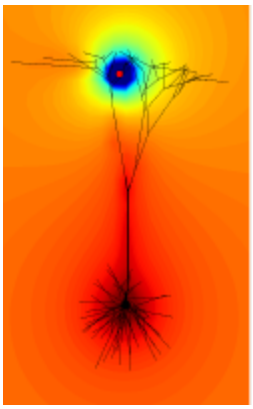
LFPy Homepage

Release 2.2.6

LFPy-team

Apr 05, 2022

CONTENTS



LFPy

Local Field Potentials in Python

(Looking for the old LFPy v1.* documentation? Follow link)

CHAPTER

ONE

CONTENTS

1.1 LFPy

1.1.1 Summary

LFPy is a Python module for calculation of extracellular potentials from multicompartment neuron models. It relies on the NEURON simulator (http://www.neuron.yale.edu/neuron) and uses the Python interface (http://www.frontiersin.org/neuroinformatics/10.3389/neuro.11.001.2009/abstract) it provides.

1.1.2 Latest changes

Just updated LFPy? Please check the latest release notes: https://github.com/LFPy/LFPy/releases

1.1.3 Usage

A basic simulation of extracellular potentials of a multicompartment neuron model set up with LFPy:

```
>>> # import modules
>>> import LFPy
>>> from LFPy import Cell, Synapse, LineSourcePotential
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> # create Cell
>>> cell = Cell(morphology=''.join(LFPy.__path__ +
>>>                                ['/test/ball_and_sticks.hoc']),
>>>             passive=True,  # NEURON 'pas' mechanism
>>>             tstop=100,  # ms
>>>            )
>>> # create Synapse
>>> synapse = Synapse(cell=cell,
>>>                   idx=cell.get_idx("soma[0]"),  # soma segment index
>>>                   syntype='Exp2Syn',  # two-exponential synapse
>>>                   weight=0.005,  # max conductance (uS)
>>>                   e=0,  # reversal potential (mV)
>>>                   tau1=0.5,  # rise time constant
>>>                   tau2=5.,  # decay time constant
>>>                   record_current=True,  # record synapse current
>>>                  )
```

(continues on next page)

(continued from previous page)

```
>>> synapse.set_spike_times(np.array([20., 40]))  # set activation times
>>> # create extracellular predictor
>>> lsp = LineSourcePotential(cell=cell,
>>>                           x=np.zeros(11) + 10,  # x-coordinates of contacts (µm)
>>>                           y=np.zeros(11),  # y-coordinates
>>>                           z=np.arange(11)*20,  # z-coordinates
>>>                           sigma=0.3,  # extracellular conductivity (S/m)
>>>                          )
>>> # execute simulation
>>> cell.simulate(probes=[lsp])  # compute measurements at run time
>>> # plot results
>>> fig, axes = plt.subplots(3, 1, sharex=True, figsize=(12, 8))
>>> axes[0].plot(cell.tvec, synapse.i)
>>> axes[0].set_ylabel('i_syn (nA)')
>>> axes[1].plot(cell.tvec, cell.somav)
>>> axes[1].set_ylabel('V_soma (nA)')
>>> axes[2].pcolormesh(cell.tvec, lsp.z, lsp.data, shading='auto')
>>> axes[2].set_ylabel('z (µm)')
>>> axes[2].set_xlabel('t (ms)')
```

You can now test some LFPy examples online without installation:

1.1.4 Code status

1.1.5 Conda-forge status

1.1.6 Information

LFPy provides a set of easy-to-use Python classes for setting up your model, running your simulations and calculating the extracellular potentials arising from activity in your model neuron. If you have a model working in NEURON (www.neuron.yale.edu) already, it is likely that it can be adapted to work with LFPy.

The extracellular potentials are calculated from transmembrane currents in multi-compartment neuron models using the line-source method (Holt & Koch, J Comp Neurosci 1999), but a simpler point-source method is also available. The calculations assume that the neuron are surrounded by an infinite extracellular medium with homogeneous and frequency independent conductivity, and compartments are assumed to be at least at a minimal distance from the electrode (which can be specified by the user). For more information on the biophysics underlying the numerical framework used see this coming book chapter:

- K.H. Pettersen, H. Linden, A.M. Dale and G.T. Einevoll: Extracellular spikes and current-source density, in *Handbook of Neural Activity Measurement*, edited by R. Brette and A. Destexhe, Cambridge, to appear (preprint PDF, 5.7MB http://www.csc.kth.se/~helinden/PettersenLindenDaleEinevoll-BookChapter-revised.pdf

The first release of LFPy (v1.x) was mainly designed for simulation extracellular potentials of single neurons, described in our paper on the package in Frontiers in Neuroinformatics entitled "LFPy: A tool for biophysical simulation of extracellular potentials generated by detailed model neurons". The article can be found at https://dx.doi.org/10.3389/fninf.2013.00041. Since version 2 (LFPy v2.x), the tool also facilitates simulations of extracellular potentials and current dipole moment from ongoing activity in recurrently connected networks of multicompartment neurons, prediction of EEG scalp surface potentials, MEG scalp surface magnetic fields, as described in the publication "Multimodal modeling of neural network activity: computing LFP, ECoG, EEG and MEG signals with LFPy2.0" by Espen Hagen, Solveig Naess, Torbjoern V Ness, Gaute T Einevoll, found at https://dx.doi.org/10.3389/fninf.2018.00092.

1.1.7 Citing LFPy

- LFPy v2.x: Hagen E, Næss S, Ness TV and Einevoll GT (2018) Multimodal Modeling of Neural Network Activity: Computing LFP, ECoG, EEG, and MEG Signals With LFPy 2.0. Front. Neuroinform. 12:92. doi: 10.3389/fninf.2018.00092. https://dx.doi.org/10.3389/fninf.2018.00092
- LFPy v1.x: Linden H, Hagen E, Leski S, Norheim ES, Pettersen KH and Einevoll GT (2013). LFPy: A tool for biophysical simulation of extracellular potentials generated by detailed model neurons. Front. Neuroinform. 7:41. doi: 10.3389/fninf.2013.00041. https://dx.doi.org/10.3389/fninf.2013.00041

LFPy was developed in the Computational Neuroscience Group, Department of Mathemathical Sciences and Technology (http://www.nmbu.no/imt), at the Norwegian University of Life Sciences (http://www.nmbu.no), in collaboration with the Laboratory of Neuroinformatics (http://www.nencki.gov.pl/en/laboratory-of-neuroinformatics), Nencki Institute of Experimental Biology (http://www.nencki.gov.pl), Warsaw, Poland. The effort was supported by International Neuroinformatics Coordinating Facility (http://incf.org), the Research Council of Norway (http://www.forskningsradet.no/english) (eScience, NevroNor), EU-FP7 (BrainScaleS, http://www.brainscales.org), the European Union Horizon 2020 Framework Programme for Research and Innovation under Specific Grant Agreement No. 785907 and No. 945539 [Human Brain Project (HBP) SGA2, SGA3 and EBRAINS].

For updated information on LFPy and online documentation, see the LFPy homepage (http://lfpy.readthedocs.io).

1.1.8 Tutorial slides on LFPy

- Slides for OCNS 2019 meeting tutorial T8: Biophysical modeling of extracellular potentials (using LFPy) hosted in Barcelona, Spain on LFPy:
- Older tutorial slides can be found at https://github.com/LFPy/LFPy.github.io/tree/master/downloads

1.1.9 Related projects

LFPy has been used extensively in ongoing and published work, and may be a required dependency by the publicly available Python modules:

- ViSAPy - Virtual Spiking Activity in Python (https://github.com/espenhgn/ViSAPy, http://software.incf.org/software/visapy), as described in Hagen, E., et al. (2015), J Neurosci Meth, DOI:10.1016/j.jneumeth.2015.01.029
- ViMEAPy that can be used to incorporate heterogeneous conductivity in calculations of extracellular potentials with LFPy (https://bitbucket.org/torbness/vimeapy, http://software.incf.org/software/vimeapy). ViMEAPy and it's application is described in Ness, T. V., et al. (2015), Neuroinform, DOI:10.1007/s12021-015-9265-6.
- hybridLFPy - biophysics-based hybrid scheme for calculating the local field potential (LFP) of spiking activity in simplified point-neuron network models (https://github.com/INM-6/hybridLFPy), as described in Hagen, E. and Dahmen, D., et al. (2016), Cereb Cortex, DOI:10.1093/cercor/bhw237
- MEArec - Fast and customizable simulation of extracellular recordings on Multi-Electrode-Arrays (https://github.com/alejoe91/MEArec) as described in Buccino, A.P., Einevoll, G.T. MEArec: A Fast and Customizable Testbench Simulator for Ground-truth Extracellular Spiking Activity. Neuroinform (2020). https://doi.org/10.1007/s12021-020-09467-7

1.1.10 Requirements

Dependencies should normally be automatically installed. For manual preinstallation of dependencies, the following packages are needed:

- Python modules numpy, scipy, matplotlib, h5py, mpi4py
- MEAutility (https://github.com/alejoe91/MEAutility)
- LFPykit (https://github.com/LFPy/LFPykit)
- NEURON (from http://www.neuron.yale.edu) and corresponding Python module. The following should execute without error in a Python console:

```
>>> import neuron
>>> neuron.test()
```

- Cython (C-extensions for python, http://cython.org) to speed up simulations of extracellular fields

1.1.11 Installation

There are few options to install LFPy:

1. From the Python Package Index with only local access using pip:

```
$ pip install LFPy --user
```

   as sudoer (in general not recommended as system Python files may be overwritten):

```
$ sudo pip install LFPy
```

   Upgrading LFPy from the Python package index (without attempts at upgrading dependencies):

```
$ pip install --upgrade --no-deps LFPy --user
```

   LFPy release candidates can be installed as:

```
$ pip install --pre --index-url https://test.pypi.org/simple/ --extra-index-url␣
↪https://pypi.org/simple LFPy --user
```

2. From the Python Package Index with only local access using easy_install:

```
$ easy_install --user LFPy
```

   As sudoer:

```
$ sudo easy_install LFPy
```

3. From source:

```
$ tar -xzf LFPy-x.x.tar.gz
$ cd LFPy-x.x
$ (sudo) python setup.py develop (--user)
```

4. Development version from the GitHub repository:

```
$ git clone https://github.com/LFPy/LFPy.git
$ cd LFPy
$ (sudo) pip install -r requirements.txt (--user) # install dependencies
$ (sudo) python setup.py develop (--user)
```

5. Anaconda Python (https://www.anaconda.com, macos/linux only):

   Add the conda-forge (https://conda-forge.org) as channel:

```
$ conda config --add channels conda-forge
$ conda config --set channel_priority strict  # suggested
```

   Create a new conda environment with LFPy and activate it:

```
$ conda create -n lfpy python=3 pip lfpy  # creates new Python 3.x conda␣
↪environment named lfpy with pip and LFPy and their dependencies
$ conda activate lfpy  # activate the lfpy environment
$ python -c "import LFPy; LFPy.run_tests()"  # check that installation is working
```

   LFPy can also be installed in existing conda environments if the dependency tree is solvable:

```
$ conda activate <environment>
$ conda install lfpy  # installs LFPy and its dependencies in the current conda␣
↪environment
```

1.1.12 Uninstall

To remove installed LFPy files it should suffice to issue (repeat until no more LFPy files are found):

```
$ (sudo) pip uninstall LFPy
```

In case LFPy was installed using conda in an environment, it can be uninstalled by issuing:

```
$ conda uninstall lfpy
```

1.1.13 Docker

We provide a Docker (https://www.docker.com) container recipe file with LFPy. To get started, install Docker and issue either:

```
# build Dockerfile from GitHub
$ docker build -t lfpy https://raw.githubusercontent.com/LFPy/LFPy/master/Dockerfile
$ docker run -it -p 5000:5000 lfpy:latest
```

or

```
# build local Dockerfile (obtained by cloning repo, checkout branch etc.)
$ docker build -t lfpy - < Dockerfile
$ docker run -it -p 5000:5000 lfpy:latest
```

If the docker file should fail for some reason it is possible to store the build log and avoid build caches by issuing

```
$ docker build --no-cache --progress=plain -t lfpy - < Dockerfile 2>&1 | tee lfpy.log
```

If the build is successful, the `--mount` option can be used to mount a folder on the host to a target folder as:

```
$ docker run --mount type=bind,source="$(pwd)",target=/opt -it -p 5000:5000 lfpy
```

which mounts the present working dirctory (`$(pwd)`) to the `/opt` directory of the container. Try mounting the `LFPy` source directory for example (by setting `source="<path-to-LFPy>"`). Various LFPy example files can then be found in the folder `/opt/LFPy/examples/` when the container is running.

