Neuronaldynamics Exercises Documentation

Release 0.3.3.dev0+gc722a63.d20170428

Wulfram Gerstner

Apr 28, 2017

Contents

This documentation is automatically generated documentation from the corresponding code repository hosted at Github. The repository contains python exercises accompanying the book Neuronal Dynamics by Wulfram Gerstner, Werner M. Kistler, Richard Naud and Liam Paninski.

CHAPTER 1

Contents

Introduction

This repository contains python exercises accompanying the book Neuronal Dynamics by Wulfram Gerstner, Werner M. Kistler, Richard Naud and Liam Paninski. References to relevant chapters will be added in the Teaching Materials section of the book homepage.

Quickstart

See *the setup instructions* for details on how to install the python classes needed for the exercises.

Every *exercise* comes with instructions and a demo function to get started. We recommend to create one jupyter notebook per exercise.

Requirements

The following requirements should be met:

- Either Python 2.7 or 3.4
- Brian2 Simulator 2.0b4
- Numpy
- Matplotlib
- Scipy (only required in some exercises)

If you are not using anaconda/miniconda, you can install all requirements by running:

```
pip install -r requirements.txt
```

Disclaimer

- You can download, use and modify the software we provide here. It has been tested but it can still contain errors.
- The content of this site can change at any moment. We may change, add or remove code/exercises without notification.

Bug reports

Did you find a bug? Open an issue on github . Thank you!

Exercises

Setting up Python and Brian

To solve the exercises you need to install Python, Brian2 and the neurodynex package. The installation procedure we described here focuses on the tools we use in the classroom sessions at EPFL. For that reason we additionally set up a **conda environment** (which we call bmnn below) and install Jupyter .

Using miniconda (recommended)

We offer anaconda packages for the most recent releases, which is the easiest way of running the exercises.

Head over to the miniconda download page and install **miniconda** (for Python 2.7 preferably).

Now execute the following commands to **install** the exercise package as well as **Brian2** and some dependencies. Note: we create a conda environment called 'bmnn'. You may want to change that name. In the last command we install Jupyter , a handy tool to create solution documents.

```
>> conda create --name bmnn python=2.7
>> source activate bmnn
>> conda install -c brian-team -c epfl-lcn neurodynex
>> conda install jupyter
```

If you need to **update** the exercise package, call:

```
>> source activate bmnn
>> conda update -c brian-team -c epfl-lcn neurodynex
```

You now have the tools you need to solve the python exercises. To get started, open a terminal, move to the folder where you want your code being stored and start a Jupyter notebook:

```
>> cd your_folder
>> source activate bmnn
>> jupyter notebook
```

We recommend you to create one notebook per exercise.

Note: **Trouble shooting:** You may get errors like 'No module named 'neurodynex'. This is the case when your jupyter notebook does not see the packages you've just installed. As a solution, try to re-install jupyter **within** the environment: .. code-block:

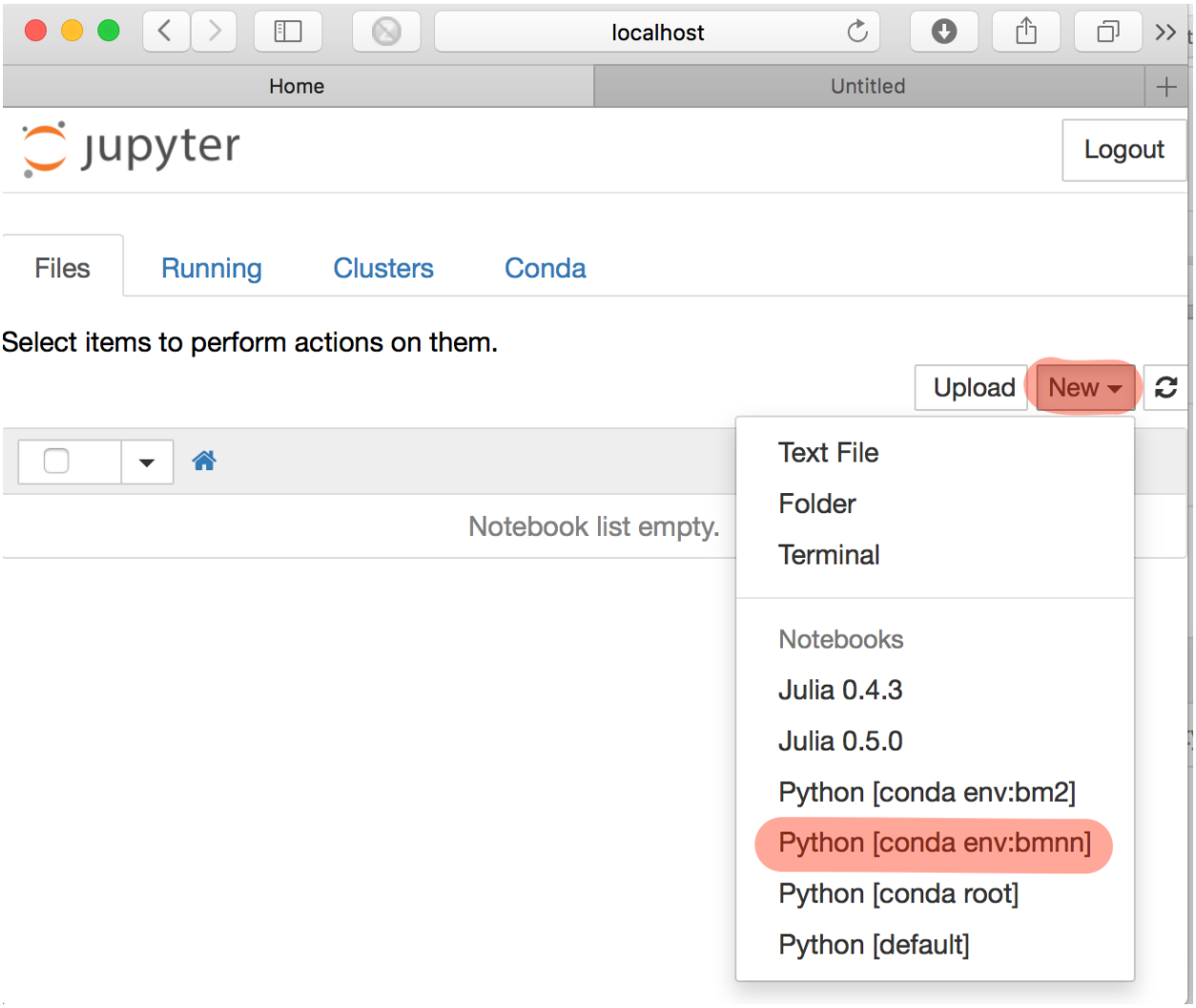
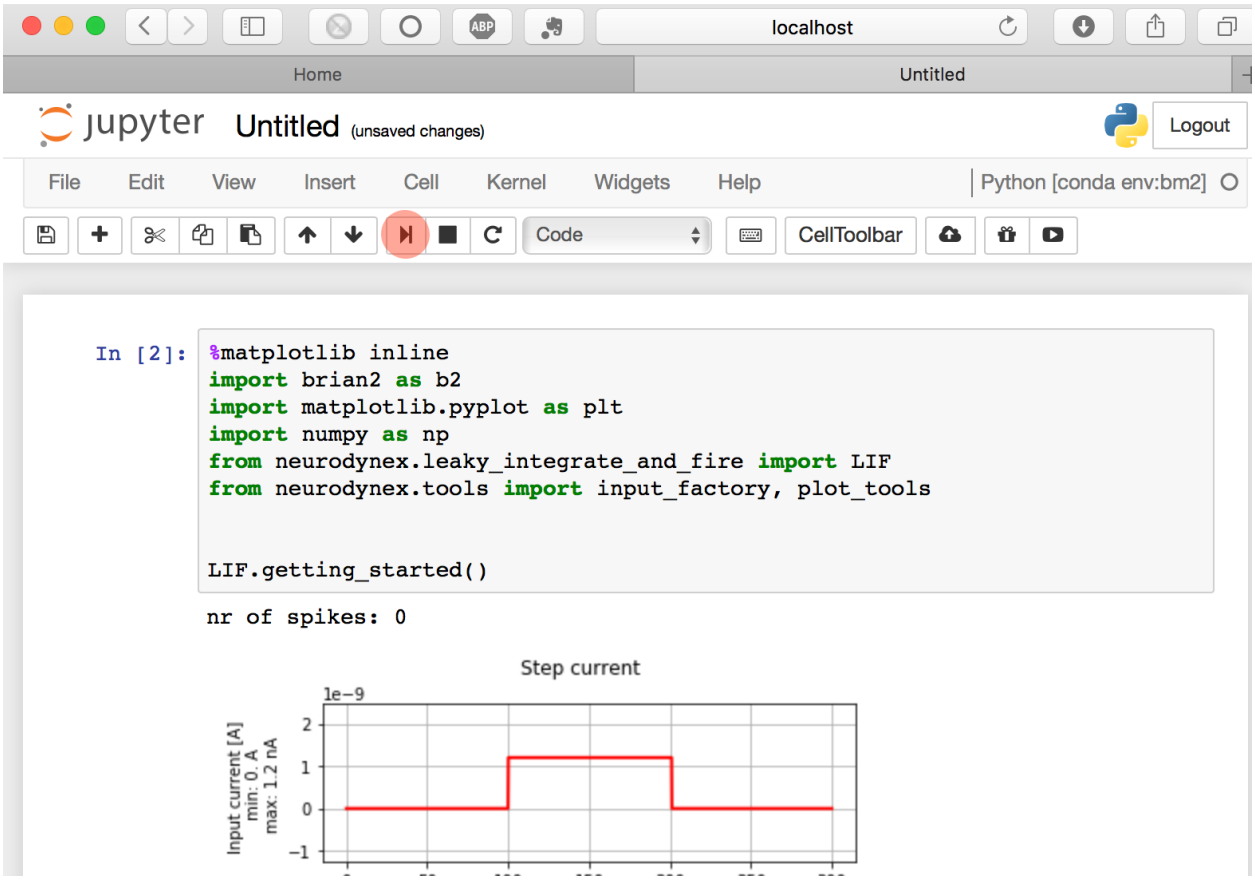


Fig. 1.1: Starting Jupyter will open your browser. Select NEW, Python2 to get a new notebook page. Depending on what else you have installed on your computer, you may have to specify the kernel. In the case shown here, it's the Python-bmnn installation.

Once you've create a new notebook, copy-paste the code of exercise one into the notebook and run it. Note that the first time you do this, the execution may take a little longer and, in some cases, you may see compilation warnings.


localhost
Home
Untitled
jupyter
Untitled (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Python [conda env:bm2]
Code
CellToolbar
In [2]:
%matplotlib inline
import brian2 as b2
import matplotlib.pyplot as plt
import numpy as np
from neurodynex.leaky_integrate_and_fire import LIF
from neurodynex.tools import input_factory, plot_tools
LIF.getting_started()
nr of spikes: 0
Step current
1e−9
Input current [A]
min: 0. A
max: 1.2 nA
2
1
0
−1

```
>> source activate bmnn
>> conda install jupyter
```

Alternative procedure: Using python and pip

If you already have Python installed and prefer using PIP, you can get the most recent versions of this repository as a pypi package called `neurodynex`.

To install the exercises using `pip` simply execute (the `--upgrade` flag will overwrite existing installations with the newest versions):

```
pip install --upgrade jupyter
pip install --upgrade neurodynex
```

Note: Should you want to run Spyder to work on the exercises, and you're running into problems (commonly, after running `conda install spyder` you can not start `spyder` due to an error related to numpy), try the following:

```
# create a new conda environment with spyder and the exercises
conda create --name neurodynex -c brian-team -c epfl-lcn neurodynex spyder

# activate the environment
source activate neurodynex
```

This creates a new conda environment (here is more information on conda environments) in which you can use spyder together with the exercises.

Links

Here are some useful links to get started with Python and Brian

- https://www.python.org/about/gettingstarted/
- https://brian2.readthedocs.io/en/latest/index.html
- http://www.scipy.org
- http://Matplotlib.sf.net

Leaky-integrate-and-fire model

Book chapters

See Chapter 1 Section 3 on general information about leaky-integrate-and-fire models.

Python classes

The *leaky_integrate_and_fire.LIF* implements a parameterizable LIF model. Call `LIF.getting_started()` and have a look at it's source code to learn how to efficiently use the *leaky_integrate_and_fire.LIF* module.

A typical Jupyter notebook looks like this:

```
    %matplotlib inline
import brian2 as b2
import matplotlib.pyplot as plt
import numpy as np
from neurodynex.leaky_integrate_and_fire import LIF
from neurodynex.tools import input_factory, plot_tools


LIF.getting_started()
LIF.print_default_parameters()
```

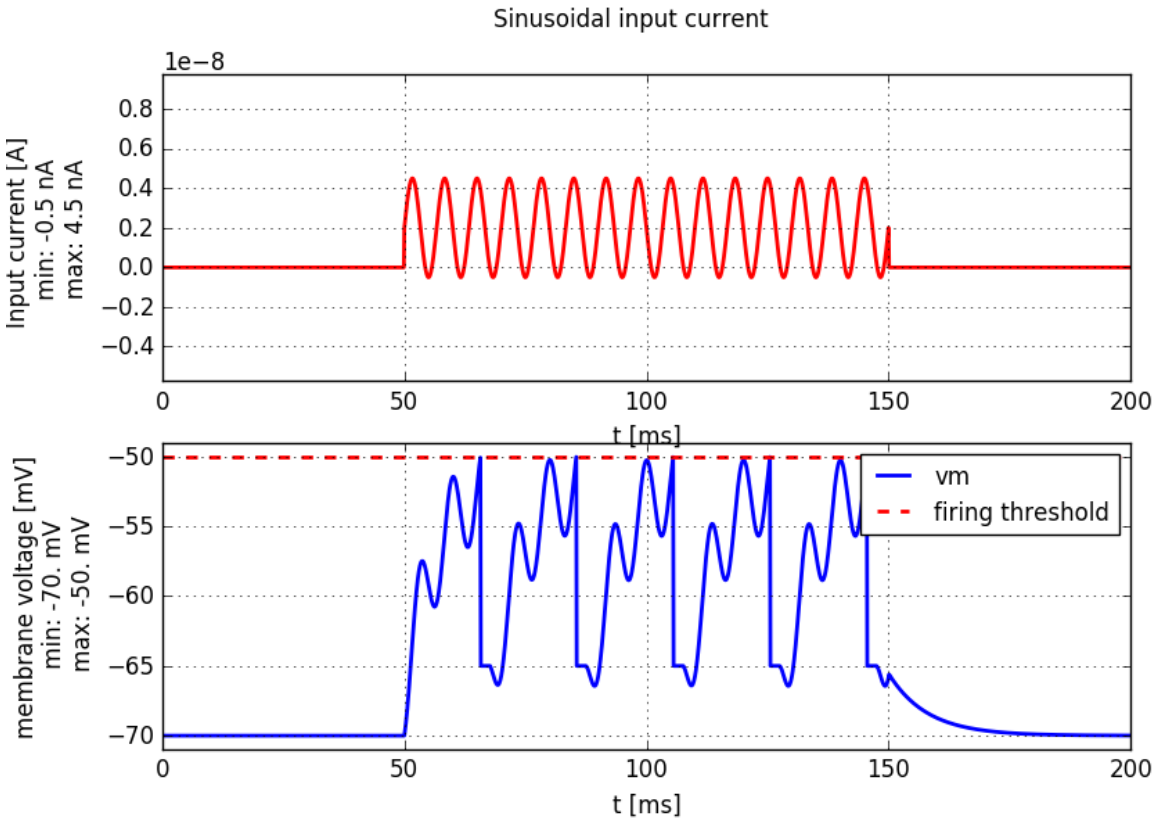


Fig. 1.2: Injection of a sinusoidal current into a leaky-integrate-and-fire neuron

Note that you can change all parameter of the LIF neuron by using the named parameters of the function *simulate_LIF_neuron()*. If you do not specify any parameter, the following default values are used:

```
V_REST = -70*b2.mV
V_RESET = -65*b2.mV
FIRING_THRESHOLD = -50*b2.mV
MEMBRANE_RESISTANCE = 10. * b2.Mohm
MEMBRANE_TIME_SCALE = 8. * b2.ms
ABSOLUTE_REFRACTORY_PERIOD = 2.0 * b2.ms
```

Exercise: minimal current

In the absence of an input current, a LIF neuron has a constant membrane voltage vm=v_rest. If an input current drives vm above the firing threshold, a spike is generated. Then, vm is reset to v_reset and the neuron ignores any input during the refractroy period.

Question: minimal current (calculation)

For the default neuron parameters (see above) compute the minimal amplitude **i_min** of a step current to elicitate a spike. You can access these default values in your code and do the calculation with correct units:

```
from neurodynex.leaky_integrate_and_fire import LIF
print("resting potential: {}".format(LIF.V_REST))
```

Question: minimal current (simulation)

Use the value **i_min** you've computed and verify your result: inject a step current of amplitude i_min for 100ms into the LIF neuron and plot the membrane voltage. Vm should approach the firing threshold but *not* fire. We have implemented a couple of helper functions to solve this task. Use this code block, but make sure you understand it and you've read the docs of the functions *LIF.simulate_LIF_neuron()*, *input_factory.get_step_current()* and *plot_tools.plot_voltage_and_current_traces()*.

```
import brian2 as b2
from neurodynex.leaky_integrate_and_fire import LIF
from neurodynex.tools import input_factory

# create a step current with amplitude= i_min
step_current = input_factory.get_step_current(
    t_start=5, t_end=100, unit_time=b2.ms,
    amplitude= i_min)  # set i_min to your value

# run the LIF model.
# Note: As we do not specify any model parameters, the simulation runs with the␣
↪default values
(state_monitor,spike_monitor) = LIF.simulate_LIF_neuron(input_current=step_current,␣
↪simulation_time = 100 * b2.ms)

# plot I and vm
plot_tools.plot_voltage_and_current_traces(
state_monitor, step_current, title="min input", firing_threshold=LIF.FIRING_THRESHOLD)
print("nr of spikes: {}".format(spike_monitor.count[0]))  # should be 0
```

Exercise: f-I Curve

For a constant input current I, a LIF neuron fires regularly with firing frequency f. If the current is to small (I < I_min) f is 0Hz; for larger I the rate increases. A neuron's firing-rate versus input-amplitude relationship is visualized in an "f-I curve".

Question: f-I Curve and refractoryness

We now study the f-I curve for a neuron with a refractory period of 3ms (see *LIF.simulate_LIF_neuron()* to learn how to set a refractory period).

1. Sketch the f-I curve you expect to see
2. What is the maximum rate at which this neuron can fire?
3. Inject currents of different amplitudes (from 0nA to 100nA) into a LIF neuron. For each current, run the simulation for 500ms and determine the firing frequency in Hz. Then plot the f-I curve. Pay attention to the low input current.

Exercise: "Experimentally" estimate the parameters of a LIF neuron

A LIF neuron is determined by the following parameters: Resting potential, Reset voltage, Firing threshold, Membrane resistance, Membrane time-scale, Absolute refractory period. By injecting a known test current into a LIF neuron (with unknown parameters), you can determine the neuron properties from the voltage response.

Question: "Read" the LIF parameters out of the vm plot

1. Get a random parameter set
2. Create an input current of your choice.
3. Simulate the LIF neuron using the random parameters and your test-current. Note that the simulation runs for a fixed duration of 50ms.
4. Plot the membrane voltage and estimate the parameters. You do not have to write code to analyse the voltage data in the StateMonitor. Simply estimate the values from the plot. For the Membrane resistance and the Membrane time-scale you might have to change your current.
5. compare your estimates with the true values.

Again, you do not have to write much code. Use the helper functions:

```
# get a random parameter. provide a random seed to have a reproducible experiment
random_parameters = LIF.get_random_param_set(random_seed=432)

# define your test current
test_current = input_factory.get_step_current(
    t_start=..., t_end=..., unit_time=b2.ms, amplitude= ... * b2.namp)

# probe the neuron. pass the test current AND the random params to the function
state_monitor, spike_monitor = LIF.simulate_random_neuron(test_current, random_
↪parameters)

# plot
plot_tools.plot_voltage_and_current_traces(state_monitor, test_current, title=
↪"experiment")

# print the parameters to the console and compare with your estimates
# LIF.print_obfuscated_parameters(random_parameters)
```

Exercise: Sinusoidal input current and subthreshold response

In the subthreshold regime (no spike), the LIF neuron is a linear system and the membrane voltage is a filtered version of the input current. In this exercise we study the properties of this linear system when it gets a sinusoidal stimulus.

Question

Create a sinusoidal input current (see example below) and inject it into the LIF neuron. Determine the phase and amplitude of the membrane voltage.

```
# note the higher resolution when discretizing the sine wave: we specify unit_time=0.
↪1 * b2.ms
sinusoidal_current = input_factory.get_sinusoidal_current(200, 1000, unit_time=0.1 *␣
↪b2.ms,
                                                amplitude= 2.5 * b2.namp,␣
↪frequency=250*b2.Hz,
                                                direct_current=0. * b2.namp)

# run the LIF model. By setting the firing threshold to to a high value, we make sure␣
↪to stay in the linear (non spiking) regime.
(state_monitor, spike_monitor) = LIF.simulate_LIF_neuron(input_current=sinusoidal_
↪current, simulation_time = 120 * b2.ms, firing_threshold=0*b2.mV)

# plot the membrane voltage
plot_tools.plot_voltage_and_current_traces(state_monitor, sinusoidal_current, title=
↪"Sinusoidal input current")
print("nr of spikes: {}".format(spike_monitor.count[0]))
```

Question

For input frequencies between $10Hz$ and $1kHz$, plot the the resulting *amplitude of subthreshold oscillations* of the membrane potential vs. input frequency.

Question

For input frequencies between $10Hz$ and $1kHz$, plot the resulting *phase shift of subthreshold oscillations* of the membrane potential vs. input frequency.

Question

To what type of filter (High-Pass, Low-Pass) does this correspond?

Note: It is not straight forward to automatically determine the phase shift in a script. For this exercise, simply get it "visually" from your plot. If you want to automatize the procedure in your Python script you could try the function scipy.signal.correlate().

The Exponential Integrate-and-Fire model

Book chapters

The Exponential Integrate-and-Fire model is introduced in Chapter 5 Section 2

Python classes

The module *exponential_integrate_fire.exp_IF* implements the dynamics given in the book (equation 5.6).

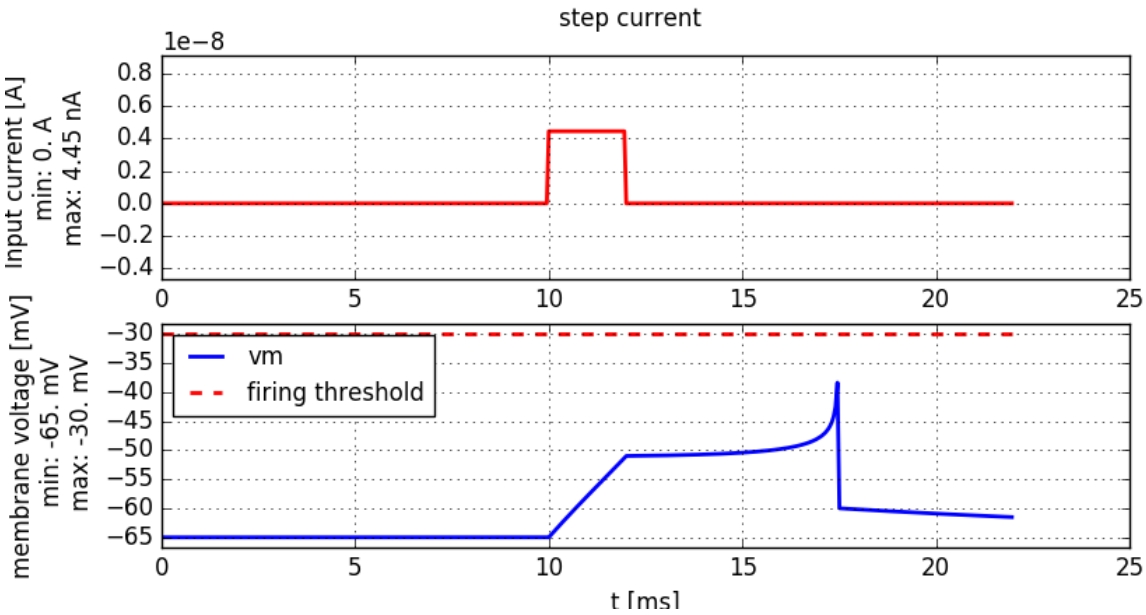


Fig. 1.3: A short pulse current of 2ms duration is injected into an Exponential-Integrate-and-Fire neuron. The current amplitude is just sufficient to elicit a spike.

