teili Documentation

Release Alpha

Moritz Milde Alpha Renner Renate Krause Matteo Cartiglia Dmitrii

May 06, 2021

CONTENT:

teili, das /tali/ swiss german diminutive for piece.

This toolbox was developed to provide computational neuroscientists, as well as neuromorphic engineers, a play ground for neuronally implemented algorithms which are simulated using brian2. By providing pre-defined neural algorithms and an intuitive way to combine different aspects of those algorithms, e.g. plasticity, connectivity etc., we aim to shorten the production time of novel neural algorithms. Furthermore, by emphasising an easy and modular way to construct those algorithms from the basic building blocks of computation, e.g. neurons, synapses and populations, we target to reduce the gap between software simulation, hardware emulation and real-world applications.

CHAPTER

ONE

GETTING STARTED

Welcome to *teili*!

To start using *teili*, follow the instructions below and see our tutorials.

1.1 Prepare a virtual environment

Before we can install **teili** we encourage to set up a virtual environment using anaconda (or miniconda). Make sure you have installed conda.

- Create a virtual environment using conda

```
conda create --name <myenv> python=3.7
```

Note: Replace <myenv> with the desired name for your virtual environment

Note: If you want to use CTXCTL add the particular python version to the conda environment

```
conda create --name <myenv> python=3.7.1
```

- Activate your conda environment

```
source activate <myenv>
```

1.2 Install latest stable release of *teili*

For installing the latest stable release run the following command

```
pip install teili
python -m teili.tools.generate_teiliApps
```

The first command will install *teili* and all its dependencies. The second command generates a `teiliApps` directory in your home folder. This folder contains a selection of neuron and synapse models, tutorials, as well as unit tests. Please run the unit tests to check if everything is working as expected by

```
cd ~/teiliApps
python -m unittest discover unit_tests/
```

Attention: Running the `unit_tests` will output a lot of `Warnings` to your terminal. This, however, does not mean that the `unit_tests` failed as we need to generate and kill test plots. As long as the last line states: **Ran 78 tests in 93.373s OK** Everything is good.

You are good to go.

Note: If you find yourself seeing an warning as shown below consider updating pyqtgraph to the current development version using **pip install git+https://github.com/pyqtgraph/pyqtgraph@develop**

```
Error in atexit._run_exitfuncs:
Traceback (most recent call last):
  File "/home/you/miniconda3/envs/teili_test/lib/python3.6/site-packages/pyqtgraph/__
↪init__.py", line 312, in cleanup
    if isinstance(o, QtGui.QGraphicsItem) and isQObjectAlive(o) and o.scene() is None:
ReferenceError: weakly-referenced object no longer exists
```

In case you want the latest stable version of *teili* you refer to our repository The following steps are only required, if you need the most recent stable version/unstable developments for your simulations. If you do not require the latest version please proceed to tutorials.

1.3 Install latest development version of *teili*

- To get the most recent version of *teili* you can either clone the repository as shown below or download the tar.gz file.

```
git clone https://gitlab.com/neuroinf/teili.git
```

Note: If you have set up git properly you can use of course **git clone git@gitlab.com:neuroinf/teili.git**

Note: For the **latest development version** of *teili* please checkout the *dev* branch: **git checkout dev**.

- Navigate to the parent folder containing the cloned repository or the downloaded `tar.gz` file and install teili using pip (make sure you activated your virtual environment).

```
# Point pip to the location of the setup.py
pip install teili/
# or point pip to the downloaded tar.gz file
pip install teili*.tar.gz
```

Note: Note that the *path* provided in the install command needs to point to the folder which contains the **setup.py** file. When using the source files the `teiliApps` directory is generated automically.

The `setup.py` will by default create a folder in your home directory called `teiliApps`. This folder contains a selection of neuron and synapse models, tutorials, as well as unit tests. Please run the unit tests to check if everything is working as expected by

```
cd ~/teiliApps
python -m unittest discover unit_tests/
```

You are good to go!

Note: Due to *pyqtgraph* the unit tests will print warnings, as we generate and close figures to test the functionality of *teili*. These warning are normal. As longer as no `Error` is returned, everything is behaving as expected.

If you want to change the location of `teiliApps`, you can do so by moving the folder manually.

The installation instructions above will install all requirements and dependencies. It will also build pre-defined neuron and synapse models and place them in `teiliApps/equations/`. Make sure you checkout our tutorials.

1.4 Re-building models after installation

Note: By default models are generated during installation. **Only if** you accidentally deleted them manually you need to rebuild models.

By default the models will be placed in `teiliApps/equations/`. If you want to place them at a different location follow the instructions below:

```
source activate <myenv>
python
```

```
from teili import neuron_models, synapse_models
neuron_models.main("/path/to/my/equations/")
synapse_models.main("/path/to/my/equations/")
```

Attention: You need to specify the absolute path. So use `/home/<YOU>/your_custom_path/`, rather than `~/your_custom_path/`.

Note, that the following folder structure is generated in the specified location: `/path/to/my/equations/teiliApps/equations/`. If you simply call the classes without a path the equations will be placed in `~/teiliApps/equations/`. Have a look at our tutorials to see how to use teili and which features it provides to you.

CHAPTER

TWO

TUTORIALS

Welcome to teili, a modular python-based framework for developing, testing and visualization of neural algorithms. Before going through our tutorials we highly recommend doing the tutorials provided by Brian2

2.1 Dynamic model generation vs. static model import

2.1.1 Working with the dynamic model generation method

See an example for how to import model classes defined in `models.neuron_models` and `models.synapse_models` below.

```
from teili.core.groups import Neurons, Connections

from teili.models.neuron_models import DPI as neuron_model
from teili.models.synapse_models import DPISyn as syn_model

test_neuron1 = Neurons(N=2,
                       equation_builder=neuron_model(num_inputs=2),
                       name="test_neuron1", verbose=True)

test_neuron2 = Neurons(N=2,
                       equation_builder=neuron_model(num_inputs=2),
                       name="test_neuron2",verbose=True)

test_synapse = Connections(test_neuron1, test_neuron2,
                           equation_builder=syn_model,
                           name="test_synapse", verbose=True)
```

If you want to see a less detailed report on which equations were used during the generation you can set `verbose=False`.

Note: The `Neurons` class a `num_inputs` property. This allows the user to define how many different afferent connections this particular neuron population has to expect. The default is set to 1. Do not define more inputs than you expect.

2.1.2 Working with the static model import method

If you prefer to use neuron and synapse models statically specified in a file you can import default models from `teiliApps/equations` directory using the `import_eq` method. You can manually tweak these models and also generate your own models. For more information please have a look at the EquationBuilder.

```
import os
from teili.core.groups import Neurons, Connections
from teili.models.builder.neuron_equation_builder import NeuronEquationBuilder
from teili.models.builder.synapse_equation_builder import SynapseEquationBuilder

path = os.path.expanduser("~")
model_path = os.path.join(path, "teiliApps", "equations", "")

my_neuron_model = NeuronEquationBuilder.import_eq(
        model_path + 'DPI.py', num_inputs=2)

my_synapse_model = SynapseEquationBuilder.import_eq(
        model_path + 'DPISyn.py')

test_neuron1 = Neurons(N=2,
                       equation_builder=my_neuron_model,
                       name="test_neuron1")
test_neuron2 = Neurons(N=2,
                       equation_builder=my_neuron_model,
                       name="test_neuron2")

test_synapse = Connections(test_neuron1, test_neuron2,
                           equation_builder=my_synapse_model,
                           name="test_synapse")
```

If you want to see a more detailed report on which equations were used during the generation you can set `verbose=True`, such that it looks like this

```
test_neuron1 = Neurons(N=2,
                       equation_builder=my_neuron_model,
                       name="test_neuron1", verbose=True)
```

2.2 Neuron & Synapse tutorial

We created a simple tutorial of how to simulate a small neural network either using the `EquationBuilder`. The complete tutorial is located in `teiliApps/tutorials/neuron_synapse_tutorial.py`.

2.2.1 Import relevant libraries

First we import all required libraries

```
from pyqtgraph.Qt import QtGui
import pyqtgraph as pg
import numpy as np

from Brian2 import ms, pA, nA, prefs,\
        SpikeMonitor, StateMonitor,\
```

(continues on next page)

(continued from previous page)

```
        SpikeGeneratorGroup

from teili.core.groups import Neurons, Connections
from teili import TeiliNetwork
from teili.models.neuron_models import DPI as neuron_model
from teili.models.synapse_models import DPISyn as syn_model
from teili.models.parameters.dpi_neuron_param import parameters as neuron_model_param

from teili.tools.visualizer.DataViewers import PlotSettings
from teili.tools.visualizer.DataControllers import Rasterplot, Lineplot
```

We now can define the target for the code generation. Typically we use the `numpy` backend. For more details on how to run your code more efficient and faster have a look at brian's standalone mode

```
prefs.codegen.target = "numpy"
```

2.2.2 Defining the input stimulus

We can now generate a simple input pattern using Brian2's `SpikeGeneratorGroup`

```
input_timestamps = np.asarray([1, 3, 4, 5, 6, 7, 8, 9]) * ms
input_indices = np.asarray([0, 0, 0, 0, 0, 0, 0, 0])
input_spikegenerator = SpikeGeneratorGroup(1, indices=input_indices,
                                           times=input_timestamps, name='gtestInp')
```

After defining the input group, we can build a `TeiliNetwork`.

2.2.3 Defining the network

```
Net = TeiliNetwork()

test_neurons1 = Neurons(N=2,
                        equation_builder=neuron_model(num_inputs=2),
                        name="test_neurons1")

test_neurons2 = Neurons(N=2,
                        equation_builder=neuron_model(num_inputs=2),
                        name="test_neurons2")



input_synapse = Connections(input_spikegenerator, test_neurons1,
                            equation_builder=syn_model(),
                            name="input_synapse")
input_synapse.connect(True)

test_synapse = Connections(test_neurons1, test_neurons2,
                           equation_builder=syn_model(),
                           name="test_synapse")
test_synapse.connect(True)
```

After initialising the populations of `Neurons` and connecting them via synaptic `Connections`, we can set model parameters.

2.2.4 Setting parameters

Note that parameters are set by default. This example only shows how you would need to go about if you want to set non-default (user defined) parameters. Example parameter dictionaries can be found `teili/models/parameters`. You can change all the parameters like this after creation of the `Neurons` or `Connections`.

```
# Example of how to set parameters, saved as a dictionary
test_neurons1.set_params(neuron_model_param)
test_neurons2.set_params(neuron_model_param)

# Example of how to set a single parameter
test_neurons1.refP = 1 * ms
test_neurons2.refP = 1 * ms

if 'Imem' in neuron_model().keywords['model']:
        input_synapse.weight = 5000
        test_synapse.weight = 800
        test_neurons1.Iconst = 10 * nA
elif 'Vm' in neuron_model().keywords['model']:
        input_synapse.weight = 1.5
        test_synapse.weight = 8.0
        test_neurons1.Iconst = 3 * nA
```

Note: The `if` condition is only there for convenience of the user to run our tutorial and switch between voltage- or current-based models. Normally, you have either current or voltage-based models in your simulation, thus you will not need the `if` condition.

Attention: The `weight` of a given `Connection` is multiplied with the `baseweight`, which is currently initialised to *7 pA* by default for the **DPI synapse model**. In order to elicit an output spike in response to a single `SpikeGenerator` input spike the weight must be greater than 3250.

Now our simple spiking neural network is defined. In order to visualize what is happening during the simulation we need to monitor the spiking behaviour of our neurons and other state variables of neurons and synapses.

2.2.5 Defining monitors

```
spikemon_input = SpikeMonitor(
        input_spikegenerator, name='spikemon_input')
spikemon_test_neurons1 = SpikeMonitor(
        test_neurons1, name='spikemon_test_neurons1')
spikemon_test_neurons2 = SpikeMonitor(
        test_neurons2, name='spikemon_test_neurons2')

statemon_input_synapse = StateMonitor(
        input_synapse, variables='I_syn',
        record=True, name='statemon_input_synapse')

statemon_test_synapse = StateMonitor(
        test_synapse, variables='I_syn',
        record=True, name='statemon_test_synapse')
```

(continues on next page)

(continued from previous page)

```
if 'Imem' in neuron_model().keywords['model']:
        statemon_test_neurons1 = StateMonitor(test_neurons1,
                                              variables=["Iin", "Imem", "Iahp"],
                                              record=[0, 1],
                                              name='statemon_test_neurons1')

        statemon_test_neurons2 = StateMonitor(test_neurons2,
                                              variables=['Imem'],
                                              record=0,
                                              name='statemon_test_neurons2')


elif 'Vm' in neuron_model().keywords['model']:
        statemon_test_neurons1 = StateMonitor(test_neurons1,
                                              variables=["Iin", "Vm", "Iadapt"],
                                              record=[0, 1],
                                              name='statemon_test_neurons1')

        statemon_test_neurons2 = StateMonitor(test_neurons2,
                                              variables=['Vm'],
                                              record=0,
                                              name='statemon_test_neurons2')
```

2.2.6 Starting the simulation

We can now finally add all defined `Neurons` and `Connections`, as well as the monitors to our `TeiliNetwork` and run the simulation.

```
Net.add(input_spikegenerator,
        test_neurons1, test_neurons2,
        input_synapse, test_synapse,
        spikemon_input, spikemon_test_neurons1,
        spikemon_test_neurons2,
        statemon_test_neurons1, statemon_test_neurons2,
        statemon_test_synapse, statemon_input_synapse)

duration = 500
Net.run(duration * ms)
```

2.2.7 Using statically defined models instead

If you, however, prefer to use the equation files located in `teiliApps/equations/`, you need to change the way the neurons and synapses are defined. The only thing that changes from the example above is the import and neuron/synapse group definition. The complete tutorial can be found in `teiliApps/tutorials/neuron_synapse_import_eq_tutorial.py`.

```
import os
from teili.models.builder.neuron_equation_builder import NeuronEquationBuilder
from teili.models.builder.synapse_equation_builder import SynapseEquationBuilder

# For this example you must first run models/neuron_models.py and synapse_models.py,
# which will create the equation template. This will be stored in models/equations
# Building neuron objects
```

(continues on next page)

(continued from previous page)

```
path = os.path.expanduser("~")
model_path = os.path.join(path, "teiliApps", "equations", "")

builder_object1 = NeuronEquationBuilder.import_eq(
        model_path + 'DPI.py', num_inputs=2)
builder_object2 = NeuronEquationBuilder.import_eq(
        model_path + 'DPI.py', num_inputs=2)

builder_object3 = SynapseEquationBuilder.import_eq(
        model_path + 'DPISyn.py')
builder_object4 = SynapseEquationBuilder.import_eq(
        model_path + 'DPISyn.py')

test_neurons1 = Neurons(2, equation_builder=builder_object1, name="test_neurons1")
test_neurons2 = Neurons(2, equation_builder=builder_object2, name="test_neurons2")

input_synapse = Connections(input_spikegenerator, test_neurons1,
                            equation_builder=builder_object3,
                            name="input_synapse", verbose=False)
input_synapse.connect(True)
test_synapse = Connections(test_neurons1, test_neurons2,
                           equation_builder=builder_object4, name="test_synapse")
test_synapse.connect(True)
```

The way parameters are set remains the same.

2.2.8 Visualising the networks activity

In order to visualize the behaviour the example script also plots a couple of spike and state monitors. For a full tutorial of how use *teili's* `Visualizer` class please refer to our visualiser tutorial

```
app = QtGui.QApplication.instance()
if app is None:
    app = QtGui.QApplication(sys.argv)
else:
    print('QApplication instance already exists: %s' % str(app))

pg.setConfigOptions(antialias=True)
labelStyle = {'color': '#FFF', 'font-size': 12}
MyPlotSettings = PlotSettings(fontsize_title=labelStyle['font-size'],
                              fontsize_legend=labelStyle['font-size'],
                              fontsize_axis_labels=10,
                              marker_size=7)

win = pg.GraphicsWindow()
win.resize(2100, 1200)
win.setWindowTitle('Simple Spiking Neural Network')

p1 = win.addPlot(title="Input spike generator")
p2 = win.addPlot(title="Input synapses")
win.nextRow()
p3 = win.addPlot(title='Intermediate test neurons 1')
p4 = win.addPlot(title="Test synapses")
win.nextRow()
```

(continues on next page)

(continued from previous page)

```
p5 = win.addPlot(title="Rasterplot of output test neurons 2")
p6 = win.addPlot(title="Output test neurons 2")


# Spike generator
Rasterplot(MyEventsModels=[spikemon_input],
                      MyPlotSettings=MyPlotSettings,
                      time_range=[0, duration],
                      neuron_id_range=None,
                      title="Input spike generator",
                      xlabel='Time (ms)',
                      ylabel="Neuron ID",
                      backend='pyqtgraph',
                      mainfig=win,
                      subfig_rasterplot=p1,
                      QtApp=app,
                      show_immediately=False)

# Input synapses
Lineplot(DataModel_to_x_and_y_attr=[(statemon_input_synapse, ('t', 'I_syn'))],
                    MyPlotSettings=MyPlotSettings,
                    x_range=[0, duration],
                    title="Input synapses",
                    xlabel="Time (ms)",
                    ylabel="EPSC (A)",
                    backend='pyqtgraph',
                    mainfig=win,
                    subfig=p2,
                    QtApp=app,
                    show_immediately=False)

# Intermediate neurons
if hasattr(statemon_test_neurons1, 'Imem'):
    MyData_intermed_neurons = [(statemon_test_neurons1, ('t', 'Imem'))]
if hasattr(statemon_test_neurons1, 'Vm'):
    MyData_intermed_neurons = [(statemon_test_neurons1, ('t', 'Vm'))]

i_current_name = 'Imem' if 'Imem' in neuron_model().keywords['model'] else 'Vm'
Lineplot(DataModel_to_x_and_y_attr=MyData_intermed_neurons,
                    MyPlotSettings=MyPlotSettings,
                    x_range=[0, duration],
                    title='Intermediate test neurons 1',
                    xlabel="Time (ms)",
                    ylabel=i_current_name,
                    backend='pyqtgraph',
                    mainfig=win,
                    subfig=p3,
                    QtApp=app,
                    show_immediately=False)

# Output synapses
Lineplot(DataModel_to_x_and_y_attr=[(statemon_test_synapse, ('t', 'I_syn'))],
                    MyPlotSettings=MyPlotSettings,
                    x_range=[0, duration],
                    title="Test synapses",
                    xlabel="Time (ms)",
                    ylabel="EPSC (A)",
```

(continues on next page)

(continued from previous page)

```
                    backend='pyqtgraph',
                    mainfig=win,
                    subfig=p4,
                    QtApp=app,
                    show_immediately=False)


Rasterplot(MyEventsModels=[spikemon_test_neurons2],
                      MyPlotSettings=MyPlotSettings,
                      time_range=[0, duration],
                      neuron_id_range=None,
                      title="Rasterplot of output test neurons 2",
                      xlabel='Time (ms)',
                      ylabel="Neuron ID",
                      backend='pyqtgraph',
                      mainfig=win,
                      subfig_rasterplot=p5,
                      QtApp=app,
                      show_immediately=False)

if hasattr(statemon_test_neurons2, 'Imem'):
    MyData_output = [(statemon_test_neurons2, ('t','Imem'))]
if hasattr(statemon_test_neurons2, 'Vm'):
    MyData_output = [(statemon_test_neurons2, ('t','Vm'))]

Lineplot(DataModel_to_x_and_y_attr=MyData_output,
                    MyPlotSettings=MyPlotSettings,
                    x_range=[0, duration],
                    title="Output test neurons 2",
                    xlabel="Time (ms)",
                    ylabel="%s" %i_current_name,
                    backend='pyqtgraph',
                    mainfig=win,
                    subfig=p6,
                    QtApp=app,
                    show_immediately=False)

app.exec()
```

In both cases of model definition (rather dynamic model generation or static model import) the resulting figure should look like this:

kernels. Here we provide a tutorial of how to use them and how they look when applied together with a neuron model. The first steps are the same as in the previous tutorial. The tutorial is located in `teiliApps/tutorials/synaptic_kernels_tutorial.py`. We first import all required libraries:

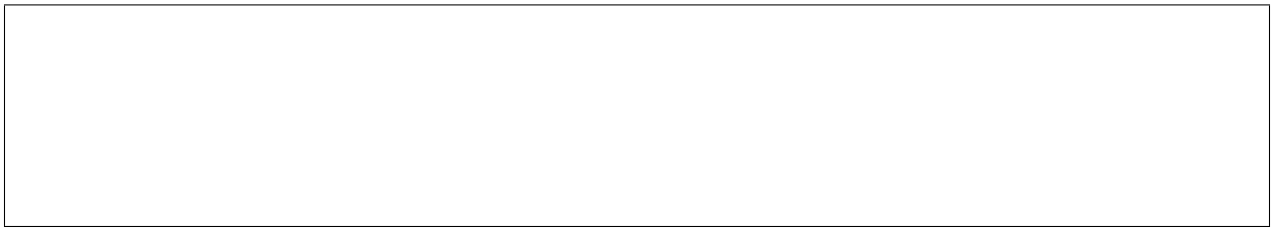
(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

we use the `numpy` backend.



one will send excitatory and the other inhibitory spikes.

```
↪5, 3]) * ms
```

(continues on next page)

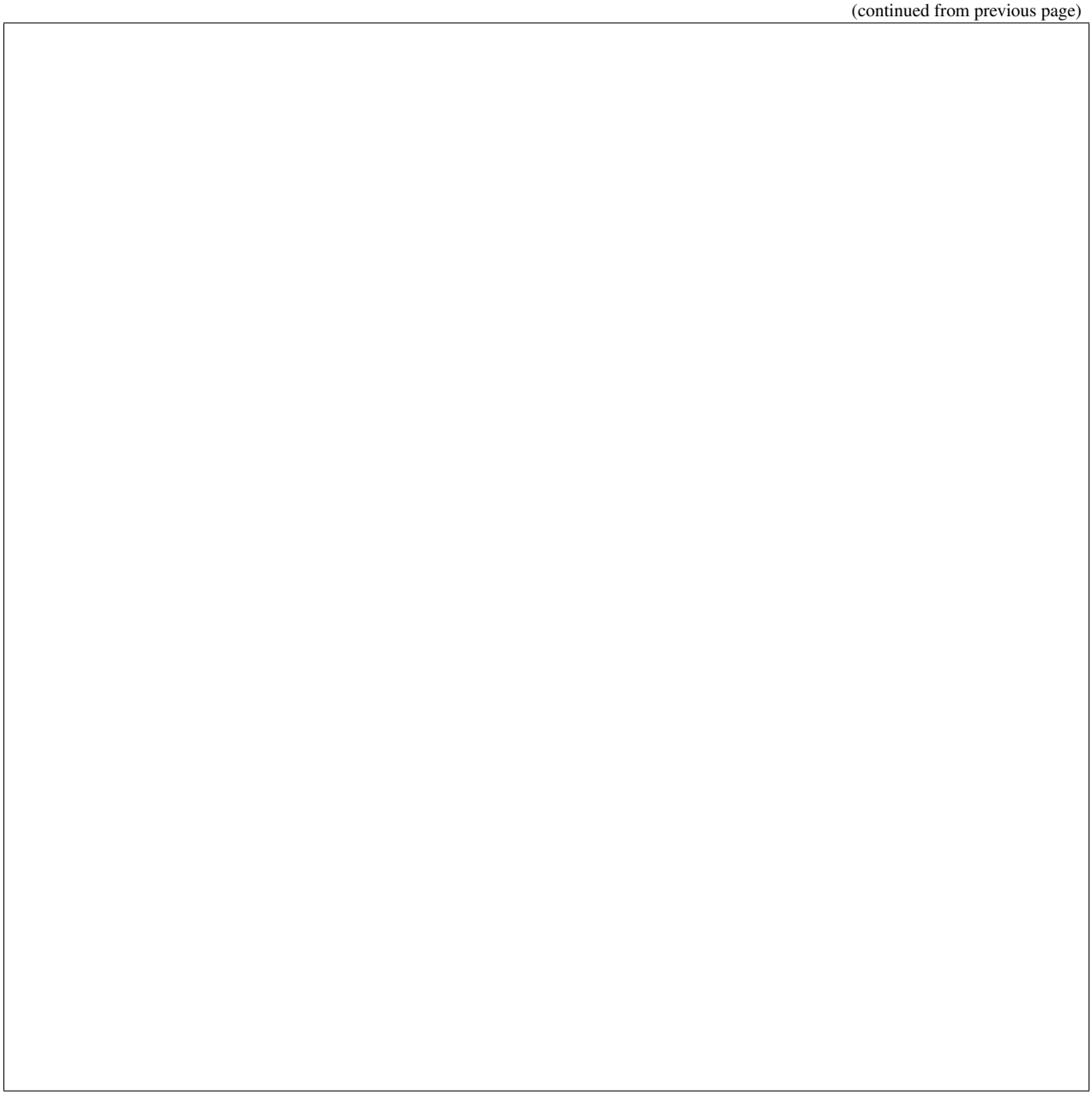
(continued from previous page)

```
↪           times=input_timestamps, name='gtestInp')
```

rons. The first one will receive synapses with an `Alpha` kernel shape, the second will receive synapses with a `Resonant` kernel shape and the third one will receive a synapse which models the `DPI synapse model`, which has an exponential decay shape. Note that a single neuron can receive synapses with different kernels at the same time. Here we split them for better visualisation.

(continues on next page)

(continued from previous page)



Note: We are using the DPI neuron model for this tutorial but the synaptic model is independent of the neuron's model and therefore other neuron models can be used.

standard parameters from a dictionary but also change single parameters as in this example with the refractory period. Now we specify the connections. The synaptic models are `Alpha`, `Resonant` and `DPI`.

```
↪builder=Alpha(), name="test_syn_alpha", verbose=False)
```

(continues on next page)

(continued from previous page)

```
↪syn_resonant", verbose=False)
```

(continues on next page)

```
↪syn_dpi", verbose=False)
```

(continued from previous page)

the `SpikeGenerator` will have an excitatory effect (weight>0) and the second neuron will have an inhibitory effect (weight<0) on the post-synaptic neuron.

pA by default for the **DPI synapse model**. In order to elicit an output spike in response to a single `SpikeGenerator` input spike the weight must be greater than 3250.

is happening during the simulation we need to monitor the spiking behaviour of our `Neurons` and other state variables of our `Neurons` and `Connections`.

(continues on next page)

(continued from previous page)

```
↪               record=True, name='statemon_syn_alpha')
```

(continues on next page)

(continued from previous page)

```
↪           record=True, name='statemon_syn_dpi')
```

(continues on next page)

(continued from previous page)

```
↪                record=0, name='statemon_test_neuron2')
```

(continues on next page)

(continued from previous page)

run the simulation.

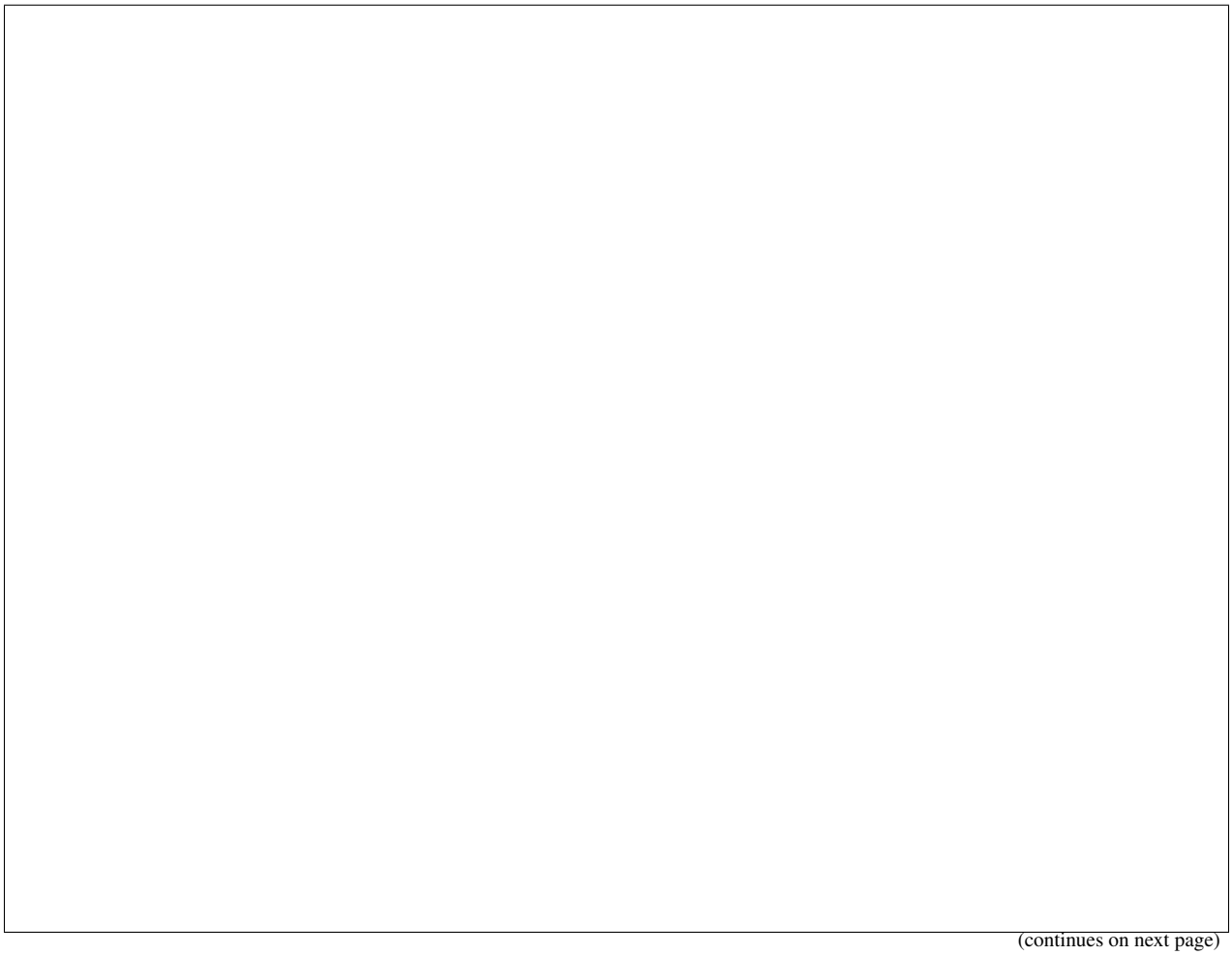
(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

spike and state monitors.



(continues on next page)

(continued from previous page)

```
↪         fontsize_legend=labelStyle['font-size'],
```

(continues on next page)

(continued from previous page)

```
↪         marker_size=7)
```

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

```
↪           state_variables_times=[statemon_syn_alpha.t])
```

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

```
↪ movable=False,)
```

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

```
↪           state_variables=[data],
```

(continues on next page)


```
↪           state_variables_times=[statemon_syn_resonant.t])
```

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

```
↪           state_variables=[data],

↪           state_variables_times=[statemon_syn_dpi.t])
```

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

```
↪ movable=False,)
```

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

Iin of the neuron. To better visualize the synapse dynamics, **we have multiplied the I_syn of the inhibitory synapse by -1**. The resulting figure should look like this:

scription of neuronal algorithms which can be formalised as scalable building blocks. One example `BuildingBlock` is the winner-takes-all (`WTA`) network. To show the basic interface of how to use a `WTA` network we start with the imports. The original file can be found in `teiliApps/tutorials/wta_tutorial.py`

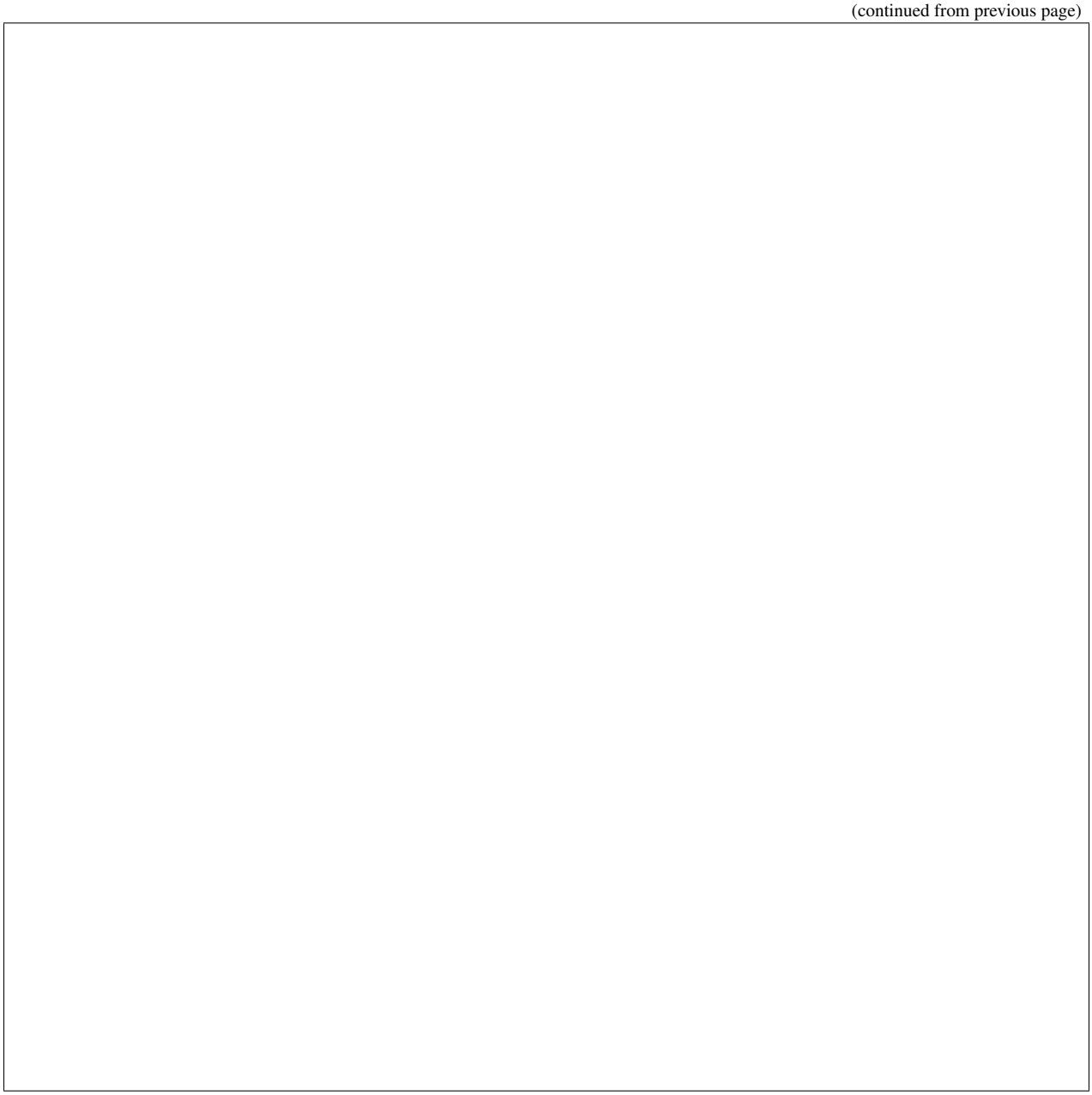
Note: For instructions on how to design a novel *BuildingBlock* please refer to BuildingBlock development

(continues on next page)

(continued from previous page)

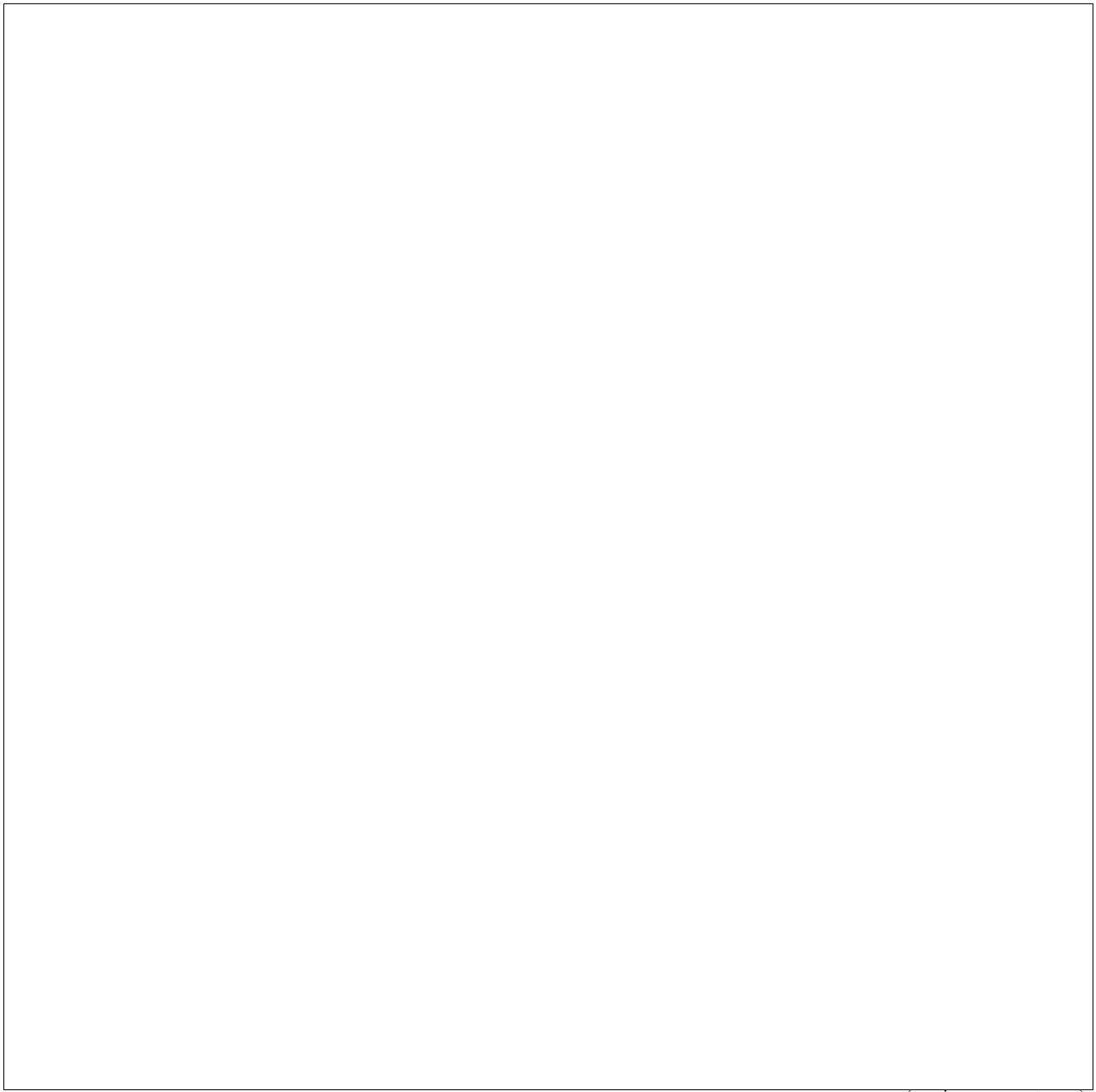
(continues on next page)

(continued from previous page)



the standard `numpy` backend, or by setting `run_as_standalone = True` the code will be compiled as C++ code before it is executed.