Jupyter notebook servers running from within the container can be accessed after invoking them by issuing:

```
$ cd /opt/LFPy/examples/
$ jupyter-notebook --ip 0.0.0.0 --port=5000 --no-browser --allow-root
```

and opening the resulting URL in a browser on the host computer, similar to: http://127.0.0.1:5000/?token=dcf8f859f859740fc858c568bdd5b015e0cf15bfc2c5b0c1

1.1.14 HTML Documentation

To generate the html documentation also hosted at https://lfpy.rtfd.io using Sphinx, issue from the LFPy source code directory:

```
$ cd doc
$ make html
```

The main html file is in `_build/html/index.html`. m2r2, Numpydoc and the Sphinx ReadTheDocs theme may be needed:

```
$ pip install m2r2 --user
$ pip install numpydoc --user
$ pip install sphinx-rtd-theme --user
```

1.1.15 Physical units in LFPy

Physical units follow the NEURON conventions found here. The units in LFPy for given quantities are:

Quantity	Symbol	Unit
Spatial dimensions	x,y,z,d	[m]
Potential	v, Phi,	[mV]
Reversal potential	E	[mV]
Current	i	[nA]
Membrane capacitance	c_m	[F/cm2]
Conductance	g	[S/cm2]
Synaptic conductance	g	[µS]
Extracellular conductivity	sigma,	[S/m]
Current dipole moment	P	[nA µm]
Magnetic field	H	[nA/µm]
Magnetic permeability	µ, mu	[T m/A]
Current Source Density	CSD	[nA/µm3]

Note: resistance, conductance and capacitance are usually specific values, i.e per membrane area (lowercase r_m, g, c_m) Depending on the mechanism files, some may use different units altogether, but this should be taken care of internally by NEURON.

1.2 Module `LFPy`

Initialization of LFPy, a Python module for simulating extracellular potentials.

Group of Computational Neuroscience, Department of Mathematical Sciences and Technology, Norwegian University of Life Sciences.

Classes

- `Cell` - object built on top of NEURON representing biological neuron
- `TemplateCell` - Similar to `Cell`, but for models using cell templates
- `NetworkCell` - Similar to `TemplateCell` with some attributes and methods for spike communication between parallel RANKs
- `PointProcess` - Parent class of `Synapse` and `StimIntElectrode`
- `Synapse` - Convenience class for inserting synapses onto `Cell` objects
- `StimIntElectrode` - Convenience class for inserting stimulating electrodes into `Cell` objects
- `Network` - Class for creating distributed populations of cells and handling connections between cells in populations
- `NetworkPopulation` - Class representing group of `Cell` objects distributed across MPI RANKs
- `RecExtElectrode` - Class for setup of simulations of extracellular potentials
- `RecMEAElectrode` - Class for setup of simulations of in vitro (slice) extracellular potentials
- `PointSourcePotential` - Base forward-model for extracellular potentials assuming point current sources in conductive media
- `LineSourcePotential` - Base forward-model for extracellular potentials assuming line current sources in conductive media
- `OneSphereVolumeConductor` - For computing extracellular potentials within and outside a homogeneous sphere
- `CurrentDipoleMoment` - For computing the current dipole moment,
- `FourSphereVolumeConductor` - For computing extracellular potentials in four-sphere head model (brain, CSF, skull, scalp)
- `InfiniteVolumeConductor` - To compute extracellular potentials with current dipoles in infinite volume conductor
- `MEG` - Class for computing magnetic field from current dipole moment

Modules

- `lfpcalc` - Misc. functions used by RecExtElectrode class
- `tools` - Some convenient functions
- `inputgenerators` - Functions for synaptic input time generation
- `eegmegcalc` - Classes for calculating current dipole moment vector P and P_tot from currents and distances.
- `run_simulations` - Functions to run NEURON simulations

1.3 Cell classes

1.3.1 class Cell

class LFPy.Cell(*morphology*, *v_init=- 70.0*, *Ra=None*, *cm=None*, *passive=False*, *passive_parameters=None*, *extracellular=False*, *tstart=0.0*, *tstop=100.0*, *dt=0.0625*, *nsegs_method='lambda100'*, *lambda_f=100.0*, *d_lambda=0.1*, *max_nsegs_length=None*, *delete_sections=True*, *custom_code=None*, *custom_fun=None*, *custom_fun_args=None*, *pt3d=False*, *celsius=None*, *verbose=False*, ***kwargs*)

Bases: object

The main cell class used in LFPy.

Parameters

morphology: str or neuron.h.SectionList File path of morphology on format that NEURON can understand (w. file ending .hoc, .asc, .swc or .xml), or neuron.h.SectionList instance filled with references to neuron.h.Section instances.

v_init: float Initial membrane potential. Defaults to -70 mV.

Ra: float or None Axial resistance. Defaults to None (unit Ohm*cm)

cm: float Membrane capacitance. Defaults to None (unit uF/cm2)

passive: bool Passive mechanisms are initialized if True. Defaults to False

passive_parameters: dict parameter dictionary with values for the passive membrane mechanism in NEURON ('pas'). The dictionary must contain keys 'g_pas' [S/cm^2] and 'e_pas' [mV], like the default: passive_parameters=dict(g_pas=0.001, e_pas=-70)

extracellular: bool Switch for NEURON's extracellular mechanism. Defaults to False

dt: float simulation timestep. Defaults to 2^-4 ms

tstart: float Initialization time for simulation <= 0 ms. Defaults to 0.

tstop: float Stop time for simulation > 0 ms. Defaults to 100 ms.

nsegs_method: 'lambda100' or 'lambda_f' or 'fixed_length' or None nseg rule, used by NEURON to determine number of compartments. Defaults to 'lambda100'

max_nsegs_length: float or None Maximum segment length for method 'fixed_length'. Defaults to None

lambda_f: float AC frequency for method 'lambda_f'. Defaults to 100. (Hz)

d_lambda: float Parameter for d_lambda rule. Defaults to 0.1

delete_sections: bool Delete pre-existing section-references. Defaults to True

custom_code: list or None List of model-specific code files ([.py/.hoc]). Defaults to None

custom_fun: list or None List of model-specific functions with args. Defaults to None

custom_fun_args: list or None List of args passed to custom_fun functions. Defaults to None

pt3d: bool Use pt3d-info of the cell geometries switch. Defaults to False

celsius: float or None Temperature in celsius. If nothing is specified here or in custom code it is 6.3 celcius

verbose: bool Verbose output switch. Defaults to False

See also:

TemplateCell

NetworkCell

Examples

Simple example of how to use the Cell class with a passive-circuit morphology (modify morphology path accordingly):

```
>>> import os
>>> import LFPy
>>> cellParameters = {
>>>     'morphology': os.path.join('examples', 'morphologies',
>>>                                'L5_Mainen96_LFPy.hoc'),
>>>     'v_init': -65.,
>>>     'cm': 1.0,
>>>     'Ra': 150,
>>>     'passive': True,
>>>     'passive_parameters': {'g_pas': 1./30000, 'e_pas': -65},
>>>     'dt': 2**-3,
>>>     'tstart': 0,
>>>     'tstop': 50,
>>> }
>>> cell = LFPy.Cell(**cellParameters)
>>> cell.simulate()
>>> print(cell.somav)
```

cellpickler(*filename*, *pickler=<built-in function dump>*)

Save data in cell to filename, using cPickle. It will however destroy any `neuron.h` objects upon saving, as c-objects cannot be pickled

Parameters

filename: str Where to save cell

Returns

None or pickle

Examples

```
>>> # To save a cell, issue:
>>> cell.cellpickler('cell.cpickle')
>>> # To load this cell again in another session:
>>> import cPickle
>>> with file('cell.cpickle', 'rb') as f:
>>>     cell = cPickle.load(f)
```

chiral_morphology(*axis='x'*)

Mirror the morphology around given axis, (default x-axis), useful to introduce more heterogeneouties in morphology shapes

Parameters

axis: str 'x' or 'y' or 'z'

`distort_geometry`(*factor=0.0*, *axis='z'*, *nu=0.0*)

Distorts cellular morphology with a relative factor along a chosen axis preserving Poisson's ratio. A ratio nu=0.5 assumes uncompressible and isotropic media that embeds the cell. A ratio nu=0 will only affect geometry along the chosen axis. A ratio nu=-1 will isometrically scale the neuron geometry along each axis. This method does not affect the underlying cable properties of the cell, only predictions of extracellular measurements (by affecting the relative locations of sources representing the compartments).

Parameters

factor: float relative compression/stretching factor of morphology. Default is 0 (no compression/stretching). Positive values implies a compression along the chosen axis.

axis: str which axis to apply compression/stretching. Default is "z".

nu: float Poisson's ratio. Ratio between axial and transversal compression/stretching. Default is 0.

`enable_extracellular_stimulation`(*electrode*, *t_ext=None*, *n=1*, *model='inf'*)

Enable extracellular stimulation with NEURON's *extracellular* mechanism. Extracellular potentials are computed from electrode currents using the point-source approximation. If `model` is `'inf'` (default), potentials are computed as (r_i is the position of a compartment i, r_n is the position of an electrode n, σ is the conductivity of the medium):

$$V_e(r_i) = \sum_n \frac{I_n}{4\pi\sigma|r_i - r_n|}$$

If `model` is `'semi'`, the method of images is used:

$$V_e(r_i) = \sum_n \frac{I_n}{2\pi\sigma|r_i - r_n|}$$

Parameters

electrode: RecExtElectrode Electrode object with stimulating currents

t_ext: np.ndarray or list Time in ms corresponding to step changes in the provided currents. If None, currents are assumed to have the same time steps as the NEURON simulation.

n: int Points per electrode for spatial averaging

model: str `'inf'` or `'semi'`. If `'inf'` the medium is assumed to be infinite and homogeneous. If `'semi'`, the method of images is used.

Returns

v_ext: np.ndarray Computed extracellular potentials at cell mid points

`get_axial_currents_from_vmem`(*timepoints=None*)

Compute axial currents from cell sim: get current magnitude, distance vectors and position vectors.

Parameters

timepoints: ndarray, dtype=int array of timepoints in simulation at which you want to compute the axial currents. Defaults to False. If not given, all simulation timesteps will be included.

Returns

i_axial: ndarray, dtype=float Shape ((cell.totnsegs-1)*2, len(timepoints)) array of axial current magnitudes I in units of (nA) in cell at all timesteps in timepoints, or at all timesteps of the simulation if timepoints=None. Contains two current magnitudes per segment, (except for the root segment): 1) the current from the mid point of the segment to the segment

start point, and 2) the current from the segment start point to the mid point of the parent segment.

d_vectors: ndarray, dtype=float Shape (3, (cell.totnsegs-1)*2) array of distance vectors traveled by each axial current in i_axial in units of (µm). The indices of the first axis, correspond to the first axis of i_axial and pos_vectors.

pos_vectors: ndarray, dtype=float Shape ((cell.totnsegs-1)*2, 3) array of position vectors pointing to the mid point of each axial current in i_axial in units of (µm). The indices of the first axis, correspond to the first axis of i_axial and d_vectors.

Raises

AttributeError Raises an exeption if the cell.vmem attribute cannot be found

`get_axial_resistance()`

Return NEURON axial resistance for all cell compartments.

