Neuronaldynamics Exercises Documentation

Release 0.1.1.dev0+ng19946fa.d20161110

Wulfram Gerstner

November 10, 2016

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

1.1 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.

1.1.1 Quickstart

See the indiviual *exercises* - they contain instructions on how to use the python code to solve them.

To install the exercises using `pip` simply execute:

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

To install the exercises with anaconda/miniconda execute:

```
conda install -c brian-team -c epfl-lcn neurodynex
```

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

1.1.2 Brian1

We are currently rewriting the python exercises to use the more recent Brian2 Simulator. The old brian1 exercises are available on the brian1 branch.

1.1.3 Requirements

The following requirements should be met:

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

1.2 Exercises

1.2.1 Setting up Python and Brian

Using python and pip

We provide 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 neurodynex
```

Using anaconda/miniconda

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). To **install or update** the exercise classes for your anaconda environment, it suffices to run:

```
conda install -c brian-team -c epfl-lcn 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.

1.2.2 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`* module contains all code required for this exercise. At the beginning of your exercise solutions, import the contained functions by running

```
from neurodynex.leaky_integrate_and_fire.LIF import *
```

You can then simply run the exercise functions by executing

```
LIF_Step()   # example Step
LIF_Sinus()   # example Sinus
```

Exercise

Use the function *LIF_Step()* to simulate a Leaky Integrate-And-Fire neuron stimulated by a current step of a given amplitude. The goal of this exercise is to modify the provided python functions and use the `numpy` and `matplotlib` packages to answer the following questions.

Question

What is the minimum current step amplitude `I_amp` to elicit a spike with model parameters as given in *LIF_Step()*? Plot the injected values of current step amplitude against the frequency of the spiking response (you can use the inter-spike interval to calculate this – let the frequency be $0Hz$ if the model does not spike, or emits only a single spike) during a $500ms$ current step.

Exercise

Use the function *LIF_Sinus()* to simulate a Leaky Integrate-And-Fire neuron stimulated by a sinusoidal current of a given frequency. The goal of this exercise is to modify the provided python functions and use the `numpy` and `matplotlib` packages to plot the amplitude and frequency gain and phase of the voltage oscillations as a function of the input current frequency.

Question

For input frequencies between $0.1kHz$ and $1.kHz$, plot the input frequency against the resulting *amplitude of subthreshold oscillations* of the membrane potential. If your neuron emits spikes at high stimulation frequencies, decrease the amplitude of the input current.

Question

For input frequencies between $0.1kHz$ and $1.kHz$, plot the input frequency against the resulting *frequency and phase of subthreshold oscillations* of the membrane potential. Again, keep your input amplitude in a regime, where the neuron does not fire action potentials.

1.2.3 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. At the beginning of your exercise solutions, import the contained functions by running

```
from neurodynex.hodgkin_huxley.HH import *
```

You can then simply run the exercise functions by executing

```
HH_Step()   # example Step-current injection
HH_Sinus()   # example Sinus-current injection
HH_Ramp()   # example Ramp-current injection
```

Exercise

Use the function *HH_Step()* to simulate a HH neuron stimulated by a current step of a given amplitude. The goal of this exercise is to modify the provided python functions and use the `numpy` and `matplotlib` packages to answer the following questions.

Question

What is the lowest step current amplitude for generating at least one spike? Hint: use binary search on `I_amp`, with a $0.1\mu A$ resolution.

Question

What is the lowest step current amplitude to generate repetitive firing?

Question

Look at *HH_Step()* for `I_amp = -5` and `I_amp = -1`. What is happening here? To which gating variable do you attribute this rebound spike?

Exercise

Use the function *HH_Ramp()* to simulate a HH neuron stimulated by a ramping curent.

Question

What is the minimum current required to make a spike when the current is slowly increased (ramp current waveform) instead of being increased suddenly?

Exercise

To solve this exercise, you will need to change the actual implementation of the model. Download directly the source file HH.py. When starting Python in the directory containing the downloaded file, you run functions from it directly as follows:

```
import HH  # import the HH module, i.e. the HH.py file
HH.HH_Step()  # access the LIF_Step function in HH.py
```

Then use any text editor to make changes in the `HH.py` file.

Note: You might have to reload the module for changes to become active - quitting and restarting the python interpreter reloads all modules. Alternatively, you can also force a reload by typing

```
reload(HH)
```

For automatic reloading you can also run `ipython` instead of `python` and set the autoreload flag. For this, make sure you have ipython installed - if you have followed *the setup instructions for anaconda/miniconda* this should already work.