To get started, copy the following code into a Jupyter notebook. It follows a common pattern used in these exercises: use the input_factory to get a specific current, inject it into the neuron model we provide, and finally use the plot_tools to visualize the state variables:

```
% matplotlib inline
import brian2 as b2
import matplotlib.pyplot as plt
import neurodynex.exponential_integrate_fire.exp_IF as exp_IF
from neurodynex.tools import plot_tools, input_factory


input_current = input_factory.get_step_current(
    t_start=20, t_end=120, unit_time=b2.ms, amplitude=0.8 * b2.namp)

state_monitor, spike_monitor = exp_IF.simulate_exponential_IF_neuron(
    I_stim=input_current, simulation_time=200*b2.ms)

plot_tools.plot_voltage_and_current_traces(
    state_monitor, input_current,title="step current",
    firing_threshold=exp_IF.FIRING_THRESHOLD_v_spike)
print("nr of spikes: {}".format(spike_monitor.count[0]))
```

Note that you can change all parameters of the neuron by using the named parameters of the function *simulate_exponential_IF_neuron()*. If you do not specify any parameter, the default values are used (see next code block). You can access these variables in your code by prefixing them with the module name (for example exp_IF.FIRING_THRESHOLD_v_spike).

```
MEMBRANE_TIME_SCALE_tau = 12.0 * b2.ms
MEMBRANE_RESISTANCE_R = 20.0 * b2.Mohm
V_REST = -65.0 * b2.mV
V_RESET = -60.0 * b2.mV
RHEOBASE_THRESHOLD_v_rh = -55.0 * b2.mV
```

```
SHARPNESS_delta_T = 2.0 * b2.mV
FIRING_THRESHOLD_v_spike = -30. * b2.mV
```

Exercise: rehobase threshold

The goal of this exercise is to study the minimal current that can elicit a spike and to understand the different notions of a firing threshold. The Exponential-Integrate-and-Fire neuron model has two threshold related parameters. They correspond to the named parameters 'v_spike' and 'v_rheobase' in the function *simulate_exponential_IF_neuron()*.

Question:

- Modify the code example given above: Call *simulate_exponential_IF_neuron()* and set the function parameter v_spike=+10mV (which overrides the default value -30mV). What do you expect to happen? How many spikes will be generated?
- Compute the minimal amplitude I_rh of a constant input current such that the neuron will elicit a spike. If you are not sure what and how to compute I_rh, have a look at Figure 5.1 and the textbox "Rheobase threshold and interpretation of parameters" in the book.
- Validate your result: Modify the code given above and inject a current of amplitude I_rh and 300 ms duration into the expIF neuron.

Exercise: strength-duration curve

The minimal amplitude to elicit a spike depends on the duration of the current. For an infinitely long current, we've just calculated the rheobase current. For short pulses and step currents, we can "experimentally" determine the minimal currents. If we plot the amplitude versus duration, we get the strength-duration curve

Question:

Have a look at the following code: for the values i = 0, 2 and 6 we did not provide the minimal amplitude, but the entries in min_amp[i] are set to 0. Complete the min_amp list.

- Set the index i to 0
- Enter an informed guess into the min_amp table
- Run the script
- Depending on the plot, increase or decrease the amplitude, repeat until you just get one spike.
- Do the same for i = 2 and i = 6

At the end of the script, the strength-duration curve is plotted. Discuss it. You may want to add a log-log plot to better see the asymptotic behaviour.

```
% matplotlib inline
import brian2 as b2
import matplotlib.pyplot as plt
import neurodynex.exponential_integrate_fire.exp_IF as exp_IF
from neurodynex.tools import plot_tools, input_factory

i=1  #change i and find the value that goes into min_amp
```

```
durations = [1,   2,    5,  10,   20,   50, 100]
min_amp =   [0., 4.42, 0., 1.10, .70, .48, 0.]

t=durations[i]
I_amp = min_amp[i]*b2.namp
title_txt = "I_amp={}, t={}".format(I_amp, t*b2.ms)

input_current = input_factory.get_step_current(t_start=10, t_end=10+t-1, unit_time=b2.
↪ms, amplitude=I_amp)

state_monitor, spike_monitor = exp_IF.simulate_exponential_IF_neuron(I_stim=input_
↪current, simulation_time=(t+20)*b2.ms)

plot_tools.plot_voltage_and_current_traces(state_monitor, input_current,
                                           title=title_txt, firing_threshold=exp_IF.
↪FIRING_THRESHOLD_v_spike,
                                          legend_location=2)
print("nr of spikes: {}".format(spike_monitor.count[0]))

plt.plot(durations, min_amp)
plt.title("Strength-Duration curve")
plt.xlabel("t [ms]")
plt.ylabel("min amplitude [nAmp]")
```

AdEx: the Adaptive Exponential Integrate-and-Fire model

Book chapters

The Adaptive Exponential Integrate-and-Fire model is introduced in Chapter 6 Section 1

Python classes

Use function *AdEx.simulate_AdEx_neuron()* to run the model for different input currents and different parameters. Get started by running the following script:

```
% matplotlib inline
import brian2 as b2
from neurodynex.adex_model import AdEx
from neurodynex.tools import plot_tools, input_factory

current = input_factory.get_step_current(10, 250, 1. * b2.ms, 65.0 * b2.pA)
state_monitor, spike_monitor = AdEx.simulate_AdEx_neuron(I_stim=current, simulation_
↪time=400 * b2.ms)
plot_tools.plot_voltage_and_current_traces(state_monitor, current)
print("nr of spikes: {}".format(spike_monitor.count[0]))
# AdEx.plot_adex_state(state_monitor)
```

Exercise: Adaptation and firing patterns

We have implemented an Exponential Integrate-and-Fire model with a single adaptation current `w`:

$$\left[\begin{array}{rcl} \tau_m \frac{du}{dt} & = & -(u-u_{rest}) + \Delta_T exp(\frac{u-\vartheta_{rh}}{\Delta_T}) - Rw + RI(t) \\ \tau_w \frac{dw}{dt} & = & a(u-u_{rest}) - w + b\tau_w \sum_{t^{(f)}} \delta(t-t^{(f)}) \end{array}\right. \tag{1.1}$$

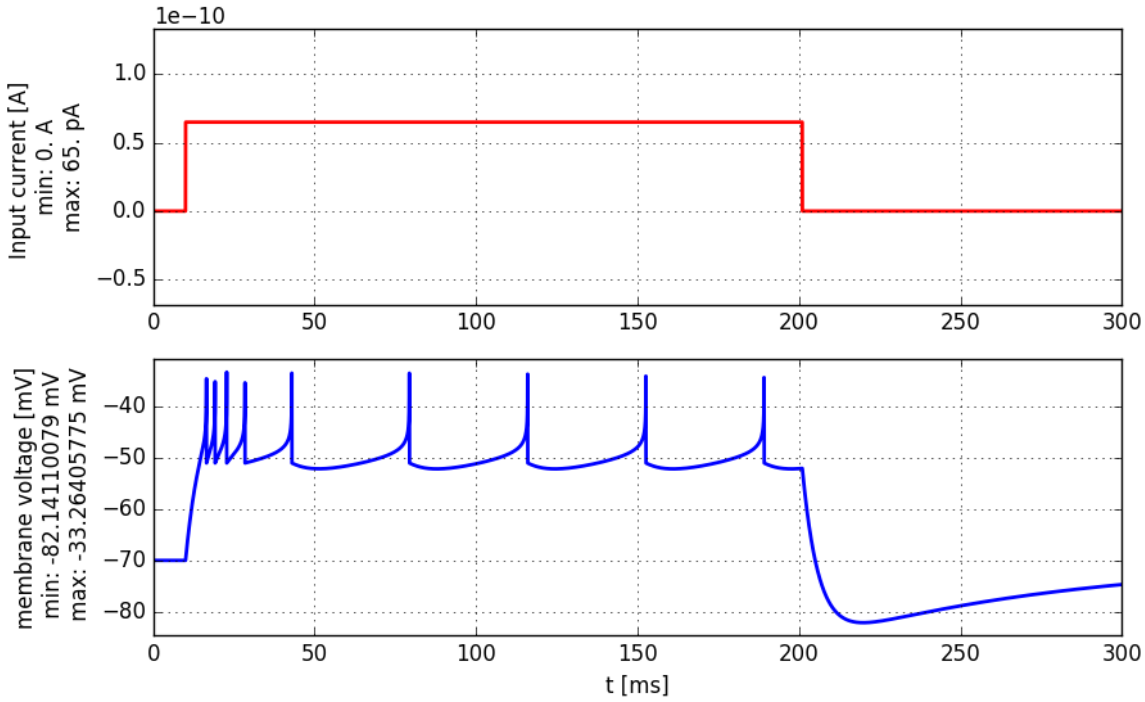


Fig. 1.4: Step current injection into an AdEx neuron.

Question: Firing pattern

- When you simulate the model with the default parameters, it produces the voltage trace shown above. Describe that firing pattern. Use the terminology of Fig. 6.1 in Chapter 6.1
- Call the function *AdEx.simulate_AdEx_neuron()* with different parameters and try to create **adapting**, **bursting** and **irregular** firing patterns. Table 6.1 in Chapter 6.1 provides a starting point for your explorations.
- In order to better understand the dynamics, it is useful to observe the joint evolution of `u` and `w` in a phase diagram. Use the function *AdEx.plot_adex_state()* to get more insights. Fig. 6.3 in Chapter 6 Section 2 shows a few trajectories in the phase diagram.

Note: If you want to set a parameter to 0, Brian still expects a unit. Therefore use `a=0*b2.nS` instead of `a=0`.

If you do not specify any parameter, the following default values are used:

```
MEMBRANE_TIME_SCALE_tau_m = 5 * b2.ms
MEMBRANE_RESISTANCE_R = 500*b2.Mohm
V_REST = -70.0 * b2.mV
V_RESET = -51.0 * b2.mV
RHEOBASE_THRESHOLD_v_rh = -50.0 * b2.mV
SHARPNESS_delta_T = 2.0 * b2.mV
ADAPTATION_VOLTAGE_COUPLING_a = 0.5 * b2.nS
ADAPTATION_TIME_CONSTANT_tau_w = 100.0 * b2.ms
SPIKE_TRIGGERED_ADAPTATION_INCREMENT_b = 7.0 * b2.pA
```

Exercise: phase plane and nullclines

First, try to get some intuition on shape of nullclines by plotting or simply sketching them on a piece of paper and answering the following questions.

1. Plot or sketch the u- and w- nullclines of the AdEx model (`I(t) = 0`)
2. How do the nullclines change with respect to `a`?
3. How do the nullclines change if a constant current `I(t) = c` is applied?
4. What is the interpretation of parameter `b`?
5. How do flow arrows change as `tau_w` gets bigger?

Question:

Can you predict what would be the firing pattern if `a` is small (in the order of `0.01 nS`) ? To do so, consider the following 2 conditions:

1. A large jump `b` and a large time scale `tau_w`.
2. A small jump `b` and a small time scale `tau_w`.

Try to simulate the above conditions, to see if your predictions were true.

Question:

To learn more about the variety of patterns the relatively simple neuron model can reproduce, have a look the following publication: Naud, R., Marcille, N., Clopath, C., Gerstner, W. (2008). Firing patterns in the adaptive exponential integrate-and-fire model. Biological cybernetics, 99(4-5), 335-347.

Dendrites and the (passive) cable equation

Book chapters

In Chapter 3 Section 2 the cable equation is derived and compartmental models are introduced.

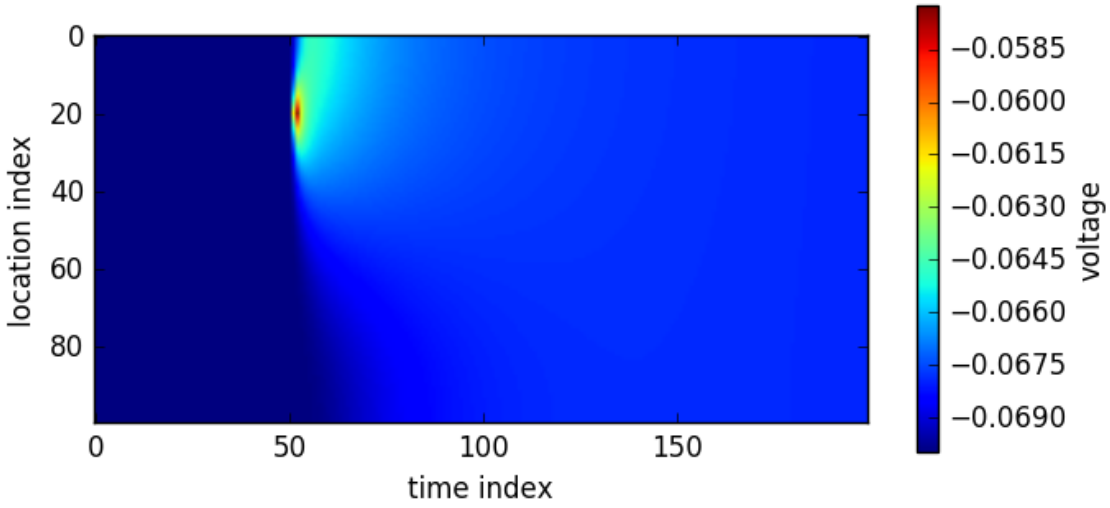
Python classes

The *cable_equation.passive_cable* module implements a passive cable using a Brian2 multicompartment model. To get started, import the module and call the demo function:

```
import brian2 as b2
import matplotlib.pyplot as plt
from neurodynex.cable_equation import passive_cable
from neurodynex.tools import input_factory
passive_cable.getting_started()
```

The function *passive_cable.getting_started()* injects a very short pulse current at (t=500ms, x=100um) into a finite length cable and then lets Brian evolve the dynamics for 2ms. This simulation produces a time x location matrix whose entries are the membrane voltage at each (time,space)-index. The result is visualized using pyplot.imshow.

Note: The axes in the figure above are not scaled to the physical units but show the raw matrix indices. These indices depend on the spatial resolution (number of compartments) and the temporal resolution (brian2.defaultclock.dt). For



the exercises make sure you correctly scale the units using Brian's unit system . As an example, to plot voltage vs. time you call

```
pyplot.plot(voltage_monitor.t / b2.ms, voltage_monitor[0].v / b2.mV)
```

This way, your plot shows voltage in mV and time in ms, which is useful for visualizations. Note that this scaling (to physical units) is different from the scaling in the theoretical derivation (e.g. chapter 3.2.1 where the quantities are rescaled to a unit-free characteristic length scale

---

Using the module *cable_equation.passive_cable*, we study some properties of the passive cable. Note: if you do not specify the cable parameters, the function *cable_equation.passive_cable.simulate_passive_cable()* uses the following default values:

```
CABLE_LENGTH = 500. * b2.um  # length of dendrite
CABLE_DIAMETER = 2. * b2.um  # diameter of dendrite
R_LONGITUDINAL = 0.5 * b2.kohm * b2.mm  # Intracellular medium resistance
R_TRANSVERSAL = 1.25 * b2.Mohm * b2.mm ** 2  # cell membrane resistance (->leak␣
↪current)
E_LEAK = -70. * b2.mV  # reversal potential of the leak current (-> resting potential)
CAPACITANCE = 0.8 * b2.uF / b2.cm ** 2  # membrane capacitance
```

You can easily access those values in your code:

```
from neurodynex.cable_equation import passive_cable
print(passive_cable.R_TRANSVERSAL)
```

Exercise: spatial and temporal evolution of a pulse input

Create a cable of length 800um and inject a 0.1ms long step current of amplitude 0.8nanoAmp at (t=1ms, x=200um). Run Brian for 3 milliseconds.

You can use the function *cable_equation.passive_cable.simulate_passive_cable()* to implement this task. For the parameters not specified here (e.g. dentrite diameter) you can rely on the default values. Have a look at the documentation of *simulate_passive_cable()* and the source code of *passive_cable.getting_started()* to learn how to efficiently solve this exercise. From the specification

of *simulate_passive_cable()* you should also note, that it returns two objects which are helpful to access the values of interest using spatial indexing:

```
voltage_monitor, cable_model = passive_cable.simulate_passive_cable(...)
probe_location = 0.123 * b2.mm
v = voltage_monitor[cable_model.morphology[probe_location]].v
```

Question:

1. What is the maximum depolarization you observe? Where and when does it occur?
2. Plot the temporal evolution (t in [0ms, 3ms]) of the membrane voltage at the locations 0um, 100um, ... , 600 um in one figure.
3. Plot the spatial evolution (x in [0um, 800um]) of the membrane voltage at the time points 1.0ms, 1.1ms, ... , 1.6ms in one plot
4. Discuss the figures.

Exercise: Spatio-temporal input pattern

While the passive cable use here is a very simplified model of a real dendrite, we can still get an idea of how input spikes would look to the soma. Imagine a dendrite of some length and the soma at x=0um. What is the depolarization at x=0 if the dendrite receives multiple spikes at different time/space locations? This is what we study in this exercise:

Create a cable of length 800uM and inject three short pulses A, B, and C at different time/space locations:

A: (t=1.0ms, x=100um)

B: (t=1.5ms, x=200um)

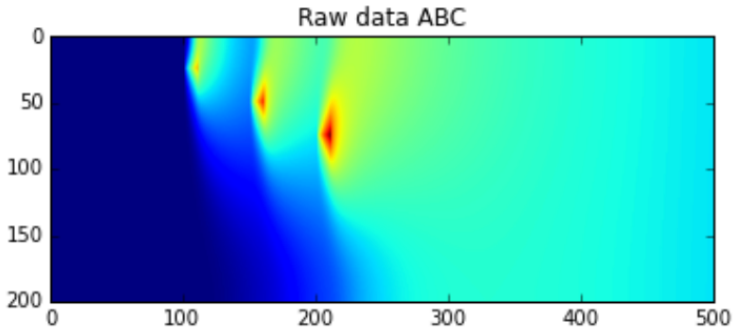
C: (t=2.0ms, x=300um)

Pulse input: 100us duration, 0.8nanoAmp amplitude

Make use of the function *input_factory.get_spikes_current()* to easily create such an input pattern:

```
t_spikes = [10, 15, 20]
l_spikes = [100. * b2.um, 200. * b2.um, 300. * b2.um]
current = input_factory.get_spikes_current(t_spikes, 100*b2.us, 0.8*b2.namp, append_
↪zero=True)
voltage_monitor_ABC, cable_model = passive_cable.simulate_passive_cable(..., current_
↪injection_location=l_spikes, input_current=current, ...)
```

Run Brian for 5 milliseconds. Your simulation for this input pattern should look similar to this figure:



Question

1. plot the temporal evolution (t in [0ms, 5ms]) of the membrane voltage at the soma (x=0). What is the maximal depolarization?
2. reverse the order of the three input spikes:

   C: (t=1.0ms, x=300um)

   B: (t=1.5ms, x=200um)

   A: (t=2.0ms, x=100um)

Again, let Brian simulate 5 milliseconds. In the same figure as before, plot the temporal evolution (t in [0ms, 5ms]) of the membrane voltage at the soma (x=0). What is the maximal depolarization? Discuss the result.

Exercise: Effect of cable parameters

So far, you have called the function *simulate_passive_cable()* without specifying the cable parameters. That means, the model was run with the default values. Look at the documentation of *simulate_passive_cable()* to see which parameters you can change.

Keep in mind that our cable model is very simple compared to what happens in dendrites or axons. But we can still observe the impact of a parameter change on the current flow. As an example, think of a myelinated fiber: it has a much **lower** membrane capacitance and **higher** membrane resistance. Let's compare these two parameter-sets:

Question

Inject a very brief pulse current at (t=.05ms, x=400um). Run Brian twice for 0.2 ms with two different parameter sets (see example below). Plot the temporal evolution of the membrane voltage at x=500um for the two parameter sets. Discuss your observations.

> Note: to better see some of the effects, plot only a short time window and increase the temporal resolution of the numerical approximation (b2.defaultclock.dt = 0.005 * b2.ms)

```
# set 1: (same as defaults)
membrane_resistance_1 = 1.25 * b2.Mohm * b2.mm ** 2
membrane_capacitance_1 = 0.8 * b2.uF / b2.cm ** 2
# set 2: (you can think of a myelinated "cable")
membrane_resistance_2 = 5.0 * b2.Mohm * b2.mm ** 2
membrane_capacitance_2 = 0.2 * b2.uF / b2.cm ** 2
```

Exercise: stationary solution and comparison with theoretical result

Create a cable of length 500um and inject a **constant current** of amplitude 0.1nanoAmp at x=0um. You can use the input_factory to create that current. Note the parameter append_zero=False. As we are not interested in the exact values of the transients, we can speed up the simulation increase the width of a timestep dt: b2.defaultclock.dt = 0.1 * b2.ms

```
b2.defaultclock.dt = 0.1 * b2.ms
current = input_factory.get_step_current(0, 0, unit_time=b2.ms, amplitude=0.1 * b2.
↪namp, append_zero=False)
voltage_monitor, cable_model = passive_cable.simulate_passive_cable(
length=0.5 * b2.mm, current_injection_location = [0*b2.um],
input_current=current, simulation_time=sim_time, nr_compartments=N_comp)
v_X0 = voltage_monitor.v[0,:]  # access the first compartment
```

```
v_Xend = voltage_monitor.v[-1,:]  # access the last compartment
v_Tend = voltage_monitor.v[:, -1]  # access the last time step
```

Question

Before running a simulation, sketch two curves, one for x=0um and one for x=500um, of the membrane potential Vm versus time. What steady state Vm do you expect?

Now run the Brian simulator for 100 milliseconds.

1. Plot Vm vs. time (t in [0ms, 100ms]) at x=0um and x=500um and compare the curves to your sketch.
2. Plot Vm vs location (x in [0um, 500um]) at t=100ms.

Question

1. Compute the characteristic length λ (= length scale = lenght constant) of the cable. Compare your value with the previous figure.

   $\lambda = \sqrt{\frac{r_{Membrane}}{r_{Longitudinal}}}$

Question (Bonus)

You observed that the membrane voltage reaches a location dependent steady-state value. Here we compare those simulation results to the analytical solution.

1. Derive the analytical steady-state solution. (finite cable length L, constant current I0 at x=0, sealed end: no longitudinal current at x=L).
2. Plot the analytical solution and the simulation result in one figure.
3. Run the simulation with different resolution parameters (change defaultclock.dt and/or the number of compartments). Compare the simulation with the analytical solution.
4. If you need help to get started, or if you're not sure about the analytical solution, you can find a solution in the Brian2 docs :

Numerical integration of the HH model of the squid axon

Book chapters

See Chapter 2 Section 2 on general information about the Hodgkin-Huxley equations and models.

Python classes

The *hodgkin_huxley.HH* module contains all code required for this exercise. It implements a Hodgkin-Huxley neuron model. At the beginning of your exercise solutions, import the modules and run the demo function.

```
%matplotlib inline
import brian2 as b2
import matplotlib.pyplot as plt
import numpy as np
from neurodynex.hodgkin_huxley import HH
from neurodynex.tools import input_factory
```

```
HH.getting_started()
```

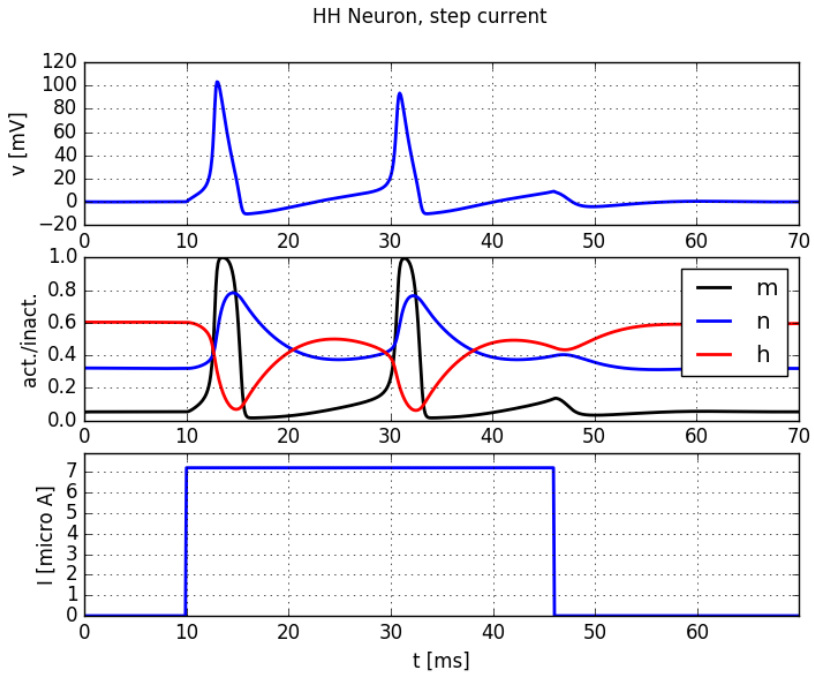


Fig. 1.5: Step current injection into a Hodgkin-Huxley neuron

Exercise: step current response

We study the response of a Hodgkin-Huxley neuron to different input currents. Have a look at the documentation of the functions *HH.simulate_HH_neuron()* and *HH.plot_data()* and the module *neurodynex.tools.input_factory*.

Question

What is the lowest step current amplitude I_min for generating **at least one spike**? Determine the value by trying different input amplitudes in the code fragment:

```
current = input_factory.get_step_current(5, 100, b2.ms, I_min *b2.uA)
state_monitor = HH.simulate_HH_neuron(current, 120 * b2.ms)
HH.plot_data(state_monitor, title="HH Neuron, minimal current")
```

Question

- What is the lowest step current amplitude to generate **repetitive firing**?
- Discuss the difference between the two regimes.

Exercise: slow and fast ramp current

The minimal current to elicit a spike does not just depend on the amplitude I or on the total charge Q of the current, but on the "shape" of the current. Let's see why:

Question

Inject a slow ramp current into a HH neuron. The current has amplitude `0A` at t in [0, 5] ms and linearly increases to an amplitude of `12.0uAmp` at `t=ramp_t_end`. At `t>ramp_t_end`, the current is set to `0A`. Using the following code, reduce `slow_ramp_t_end` to the maximal duration of the ramp current, such that the neuron does **not** spike. Make sure you simulate system for at least 20ms after the current stops.

