Cleo

Kyle Johnsen, Nathan Cruzado

Sep 01, 2022

CONTENTS

Hello there! Cleo has the goal of bridging theory and experiment for mesoscale neuroscience, facilitating electrode recording, optogenetic stimulation, and closed-loop experiments (e.g., real-time input and output processing) with the Brian 2 spiking neural network simulator. We hope users will find these components useful for prototyping experiments, innovating methods, and testing observations about a hypotheses *in silico*, incorporating into spiking neural network models laboratory techniques ranging from passive observation to complex model-based feedback control. Cleo also serves as an extensible, modular base for developing additional recording and stimulation modules for Brian simulations.

This package was developed by Kyle Johnsen and Nathan Cruzado under the direction of Chris Rozell at Georgia Institute of Technology.

CHAPTER ONE

CLOSED LOOP PROCESSING

Cleo allows for flexible I/O processing in real time, enabling the simulation of closed-loop experiments such as event-triggered or feedback control. The user can also add latency to closed-loop stimulation to study the effects of computation delays.

CHAPTER

TWO

ELECTRODE RECORDING

Cleo provides functions for configuring electrode arrays and placing them in arbitrary locations in the simulation. The user can then specify parameters for probabilistic spike detection or a spike-based LFP approximation developed by Teleńczuk et al., 2020.

CHAPTER

THREE

OPTOGENETIC STIMULATION

By providing an optic fiber-light propagation model, Cleo enables users to flexibly add photostimulation to their model. Both a four-state Markov state model of opsin dynamics is available, as well as a minimal proportional current option for compatibility with simple neuron models. Parameters are provided for the common blue light/ChR2 setup.

CHAPTER

FOUR

GETTING STARTED

Just use pip to install—the name on PyPI is `cleosim`:

```
pip install cleosim
```

Then head to the overview section of the documentation for a more detailed discussion of motivation, structure, and basic usage.

CHAPTER FIVE

RELATED RESOURCES

Those using Cleo to simulate closed-loop control experiments may be interested in software developed for the execution of real-time, *in-vivo* experiments. Developed by members of Chris Rozell's and Garrett Stanley's labs at Georgia Tech, the CLOCTools repository can serve these users in two ways:

1. By providing utilities and interfaces with experimental platforms for moving from simulation to reality.
2. By providing performant control and estimation algorithms for feedback control. Although Cleo enables closed-loop manipulation of network simulations, it does not include any advanced control algorithms itself. The `ldsCtrlEst` library implements adaptive linear dynamical system-based control while the `hmm` library can generate and decode systems with discrete latent states and observations.

5.1 Publications

CLOC Tools: A Library of Tools for Closed-Loop Neuroscience A.A. Willats, M.F. Bolus, K.A. Johnsen, G.B. Stanley, and C.J. Rozell. *In prep*, 2022.

State-Aware Control of Switching Neural Dynamics A.A. Willats, M.F. Bolus, C.J. Whitmire, G.B. Stanley, and C.J. Rozell. *In prep*, 2022.

Closed-Loop Identifiability in Neural Circuits A. Willats, M. O'Shaughnessy, and C. Rozell. *In prep*, 2022.

State-space optimal feedback control of optogenetically driven neural activity M.F. Bolus, A.A. Willats, C.J. Rozell and G.B. Stanley. *Journal of Neural Engineering*, 18(3), pp. 036006, March 2021.

Design strategies for dynamic closed-loop optogenetic neurocontrol in vivo M.F. Bolus, A.A. Willats, C.J. Whitmire, C.J. Rozell and G.B. Stanley. *Journal of Neural Engineering*, 15(2), pp. 026011, January 2018.

CHAPTER

SIX

DOCUMENTATION CONTENTS

6.1 Overview

6.1.1 Introduction

Who is this package for?

Cleo (Closed Loop, Electrophysiology, and Optogenetics Simulator) is a Python package developed to bridge theory and experiment for mesoscale neuroscience. We envision two primary uses cases:

1. For prototyping closed-loop control of neural activity *in silico*. Animal experiments are costly to set up and debug, especially with the added complexity of real-time intervention—our aim is to enable researchers, given a decent spiking model of the system of interest, to assess whether the type of control they desire is feasible and/or what configuration(s) would be most conducive to their goals.
2. The complexity of experimental interfaces means it's not always clear what a model would look like in a real experiment. Cleo can help anyone interested in observing or manipulating a model while taking into account the constraints present in real experiments. Because Cleo is built around the Brian simulator, we especially hope this is helpful for existing Brian users who for whatever reason would like a convenient way to inject recorders (e.g., electrodes) or stimulators (e.g., optogenetics) into the core network simulation.

What is closed-loop control?

In short, determining the inputs to deliver to a system from its outputs. In neuroscience terms, making the stimulation parameters a function of the data recorded in real time.

Structure and design

Cleo wraps a spiking network simulator and allows for the injection of stimulators and/or recorders. The models used to emulate these devices are often non-trivial to implement or use in a flexible manner, so Cleo aims to make device injection and configuration as painless as possible, requiring minimal modification to the original network.

Cleo also orchestrates communication between the simulator and a user-configured `IOProcessor` object, modeling how experiment hardware takes samples, processes signals, and controls stimulation devices in real time.

For an explanation of why we choose to prioritize spiking network models and how we chose Brian as the underlying simulator, see *Design rationale*.

Why closed-loop control in neuroscience?

Fast, real-time, closed-loop control of neural activity enables intervention in processes that are too fast or unpredictable to control manually or with pre-defined stimulation, such as sensory information processing, motor planning, and oscillatory activity. Closed-loop control in a *reactive* sense enables the experimenter to respond to discrete events of interest, such as the arrival of a traveling wave or sharp wave ripple, whereas *feedback* control deals with driving the system towards a desired point or along a desired state trajectory. The latter has the effect of rejecting noise and disturbances, reducing variability across time and across trials, allowing the researcher to perform inference with less data and on a finer scale. Additionally, closed-loop control can compensate for model mismatch, allowing it to reach more complex targets where open-loop control based on imperfect models is bound to fail.

6.1.2 Installation

Make sure you have Python >=3.7, then use pip: `pip install cleosim`.

Note: The name on PyPI is `cleosim` since `cleo` was already taken, but in code it is still used as `import cleo`. The other Cleo appears to actually be a fairly well developed package, so I'm sorry if you need to use it along with this Cleo in the same environment. In that case, there are workarounds.

Or, if you're a developer, install poetry and run `poetry install` from the repository root.

6.1.3 Usage

Brian network model

The starting point for using Cleo is a Brian spiking neural network model of the system of interest. For those new to Brian, the docs are a great resource. If you have a model built with another simulator or modeling language, you may be able to import it to Brian via NeuroML.

Perhaps the biggest change you may have to make to an existing model to make it compatible with Cleo's optogenetics and electrode recording is to give the neurons of interest coordinates in space. See the *Tutorials* or the `cleo.coords` module for more info.

You'll need your model in a Brian `Network` object before you move on. E.g.,:

```
net = brian2.Network(...)
```

CLSimulator

Once you have a network model, you can construct a `CLSimulator` object:

```
sim = cleo.CLSimulator(net)
```

The simulator object wraps the Brian network and coordinates device injection, processing input and output, and running the simulation.

Recording

Recording devices take measurements of the Brian network. Some extremely simple implementations (which do little more than wrap Brian monitors) are available in the `cleo.recorders` module.

To use a `Recorder`, you must inject it into the simulator via `inject_recorder()`:

```
rec = MyRecorder('recorder_name', ...)  # note that all devices need a unique name
sim.inject_recorder(rec, neuron_group1, neuron_group2, ...)  # can pass in additional␣
↪arguments
```

The recorder will only record from the neuron groups specified on injection, allowing for such scenarios as singling out a cell type to record from.

Electrodes

Electrode recording is the main recording modality currently implemented in Cleo. See the *Electrode recording* tutorial for more detail, but in brief, usages consists of:

1. Constructing a `Probe` object with coordinates at the desired contact locations
   - Convenience functions for generating shank probe coordinates exist. See *Specifying electrode coordinates*.
2. Specifying the signals to be recorded. Currently there are three implemented. See *Specifying signals to record*.
   - Multi-unit activity
   - Sorted spikes
   - TKLFP: Teleńczuk kernel approximation of LFP
3. Injection into the simulator

Stimulation

Stimulator devices manipulate the Brian network. Usage is similar to recorders:

```
stim = MyStimulator('stimulator_name', ...)  # again, all devices need a unique name
# again, specify neuron groups device will affect and any additional arguments needed
sim.inject_stimulator(stim, neuron_group1, neuron_group2, ...)
```

As with recorders, you can inject stimulators per neuron group to produce a targeted effect.

Optogenetics

Optogenetics is the main stimulator device currently implemented by Cleo. This take the form of an `OptogeneticIntervention`, which, on injection, adds a light source at the specified location and transfects the neurons (via Brian "synapses" that deliver current according to an opsin model, leaving the neuron model equations untouched).

Out of the box you can access a four-state Markov model of channelrhodopsin-2 (ChR2) and parameters for a 473-nm blue optic fiber light source.:

```python
from cleo.opto import *
opto = OptogeneticIntervention(
    name="...",
    opsin_model=FourStateModel(params=ChR2_four_state),
    light_model_params=default_blue,
    location=(0, 0, 0.5) * mm,
)
```

Note, however, that Markov opsin dynamics models require target neurons to have membrane potentials in realistic ranges and an *Iopto* term defined in amperes. If you need to interface with a model without these features, you may want to use the simplified `ProportionalCurrentModel`. You can find more details, including a comparison between the two model types, in the *optogenetics tutorial*.

These model and parameter settings were designed to be flexible enough that an interested user should be able to imitate and replace them with other opsins, light sources, etc. See the *Optogenetic stimulation* tutorial for more detail.

IO Processor

Just as in a real experiment where the experiment hardware must be connected to signal processing equipment and/or computers for recording and control, the `CLSimulator` must be connected to an `IOProcessor`:

```python
sim.set_io_processor(...)
```

If you are only recording, you may want to use the `RecordOnlyProcessor`. Otherwise you will want to implement the `LatencyIOProcessor`, which not only takes samples at the specified rate, but processes the data and delivers input to the network after a user-defined delay, emulating the latency inherent in real experiments. You define your processor by creating a subclass and defining the `process()` function:

```python
class MyProcessor(LatencyIOProcessor):

    def process(self, state_dict, sample_time_ms):
        # state_dict contains a {'recorder_name': value} dict of network
        foo = state_dict['foo_recorder']
        out = ... # do something with sampled spikes
        delay_ms = 3
        t_out_ms = sample_time_ms + delay_ms
        # output must be a {'stimulator_name': value} dict setting stimulator values
        return {'stim': out}, t_out_ms

my_proc = MyProcessor(sample_period_ms=1)
sim.set_io_processor(my_proc)
```

See *On-off control* for a minimal working example or *PI control* for more advanced features, including decomposing the processing into blocks with accompanying stochastic delay objects.

Running experiments

Use CLSimulator's *run()* function with the desired duration:

```
sim.run(500*ms, ...)  # kwargs are passed to Brian's run function
```

Use CLSimulator's *reset()* function to restore the default state (right after initialization/injection) for the network and all devices. This could be useful for running a simulation multiple times under different conditions.

To facilitate access to data after the simulation, many classes offer a `save_history` option on construction. If true, that object will store relevant variables as attributes. For example,:

```
sorted_spikes = cleo.ephys.SortedSpiking(...)
...
sim.run(...)

plt.plot(sorted_spikes.t_ms, sorted_spikes.i)
```

6.1.4 Design rationale

Why not prototype with more abstract models?

Cleo aims to be practical, and as such provides models at the level of abstraction corresponding to the variables the experimenter has available to manipulate. This means models of spatially defined, spiking neural networks.

Of course, neuroscience is studied at many spatial and temporal scales. While other projects may be better suited for larger segments of the brain and/or longer timescales (such as HNN or BMTK's PopNet or FilterNet), this project caters to finer-grained models because they can directly simulate the effects of alternate experimental configurations. For example, how would the model change when swapping one opsin for another, using multiple opsins simultaneously, or with heterogeneous expression? How does recording or stimulating one cell type vs. another affect the experiment? Would using a more sophisticated control algorithm be worth the extra compute time, and thus later stimulus delivery, compared to a simpler controller?

Questions like these could be answered using an abstract dynamical system model of a neural circuit, but they would require the extra step of mapping the afore-mentioned details to a suitable abstraction—e.g., estimating a transfer function to model optogenetic stimulation for a given opsin and light configuration. Thus, we haven't emphasized these sorts of models so far in our development of Cleo, though they should be possible to implement in Brian if you are interested. For example, one could develop a Poisson linear dynamical system (PLDS), record spiking output, and configure stimulation to act directly on the system's latent state.

And just as experiment prototyping could be done on a more abstract level, it could also be done on an even more realistic level, which we did not deem necessary. That brings us to the next point…

Why Brian?

Brian is a relatively new spiking neural network simulator written in Python. Here are some of its advantages:

- Flexibility: allowing (and requiring!) the user to define models mathematically rather than selecting from a pre-defined library of cell types and features. This enables us to define arbitrary models for recorders and stimulators and easily interface with the simulation
- Ease of use: it's all just Python
- Speed

NEST is a popular alternative to Brian also strong in point neuron simulations. However, it appears to be less flexible, and thus harder to extend. NEURON is another popular alternative to Brian. Its main advantage is its first-class support of detailed, morphological, multi-compartment neurons. In fact, strong alternatives to Brian for this project were BioNet (docs, paper) and NetPyNE (docs, paper), which already offer a high-level interface to NEURON with extracellular potential recording. Optogenetics could be incorporated with pre-existing .hoc code, though the light model would need to be implemented. From brief examination of the source code of BioNet, it appears that closed-loop stimulation would not be too difficult to add. It is unclear for NetPyNE.

In the end, we chose Brian since our priority was to model circuit/population-level dynamics over molecular/intra-neuron dynamics. Also, Brian does have support for multi-compartment neurons, albeit less fully featured, if that is needed.

6.1.5 Future development

Here are some features which are missing but could be useful to add:

- Better support for multiple opsins simultaneously. At present the user would have to include a separate variable for each new opsin current, which makes changing the number of different opsins inconvenient
- Support for multiple light sources affecting a single opsin transfection—whether the light sources have the same or different wavelengths
- Electrode microstimulation
- A more accurate LFP signal (only usable for morphological neurons) based on the volume conductor forward model as in LFPy or Vertex
- The Mazzoni-Lindén LFP approximation for LIF point-neuron networks
- Imaging as a recording modality

6.2 Tutorials

6.2.1 Electrode recording

How to insert electrodes to measure different spiking and extracellular signals from a Brian network simulation.

Preamble:

```
from brian2 import *  # includes numpy
import cleo
from cleo import *
# the default cython compilation target isn't worth it for
# this trivial example
prefs.codegen.target = "numpy"
np.random.seed(1919)

cleo.utilities.style_plots_for_docs()

# colors
c = {
    'light': '#df87e1',
    'main': '#C500CC',
    'dark': '#8000B4',
```

(continues on next page)

(continued from previous page)

```
    'exc': '#d6755e',
    'inh': '#056eee',
    'accent': '#36827F',
}
```

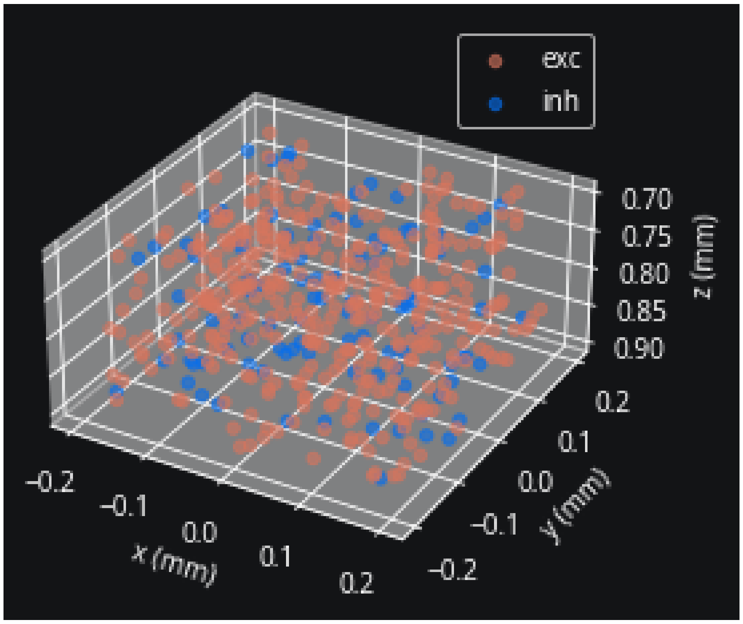
```
INFO       Cache size for target 'cython': 1933664869 MB.
You can call clear_cache('cython') to delete all files from the cache or manually delete␣
↪files in the '/home/kyle/.cython/brian_extensions' directory. [brian2]
```

Network setup

First we create a toy E-I network with Poisson firing rates and assign coordinates:

```
N = 500
n_e = int(N * 0.8)
n_i = int(N * 0.2)

exc = PoissonGroup(n_e, 10 * Hz, name="exc")
inh = PoissonGroup(n_i, 30 * Hz, name="inh")

net = Network([exc, inh])
sim = CLSimulator(net)

cleo.coords.assign_coords_rand_rect_prism(
    exc, xlim=(-0.2, 0.2), ylim=(-0.2, 0.2), zlim=(0.7, 0.9)
)
cleo.coords.assign_coords_rand_rect_prism(
    inh, xlim=(-0.2, 0.2), ylim=(-0.2, 0.2), zlim=(0.7, 0.9)
)
cleo.viz.plot(exc, inh, colors=[c['exc'], c['inh']], scatterargs={'alpha': .6})
```

```
(<Figure size 432x288 with 1 Axes>,
 <Axes3DSubplot:xlabel='x (mm)', ylabel='y (mm)'>)
```



Specifying electrode coordinates

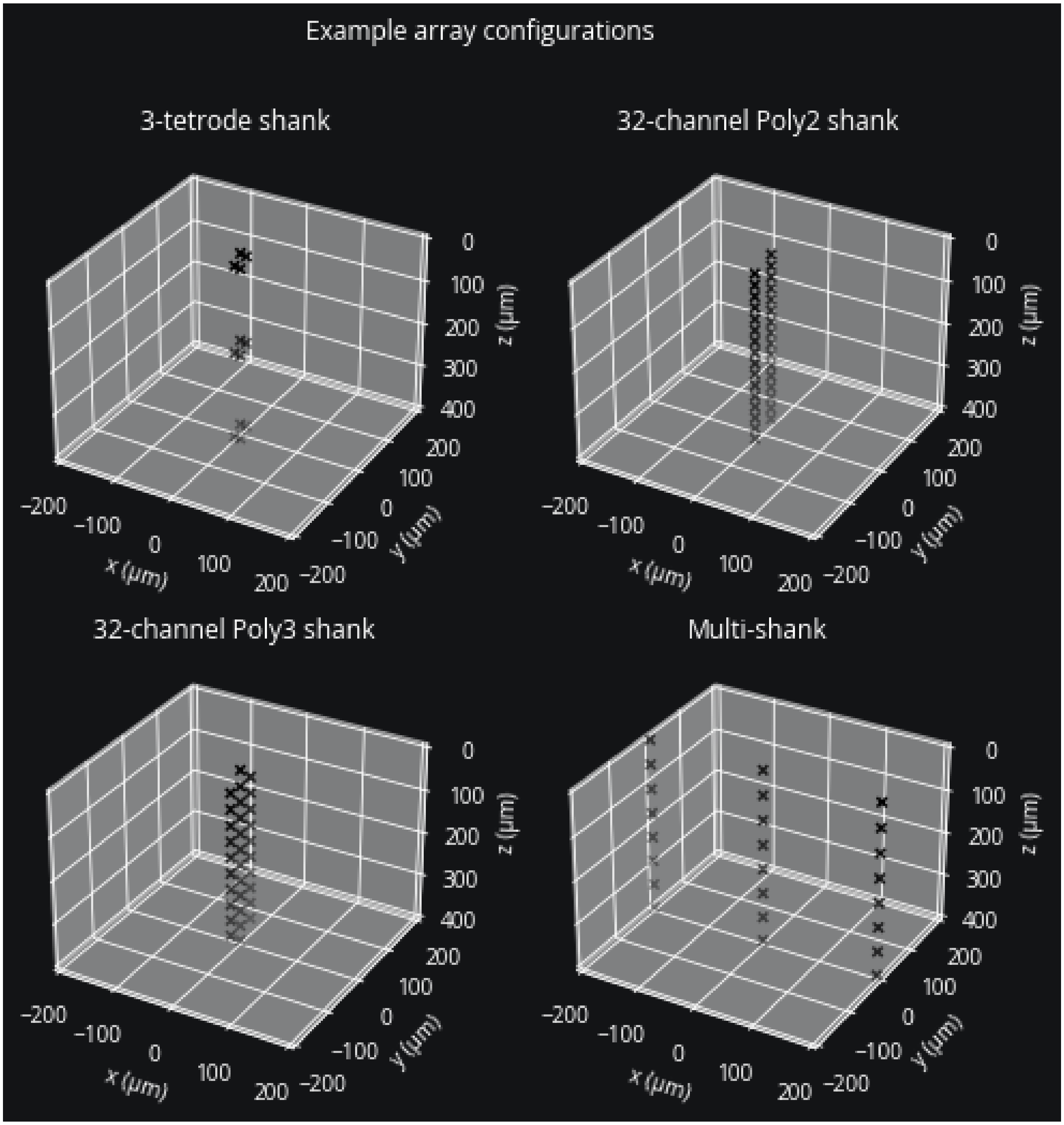
Now we insert an electrode shank probe in the center of the population by injecting an `Probe` device. Note that `Probe` takes arbitrary coordinates as arguments, so you can place contacts wherever you wish. However, the `cleo.ephys` module provides convenience functions to easily generate coordinates common in NeuroNexus probes. Here are some examples:

```
from cleo import ephys
from mpl_toolkits.mplot3d import Axes3D

array_length = 0.4 * mm  # length of the array itself, not the shank
tetr_coords = ephys.tetrode_shank_coords(array_length, tetrode_count=3)
poly2_coords = ephys.poly2_shank_coords(
    array_length, channel_count=32, intercol_space=50 * umeter
)
poly3_coords = ephys.poly3_shank_coords(
    array_length, channel_count=32, intercol_space=30 * umeter
)
# by default start_location (location of first contact) is at (0, 0, 0)
single_shank = ephys.linear_shank_coords(
    array_length, channel_count=8, start_location=(-0.2, 0, 0) * mm
)
# tile vector determines length and direction of tiling (repeating)
multishank = ephys.tile_coords(single_shank, num_tiles=3, tile_vector=(0.4, 0, 0) * mm)

fig = plt.figure(figsize=(8, 8))
fig.suptitle("Example array configurations")
for i, (coords, title) in enumerate(
    [
        (tetr_coords, "3-tetrode shank"),
        (poly2_coords, "32-channel Poly2 shank"),
        (poly3_coords, "32-channel Poly3 shank"),
```

(continues on next page)

(continued from previous page)

```
        (multishank, "Multi-shank"),
    ],
    start=1,
):
    ax = fig.add_subplot(2, 2, i, projection="3d")
    x, y, z = coords.T / umeter
    ax.scatter(x, y, z, marker="x", c="black")
    ax.set(
        title=title,
        xlabel="x (m)",
        ylabel="y (m)",
        zlabel="z (m)",
        xlim=(-200, 200),
        ylim=(-200, 200),
        zlim=(400, 0),
    )
```



As seen above, the `tile_coords` function can be used to repeat a single shank to produce coordinates for a multi-shank probe. Likewise it can be used to repeat multi-shank coordinates to achieve a 3D recording array (what NeuroNexus calls a MatrixArray).

For our example we will use a simple linear array. We configure the probe so it has 32 contacts ranging from 0.2 to 1.2 mm in depth. We could specify the orientation, but by default shank coordinates extend downwards (in the positive z direction).