Returns

ri_list: ndarray, dtype=float Shape (cell.totnsegs,) array containing neuron.h.ri(seg.x) in units of (MOhm) for all segments in cell calculated using the neuron.h.ri(seg.x) method. neuron.h.ri(seg.x) returns the axial resistance from the middle of the segment to the middle of the parent segment. Note: If seg is the first segment in a section, i.e. the parent segment belongs to a different section or there is no parent section, then neuron.h.ri(seg.x) returns the axial resistance from the middle of the segment to the node connecting the segment to the parent section (or a ghost node if there is no parent)

`get_closest_idx(x=0.0, y=0.0, z=0.0, section='allsec')`

Get the index number of a segment in specified section which midpoint is closest to the coordinates defined by the user

Parameters

x: float x-coordinate

y: float y-coordinate

z: float z-coordinate

section: str String matching a section-name. Defaults to 'allsec'.

Returns

int segment index

`get_dict_of_children_idx()`

Return dictionary with children segment indices for all sections.

Returns

children_dict: dictionary Dictionary containing a list for each section, with the segment index of all the section's children. The dictionary is needed to find the sibling of a segment.

`get_dict_parent_connections()`

Return dictionary with parent connection point for all sections.

Returns

connection_dict: dictionary Dictionary containing a float in range [0, 1] for each section in cell. The float gives the location on the parent segment to which the section is connected. The dictionary is needed for computing axial currents.

get_idx(*section='allsec'*, *z_min=- inf*, *z_max=inf*)

Returns compartment idx of segments from sections with names that match the pattern defined in input section on interval [z_min, z_max].

Parameters

section: str Any entry in cell.allsecnames or just 'allsec'.

z_min: float Depth filter. Specify minimum z-position

z_max: float Depth filter. Specify maximum z-position

Returns

ndarray, dtype=int segment indices

Examples

```
>>> idx = cell.get_idx(section='allsec')
>>> print(idx)
>>> idx = cell.get_idx(section=['soma', 'dend', 'apic'])
>>> print(idx)
```

get_idx_children(*parent='soma[0]'*)

Get the idx of parent's children sections, i.e. compartments ids of sections connected to parent-argument

Parameters

parent: str name-pattern matching a sectionname. Defaults to "soma[0]"

Returns

ndarray, dtype=int

get_idx_name(*idx=array([0])*)

Return NEURON convention name of segments with index idx. The returned argument is an array of tuples with corresponding segment idx, section name, and position along the section, like; [(0, 'neuron.h.soma[0]', 0.5),]

Parameters

idx: ndarray, dtype int segment indices, must be between 0 and cell.totnsegs

Returns

ndarray, dtype=object tuples with section names of segments

get_idx_parent_children(*parent='soma[0]'*)

Get all idx of segments of parent and children sections, i.e. segment idx of sections connected to parent-argument, and also of the parent segments

Parameters

parent: str name-pattern matching a sectionname. Defaults to "soma[0]"

Returns

ndarray, dtype=int

get_idx_polygons(*projection=('x', 'z')*)

For each segment idx in cell create a polygon in the plane determined by the projection kwarg (default ('x', 'z')), that can be visualized using plt.fill() or mpl.collections.PolyCollection

Parameters

projection: tuple of strings Determining projection. Defaults to ('x', 'z')

Returns

polygons: list list of (ndarray, ndarray) tuples giving the trajectory of each section

Examples

```
>>> from matplotlib.collections import PolyCollection
>>> import matplotlib.pyplot as plt
>>> cell = LFPy.Cell(morphology='PATH/TO/MORPHOLOGY')
>>> zips = []
>>> for x, z in cell.get_idx_polygons(projection=('x', 'z')):
>>>     zips.append(list(zip(x, z)))
>>> polycol = PolyCollection(zips,
>>>                          edgecolors='none',
>>>                          facecolors='gray')
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.add_collection(polycol)
>>> ax.axis(ax.axis('equal'))
>>> plt.show()
```

get_intersegment_distance(*idx0=0*, *idx1=0*)

Return the Euclidean distance between midpoints of two segments.

Parameters

idx0: int

idx1: int

Returns

float distance (µm).

get_intersegment_vector(*idx0=0*, *idx1=0*)

Return the distance between midpoints of two segments with index idx0 and idx1. The argument returned is a list [x, y, z], where x = self.x[idx1].mean(axis=-1) - self.x[idx0].mean(axis=-1) etc.

Parameters

idx0: int

idx1: int

Returns

list of floats distance between midpoints along x,y,z axis in µm

get_multi_current_dipole_moments(*timepoints=None*)

Return 3D current dipole moment vector and middle position vector from each axial current in space.

Parameters

timepoints: ndarray, dtype=int or None array of timepoints at which you want to compute the current dipole moments. Defaults to None. If not given, all simulation timesteps will be included.

Returns

multi_dipoles: ndarray, dtype = float Shape (n_axial_currents, 3, n_timepoints) array containing the x-,y-,z-components of the current dipole moment from each axial current in cell, at all timepoints. The number of axial currents, n_axial_currents = (cell.totnsegs-1) * 2 and the number of timepoints, n_timepoints = cell.tvec.size. The current dipole moments are given in units of (nA µm).

pos_axial: ndarray, dtype = float Shape (n_axial_currents, 3) array containing the x-, y-, and z-components giving the mid position in space of each multi_dipole in units of (µm).

Examples

Get all current dipole moments and positions from all axial currents in a single neuron simulation:

```
>>> import LFPy
>>> import numpy as np
>>> cell = LFPy.Cell('PATH/TO/MORPHOLOGY', extracellular=False)
>>> syn = LFPy.Synapse(cell, idx=cell.get_closest_idx(0,0,1000),
>>>                    syntype='ExpSyn', e=0., tau=1., weight=0.001)
>>> syn.set_spike_times(np.mgrid[20:100:20])
>>> cell.simulate(rec_vmem=True, rec_imem=False)
>>> timepoints = np.array([1,2,3,4])
>>> multi_dipoles, dipole_locs = cell.get_multi_current_dipole_moments(
>>>     timepoints=timepoints)
```

get_pt3d_polygons(*projection=('x', 'z')*)

For each section create a polygon in the plane determined by keyword argument projection=('x', 'z'), that can be visualized using e.g., plt.fill()

Parameters

projection: tuple of strings Determining projection. Defaults to ('x', 'z')

Returns

list list of (x, z) tuples giving the trajectory of each section that can be plotted using PolyCollection

Examples

```
>>> from matplotlib.collections import PolyCollection
>>> import matplotlib.pyplot as plt
>>> cell = LFPy.Cell(morphology='PATH/TO/MORPHOLOGY')
>>> zips = []
>>> for x, z in cell.get_pt3d_polygons(projection=('x', 'z')):
>>>     zips.append(list(zip(x, z)))
>>> polycol = PolyCollection(zips,
>>>                          edgecolors='none',
>>>                          facecolors='gray')
>>> fig = plt.figure()
```

(continues on next page)

(continued from previous page)

```
>>> ax = fig.add_subplot(111)
>>> ax.add_collection(polycol)
>>> ax.axis(ax.axis('equal'))
>>> plt.show()
```

get_rand_idx_area_and_distribution_norm(*section='allsec'*, *nidx=1*, *z_min=-1000000.0*, *z_max=1000000.0*, *fun=<scipy.stats._continuous_distns.norm_gen object>*, *funargs={'loc': 0, 'scale': 100}*, *funweights=None*)

Return nidx segment indices in section with random probability normalized to the membrane area of each segment multiplied by the value of the probability density function of "fun", a function in the scipy.stats module with corresponding function arguments in "funargs" on the interval [z_min, z_max]

Parameters

section: str string matching a section name

nidx: int number of random indices

z_min: float lower depth interval

z_max: float upper depth interval

fun: function or str, or iterable of function or str if function a scipy.stats method, if str, must be method in scipy.stats module with the same name (like 'norm'), if iterable (list, tuple, numpy.array) of function or str some probability distribution in scipy.stats module

funargs: dict or iterable iterable (list, tuple, numpy.array) of dict, arguments to fun.pdf method (e.g., w. keys 'loc' and 'scale')

funweights: None or iterable iterable (list, tuple, numpy.array) of floats, scaling of each individual fun (i.e., introduces layer specificity)

Examples

```
>>> import LFPy
>>> import numpy as np
>>> import scipy.stats as ss
>>> import matplotlib.pyplot as plt
>>> from os.path import join
>>> cell = LFPy.Cell(morphology=join('cells', 'cells', 'j4a.hoc'))
>>> cell.set_rotation(x=4.99, y=-4.33, z=3.14)
>>> idx = cell.get_rand_idx_area_and_distribution_norm(
    nidx=10000, fun=ss.norm, funargs=dict(loc=0, scale=200))
>>> bins = np.arange(-30, 120)*10
>>> plt.hist(cell.zmid[idx], bins=bins, alpha=0.5)
>>> plt.show()
```

get_rand_idx_area_norm(*section='allsec'*, *nidx=1*, *z_min=- 1000000.0*, *z_max=1000000.0*)

Return nidx segment indices in section with random probability normalized to the membrane area of segment on interval [z_min, z_max]

Parameters

section: str String matching a section-name

nidx: int Number of random indices

z_min: float Depth filter

z_max: float Depth filter

Returns

ndarray, dtype=int segment indices

`get_rand_prob_area_norm`(*section='allsec'*, *z_min=- 10000*, *z_max=10000*)

Return the probability (0-1) for synaptic coupling on segments in section sum(prob)=1 over all segments in section. Probability normalized by area.

Parameters

section: str string matching a section-name. Defaults to 'allsec'

z_min: float depth filter

z_max: float depth filter

Returns

ndarray, dtype=float

`get_rand_prob_area_norm_from_idx`(*idx=array([0])*)

Return the normalized probability (0-1) for synaptic coupling on segments in idx-array. Normalised probability determined by area of segments.

Parameters

idx: ndarray, dtype=int. array of segment indices

Returns

ndarray, dtype=float

`insert_v_ext`(*v_ext*, *t_ext*)

Set external extracellular potential around cell. Playback of some extracellular potential v_ext on each cell.totnseg compartments. Assumes that the "extracellular"-mechanism is inserted on each compartment. Can be used to study ephaptic effects and similar The inputs will be copied and attached to the cell object as cell.v_ext, cell.t_ext, and converted to (list of) neuron.h.Vector types, to allow playback into each compartment e_extracellular reference. Can not be deleted prior to running cell.simulate()

Parameters

v_ext: ndarray Numpy array of size cell.totnsegs x t_ext.size, unit mV

t_ext: ndarray Time vector of v_ext in ms

Examples

```
>>> import LFPy
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> #create cell
>>> cell = LFPy.Cell(morphology='morphologies/example_morphology.hoc',
>>>                  passive=True)
>>> #time vector and extracellular field for every segment:
>>> t_ext = np.arange(cell.tstop / cell.dt+ 1) * cell.dt
>>> v_ext = np.random.rand(cell.totnsegs, t_ext.size)-0.5
>>> #insert potentials and record response:
```

(continues on next page)

(continued from previous page)

```
>>> cell.insert_v_ext(v_ext, t_ext)
>>> cell.simulate(rec_imem=True, rec_vmem=True)
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(311)
>>> ax2 = fig.add_subplot(312)
>>> ax3 = fig.add_subplot(313)
>>> eim = ax1.matshow(np.array(cell.v_ext), cmap='spectral')
>>> cb1 = fig.colorbar(eim, ax=ax1)
>>> cb1.set_label('v_ext')
>>> ax1.axis(ax1.axis('tight'))
>>> iim = ax2.matshow(cell.imem, cmap='spectral')
>>> cb2 = fig.colorbar(iim, ax=ax2)
>>> cb2.set_label('imem')
>>> ax2.axis(ax2.axis('tight'))
>>> vim = ax3.matshow(cell.vmem, cmap='spectral')
>>> ax3.axis(ax3.axis('tight'))
>>> cb3 = fig.colorbar(vim, ax=ax3)
>>> cb3.set_label('vmem')
>>> ax3.set_xlabel('tstep')
>>> plt.show()
```

set_point_process(*idx*, *pptype*, *record_current=False*, *record_potential=False*, ***kwargs*)

Insert pptype-electrode type pointprocess on segment numbered idx on cell object

Parameters

idx: int Index of compartment where point process is inserted

pptype: str Type of pointprocess. Examples: SEClamp, VClamp, IClamp, SinIClamp, ChirpIClamp

record_current: bool Decides if current is stored

kwargs Parameters passed on from class StimIntElectrode

Returns

int index of point process on cell

set_pos(*x=0.0*, *y=0.0*, *z=0.0*)

Set the cell position. Move the cell geometry so that midpoint of soma section is in (x, y, z). If no soma pos, use the first segment

Parameters

x: float x position defaults to 0.0

y: float y position defaults to 0.0

z: float z position defaults to 0.0

set_rotation(*x=None*, *y=None*, *z=None*, *rotation_order='xyz'*)

Rotate geometry of cell object around the x-, y-, z-axis in the order described by rotation_order parameter.