- What is the membrane voltage at the time when the current injection stops (t=slow_ramp_t_end)?

```
b2.defaultclock.dt = 0.02*b2.ms
slow_ramp_t_end = 60  # no spike. make it shorter
slow_ramp_current = input_factory.get_ramp_current(5, slow_ramp_t_end, b2.ms, 0.*b2.
↪uA, 12.0*b2.uA)
state_monitor = HH.simulate_HH_neuron(slow_ramp_current, 90 * b2.ms)
idx_t_end = int(round(slow_ramp_t_end*b2.ms / b2.defaultclock.dt))
voltage_slow = state_monitor.vm[0,idx_t_end]
print("voltage_slow={}".format(voltage_slow))
```

Question

Do the same as before but for a fast ramp current: The maximal amplitude at `t=ramp_t_end` is `4.5uAmp`. Start with `fast_ramp_t_end = 8ms` and then increase it until you observe a spike. Note: Technically the input current is implemented using a TimedArray. For a short, steep ramp, the one milliseconds discretization for the current is not high enough. You can create a finer resolution by setting the parameter `unit_time` in the function *`input_factory.get_ramp_current()`* (see next code block)

- What is the membrane voltage at the time when the current injection stops (t=fast_ramp_t_end)?

```
b2.defaultclock.dt = 0.02*b2.ms
fast_ramp_t_end = 80  # no spike. make it longer
fast_ramp_current = input_factory.get_ramp_current(50, fast_ramp_t_end, 0.1*b2.ms, 0.
↪*b2.uA, 4.5*b2.uA)
state_monitor = HH.simulate_HH_neuron(fast_ramp_current, 40 * b2.ms)
idx_t_end = int(round(fast_ramp_t_end*0.1*b2.ms / b2.defaultclock.dt))
voltage_fast = state_monitor.vm[0,idx_t_end]
print("voltage_fast={}".format(voltage_fast))
```

Question

Use the function *`HH.plot_data()`* to visualize the dynamics of the system for the fast and the slow case above. Discuss the differences between the two situations. Why are the two "threshold" voltages different? Link your observation to the gating variables m,n, and h. Hint: have a look at Chapter 2 Figure 2.3

Exercise: Rebound Spike

A HH neuron can spike not only if it receives a sufficiently strong depolarizing input current but also after a hyperpolarizing current. Such a spike is called a *rebound spike*.

Question

Inject a hyperpolarizing step current `I_amp = -1 uA` for 20ms into the HH neuron. Simulate the neuron for 50 ms and plot the voltage trace and the gating variables. Repeat the simulation with `I_amp = -5 uA` What is happening here? To which gating variable do you attribute this rebound spike?

Exercise: Brian implementation of a HH neuron

In this exercise you will learn to work with the Brian2 model equations. To do so, get the source code of the function *`HH.simulate_HH_neuron()`* (follow the link to the documentation and then click on the [source] link). Copy the function code and paste it into your Jupyter Notebook. Change the function name from simulate_HH_neuron to a name of your choice. Have a look at the source code and find the conductance parameters gK and gNa.

Question

In the source code of your function, change the density of sodium channels. Increase it by a factor of 1.4. Stimulate this modified neuron with a step current.

- What is the minimal current leading to repetitive spiking? Explain.
- Run a simulation with no input current to determine the resting potential of the neuron. Link your observation to the Goldman–Hodgkin–Katz voltage equation.
- If you increase the sodium conductance further, you can observe repetitive firing even in the absence of input, why?

FitzHugh-Nagumo: Phase plane and bifurcation analysis

Book chapters

See Chapter 4 and especially Chapter 4 Section 3 for background knowledge on phase plane analysis.

Python classes

In this exercise we study the phase plane of a two dimensional dynamical system implemented in the module *`phase_plane_analysis.fitzhugh_nagumo`*. To get started, copy the following code block into your Jupyter Notebook. Check the documentation to learn how to use these functions. Make sure you understand the parameters the functions take.

```
%matplotlib inline
import brian2 as b2
import matplotlib.pyplot as plt
import numpy as np
from neurodynex.phase_plane_analysis import fitzhugh_nagumo

fitzhugh_nagumo.plot_flow()

fixed_point = fitzhugh_nagumo.get_fixed_point()
print("fixed_point: {}".format(fixed_point))

plt.figure()
trajectory = fitzhugh_nagumo.get_trajectory()
plt.plot(trajectory[0], trajectory[1])
```

Exercise: Phase plane analysis

We have implemented the following Fitzhugh-Nagumo model.

$$\left[\begin{array}{rcl} \frac{du}{dt} & = & u\left(1-u^2\right)-w+I \equiv F(u,w) \\ \frac{dw}{dt} & = & \varepsilon\left(u-0.5w+1\right) \equiv \varepsilon G(u,w)\,, \end{array}\right. \tag{1.2}$$

Question

Use the function `plt.plot` to plot the two nullclines of the Fitzhugh-Nagumo system given in Eq. (1.2) for $I = 0$ and $\varepsilon = 0.1$.

Plot the nullclines in the $u - w$ plane, for voltages in the region $u \in [-2.5, 2.5]$.

Note: For instance the following example shows plotting the function $y(x) = -\frac{x^2}{2} + x + 1$:

```
x = np.arange(-2.5, 2.51, .1)  # create an array of x values
y = -x**2 / 2. + x + 1  # calculate the function values for the given x values
plt.plot(x, y, color='black')  # plot y as a function of x
plt.xlim(-2.5, 2.5)  # constrain the x limits of the plot
```

You can use similar code to plot the nullclines, inserting the appropriate equations.

Question

Get the lists `t`, `u` and `w` by calling `t, u, w = fitzhugh_nagumo.get_trajectory(u_0, w_0, I)` for $u_0 = 0$, $w_0 = 0$ and $I = 1.3$. They are corresponding values of t, $u(t)$ and $w(t)$ during trajectories starting at the given point (u_0, w_0) for a given **constant** input current I. Plot the nullclines for this given current and the trajectories into the $u - w$ plane.

Question

At this point for the same current I, call the function `plot_flow`, which adds the flow created by the system Eq. (1.2) to your plot. This indicates the direction that trajectories will take.

Note: If everything went right so far, the trajectories should follow the flow. First, create a new figure by calling `plt.figure()` and then plot the u data points from the trajectory obtained in *the previous exercise* on the ordinate.

You can do this by using the `plt.plot` function and passing only the array of u data points:

```
u = [1,2,3,4]  # example data points of the u trajectory
plot(u, color='blue')  # plot will assume that u is the ordinate data
```

Question

Finally, change the input current in your python file to other values $I > 0$ and reload it. You might have to first define I as a variable and then use this variable in all following commands if you did not do so already. At which value of I do you observe the change in stability of the system?

Exercise: Jacobian & Eigenvalues

The linear stability of a system of differential equations can be evaluated by calculating the eigenvalues of the system's Jacobian at the fixed points. In the following we will graphically explore the linear stability of the fixed point of the system Eq. (1.2). We will find that the linear stability changes as the input current crosses a critical value.

Question

Set $\varepsilon = .1$ and I to zero for the moment. Then, the Jacobian of Eq. (1.2) as a function of the fixed point is given by

$$J(u_0, w_0) = \begin{pmatrix} 1 - 3u_0^2 & -1 \\ 0.1 & -0.05 \end{pmatrix}$$

Write a python function `get_jacobian(u_0,w_0)` that returns the Jacobian evaluated for a given fixed point (u_0, v_0) as a python list.

Note: An example for a function that returns a list corresponding to the matrix $M(a,b) = \begin{pmatrix} a & 1 \\ 0 & b \end{pmatrix}$ is:

```
def get_M(a,b):
        return [[a,1],[0,b]] # return the matrix
```

Question

The function *`u0,w0 = get_fixed_point(I)`* gives you the numerical coordinates of the fixed point for a given current I. Use the function you created in *the previous exercise* to evaluate the Jacobian at this fixed point and store it in a new variable `J`.

Question

Calculate the eigenvalues of the Jacobian `J`, which you computed in *the previous exercise* , by using the function `np.linalg.eigvals(J)`. Both should be negative for $I = 0$.

Exercise: Bifurcation analysis

Wrap the code you wrote so far by a loop, to calculate the eigenvalues for increasing values of I. Store the changing values of each eigenvalue in seperate lists, and finally plot their real values against I.

Note:

You can use this example loop to help you getting started

```
import numpy as np
list1 = []
list2 = []
currents = np.arange(0,4,.1) # the I values to use
for I in currents:
    # your code to calculate the eigenvalues e = [e1,e2] for a given I goes here
    list1.append(e[0].real) # store each value in a separate list
    list2.append(e[1].real)
```

```
# your code to plot list1 and list 2 against I goes here
```

Question

In what range of I are the real parts of eigenvalues positive?

Question

Compare this *with your earlier result* for the critical I. What does this imply for the stability of the fixed point? What has become stable in this system instead of the fixed point?

Hopfield Network model of associative memory

Book chapters

See Chapter 17 Section 2 for an introduction to Hopfield networks.

Python classes

Hopfield networks can be analyzed mathematically. In this Python exercise we focus on visualization and simulation to develop our intuition about Hopfield dynamics.

We provide a couple of functions to easily create patterns, store them in the network and visualize the network dynamics. Check the modules *hopfield_network.network*, *hopfield_network.pattern_tools* and *hopfield_network.plot_tools* to learn the building blocks we provide.

Note: If you instantiate a new object of class `network.HopfieldNetwork` it's default dynamics are **deterministic** and **synchronous**. That is, all states are updated at the same time using the sign function. We use this dynamics in all exercises described below.

Getting started:

Run the following code. Read the inline comments and check the documentation. The patterns and the flipped pixels are randomly chosen. Therefore the result changes every time you execute this code. Run it several times and change some parameters like nr_patterns and nr_of_flips.

```
%matplotlib inline
from neurodynex.hopfield_network import network, pattern_tools, plot_tools

pattern_size = 5

# create an instance of the class HopfieldNetwork
hopfield_net = network.HopfieldNetwork(nr_neurons= pattern_size**2)
# instantiate a pattern factory
factory = pattern_tools.PatternFactory(pattern_size, pattern_size)
# create a checkerboard pattern and add it to the pattern list
checkerboard = factory.create_checkerboard()
pattern_list = [checkerboard]
```

```
# add random patterns to the list
pattern_list.extend(factory.create_random_pattern_list(nr_patterns=3, on_
↪probability=0.5))
plot_tools.plot_pattern_list(pattern_list)
# how similar are the random patterns and the checkerboard? Check the overlaps
overlap_matrix = pattern_tools.compute_overlap_matrix(pattern_list)
plot_tools.plot_overlap_matrix(overlap_matrix)

# let the hopfield network "learn" the patterns. Note: they are not stored
# explicitly but only network weights are updated !
hopfield_net.store_patterns(pattern_list)

# create a noisy version of a pattern and use that to initialize the network
noisy_init_state = pattern_tools.flip_n(checkerboard, nr_of_flips=4)
hopfield_net.set_state_from_pattern(noisy_init_state)

# from this initial state, let the network dynamics evolve.
states = hopfield_net.run_with_monitoring(nr_steps=4)

# each network state is a vector. reshape it to the same shape used to create the␣
↪patterns.
states_as_patterns = factory.reshape_patterns(states)
# plot the states of the network
plot_tools.plot_state_sequence_and_overlap(states_as_patterns, pattern_list,␣
↪reference_idx=0, suptitle="Network dynamics")
```

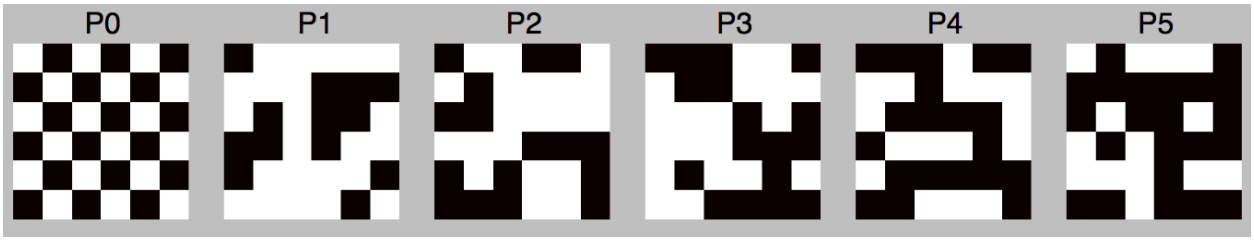


Fig. 1.6: Six patterns are stored in a Hopfield network.

Note: The network state is a vector of N neurons. For visualization we use 2d patterns which are two dimensional numpy.ndarrays of size = (length, width). To store such patterns, initialize the network with N = length x width neurons.

Introduction: Hopfield-networks

This exercise uses a model in which neurons are pixels and take the values of -1 (*off*) or +1 (*on*). The network can store a certain number of pixel patterns, which is to be investigated in this exercise. During a retrieval phase, the network is started with some initial configuration and the network dynamics evolves towards the stored pattern (attractor) which is closest to the initial configuration.

The dynamics is that of equation:

$$S_i(t+1) = sgn\left(\sum_j w_{ij} S_j(t)\right)$$

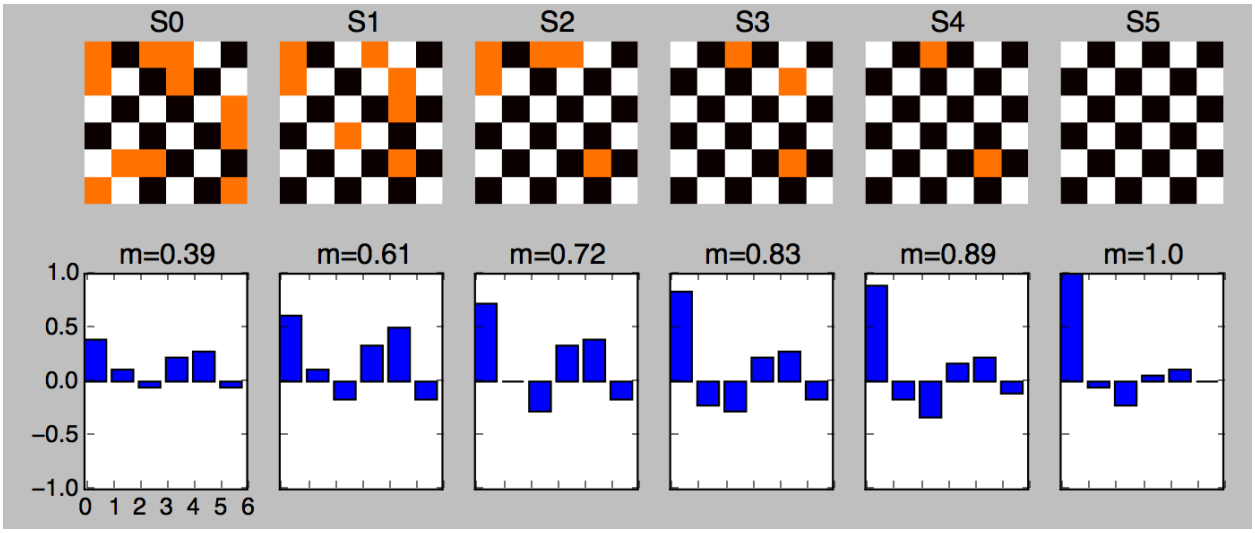


Fig. 1.7: The network is initialized with a (very) noisy pattern S(t=0). Then, the dynamics recover pattern P0 in 5 iterations.

In the Hopfield model each neuron is connected to every other neuron (full connectivity). The connection matrix is

$$w_{ij} = \frac{1}{N} \sum_{\mu} p_i^{\mu} p_j^{\mu}$$

where N is the number of neurons, p_i^{μ} is the value of neuron i in pattern number μ and the sum runs over all patterns from $\mu = 1$ to $\mu = P$. This is a simple correlation based learning rule (Hebbian learning). Since it is not a iterative rule it is sometimes called one-shot learning. The learning rule works best if the patterns that are to be stored are random patterns with equal probability for on (+1) and off (-1). In a large networks (N to infinity) the number of random patterns that can be stored is approximately 0.14 times N.

Exercise: N=4x4 Hopfield-network

We study how a network stores and retrieve patterns. Using a small network of only 16 neurons allows us to have a close look at the network weights and dynamics.

Question: Storing a single pattern

Modify the Python code given above to implement this exercise:

1. Create a network with N=16 neurons.
2. Create a single 4 by 4 checkerboard pattern.
3. Store the checkerboard in the network.
4. Set the initial state of the network to a noisy version of the checkerboard (nr_flipped_pixels = 5).
5. Let the network dynamics evolve for 4 iterations.
6. Plot the sequence of network states along with the overlap of network state with the checkerboard.

Now test whether the network can still retrieve the pattern if we increase the number of flipped pixels. What happens at nr_flipped_pixels = 8, what if nr_flipped_pixels > 8 ?

Question: the weights matrix

The patterns a Hopfield network learns are not stored explicitly. Instead, the network learns by adjusting the weights to the pattern set it is presented during learning. Let's visualize this.

1. Create a new 4x4 network. Do not yet store any pattern.
2. What is the size of the network matrix?
3. Visualize the weight matrix using the function *plot_tools.plot_nework_weights()*. It takes the network as a parameter.
4. Create a checkerboard, store it in the network.
5. Plot the weights matrix. What weight values do occur?
6. Create a new 4x4 network
7. Create an L-shaped pattern (look at the pattern factory doc), store it in the network
8. Plot the weights matrix. What weight values do occur?
9. Create a new 4x4 network
10. Create a checkerboard and an L-shaped pattern. Store **both** patterns in the network
11. Plot the weights matrix. What weight values do occur? How does this matrix compare to the two previous matrices?

Note: The mapping of the 2 dimensional patterns onto the one dimensional list of network neurons is internal to the implementation of the network. You cannot know which pixel (x,y) in the pattern corresponds to which network neuron i.

Question (optional): Weights Distribution

It's interesting to look at the weights distribution in the three previous cases. You can easily plot a histogram by adding the following two lines to your script. It assumes you have stored your network in the variable 'hopfield_net'.

```
plt.figure()
plt.hist(hopfield_net.weights.flatten())
```

Exercise: Capacity of an N=100 Hopfield-network

Larger networks can store more patterns. There is a theoretical limit: the capacity of the Hopfield network. Read chapter "17.2.4 Memory capacity" to learn how memory retrieval, pattern completion and the network capacity are related.

Question:

A Hopfield network implements so called **associative** or **content-adressable** memory. Explain what this means.

Question:

Using the value C_{store} given in the book, how many patterns can you store in a N=10x10 network? Use this number **K** in the next question:

Question:

Create an N=10x10 network and store a checkerboard pattern together with **(K-1) random patterns**. Then initialize the network with the **unchanged** checkerboard pattern. Let the network evolve for five iterations.

Rerun your script a few times. What do you observe?

Exercise: Non-random patterns

In the previous exercises we used random patterns. Now we us a list of structured patterns: the letters A to Z. Each letter is represented in a 10 by 10 grid.

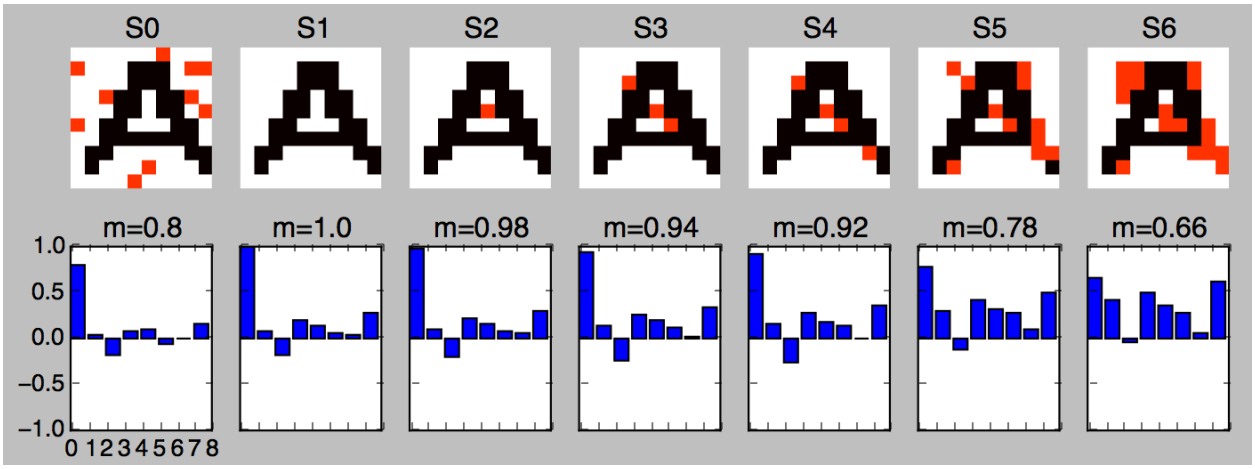


Fig. 1.8: Eight letters (including 'A') are stored in a Hopfield network. The letter 'A' is not recovered.

Question:

Run the following code. Read the inline comments and look up the doc of functions you do not know.

```
%matplotlib inline
import matplotlib.pyplot as plt
from neurodynex.hopfield_network import network, pattern_tools, plot_tools
import numpy

# the letters we want to store in the hopfield network
letter_list = ['A', 'B', 'C', 'S', 'X', 'Y', 'Z']

# set a seed to reproduce the same noise in the next run
# numpy.random.seed(123)

abc_dictionary =pattern_tools.load_alphabet()
print("the alphabet is stored in an object of type: {}".format(type(abc_dictionary)))
```

```
# access the first element and get it's size (they are all of same size)
pattern_shape = abc_dictionary['A'].shape
print("letters are patterns of size: {}. Create a network of corresponding size".
↪format(pattern_shape))
# create an instance of the class HopfieldNetwork
hopfield_net = network.HopfieldNetwork(nr_neurons= pattern_shape[0]*pattern_shape[1])

# create a list using Pythons List Comprehension syntax:
pattern_list = [abc_dictionary[key] for key in letter_list ]
plot_tools.plot_pattern_list(pattern_list)

# store the patterns
hopfield_net.store_patterns(pattern_list)

# # create a noisy version of a pattern and use that to initialize the network
noisy_init_state = pattern_tools.get_noisy_copy(abc_dictionary['A'], noise_level=0.2)
hopfield_net.set_state_from_pattern(noisy_init_state)

# from this initial state, let the network dynamics evolve.
states = hopfield_net.run_with_monitoring(nr_steps=4)

# each network state is a vector. reshape it to the same shape used to create the␣
↪patterns.
states_as_patterns = pattern_tools.reshape_patterns(states, pattern_list[0].shape)

# plot the states of the network
plot_tools.plot_state_sequence_and_overlap(
    states_as_patterns, pattern_list, reference_idx=0, suptitle="Network dynamics")
```

Question:

Add the letter 'R' to the letter list and store it in the network. Is the pattern 'A' still a fixed point? Does the overlap between the network state and the reference pattern 'A' always decrease?

Question:

Make a guess of how many letters the network can store. Then create a (small) set of letters. Check if **all** letters of your list are fixed points under the network dynamics. Explain the discrepancy between the network capacity C (computed above) and your observation.

Exercise: Bonus

The implementation of the Hopfield Network in *hopfield_network.network* offers a possibility to provide a custom update function *HopfieldNetwork.set_dynamics_to_user_function()*. Have a look at the source code of *HopfieldNetwork.set_dynamics_sign_sync()* to learn how the update dynamics are implemented. Then try to implement your own function. For example, you could implement an asynchronous update with stochastic neurons.

Type I and type II neuron models

Book chapters

See Chapter 4 and especially Chapter 4 Section 4 for background knowledge on Type I and Type II neuron models.

Python classes

The *neurodynex.neuron_type.neurons* module contains all classes required for this exercise. To get started, call *getting_started* or copy the following code into your Jupyter notebook:

```
%matplotlib inline  # needed in Notebooks, not in Python scripts
import brian2 as b2
import matplotlib.pyplot as plt
import numpy as np
from neurodynex.tools import input_factory, plot_tools, spike_tools
from neurodynex.neuron_type import neurons

# create an input current
input_current = input_factory.get_step_current(50, 150, 1.*b2.ms, 0.5*b2.pA)

# get one instance of class NeuronX and save that object in the variable 'a_neuron_of_
↪type_X'
a_neuron_of_type_X = neurons.NeuronX()  # we do not know if it's type I or II
# simulate it and get the state variables
state_monitor = a_neuron_of_type_X.run(input_current, 200*b2.ms)
# plot state vs. time
neurons.plot_data(state_monitor, title="Neuron of Type X")

# get an instance of class NeuronY
a_neuron_of_type_Y = neurons.NeuronY()  # we do not know if it's type I or II
state_monitor = a_neuron_of_type_Y.run(input_current, 200*b2.ms)
neurons.plot_data(state_monitor, title="Neuron of Type Y")
```

Note: For those who are interested, here is more about classes and inheritance in Python.

Exercise: Probing Type I and Type II neuron models

This exercise deals not only with Python functions, but with python objects. The classes *NeuronX* and *NeuronY* both are neurons, that have different dynamics: **one is Type I and one is Type II**. Finding out which class implements which dynamics is the goal of the exercise.

The types get randomly assigned each time you load the module or you call the function *neurons.neurontype_random_reassignment()*.

Question: Estimating the threshold

What is the threshold current for repetitive firing for *NeuronX* and *NeuronY*?