Question

What is the current threshold for repetitive spiking if the density of sodium channels is increased by a factor of 1.5? To solve this, change the maximum conductance of sodium channel in *HH_Neuron()*.

1.2.4 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

The *phase_plane_analysis.fitzhugh_nagumo* module contains all code required for this exercise. At the beginning of your exercise solutions, import the contained functions by running

```
from neurodynex.phase_plane_analysis.fitzhugh_nagumo import *
```

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

```
get_trajectory()
get_fixed_point()
plot_flow()
```

Exercise: Phase plane analysis

Create a script file (e.g. `answers.py`) and add the following header:

```
from neurodynex.phase_plane_analysis.fitznagumo import *

# your code here ..
```

You will type the code for your answers right below, adding more code with each exercise. When you want to execute your code, open ipython in a terminal and type

```
run answers.py
```

Or alternatively, from the command line execute

```
python answers.py
```

For some exercises you will have to plot and analyze data. This can be done by importing `matplotlib` and `numpy`:

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

Question

Use the function `plt.plot` to plot the two nullclines of the Fitzhugh-Nagumo system given in Eq. (1.1) 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 = 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 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. (1.1) 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

Consider the following two-dimensional Fitzhugh-Nagumo model:

$$\left[\begin{array}{rcl} \frac{du}{dt} & = & u\left(1-u^2\right) - w + I \equiv F(u,w) \\ \frac{dw}{dt} & = & \varepsilon\left(u - 0.5w + 1\right) \equiv \varepsilon G(u,w)\,, \end{array}\right. \tag{1.1}$$

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. (1.1). We will find that the linear stability changes as the input current crosses a critical value.

Set $\varepsilon = .1$. Create the variable I and set it to zero for the moment.

Question

The Jacobian of Eq. (1.1) as a function of the fixed point is given by

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

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 $M(a,b) = \left(\begin{array}{cc} a & 1 \\ 0 & b \end{array}\right)$ 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

```
list1 = []
list2 = []
currents = 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?

1.2.5 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.typeXY` module contains all classes required for this exercise. For the exercises you will need to import the classes `NeuronX` and `NeuronY` by running

```
from neurodynex.neuron_type.typeXY import NeuronX, NeuronY
```

Note: Both `NeuronX` and `NeuronY` inherit from a common base class `neuron_type.neurons.NeuronAbstract` and thus implement similar methods.

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.

To run the exercises you will have to instantiate these classes. You can then plot step_current injections (using the `step` method) or extract the firing rate for a given step current (using the `get_rate` method):

```
from neurodynex.neuron_type.typeXY import NeuronX, NeuronY

n1 = NeuronX()  # instantiates a new neuron of type X

n1.step(do_plot=True)  # plot a step current injection
```

To check your results, you can use the `get_neuron_type` function, e.g.:

```
>> n1 = NeuronX()  # instantiates a new neuron of type X
>> n1.get_neuron_type()
neurodynex.neuron_type.neurons.NeuronTypeOne
```

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.

Note: As shown abve, use the `step` functions to 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):

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

Question: Pulse response

Plot the response to short current pulses near threshold, and interpret the results: which class is Type I, which is II?

For example:

```
import matplotlib.pyplot as plt
plt.figure()  # new figure
n1 = NeuronX()  # instantiates a new neuron of type X

t, v, w, I = n1.step(I_amp=1.05, I_tstart=100, I_tend=110, t_end=300)
plt.plot(t,v)

t, v, w, I = n1.step(I_amp=1.1, I_tstart=100, I_tend=110, t_end=300)
plt.plot(t,v)

# can you simplify this in a loop?

plt.show()
```

Exercise: f-I curves

During the questions of 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 *get_spiketimes* to determine the spike times from given timeseries `t` and `v`:

```
>> from neurodynex.neuron_type.neurons import get_spiketimes
>> t, v, w, I = n1.step(I_amp=1.0, I_tstart=100, I_tend=1000., t_end=1000.)
>> st = get_spiketimes(t, v)
>> print st
[ 102.9  146.1   189.1 ... ]
```

Use this function to write a Python function (in your own *.py* file) that calculates an estimate of the firing rate, given a neuron instance and an input current:

```
def get_firing_rate(neuron, I_amp):

    # run a step on the neuron via neuron.step()
    # get 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 and take the reciprocal (pay attention when converting from 1/ms to Hz) to yield the firing-rate:

```
f = 1000.0/mean(isi)
```

Note: You can check your results by calling:

```
# get firing rate and plot the dynamics for an injection of I_amp
n1.get_rate(I_amp, do_plot=True)
```

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(0.0,1.05,0.1)  # 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 to zoom in near the threshold, and try running it again for both classes.