Parameters

x: float or None rotation angle in radians. Default: None

y: float or None rotation angle in radians. Default: None

z: float or None rotation angle in radians. Default: None

rotation_order: str string with 3 elements containing x, y and z e.g. 'xyz', 'zyx'. Default: 'xyz'

Examples

```
>>> cell = LFPy.Cell(**kwargs)
>>> rotation = {'x': 1.233, 'y': 0.236, 'z': np.pi}
>>> cell.set_rotation(**rotation)
```

`set_synapse`(*idx*, *syntype*, *record_current=False*, *record_potential=False*, *weight=None*, ***kwargs*)

Insert synapse on cell segment

Parameters

idx: int Index of compartment where synapse is inserted

syntype: str Type of synapse. Built-in types in NEURON: ExpSyn, Exp2Syn

record_current: bool If True, record synapse current

record_potential: bool If True, record postsynaptic potential seen by the synapse

weight: float Strength of synapse

kwargs arguments passed on from class Synapse

Returns

int index of synapse object on cell

`simulate`(*probes=None*, *rec_imem=False*, *rec_vmem=False*, *rec_ipas=False*, *rec_icap=False*, *rec_variables=[]*, *variable_dt=False*, *atol=0.001*, *rtol=0.0*, *to_memory=True*, *to_file=False*, *file_name=None*, ***kwargs*)

This is the main function running the simulation of the NEURON model. Start NEURON simulation and record variables specified by arguments.

Parameters

probes: list of [obj:, optional] None or list of LFPykit.RecExtElectrode like object instances that each have a public method *get_transformation_matrix* returning a matrix that linearly maps each compartments' transmembrane current to corresponding measurement as

$$\mathbf{P} = \mathbf{MI}$$

rec_imem: bool If true, segment membrane currents will be recorded If no electrode argument is given, it is necessary to set rec_imem=True in order to make predictions later on. Units of (nA).

rec_vmem: bool Record segment membrane voltages (mV)

rec_ipas: bool Record passive segment membrane currents (nA)

rec_icap: bool Record capacitive segment membrane currents (nA)

rec_variables: list List of segment state variables to record, e.g. arg=['cai',]

variable_dt: bool Use NEURON's variable timestep method

atol: float Absolute local error tolerance for NEURON variable timestep method

rtol: float Relative local error tolerance for NEURON variable timestep method

to_memory: bool Only valid with probes=[:obj:], store measurements as *:obj:.data*

to_file: bool Only valid with probes, save simulated data in hdf5 file format

file_name: str Name of hdf5 file, '.h5' is appended if it doesnt exist

`strip_hoc_objects`()

Destroy any NEURON hoc objects in the cell object

1.3.2 class TemplateCell

class LFPy.**TemplateCell**(*templatefile='LFPyCellTemplate.hoc'*, *templatename='LFPyCellTemplate'*, *templateargs=None*, *verbose=False*, ***kwargs*)

Bases: *LFPy.cell.Cell*

`LFPy.Cell` like class allowing use of NEURON templates with some limitations.

This takes all the same parameters as the Cell class, but requires three more template related parameters `templatefile`, `templatename` and `templateargs`

Parameters

morphology [str] path to morphology file

templatefile [str] File with cell template definition(s)

templatename [str] Cell template-name used for this cell object

templateargs [str] Parameters provided to template-definition

v_init [float] Initial membrane potential. Default to -65.

Ra [float] axial resistance. Defaults to 150.

cm [float] membrane capacitance. Defaults to 1.0

passive [bool] Passive mechanisms are initialized if True. Defaults to True

passive_parameters [dict] parameter dictionary with values for the passive membrane mechanism in NEURON ('pas'). The dictionary must contain keys 'g_pas' and 'e_pas', like the default: passive_parameters=dict(g_pas=0.001, e_pas=-70)

extracellular [bool] switch for NEURON's extracellular mechanism. Defaults to False

dt: float Simulation time step. Defaults to 2**-4

tstart [float] initialization time for simulation <= 0 ms. Defaults to 0.

tstop [float] stop time for simulation > 0 ms. Defaults to 100.

nsegs_method ['lambda100' or 'lambda_f' or 'fixed_length' or None] nseg rule, used by NEURON to determine number of compartments. Defaults to 'lambda100'

max_nsegs_length [float or None] max segment length for method 'fixed_length'. Defaults to None

lambda_f [int] AC frequency for method 'lambda_f'. Defaults to 100

d_lambda [float] parameter for d_lambda rule. Defaults to 0.1

delete_sections [bool] delete pre-existing section-references. Defaults to True

custom_code [list or None] list of model-specific code files ([.py/.hoc]). Defaults to None

custom_fun [list or None] list of model-specific functions with args. Defaults to None

custom_fun_args [list or None] list of args passed to custom_fun functions. Defaults to None

pt3d [bool] use pt3d-info of the cell geometries switch. Defaults to False

celsius [float or None] Temperature in celsius. If nothing is specified here or in custom code it is 6.3 celcius

verbose [bool] verbose output switch. Defaults to False

See also:

Cell

NetworkCell

Examples

```
>>> import LFPy
>>> cellParameters = {
>>>     'morphology' : '<path to morphology.hoc>',
>>>     'templatefile' :  '<path to template_file.hoc>'
>>>     'templatename' :  'templatename'
>>>     'templateargs' :  None
>>>     'v_init' : -65,
>>>     'cm' : 1.0,
>>>     'Ra' : 150,
>>>     'passive' : True,
>>>     'passive_parameters' : {'g_pas' : 0.001, 'e_pas' : -65.},
>>>     'dt' : 2**-3,
>>>     'tstart' : 0,
>>>     'tstop' : 50,
>>> }
>>> cell = LFPy.TemplateCell(**cellParameters)
>>> cell.simulate()
```

cellpickler(*filename*, *pickler=<built-in function dump>*)

Save data in cell to filename, using cPickle. It will however destroy any `neuron.h` objects upon saving, as c-objects cannot be pickled

Parameters

filename: str Where to save cell

Returns

None or pickle

Examples

```
>>> # To save a cell, issue:
>>> cell.cellpickler('cell.cpickle')
>>> # To load this cell again in another session:
>>> import cPickle
>>> with file('cell.cpickle', 'rb') as f:
>>>     cell = cPickle.load(f)
```

chiral_morphology(*axis='x'*)

Mirror the morphology around given axis, (default x-axis), useful to introduce more heterogeneouties in morphology shapes

Parameters

axis: str 'x' or 'y' or 'z'

distort_geometry(*factor=0.0*, *axis='z'*, *nu=0.0*)

Distorts cellular morphology with a relative factor along a chosen axis preserving Poisson's ratio. A ratio nu=0.5 assumes uncompressible and isotropic media that embeds the cell. A ratio nu=0 will only affect geometry along the chosen axis. A ratio nu=-1 will isometrically scale the neuron geometry along each axis. This method does not affect the underlying cable properties of the cell, only predictions of extracellular measurements (by affecting the relative locations of sources representing the compartments).

Parameters

factor: float relative compression/stretching factor of morphology. Default is 0 (no compression/stretching). Positive values implies a compression along the chosen axis.

axis: str which axis to apply compression/stretching. Default is "z".

nu: float Poisson's ratio. Ratio between axial and transversal compression/stretching. Default is 0.

enable_extracellular_stimulation(*electrode*, *t_ext=None*, *n=1*, *model='inf'*)

Enable extracellular stimulation with NEURON's *extracellular* mechanism. Extracellular potentials are computed from electrode currents using the point-source approximation. If `model` is `'inf'` (default), potentials are computed as (r_i is the position of a compartment i, r_n is the position of an electrode n, σ is the conductivity of the medium):

$$V_e(r_i) = \sum_n \frac{I_n}{4\pi\sigma|r_i - r_n|}$$

If `model` is `'semi'`, the method of images is used:

$$V_e(r_i) = \sum_n \frac{I_n}{2\pi\sigma|r_i - r_n|}$$

Parameters

electrode: RecExtElectrode Electrode object with stimulating currents

t_ext: np.ndarray or list Time in ms corresponding to step changes in the provided currents. If None, currents are assumed to have the same time steps as the NEURON simulation.

n: int Points per electrode for spatial averaging

model: str `'inf'` or `'semi'`. If `'inf'` the medium is assumed to be infinite and homogeneous. If `'semi'`, the method of images is used.

Returns

v_ext: np.ndarray Computed extracellular potentials at cell mid points

`get_axial_currents_from_vmem`(*timepoints=None*)

Compute axial currents from cell sim: get current magnitude, distance vectors and position vectors.

Parameters

timepoints: ndarray, dtype=int array of timepoints in simulation at which you want to compute the axial currents. Defaults to False. If not given, all simulation timesteps will be included.

Returns

i_axial: ndarray, dtype=float Shape ((cell.totnsegs-1)*2, len(timepoints)) array of axial current magnitudes I in units of (nA) in cell at all timesteps in timepoints, or at all timesteps of the simulation if timepoints=None. Contains two current magnitudes per segment, (except for the root segment): 1) the current from the mid point of the segment to the segment start point, and 2) the current from the segment start point to the mid point of the parent segment.

d_vectors: ndarray, dtype=float Shape (3, (cell.totnsegs-1)*2) array of distance vectors traveled by each axial current in i_axial in units of (µm). The indices of the first axis, correspond to the first axis of i_axial and pos_vectors.

pos_vectors: ndarray, dtype=float Shape ((cell.totnsegs-1)*2, 3) array of position vectors pointing to the mid point of each axial current in i_axial in units of (µm). The indices of the first axis, correspond to the first axis of i_axial and d_vectors.

Raises

AttributeError Raises an exeption if the cell.vmem attribute cannot be found

`get_axial_resistance`()

Return NEURON axial resistance for all cell compartments.

Returns

ri_list: ndarray, dtype=float Shape (cell.totnsegs,) array containing neuron.h.ri(seg.x) in units of (MOhm) for all segments in cell calculated using the neuron.h.ri(seg.x) method. neuron.h.ri(seg.x) returns the axial resistance from the middle of the segment to the middle of the parent segment. Note: If seg is the first segment in a section, i.e. the parent segment belongs to a different section or there is no parent section, then neuron.h.ri(seg.x) returns the axial resistance from the middle of the segment to the node connecting the segment to the parent section (or a ghost node if there is no parent)

`get_closest_idx`(*x=0.0*, *y=0.0*, *z=0.0*, *section='allsec'*)

Get the index number of a segment in specified section which midpoint is closest to the coordinates defined by the user

Parameters

x: float x-coordinate

y: float y-coordinate

z: float z-coordinate

section: str String matching a section-name. Defaults to 'allsec'.

Returns

int segment index

get_dict_of_children_idx()

Return dictionary with children segment indices for all sections.

Returns

children_dict: dictionary Dictionary containing a list for each section, with the segment index of all the section's children. The dictionary is needed to find the sibling of a segment.

get_dict_parent_connections()

Return dictionary with parent connection point for all sections.