Exploring various values of `I_amp`, find the range in which the threshold occurs, to a precision of 0.01.

Plot the responses to step current which starts after 100ms (to let the system equilibrate) and lasting at least 1000ms (to detect repetitive firing with a long period). You can do this by modifying the code example given above. Make sure to check the documentation of the functions you use: *input_factory.get_step_current()*, `neuron_type.neurons.run()` and *neuron_type.neurons.plot_data()*.

Already from the voltage response near threshold you might have an idea which is type I or II, but let's investigate further.

Exercise: f-I curves

In this exercise you will write a python script that plots the f-I curve for type I and type II neuron models.

Get firing rates from simulations

We provide you with a function *spike_tools.get_spike_time()* to determine the spike times from a StateMonitor. The following code shows how to use that function. Note that the return value is a Brian Quantity: it has units. If you write code using units, you'll get consistency checks done by Brian.

```
input_current = input_factory.get_step_current(100, 110, b2.ms, 0.5*b2.pA)
state_monitor = a_neuron_of_type_X.run(input_current, ...)
spike_times = spike_tools.get_spike_time(state_monitor, ...)
print(spike_times)
print(type(spike_times))  # it's a Quantity
```

Now **write a new function** (in your own *.py* file or in your Jupyter Notebook) that calculates an estimate of the firing rate. In your function use *spike_tools.get_spike_time()*

```
def get_firing_rate(neuron, input_current, spike_threshold):

    # inject a test current into the neuron and call it's run() function.
    # get the spike times using spike_tools.get_spike_times
    # from the spike times, calculate the firing rate f

    return f
```

Note: To calculate the firing rate, first calculate the inter-spike intervals (time difference between spikes) from the spike times using this elegant indexing idiom

```
isi = st[1:]-st[:-1]
```

Then find the mean isi and take the reciprocal to yield the firing-rate. As these are standard operations, you can expect that someone else has already implemented it. Have a look at the numpy package and look up the functions diff and mean. Once you have implemented your function, you should verify it's correctness: inject a few currents into your neuron, plot the voltage response and compare the plot with the firing rate computed by your function.

Note: You can check your results by calling:

```
spike_tools.pretty_print_spike_train_stats(...)
```

Plot the f-I curve

Now let's use your function `get_firing_rate` to plot an f-vs-I curve for both neuron classes.

Add the following function skeleton to your code and complete it to plot the f-I curve, given the neuron class as an argument:

```
import matplotlib.pyplot as plt
import numpy as np
```

```
def plot_fI_curve(NeuronClass):

    plt.figure()  # new figure

    neuron = NeuronClass()  # instantiate the neuron class

    I = np.arange(0.0,1.1,0.1)  # a range of current inputs
    f = []

    # loop over current values
    for I_amp in I:

        firing_rate = # insert here a call to your function get_firing_rate( ... )

        f.append(firing_rate)

    plt.plot(I, f)
    plt.xlabel('Amplitude of Injecting step current (pA)')
    plt.ylabel('Firing rate (Hz)')
    plt.grid()
    plt.show()
```

- Call your `plot_fI_curve` function with each class `NeuronX` and `NeuronY` as argument.
- Change the `I` range (and reduce the step size) to zoom in near the threshold, and try running it again for both classes.

Which class is Type I and which is Type II? Check your result:

```
print("a_neuron_of_type_X is : {}".format(a_neuron_of_type_X.get_neuron_type()))
print("a_neuron_of_type_Y is : {}".format(a_neuron_of_type_Y.get_neuron_type()))
```

Oja's hebbian learning rule

Book chapters

See Chapter 19 Section 2 on the learning rule of Oja.

Python classes

The *ojas_rule.oja* module contains all code required for this exercise. At the beginning of your exercise solution file, import the contained functions by

```
import neurodynex.ojas_rule.oja as oja
```

You can then simply run the exercise functions by executing, e.g.

```
cloud = oja.make_cloud()  # generate data points
wcourse = oja.learn(cloud)  # learn weights and return timecourse
```

A complete script using these functions could look like this:

```
%matplotlib inline  # used for Jupyter Notebook
import neurodynex.ojas_rule.oja as oja
import matplotlib.pyplot as plt

cloud = oja.make_cloud(n=200, ratio=.3, angle=60)
wcourse = oja.learn(cloud, initial_angle=-20, eta=0.04)
```

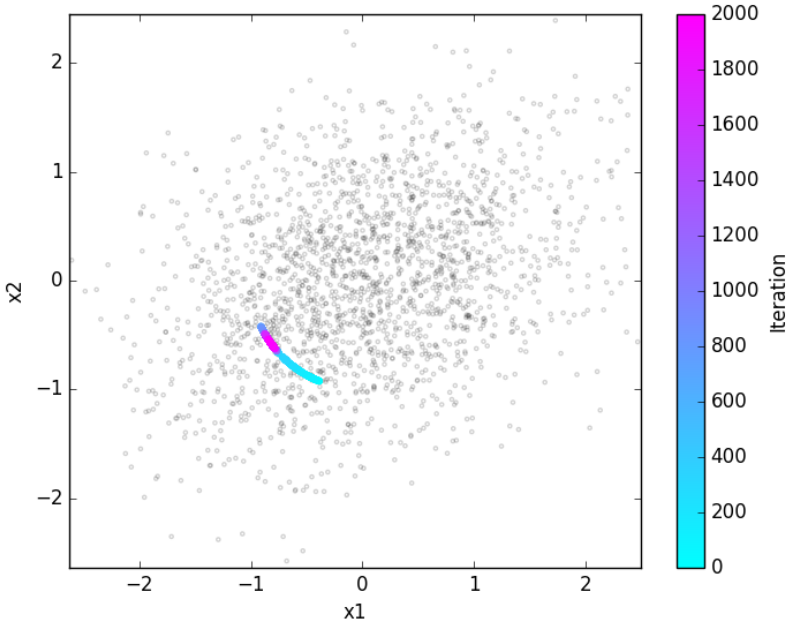


Fig. 1.9: Grey points: Datapoints (two presynaptic firing rates, presented sequentially in random order). Colored points: weight change under Oja's rule.

```
plt.scatter(cloud[:, 0], cloud[:, 1], marker=".", alpha=.2)
plt.plot(wcourse[-1, 0], wcourse[-1, 1], "or", markersize=10)
plt.axis('equal')
plt.figure()
plt.plot(wcourse[:, 0], "g")
plt.plot(wcourse[:, 1], "b")
print("The final weight vector w is: ({},{})".format(wcourse[-1,0],wcourse[-1,1]))
```

Exercise: getting started

The figure below shows the configuration of a neuron learning from the joint input of two presynaptic neurons. Run the above script. Make sure you understand what the functions are doing.

Question: The norm of the weights

- Run the script with a large learning rate eta = 0.2. What do you observe?
- Modify the script: plot the time course of the norm of the weights vector.

Exercise: Circular data

Now we study Oja's rule on a data set which has no correlations. Use the functions *make_cloud* and *learn* to get the timecourse for weights that are learned on a **circular** data cloud (`ratio=1`). Plot the time course of both components of the weight vector. Repeat this many times (*learn* will choose random initial conditions on each run), and plot this into the same plot. Can you explain what happens? Try different learning rates eta.

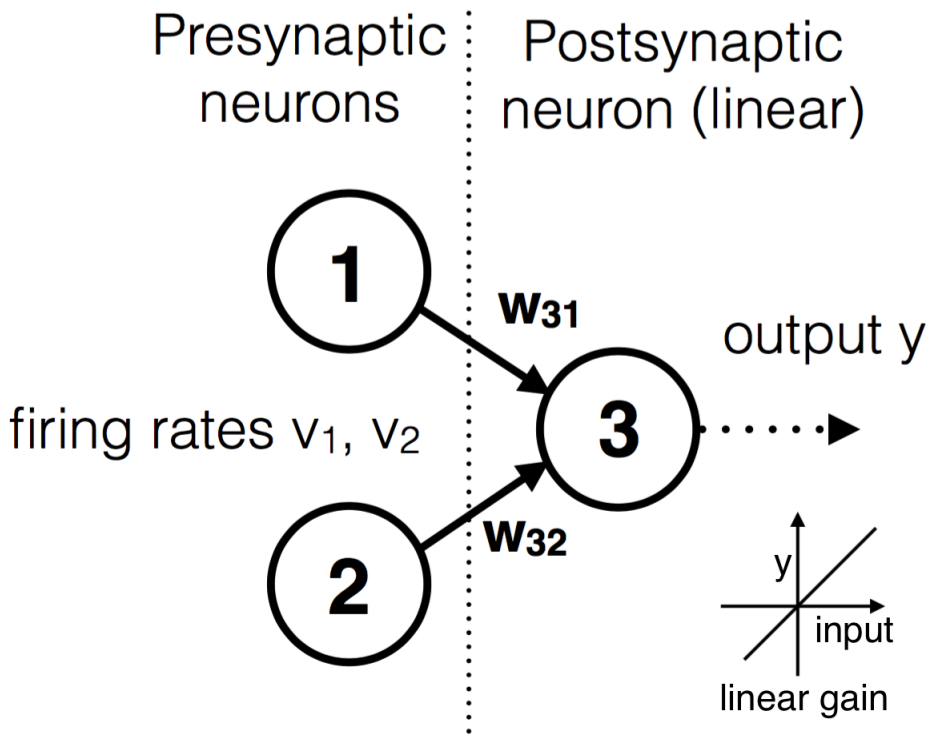


Fig. 1.10: One linear neuron gets input from two presynaptic neurons.

Exercise: What is the neuron leaning?

- Repeat the previous question with an **elongated** elliptic data cloud (e.g. `ratio=0.3`). Again, repeat this several times.
- What difference in terms of learning do you observe with respect to the circular data clouds?
- The "goal" of the neuron is not to change weights, but to produce a meaningful output y. After learning, what does the output y tell about the input?
- Take the final weights [w31, w32], then calculate a single input vector (v1=?, v2=?) that leads to a **maximal** output firing y. Constrain your input to norm([v1,v2]) =1.
- Calculate an input which leads to a **minimal** output firing y.

Exercise: Non-centered data

The above exercises assume that the input activities can be negative (indeed the inputs were always statistically centered). In actual neurons, if we think of their activity as their firing rate, this cannot be less than zero.

Try again the previous exercise, but applying the learning rule on a noncentered data cloud. E.g., use `cloud = (3,5) + oja.make_cloud(n=1000, ratio=.4, angle=-45)`, which centers the data around `(3,5)`. What conclusions can you draw? Can you think of a modification to the learning rule?

Bonus: 3 D

By modifying the source code of the given functions, try to visualize learning from a 3 dimensional time series. Here's an example of a 3D scatter plot: scatter3d

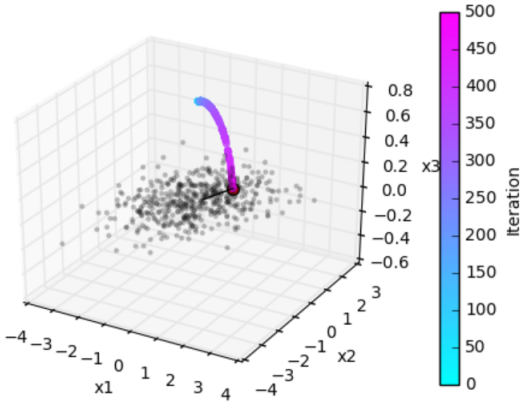


Fig. 1.11: Learning from a 3D input.

Network of LIF neurons (Brunel)

In this exercise we study a well known network of sparsely connected Leaky-Integrate-And-Fire neurons (Brunel, 2000).

Book chapters

The Brunel model is introduced in Chapter 13 Section 4.2 . The network structure is shown in figure 13.6b. Read the section "Synchrony, oscillations, and irregularity" and have a look at Figure 13.7. For this exercise, you can skip the explanations related to the Fokker-Planck equation.

Python classes

The module *brunel_model.LIF_spiking_network* implements a parametrized network. The figure below shows the simulation result using the default configuration.

To get started, call the function *brunel_model.LIF_spiking_network.getting_started()* or copy the following code into a Jupyter notebook.

```
%matplotlib inline
from neurodynex.brunel_model import LIF_spiking_network
from neurodynex.tools import plot_tools
import brian2 as b2

rate_monitor, spike_monitor, voltage_monitor, monitored_spike_idx = LIF_spiking_
↪network.simulate_brunel_network(sim_time=250. * b2.ms)
plot_tools.plot_network_activity(rate_monitor, spike_monitor, voltage_monitor, spike_
↪train_idx_list=monitored_spike_idx, t_min=0.*b2.ms)
```

Note that you can change all parameters of the neuron by using the named parameters of the function *simulate_brunel_network()*. If you do not specify any parameter, the default values are used (see next code block). You can access these variables in your code by prefixing them with the module name (for example LIF_spiking_network.POISSON_INPUT_RATE).

```
# Default parameters of a single LIF neuron:
V_REST = 0. * b2.mV
V_RESET = +10. * b2.mV
FIRING_THRESHOLD = +20. * b2.mV
MEMBRANE_TIME_SCALE = 20. * b2.ms
ABSOLUTE_REFRACTORY_PERIOD = 2.0 * b2.ms
```

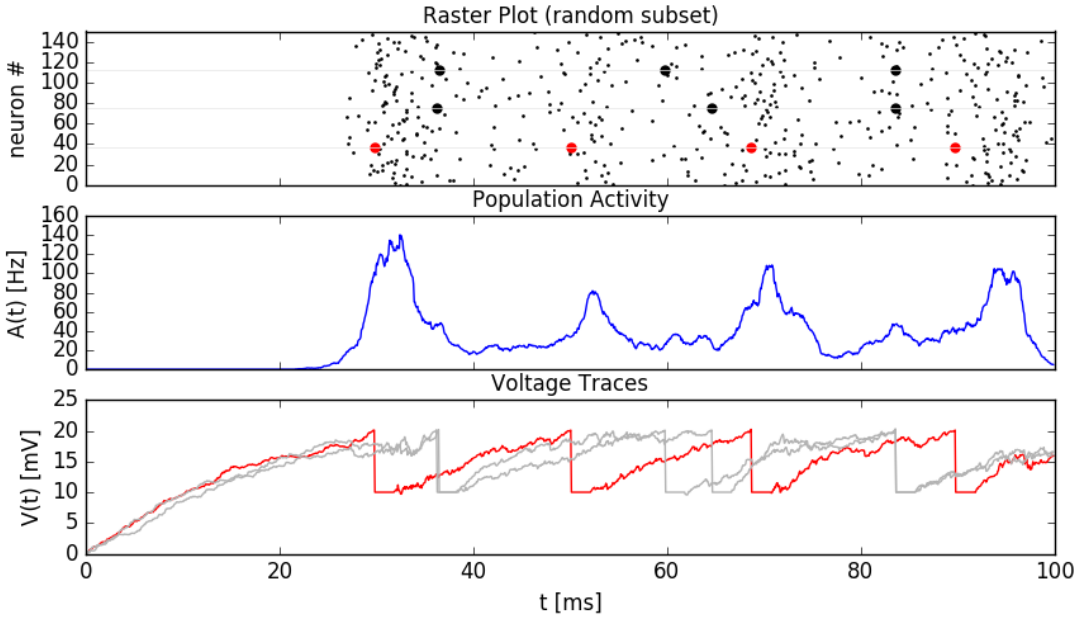


Fig. 1.12: Simulation result. Top: raster plot of 150 randomly selected neurons. Three spike trains are visually highlighted. Middle: time evolution of the population activity A(t). Bottom: Membrane voltage of three neurons. The red color in the top and bottom panels identifies the same neuron.

```
# Default parameters of the network
SYNAPTIC_WEIGHT_W0 = 0.1 * b2.mV  # note: w_ee=w_ie = w0 and = w_ei=w_ii = -g*w0
RELATIVE_INHIBITORY_STRENGTH_G = 4.  # balanced
CONNECTION_PROBABILITY_EPSILON = 0.1
SYNAPTIC_DELAY = 1.5 * b2.ms
POISSON_INPUT_RATE = 12. * b2.Hz
N_POISSON_INPUT = 1000
```

Exercise: model parameters and threshold rate

In the first exercise, we get familiar with the model and parameters. Make sure you have read the book chapter . Then have a look at the documentation of `simulate_brunel_network()`. Note that in our implementation, the number of excitatory presynaptic poisson neurons (input from the external population) is a parameter *N_extern* and thus independent of *CE*.

Question:

- Run the simulation with the default parameters (see code block above). In that default configuration, what values take the variables N_E, N_I, C_E, C_I, w_{EE}, w_{EI}, w_{IE}, and w_{II}? The variables are described in the book and in Fig. 13.6
- What are the units of the weights w?
- The frequency $\nu_{threshold}$ is is the poisson rate of the external population sufficient to drive the neurons in the network to the firing threshold. Using Eq. (1.3), compute $\nu_{threshold}$. You can do this in Python, e.g. use LIF_spiking_network.FIRING_THRESHOLD for u_{thr}, etc.

- Refering to Figure 13.7, left panel, what is the meaning of the value 1 on the y-axis (Input). What is the horizontal dashed line designating? How is it related to u_{thr}?
- Run a simulation for 500ms. Set *poisson_input_rate* to $\nu_{threshold}$. Plot the network activity in the time interval [0ms, 500ms]. Is the network quiet (Q)?
- During the simulation time, what is the average firing rate of a single neuron? You can access the total number of spikes from the Brian2.SpikeMonitor: spike_monitor.num_spikes and the number of neurons in the network from spike_monitor.source.N .

$$\nu_{threshold} = \frac{u_{thr}}{N_{extern} w_0 \tau_m} \tag{1.3}$$

Exercise: Population activity

The network of spiking LIF-neurons shows characteristic population activities. In this exercise we investigate the patterns asynchronous irregular (AI), synchronous regular (SR), fast synchronous irregular (SI fast) and slow synchronous irregular (SI slow).

Question: Network states

- The function `simulate_brunel_network()` gives you three options to vary the input strength (y-axis in figure 13.7, a). What options do you have?
- Which parameter of the function `simulate_brunel_network()` lets you change the relative strength of inhibition (the x-axis in figure 13.7, a)?
- Define a network of 6000 excitatory and 1500 inhibitory neurons. Find the appropriate parameters and simulate the network in the regimes AI, SR, SI-fast and SI-slow. For each of the four configurations, plot the network activity and compute the average firing rate. Run each simulation for at least 1000ms and plot two figures for each simulation: one showing the complete simulation time and one showing only the last ~50ms.
- What is the population activity A(t) in each of the four conditions (in Hz, averaged over the last 200ms of your simulation)?

Question: Interspike interval (ISI) and Coefficient of Variation (CV)

Before answering the questions, make sure you understand the notions ISI and CV. If necessary, read Chapter 7.3.1 .

- What is the CV of a Poisson neuron?
- From the four figures plotted in the previous question, qualitatively interpret the spike trains and the population activity in each of the four regimes:
  - What is the mean firing rate of a single neuron (only a rough estimate).
  - Sketch the ISI histogram. (is it peaked or broad? where's the maximum?)
  - Estimate the CV. (is it <1, <<1, =1, >1 ?)
- Validate your estimates using the functions `spike_tools.get_spike_train_stats()` and `plot_tools.plot_ISI_distribution()`. Use the code block provided here.
- Make sure you understand the code block. Why is the function *.spike_tools.get_spike_train_stats* called with the parameter *window_t_min=100.*b2.ms*?

```
%matplotlib inline
from neurodynex.brunel_model import LIF_spiking_network
from neurodynex.tools import plot_tools, spike_tools
import brian2 as b2

poisson_rate = ??? *b2.Hz
g = ???
CE = ???
simtime = ??? *b2.ms

rate_monitor, spike_monitor, voltage_monitor, monitored_spike_idx = LIF_spiking_
↪network.simulate_brunel_network(N_Excit=CE, poisson_input_rate=poisson_rate, g=g,␣
↪sim_time=simtime)
plot_tools.plot_network_activity(rate_monitor, spike_monitor, voltage_monitor, spike_
↪train_idx_list=monitored_spike_idx, t_min = 0*b2.ms)
plot_tools.plot_network_activity(rate_monitor, spike_monitor, voltage_monitor, spike_
↪train_idx_list=monitored_spike_idx, t_min = simtime - ??? *b2.ms)
spike_stats = spike_tools.get_spike_train_stats(spike_monitor, window_t_min= 100 *b2.
↪ms)
plot_tools.plot_ISI_distribution(spike_stats, hist_nr_bins=100, xlim_max_ISI= ??? *b2.
↪ms)
```

- In the Synchronous Repetitive (SR) state, what is the dominant frequency of the population activity A(t)? Compare this frequency to the firing frequency of a single neuron. You can do this "visually" using the plots created by *plot_tools.plot_network_activity()* or by solving the bonus exercise below.

Exercise: Emergence of Synchronization

The different regimes emerge from from the recurrence and the relative strength of inhibition g. In the absence of recurrent feedback from the network, the network would approach a constant mean activity A(t).

Question:

- Simulate a network of 6000 excitatory and 1500 inhibitory neurons. Set the following parameters: poisson_rate = 14*b2.Hz, g=2.5. In which state is this network?
- What would be the population activity caused by the external input only? We can simulate this. Run a simulation of the same network, but disable the recurrent feedback: simulate_brunel_network(...,w0=0.*b2.mV, w_external = LIF_spiking_network.SYNAPTIC_WEIGHT_W0).
- Explain why the non-recurrent network shows a strong synchronization in the beginning and why this synchronization fades out.
- The non recurrent network is strongly synchronized in the beginning. Is the connected network simply "locked" to this initial synchronization? You can falsify this hypothesis by initializing each neuron in the network with a random vm. Run the simulation with *random_vm_init=True* to see how the synchronization emerges over time.

Bonus: Power Spectrum of the Population Activity

We can get more insights into the statistics of the network activity by analysing the power spectrum of the spike trains and the population activity. The four regimes (SR, AI, SI fast, SI slow) are characterized by *two* properties: the regularity/irregularity of individual neuron's spike trains *and* the stationary/oscillatory pattern of the population activity A(t). We transform the spike trains and A(t) into the frequency domain to identify regularities.

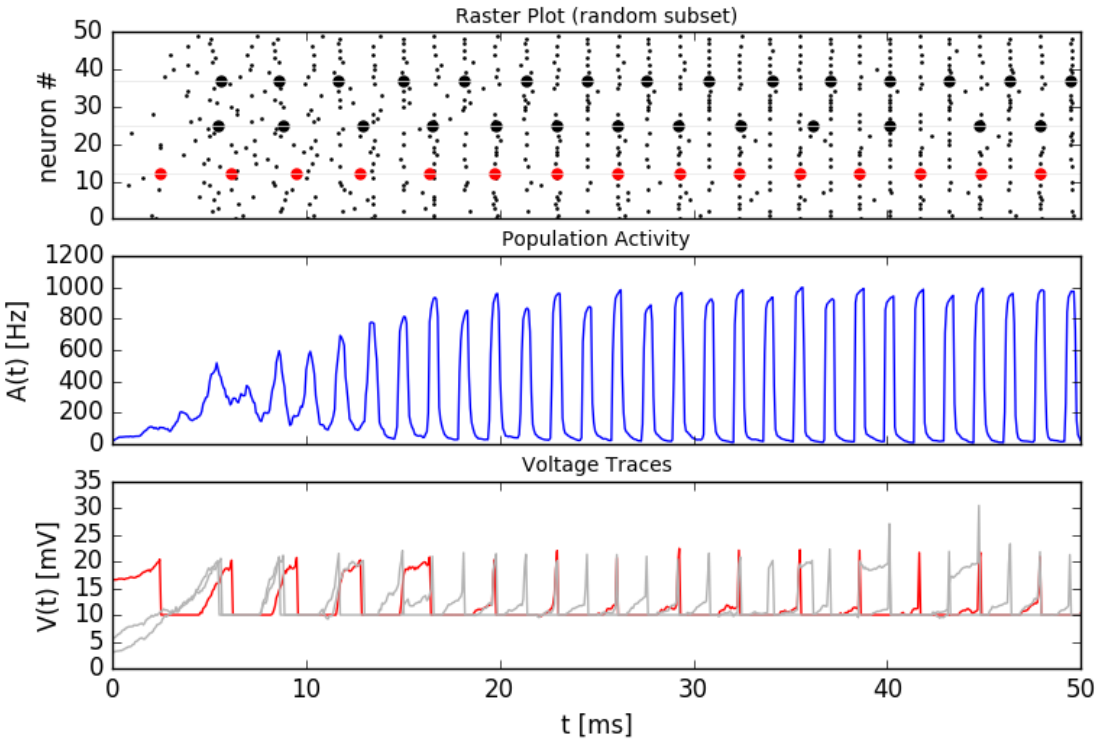


Fig. 1.13: Simulation of a network with random v_m initialization. The synchronization of the neurons is not a residue of shared initial conditions, but emerges over time.

Question: Sampling the Population Activity

- When analysing the population activity A(t), what is the lowest/highest frequency we are interested?

The highest frequency f_{max} one can resolve from the time series A(t) is determined by Δt. Even if we are not interested in very high frequencies, we should not increase Δt (too much) because it may affect the accuracy of the simulation.