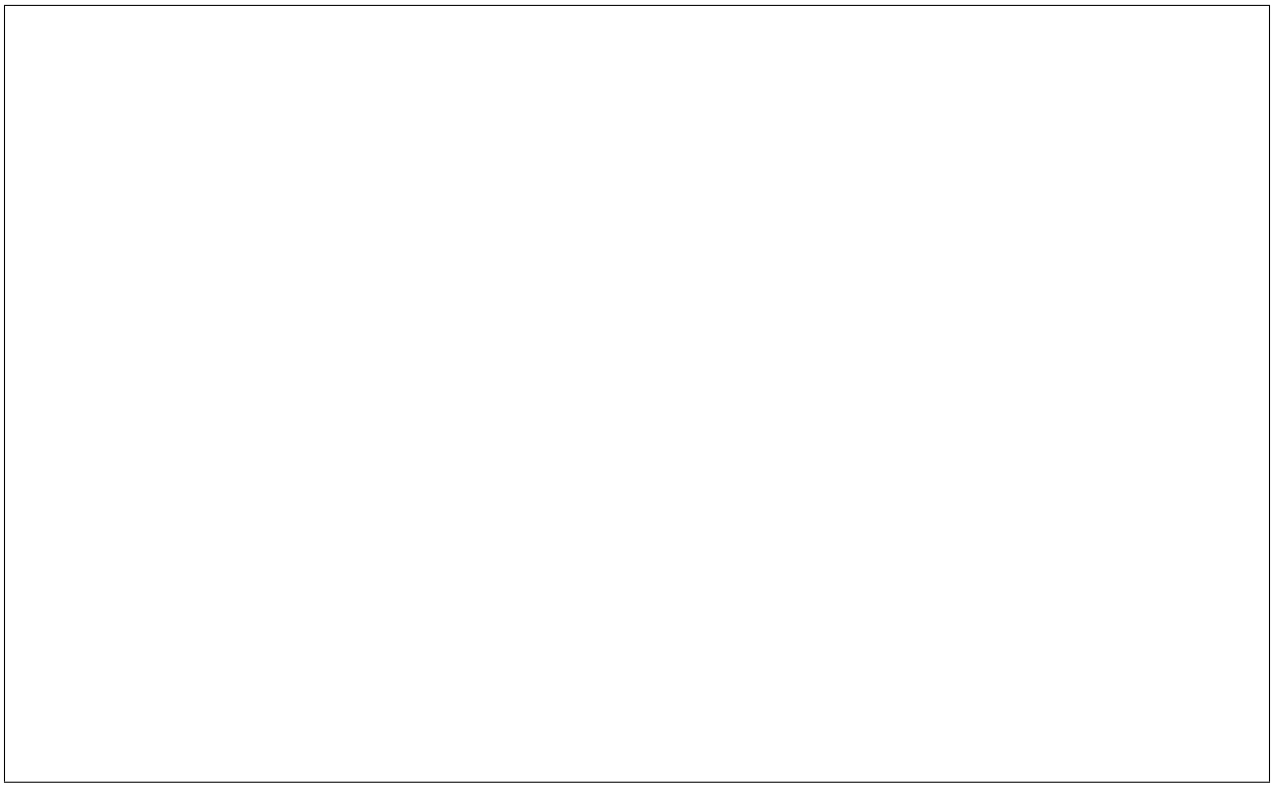
Note: To run the WTA `BuildingBlock` in standalone mode please refer to our standalone tutorial which is located in `teiliApps/tutorials/wta_standalone_tutorial.py`.



(continues on next page)

(continued from previous page)

its working behaviour, we initialize an instance of a stimulus test class specifically designed for WTA's.



complicated. When we generate our `BuildingBlock`, we need to pass specific parameters, which set internal synaptic weights, connectivity kernels and connectivity probabilities. For more information see BuildingBlocks docu-

mentation and the source code, respectively. To do so we define a dictionary, which is passed to the `BuildingBlock` class. Feel free to change the parameters to see what effect it has on the stability and signal-to-noise ratio.

```
↪
↪550,
```

(continues on next page)

(continued from previous page)

```
↪ms,
```

(continues on next page)

(continued from previous page)

```
↪7,
```

WTA network.

(continues on next page)

(continued from previous page)

```
↪
```

(continues on next page)

(continued from previous page)

```
↪ end_time=duration)
```

(continues on next page)

(continued from previous page)

```
↪    test_WTA._groups['n_exc'],

↪    equation_builder=DPISyn(),
```

(continues on next page)


```
↪    name="noise_syn")
```

(continued from previous page)

to *7 pA* by default for the **DPI synapse model**. In order to elicit an output spike in response to a single `SpikeGenerator` input spike the weight must be greater than 3250.

(continues on next page)

(continued from previous page)

```
↪          ('Iin0', 'Iin1', 'Iin2', 'Iin3'),
```

(continues on next page)

(continued from previous page)

```
↪           record=True,

↪           name='statemon_wta_input')
```

(continues on next page)

(continued from previous page)

fine standalone parameters, if you are using standalone mode.

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

```
↪               ('stestWTA_e_latSigma', 2),

↪               ('stestWTA_Inpe_weight', 900),
```

(continues on next page)


```
↪               ('stestWTA_Inhe_weight', 500),
```

(continued from previous page)

```
↪               ('stestWTA_Inhi_weight', -550),

↪               ('test_WTA_refP', 1. * msecond),

↪               ('testWTA_Inh_refP', 1. * msecond)])
```

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

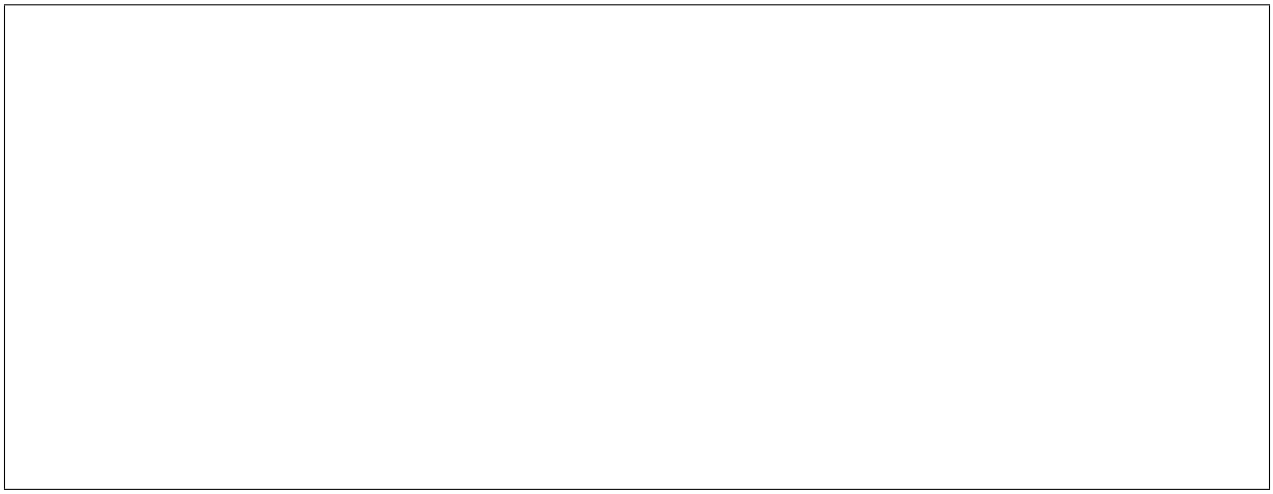
(continues on next page)

(continued from previous page)

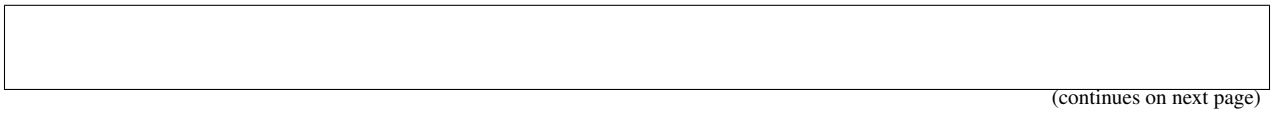
tended to provide additional functionality, such as extending a given synapse model with for example a Spike-Timing Dependent Plasticity (STDP) mechanism. STDP is one mechanism which has been identified experimentally how neurons adjust their synaptic weight according to correlated firing patterns. Feel free to read more about STDP. The following tutorial can be found at `teiliApps/tutorials/stdp_tutorial.py` If we want to add an activity dependent plasticity mechanism to our network we again start by importing the required packages.

(continues on next page)

(continued from previous page)

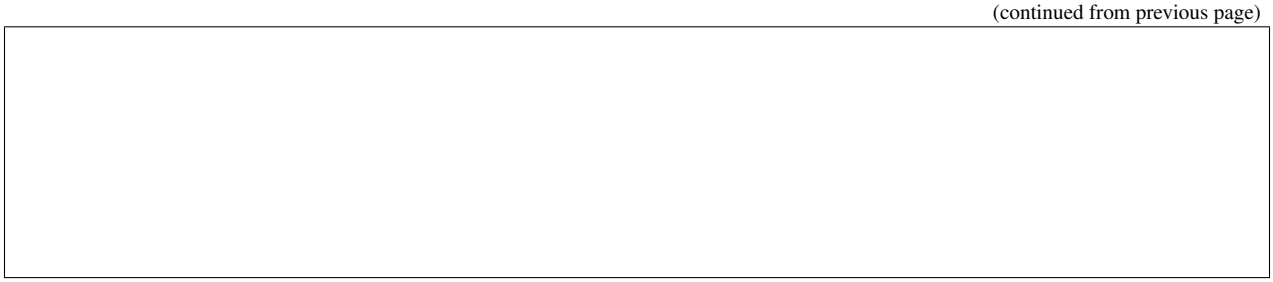


tegration error and to be sure that the network performs the desired computation. But keep in mind by decreasing the `defaultclock.dt` the simulation takes longer! In the next step we will load a simple STDP-protocol from our STDP testbench, which provides us with pre-defined pre-post spikegenerators with specific delays between pre and post spiking activity.

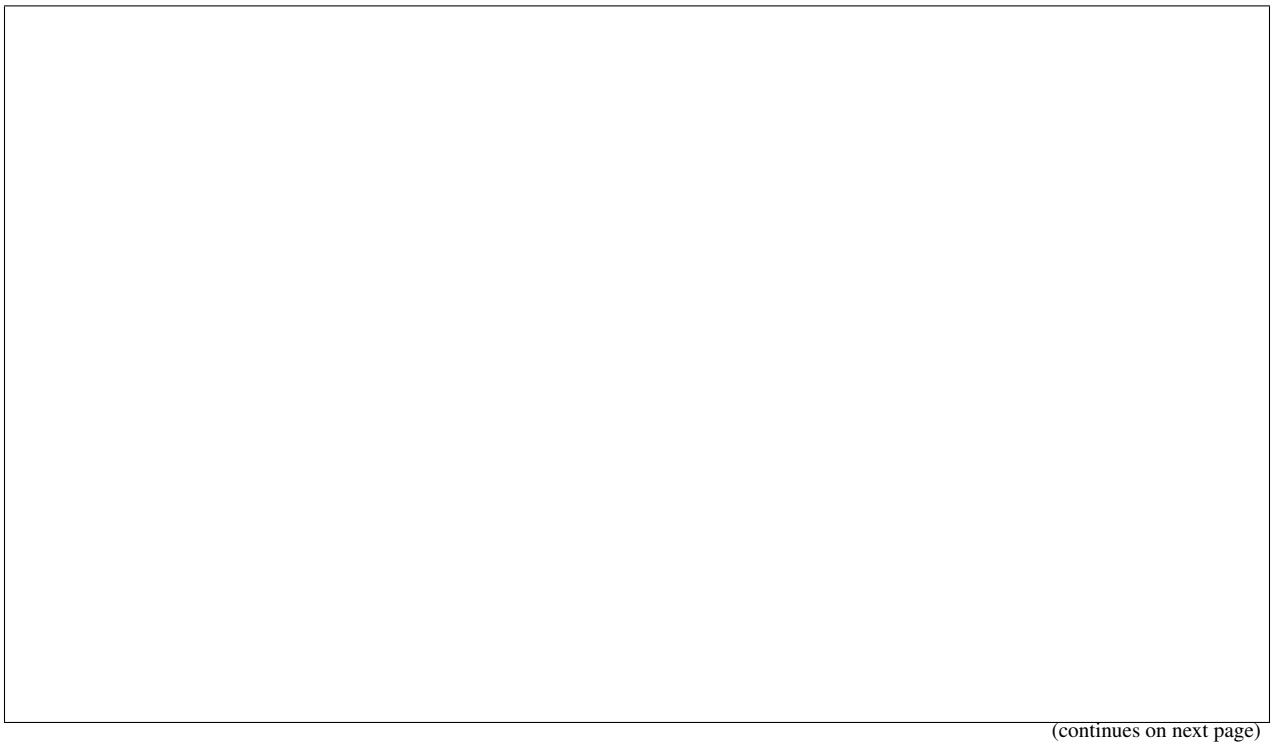


(continues on next page)

(continued from previous page)



tic synapses between them.



(continues on next page)

```
↪ equation_builder=DPI(num_inputs=1),
```

(continued from previous page)

```
↪   name='pre_neurons')

↪   equation_builder=DPI(num_inputs=2),
```

(continues on next page)

(continued from previous page)

```
↪       equation_builder=DPISyn(),

↪       name='pre_synapse')
```

(continues on next page)

(continued from previous page)

```
↪       equation_builder=DPISyn(),

↪       name='post_synapse')
```

(continues on next page)

(continued from previous page)

```
↪        name='stdp_synapse')
```

Note: Note that we define the temporal window of the STDP kernel using `taupost` and `taupost` bias. The learning rate, i.e. the amount of maximal weight change, is set by `dApre`.

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

col and the respective weight change.

(continues on next page)

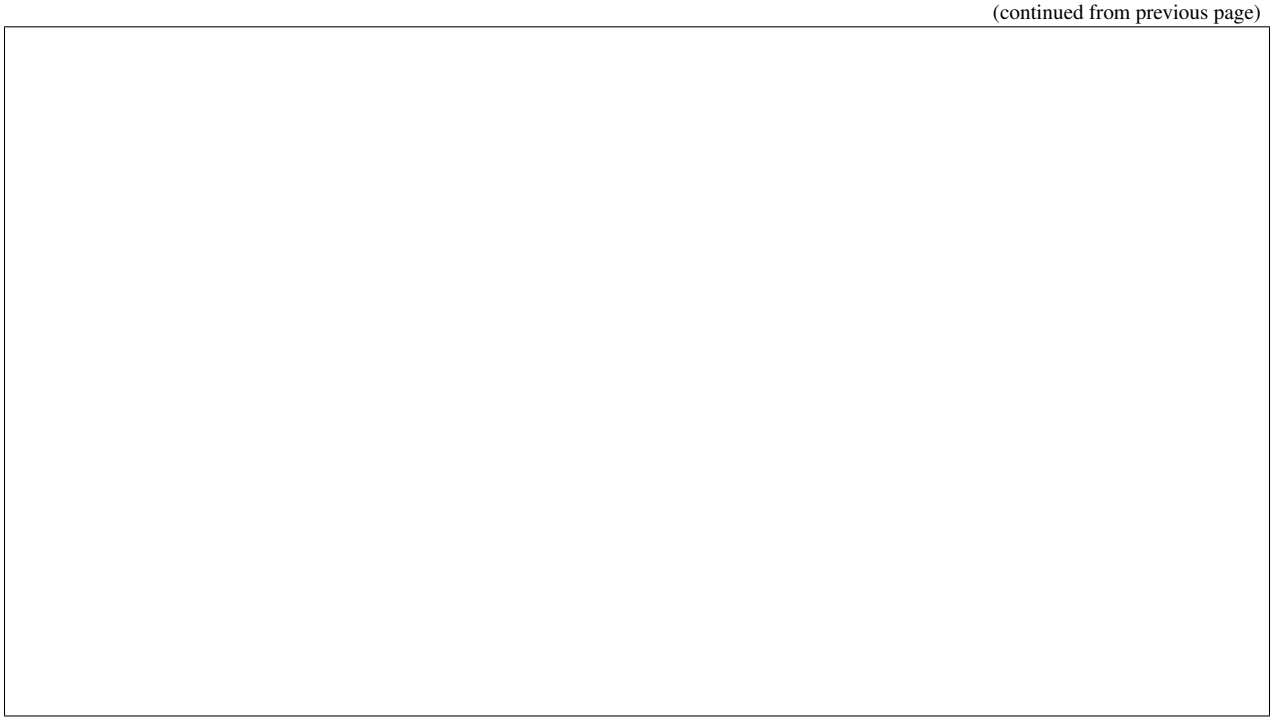
(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

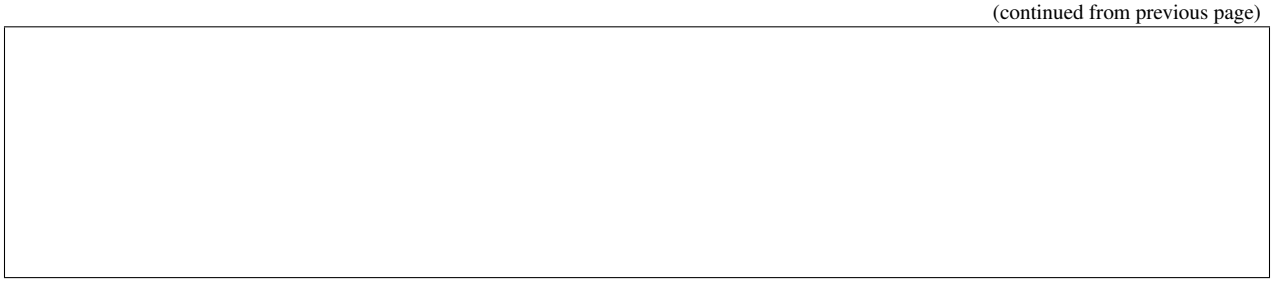


(continues on next page)

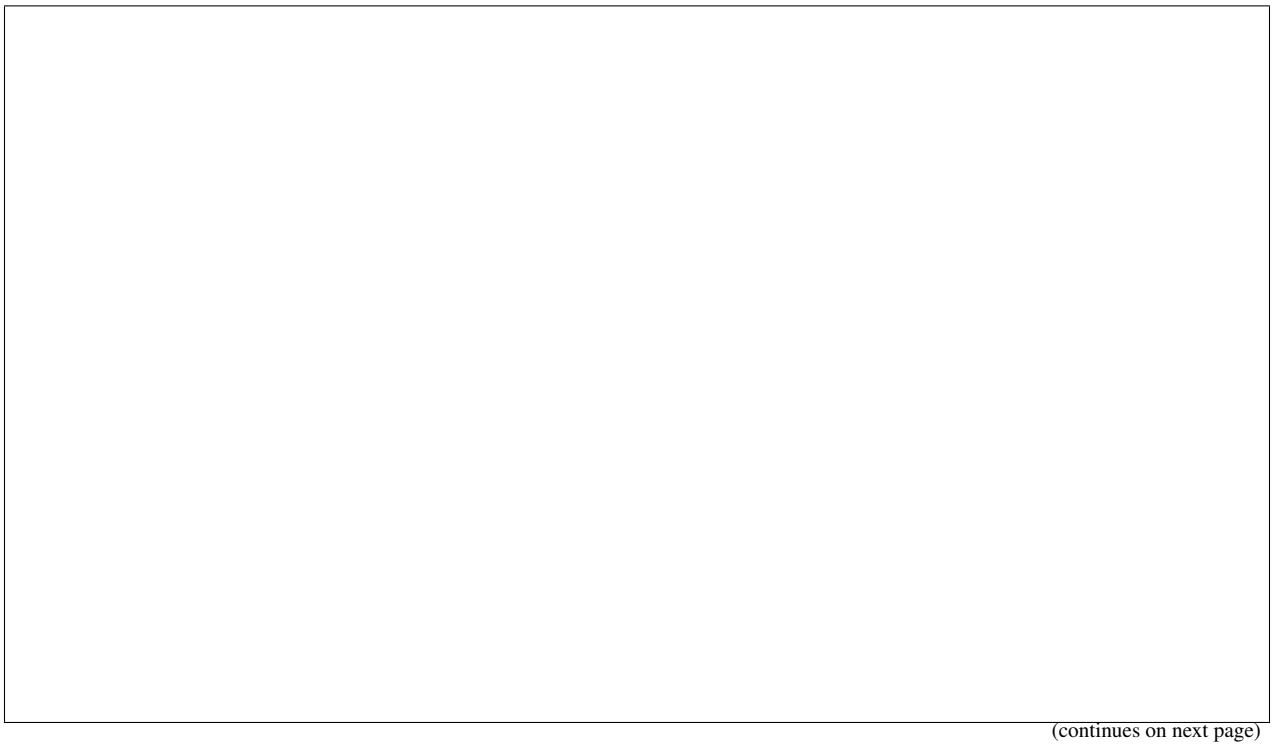
(continued from previous page)

(continues on next page)

(continued from previous page)



the STDP synapse.



(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

```
↪            state_variables=[np.asarray(statemon_post_synapse.I_syn[1])],
```

(continues on next page)

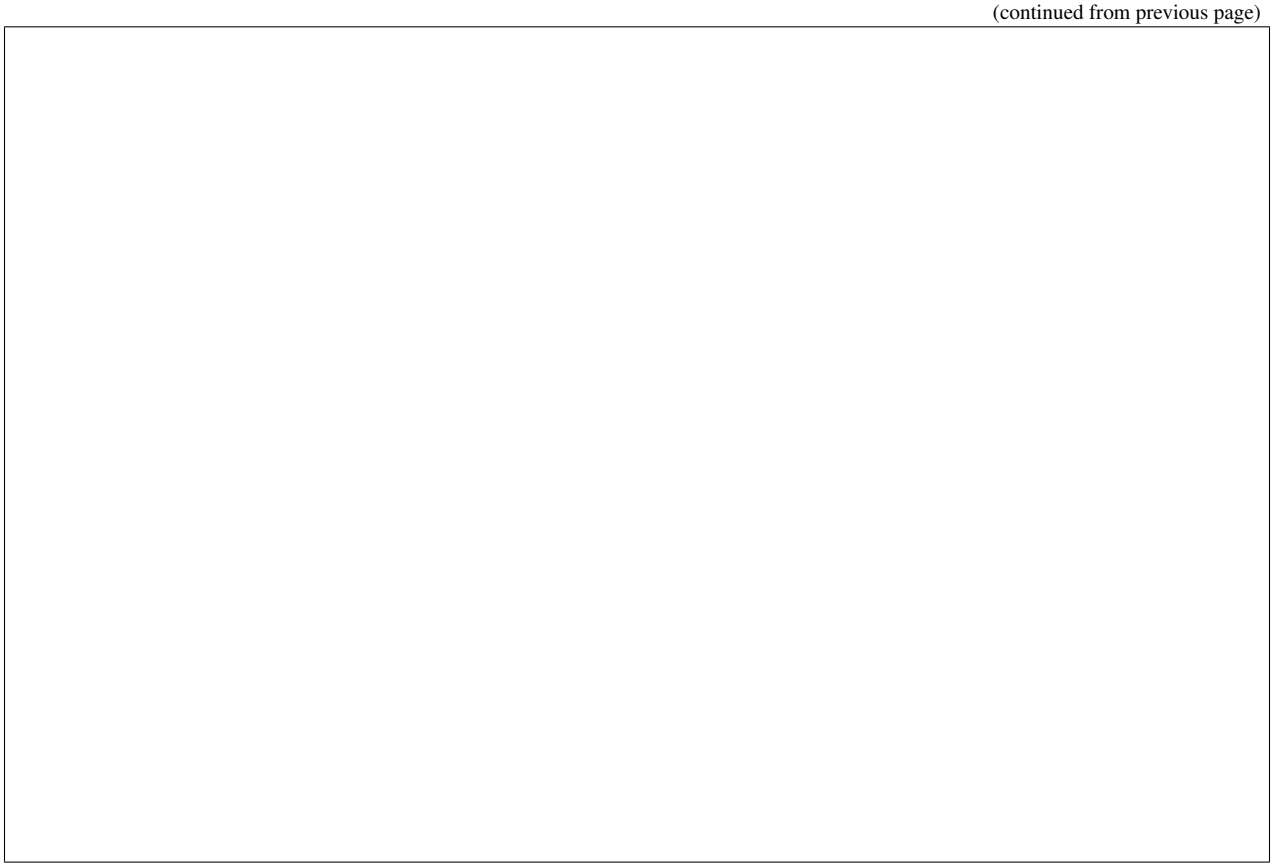
(continued from previous page)

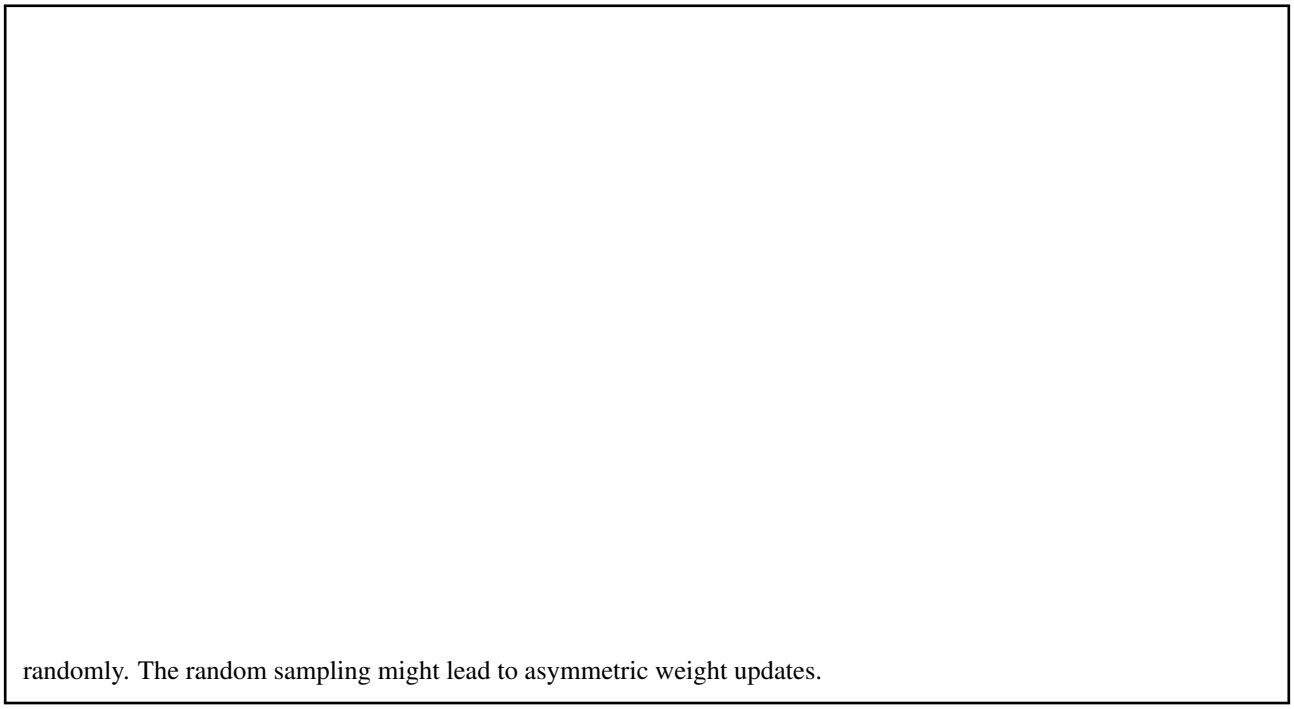
(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)



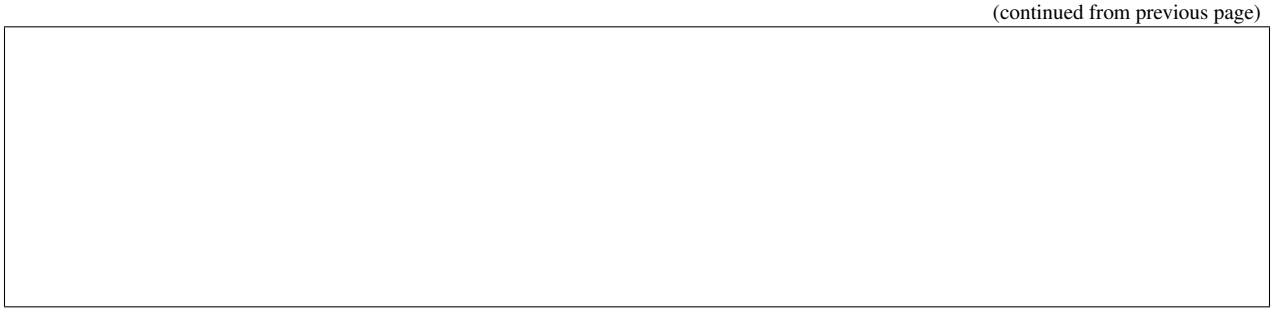


randomly. The random sampling might lead to asymmetric weight updates.

spike pairs we can visualize the STDP kernel. The following tutorial can be found at `~/teiliApps/tutorials/stdp_kernel_tutorial.py` We start again by importing the required dependencies.

(continues on next page)

(continued from previous page)



the visualization.

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

alize the STDP kernel. Now we can define our neuronal populations and connect them via a STDP synapse.

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

```
↪      tspike:second''',
```

```
↪', refractory=100 * ms)
```

(continues on next page)

(continued from previous page)

(continues on next page)

```
↪ equation_builder=DPIstdp(),
```

(continued from previous page)

```
↪ name='stdp_synapse')
```

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

```
↪       state_variables_times=[np.asarray((post_neurons.tspike - pre_neurons.tspike) /
↪ ms)])
```

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

as synaptic time constants, spiking thresholds etc. Biological neurons and synapses, however, show high diversity in their morphology and thus in their behaviour which is modelled using model parameters. This diversity, i.e. heterogeneity, leads to highly variable responses which are usually not considered. On the one hand, this heterogeneity might be actually relevant for computation and stability, thus should be encapsulated in spiking neural network models. On the other hand neuromorphic mixed-signal analogue-digital sensory-processing systems are subject to so-called device mismatch due to imperfections in the manufacturing process. To better assess if a given model is suited for an implementation on neuromorphic hardware and to advance our understanding of computation in heterogeneous population of neurons *teili* provides a *Group* level method to add device mismatch. This example shows how to add device mismatch to a neural network with one input neuron connected to 1000 output neurons. Once our population is created, we will add device mismatch to the selected parameters by specifying a dictionary with parameter names as keys and mismatch standard deviation as values. The following tutorial can be found at `~/teiliApps/examples/mismatch_tutorial.py`. Here the selected parameters for the neuron and synapse models are specified in `mismatch_neuron_param` and `mismatch_synap_param` respectively.

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

synaptic weight (`baseweight`), with a standard deviation of 20% of the current value for both parameters. Let's first create the input `SpikeGeneratorGroup`, the output layer and the synapses. Notice that a constant input current has been set for the output neurons.

(continues on next page)

(continued from previous page)

```
↪                 times=ts_input, name='gtestInp')
```

(continues on next page)

(continued from previous page)

```
↪     name='output_neurons')
```

(continues on next page)

(continued from previous page)

```
```

rent values of `refP` and `baseweight` to be able to compare them to those generated by adding mismatch (see mismatch distribution plot below). Assuming that mismatch has not been added yet (e.g. if you have just created the neuron population), the values of the selected parameter will be the same for all the neurons in the population. Here we will arbitrarily choose to store the first one.

```
```

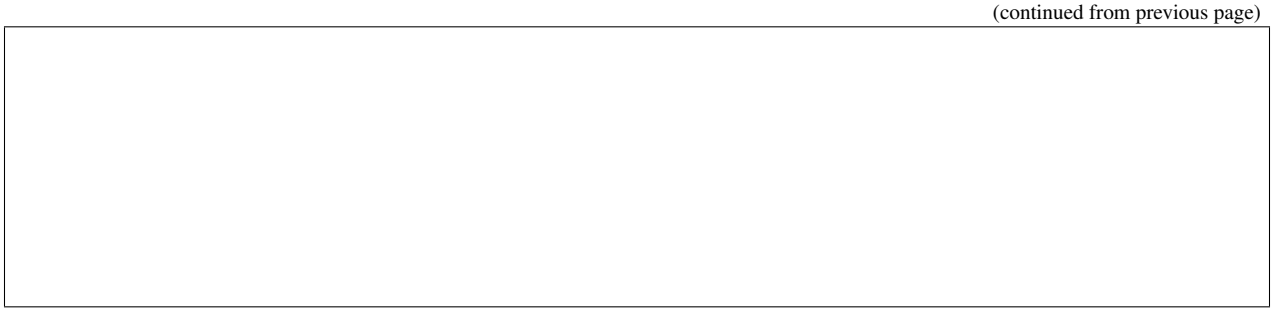
(continues on next page)

(continued from previous page)

to reproduce the same mismatch across multiple simulations and to use the same random distribution across bigger populations, we can also set the `seed`.

(continues on next page)

(continued from previous page)



mismatch on the *EPSC* and on the output membrane current `Imem` of five randomly selected neurons, and the parameter distribution across neurons.