Which class is Type I and which is Type II?

1.2.6 Hopfield Network model of associative memory

Book chapters

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

Python classes

The *`hopfield_network.hopfield`* module contains all code required for this exercise. At the beginning of your exercise solutions, import the contained functions by running

```
from neurodynex.phase_plane_analysis.fitzhugh_nagumo import *
```

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

```
get_trajectory()
get_fixed_point()
plot_flow()
```

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:

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

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: 4x4 Hopfield-network

This exercise deals not only with Python functions, but with Python classes and objects. The class `HopfieldNetwork` implements a Hopfield network. To run the exercises you will have to instantiate the network:

```
from neurodynex.hopfield_network.hopfield import HopfieldNetwork
n = HopfieldNetwork(4)   # instantiates a new HopfieldNetwork
```

Note: See the `documentation for the HopfieldNetwork class` to see all methods you can use on a instantiated HopfieldNetwork.

Storing patterns

Create an instance of the `HopfieldNetwork` with N=4. Use the `make_pattern` method to store a pattern (default is one random pattern with half of its pixels *on*) and test whether it can be retrieved with the `run` method:

```
n.run()  # Note: this will fail with a RuntimeError if no patterns have been stored before
```

The `run` method, by defaults, runs the dynamics for the first pattern with no pixel flipped.

Question: Capacity of the 4x4 network

What is the experimental maximum number of random patterns the 4x4 network is able to memorize?

Store more and more random patterns and test retrieval of some of them. The first few patterns should be stored perfectly, but then the performance gets worse.

Does this correspond to the theoretical maximum number of random patterns the network should be able to memorize?

Exercise: 10x10 Hopfield-network

Question: Capacity of the 10x10 network

Increase the network size to 10x10 and repeat the steps of the previous exercise.

Question: Error correction

Instatiate a network and store a finite number of random patterns, e.g. 8.

How many wrong pixels can the network tolerate in the initial state, such that it still settles into the correct pattern?

Note: See the documentation for the *run method* to see how to control which percentage of pixels is flipped.

Question: Storing alphabet letters

Try to store alphabetic characters as the relevant patterns. How good is the retrieval of patterns? What is the reason?

Note: See the documentation for the *make_pattern method* on how to store alphabet characters.

Exercise: Bonus

Try one of the preceding points in bigger networks.

Try downloading the source code for the network, and modify it by adding a smooth transfer function g to the neurons. A short introducion on how to run the downloaded file *can be found here*.

1.2.7 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

```
from neurodynex.ojas_rule.oja import *
```

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

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

Exercise: Circular data

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?

Exercise: Elliptic data

Repeat the previous question with an **elongated** elliptic data cloud (e.g. `ratio=0.3`). Again, repeat this several times.

Question

What difference in terms of learning do you observe with respect to the circular data clouds?

Question

Try to change the orientation of the ellipsoid (try several different angles). Can you explain what Oja's rule does?

Note: To gain more insight, plot the learned weight vector in 2D space, and relate its orientation to that of the ellipsoid of data clouds.

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 `5 + make_cloud(...)`, which centers the data around `(5,5)`. What conclusions can you draw? Can you think of a modification to the learning rule?

1.3 Python exercise modules

All exercises are contained in subpackages of the python package *neurodynex*. The subpackages contain modules used for each exercise, along with a file called *exercise.pdf* with the actual exercises using the python code.

1.3.1 neurodynex package

Subpackages

neurodynex.hodgkin_huxley package

Submodules

neurodynex.hodgkin_huxley.HH module This file implements Hodgkin-Huxley (HH) model. You can inject a step current, sinusoidal current or ramp current into neuron using HH_Step(), HH_Sinus() or HH_Ramp() methods respectively.

Relevant book chapters:

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

`neurodynex.hodgkin_huxley.HH.`**`HH_Neuron`**(*curr*, *simtime*)

Simple Hodgkin-Huxley neuron implemented in Brian2.