We can add the electrode to the plotting function to visualize it along with the neurons:

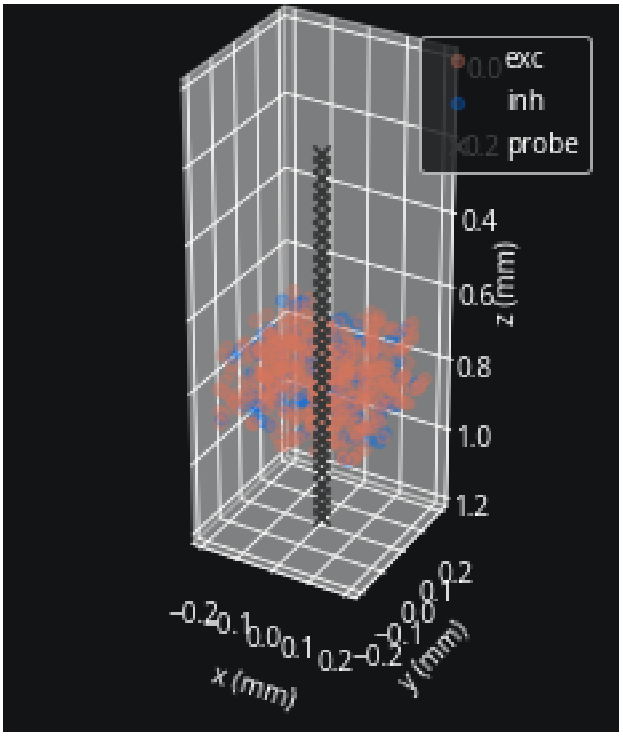
```
coords = ephys.linear_shank_coords(1 * mm, 32, start_location=(0, 0, 0.2) * mm)
probe = ephys.Probe("probe", coords)
```

(continues on next page)

(continued from previous page)

```
cleo.viz.plot(
    exc, inh, colors=[c['exc'], c['inh']], zlim=(0, 1.2), devices=[probe], scatterargs={
↪'alpha': .3}
)
```

```
(<Figure size 432x288 with 1 Axes>,
 <Axes3DSubplot:xlabel='x (mm)', ylabel='y (mm)'>)
```



Specifying signals to record

This looks right, but we need to specify what signals we want to pick up with our electrode. Let's try the two basic spiking signals and an LFP approximation for point neurons.

The two spiking signals (sorted and multi-unit) take the same parameters, mainly `perfect_detection_radius`, within which all spikes will be detected, and `half_detection_radius`, at which distance a spike has only a 50% chance of being detected. My choice to set these parameters at 50 and 100 m is arbitary, though from at least some published data that seems reasonable.

We use default parameters for the Teleńczuk kernel LFP approximation method (TKLFP), but will need to specify cell type (excitatory or inhibitory) and sampling period (if unavailable from a connected IO processor) upon injection.

```
mua = ephys.MultiUnitSpiking(
    "mua",
    perfect_detection_radius=0.05 * mm,
    half_detection_radius=0.1 * mm,
    save_history=True,
)
ss = ephys.SortedSpiking("ss", 0.05 * mm, 0.1 * mm, save_history=True)
```

(continues on next page)

(continued from previous page)

```
tklfp = ephys.TKLFPSignal("tklfp", save_history=True)

probe.add_signals(mua, ss, tklfp)

from cleo.ioproc import RecordOnlyProcessor
sim.set_io_processor(RecordOnlyProcessor(sample_period_ms=1))
sim.inject_recorder(probe, exc, tklfp_type="exc")
sim.inject_recorder(probe, inh, tklfp_type="inh")
```

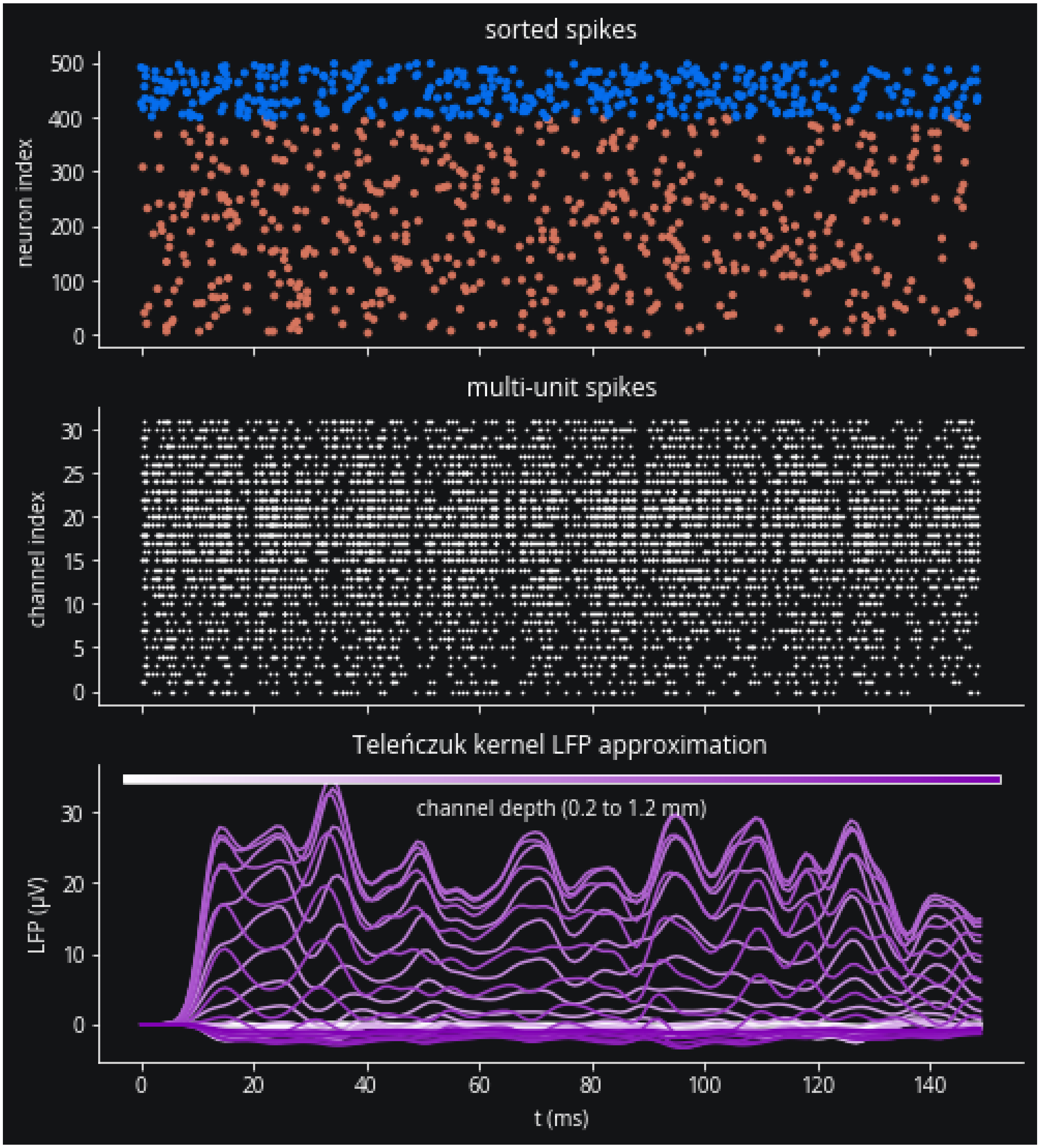
Simulation and results

Now we'll run the simulation:

```
sim.run(150*ms)
```

And plot the output of the three signals we've recorded:

```
from matplotlib.colors import ListedColormap, LinearSegmentedColormap
fig, axs = plt.subplots(3, 1, figsize=(8, 9), sharex=True)

# assuming all neurons are detectable for c=ss.i >= n_e to work
# in practice this will often not be the case and we'd have to map
# from probe index to neuron group index using ss.i_probe_by_i_ng.inverse
exc_inh_cmap = ListedColormap([c['exc'], c['inh']])
axs[0].scatter(ss.t_ms, ss.i, marker=".", c=ss.i >= n_e, cmap=exc_inh_cmap)
axs[0].set(title="sorted spikes", ylabel="neuron index")

axs[1].scatter(mua.t_ms, mua.i, marker=".", s=2, c='white')
axs[1].set(title="multi-unit spikes", ylabel="channel index")

lines = axs[2].plot(tklfp.lfp_uV)
axs[2].set(
    title="Teleńczuk kernel LFP approximation", xlabel="t (ms)", ylabel="LFP (V)"
)

# color-code channel depth
from mpl_toolkits.axes_grid1.inset_locator import inset_axes
depth_cmap = LinearSegmentedColormap.from_list('cleo', ['white', c['dark']])
axins = inset_axes(axs[2], width='95%', height='3%', loc="upper center")
for i in range(32):
    l = lines[i]
    l.set_color(depth_cmap(i / 31))
from matplotlib.colors import Normalize
channel_mappable = plt.cm.ScalarMappable(Normalize(0, 1.2), depth_cmap)
fig.colorbar(
    channel_mappable,
    cax=axins,
    orientation="horizontal",
    ticks=[],
    label="channel depth (0.2 to 1.2 mm)",
);
```



Or, to see the LFP as a function of depth better:

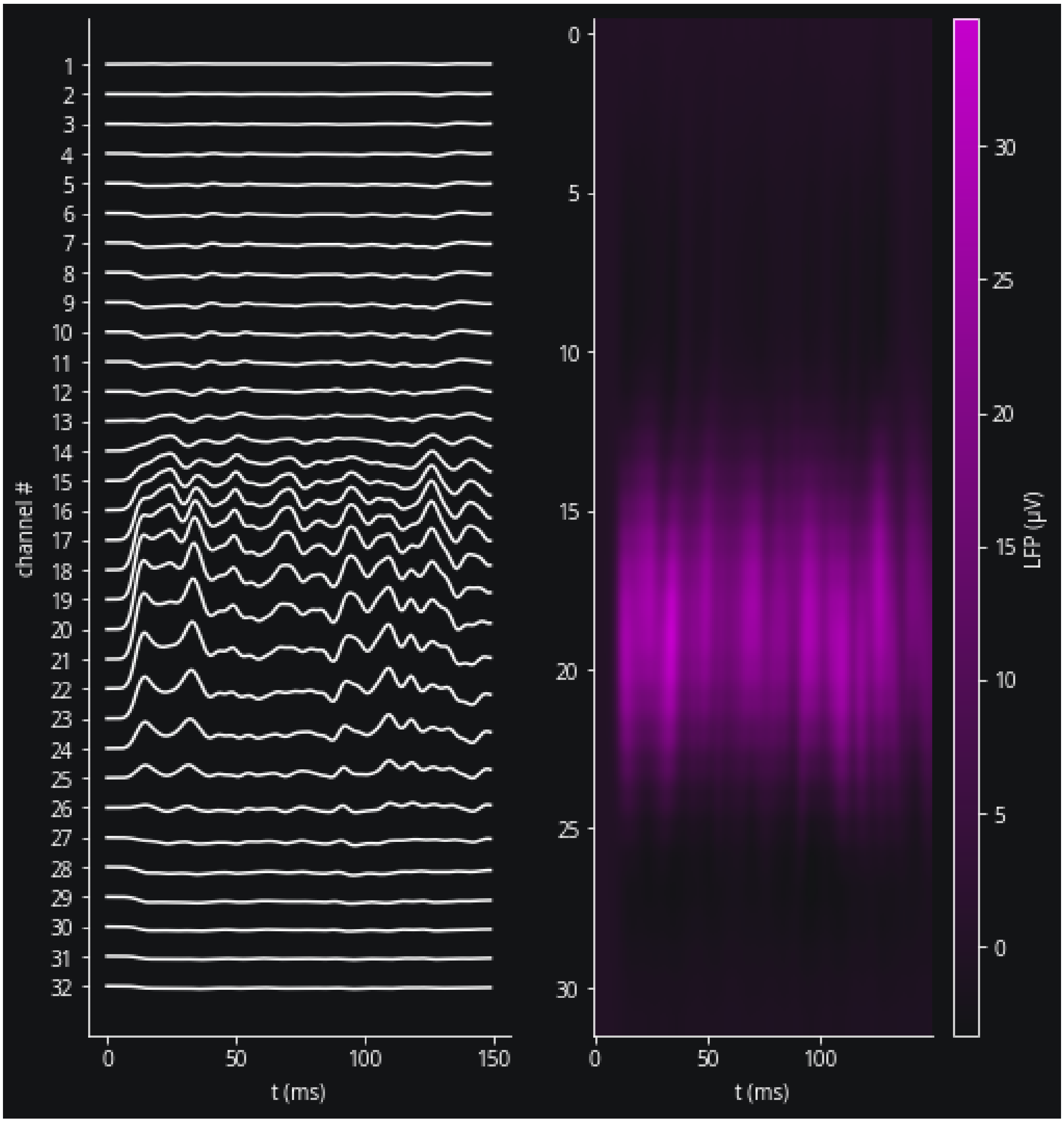
```
fig, axs = plt.subplots(1, 2, figsize=(8, 9))
channel_offsets = -12 * np.arange(32)
lfp_to_plot = tklfp.lfp_uV + channel_offsets
axs[0].plot(lfp_to_plot, color="w")
axs[0].set(
    yticks=channel_offsets,
    yticklabels=range(1, 33),
```

(continues on next page)

(continued from previous page)

```
    xlabel="t (ms)",
    ylabel="channel #",
)
cmap = LinearSegmentedColormap.from_list('lfp', ['#131416', c['main']])
im = axs[1].imshow(tklfp.lfp_uV.T, aspect="auto", cmap=cmap)
axs[1].set(xlabel="t (ms)")
fig.colorbar(im, aspect=40, label="LFP (V)")
```

```
<matplotlib.colorbar.Colorbar at 0x7ff1ee368490>
```



6.2.2 Optogenetic stimulation

How to inject an optogenetic intervention (opsin and optic fiber) into a simulation.

Preamble:

```
from brian2 import *
import matplotlib.pyplot as plt

import cleo
from cleo import *

cleo.utilities.style_plots_for_docs()

# numpy faster than cython for lightweight example
prefs.codegen.target = 'numpy'
# for reproducibility
np.random.seed(1866)
```

```
INFO       Cache size for target 'cython': 1933664869 MB.
You can call clear_cache('cython') to delete all files from the cache or manually delete␣
↪files in the '/home/kyle/.cython/brian_extensions' directory. [brian2]
```

Create a Markov opsin-compatible network

Cleo enables two basic approaches to modeling opsin currents. One is a fairly accurate Markov state model and the other is a simple proportional current model. We will look at the Markov model first.

The established Markov opsin models (as presented in Evans et al., 2016), are conductance-based and so depend on somewhat realistic membrane voltages. Note that we follow the conventions used in neuron modeling, where current is positive, rather than the conventions in opsin modeling, where the photocurrent is negative.

We'll use a small neuron group, biased by Poisson input spikes.

```
n = 10
ng = NeuronGroup(
    n,
    """
    dv/dt = (-(v - E_L) + Delta_T*exp((v-theta)/Delta_T) + Rm*I) / tau_m : volt
    I : amp
    """,
    threshold="v>30*mV",
    reset="v=-55*mV",
    namespace={
        "tau_m": 20 * ms,
        "Rm": 500 * Mohm,
        "theta": -50 * mV,
        "Delta_T": 2 * mV,
        "E_L": -70*mV,
    },
)
ng.v = -70 * mV

input_group = PoissonInput(ng, "v", n, 100 * Hz, 1 * mV)
```

(continues on next page)

(continued from previous page)

```
mon = SpikeMonitor(ng)

net = Network(ng, input_group, mon)
ng.equations
```

$$\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{\Delta_T e^{\frac{-\theta+v}{\Delta_T}} + E_L + IRm - v}{\tau_m} \qquad \text{(unit of } v\text{: V)}$$

$$I \qquad \text{(unit: A)}$$

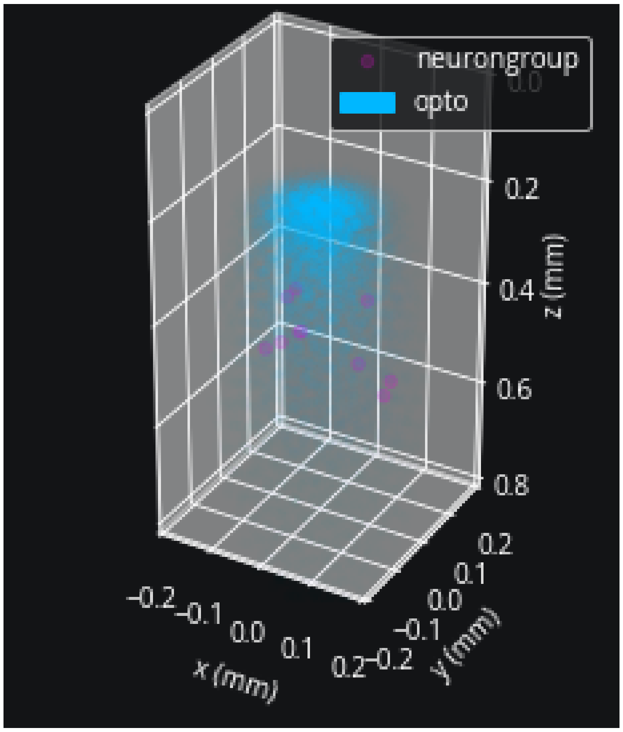
Assign coordinates and configure optogenetic model

The `OptogeneticIntervention` class implements the chosen opsin kinetics model with specified parameters. A standard four-state Markov model as well as channelrhodopsin-2 (ChR2) parameters are included with cleo and are accessible in the `cleo.opto` module. For extending to other models (such as three-state or six-state), see the source code—the state equations, opsin-specific parameters, and light wavelength-specific parameters (if not using 473-nm blue) would be needed.

For reference, cleo draws heavily on Foutz et al., 2012 for the light propagation model and on Evans et al., 2016 for the opsin kinetics model.

```
from cleo.coords import assign_coords_rand_rect_prism
from cleo.opto import *

assign_coords_rand_rect_prism(ng, xlim=(-0.1, 0.1), ylim=(-0.1, 0.1), zlim=(0.4, 0.6))

opto = OptogeneticIntervention(
    name="opto",
    opsin_model=FourStateModel(ChR2_four_state),
    light_model_params=default_blue,
    location=(0, 0, 0.2) * mm,
)

cleo.viz.plot(
    ng,
    colors=["xkcd:fuchsia"],
    xlim=(-0.2, 0.2),
    ylim=(-0.2, 0.2),
    zlim=(0, 0.8),
    devices=[(opto, {'n_points': 3e4, 'intensity': 1})],  # num points to visualize
)
```

```
(<Figure size 432x288 with 1 Axes>,
 <Axes3DSubplot:xlabel='x (mm)', ylabel='y (mm)'>)
```



Open-loop optogenetic stimulation

We need to inject our optogenetic intervention into the simulator. `cleo` handles all the object creation and equations needed to interact with the existing Brian model without the need to alter it, with the possible exception of adding a variable to represent the opsin current. This needs to be specified upon injection with `Iopto_var_name=...` if not the default `Iopto`. The membrane potential variable name also needs to be specified (with `v_var_name=...`) if not the default `v`.

```
sim = CLSimulator(net)
sim.inject_stimulator(opto, ng, Iopto_var_name='I')
```

IO processor setup

Here we design an IO processor that ignores measurements and simply sets the light intensity according to the `stimulus(t)` function:

```
from cleo.ioproc import LatencyIOProcessor

def stimulus(time_ms):
    f = 30
    return 1 * (1 + np.sin(2*np.pi*f * time_ms/1000))

class OpenLoopOpto(LatencyIOProcessor):
    def __init__(self):
        super().__init__(sample_period_ms=1)

    # since this is open-loop, we don't use state_dict
    def process(self, state_dict, time_ms):
```

(continues on next page)

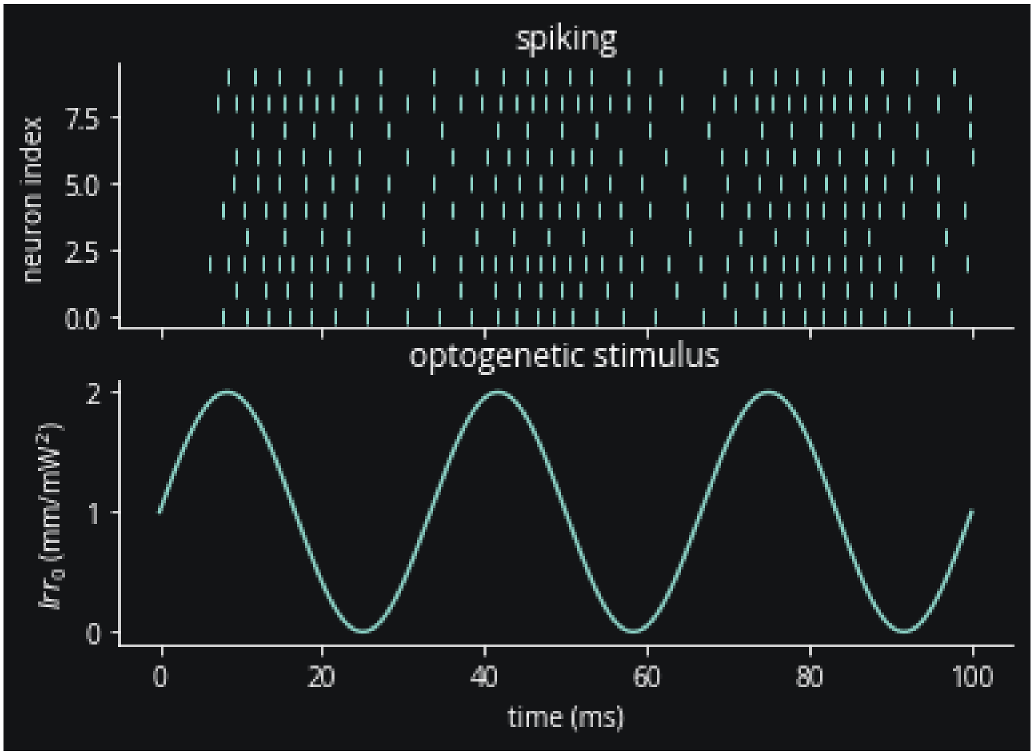
(continued from previous page)

```
        opto_intensity = stimulus(time_ms)
        # return output dict and time
        return ({"opto": opto_intensity}, time_ms)

sim.set_io_processor(OpenLoopOpto())
```

Run simulation and plot results

```
sim.run(100*ms)

fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True)
ax1.plot(mon.t / ms, mon.i[:], '|')
ax1.set(ylabel='neuron index', title='spiking')
t_sim=np.linspace(0, 100, 1000)
ax2.plot(t_sim, stimulus(t_sim))
ax2.set(ylabel=r'$Irr_0$ (mm/mW$^2$)', title='optogenetic stimulus', xlabel='time (ms)');
```

```
INFO       No numerical integration method specified for group 'neurongroup', using␣
↪method 'euler' (took 0.01s, trying other methods took 0.05s). [brian2.stateupdaters.
↪base.method_choice]
```



Conclusion

We can see clearly that firing rate correlates with light intensity as expected.

As a recap, in this tutorial we've seen how to:

- configure an `OptogeneticIntervention`,
- inject it into the simulation,
- and control its light intensity in an open-loop fashion.

Appendix: alternative opsin and neuron models

Because it would be a pain and an obstacle to reproducibility to have to replace all pre-existing simple neuron models with more sophisticated ones with proper voltage ranges and units, we provide an approximation that is much more flexible, requiring only a current term, of any unit, in the target neurons.

The Markov models of opsin dynamics we've used so far produce a rise, peak, and fall to a steady-state plateau current when subjected to sustained light. Since they are conductance-based, the current also varies with membrane voltage, including during spikes. The `ProportionalCurrentModel`, on the other hand, simply delivers current proportional to light intensity. This should be adequate for a wide range of use cases where the exact opsin current dynamics on short timescales don't matter so much and a sort of average current-light relationship will suffice.

Speaking of realistic membrane voltages, does the Markov model's voltage-dependent current render it unsuitable for the most basic leaky integrate-and-fire (LIF) neuron model? LIF neurons reset on reaching their rheobase threshold, staying perpetually in a subthreshold region producing exaggerated opsin currents. How much does this affect the output? We will explore this question by comparing a variety of opsin/neuron model combinations.