(continues on next page)

(continued from previous page)

```
↪               variables=['Imem'],

↪               record=True,
```

(continues on next page)

(continued from previous page)

```
↪                variables='I_syn',

↪                record=True,
```

(continues on next page)

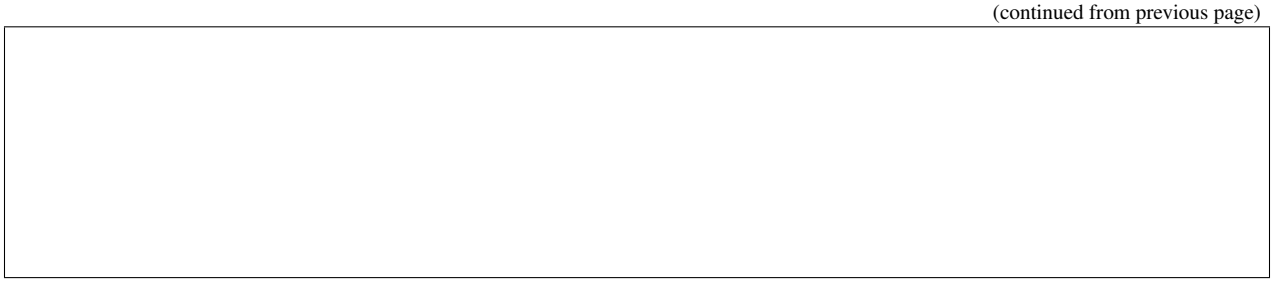
(continued from previous page)

```
↪output,
```

```
↪input_syn)
```

(continues on next page)

(continued from previous page)



(continues on next page)

(continued from previous page)

```
↪           fontsize_legend=12,

↪           fontsize_axis_labels=12,

↪           marker_size=2)
```

(continues on next page)

(continued from previous page)

```
↪per subgroup
```

(continues on next page)

(continued from previous page)

```
↪                    state_variables=[statemon_input_syn.I_syn[neuron_idx]],
```

(continues on next page)

(continued from previous page)

```
↪                        state_variables_times=[statemon_input_syn.t])

↪'))))
```

(continues on next page)

(continued from previous page)

```
↪              state_variables=[statemon_input_syn.I_syn[neuron_ids_to_plot].T],
```

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

```
↪                    state_variables=[statemon_output.Imem[neuron_id].T],
```

(continues on next page)

```
↪                    state_variables_times=[statemon_output.t])
```

(continued from previous page)

```
↪' ) ) )
```

(continues on next page)

(continued from previous page)

```
↪                 state_variables=[statemon_output.Imem[neuron_ids_to_plot].T],

↪                 state_variables_times=[statemon_output.t])
```

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

```
↪  MyPlotSettings=MyPlotSettings,

↪  time_range=[0, duration],
```

(continues on next page)


```
↪  title="Spike generator", xlabel="Time (ms)", ylabel="Neuron ID",
```

(continued from previous page)

```
    backend='pyqtgraph', mainfig=mainfig, subfig_rasterplot=subfig1, QtApp=QtApp,
    show_immediately=False)
```

(continues on next page)

(continued from previous page)

```
↪ time_range=[0, duration],

↪ title="Output layer", xlabel="Time (ms)", ylabel="Neuron ID",
```

(continues on next page)


```
↪ backend='pyqtgraph', mainfig=mainfig, subfig_rasterplot=subfig2, QtApp=QtApp,
```

(continued from previous page)

```
↪ show_immediately=False)

↪
```

(continues on next page)

(continued from previous page)

```
↪"EPSC", xlabel="Time (ms)", ylabel="EPSC (pA)",

↪'pyqtgraph', mainfig=mainfig, subfig=subfig3, QtApp=QtApp,
```

(continues on next page)


```
↪immediately=False)
```

(continued from previous page)

```
↪
↪range=[0, duration],
```

(continues on next page)

(continued from previous page)

```
↪'pyqtgraph', mainfig=mainfig, subfig=subfig4, QtApp=QtApp,
↪immediately=True)
```

(continues on next page)

(continued from previous page)

```
↪                       state_variables=[input_syn_baseweights])  # to pA
```

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

```
↪title='baseweight', xlabel='(pA)', ylabel='count',
```

```
↪backend='pyqtgraph',
```

(continues on next page)


```
↪mainfig=mainfig, subfig=subfig1, QtApp=QtApp,
```

(continued from previous page)

```
↪show_immediately=False)
```

(continues on next page)

(continued from previous page)

```
↪MyPlotSettings=MyPlotSettings,
```

(continues on next page)

(continued from previous page)

```
↪backend='pyqtgraph',

↪mainfig=mainfig, subfig=subfig2, QtApp=QtApp,

↪show_immediately=False)
```

(continues on next page)

(continued from previous page)

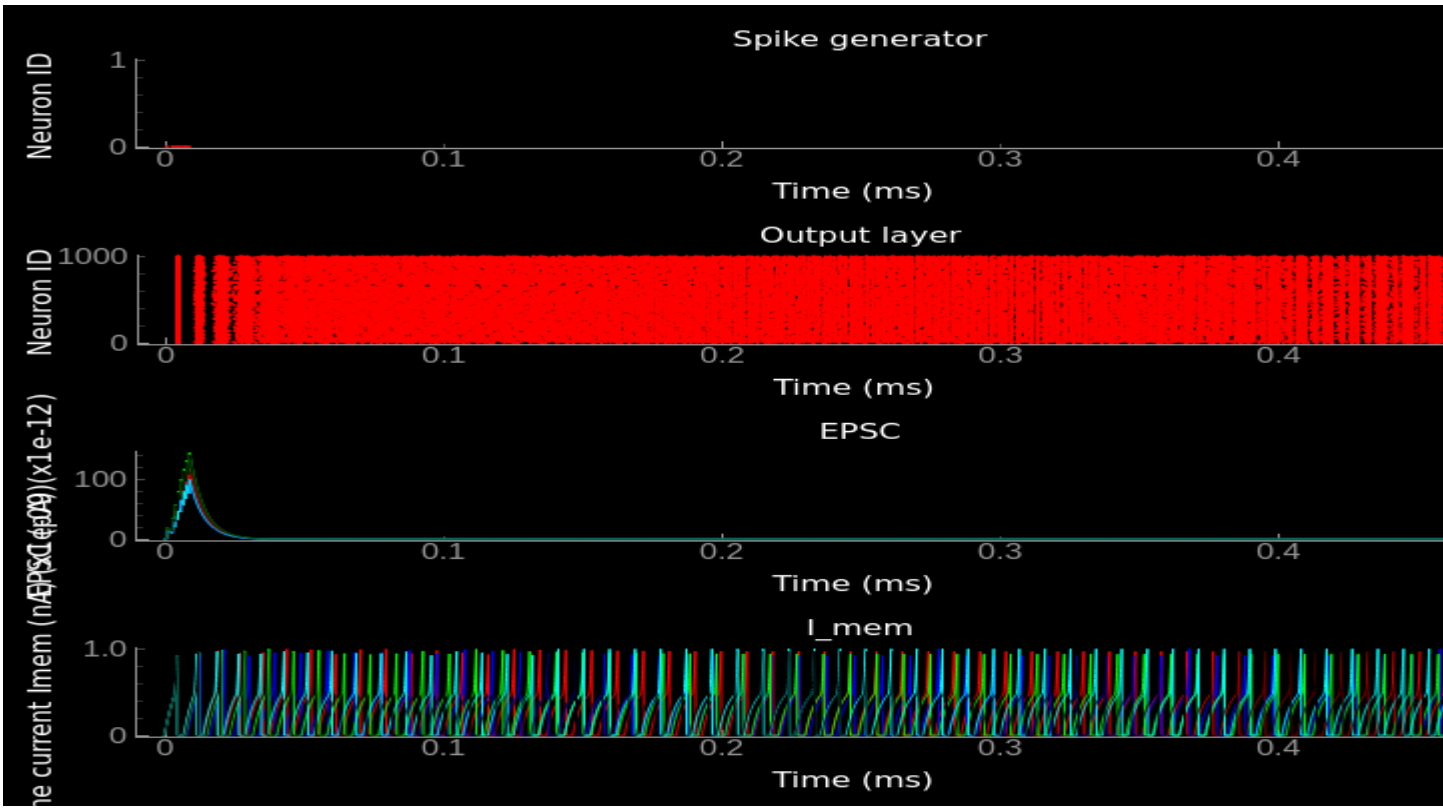
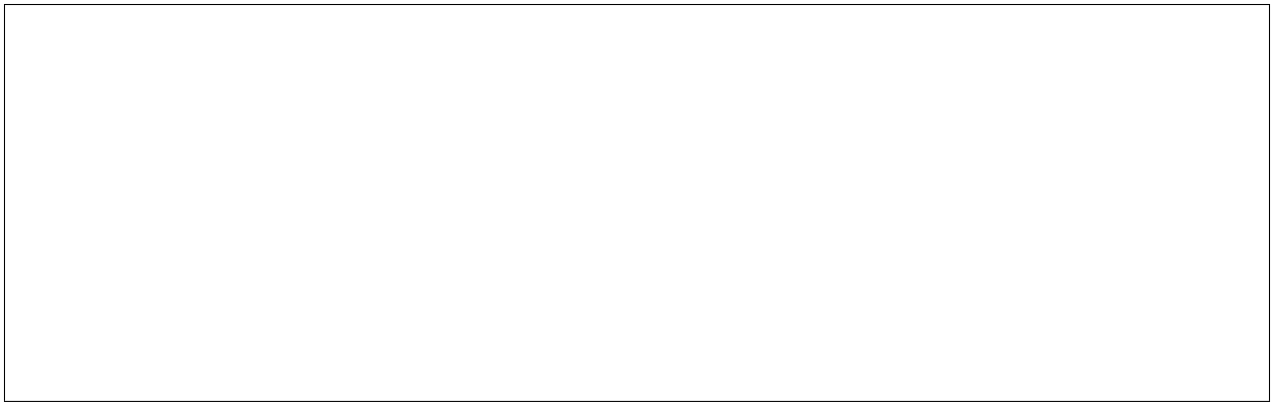
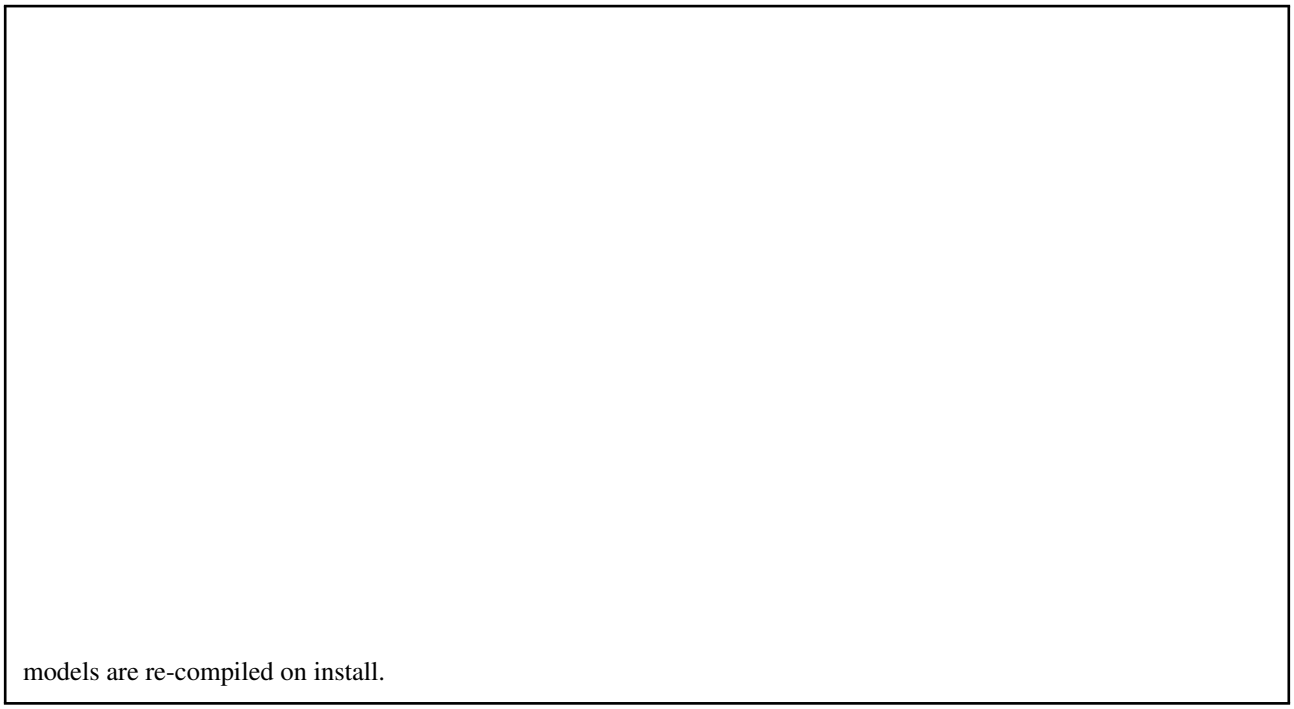


Fig. 6: Effect of mismatch on neuron and synapse dynamics. **Top)** Input spike raster plot. **Top/Middle)** Output spike raster plot. **Bo** EPSC traces of different synapses. Note that the input spike time is the same but the temporal evolution, which is set by the synapt different. **Bottom)** Traces of *Imem* of all neurons in the output population. Note that due to the heterogeneity the spike timing an dynamics differ for different neurons.

`~/teiliApps/equations/`, please execute the following two scripts:

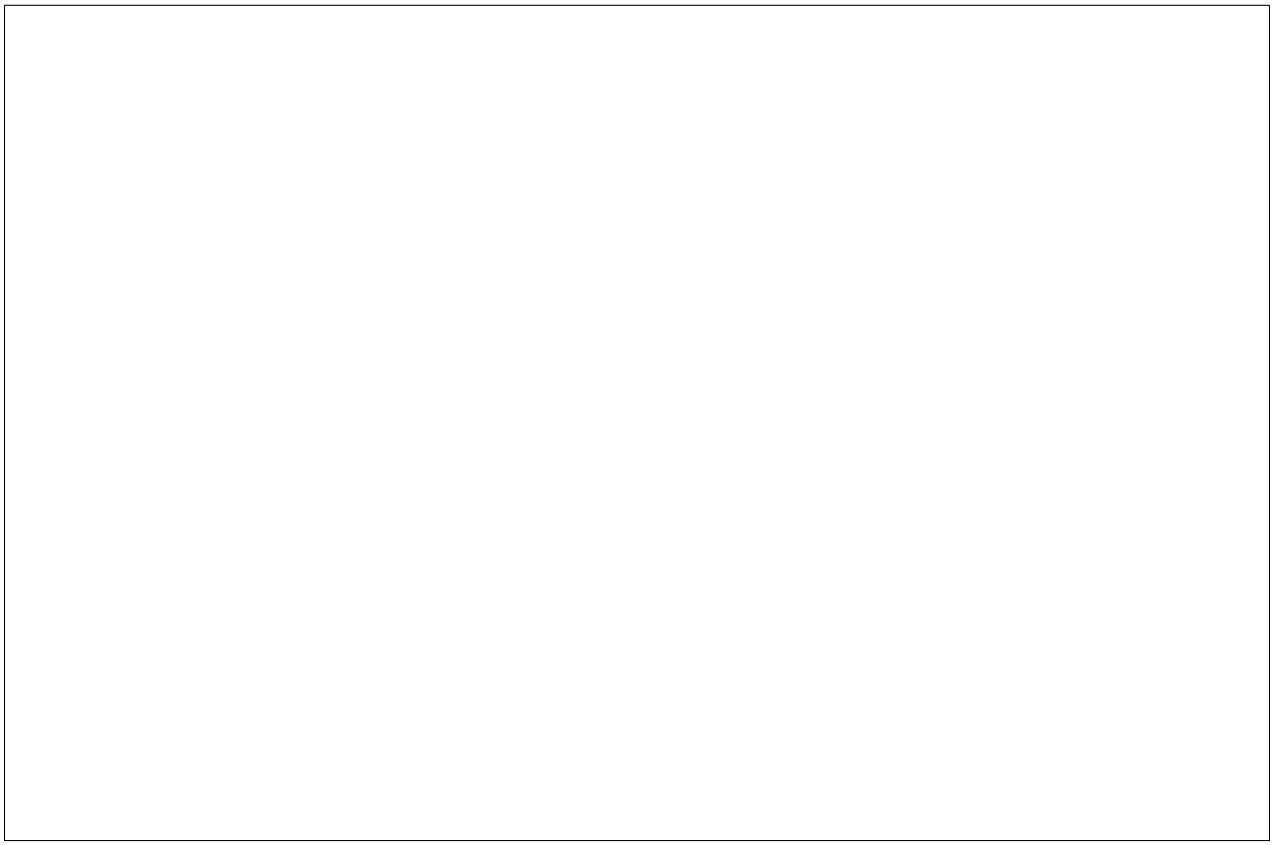
Note: During installation the static models are generated by default. You only need to re-generate them if you manually deleted or changed them and want the default static models.





models are re-compiled on install.

a different location follow the instructions below:



```
~/your_custom_path/.
```

tion: `/path/to/my/equations/teiliApps/equations/`. If you simply call the classes without a path the equations will be placed in `~/teiliApps/equations/`.

CHAPTER

THREE

ADVANCED TUTORIALS

3.1 Online Clustering of Temporal Activity (OCTA)

The `OCTA` network is a `Buildingblock` which implements a model of the canonical microcircuit [1]. The canonical microcircuit, which itself is an abstraction and generalisation of anatomical reality, is found throughout cortical domains, i.e. visual, auditory, motor etc., and throughout the hierarchy, i.e. primary sensory and pre-frontal cortex. The objective of the `OCTA BuildingBlock` is to leverage the temporal information of so-called event-based sensory systems which are completely time-continuous [2]. The temporal information, or the precise timing information of events, is used to extract spatio-temporal correlated patterns from incoming streams of events. But also to learn temporally structured, i.e ordered, sequences of patterns in order to perform temporal predictions of future inputs. A single `OCTA BuildingBlock` consists of two two-dimensional `WTA` networks (compression and prediction) connected in a recurrent manner to a relay (projection) population.

It is inspired by the connectivity between the different layers of the mammalian cortex: every element in the *teili* implementation has a cortical counterpart for which the connectivity and function is preserved:

- compression['n_proj'] : Layer 4
- compression['n_exc'] : Layer 2/3
- prediction['n_exc'] : Layer 5/6

Given a high dimensional input in L2/3, the network extracts in the recurrent connections of L4 a lower dimensional representation of temporal dependencies by learning spatio-temporal features.

As in the other tutorials we start with importing the relevant libraries

```
import sys
import numpy as np

from pyqtgraph.Qt import QtGui
import pyqtgraph as pg
from brian2 import us, ms, prefs, defaultclock, core, float64
import copy as cp

from teili import TeiliNetwork
from teili.building_blocks.octa import Octa
from teili.models.parameters.octa_params import wta_params, octa_params,\
    mismatch_neuron_param, mismatch_synap_param
from teili.models.neuron_models import OCTA_Neuron as octa_neuron
from teili.stimuli.testbench import OCTA_Testbench
from teili.tools.sorting import SortMatrix
from teili.tools.visualizer.DataViewers import PlotSettings
from teili.tools.visualizer.DataControllers.Rasterplot import Rasterplot
from teili.tools.io import monitor_init
```

Now we set the code-generation target as well as the simulation time step. Please note the small `dt`. In order to avoid integration errors and make sure that the timing of spikes/events can be used we need to lower the simulation time step much more than in our other (simpler) tutorials.

Note: The `OCTA` building block can **not** be run in `standalone` mode as it requires quite complicated `run_regularly` functions which are currently not available in c++.

```
prefs.codegen.target = "numpy"
defaultclock.dt = 0.1 * ms
core.default_float_dtype = float64
```

Now we can create the `TeiliNetwork` and load the specifically designed `OCTA_Testbench`.

```
# create the network
Net = TeiliNetwork()
OCTA_net = Octa(name='OCTA_net')

#Input into the Layer 4 block: compression['n_proj']
testbench_stim = OCTA_Testbench()
testbench_stim.rotating_bar(length=10, nrows=10,
                            direction='cw',
                            ts_offset=3, angle_step=10,
                            noise_probability=0.2,
                            repetitions=300,
                            debug=False)

OCTA_net.groups['spike_gen'].set_spikes(indices=testbench_stim.indices,
                                        times=testbench_stim.times * ms)
```

As in the other tutorials we can now add the different `BuildingBlocks` and `sub_blocks` to the network. In order to visualise the input we need to explicitely add the monitor again, as we changed the `Neurons` it is monitoring.

```
Net.add(OCTA_net,
        OCTA_net.monitors['spikemon_proj'],
        OCTA_net.sub_blocks['compression'],
        OCTA_net.sub_blocks['prediction'])

Net.run(np.max(testbench_stim.times) * ms,
        report='text')
```

The simulation will take about 10 minutes. In contrast to other tutorials we only provide a pyqtgraph backend visualisation, as the amount of data is too high and the way we want to look at the spiking activity needs a more sophisticated sub plot arrangement.

```
app = QtGui.QApplication.instance()
if app is None:
    app = QtGui.QApplication(sys.argv)
else:
    print('QApplication instance already exists: %s' % str(app))

pg.setConfigOptions(antialias=True)
labelStyle = {'color': '#FFF', 'font-size': 18}
MyPlotSettings = PlotSettings(fontsize_title=18,
                              fontsize_legend=12,
                              fontsize_axis_labels=14,
                              marker_size=10)
```

(continues on next page)

(continued from previous page)

```
sort_rasterplot = True
win = pg.GraphicsWindow(title="Network activity")
win.resize(1024, 768)
p1 = win.addPlot(title="Spike raster plot: L4")
p2 = win.addPlot(title="Zoomed in spike raster plot: L2/3")
win.nextRow()
p3 = win.addPlot(title="Zoomed in spike raster plot: L5/6",
                 colspan=2)

p1.showGrid(x=True, y=True)
p2.showGrid(x=True, y=True)
p3.showGrid(x=True, y=True)

region = pg.LinearRegionItem()
region.setZValue(10)

p1.addItem(region, ignoreBounds=True)


monitor_p1 = OCTA_net.monitors['spikemon_proj']
monitor_p2 = monitor_init()
monitor_p2.i = cp.deepcopy(np.asarray(
    OCTA_net.sub_blocks['compression'].monitors['spikemon_exc'].i))
monitor_p2.t = cp.deepcopy(np.asarray(
    OCTA_net.sub_blocks['compression'].monitors['spikemon_exc'].t))
monitor_p3 = monitor_init()
monitor_p3.i = cp.deepcopy(np.asarray(
    OCTA_net.sub_blocks['prediction'].monitors['spikemon_exc'].i))
monitor_p3.t = cp.deepcopy(np.asarray(
    OCTA_net.sub_blocks['prediction'].monitors['spikemon_exc'].t))

if sort_rasterplot:
    weights_23 = cp.deepcopy(np.asarray(
        OCTA_net.sub_blocks['compression'].groups['s_exc_exc'].w_plast))
    s_23 = SortMatrix(nrows=OCTA_net.sub_blocks['compression'].groups['s_exc_exc'].
↪source.N,
                      ncols=OCTA_net.sub_blocks['compression'].groups['s_exc_exc'].
↪target.N,
                      matrix=weights_23,
                      axis=1)

    weights_23_56 = cp.deepcopy(np.asarray(
        OCTA_net.sub_blocks['prediction'].groups['s_inp_exc'].w_plast))
    s_23_56 = SortMatrix(nrows=OCTA_net.sub_blocks['prediction'].groups['s_inp_exc'].
↪source.N,
                         ncols=OCTA_net.sub_blocks['prediction'].groups['s_inp_exc
↪'].target.N,
                         matrix=weights_23_56,
                         axis=1)

    monitor_p2.i = np.asarray([np.where(
        np.asarray(s_23.permutation) == int(i))[0][0] for i in monitor_p2.i])
    monitor_p3.i = np.asarray([np.where(
        np.asarray(s_23_56.permutation) == int(i))[0][0] for i in monitor_p3.i])


duration = np.max(testbench_stim.times)
```

(continues on next page)

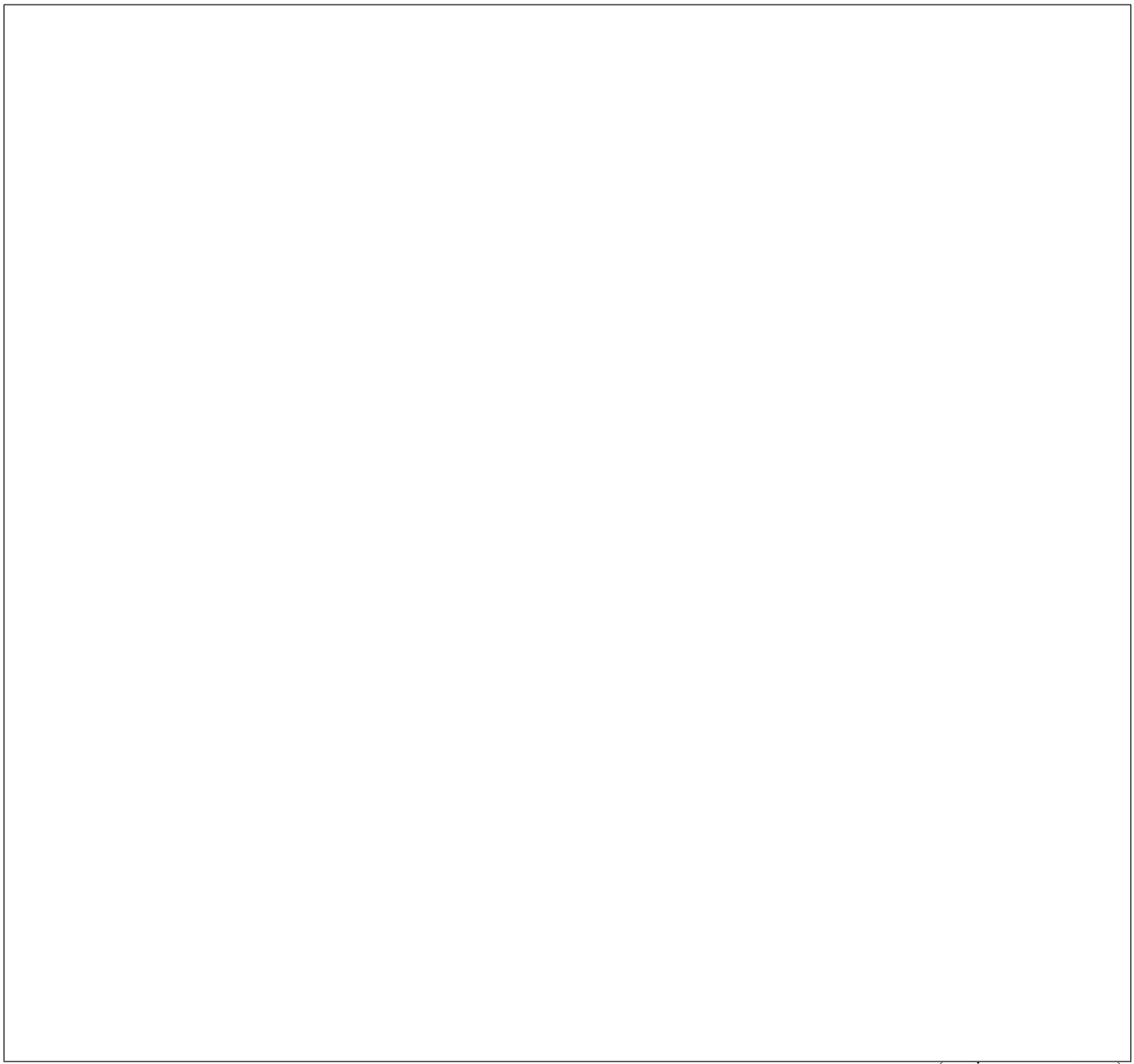
(continued from previous page)

```
Rasterplot(MyEventsModels=[monitor_p1],
            MyPlotSettings=MyPlotSettings,
            time_range=[0, duration],
            neuron_id_range=None,
            title="Input rotating bar",
            xlabel='Time (s)',
            ylabel="Neuron ID",
            backend='pyqtgraph',
            mainfig=win,
            subfig_rasterplot=p1,
            QtApp=app,
            show_immediately=False)

Rasterplot(MyEventsModels=[monitor_p2],
            MyPlotSettings=MyPlotSettings,
            time_range=[0, duration],
            neuron_id_range=None,
            title="Spike raster plot of L2/3",
            xlabel='Time (s)',
            ylabel="Neuron ID",
            backend='pyqtgraph',
            mainfig=win,
            subfig_rasterplot=p2,
            QtApp=app,
            show_immediately=False)

Rasterplot(MyEventsModels=[monitor_p3],
            MyPlotSettings=MyPlotSettings,
            time_range=[0, duration],
            neuron_id_range=None,
            title="Spike raster plot of L5/6",
            xlabel='Time (s)',
            ylabel="Neuron ID",
            backend='pyqtgraph',
            mainfig=win,
            subfig_rasterplot=p3,
            QtApp=app,
            show_immediately=False)

region.sigRegionChanged.connect(update)
p2.sigRangeChanged.connect(updateRegion)
p3.sigRangeChanged.connect(updateRegion)
region.setRegion([29.6, 30])
p1.setXRange(25, 30, padding=0)

app.exec_()
```

The generated plot should look like this:

A, B and C, connected to a hidden two-dimensional `WTA` population H. The role of the hidden population is to encode a relation between A, B and C, which serve as inputs andor outputs.

a relation A + B = C to each other, which is hardcoded into connectivity of the hidden population.



(continues on next page)

(continued from previous page)

(continues on next page)

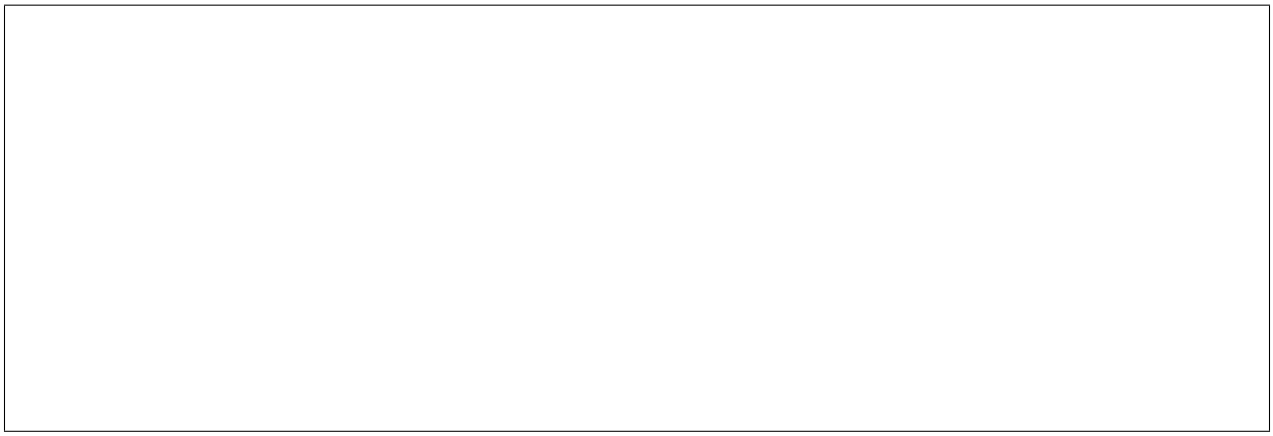
(continued from previous page)

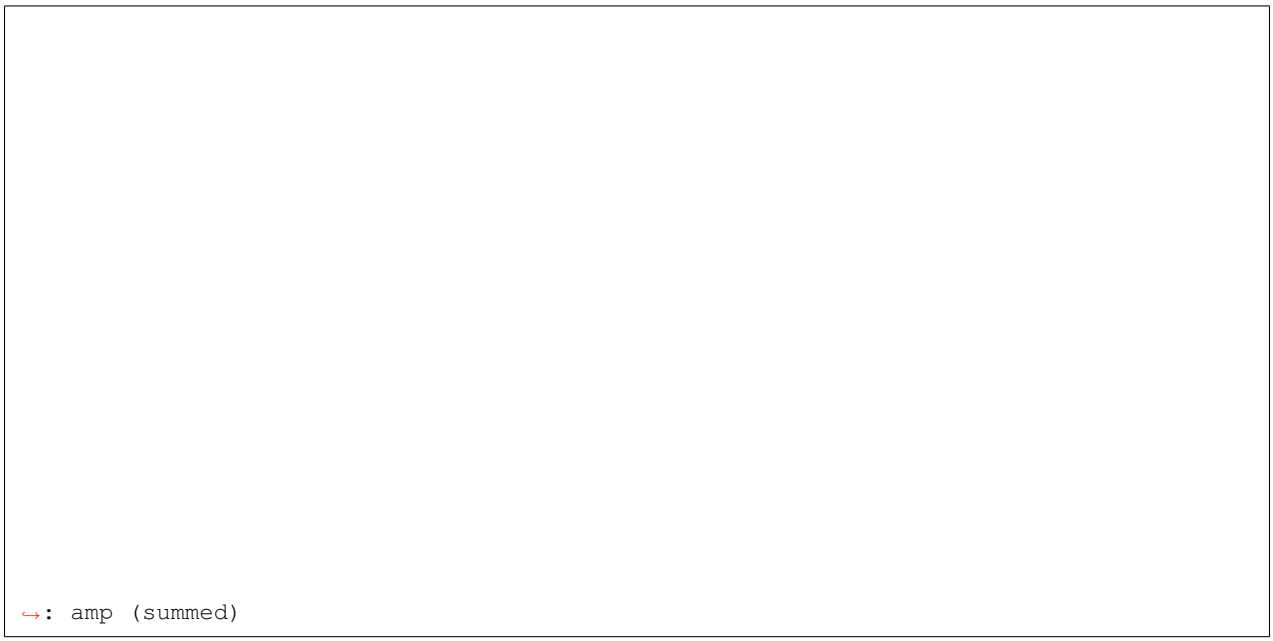
(continues on next page)

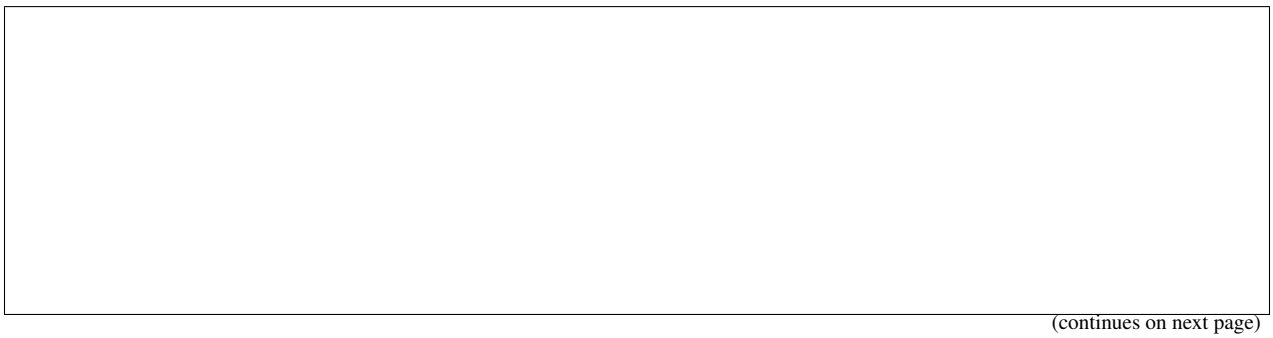
(continued from previous page)

Here we send A=0.2, B=0.4 and activity in population C is inferred via H, shaping in an activity bump encoding ~0.6:

ecuted on a nVidia graphics card. Make sure to change the `DPIsyn` model located in *teiliApps/equations/DPIsyn.py*. To be able to use brian2genn with `TeiliNetwork` change this line:

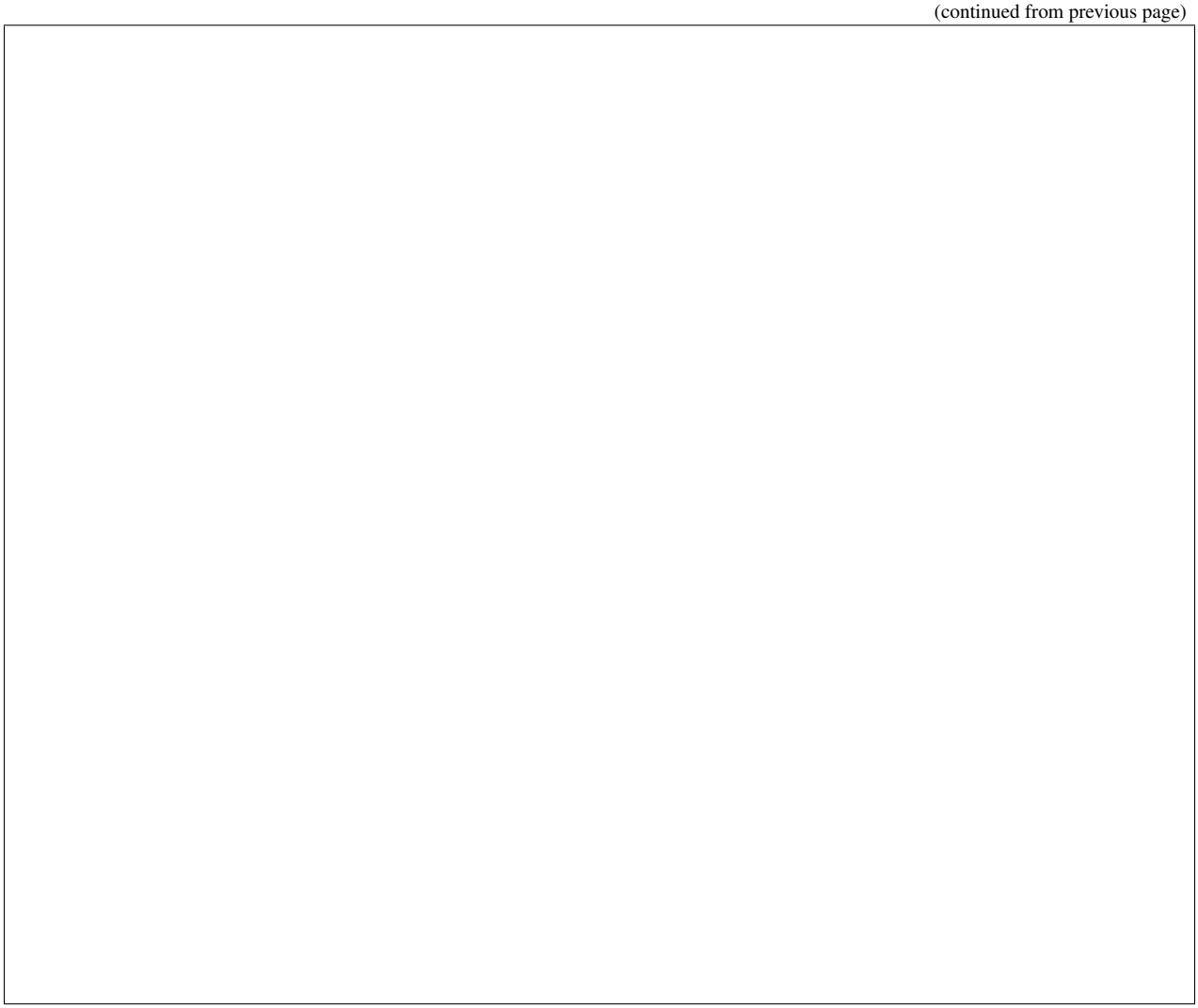






(continues on next page)

(continued from previous page)



(continues on next page)

(continued from previous page)

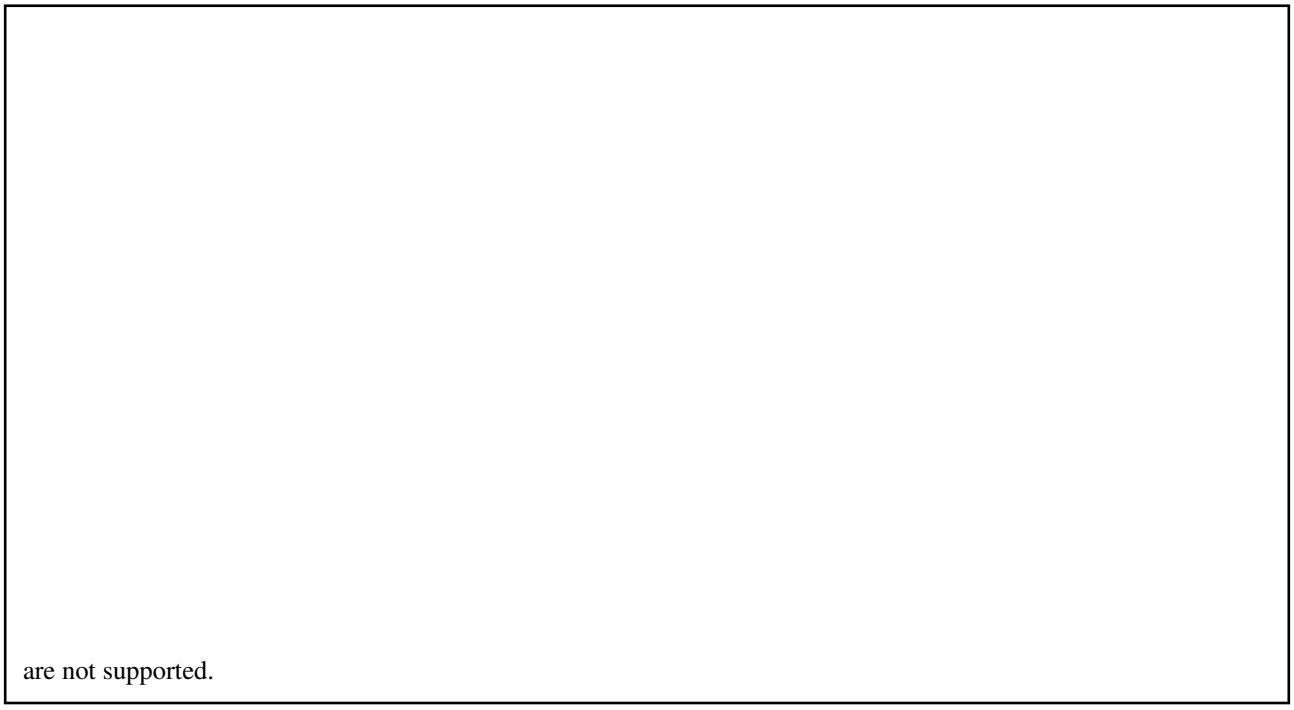
```
↪   : amp
```

```
↪amp
```

(continues on next page)


```
↪((I_gain/I_syn)+1))
```

(continued from previous page)



are not supported.

`teiliApps/tutorials/`. The `TeiliNetwork` is the same as in `neuron_synapse_tutorial` but with the specific commands to use the **genn-backend**.

They are imported as follows:

or subgroup with a line plot.

(continues on next page)

(continued from previous page)

```
```

are the same for the whole group. Or, as shown here in the tutorial, you can also just add a new state variable and copy over the value (or do other things with it) in a run_regularly:

```
```

(continues on next page)

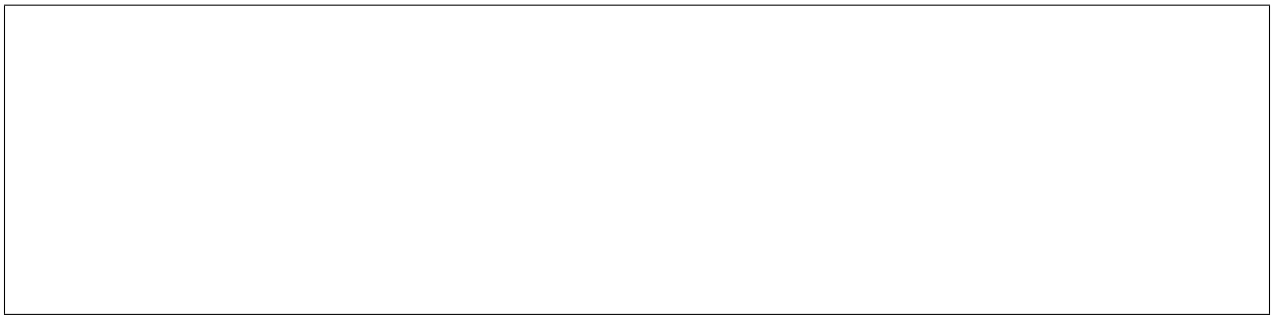
(continued from previous page)

Note: The brian2 simulator is not made for real time plotting. This currently only works with numpy code generation and timestep length can vary.

make a plot of 2d neural activity or DVS recordings. In this tutorials, you are asked to provide a path for 2 .aedat or .npz files that are then plotted next to each other.

and replacing of parameters in the c++ code so we can run the same compiled standalone program several times with different parameters. The tutorial re-implements the sequence learning architecture described by Kreiser et al. 2018. The full tutorial can be found at `~/teiliApps/tutorials/sequence_learning_standalone_tutorial.py`

is built:



piling) and the results can be plotted in between.

ent parameters in a loop.