The lowest frequency Δf is determined by the signal length $T_{Simulation}$. We could therefore decrease the simulation duration if we accept decreasing the resolution in the frequency domain. But there is another option: We still use a "too long" simulation time $T_{Simulation}$ but then split the RateMonitor.rate signal into k chunks of duration T_{Signal}. We can then average the power across the k repetitions. This is what the function *spike_tools.get_population_activity_power_spectrum()* does - we just have to get the parameters first:

- Given the values $\Delta f = 5Hz, \Delta t = 0.1ms, T_{init} = 100ms, k = 5$, compute T_{Signal} and $T_{Simulation}$.

$$\begin{aligned} f_{max} &= \frac{f_{Sampling}}{2} = \frac{1}{2 \cdot \Delta t} \\ N \cdot \Delta t &= T_{Signal} \\ 2 \cdot f_{max} &= N \cdot \Delta f \\ T_{Simulation} &= k \cdot T_{Signal} + T_{init}; k \in N \end{aligned} \tag{1.4}$$

$f_{Sampling}$: sampling frequency of the signal; f_{max}: highest frequency component; Δf: frequency resolution in fourier domain = lowest frequency component; T_{Signal} length of the signal; Δt: temporal resolution of the signal; N: Number of samples (same in time- and frequency- domain) $T_{Simulation}$: simulation time; k: k repetitions of the signal; T_{init}: initial part of the simulation (not used for data analysis);

Question: Sampling a Single Neuron Spike Train

- The sampling of the individual neuron's spike train is different because in that case, the signal is given as a list of timestamps (SpikeMonitor.spike_trains) and needs to be transformed into a binary vector. This is done inside the function *spike_tools.get_averaged_single_neuron_power_spectrum()*. Read the doc to learn how to control the sampling rate.
- The firing rate of a single neuron can be very low and very different from one neuron to another. For that reason, we do not split the spike train into k realizations but we analyse the full spike train ($T_{Simulation} - T_{init}$). From the simulation, we get many (CE+CI) spike trains and we can average across a subset of neurons. Check the doc of *spike_tools.get_averaged_single_neuron_power_spectrum()* to learn how to control the number of neurons of this subset.

Question: Single Neuron activity vs. Population Activity

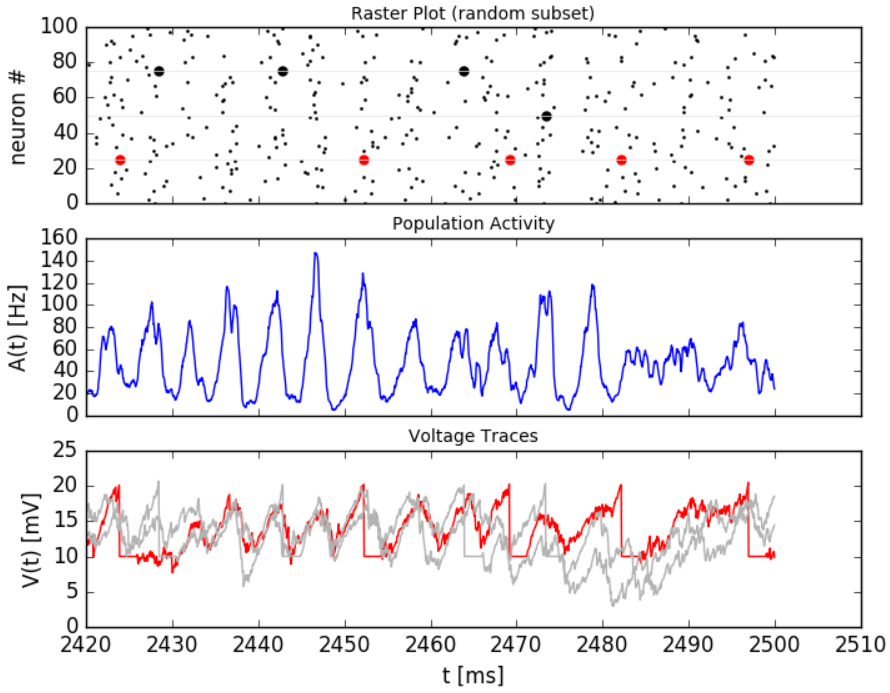
We can now compute and plot the power spectrum.

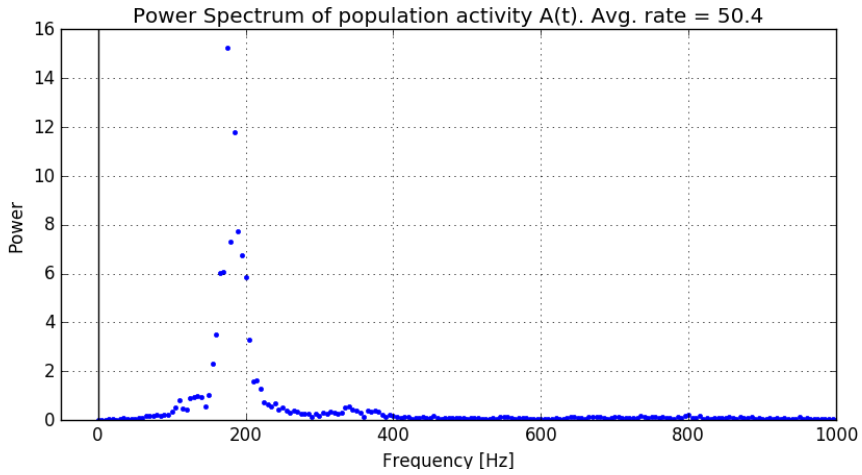
- For each network states SR, AI, SI fast, SI slow, find the parameters, then compute and plot the power spectrum using the script given here. Make sure you understand the script and read the documentation of the functions *spike_tools.get_averaged_single_neuron_power_spectrum()*, *plot_tools.plot_spike_train_power_spectrum()*, *spike_tools.get_population_activity_power_spectrum()*, and *plot_tools.plot_population_activity_power_spectrum()*.
- Discuss power spectra of the states SR, AI, SI fast and SI slow. Compare the individual neuron's spike train powers to the averaged power spectrum and to the power spectrum of A(t).

```
%matplotlib inline
from neurodynex.brunel_model import LIF_spiking_network
from neurodynex.tools import plot_tools, spike_tools
import brian2 as b2

# Specify the parameters of the desired network state (e.g. SI fast)
poisson_rate = ??? *b2.Hz
g = ???
CE = ???

# Specify the signal and simulation properties:
delta_t = ??? * b2.ms
delta_f = ??? * b2.Hz
T_init = ??? * b2.ms
k = ???

# compute the remaining values:
f_max = ???
N_samples = ???
T_signal = ???
T_sim = k * T_signal + T_init

# replace the ??? by appropriate values:

print("Start simulation. T_sim={}, T_signal={}, N_samples={}".format(T_sim, T_signal,␣
↪N_samples))
b2.defaultclock.dt = delta_t
# for technical reason (solves rounding issues), we add a few extra samples:
stime = T_sim + (10 + k) * b2.defaultclock.dt
rate_monitor, spike_monitor, voltage_monitor, monitored_spike_idx = \
    LIF_spiking_network.simulate_brunel_network(
        N_Excit=CE, poisson_input_rate=poisson_rate, g=g, sim_time=stime)

plot_tools.plot_network_activity(rate_monitor, spike_monitor, voltage_monitor,
                                 spike_train_idx_list=monitored_spike_idx, t_min=0*b2.
↪ms)
plot_tools.plot_network_activity(rate_monitor, spike_monitor, voltage_monitor,
                                 spike_train_idx_list=monitored_spike_idx, t_min=T_
↪sim - ??? *b2.ms)
spike_stats = spike_tools.get_spike_train_stats(spike_monitor, window_t_min= T_init)
plot_tools.plot_ISI_distribution(spike_stats, hist_nr_bins= ???, xlim_max_ISI= ???␣
↪*b2.ms)

#  Power Spectrum
pop_freqs, pop_ps, average_population_rate = \
    spike_tools.get_population_activity_power_spectrum(
        rate_monitor, delta_f, k, T_init)
plot_tools.plot_population_activity_power_spectrum(pop_freqs, pop_ps, ??? *b2.Hz,␣
↪average_population_rate)
freq, mean_ps, all_ps, mean_firing_rate, all_mean_firing_freqs = \
    spike_tools.get_averaged_single_neuron_power_spectrum(
        spike_monitor, sampling_frequency=1./delta_t, window_t_min= T_init,
        window_t_max=T_sim, nr_neurons_average= ??? )
plot_tools.plot_spike_train_power_spectrum(freq, mean_ps, all_ps, max_freq= ??? * b2.
↪Hz,
                                           mean_firing_freqs_per_neuron=all_mean_
↪firing_freqs,
                                           nr_highlighted_neurons=2)
```

```
print("done")
```

The figures below show the type of analysis you can do with this script. The first figure shows the last 80ms of a network simulation. The second figure the power spectrum of the population activity A(t) and the third figure shows the power spectrum of single neurons (individual neurons and averaged across neurons). Note the qualitative difference between the spectral density of the population and that of the individual neurons.





Spatial Working Memory (Compte et. al.)

In this exercise we study a model of spatial working memory. The model has been introduced by Compte et. al. [1]. The parameters used here differ from the original paper. They are changed such that we can still study some effects

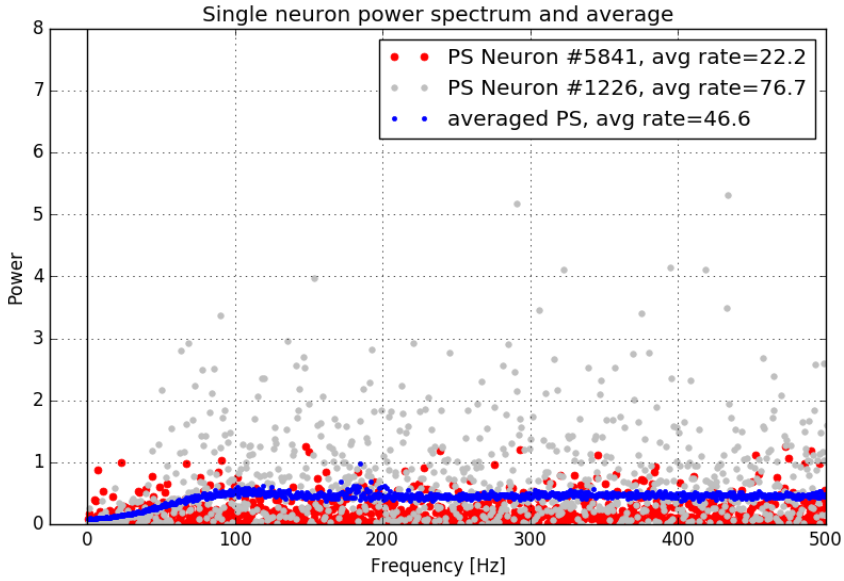


Fig. 1.14: Single neurons (red, grey) fire irregularly **(I)** while the population activity oscillates **(S)**.

while simulating a small network.

Figure 18.4 in chapter 18.1 shows the kind of ring model we are studying here.

Book chapters

Read the introduction of chapter 18, Cortical field models for perceptions and the chapters 18.1, 18.2 and 18.3 . Figure 18.4 in chapter 18.1 shows the kind of ring model we are studying here.

If you have access to a scientific library, you may also want to read the original publication [1].

Python classes

The module *working_memory_network.wm_model* implements a working memory circuit adapted from [1, 2]. To get started, call the function *working_memory_network.wm_model.getting_started()* or copy the following code into a Jupyter notebook.

```
%matplotlib inline
from neurodynex.working_memory_network import wm_model
from neurodynex.tools import plot_tools
import brian2 as b2

wm_model.getting_started()
```

Exercise: Spontanous bump formation

We study the structure and activity of the following network.

Question: External poisson population

Parameters that are not explicitly specified are set to default values. Read the documentation of the function *working_memory_network.wm_model.simulate_wm()* to answer the following questions:

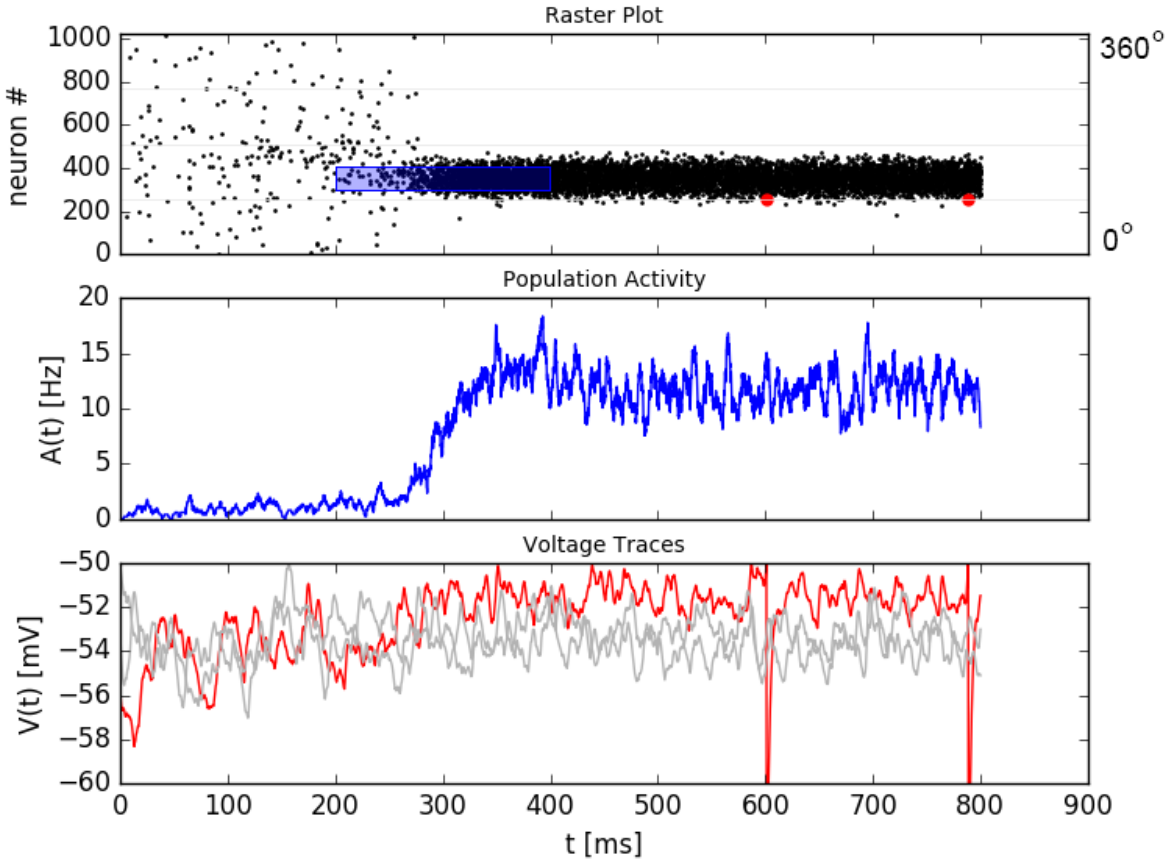


Fig. 1.15: *Top:* A weak stimulus, centered at 120deg, is applied to a subset of the excitatory population from t=200ms to t=400ms (blue box in top panel). This creates an activity bump in the excitatory subpopulation. The activity sustains after the end of the stimulation. The active neurons have a preferred direction close to the stimulus location. *Middle:* The population activity increases over time when the stimulus is applied. *Bottom:* Voltage traces for three selected neurons. The spikes of the red neuron are visible in the top and bottom panel.

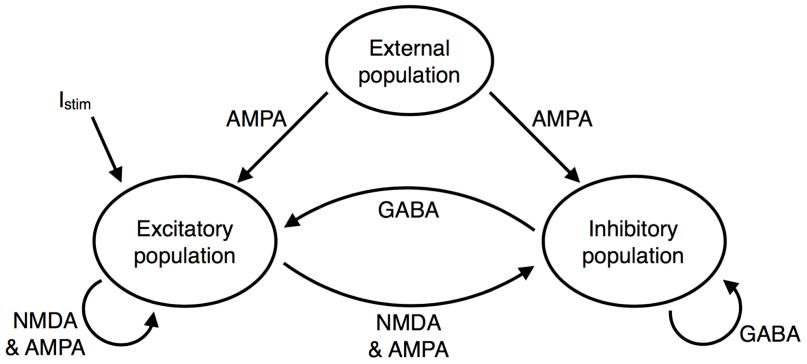


Fig. 1.16: Network structure. Look at Figure 18.4 in chapter 18.1 to see how the excitatory population is spatially arranged on a ring and has a specific connectivity profile. In our implementation, every excitatory neuron receives unstructured input from all inhibitory neurons and structured input from all excitatory neurons. The inhibitory neurons receive unstructured input from all excitatory and all inhibitory neurons.

- By default, how many neurons are in the external poisson population?
- Using the default parameters, what is the average number of spikes/second an excitatory neuron receives from the external population?

From the documentation, follow the 'source' link to go to the implementation of *simulate_wm()*. Answer the following questions about the external poisson population:

- We use the Brian2 PoissonInput to implement the external population. Which post-synaptic variable is targeted by a presynaptic (poisson) spike?
- The dynamics of that variable are defined in the equations `excit_lif_dynamics` (still in the source code of simulate_wm). What is the time-scale of that variable (in milliseconds)?

Question: Unstructured input

Run the following code to simulate a network that receives unstructured poisson input.

```
%matplotlib inline
import brian2 as b2
from neurodynex.working_memory_network import wm_model
from neurodynex.tools import plot_tools

rate_monitor_excit, spike_monitor_excit, voltage_monitor_excit, idx_monitored_neurons_
↪excit, rate_monitor_inhib, spike_monitor_inhib, voltage_monitor_inhib, idx_
↪monitored_neurons_inhib, w_profile = wm_model.simulate_wm(sim_time=800. * b2.ms,␣
↪poisson_firing_rate=1.3 * b2.Hz, sigma_weight_profile=20., Jpos_excit2excit=1.6)
plot_tools.plot_network_activity(rate_monitor_excit, spike_monitor_excit, voltage_
↪monitor_excit, t_min=0. * b2.ms)
```

- Without coding, from the plot: What is the population activity (mean firing rate) of the excitatory population at different points in time?
- Change the firing rate of the external population to 2.2Hz. What do you observe?
- Run the simulation a few times with r_ext = 2.2 Hz. Describe your observations.

Question: Weight profile

The function *simulate_wm()* takes two parameters to define the weight profile: `sigma_weight_profile` and `Jpos_excit2excit`. After the simulation you can access the return value weight_profile_45. This array contains the synaptic weights between the one postsynaptic neuron whose preferred direction is 45deg and all other (presynaptic) neurons. Our choice of 45deg is arbitrary, the profile for other neurons are shifted versions of this one.

- Run the following code to simulate the network.
- Increase `Jpos_excit2excit`. How does the weight profile change (look at short and long ranges)?
- Simulate with `Jpos_excit2excit` = 2.3. What do you observe?
- How does the weight profile change with the parameter `sigma_weight_profile`? How does the bump change with this parameter?

```
%matplotlib inline
import brian2 as b2
from neurodynex.working_memory_network import wm_model
from neurodynex.tools import plot_tools
import matplotlib.pyplot as plt
```

```
rate_monitor_excit, spike_monitor_excit, voltage_monitor_excit, idx_monitored_neurons_
↪excit, rate_monitor_inhib, spike_monitor_inhib, voltage_monitor_inhib, idx_
↪monitored_neurons_inhib, weight_profile_45 = wm_model.simulate_wm(sim_time=800. *␣
↪b2.ms, poisson_firing_rate=1.3 * b2.Hz, sigma_weight_profile=20., Jpos_
↪excit2excit=1.6)
plot_tools.plot_network_activity(rate_monitor_excit, spike_monitor_excit, voltage_
↪monitor_excit, t_min=0. * b2.ms)

plt.figure()
plt.plot(weight_profile_45)
```

Exercise: Network response to a structured input stimulus

We now apply a stimulus to a subset of the excitatory population. The network has the property of integrating input over time and keep a memory of the input stimulus. Using the following code, you can run a simulation with a weak input stimulus.

```
import brian2 as b2
from neurodynex.working_memory_network import wm_model
from neurodynex.tools import plot_tools
import matplotlib.pyplot as plt


rate_monitor_excit, spike_monitor_excit, voltage_monitor_excit, idx_monitored_neurons_
↪excit, rate_monitor_inhib, spike_monitor_inhib, voltage_monitor_inhib, idx_
↪monitored_neurons_inhib, w_profile = wm_model.simulate_wm(stimulus_center_deg=120,␣
↪stimulus_width_deg=30, stimulus_strength=.06 * b2.namp, t_stimulus_start=100 * b2.
↪ms, t_stimulus_duration=200 * b2.ms, sim_time=500. * b2.ms)
fig, ax_raster, ax_rate, ax_voltage = plot_tools.plot_network_activity(rate_monitor_
↪excit, spike_monitor_excit, voltage_monitor_excit, t_min=0. * b2.ms)
plt.show()
```

Question: Integration of input

Run the stimulation given above. Then answer the following questions qualitatively (by eye, from the raster plot)

- At which time can you identify a change in the population activity? How does that compare to the time when the stimulus is applied?
- What is the population activity at the end of the simulation?
- For the time point t=400ms, sketch the firing rate across the population (neuron index on the x-axis, per-neuron firing rate on the y-axis).
- Increase the stimulus strength to 0.5namp. What happens when the stimulus stops?
- Increase the stimulus width to 60deg (stimulus_strength=0.1 * b2.namp, stimulus center = 120deg). How does the bump shape change?

Question: Role of the inhibitory population

We can remove the inhibitory population by setting its size to the minimal size N_inhibitory = 1. If we also deactivate the external input we can study the effect of the recurrent weights within the excitatory population:

Parameters: N_inhibitory = 1, stimulus_strength=0.65 * b2.namp, t_stimulus_start=5 * b2.ms, t_stimulus_duration=25 * b2.ms, sim_time=80. * b2.ms

- Before running the simulation: What do you expect to see?
- Run the simulation with the given parameters. Describe your observations.

Now run again a "normal" simulation:

```
rate_monitor_excit, spike_monitor_excit, voltage_monitor_excit, idx_monitored_neurons_
↪excit, rate_monitor_inhib, spike_monitor_inhib, voltage_monitor_inhib, idx_
↪monitored_neurons_inhib, w_profile = wm_model.simulate_wm(stimulus_center_deg=120,␣
↪stimulus_width_deg=30, stimulus_strength=.06 * b2.namp, t_stimulus_start=100 * b2.
↪ms, t_stimulus_duration=200 * b2.ms, sim_time=500. * b2.ms)
```

- As for the excitatory population, plot the raster, population activity and voltage traces for the inhibitory population.
- What is the role of the inhibitory population?

Exercise: Decoding the population activity into a population vector

In the raster plot above we see that the population of spiking neurons keeps a memory of the stimulus. In this exercise we decode the population vector (i.e. the angle `theta` stored in the working memory) from the spiking activity. The population vector is defined as the **weighted (by spike counts) mean of the preferred directions of the neurons**. We access the data in the Brian2 SpikeMonitor returned by the simulation to calculate the population vector. Read the Brian2 documentation to see how one can access spike trains. Then implement the readout following the steps given here:

Mapping the neuron index onto its preferred direction

Write a function **get_orientation(idx_list, N)** which maps a vector of neuron indices `idx_list` onto a vector of preferred directions. `idx_list` is the subset of `k` monitored neurons. The second parameter `N` is the total number of neurons in the excitatory population. Verify your implementation by calling the function with the following example input:

```
> get_orientation([0, 1, 5, 10], 11)
> [16.36, 49.09, 180.0, 343.64]
>
> get_orientation([0, 1, 499, 500, 999], 1000)
> [0.18, 0.54, 179.82, 180.18, 359.82]
```

Extracting spikes from the spike monitor

The population vector `theta` changes over time due to drift and diffusion which is why we are interested in `theta(t)`. As we are dealing with spikes (discrete point events), and a small number of neurons, we have to average the population activity over some time window around t, [t_min=t - t_window_width/2, t_max =t + t_window_width/2], to get an estimate of `theta(t)`.

Write a function `get_spike_count(spike_monitor, spike_index_list, t_min, t_max)` which returns an array of spike counts (one value for each neuron in `spike_index_list`). Be careful about the indexing: `spike_index_list` is a list of `k` neuron indices in [0, N-1] while the returned array `spike_count_list` is of length `k`.