First, we introduce exponential integrate-and-fire neurons, which maintain simplicity while modeling an upward membrane potential swing during a spike. For more info, see the related section in the Neuronal Dynamics online textbook and their example parameters table.

```
neuron_params = {
    "tau_m": 20 * ms,
    "Rm": 500 * Mohm,
    "theta": -50 * mV,
    "Delta_T": 2 * mV,
    "E_L": -70*mV,
}

def prep_ng(ng, neuron_type, markov_opsin):
    ng.v = neuron_params['E_L']
    assign_coords_rand_rect_prism(ng, xlim=(0, 0), ylim=(0, 0), zlim=(0, 0))
    state_mon = StateMonitor(ng,("Iopto", "v"), record=True)
    spike_mon = SpikeMonitor(ng)
    return neuron_type, ng, state_mon, spike_mon, markov_opsin

experiments = []

eif = NeuronGroup(
    1,
    """
    dv/dt = (-(v - E_L) + Delta_T*exp((v-theta)/Delta_T) + Rm*Iopto) / tau_m : volt
    Iopto : amp
    """,
```

(continues on next page)

(continued from previous page)

```
    threshold="v > -10*mV",
    reset="v = E_L - 0*mV",
    namespace=neuron_params,
)

experiments.append(prep_ng(eif, 'EIF', True))
```

Configure LIF models

Here we define LIF neurons with biological parameters for the sake of comparison, but the `ProportionalCurrentModel` is compatible with models of any voltage range and units, so long as it has an `Iopto` term.

```
def prep_lif(markov_opsin):
    ng = NeuronGroup(
        1,
        """dv/dt = (-(v - E_L) + Rm*Iopto) / tau_m : volt
        Iopto : amp""",
        threshold="v > theta + 4*mV",
        reset="v = E_L - 0*mV",
        namespace=neuron_params,
    )
    return prep_ng(ng, "LIF", markov_opsin)

experiments.append(prep_lif(True))
experiments.append(prep_lif(False))
```

Comparing to more realistic models

To see how well simplified neuron and opsin models do, we'll also compare to the more complex AdEx neuron (with "tonic" firing pattern parameters) and a Hodgkin-Huxley model (code from Neuronal Dynamics).

```
adex = NeuronGroup(
    1,
    """dv/dt = (-(v - E_L) + 2*mV*exp((v-theta)/Delta_T) + Rm*(Iopto-w)) / tau_m : volt
    dw/dt = (0*nsiemens*(v-E_L) - w) / (100*ms) : amp
    Iopto : amp""",
    threshold="v>=-10*mV",
    reset="v=-55*mV; w+=5*pamp",
    namespace=neuron_params,
)
experiments.append(prep_ng(adex, "AdEx", True))

# Parameters
# Cm = 1*ufarad*cm**-2 * area
Cm = neuron_params["tau_m"] / neuron_params["Rm"]
# area = 5000*umetre**2
area = Cm / (1*ufarad*cm**-2)
gl = 0.3*msiemens*cm**-2 * area
```

(continues on next page)

(continued from previous page)

```
El = -65*mV
EK = -90*mV
ENa = 50*mV
g_na = 40*msiemens*cm**-2 * area
g_kd = 35*msiemens*cm**-2 * area
VT = -63*mV

# The model
eqs = Equations('''
dv/dt = (gl*(El-v) - g_na*(m*m*m)*h*(v-ENa) - g_kd*(n*n*n*n)*(v-EK) + Iopto)/Cm : volt
dm/dt = 0.32*(mV**-1)*4*mV/exprel((13.*mV-v+VT)/(4*mV))/ms*(1-m)-0.28*(mV**-1)*5*mV/
↪exprel((v-VT-40.*mV)/(5*mV))/ms*m : 1
dn/dt = 0.032*(mV**-1)*5*mV/exprel((15.*mV-v+VT)/(5*mV))/ms*(1.-n)-.5*exp((10.*mV-v+VT)/
↪(40.*mV))/ms*n : 1
dh/dt = 0.128*exp((17.*mV-v+VT)/(18.*mV))/ms*(1.-h)-4./(1+exp((40.*mV-v+VT)/(5.*mV)))/
↪ms*h : 1
Iopto : amp
''')
# Threshold and refractoriness are only used for spike counting
hh = NeuronGroup(1, eqs,
                 threshold='v > -40*mV',
                 reset='',
                 method='exponential_euler')

experiments.append(prep_ng(hh, "HH", True))
```

Opsin configuration

Note that the only parameter we need to set for the simple opsin model is the gain on light intensity, `Iopto_per_mW_per_mm2`. This term defines what the neuron receives for every 1 mW/mm2 of light intensity. Here that term is defined in amperes, but it could have been unitless for a simpler model.

The gain is tuned somewhat by hand (in relation to the membrane resistance and the 20 mV gap between rest and threshold potential) to achieve similar outputs to the Markov model.

```
simple_opto = OptogeneticIntervention(
    name="simple_opto",
    # handpicked gain to make firing rate roughly comparable to EIF
    opsin_model=ProportionalCurrentModel(Iopto_per_mW_per_mm2=240/neuron_params['Rm
↪']*20*mV),
    light_model_params=default_blue,
)

markov_opto = OptogeneticIntervention(
    name="markov_opto",
    opsin_model=FourStateModel(ChR2_four_state),
    light_model_params=default_blue,
)
```

Simulation

And we set up the simulator:

```
net = Network()
sim = CLSimulator(net)
for ng_type, ng, state_mon, spike_mon, markov_opsin in experiments:
    net.add(ng, state_mon, spike_mon)
    if markov_opsin:
        sim.inject_stimulator(markov_opto, ng)
    else:
        sim.inject_stimulator(simple_opto, ng)
```

We'll now run the simulation with light pulses of increasing amplitudes to observe the effect on the current.

```
# hand-picked range of amplitudes to show 0 to moderate firing rates
for Irr0_mW_per_mm2 in np.linspace(0.005, 0.03, 5):
    markov_opto.update(Irr0_mW_per_mm2)
    simple_opto.update(Irr0_mW_per_mm2)
    sim.run(60 * ms)
    markov_opto.update(0)
    simple_opto.update(0)
    sim.run(60 * ms)
```

```
INFO       No numerical integration method specified for group 'neurongroup_1', using␣
↪method 'euler' (took 0.01s, trying other methods took 0.04s). [brian2.stateupdaters.
↪base.method_choice]
```

```
INFO       No numerical integration method specified for group 'neurongroup_2', using␣
↪method 'exact' (took 0.04s). [brian2.stateupdaters.base.method_choice]
```

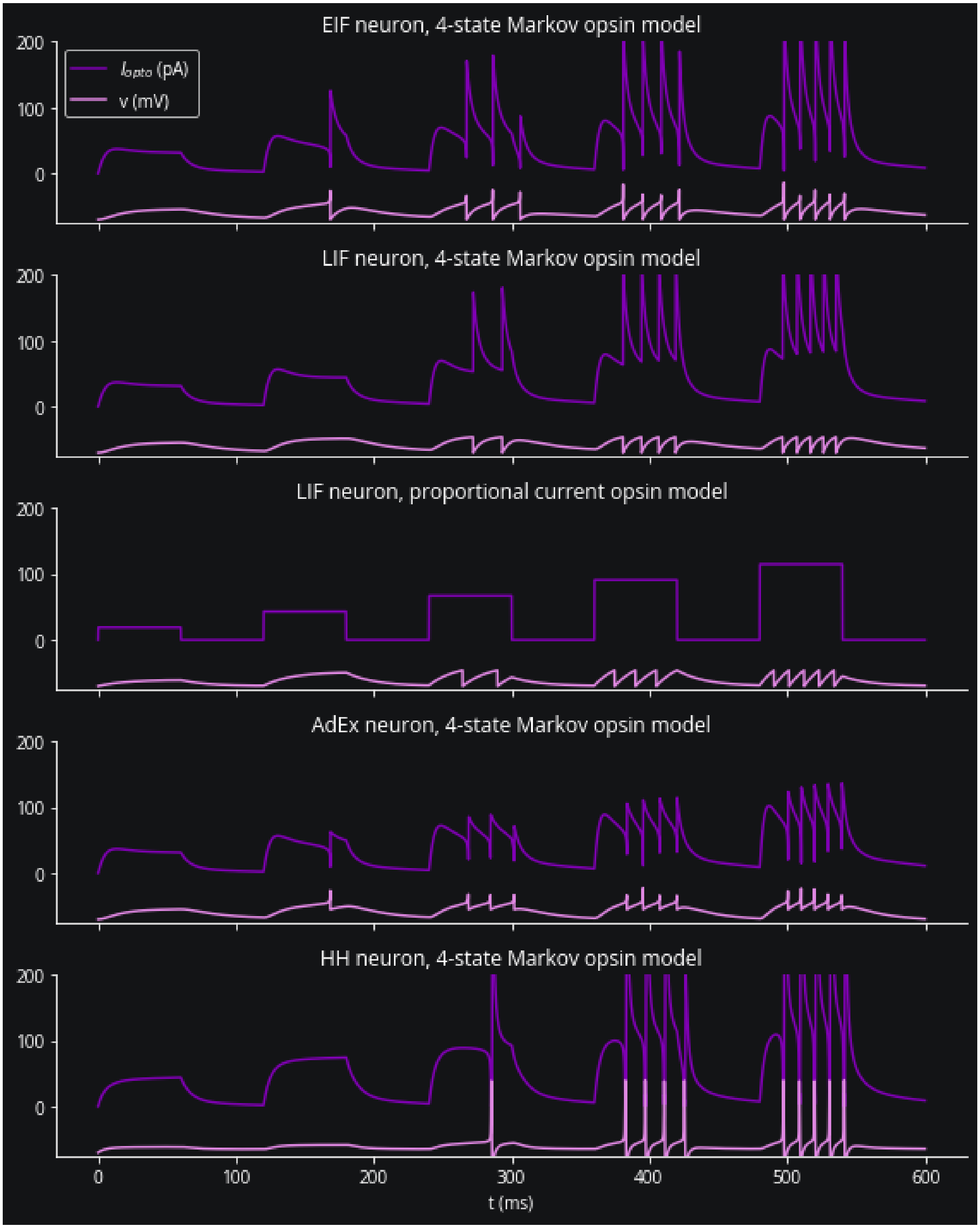
```
INFO       No numerical integration method specified for group 'neurongroup_3', using␣
↪method 'exact' (took 0.02s). [brian2.stateupdaters.base.method_choice]
```

```
INFO       No numerical integration method specified for group 'neurongroup_4', using␣
↪method 'euler' (took 0.01s, trying other methods took 0.05s). [brian2.stateupdaters.
↪base.method_choice]
```

```
WARNING    'n' is an internal variable of group 'neurongroup_5', but also exists in the␣
↪run namespace with the value 10. The internal variable will be used. [brian2.groups.
↪group.Group.resolve.resolution_conflict]
```

Results

```
c1 = '#8000b4'
c2 = '#df87e1'

fig, axs = plt.subplots(
    len(experiments), 1, figsize=(8, 2*len(experiments)), sharex=True
)

for ax, (ng_type, _, state_mon, spike_mon, markov_opsin) in zip(axs, experiments):
    ax.plot(state_mon.t / ms, state_mon.Iopto[0] / pamp, c=c1, label="$I_{opto}$ (pA)")
    ax.plot(state_mon.t / ms, state_mon.v[0] / mV, c=c2, label="v (mV)")
    opsin_name = "4-state Markov" if markov_opsin else "proportional current"
    ax.set(title=f"{ng_type} neuron, {opsin_name} opsin model")

axs[-1].set_xlabel('t (ms)')
axs[0].legend();

max_ylim = max([ax.get_ylim()[1] for ax in axs])
for ax in axs:
    ax.set_ylim([-75, 200])

fig.tight_layout()
```



Qualitatively we can see that the proportional current model doesn't capture the rise, peak, plateau, and fall dynamics that a Markov model can produce, but is a reasonable approximation if all you need is a roughly linear light intensity-firing rate relationship. We also see that a variety of neuron/opsin model combinations all produce similar firing responses to light.

6.2.3 On-off control

Here we will see how to set up a minimum, working closed loop with a very simple threshold-triggered control scheme.

Preamble:

```
from brian2 import *
from cleo import *

import matplotlib.pyplot as plt

utilities.style_plots_for_docs()

# the default cython compilation target isn't worth it for
# this trivial example
prefs.codegen.target = "numpy"
```

```
INFO       Cache size for target 'cython': 1933664869 MB.
You can call clear_cache('cython') to delete all files from the cache or manually delete␣
↪files in the '/home/kyle/.cython/brian_extensions' directory. [brian2]
```

Set up network

We will use a simple leaky integrate-and-fire network with Poisson spike train input. We use Brian's standard SpikeMonitor to view resulting spikes here for simplicity, but see the electrodes tutorial for a more realistic electrode recording scheme.

```
n = 10
population = NeuronGroup(n, '''
            dv/dt = (-v - 70*mV + Rm*I) / tau : volt
            tau: second
            Rm: ohm
            I: amp''',
        threshold='v>-50*mV',
        reset='v=-70*mV'
)
population.tau = 10*ms
population.Rm = 100*Mohm
population.I = 0*mA
population.v = -70*mV

input_group = PoissonGroup(n, np.linspace(0, 100, n)*Hz + 10*Hz)

S = Synapses(input_group, population, on_pre='v+=5*mV')
S.connect(condition='abs(i-j)<=3')

pop_mon = SpikeMonitor(population)

net = Network([population, input_group,S, pop_mon])

print("Recorded population's equations:")
population.user_equations
```

```
Recorded population's equations:
```

$$\frac{dv}{dt} = \frac{IRm - 70mV - v}{\tau} \quad \text{(unit of } v\text{: V)}$$
$$\tau \quad \text{(unit: s)}$$
$$Rm \quad \text{(unit: ohm)}$$
$$I \quad \text{(unit: A)}$$

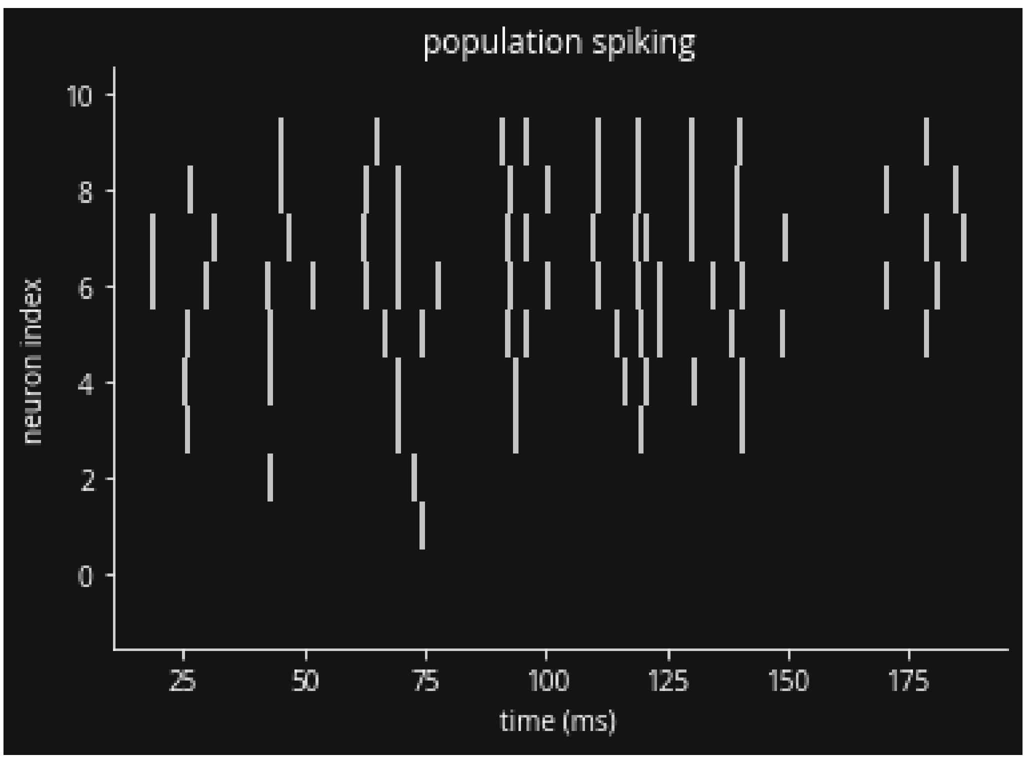
Run simulation

```
net.run(200*ms)
```

```
INFO       No numerical integration method specified for group 'neurongroup', using␣
↪method 'exact' (took 0.08s). [brian2.stateupdaters.base.method_choice]
```

```
sptrains = pop_mon.spike_trains()
fig, ax = plt.subplots()
ax.eventplot([t / ms for t in sptrains.values()], lineoffsets=list(sptrains.keys()))
ax.set(title='population spiking', ylabel='neuron index', xlabel='time (ms)')
```

```
[Text(0.5, 1.0, 'population spiking'),
 Text(0, 0.5, 'neuron index'),
 Text(0.5, 0, 'time (ms)')]
```



Because lower neuron indices receive very little input, we see no spikes for neuron 0. Let's change that with closed-loop control.

IO processor setup

We use the `IOProcessor` class to define interactions with the network. To achieve our goal of making neuron 0 fire, we'll use a contrived, simplistic setup where

1. the recorder reports the voltage of a given neuron (of index 5 in our case),
2. the controller outputs a pulse whenever that voltage is below a certain threshold, and
3. the stimulator applies that pulse to the specified neuron.

So if everything is wired correctly, we'll see bursts of activity in just the first neuron.

```
from cleo.recorders import RateRecorder, VoltageRecorder
from cleo.stimulators import StateVariableSetter

i_rec = int(n / 2)
i_ctrl = 0
sim = CLSimulator(net)
v_rec = VoltageRecorder("rec")
sim.inject_recorder(v_rec, population[i_rec])
sim.inject_stimulator(
    StateVariableSetter("stim", variable_to_ctrl="I", unit=nA), population[i_ctrl]
)
```

We need to implement the `LatencyIOProcessor` object. For a more sophisticated case we'd use `ProcessingBlock` objects to decompose the computation in the `process` function.

```
from cleo.ioproc import LatencyIOProcessor

trigger_threshold = -60*mV
class ReactivePulseIOProcessor(LatencyIOProcessor):
    def __init__(self, pulse_current=1):
        super().__init__(sample_period_ms=1)
        self.pulse_current = pulse_current
        self.out = {}

    def process(self, state_dict, time_ms):
        v = state_dict['rec']
        if v is not None and v < trigger_threshold:
            self.out['stim'] = self.pulse_current
        else:
            self.out['stim'] = 0

        return (self.out, time_ms)

sim.set_io_processor(ReactivePulseIOProcessor(pulse_current=1))
```

And run the simulation:

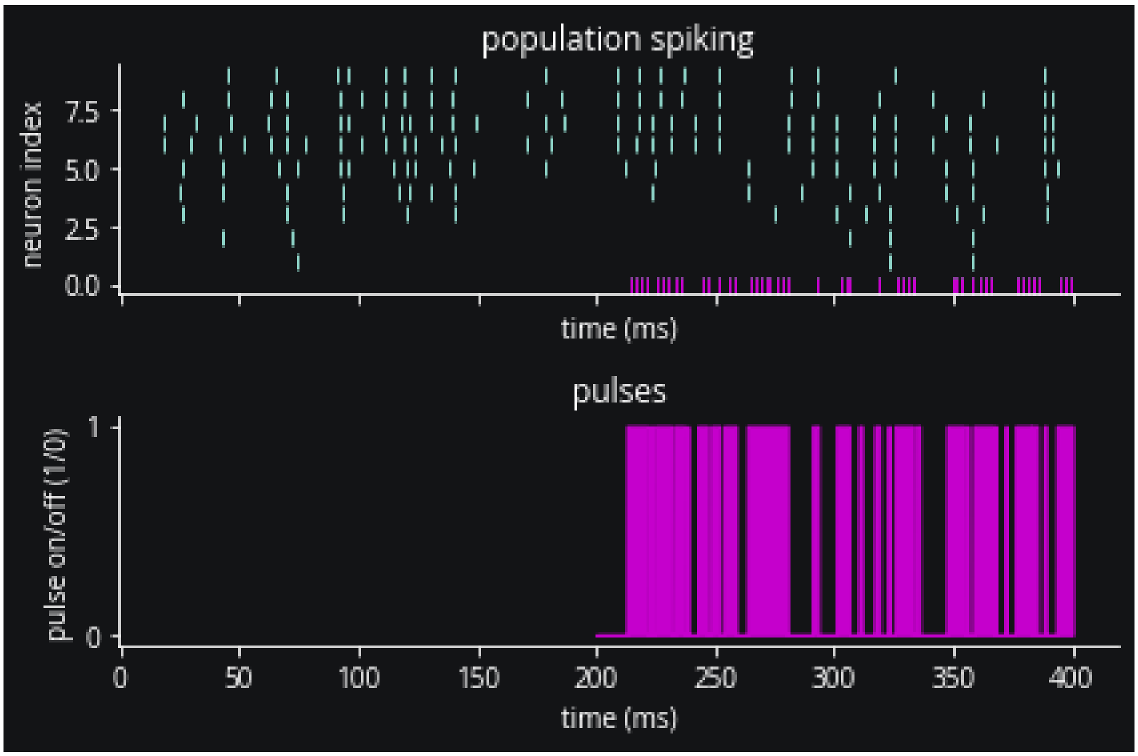
```
sim.run(200*ms)
```

```
fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True)
ax1.plot(pop_mon.t / ms, pop_mon.i[:], "|")
ax1.plot(
```

(continues on next page)

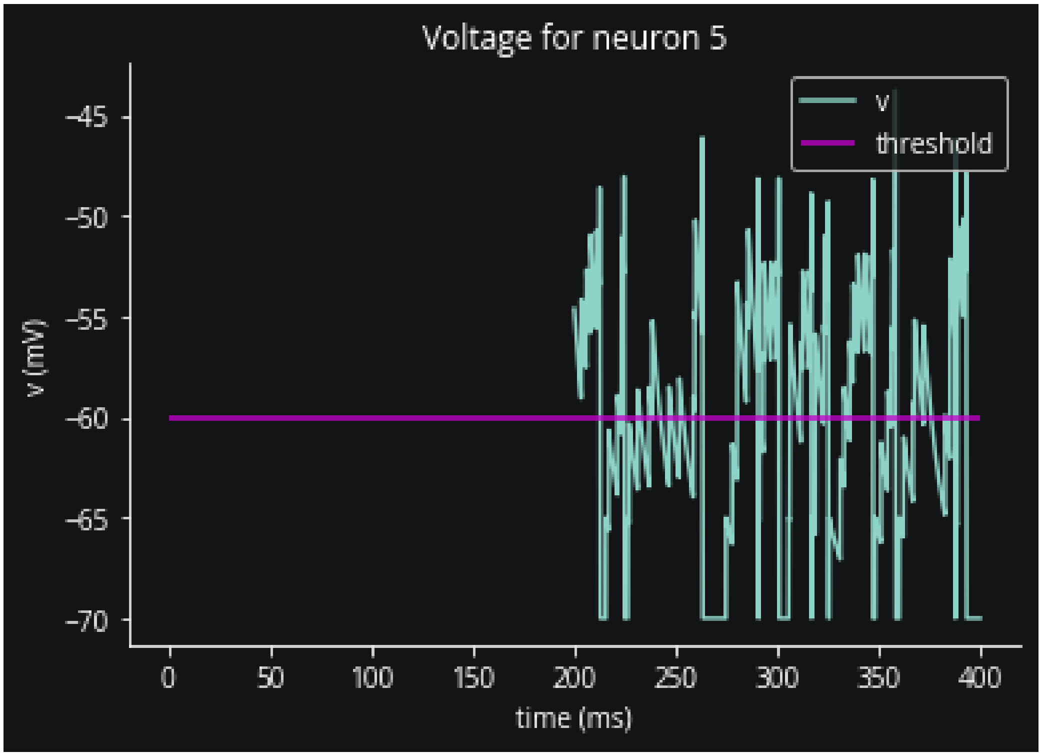
(continued from previous page)

```
    pop_mon.t[pop_mon.i == i_ctrl] / ms,
    pop_mon.i[pop_mon.i == i_ctrl],
    "|",
    c="#C500CC",
)
ax1.set(title="population spiking", ylabel="neuron index", xlabel="time (ms)")
ax2.fill_between(
    v_rec.mon.t / ms, (v_rec.mon.v.T < trigger_threshold)[:, 0], color="#C500CC"
)
ax2.set(title="pulses", xlabel="time (ms)", ylabel="pulse on/off (1/0)", yticks=[0, 1])
plt.tight_layout()
```



Yes, we see the IO processor triggering pulses as expected. And here's a plot of neuron 5's voltage to confirm that those pulses are indeed where we expect them to be, whenever the voltage is below -60 mV.

```
fig, ax = plt.subplots()
ax.set(title=f"Voltage for neuron {i_rec}", ylabel="v (mV)", xlabel='time (ms)')
ax.plot(v_rec.mon.t/ms, v_rec.mon.v.T / mV);
ax.hlines(-60, 0, 400, color='#c500cc');
ax.legend(['v', 'threshold'], loc='upper right');
```



Conclusion

In this tutorial we've seen the basics of configuring an `IOProcessor` to implement a closed-loop intervention on a Brian network simulation.