CHAPTER

FOUR

CORE

4.1 Network

`TeiliNetwork` is a subclass of `brian2.Network`. It does the same thing plus some additional methods for convenience. There are properties to get all `monitors`, `Neurons` and `Connections` that were added to the Network.

Like in *Brian2*, there is an `add` method to which all `Groups` have to be added, for usage, please refer to the *teili* examples (in particular neuron_synapse_tutorial) and to the Brian2 documentation ().

4.2 Groups

4.2.1 Neurons

`Neurons` is a subclass of `brian2.NeuronGroup` and can be used in the same way. Have a look at neuron_synapse_tutorial for an introduction. In teili there are different ways to initialize a `Neurons` object:

```
import os
from teili.core.groups import Neurons
from teili.models.neuron_models import DPI
from teili.models.builder.neuron_equation_builder import NeuronEquationBuilder
# the teili way
G = Neurons(100, equation_builder=DPI(num_inputs=2))
# from a static file
path = os.path.expanduser("~")
model_path = os.path.join(path, "teiliApps", "equations", "")
neuron_model = NeuronEquationBuilder.import_eq(
    filename=model_path + 'DPI.py', num_inputs=2)
G = Neurons(100, equation_builder=neuron_model)
# or the brian2 way
G = Neurons(100, model='dv/dt = -v / tau : 1')
```

As in brian2 we provide a `Neuron` class which inherits from brian2's `NeuronGroup` class. The required keyword arguments are the same as described in brian2's neuron tutorial. See below a example use case of pre-defined `neuron_models`. For static `neuron_model` usage please refer to `teiliApps/tutorials/neuron_synapse_import_eq_tutorial.py`.

```
from teili import Neurons
from teili.models.neuron_models import DPI as neuron_model

```

(continues on next page)

(continued from previous page)

```
test_neurons = Neurons(2, equation_builder=neuron_model(num_inputs=2),
                       name="test_neurons")
```

where **num_inputs** defines how many distinct inputs the `NeuronGroup` is expecting. This allows us to potentially treat each synaptic connection as independent and not to perform a linear summation before each current is injected into the neuron. For many simulations this is an unnecessary feature as most models expect a linear summation of all synaptic inputs. By defining the number of inputs explicitly, however, one can study branch specific inputs with a distribution of synaptic time constants which are asynchronously integrated.

Each model, whether `Neuron` or `Connection` is internally generated dynamically using the EquationBuilder. For more details please refer to NeuronEquationBuilder.

An example of the `neuron_model` class is shown below:

```
class DPI(NeuronEquationBuilder):
    """This class provides you with all equations to simulate a current-based
    exponential, adaptive leaky integrate and fire neuron as implemented on
    the neuromorphic chips by the NCS group. The neuron model follows the DPI neuron
    which was published in 2014 (Chicca et al. 2014).
    """

    def __init__(self, num_inputs=1):
        """This initializes the NeuronEquationBuilder with DPI neuron model.

        Args:
            num_inputs (int, optional): Description
        """
        NeuronEquationBuilder.__init__(self, base_unit='current', adaptation='calcium_
↪feedback',
                                       integration_mode='exponential', leak='leaky',
                                       position='spatial', noise='none')
        self.add_input_currents(num_inputs)
```

The `NeuronEquationBuilder` has the following keyword arguments:

- **base_unit**: Either set to `current` or `voltage` depending whether you want to simulate current-based hardware neuron models
- **adaptation**: Toggles spike-frequency adaptation mechanism in `neuron_model`. Can either be set to `None` or `calcium_feedback`.
- **integration_mode**: Can be set to `linear`, `quadratic` or `exponential`
- **leak**: Toggles leaky integration. Possible values are `leaky` or `non_leaky`.
- **position**: Adds positional x, y attribute to neuron in order to spatially arrange the neurons. Once the `neuron_model` has these attributes the user can access and set them by `neuron_obj.x`.
- **noise**: Adds constant noise to `neuron_model`

The reason behind this is that the `EquationBuilder` has access to a set of templates defined in `teili/models/builder/templates/` such that the same neuron model can easily be simulated with and without leak for example. Of course we offer the possibility of a work-around so that statically defined models can be simulated. For details please refer to the tutorial

For more information please consult the EquationBuilder section. Let's connect neurons to one another.

4.2.2 Connections

The `Connections` class is a subclass of `brian2.Synapses` and can be used in the same way. Have a look at neuron_synapse_tutorial for an introduction. In *teili* there are different ways to initialize a `Connections` object:

```
import os
from teili.core.groups import Connections
from teili.models.synapse_models import DPISyn
from teili.models.builder.synapse_equation_builder import SynapseEquationBuilder
# the teili way
S = Connections(pre_neuron, post_neuron,
                equation_builder=DPISyn(), name="synapse_name")
# from a static file
path = os.path.expanduser("~")
model_path = os.path.join(path, "teiliApps", "equations", "")
synapse_model = = SynapseEquationBuilder.import_eq(
    model_path + 'DPISyn.py')
S = Connections(pre_neuron, post_neuron,
                equation_builder=synapse_model, name="synapse_name")
# or the brian2 way
S = Connections(pre_neuron, post_neuron, model='w : volt', on_pre='v += w')
```

As in brian2 we provide a `Connections` class which inherits from brian2's `Synapses` class. The required keyword arguments are the same as described in brian2's synapse tutorial. See below a example use case of predefined `synapse_models`. For static `synapse_model` usage please refer to `~/teiliApps/tutorials/neuron_synapse_builderobj_tutorial.py`.

```
from teili.core.groups import Neurons, Connections
from teili.models.synapse_models import DPISyn as syn_model

test_synapse = Connections(test_neurons1, test_neurons2,
                           equation_builder=syn_model(),
                           name="test_synapse")
```

Each model, whether `Neuron` or `Connection` is internally generated dynamically using the EquationBuilder. For more details please refer to NeuronEquationBuilder or SynapseEquationBuilder

An example of the `synapse_model` class is shown below:

```
class DPISyn(SynapseEquationBuilder):
    """This class provides you with all the equations to simulate a Differential Pair
    Integrator (DPI) synapse as published in Chicca et al. 2014.
    """

    def __init__(self):
        """This class provides you with all the equations to simulate a Differential␣
↪Pair
        Integrator (DPI) synapse as published in Chicca et al. 2014.
        """
        SynapseEquationBuilder.__init__(self, base_unit='DPI',
                                        plasticity='non_plastic')
```

The `SynapseEquationBuilder` has the following keyword arguments:

- **base_unit**: Set to `current` or `conductance` depending whether you want to simulate current-based hardware neuron models. This keyword argument can also be set to `DPI` or `DPIShunting` for specific hardware model simulation.

- **kernel**: Can be set to `exponential`, `alpha` or `resonant` which ultimately sets the shapes of the EPSC and IPSC.
- **plasticity**: This keyword argument lets you easily generate any `synapse_model` with either an `stdp` or `fusi` learning rule.

The reason behind this is that the `EquationBuilder` has access to a set of templates defined in `teili/models/builder/templates/` such that the same `synapse_model` can easily be simulated with and without plasticity or with different plasticity rules for example. Of course we offer the possibility of a work-around so that statically defined models can be simulated. For details please refer to the plasticity tutorial

Note: TBA Contributing guide for new templates

4.3 Tags

Each `TeiliGroup` has an attribute called `_tags`. For more information please see here for more detailed explanation of how to set and get tags from `Groups`.

Tags should be set as the network expands and the functionality changes. Tags are defined as:

- **mismatch**: (bool) Flag to indicate if mismatch is present in the `Group`
- **noise**: (bool) Noise input, noise connection or noise presence
- **level**: (int) Level of BuildingBlock in the hierarchy. A WTA BuildingBlock which is connected directly to a sensor array is level 1. An OCTA BuildinBlock, however, is level 2 as it consists of level 1 WTAs
- **sign**: (str : exc/inh/None) Sign on neuronal population. Following Dale law.
- **target_sign**: (str : exc/inh/None) Sign of target population. None if not applicable.
- **num_inputs**: (int) Number of inputs in Neuron population. None if not applicable.
- **bb_type**: (str : WTA/ OCTA/ 3-WAY) Building block type.
- **group_type**: (str : Neuron/Connection/ SpikeGen) Group type
- **connection_type**: (str : rec/lateral/fb/ff/None) Connection type

4.3.1 Setting Tags

Tags can be set: .. code-block:: python

    test_wta._set_tags({'custom_tag' : custom_tag }}, target_group)

4.3.2 Getting Tags

Specific groups can be filtered using specific tags:

```
test_wta.get_groups({'group_type': 'SpikeGenerator'})
```

All tags of a group can be obtained by:

```
test_wta.print_tags('n_exc')
```

4.4 Device Mismatch

Mismatch is an inherent property of analog VLSI devices due to fabrication variability[1]. The effect of mismatch on chip behavior can be studied, for example, with Monte Carlo simulations[2]. Thus, if you are simulating neuron and synapse models of neuromorphic chips, e.g. the DPI neuron (DPI) and the DPI synapse (DPISyn), you might also want to simulate device mismatch. To this end, the class method `add_mismatch()` allows you to add a Gaussian distributed mismatch with mean equal to the current parameter value and standard deviation set as a fraction of the current parameter value.

As an example, once `Neurons` and `Connections` are created, device mismatch can be added to some selected parameters (e.g. *Itau* and *refP* for the *DPI neuron*) by specifying a dictionary with parameter names as `keys` and standard deviation as `values`, as shown in the example below. If no dictionary is passed to `add_mismatch()` 20% mismatch will be added to all variables except for variables that are found in `teili/models/parameters/no_mismatch_parameter.py`.

```
import numpy as np
from brian2 import seed
from teili.core.groups import Neurons
from teili.models.neuron_models import DPI

test_neurons = Neurons(100, equation_builder=DPI(num_inputs=2))
```

Let's assume that the estimated mismatch distribution has a standard deviation of 10% of the current value for both parameters. Then:

```
mismatch_param = {'Itau': 0.1, 'refP': 0.1}
test_neurons.add_mismatch(mismatch_param, seed=10)
```

This will change the current parameter values by drawing random values from the specified Gaussian distribution.

If you set the mismatch seed in the input parameters, the random samples will be reproducible across simulations.

Note: Note that `self.add_mismatch()` will automatically truncate the Gaussian distribution

---

at zero for the lower bound. This will prevent neuron or synapse parameters (which are mainly transistor currents for the DPI model) from being set to negative values. No upper bound is specified by default. However, if you want to manually specify the lower bound and upper bound of the mismatch Gaussian distribution, you can use the method `_add_mismatch_param()`, as shown below. With old_param being the current parameter value, this will draw samples from a Gaussian distribution with the following parameters:

- **mean**: old_param
- **standard deviation**: std * old_param
- **lower bound**: lower * std * old_param + old_param
- **upper bound**: upper * std * old_param + old_param

```
import numpy as np
from brian2 import seed
from teili.core.groups import Neurons
from teili.models.neuron_models import DPI
```

(continues on next page)

[1] Sheik, Sadique, Elisabetta Chicca, and Giacomo Indiveri. "Exploiting device mismatch in neuromorphic VLSI systems to implement axonal delays." Neural Networks (IJCNN), The 2012 International Joint Conference on. IEEE, 2012.

[2] Hung, Hector, and Vladislav Adzic. "Monte Carlo simulation of device variations and mismatch in analog integrated circuits." Proc. NCUR 2006 (2006): 1-8.

(continued from previous page)

```
test_neurons = Neurons(100, equation_builder=DPI(num_inputs=2))
test_neurons._add_mismatch_param(param='Itau', std=0.1, lower=-0.2, upper = 0.2)
```

Note: that this option allows you to add mismatch only to one parameter at a time.

CHAPTER

FIVE

MODELS

Teili comes equipped with pre-built neuron and synapse models. Example models can be found in your **teiliApps** folder after successful installation. Below we provide a brief overview of the different keywords and properties. Examples how the pre-built and the dynamic models can be used in your simulations can be found in `teiliApps/tutorials/neuron_synapse_tutorial.py` and `teiliApps/tutorials/neuron_synapse_import_eq_tutorial.py`

Note: TBA detailed explanation of statically defined neuron and synapse models which can be found in `teiliApps/equations/`

CHAPTER

SIX

BUILDING BLOCKS

The core of the motivation for the development of teili was to provide users with a toolbox to easily build and combine neural `BuildingBlocks` which represent basic algorithms implemented using neurons and synapses. In order to provide these functionalities, all `BuildingBlocks` share the same parent class which, amongst other things, provides I/O groups and properties to combine `BuildingBlocks` hierarchically.

6.1 BuildingBlock

Every `BuildingBlock` has a set of parameters such as weights and refractory period, which can be specified outside the `BuildingBlock``generation in a dictionary and are unpacked to the ``BuildingBlock` upon creation.. Each `BuildingBlock` has the following attributes:

Attributes:

- **name** (str, required): Name of the building_block population
- **neuron_eq_builder** (class, optional): neuron class as imported from models/neuron_models
- **synapse_eq_builder** (class, optional): synapse class as imported from models/synapse_models
- **params** (dictionary, optional): Dictionary containing all relevant parameters for each building block
- **debug** (bool, optional): Flag to gain additional information
- **groups** (property): Class property to collect all keys to all neuron and synapse groups
- **monitors** (dictionary): Keys to all spike and state monitors
- **monitor** (bool, optional): Flag to auto-generate spike and state monitors
- **standalone_params** (dictionary): Dictionary for all parameters to create a standalone network
- **sub_blocks** (dictionary): Dictionary for all children building blocks
- **input_groups** (dictionary): Dictionary containing all possible groups which are potential inputs
- **output_groups** (dictionary): Dictionary containing all possible groups which are potential outputs
- **hidden_groups** (dictionary): Dictionary containing all remaining groups which are neither inputs nor outputs

And as each `BuildingBlock` inherits from this parent class, all `BuildingBlocks` share the same attributes and properties. To assure this every `BuildingBlock` initialises the `BuildingBlock` class:

```
BuildingBlock.__init__(self,
                       name,
                       neuron_eq_builder,
                       synapse_eq_builder,
                       block_params,
```

(continues on next page)

(continued from previous page)

```
                        debug,
                        monitor)
```

Furthermore, as described above, as soon the parent class is initialised, each `BuildingBlock` has a set of dictionaries which handle I/O to other `Neuron` and `Connection` groups or `BuildingBlocks`.

The `BuildingBlock` class comes with a set of `__setter__` and `__getter__` functions for collecting all `groups` involved or identifying a subset of groups which share the same tags

To retrieve all `Neurons`, `Connections`, `SpikeGeneratorGroups` etc. simply call the `groups` property:

```
test_wta= WTA(name='test_wta', dimensions=1, num_neurons=16, debug=False)
bb_groups = test_wta.groups
```

6.2 Tags

Each `TeiliGroup` has an attribute called `_tags`. The idea behind the `_tags` are that the user can easily define a dictionary and use this dictionary to obtain all `TeiliGroups` which share the same `_tags`. Tags are defined as:

- **mismatch**: (bool) Mismatch present of group
- **noise**: (bool) Noise input, noise connection or noise presence
- **level**: (int) Level of BuildingBlock in the hierarchy.
- **sign**: (str : exc/inh/None) Sign on neuronal population. Follows Dale law.
- **target sign**: (str : exc/inh/None) Sign of target population. None if not applicable.
- **num_inputs**: (int) Number of inputs in Neuron population. None if not applicable.
- **bb_type**: (str : WTA/ OCTA/ 3-WAY..) Building block type.
- **group_type**: (str : Neuron/Connection/ SpikeGen) Group type
- **connection_type**: (str : rec/lateral/fb/ff/None) Connection type

6.2.1 Setting Tags

Tags can be set using an entire dictionary. See tags for additional information.

```
test_wta = WTA(name='test_wta', dimensions=1, num_neurons=16, debug=False)
target_group = test_wta._groups['n_exc']
basic_tags_empty = {'mismatch' : 0,
                    'noise' : 0,
                    'level': 0 ,
                    'sign': 'None',
                    'target sign': 'None',
                    'num_inputs' : 0,
                    'bb_type' : 'None',
                    'group_type' : 'None',
                    'connection_type' : 'None',
                    }

test_wta._set_tags(basic_tags_empty, target_group)
```

and updated:

```
test_wta._tags['mismatch'] = True
```

6.2.2 Getting Tags

Specific groups can be filtered using specific tags:

```
test_wta.get_groups({'group_type': 'SpikeGenerator'})
```

All tags of a group can be obtained by:

```
test_wta.print_tags('n_exc')
```

6.3 Winner-takes-all (WTA)

For the WTA `BuildingBlock` the parameter dictionary looks as follows:

```
wta_params = {'we_inp_exc': 1.5,
              'we_exc_inh': 1,
              'wi_inh_exc': -1,
              'we_exc_exc': 0.5,
              'sigm': 3,
              'rp_exc': 3 * ms,
              'rp_inh': 1 * ms,
              'ei_connection_probability': 1,
              'ie_connection_probability': 1,
              'ii_connection_probability': 0}
```

where each key is defined as:

- **we_inp_exc**: Excitatory synaptic weight between input SpikeGenerator and excitatory neurons.
- **we_exc_inh**: Excitatory synaptic weight between excitatory population and inhibitory interneuron.
- **wi_inh_exc**: Inhibitory synaptic weight between inhibitory interneurons and excitatory population.
- **we_exc_exc**: Self-excitatory synaptic weight.
- **wi_inh_inh**: Self-inhibitory weight of the interneuron population.
- **sigm**: Standard deviation in number of neurons for Gaussian connectivity kernel.
- **rp_exc**: Refractory period of excitatory neurons.
- **rp_inh**: Refractory period of inhibitory neurons.
- **ei_connection_probability**: Excitatory to interneuron connectivity probability.
- **ie_connection_probability**: Interneuron to excitatory connectivity probability
- **ii_connection_probability**: Interneuron to Interneuron connectivity probability.

Now we can import the necessary modules and build our building block.

```
from teili.building_blocks.wta import WTA
from teili.models.neuron_models import DPI
```

6.3.1 1 Dimensional WTA

The WTA `BuildingBlock` comes in two slightly different versions. The versions only differ in the dimensionality of the WTA.

```
# The number of neurons in your WTA population.
# Note that this number is squared in the 2D WTA
num_neurons = 50
# The number of neurons which project to your WTA.
# Note that this number is squared in the 2D WTA
num_input_neurons = 50
my_wta = WTA(name='my_wta', dimensions=1,
             neuron_eq_builder=DPI,
             num_neurons=num_neurons, num_inh_neurons=int(num_neurons/4),
             num_input_neurons=num_input_neurons, num_inputs=2,
             block_params=wta_params,
             monitor=True)
```

6.3.2 2 Dimensional WTA

To generate a 2-dimensional WTA population you can do the following:

```
# The number of neurons in your WTA population.
# Note that this number is squared in the 2D WTA
num_neurons = 7
# The number of neurons which project to your WTA.
# Note that this number is squared in the 2D WTA
num_input_neurons = 10
my_wta = WTA(name='my_wta', dimensions=2,
             neuron_eq_builder=DPI,
             num_neurons=num_neurons, num_inh_neurons=int(num_neurons**2/4),
             num_input_neurons=num_input_neurons, num_inputs=2,
             block_params=wta_params,
             monitor=True)
```

Attention: The generation of the 2D WTA internally squares the number of neurons specified in `num_neurons` only for the excitatory population, **not** for the inhibitory population.

Changing a certain `Connections` group from being *static* to *plastic*:

```
from teili.core.groups import Connections
from teili.models.synapse_models import DPIstdp
my_wta._groups['s_exc_exc'] = Connections(my_wta._groups['n_exc'],
                                          my_wta._groups['n_exc'],
                                          equation_builder=DPIstdp
                                          method='euler',
                                          name=my_wta._groups['s_exc_exc'].name)
my_wta._groups['s_exc_exc'].connect(True)
```

Now we replaced the standard DPI synapse for the recurrent connection within a WTA population with an All-to-All STDP-based DPI synapse. In order to initialize the plastic weight `w_plast` we need to do:

```
my_wta._groups['s_exc_exc'].weight = 45
my_wta._groups['s_exc_exc'].namespace.update({'w_mean': 0.45})
```

(continues on next page)

(continued from previous page)

```
my_wta._groups['s_exc_exc'].namespace.update({'w_std': 0.35})
# Initializing the plastic weight randomly
my_wta._groups['s_exc_exc'].w_plast = 'w_mean + randn() * w_std'
```

6.4 Chain

Note: TBA by Alpha Renner

6.5 Sequence learning

Note: TBA by Alpha Renner

6.6 Threeway network

`Threeway` block consists of three 1D `WTA` blocks and one 2D `WTA`, thus no additional parameters are passed in the `block_params` dictionary, only the ones needed to configure the `WTA`.

To initialize the block provide it with the connectivity pattern in the hidden layer and the cutoff setting used for all `WTA` blocks:

```
from teili.building_blocks.threeway import Threeway
from teili.tools.three_way_kernels import A_plus_B_equals_C
TW = Threeway('TestTW',
              hidden_layer_gen_func = A_plus_B_equals_C,
              cutoff = 2,
              monitor=True)
```

Note: You always have to set **monitor** to **True** to be able to use the method **get_values()** to calculate the population vectors.

In addition to standard `BuildingBlock` arguments you can also specify these optional parameters:

- **num_input_neurons** (int): Sizes of input/output populations A, B and C
- **num_hidden_neurons** (int): Size of the hidden population H
- **hidden_layer_gen_func** (function): A function providing connectivity pattern

A list of attributes available specific to the block:

- **A**, **B** and **C** (WTA): Shortcuts for input/output 1d `WTA` building blocks
- **H** (WTA): A shortcut for the hidden population H implemented with a 2d `WTA` building block
- **Inp_A**, **Inp_B**, **Inp_C** (PoissonGroup): Shortcuts to input spike generators
- **value_a**, **value_b**, **value_c** (double): Population vector values decoded with **get_values()** input for A, B and C

`Threeway` class also implements the following methods unique to the block:

- **set_A(float)**, **set_B(float)** and **set_C(float)**: Sets spiking rates of neurons of the PoissonGroup `Inp_A`, `Inp_B` and `Inp_C`, respectively, to satisfy a shape of a gaussian bump centered at 'value' between 0 and 1
- **reset_A()**, **reset_B()** and **reset_C()**: Resets spiking rates of the neurons of the respective `PoissonGroup` s to zero (e.g. turns the inputs off)
- **reset_inputs()**: turns all three inputs off
- **get_values(ms)**: Extracts and updates encoded values of A, B and C from the spiking rates of the corresponding populations. Must be called to get the numerical results
- **plot()**: calls a preconfigured instance of the `Visualizer` to plot the raster for populations A, B and C

6.7 Online Clustering of Temporal Activity (OCTA)

Online Clustering of Temporal Activity (OCTA) is a second generation `BuildingBlock`: it uses multiple WTA networks recurrently connected to create a cortex-inspired microcircuit that, leveraging the spike timing information, enables investigations of emergent network dynamics [1] (Download).

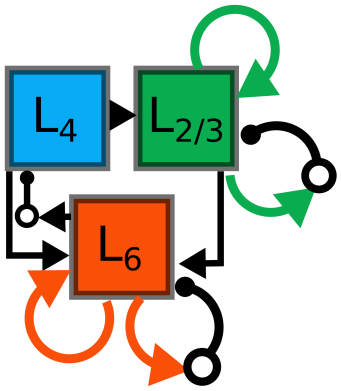


Fig. 1: Schematic overview of a single OCTA `BuildingBlock`

The basic OCTA module consists of a projection (L4), a clustering (Layer2/3) and a prediction (L5/6) sub-module. Given that all connections are subject to learning, the objective of one OCTA module is to continuously adjust its parameters, e.g. synaptic weights and time constants, based on local information to best capture the spatio-temporal statistics of its input.

Parameters for the network are stored in two dictionaries located in `tools/octa_tools/octa_params.py`

The WTA keys are explained above, the OCTA keys are defined as:

- **duration** (int): Duration of the simulation.
- **revolutions** (int): Number of times input is presented.
- **num_neurons** (int): Number of neurons in the compression WTA group. Keep in mind OCTA uses 2D WTAs.
- **num_input_neurons** (int): Number of neurons in the projection and prediction WTA.
- **distribution** (bool): Distribution from which to initialize the weights. Gamma (1) or normal (0) distributions.
- **dist_param_init** (int): Shape for gamma distribution or mean of Gaussian distribution to be used at initialisation.
- **scale_init** (int): Scale for gamma distribution or std of normal distribution.

- **dist_param_re_init** (int): Shape of gamma distribution or mean of normal distribution used during the run regular functions.
- **scale_re_init** (int): Scale for gamma distribution or std of normal distribution used during the run regular functions.
- **re_init_threshold** (float): Parameter between 0 and 1.0. The weights gets reinitialized if the mean weight of a synapse is below the given value or above `1 - re_init_threshold`.
- **buffer_size_plast** (int): Length of the buffer used by the activity dependent plasticity (ADP) mechanism. ADP acts as homeostatic regulariser.
- **noise_weight** (int): Synaptic weight the PoissonSpikeGenerator which injects noise to the network.
- **variance_th_c** (float): Variance threshold for the compression group. Parameter included in the `activity` synapse template used for ADP.
- **variance_th_p** (float): Variance threshold for the prediction group. Parameter included in the `activity` synapse template used for ADP.
- **learning_rate** (float): Learning rate.
- **inh_learning_rate** (float): Inhibitory learning rate.
- **decay** (int): Decay parameter of the decay in the activity dependent run_regular.
- **seed** (int): Seed for mismatch. Default is 42.
- **tau_stdp** (int): Time constant in ms that defines the STDP plasticty.

Initialisation of the building block goes as follows:

```
from brian2 import ms
from teili import TeiliNetwork
from teili.building_blocks.octa import Octa
from teili.models.parameters.octa_params import wta_params, octa_params
from teili.models.neuron_models import OCTA_Neuron as octa_neuron
from teili.stimuli.testbench import OCTA_Testbench

 Net = TeiliNetwork()

 OCTA =  Octa(name='OCTA',
              wta_params=wta_params,
              octa_params=octa_params,
              neuron_eq_builder=octa_neuron,
              num_input_neurons=10,
              num_neurons=7,
              external_input=True,
              noise=True,
              monitor=True,
              debug=False)

testbench_stim = OCTA_Testbench()

testbench_stim.rotating_bar(length=10, nrows=10,
                            direction='cw',
                            ts_offset=3, angle_step=10,
                            noise_probability=0.2,
                            repetitions=90,
                            debug=False)

OCTA_net.groups['spike_gen'].set_spikes(indices=testbench_stim.indices,
```

(continues on next page)

(continued from previous page)

```
                                        times=testbench_stim.times * ms)

Net.add(
        OCTA_net,
        OCTA_net.sub_blocks['prediction'],
        OCTA_net.sub_blocks['compression']
        )

Net.run(octa_params['duration']*ms, report='text')
```

Attention: When `Neurons` or `Connections` groups of a `BuildingBlock` are changed from their default, one needs to `add` the affected `sub_blocks` explicitly.

The additional keyword arguments are defined as:

- **external_input**: Flag to include an input to the network
- **noise**: Flag to include 10 Hz Poisson noise generator on `n_exc` of compression and prediction
- **monitor**: Flag to return monitors of the network
- **debug**: Flag for verbose debug

CHAPTER

SEVEN

EQUATION BUILDER

The equation builder serves as a dynamic model generator. It takes model templates, located in `teili/models/builder/templates/` and combines these template snippets using `teili/models/builder/combine.py`.

There are two distinct equation builder classes:

- `NeuronEquationBuilder`
- `SynapseEquationBuilder`

Each builder is wrapped by a neuron/synapse model generator class located in `teili/models/`:

- `neuron_model`
- `synapse_model`

7.1 Keyword arguments for builder

In order to generate a neuron/synapse model, its builder needs to be initialized by specifying a `base_unit` and a set of values which will define the model itself and thus which template equation/parameters are combined.

The values that determine the model should be passed by defining a keyword which explains the functionality.

7.1.1 NeuronEquationBuilder keywords

```
from teili.models.builder.neuron_equation_builder import NeuronEquationBuilder
num_inputs = 2
my_neuron_model = NeuronEquationBuilder.__init__(base_unit='current',
                                                 adaptation='calcium_feedback',
                                                 integration_mode='exponential',
                                                 leak='leaky',
                                                 position='spatial',
                                                 noise = 'None')
my_neuron.add_input_currents(num_inputs)
```

The keywords used in the example and the values are explained below:

- **base_unit**: Indicates whether the neuron model is `current` or `voltage` based.
- **adaptation**: Determines what type of adaptive feedback should be used. Can be `calciumfeedback` or `None`.
- **integration_mode**: Determines how the neuron integrates up to spike-generation. Can be `linear` or `exponential`.

- **leak**: Enables leaky integration. Can be `leaky` or `non_leaky`.
- **position**: To enable spatial-like position indices (x, y) about the position of a neuron in space. Can be `spatial` or `None`.
- **noise**: Determines what type of distribution is used to inject noise into the neuron. Can be `gaussian` or `None`.

Custom keywords (such as gain_modulation or activity_modulation) can be added by defining a custom equation template in `teili/models/builder/templates/neuron_templates.py` and adding the keyword to either the `current_equation_sets` or to the `voltage_equation_sets` dictionary. When defining a new neuron model, import the new feature by passing the newly constructed keyword to the `NeuronEquationBuilder`.

Note: The `Neurons` class a `num_inputs` property. This allows the user to define how many different afferent connections this particular neuron population has to expect. The default is set to 1. Do not define more inputs than you expect. See below for how to use this property.

7.1.2 SynapseEquationBuilder keywords

```
from teili.models.builder.synapse_equation_builder import SynapseEquationBuilder
my_synapse_model = SynapseEquationBuilder.__init__(base_unit='DPI',
                                                   plasticity='non_plastic')
```

The keywords used in the example and the values are explained below:

- **base_unit**: Indicates whether synapse uses `current`, `conductance` or `DPI` current models.
- **kernel**: Specifies temporal kernel with which each spike gets convolved. Can be `exponential`, `resonant` or `alpha`.
- **plasticity**: Plasticity algorithm for the synaptic weight. Can either be `non_plastic`, `fusi` or `stdp`.

Custom keywords (such as new learning rules or new kernels) can be added by defining a custom equation template in `teili/models/builder/templates/synapse_templates.py` and adding the keywords to the `synaptic_equations` dictionary. When defining a new synapse model, import the new feature by passing the newly constructed keyword to the `SynapseEquationBuilder`.

Equations that do not fit into the existing synaptic modes: *current*, **conductance**, **DPI**, **DPI shunting** can be grouped into the **unit_less** mode and the equation needs to be added to the **unit_less_parameters** dictionary.

7.2 Dictionary structure

Both `EquationBuilders` a dictionary attribute, the keys of which represent the keywords necessary to generate a neuron or synapse model in order to simulate it using *brian2*. The keywords given to the EquationBuilder class are used to select template dictionaries which are combined. This is done by passing these keywords to `current_equation_sets` and `current_parameters` in the case of neurons and to `modes`, `kernels`, `plasticity_models` and `current_parameters` in the case of synapses.

```
# In the case of neurons
keywords = combine_neu_dict(eq_templ, param_templ)
# In the case of synapses
keywords = combine_syn_dict(eq_tmpl, param_templ)
```

7.2.1 Neuron model keywords

The dictionary `keywords` has the following keys:

```
keywords = {'model': keywords['model'],
            'threshold': keywords['threshold'],
            'reset': keywords['reset'],
            'refractory': 'refP',
            'parameters': keywords['parameters']}
```

7.2.2 Synapse model keywords

The dictionary `keywords` has the following keys:

```
keywords = {'model': keywords['model'],
            'on_pre': keywords['on_pre'],
            'on_post': keywords['on_post'],
            'parameters': keywords['parameters']}
```

7.3 Class methods

7.3.1 import_eq

A function to import pre-defined neuron_model. This function can load a dictionary and its keywords in order to initialize the `EquationBuilder`.

```
from teili.models.builder.neuron_equation_builder import NeuronEquationBuilder

my_neu_model = NeuronEquationBuilder.import_eq(
    '~/teiliApps/equations/DPI', num_inputs=2)
```

where `num_inputs` specifies how many distinct neuron populations project to the target population.

For synapses the import works as follows:

```
from teili.models.builder.synapse_equation_builder import SynapseEquationBuilder

my_syn_model = SynapseEquationBuilder.import_eq(
    'teiliApps/equations/DPISyn')
```

7.3.2 export_eq

In order to generate models which can later be changed manually and imported again, the `EquationBuilder` class features an export method which can be used as follows:

```
path = '/home/YOU/teiliApps/equations/'
DPI = NeuronEquationBuilder(base_unit='current', adaptation='calcium_feedback',
                            integration_mode='exponential', leak='leaky',
                            position='spatial', noise='none')
DPI.add_input_currents(num_inputs)
DPI.export_eq(os.path.join(path, "DPI"))
```

For synapse models:

```
path = '/home/YOU/teiliApps/equations/`)
dpi_syn = SynapseEquationBuilder(base_unit='DPI',
                                 plasticity='non_plastic')

dpi_syn.export_eq(os.path.join(path, "DPISyn"))
```

Note: The path can be any existing path. You do not need to store your models within the teiliApps directory.

7.3.3 var_replacer

This function takes two equation sets in the form of strings and replaces all lines which start with '%'.

```
'%x = theta' --> 'x = theta'
'%x' --> ''
```

This feature allows equations that we don't want to compute to be removed from the template by writing '%[variable]' in the other equation blocks.

To replace variables and lines:

```
from teili.models.builder.combine import var_replacer
var_replacer(first_eq, second_eq, params)
```

CHAPTER

EIGHT

DEVELOPING EQUATION TEMPLATES

For using existing models please refer to our neuron and synapse tutorials If you need a different model or you want to test a new idea it easy to use all of *teili's* functionality with your custom model. There are two ways to test, develop and use your custom model

1. Defining a static static dictionary
2. Defining a set of templates

The first way allows you to quickly define a model inside a *.py* file as a dictionary. This way you can easily debug and test your new idea with fiddling with the *EquationBuilder* class. You can use the *import_eq* method to import your custm model into *teili*. See **below_** for more details of how to create your custom dictionary. The second way is to define a template of your model and use the *EquationBuilder* class to dynamically build your model. This is a bit more tricky, but has the advantage that others can potentially use your model in the future. The second advantage is that you might just want to add a custom learning rule or a set of additional equations to calculate some proxy values, required for learning. So you intend to use the existing model, but want to change it's e.g. learning dynamics. 'Teili` provides a *combine_equation* method exactly for this case. For more details of how to do so, please see here

8.1 Defining static models (import_eq)

To create a custom model (**locally**) please follow these steps:

1. Create a file in your desired directory: `my/awesome/path/<my_cool_model.py>`
2. Import all necessary units from *brian2*
3. Make a dictionary with the same name as your file: `my_cool_model = {}`
4. Create four keys in case of a neuron model (*model*, *threshold*, *reset* and *parameters*) or in case of a synapse model (*model*, *on_pre*, *on_post* and *parameters*).

> **Attention:** The name of the file and the name of the dictionary need to be same (without the *.py* extension).

The file (`my/awesome/path/<my_cool_model.py>`) should now look like this for **neuron model**:

```
from brian2.units import *

my_cool_model = {
    'model':'''
        ''',
    'threshold':'''
        ''',
```

(continues on next page)

(continued from previous page)

```
    'reset':'''
        ''',
    'parameters':{
        'refP' : '0.*second',
        }
    }
```

Attention: The neuron parameters dictionary **needs** at least a entry for the refractory period `refP`.

The file (`my/awesome/path/<my_cool_model.py>`) should look like this for a **synapse model**

```
from brian2.units import *

my_cool_model = {
    'model':'''
        ''',
    'on_pre':'''
        ''',
    'on_post':'''
        ''',
    'parameters':{
        }
    }
```

Once you filled your dictionary with your model and/or standard model + custom equations, you can use the `import_eq` method to start using it.

```
from teili.core.groups import Neurons
from teili.models.builder.neuron_equation_builder import NeuronEquationBuilder

my_cool_neuron_model = NeuronEquationBuilder.import_eq('my/awesome/path/<my_cool_
↪model.py>')
N = Neurons(1, equation_builder=my_cool_neuron_model,  name='my_cool_neuron' )
```

Note: Please make sure you remove '<' and '>' from your strings.

8.2 Create new templates for dynamic model generation

As described above the second way to create and use your custom model is to extend the provided neuron/synapse templates and the `neuron_models` or `synapse_models` respectively.

Attention: To add new templates you have to fork and clone the repository. Details of how to contribute are given below. We highly recommend building and test your custom model using the static model import method described above, before fiddling with the dynamic model generation.

8.2.1 Neuronal templates

Navigate to the template sub-directory (`teili/models/builder/templates`) and open `neuron_templates.py`. As described above the neuron model is defined as a dictionary in which the following `keys` are required: * 'model' * 'threshold' * 'reset'

The parameters are defined **separately** here. your new entry to `neuron_templates` should look like this:

```
#As an example lets define a new voltage-based model
new_neuron_model = {
  'model': '''
      ''',
  'threshold': ''' ''',
  'reset':'''
      '''
  }

#define new parameters
new_neuron_model_params = {
    }
```

At the end of the file, we need to associate the newly defined model and its parameters with `keywords` used by the `EquiationBuilder`. Each neuron model as two corresponding dictionaries. The first one is the `equation_sets` dictionary which depending on the `base_unit` is either called `current_equation_sets` or `voltage_equation_sets`. The second dictionary is the `parameters` dictionary which depending on the `base_unit` is either called `current_parameters` or `voltage_parameters`. The `key` in this dictionary needs to match die `**kwargs` given in the class initilisation and the value needs to match the name of the dictionary defined in `neuron_templates.py`. That allows the `EquationBuilder` upon initialisation to dynamically assemble the respective equations into a coherent model. This functionality is especially useful when you e.g. just createdevelop a new threshold adaption mechanism, a different adaption current dynamics or a new plasticity rule. After the template was added the newly defined neuron model must be added to `neuron_models.py` to be generated dynamically. The entry should look similar to this

```
class my_neuron_model(NeuronEquationBuilder):
    """This class provides you with all equations to simulate a current-based
    awesome model...
    """

    def __init__(self, num_inputs=1):
        NeuronEquationBuilder.__init__(self, base_unit='current', adaptation='none',
                                       integration_mode='exponential', leak='leaky',
                                       position='spatial', awesome_new_feature='adp')
        self.add_input_currents(num_inputs)
```

Note: This is just an example.