Returns

connection_dict: dictionary Dictionary containing a float in range [0, 1] for each section in cell. The float gives the location on the parent segment to which the section is connected. The dictionary is needed for computing axial currents.

get_idx(*section='allsec'*, *z_min=- inf*, *z_max=inf*)

Returns compartment idx of segments from sections with names that match the pattern defined in input section on interval [z_min, z_max].

Parameters

section: str Any entry in cell.allsecnames or just 'allsec'.

z_min: float Depth filter. Specify minimum z-position

z_max: float Depth filter. Specify maximum z-position

Returns

ndarray, dtype=int segment indices

Examples

```
>>> idx = cell.get_idx(section='allsec')
>>> print(idx)
>>> idx = cell.get_idx(section=['soma', 'dend', 'apic'])
>>> print(idx)
```

get_idx_children(*parent='soma[0]'*)

Get the idx of parent's children sections, i.e. compartments ids of sections connected to parent-argument

Parameters

parent: str name-pattern matching a sectionname. Defaults to "soma[0]"

Returns

ndarray, dtype=int

get_idx_name(*idx=array([0])*)

Return NEURON convention name of segments with index idx. The returned argument is an array of tuples with corresponding segment idx, section name, and position along the section, like; [(0, 'neuron.h.soma[0]', 0.5),]

Parameters

idx: ndarray, dtype int segment indices, must be between 0 and cell.totnsegs

Returns

ndarray, dtype=object tuples with section names of segments

get_idx_parent_children(*parent='soma[0]'*)

Get all idx of segments of parent and children sections, i.e. segment idx of sections connected to parent-argument, and also of the parent segments

Parameters

parent: str name-pattern matching a sectionname. Defaults to "soma[0]"

Returns

ndarray, dtype=int

get_idx_polygons(*projection=('x', 'z')*)

For each segment idx in cell create a polygon in the plane determined by the projection kwarg (default ('x', 'z')), that can be visualized using plt.fill() or mpl.collections.PolyCollection

Parameters

projection: tuple of strings Determining projection. Defaults to ('x', 'z')

Returns

polygons: list list of (ndarray, ndarray) tuples giving the trajectory of each section

Examples

```
>>> from matplotlib.collections import PolyCollection
>>> import matplotlib.pyplot as plt
>>> cell = LFPy.Cell(morphology='PATH/TO/MORPHOLOGY')
>>> zips = []
>>> for x, z in cell.get_idx_polygons(projection=('x', 'z')):
>>>     zips.append(list(zip(x, z)))
>>> polycol = PolyCollection(zips,
>>>                          edgecolors='none',
>>>                          facecolors='gray')
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.add_collection(polycol)
>>> ax.axis(ax.axis('equal'))
>>> plt.show()
```

get_intersegment_distance(*idx0=0*, *idx1=0*)

Return the Euclidean distance between midpoints of two segments.

Parameters

idx0: int

idx1: int

Returns

float distance (µm).

get_intersegment_vector(*idx0=0*, *idx1=0*)

Return the distance between midpoints of two segments with index idx0 and idx1. The argument returned is a list [x, y, z], where x = self.x[idx1].mean(axis=-1) - self.x[idx0].mean(axis=-1) etc.

Parameters

idx0: int

idx1: int

Returns

list of floats distance between midpoints along x,y,z axis in µm

`get_multi_current_dipole_moments`(*timepoints=None*)

Return 3D current dipole moment vector and middle position vector from each axial current in space.

Parameters

timepoints: ndarray, dtype=int or None array of timepoints at which you want to compute the current dipole moments. Defaults to None. If not given, all simulation timesteps will be included.

Returns

multi_dipoles: ndarray, dtype = float Shape (n_axial_currents, 3, n_timepoints) array containing the x-,y-,z-components of the current dipole moment from each axial current in cell, at all timepoints. The number of axial currents, n_axial_currents = (cell.totnsegs-1) * 2 and the number of timepoints, n_timepoints = cell.tvec.size. The current dipole moments are given in units of (nA µm).

pos_axial: ndarray, dtype = float Shape (n_axial_currents, 3) array containing the x-, y-, and z-components giving the mid position in space of each multi_dipole in units of (µm).

Examples

Get all current dipole moments and positions from all axial currents in a single neuron simulation:

```
>>> import LFPy
>>> import numpy as np
>>> cell = LFPy.Cell('PATH/TO/MORPHOLOGY', extracellular=False)
>>> syn = LFPy.Synapse(cell, idx=cell.get_closest_idx(0,0,1000),
>>>                    syntype='ExpSyn', e=0., tau=1., weight=0.001)
>>> syn.set_spike_times(np.mgrid[20:100:20])
>>> cell.simulate(rec_vmem=True, rec_imem=False)
>>> timepoints = np.array([1,2,3,4])
>>> multi_dipoles, dipole_locs = cell.get_multi_current_dipole_moments(
>>>     timepoints=timepoints)
```

`get_pt3d_polygons`(*projection=('x', 'z')*)

For each section create a polygon in the plane determined by keyword argument projection=('x', 'z'), that can be visualized using e.g., plt.fill()

Parameters

projection: tuple of strings Determining projection. Defaults to ('x', 'z')

Returns

list list of (x, z) tuples giving the trajectory of each section that can be plotted using PolyCollection

Examples

```
>>> from matplotlib.collections import PolyCollection
>>> import matplotlib.pyplot as plt
>>> cell = LFPy.Cell(morphology='PATH/TO/MORPHOLOGY')
>>> zips = []
>>> for x, z in cell.get_pt3d_polygons(projection=('x', 'z')):
>>>     zips.append(list(zip(x, z)))
>>> polycol = PolyCollection(zips,
>>>                          edgecolors='none',
>>>                          facecolors='gray')
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.add_collection(polycol)
>>> ax.axis(ax.axis('equal'))
>>> plt.show()
```

get_rand_idx_area_and_distribution_norm(*section='allsec'*, *nidx=1*, *z_min=-1000000.0*, *z_max=1000000.0*, *fun=<scipy.stats._continuous_distns.norm_gen object>*, *funargs={'loc': 0, 'scale': 100}*, *funweights=None*)

Return nidx segment indices in section with random probability normalized to the membrane area of each segment multiplied by the value of the probability density function of "fun", a function in the scipy.stats module with corresponding function arguments in "funargs" on the interval [z_min, z_max]

Parameters

section: str string matching a section name

nidx: int number of random indices

z_min: float lower depth interval

z_max: float upper depth interval

fun: function or str, or iterable of function or str if function a scipy.stats method, if str, must be method in scipy.stats module with the same name (like 'norm'), if iterable (list, tuple, numpy.array) of function or str some probability distribution in scipy.stats module

funargs: dict or iterable iterable (list, tuple, numpy.array) of dict, arguments to fun.pdf method (e.g., w. keys 'loc' and 'scale')

funweights: None or iterable iterable (list, tuple, numpy.array) of floats, scaling of each individual fun (i.e., introduces layer specificity)

Examples

```
>>> import LFPy
>>> import numpy as np
>>> import scipy.stats as ss
>>> import matplotlib.pyplot as plt
>>> from os.path import join
>>> cell = LFPy.Cell(morphology=join('cells', 'cells', 'j4a.hoc'))
>>> cell.set_rotation(x=4.99, y=-4.33, z=3.14)
>>> idx = cell.get_rand_idx_area_and_distribution_norm(
```

(continues on next page)

(continued from previous page)

```
    nidx=10000, fun=ss.norm, funargs=dict(loc=0, scale=200))
>>> bins = np.arange(-30, 120)*10
>>> plt.hist(cell.zmid[idx], bins=bins, alpha=0.5)
>>> plt.show()
```

get_rand_idx_area_norm(*section='allsec'*, *nidx=1*, *z_min=- 1000000.0*, *z_max=1000000.0*)

Return nidx segment indices in section with random probability normalized to the membrane area of segment on interval [z_min, z_max]

Parameters

section: str String matching a section-name

nidx: int Number of random indices

z_min: float Depth filter

z_max: float Depth filter

Returns

ndarray, dtype=int segment indices

get_rand_prob_area_norm(*section='allsec'*, *z_min=- 10000*, *z_max=10000*)

Return the probability (0-1) for synaptic coupling on segments in section sum(prob)=1 over all segments in section. Probability normalized by area.

Parameters

section: str string matching a section-name. Defaults to 'allsec'

z_min: float depth filter

z_max: float depth filter

Returns

ndarray, dtype=float

get_rand_prob_area_norm_from_idx(*idx=array([0])*)

Return the normalized probability (0-1) for synaptic coupling on segments in idx-array. Normalised probability determined by area of segments.

Parameters

idx: ndarray, dtype=int. array of segment indices

Returns

ndarray, dtype=float

insert_v_ext(*v_ext*, *t_ext*)

Set external extracellular potential around cell. Playback of some extracellular potential v_ext on each cell.totnseg compartments. Assumes that the "extracellular"-mechanism is inserted on each compartment. Can be used to study ephaptic effects and similar The inputs will be copied and attached to the cell object as cell.v_ext, cell.t_ext, and converted to (list of) neuron.h.Vector types, to allow playback into each compartment e_extracellular reference. Can not be deleted prior to running cell.simulate()

Parameters

v_ext: ndarray Numpy array of size cell.totnsegs x t_ext.size, unit mV

t_ext: ndarray Time vector of v_ext in ms

Examples

```
>>> import LFPy
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> #create cell
>>> cell = LFPy.Cell(morphology='morphologies/example_morphology.hoc',
>>>                  passive=True)
>>> #time vector and extracellular field for every segment:
>>> t_ext = np.arange(cell.tstop / cell.dt+ 1) * cell.dt
>>> v_ext = np.random.rand(cell.totnsegs, t_ext.size)-0.5
>>> #insert potentials and record response:
>>> cell.insert_v_ext(v_ext, t_ext)
>>> cell.simulate(rec_imem=True, rec_vmem=True)
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(311)
>>> ax2 = fig.add_subplot(312)
>>> ax3 = fig.add_subplot(313)
>>> eim = ax1.matshow(np.array(cell.v_ext), cmap='spectral')
>>> cb1 = fig.colorbar(eim, ax=ax1)
>>> cb1.set_label('v_ext')
>>> ax1.axis(ax1.axis('tight'))
>>> iim = ax2.matshow(cell.imem, cmap='spectral')
>>> cb2 = fig.colorbar(iim, ax=ax2)
>>> cb2.set_label('imem')
>>> ax2.axis(ax2.axis('tight'))
>>> vim = ax3.matshow(cell.vmem, cmap='spectral')
>>> ax3.axis(ax3.axis('tight'))
>>> cb3 = fig.colorbar(vim, ax=ax3)
>>> cb3.set_label('vmem')
>>> ax3.set_xlabel('tstep')
>>> plt.show()
```

set_point_process(*idx*, *pptype*, *record_current=False*, *record_potential=False*, ***kwargs*)

Insert pptype-electrode type pointprocess on segment numbered idx on cell object

Parameters

idx: int Index of compartment where point process is inserted

pptype: str Type of pointprocess. Examples: SEClamp, VClamp, IClamp, SinIClamp, ChirpIClamp

record_current: bool Decides if current is stored

kwargs Parameters passed on from class StimIntElectrode

Returns

int index of point process on cell

set_pos(*x=0.0*, *y=0.0*, *z=0.0*)

Set the cell position. Move the cell geometry so that midpoint of soma section is in (x, y, z). If no soma pos, use the first segment

Parameters

x: float x position defaults to 0.0

y: float y position defaults to 0.0

z: float z position defaults to 0.0

set_rotation(*x=None*, *y=None*, *z=None*, *rotation_order='xyz'*)

Rotate geometry of cell object around the x-, y-, z-axis in the order described by rotation_order parameter.