6.2.4 PI control

In this tutorial we'll introduce

1. PI control, a commonly used model-free control method,
2. the concept of decomposing the `IOProcessor`'s computation into `ProcessingBlock`s, and
3. modeling computation delays on those blocks to reflect hardware and algorithmic speed limitations present in a real experiment.

Preamble:

```
from brian2 import *
import matplotlib.pyplot as plt
from cleo import *

utilities.style_plots_for_docs()

np.random.seed(7000)

# the default cython compilation target isn't worth it for
# this trivial example
prefs.codegen.target = "numpy"
```

```
INFO       Cache size for target 'cython': 1933664869 MB.
You can call clear_cache('cython') to delete all files from the cache or manually delete␣
↪files in the '/home/kyle/.cython/brian_extensions' directory. [brian2]
```

Create the Brian network

We'll create a population of 10 LIF neurons mainly driven by feedforward input but with some recurrent connections as well.

```
n = 10
population = NeuronGroup(n, '''
            dv/dt = (-v - 70*mV + Rm*I) / tau : volt
            tau: second
            Rm: ohm
            I: amp''',
        threshold='v>-50*mV',
        reset='v=-70*mV'
)
population.tau = 10*ms
population.Rm = 100*Mohm
population.I = 0*mA
population.v = -70*mV

input_group = PoissonGroup(n, np.linspace(20, 200, n)*Hz)

S = Synapses(input_group, population, on_pre='v+=5*mV')
S.connect(condition=f'abs(i-j)<={3}')
S2 = Synapses(population, population, on_pre='v+=2*mV')
S2.connect(p=0.2)

pop_mon = SpikeMonitor(population)

net = Network(population, input_group, S, S2, pop_mon)
population.equations
```

$$\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{IRm - 70mV - v}{\tau} \qquad \text{(unit of } v\text{: V)}$$

$$I \qquad \text{(unit: A)}$$

$$Rm \qquad \text{(unit: ohm)}$$

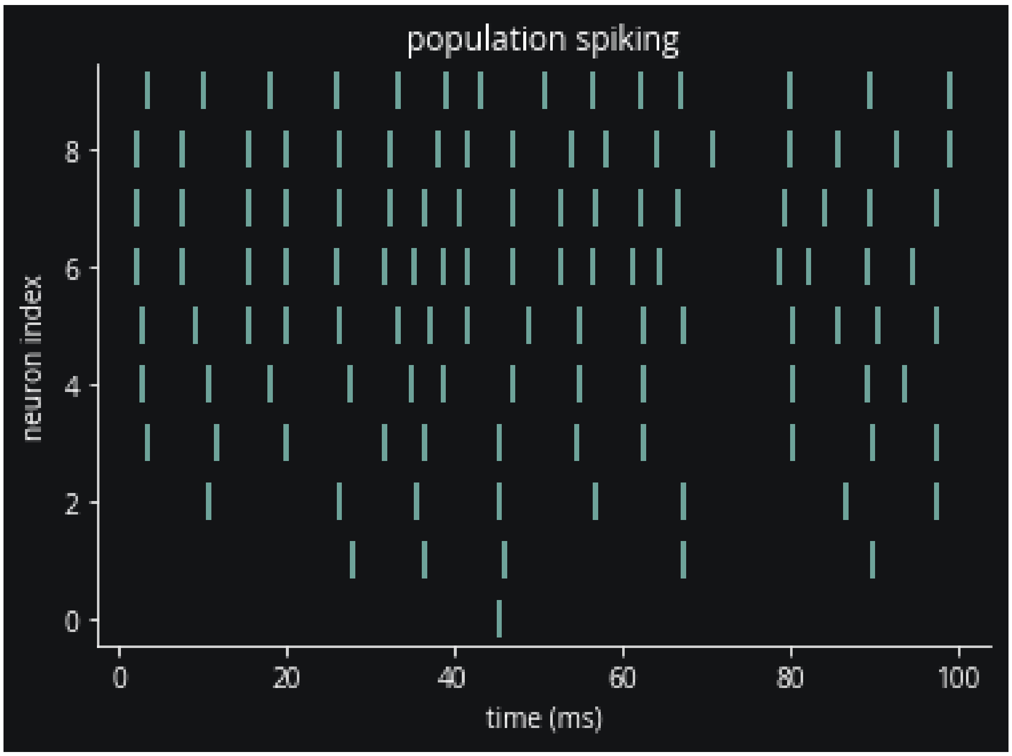
$$\tau \qquad \text{(unit: s)}$$

Run simulation without control:

```
net.run(100*ms)
```

```
INFO       No numerical integration method specified for group 'neurongroup', using␣
↪method 'exact' (took 0.08s). [brian2.stateupdaters.base.method_choice]
```

```
fig, ax = plt.subplots()
ax.scatter(pop_mon.t / ms, pop_mon.i, marker='|', s=200);
ax.set(title='population spiking', ylabel='neuron index', xlabel='time (ms)');
```



Constructing a closed-loop simulation

We will use the popular model-free PI control to control a single neuron's firing rate. PI stands for proportional-integral, referring to a feedback gain *proportional* to the instantaneous error as well as the *integrated* error over time.

First we construct a `CLSimulator` from the network:

```
from cleo import CLSimulator
sim = CLSimulator(net)
```

Then, to control neuron i, we need to:

1. capture spiking using a `GroundTruthSpikeRecorder`

```
from cleo.recorders import GroundTruthSpikeRecorder
i = 0  # neuron to control
rec = GroundTruthSpikeRecorder('spike_rec')
sim.inject_recorder(rec, population[i])
```

1. define the firing rate trajectory we want our target neuron to follow

```
# the target firing rate trajectory, as a function of time
def target_Hz(t_ms):
    if t_ms < 250:  # constant target at first
        return 400
    else:  # sinusoidal afterwards
        a = 200
        t_s = t_ms / 1000
        return a + a * np.sin(2 * np.pi * 20 * t_s)
```

1. estimating its firing rate from incoming spikes using a `FiringRateEstimator`
2. compute the stimulus intensity with a `PIController`
3. output that value for a `StateVariableSetter` stimulator to use

Here we initialize blocks when the `IOProcessor` is created and define how to process network output and set the control signal in the `process` function.

```
from cleo.ioproc import (
    LatencyIOProcessor,
    FiringRateEstimator,
    ConstantDelay,
    PIController,
)

class PIRateIOProcessor(LatencyIOProcessor):
    delta = 1  # ms

    def __init__(self):
        super().__init__(sample_period_ms=self.delta, processing="parallel")
        self.rate_estimator = FiringRateEstimator(
            tau_ms=15,
            sample_period_ms=self.delta,
            delay=ConstantDelay(4.1),  # latency in ms
            save_history=True,  # lets us plot later
        )

        # using hand-tuned gains that seem reasonable
        self.pi_controller = PIController(
            target_Hz,
            Kp=0.005,
            Ki=0.04,
            sample_period_ms=self.delta,
            delay=ConstantDelay(2.87),  # latency in ms
            save_history=True,  # lets us plot later
        )

    def process(self, state_dict, sample_time_ms):
        spikes = state_dict["spike_rec"]
        # feed output and out_time through each block
        out, time_ms = self.rate_estimator.process(
            spikes, sample_time_ms, sample_time_ms=sample_time_ms
        )
```

(continues on next page)

(continued from previous page)

```
        out, time_ms = self.pi_controller.process(
            out, time_ms, sample_time_ms=sample_time_ms
        )
        # this dictionary output format allows for the flexibility
        # of controlling multiple stimulators
        if out < 0:  # limit to positive current
            out = 0
        out_dict = {"I_stim": out}
        # time_ms at the end reflects the delays added by each block
        return out_dict, time_ms

io_processor = PIRateIOProcessor()
sim.set_io_processor(io_processor)
```

Note that we can set delays for individual `ProcessingBlocks` in the IO processor to better approximate the experiment. We use simple constant delays here, but a `GaussianDelay` class is also available and others could be easily implemented.

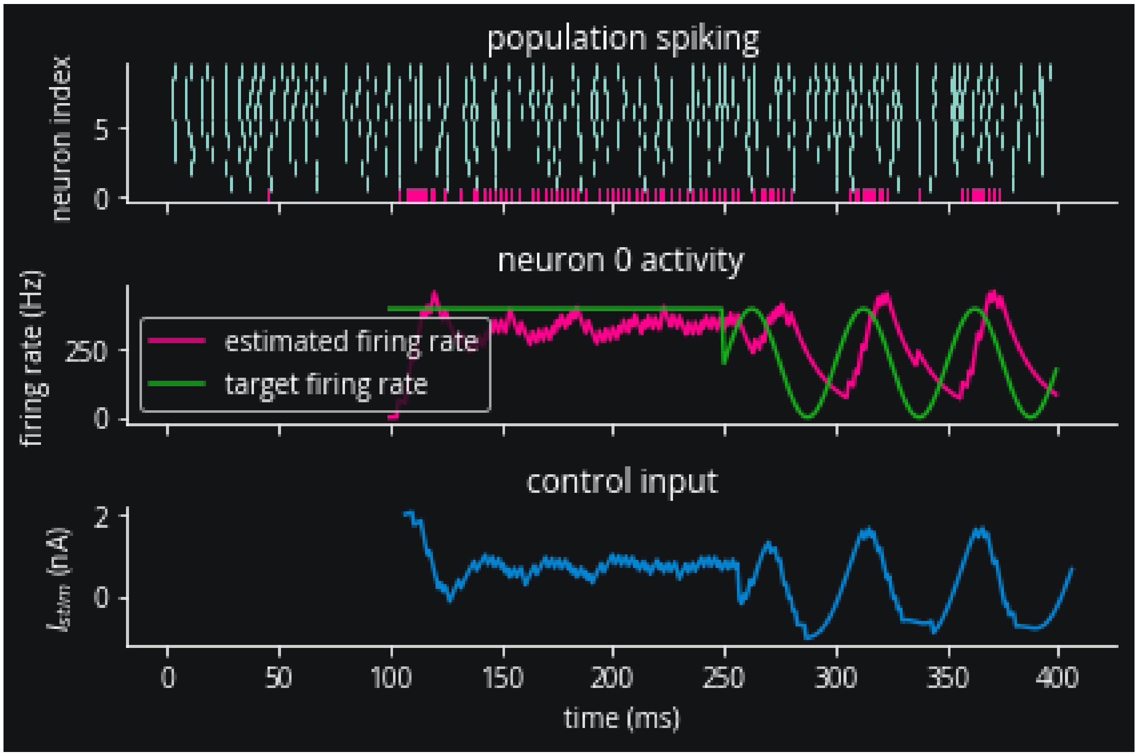
Now we inject the stimulator:

```
from cleo.stimulators import StateVariableSetter
sim.inject_stimulator(
        StateVariableSetter(
            'I_stim', variable_to_ctrl='I', unit=nA),
        population[i]
)
```

Run the simulation

```
sim.run(300*ms)
```

```
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, sharex=True);
ax1.plot(pop_mon.t / ms, pop_mon.i[:], '|');
ax1.plot(pop_mon.t[pop_mon.i == i]/ms, pop_mon.i[pop_mon.i==i], '|', c='xkcd:hot pink')
ax1.set(title='population spiking', ylabel='neuron index')

ax2.plot(io_processor.rate_estimator.t_in_ms, io_processor.rate_estimator.values, c=
↪'xkcd:hot pink');
ax2.plot(io_processor.rate_estimator.t_in_ms, [target_Hz(t) for t in io_processor.rate_
↪estimator.t_in_ms],\
         c='xkcd:green');
ax2.set(ylabel='firing rate (Hz)', title=f'neuron {i} activity');
ax2.legend(['estimated firing rate', 'target firing rate']);

ax3.plot(io_processor.pi_controller.t_out_ms, io_processor.pi_controller.values, c=
↪'xkcd:cerulean')
ax3.set(title='control input', ylabel='$I_{stim}$ (nA)', xlabel='time (ms)')

fig.tight_layout()
fig.show()
```



Note the lag in keeping up with the target firing rate, which can be directly attributed to the ~7 ms delay we coded in to the IO processor.

Conclusion

In this tutorial, we've learned how to

- use PI control to interact with a Brian simulation,
- decompose processing steps into blocks, and
- assign delays to processing blocks to model real-life latency.

6.2.5 LQR optimal control using `ldsctrlest`

This tutorial will be more comprehensive than the others, bringing together all of `cleo`'s main capabilities—electrode recording, optogenetics, and latency modeling—as well as introducing more sophisticated model-based feedback control. To achieve the latter, we will use the `ldsctrlest` Python bindings to the `ldsCtrlEst` C++ library.

Preamble:

```
from brian2 import *
import matplotlib.pyplot as plt
import cleo

cleo.utilities.style_plots_for_docs()

# numpy faster than cython for lightweight example
prefs.codegen.target = 'numpy'
np.random.seed(1856)
```

```
INFO       Cache size for target 'cython': 1933664869 MB.
You can call clear_cache('cython') to delete all files from the cache or manually delete␣
↪files in the '/home/kyle/.cython/brian_extensions' directory. [brian2]
```

Network setup

As in the optogenetics tutorial, we'll use a trivial network of a small neuron group biased by Poisson input spikes. We'll use the exponential integrate-and-fire neuron model, which maintains simplicity while modeling an upward membrane potential swing when spiking.

```
n = 2
ng = NeuronGroup(
    n,
    """
    dv/dt = (-(v - E_L) + Delta_T*exp((v-theta)/Delta_T) + Rm*I) / tau_m : volt
    I : amp
    """,
    threshold="v>30*mV",
    reset="v=-55*mV",
    namespace={
        "tau_m": 20 * ms,
        "Rm": 500 * Mohm,
        "theta": -50 * mV,
        "Delta_T": 2 * mV,
        "E_L": -70 * mV,
    },
)
ng.v = -70 * mV

input_group = PoissonInput(ng, "v", 10, 100 * Hz, 2.5 * mV)

net = Network(ng, input_group)
```

Coordinates, stimulation, and recording

Here we assign coordinates to the neurons and configure the optogenetic intervention and recording setup:

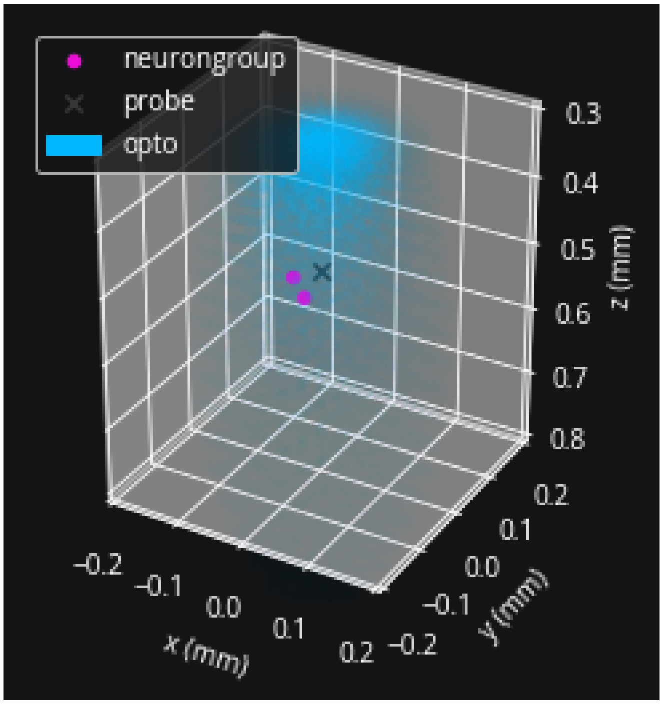
```
from cleo.coords import assign_coords_rand_rect_prism
from cleo.opto import *
from cleo.ephys import Probe, SortedSpiking

hor_lim = .05
assign_coords_rand_rect_prism(ng, xlim=(-hor_lim, hor_lim), ylim=(-hor_lim, hor_lim),␣
↪zlim=(0.4, 0.6))

opto = OptogeneticIntervention(
    name="opto",
    opsin_model=FourStateModel(ChR2_four_state),
    light_model_params=default_blue,
    location=(0, 0, 0.3) * mm,
)
```

(continues on next page)

(continued from previous page)

```
spikes = SortedSpiking(
    "spikes",
    perfect_detection_radius=40 * umeter,
    half_detection_radius=80 * umeter,
    save_history=True,
)
probe = Probe(
    "probe",
    coords=[0, 0, 0.5] * mm,
    signals=[spikes],
)

cleo.viz.plot(
    ng,
    colors=["xkcd:fuchsia"],
    xlim=(-0.2, 0.2),
    ylim=(-0.2, 0.2),
    zlim=(0.3, 0.8),
    devices=[probe, (opto, {'n_points': 1e5})],
    scatterargs={'alpha': 1}
)
```

```
(<Figure size 432x288 with 1 Axes>,
 <Axes3DSubplot:xlabel='x (mm)', ylabel='y (mm)'>)
```



Looks right. Let's set up the simulation and inject the devices:

```
sim = cleo.CLSimulator(net)
```

(continues on next page)

(continued from previous page)

```
sim.inject_stimulator(opto, ng, Iopto_var_name='I')
sim.inject_recorder(probe, ng)
```

Prepare controller

Our goal will be to control two neuron's firing rates simultaneously. To do this, we will use the LQR technique explained in Bolus et al., 2021 ("State-space optimal feedback control of optogenetically driven neural activity)".

Fit model

Our controller needs a model of the system's dynamics, which we can obtain by fitting to training data. We will generate training data using Gaussian random walk inputs. `ldsCtrlEst` is designed for data coming from an experiment, organized into trials, so we will run the simulation repeatedly, resetting after each run. Here u represents the input and z the spike output.

We will intentionally use very little training data so the importance of adaptive control will become apparent later on.

```
n_trials = 5
n_samp = 100
u = []
z = []
n_u = 1  # 1-dimensional input (just one optogenetic actuator)
n_z = 2  # we'll be controlling two neurons
for trial in range(n_trials):
    # one-sided normally distributed training data, stdev of 10 mW/mm2
    u_trial = 10*np.abs(np.random.randn(n_u, n_samp))
    u.append(u_trial)
    z.append(np.zeros((n_z, n_samp)))
```

The IO processor is simple enough here that we won't bother separating steps using `ProcessingBlock` objects, which is recommended for more complex scenarios where modularity is more important.

```
from cleo.ioproc import LatencyIOProcessor

class TrainingStimIOP(LatencyIOProcessor):
    i_samp = 0
    i_trial = 0

    # here we just feed in the training inputs and record the outputs
    def process(self, state_dict, sample_time_ms):
        i, t, z_t = state_dict['probe']['spikes']
        z[self.i_trial][:, self.i_samp] = z_t[:n_z]  # just first two neurons
        out = {'opto': u[self.i_trial][:, self.i_samp]}
        self.i_samp += 1
        return out, sample_time_ms

training_stim_iop = TrainingStimIOP(sample_period_ms=1)
sim.set_io_processor(training_stim_iop)

for i_trial in range(n_trials):
    training_stim_iop.i_trial = i_trial
```

(continues on next page)

(continued from previous page)

```
    training_stim_iop.i_samp = 0
    sim.run(n_samp*ms)
    sim.reset()
```

```
INFO       No numerical integration method specified for group 'neurongroup', using␣
↪method 'euler' (took 0.01s, trying other methods took 0.06s). [brian2.stateupdaters.
↪base.method_choice]
```

Now we have `u` and `z` in the form we need for `ldsctrlest`'s fitting functions: `n_trial`-length lists of `n` by `n_samp` arrays. We will now fit Gaussian linear dynamical systems using the SSID algorithm. See the documentation for more detailed explanations.

```
import ldsctrlest as lds
import ldsctrlest.gaussian as glds
n_x_fit = 2  # latent dimensionality of system
n_h = 50  # size of block Hankel data matrix
dt = 0.001  # timestep (in seconds)
u_train = lds.UniformMatrixList(u, free_dim=2)
z_train = lds.UniformMatrixList(z, free_dim=2)
ssid = glds.FitSSID(n_x_fit, n_h, dt, u_train, z_train)
fit, sing_vals = ssid.Run(lds.SSIDWt.kMOESP)
```

Design controller

LQR optimal control

We now use the fit parameters to create the controller system and set additional parameters. The feedback gain, K_c, is especially important, determining how the controller responds to the current "error"—the difference between where the system is (estimated to be) now and where we want it to be. The field of optimal control deals with how to design the controller so as to minimize a cost function reflecting what we care about.