The parameter `spike_monitor` is the spike_monitor_excit returned by the function *`simulate_wm()`*. The following pseudo-code and fragments are useful to implement `get_spike_count`:

```
def get_spike_count(spike_monitor, spike_index_list, t_min, t_max):
    nr_neurons = len(spike_index_list)
    spike_count_list = numpy.zeros(nr_neurons)
    spike_trains = spike_monitor.spike_trains()
    ...
    # loop over the list of neurons and get the spikes within the time window:
        (spike_trains[i]>=t_min) & (spike_trains[i]<(t_max))  # try sum(list of␣
↪booleans)
    ...
return spike_count_list
```

Do a plausibility check of your implementation: In one of the previous questions you have sketched the firing rates across the population at t=400ms. Use `get_spike_count` to plot the profile. Compare to your sketch. You can use the following code block. It's assumed you have run a simulation and the two variables `spike_monitor_excit` and `idx_monitored_neurons_excit` are defined. Then play with the `t_window` parameter to get an intuition for 'good' values.

```
import matplotlib.pyplot as plt

t = 400*b2.ms  # time point of interest
t_window = 10*b2.ms # width of the window over which the average is taken

t_min = t-t_window/2
t_max = t+t_window/2
spike_counts = get_spike_count(spike_monitor_excit, idx_monitored_neurons_excit, t_
↪min, t_max)
spike_rates = spike_counts/(t_max-t_min)/b2.second
plt.plot(spike_rates, ".b")
plt.title("Bump profile in the time interval[{},{}]".format(t_min, t_max))
plt.xlabel("Neuron index")
plt.ylabel("Spike rate [Hz]")
```

Computing the population vector

- Combine the two previous functions to calculate theta(t). For our purpose, it is sufficient to calculate a weighted mean of preferred directions. It is not necessary to correctly decode an angle close to 0deg = 360deg (You can stimulate the network at 350deg to see the problem).
- Run a simulation and decode the population vector at the time when the **stimulation** ends. You should get a value close to the stimulus location.
- Pack the calculation of theta(t) into a function `get_theta_time_series` which takes an additional parameter `t_snapshots` (an array of time points at which you want to decode the population vector). `get_theta_time_series` loops over all t_snapshots and calls `get_spike_count`. Use your function to readout and visualize the evolution of theta. You can take some inspiration from the following code fragment:

```
# Example how to create an array of timestamps spaced by snapshot_interval in the␣
↪interval of interest.
t_snapshots = range(
    int(math.floor((t_stimulus_start+t_stimulus_duration)/b2.ms)),  # lower bound
    int(math.floor((t_sim-t_window_width/2)/b2.ms)),  # Subtract half window. Avoids␣
↪an out-of-bound error later.
    int(round(snapshot_interval/b2.ms))  # spacing between time stamps
    )*b2.ms
```

```
# how your function get_theta_time_series could be called:
theta_ts = get_theta_time_series(spike_monitor, idx_monitored_neurons, t_snapshots, t_
↪window_width)

# plot theta vs time using pyplot
import matplotlib.pyplot as plt
plt.plot(t_snapshots/b2.ms, theta_ts)
```

Exercise: Visualize the diffusion of the population vector

As mentioned above, the population vector changes over time due to drift and diffusion. In our implementation, because of homogeneous network properties (equal parameters, equal weights, shared presynaptic neurons) the drift is zero.

Use your functions developed in the previous questions to study the diffusion of the population vector:

- Simulate a network of size `N_excitatory` = 2048. Apply a stimulus from t=100ms to t=300ms. Plot theta(t). *Note that when you increase the size of the excitatory population you also have to increase the inhibitory population and the weights (''N_inhibitory" and ''weight_scaling_factor''). When doubling the number of presynaptic neurons, you have to scale the weights by 0.5 to keep the total synaptic input the same.*
- Repeat the simulation at least 3 times. Plot each time series theta(t) into the same figure.
- Change the size of the network to `N_excitatory` = 512 and redo the previous steps.
- Discuss your observations.

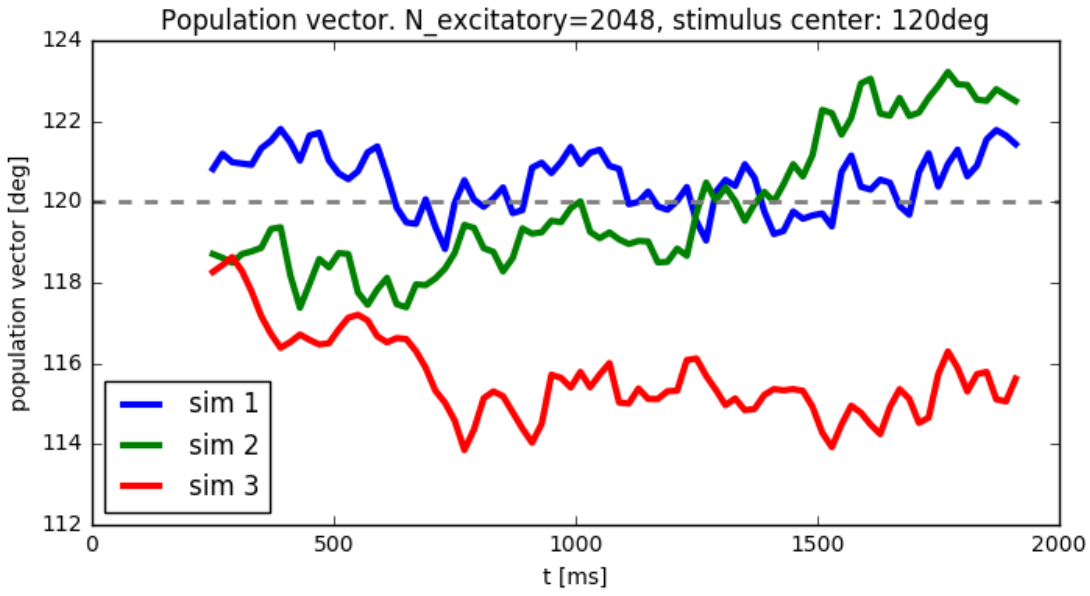


Fig. 1.17: Diffusion of the population vector for three different simulations.

Reading exercise: slow and fast channels

The working memory circuit we study in this exercise combines three different receptors: NMDA and AMPA at excitatory synapses, and GABA at inhibitory synapses. A crucial element for this circuit is the slow dynamics of the

NMDA receptor. Read the chapters 3.1 Synapses and look at Figure 3.2 to understand the dynamics of the receptors.

Question:

The dynamics of the NMDA receptor are implemented in the function `simulate_wm()`. Look for the equations `excit_lif_dynamics` in the source code.

- In the model used here, what is the timescale (in milliseconds) of the fast rise? What is the timescale of the slow decay?

References

[1] Compte, A., Brunel, N., Goldman-Rakic, P. S., & Wang, X. J. (2000). Synaptic mechanisms and network dynamics underlying spatial working memory in a cortical network model. Cerebral Cortex, 10(9), 910-923.

[2] Parts of this exercise and parts of the implementation are inspired by material from *Stanford University, BIOE 332: Large-Scale Neural Modeling, Kwabena Boahen & Tatiana Engel, 2013*, online available.

Perceptual Decision Making (Wong & Wang)

In this exercise we study decision making in a network of competing populations of spiking neurons. The network has been proposed by Wong and Wang in 2006 [1] as a model of decision making in a visual motion detection task. The decision making task and the network are described in the book and in the original publication (see *References* [1]).

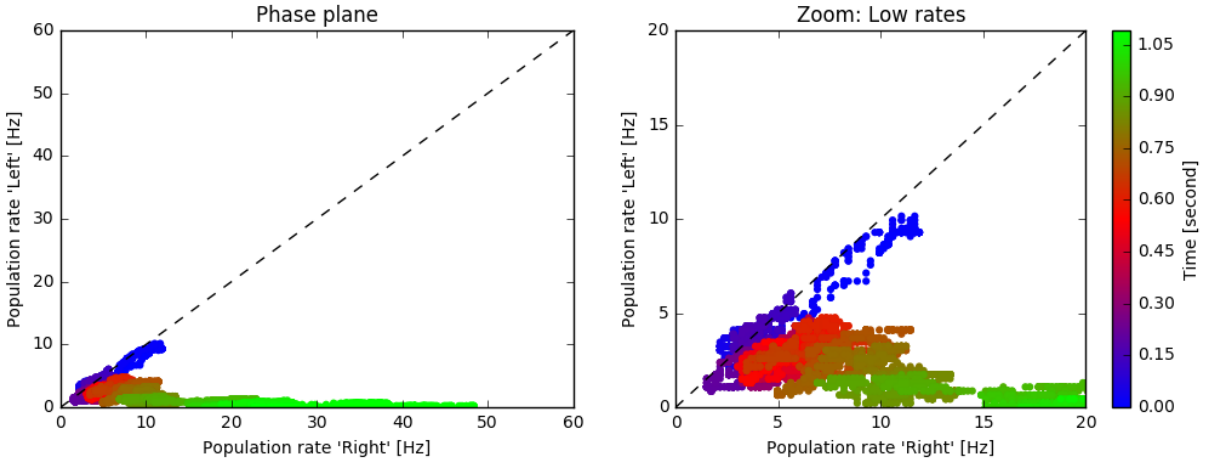


Fig. 1.18: Decision Space. Each point represents the firing rates of the two subpopulations "Left" and "Right" at a given point in time (averaged over a short time window). The color encodes time. In this example, the decision "Right" is made after about 900 milliseconds.

To get a better understanding of the network dynamics, we recommend to solve the exercise *Spatial Working Memory (Compte et. al.)*.

The parameters of our implementation differ from the original paper. In particular, the default network simulates only 480 spiking neurons which leads to relatively short simulation time even on less powerful computers.

Book chapters

Read the introduction of chapter 16, Competing populations and decision making. To understand the mechanism of decision making in a network, read 16.2, Competition through common inhibition.

If you have access to a scientific library, you may also want to read the original publication, *References* [1].

Python classes

The module `competing_populations.decision_making` implements the network adapted from *References* [1, 2]. To get started, call the function `competing_populations.decision_making.getting_started()` or copy the following code into a Jupyter notebook.

```
%matplotlib inline
from neurodynex.competing_populations import decision_making

decision_making.getting_started()
```

Exercise: The network implementation

Before we can analyse the decision making process and the simulation results, we first need to understand the structure of the network and how we can access the state variables of the respective subpopulations.

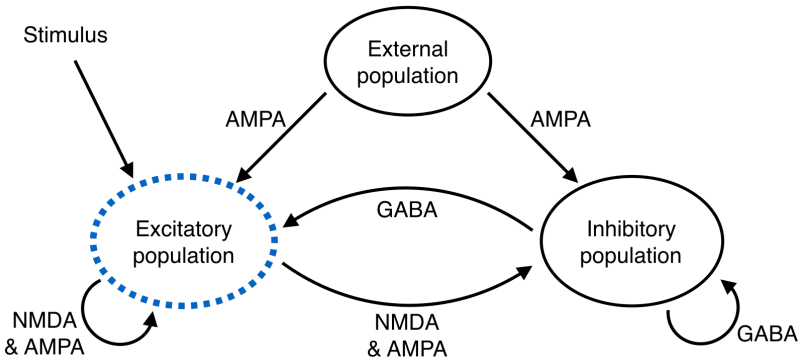


Fig. 1.19: Network structure. The excitatory population is divided into three subpopulations, shown in the next figure.

Question: Understanding Brian2 Monitors

The network shown in the figure above is implemented in Brian2 in the function `competing_populations.decision_making.sim_decision_making_network()`. Each subpopulation is a Brian2 NeuronGroup. Look at the source code of the function `sim_decision_making_network()` to answer the following questions:

- For each of the four subpopulations, find the variable name of the corresponding NeuronGroup.
- Each NeuronGroup is monitored with a PopulationRateMonitor, a SpikeMonitor, and a StateMonitor. Find the variable names for those monitors. Have a look at the Brian2 documentation if you are not familiar with the concept of monitors.
- Which state variable of the neurons is recorded by the StateMonitor?

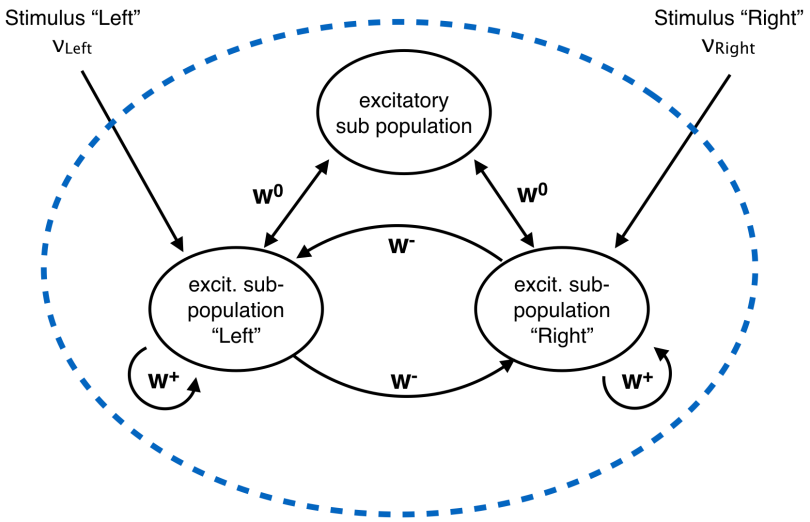


Fig. 1.20: Structure within the excitatory population. The “Left” and “Right” subpopulations have strong recurrent weights ($w^+ > w^0$) and weak projections to the other ($w^- < w^0$). All neurons receive a poisson input from an external source. Additionally, the neurons in the “Left” subpopulation receive poisson input with some rate ν_{Left}; the “Right” subpopulation receives a poisson input with a different rate ν_{right}.

Question: Accessing a dictionary to plot the population rates

The monitors are returned in a Python dictionary providing access to objects by name. Read the Python documentation and look at the code block below or the function *competing_populations.decision_making.getting_started()* to learn how dictionaries are used.

- Extend the following code block to include plots for all four subpopulations.
- Run the simulation for 800ms. What are the “typical” population rates of the four populations towards the end of the simulation? (In case the network did not decide, run the simulation again).
- Without running the simulation again, but by using the same `results` dictionary, plot the rates using different values for `avg_window_width`.
- Interpret the effect of a very short and a very long averaging window.
- Find a value `avg_window_width` for which the population activity plot gives meaningful rates.

```
import brian2 as b2
from neurodynex.tools import plot_tools
from neurodynex.competing_populations import decision_making
import matplotlib.pyplot as plt

results = decision_making.sim_decision_making_network(t_stimulus_start= 50.␣
↪* b2.ms,
                                                      coherence_level=-0.6,␣
↪max_sim_time=1000. * b2.ms)
plot_tools.plot_network_activity(results["rate_monitor_A"], results["spike_
↪monitor_A"],
                                 results["voltage_monitor_A"], t_min=0. * b2.
↪ms, avg_window_width=2. * b2.ms,
                                 sup_title="Left")
plot_tools.plot_network_activity(results["rate_monitor_B"], results["spike_
↪monitor_B"],
```

```
                                    results["voltage_monitor_B"], t_min=0. * b2.
↪ms, avg_window_width=2. * b2.ms,
                                    sup_title="Right")
plt.show()
```

Remark: The parameter `avg_window_width` is passed to the function PopulationRateMonitor.smooth_rate() . This function is useful to solve one of the next exercises.

```
avg_window_width = 123*b2.ms
sr = results["rate_monitor_A"].smooth_rate(window="flat", width=avg_window_width)/b2.
↪Hz
```

Exercise: Stimulating the decision making circuit

The input stimulus is implemented by two inhomogenous Poisson processes: The subpopulation "Left" and "Right" receive input from two different PoissonGroups (see Figure "Network Structure"). The input has a `coherence level c` and is noisy. We have implemented this in the following way: every 70ms, the firing rates ν_{left} and ν_{right} of each of the two PoissonGroups are drawn from a normal distribution:

$$\nu_{left} \sim \mathcal{N}(\mu_{left}, \sigma^2)$$

$$\nu_{right} \sim \mathcal{N}(\mu_{right}, \sigma^2)$$

$$\mu_{left} = \mu_0 * (0.5 + 0.5c)$$

$$\mu_{right} = \mu_0 * (0.5 - 0.5c)$$

$$c \in [-1, +1]$$

The coherence level `c`, the maximum mean μ_0 and the standard deviation σ are parameters of *sim_decision_making_network()*.

Question: Coherence Level

- From the equation above, express the difference $\mu_{left} - \mu_{right}$ in terms of μ_0 and c.
- Find the distribution of the difference $\nu_{left} - \nu_{right}$. Hint: the difference of two Gaussian distributions is another Gaussian distribution.

Now look at the documentation of the function *sim_decision_making_network()* and find the default values of μ_0 and σ. Using those values, answer the following questions:

- What are the mean firing rates (in Hz) μ_{left} and μ_{right} for the coherence level c= -0.2?
- For c= -0.2, how does the difference $\mu_{left} - \mu_{right}$ compare to the variance of $\nu_{left} - \nu_{right}$.

Question: Input stimuli with different coherence levels

Run a few simulations with `c=-0.1` and `c=+0.6`. Plot the network activity.

- Does the network always make the correct decision?

- Look at the population rates and estimate how long it takes the network to make a decision.

Exercise: Decision Space

We can visualize the dynamics of the decision making process by plotting the activities of the two subpopulations "Left" / "Right" in a phase plane (see figure at the top of this page). Such a phase plane of competing states is also known as the *Decision Space*. A discussion of the decision making process in the decision space is out of the scope of this exercise but we refer to *References* [1].

Question: Plotting the Decision Space

- Write a function that takes two RateMonitors and plots the *Decision Space*.
- Add a parameter `avg_window_width` to your function (same semantics as in the exercise above). Run a few simulations and plot the phase plane for different values of `avg_window_width`.
- We can use a rate threshold as a decision criterion: We say the network has made a decision if one of the (smoothed) rates crosses a threshold. What are appropriate values for `avg_window_width` and `rate threshold` to detect a decision from the two rates?

Hint: Use Brian's smooth_rate function:

```
avg_window_width = 123*b2.ms
sr = results["rate_monitor_A"].smooth_rate(window="flat", width=avg_window_width)/b2.
↪Hz
```

Question: Implementing a decision criterion

- Using your insights from the previous questions, implement a function **get_decision_time** that takes two RateMonitors , a `avg_window_width` and a `rate_threshold`. The function should return a tuple (decision_time_Left, decision_time_right). The decision time is the time index when some decision boundary is crossed. Possible return values are (1234.5ms, 0ms) for decision "Left", (0ms, 987.6ms) for decision "Right" and (0ms, 0ms) for the case when no decision is made within the simulation time. A return value like (123ms, 456ms) is an error and occurs if your function is called with inappropriate values for `avg_window_width` and `rate_threshold`.

  The following code block shows how your function is called.

```
>> get_decision_time(results["rate_monitor_A"], results["rate_monitor_B"], avg_window_
↪width=123*b2.ms, rate_threshold=45.6*b2.Hz)
>> (0.543 * second, 0. * second)
```

The following code fragments could be useful:

```
smoothed_rates_A = rate_monitor_A.smooth_rate(window="flat", width=avg_window_width) /
↪ b2.Hz
idx_A = numpy.argmax(smoothed_rates_A > rate_threshold/b2.Hz)
t_A = idx_A * b2.defaultclock.dt
```

Run a few simulations to test your function.

Exercise: Percent-correct and Decision-time as a function of coherence level

We now systematically investigate how the coherence level influences the decision making process. Running multiple repetitions for different coherence levels, we can study how well the network is able to make correct decisions.

You can pass your function *get_decision_time* as an argument to *competing_populations.decision_making.run_multiple_simulations()* as shown here:

```
coherence_levels = [0.15, 0.8]
nr_repetitions = 3

time_to_A, time_to_B, count_A, count_B, count_No = decision_making.run_multiple_
↪simulations(get_decision_time,coherence_levels, nr_repetitions, max_sim_time=??,␣
↪rate_threshold=??, avg_window_width=??)
```

The return value `time_to_A` is a matrix of size [nr_of_c_levels x nr_of_repetitions]. `count_A` is the number of times the network decides for A (= "Left" by convention). The other values are analogous.

Check the documentation of *run_multiple_simulations()* and set the parameters according to the findings in previous questions.

Question: Percent-Correct, Time-to-decision, Time-to-wrong-decision

Using *run_multiple_simulations()*, run at least 10 simulations for each of the two `coherence_levels = [0.1, 0.8]`. The simulation stops either when a decision is made or after a maximum simulation time (set it to ~1200ms). If you have sufficient time/computing-power, you should run more repetitions and more levels, and you could even try larger networks. Then analyse your simulation results by visualizing the following statistics. For each of the questions, ignore the simulations with "no decision".

- Visualize `Percent correct` versus `coherence level`.
- Visualize `Time to decision` versus `coherence level`.
- Discuss your results.

References

[1] Wong, K.-F. & Wang, X.-J. A Recurrent Network Mechanism of Time Integration in Perceptual Decisions. J. Neurosci. 26, 1314–1328 (2006).

[2] Parts of this exercise and parts of the implementation are inspired by material from *Stanford University, BIOE 332: Large-Scale Neural Modeling, Kwabena Boahen & Tatiana Engel, 2013*, online available.

Python exercise modules

All exercises are contained in subpackages of the python package *neurodynex*. The subpackages contain modules used for each exercise. The module *neurodynex.tools* provides functions shared across exercises. Note that the code is not optimized for performance and there is no guarantee for correctness.

neurodynex package

Subpackages

neurodynex.adex_model package

Submodules

neurodynex.adex_model.AdEx module

Implementation of the Adaptive Exponential Integrate-and-Fire model.

See Neuronal Dynamics Chapter 6 Section 1

neurodynex.adex_model.AdEx.**getting_started**()
Simple example to get started

neurodynex.adex_model.AdEx.**plot_adex_state**(*adex_state_monitor*)
Visualizes the state variables: w-t, v-t and phase-plane w-v

Parameters **adex_state_monitor** (*StateMonitor*) – States of "v" and "w"

neurodynex.adex_model.AdEx.**simulate_AdEx_neuron**(*tau_m=5. * msecond*, *R=0.5 * Gohm*, *v_rest=-70. * mvolt*, *v_reset=-51. * mvolt*, *v_rheobase=-50. * mvolt*, *a=0.5 * nsiemens*, *b=7. * pamp*, *v_spike=-30. * mvolt*, *delta_T=2. * mvolt*, *tau_w=100. * msecond*, *I_stim=<brian2.input.timedarray.TimedArray object>*, *simulation_time=200. * msecond*)
Implementation of the AdEx model with a single adaptation variable w.

The Brian2 model equations are:

$$\frac{dv}{dt} = (-(v - v_rest) + delta_T * exp((v - v_rheobase)/delta_T) + R * I_stim(t,i) - R * w)/(tau_m) : volt$$

$$\frac{dw}{dt} = (a * (v - v_rest) - w)/tau_w : amp$$

Parameters

- **tau_m** (*Quantity*) – membrane time scale
- **R** (*Quantity*) – membrane restistance
- **v_rest** (*Quantity*) – resting potential
- **v_reset** (*Quantity*) – reset potential
- **v_rheobase** (*Quantity*) – rheobase threshold
- **a** (*Quantity*) – Adaptation-Voltage coupling
- **b** (*Quantity*) – Spike-triggered adaptation current (=increment of w after each spike)
- **v_spike** (*Quantity*) – voltage threshold for the spike condition
- **delta_T** (*Quantity*) – Sharpness of the exponential term
- **tau_w** (*Quantity*) – Adaptation time constant
- **I_stim** (*TimedArray*) – Input current
- **simulation_time** (*Quantity*) – Duration for which the model is simulated

Returns A b2.StateMonitor for the variables "v" and "w" and a b2.SpikeMonitor

Return type (state_monitor, spike_monitor)

Module contents

neurodynex.brunel_model package

Submodules

neurodynex.brunel_model.LIF_spiking_network module

Implementation of the Brunel 2000 network: sparsely connected network of identical LIF neurons (Model A).

neurodynex.brunel_model.LIF_spiking_network.**getting_started**()
A simple example to get started

neurodynex.brunel_model.LIF_spiking_network.**simulate_brunel_network**(*N_Excit=5000*, *N_Inhib=None*, *N_extern=1000*, *connection_probability=0.1*, *w0=100. * uvolt*, *g=4.0*, *synaptic_delay=1.5 * msecond*, *poisson_input_rate=13. * hertz*, *w_external=None*, *v_rest=0. * volt*, *v_reset=10. * mvolt*, *firing_threshold=20. * mvolt*, *membrane_time_scale=20. * msecond*, *abs_refractory_period=2. * msecond*, *monitored_subset_size=100*, *random_vm_init=False*, *sim_time=100. * msecond*)
Fully parametrized implementation of a sparsely connected network of LIF neurons (Brunel 2000)

Parameters

- **N_Excit** (*int*) – Size of the excitatory popluation
- **N_Inhib** (*int*) – optional. Size of the inhibitory population. If not set (=None), N_Inhib is set to N_excit/4.
- **N_extern** (*int*) – optional. Number of presynaptic excitatory poisson neurons. Note: if set to a value, this number does NOT depend on N_Excit and NOT depend on connection_probability (this is different from the book and paper. Only if N_extern is set to 'None', then N_extern is computed as N_Excit*connection_probability.
- **connection_probability** (*float*) – probability to connect to any of the (N_Excit+N_Inhib) neurons CE = connection_probability*N_Excit CI = connection_probability*N_Inhib Cexternal = N_extern
- **w0** (*float*) – Synaptic strength J
- **g** (*float*) – relative importance of inhibition. J_exc = w0. J_inhib = -g*w0
- **synaptic_delay** (*Quantity*) – Delay between presynaptic spike and postsynaptic increase of v_m
- **poisson_input_rate** (*Quantity*) – Poisson rate of the external population
- **w_external** (*float*) – optional. Synaptic weight of the excitatory external poisson neurons onto all neurons in the network. Default is None, in that case w_external is set to w0, which is the standard value in the book and in the paper Brunel2000. The purpose of this parameter is to see the effect of external input in the absence of network feedback(setting w0 to 0mV and w_external>0).
- **v_rest** (*Quantity*) – Resting potential
- **v_reset** (*Quantity*) – Reset potential
- **firing_threshold** (*Quantity*) – Spike threshold
- **membrane_time_scale** (*Quantity*) – tau_m
- **abs_refractory_period** (*Quantity*) – absolute refractory period, tau_ref
- **monitored_subset_size** (*int*) – nr of neurons for which a VoltageMonitor is recording Vm
- **random_vm_init** (*bool*) – if true, the membrane voltage of each neuron is initialized with a random value drawn from Uniform(v_rest, firing_threshold)
- **sim_time** (*Quantity*) – Simulation time

Returns (rate_monitor, spike_monitor, voltage_monitor, idx_monitored_neurons) PopulationRateMonitor: Rate Monitor SpikeMonitor: SpikeMonitor for ALL (N_Excit+N_Inhib) neurons StateMonitor: membrane voltage for a selected subset of neurons list: index of monitored neurons. length = monitored_subset_size

Module contents

neurodynex.cable_equation package

Submodules

neurodynex.cable_equation.passive_cable module

Implements compartmental model of a passive cable. See Neuronal Dynamics Chapter 3 Section 2

neurodynex.cable_equation.passive_cable.**getting_started**()
A simple code example to get started.

neurodynex.cable_equation.passive_cable.**simulate_passive_cable**(*current_injection_location=[166.666666 * umetre], input_current=<brian2.input.timedarray.T object>, length=0.5 * mmetre, diameter=2. * umetre, r_longitudinal=0.5 * metre ** 3 * kilogram * second ** -3 * amp ** -2, r_transversal=1.25 * metre ** 4 * kilogram * second ** -3 * amp ** -2, e_leak=-70. * mvolt, initial_voltage=-70. * mvolt, capacitance=0.008 * metre ** -4 * kilogram ** -1 * second ** 4 * amp ** 2, nr_compartments=200, simulation_time=5. * msecond*)

Builds a multicompartment cable and numerically approximates the cable equation.