Parameters

x: float or None rotation angle in radians. Default: None

y: float or None rotation angle in radians. Default: None

z: float or None rotation angle in radians. Default: None

rotation_order: str string with 3 elements containing x, y and z e.g. 'xyz', 'zyx'. Default: 'xyz'

Examples

```
>>> cell = LFPy.Cell(**kwargs)
>>> rotation = {'x': 1.233, 'y': 0.236, 'z': np.pi}
>>> cell.set_rotation(**rotation)
```

set_synapse(*idx*, *syntype*, *record_current=False*, *record_potential=False*, *weight=None*, ***kwargs*)

Insert synapse on cell segment

Parameters

idx: int Index of compartment where synapse is inserted

syntype: str Type of synapse. Built-in types in NEURON: ExpSyn, Exp2Syn

record_current: bool If True, record synapse current

record_potential: bool If True, record postsynaptic potential seen by the synapse

weight: float Strength of synapse

kwargs arguments passed on from class Synapse

Returns

int index of synapse object on cell

simulate(*probes=None*, *rec_imem=False*, *rec_vmem=False*, *rec_ipas=False*, *rec_icap=False*, *rec_variables=[]*, *variable_dt=False*, *atol=0.001*, *rtol=0.0*, *to_memory=True*, *to_file=False*, *file_name=None*, ***kwargs*)

This is the main function running the simulation of the NEURON model. Start NEURON simulation and record variables specified by arguments.

Parameters

probes: list of [obj:, optional] None or list of LFPykit.RecExtElectrode like object instances that each have a public method *get_transformation_matrix* returning a matrix that linearly maps each compartments' transmembrane current to corresponding measurement as

$$\mathbf{P} = \mathbf{MI}$$

rec_imem: bool If true, segment membrane currents will be recorded If no electrode argument is given, it is necessary to set rec_imem=True in order to make predictions later on. Units of (nA).

rec_vmem: bool Record segment membrane voltages (mV)

rec_ipas: bool Record passive segment membrane currents (nA)

rec_icap: bool Record capacitive segment membrane currents (nA)

rec_variables: list List of segment state variables to record, e.g. arg=['cai',]

variable_dt: bool Use NEURON's variable timestep method

atol: float Absolute local error tolerance for NEURON variable timestep method

rtol: float Relative local error tolerance for NEURON variable timestep method

to_memory: bool Only valid with probes=[:obj:], store measurements as *:obj:.data*

to_file: bool Only valid with probes, save simulated data in hdf5 file format

file_name: str Name of hdf5 file, '.h5' is appended if it doesnt exist

`strip_hoc_objects()`

Destroy any NEURON hoc objects in the cell object

1.3.3 class NetworkCell

`class LFPy.NetworkCell(**args)`

Bases: *LFPy.templatecell.TemplateCell*

Similar to *LFPy.TemplateCell* with the addition of some attributes and methods allowing for spike communication between parallel RANKs.

This class allow using NEURON templates with some limitations.

This takes all the same parameters as the Cell class, but requires three more template related parameters

Parameters

morphology: str path to morphology file

templatefile: str File with cell template definition(s)

templatename: str Cell template-name used for this cell object

templateargs: str Parameters provided to template-definition

v_init: float Initial membrane potential. Default to -65.

Ra: float axial resistance. Defaults to 150.

cm: float membrane capacitance. Defaults to 1.0

passive: bool Passive mechanisms are initialized if True. Defaults to True

passive_parameters: dict parameter dictionary with values for the passive membrane mechanism in NEURON ('pas'). The dictionary must contain keys 'g_pas' and 'e_pas', like the default: passive_parameters=dict(g_pas=0.001, e_pas=-70)

extracellular: bool switch for NEURON's extracellular mechanism. Defaults to False

dt: float Simulation time step. Defaults to 2**-4

tstart: float initialization time for simulation <= 0 ms. Defaults to 0.

tstop: float stop time for simulation > 0 ms. Defaults to 100.

nsegs_method: 'lambda100' or 'lambda_f' or 'fixed_length' or None nseg rule, used by NEURON to determine number of compartments. Defaults to 'lambda100'

max_nsegs_length: float or None max segment length for method 'fixed_length'. Defaults to None

lambda_f: int AC frequency for method 'lambda_f'. Defaults to 100

d_lambda: float parameter for d_lambda rule. Defaults to 0.1

delete_sections: bool delete pre-existing section-references. Defaults to True

custom_code: list or None list of model-specific code files ([.py/.hoc]). Defaults to None

custom_fun: list or None list of model-specific functions with args. Defaults to None

custom_fun_args: list or None list of args passed to custom_fun functions. Defaults to None

pt3d: bool use pt3d-info of the cell geometries switch. Defaults to False

celsius: float or None Temperature in celsius. If nothing is specified here or in custom code it is 6.3 celcius

verbose: bool verbose output switch. Defaults to False

See also:

Cell

TemplateCell

Examples

```
>>> import LFPy
>>> cellParameters = {
>>>     'morphology': '<path to morphology.hoc>',
>>>     'templatefile':  '<path to template_file.hoc>',
>>>     'templatename':  'templatename',
>>>     'templateargs':  None,
>>>     'v_init': -65,
>>>     'cm': 1.0,
>>>     'Ra': 150,
>>>     'passive': True,
>>>     'passive_parameters': {'g_pas': 0.001, 'e_pas': -65.},
>>>     'dt': 2**-3,
>>>     'tstart': 0,
>>>     'tstop': 50,
>>> }
>>> cell = LFPy.NetworkCell(**cellParameters)
>>> cell.simulate()
```

cellpickler(*filename*, *pickler=<built-in function dump>*)

Save data in cell to filename, using cPickle. It will however destroy any `neuron.h` objects upon saving, as c-objects cannot be pickled

Parameters

filename: str Where to save cell

Returns

None or pickle

Examples

```
>>> # To save a cell, issue:
>>> cell.cellpickler('cell.cpickle')
>>> # To load this cell again in another session:
>>> import cPickle
>>> with file('cell.cpickle', 'rb') as f:
>>>     cell = cPickle.load(f)
```

chiral_morphology(*axis='x'*)

Mirror the morphology around given axis, (default x-axis), useful to introduce more heterogeneouties in morphology shapes

Parameters

axis: str 'x' or 'y' or 'z'

create_spike_detector(*target=None*, *threshold=- 10.0*, *weight=0.0*, *delay=0.0*)

Create spike-detecting NetCon object attached to the cell's soma midpoint, but this could be extended to having multiple spike-detection sites. The NetCon object created is attached to the cell's *_hoc_sd_netconlist* attribute, and will be used by the Network class when creating connections between all presynaptic cells and postsynaptic cells on each local RANK.

Parameters

target: None (default) or a NEURON point process

threshold: float spike detection threshold

weight: float connection weight (not used unless target is a point process)

delay: float connection delay (not used unless target is a point process)

create_synapse(*cell*, *sec*, *x=0.5*, *syntype=ExpSyn()*, *synparams={'e': 0.0, 'tau': 2.0}*, *assert_syn_values=False*)

Create synapse object of type syntype on sec(x) of cell and append to list cell.netconsynapses

TODO: Use LFPy.Synapse class if possible.

Parameters

cell: object instantiation of class NetworkCell or similar

sec: neuron.h.Section object, section reference on cell

x: float in [0, 1], relative position along section

syntype: hoc.HocObject NEURON synapse model reference, e.g., neuron.h.ExpSyn

synparams: dict

parameters for syntype, e.g., for neuron.h.ExpSyn we have: tau: float, synapse time constant e: float, synapse reversal potential

assert_syn_values: bool if True, raise AssertionError if synapse attribute values do not match the values in the synparams dictionary

Raises

AssertionError

`distort_geometry`(*factor=0.0*, *axis='z'*, *nu=0.0*)

Distorts cellular morphology with a relative factor along a chosen axis preserving Poisson's ratio. A ratio nu=0.5 assumes uncompressible and isotropic media that embeds the cell. A ratio nu=0 will only affect geometry along the chosen axis. A ratio nu=-1 will isometrically scale the neuron geometry along each axis. This method does not affect the underlying cable properties of the cell, only predictions of extracellular measurements (by affecting the relative locations of sources representing the compartments).

Parameters

factor: float relative compression/stretching factor of morphology. Default is 0 (no compression/stretching). Positive values implies a compression along the chosen axis.

axis: str which axis to apply compression/stretching. Default is "z".

nu: float Poisson's ratio. Ratio between axial and transversal compression/stretching. Default is 0.

`enable_extracellular_stimulation`(*electrode*, *t_ext=None*, *n=1*, *model='inf'*)

Enable extracellular stimulation with NEURON's *extracellular* mechanism. Extracellular potentials are computed from electrode currents using the point-source approximation. If `model` is `'inf'` (default), potentials are computed as (r_i is the position of a compartment i, r_n is the position of an electrode n, σ is the conductivity of the medium):

$$V_e(r_i) = \sum_n \frac{I_n}{4\pi\sigma|r_i - r_n|}$$

If `model` is `'semi'`, the method of images is used:

$$V_e(r_i) = \sum_n \frac{I_n}{2\pi\sigma|r_i - r_n|}$$

Parameters

electrode: RecExtElectrode Electrode object with stimulating currents

t_ext: np.ndarray or list Time in ms corresponding to step changes in the provided currents. If None, currents are assumed to have the same time steps as the NEURON simulation.

n: int Points per electrode for spatial averaging

model: str `'inf'` or `'semi'`. If `'inf'` the medium is assumed to be infinite and homogeneous. If `'semi'`, the method of images is used.

Returns

v_ext: np.ndarray Computed extracellular potentials at cell mid points

`get_axial_currents_from_vmem`(*timepoints=None*)

Compute axial currents from cell sim: get current magnitude, distance vectors and position vectors.

Parameters

timepoints: ndarray, dtype=int array of timepoints in simulation at which you want to compute the axial currents. Defaults to False. If not given, all simulation timesteps will be included.

Returns

i_axial: ndarray, dtype=float Shape ((cell.totnsegs-1)*2, len(timepoints)) array of axial current magnitudes I in units of (nA) in cell at all timesteps in timepoints, or at all timesteps of the simulation if timepoints=None. Contains two current magnitudes per segment, (except for the root segment): 1) the current from the mid point of the segment to the segment

start point, and 2) the current from the segment start point to the mid point of the parent segment.

d_vectors: ndarray, dtype=float Shape (3, (cell.totnsegs-1)*2) array of distance vectors traveled by each axial current in i_axial in units of (µm). The indices of the first axis, correspond to the first axis of i_axial and pos_vectors.

pos_vectors: ndarray, dtype=float Shape ((cell.totnsegs-1)*2, 3) array of position vectors pointing to the mid point of each axial current in i_axial in units of (µm). The indices of the first axis, correspond to the first axis of i_axial and d_vectors.

Raises

AttributeError Raises an exeption if the cell.vmem attribute cannot be found

`get_axial_resistance()`

Return NEURON axial resistance for all cell compartments.

Returns

ri_list: ndarray, dtype=float Shape (cell.totnsegs,) array containing neuron.h.ri(seg.x) in units of (MOhm) for all segments in cell calculated using the neuron.h.ri(seg.x) method. neuron.h.ri(seg.x) returns the axial resistance from the middle of the segment to the middle of the parent segment. Note: If seg is the first segment in a section, i.e. the parent segment belongs to a different section or there is no parent section, then neuron.h.ri(seg.x) returns the axial resistance from the middle of the segment to the node connecting the segment to the parent section (or a ghost node if there is no parent)

`get_closest_idx(x=0.0, y=0.0, z=0.0, section='allsec')`

Get the index number of a segment in specified section which midpoint is closest to the coordinates defined by the user

Parameters

x: float x-coordinate

y: float y-coordinate

z: float z-coordinate

section: str String matching a section-name. Defaults to 'allsec'.

Returns

int segment index

`get_dict_of_children_idx()`

Return dictionary with children segment indices for all sections.

Returns

children_dict: dictionary Dictionary containing a list for each section, with the segment index of all the section's children. The dictionary is needed to find the sibling of a segment.

`get_dict_parent_connections()`

Return dictionary with parent connection point for all sections.