Parameters

- **`curr`** (*`TimedArray`*) – Input current injected into the HH neuron
- **`simtime`** (*`float`*) – Simulation time [seconds]

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

Return type StateMonitor

`neurodynex.hodgkin_huxley.HH.`**`HH_Ramp`**(*I_tstart=30*, *I_tend=270*, *I_amp=20.0*, *tend=300*, *dt=0.1*, *do_plot=True*)

Run the HH model for a sinusoidal current

Parameters

- **`tend`** (*`float, optional`*) – the simulation time of the model [ms]
- **`I_tstart`** (*`float, optional`*) – start of current ramp [ms]
- **`I_tend`** (*`float, optional`*) – end of the current ramp [ms]
- **`I_amp`** (*`float, optional`*) – final amplitude of current ramp [uA]
- **`do_plot`** (*`bool, optional`*) – plot the resulting simulation

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

Return type StateMonitor

`neurodynex.hodgkin_huxley.HH.`**`HH_Sinus`**(*I_freq=0.01*, *I_offset=0.5*, *I_amp=7.0*, *tend=600*, *dt=0.1*, *do_plot=True*)

Run the HH model for a sinusoidal current

Parameters

- **`tend`** (*`float, optional`*) – the simulation time of the model [ms]
- **`I_freq`** (*`float, optional`*) – frequency of current sinusoidal [kHz]
- **`I_offset`** (*`float, optional`*) – DC offset of current [nA]
- **`I_amp`** (*`float, optional`*) – amplitude of sinusoidal [nA]
- **`do_plot`** (*`bool, optional`*) – plot the resulting simulation

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

Return type StateMonitor

`neurodynex.hodgkin_huxley.HH.`**`HH_Step`**(*I_tstart=20*, *I_tend=180*, *I_amp=7*, *tend=200*, *do_plot=True*)

Run the Hodgkin-Huley neuron for a step current input.

Parameters

- **`I_tstart`** (*`float, optional`*) – start of current step [ms]

- **I_tend** (*float, optional*) – start of end step [ms]
- **I_amp** (*float, optional*) – amplitude of current step [uA]
- **tend** (*float, optional*) – the simulation time of the model [ms]
- **do_plot** (*bool, optional*) – plot the resulting simulation

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

Return type StateMonitor

`neurodynex.hodgkin_huxley.HH.`**`plot_data`**`(`*rec*, *title=None*`)`

Plots a TimedArray for values I and v

Parameters

- **rec** (*TimedArray*) – the data to plot
- **title** (*string, optional*) – plot title to display

Module contents

neurodynex.hopfield_network package

Submodules

neurodynex.hopfield_network.hopfield module This file implements a Hopfield Network model.

Relevant book chapters:

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

class `neurodynex.hopfield_network.hopfield.`**`HopfieldNetwork`**`(`*N*`)`

Implements a Hopfield network of size N.

`N`

int

Square root of number of neurons

`patterns`

numpy.ndarray

Array of stored patterns

`weight`

numpy.ndarray

Array of stored weights

`x`

numpy.ndarray

Network state (of size N**2)

`dynamic``()`

Executes one timestep of the dynamics

`grid``(`*mu=None*`)`

Reshape an array of length NxN to a matrix NxN

Parameters **mu** (*TYPE, optional*) – If None, return the reshaped network state. For an integer i < P, return the reshaped pattern i.

Returns Reshaped network state or pattern

Return type numpy.ndarray

make_pattern(*P=1*, *ratio=0.5*, *letters=None*)

Creates and stores additional patterns to the network.

Parameters

- **P** (*int, optional*) – number of patterns (used only for random patterns)
- **ratio** (*float, optional*) – percentage of 'on' pixels for random patterns
- **letters** (*TYPE, optional*) – to store characters use as input a string with the desired letters. Example: `make_pattern(letters='abcdjft')`

Raises `ValueError` – Raised if N!=10 and letters!=None. For now letters are hardcoded for N=10.

overlap(*mu*)

Computes the overlap of the current state with pattern number mu.

Parameters **mu** (*int*) – The index of the pattern to compare with.

run(*t_max=20*, *mu=0*, *flip_ratio=0*, *do_plot=True*)

Runs the dynamics and optionally plots it.