With a linear system (obtained from the fitting procedure above) and quadratic per-timestep cost function L penalizing distance from the reference x^* and the input u

$$L = \frac{1}{2}(x - x^*)^T Q(x - x^*) + \frac{1}{2}u^T Ru$$

we can use the closed-form optimal solution called the Linear Quadratic Regulator (LQR).

$$K = (R + B^T PB)^{-1}(B^T PA) \qquad u = -Kx$$

The P matrix is obtained by numerically solving the discrete algebraic Riccati equation:

$$P = A^T PA - \left(A^T PB\right)\left(R + B^T PB\right)^{-1}\left(B^T PA\right) + Q$$

```
fit_sys = glds.System(fit)
# upper and lower bounds on control signal (optic fiber light intensity)
u_lb = 0  # mW/mm2
u_ub = 30  # mW/mm2
controller = glds.Controller(fit_sys, u_lb, u_ub)
```

(continues on next page)

(continued from previous page)

```
# careful not to use this anymore since controller made a copy
del fit_sys

from scipy.linalg import solve_discrete_are
# cost matrices
# Q reflects how much we care about state error
# we use C'C since we really care about output error, not latent state
Q_cost = controller.sys.C.T @ controller.sys.C
R_cost = 1e-4 * np.eye(n_u)  # reflects how much we care about minimizing the stimulus
A, B = controller.sys.A, controller.sys.B
P = solve_discrete_are(A, B, Q_cost, R_cost)
controller.Kc = np.linalg.inv(R_cost + B.T @ P @ B) @ (B.T @ P @ A)
controller.Print()
```

We now configure the IOProcessor to use our controller:

```
class CtrlLoop(LatencyIOProcessor):
    def __init__(self, samp_period_ms, controller, y_ref: callable):
        super().__init__(samp_period_ms)
        self.controller = controller
        self.sys = controller.sys
        self.y_ref = y_ref
        self.do_control = False  # allows us to turn on and off control

        # for post hoc visualization/analysis:
        self.u = np.empty((n_u, 0))
        self.x_hat = np.empty((n_x_fit, 0))
        self.y_hat = np.empty((n_z, 0))
        self.z = np.empty((n_z, 0))

    def process(self, state_dict, sample_time_ms):
        i, t, z_t = state_dict["probe"]["spikes"]
        z_t = z_t[:n_z].reshape((-1, 1))  # just first n_z neurons
        self.controller.y_ref = self.y_ref(sample_time_ms)

        u_t = self.controller.ControlOutputReference(z_t, do_control=self.do_control)
        out = {opto.name: u_t}

        # record variables from this timestep
        self.u = np.hstack([self.u, u_t])
        self.y_hat = np.hstack([self.y_hat, self.sys.y])
        self.x_hat = np.hstack([self.x_hat, self.sys.x])
        self.z = np.hstack((self.z, z_t))

        return out, sample_time_ms + 3  # 3 ms delay


y_ref = 200 * dt  # target rate in Hz
ctrl_loop = CtrlLoop(
    samp_period_ms=1, controller=controller, y_ref=lambda t: np.ones((n_z, 1)) * y_ref
)
```

Run the experiment

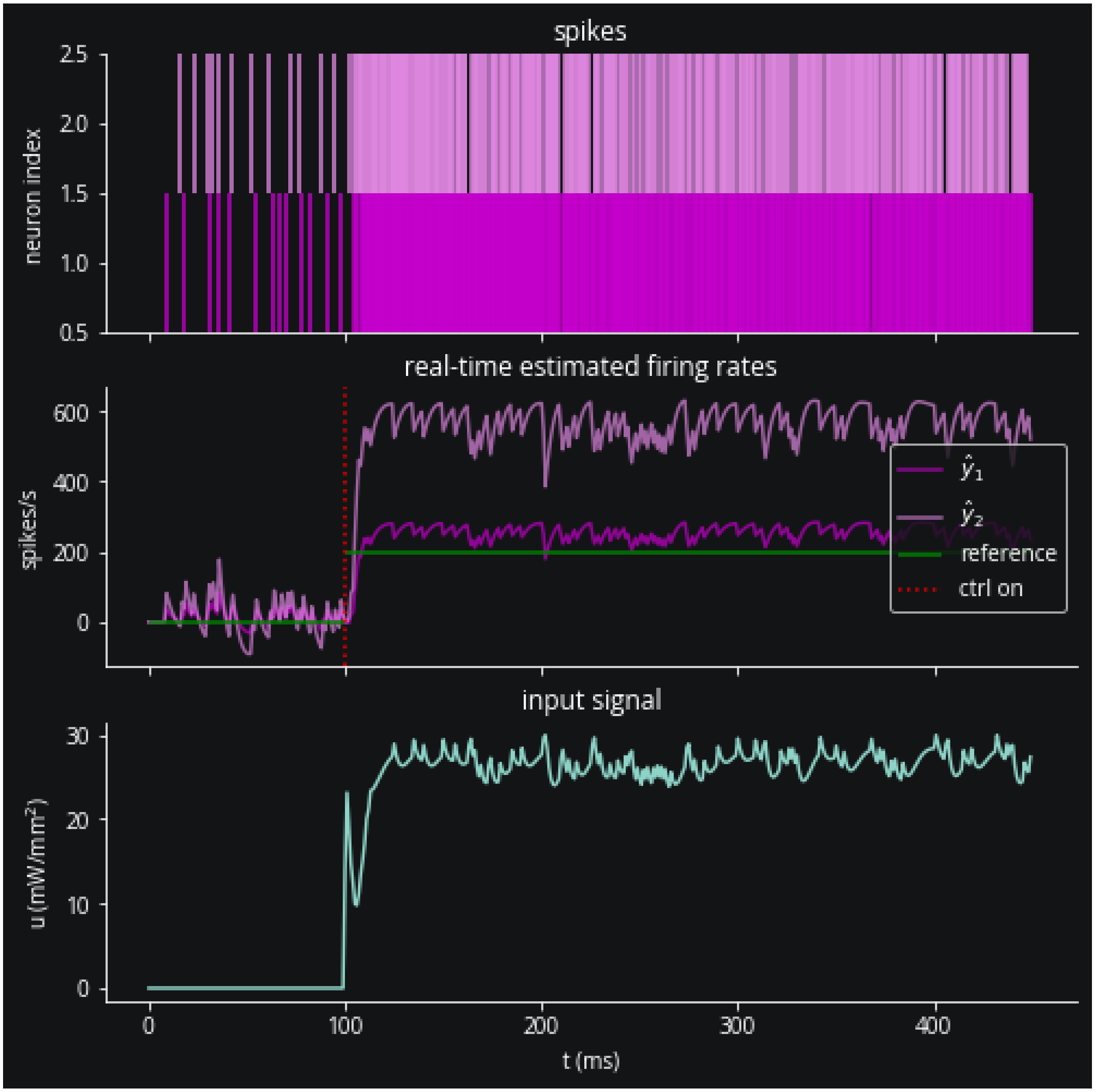
We'll now run the simulation with and without control to compare.

```
sim.set_io_processor(ctrl_loop)
T0 = 100
sim.run(T0*ms)

ctrl_loop.do_control = True
T1 = 350
sim.run(T1*ms)
```

```
WARNING    'dt' is an internal variable of group 'synapses_opto_neurongroup', but also␣
↪exists in the run namespace with the value 0.001. The internal variable will be used.␣
↪[brian2.groups.group.Group.resolve.resolution_conflict]
```

Now we plot the results to see how well the controller was able to match the desired firing rate:

```
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, sharex=True, figsize=(8,8))
c1 = "#C500CC"
c2 = "#df87e1"
spikes1 = spikes.t_ms[spikes.i == 0]
spikes2 = spikes.t_ms[spikes.i == 1]
ax1.eventplot([spikes1, spikes2], lineoffsets=[1, 2], colors=[c1, c2])
ax1.set(ylabel='neuron index', ylim=(0.5, 2.5), title='spikes')
ax2.set(ylabel='spikes/s', title='real-time estimated firing rates')
ax2.plot(ctrl_loop.y_hat[0]/dt, c=c1, alpha=0.7, label='$\hat{y}_1$')
ax2.plot(ctrl_loop.y_hat[1]/dt, c=c2, alpha=0.7, label='$\hat{y}_2$')
ax2.hlines(y_ref/dt, 100, T0+T1, color='green', label='reference')
ax2.hlines(0, 0, 100, color='green')
ax2.axvline(T0, c='xkcd:red', linestyle=':', label='ctrl on')
ax2.legend(loc="right")
ax3.plot(range(T0+T1), ctrl_loop.u.T)
ax3.set(xlabel='t (ms)', ylabel='u (mW/mm$^2$)', title='input signal');
```



Looks all right, but in addition to the system's estimated firing rate let's count the spikes over the control period to see how well we hit the target on average:

```
print("Results (spikes/second):")
print('baseline =', np.sum(ctrl_loop.z[:, :T0], axis=1)/(T0/1000))
print('target =', [y_ref*1000, y_ref*1000])
print('lqr achieved =', (np.sum(ctrl_loop.z[:, T0:T0+T1], axis=1)/(T1/1000)).round(1))
```

```
Results (spikes/second):
baseline = [130. 120.]
target = [200.0, 200.0]
lqr achieved = [965.7 800. ]
```

We can see that the system consistently underestimates the true firing rate. And as we could expect, we weren't able to maintain the target firing rate with both neurons simultaneously since one was exposed to more light than the other. However, the controller was able to achieve something. See the appendix for how we can avoid overshooting with both neurons, which should be avoidable.

Conclusion

As a recap, in this tutorial we've seen how to:

- inject optogenetic stimulation into an existing Brian network
- inject an electrode into an existing Brian network to record spikes
- generate training data and fit a Gaussian linear dynamical system to the spiking output using `ldsctrlest`
- configure an `ldsctrlest` LQR controller based on that linear system and design optimal gains
- use that controller in running a complete simulated feedback control experiment

Appendix

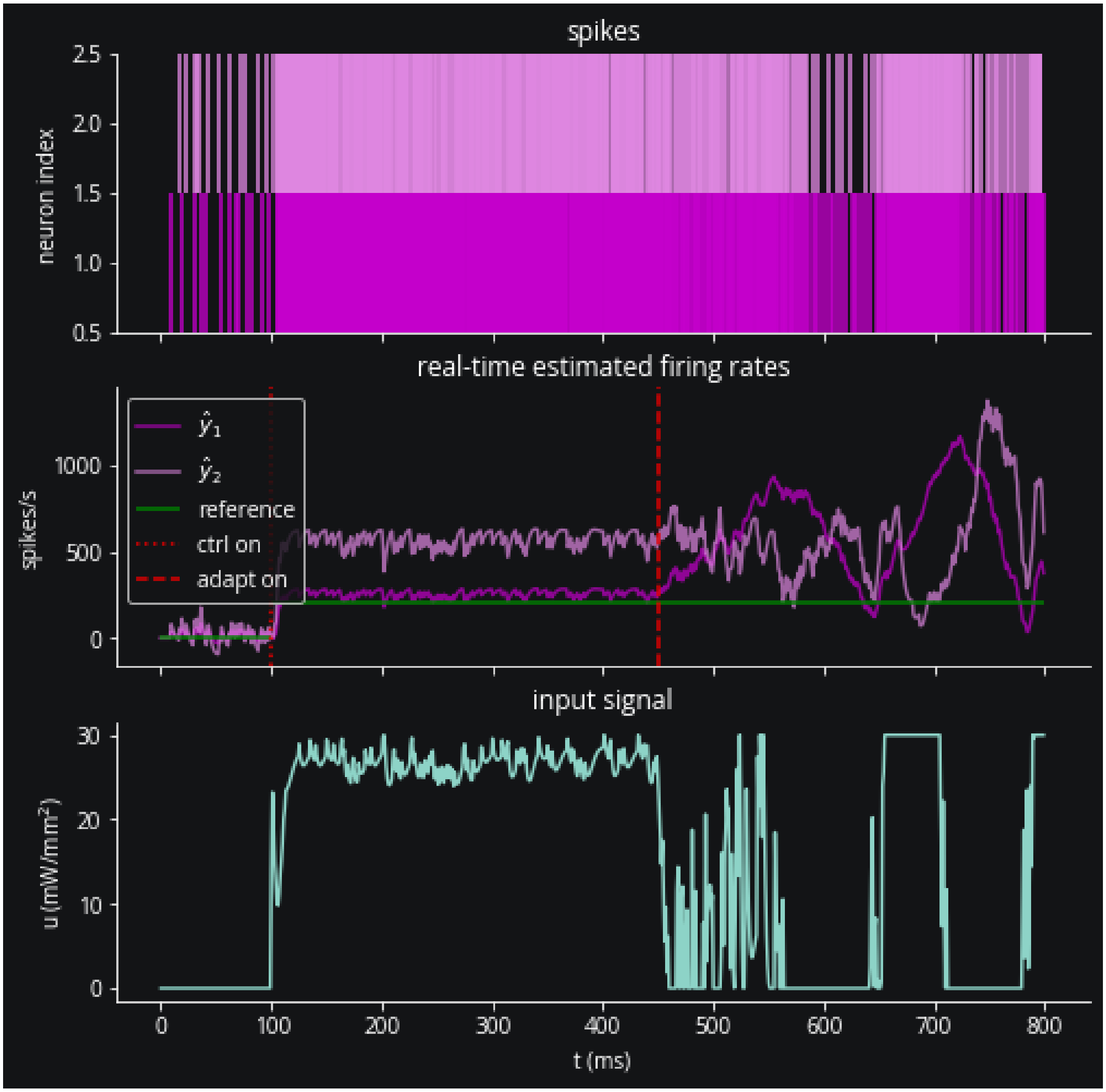
Adaptive control

ldsCtrlEst also provides an *adaptive* variation on LQR, capable of inferring state beyond our static, linear model and thus able to account for unmodeled disturbances and noise. Let's see how it compares:

```
controller.sys.do_adapt_m = True  # enable adaptive disturbance estimation
# set covariance for the disturbance state
# larger values mean the system more readily ascribes changes to unmodeled disturbance
controller.sys.Q_m = 1e-2 * np.eye(n_x_fit)
controller.control_type = lds.kControlTypeAdaptM  # enable adaptive control
```

```
ctrl_loop.sys.do_adapt_m = True
T2 = 350
sim.run(T2*ms)
T = T0 + T1 + T2
```

```
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, sharex=True, figsize=(8,8))
spikes1 = spikes.t_ms[spikes.i == 0]
spikes2 = spikes.t_ms[spikes.i == 1]
ax1.eventplot([spikes1, spikes2], lineoffsets=[1, 2], colors=[c1, c2])
ax1.set(ylabel='neuron index', ylim=(.5, 2.5), title='spikes')
ax2.set(ylabel='spikes/s', title='real-time estimated firing rates')
ax2.plot(ctrl_loop.y_hat[0]/dt, c=c1, alpha=0.7, label='$\hat{y}_1$')
ax2.plot(ctrl_loop.y_hat[1]/dt, c=c2, alpha=0.7, label='$\hat{y}_2$')
ax2.hlines(y_ref/dt, 100, T, color='green', label='reference')
ax2.hlines(0, 0, 100, color='green')
ax2.axvline(T0, c='xkcd:red', linestyle=':', label='ctrl on')
ax2.axvline(T0+T1, c='xkcd:red', linestyle='--', label='adapt on')
ax2.legend(loc="upper left")
ax3.plot(range(T), ctrl_loop.u.T)
ax3.set(xlabel='t (ms)', ylabel='u (mW/mm$^2$)', title='input signal');
```



We can see the effect most easily in the input signal, which has much more variation now. Let's confirm that the firing rates were better balanced around the target:

```
print("Results (spikes/second):")
print('baseline =', np.sum(ctrl_loop.z[:, :T0], axis=1)/(T0/1000))
print('target =', [y_ref*1000, y_ref*1000])
print('static achieved =', (np.sum(ctrl_loop.z[:, T0:T0+T1], axis=1)/(T1/1000)).round(1))
print('adaptive achieved =', (np.sum(ctrl_loop.z[:, T0+T1:T], axis=1)/(T2/1000)).
↪round(1))
```

```
Results (spikes/second):
baseline = [130. 120.]
target = [200.0, 200.0]
static achieved = [965.7 800. ]
```

(continues on next page)

(continued from previous page)

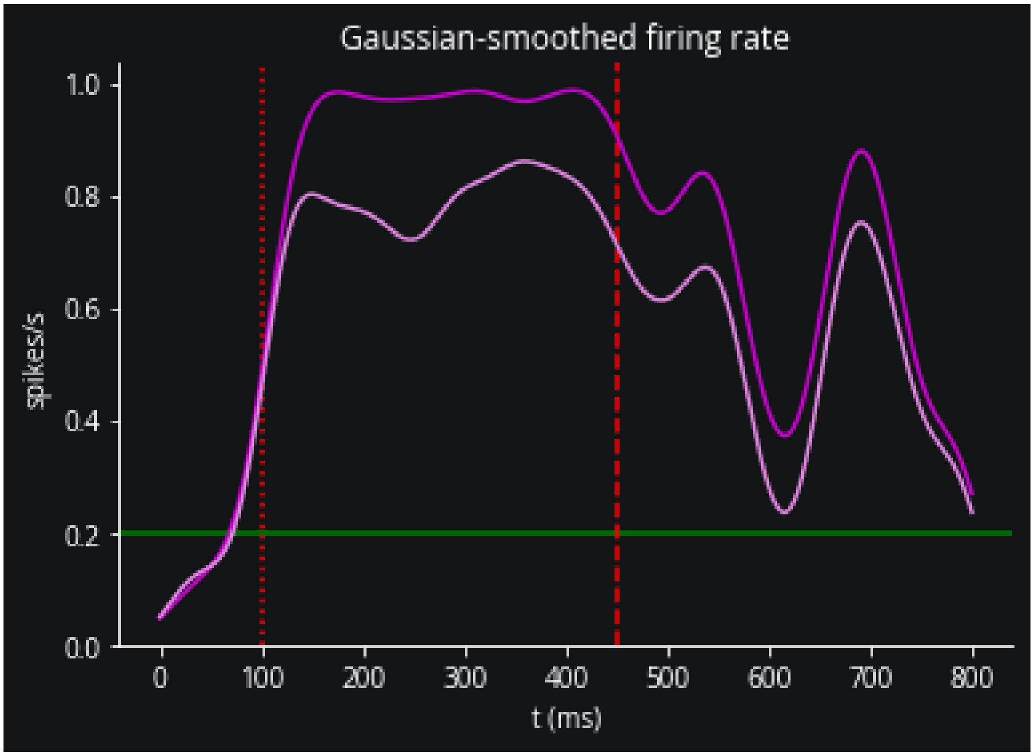
```
adaptive achieved = [657.1 534.3]
```

That looks better. Adaptive control achieves a balance between the two neurons, as we would expect.

Post-hoc firing rate estimate

To see if the system's online estimation of firing rates is reasonable, we compute a Gaussian-smoothed version with a 25-ms standard deviation:

```
from scipy.stats import norm
kernel = norm.pdf(np.linspace(-75, 75, 151), scale=25)  # 25-ms  Gaussian window
smoothed1 = np.convolve(ctrl_loop.z[0, :], kernel, mode='same')
smoothed2 = np.convolve(ctrl_loop.z[1, :], kernel, mode='same')
plt.axhline(y_ref, c='g')
plt.axvline(T0, c='r', ls=':')
plt.axvline(T0+T1, c='r', ls='--')
plt.xlabel("t (ms)")
plt.ylabel("spikes/s")
plt.title("Gaussian-smoothed firing rate")
plt.plot(smoothed1, c=c1)
plt.plot(smoothed2, c=c2);
```



6.2.6 Video visualization

In this tutorial we'll see how to inject a video visualizer into a simulation.

Preamble:

```
from brian2 import *
import matplotlib.pyplot as plt

from cleo import *

utilities.style_plots_for_docs()

# numpy faster than cython for lightweight example
prefs.codegen.target = 'numpy'
# for reproducibility
np.random.seed(1866)

c_exc = 'xkcd:tomato'
c_inh = 'xkcd:cerulean blue'
```

```
INFO       Cache size for target 'cython': 1933664869 MB.
You can call clear_cache('cython') to delete all files from the cache or manually delete␣
↪files in the '/home/kyle/.cython/brian_extensions' directory. [brian2]
```

Set up the simulation

Network

We'll use excitatory and inhibitory populations of exponential integrate-and-fire neurons.

```
n_e = 400
n_i = n_e // 4
def eif(n, name):
    ng = NeuronGroup(
        n,
        """
        dv/dt = (-(v - E_L) + Delta_T*exp((v-theta)/Delta_T) + Rm*I) / tau_m : volt
        I : amp
        """,
        threshold="v>30*mV",
        reset="v=-55*mV",
        namespace={
            "tau_m": 20 * ms,
            "Rm": 500 * Mohm,
            "theta": -50 * mV,
            "Delta_T": 2 * mV,
            "E_L": -70*mV,
        },
        name=name,
    )
    ng.v = -70 * mV
```

(continues on next page)

(continued from previous page)

```
    return ng

exc = eif(n_e, "exc")
inh = eif(n_i, "inh")
W = 250
p_S = 0.3
S_ei = Synapses(exc, inh, on_pre="v_post+=W*mV/n_e")
S_ei.connect(p=p_S)
S_ie = Synapses(inh, exc, on_pre="v_post-=W*mV/n_i")
S_ie.connect(p=p_S)
S_ee = Synapses(exc, exc, on_pre="v_post+=W*mV/n_e")
S_ee.connect(condition='abs(i-j)<=20')

mon_e = SpikeMonitor(exc)
mon_i = SpikeMonitor(inh)

net = Network(exc, inh, S_ei, S_ie, S_ee, mon_e, mon_i)
```

Coordinates and optogenetics

Here we configure the coordinates and optogenetic stimulation. For more details, see the "Optogenetic stimulation" tutorial. Note that we save the arguments used in the plotting function for reuse later on when generating the video.

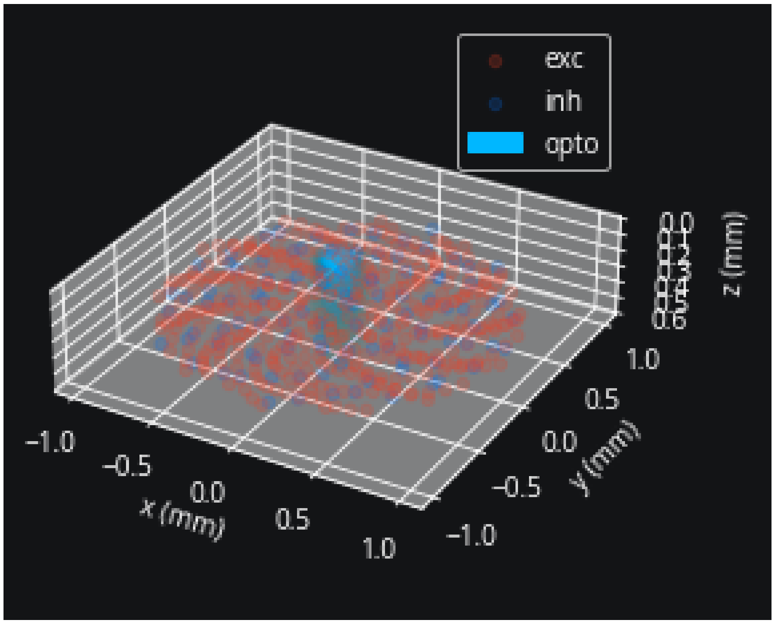
```
from cleo.coords import assign_coords_uniform_cylinder
from cleo.viz import plot

r = 1
assign_coords_uniform_cylinder(
    exc, xyz_start=(0, 0, 0.3), xyz_end=(0, 0, 0.4), radius=r
)
assign_coords_uniform_cylinder(
    inh, xyz_start=(0, 0, 0.3), xyz_end=(0, 0, 0.4), radius=r
)

from cleo.opto import (
    OptogeneticIntervention,
    FourStateModel,
    ChR2_four_state,
    default_blue,
)

opto = OptogeneticIntervention(
    name="opto",
    opsin_model=FourStateModel(ChR2_four_state),
    light_model_params=default_blue,
    max_Irr0_mW_per_mm2=30,
)

plotargs = {
    "colors": [c_exc, c_inh],
    "zlim": (0, 0.6),
```

(continues on next page)

(continued from previous page)

```
    "scatterargs": {"s": 20},  # to adjust neuron marker size
}

plot(
    exc,
    inh,
    **plotargs,
    devices=[(opto, {"n_points": 2e4})],
)
```

```
(<Figure size 432x288 with 1 Axes>,
 <Axes3DSubplot:xlabel='x (mm)', ylabel='y (mm)'>)
```



Simulator, optogenetics injection

Here we create the simulator and inject the `OptogeneticIntervention`.

```
sim = CLSimulator(net)
sim.inject_stimulator(opto, exc, Iopto_var_name='I')
```

Processor

And we set up open-loop optogenetic stimulation:

```
from cleo.ioproc import LatencyIOProcessor

opto.update(0)
stim_vals = []
stim_t = []
```

(continues on next page)

(continued from previous page)

```
class OpenLoopOpto(LatencyIOProcessor):
    def process(self, state_dict, time_ms):
        # random walk stimulation
        opto_intensity = opto.value + np.random.randn()*.5
        if opto_intensity < 0:
            opto_intensity = 0
        # save values for plotting
        stim_vals.append(opto_intensity)
        stim_t.append(time_ms)
        return ({"opto": opto_intensity}, time_ms)

sim.set_io_processor(OpenLoopOpto(sample_period_ms=1))
```

Inject VideoVisualizer

A `VideoVisualizer` is an `InterfaceDevice` like recorders and stimulators and needs to be injected in order to properly interact with the Brian network. Keep in mind the following:

- It must be injected *after* all other devices for the `devices='all'` argument to work as expected.
- Similarly to recording and stimulation, you must specify the target neuron groups (to display, in this case) on injection
- The `dt` argument makes a huge difference on the amount of time it takes to generate the video. You may want to keep this high while experimenting and only lower it when you are ready to generate a high-quality video since the process is so slow.

```
from cleo.viz import VideoVisualizer

vv = VideoVisualizer(dt=1 * ms, devices="all")
sim.inject_device(vv, exc, inh)
```