Returns

connection_dict: dictionary Dictionary containing a float in range [0, 1] for each section in cell. The float gives the location on the parent segment to which the section is connected. The dictionary is needed for computing axial currents.

get_idx(*section='allsec'*, *z_min=-inf*, *z_max=inf*)

Returns compartment idx of segments from sections with names that match the pattern defined in input section on interval [z_min, z_max].

Parameters

section: str Any entry in cell.allsecnames or just 'allsec'.

z_min: float Depth filter. Specify minimum z-position

z_max: float Depth filter. Specify maximum z-position

Returns

ndarray, dtype=int segment indices

Examples

```
>>> idx = cell.get_idx(section='allsec')
>>> print(idx)
>>> idx = cell.get_idx(section=['soma', 'dend', 'apic'])
>>> print(idx)
```

get_idx_children(*parent='soma[0]'*)

Get the idx of parent's children sections, i.e. compartments ids of sections connected to parent-argument

Parameters

parent: str name-pattern matching a sectionname. Defaults to "soma[0]"

Returns

ndarray, dtype=int

get_idx_name(*idx=array([0])*)

Return NEURON convention name of segments with index idx. The returned argument is an array of tuples with corresponding segment idx, section name, and position along the section, like; [(0, 'neuron.h.soma[0]', 0.5),]

Parameters

idx: ndarray, dtype int segment indices, must be between 0 and cell.totnsegs

Returns

ndarray, dtype=object tuples with section names of segments

get_idx_parent_children(*parent='soma[0]'*)

Get all idx of segments of parent and children sections, i.e. segment idx of sections connected to parent-argument, and also of the parent segments

Parameters

parent: str name-pattern matching a sectionname. Defaults to "soma[0]"

Returns

ndarray, dtype=int

get_idx_polygons(*projection=('x', 'z')*)

For each segment idx in cell create a polygon in the plane determined by the projection kwarg (default ('x', 'z')), that can be visualized using plt.fill() or mpl.collections.PolyCollection

Parameters

projection: tuple of strings Determining projection. Defaults to ('x', 'z')

Returns

polygons: list list of (ndarray, ndarray) tuples giving the trajectory of each section

Examples

```
>>> from matplotlib.collections import PolyCollection
>>> import matplotlib.pyplot as plt
>>> cell = LFPy.Cell(morphology='PATH/TO/MORPHOLOGY')
>>> zips = []
>>> for x, z in cell.get_idx_polygons(projection=('x', 'z')):
>>>     zips.append(list(zip(x, z)))
>>> polycol = PolyCollection(zips,
>>>                          edgecolors='none',
>>>                          facecolors='gray')
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.add_collection(polycol)
>>> ax.axis(ax.axis('equal'))
>>> plt.show()
```

get_intersegment_distance(*idx0=0*, *idx1=0*)

Return the Euclidean distance between midpoints of two segments.

Parameters

idx0: int

idx1: int

Returns

float distance (µm).

get_intersegment_vector(*idx0=0*, *idx1=0*)

Return the distance between midpoints of two segments with index idx0 and idx1. The argument returned is a list [x, y, z], where x = self.x[idx1].mean(axis=-1) - self.x[idx0].mean(axis=-1) etc.

Parameters

idx0: int

idx1: int

Returns

list of floats distance between midpoints along x,y,z axis in µm

get_multi_current_dipole_moments(*timepoints=None*)

Return 3D current dipole moment vector and middle position vector from each axial current in space.

Parameters

timepoints: ndarray, dtype=int or None array of timepoints at which you want to compute the current dipole moments. Defaults to None. If not given, all simulation timesteps will be included.

Returns

multi_dipoles: ndarray, dtype = float Shape (n_axial_currents, 3, n_timepoints) array containing the x-,y-,z-components of the current dipole moment from each axial current in cell, at all timepoints. The number of axial currents, n_axial_currents = (cell.totnsegs-1) * 2 and the number of timepoints, n_timepoints = cell.tvec.size. The current dipole moments are given in units of (nA µm).

pos_axial: ndarray, dtype = float Shape (n_axial_currents, 3) array containing the x-, y-, and z-components giving the mid position in space of each multi_dipole in units of (µm).

Examples

Get all current dipole moments and positions from all axial currents in a single neuron simulation:

```
>>> import LFPy
>>> import numpy as np
>>> cell = LFPy.Cell('PATH/TO/MORPHOLOGY', extracellular=False)
>>> syn = LFPy.Synapse(cell, idx=cell.get_closest_idx(0,0,1000),
>>>                    syntype='ExpSyn', e=0., tau=1., weight=0.001)
>>> syn.set_spike_times(np.mgrid[20:100:20])
>>> cell.simulate(rec_vmem=True, rec_imem=False)
>>> timepoints = np.array([1,2,3,4])
>>> multi_dipoles, dipole_locs = cell.get_multi_current_dipole_moments(
>>>     timepoints=timepoints)
```

get_pt3d_polygons(*projection=('x', 'z')*)

For each section create a polygon in the plane determined by keyword argument projection=('x', 'z'), that can be visualized using e.g., plt.fill()

Parameters

projection: tuple of strings Determining projection. Defaults to ('x', 'z')

Returns

list list of (x, z) tuples giving the trajectory of each section that can be plotted using PolyCollection

Examples

```
>>> from matplotlib.collections import PolyCollection
>>> import matplotlib.pyplot as plt
>>> cell = LFPy.Cell(morphology='PATH/TO/MORPHOLOGY')
>>> zips = []
>>> for x, z in cell.get_pt3d_polygons(projection=('x', 'z')):
>>>     zips.append(list(zip(x, z)))
>>> polycol = PolyCollection(zips,
>>>                          edgecolors='none',
>>>                          facecolors='gray')
>>> fig = plt.figure()
```

(continues on next page)

(continued from previous page)

```
>>> ax = fig.add_subplot(111)
>>> ax.add_collection(polycol)
>>> ax.axis(ax.axis('equal'))
>>> plt.show()
```

get_rand_idx_area_and_distribution_norm(*section='allsec'*, *nidx=1*, *z_min=-1000000.0*, *z_max=1000000.0*, *fun=<scipy.stats._continuous_distns.norm_gen object>*, *funargs={'loc': 0, 'scale': 100}*, *funweights=None*)

Return nidx segment indices in section with random probability normalized to the membrane area of each segment multiplied by the value of the probability density function of "fun", a function in the scipy.stats module with corresponding function arguments in "funargs" on the interval [z_min, z_max]

Parameters

section: str string matching a section name

nidx: int number of random indices

z_min: float lower depth interval

z_max: float upper depth interval

fun: function or str, or iterable of function or str if function a scipy.stats method, if str, must be method in scipy.stats module with the same name (like 'norm'), if iterable (list, tuple, numpy.array) of function or str some probability distribution in scipy.stats module

funargs: dict or iterable iterable (list, tuple, numpy.array) of dict, arguments to fun.pdf method (e.g., w. keys 'loc' and 'scale')

funweights: None or iterable iterable (list, tuple, numpy.array) of floats, scaling of each individual fun (i.e., introduces layer specificity)

Examples

```
>>> import LFPy
>>> import numpy as np
>>> import scipy.stats as ss
>>> import matplotlib.pyplot as plt
>>> from os.path import join
>>> cell = LFPy.Cell(morphology=join('cells', 'cells', 'j4a.hoc'))
>>> cell.set_rotation(x=4.99, y=-4.33, z=3.14)
>>> idx = cell.get_rand_idx_area_and_distribution_norm(
    nidx=10000, fun=ss.norm, funargs=dict(loc=0, scale=200))
>>> bins = np.arange(-30, 120)*10
>>> plt.hist(cell.zmid[idx], bins=bins, alpha=0.5)
>>> plt.show()
```

get_rand_idx_area_norm(*section='allsec'*, *nidx=1*, *z_min=- 1000000.0*, *z_max=1000000.0*)

Return nidx segment indices in section with random probability normalized to the membrane area of segment on interval [z_min, z_max]

Parameters

section: str String matching a section-name

nidx: int Number of random indices

z_min: float Depth filter

z_max: float Depth filter

Returns

ndarray, dtype=int segment indices

`get_rand_prob_area_norm`(*section='allsec'*, *z_min=- 10000*, *z_max=10000*)

Return the probability (0-1) for synaptic coupling on segments in section sum(prob)=1 over all segments in section. Probability normalized by area.

Parameters

section: str string matching a section-name. Defaults to 'allsec'

z_min: float depth filter

z_max: float depth filter

Returns

ndarray, dtype=float

`get_rand_prob_area_norm_from_idx`(*idx=array([0])*)

Return the normalized probability (0-1) for synaptic coupling on segments in idx-array. Normalised probability determined by area of segments.

Parameters

idx: ndarray, dtype=int. array of segment indices

Returns

ndarray, dtype=float

`insert_v_ext`(*v_ext*, *t_ext*)

Set external extracellular potential around cell. Playback of some extracellular potential v_ext on each cell.totnseg compartments. Assumes that the "extracellular"-mechanism is inserted on each compartment. Can be used to study ephaptic effects and similar The inputs will be copied and attached to the cell object as cell.v_ext, cell.t_ext, and converted to (list of) neuron.h.Vector types, to allow playback into each compartment e_extracellular reference. Can not be deleted prior to running cell.simulate()

Parameters

v_ext: ndarray Numpy array of size cell.totnsegs x t_ext.size, unit mV

t_ext: ndarray Time vector of v_ext in ms

Examples

```
>>> import LFPy
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> #create cell
>>> cell = LFPy.Cell(morphology='morphologies/example_morphology.hoc',
>>>                  passive=True)
>>> #time vector and extracellular field for every segment:
>>> t_ext = np.arange(cell.tstop / cell.dt+ 1) * cell.dt
>>> v_ext = np.random.rand(cell.totnsegs, t_ext.size)-0.5
>>> #insert potentials and record response:
```

(continues on next page)

(continued from previous page)

```
>>> cell.insert_v_ext(v_ext, t_ext)
>>> cell.simulate(rec_imem=True, rec_vmem=True)
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(311)
>>> ax2 = fig.add_subplot(312)
>>> ax3 = fig.add_subplot(313)
>>> eim = ax1.matshow(np.array(cell.v_ext), cmap='spectral')
>>> cb1 = fig.colorbar(eim, ax=ax1)
>>> cb1.set_label('v_ext')
>>> ax1.axis(ax1.axis('tight'))
>>> iim = ax2.matshow(cell.imem, cmap='spectral')
>>> cb2 = fig.colorbar(iim, ax=ax2)
>>> cb2.set_label('imem')
>>> ax2.axis(ax2.axis('tight'))
>>> vim = ax3.matshow(cell.vmem, cmap='spectral')
>>> ax3.axis(ax3.axis('tight'))
>>> cb3 = fig.colorbar(vim, ax=ax3)
>>> cb3.set_label('vmem')
>>> ax3.set_xlabel('tstep')
>>> plt.show()
```

set_point_process(*idx*, *pptype*, *record_current=False*, *record_potential=False*, ***kwargs*)

Insert pptype-electrode type pointprocess on segment numbered idx on cell object

Parameters

idx: int Index of compartment where point process is inserted

pptype: str Type of pointprocess. Examples: SEClamp, VClamp, IClamp, SinIClamp, ChirpIClamp

record_current: bool Decides if current is stored

kwargs Parameters passed on from class StimIntElectrode

Returns

int index of point process on cell

set_pos(*x=0.0*, *y=0.0*, *z=0.0*)

Set the cell position. Move the cell geometry so that midpoint of soma section is in (x, y, z). If no soma pos, use the first segment

Parameters

x: float x position defaults to 0.0

y: float y position defaults to 0.0

z: float z position defaults to 0.0

set_rotation(*x=None*, *y=None*, *z=None*, *rotation_order='xyz'*)

Rotate geometry of cell object around the x-, y-, z-axis in the order described by rotation_order parameter.