Parameters

- **t_max** (*float, optional*) – Timesteps to simulate
- **mu** (*int, optional*) – Pattern number to use as initial pattern for the network state (< P)
- **flip_ratio** (*int, optional*) – ratio of randomized pixels. For example, to run pattern #5 with 5% flipped pixels use `run(mu=5,flip_ratio=0.05)`
- **do_plot** (*bool, optional*) – Plot the network as it is updated

Raises

- `IndexError` – Raised if given pattern index is too high.
- `RuntimeError` – Raised if no patterns have been created.

`neurodynex.hopfield_network.hopfield.`**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.

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.`**`LIF_Neuron`**(*curr*, *simtime*)

Simple LIF neuron implemented in Brian2.

Parameters

- **`curr`** (*`TimedArray`*) – Input current injected into the neuron
- **`simtime`** (*`float`*) – Simulation time [seconds]

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

Return type StateMonitor

`neurodynex.leaky_integrate_and_fire.LIF.`**`LIF_Sinus`**(*I_freq=0.1*, *I_offset=0.5*, *I_amp=0.5*, *tend=100*, *dt=0.1*, *do_plot=True*)

Run the LIF for a sinusoidal current

Parameters

- **`tend`** (*`float, optional`*) – the simulation time of the model [ms]
- **`I_freq`** (*`float, optional`*) – frequency of current sinusoidal [kHz]
- **`I_offset`** (*`float, optional`*) – DC offset of current [nA]
- **`I_amp`** (*`float, optional`*) – amplitude of sinusoidal [nA]
- **`do_plot`** (*`bool, optional`*) – plot the resulting simulation

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

Return type StateMonitor

`neurodynex.leaky_integrate_and_fire.LIF.`**`LIF_Step`**(*I_tstart=20*, *I_tend=70*, *I_amp=1.005*, *tend=100*, *do_plot=True*)

Run the LIF and give a step current input.

Parameters

- **`tend`** (*`float, optional`*) – the simulation time of the model [ms]
- **`I_tstart`** (*`float, optional`*) – start of current step [ms]
- **`I_tend`** (*`float, optional`*) – start of end step [ms]
- **`I_amp`** (*`float, optional`*) – amplitude of current step [nA]
- **`do_plot`** (*`bool, optional`*) – plot the resulting simulation

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

Return type StateMonitor

`neurodynex.leaky_integrate_and_fire.LIF.`**`plot_data`**(*rec*, *v_threshold=1.0*, *title=None*)

Plots a TimedArray for values I and v

Parameters

- **`rec`** (*`TimedArray`*) – the data to plot

- **v_threshold** (*float*) – plots a threshold at this level [mV]
- **title** (*string*) – plot title to display

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_rate`(*I_amp*, *t_end=1000.0*, *do_plot=False*)

Return the firing rate under a current step.

Parameters

- **NeuronClass** (*type*) – Subclass of neurons.AbstractNeuron
- **I_amp** (*float*) – Amplitude of voltage step
- **t_end** (*float*) – Length of simulation
- **do_plot** (*bool, optional*) – plot the results

Returns firing rate of neuron

Return type float

`make_neuron`()

Abstract function, which creates neuron attribute for this class.

`run`(*curr*, *simtime*)

Runs the neuron for a given current.

Parameters

- **curr** (*TimedArray*) – Input current injected into the neuron
- **simtime** (*float*) – Simulation time [seconds]

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

Return type StateMonitor

`step`(*t_end=300.0*, *I_tstart=20*, *I_tend=270*, *I_amp=0.5*, *do_plot=True*, *show=True*)

Runs the neuron for a step current and plots the data.

Parameters

- **t_end** (*float, optional*) – the simulation time of the model [ms]
- **I_tstart** (*float, optional*) – start of current step [ms]
- **I_tend** (*float, optional*) – start of end step [ms]
- **I_amp** (*float, optional*) – amplitude of current step [nA]
- **do_plot** (*bool, optional*) – plot the resulting simulation
- **show** (*bool, optional*) – call plt.show for the plot

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

Return type StateMonitor

class `neurodynex.neuron_type.neurons.`**`NeuronTypeOne`**

Bases: *neurodynex.neuron_type.neurons.NeuronAbstract*

`make_neuron``()`

Sets the self.neuron attribute.

class `neurodynex.neuron_type.neurons.`**`NeuronTypeTwo`**

Bases: *neurodynex.neuron_type.neurons.NeuronAbstract*

`make_neuron``()`

Sets the self.neuron attribute.