Run simulation and visualize

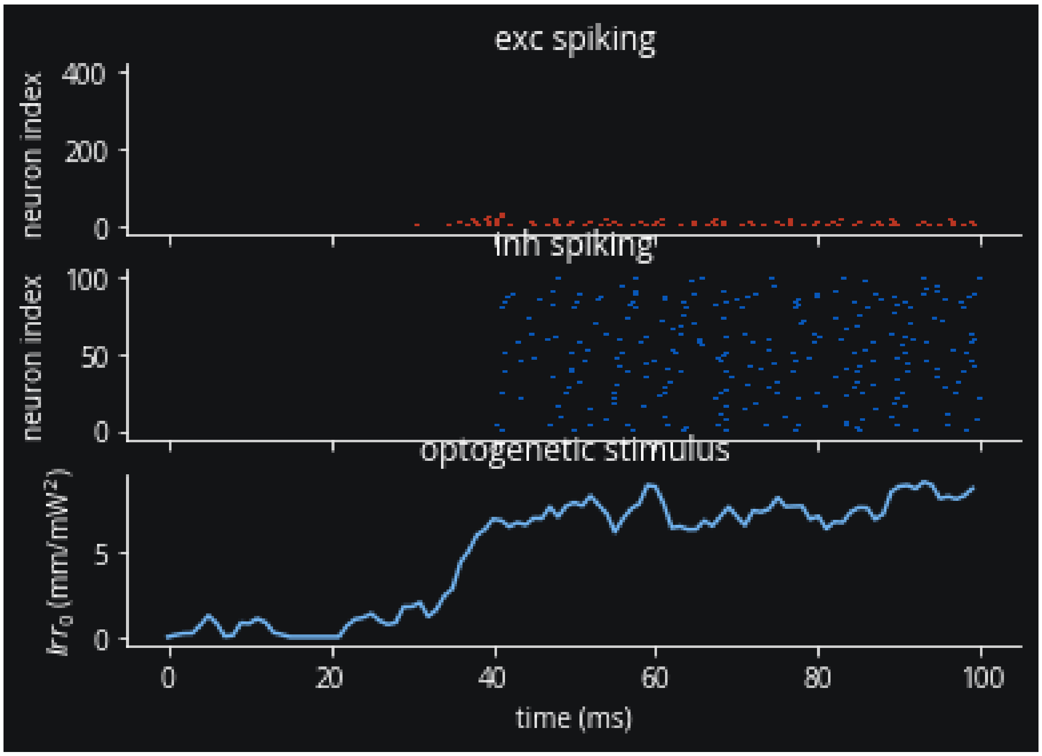
Here we display a quick plot before generating the video:

```
T = 100
sim.run(T * ms)

fig, (ax1, ax2, ax3) = plt.subplots(3, 1, sharex=True)
sptexc = mon_e.spike_trains()
ax1.eventplot([t/ms for t in sptexc.values()], lineoffsets=list(sptexc.keys()), color=c_
↪exc)
ax1.set(ylabel="neuron index", title="exc spiking")
sptinh = mon_i.spike_trains()
ax2.eventplot([t/ms for t in sptinh.values()], lineoffsets=list(sptinh.keys()), color=c_
↪inh)
ax2.set(ylabel="neuron index", title="inh spiking")
ax3.plot(stim_t, stim_vals, c="#72b5f2")
ax3.set(ylabel=r"$Irr_0$ (mm/mW$^2$)", title="optogenetic stimulus", xlabel="time (ms)");
```

```
INFO       No numerical integration method specified for group 'exc', using method 'euler
↪' (took 0.01s, trying other methods took 0.04s). [brian2.stateupdaters.base.method_
↪choice]
```

```
INFO       No numerical integration method specified for group 'inh', using method 'euler
↪' (took 0.00s, trying other methods took 0.02s). [brian2.stateupdaters.base.method_
↪choice]
```

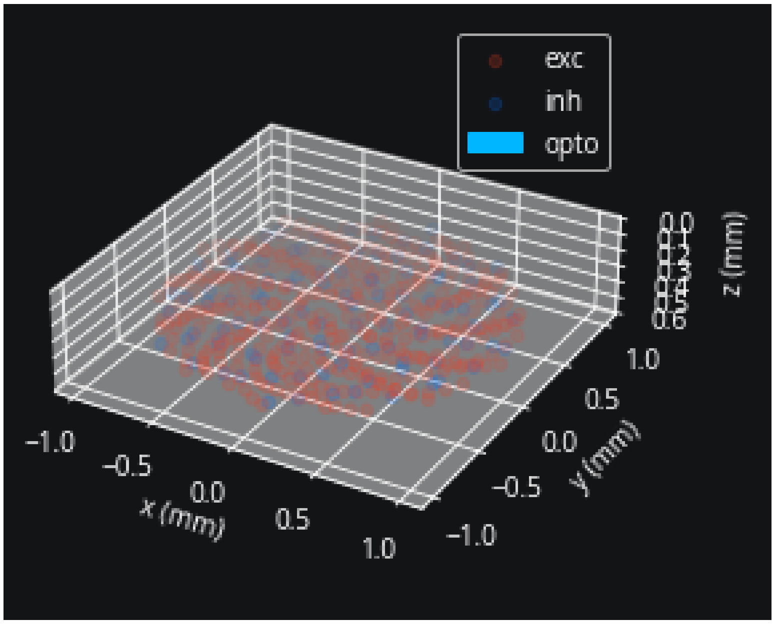
```
WARNING    'T' is an internal variable of group 'synapses_opto_exc', but also exists in␣
↪the run namespace with the value 100. The internal variable will be used. [brian2.
↪groups.group.Group.resolve.resolution_conflict]
```



The `VideoVisualizer` stores the data it needs during the simulation, but hasn't yet produced any visual output. We first use the `generate_Animation()`, plugging in the arguments we used for the original plot.

Also, we set the `max_Irr0_mW_per_mm2_viz` attribute of the optogenetic intervention. This effectively scales how bright the light appears in the visualization. That is, a high maximum irradiance makes the stimulus values small in comparison and produces a faint light, while a low ceiling makes the values relatively large and produces a bright light in the resulting video.

```
opto.max_Irr0_mW_per_mm2_viz = max(stim_vals)
ani = vv.generate_Animation(plotargs, slowdown_factor=10)
```



The `generate_Animation()` function returns a matplotlib `FuncAnimation` object, which you can then use however you want. You will probably want to save a video.

Note that at this point the video still hasn't been rendered; that happens when you try and save or visualize the animation. This step takes a while if your temporal resolution is high, so we suggest you do this only after your experiment is finalized and after you've experimented with low framerate videos to finalize video parameters.

Here we embed the video using HTML so you can see the output:

```
from matplotlib import rc
rc('animation', html='jshtml')

ani
```

```
<matplotlib.animation.FuncAnimation at 0x7fad6232ccd0>
```

6.3 Reference

6.3.1 cleo module

class `cleo.CLSimulator`(*brian_network: brian2.core.network.Network*)

Bases: `object`

The centerpiece of cleo. Integrates simulation components and runs.

Parameters **brian_network** (*Network*) – The Brian network forming the core model

`get_state()` → dict

Return current recorder measurements.

Returns A dictionary of *name*: *state* pairs for all recorders in the simulator.

Return type dict

inject_device(*device: cleo.base.InterfaceDevice*, **neuron_groups: brian2.groups.neurongroup.NeuronGroup*, ***kwparams: Any*) → None

Inject InterfaceDevice into the network, connecting to specified neurons.

Calls `connect_to_neuron_group()` for each group with kwparams and adds the device's `brian_objects` to the simulator's `network`.

Automatically called by `inject_recorder()` and `inject_stimulator()`.

Parameters **device** (`InterfaceDevice`) – Device to inject

inject_recorder(*recorder: cleo.base.Recorder*, **neuron_groups: brian2.groups.neurongroup.NeuronGroup*, ***kwparams*) → None

Inject recorder into given neuron groups.

`Recorder.connect_to_neuron_group()` is called for each *group*.

Parameters

- **recorder** (`Recorder`) – The recorder to inject into the simulation
- ***neuron_groups** (*NeuronGroup*) – The groups to inject the recorder into
- ****kwparams** (*any*) – Passed on to `Recorder.connect_to_neuron_group()` function. Necessary for parameters that can vary by neuron group, such as inhibitory/excitatory cell type

inject_stimulator(*stimulator: cleo.base.Stimulator*, **neuron_groups: brian2.groups.neurongroup.NeuronGroup*, ***kwparams*) → None

Inject stimulator into given neuron groups.

`Stimulator.connect_to_neuron_group()` is called for each *group*.

Parameters

- **stimulator** (`Stimulator`) – The stimulator to inject
- ***neuron_groups** (*NeuronGroup*) – The groups to inject the stimulator into
- ****kwparams** (*any*) – Passed on to `Stimulator.connect_to_neuron_group()` function. Necessary for parameters that can vary by neuron group, such as opsin expression levels.

io_processor: *cleo.base.IOProcessor*

network: brian2.core.network.Network

recorders = 'set[Recorder]'

reset(***kwargs*)

Reset the simulator to a neutral state

Restores the Brian Network to where it was when the CLSimulator was last modified (last injection, IOProcessor change). Calls reset() on stimulators, recorders, and IOProcessor.

run(*duration: brian2.units.fundamentalunits.Quantity*, ***kwparams*) → None

Run simulation.

Parameters

- **duration** (*brian2 temporal Quantity*) – Length of simulation
- ****kwparams** (*additional arguments passed to brian2.run()*) – level has a default value of 1

`set_io_processor`(*io_processor*, *communication_period=None*) → None

Set simulator IO processor

Will replace any previous IOProcessor so there is only one at a time. A Brian NetworkOperation is created to govern communication between the Network and the IOProcessor.

Parameters **io_processor** (`IOProcessor`) –

`stimulators = 'set[Stimulator]'`

`update_stimulators`(*ctrl_signals*) → None

Update stimulators with output from the *IOProcessor*

Parameters **ctrl_signals** (*dict*) – {*stimulator_name*: *ctrl_signal*} dictionary with values to update each stimulator.

class cleo.**IOProcessor**

Bases: `abc.ABC`

Abstract class for implementing sampling, signal processing and control

This must be implemented by the user with their desired closed-loop use case, though most users will find the `LatencyIOProcessor()` class more useful, since delay handling is already defined.

abstract `get_ctrl_signal`(*time*) → dict

Get per-stimulator control signal from the *IOProcessor*.

Parameters **time** (*Brian 2 temporal Unit*) – Current timestep

Returns A {'stimulator_name': value} dictionary for updating stimulators.

Return type dict

abstract `is_sampling_now`(*time*) → bool

Determines whether the processor will take a sample at this timestep.

Parameters **time** (*Brian 2 temporal Unit*) – Current timestep.

Return type bool

abstract `put_state`(*state_dict: dict*, *time*) → None

Deliver network state to the *IOProcessor*.

Parameters

- **state_dict** (*dict*) – A dictionary of recorder measurements, as returned by *get_state()*
- **time** (*brian2 temporal Unit*) – The current simulation timestep. Essential for simulating control latency and for time-varying control.

`reset`(***kwargs*) → None

`sample_period_ms: float`

Determines how frequently the processor takes samples

class cleo.**InterfaceDevice**(*name: str*)

Bases: `abc.ABC`

Base class for devices to be injected into the network

Parameters **name** (*str*) – Unique identifier for the device.

add_self_to_plot(*ax: mpl_toolkits.mplot3d.axes3d.Axes3D*, *axis_scale_unit: brian2.units.fundamentalunits.Unit*, ***kwargs*) → list[matplotlib.artist.Artist]

Add device to an existing plot

Should only be called by `plot()`.

Parameters

- **ax** (*Axes3D*) – The existing matplotlib Axes object
- **axis_scale_unit** (*Unit*) – The unit used to label axes and define chart limits
- ****kwargs** (*optional*) –

Returns A list of artists used to render the device. Needed for use in conjunction with `VideoVisualizer`.

Return type list[Artist]

brian_objects: set

All the Brian objects added to the network by this device. Must be kept up-to-date in `connect_to_neuron_group()` and other functions so that those objects can be automatically added to the network when the device is injected.

abstract connect_to_neuron_group(*neuron_group: brian2.groups.neurongroup.NeuronGroup*, ***kwparams*) → None

Connect device to given *neuron_group*.

If your device introduces any objects which Brian must keep track of, such as a NeuronGroup, Synapses, or Monitor, make sure to add these to *self.brian_objects*.

Parameters

- **neuron_group** (*NeuronGroup*) –
- ****kwparams** (optional, passed from *inject_recorder* or) – *inject_stimulator*

init_for_simulator(*simulator: cleo.base.CLSimulator*) → None

Initialize device for simulator on initial injection

This function is called only the first time a device is injected into a simulator and performs any operations that are independent of the individual neuron groups it is connected to.

Parameters **simulator** (`CLSimulator`) – simulator being injected into

name: str

Unique identifier for device. Used as a key in output/input dicts

sim: cleo.base.CLSimulator

The simulator the device is injected into

update_artists(**args*, ***kwargs*) → list[matplotlib.artist.Artist]

Update the artists used to render the device

Used to set the artists' state at every frame of a video visualization. The current state would be passed in **args* or ***kwargs*

Parameters **artists** (*list[Artist]*) – the artists used to render the device originally, i.e., which were returned from the first `add_self_to_plot()` call.

Returns The artists that were actually updated. Needed for efficient blit rendering, where only updated artists are re-rendered.

Return type list[Artist]

class `cleo.`**`Recorder`**(*name: str*)

Bases: *`cleo.base.InterfaceDevice`*

Device for taking measurements of the network.

Parameters **`name`** (*`str`*) – Unique identifier for the device.

`brian_objects: set`

All the Brian objects added to the network by this device. Must be kept up-to-date in `connect_to_neuron_group()` and other functions so that those objects can be automatically added to the network when the device is injected.

`abstract get_state``()` → Any

Return current measurement.

`name: str`

Unique identifier for device. Used as a key in output/input dicts

`reset`(***kwargs*) → None

Reset the recording device to a neutral state

`sim:` *`cleo.base.CLSimulator`*

The simulator the device is injected into

class `cleo.`**`Stimulator`**(*name: str*, *default_value*, *save_history: bool = False*)

Bases: *`cleo.base.InterfaceDevice`*

Device for manipulating the network

Parameters

- **`name`** (*`str`*) – Unique device name used in *`CLSimulator.update_stimulators()`*
- **`default_value`** (*`any`*) – The stimulator's default value

`default_value: Any`

The default value of the device—used on initialization and on *`reset()`*

`init_for_simulator`(*simulator:* cleo.base.CLSimulator) → None

Initialize device for simulator on initial injection

This function is called only the first time a device is injected into a simulator and performs any operations that are independent of the individual neuron groups it is connected to.

Parameters **`simulator`** (`CLSimulator`) – simulator being injected into

`reset`(***kwargs*) → None

Reset the stimulator device to a neutral state

`save_history: bool`

Determines whether *`t_ms`* and *`values`* are recorded

`t_ms: list[float]`

Times stimulator was updated, stored if *`save_history`*

`update`(*ctrl_signal*) → None

Set the stimulator value.

By default this simply sets *value* to *ctrl_signal*. You will want to implement this method if your stimulator requires additional logic. Use super.update(self, value) to preserve the self.value attribute logic

Parameters **`ctrl_signal`** (*`any`*) – The value the stimulator is to take.

value: Any

The current value of the stimulator device

values: list[Any]

Values taken by the stimulator at each `update()` call, stored if `save_history`

6.3.2 cleo.coords module

Contains functions for assigning neuron coordinates and visualizing

`cleo.coords.`**`assign_coords`**(*neuron_group: brian2.groups.neurongroup.NeuronGroup*, *x: numpy.ndarray*, *y: numpy.ndarray*, *z: numpy.ndarray*, *unit: brian2.units.fundamentalunits.Unit = mmetre*)

Assign arbitrary coordinates to neuron group.

Parameters

- **neuron_group** (*NeuronGroup*) – neurons to be assigned coordinates
- **x** (*np.ndarray*) – x positions to assign (preferably 1D with no unit)
- **y** (*np.ndarray*) – y positions to assign (preferably 1D with no unit)
- **z** (*np.ndarray*) – z positions to assign (preferably 1D with no unit)
- **unit** (*Unit, optional*) – Brian unit determining what scale to use for coordinates, by default mm

`cleo.coords.`**`assign_coords_grid_rect_prism`**(*neuron_group: brian2.groups.neurongroup.NeuronGroup*, *xlim: Tuple[float, float]*, *ylim: Tuple[float, float]*, *zlim: Tuple[float, float]*, *shape: Tuple[int, int, int]*, *unit: brian2.units.fundamentalunits.Unit = mmetre*) → None

Assign grid coordinates to neurons in a rectangular grid

Parameters

- **neuron_group** (*NeuronGroup*) – The neuron group to assign coordinates to
- **xlim** (*Tuple[float, float]*) – xmin, xmax, with no unit
- **ylim** (*Tuple[float, float]*) – ymin, ymax, with no unit
- **zlim** (*Tuple[float, float]*) – zmin, zmax with no unit
- **shape** (*Tuple[int, int, int]*) – n_x, n_y, n_z tuple representing the shape of the resulting grid
- **unit** (*Unit, optional*) – Brian unit determining what scale to use for coordinates, by default mm

Raises **ValueError** – When the shape is incompatible with the number of neurons in the group

`cleo.coords.`**`assign_coords_rand_cylinder`**(*neuron_group: brian2.groups.neurongroup.NeuronGroup*, *xyz_start: Tuple[float, float, float]*, *xyz_end: Tuple[float, float, float]*, *radius: float*, *unit: brian2.units.fundamentalunits.Unit = mmetre*) → None

Assign random coordinates within a cylinder.

Parameters

- **neuron_group** (*NeuronGroup*) – neurons to assign coordinates to

- **xyz_start** (*Tuple[float, float, float]*) – starting position of cylinder without unit
- **xyz_end** (*Tuple[float, float, float]*) – ending position of cylinder without unit
- **radius** (*float*) – radius of cylinder without unit
- **unit** (*Unit, optional*) – Brian unit to scale other params, by default mm

cleo.coords.**assign_coords_rand_rect_prism**(*neuron_group: brian2.groups.neurongroup.NeuronGroup, xlim: Tuple[float, float], ylim: Tuple[float, float], zlim: Tuple[float, float], unit: brian2.units.fundamentalunits.Unit = mmetre*) → None

Assign random coordinates to neurons within a rectangular prism

Parameters

- **neuron_group** (*NeuronGroup*) – neurons to assign coordinates to
- **xlim** (*Tuple[float, float]*) – xmin, xmax without unit
- **ylim** (*Tuple[float, float]*) – ymin, ymax without unit
- **zlim** (*Tuple[float, float]*) – zmin, zmax without unit
- **unit** (*Unit, optional*) – Brian unit to specify scale implied in limits, by default mm

cleo.coords.**assign_coords_uniform_cylinder**(*neuron_group: brian2.groups.neurongroup.NeuronGroup, xyz_start: Tuple[float, float, float], xyz_end: Tuple[float, float, float], radius: float, unit: brian2.units.fundamentalunits.Unit = mmetre*) → None

Assign uniformly spaced coordinates within a cylinder.

Parameters

- **neuron_group** (*NeuronGroup*) – neurons to assign coordinates to
- **xyz_start** (*Tuple[float, float, float]*) – starting position of cylinder without unit
- **xyz_end** (*Tuple[float, float, float]*) – ending position of cylinder without unit
- **radius** (*float*) – radius of cylinder without unit
- **unit** (*Unit, optional*) – Brian unit to scale other params, by default mm

6.3.3 cleo.ephys module

Contains probes, coordinate convenience functions, signals, spiking, and LFP

class cleo.ephys.**MultiUnitSpiking**(*name: str, perfect_detection_radius: brian2.units.fundamentalunits.Quantity, half_detection_radius: Optional[brian2.units.fundamentalunits.Quantity] = None, cutoff_probability: float = 0.01, save_history: bool = False*)

Bases: `cleo.ephys.spiking.Spiking`

Detects spikes per channel, that is, unsorted.

Parameters

- **name** (*str*) – Unique identifier for signal
- **perfect_detection_radius** (*Quantity*) – Radius (with Brian unit) within which all spikes are detected

- **half_detection_radius** (*Quantity, optional*) – Radius (with Brian unit) within which half of all spikes are detected
- **cutoff_probability** (*float, optional*) – Spike detection probability below which neurons will not be considered, by default 0.01. For computational efficiency.
- **save_history** (*bool, optional*) – If True, will save t_ms (spike times), i (neuron or channel index), and t_samp_ms (sample times) as attributes. By default False

connect_to_neuron_group(*neuron_group: brian2.groups.neurongroup.NeuronGroup*, ***kwparams*) → None

Configure signal to record from specified neuron group

Parameters **neuron_group** (*NeuronGroup*) – group to record from

cutoff_probability: float

Spike detection probability below which neurons will not be considered. For computational efficiency.

get_state() → tuple[NDArray[Any, ..., UInt[64]], NDArray[Any, ..., Float[64]], NDArray[Any, ..., UInt[64]]]

Return spikes since method was last called (i, t_ms, y)

Returns (i, t_ms, y) where i is channel (for multi-unit) or neuron (for sorted) spike indices, t_ms is spike times, and y is a spike count vector suitable for control- theoretic uses—i.e., a 0 for every channel/neuron that hasn't spiked and a 1 for a single spike.

Return type tuple[NDArray[np.uint], NDArray[float], NDArray[np.uint]]

half_detection_radius: Quantity

Radius (with Brian unit) within which half of all spikes are detected

i: NDArray[Any, np.uint]

Channel (for multi-unit) or neuron (for sorted) indices of spikes, stored if *save_history*

i_probe_by_i_ng: bidict

(neuron_group, i_ng) keys, i_probe values. bidict for converting between neuron group indices and the indices the probe uses

perfect_detection_radius: Quantity

Radius (with Brian unit) within which all spikes are detected

save_history: bool

Determines whether *t_ms*, *i*, and *t_samp_ms* are recorded

t_ms: NDArray[Any, float]

Spike times in ms, stored if *save_history*

t_samp_ms: NDArray[Any, float]

Sample times in ms when each spike was recorded, stored if *save_history*

class cleo.ephys.**Probe**(*name: str*, *coords: brian2.units.fundamentalunits.Quantity*, *signals: collections.abc.Iterable[cleo.ephys.probes.Signal] = []*)

Bases: *cleo.base.Recorder*

Picks up specified signals across an array of electrodes.

Visualization kwargs

- **marker** (*str, optional*) – The marker used to represent each contact. "x" by default.
- **size** (*float, optional*) – The size of each contact marker. 40 by default.

- **color** (*Any, optional*) – The color of contact markers. "xkcd:dark gray" by default.

Parameters

- **name** (*str*) – Unique identifier for device
- **coords** (*Quantity*) – Coordinates of n electrodes. Must be an n x 3 array (with unit) where columns represent x, y, and z
- **signals** (*Iterable[Signal], optional*) – Signals to record with probe, by default []. Can be specified later with `add_signals()`.

Raises **ValueError** – When coords aren't n x 3

add_self_to_plot(*ax: mpl_toolkits.mplot3d.axes3d.Axes3D, axis_scale_unit: brian2.units.fundamentalunits.Unit, **kwargs*) → list[matplotlib.artist.Artist]

Add device to an existing plot

Should only be called by `plot()`.

Parameters

- **ax** (*Axes3D*) – The existing matplotlib Axes object
- **axis_scale_unit** (*Unit*) – The unit used to label axes and define chart limits
- ****kwargs** (*optional*) –

Returns A list of artists used to render the device. Needed for use in conjunction with `VideoVisualizer`.

Return type list[Artist]

add_signals(**signals: cleo.ephys.probes.Signal*) → None

Add signals to the probe for recording

Parameters ***signals** (*Signal*) – signals to add

connect_to_neuron_group(*neuron_group: brian2.groups.neurongroup.NeuronGroup, **kwparams: Any*) → None

Configure probe to record from given neuron group

Will call `Signal.connect_to_neuron_group()` for each signal

Parameters

- **neuron_group** (*NeuronGroup*) – neuron group to connect to, i.e., record from
- ****kwparams** (*Any*) – Passed in to signals' connect functions, needed for some signals

coords: Quantity

n x 3 array (with Brian length unit) specifying contact locations

get_state() → dict

Get current state from probe, i.e., all signals

Returns {'signal_name': value} dict with signal states

Return type dict

n: int

Number of electrode contacts in the probe `reset(**kwargs)`

Reset the probe to a neutral state

Calls reset() on each signal

`signals: list[Signal]`

Signals recorded by the probe

`property xs: brian2.units.fundamentalunits.Quantity`

x coordinates of recording contacts

Returns x coordinates represented as a Brian quantity, that is, including units. Should be like a 1D array.