8.2.2 Synapes templates

Navigate to the template sub-directory (`teili/models/builder/templates`) and open `synapse_templates.py`. * Define your new model equations and the corresponding parameters in synapse models. * Synapse models have the following keywords: **model**, **on_pre**, **on_post** and **parameters** * Make sure that both the new model equations and the corresponding parameters are added in the *Dictionary of keywords* at the bottom of the file. * Neuron templates are divided into two main modes: **current** and **voltage** based equations. Each mode supports equations and parameters. * Synapse templates are divided based on function. Equations are divided based on **modes**, **kernels**, **plasticity_models** and new **synaptic equations**. * Synaptic **modes** are further devided into subcategories: **current**, **DPI**, **conductance**, **DPIShunting** or **unit_less**. * Create your model using the Neuron or SynapseEquationBuilder.

```
#As an example lets define a new current based kernel and create a new model
new_synapse_kernel = {
  'model': '''

      ''',
  'on_pre': '''

      ''',
  'on_post':
      '''
      '''
  }

#define new parameters
new_synapse_kernel_params = {
    }

#include model in equations
kernels = {
      'new': new_synapse_kernel
      }

#include parameters
current_parameters = {
  'new_kernel': new_synapse_kernel_params,
  }
```

Once the templates are extendend you can add the model to `synapse_models.py` located in `teili/models/`.

```
# Define the new model
class my_model(SynapseEquationBuilder):
    def __init__(self):
        SynapseEquationBuilder.__init__(self, base_unit='current', kernel = 'new_
↪kernel')

 my_model = my_model()
 my_model.export_eq(os.path.join(path, "my_model"))
```

8.2.3 Create a new template using the unit_less dictionary

You might want to develop, test or define a plasticity mechanism or part of a neuron model which neither uses currents or voltages. Therefore, we provide a third dictionary called `unit_less`.

- `unit_less` models follow the same proceduce as `current` or `voltage` based dictionaries.
- **Parameters** to be defined to the `unit_less` dictionary while the model needs to be added to one of the model dictionaries.
- When creating the synapse model **base_unit** should be defined as **unit_less**.
- This can be usefull to define learning rules that involve gain modulation or activity modulation. (e.g. `STDGM`)

```
# Define the new model
class my_model(SynapseEquationBuilder):
    def __init__(self):
        SynapseEquationBuilder.__init__(self, base_unit='unit_less')
```

8.2.4 Combine equations and replace variables

A major strength of *teili* is its **modularity**. This starts already at the equation level. If you want to to e.g. test an existing synapse model with let's say Spike-Timing Dependent Plasticity (STDP), you don't need to specify a complete new model. Instead you can initialise the `EquationBuilder` differently, such that the STDP template is combined with a `DPI` synapse using a `Alpha` -shaped kernel. Internally, the different equation sets are combined depending on the provided keyword arguments and equations which have a default definition but are defined differently in a given plasticity mechanism are replaced. We provide a method call `var_replacer` which uses the '%' symbol to replace equations in the original set of equations. Compare e.g. the `synapse_models`: **Alpha** and **AlphaStdp** (located in `teili/models/synapapse_models.py`) and their respective templates (located in `teili/models/builder/templates/synapse_templates.py`) For more information see our documentation on the equation builder

8.3 How to contribute and publish your custom model

Once you tested your custom model locally, you can add a new template. To do so you need to work directly inside the library code. Head to the **repository_** and do the following steps

1. Fork the `dev` branch
2. Clone the forked repository to your local system
3. Create a new branch within your forked repository (e.g. *git checkout -b new_branch + git remote add upstream URL_of_project*)
4. Follow the steps from our contribution guide
5. Make your changes, add appropriate unite tests and push it to your repository.

You can create a pull request to add your remote change to *teili*

- Click *compare & pull request* button on github.
- Click *create pull request* to open a new pull request
- Wait for us to approve it and give you feedback :)

CHAPTER

NINE

DEVELOPING BUILDING BLOCKS

This section explains the generation of new and more complex `BuildingBlocks`. An example of such a `BuildingBlock` can be found in the Building Block documentation, as well as in the respective advanced tutorials such as the `threeway network` or the `OCTA network`.

Every building block inherits from the class `BuildingBlocks` which has attributes such as `sub_blocks`, `input_groups`, `output_groups` and `hidden_groups`.

Recommended practices for creating custom `BuildingBlocks` are as follow:

- Keep the class initialization as concise as possible.
- Create a generation function which implements the desired connectivity.
- Label correctly `sub_blocks`, `input_groups`, `output_groups` and `hidden_groups`.
- Remember to `_set_tags` as you expand the network and its functionality; it really helps to keep track of the properties of individual groups.

Important Notes:

- When running the network, add the newly generated `BuildingBlock` as well as all the `sub_blocks` the network depends on.
- There is a fundamental difference between the attributes `groups` and `_groups`. `_groups` is a dictionary containing the objects specific to that `BuildingBlock`. `groups` is a property of the `BuildingBlock` class which returns all `_groups` included in the `BuildingBlock` and its `sub_blocks`.
- When overwriting an existing population in one of the `sub_blocks._groups`, remember to re-initialize all the `Connections` and `monitors` regarding that population. Which will now be specific to the parent class.

Have fun developing your own `BuildingBlocks` and advancing neuroscience! :)

CHAPTER

TEN

TOOLS

This is a collection of useful tools and functions mainly for use with *Brian2*. Or other spiking network software or hardware.

10.1 converter

Functions in this module convert data from or to *Brian2* compatible formats. In particular, there are functions to convert and process data from Dynamic Vision Sensors (DVSs), which are event-based cameras. If you want to learn more about a **DVS** checkout the original publication

10.2 cpptools

This provides functions that are used by the TeiliNetwork class to allow running the network several times with different parameters without recompiling. This is in particular useful if you have small networks and you would like to do parameter optimization. Have a look at sequence_learning_standalone_tutorial as an example for how to use it.

```
# net is built once
net.build(standalone_params = standaloneParams)
# and can then be run several times with different parameters without recompilation
net.run(duration,standaloneParams = standaloneParams)
```

10.3 distance

These functions to compute different distance measures are mainly there to provide an easy to use cpp implementation for *Brian2* cpp code generation. They are used by the synaptic kernel functions, but can also be added into any *Brian2* string. Make sure to add the functions you use to the namespace of the respective groups as follows:

```
from teili.tools.distance import function
group.namespace['functionname_used_in_string'] = function
```

10.4 indexing

Functions that convert 1d indices to x, y coordinates and vice versa including cpp implementation. As *Brian2* uses 1d indexing for neurons, it is necessary to convert 1d to 2d indices every so often when e.g. generating synapses. Numpy provides a good API for that, which we use here, but we also add a cpp implementation so the functions can be used in standalone mode.

10.5 live

WIP Live plotting and changing of parameters during numpy based simulation could be done like this.

10.6 misc

Functions that didn't fit in any of the other categories so far.

10.7 plotter2d

Will be deprecated soon once implemented into the visualizer. Provides 2d plotting functionality (video plot and gif generation)

10.8 plotting

Will be deprecated soon once implemented into the visualizer?

10.9 prob_distributions

Probability density functions with cpp implementation.

This is e.g. a plot of a 1d Gaussian:

```
import matplotlib.pyplot as plt
dx = 0.1
normal1drange = np.arange(-10, 10, dx)
gaussian = [normal1d_density(x, 0, 1, True) for x in normal1drange]
print(np.sum(gaussian) * dx)

plt.figure()
plt.plot(normal1drange, gaussian)
plt.show()
```

10.10 random_sampling

This module provides functions to sample from a random distribution, e.g. for random initialization of weights. All functions in should be callable in a similar way as rand() or randn() that are provided by *Brian2*.

Here is an example how this works in standalone mode:

```
from teili.tools.random_sampling import Rand_gamma, Randn_trunc

n_samples = 10000
standaloneDir = os.path.expanduser('~/gamma_standalone')
set_device('cpp_standalone', directory=standaloneDir, build_on_run=True)

ng = NeuronGroup(n_samples, '''
testvar : 1
testvar2 : 1''', name = 'ng_test')

ng.namespace.update({'rand_gamma': Rand_gamma(4.60, -10750.0),
                     'randn_trunc': Randn_trunc(-1.5,1.5)
                    })

ng.testvar = 'rand_gamma()'
ng.testvar2 = '5*randn_trunc()'

run(10 * ms)

plt.figure()
plt.title('rand_gamma')
plt.hist(ng.testvar, 50, histtype='step')
plt.show()

plt.figure()
plt.title('randn_trunc')
plt.hist(ng.testvar2, 50, histtype='step')
plt.show()
```

10.11 random_walk

Functions that generate a random walk. E.g. as artificial input.

10.12 sorting

To understand the structure in spiking activity of a network or more specifically the structure in the spike rasterplots of a neuronal population we need to sort the neuronal indicies. But also if we want to understand the strucuture of a learned weight matrix we need to be able to sort this matrix. This set of tools allows the user to sort a given weight matrix according to some similarity measure, such as euclidean distance. The class returns a list of permutated indices which can be used to sort a spike rasterplot or the weight matrix itself, before it is being displayed. However, the sorting algorithm is completely agnostic to the similarity measure. It connects each node with maximum two edges and constructs a directed graph. This is similar to the travelling salesman problem.

Example: In order to use this class you need to initialize it either without a filename:

```
from teili.tools.sorting import SortMatrix
import numpy as np
matrix = np.random.randint((49, 49))
obj = SortMatrix(nrows=49, matrix=matrix)
print(obj.matrix)
print(obj.permutation)
print(ob.sorted_matrix)
```

or instead of using a matrix you can also specify a path to a stored matrix:

```
filename = '/path/to/your/matrix.npy'
obj = SortMatrix(nrows=49, filename=filename)
```

10.13 stimulus_generators

The idea is to generate inputs based on a function instead of having to use a fixed spikegenerator that is filled before the simulation. This avoids having to read large datafiles and makes generation of input easier.

Use it as follows (also teili groups and network can be used):

```
import matplotlib.pyplot as plt

from brian2 import SpikeMonitor, Network, prefs, ms

from teili.tools.stimulus_generators import StimulusSpikeGenerator
from teili import normal2d_density, Plotter2d

prefs.codegen.target = 'numpy'

nrows = 80
ncols = 80

# Create a moving Gaussian with increasing sigma
# the update that happens every dt is given in the trajectory_eq
# the center coordinates move 5 to the right and 2 upwards every dt
# the sigma is increased by 0.1 in both directions every dt
trajectory_eq = '''
                mu_x = (mu_x + 5)%nrows
                mu_y = (mu_y + 2)%nrows
                sigma_x += 0.1
                sigma_y += 0.1
                '''

stimgen = StimulusSpikeGenerator(
    nrows, ncols, dt=50 * ms, trajectory_eq=trajectory_eq, amplitude=200,
    spike_generator='poisson', pattern_func=normal2d_density,
    name="moving_gaussian_stimgen",
    mu_x=40.0, mu_y=40.0, sigma_x=1.0, sigma_y=1.0, rho=0.0, normalized=False)

poissonspmon = SpikeMonitor(stimgen, record=True)

net = Network()
net.add((stimgen, poissonspmon))
net.run(3000 * ms)
```

(continues on next page)

(continued from previous page)

```
plt.plot(poissonspmon.t, poissonspmon.i, ".")

plotter2d = Plotter2d(poissonspmon, (nrows, ncols))
imv = plotter2d.plot3d(plot_dt=10 * ms, filtersize=20 * ms)
imv.show()
```

10.14 synaptic_kernel

This module provides functions that can be used for synaptic connectivity kernels (generate weight matrices). E.g. Gaussian, Mexican hat, Gabor with different dimensionality, also using different distance metrics. In order to also use them with C++ code generation, all functions have a cpp implementation given by the @implementation decorator.

```
# SPDX-License-Identifier: MIT
# Copyright (c) 2018 University of Zurich
```

CHAPTER

ELEVEN

VISUALIZER

Model: The model represents the data, and does nothing else. The model does NOT depend on the controller or the view.

- EventsModel
- StateVariablesModel
- ...

View: The view displays the model data, and sends user actions (e.g. button clicks) to the controller. The view can be independent of both the model and the controller or actually be the controller, and therefore depend on the model.

- HistogramViewer
- RasterplotViewer
- LineplotViewer
- ...

Controller: The controller provides model data to the view, and interprets user actions such as button clicks. The controller depends on the view and the model. In some cases, the controller and the view are the same object.

- HistogramController (called Histogram)
- RasterplotController (called Rasterplot)
- LineplotController (called Lineplot)
- ...

PlotSettings()

- font sizes
- ...

Backends Currently, the visualizer supports matplotlib and pyqtgraph as backends. To switch between the two backends you simply have to change one input flag of the controller. This is shown below in the examples which are given for pyqtgraph and for matplotlib, however the figure are only shown for matplotlib.

11.1 How to use it - in short

1) get data

```
spikemonN1, spikemonN2, statemonN1, statemonN2 = run_your_own_brian_network()
```

2) define PLOTSETTING

```
from teili.tools.visualizer.DataViewers import PlotSettings
MyPlotSettings = PlotSettings(fontsize_title=20, fontsize_legend=14,
    Fontsize_axis_labels=14, marker_size = 30,colors = ['r', 'b', 'g', 'c',
    'k', 'm', 'y'])
```

3) call CONTROLLER of desired type of plot, e.g. Histogram, Rasterplot, Lineplot, ...

```
from teili.tools.visualizer.DataControllers import Histogram
DataModel_to_attr = [(spikemonN1, 'i'), (spikemonN2, 'i')]
subgroup_labels = ['N1', 'N2']
HC = Histogram(MyPlotSettings=MyPlotSettings, DataModel_to_attr=DataModel_to_attr,␣
↪subgroup_labels=subgroup_labels, backend='matplotlib')
```

If you want to change anything else on the main figure/window or one of the subplots you can directly access the mainfigure (matplotlib: figure, pyqtgraph: qt-window) as *my_controller.mainfig* or the corresponding subplot under *my_controller.subfig*

11.2 How to use it - the slightly longer version

Import the required modules and functions ...

```
%pylab inline
import numpy as np
import os

from brian2 import us, ms, second, prefs, defaultclock, start_scope,␣
↪SpikeGeneratorGroup, SpikeMonitor, StateMonitor
import matplotlib.pylab as plt
import pyqtgraph as pg
from PyQt5 import QtGui

from teili.core.groups import Neurons, Connections
from teili import TeiliNetwork
from teili.models.neuron_models import DPI
from teili.models.synapse_models import DPISyn
from teili.models.parameters.dpi_neuron_param import parameters as neuron_model_param
from teili.models.parameters.dpi_synapse_param import parameters as synapse_model_
↪param

QtApp = QtGui.QApplication([])
```

```
def run_brian_network():
    prefs.codegen.target = "numpy"
    defaultclock.dt = 10 * us

    start_scope()
```

(continues on next page)

(continued from previous page)

```
    N_input, N_N1, N_N2 = 1, 5, 3
    duration_sim = 100
    Net = TeiliNetwork()
    # setup spike generator
    spikegen_spike_times = np.sort(np.random.choice(size=500, a=np.
↪arange(float(defaultclock.dt), float(duration_sim*ms)*0.9,
                                                                    ␣
↪float(defaultclock.dt*5)), replace=False)) * second
    spikegen_neuron_ids  = np.zeros_like(spikegen_spike_times) / ms
    gInpGroup = SpikeGeneratorGroup(N_input, indices=spikegen_neuron_ids,
                                    times=spikegen_spike_times, name='gtestInp')
    # setup neurons
    testNeurons1 = Neurons(N_N1, equation_builder=DPI(num_inputs=2), name="testNeuron
↪")
    testNeurons1.set_params(neuron_model_param)
    testNeurons2 = Neurons(N_N2, equation_builder=DPI(num_inputs=2), name="testNeuron2
↪")
    testNeurons2.set_params(neuron_model_param)
    # setup connections
    InpSyn = Connections(gInpGroup, testNeurons1, equation_builder=DPISyn(), name=
↪"testSyn", verbose=False)
    InpSyn.connect(True)
    InpSyn.weight = '200 + rand() * 100'
    Syn = Connections(testNeurons1, testNeurons2, equation_builder=DPISyn(), name=
↪"testSyn2", verbose=False)
    Syn.connect(True)
    Syn.weight = '200 + rand() * 100'
    # spike monitors input and network
    spikemonInp = SpikeMonitor(gInpGroup, name='spikemonInp')
    spikemonN1 = SpikeMonitor(testNeurons1, name='spikemon')
    spikemonN2 = SpikeMonitor(testNeurons2, name='spikemonOut')
    # state monitor neurons
    # statemonN1 = StateMonitor(testNeurons1, variables=["Iin", "Imem"], record=[0,␣
↪3], name='statemonNeu')
    statemonN1 = StateMonitor(testNeurons1, variables=["Iin", "Iahp"], record=True,␣
↪name='statemonNeu')
    # statemonN2 = StateMonitor(testNeurons2, variables=['Iahp'], record=0, name=
↪'statemonNeuOut')
    statemonN2 = StateMonitor(testNeurons2, variables=['Imem'], record=True, name=
↪'statemonNeuOut')

    Net.add(gInpGroup, testNeurons1, testNeurons2, InpSyn, Syn, spikemonN1,␣
↪spikemonN2, statemonN1, statemonN2)
    # run simulation
    Net.run(duration_sim * ms)
    print ('Simulation run for {} ms'.format(duration_sim))
    return spikemonN1, spikemonN2, statemonN1, statemonN2
```

11.3 Get the data to plot

Option A: run brian network to get SpikeMonitors and StateMonitors

```
spikemonN1, spikemonN2, statemonN1, statemonN2 = run_brian_network()
```

Option B: create DataModel instance from arrays, lists or brian-SpikeMonitors/StateMonitors

Available DataModels:

EventsModel: stores neuron_ids and spike_times

Example shows how to create it from array/list . . .

```
from teili.tools.visualizer.DataModels import EventsModel
neuron_ids  = [1, 1, 1, 2, 3, 1, 4, 5]
spike_times = [11, 14, 14, 16, 17, 25, 36, 40]
EM = EventsModel(neuron_ids=neuron_ids, spike_times=spike_times)
```

. . . or create from brian spike monitor

```
EM = EventsModel.from_brian_spike_monitor(spikemonN1)
```

Then the created EventsModel EM has the following attributes:

```
neuron_ids :
 [3 4 1 0 2 3 4 1 0 2 3 4 1 0 2 3 4 1 0 2 3 4 1 0 2 3 4 1 0 2 3 4
 1 0 2 3 4 1 0 2 3 4 1 0 2 3 4 1 0 2 3 4 1 0 3 2 4 1 3 0 2 4 1 3
 0 2 4 1 3 0 2 4 1 3 0 2 4 3 1 0 2 4 3 1 0 2 4 3 1 0 2 4 3 1 0 2
 4 3 1 0 2 4 3 1 0 2 4 3 1 0 2 3 4 1 0 2 3 4 1 0 3 2 4 1 0 3 4 2 1 3 0 4 2
 1 3 0 4 2 1 3 0 4 2 1 3 0 4 2 1 3 4 0 2 1 3 4 0 2 1 3 4 0 2 1 3 4 0 2 3 1
 4 0 2 3 1 4 0 2 3 1 4 0 3 2 1 4 0 3 2 1]
spike_times :
 [0.00387 0.004   0.00405 0.00411 0.00413 0.00628 0.00651 0.00659 0.00669
 0.00673 0.0085  0.0088  0.00891 0.00903 0.00908 0.0107  0.0111  0.01124
 0.01139 0.01145 0.01278 0.01326 0.01344 0.01363 0.0137  0.01491 0.01552
 0.01574 0.01595 0.01603 0.01699 0.01764 0.01788 0.01812 0.01821 0.01907
 0.01984 0.02012 0.02039 0.02049 0.02108 0.02183 0.02216 0.02247 0.02259
 0.02304 0.02391 0.02426 0.0246  0.02473 0.02506 0.02604 0.02644 0.02684
 0.027   0.02719 0.02822 0.02867 0.02914 0.02932 0.02935 0.03047 0.03094
 0.03139 0.03144 0.03155 0.03261 0.03313 0.03351 0.03359 0.03375 0.0347
 0.03525 0.03554 0.03576 0.03594 0.03682 0.03731 0.03748 0.03781 0.038
 0.03885 0.03937 0.03945 0.03987 0.04005 0.04084 0.04141 0.04142 0.042
 0.04222 0.04304 0.04359 0.04372 0.04438 0.04459 0.04526 0.04569 0.04591
 0.04663 0.04688 0.04752 0.04786 0.04816 0.04878 0.04902 0.04953 0.0498
 0.05021 0.0509  0.05117 0.05161 0.0518  0.05233 0.05309 0.05338 0.05373
 0.05384 0.05449 0.05522 0.0555  0.05575 0.05579 0.05655 0.05743 0.05774
 0.05788 0.05792 0.0587  0.05949 0.05978 0.05978 0.05989 0.06072 0.06163
 0.0618  0.06194 0.06198 0.06289 0.06389 0.06391 0.06419 0.06423 0.06514
 0.06605 0.06618 0.06643 0.06656 0.06748 0.06828 0.06855 0.06871 0.0689
 0.06966 0.07033 0.07075 0.07086 0.07114 0.07195 0.0725  0.07303 0.07306
 0.07343 0.07423 0.07471 0.0754  0.07545 0.0759  0.07656 0.07688 0.07763
 0.07774 0.07815 0.07875 0.07899 0.07997 0.08017 0.08062 0.0811  0.08119
 0.08212 0.08238 0.08286 0.08328 0.0833  0.08428 0.08459 0.08509 0.08537
 0.0855  0.08647 0.08687 0.08744 0.08757 0.08783 0.08879 0.08929 0.08982
 0.08987 0.0902  0.09128 0.0921  0.09261 0.09306 0.09351]
```

StateVariablesModel: stores any number of variables with their name and the list of timepoints when the variable was sampled

Example shows how to create it from array/list or from brian spike monitor

```
from teili.tools.visualizer.DataModels import StateVariablesModel

# create from array/list
state_variable_names = ['var_name']
num_neurons, num_timesteps = 6, 50
state_variables       = [np.random.random((num_neurons, num_timesteps))]
state_variables_times = [np.linspace(0, 100, num_timesteps)]
SVM = StateVariablesModel(state_variable_names, state_variables, state_variables_
↪times)

# create from brian state monitorS
skip_not_rec_neuron_ids=False
SVM = StateVariablesModel.from_brian_state_monitors([statemonN1, statemonN2], skip_
↪not_rec_neuron_ids)
skip_not_rec_neuron_ids=True
SVM = StateVariablesModel.from_brian_state_monitors([statemonN1, statemonN2], skip_
↪not_rec_neuron_ids)
```

Then the created StateVariablesModel SVM has the following attributes:

```
Iin :
 [[0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
  0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
  0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
  0.00000000e+00]
 ...
 [6.82521123e-09 7.02939025e-09 6.74769896e-09 7.76629202e-09
  7.21872104e-09]
 [6.81237889e-09 7.01617406e-09 6.73501234e-09 7.75169045e-09
  7.20514890e-09]
 [6.79957068e-09 7.00298271e-09 6.72234958e-09 7.73711634e-09
  7.19160228e-09]]
t_Iin :
 [0.000e+00 1.000e-05 2.000e-05 ... 9.997e-02 9.998e-02 9.999e-02]
Iahp :
 [[5.00000000e-13 5.00000000e-13 5.00000000e-13 5.00000000e-13
  5.00000000e-13]
 [5.00000000e-13 5.00000000e-13 5.00000000e-13 5.00000000e-13
  5.00000000e-13]
 [5.00000000e-13 5.00000000e-13 5.00000000e-13 5.00000000e-13
  5.00000000e-13]
 ...
 [2.35349697e-11 2.45322975e-11 2.37997191e-11 2.54116963e-11
  2.38412778e-11]
 [2.35283328e-11 2.45253794e-11 2.37930076e-11 2.54045302e-11
  2.38345545e-11]
 [2.35216978e-11 2.45184633e-11 2.37862979e-11 2.53973662e-11
  2.38278332e-11]]
t_Iahp :
 [0.000e+00 1.000e-05 2.000e-05 ... 9.997e-02 9.998e-02 9.999e-02]
Imem :
 [[0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [4.74578721e-33 4.74578721e-33 4.74578721e-33]
 [9.49157441e-33 9.49157441e-33 9.49157441e-33]
```

(continues on next page)

(continued from previous page)

```
 ...
 [1.14559533e-10 2.80317027e-10 3.29995059e-10]
 [1.15005619e-10 2.80084652e-10 3.29576244e-10]
 [1.15447969e-10 2.79851599e-10 3.29157605e-10]]
t_Imem :
 [0.000e+00 1.000e-05 2.000e-05 ... 9.997e-02 9.998e-02 9.999e-02]
```

11.4 Plot the collected data

11.4.1 Define PlotSettings

- The PlotSettings are defined only once for all the plots that will be created. This should make it easier to get consistent color-codings, fontsizes and markersize across different plots.
- The colors can be defined as RGBA to additionally define the transparency

```
from teili.tools.visualizer.DataViewers import PlotSettings
MyPlotSettings = PlotSettings(fontsize_title=20, fontsize_legend=14, fontsize_axis_
↪labels=14,
                              marker_size = 30,             # default 5
                              colors = ['r', 'b'],          # default ['r', 'b', 'g',
↪ 'c', 'k', 'm', 'y']
)
```

11.4.2 Call the DataController of the desired type of plot

So far in teili:

- Histogram
- Rasterplot
- Lineplot

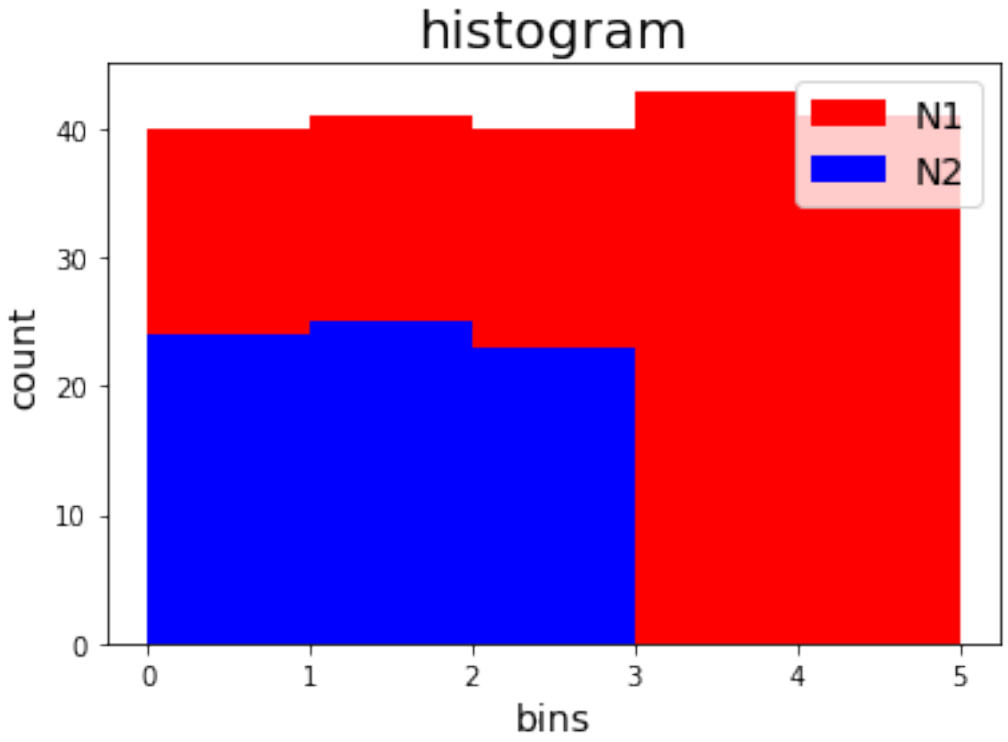
11.4.3 Histogram

Histogram - Inputs

```
* DataModel_to_attr             --> e.g. [(spikemonN1, 'i'), (spikemonN2, 'i')] OR
                                        [(EventsModel, 'i'), (EventsModel, 'i')]
* MyPlotSettings=PlotSettings()
* subgroup_labels=None          --> e.g. ['Neurongroup N1', 'Neurongroup N2']
* bins=None                     --> e.g. range(0,9)
* orientation='vertical'        --> 'horizontal' OR 'vertical'
* title='histogram
* xlabel='bins'
* ylabel='count',
* backend='matplotlib'
* show_immediately=False
```

Simple example to plot a histogram of two NeuronGroups to plot data from BrianSpikeMontiors/StateMonitors (or Datamodels) with matplotlib backend:

```
from teili.tools.visualizer.DataControllers import Histogram
# brian spike monitor
DataModel_to_attr =  [(spikemonN1, 'i'), (spikemonN2, 'i')]

# or plot data from DataModels
# EM1 = EventsModel.from_brian_spike_monitor(spikemonN1)
# EM2 = EventsModel.from_brian_spike_monitor(spikemonN2)
# DataModel_to_attr = {EM1: 'neuron_ids', EM2:'neuron_ids'}
subgroup_labels = ['N1', 'N2']

HC = Histogram(DataModel_to_attr=DataModel_to_attr,
                MyPlotSettings=MyPlotSettings,
                subgroup_labels=subgroup_labels,
                backend='matplotlib')
```



or PYQTGRAPH backend:

```
HC = Histogram(DataModel_to_attr=DataModel_to_attr,
                MyPlotSettings=MyPlotSettings,
                subgroup_labels=subgroup_labels,
                backend='pyqtgraph',
                QtApp=QtApp, show_immediately=True)
```

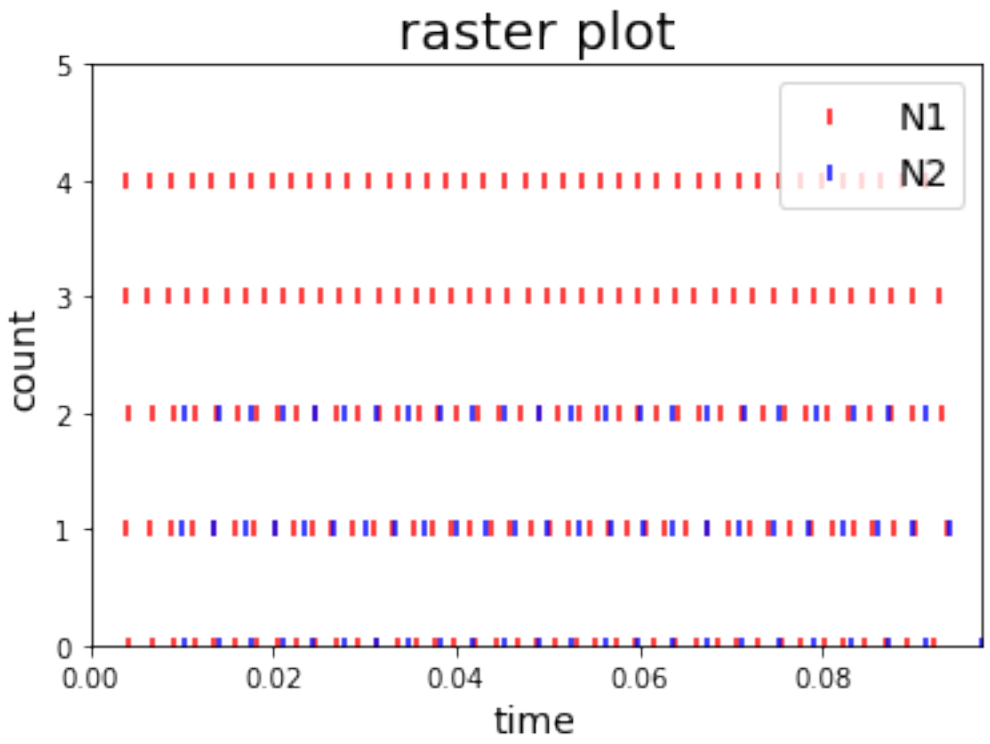
11.4.4 Rasterplot

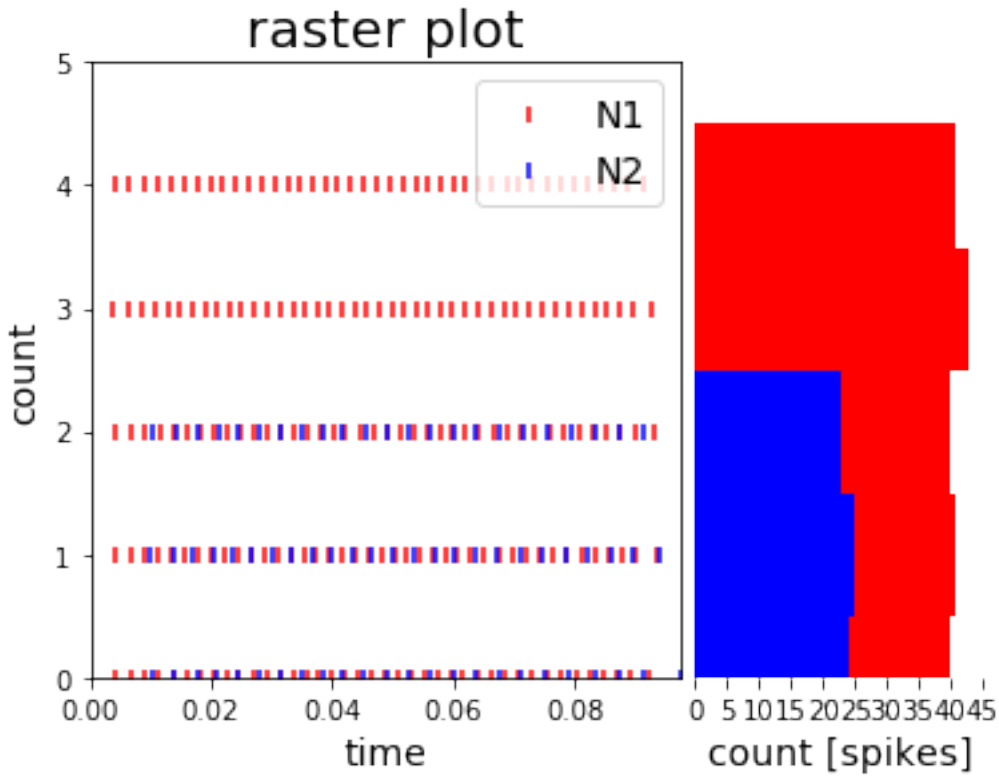
Rasterplot - Inputs

```
* MyEventsModels                --> list of EventsModel or BrianSpikeMonitors
* MyPlotSettings=PlotSettings()
* subgroup_labels=None          --> ['N1', 'N2']
* time_range=None               --> (0, 0.9)
* neuron_id_range=None,         --> (0, 4)
* title='raster plot'
* xlabel='time
* ylabel='count',
* backend='matplotlib'
* add_histogram=False           --> show histogram of spikes per neuron id next to␣
↪rasterplot
* show_immediately=False
```

Simple example to plot a raster plot of two NeuronGroups to plot data from BrianSpikeMontiors/StateMonitors (or Datamodels) with matplotlib backend:

```
from teili.tools.visualizer.DataControllers import Rasterplot
# plot data from BrianSpikeMontiors
MyEventsModels = [spikemonN1, spikemonN2]

# or plot data from EventsModel
# EM1 = EventsModel.from_brian_spike_monitor(spikemonN1)
# EM2 = EventsModel.from_brian_spike_monitor(spikemonN2)
# MyEventsModels = [EM1, EM2]

subgroup_labels = ['N1', 'N2']

# MATPLOTLIB backend - WITHOUT HISTOGRAM
RC = Rasterplot(MyEventsModels=MyEventsModels, MyPlotSettings=MyPlotSettings,␣
↪subgroup_labels=subgroup_labels, backend='matplotlib')
# MATPLOTLIB backend - WITH HISTOGRAM
RC = Rasterplot(MyEventsModels=MyEventsModels, MyPlotSettings=MyPlotSettings,␣
↪subgroup_labels=subgroup_labels, add_histogram=True)
```





or PYQTGRAPH backend:

```
# PYQTGRAPH backend - WITHOUT HISTOGRAM
```

(continues on next page)

(continued from previous page)

```
RC = Rasterplot(MyEventsModels=MyEventsModels, MyPlotSettings=MyPlotSettings,␣
↪subgroup_labels=subgroup_labels, backend='pyqtgraph', QtApp=QtApp)
# PYQTGRAPH backend - WITH HISTOGRAM
RC = Rasterplot(MyEventsModels=MyEventsModels, MyPlotSettings=MyPlotSettings,␣
↪subgroup_labels=subgroup_labels,
                    add_histogram=True, backend='pyqtgraph', QtApp=QtApp, show_
↪immediately=True)
```

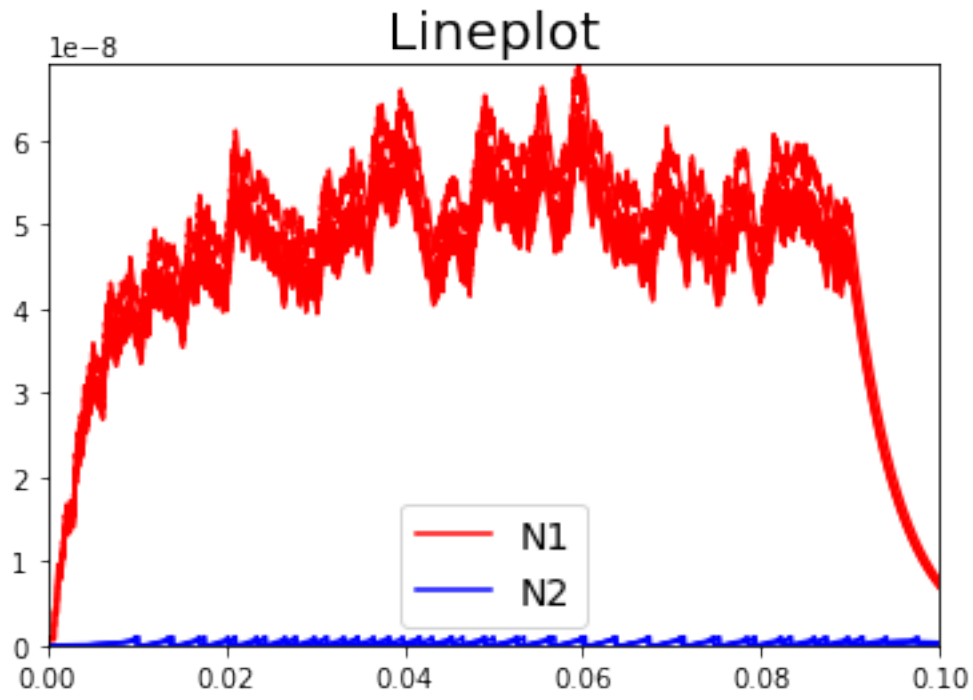
11.4.5 LinePlot

Lineplot - Inputs

```
* DataModel_to_x_and_y_attr --> e.g. [(statemonN1, ('Imem', 't_Imem')),
                                      (statemonN2, ('Iahp', 't_Iahp'))]
                                  OR
                                [(StateVariablesModel_N1, ('Imem', 't_Imem')),
                                 (StateVariablesModel_N2, ('Iahp', 't_
↪Iahp'))]
* MyPlotSettings=PlotSettings()
* subgroup_labels=None      --> ['N1', 'N2']
* x_range=None,             --> (0, 0.9)
* y_range=None,             --> (0, 4)
* title='Lineplot'
* xlabel=None
* ylabel=None
* backend='matplotlib'
* show_immediately=False
```

Simple example to plot a line plot of two NeuronGroups to plot data from BrianSpikeMontiors/StateMonitors (or Datamodels) with matplotlib backend:

```
from teili.tools.visualizer.DataControllers import Lineplot
# plot data from BrianSpikeMontiors
DataModel_to_x_and_y_attr = [(statemonN1, ('t', 'Iin')), (statemonN2, ('t', 'Imem'))]
# or plot data from StateVariablesModel
SVM_N1 = StateVariablesModel.from_brian_state_monitors([statemonN1])
SVM_N2 = StateVariablesModel.from_brian_state_monitors([statemonN2])
DataModel_to_x_and_y_attr = [(SVM_N1, ('t_Iin', 'Iin')), (SVM_N2, ('t_Imem', 'Imem'))]

subgroup_labels = ['N1', 'N2']
LC = Lineplot(DataModel_to_x_and_y_attr=DataModel_to_x_and_y_attr,
                  MyPlotSettings=MyPlotSettings,
                  subgroup_labels=subgroup_labels,
                  backend='matplotlib')
```



or PYQTGRAPH backend:

```
LC = Lineplot(DataModel_to_x_and_y_attr=DataModel_to_x_and_y_attr,
                   MyPlotSettings=MyPlotSettings,
                   subgroup_labels=subgroup_labels,
                   backend='pyqtgraph', QtApp=QtApp, show_immediately=True)
```

11.5 Additional functionalities

11.5.1 Combine different plots

You can combine different plot in any kind of way. Just specifiy the location and size of the subplots you would want to have and then pass the respective subplots on to the different controller. When using more than one controller make sure to set show_immediately only to true in the controller that is called last.