Parameters

- **t_spikes** (*int*) – list of spike times
- **current_injection_location** (*list*) – List [] of input locations (Quantity, Length): [123.*b2.um]
- **input_current** (*TimedArray*) – TimedArray of current amplitudes. One column per current_injection_location.
- **length** (*Quantity*) – Length of the cable: 0.8*b2.mm
- **diameter** (*Quantity*) – Diameter of the cable: 0.2*b2.um
- **r_longitudinal** (*Quantity*) – The longitudinal (axial) resistance of the cable: 0.5*b2.kohm*b2.mm
- **r_transversal** (*Quantity*) – The transversal resistance (=membrane resistance): 1.25*b2.Mohm*b2.mm**2

- **e_leak** (*Quantity*) – The reversal potential of the leak current (=resting potential): -70.*b2.mV
- **initial_voltage** (*Quantity*) – Value of the potential at t=0: -70.*b2.mV
- **capacitance** (*Quantity*) – Membrane capacitance: 0.8*b2.uF/b2.cm**2
- **nr_compartments** (*int*) – Number of compartments. Spatial discretization: 200
- **simulation_time** (*Quantity*) – Time for which the dynamics are simulated: 5*b2.ms

Returns The state monitor contains the membrane voltage in a Time x Location matrix. The SpatialNeuron object specifies the simulated neuron model and gives access to the morphology. You may want to use those objects for spatial indexing: myVoltageStateMonitor[mySpatialNeuron.morphology[0.123*b2.um]].v

Return type (StateMonitor, SpatialNeuron)

Module contents

neurodynex.competing_populations package

Submodules

neurodynex.competing_populations.decision_making module

Implementation of a decision making model of [1] Wang, Xiao-Jing. "Probabilistic decision making by slow reverberation in cortical circuits." Neuron 36.5 (2002): 955-968.

Some parts of this implementation are inspired by material from *Stanford University, BIOE 332: Large-Scale Neural Modeling, Kwabena Boahen & Tatiana Engel, 2013*, online available.

Note: Most parameters differ from the original publication.

neurodynex.competing_populations.decision_making.**getting_started**()
A simple example to get started. Returns:

neurodynex.competing_populations.decision_making.**print_version**()

neurodynex.competing_populations.decision_making.**run_multiple_simulations**(*f_get_decision_time*, *coherence_levels*, *nr_repetitions*, *max_sim_time=1. * msecond*, *rate_threshold=1. * hertz*, *avg_window_width=1. * msecond*, *N_excit=384*, *N_inhib=96*, *weight_scaling=5.33*, *w_pos=1.9*, *t_stim_start=100. * msecond*, *t_stim_duration=9.999 * second*, *nu0_mean_stim_Hz=160*, *nu0_std_stim_Hz=20.0*, *stim_upd_interval=30. * msecond*, *N_extern=1000*, *firing_rate_extern=9.8 * hertz*)

Parameters

- **f_get_decision_time** (*Function*) – a function that implements the decision criterion.
- **coherence_levels** (*array*) – A list of coherence levels
- **nr_repetitions** (*int*) – Number of repetitions (independent simulations).
- **max_sim_time** (*Quantity*) – max simulation time
- **rate_threshold** (*Quantity*) – A firing rate threshold. The simulation stops before max_sim_time if one of the two subpopulations "Left" or "Right" fires above this threshold (averaged over some time window)
- **avg_window_width** (*Quantity*) – window size when smoothing the firing rates.
- **N_excit** (*int*) – total number of neurons in the excitatory population

- **N_inhib** (*int*) – nr of neurons in the inhibitory populations
- **weight_scaling** (*float*) – When increasing the number of neurons by 2, the weights should be scaled down by 1/2
- **t_stim_start** (*Quantity*) – Start of the stimulation
- **t_stim_duration** (*Quantity*) – Duration of the stimulation
- **nu0_mean_stimulus_Hz** (*float*) – maximum mean firing rate of the stimulus if c=1
- **nu0_std_stimulus_Hz** (*float*) – std deviation of the stimulating PoissonGroups.
- **stim_upd_interval** (*Quantity*) – the mean of the stimulating PoissonGroups is re-sampled every at this interval
- **N_extern=1000** (*int*) – Size of the external PoissonGroup (unstructured input)
- **firing_rate_extern** (*Quantity*) – Firing frequency of the external PoissonGroup

Returns Five values are returned. [1] time_to_A: A matrix of size [nr_of_c_levels x nr_of_repetitions], where for each entry the time stamp for decision A is recorded. If decision B was made, the entry is 0ms. [2] time_to_B (array): A matrix of size [nr_of_c_levels x nr_of_repetitions], where for each entry the time stamp for decision B is recorded. If decision A was made, the entry is 0ms. [3] count_A (int): Nr of times decision A is made. [4] count_B (int): Nr of times decision B is made. [5] count_No (int): Nr of times no decision is made within the simulation time.

Return type results_tuple (array)

neurodynex.competing_populations.decision_making.**sim_decision_making_network**(*N_Excit=384*, *N_Inhib=96*, *weight_scaling_fact* *t_stimulus_start=10* * *msec-* *ond*, *t_stimulus_duration* * *sec-* *ond*, *co-* *her-* *ence_level=0.0*, *stim-* *u-* *lus_update_interval* * *msec-* *ond*, *nu0_mean_stimulus* *nu0_std_stimulus_H* *N_extern=1000*, *fir-* *ing_rate_extern=9.8* * *hertz*, *w_pos=1.9*, *f_Subpop_size=0.25* *max_sim_time=1.* * *sec-* *ond*, *stop_condition_rate* *mon-* *i-* *tored_subset_size=5*

Parameters

- **N_Excit** (*int*) – total number of neurons in the excitatory population
- **N_Inhib** (*int*) – nr of neurons in the inhibitory populations
- **weight_scaling_factor** – When increasing the number of neurons by 2, the weights should be scaled down by 1/2
- **t_stimulus_start** (*Quantity*) – time when the stimulation starts
- **t_stimulus_duration** (*Quantity*) – duration of the stimulation
- **coherence_level** (*int*) – coherence of the stimulus. Difference in mean between the PoissonGroups "left" stimulus and "right" stimulus
- **stimulus_update_interval** (*Quantity*) – the mean of the stimulating PoissonGroups is re-sampled every at this interval
- **nu0_mean_stimulus_Hz** (*float*) – maximum mean firing rate of the stimulus if c=1

- **nu0_std_stimulus_Hz** (*float*) – std deviation of the stimulating PoissonGroups.
- **N_extern** (*int*) – nr of neurons in the stimulus independent poisson background population
- **firing_rate_extern** (*int*) – firing rate of the stimulus independent poisson background population
- **w_pos** (*float*) – Scaling (strengthening) of the recurrent weights within the subpopulations "Left" and "Right"
- **f_Subpop_size** (*float*) – fraction of the neurons in the subpopulations "Left" and "Right". #left = #right = f_Subpop_size*N_Excit.
- **max_sim_time** (*Quantity*) – simulated time.
- **stop_condition_rate** (*Quantity*) – An optional stopping criteria: If not None, the simulation stops if the firing rate of either subpopulation "Left" or "Right" is above stop_condition_rate.
- **monitored_subset_size** (*int*) – max nr of neurons for which a state monitor is registered.

Returns

"rate_monitor_A", "spike_monitor_A", "voltage_monitor_A", "idx_monitored_neurons_A", "rate_monitor_B", "spike_monitor_B", "voltage_monitor_B", "idx_monitored_neurons_B", "rate_monitor_Z", "spike_monitor_Z", "voltage_monitor_Z", "idx_monitored_neurons_Z", "rate_monitor_inhib", "spike_monitor_inhib", "voltage_monitor_inhib", "idx_monitored_neurons_inhib"

Return type A dictionary with the following keys (strings)

Module contents

neurodynex.exponential_integrate_fire package

Submodules

neurodynex.exponential_integrate_fire.exp_IF module

Exponential Integrate-and-Fire model. See Neuronal Dynamics, Chapter 5 Section 2

neurodynex.exponential_integrate_fire.exp_IF.**getting_started**()
A simple example

neurodynex.exponential_integrate_fire.exp_IF.**simulate_exponential_IF_neuron**(*tau=12. * msec-ond, R=20. * Mohm, v_rest=-65. * mvolt, v_reset=-60. * mvolt, v_rheobase=-55. * mvolt, v_spike=-30. * mvolt, delta_T=2. * mvolt, I_stim=<brian2.inpu ob-ject>, sim-u-la-tion_time=200. * msec-ond*)

Implements the dynamics of the exponential Integrate-and-fire model

Parameters

- **tau** (*Quantity*) – Membrane time constant
- **R** (*Quantity*) – Membrane resistance
- **v_rest** (*Quantity*) – Resting potential
- **v_reset** (*Quantity*) – Reset value (vm after spike)
- **v_rheobase** (*Quantity*) – Rheobase threshold
- **v_spike** (*Quantity*) – voltage threshold for the spike condition
- **delta_T** (*Quantity*) – Sharpness of the exponential term
- **I_stim** (*TimedArray*) – Input current
- **simulation_time** (*Quantity*) – Duration for which the model is simulated

Returns A b2.StateMonitor for the variable "v" and a b2.SpikeMonitor

Return type (voltage_monitor, spike_monitor)

Module contents

neurodynex.hodgkin_huxley package

Submodules

neurodynex.hodgkin_huxley.HH module

Implementation of a Hodging-Huxley neuron Relevant book chapters:

- http://neuronaldynamics.epfl.ch/online/Ch2.S2.html

neurodynex.hodgkin_huxley.HH.**getting_started**()
An example to quickly get started with the Hodgkin-Huxley module.

neurodynex.hodgkin_huxley.HH.**plot_data**(*state_monitor*, *title=None*)
Plots the state_monitor variables [”vm”, “I_e”, “m”, “n”, “h”] vs. time.

Parameters

- **state_monitor** (*StateMonitor*) – the data to plot
- **title** (*string, optional*) – plot title to display

neurodynex.hodgkin_huxley.HH.**simulate_HH_neuron**(*input_current*, *simulation_time*)
A Hodgkin-Huxley neuron implemented in Brian2.

Parameters

- **input_current** (*TimedArray*) – Input current injected into the HH neuron
- **simulation_time** (*float*) – Simulation time [seconds]

Returns Brian2 StateMonitor with recorded fields [”vm”, “I_e”, “m”, “n”, “h”]

Return type StateMonitor

Module contents

neurodynex.hopfield_network package

Submodules

neurodynex.hopfield_network.demo module

neurodynex.hopfield_network.demo.**run_demo**()
Simple demo

neurodynex.hopfield_network.demo.**run_hf_demo**(*pattern_size=4*, *nr_random_patterns=3*, *reference_pattern=0*, *initially_flipped_pixels=3*, *nr_iterations=6*, *random_seed=None*)
Simple demo.

Parameters

- **pattern_size** –
- **nr_random_patterns** –
- **reference_pattern** –
- **initially_flipped_pixels** –
- **nr_iterations** –
- **random_seed** –

Returns:

neurodynex.hopfield_network.demo.**run_hf_demo_alphabet**(*letters*, *initialization_noise_level=0.2*, *random_seed=None*)

Simple demo

Parameters

- **letters** –
- **initialization_noise_level** –
- **random_seed** –

Returns:

neurodynex.hopfield_network.demo.**run_user_function_demo**()

neurodynex.hopfield_network.network module

This file implements a Hopfield network. It provides functions to set and retrieve the network state, store patterns.

Relevant book chapters:

- http://neuronaldynamics.epfl.ch/online/Ch17.S2.html

class neurodynex.hopfield_network.network.**HopfieldNetwork**(*nr_neurons*)

Implements a Hopfield network.

nrOfNeurons
int – Number of neurons

weights
numpy.ndarray – nrOfNeurons x nrOfNeurons matrix of weights

state
numpy.ndarray – current network state. matrix of shape (nrOfNeurons, nrOfNeurons)

iterate()
Executes one timestep of the dynamics

reset_weights()
Resets the weights to random values

run(*nr_steps=5*)
Runs the dynamics.

Parameters **nr_steps** (*float, optional*) – Timesteps to simulate

run_with_monitoring(*nr_steps=5*)
Iterates at most nr_steps steps. records the network state after every iteration

Parameters **nr_steps** –

Returns a list of 2d network states

set_dynamics_sign_async()
Sets the update dynamics to the g(h) = sign(h) functions. Neurons are updated asynchronously: In random order, all neurons are updated sequentially

set_dynamics_sign_sync()
sets the update dynamics to the synchronous, deterministic g(h) = sign(h) function

set_dynamics_to_user_function(*update_function*)
Sets the network dynamics to the given update function

Parameters **update_function** – upd(state_t0, weights) -> state_t1. Any function mapping a state s0 to the next state s1 using a function of s0 and weights.

set_state_from_pattern(*pattern*)
Sets the neuron states to the pattern pixel. The pattern is flattened.

Parameters **pattern** – pattern

store_patterns(*pattern_list*)
Learns the patterns by setting the network weights. The patterns themselves are not stored, only the weights are updated! self connections are set to 0.

Parameters **pattern_list** – a nonempty list of patterns.

neurodynex.hopfield_network.pattern_tools module

Functions to create 2D patterns. Note, in the hopfield model, we define patterns as vectors. To make the exercise more visual, we use 2D patterns (N by N ndarrays).

class neurodynex.hopfield_network.pattern_tools.**PatternFactory**(*pattern_length*, *pattern_width=None*)
Creates square patterns of size pattern_length x pattern_width If pattern length is omitted, square patterns are produced

create_L_pattern(*l_width=1*)
creates a pattern with column 0 (left) and row n (bottom) set to +1. Increase l_width to set more columns and rows (default is 1)

Parameters **l_width** (*int*) – nr of rows and columns to set

Returns an L shaped pattern.

create_all_off()

Returns 2d pattern, all pixels off

create_all_on()

Returns 2d pattern, all pixels on

create_checkerboard()
creates a checkerboard pattern of size (pattern_length x pattern_width) :returns: checkerboard pattern

create_random_pattern(*on_probability=0.5*)
Creates a pattern_length by pattern_width 2D random pattern :param on_probability:

Returns a new random pattern

create_random_pattern_list(*nr_patterns*, *on_probability=0.5*)
Creates a list of nr_patterns random patterns :param nr_patterns: length of the new list :param on_probability:

Returns a list of new random patterns of size (pattern_length x pattern_width)

create_row_patterns(*nr_patterns=None*)
creates a list of n patterns, the i-th pattern in the list has all states of the i-th row set to active. This is convenient to create a list of orthogonal patterns which are easy to visually identify

Parameters **nr_patterns** –

Returns list of orthogonal patterns

reshape_patterns(*pattern_list*)
reshapes all patterns in pattern_list to have shape = (self.pattern_length, self.pattern_width)

Parameters

- **self** –
- **pattern_list** –

Returns:

neurodynex.hopfield_network.pattern_tools.**compute_overlap**(*pattern1*, *pattern2*)
compute overlap

Parameters

- **pattern1** –
- **pattern2** –

Returns: Overlap between pattern1 and pattern2

neurodynex.hopfield_network.pattern_tools.**compute_overlap_list**(*reference_pattern*, *pattern_list*)
Computes the overlap between the reference_pattern and each pattern in pattern_list

Parameters

- **reference_pattern** –
- **pattern_list** – list of patterns

Returns A list of the same length as pattern_list

neurodynex.hopfield_network.pattern_tools.**compute_overlap_matrix**(*pattern_list*)
For each pattern, it computes the overlap to all other patterns.

Parameters **pattern_list** –

Returns the matrix m(i,k) = overlap(pattern_list[i], pattern_list[k]

neurodynex.hopfield_network.pattern_tools.**flip_n**(*template*, *nr_of_flips*)
makes a copy of the template pattern and flips exactly n randomly selected states. :param template: :param nr_of_flips:

Returns a new pattern

neurodynex.hopfield_network.pattern_tools.**get_noisy_copy**(*template*, *noise_level*)
Creates a copy of the template pattern and reassigns N pixels. N is determined by the noise_level Note: reassigning a random value is not the same as flipping the state. This function reassigns a random value.

Parameters

- **template** –
- **noise_level** – a value in [0,1]. for 0, this returns a copy of the template.

- **1, a random pattern of the same size as template is returned.** (*for*) –

Returns:

neurodynex.hopfield_network.pattern_tools.**get_pattern_diff**(*pattern1*, *pattern2*, *diff_code=0*)

Creates a new pattern of same size as the two patterns. the diff pattern has the values pattern1 = pattern2 where the two patterns have the same value. Locations that differ between the two patterns are set to diff_code (default = 0)

Parameters

- **pattern1** –
- **pattern2** –
- **diff_code** – the values of the new pattern, at locations that differ between
- **two patterns are set to diff_code.** (*the*) –

Returns the diff pattern.

neurodynex.hopfield_network.pattern_tools.**load_alphabet**()

Load alphabet dict from the file `data/alphabet.pickle.gz`, which is included in the neurodynex release.

Returns Dictionary of 10x10 patterns

Return type dict

Raises `ImportError` – Raised if `neurodynex` can not be imported. Please install neurodynex.

neurodynex.hopfield_network.pattern_tools.**reshape_patterns**(*pattern_list*, *shape*)

reshapes each pattern in pattern_list to the given shape

Parameters

- **pattern_list** –
- **shape** –

Returns:

neurodynex.hopfield_network.plot_tools module

Helper tools to visualize patterns and network state

neurodynex.hopfield_network.plot_tools.**plot_nework_weights**(*hopfield_network*, *color_map='jet'*)

Visualizes the network's weight matrix

Parameters

- **hopfield_network** –
- **color_map** –

neurodynex.hopfield_network.plot_tools.**plot_overlap_matrix**(*overlap_matrix*, *color_map='bwr'*)

Visualizes the pattern overlap

Parameters

- **overlap_matrix** –
- **color_map** –

neurodynex.hopfield_network.plot_tools.**plot_pattern**(*pattern*, *reference=None*, *color_map='brg'*, *diff_code=0*)

Plots the pattern. If a (optional) reference pattern is provided, the pattern is plotted with differences highlighted

Parameters

- **pattern** (*numpy.ndarray*) – N by N pattern to plot
- **reference** (*numpy.ndarray*) – optional. If set, differences between pattern and reference are highlighted

neurodynex.hopfield_network.plot_tools.**plot_pattern_list**(*pattern_list*, *color_map='brg'*)

Plots the list of patterns

Parameters

- **pattern_list** –
- **color_map** –

Returns:

neurodynex.hopfield_network.plot_tools.**plot_state_sequence_and_overlap**(*state_sequence*, *pattern_list*, *reference_idx*, *color_map='brg'*, *suptitle=None*)

For each time point t (= index of state_sequence), plots the sequence of states and the overlap (barplot) between state(t) and each pattern.

Parameters

- **state_sequence** – (list(numpy.ndarray))
- **pattern_list** – (list(numpy.ndarray))
- **reference_idx** – (int) identifies the pattern in pattern_list for which wrong pixels are colored.

Module contents

neurodynex.leaky_integrate_and_fire package

Submodules

neurodynex.leaky_integrate_and_fire.LIF module

This file implements a leaky intergrate-and-fire (LIF) model. You can inject a step current or sinusoidal current into neuron using LIF_Step() or LIF_Sinus() methods respectively.

Relevant book chapters:

- http://neuronaldynamics.epfl.ch/online/Ch1.S3.html

neurodynex.leaky_integrate_and_fire.LIF.**get_random_param_set**(*random_seed=None*)
creates a set of random parameters. All values are constrained to their typical range :returns: a list of (obfuscated) parameters. Use this vector when calling simulate_random_neuron() :rtype: list

neurodynex.leaky_integrate_and_fire.LIF.**getting_started**()
An example to quickly get started with the LIF module. Returns:

neurodynex.leaky_integrate_and_fire.LIF.**print_default_parameters**()
Prints the default values Returns:

neurodynex.leaky_integrate_and_fire.LIF.**print_obfuscated_parameters**(*obfuscated_params*)
Print the de-obfuscated values to the console

Parameters **obfuscated_params** –

Returns:

neurodynex.leaky_integrate_and_fire.LIF.**simulate_LIF_neuron**(*input_current*, *simulation_time=5. * msecond*, *v_rest=-70. * mvolt*, *v_reset=-65. * mvolt*, *firing_threshold=-50. * mvolt*, *membrane_resistance=10. * Mohm*, *membrane_time_scale=8. * msecond*, *abs_refractory_period=2. * msecond*)
Basic leaky integrate and fire neuron implementation.

Parameters

- **input_current** (*TimedArray*) – TimedArray of current amplitudes. One column per current_injection_location.
- **simulation_time** (*Quantity*) – Time for which the dynamics are simulated: 5ms
- **v_rest** (*Quantity*) – Resting potential: -70mV
- **v_reset** (*Quantity*) – Reset voltage after spike - 65mV
- **firing_threshold** (*Quantity*) –
- **membrane_resistance** (*Quantity*) – 10Mohm
- **membrane_time_scale** (*Quantity*) – 8ms
- **abs_refractory_period** (*Quantity*) – 2ms

Returns Brian2 StateMonitor for the membrane voltage "v" SpikeMonitor: Biran2 SpikeMonitor

Return type StateMonitor

neurodynex.leaky_integrate_and_fire.LIF.**simulate_random_neuron**(*input_current*, *obfuscated_param_set*)
Simulates a LIF neuron with unknown parameters (obfuscated_param_set) :param input_current: The current to probe the neuron :type input_current: TimedArray :param obfuscated_param_set: obfuscated parameters :type obfuscated_param_set: list

Returns Brian2 StateMonitor for the membrane voltage "v" SpikeMonitor: Biran2 SpikeMonitor

Return type StateMonitor

Module contents

neurodynex.neuron_type package

Submodules

neurodynex.neuron_type.neurons module

This file implements a type I and a type II model from the abstract base class NeuronAbstract.

You can inject step currents and plot the responses, as well as get firing rates.

Relevant book chapters:

- http://neuronaldynamics.epfl.ch/online/Ch4.S4.html

class neurodynex.neuron_type.neurons.**NeuronAbstract**
Bases: object

Abstract base class for both neuron types.

This stores its own recorder and network, allowing each neuron to be run several times with changing currents while keeping the same neurogroup object and network internally.

get_neuron_type()
Type I or II.

Returns type as a string "Type I" or "Type II"

run(*input_current*, *simtime*)
Runs the neuron for a given current.

Parameters

- **input_current** (*TimedArray*) – Input current injected into the neuron
- **simtime** (*Quantity*) – Simulation time in correct Brian units.

Returns Brian2 StateMonitor with input current (I) and voltage (V) recorded

Return type StateMonitor

class neurodynex.neuron_type.neurons.**NeuronX**
Bases: *neurodynex.neuron_type.neurons.NeuronAbstract*

class neurodynex.neuron_type.neurons.**NeuronY**
Bases: *neurodynex.neuron_type.neurons.NeuronAbstract*

neurodynex.neuron_type.neurons.**getting_started**()
simple demo to get started

Returns:

neurodynex.neuron_type.neurons.**neurontype_random_reassignment**()
Randomly reassign the two types: Returns:

neurodynex.neuron_type.neurons.**plot_data**(*state_monitor*, *title=None*, *show=True*)
Plots a TimedArray for values I, v and w

Parameters

- **state_monitor** (*StateMonitor*) – the data to plot. expects [”v”, “w”, “I”] and (by default) “t”
- **title** (*string, optional*) – plot title to display
- **show** (*bool, optional*) – call plt.show for the plot

Returns

Brian2 StateMonitor with input current (I) and voltage (V) recorded

Return type StateMonitor

Module contents

neurodynex.ojas_rule package

Submodules

neurodynex.ojas_rule.oja module

This file implements Oja’s hebbian learning rule.

Relevant book chapters:

- http://neuronaldynamics.epfl.ch/online/Ch19.S2.html#SS1.p6

neurodynex.ojas_rule.oja.**learn**(*cloud*, *initial_angle=None*, *eta=0.005*)

Run one batch of Oja’s learning over a cloud of datapoints.