Return type Quantity

`property ys: brian2.units.fundamentalunits.Quantity`

y coordinates of recording contacts

Returns y coordinates represented as a Brian quantity, that is, including units. Should be like a 1D array.

Return type Quantity

`property zs: brian2.units.fundamentalunits.Quantity`

z coordinates of recording contacts

Returns z coordinates represented as a Brian quantity, that is, including units. Should be like a 1D array.

Return type Quantity

`class cleo.ephys.Signal(name: str)`

Bases: `abc.ABC`

Base class representing something an electrode can record

Constructor must be called at beginning of children constructors.

Parameters **name** (*str*) – Unique identifier used when reading the state from the network

`brian_objects: set`

All Brian objects created by the signal. Must be kept up-to-date for automatic injection into the network

`abstract connect_to_neuron_group(neuron_group: brian2.groups.neurongroup.NeuronGroup, **kwparams)`

Configure signal to record from specified neuron group

Parameters **neuron_group** (*NeuronGroup*) – group to record from

`abstract get_state()` → Any

Get the signal's current value

`init_for_probe(probe: cleo.ephys.probes.Probe)` → None

Called when attached to a probe.

Ensures signal can access probe and is only attached to one

Parameters **probe** (`Probe`) – Probe to attach to

Raises `ValueError` – When signal already attached to another probe

name: str

Unique identifier used to organize probe output

probe: *cleo.ephys.probes.Probe*

The probe the signal is configured to record for.

reset(***kwargs*) → None

Reset signal to a neutral state

class cleo.ephys.**SortedSpiking**(*name: str*, *perfect_detection_radius: brian2.units.fundamentalunits.Quantity*, *half_detection_radius: Optional[brian2.units.fundamentalunits.Quantity] = None*, *cutoff_probability: float = 0.01*, *save_history: bool = False*)

Bases: *cleo.ephys.spiking.Spiking*

Detect spikes identified by neuron indices.

The indices used by the probe do not correspond to those coming from neuron groups, since the probe must consider multiple potential groups and within a group ignores those neurons that are too far away to be easily detected.

Parameters

- **name** (*str*) – Unique identifier for signal
- **perfect_detection_radius** (*Quantity*) – Radius (with Brian unit) within which all spikes are detected
- **half_detection_radius** (*Quantity, optional*) – Radius (with Brian unit) within which half of all spikes are detected
- **cutoff_probability** (*float, optional*) – Spike detection probability below which neurons will not be considered, by default 0.01. For computational efficiency.
- **save_history** (*bool, optional*) – If True, will save t_ms (spike times), i (neuron or channel index), and t_samp_ms (sample times) as attributes. By default False

connect_to_neuron_group(*neuron_group: brian2.groups.neurongroup.NeuronGroup*, ***kwparams*) → None

Configure sorted spiking signal to record from given neuron group

Parameters **neuron_group** (*NeuronGroup*) – group to record from

cutoff_probability: float

Spike detection probability below which neurons will not be considered. For computational efficiency.

get_state() → tuple[NDArray[Any, ..., UInt[64]], NDArray[Any, ..., Float[64]], NDArray[Any, ..., UInt[64]]]

Return spikes since method was last called (i, t_ms, y)

Returns (i, t_ms, y) where i is channel (for multi-unit) or neuron (for sorted) spike indices, t_ms is spike times, and y is a spike count vector suitable for control- theoretic uses—i.e., a 0 for every channel/neuron that hasn't spiked and a 1 for a single spike.

Return type tuple[NDArray[np.uint], NDArray[float], NDArray[np.uint]]

half_detection_radius: Quantity

Radius (with Brian unit) within which half of all spikes are detected

i: NDArray[Any, np.uint]

Channel (for multi-unit) or neuron (for sorted) indices of spikes, stored if *save_history*

i_probe_by_i_ng: bidict

(neuron_group, i_ng) keys, i_probe values. bidict for converting between neuron group indices and the indices the probe uses

perfect_detection_radius: Quantity

Radius (with Brian unit) within which all spikes are detected

save_history: bool

Determines whether `t_ms`, `i`, and `t_samp_ms` are recorded

t_ms: NDArray[Any, float]

Spike times in ms, stored if `save_history`

t_samp_ms: NDArray[Any, float]

Sample times in ms when each spike was recorded, stored if `save_history`

class cleo.ephys.Spiking(*name: str*, *perfect_detection_radius: brian2.units.fundamentalunits.Quantity*, *half_detection_radius: Optional[brian2.units.fundamentalunits.Quantity] = None*, *cutoff_probability: float = 0.01*, *save_history: bool = False*)

Bases: `cleo.ephys.probes.Signal`

Base class for probabilistically detecting spikes

Parameters

- **name** (*str*) – Unique identifier for signal
- **perfect_detection_radius** (*Quantity*) – Radius (with Brian unit) within which all spikes are detected
- **half_detection_radius** (*Quantity, optional*) – Radius (with Brian unit) within which half of all spikes are detected
- **cutoff_probability** (*float, optional*) – Spike detection probability below which neurons will not be considered, by default 0.01. For computational efficiency.
- **save_history** (*bool, optional*) – If True, will save t_ms (spike times), i (neuron or channel index), and t_samp_ms (sample times) as attributes. By default False

connect_to_neuron_group(*neuron_group: brian2.groups.neurongroup.NeuronGroup*, ***kwparams*) → numpy.ndarray

Configure signal to record from specified neuron group

Parameters **neuron_group** (*NeuronGroup*) – Neuron group to record from

Returns num_neurons_to_consider x num_channels array of spike detection probabilities, for use in subclasses

Return type np.ndarray

cutoff_probability: float

Spike detection probability below which neurons will not be considered. For computational efficiency.

abstract get_state() → tuple[NDArray[Any, ..., UInt[64]], NDArray[Any, ..., Float[64]], NDArray[Any, ..., UInt[64]]]

Return spikes since method was last called (i, t_ms, y)

Returns (i, t_ms, y) where i is channel (for multi-unit) or neuron (for sorted) spike indices, t_ms is spike times, and y is a spike count vector suitable for control- theoretic uses—i.e., a 0 for every channel/neuron that hasn't spiked and a 1 for a single spike.

Return type tuple[NDArray[np.uint], NDArray[float], NDArray[np.uint]]

half_detection_radius: Quantity

Radius (with Brian unit) within which half of all spikes are detected

i: NDArray[Any, np.uint]

Channel (for multi-unit) or neuron (for sorted) indices of spikes, stored if *save_history*

i_probe_by_i_ng: bidict

(neuron_group, i_ng) keys, i_probe values. bidict for converting between neuron group indices and the indices the probe uses

perfect_detection_radius: Quantity

Radius (with Brian unit) within which all spikes are detected

reset(***kwargs*) → None

Reset signal to a neutral state

save_history: bool

Determines whether *t_ms*, *i*, and *t_samp_ms* are recorded

t_ms: NDArray[Any, float]

Spike times in ms, stored if *save_history*

t_samp_ms: NDArray[Any, float]

Sample times in ms when each spike was recorded, stored if *save_history*

class `cleo.ephys.`**TKLFPSignal**(*name: str*, *uLFP_threshold_uV: float = 0.001*, *save_history: bool = False*)

Bases: *cleo.ephys.probes.Signal*

Records the Teleńczuk kernel LFP approximation.

Requires `tklfp_type='exc'|'inh'` to specify cell type on injection.

An `orientation` keyword argument can also be specified on injection, which should be an array of shape `(n_neurons, 3)` representing which way is "up," that is, towards the surface of the cortex, for each neuron. If a single vector is given, it is taken to be the orientation for all neurons in the group. [0, 0, -1] is the default, meaning the negative z axis is "up." As stated elsewhere, Cleo's convention is that z=0 corresponds to the cortical surface and increasing z values represent increasing depth.

TKLFP is computed from spikes using the tklfp package.

Parameters

- **name** (*str*) – Unique identifier for signal, used to identify signal output in *Probe.get_state()*
- **uLFP_threshold_uV** (*float, optional*) – Sets *uLFP_threshold_uV*, by default 1e-3
- **save_history** (*bool, optional*) – Sets *save_history* to determine whether output is recorded, by default False

connect_to_neuron_group(*neuron_group: brian2.groups.neurongroup.NeuronGroup*, ***kwparams*)

Configure signal to record from specified neuron group

Parameters **neuron_group** (*NeuronGroup*) – group to record from

get_state() → numpy.ndarray

Get the signal's current value

init_for_probe(*probe: cleo.ephys.probes.Probe*)

Called when attached to a probe.

Ensures signal can access probe and is only attached to one

Parameters **probe** (Probe) – Probe to attach to

Raises **ValueError** – When signal already attached to another probe

lfp_uV: nptyping.types._ndarray.NDArray[Any, Any, nptyping.types._number.Float]

Approximated LFP from every call to *get_state()*, recorded if *save_history*. Shape is (n_samples, n_channels).

reset(***kwargs*) → None

Reset signal to a neutral state

save_history: bool

Whether to record output from every timestep in *lfp_uV*. Output is stored every time *get_state()* is called.

t_ms: nptyping.types._ndarray.NDArray[Any, nptyping.types._number.Float]

Times at which LFP is recorded, in ms, stored if *save_history*

uLFP_threshold_uV: float

Threshold, in microvolts, above which the uLFP for a single spike is guaranteed to be considered. This determines the buffer length of past spikes, since the uLFP from a long-past spike becomes negligible and is ignored.

cleo.ephys.**concat_coords**(**coords: brian2.units.fundamentalunits.Quantity*) → brian2.units.fundamentalunits.Quantity

Combine multiple coordinate Quantity arrays into one

Parameters ***coords** (*Quantity*) – Multiple coordinate n x 3 Quantity arrays to combine

Returns A single n x 3 combined Quantity array

Return type Quantity

cleo.ephys.**linear_shank_coords**(*array_length: brian2.units.fundamentalunits.Quantity, channel_count: int, start_location: brian2.units.fundamentalunits.Quantity = array([0., 0., 0.]) * metre, direction: Tuple[float, float, float] = (0, 0, 1)*) → brian2.units.fundamentalunits.Quantity

Generate coordinates in a linear pattern

Parameters

- **array_length** (*Quantity*) – Distance from the first to the last contact (with a Brian unit)
- **channel_count** (*int*) – Number of coordinates to generate, i.e. electrode contacts
- **start_location** (*Quantity, optional*) – x, y, z coordinate (with unit) for the start of the electrode array, by default (0, 0, 0)*mm
- **direction** (*Tuple[float, float, float], optional*) – x, y, z vector indicating the direction in which the array extends, by default (0, 0, 1), meaning pointing straight down

Returns channel_count x 3 array of coordinates, where the 3 columns represent x, y, and z

Return type Quantity

cleo.ephys.**poly2_shank_coords**(*array_length: brian2.units.fundamentalunits.Quantity, channel_count: int, intercol_space: brian2.units.fundamentalunits.Quantity, start_location: brian2.units.fundamentalunits.Quantity = array([0., 0., 0.]) * metre, direction: Tuple[float, float, float] = (0, 0, 1)*) → brian2.units.fundamentalunits.Quantity

Generate NeuroNexus-style Poly2 array coordinates

Poly2 refers to 2 parallel columns with staggered contacts. See https://www.neuronexus.com/products/electrode-arrays/up-to-15-mm-depth for more detail.

Parameters

- **array_length** (*Quantity*) – Length from the beginning to the end of the two-column array, as measured in the center
- **channel_count** (*int*) – Total (not per-column) number of coordinates (recording contacts) desired
- **intercol_space** (*Quantity*) – Distance between columns (with Brian unit)
- **start_location** (*Quantity, optional*) – Where to place the beginning of the array, by default (0, 0, 0)*mm
- **direction** (*Tuple[float, float, float], optional*) – x, y, z vector indicating the direction in which the two columns extend; by default (0, 0, 1), meaning straight down.

Returns channel_count x 3 array of coordinates, where the 3 columns represent x, y, and z

Return type Quantity

cleo.ephys.**poly3_shank_coords**(*array_length: brian2.units.fundamentalunits.Quantity, channel_count: int, intercol_space: brian2.units.fundamentalunits.Quantity, start_location: brian2.units.fundamentalunits.Quantity = array([0., 0., 0.]) * metre, direction: Tuple[float, float, float] = (0, 0, 1)*) → brian2.units.fundamentalunits.Quantity

Generate NeuroNexus Poly3-style array coordinates

Poly3 refers to three parallel columns of electrodes. The middle column will be longest if the channel count isn't divisible by three and the side columns will be centered vertically with respect to the middle.

Parameters

- **array_length** (*Quantity*) – Length from beginning to end of the array as measured along the center column
- **channel_count** (*int*) – Total (not per-column) number of coordinates to generate (i.e., electrode contacts)
- **intercol_space** (*Quantity*) – Spacing between columns, with Brian unit
- **start_location** (*Quantity, optional*) – Location of beginning of the array, that is, the first contact in the center column, by default (0, 0, 0)*mm
- **direction** (*Tuple[float, float, float], optional*) – x, y, z vector indicating the direction along which the array extends, by default (0, 0, 1), meaning straight down

Returns channel_count x 3 array of coordinates, where the 3 columns represent x, y, and z

Return type Quantity

cleo.ephys.**tetrode_shank_coords**(*array_length: brian2.units.fundamentalunits.Quantity, tetrode_count: int, start_location: brian2.units.fundamentalunits.Quantity = array([0., 0., 0.]) * metre, direction: Tuple[float, float, float] = (0, 0, 1), tetrode_width: brian2.units.fundamentalunits.Quantity = 25. * umetre*) → brian2.units.fundamentalunits.Quantity

Generate coordinates for a linear array of tetrodes

See https://www.neuronexus.com/products/electrode-arrays/up-to-15-mm-depth to visualize NeuroNexus-style arrays.

Parameters

- **array_length** (*Quantity*) – Distance from the center of the first tetrode to the last (with a Brian unit)
- **tetrode_count** (*int*) – Number of tetrodes desired
- **start_location** (*Quantity, optional*) – Center location of the first tetrode in the array, by default (0, 0, 0)*mm
- **direction** (*Tuple[float, float, float], optional*) – x, y, z vector determining the direction in which the linear array extends, by default (0, 0, 1), meaning straight down.
- **tetrode_width** (*Quantity, optional*) – Distance between contacts in a single tetrode. Not the diagonal distance, but the length of one side of the square. By default 25*umeter, as in NeuroNexus probes.

Returns (tetrode_count*4) x 3 array of coordinates, where 3 columns represent x, y, and z

Return type Quantity

cleo.ephys.**tile_coords**(*coords: brian2.units.fundamentalunits.Quantity*, *num_tiles: int*, *tile_vector: brian2.units.fundamentalunits.Quantity*) → brian2.units.fundamentalunits.Quantity

Tile (repeat) coordinates to produce multi-shank/matrix arrays

Parameters

- **coords** (*Quantity*) – The n x 3 coordinates array to tile
- **num_tiles** (*int*) – Number of times to tile (repeat) the coordinates. For example, if you are tiling linear shank coordinates to produce multi-shank coordinates, this would be the desired number of shanks
- **tile_vector** (*Quantity*) – x, y, z array with Brian unit determining both the length and direction of the tiling

Returns (n * num_tiles) x 3 array of coordinates, where the 3 columns represent x, y, and z

Return type Quantity

6.3.4 cleo.opto module

Contains opsin models, parameters, and OptogeneticIntervention device

cleo.opto.**ChR2_four_state = {'E': 0. * volt, 'Gb0': 16.1 * hertz, 'Gd1': 105. * hertz, 'Gd2': 13.8 * hertz, 'Gf0': 37.3 * hertz, 'Gr0': 0.33 * hertz, 'g0': 114. * nsiemens, 'gamma': 0.00742, 'k1': 4.15 * khertz, 'k2': 0.868 * khertz, 'kb': 63. * hertz, 'kf': 58.1 * hertz, 'p': 0.833, 'phim': 2.33e+23 * metre ** -2 * second ** -1, 'q': 1.94, 'v0': 43. * mvolt, 'v1': 17.1 * mvolt}**

Parameters for the 4-state ChR2 model.

Taken from try.projectpyrho.org's default 4-state params.

class cleo.opto.**FourStateModel**(*params: dict*)

Bases: *cleo.opto.MarkovModel*

4-state model from PyRhO (Evans et al. 2016).

rho_rel is channel density relative to standard model fit; modifying it post-injection allows for heterogeneous opsin expression.

IOPTO_VAR_NAME and V_VAR_NAME are substituted on injection.

Parameters **params** (*dict*) – dict defining params in the *model*

init_opto_syn_vars(*opto_syn: brian2.synapses.synapses.Synapses*) → None

Initializes appropriate variables in Synapses implementing the model

Can also be used to reset the variables.

Parameters **opto_syn** (*Synapses*) – The synapses object implementing this model

model: str = '\n dC1/dt = Gd1*O1 + Gr0*C2 - Ga1*C1 : 1 (clock-driven)\n dO1/dt = Ga1*C1 + Gb*O2 - (Gd1+Gf)*O1 : 1 (clock-driven)\n dO2/dt = Ga2*C2 + Gf*O1 - (Gd2+Gb)*O2 : 1 (clock-driven)\n C2 = 1 - C1 - O1 - O2 : 1\n # dC2/dt = Gd2*O2 - (Gr0+Ga2)*C2 : 1 (clock-driven)\n\n Theta = int(phi > 0*phi) : 1\n Hp = Theta * phip/(phi**p + phim**p) : 1\n Ga1 = k1*Hp : hertz\n Ga2 = k2*Hp : hertz\n Hq = Theta * phi**q/(phi**q + phim**q) : 1\n Gf = kf*Hq + Gf0 : hertz\n Gb = kb*Hq + Gb0 : hertz\n\n fphi = O1 + gamma*O2 : 1\n fv = (1 - exp(-(V_VAR_NAME_post-E)/v0)) / -2 : 1\n\n IOPTO_VAR_NAME_post = -g0*fphi*fv*(V_VAR_NAME_post-E)*rho_rel : ampere (summed)\n rho_rel : 1\n '**

Basic Brian model equations string.

Should contain a *rho_rel* term reflecting relative expression levels. Will likely also contain special NeuronGroup-dependent symbols such as V_VAR_NAME to be replaced on injection in `modify_model_and_params_for_ng()`.

class cleo.opto.**MarkovModel**(*params: dict*)

Bases: `cleo.opto.OpsinModel`

Base class for Markov state models à la Evans et al., 2016

Parameters **params** (*dict*) – dict defining params in the `model`

required_vars: list[Tuple[str, brian2.units.fundamentalunits.Unit]] = [('Iopto', amp), ('v', volt)]

Default names of state variables required in the neuron group, along with units, e.g., [('Iopto', amp)].

It is assumed that non-default values can be passed in on injection as a keyword argument `[default_name]_var_name=[non_default_name]` and that these are found in the model string as `[DEFAULT_NAME]_VAR_NAME` before replacement.

class cleo.opto.**OpsinModel**

Bases: `abc.ABC`

Base class for opsin model

init_opto_syn_vars(*opto_syn: brian2.synapses.synapses.Synapses*) → None

Initializes appropriate variables in Synapses implementing the model

Can also be used to reset the variables.

Parameters **opto_syn** (*Synapses*) – The synapses object implementing this model

model: str

Basic Brian model equations string.

Should contain a *rho_rel* term reflecting relative expression levels. Will likely also contain special NeuronGroup-dependent symbols such as V_VAR_NAME to be replaced on injection in `modify_model_and_params_for_ng()`.

`modify_model_and_params_for_ng`(*neuron_group: brian2.groups.neurongroup.NeuronGroup, injct_params: dict, model='class-defined'*) → Tuple[brian2.equations.equations.Equations, dict]

Adapt model for given neuron group on injection

This enables the specification of variable names differently for each neuron group, allowing for custom names and avoiding conflicts.

Parameters

- **neuron_group** (*NeuronGroup*) – NeuronGroup this opsin model is being connected to
- **injct_params** (*dict*) – kwargs passed in on injection, could contain variable names to plug into the model

Keyword Arguments **model** (*str, optional*) – Model to start with, by default that defined for the class. This allows for prior string manipulations before it can be parsed as an *Equations* object.

Returns A tuple containing an Equations object and a parameter dictionary, constructed from `model` and `params`, respectively, with modified names for use in `opto_syns`

Return type Equations, dict

params: dict

Parameter values for model, passed in as a namespace dict

required_vars: list[Tuple[str, brian2.units.fundamentalunits.Unit]]

Default names of state variables required in the neuron group, along with units, e.g., [('Iopto', amp)].

It is assumed that non-default values can be passed in on injection as a keyword argument `[default_name]_var_name=[non_default_name]` and that these are found in the model string as `[DEFAULT_NAME]_VAR_NAME` before replacement.

class cleo.opto.**OptogeneticIntervention**(*name: str, opsin_model: cleo.opto.OpsinModel, light_model_params: dict, location: brian2.units.fundamentalunits.Quantity = array([0., 0., 0.]) * metre, direction: Tuple[float, float, float] = (0, 0, 1), max_Irr0_mW_per_mm2: Optional[float] = None, save_history: bool = False*)

Bases: `cleo.base.Stimulator`

Enables optogenetic stimulation of the network.

Essentially "transfects" neurons and provides a light source. Under the hood, it delivers current via a Brian `Synapses` object.

Requires neurons to have 3D spatial coordinates already assigned. Also requires that the neuron model has a current term (by default Iopto) which is assumed to be positive (unlike the convention in many opsin modeling papers, where the current is described as negative).

See `connect_to_neuron_group()` for optional keyword parameters that can be specified when calling `cleo.CLSimulator.inject_stimulator()`.

Visualization kwargs

- **n_points** (*int, optional*) – The number of points used to represent light intensity in space. By default 1e4.
- **T_threshold** (*float, optional*) – The transmittance below which no points are plotted. By default 1e-3.

- **intensity** (*float, optional*) – How bright the light appears, should be between 0 and 1. By default 0.5.
- **rasterized** (*bool, optional*) – Whether to render as rasterized in vector output, True by default. Useful since so many points makes later rendering and editing slow.

Parameters

- **name** (*str*) – Unique identifier for stimulator
- **opsin_model** (OpsinModel) – OpsinModel object defining how light affects target neurons. See *FourStateModel* and *ProportionalCurrentModel* for examples.
- **light_model_params** (*dict*) – Parameters for the light propagation model in Foutz et al., 2012. See *default_blue* for an example.
- **location** (*Quantity, optional*) – (x, y, z) coords with Brian unit specifying where to place the base of the light source, by default (0, 0, 0)*mm
- **direction** (*Tuple[float, float, float], optional*) – (x, y, z) vector specifying direction in which light source is pointing, by default (0, 0, 1)
- **max_Irr0_mW_per_mm2** (*float, optional*) – Set *max_Irr0_mW_per_mm2*.
- **save_history** (*bool, optional*) – Determines whether `values` and `t_ms` are saved.

add_self_to_plot(*ax*, *axis_scale_unit*, ***kwargs*) → matplotlib.collections.PathCollection

Add device to an existing plot

Should only be called by *plot()*.

Parameters

- **ax** (*Axes3D*) – The existing matplotlib Axes object
- **axis_scale_unit** (*Unit*) – The unit used to label axes and define chart limits
- ****kwargs** (*optional*) –

Returns A list of artists used to render the device. Needed for use in conjunction with *VideoVisualizer*.

Return type list[Artist]

connect_to_neuron_group(*neuron_group: brian2.groups.neurongroup.NeuronGroup*, ***kwparams: Any*) → None

Configure opsin and light source to stimulate given neuron group.

Parameters **neuron_group** (*NeuronGroup*) – The neuron group to stimulate with the given opsin and light source

Keyword Arguments

- **p_expression** (*float*) – Probability (0 <= p <= 1) that a given neuron in the group will express the opsin. 1 by default.
- **rho_rel** (*float*) – The expression level, relative to the standard model fit, of the opsin. 1 by default. For heterogeneous expression, this would have to be modified in the opsin synapse post-injection, e.g., `opto.opto_syns["neuron_group_name"].rho_rel = ...`
- **Iopto_var_name** (*str*) – The name of the variable in the neuron group model representing current from the opsin

- **v_var_name** (*str*) – The name of the variable in the neuron group model representing membrane potential

max_Irr0_mW_per_mm2: float

The maximum irradiance the light source can emit.