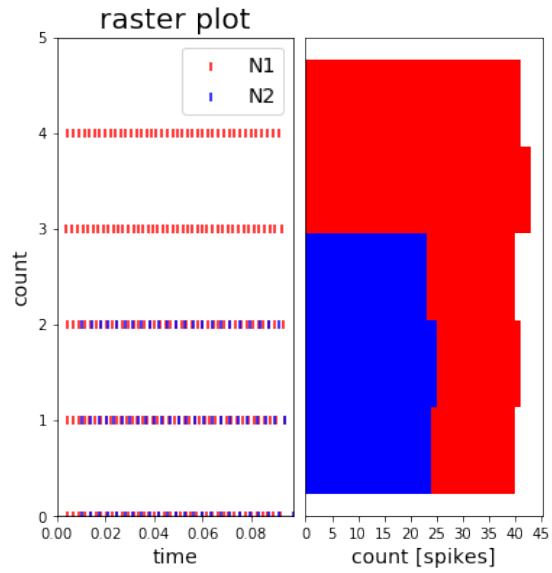
. . . with matplotlib

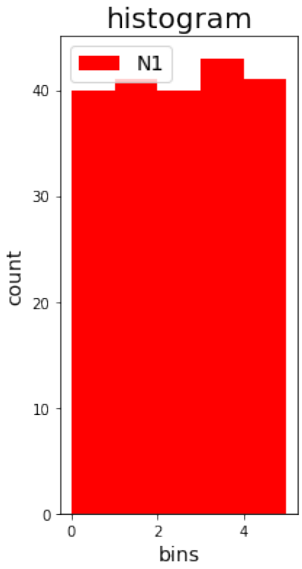
```
# define plot structure BEFOREHAND
mainfig = plt.figure()
subfig1 = mainfig.add_subplot(321)
subfig2 = mainfig.add_subplot(322)
subfig3 = mainfig.add_subplot(324)
subfig4 = mainfig.add_subplot(325)

plt.subplots_adjust(left=0.125, right=0.9, bottom=0.1, top=4., wspace=0.05, hspace=0.
↪2)

MyEventsModels = [spikemonN1, spikemonN2]
subgroup_labels = ['N1', 'N2']
RC = Rasterplot(MyEventsModels=MyEventsModels, MyPlotSettings=MyPlotSettings,␣
↪subgroup_labels=subgroup_labels,
```

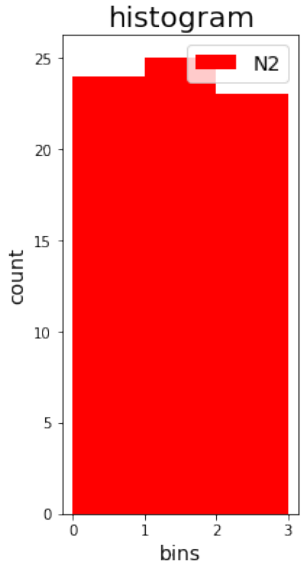
(continues on next page)

(continued from previous page)

```
                mainfig=mainfig, subfig_rasterplot=subfig1, subfig_histogram =␣
↪subfig2,
                add_histogram=True, show_immediately=False)

DataModel_to_attr = [(spikemonN1, 'i')]
subgroup_labels = ['N1']
HC = Histogram(DataModel_to_attr=DataModel_to_attr, MyPlotSettings=MyPlotSettings,
                subgroup_labels=subgroup_labels, mainfig=mainfig, subfig=subfig3,␣
↪show_immediately=False)

DataModel_to_attr = [(spikemonN2, 'i')]
subgroup_labels = ['N2']
HC = Histogram(DataModel_to_attr=DataModel_to_attr, MyPlotSettings=MyPlotSettings,
            subgroup_labels=subgroup_labels, mainfig=mainfig, subfig=subfig4, show_
↪immediately=True)
```


raster plot
N1
N2
count
time
count [spikes]


histogram
N1
count
bins


histogram
N2
count
bins

. . . with pyqtgraph

```
# define plot structure BEFOREHAND
mainfig = pg.GraphicsWindow()
subfig1 = mainfig.addPlot(row=0, col=0)
subfig2 = mainfig.addPlot(row=0, col=1)
subfig2.setYLink(subfig1)
subfig3 = mainfig.addPlot(row=1, col=1)
subfig4 = mainfig.addPlot(row=2, col=0)

plt.subplots_adjust(left=0.125, right=0.9, bottom=0.1, top=4., wspace=0.05, hspace=0.
↪2)

MyEventsModels = [spikemonN1, spikemonN2]
subgroup_labels = ['N1', 'N2']
RC = Rasterplot(MyEventsModels=MyEventsModels, MyPlotSettings=MyPlotSettings,␣
↪subgroup_labels=subgroup_labels,
                        mainfig=mainfig, subfig_rasterplot=subfig1, subfig_histogram␣
↪= subfig2, QtApp=QtApp,
                        backend='pyqtgraph', add_histogram=True, show_
↪immediately=False)

DataModel_to_attr = [(spikemonN1, 'i')]
subgroup_labels = ['N1']
HC = Histogram(DataModel_to_attr=DataModel_to_attr, MyPlotSettings=MyPlotSettings,
                        subgroup_labels=subgroup_labels,
                        backend='pyqtgraph', mainfig=mainfig, subfig=subfig3, ␣
↪QtApp=QtApp,
                        show_immediately=False)

DataModel_to_attr = [(spikemonN1, 'i')]
subgroup_labels = ['N2']
HC = Histogram(DataModel_to_attr=DataModel_to_attr, MyPlotSettings=MyPlotSettings,
                        subgroup_labels=subgroup_labels,
                        backend='pyqtgraph', mainfig=mainfig, subfig=subfig4,␣
↪QtApp=QtApp,
                        show_immediately=True)
```

11.5.2 Add second plot with a detailed view of a given plot

Create original plot of which you would like to have a detailed version as well TWICE

```
MyEventsModels = [spikemonN1, spikemonN2]
subgroup_labels = ['N1', 'N2']

mainfig = pg.GraphicsWindow()
subfig1 = mainfig.addPlot(row=0, col=0)
mainfig.nextRow()
subfig2 = mainfig.addPlot(row=1, col=0)


RC_org = Rasterplot(MyEventsModels=MyEventsModels, MyPlotSettings=MyPlotSettings,␣
↪subgroup_labels=subgroup_labels,
                            mainfig=mainfig, subfig_rasterplot=subfig1,
                        QtApp=QtApp, backend='pyqtgraph', show_immediately=False)
RC_detail = Rasterplot(MyEventsModels=MyEventsModels, MyPlotSettings=MyPlotSettings,␣
↪subgroup_labels=subgroup_labels,
```

(continues on next page)

(continued from previous page)

```
                                mainfig=mainfig, subfig_rasterplot=subfig2,
                            QtApp=QtApp, backend='pyqtgraph', show_immediately=False)

RC_org.connect_detailed_subplot(filled_subplot_original_view=RC_org.viewer.subfig_
↪rasterplot,
                                filled_subplot_detailed_view=RC_detail.viewer.subfig_
↪rasterplot,
                                ~
```

CHAPTER

TWELVE

FREQUENTLY ASKED QUESTIONS

This page contains a collection of frequently asked questions. We are trying constantly to update this list. If you would like to add questions feel free to open first open an issue using our issue tracker. After we answered your question we will consider adding it our FAQ depending on how general the problem is the question addresses.

12.1 Installation

12.1.1 I ran the first basic tutorial but I did not get any output. Is my installation broken?

TBA

12.2 Models

12.2.1 Where can I find more information on the Winner-Takes-All (WTA) network?

TBA

12.2.2 Where can I find more information on the Spike-Time Dependent Plasticity (STDP) synapse model?

TBA

12.2.3 What is the Differential Pair Integrator (DPI) neuron? Where can I find more information on this model?

TBA

12.3 Classes & Properties

12.3.1 In which situation would the `num_input` property of class Neuron be useful?

TBA

12.3.2 Why do I need `connect(True)` after creating a Connection class?

TBA

12.3.3 Why do I need to update the namespace using `namespace.update`?

TBA

Contributors

- **Moritz Milde** - Initial work, equation builder, building blocks, neuron and synapse models, testbench -
- **Alpha Renner** - Initial work, core, visualizer, building blocks, tools -
- **Karla Burelo** - Synaptic kernels -
- **Sergio Solinas** - Synaptic kernels, building blocks, neuron models -
- **Nicoletta Risi** - Mismatch, DYNAPSE interface -
- **Matteo Cartiglia** - Building blocks, documentation -
- **Marco Rasetto** - Equation builder -
- **Germain Haessig** - Testing, documentation -
- **Dmitrii Zendrikov** - Threeway building block -
- **Adrian M. Whatley** - Code management, review, integration -
- **Vanessa Leite** - Integration, review -
- **Dmitrii Zendrikov** - Threeway building block -

CHAPTER

THIRTEEN

TEILI

13.1 teili package

13.1.1 Subpackages

teili.building_blocks package

Submodules

teili.building_blocks.building_block module

This module provides the parent class for (neuronal) building blocks.

Todo:

- **Hierarchy of building_blocks.** There is a problem when 2 BBs of the same kind are added to another BB, as the names collide.
- **Monitor naming.** Important all Monitors of Building blocks have to be named and named uniquely! Otherwise they will not be found, when a Network is rerun in standalone mode after rebuild without recompile

---

class teili.building_blocks.building_block.**BuildingBlock**(**args*, ***kw*)

Bases: brian2.core.names.Nameable

This class is the parent class to all building blocks, e.g. WTA, SOM.

name
Name of the building_block population

Type str, required

neuron_eq_builder
neuron class as imported from models/neuron_models

Type class, optional

synapse_eq_builder
synapse class as imported from models/synapse_models

Type class, optional

params
Dictionary containing all relevant parameters for each building block

Type dictionary, optional

verbose
Flag to gain additional information

Type bool, optional

groups
Keys to all neuron and synapse groups

Type dictionary

monitors
Keys to all spike and state monitors

Type dictionary

monitor
Flag to auto-generate spike and state monitors

Type bool, optional

standalone_params
Dictionary for all parameters to create a standalone network

Type dictionary

sub_blocks
Dictionary for all parent building blocks

Type dictionary

input_groups
Dictionary containing all possible groups which are potential inputs

Type dictionary

output_groups
Dictionary containing all possible groups which are potential outputs

Type dictionary

hidden_groups
Dictionary containing all remaining groups which are neither inputs nor outputs

Type dictionary

__iter__()
this allows us to iterate over the BrianObjects and directly add the Block to a Network

Returns Returns a dictionary wich contains all brian objects

Return type TYPE

_set_tags(*tags*, *target_group*)
This method allows the user to set a list of tags to a specific target group. Normally the tags are already assigned by each building block. So this method only adds convinience and a way to replace the default tags if this is needed by any user. Typically this should not be the user's concern, that's why it is private method.

Parameters

- **tags** (*dict*) – A dictionary of tags {'mismatch' : False, 'noise' : False, 'level': 0 , 'sign': 'None', 'target sign': 'None', 'num_inputs' : 0, 'bb_type' : 'None', 'group_type' : 'None', 'connection_type' : 'None', }

- **target_group** (*str*) – Name of group to set tags

get_groups(*tags*)
Get all groups which have a certain set of tags

Parameters **tags** (*dict*) – A dictionary of tags

Returns

List of all group objects which share the same tags as specified.

Return type target_dict (dict)

get_run_args()
this collects the arguments to cpp main() for standalone run

Returns Arguments which can be changed during/between runs

Return type TYPE

get_tags(*target_group*)
Get the currently set tags for a given group.

Parameters **target_group** (*str*) – Name of group to get tags from or direct *TeiliGroup*

Returns

Dictionary containing all assigned _tags of provided group

Return type (dict)

property groups
This property will collect all available groups from the respective building block. The property follows a recursive strategy to collect all available groups. The intention is to easily update all available groups for stacked building blocks. NOTE Avoid any kind of loops between Building Blocks. Loops are forbidden as they lead to infinite collection of groups.

Returns Dictionary containing all groups of all sub_blocks

Return type tmp_groups (dict)

print_tags(*target_group*)
Get the currently set tags for a given group.

Parameters **target_group** (*str*) – Name of group to get tags from

teili.building_blocks.chain module

This is a simple syn-fire chain of neurons

teili.building_blocks.chain.**chain_params**
Dictionary of default parameters for syn-fire chain

Type dict

Todo:

- **Update docstrings**

Not all Description of attributes are set. Please provide meaningful docstrings

---

Example

To use the syn-fire chain building block in your simulation you need to create an object of the class by:

```
>>> from teili.building_blocks.chain import Chain
>>> my_bb = Chain(name='my_chain')
```

If you want to change the underlying neuron and synapse model you need to provide a different equation_builder class:

```
>>> from teili.models.neuron_models import DPI
>>> from teili.models.synapse_models import DPISyn
>>> my_bb = Chain(name='my_chain',
                  neuron_eq_builder=DPI,
                  synapse_eq_builder=DPISyn)
```

If you want to change the default parameters of your building block you need to define a dictionary, which you pass to the building_block

```
>>> chain_params = {'num_chains': 4,
                    'num_neurons_per_chain': 15,
                    'synChaCha1e_weight': 4,
                    'synInpCha1e_weight': 1,
                    'gChaGroup_refP': 1 * ms}
>>> my_bb = Chain(name='my_chain', block_params=chain_params)
```

class teili.building_blocks.chain.**Chain**(**args*, ***kw*)

Bases: *teili.building_blocks.building_block.BuildingBlock*

This is a simple syn-fire chain of neurons.

gChaGroup_refP
Parameter specifying the refractory period.

Type str, optional

group
List of keys of neuron population.

Type dict

inputGroup
SpikeGenerator obj. to stimulate syn-fire chain.

Type SpikeGenerator

num_chains
Number of chains to generate.

Type int, optional

num_neurons_per_chain
Number of neurons within one chain.

Type int, optional

spikemon_cha
Description.

Type brian2 SpikeMonitor obj.

spikemon_cha_inp
Description.

Type brian2 SpikeMonitor obj.

standalone_params
Keys for all standalone parameters necessary for cpp code generation.

Type dict

synapse
Description.

Type TYPE

synChaCha1e_weight
Parameter specifying the recurrent weight.

Type int, optional

synInpCha1e_weight
Parameter specifying the input weight.

Type int, optional

plot(*savedir=None*)
Simple function to plot recorded state and spikemonitors.

Parameters **savedir** (*str, optional*) – Path to directory to save plot.

Returns Returns figure.

Return type matplotlib.pyplot object

teili.building_blocks.chain.**gen_chain**(*groupname='Cha'*, *neuron_eq_builder=<teili.models.neuron_models.ExpAdaptIF object>*, *synapse_eq_builder=<teili.models.synapse_models.ReversalSynV object>*, *num_chains=4*, *num_neurons_per_chain=15*, *num_inputs=1*, *synChaCha1e_weight=4*, *synInpCha1e_weight=1*, *gChaGroup_refP=1. * msecond*, *debug=False*)
Creates chains of neurons

Parameters

- **groupname** (*str, optional*) – Base name for building block.
- **neuron_eq_builder** (*TYPE, optional*) – Neuron equation builder object.
- **synapse_eq_builder** (*TYPE, optional*) – Synapse equation builder object.
- **num_chains** (*int, optional*) – Number of chains to generate.
- **num_neurons_per_chain** (*int, optional*) – Number of neurons within one chain.
- **num_inputs** (*int, optional*) – Number of inputs from different source populations.
- **synChaCha1e_weight** (*int, optional*) – Parameter specifying the recurrent weight.
- **synInpCha1e_weight** (*int, optional*) – Parameter specifying the input weight.
- **gChaGroup_refP** (*TYPE, optional*) – Parameter specifying the refractory period.
- **debug** (*bool, optional*) – Debug flag.

Returns Keys to all neuron and synapse groups. Monitors (dictionary): Keys to all spike- and statemonitors. standalone_params (dictionary): Dictionary which holds all parameters to create a standalone network.

Return type Groups (dictionary)

teili.building_blocks.reservoir module

This module provides a reservoir network, as described in Nicola and Clopath 2017.

`teili.building_blocks.reservoir.`**`reservoir_params`**
Dictionary of default parameters for reservoir.

Type dict

Example

To use the Reservoir building block in your simulation you need to create an object of the class by:

```
>>> from teili.building_blocks.reservoir import Reservoir
>>> my_bb = Reservoir(name='my_reservoir')
```

If you want to change the underlying neuron and synapse model you need to provide a different equation_builder class:

```
>>> from teili.models.neuron_models import DPI as neuron_model
>>> from teili.models.synapse_models import DPISyn as synapse_model
>>> my_bb = Reservoir(name='my_reservoir',
                      neuron_eq_builder=DPI,
                      synapse_eq_builder=DPISyn)
```

If you want to change the default parameters of your building block you need to define a dictionary, which you pass to the building_block

```
>>> reservoir_params = {'weInpR': 1.5,
                        'weRInh': 1,
                        'wiInhR': -1,
                        'weRR': 0.5,
                        'sigm': 3,
                        'rpR': 0 * ms,
                        'rpInh': 0 * ms
                        }
>>> my_bb = Reservoir(name='my_reservoir', block_params=reservoir_params)
```

class teili.building_blocks.reservoir.**Reservoir**(**args*, ***kw*)
Bases: *teili.building_blocks.building_block.BuildingBlock*

A recurrent Neural Net implementing a Reservoir.

group
List of keys of neuron population.

Type dict

input_group
SpikeGenerator object to stimulate Reservoir.

Type SpikeGenerator

num_neurons
Size of Reservoir neuron population.

Type int, optional

fraction_inh_neurons
Set to None to skip Dale's principle.

Type float, optional

spikemonR
A spikemonitor which monitors the activity of the reservoir population.

Type brian2.SpikeMonitor object

standalone_params
Keys for all standalone parameters necessary for cpp code generation.

Type dict

teili.building_blocks.reservoir.**gen_reservoir**(*groupname, neuron_eq_builder=<class 'teili.models.neuron_models.Izhikevich'>, synapse_eq_builder=<class 'teili.models.synapse_models.DoubleExponential'>, neuron_model_params={'Cm': 250. * pfarad, 'Iconst': 0. * amp, 'Ileak': 0. * amp, 'Inoise': 0. * amp, 'VR': -60. * mvolt, 'VT': -20. * mvolt, 'a': 10. * hertz, 'b': 0. * siemens, 'c': -65. * mvolt, 'd': 200. * pamp, 'k': 2.5e-09 * metre ** -4 * kilogram ** -2 * second ** 6 * amp ** 3, 'refP': 1. * msecond}, weInpR=1.5, weRInh=1, wiInhR=-1, weRR=0.5, sigm=3, rpR=0. * second, rpInh=0. * second, num_neurons=64, Rconn_prob=None, adjecency_mtr=None, num_input_neurons=0, num_output_neurons=1, output_weights_init=[0], taud=0, taur=0, num_inputs=1, fraction_inh_neurons=0.2, spatial_kernel='kernel_mexican_1d', monitor=True, additional_statevars=[], debug=False*)

Generates a reservoir network.

Parameters

- **groupname** (*str, required*) – Name of the Reservoir population.
- **neuron_eq_builder** (*class, optional*) – neuron class as imported from models/neuron_models.
- **synapse_eq_builder** (*class, optional*) – synapse class as imported from models/synapse_models.
- **weInpR** (*float, optional*) – Excitatory synaptic weight between input SpikeGenerator and Reservoir neurons.
- **weRInh** (*int, optional*) – Excitatory synaptic weight between Reservoir population and inhibitory interneuron.
- **wiInhR** (*TYPE, optional*) – Inhibitory synaptic weight between inhibitory interneuron and Reservoir population.
- **weRR** (*float, optional*) – Self-excitatory synaptic weight (Reservoir).

- **sigm** (*int, optional*) – Standard deviation in number of neurons for Gaussian connectivity kernel.
- **rpR** (*float, optional*) – Refractory period of Reservoir neurons.
- **rpInh** (*float, optional*) – Refractory period of inhibitory neurons.
- **num_neurons** (*int, optional*) – Size of Reservoir neuron population.
- **Rconn_prob** (*float, optional*) – Float 0<p<1 to set connection probability within the reservoir
- **adjecency_mtr** (*numpy ndarray 3D of int and float, optional*) – Uses the adjecency matrix to set connections in the reservoir
- **fraction_inh_neurons** (*int, optional*) – Set to None to skip Dale's principle.
- **cutoff** (*int, optional*) – Radius of self-excitation.
- **num_inputs** (*int, optional*) – Number of input currents to each neuron in the Reservoir.
- **num_input_neurons** (*int, optional*) – Number of neurons in the input stage.
- **num_readout_neurons** (*int, optional*) – Number of neurons in the readout stage.
- **spatial_kernel** (*str, optional*) – None defaults to kernel_mexican_1d
- **monitor** (*bool, optional*) – Flag to auto-generate spike and statemonitors.
- **additional_statevars** (*list, optional*) – List of additional state variables which are not standard.
- **debug** (*bool, optional*) – Flag to gain additional information.

Returns Keys to all neuron and synapse groups. Monitors (dictionary): Keys to all spike- and statemonitors. standalone_params (dictionary): Dictionary which holds all parameters to create a standalone network.

Return type Groups (dictionary)

teili.building_blocks.sequence_learning module

This module provides a sequence learning building block. This building block can be used to learn sequences of items

teili.building_blocks.sequence_learning.**sl_params**
Dictionary of default parameters for reservoir.

Type dict

Example

To use the Reservoir building block in your simulation you need to create an object of the class by doing:

```
>>> from teili.building_blocks.reservoir import Reservoir
>>> my_bb = Reservoir(name='my_sequence')
```

If you want to change the underlying neuron and synapse model you need to provide a different equation_builder class:

```
>>> from teili.models.neuron_models import DPI
>>> from teili.models.synapse_models import DPISyn
>>> my_bb = Reservoir(name='my_sequence',
                      neuron_eq_builder=DPI,
                      synapse_eq_builder=DPISyn)
```

If you want to change the default parameters of your building block you need to define a dictionary, which you pass to the building_block:

```
>>> sl_params = {'synInpOrd1e_weight': 1.3,
                 'synOrdMem1e_weight': 1.1,
                 'synMemOrd1e_weight': 0.16,
                 # local
                 'synOrdOrd1e_weight': 1.04,
                 'synMemMem1e_weight': 1.54,
                 # inhibitory
                 'synOrdOrd1i_weight': -1.95,
                 'synMemOrd1i_weight': -0.384,
                 'synCoSOrd1i_weight': -1.14,
                 'synResetOrd1i_weight': -1.44,
                 'synResetMem1i_weight': -2.6,
                 # refractory
                 'gOrdGroups_refP': 1.7 * ms,
                 'gMemGroups_refP': 2.3 * ms
                 }
>>> my_bb = Reservoir(name='my_sequence', block_params=sl_params)
```

class teili.building_blocks.sequence_learning.**SequenceLearning**(**args*, ***kw*)

Bases: *teili.building_blocks.building_block.BuildingBlock*

Sequence Learning Network.

cos_group
Condition of Satisfaction group.

Type neuron group

group
List of keys of neuron population.

Type dict

input_group
SpikeGenerator object to stimulate Reservoir.

Type SpikeGenerator

reset_group
Reset group, to reset network after CoS is met.

Type neuron group

standalone_params
Keys for all standalone parameters necessary for cpp code generation.

Type dict

plot(*duration=None*)
Simple plot for sequence learning network.

Returns The window containing the plot.

Return type pyqtgraph window

teili.building_blocks.sequence_learning.**gen_sequence_learning**(*groupname='Seq'*, *neuron_eq_builder=<class 'teili.models.neuron_models.ExpAdaptIF'>*, *synapse_eq_builder=<class 'teili.models.synapse_models.ReversalSyn*, *num_elements=4*, *num_neurons_per_group=8*, *synInpOrd1e_weight=1.3*, *synOrdMem1e_weight=1.1*, *synMemOrd1e_weight=0.16*, *synOrdOrd1e_weight=1.04*, *synMemMem1e_weight=1.54*, *synOrdOrd1i_weight=-1.95*, *synMemOrd1i_weight=-0.384*, *synCoSOrd1i_weight=-1.14*, *synResetOrd1i_weight=-1.44*, *synResetMem1i_weight=-2.6*, *gOrdGroups_refP=1.7 * msecond*, *gMemGroups_refP=2.3 * msecond*, *num_inputs=1*, *verbose=False*)
Create Sequence Learning Network after the model from Sandamirskaya and Schoener (2010).

Parameters

- **groupname** (*str, optional*) – Base name for building block.
- **neuron_eq_builder** (*teili.models.builder obj, optional*) – Neuron equation builder object.
- **synapse_eq_builder** (*teili.models.builder obj, optional*) – Synapse equation builder object.

- **num_elements** (*int, optional*) – Number of elements in the sequence.
- **num_neurons_per_group** (*int, optional*) – Number of neurons used to remember each item.
- **synInpOrd1e_weight** (*float, optional*) – Parameter specifying the input weight.
- **synOrdMem1e_weight** (*float, optional*) – Parameter specifying the ordinary to memory weight.
- **synMemOrd1e_weight** (*float, optional*) – Parameter specifying the memory to ordinary weight.
- **synOrdOrd1e_weight** (*float, optional*) – Parameter specifying the recurrent weight (ord).
- **synMemMem1e_weight** (*float, optional*) – Parameter specifying the recurrent weight (memory).
- **synOrdOrd1i_weight** (*TYPE, optional*) – Parameter specifying the recurrent inhibitory weight.
- **synMemOrd1i_weight** (*TYPE, optional*) – Parameter specifying the memory to ordinary inhibitory weight.
- **synCoSOrd1i_weight** (*TYPE, optional*) – Parameter specifying the inhibitory weight from cos to ord.
- **synResetOrd1i_weight** (*TYPE, optional*) – Parameter specifying the the inhibitory weight from reset to ord.
- **synResetMem1i_weight** (*TYPE, optional*) – Parameter specifying the the inhibitory weight from reset cos to memory.
- **gOrdGroups_refP** (*TYPE, optional*) – Parameter specifying the refractory period.
- **gMemGroups_refP** (*TYPE, optional*) – Parameter specifying the refractory period.
- **num_inputs** (*int, optional*) – Number of inputs from different source populations.
- **debug** (*bool, optional*) – Debug flag.

Returns Keys to all neuron and synapse groups. Monitors (dictionary): Keys to all spike- and statemonitors. standalone_params (dictionary): Dictionary which holds all parameters to create a standalone network.

Return type Groups (dictionary)

teili.building_blocks.sequence_learning.**plot_sequence_learning**(*Monitors*, *duration=None*)

A simple matplotlib wrapper function to plot network activity.

Parameters **Monitors** (*building_block.monitors*) – Dictionary containing all monitors created by gen_sequence_learning().

Returns Matplotlib figure.

Return type plt.fig

teili.building_blocks.wta module

This module provides different Winner-Takes_all (WTA) circuits.

Beside different dimensionality of the WTA, i.e 1D & 2D, you can select different spatial connectivity and neuron and synapse models.

teili.building_blocks.wta.**wta_params**
Dictionary of default parameters for wta.

Type dict

Todo:

- Generalize for n dimensions

Example

To use the WTA building block in your simulation you need to create an object of the class by doing:

```
>>> from teili.building_blocks.wta import WTA
>>> my_bb = WTA(name='my_wta')
```

If you want to change the underlying neuron and synapse model you need to provide a different equation_builder class:

```
>>> from teili.models.neuron_models import ExpAdaptIF
>>> from teili.models.synapse_models import ReversalSynV
>>> my_bb = WTA(name='my_wta',
                neuron_eq_builder=ExpAdaptIF,
                synapse_eq_builder=ReversalSynV)
```

If you want to change the default parameters of your building block you need to define a dictionary, which you pass to the building_block:

```
>>> wta_params = {'we_inp_exc': 1.5,
                  'we_exc_inh': 1,
                  'wi_inh_exc': -1,
                  'we_exc_exc': 0.5,
                  'wi_inh_inh': -1,
                  'sigm': 3,
                  'rp_exc': 3 * ms,
                  'rp_inh': 1 * ms,
                  'ei_connection_probability': 1,
                  'ie_connection_probability': 1,
                  'ii_connection_probability': 0
                  }
    >>> my_bb = WTA(name='my_wta', block_params=wta_params)
```

class teili.building_blocks.wta.**WTA**(**args*, ***kw*)
Bases: *teili.building_blocks.building_block.BuildingBlock*

A 1 or 2D square Winner-Takes_all (WTA) Building block.

dimensions
Specifies if 1 or 2 dimensional WTA is created.

Type int, optional

num_neurons
Size of WTA neuron population.

Type int, optional

spike_gen
SpikeGenerator group of the BB

Type brian2.Spikegenerator obj.

spikemon_exc
A spike monitor which monitors the activity of the WTA population.

Type brian2.SpikeMonitor obj.

standalone_params
Dictionary of parameters to be changed after standalone code generation.

Type dict

teili.building_blocks.wta.**gen1dWTA**(*groupname*, *neuron_eq_builder=<class 'teili.models.neuron_models.DPI'>*, *synapse_eq_builder=<class 'teili.models.synapse_models.DPISyn'>*, *we_inp_exc=1.5*, *we_exc_inh=1*, *wi_inh_inh=-1*, *wi_inh_exc=-1*, *we_exc_exc=0.5*, *sigm=3*, *rp_exc=3. * msecond*, *rp_inh=1. * msecond*, *num_neurons=64*, *num_inh_neurons=5*, *num_input_neurons=None*, *num_inputs=1*, *num_inh_inputs=2*, *cutoff=10*, *spatial_kernel='kernel_gauss_1d'*, *ei_connection_probability=1*, *ie_connection_probability=1*, *ii_connection_probability=0*, *additional_statevars=[]*, *monitor=True*, *verbose=False*)

Creates a 1D WTA population of neurons, including the inhibitory interneuron population

Parameters

- **groupname** (*str, required*) – Name of the WTA population.
- **neuron_eq_builder** (*class, optional*) – neuron class as imported from models/neuron_models.
- **synapse_eq_builder** (*class, optional*) – synapse class as imported from models/synapse_models.
- **we_inp_exc** (*float, optional*) – Excitatory synaptic weight between input SpikeGenerator and WTA neurons.
- **we_exc_inh** (*float, optional*) – Excitatory synaptic weight between WTA population and inhibitory interneuron.
- **wi_inh_inh** (*float, optional*) – Inhibitory synaptic weight between interneurons.
- **wi_inh_exc** (*float, optional*) – Inhibitory synaptic weight between inhibitory interneuron and WTA population.
- **we_exc_exc** (*float, optional*) – Self-excitatory synaptic weight (WTA).
- **sigm** (*int, optional*) – Standard deviation in number of neurons for Gaussian connectivity kernel.
- **rp_exc** (*float, optional*) – Refractory period of WTA neurons.

- **rp_inh** (*float, optional*) – Refractory period of inhibitory neurons.
- **num_neurons** (*int, optional*) – Size of WTA neuron population.
- **num_inh_neurons** (*int, optional*) – Size of inhibitory interneuron population.
- **num_input_neurons** (*int, optional*) – Size of input population. If None, equal to size of WTA population.
- **num_inputs** (*int, optional*) – Number of input currents to WTA.
- **num_inh_inputs** (*int, optional*) – Number of input currents to the inhibitory group.
- **cutoff** (*int, optional*) – Radius of self-excitation.
- **spatial_kernel** (*str, optional*) – Connectivity kernel for lateral connectivity. Default is 'kernel_gauss_1d'. See tools.synaptic_kernel for more detail.
- **ei_connection_probability** (*float, optional*) – WTA to interneuron connectivity probability.
- **ie_connection_probability** (*float, optional*) – Interneuron to WTA connectivity probability
- **ii_connection_probability** (*float, optional*) – Interneuron to Interneuron connectivity probability.
- **additional_statevars** (*list, optional*) – List of additional state variables which are not standard.
- **monitor** (*bool, optional*) – Flag to auto-generate spike and state monitors.
- **verbose** (*bool, optional*) – Flag to gain additional information.

Returns

Keys to all neuron and synapse groups. monitors (dictionary): Keys to all spike and state monitors. standalone_params (dictionary): Dictionary which holds all

parameters to create a standalone network.

Return type _groups (dictionary)

teili.building_blocks.wta.**gen2dWTA**(*groupname*, *neuron_eq_builder=<class 'teili.models.neuron_models.DPI'>*, *synapse_eq_builder=<class 'teili.models.synapse_models.DPISyn'>*, *we_inp_exc=1.5*, *we_exc_inh=1*, *wi_inh_inh=-1*, *wi_inh_exc=-1*, *we_exc_exc=2.0*, *sigm=2.5*, *rp_exc=3. * msecond*, *rp_inh=1. * msecond*, *num_neurons=64*, *num_inh_neurons=5*, *num_input_neurons=None*, *num_inputs=1*, *num_inh_inputs=2*, *cutoff=10*, *spatial_kernel='kernel_gauss_2d'*, *ei_connection_probability=1.0*, *ie_connection_probability=1.0*, *ii_connection_probability=0.1*, *additional_statevars=[]*, *monitor=True*, *verbose=False*)

Creates a 2D square WTA population of neurons, including the inhibitory interneuron population

Parameters

- **groupname** (*str, required*) – Name of the WTA population.

- **neuron_eq_builder** (*class, optional*) – neuron class as imported from models/neuron_models.
- **synapse_eq_builder** (*class, optional*) – synapse class as imported from models/synapse_models.
- **we_inp_exc** (*float, optional*) – Excitatory synaptic weight between input SpikeGenerator and WTA neurons.
- **we_exc_inh** (*int, optional*) – Excitatory synaptic weight between WTA population and inhibitory interneuron.
- **wi_inh_inh** (*int, optional*) – Self-inhibitory weight of the interneuron population.
- **wi_inh_exc** (*TYPE, optional*) – Inhibitory synaptic weight between inhibitory interneuron and WTA population.
- **we_exc_exc** (*float, optional*) – Self-excitatory synaptic weight (WTA).
- **sigm** (*int, optional*) – Standard deviation in number of neurons for Gaussian connectivity kernel.
- **rp_exc** (*float, optional*) – Refractory period of WTA neurons.
- **rp_inh** (*float, optional*) – Refractory period of inhibitory neurons.
- **num_neurons** (*int, optional*) – Size of WTA neuron population.
- **num_inh_neurons** (*int, optional*) – Size of inhibitory interneuron population.
- **num_input_neurons** (*int, optional*) – Size of input population. If None, equal to size of WTA population.
- **num_inputs** (*int, optional*) – Number of input currents to WTA.
- **num_inh_inputs** (*int, optional*) – Number of input currents to the inhibitory group.
- **cutoff** (*int, optional*) – Radius of self-excitation.
- **spatial_kernel** (*str, optional*) – Description
- **ei_connection_probability** (*float, optional*) – WTA to interneuron connectivity probability.
- **ie_connection_probability** (*float, optional*) – Interneuron to excitory neuron connectivity probability.
- **ii_connection_probability** (*float, optional*) – Interneuron to interneuron neuron connectivity probability.
- **additional_statevars** (*list, optional*) – List of additional state variables which are not standard.
- **monitor** (*bool, optional*) – Flag to auto-generate spike and statemonitors.
- **verbose** (*bool, optional*) – Flag to gain additional information.

Returns

Keys to all neuron and synapse groups. monitors (dictionary): Keys to all spike and state monitors. standalone_params (dictionary): Dictionary which holds all

parameters to create a standalone network.

Return type _groups (dictionary)

teili.building_blocks.wta.**set_wta_tags**(*self*, *_groups*)
Sets default tags to a WTA network

Parameters **_groups** (*dictionary*) – Keys to all neuron and synapse groups.

No Longer Returned: Tags will be added to all _groups passed. They follow this structure:

tags = {'mismatch' [(bool, 0/1)]

'noise : (bool, 0/1) 'level': int 'sign': str (exc/inh/None) 'target sign': str (exc/inh/None) 'num_inputs' : int (0 if not Neuron group), 'bb_type' : str (WTA/3-WAY), 'group_type' : str (Neuron/Connection/ SpikeGen) 'connection_type' : str (rec/lateral/fb/ff/None)

}

Module contents

teili.core package

Submodules

teili.core.groups module

Wrapper class for brian2 Group class.

Todo:

- Check if *shared* works for neuron as well.
- **Raise error (understandable)** if addStateVariable is called before synapses are connected.
- Find out, if it is possible to have delay as state variable for Connections.
- Some functionality of the package is not compatible with subgroups yet.
- **This: *self.register_synapse = self.source.register_synapse* is not ideal,** as it is not necessary to register a synapse for subgroups!

---

class teili.core.groups.**Connections**(**args*, ***kw*)
Bases: brian2.synapses.synapses.Synapses, *teili.core.groups.TeiliGroup*, brian2.core.names.Nameable

This class is a subclass of Synapses.