`neurodynex.neuron_type.neurons.`**`get_spiketimes`**(*t*, *v*, *v_th=0.5*, *do_plot=False*)

Returns numpy.ndarray of spike times, for a given time and voltage series.

Parameters

- **t** (*numpy.ndarray*) – time dimension of timeseries [ms]
- **v** (*numpy.ndarray*) – voltage dimension of timeseries [mV]
- **v_th** (*float, optional*) – threshold voltage for spike detection [mV]
- **do_plot** (*bool, optional*) – plot the results

Returns detected spike times

Return type np.ndarray

`neurodynex.neuron_type.neurons.`**`get_step_curr`**(*I_tstart=20*, *I_tend=270*, *I_amp=0.5*)

Returns a pA step current TimedArray.

Parameters

- **I_tstart** (*float, optional*) – start of current step [ms]
- **I_tend** (*float, optional*) – start of end step [ms]
- **I_amp** (*float, optional*) – amplitude of current step [pA]

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

Return type StateMonitor

`neurodynex.neuron_type.neurons.`**`plot_data`**(*rec*, *title=None*, *show=False*)

Plots a TimedArray for values I, v and w

Parameters

- **rec** (*TimedArray*) – the data to plot
- **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

`neurodynex.neuron_type.neurons.`**`rec_to_tuple`**(*rec*)

Extracts a tuple of numpy arrays from a brian2 StateMonitor.

Parameters **rec** (*StateMonitor*) – state monitor with v, w, I recorded

Returns (t, v, w, I) tuple of numpy.ndarrays

Return type tuple

neurodynex.neuron_type.typeXY module This module has two neuron models, NeuronX and NeuronY. One of them is Type I, the other is Type II - the assignment is randomly generated when the module is loaded.

Relevant book chapters:

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

class `neurodynex.neuron_type.typeXY.`**`NeuronX`**

Bases: *neurodynex.neuron_type.neurons.NeuronTypeOne*

classmethod **`get_neuron_type`**(*x*)

Returns the underlying neuron type.

Returns Class of the underlying neuron model

Return type type

class `neurodynex.neuron_type.typeXY.`**`NeuronY`**

Bases: *neurodynex.neuron_type.neurons.NeuronTypeTwo*

classmethod **`get_neuron_type`**(*x*)

Returns the underlying neuron type.

Returns Class of the underlying neuron model

Return type type

`neurodynex.neuron_type.typeXY.`**`create_models`**()

Creates classes NeuronX and NeuronY in this module that are random assignments of Type1 and Type2 neuron models.

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=0.001*)

Run one batch of Oja's learning over a cloud of datapoints

Parameters

- **cloud** (*numpy.ndarray*) – array of datapoints
- **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=10000*, *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.`**`run_oja`**(*n=10000*, *ratio=1.0*, *angle=0.0*, *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_HH module

`neurodynex.test.test_HH.`**`test_runnable_Ramp`**()

Test if HH_Ramp is runnable.

`neurodynex.test.test_HH.`**`test_runnable_Sinus`**()

Test if HH_Sinus is runnable.

`neurodynex.test.test_HH.`**`test_runnable_Step`**()

Test if HH_Step is runnable.

neurodynex.test.test_LIF module

`neurodynex.test.test_LIF.`**`test_runnable_Sinus`**`()`
  Test if LIF_Sinus is runnable.

`neurodynex.test.test_LIF.`**`test_runnable_Step`**`()`
  Test if LIF_Step is runnable.

neurodynex.test.test_hopfield module

`neurodynex.test.test_hopfield.`**`test_alphabet`**`()`
  Test if alphabet is loadable.

`neurodynex.test.test_hopfield.`**`test_net_alphabet`**`()`
  Test hopfield network with alphabet patterns.

`neurodynex.test.test_hopfield.`**`test_net_random`**`()`
  Test hopfield network with random patterns.

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_nagumo.`**`test_runnable_plot_flow`**`()`
  Test if plot_flow is runnable.

neurodynex.test.test_neuron_type module

`neurodynex.test.test_neuron_type.`**`run_neuron`**`(`*c*`)`

`neurodynex.test.test_neuron_type.`**`test_class_assignment`**`()`
  Test if NeuronX and NeuronY are properly assigned to NeuronTypeOne and NeuronTypeTwo.

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

neurodynex.test.test_oja module

`neurodynex.test.test_oja.`**`test_oja`**`()`
  Test if Oja learning rule is runnable.

Module contents

Module contents

1.4 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