Usually determined by hardware in a real experiment.

max_Irr0_mW_per_mm2_viz: float

Maximum irradiance for visualization purposes.

i.e., the level at or above which the light appears maximally bright. Only relevant in video visualization.

opto_syns: dict[str, Synapses]

Stores the synapse objects implementing the opsin model, with NeuronGroup name keys and Synapse values.

reset(***kwargs*)

Reset the stimulator device to a neutral state

update(*Irr0_mW_per_mm2: float*)

Set the light intensity, in mW/mm2 (without unit)

Parameters **Irr0_mW_per_mm2** (*float*) – Desired light intensity for light source

update_artists(*artists: list[matplotlib.artist.Artist], value, *args, **kwargs*) → list[matplotlib.artist.Artist]

Update the artists used to render the device

Used to set the artists' state at every frame of a video visualization. The current state would be passed in **args* or ***kwargs*

Parameters **artists** (*list[Artist]*) – the artists used to render the device originally, i.e., which were returned from the first `add_self_to_plot()` call.

Returns The artists that were actually updated. Needed for efficient blit rendering, where only updated artists are re-rendered.

Return type list[Artist]

class cleo.opto.**ProportionalCurrentModel**(*Iopto_per_mW_per_mm2: brian2.units.fundamentalunits.Quantity*)

Bases: `cleo.opto.OpsinModel`

A simple model delivering current proportional to light intensity

Parameters **Iopto_per_mW_per_mm2** (*Quantity*) – How much current (in amps or unitless, depending on neuron model) to deliver per mW/mm2

model: str = '\n IOPTO_VAR_NAME_post = gain * Irr * rho_rel : IOPTO_UNIT (summed)\n rho_rel : 1\n '

Basic Brian model equations string.

Should contain a *rho_rel* term reflecting relative expression levels. Will likely also contain special NeuronGroup-dependent symbols such as V_VAR_NAME to be replaced on injection in `modify_model_and_params_for_ng()`.

modify_model_and_params_for_ng(*neuron_group: brian2.groups.neurongroup.NeuronGroup, injct_params: dict*) → Tuple[brian2.equations.equations.Equations, dict]

Adapt model for given neuron group on injection

This enables the specification of variable names differently for each neuron group, allowing for custom names and avoiding conflicts.

Parameters

- **neuron_group** (*NeuronGroup*) – NeuronGroup this opsin model is being connected to
- **injct_params** (*dict*) – kwargs passed in on injection, could contain variable names to plug into the model

Keyword Arguments **model** (*str, optional*) – Model to start with, by default that defined for the class. This allows for prior string manipulations before it can be parsed as an *Equations* object.

Returns A tuple containing an Equations object and a parameter dictionary, constructed from `model` and `params`, respectively, with modified names for use in `opto_syns`

Return type Equations, dict

`cleo.opto.`**`default_blue = {'K': 125. * metre ** -1, 'NAfib':  0.37, 'R0':  100. * umetre, 'S': 7370. * metre ** -1, 'ntis':  1.36, 'wavelength':  0.473 * umetre}`**

Light parameters for 473 nm wavelength delivered via an optic fiber.

From Foutz et al., 2012

6.3.5 cleo.ioproc module

Classes and functions for constructing and configuring an `IOProcessor`.

class `cleo.ioproc.`**`ConstantDelay`**(*delay_ms: float*)

Bases: `cleo.ioproc.delays.Delay`

Simply adds a constant delay to the computation

Parameters **delay_ms** (*float*) – Desired delay in milliseconds

compute()

Compute delay.

class `cleo.ioproc.`**`Delay`**

Bases: `abc.ABC`

Abstract base class for computing delays.

abstract compute() → float

Compute delay.

class `cleo.ioproc.`**`FiringRateEstimator`**(*tau_ms: float*, *sample_period_ms: float*, ***kwargs*)

Bases: `cleo.ioproc.base.ProcessingBlock`

Exponential filter to estimate firing rate.

Requires *sample_time_ms* kwarg when process is called.

Parameters

- **tau_ms** (*float*) – Time constant of filter
- **sample_period_ms** (*float*) – Sampling period in milliseconds

compute_output(*input: nptyping.types._ndarray.NDArray[Any, nptyping.types._number.UInt]*, ***kwargs*) → nptyping.types._ndarray.NDArray[Any, nptyping.types._number.Float]

Estimate firing rate given past and current spikes.

Parameters **input** (`NDArray[(n,), np.uint]`) – n-length vector of spike counts

Keyword Arguments **sample_time_ms** (`float`) – Time measurement was taken in milliseconds

Returns n-length vector of firing rates

Return type NDArray[(n,), float]

delay: `Delay`

The delay object determining compute latency for the block

save_history: **bool**

Whether to record `t_in_ms`, `t_out_ms`, and `values` with every timestep

t_in_ms: **list[float]**

The walltime the block received each input. Only recorded if `save_history`

t_out_ms: **list[float]**

The walltime of each of the block's outputs. Only recorded if `save_history`

values: **list[Any]**

Each of the block's outputs. Only recorded if `save_history`

class cleo.ioproc.**GaussianDelay**(*loc: float*, *scale: float*)

Bases: `cleo.ioproc.delays.Delay`

Generates normal-distributed delay.

Will return 0 when a negative value is sampled.

Parameters

- **loc** (`float`) – Center of distribution
- **scale** (`float`) – Standard deviation of delay distribution

compute() → float

Compute delay.

class cleo.ioproc.**LatencyIOProcessor**(*sample_period_ms: float*, ***kwargs*)

Bases: `cleo.base.IOProcessor`

IOProcessor capable of delivering stimulation some time after measurement.

Parameters **sample_period_ms** (`float`) – Determines how frequently samples are taken from the network.

Keyword Arguments

- **sampling** (`str`) – "fixed" or "when idle"; "fixed" by default

  "fixed" sampling means samples are taken on a fixed schedule, with no exceptions.

  "when idle" sampling means no samples are taken before the previous sample's output has been delivered. A sample is taken ASAP after an over-period computation: otherwise remains on schedule.

- **processing** (*str*) – "parallel" or "serial"; "parallel" by default

  "parallel" computes the output time by adding the delay for a sample onto the sample time, so if the delay is 2 ms, for example, while the sample period is only 1 ms, some of the processing is happening in parallel. Output order matches input order even if the computed output time for a sample is sooner than that for a previous sample.

  "serial" computes the output time by adding the delay for a sample onto the output time of the previous sample, rather than the sampling time. Note this may be of limited utility because it essentially means the *entire* round trip cannot be in parallel at all. More realistic is that simply each block or phase of computation must be serial. If anyone cares enough about this, it will have to be implemented in the future.

Note: It doesn't make much sense to combine parallel computation with "when idle" sampling, because "when idle" sampling only produces one sample at a time to process.

Raises `ValueError` – For invalid *sampling* or *processing* kwargs

`get_ctrl_signal`(*query_time_ms*)

Get per-stimulator control signal from the *IOProcessor*.

Parameters `time` (*Brian 2 temporal Unit*) – Current timestep

Returns A {'stimulator_name': value} dictionary for updating stimulators.

Return type dict

`is_sampling_now`(*query_time_ms*)

Determines whether the processor will take a sample at this timestep.

Parameters `time` (*Brian 2 temporal Unit*) – Current timestep.

Return type bool

`abstract process`(*state_dict: dict*, *sample_time_ms: float*) → Tuple[dict, float]

Process network state to generate output to update stimulators.

This is the function the user must implement to define the signal processing pipeline.

Parameters

- `state_dict` (*dict*) – {*recorder_name*: *state*} dictionary from *get_state()*
- `time_ms` (*float*) –

Returns {'stim_name': *ctrl_signal*} dictionary and output time in milliseconds.

Return type Tuple[dict, float]

`put_state`(*state_dict: dict*, *sample_time_ms*)

Deliver network state to the `IOProcessor`.

Parameters

- `state_dict` (*dict*) – A dictionary of recorder measurements, as returned by *get_state()*
- `time` (*brian2 temporal Unit*) – The current simulation timestep. Essential for simulating control latency and for time-varying control.

t_samp_ms: list[float]

Record of sampling times—each time `put_state()` is called.

class `cleo.ioproc.`**PIController**(*ref_signal: callable*, *Kp: float*, *Ki: float = 0*, *sample_period_ms: float = 0*, ***kwargs: Any*)

Bases: `cleo.ioproc.base.ProcessingBlock`

Simple PI controller.

`compute_output()` requires a `sample_time_ms` keyword argument. Only tested on controlling scalar values, but could be easily adapted to controlling a multi-dimensional state.

Parameters

- **ref_signal** (*callable*) – Must return the target as a function of time in ms
- **Kp** (*float*) – Gain on the proportional error
- **Ki** (*float, optional*) – Gain on the integral error, by default 0
- **sample_period_ms** (*float, optional*) – Rate at which processor takes samples, by default 0. Only used to compute integrated error on first sample

compute_output(*input: float*, ***kwargs*) → float

Compute control input to the system using previously specified gains.

Parameters **input** (*Any*) – Current system state

Returns Control signal

Return type float

ref_signal: callable[[float], Any]

Callable returning the target as a function of time in ms

class `cleo.ioproc.`**ProcessingBlock**(***kwargs*)

Bases: `abc.ABC`

Abstract signal processing stage or control block.

It's important to use *super().__init__(**kwargs)* in the base class to use the parent-class logic here.

Keyword Arguments **delay** (`Delay`) – Delay object which adds to the compute time

Raises **TypeError** – When *delay* is not a *Delay* object.

abstract compute_output(*input: Any*, ***kwargs*) → Any

Computes output for given input.

This is where the user will implement the desired functionality of the *ProcessingBlock* without regard for latency.

Parameters

- **input** (*Any*) – Data to be processed. Passed in from `process()`.
- ****kwargs** (*Any*) – optional key-value argument pairs passed from `process()`. Could be used to pass in such values as the IO processor's walltime or the measurement time for time- dependent functions.

Returns output

Return type Any

delay: *cleo.ioproc.delays.Delay*

The delay object determining compute latency for the block

process(*input: Any*, *t_in_ms: float*, ***kwargs*) → Tuple[Any, float]

Computes and saves output and output time given input and input time.

The user should implement *compute_output()* for their child classes, which performs the computation itself without regards for timing or saving variables.

Parameters

- **input** (*Any*) – Data to be processed
- **t_in_ms** (*float*) – Time the block receives the input data
- ****kwargs** (*Any*) – Key-value list of arguments passed to *compute_output()*

Returns output, out time in milliseconds

Return type Tuple[Any, float]

save_history: bool

Whether to record *t_in_ms*, *t_out_ms*, and *values* with every timestep

t_in_ms: list[float]

The walltime the block received each input. Only recorded if *save_history*

t_out_ms: list[float]

The walltime of each of the block's outputs. Only recorded if *save_history*

values: list[Any]

Each of the block's outputs. Only recorded if *save_history*

class cleo.ioproc.**RecordOnlyProcessor**(*sample_period_ms*, ***kwargs*)

Bases: *cleo.ioproc.base.LatencyIOProcessor*

Take samples without performing any control.

Use this if all you are doing is recording.

Parameters **sample_period_ms** (*float*) – Determines how frequently samples are taken from the network.

Keyword Arguments

- **sampling** (*str*) – "fixed" or "when idle"; "fixed" by default

  "fixed" sampling means samples are taken on a fixed schedule, with no exceptions.

  "when idle" sampling means no samples are taken before the previous sample's output has been delivered. A sample is taken ASAP after an over-period computation: otherwise remains on schedule.

- **processing** (*str*) – "parallel" or "serial"; "parallel" by default

  "parallel" computes the output time by adding the delay for a sample onto the sample time, so if the delay is 2 ms, for example, while the sample period is only 1 ms, some of the processing is happening in parallel. Output order matches input order even if the computed output time for a sample is sooner than that for a previous sample.

  "serial" computes the output time by adding the delay for a sample onto the output time of the previous sample, rather than the sampling time. Note this may be of limited utility because it essentially means the *entire* round trip cannot be in parallel at all. More realistic is that simply each block or phase of computation must be serial. If anyone cares enough about this, it will have to be implemented in the future.

Note: It doesn't make much sense to combine parallel computation with "when idle" sampling, because "when idle" sampling only produces one sample at a time to process.

Raises `ValueError` – For invalid *sampling* or *processing* kwargs

`process`(*state_dict: dict*, *sample_time_ms: float*) → Tuple[dict, float]

Process network state to generate output to update stimulators.

This is the function the user must implement to define the signal processing pipeline.

Parameters

- **state_dict** (*dict*) – {*recorder_name*: *state*} dictionary from `get_state()`
- **time_ms** (*float*) –

Returns {'stim_name': *ctrl_signal*} dictionary and output time in milliseconds.

Return type Tuple[dict, float]

`sample_period_ms: float`

Determines how frequently the processor takes samples

`t_samp_ms: list[float]`

Record of sampling times—each time `put_state()` is called.

6.3.6 cleo.viz module

Tools for visualizing models and simulations

`class cleo.viz.VideoVisualizer`(*devices: collections.abc.Iterable[Union[cleo.base.InterfaceDevice, Tuple[cleo.base.InterfaceDevice, dict]]] = 'all'*, *dt: brian2.units.fundamentalunits.Quantity = 1. * msecond*)

Bases: `cleo.base.InterfaceDevice`

Device for visualizing a simulation.

Must be injected after all other devices and before the simulation is run.

Parameters

- **devices** (*Iterable[Union[InterfaceDevice, Tuple[InterfaceDevice, dict]]], optional*) – list of devices or (device, vis_kwargs) tuples to include in the plot, just as in the `plot()` function, by default "all", which will include all recorders and stimulators currently injected when this visualizer is injected into the simulator.
- **dt** (*Brian 2 temporal Quantity, optional*) – length of each frame—that is, every *dt* the visualizer takes a snapshot of the network, by default 1*ms

`brian_objects: set`

All the Brian objects added to the network by this device. Must be kept up-to-date in `connect_to_neuron_group()` and other functions so that those objects can be automatically added to the network when the device is injected.

connect_to_neuron_group(*neuron_group: brian2.groups.neurongroup.NeuronGroup*, ***kwparams*) → None

Connect device to given *neuron_group*.

If your device introduces any objects which Brian must keep track of, such as a NeuronGroup, Synapses, or Monitor, make sure to add these to *self.brian_objects*.

Parameters

- **neuron_group** (`NeuronGroup`) –
- ****kwparams** (optional, passed from *inject_recorder* or) – *inject_stimulator*

generate_Animation(*plotargs: dict*, *slowdown_factor: float = 10*, ***figargs: Any*) → matplotlib.animation.Animation

Create a matplotlib Animation object from the recorded simulation

Parameters

- **plotargs** (`dict`) – dictionary of arguments as taken by `plot()`. can include *xlim*, *ylim*, *zlim*, *colors*, *axis_scale_unit*, *invert_z*, and/or *scatterargs*. neuron groups and devices are automatically added and ***figargs* are specified separately.
- **slowdown_factor** (`float, optional`) – how much slower the animation will be rendered, as a multiple of real-time, by default 10
- ****figargs** (`Any, optional`) – keyword arguments passed to plt.figure(), such as figsize

Returns An Animation object capturing the desired visualization. See matplotlib's docs for saving and rendering options.

Return type `matplotlib.animation.Animation`

init_for_simulator(*simulator: cleo.base.CLSimulator*)

Initialize device for simulator on initial injection

This function is called only the first time a device is injected into a simulator and performs any operations that are independent of the individual neuron groups it is connected to.

Parameters **simulator** (`CLSimulator`) – simulator being injected into

name: str

Unique identifier for device. Used as a key in output/input dicts

sim: *`cleo.base.CLSimulator`*

The simulator the device is injected into

`cleo.viz.`**plot**(**neuron_groups: brian2.groups.neurongroup.NeuronGroup*, *xlim: Optional[Tuple[float, float]] = None*, *ylim: Optional[Tuple[float, float]] = None*, *zlim: Optional[Tuple[float, float]] = None*, *colors: Optional[collections.abc.Iterable] = None*, *axis_scale_unit: brian2.units.fundamentalunits.Unit = mmetre*, *devices: collections.abc.Iterable[Union[cleo.base.InterfaceDevice, Tuple[cleo.base.InterfaceDevice, dict]]] = []*, *invert_z: bool = True*, *scatterargs: dict = {}*, ***figargs: Any*) → None

Visualize neurons and interface devices

Parameters

- **xlim** (`Tuple[float, float], optional`) – xlim for plot, determined automatically by default
- **ylim** (`Tuple[float, float], optional`) – ylim for plot, determined automatically by default

- **zlim** (*Tuple[float, float], optional*) – zlim for plot, determined automatically by default
- **colors** (*Iterable, optional*) – colors, one for each neuron group, automatically determined by default
- **axis_scale_unit** (*Unit, optional*) – Brian unit to scale lim params, by default mm
- **devices** (*Iterable[Union[InterfaceDevice, Tuple[InterfaceDevice, dict]]], optional*) – devices to add to the plot or (device, kwargs) tuples. *add_self_to_plot()* is called for each, using the kwargs dict if given. By default []
- **invert_z** (*bool, optional*) – whether to invert z-axis, by default True to reflect the convention that +z represents depth from cortex surface
- **scatterargs** (*dict, optional*) – arguments passed to plt.scatter() for each neuron group, such as marker
- ****figargs** (*Any, optional*) – keyword arguments passed to plt.figure(), such as figsize

Raises **ValueError** – When neuron group doesn't have x, y, and z already defined

6.3.7 cleo.recorders module

Contains basic recorders.

class cleo.recorders.**GroundTruthSpikeRecorder**(*name*)

Bases: *cleo.base.Recorder*

Reports the number of spikes seen since last queried for each neuron.

This amounts effectively to the number of spikes per control period. Note: this will only work for one neuron group at the moment.

Parameters **name** (*str*) – Unique identifier for the device.

brian_objects: set

All the Brian objects added to the network by this device. Must be kept up-to-date in *connect_to_neuron_group()* and other functions so that those objects can be automatically added to the network when the device is injected.

connect_to_neuron_group(*neuron_group*)

Connect device to given *neuron_group*.

If your device introduces any objects which Brian must keep track of, such as a NeuronGroup, Synapses, or Monitor, make sure to add these to *self.brian_objects*.

Parameters

- **neuron_group** (*NeuronGroup*) –
- ****kwparams** (optional, passed from *inject_recorder* or) – *inject_stimulator*

get_state() → nptyping.types._ndarray.NDArray[Any, nptyping.types._number.UInt]

Returns n_neurons-length array with spike counts over the latest control period.

Return type NDArray[(n_neurons,), np.uint]

name: str

Unique identifier for device. Used as a key in output/input dicts

sim: *CLSimulator*

The simulator the device is injected into

class cleo.recorders.RateRecorder(*name: str*, *index: int*)

Bases: *cleo.base.Recorder*

Records firing rate from a single neuron.

Firing rate comes from Brian's `PopulationRateMonitor`

Parameters

- **name** (*str*) – Unique device name
- **index** (*int*) – index of neuron to record

brian_objects: set

All the Brian objects added to the network by this device. Must be kept up-to-date in *connect_to_neuron_group()* and other functions so that those objects can be automatically added to the network when the device is injected.

connect_to_neuron_group(*neuron_group*)

Connect device to given *neuron_group*.

If your device introduces any objects which Brian must keep track of, such as a NeuronGroup, Synapses, or Monitor, make sure to add these to *self.brian_objects*.

Parameters

- **neuron_group** (*NeuronGroup*) –
- ****kwparams** (optional, passed from *inject_recorder* or) – *inject_stimulator*

get_state()

Return current measurement.

name: str

Unique identifier for device. Used as a key in output/input dicts

sim: *CLSimulator*

The simulator the device is injected into

class cleo.recorders.VoltageRecorder(*name: str*, *voltage_var_name: str = 'v'*)

Bases: *cleo.base.Recorder*

Records the voltage of a single neuron group.

Parameters

- **name** (*str*) – Unique device name
- **voltage_var_name** (*str, optional*) – Name of variable representing membrane voltage, by default "v"

brian_objects: set

All the Brian objects added to the network by this device. Must be kept up-to-date in *connect_to_neuron_group()* and other functions so that those objects can be automatically added to the network when the device is injected.

connect_to_neuron_group(*neuron_group*)

Connect device to given *neuron_group*.

If your device introduces any objects which Brian must keep track of, such as a NeuronGroup, Synapses, or Monitor, make sure to add these to *self.brian_objects*.

Parameters

- **neuron_group** (*NeuronGroup*) –
- ****kwparams** (optional, passed from *inject_recorder* or) – *inject_stimulator*

get_state() → brian2.units.fundamentalunits.Quantity

Returns Current voltage of target neuron group

Return type Quantity

name: str

Unique identifier for device. Used as a key in output/input dicts

sim: *CLSimulator*

The simulator the device is injected into

6.3.8 cleo.stimulators module

Contains basic stimulators.

class cleo.stimulators.**StateVariableSetter**(*name: str*, *variable_to_ctrl: str*, *unit: brian2.units.fundamentalunits.Unit*, *start_value: float = 0*)

Bases: *cleo.base.Stimulator*

Sets the given state variable of target neuron groups.

Parameters

- **name** (*str*) – Unique device name
- **variable_to_ctrl** (*str*) – Name of state variable to control
- **unit** (*Unit*) – Unit of that state variable: will be used in *update()*
- **start_value** (*float, optional*) – Starting variable value, by default 0

brian_objects: set

All the Brian objects added to the network by this device. Must be kept up-to-date in *connect_to_neuron_group()* and other functions so that those objects can be automatically added to the network when the device is injected.

connect_to_neuron_group(*neuron_group*)

Connect device to given *neuron_group*.

If your device introduces any objects which Brian must keep track of, such as a NeuronGroup, Synapses, or Monitor, make sure to add these to *self.brian_objects*.

Parameters

- **neuron_group** (*NeuronGroup*) –
- ****kwparams** (optional, passed from *inject_recorder* or) – *inject_stimulator*

default_value: Any

The default value of the device—used on initialization and on `reset()`

name: str

Unique identifier for device. Used as a key in output/input dicts

save_history: bool

Determines whether *t_ms* and *values* are recorded

sim: *CLSimulator*

The simulator the device is injected into

t_ms: list[float]

Times stimulator was updated, stored if *save_history*

update(*ctrl_signal: float*) → None

Set state variable of target neuron groups

Parameters **ctrl_signal** (*float*) – Value to update variable to, without unit. The unit provided on initialization is automatically multiplied.

value: Any

The current value of the stimulator device

values: list[Any]

Values taken by the stimulator at each *update()* call, stored if *save_history*

6.3.9 cleo.utilities module

Assorted utilities for developers.

cleo.utilities.**get_orth_vectors_for_v**(*v*)

Returns w1, w2 as 1x3 row vectors

cleo.utilities.**modify_model_with_eqs**(*neuron_group*, *eqs_to_add*)

Adapted from _create_variables() from neurongroup.py from Brian2 source code v2.3.0.2

cleo.utilities.**style_plots_for_docs**()

cleo.utilities.**uniform_cylinder_rz**(*n*, *rmax*, *zmax*)

cleo.utilities.**wavelength_to_rgb**(*wavelength_nm*, *gamma=0.8*)

taken from http://www.noah.org/wiki/Wavelength_to_RGB_in_Python This converts a given wavelength of light to an approximate RGB color value. The wavelength must be given in nanometers in the range from 380 nm through 750 nm (789 THz through 400 THz).

Based on code by Dan Bruton http://www.physics.sfasu.edu/astro/color/spectra.html

cleo.utilities.**xyz_from_rz**(*rs*, *thetas*, *zs*, *xyz_start*, *xyz_end*)

Convert from cylindrical to Cartesian coordinates.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

D

F

G

H

I

L

M

T

U

V

W

X

Y

Z