You can use it as a Synapses, and everything will be passed to Synapses. Alternatively, you can also pass an EquationBuilder object that has all keywords and parameters.

equation_builder
Class which builds the synapse model.

Type teili

input_number
Number of input to post synaptic neuron. This variable takes care of the summed issue present in brian2.

Type int

parameters
Dictionary of parameter keys and values of the synapse model.

Type dict

verbose
Flag to print more detail about synapse generation.

Type bool

__setattr__(*key*, *value*)
Function to set arguments to synapses

Parameters

- **key** (*str*) – Name of attribute to be set
- **value** (*TYPE*) – Description

connect(*condition=None*, *i=None*, *j=None*, *p=1.0*, *n=1*, *skip_if_invalid=False*, *namespace=None*, *level=0*, ***Kwargs*)
Wrapper function to make synaptic connections among neurongroups.

Parameters

- **condition** (*bool, str, optional*) – A boolean or string expression that evaluates to a boolean. The expression can depend on indices i and j and on pre- and post-synaptic variables. Can be combined with arguments n, and p but not i or j.
- **i** (*int, str, optional*) – Source neuron index.
- **j** (*int, str, optional*) – Target neuron index.
- **p** (*float, optional*) – Probability of connection.
- **n** (*int, optional*) – The number of synapses to create per pre/post connection pair. Defaults to 1.
- **skip_if_invalid** (*bool, optional*) – Flag to skip connection if invalid indices are given.
- **namespace** (*str, optional*) – namespace of this synaptic connection.
- **level** (*int, optional*) – Distance to the input layer needed for tags.
- ****Kwargs** – Additional keyword arguments.

plot()
Simple visualization of synapse connectivity (connected dots and connectivity matrix)

class teili.core.groups.**Neurons**(**args*, ***kw*)
Bases: brian2.groups.neurongroup.NeuronGroup, *teili.core.groups.TeiliGroup*

This class is a subclass of NeuronGroup.

You can use it as a NeuronGroup, and everything will be passed to NeuronGroup. Alternatively, you can also pass an EquationBuilder object that has all keywords and parameters.

equation_builder
Class which describes the neuron model equation and all properties and default parameters. See /model/builder/neuron_equation_builder.py and models/neuron_models.py.

Type TYPE

initialized
Flag to register Neurons' population with TeiliGroups.

Type bool

num_inputs
Number of possible synaptic inputs. This overcomes the summed issue present in brian2.

Type int

num_synapses
Number of synapses projecting to post-synaptic neuron group.

Type int

synapses_dict
Dictionary with all synapse names and their respective synapse index.

Type dict

verbose
Flag to print more details of neuron group generation.

Type bool

__getitem__(*item*)
Taken from brian2/brian2/groups/neurongroup.py

Parameters **item** (*TYPE*) – Description

Returns The respective neuron subgroup.

Return type *TeiliSubgroup*

Raises

- **IndexError** – Error that indicates that size of subgroup set by start and stop is out of bounds.
- **TypeError** – Error to indicate that wrong syntax has been used.

__setattr__(*key*, *value*)
Set attribute method.

Parameters

- **key** (*TYPE*) – key of attribute to be set.
- **value** (*TYPE*) – value of respective key to be set.

register_synapse(*synapsename*)
Registers a Synapse so we know the input number.

It counts all synapses connected with one neuron group.

Raises **ValueError** – If too many synapses project to a given post-synaptic neuron group this error is raised. You need to increae the number of inputs parameter.

Parameters **synapsename** (*str*) – Name of the synapse group to be registered.

Returns

dictionary with all synapse names and their respective synapse index.

Return type dict

class teili.core.groups.**TeiliGroup**(**args*, ***kw*)
Bases: brian2.groups.group.Group

just a bunch of methods that are shared between neurons and connections class Group is already used by brian2.

standalone_params
Dictionary of standalone parameters.

Type dict

standalone_vars
List of standalone variables.

Type list

str_params
Name of parameters to be updated.

Type dict

_add_mismatch_param(*param*, *std=0*, *lower=None*, *upper=None*, *seed=None*)
This function sets the input parameter (param) to a value (new_param) drawn from a normal distribution with standard deviation (std) expressed as a fraction of the current value (old_param).

Parameters

- **param** (*str*) – name of the parameter to which the mismatch has to be added
- **std** (*float*) – normalized standard deviation, expressed as a fraction of the current parameter (e.g.: std = 0.1 means that the new value will be sampled from a normal distribution with standard deviation of 0.1*old_param, with old_value being the current value of the attribute param) (default: 0)
- **lower** (*float, optional*) – lower bound for the parameter mismatch, expressed as a fraction of the standard deviation, see note below. (default: -1/std)
- **upper** (*float, optional*) – upper bound for the parameter mismatch, expressed as a fraction of the standard deviation, see note below. (default: inf)
- **seed** (*int, optional*) – seed value for the random generator. Set the seed if you want to make the mismatch values reproducible across simulations. (default: None)

NOTE: the outuput value (new_param) is drawn from a Gaussian distribution with parameters: mean: old_param standard deviation: std * old_param lower bound: lower * std * old_param + old_param (default: 0, i.e. lower = -1/std) upper bound: upper * std * old_param + old_param (default: inf)

using the function truncnorm. For details, see: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.truncnorm.html

The seed will not work in standalone mode so far (TODO)

Raises

- **NameError** – if one of the specified parameters in the disctionary is not included in the model.
- **AttributeError** – if the input parameter to be changed does not have units
- **UserWarning** – if the lower bound is negative (i.e. if lower < -1/std) (e.g. if the specified parameter is a current, negative values are meaningless)

Example

Adding mismatch to Itau in a population of 100 DPI neurons using _add_mismatch_param(). >>> from teili.models.neuron_models import DPI >>> testNeurons = Neurons(100, equation_builder=DPI(num_inputs=2)) >>> testNeurons._add_mismatch_param(param='Itau', std=0.1)

This will truncate the distribution at 0, to prevent Itau to become negative.

To specify also the lower bound as 2 times the standard deviation: >>> testNeurons._add_mismatch_param(param='Ith',

std=0.1, lower=-2)

TODO: Consider the mismatch for the parameter 'Cm' as a separate case.

TODO: Add UserWarning if mismatch has been added twice both in numpy and standalone mode.

add_mismatch(*std_dict=None*, *seed=None*, *verbose=False*)

This function is a wrapper for the method _add_mismatch_param() to add mismatch to a dictionary of parameters specified in the input dictionary (std_dict). Mismatch is drawn from a Gaussian distribution with mean equal to the parameter's current value.

If no dictionary is given, 20% mismatch is added to all the parameters of the model except variables specified in the no_mismatch_parameter file.

Note: if you want to specify also lower and upper bound of the mismatch distribution see _add_mismatch_param() which adds mismatch to a single parameter.

Parameters

- **std_dict** (*dict, optional*) – dictionary of parameter names as keys and standard deviation as values. Standard deviations are expressed as fraction of the current parameter value. If empty, 20% of missmatch will be added to all variables (example: if std_dict = {'Itau': 0.1}, the new parameter value will be sampled from a normal distribution with standard deviation of 0.1*old_param, with old_param being the old parameter value)
- **seed** (*int, optional*) – seed value for the random generator. Set the seed if you want to make the mismatch values reproducible across simulations. The random generator state before calling this method will be restored after the call in order to avoid effects to the rest of your simulation (default = None)

Example

Adding mismatch to 100 DPI neurons.

First create the neuron population (this sets parameter default values): >>> from teili.models.neuron_models import DPI >>> testNeurons = Neurons(100, equation_builder=DPI(num_inputs=2))

Store the old values as array: >>> old_param_value = np.copy(getattr(testNeurons, 'Itau'))

Add mismatch to the neuron Itau with a standard deviation of 10% of the current bias values: >>> testNeurons.add_mismatch({'Itau': 0.1})

add_state_variable(*name*, *unit=1*, *shared=False*, *constant=False*, *changeInStandalone=True*)

This method allows you to add a state variable.

Usually a state variable is defined in equations, that is changeable in standalone mode. If you pass a value, it will directly set it and decide based on that value, if the variable should be shared (scalar) or not (vector).

Parameters

- **name** (*str*) – Name of state variable.
- **unit** (*int, optional*) – Unit of respective state variable.
- **shared** (*bool, optional*) – Flag to indicate if state variable is shared.
- **constant** (*bool, optional*) – Flag to indicate if state variable is constant.
- **changeInStandalone** (*bool, optional*) – Flag to indicate if state variable should be subject to on-line change in cpp standalone mode.

add_subexpression(*name*, *dimensions*, *expr*)

This method allows you to add a subexpression (like a state variable but a string that can be evaluated over time) You can e.g. add a timedArray like that: >>> neuron_group.add_subexpression('I_arr',nA.dim,'timed_array(t)') (be aware, that you need to add a state variable I_arr first, that is somehow connected to other variables, so run_regularly may be an easier solution for your problem)

Parameters

- **name** (*str*) – name of the expression.
- **dimensions** (*brian2.units.fundamentalunits.Dimension*) – dimension of the expression.
- **expr** (*str*) – the expression.

get_params(*params=None*, *verbose=False*)

This function gets parameter values of neurons or synapses. In standalone mode, it only works after the simulation has been run.

Parameters **params** (*list, optional*) – list of parameters that should be retrieved. If params = None (default), a dictionary of all parameters with their current values is returned

Returns dictionary of parameters with their values

Return type dict

import_eq(*filename*)

Function to import pre-defined neuron/synapse models.

Parameters **filename** (*str*) – path/to/your/model.py Usually synapse models can be found in teiliApps/models/equations.

Returns

Dictionary keywords with all relevant kwargs, to generate model.

Return type Dictionary

property model

This property allows the user to only show the model of a given member.

Returns Dictionary only containing model equations.

Return type dict

print_equations()

This function print the equation underlying the TeiliGroup member.

set_params(*params*, ***kwargs*)
This function sets parameters on members of a Teiligroup.

Parameters

- **params** (*dict*) – Key and value of parameter to be set.
- ****kwargs** – Additional keyword arguments.

Returns The parameters set.

Return type dict

update_param(*parameter_name*, *verbose=True*)
This is used to update string based params during run (e.g. with gui).

Parameters **parameter_name** (*str*) – Name of parameter to be updated.

class teili.core.groups.**TeiliSubgroup**(**args*, ***kw*)
Bases: brian2.groups.subgroup.Subgroup

This helps to make Subgroups compatible, otherwise the same as Subgroup.

register_synapse
Register a synapse group to TeiliGroup.

Type fct

property num_synapses
Property to overcome summed issue present in brian2.

Returns

Number of synapses that converge to the same post-synaptic neuron group.

Return type int

teili.core.groups.**print_paramdict**(*paramdict*)
This function prints a params dictionary for get and set_params.

Parameters **paramdict** (*dict, required*) – Parameter keys and values to be set.

teili.core.groups.**set_params**(*briangroup*, *params*, *ndargs=None*, *raise_error=False*, *verbose=False*)
This function takes a params dictionary and sets the parameters of a briangroup.

Parameters

- **brianggroup** (*brian2.groups.group, required*) – Neuron or Synapsegroup to set parameters on.
- **params** (*dict, required*) – Parameter keys and values to be set.
- **raise_error** (*boolean, optional*) – determines if an error is raised if a parameter does not exist as a state variable of the group.
- **ndargs** (*dict, optional*) – Addtional attribute arguments.
- **verbose** (*bool, optional*) – Flag to get more details about parameter setting process. States are not printed in cpp standalone mode before the simulation has been run

teili.core.network module

Wrapper class for Network class of brian2.

This wrapper provides a more flexible interface, especially to change parameters on the fly after compilation

---

Todo:

- function that plots the whole network

---

class teili.core.network.**TeiliNetwork**(**args*, ***kw*)

Bases: brian2.core.network.Network

This is a subclass of brian2.Network.

This subclass does the same thing plus some additional methods for convenience and functionality to allow real time plotting and gui.

blocks

Description

Type list

has_run

Flag to indicate if network has been simulated already.

Type bool

standalone_params

Dictionary of standalone parameters.

Type dict

thread

Description

Type TYPE

add(**objs*)

Does the same thing as Network.add (adding Groups to the Network)

It also adds the groups to a list for the parameter gui.

Parameters ***objs** – arguments (brian2 objects which should be added to the network).

add_standalone_params(***params*)

Function to a add standalone parameter to the standaloneParam dict.

These parameters can be changed after building w/o recompiling the network.

Parameters ****params** (*dict, required*) – Dictionary with parameter to be added to standalone_params.

build(*report='stdout'*, *report_period=10.0 * second*, *namespace=None*, *profile=True*, *level=0*, *recompile=False*, *standalone_params=None*, *clean=True*, *verbose=True*)

Building the network.

Parameters

- **report** (*bool, optional*) – Flag to provide more detailed information during run.
- **report_period** (*brian2.unit, optional*) – How often should be reported (unit time).

- **namespace** (*None, optional*) – Namespace containing all names of the network to be built.
- **profile** (*bool, optional*) – Flag to enable profiling of the network in terms of execution time, resources etc. .
- **level** (*int, optional*) – Description.
- **recompile** (*bool, optional*) – Flag to indicate if network should rather be recompiled than used based on a prior build. Set this to False if you want to only change parameters rather than network topology.
- **standalone_params** (*dict, optional*) – Dictionary with standalone parameters which should be changed.
- **clean** (*bool, optional*) – Flag to clean-up standalone directory.

has_run = False

property neurongroups
property to conveniently get all neurongroups in the network

Returns A dictionary of all neurongroups (e.g. for looping over them)

Return type dict

print_params()
This functions prints all standalone parameters (cpp standalone network).

run(*duration=None*, *standalone_params={}*, *verbose=True*, ***kwargs*)
Wrapper function to simulate a network for the given duration.

Parameters which should be changeable, especially after cpp compilation, need to be provided to standalone_params.

Parameters

- **duration** (*brain2.unit, optional*) – Simulation time in s, i.e. 1000 * ms.
- **standalone_params** (*dict, optional*) – Dictionary whose keys refer to parameters which should be changeable in cpp standalone mode.
- **verbose** (*bool, optional*) – set to False if you don't want the prints, it is set to True by default, as there are things that can go wrong during string replacement etc. so it is better to have a look manually.
- ****kwargs** (*optional*) – Additional keyword arguments.

run_as_thread(*duration*, ***kwargs*)
Running network in a thread.

Parameters

- **duration** (*brain2.unit, optional*) – Simulation time in ms, i.e. 100 * ms.
- ****kwargs** (*optional*) – Additional keyword arguments.

property spikemonitors
property to conveniently get all spikemonitors in the network

Returns A dictionary of all spike monitors (e.g. for looping over them)

Return type dict

property statemonitors
property to conveniently get all statemonitors in the network

Returns A dictionary of all statemonitors (e.g. for looping over them)

Return type dict

property synapses
property to conveniently get all synapses in the network

Returns A dictionary of all synapses (e.g. for looping over them).

Return type dict

Module contents

teili.models package

Subpackages

teili.models.builder package

Subpackages

teili.models.builder.templates package

Submodules

teili.models.builder.templates.neuron_templates module

This file contains dictionaries of neuron equations or modules, combined by the neuron equation builder. Each template consists of a dictionary containing the relevant equations and a corresponding parameter dictionary.

For usage example plaese refer to *teili/models/neuron_models.py*

Contributing guide: * Dictionary describing the neuron model

- Describing the model dynamics, including all used variables and their units.
- Required keys in the dictionary: 'model', 'threshold', 'reset'.
- name: modelname_template
- **Corresponding dictionary containing default/init parameters.**
  - name: modelname_template_params

TBA: How to add dictionaries to Model dictionaries (see bottom)

teili.models.builder.templates.neuron_templates.**none_params = {}**
LIF neuron model with stochastic decay taken from Wang et al. (2018). Please refer to this paper for more information. Note that this model was conceptualized in discrete time with backward euler scheme and an integer operation. An state updader with x_new = f(x,t) and defaultclock.dt = 1*ms in the code using this model.

teili.models.builder.templates.neuron_templates.**v_noise = {'model': '\n %Inoise = xi*Anois**
Adds spatial location to neuron locate at the soma. This additional information is **not** set by default.

teili.models.builder.templates.neuron_templates.**v_quad_current = {'model': '\n %Iexp = k***
Paramters for the quadratic model taken from Nicola & Clopath 2017. Please refer to this paper for more information. The parameter k represents k = 1/Rin in the original study.

teili.models.builder.templates.synapse_templates module

This file contains dictionaries of synapse equations or modules, combined by the synapse equation builder. Each template consists of a dictionary containing the relevant equations and a corresponding parameter dictionary.

For usage example plaese refer to *teili/models/synapse_models.py*

Contributing guide: * Dictionary describing the synapse model

- Describing the model dynamics, including all used variables and their units.
- Required keys in the dictionary: 'model', 'on_pre', 'on_post'.
- name: modelname_template
- **Corresponding dictionary containing default/init parameters.**
  - name: modelname_template_params

If you want to override an equation add '%' before the variable of your block's explicit equation.

Example: Let's say we have the simplest model (current one with template equation), and you're implementing a new block with this explicit equation : d{synvar_e}/dt = (-{synvar_e})**2 / synvar_e. If you want to override the equation already declared in the template: d{synvar_e}/dt = (-{synvar_e}) / tausyne + kernel_e, your equation will be : %d{synvar_e}/dt = (-{synvar_e})**2 / synvar_e

teili.models.builder.templates.synapse_templates.**conductance = {'model': '\n dgI/dt = (-g**

Standard parameters for conductance based models TODO: For inhibitory synapse EIe is negative. Could this thus be a problem?

teili.models.builder.templates.synapse_templates.**deterministic_counter_params = {'re_init_**

Dictionary of keywords:

These dictionaries contains keyword and models and parameters names useful for the __init__ subroutine Every new block dictionaries must be added to these definitions. synaptic_equations is a dictionary that gathers all models and parameters.

teili.models.builder.templates.synapse_templates.**dpi_shunt_params = {'Csyn': 1.5 * pfarad**

Exponentially decaying synapse model using quantized stochastic decay taken from Wang et al. (2018). Please refer to this paper for more information. Note that this model was conceptualized in discrete time with backward euler scheme and an integer operation. An state updader with x_new = f(x,t) and defaultclock.dt = 1*ms in the code using this model is necessary.

teili.models.builder.templates.synapse_templates.**quantized_stochastic_params = {'gain_syn'**

Plasticity blocks You need to declare two set of parameters for every block: * current based models * conductance based models

teili.models.builder.templates.synapse_templates.**quantized_stochastic_stdp_params = {'A_ga**

Kernels Blocks: You need to declare two set of parameters for every block: * current based models * conductance based models

teili.models.builder.templates.synapse_templates.**stdgm_params = {'Ipred_plast': '0.0', 'Q_**

The set of activation encapsulates *StateVariables* which are needed for Activity Dependent Plasticity (ADP) paradigm. ADP adjusts the inhibitory weight according to the 'activity' of the post-synaptic neuron. These equations are required by synapses projecting to _adp neuronal populations.

teili.models.builder.templates.synapse_templates.**unit_less = {'model': '\n ', 'on_post':**

Structural plasticity blocks: You need to declare two set of parameters for every block: * current based models * conductance based models

These blocks add a counter that keep track of weight updates so that we can measure how active a synapse is. Note that stochastic structural plasticity should only be used in conjunction with synaptic plasticity and Teili's run_regularly functions.

Module contents

Submodules

teili.models.builder.combine module

This file contains a set of functions used by the library to combine models.

Combine_neu_dict is used for combining neuron models Combine_syn_dict is used for combining neuron models

Both functions use the var_replacer when the special overwrite character '%' is found in the equations. If '%' is found in the middle of the equation, like a modulo operation, substitutions are not performed.

Example

To use combine_neu_dict:

```
>>> from teili.models.builder.combine import combine_neu_dict
>>> combine_neu_dict(eq_templ, param_templ)
```

To use combine_syn_dict: >>> from teili.models.builder.combine import combine_syn_dict >>> combine_syn_dict(eq_templ, param_templ)

teili.models.builder.combine.**combine_neu_dict**(*eq_templ*, *param_templ*)

Function to combine neuron models into a single neuron model.

This library offers this ability in order to combine single blocks into bigger and more complex Brian2 compatible models. combine_neu_dict also makes it possible to delete or overwrite an explicit function with the use of the special character '%'. .. rubric:: Example

Example with two different dictionaries both containing the explicit function for the variable 'x':

```
>>> x = theta
>>> %x = gamma
>>> x = gamma
```

'%x' without any assignment will simply delete the variable from the output and from the parameter dictionary.

To combine two dictionaries:

```
>>> combine_neu_dict(eq_templ, param_templ)
```

Parameters

- **eq_templ** (*dict*) – Dictionary containing different keywords and equations
- **param_templ** (*dict*) – Dictionary containing different parameters for the equation

Returns ['model'] Actually neuron model differential equation. dict: ['threshold'] Dictionary with equations specifying behaviour of synapse to post-synaptic spike. dict: ['reset'] Dictionary with equations specifying behaviour of synapse to pre-synaptic spike. dict: ['parameters'] Dictionary of parameters.

Return type dict

teili.models.builder.combine.**combine_syn_dict**(*eq_templ*, *param_templ*)

Function to combine synapse models into a single synapse model.

This library offers this ability in order to combine single blocks into bigger and more complex Brian2 compatible models. combine_syn_dict also makes it possible to delete or overwrite an explicit function with the use of the special character '%'. Example with two different dictionaries both containing the explicit function for the variable 'x': .. rubric:: Example

Example with two different dictionaries both containing the explicit function for the variable 'x':

```
>>> x = theta
>>> %x = gamma
>>> x = gamma
```

'%x' without any assignment will simply delete the variable from the output and from the parameter dictionary.

To combine two dictionaries:

```
>>> combine_syn_dict(eq_templ, param_templ)
```

Parameters

- **eq_templ** (*dict*) – Dictionary containing different keywords and equations.
- **param_templ** (*dict*) – Dictionary containing different parameters for the equation.

Returns ['model'] Actually neuron model differential equation. dict: ['on_post'] Dictionary with equations specifying behaviour of synapse to post-synaptic spike. dict: ['on_pre'] Dictionary with equations specifying behaviour of synapse to pre-synaptic spike. dict: ['parameters'] Dictionary of parameters.

Return type dict

teili.models.builder.combine.**var_replacer**(*first_eq*, *second_eq*, *params*)

Function to delete variables from equations and parameters.

It works with couples of strings and a dict of parameters: first_eq, second_eq and params It searches for every line in second_eq for the special character '%' removing it, and then searching for the variable (even if in differential form '%dx/dt') and erasing every line in first_eq starting with that variable (every explicit equation). If the character '=' or ':' is not in the line containing the variable in second_eq the entire line would be erased.

Parameters

- **first_eq** (*string*) – The first subset of equations that we want to expand or overwrite.
- **second_eq** (*string*) – The second subset of equations which will be added to first_eq. It also contains '%' for overwriting or erasing lines in first_eq.
- **params** (*dict*) – Dictionary of parameters to be replaced.

Returns The first_eq string containing the replaced variable equations. result_second_eq: The second_eq string without the lines containing the special character '%'. params: The parameter dictionary not containing the removed/replaced variables.

Return type result_first_eq

Examples

```
>>> '%x = theta' --> 'x = theta'
>>> '%x' --> ''
```

This feature allows to remove equations in the template that we don't want to compute by writing '%[variable]' in the other equation blocks.

To replace variables and lines:

```
>>> from teili.models.builder.combine import var_replacer
>>> var_replacer(first_eq, second_eq, params)
```

teili.models.builder.neuron_equation_builder module

This file contains a class that manages a neuron equation.

And it prepares a dictionary of keywords for easy synapse creation. It also provides a function to add lines to the model.

Example

To use the NeuronEquationBuilder "on the fly" you can initialize it as DPI neuron:

```
>>> from teili.models.builder.neuron_equation_builder import NeuronEquationBuilder
>>> num_inputs = 2
>>> my_neu_model = NeuronEquationBuilder.__init__(base_unit='current', adaptation=
↪'calcium_feedback',
                                  integration_mode='exponential', leak='leaky',
                                  position='spatial', noise='none')
>>> my_neuron.add_input_currents(num_inputs)
```

Or if you have a pre-defined neuron model you can import this dictionary as:

```
>>> from teili.models.builder.neuron_equation_builder import NeuronEquationBuilder
>>> my_neu_model = NeuronEquationBuilder.import_eq(
    '~/teiliApps/equations/DPI', num_inputs=2)
```

in both cases you can pass it to Neurons:

```
>>> from teili.core.groups import Neurons
>>> my_neuron = Neurons(2, equation_builder=my_neu_model, name="my_neuron")
```

Another way of using it is to import the DPI class directly:

```
>>> from teili.models.neuron_models import DPI
>>> from teili.core.groups import Neurons
>>> my_neuron = Neurons(2, equation_builder=DPI(num_inputs=2), name="my_neuron")
```

class teili.models.builder.neuron_equation_builder.**NeuronEquationBuilder**(*keywords=None*, *base_unit='current'*, *num_inputs=1*, *verbose=False*, ***kwargs*)

Bases: object

Class which builds neuron equation according to pre-defined properties such as spike-frequency adaptation, leakage etc.

keywords
Dictionary containing all relevant keywords and equations for brian2, such as model, refractory, reset, threshold and parameters.

Type dict

num_inputs
Number specifying how many distinct neuron population project to the target neuron population.

Type int

verbose
Flag to print more detailed output of neuron equation builder.

Type bool

__call__ (*num_inputs*)
In the recommended way of using Neurons as provided bey teili the neuron model is imported from teili.models.neuron_models as properly initialised python object in which the number of incoming current, i.e. num_inputs, is set during the initialisation of the class. However, teili also supports to initialise the *Equation_builder* using a user-specified model without the need to implement the model directly in the existing software stack. This allows faster development time and mre flexibility as all functionality of teili is provided to user-specified models. This function allows the user to set the num_inputs argument to non-standard neuron model.

An usage example can be found in *teiliApps/tutorials/neuron_synapse_builderobj_tutorial.py* :param num_inputs: Number specifying how many distinct

neuron populations project to the target neuron population.

Returns A deep copy of the NeuronEquationBuilder object.

Return type NeuronEquationBuilder obj.

add_input_currents (*num_inputs*)
Automatically adds the input current line according to num_inputs.

It also adds all these input currents as state variables.

Example

```
>>> Iin = Ie0 + Ii0 + Ie1 + Ii1 + ... + IeN + IiN (with N = num_inputs)
```

Parameters **num_inputs** (*int*) – Number of inputs to the post-synaptic neuron

add_state_vars (*stateVars*)
this function adds state variables to neuron equation by just adding a line to the neuron model equation.

Parameters **stateVars** (*dict*) – State variable to be added to neuron model

export_eq (*filename*)
Function to export generated neuron model to a file.

Parameters **filename** (*str*) – Path/to/location/to/store/neuron_model.py.

classmethod import_eq(*filename*, *num_inputs=1*)
Function to import pre-defined neuron_model.

num_inputs is used to add additional input currents that are used for different synapses that are summed.

Parameters

- **filename** (*str*) – Path/to/location/where/model/is/stored/neuron_model.py.
- **num_inputs** (*int*) – Number of inputs to the post-synaptic neuron.

Returns

Object containing keywords (dict) with all relevant keys, to generate neuron_model.

Return type NeuronEquationBuilder obj.

print_all()
Method to print all dictionaries within a neuron model

teili.models.builder.neuron_equation_builder.**print_neuron_model**(*Neuron_group*)
Function to print keywords of a Neuron model Usefull to check the entire equation and parameter list

Parameters **group** (*Neuron*) – Synaptic group

NOTE: Even if mismatch is added, the values that are shown and not subject to mismatch

teili.models.builder.neuron_equation_builder.**print_param_dictionaries**(*Dict*)
Function to print dictionaries of parameters in an ordered way.

Parameters **Dict** (*dict*) – Parameter dictionary to be printed.

teili.models.builder.synapse_equation_builder module

This file contains a class that manages a synapse equation.

And it prepares a dictionary of keywords for easy synapse creation. It also provides a function to add lines to the model.

Example

To use the SynapseEquationBuilder "on the fly" you can initialize it as a DPI neuron:

```
>>> from teili.models.builder.synapse_equation_builder import SynapseEquationBuilder
>>> my_syn_model = SynapseEquationBuilder.__init__(base_unit='DPI',
                                                   plasticity='non_plastic')
```

Or if you have a pre-defined synapse model you can import this dictionary as follows:

```
>>> from teili.models.builder.synapse_equation_builder import SynapseEquationBuilder
>>> my_syn_model = SynapseEquationBuilder.import_eq(
    'teiliApps/equations/DPISyn')
```

In both cases you can pass it to Connections:

```
>>> from teili.core.groups import Connections
>>> my_synapse = Connections(testNeurons, testNeurons2,
                             equation_builder=DPISyn, name="my_synapse")
```

Another way of using it is to import the DPI class directly:

```
>>> from teili.models.synapse_models import DPISyn
>>> from teili.core.groups import Connections
>>> my_synapse = Connections(testNeurons, testNeurons2,
              equation_builder=DPISyn, name="my_synapse")
```

class teili.models.builder.synapse_equation_builder.**SynapseEquationBuilder**(*keywords=None*, *base_unit='current'*, *verbose=False*, ***kwargs*)

Bases: object

Class which builds synapse equation.

keywords
Dictionary containing all relevant keywords and equations for brian2, such as model, on_post, on_pre and parameters

Type dict

keywords_original
Dictionary containing all needed keywords for equation builder.

Type dict

verbose
Flag to print more detailed output of neuron equation builder.

Type bool

__call__()
This allows the user to call the object like a class in order to make new objects.

Maybe this use is a bit confusing, so rather not use it.

Returns A deep copy of the SynapseEquationBuilder object.

Return type SynapseEquationBuilder obj.

export_eq(*filename*)
Function to export generated neuron model to a file.

Parameters **filename** (*str*) – path/where/you/store/your/model.py Usually synapse models are stored in teili/models/equations

classmethod import_eq(*filename*)
Function to import pre-defined synapse_model.

Parameters **filename** (*str*) – path/to/your/synapse/model.py Usually synapse models can be found in teili/models/equations.

Returns

Object containing keywords (dict) with all relevant keys, to generate synapse_model.

Return type SynapseEquationBuilder obj.

Examples

synapse_object = SynapseEquationBuilder.import_eq('teili/models/equations/DPISyn')

print_all()
Method to print all dictionaries within a synapse model

set_input_number(*input_number*)
Sets the input number of synapse.

This is needed to overcome the summed issue in brian2.

Parameters **input_number** (*int*) – Synapse's input number

teili.models.builder.synapse_equation_builder.**print_param_dictionaries**(*Dict*)
Function to print dictionaries of parameters in an ordered way.

Parameters **Dict** (*dict*) – Parameter dictionary to be printed.

teili.models.builder.synapse_equation_builder.**print_synaptic_model**(*synapse_group*)

Function to print keywords of a synaptic model

Usefull to check the entire equation and parameter list

Args: Synaptic group(Connections) : Synaptic group

Note: Even if mismatch is added, the values that are shown and not subject to mismatch

Module contents

teili.models.equations package

Submodules

teili.models.equations.BraderFusiSynapses module

teili.models.equations.DPI module

teili.models.equations.DPIShunt module

teili.models.equations.DPISyn module

teili.models.equations.DPIstdp module

teili.models.equations.ExpAdaptIF module

teili.models.equations.ReversalSynV module

teili.models.equations.ReversalSynVfusi module

teili.models.equations.StdpSynV module

Module contents

teili.models.parameters package

Submodules

teili.models.parameters.constants module

This file contains shared constants for synapses and neuron models

teili.models.parameters.dpi_neuron_param module

This file contains default parameter for dpi neuron. For more details on model see models/equations/dpi_neuron.py

teili.models.parameters.dpi_neuron_param.**parameters**
Neuron parameters

Type dict

teili.models.parameters.dpi_shunting_synapse_param module

This file contains default parameter for dpi shunting synapse. For more details on model see models/equations/dpi_shunting_synapse.py

teili.models.parameters.dpi_shunting_synapse_param.**parameters**
Synapse parameters

Type dict

teili.models.parameters.dpi_synapse_param module

This file contains default parameter for dpi dendritic synapse. For more details on model see models/equations/dpi_synapse.py

teili.models.parameters.dpi_synapse_param.**parameters**
Synapse parameters

Type dict

teili.models.parameters.exp_adapt_if_param module

teili.models.parameters.exp_chip_stdp_syn_param module

teili.models.parameters.exp_syn_param module

teili.models.parameters.lif_chip_param module

Module contents

Submodules

teili.models.neuron_models module

This contains subclasses of NeuronEquationBuilder with predefined common parameters

class teili.models.neuron_models.**DPI**(*num_inputs=1*)
Bases: *teili.models.builder.neuron_equation_builder.NeuronEquationBuilder*

This class provides you with all equations to simulate a current-based exponential, adaptive leaky integrate and fire neuron as implemented on the neuromorphic chips by the NCS group. The neuronmodel follows the DPI neuron which was published in 2014 (Chicca et al. 2014).

class teili.models.neuron_models.**ExpAdaptIF**(*num_inputs=1*)
Bases: *teili.models.builder.neuron_equation_builder.NeuronEquationBuilder*

This class provides you with all equations to simulate a voltage-based exponential, adaptive integrate and fire neuron.

class teili.models.neuron_models.**ExpAdaptLIF**(*num_inputs=1*)
Bases: *teili.models.builder.neuron_equation_builder.NeuronEquationBuilder*

This class provides you with all equations to simulate a voltage-based exponential, adaptive integrate and fire neuron.

class teili.models.neuron_models.**ExpLIF**(*num_inputs=1*)
Bases: *teili.models.builder.neuron_equation_builder.NeuronEquationBuilder*

This class provides you with all equations to simulate a voltage-based exponential leaky integrate and fire neuron.

class teili.models.neuron_models.**Izhikevich**(*num_inputs=1*)
Bases: *teili.models.builder.neuron_equation_builder.NeuronEquationBuilder*

This class provides you with all equations to simulate a voltage-based quadratic, adaptive integrate and fire neuron.

class teili.models.neuron_models.**LinearLIF**(*num_inputs=1*)
Bases: *teili.models.builder.neuron_equation_builder.NeuronEquationBuilder*

This class provides you with all equations to simulate a voltage-based exponential, adaptive integrate and fire neuron.

class teili.models.neuron_models.**OCTA_Neuron**(*num_inputs=2*)
Bases: *teili.models.builder.neuron_equation_builder.NeuronEquationBuilder*

Custom equations for the OCTA network.

octa_neuron [neuron_equation that comprises of all the components needed for octa.] In some synaptic connections not all features are used.

class teili.models.neuron_models.**QuantStochLIF**(*num_inputs=1*)
Bases: *teili.models.builder.neuron_equation_builder.NeuronEquationBuilder*

This class provides you with all equations to simulate an integrate and fire neuron with quantized stochastic decay as implemented in Wang et al. (2018)

teili.models.neuron_models.**main**(*path=None*)

teili.models.synapse_models module

This contains subclasses of SynapseEquationBuilder with predefined common parameters

class teili.models.synapse_models.**Alpha**
Bases: *teili.models.builder.synapse_equation_builder.SynapseEquationBuilder*

This class provides you with all equations to simulate synapses with double exponential dynamics.

class teili.models.synapse_models.**AlphaStdp**
Bases: *teili.models.builder.synapse_equation_builder.SynapseEquationBuilder*

This class provides you with all equations to simulate synapses with double exponential dynamics.

class teili.models.synapse_models.**BraderFusiSynapses**
Bases: *teili.models.builder.synapse_equation_builder.SynapseEquationBuilder*

This class provides you with all the equations to simulate a bistable Brader-Fusi synapse as published in Brader and Fusi 2007.

class teili.models.synapse_models.**DPIShunt**
Bases: *teili.models.builder.synapse_equation_builder.SynapseEquationBuilder*

This class provides you with all the equations to simulate a Differential Pair Integrator (DPI) synapse as published in Chicca et al. 2014.

class teili.models.synapse_models.**DPISyn**
Bases: *teili.models.builder.synapse_equation_builder.SynapseEquationBuilder*

This class provides you with all the equations to simulate a Differential Pair Integrator (DPI) synapse as published in Chicca et al. 2014.

class teili.models.synapse_models.**DPISyn_alpha**
Bases: *teili.models.builder.synapse_equation_builder.SynapseEquationBuilder*

This class provides you with all the equations to simulate a Differential Pair Integrator (DPI) synapse as published in Chicca et al. 2014.

class teili.models.synapse_models.**DPIadp**
Bases: *teili.models.builder.synapse_equation_builder.SynapseEquationBuilder*

class teili.models.synapse_models.**DPIstdgm**
Bases: *teili.models.builder.synapse_equation_builder.SynapseEquationBuilder*

class teili.models.synapse_models.**DPIstdp**
Bases: *teili.models.builder.synapse_equation_builder.SynapseEquationBuilder*

This class provides the well-known DPI synapse with Spike-Timing Dependent Plasticity mechanism.

class teili.models.synapse_models.**DoubleExponential**
Bases: *teili.models.builder.synapse_equation_builder.SynapseEquationBuilder*

This class provides you with all equations to simulate synapses with double exponential dynamics.

class teili.models.synapse_models.**Exponential**
Bases: *teili.models.builder.synapse_equation_builder.SynapseEquationBuilder*

This class provides you with all the equations to simulate an exponential decaying voltage-based synapse without learning.

class teili.models.synapse_models.**ExponentialStdp**
Bases: *teili.models.builder.synapse_equation_builder.SynapseEquationBuilder*

This class provides you with all the equations to simulate an exponential decaying voltage-based synapse without learning.

class teili.models.synapse_models.**QuantStochSyn**
Bases: *teili.models.builder.synapse_equation_builder.SynapseEquationBuilder*

This class provides you with all the equations to simulate a synapse with quantized stochastic decay as published by Wang et al. (2018).

class teili.models.synapse_models.**QuantStochSynStdp**
Bases: *teili.models.builder.synapse_equation_builder.SynapseEquationBuilder*

This class provides you with all the equations to simulate a synapse with stochastic decay with STDP as published by Wang et al. (2018)

class teili.models.synapse_models.**Resonant**
Bases: *teili.models.builder.synapse_equation_builder.SynapseEquationBuilder*

This class provides you with all equations to simulate synapses with double exponential dynamics.

class teili.models.synapse_models.**ResonantStdp**
Bases: *teili.models.builder.synapse_equation_builder.SynapseEquationBuilder*

This class provides you with all equations to simulate synapses with resonant function dynamics with STDP learning.

class teili.models.synapse_models.**ReversalSynV**
Bases: *teili.models.builder.synapse_equation_builder.SynapseEquationBuilder*

This class provides you with all the equations to simulate synapses with reversal potential.

class teili.models.synapse_models.**StdpSynV**
Bases: *teili.models.builder.synapse_equation_builder.SynapseEquationBuilder*

This class provides you with all the equations to simulate an exponential decaying voltage-based synapse with learning based on Spike-Timing Dependent Plasticity (STDP).

teili.models.synapse_models.**main**(*path=None*)

Module contents

teili.stimuli package

Submodules

teili.stimuli.testbench module

This class holds different pre-defined testbench stimuli.