Parameters

- **cloud** (*numpy.ndarray*) – An N by 2 array of datapoints. You can think of each of the two columns as the time series of firing rates of one presynaptic neuron.
- **initial_angle** (*float, optional*) – angle of initial set of weights [deg]. If None, this is random.
- **eta** (*float, optional*) – learning rate

Returns time course of the weight vector

Return type numpy.ndarray

neurodynex.ojas_rule.oja.**make_cloud**(*n=2000*, *ratio=1*, *angle=0*)

Returns an oriented elliptic gaussian cloud of 2D points

Parameters

- **n** (*int, optional*) – number of points in the cloud
- **ratio** (*int, optional*) – (std along the short axis) / (std along the long axis)
- **angle** (*int, optional*) – rotation angle [deg]

Returns array of datapoints

Return type numpy.ndarray

neurodynex.ojas_rule.oja.**plot_oja_trace**(*data_cloud*, *weights_course*)

Plots the datapoints and the time series of the weights :param data_cloud: n by 2 data :type data_cloud: numpy.ndarray :param weights_course: n by 2 weights :type weights_course: numpy.ndarray

Returns:

neurodynex.ojas_rule.oja.**run_oja**(*n=2000*, *ratio=1.0*, *angle=0.0*, *learning_rate=0.01*, *do_plot=True*)

Generates a point cloud and runs Oja's learning rule once. Optionally plots the result.

Parameters

- **n** (*int, optional*) – number of points in the cloud
- **ratio** (*float, optional*) – (std along the short axis) / (std along the long axis)
- **angle** (*float, optional*) – rotation angle [deg]
- **do_plot** (*bool, optional*) – plot the result

Module contents

neurodynex.phase_plane_analysis package

Submodules

neurodynex.phase_plane_analysis.fitzhugh_nagumo module

This file implements functions to simulate and analyze Fitzhugh-Nagumo type differential equations with Brian2.

Relevant book chapters:

- http://neuronaldynamics.epfl.ch/online/Ch4.html
- http://neuronaldynamics.epfl.ch/online/Ch4.S3.html.

neurodynex.phase_plane_analysis.fitzhugh_nagumo.**get_fixed_point**(*I=0.0*, *eps=0.1*, *a=2.0*)

Computes the fixed point of the FitzHugh Nagumo model as a function of the input current I.

We solve the 3rd order poylnomial equation: v**3 + V + a - I0 = 0

Parameters

- **I** – Constant input [mV]
- **eps** – Inverse time constant of the recovery variable w [1/ms]
- **a** – Offset of the w-nullcline [mV]

Returns (v_fp, w_fp) fixed point of the equations

Return type tuple

neurodynex.phase_plane_analysis.fitzhugh_nagumo.**get_trajectory**(*v0=0.0*, *w0=0.0*, *I=0.0*, *eps=0.1*, *a=2.0*, *tend=500.0*)

Solves the following system of FitzHugh Nagumo equations for given initial conditions:

dv/dt = 1/1ms * v * (1-v**2) - w + I dw/dt = eps * (v + 0.5 * (a - w))

Parameters

- **v0** – Intial condition for v [mV]
- **w0** – Intial condition for w [mV]
- **I** – Constant input [mV]
- **eps** – Inverse time constant of the recovery variable w [1/ms]
- **a** – Offset of the w-nullcline [mV]
- **tend** – Simulation time [ms]

Returns (t, v, w) tuple for solutions

Return type tuple

neurodynex.phase_plane_analysis.fitzhugh_nagumo.**plot_flow**(*I=0.0*, *eps=0.1*, *a=2.0*)

Plots the phase plane of the Fitzhugh-Nagumo model for given model parameters.

Parameters

- **I** – Constant input [mV]
- **eps** – Inverse time constant of the recovery variable w [1/ms]
- **a** – Offset of the w-nullcline [mV]

Module contents

neurodynex.test package

Submodules

neurodynex.test.test_AdEx module

neurodynex.test.test_AdEx.**test_simulate_exponential_IF_neuron**()

Test if simulates simulate_AdEx_neuron generates two spikes

neurodynex.test.test_HH module

neurodynex.test.test_HH.**test_simulate_HH_neuron**()

Test Hodgkin-Huxley model: simulate_HH_neuron()

neurodynex.test.test_LIF module

neurodynex.test.test_LIF.**test_simulate_LIF_neuron**()

Test LIF model: simulate_LIF_neuron(short pulse, 1ms, default values)

neurodynex.test.test_LIF_spiking_network module

neurodynex.test.test_LIF_spiking_network.**test_LIF_spiking_network**()

Test LIF spiking network: simulate_brunel_network(short pulse, 1ms, default values)

neurodynex.test.test_cable_equation module

neurodynex.test.test_cable_equation.**test_simulate_passive_cable**()
Test cable_equation.passive_cable.simulate_passive_cable

neurodynex.test.test_decision_making module

neurodynex.test.test_decision_making.**test_sim_decision_making_network**()
Test if the decision making circuit is initialized and simulated for 3ms

neurodynex.test.test_exponential_IF module

neurodynex.test.test_exponential_IF.**test_simulate_exponential_IF_neuron**()
Test exponential-integrate-and-fire model

neurodynex.test.test_hopfield module

neurodynex.test.test_hopfield.**test_load_alphabet**()
Test if the alphabet patterns can be loaded

neurodynex.test.test_hopfield.**test_overlap**()
Test hopfield_network.pattern_tools overlap

neurodynex.test.test_hopfield.**test_pattern_factory**()
Test hopfield_network.pattern_tools

neurodynex.test.test_nagumo module

neurodynex.test.test_nagumo.**test_runnable_get_fixed_point**()
Test if get_fixed_point is runnable.

neurodynex.test.test_nagumo.**test_runnable_get_trajectory**()
Test if get_trajectory is runnable.

neurodynex.test.test_neuron_type module

neurodynex.test.test_neuron_type.**test_neurons_run**()
Test if neuron functions are runnable.

neurodynex.test.test_neuron_type.**test_neurons_type**()
Test if NeuronX and NeuronY constructors are callable

neurodynex.test.test_oja module

neurodynex.test.test_oja.**test_oja**()
Test if Oja learns from a cloud

neurodynex.test.test_spike_tools module

neurodynex.test.test_spike_tools.**test_filter_spike_trains**()
Test filtering of spike train dict

neurodynex.test.test_working_memory module

neurodynex.test.test_working_memory.**test_woking_memory_sim**()
Test if the working memory circuit is initialized and simulated for 1ms

Module contents

neurodynex.test.**run_nose**()
Runs nose tests, used maily for anaconda deployment

neurodynex.tools package

Submodules

neurodynex.tools.input_factory module

neurodynex.tools.input_factory.**get_ramp_current**(*t_start*, *t_end*, *unit_time*, *amplitude_start*, *amplitude_end*, *append_zero=True*)
Creates a ramp current. If t_start == t_end, then ALL entries are 0.

Parameters

- **t_start** (*int*) – start of the ramp
- **t_end** (*int*) – end of the ramp
- **unit_time** (*Brian2 unit*) – unit of t_start and t_end. e.g. 0.1*brian2.ms
- **amplitude_start** (*Quantity*) – amplitude of the ramp at t_start. e.g. 3.5*brian2.uamp
- **amplitude_end** (*Quantity*) – amplitude of the ramp at t_end. e.g. 4.5*brian2.uamp
- **append_zero** (*bool, optional*) – if true, 0Amp is appended at t_end+1. Without that trailing 0, Brian reads out the last value in the array (=amplitude_end) for all indices > t_end.

Returns Brian2.TimedArray

Return type TimedArray

neurodynex.tools.input_factory.**get_sinusoidal_current**(*t_start*, *t_end*, *unit_time*, *amplitude*, *frequency*, *direct_current*, *phase_offset=0.0*, *append_zero=True*)
Creates a sinusoidal current. If t_start == t_end, then ALL entries are 0.

Parameters

- **t_start** (*int*) – start of the sine wave
- **t_end** (*int*) – end of the sine wave
- **unit_time** (*Quantity, Time*) – unit of t_start and t_end. e.g. 0.1*brian2.ms
- **amplitude** (*Quantity, Current*) – maximum amplitude of the sinus e.g. 3.5*brian2.uamp
- **frequency** (*Quantity, Hz*) – Frequency of the sine. e.g. 0.5*brian2.kHz
- **direct_current** (*Quantity, Current*) – DC-component (=offset) of the current
- **phase_offset** (*float, Optional*) – phase at t_start. Default = 0.
- **append_zero** (*bool, optional*) – if true, 0Amp is appended at t_end+1. Without that trailing 0, Brian reads out the last value in the array for all indices > t_end.

Returns Brian2.TimedArray

Return type TimedArray

neurodynex.tools.input_factory.**get_spikes_current**(*t_spikes*, *unit_time*, *amplitude*, *append_zero=True*)

Creates a two dimensional TimedArray wich has one column for each value in t_spikes. All values in each column are 0 except one, the spike time as specified in t_spikes is set to amplitude. Note: This function is provided to easily insert pulse currents into a cable. For other use of spike input, search the Brian2 documentation for SpikeGeneration.

Parameters

- **t_spikes** (*int*) – list of spike times
- **unit_time** (*Quantity, Time*) – unit of t_spikes . e.g. 1*brian2.ms
- **amplitude** (*Quantity, Current*) – amplitude of the spike. All spikes have the sampe amplitude
- **append_zero** (*bool, optional*) – if true, 0Amp is appended at t_end+1. Without that trailing 0, Brian reads out the last value in the array for all indices > t_end.

Returns Brian2.TimedArray

Return type TimedArray

neurodynex.tools.input_factory.**get_step_current**(*t_start*, *t_end*, *unit_time*, *amplitude*, *append_zero=True*)

Creates a step current. If t_start == t_end, then a single entry in the values array is set to amplitude.

Parameters

- **t_start** (*int*) – start of the step
- **t_end** (*int*) – end of the step
- **unit_time** (*Brian2 unit*) – unit of t_start and t_end. e.g. 0.1*brian2.ms
- **amplitude** (*Quantity*) – amplitude of the step. e.g. 3.5*brian2.uamp
- **append_zero** (*bool, optional*) – if true, 0Amp is appended at t_end+1.
- **that trailing 0, Brian reads out the last value in the array** (*Without*) –

Returns Brian2.TimedArray

Return type TimedArray

neurodynex.tools.input_factory.**get_zero_current**()
Returns a TimedArray with one entry: 0 Amp

Returns TimedArray

neurodynex.tools.input_factory.**getting_started**()

neurodynex.tools.input_factory.**plot_ramp_current_example**()
Example for get_ramp_current

neurodynex.tools.input_factory.**plot_sinusoidal_current_example**()
Example for get_sinusoidal_current

neurodynex.tools.input_factory.**plot_step_current_example**()
Example for get_step_current.

neurodynex.tools.plot_tools module

neurodynex.tools.plot_tools.**plot_ISI_distribution**(*spike_stats*, *hist_nr_bins=50*, *xlim_max_ISI=None*)
Computes the ISI distribution of the given spike_monitor and displays the distribution in a histogram

Parameters

- **spike_stats** (neurodynex.tools.spike_tools.PopulationSpikeStats) – statistics of a population activity
- **hist_nr_bins** (*int*) – Number of histrogram bins. Default:50
- **xlim_max_ISI** (*Quantity*) – Default: None. In not None, the upper xlim of the plot is set to xlim_max_ISI. The CV does not change if this bound is set.

Returns the figure

neurodynex.tools.plot_tools.**plot_network_activity**(*rate_monitor*, *spike_monitor*, *voltage_monitor=None*, *spike_train_idx_list=None*, *t_min=None*, *t_max=None*, *N_highlighted_spiketrains=3*, *avg_window_width=1. * msecond*, *sup_title=None*, *figure_size=(10, 4)*)
Visualizes the results of a network simulation: spike-train, population activity and voltage-traces.

Parameters

- **rate_monitor** (*PopulationRateMonitor*) – rate of the population
- **spike_monitor** (*SpikeMonitor*) – spike trains of individual neurons
- **voltage_monitor** (*StateMonitor*) – optional. voltage traces of some (same as in spike_train_idx_list) neurons
- **spike_train_idx_list** (*list*) – optional. A list of neuron indices whose spike-train is plotted. If no list is provided, all (up to 500) spike-trains in the spike_monitor are plotted. If None, the the list in voltage_monitor.record is used.
- **t_min** (*Quantity*) – optional. lower bound of the plotted time interval. if t_min is None, it is set to the larger of [0ms, (t_max - 100ms)]
- **t_max** (*Quantity*) – optional. upper bound of the plotted time interval. if t_max is None, it is set to the timestamp of the last spike in

- **N_highlighted_spiketrains** (*int*) – optional. Number of spike trains visually highlighted, defaults to 3 If N_highlighted_spiketrains==0 and voltage_monitor is not None, then all voltage traces of the voltage_monitor are plotted. Otherwise N_highlighted_spiketrains voltage traces are plotted.
- **avg_window_width** (*Quantity*) – optional. Before plotting the population rate (PopulationRateMonitor), the rate is smoothed using a window of width = avg_window_width. Defaults is 1.0ms
- **sup_title** (*String*) – figure suptitle. Default is None.
- **figure_size** (*tuple*) – (width,height) tuple passed to pyplot's figsize parameter.

Returns The whole figure Axes: Top panel, Raster plot Axes: Middle panel, population activity Axes: Bottom panel, voltage traces. None if no voltage monitor is provided.

Return type Figure

neurodynex.tools.plot_tools.**plot_population_activity_power_spectrum**(*freq*, *ps*, *max_freq*, *average_At=None*, *plot_f0=False*)

Plots the power spectrum of the population activity A(t)

Parameters

- **freq** – frequencies (= x axis)
- **ps** – power spectrum of the population activity
- **max_freq** (*Quantity*) – The data is plotted in the interval [-.05*max_freq, max_freq]
- **plot_f0** (*bool*) – if true, the power at frequency 0 is plotted. Default is False and the value is not plotted.

Returns the figure

neurodynex.tools.plot_tools.**plot_spike_train_power_spectrum**(*freq*, *mean_ps*, *all_ps*, *max_freq*, *nr_highlighted_neurons=2*, *mean_firing_freqs_per_neuron=None*, *plot_f0=False*)

Visualizes the power spectrum of the spike trains.

Parameters

- **freq** – frequencies (= x axis)
- **mean_ps** – average power taken over all neurons (typically all of a subsample).
- **all_ps** (*dict*) – power spectra for each single neuron
- **max_freq** (*Quantity*) – The x-lim of the plot is [-0.05*max_freq, max_freq]
- **mean_firing_freqs_per_neuron** (*float*) – None or the mean firing rate averaged across the neurons. Default is None in which case the value is not shown in the legend
- **plot_f0** (*bool*) – if true, the power at frequency 0 is plotted. Default is False and the value is not plotted.

Returns all_ps[random_neuron_index]

Return type the figure and the index of the random neuron for which the PS is computed

neurodynex.tools.plot_tools.**plot_voltage_and_current_traces**(*voltage_monitor*, *current*, *title=None*, *firing_threshold=None*, *legend_location=0*)

plots voltage and current .

Parameters

- **voltage_monitor** (*StateMonitor*) – recorded voltage
- **current** (*TimedArray*) – injected current
- **title** (*string, optional*) – title of the figure
- **firing_threshold** (*Quantity, optional*) – if set to a value, the firing threshold is plotted.
- **legend_location** (*int*) – legend location. default = 0 (="best")

Returns the figure

neurodynex.tools.spike_tools module

This the spike_tools submodule provides functions to analyse the Brian2 SpikeMonitors and Brian2 StateMonitors. The code provided here is not optimized for performance and there is no guarantee for correctness.

Relevant book chapters:

- http://neuronaldynamics.epfl.ch/online/Ch19.S2.html#SS1.p6

class neurodynex.tools.spike_tools.**PopulationSpikeStats**(*nr_neurons*, *nr_spikes*, *all_ISI*, *filtered_spike_trains*)

Wraps a few spike-train related properties.

CV
Coefficient of Variation

all_ISI
all ISIs in no specific order

filtered_spike_trains
a time-window filtered copy of the original spike_monitor.all_spike_trains

mean_isi
Mean Inter Spike Interval

nr_neurons
Number of neurons in the original population

nr_spikes
Nr of spikes

std_isi
Standard deviation of the ISI

neurodynex.tools.spike_tools.**filter_spike_trains**(*spike_trains*, *window_t_min=0. * second*, *window_t_max=None*, *idx_subset=None*)

creates a new dictionary neuron_idx=>spike_times where all spike_times are in the half open interval [window_t_min,window_t_max)

Parameters

- **spike_trains** (*dict*) – a dictionary of spike trains. Typically obtained by calling spike_monitor.spike_trains()
- **window_t_min** (*Quantity*) – Lower bound of the time window: t>=window_t_min. Default is 0ms.
- **window_t_max** (*Quantity*) – Upper bound of the time window: t<window_t_max. Default is None, in which case no upper bound is set.
- **idx_subset** (*list, optional*) – a list of neuron indexes (dict keys) specifying a subset of neurons. Neurons NOT in the key list are NOT added to the resulting dictionary. Default is None, in which case all neurons are added to the resulting list.

Returns a filtered copy of spike_trains

neurodynex.tools.spike_tools.**get_averaged_single_neuron_power_spectrum**(*spike_monitor*, *sampling_frequency*, *window_t_min*, *window_t_max*, *nr_neurons_average=100*, *subtract_mean=False*)

averaged power-spectrum of spike trains in the time window [window_t_min, window_t_max). The power spectrum of every single neuron's spike train is computed. Then the average across all single-neuron powers is computed. In order to limit the compuation time, the number of neurons taken to compute the average is limited to nr_neurons_average which defaults to 100

Parameters

- **spike_monitor** (*SpikeMonitor*) – Brian2 SpikeMonitor
- **sampling_frequency** (*Quantity*) – sampling frequency used to discretize the spike trains.
- **window_t_min** (*Quantity*) – Lower bound of the time window: t>=window_t_min. Spikes before window_t_min are not taken into account (set a lower bound if you want to exclude an initial transient in the population activity)
- **window_t_max** (*Quantity*) – Upper bound of the time window: t<window_t_max.
- **nr_neurons_average** (*int*) – Number of neurons over which the average is taken.
- **subtract_mean** (*bool*) – If true, the mean value of the signal is subtracted before FFT. Default is False

Returns freq, mean_ps, all_ps_dict, mean_firing_rate, mean_firing_freqs_per_neuron

neurodynex.tools.spike_tools.**get_population_activity_power_spectrum**(*rate_monitor*, *delta_f*, *k_repetitions*, *T_init=100. * msecond*, *subtract_mean_activity=False*)

Computes the power spectrum of the population activity A(t) (=rate_monitor.rate)

Parameters

- **rate_monitor** (*RateMonitor*) – Brian2 rate monitor. rate_monitor.rate is the signal being analysed here. The temporal resolution is read from rate_monitor.clock.dt
- **delta_f** (*Quantity*) – The desired frequency resolution.
- **k_repetitions** (*int*) – The data rate_monitor.rate is split into k_repetitions which are FFT'd independently and then averaged in frequency domain.
- **T_init** (*Quantity*) – Rates in the time interval [0, T_init] are removed before doing the Fourier transform. Use this parameter to ignore the initial transient signals of the simulation.
- **subtract_mean_activity** (*bool*) – If true, the mean value of the signal is subtracted. Default is False

Returns freqs, ps, average_population_rate

neurodynex.tools.spike_tools.**get_spike_stats**(*voltage_monitor*, *spike_threshold*)

Detects spike times and computes ISI, mean ISI and firing frequency. Note: meanISI and firing frequency are set to numpy.nan if less than two spikes are detected Note: currently only the spike times of the first column in voltage_monitor are detected. Matrix-like monitors are not supported. :param voltage_monitor: A state monitor with at least the fields "v: and "t" :type voltage_monitor: StateMonitor :param spike_threshold: The spike threshold voltage. e.g. -50*b2.mV :type spike_threshold: Quantity

Returns (nr_of_spikes, spike_times, isi, mean_isi, spike_rate)

Return type tuple

neurodynex.tools.spike_tools.**get_spike_time**(*voltage_monitor*, *spike_threshold*)

Detects the spike times in the voltage. The spike time is the value in voltage_monitor.t for which voltage_monitor.v[idx] is above threshold AND voltage_monitor.v[idx-1] is below threshold (crossing from below). Note: currently only the spike times of the first column in voltage_monitor are detected. Matrix-like monitors are not supported. :param voltage_monitor: A state monitor with at least the fields "v: and "t" :type voltage_monitor: StateMonitor :param spike_threshold: The spike threshold voltage. e.g. -50*b2.mV :type spike_threshold: Quantity

Returns A list of spike times (Quantity)

neurodynex.tools.spike_tools.**get_spike_train_stats**(*spike_monitor*, *window_t_min=0. * second*, *window_t_max=None*)

Analyses the spike monitor and returns a PopulationSpikeStats instance.

Parameters

- **spike_monitor** (*SpikeMonitor*) – Brian2 spike monitor
- **window_t_min** (*Quantity*) – Lower bound of the time window: t>=window_t_min. The stats are computed for spikes within the time window. Default is 0ms
- **window_t_max** (*Quantity*) – Upper bound of the time window: t<window_t_max. The stats are computed for spikes within the time window. Default is None, in which case no upper bound is set.

Returns PopulationSpikeStats

neurodynex.tools.spike_tools.**pretty_print_spike_train_stats**(*voltage_monitor*, *spike_threshold*)

Computes and returns the same values as get_spike_stats. Additionally prints these values to the console. :param voltage_monitor: :param spike_threshold:

Returns (nr_of_spikes, spike_times, isi, mean_isi, spike_rate)

Return type tuple

Module contents

neurodynex.working_memory_network package

Submodules

neurodynex.working_memory_network.wm_model module

Implementation of a working memory model. Literature: Compte, A., Brunel, N., Goldman-Rakic, P. S., & Wang, X. J. (2000). Synaptic mechanisms and network dynamics underlying spatial working memory in a cortical network model. Cerebral Cortex, 10(9), 910-923.

Some parts of this implementation are inspired by material from *Stanford University, BIOE 332: Large-Scale Neural Modeling, Kwabena Boahen & Tatiana Engel, 2013*, online available.

Note: Most parameters differ from the original publication.

neurodynex.working_memory_network.wm_model.**getting_started**()

neurodynex.working_memory_network.wm_model.**simulate_wm**(*N_excitatory=1024*, *N_inhibitory=256*, *N_extern_poisson=1000*, *poisson_firing_rate=1.4 * hertz*, *weight_scaling_factor=2.0*, *sigma_weight_profile=20.0*, *Jpos_excit2excit=1.6*, *stimulus_center_deg=180*, *stimulus_width_deg=40*, *stimulus_strength=70. * pamp*, *t_stimulus_start=0. * second*, *t_stimulus_duration=0. * second*, *monitored_subset_size=1024*, *sim_time=0.8 * second*)

Parameters

- **N_excitatory** (*int*) – Size of the excitatory population
- **N_inhibitory** (*int*) – Size of the inhibitory population
- **weight_scaling_factor** (*float*) – weight prefactor. When increasing the size of the populations, the synaptic weights have to be decreased. Using the default values, we have N_excitatory*weight_scaling_factor = 2048 and N_inhibitory*weight_scaling_factor=512
- **N_extern_poisson** (*int*) – Size of the external input population (Poisson input)
- **poisson_firing_rate** (*Quantity*) – Firing rate of the external population
- **sigma_weight_profile** (*float*) – standard deviation of the gaussian input profile in the excitatory population.

- **Jpos_excit2excit** (*float*) – Strength of the recurrent input within the excitatory population. Jneg_excit2excit is computed from sigma_weight_profile, Jpos_excit2excit and the normalization condition.
- **stimulus_center_deg** (*float*) – Center of the stimulus in [0, 360]
- **stimulus_width_deg** (*float*) – width of the stimulus. All neurons in stimulus_center_deg +- (stimulus_width_deg/2) receive the same input current
- **stimulus_strength** (*Quantity*) – Input current to the neurons at stimulus_center_deg +- (stimulus_width_deg/2)
- **t_stimulus_start** (*Quantity*) – time when the input stimulus is turned on
- **t_stimulus_duration** (*Quantity*) – duration of the stimulus.
- **monitored_subset_size** (*int*) – nr of neurons for which a Spike- and Voltage monitor is registered.
- **sim_time** (*Quantity*) – simulation time

Returns

rate_monitor_excit (Brian2 PopulationRateMonitor for the excitatory population), spike_monitor_excit, voltage_monitor_excit, idx_monitored_neurons_excit, rate_monitor_inhib, spike_monitor_inhib, voltage_monitor_inhib, idx_monitored_neurons_inhib, weight_profile_45 (The weights profile for the neuron with preferred direction = 45deg).

Return type results (tuple)

Module contents

Module contents

License

This free software: you can redistribute it and/or modify it under the terms of the GNU General Public License 2.0 as published by the Free Software Foundation. You should have received a copy of the GNU General Public License along with the repository. If not, see http://www.gnu.org/licenses/.

Should you reuse and publish the code for your own purposes, please point to the webpage http://neuronaldynamics.epfl.ch or cite the book: *Wulfram Gerstner, Werner M. Kistler, Richard Naud, and Liam Paninski. Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition. Cambridge University Press, 2014.*

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

Index

H

I

L

M

N

P

R

S

T

W