The idea is to test certain aspects of you network with common stimuli.

Example

```
>>> import numpy as np
>>> from brian2 import us, ms
>>> from pyqtgraph.Qt import QtCore, QtGui
>>> import pyqtgraph as pg
>>> from teili.stimuli.testbench import OCTA_Testbench
>>> from teili.tools.plotter2d import Plotter2d
```

```
>>> app = QtGui.QApplication.instance()
>>> if app is None:
        app = QtGui.QApplication(sys.argv)
>>> else:
        print('QApplication instance already exists: %s' % str(app))
```

```
>>> testbench = OCTA_Testbench()
>>> testbench.rotating_bar(length=10, nrows=10, direction='ccw', ts_offset=3,
                           angle_step=10, noise_probability=0.2, repetitions=90,␣
↪debug=False)
```

In order to visualize it:

```
>>> event_monitor = Plotter2d.loaddvs(testbench.events)
>>> imv1 = event_monitor.plot3d_on_off(plot_dt=10*ms, filtersize=15*ms)
```

```
>>> win = pg.GraphicsWindow(title="DVS Spikes")
>>> gridlayout = QtGui.QGridLayout(win)
>>> gridlayout.addWidget(imv1, 1, 1)
>>> win.resize(1500, 1000)
>>> win.setLayout(gridlayout)
>>> win.show()
>>> win.setWindowTitle('DVS plot')
>>> imv1.play(10)
```

```
>>> app.exec_()
```

Todo:

- As soon as visualizer class is updated, change imports!

class teili.stimuli.testbench.**OCTA_Testbench**(*DVS_SHAPE=(240, 180)*)

Bases: object

This class holds all relevant stimuli to test modules provided with the Online Clustering of Temporal Activity (OCTA) framework.

angles

List of angles of orientation.

Type numpy.ndarray

DVS_SHAPE

Input shape of the simulated DVS/DAVIS vision sensor.

Type TYPE

end

End pixel location of the line.

Type TYPE

events

Attribute storing events of testbench stimulus.

Type TYPE

indices
Attribute storing neuron index of testbench stimulus.

Type TYPE

line
Stimulus of the testbench which is used to either generate an interactive plot to record stimulus with a DVS/DAVIS camera or coordinates are used to generate a SpikeGenerator.

Type TYPE

start
Start pixel location of the line.

Type TYPE

times
Attribute storing spike times of testbench stimulus.

Type list

aedat2events(*rec*, *camera='DVS128'*)
Wrapper function of the original aedat2numpy function in teili.tools.converter.

This function will save events for later usage and will directly return them if no SpikeGeneratorGroup is needed.

Parameters

- **rec** (*str*) – Path to stored .aedat file.
- **camera** (*str, optional*) – Can either be string ('DAVIS240') or int 240, which specifies the larger of the 2 pixel dimension to unravel the coordinates into indices.

Returns 4D numpy array with #events entries. Array is organized as x, y, ts, pol. See aedat2numpy for more details.

Return type events (np.ndarray)

ball(*rec_path*)
This function loads a simple recording of a ball moving in a small arena. The idea is to test the Online Clustering and Prediction module of OCTAPUS. The aim is to learn spatio-temporal features based on the ball's trajectory and learn to predict its movement.

Parameters **rec_path** (*str, required*) – Path to recording.

Returns A SpikeGenerator which has index (i) and spiketimes (t) as attributes

Return type SpikeGeneratorGroup (brian2.obj)

Raises **UserWarning** – If no filename is given but aedat reacording should be loaded

dda_round(*x*)
Simple round funcion.

Parameters **x** (*float*) – Value to be rounded.

Returns Ceiled value of x.

Return type (int)

infinity(*cAngle*)
Given an angle cAngle this function returns the current position on an infinity trajectory.

Parameters **cAngle** (*float*) – current angle in rad which determines position on infinity trajectory.

Returns Postion in x, y coordinates.

Return type position (tuple)

rotating_bar(*length=10*, *nrows=10*, *ncols=None*, *direction='ccw'*, *ts_offset=10*, *angle_step=10*, *artifical_stimulus=True*, *rec_path=None*, *save_path=None*, *noise_probability=None*, *repetitions=1*, *debug=False*)

This function returns a single SpikeGeneratorGroup (Brian object).

The purpose of this function is to provide a simple test stimulus. A bar is rotating in the center. The goal is to learn necessary spatio-temporal features of the moving bar and be able to make predictions about where the bar will move.

Parameters

- **length** (*int*) – Length of the bar in pixel.
- **nrows** (*int, optional*) – X-Axis size of the pixel array.
- **ncols** (*int, optional*) – Y-Axis size of the pixel array.
- **orientation** (*str*) – Orientation of the bar. Can either be 'vertical' or 'horizontal'.
- **ts_offset** (*int*) – time between two pixel location.
- **angle_step** (*int, optional*) – Angular velocity. Sets step width in np.arrange.
- **artifical_stimulus** (*bool, optional*) – Flag if stimulus should be created or loaded from aedat file.
- **rec_path** (*str, optional*) – Path/to/stored/location/of/recorded/stimulus.aedat.
- **save_path** (*str, optional*) – Path to store generated events.
- **noise_probability** (*float, optional*) – Probability of noise events between 0 and 1.
- **repetitions** (*int, optional*) – Number of revolutions of the rotating bar.
- **debug** (*bool, optional*) – Flag to print more detailed output of testbench.

Returns

Brian2 objects which holds the spike times as well as the respective neuron indices

Return type SpikeGenerator obj

Raises **UserWarning** – If no filename is given but aedat recording should be loaded

rotating_bar_infinity(*length=10*, *nrows=64*, *ncols=None*, *orthogonal=False*, *shift=32*, *ts_offset=10*, *artifical_stimulus=True*, *rec_path=None*, *return_events=False*)

This function will either load recorded DAVIS/DVS recordings or generate artificial events of a bar moving on an infinity trajectory with fixed orientation, i.e. no super-imposed rotation. In both cases, the events are provided to a SpikeGeneratorGroup which is returned.

Parameters

- **length** (*int, optional*) – Length of the bar in pixel.
- **nrows** (*int, optional*) – X-Axis size of the pixel array.
- **ncols** (*int, optional*) – Y-Axis size of the pixel array.
- **orthogonal** (*bool, optional*) – Flag which determines if bar is kept always orthogonal to trajectory, if it kept aligned with the trajectory or if it returns in a "chaotic" way.

- **shift** (*int, optional*) – Offset in x where the stimulus will start.
- **ts_offset** (*int, optional*) – Time in ms between consecutive pixels (stimulus velocity).
- **artifical_stimulus** (*bool, optional*) – Flag if stimulus should be created or loaded from aedat file.
- **rec_path** (*str, optional*) – Path/to/stored/location/of/recorded/stimulus.aedat.
- **return_events** (*bool, optional*) – Flag to return events instead of SpikeGenerator.

Returns A SpikeGenerator which has index (i) and spiketimes (t) as attributes. events (numpy.ndarray, optional): If return_events is set, events will be returned.

Return type SpikeGeneratorGroup (brian2.obj)

Raises **UserWarning** – If no filename is given but aedat recording should be loaded.

translating_bar_infinity(*length=10*, *nrows=64*, *ncols=None*, *orientation='vertical'*, *shift=32*, *ts_offset=10*, *artifical_stimulus=True*, *rec_path=None*, *return_events=False*)

This function will either load recorded DAVIS/DVS recordings or generate artificial events of a bar moving on an infinity trajectory with fixed orientation, i.e. no super-imposed rotation. In both cases, the events are provided to a SpikeGeneratorGroup which is returned.

Parameters

- **length** (*int, optional*) – length of the bar in pixel.
- **nrows** (*int, optional*) – X-Axis size of the pixel array.
- **ncols** (*int, optional*) – Y-Axis size of the pixel array.
- **orientation** (*str, optional*) – lag which determines if bar is orientated vertically or horizontally.
- **shift** (*int, optional*) – offset in x where the stimulus will start.
- **ts_offset** (*int, optional*) – Time in ms between consecutive pixels (stimulus velocity).
- **artifical_stimulus** (*bool, optional*) – Flag if stimulus should be created or loaded from aedat file.
- **rec_path** (*str, optional*) – Path/to/stored/location/of/recorded/stimulus.aedat.
- **return_events** (*bool, optional*) – Flag to return events instead of SpikeGenerator.

Returns A SpikeGenerator which has index (i) and spiketimes (t) as attributes. events (numpy.ndarray, optional): If return_events is set, events will be returned.

Return type SpikeGeneratorGroup (brian2.obj)

Raises **UserWarning** – If no filename is given but aedat recording should be loaded.

class teili.stimuli.testbench.**STDGM_Testbench**(*N=1*, *stimulus_length=1200*)

Bases: object

This class provides a stimulus to test your spike-timing dependent gain modulation algorithm.

stimuli(*isi*)

Stimulus gneration for STDGM protocols.

This function returns two brian2 objects. Both are Spikegeneratorgroups which hold a single index each and varying spike times. The protocol follows homoeostasis, weak LTP, weak LTD, strong LTP, strong LTD, homoeostasis.

Parameters **isi** (*int, optional*) – Interspike Interval. How many spikes per stimulus phase.

Returns

Brian2 objects which hold the spiketimes and the respective neuron indices.

Return type SpikeGeneratorGroup (brian2.obj

class teili.stimuli.testbench.**STDP_Testbench**(*N=1*, *stimulus_length=1200*)

Bases: object

This class provides a stimulus to test your spike-timing dependent plasticity algorithm.

N

Size of the pre and post neuronal population.

Type int

stimulus_length

Length of stimuli in ms.

Type int

stimuli(*isi=10*)

Stimulus gneration for STDP protocols.

This function returns two brian2 objects. Both are Spikegeneratorgroups which hold a single index each and varying spike times. The protocol follows homoeostasis, weak LTP, weak LTD, strong LTP, strong LTD, homoeostasis.

Parameters **isi** (*int, optional*) – Interspike Interval. How many spikes per stimulus phase.

Returns

Brian2 objects which hold the spiketimes and the respective neuron indices.

Return type SpikeGeneratorGroup (brian2.obj

class teili.stimuli.testbench.**SequenceTestbench**(*n_channels*, *n_items*, *cycle_length*, *noise_probability=None*, *rate=None*)

Bases: object

This class provides a simple poisson encoded sequence testbench. This class returns neuron indices and spike times, which can used to instantiate a SpikeGeneratorGroup in brian2.

add_noise()

This function adds noise spike given the noise_probability.

create_poisson_items()

This function creates the Poisson distributed items with the specificed rate.

stimuli()

This function creates the stimuli and returns neuron indices and spike times.

class teili.stimuli.testbench.**WTA_Testbench**

Bases: object

Collection of functions to test the computational properties of the WTA building_block.

indices
Array with neuron indices.

Type numpy.ndarray

noise_input
PoissonGroup which provides noise events.

Type brian2.PoissonGroup

times
Array with neuron spike times.

Type numpy.ndarray

background_noise(*num_neurons=10*, *rate=10*)
Provides background noise as Poisson spike trains

Parameters

- **num_neurons** (*int, optional*) – 1D size of WTA population.
- **rate** (*int, optional*) – Spike frequency f Poisson noise process.

stimuli(*num_neurons=16*, *dimensions=2*, *start_time=10*, *end_time=500*, *isi=2*)
This function provides simple test stimuli to test the selection mechanism of a WTA population.

Parameters

- **num_neurons** (*int, optional*) – 1D size of WTA population.
- **dimensions** (*int, optional*) – Dimension of WTA. Can either be 1 or 2
- **start_time** (*int, optional*) – Start time when stimulus should start.
- **end_time** (*int, optional*) – End time when stimulus should stop.
- **isi** (*int, optional*) – Inter-spike between spike times.

Raises **NotImplementedError** – If dimension is not 1 or 2 this error is raised

Module contents

teili.tools package

Submodules

teili.tools.converter module

Function for external interfaces such as an event-based camera, e.g. DVS.

Functions in this module convert data from or to brian2 compatible formats. In particular, there are functions to convert data coming from DVS cameras.

teili.tools.converter.**aedat2numpy**(*datafile*, *length=0*, *version='V2'*, *debug=0*, *camera='DVS128'*, *unit='ms'*)
Loads AER data file and parses these properties of AE events.

Properties:

- timestamps (in us).
- x,y-position [0..127]x[0..127] for DVS128 [0..239]x[0..127] for DAVIS240.

- polarity (0/1).

Parameters

- **datafile** (*str, optional*) – Aedat recording as provided by jAER or cAER.
- **length** (*int, optional*) – how many bytes(B) should be read; default 0=whole file.
- **version** (*str, optional*) – which file format version is used: - "dat" = V1 (old) - "aedat" jAER AEDAT 2.0 = V2 - "aedat" cAER AEDAT 3.1 = V3. - "aedat" DV AEDAT 4.0 = V4
- **debug** (*int, optional*) – Flag to provide more detailed report. 0 = silent, 1 (default) = print summary. >=2 = print all debug.
- **camera** (*str, optional*) – Type of event-based camera.
- **unit** – output unit of timestamps specified as a string: - 'ms' (default), 'us' or 'sec'.

Returns (xpos, ypos, ts, pol) 2D numpy array containing data of all events.

Return type numpy.ndarray

Raises **ValueError** – Indicates that a camera was specified which is not supported or the AEDAT file version is not supported.

teili.tools.converter.**delete_doublets**(*spiketimes*, *indices*, *verbose=False*)

Removes spikes that happen at the same time and at the same index. This happens when you donwnsample, but Brian2 cannot cope with more that 1 spike per ts. :param spiketimes: numpy array of spike times :param indices: numpy array of indices :return: same as input but with removed doublets

teili.tools.converter.**dvs2ind**(*events=None*, *event_directory=None*, *resolution='DAVIS240'*, *scale=True*)

Function which converts events extracted from an aedat file using aedat2numpy into 1D vectors of neuron indices and timestamps.

Function only returns index and timestamp list for existing types (e.g. On & Off events).

Parameters

- **Events** (*None, optional*) – 4D numpy.ndarray which contains pixel location (x,y), timestamps and polarity ((4,#events)).
- **event_directory** (*None, optional*) – Path to stored events.
- **resolution** (*str/int, optional*) – Resolution of the camera.
- **scale** (*bool, optional*) – Flag to rescale the timestamps from microseconds to milliseconds.

Returns Unique indices which maps the pixel location of the camera to the 1D neuron indices of ON events. ts_on (1d numpy.array): Unique timestamps of active indices of ON events. indices_off (1d numpy.array): Unique indices which maps the pixel location of the camera to the 1D neuron indices of OFF events. ts_off (1d numpy.array): Unique timestamps of active indices of OFF events.

Return type indices_on (1d numpy.array)

teili.tools.converter.**dvs_csv2numpy**(*datafile='tmp/aerout.csv'*, *debug=False*)

Loads AER csv logfile and parses these properties of AE events

Properties:

- timestamps (in us).

- x,y-position [0..127].
- polarity (0/1).

Parameters

- **datafile** (*str, optional*) – path to the csv file to read.
- **debug** (*bool, optional*) – Flag to print more details about conversion.

Returns (ts, xpos, ypos, pol) 4D numpy array containing data of all events.

Return type numpy.ndarray

teili.tools.converter.**read_events**(*file_read*, *x_dim*, *y_dim*)

A simple function that reads events from cAER tcp.

Parameters

- **file_read** (*TYPE*) – Description
- **xdim** (*TYPE*) – Description
- **ydim** (*TYPE*) – Description

Returns Description

Return type TYPE

teili.tools.converter.**skip_header**(*file_read*)

skip header.

Parameters **file_read** (*TYPE*) – File

teili.tools.cpptools module

Collection of tools to add some features to cpp standalone mode

teili.tools.cpptools.**activate_standalone**(*directory='Brian2Network_standalone'*, *build_on_run=False*)

Enables cpp standalone mode

Parameters

- **directory** (*str, optional*) – Standalone directory containing all compiled files
- **build_on_run** (*bool, optional*) – Flag to (re-)build network before simulating

teili.tools.cpptools.**build_cpp_and_replace**(*standalone_params*, *standalone_dir='/home/docs/Brian2Standalone'*, *clean=True*, *do_compile=True*, *verbose=True*)

Builds cpp standalone network and replaces variables/parameters with standalone_params This does string replacement in the generated c++ code.

Parameters

- **standalone_params** (*dict, required*) – Dictionary containing all parameters which can be changed after building the network
- **standalone_dir** (*str, optional*) – Directory containing output generated by network
- **clean** (*bool, optional*) – Flag to clean build network

- **do_compile** (*bool, optional*) – Flag to compile network

teili.tools.cpptools.**collect_standalone_params**(*params={}*, **building_blocks*)
This just collect the parameters of all buildingblocks and adds additional parameters (not from buildingblocks)

Parameters

- **params** (*OrderedDict, optional*) – Dictionary with parameters. Needs to be ordered.
- ***buildingBlocks** – The network building block to assign standalone parameters to

Returns standalone parameters

Return type dict

teili.tools.cpptools.**deactivate_standalone**()
Disables cpp standalone mode

teili.tools.cpptools.**params2run_args**(*standalone_params*)
Add standalone parameter to run arguments

Parameters **standalone_params** (*dict*) – Dictionary containing standalone parameters to be added to run arguments

Returns run arguments

Return type list

teili.tools.cpptools.**print_dict**(*pdict*)
Wrapper function to print dictionary

Parameters **pdict** (*dictionary*) – Dictionary to be printed

teili.tools.cpptools.**replace_variables_in_cpp_code**(*replace_vars*, *replace_file_location*, *verbose=True*)
Replaces a list of variables in CPP code for standalone code generation with changeable parameters and it adds duration as a changeable parameter (it is always the first argument)

Parameters

- **replace_vars** (*list, str*) – List of strings, variables that are replaced
- **replace_file_location** (*str*) – Location of the file in which the variables are replaced

teili.tools.cpptools.**run_standalone**(*standalone_params*)
Runnung standalone networks

Parameters **standalone_params** (*dict*) – Dictionary of standalone parameters

teili.tools.distance module

Functions to compute distance (e.g. in 2D).

The suffix "_cpp" avoids variables being string-replaced by brian2 if the same name is used in the network.

teili.tools.distance.**circle_dist1d**(*x*, *y*, *N*)

teili.tools.indexing module

Collections of functions which convert indices to x, y coordinates and vice versa.

The suffix "_cpp" avoids variables being string-replaced by brian2 if the same name is used in the network.

Todo:

- **TODO: make *ind2events* consistent with the other functions** in this module! (or maybe it is not at the right place here).

---

teili.tools.math_functions module

teili.tools.misc module

A collection of helpful miscellaneous functions when working with brian2

`teili.tools.misc.`**`neuron_group_from_spikes`**(*num_inputs*, *simulation_dt*, *duration*, *poisson_group=None*, *spike_indices=None*, *spike_times=None*)

Converts spike activity in a neuron poisson_group with the same activity.

Parameters

- **num_inputs** (*int*) – Number of input channels from source.
- **simulation_dt** (*brian2.unit.ms*) – Time step of simulation.
- **duration** (*int*) – Duration of simulation in brian2.ms.
- **poisson_group** (*brian2.poissonGroup*) – Poisson poisson_group that is passed instead of spike times and indices.
- **spike_indices** (*numpy.array*) – Indices of the original source.
- **spike_times** (*numpy.array*) – Time stamps with unit of original spikes in ms.

Returns Neuron poisson_group with mimicked activity.

Return type neu_group (brian2 object)

`teili.tools.misc.`**`print_states`**(*briangroup*)

Wrapper function to print states of a brian2 groups such as NeuronGroup or Synapses

Parameters **briangroup** (*brian2.group*) – Brain object/group which states/statevariables should be printed

`teili.tools.misc.`**`spikemon2firing_rate`**(*spikemon*, *start_time=0.0 * second*, *end_time='max'*)

Calculates the instantaneous firing rate within a window of interest from a SpikeMonitor

Parameters

- **spikemon** (*brian2.SpikeMonitor*) – Brian2 SpikeMoitor object
- **start_time** (*brain2.unit.ms, optional*) – Starting point for window to calculate the firing rate. Must be provided as desired time in ms, e.g. 5 * ms
- **end_time** (*str, optional*) – End point for window to calculate the firing rate. Must be provided as desired time in ms, e.g. 5 * ms

Returns Firing rate in Hz

Return type int

teili.tools.plotter2d module

Created on 28 Dec 2017

@author: Alpha Renner

This class is a 2d plotter and provides functionality for analysis of 2d neuron fields To be extended!

`teili.tools.plotter2d.`**`CM_JET`**
Description

Type TYPE

`teili.tools.plotter2d.`**`CM_ONOFF`**
Description

Type TYPE

class `teili.tools.plotter2d.`**`DVSmonitor`**(*xi*, *yi*, *t*, *pol*, *unit=None*)
Bases: `object`

Summary

`pol`
Description

Type TYPE

`t`
Description

Type TYPE

`xi`
Description

Type TYPE

`yi`
Description

Type TYPE

class `teili.tools.plotter2d.`**`Plotter2d`**(*monitor*, *dims*, *plotrange=None*)
Bases: `object`

Plotter2d is a class that contains a number of functions to create 2d plots over time, in particular events/spikes that are arranged in 2d such as DVS camera recordings or 2d neural fields. The class offers filtering, plotting and generation of gifs.

Data is passed into the plotter as a monitor (either from brian2 or using the DVSmonitor class)

`dims`
the dimensions of the 2d data (number of rows and columns)

Type tuple

`monitor`
A monitor to sparsely store event data (e.g. from brian2), it has a t (timestamps), xi (x event coordinates) and yi (y coordinates) property or an i property (flat coordinates that are reshaped to 2d)

Type TYPE

plotrange
Masks the monitor outside of the given range (in units of t)

Type tuple

shape
3d shape of the data

Type tuple

rows
number of rows (dims[0])

Type int

cols
number of columns (dims[1])

Type int

mask
mask that masks out part of the data

Type array

calculate_pop_vector_trajectory(*dt=50.0 * msecond*, *plot=False*, *frames_timestamps=None*)
Calculates the trajectory of the center of mass over time.

generate_movie(*filename*, *scale=None*, *speed=1*, *plotfunction='plot3d'*, *plot_dt=10.0 * msecond*, *tempfolder='/home/docs'*, *ffmpegoptions=''*, ***plotkwargs*)
This exports a movie or gif from an imageview Existing outputfiles will be overwritten This needs ffmpeg wich is installed on most linux distributions and also available for windows and mac Have a loo here: https://ffmpeg.org/

Parameters

- **filename** (*str*) – The filename in which to store the generated movie. You can choose a format that can be generated with ffmpeg like '.gif', '.mpg', '.mp4',...
- **scale** (*str, optional*) – give pixel size as string e.g. '100x100'
- **speed** (*num, optional*) – if the video should run faster, specify a multiplier
- **plot_dt** (*given in brian2 time units*) – is passed to the plotfunction and determines the fps
- **tempfolder** (*str, optional*) – the directory in which the temporary folder to store files created in the process. The temporary folder will be deleted afterwards. By default it will be created in your home directory
- **plotfunction** (*str or function, optional*) – the function that should be used to create the gif. it has to be a function that returns a pyqtgraph imageview (or at least something similar that can export single images) like the methods of this class (plot3d, ...). For the methods, you can also pass a string to identify the plotfunction. The plotfunction has to take plot_dt as an argument
- **ffmepgoptions** (*str, optional*) –
- **kwargs** – all other keyword agruments will be passed to the plotfunction
- **usage** (*Example*) –
- **plotter2dobject.generate_gif** (*'~/gifname.gif', plotfunction = 'plot3d_on_off', filtersize=100 * ms, plot_dt=50 * ms*) –

get_dense3d(*dt*)
Transforms the sparse spike time representation in a dense representation, where every spike is given as a 1 in a 3d matrix (time + 2 spatial dimensions) The data is binned using dt. If there is more than one spike in a bin, the bin will not have the value 1, but the number of spikes.

Parameters **dt** (*TYPE*) – Description

Returns

dense array of the data. E.g. if we have a single spike in the neuron at location (3, 5) at timestamp 10, in the dense array, all locations have a value of 0 at all timesteps apart from (10, 3, 5), where the value is 1.

Return type array

get_dense_ifr(*dt=50.0 * msecond*, *plot=False*, *frames_timestamps=None*)
calculates a vector of instantaneous frequencies for every timestep dt. IFRs on timesteps without a spike are interpolated between the last two spikes :return: matrix of IFRs for every neuron and every timestep

get_filtered(*dt*, *filtersize*)
applies a rectangular filter (convolution) of length filtersize over time (dimension 0). It returns a 3d matrix with the firing rate. Spiketimes will be binned with a step size of dt that means that the filtersize should always be a int multiple of dt

Parameters

- **dt** (*brian2.Quantity*) – the time step with which the spike times are binned
- **filtersize** (*brian2.Quantity*) – length of the filter (in brian2 time units)

Returns Description

Return type TYPE

get_sparse3d(*dt*, *align_to_min_t=True*)
Using the package sparse (based of scipy sparse, but for 3d), the spiketimes are converted into a sparse matrix. This step is basically just for easy conversion into a dense matrix later, as you cannot do so many computations with the sparse representation. sparse documentation can be found here: http://sparse.pydata.org/en/latest/

Parameters **dt** (*float*) – the t dimension in the sparse representation is given in timesteps, so t is divided by dt

Returns Sparse representation of the data used to create the dense one efficiently.

Return type sparse.COO

property i
flattened indices (in 1d)

ifr_histogram(*filename=None*, *num_bins=50*)
histogram of instantaneous frequencies

Parameters

- **filename** (*str*) – filename to save the histogram
- **num_bins** (*int, optional*) – number of bins of the histogram

classmethod loaddvs(*eventsfile*, *dims=None*)
loads a dvs numpy (events file) from aedat2numpy and returns a SpikeMonitor2d object, you can also directly pass an events array

usage: spikemonObject = SpikeMonitor2d.loadz(myfilename) #e.g. spikemonObject.plot3d()

Parameters **eventsfile** (*str*) – filename from where to load the data of the plotter object

classmethod loadz(*filename*)
loads a file that has previously been saved with savez and returns a SpikeMonitor2d object

usage: spikemonObject = SpikeMonitor2d.loadz(myfilename) #e.g. spikemonObject.plot3d()

Parameters **filename** (*str*) – filename from where to load the data of the plotter object

Returns Description

Return type TYPE

plot3d(*plot_dt=100.0 * usecond*, *filtersize=10.0 * msecond*, *colormap=pyqtgraph.colormap.ColorMap*, *levels=None*, *flipy=False*)

Parameters

- **plot_dt** (*brian2.Quantity, optional*) – timestep in which events are binned for plotting
- **filtersize** (*brian2.Quantity, optional*) – filtersize of rectangular filter
- **colormap** (*pyqtgraph.colormap.ColorMap, optional*) – colormap for on off plot
- **levels** (*tuple*) – (min, max); the white and black level values to use (passed to pyqtgraph)

Returns ImageView object for usage in a larger pyqtgraph plot

Return type pyqtgraph.ImageView

plot3d_on_off(*plot_dt=100.0 * usecond*, *filtersize=10.0 * msecond*, *colormap=pyqtgraph.colormap.ColorMap*, *flipy=False*)

Parameters

- **plot_dt** (*brian2.Quantity, optional*) – timestep in which events are binned for plotting
- **filtersize** (*brian2.Quantity, optional*) – filtersize of rectangular filter
- **colormap** (*pyqtgraph.colormap.ColorMap, optional*) – colormap for on off plot

Returns ImageView object for usage in a larger pyqtgraph plot

Return type pyqtgraph.ImageView

plot_panes(*num_panes=None*, *timestep=None*, *filtersize=50. * msecond*, *num_rows=2*, *plotfunction='plot3d'*, *filename=None*, *colormap=<matplotlib.colors.LinearSegmentedColormap object>*, ***plotkwargs*)
plots the 3d data as time slices (2d images at several timepoints) :param num_panes: number of panes to plot :type num_panes: int, optional :param timestep: timestep between panes :type timestep: TYPE, optional :param filtersize: filtersize at which the spikes are filtered to generate the images :type filtersize: TYPE, optional :param num_rows: number of rows of the pane plot :type num_rows: int, optional :param filename: location where to save the plot :type filename: str, optional

Returns Description

Return type TYPE

property plotlength
number of timesteps

property plotrange

plotshape(*dt*)
3d shape of the data (num_timestamps, num_rows, num_cols)

Parameters **dt** (*float*) – timestep length

Returns (num_timestamps, num_rows, num_cols)

Return type tuple

property pol
polarity of DVS spikes

rate_histogram(*filename=None*, *filtersize=50.0 * msecond*, *plot_dt=10.0 * msecond*, *num_bins=50*)
plots a histogram of rates

Parameters

- **filename** (*str*) – filename to save the histogram
- **filtersize** (*brian2.Quantity, optional*) – filtersize of the retangular filter to calculate the rate
- **plot_dt** (*brian2.Quantity, optional*) – binsize in which the data is binned
- **num_bins** (*int, optional*) – number of bins of the histogram

savecsv(*filename*)
export data as csv not tested

Parameters **filename** (*TYPE*) – Description

savez(*filename*)
saves the object in a sparse way. only i,t, rows and cols are saved to an npz

Parameters **filename** (*str*) – filename under which to save the data of the plotter object

set_range(*plotrange=None*)
set a range with unit that is applied for all computations with this monitor

Parameters **plotrange** (*tuple*) – (from, to))

property t
timestamps of events

property t_
unitless t in ms

property xi
row coordinates of events

property yi
columns coordinates of events

teili.tools.plotter2d.**create_panes**(*video*, *num_rows*, *slice_indices=None*, *colormap=<matplotlib.colors.LinearSegmentedColormap object>*)

Parameters

- **video** – the 3d matrix that should be plotted
- **slice_indices** – the indices of the plotted slices in the first dim of the 3d matrix
- **num_rows** – the number of rows for the pane plot

Returns the GraphicsWindow object

teili.tools.plotter2d.**export_panes**(*gw_paneplot*, *filename*)
generates a figure file from a GraphicsWindow object :param gw_paneplot: :param filename: :return:

teili.tools.plotter2d.**interpolate_isi**(*ind*, *t=None*, *i=None*, *densetimes=None*)
interplolate interspike intervals so that there is a value at all timesteps and not just at the spike times

teili.tools.plotter2d.**visualize_3d**(*video*)

teili.tools.plotting module

Summary

teili.tools.plotting.**colors**
Description

Type TYPE

teili.tools.plotting.**labelStyle**
Description

Type dict

teili.tools.plotting.**plot_spikemon**(*start_time*, *end_time*, *monitor*, *num_neurons*, *ylab='ind'*)
Summary

Parameters

- **start_time** (*TYPE*) – Time from which spikes should be visualized
- **end_time** (*TYPE*) – Time until which spikes should be visualized
- **monitor** (*brian2.monitoritor*) – Monitor which serve as basis for plotting
- **num_neurons** (*int*) – Number of neurons to visualize
- **ylab** (*str, optional*) – Description

teili.tools.plotting.**plot_spikemon_qt**(*monitor*, *start_time=None*, *end_time=None*, *num_neurons=16*, *window=None*, *unit=None*)
Generic plotting function to plot spikemonitors using pyqtgraph

Parameters

- **start_time** (*int, optional*) – Time from which spikes should be visualized
- **end_time** (*int, optional*) – Time until which spikes should be visualized
- **monitor** (*brian2.obj*) – Monitor which serve as basis for plotting
- **num_neurons** (*int*) – Number of neurons to be plotted
- **window** (*TYPE*) – PyQtGraph window to which the plot should be added

Raises **UserWarning** – Description

teili.tools.plotting.**plot_statemon**(*start_time*, *end_time*, *monitor*, *neuron_id*, *variable='Vm'*, *unit=mvolt*, *name=''*)
Summary

Parameters

- **start_time** (*int, optional*) – Time from which spikes should be visualized
- **end_time** (*int, optional*) – Time until which spikes should be visualized

- **monitor** (*brian2.obj*) – Monitor which serve as basis for plotting
- **neuron_id** (*int*) – ID of neuron to be visualized
- **variable** (*str*) – State variable to visualize
- **unit** (*brian2.unit, optional*) – Unit of state variable
- **name** (*str, optional*) – Description

teili.tools.plotting.**plot_statemon_qt**(*start_time=None*, *end_time=None*, *monitor=None*, *neuron_id=True*, *variable='Imem'*, *unit=pamp*, *window=None*, *name=''*)

Generic plotting function to plot statemonitors using pyqtgraph

Parameters

- **start_time** (*int, optional*) – Time from which spikes should be visualized
- **end_time** (*int, optional*) – Time until which spikes should be visualized
- **monitor** (*brian2.obj*) – Monitor which serve as basis for plotting
- **neuron_id** (*int*) – ID of neuron to be visualized
- **variable** (*str*) – State variable to visualize
- **unit** (*brian2.unit, optional*) – Unit of state variable
- **window** (*pyqtgraph.window*) – PyQtGraph window to which the plot should be added
- **name** (*str, optional*) – Name of window

Raises **UserWarning** – Description

teili.tools.plotting.**plot_weights_group2wta**(*name*, *n_wta2d_neurons*, *syn_g_wta*, *n_col*)

Summary

Parameters

- **name** (*str*) – Name of the plot to be saved
- **n_wta2d_neurons** (*int*) – Number of 2d WTA population
- **syn_g_wta** (*brian2.synapses*) – Synapse group which weights should be plotted
- **n_col** (*int*) – Number of column to visualize

teili.tools.plotting.**plot_weights_wta2group**(*name*, *n_wta2d_neurons*, *syn_g_wta*, *n_col*)

Summary

Parameters

- **name** (*str*) – Name of the plot to be saved
- **n_wta2d_neurons** (*TYPE*) – Number of 2d WTA population
- **syn_g_wta** (*TYPE*) – Synapse group which weights should be plotted
- **n_col** (*TYPE*) – Number of column to visualize

teili.tools.sorting module

To understand the structure of in the rasterplots but also in the learned weight matrices, we need to sort the weight matrices according to some similarity measure, such as euclidean and jaccard distance. However, the sorting algorithm is completely agnostic to the similarity measure. It connects each node with maximum two edges and constructs a directed graph. This is similar to the travelling salesman problem.

Example

In order to use this class you need to initialize it either without a filename:

```
>>> from teili.tools.sorting import SortMatrix
>>> import numpy as np
>>> matrix = np.random.randint((49, 49))
>>> obj = SortMatrix(nrows=49, matrix=matrix)
>>> print(obj.matrix)
>>> print(obj.permutation)
>>> print(ob.sorted_matrix)
```

or instead of using a matrix you can also specify a path to a stored matrix:

```
>>> filename = '/path/to/your/matrix.npy'
>>> obj = SortMatrix(nrows=49, filename=filename)
```

class teili.tools.sorting.**SortMatrix**(*nrows*, *ncols=None*, *filename=None*, *matrix=None*, *axis=0*, *target_indices=None*, *rec_matrix=False*, *similarity_metric='euclidean'*)

Bases: object

Class which can sort your matrix based on similarity

filename
path/to/matrix/name.npy.

Type str, optional

matrix
matrix as provided by load_matrix.

Type ndarray, optional

ncols
number of columns of the 2d array.

Type int, optional

nrows
number of rows of the 2d array.

Type int, required

permutation
List of indices which are more similar to each other in (euclidean, jaccard) distance.

Type list

similarity_matrix
Matrix containing similarities.

Type ndarray, optional

sorted_matrix
Sorted matrix according to permutation.

Type TYPE

recurrent_matrix
Indicates whether it is a recurrent matrix or not.

Type boolean, optional

similarity_metric
Indicates metric for calculating similarity.

Type str, optional

max_val
Maximum value of the provided matrix

Type float

compute_distance(*x*, *y*, *similarity_metric*, *threshold=0.8*)
This function returns the distance of any to vectors x and y

Parameters

- **x** (*ndarray, required*) – 1d vector.
- **y** (*ndarray, required*) – 1d vector.
- **similarity_metric** (*str, required*) – Metric used to measure similarity

Returns Element-wise distance of two 1d vectors.

Return type ndarray

get_permutation(*axis=0*)
To sort a given matrix according to its similarity we need to construct permutation indices, which are used to sort the matrix. First we find the most similar entry in the similarity matrix. This function allows each node in the similarity graph to be only used twice, i.e. each node has maximally two edges connected to it. The vector 'degree' keeps track of this. To prevent a loop closure in the similarity graph a proxy vector called 'partner' is used to set the distance between the two ends of the similarity graph to infinity.

Parameters **axis** (*int, optional*) – Axis along which similarity should be computed.

Returns Vector of permuted indices.

Return type list

get_similarity_matrix(*axis=0*)
This function computes a similarity matrix of a given matrix.

Parameters **axis** (*int, optional*) – Axis along which similarity should be computed.

Returns Matrix containing similarities.

Return type ndarray

load_matrix()
Load matrix from .npy file

Returns loaded matrix from file.

Return type ndarray

sort_matrix()
This function returns the sorted matrix given the permutation indices.

Returns Sorted matrix according to similarity.

Return type ndarray

teili.tools.stimulus_generators module

The idea is to generate inputs based on a function. This avoids having to read large datafiles and makes generation of input easier.

How to use them: See example below.

class teili.tools.stimulus_generators.**StimulusSpikeGenerator**(**args*, ***kw*)

Bases: *teili.core.groups.TeiliGroup*, brian2.input.poissongroup.PoissonGroup

idea: add a run_regularly with dt, that dynamically changes the rates of a PoissonGroup or the input current of an IF neuron

The class is the neuron or the generator and can be added to a network

You can add equations that specify the trajectory of any pattern-variable. The two variables that are given by default are trajectory_x and trajectory_y But you can always specify the trajectory of any other variable in the trajectory_eq

trajectory_eq can either use a TimedArray that contains a prerecorded trajectory or you can also set an equation that describes the trajectory

The first 2 args of any pattern function must always be the x/y coordinates

Any user provided distance function has to have the brian2 decorators and take the following arguments: i, j, nrows, ncols (i and j are 1d indices in the 2d array)

please have a look at the example of a moving gaussian below (if __name__ == '__main__':)

plot_rates()

teili.tools.synaptic_kernel module

This module provides functions that can be used for synaptic connectivity kernels (generate weight matrices). In order to also use them with C++ code generation, all functions that are added here need to have a cpp implementation given by the @implementation decorator.

TODO: It would be good, if one could easily use different distance functions like on a ring or torus (1d/2d periodic boundary conditions). For numpy, this would be easy to implement by just using the respective function (from tools.distance) and add a selector as a parameter. For cpp, we would have to make sure that the functions used are known.

Module contents

13.1.2 Module contents

CHAPTER

FOURTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y