Parameters

x: float or None rotation angle in radians. Default: None

y: float or None rotation angle in radians. Default: None

z: float or None rotation angle in radians. Default: None

rotation_order: str string with 3 elements containing x, y and z e.g. 'xyz', 'zyx'. Default: 'xyz'

Examples

```
>>> cell = LFPy.Cell(**kwargs)
>>> rotation = {'x': 1.233, 'y': 0.236, 'z': np.pi}
>>> cell.set_rotation(**rotation)
```

`set_synapse`(*idx*, *syntype*, *record_current=False*, *record_potential=False*, *weight=None*, ***kwargs*)

Insert synapse on cell segment

Parameters

idx: int Index of compartment where synapse is inserted

syntype: str Type of synapse. Built-in types in NEURON: ExpSyn, Exp2Syn

record_current: bool If True, record synapse current

record_potential: bool If True, record postsynaptic potential seen by the synapse

weight: float Strength of synapse

kwargs arguments passed on from class Synapse

Returns

int index of synapse object on cell

`simulate`(*probes=None*, *rec_imem=False*, *rec_vmem=False*, *rec_ipas=False*, *rec_icap=False*, *rec_variables=[]*, *variable_dt=False*, *atol=0.001*, *rtol=0.0*, *to_memory=True*, *to_file=False*, *file_name=None*, ***kwargs*)

This is the main function running the simulation of the NEURON model. Start NEURON simulation and record variables specified by arguments.

Parameters

probes: list of [obj:, optional] None or list of LFPykit.RecExtElectrode like object instances that each have a public method *get_transformation_matrix* returning a matrix that linearly maps each compartments' transmembrane current to corresponding measurement as

$$\mathbf{P} = \mathbf{MI}$$

rec_imem: bool If true, segment membrane currents will be recorded If no electrode argument is given, it is necessary to set rec_imem=True in order to make predictions later on. Units of (nA).

rec_vmem: bool Record segment membrane voltages (mV)

rec_ipas: bool Record passive segment membrane currents (nA)

rec_icap: bool Record capacitive segment membrane currents (nA)

rec_variables: list List of segment state variables to record, e.g. arg=['cai',]

variable_dt: bool Use NEURON's variable timestep method

atol: float Absolute local error tolerance for NEURON variable timestep method

rtol: float Relative local error tolerance for NEURON variable timestep method

to_memory: bool Only valid with probes=[:obj:], store measurements as *:obj:.data*

to_file: bool Only valid with probes, save simulated data in hdf5 file format

file_name: str Name of hdf5 file, '.h5' is appended if it doesnt exist

`strip_hoc_objects`()

Destroy any NEURON hoc objects in the cell object

1.4 Point processes

1.4.1 class PointProcess

`class` LFPy.**PointProcess**(*cell*, *idx*, *record_current=False*, *record_potential=False*, ***kwargs*)

Bases: `object`

Parent class of Synapse, StimIntElectrode. Created in order to import and set some shared variables and extract Cartesian coordinates of segments

Parameters

cell: obj LFPy.Cell object

idx: int index of segment

record_current: bool Must be set to True for recording of pointprocess currents

record_potential: bool Must be set to True for recording potential of pointprocess target idx

kwargs: pointprocess specific variables passed on to cell/neuron

See also:

Synapse

StimIntElectrode

`update_pos`(*cell*)

Extract coordinates of point-process

1.4.2 class Synapse

`class` LFPy.**Synapse**(*cell*, *idx*, *syntype*, *record_current=False*, *record_potential=False*, ***kwargs*)

Bases: *LFPy.pointprocess.PointProcess*

The synapse class, pointprocesses that spawn membrane currents. See http://www.neuron.yale.edu/neuron/static/docs/help/neuron/neuron/ mech.html#pointprocesses for details, or corresponding mod-files.

This class is meant to be used with synaptic mechanisms, giving rise to currents that will be part of the membrane currents at times governed by the methods *set_spike_times* or *set_spike_times_w_netstim*.

Parameters

cell: obj *LFPy.Cell* or *LFPy.TemplateCell* instance to receive synapptic input

idx: int Cell index where the synaptic input arrives

syntype: str Type of synapse, such as 'ExpSyn', 'Exp2Syn', 'AlphaSynapse'

record_current: bool If True, record synapse to *<synapse>.i* in units of nA

****kwargs** Additional arguments to be passed on to NEURON in *Cell.set_synapse*

See also:

StimIntElectrode

Examples

```
>>> import pylab as pl
>>> pl.interactive(1)
>>> import LFPy
>>> import os
>>> cellParameters = {
>>>     'morphology':  os.path.join('examples', 'morphologies',
>>>                                 'L5_Mainen96_LFPy.hoc'),
>>>     'passive': True,
>>>     'tstop':     50,
>>> }
>>> cell = LFPy.Cell(**cellParameters)
```

```
>>> synapseParameters = {
>>>     'idx': cell.get_closest_idx(x=0, y=0, z=800),
>>>     'e': 0,                                  # reversal potential
>>>     'syntype': 'ExpSyn',                     # synapse type
>>>     'tau': 2,                                # syn. time constant
>>>     'weight': 0.01,                          # syn. weight
>>>     'record_current': True                   # syn. current record
>>> }
>>> synapse = LFPy.Synapse(cell, **synapseParameters)
>>> synapse.set_spike_times(pl.array([10, 15, 20, 25]))
>>> cell.simulate()
```

```
>>> pl.subplot(211)
>>> pl.plot(cell.tvec, synapse.i)
>>> pl.title('Synapse current (nA)')
>>> pl.subplot(212)
>>> pl.plot(cell.tvec, cell.somav)
>>> pl.title('Somatic potential (mV)')
```

collect_current(*cell*)

Collect synapse current. Sets <synapse>.i

Parameters

cell: LFPy.Cell like object

collect_potential(*cell*)

Collect membrane potential of segment with synapse. Sets <synapse>.v

Parameters

cell: LFPy.Cell like object

set_spike_times(*sptimes=array([], dtype=float64)*)

Set the spike times explicitly using numpy arrays

Parameters

ndarray, dtype=float Sequence of synapse activation times

set_spike_times_w_netstim(*noise=1.0*, *start=0.0*, *number=1000.0*, *interval=10.0*, *seed=1234.0*)

Generate a train of pre-synaptic stimulus times by setting up the neuron NetStim object associated with this synapse

Parameters

noise: float in range [0, 1] Fractional randomness, from deterministic to intervals that drawn from negexp distribution (Poisson spiketimes).

start: float ms, (most likely) start time of first spike

number: int (average) number of spikes

interval: float ms, (mean) time between spikes

seed: float Random seed value

1.4.3 class StimIntElectrode

class LFPy.**StimIntElectrode**(*cell*, *idx*, *pptype='SEClamp'*, *record_current=False*, *record_potential=False*, ***kwargs*)

Bases: *LFPy.pointprocess.PointProcess*

Class for NEURON point processes representing electrode currents, such as VClamp, SEClamp and ICLamp.

Membrane currents will no longer sum to zero if these mechanisms are used, as the equivalent circuit is akin to a current input to the compartment from a far away extracellular location ("ground"), not immediately from the surface to the inside of the compartment as with transmembrane currents.

Refer to NEURON documentation @ neuron.yale.edu for keyword arguments or class documentation in Python issuing e.g.

help(neuron.h.VClamp)

Will insert pptype on cell-instance, pass the corresponding kwargs onto cell.set_point_process.

Parameters

cell: obj

***LFPy.Cell* or *LFPy.TemplateCell* instance to receive Stimulation** electrode input

idx: int Cell segment index where the stimulation electrode is placed

pptype: str Type of point process. Built-in examples: VClamp, SEClamp and ICLamp. Defaults to 'SEClamp'.

record_current: bool Decides if current is recorded

record_potential: bool switch for recording the potential on postsynaptic segment index

****kwargs** Additional arguments to be passed on to NEURON in *cell.set_point_process*

See also:

Synapse

Examples

```
>>> import pylab as pl
>>> pl.ion()
>>> import os
>>> import LFPy
>>> # define a list of different electrode implementations from NEURON
>>> pointprocesses = [
>>>     {
>>>         'idx': 0,
>>>         'record_current': True,
>>>         'pptype': 'IClamp',
>>>         'amp': 1,
>>>         'dur': 20,
>>>         'delay': 10,
>>>     },
>>>     {
>>>         'idx': 0,
>>>         'record_current': True,
>>>         'pptype': 'VClamp',
>>>         'amp': [-70, 0, -70],
>>>         'dur': [10, 20, 10],
>>>    },
>>>    {
>>>        'idx': 0,
>>>        'record_current': True,
>>>        'pptype': 'SEClamp',
>>>        'dur1': 10,
>>>        'amp1': -70,
>>>        'dur2': 20,
>>>        'amp2': 0,
>>>        'dur3': 10,
>>>        'amp3': -70,
>>>     },
>>> ]
>>> # create a cell instance for each electrode
>>> fix, axes = pl.subplots(2, 1, sharex=True)
>>> for pointprocess in pointprocesses:
>>>     cell = LFPy.Cell(morphology=os.path.join('examples',
>>>                                              'morphologies',
>>>                                              'L5_Mainen96_LFPy.hoc'),
>>>                      passive=True)
>>>     stimulus = LFPy.StimIntElectrode(cell, **pointprocess)
>>>     cell.simulate()
>>>     axes[0].plot(cell.tvec, stimulus.i, label=pointprocess['pptype'])
>>>     axes[0].legend(loc='best')
>>>     axes[0].set_title('Stimulus currents (nA)')
>>>     axes[1].plot(cell.tvec, cell.somav, label=pointprocess['pptype'])
>>>     axes[1].legend(loc='best')
>>>     axes[1].set_title('Somatic potential (mV)')
```

collect_current(*cell*)

Fetch electrode current. Sets Sets ``<stimintelectrode>.i

Parameters

cell: LFPy.Cell like object

`collect_potential`(*cell*)

Collect membrane potential of segment with PointProcess. Sets `<stimintelectrode>.v`

Parameters

cell: LFPy.Cell like object

1.5 Networks

1.5.1 class `Network`

`class LFPy.Network`(*dt=0.1*, *tstart=0.0*, *tstop=1000.0*, *v_init=- 65.0*, *celsius=6.3*, *OUTPUTPATH='example_parallel_network'*, *verbose=False*)

Bases: `object`

Network class, creating distributed populations of cells of type Cell and handling connections between cells in the respective populations.

Parameters

dt: float Simulation timestep size

tstart: float Start time of simulation

tstop: float End time of simulation

v_init: float Membrane potential set at first timestep across all cells

celsius: float Global control of temperature, affect channel kinetics. It will also be forced when creating the different Cell objects, as LFPy.Cell and LFPy.TemplateCell also accept the same keyword argument.

verbose: bool if True, print out misc. messages

`connect`(*pre, post, connectivity, syntype=ExpSyn(), synparams={'e': 0.0, 'tau': 2.0}, weightfun=<built-in method normal of numpy.random.mtrand.RandomState object>, weightargs={'loc': 0.1, 'scale': 0.01}, minweight=0, delayfun=<scipy.stats._continuous_distns.truncnorm_gen object>, delayargs={'a': 0.3, 'b': inf, 'loc': 2, 'scale': 0.2}, mindelay=None, multapsefun=<scipy.stats._continuous_distns.truncnorm_gen object>, multapseargs={'a': -3.0, 'b': 6.0, 'loc': 4, 'scale': 1}, syn_pos_args={'fun': [<scipy.stats._continuous_distns.norm_gen object>, <scipy.stats._continuous_distns.norm_gen object>], 'funargs': [{'loc': 0, 'scale': 100}, {'loc': 0, 'scale': 100}], 'funweights': [0.5, 0.5], 'section': ['soma', 'dend', 'apic'], 'z_max': 1000000.0, 'z_min': -1000000.0}, save_connections=False*)

Connect presynaptic cells to postsynaptic cells. Connections are drawn from presynaptic cells to postsynaptic cells, hence connectivity array must only be specified for postsynaptic units existing on this RANK.

Parameters

pre: str presynaptic population name

post: str postsynaptic population name

connectivity: ndarray / (scipy.sparse array) boolean connectivity matrix between pre and post.

syntype: hoc.HocObject reference to NEURON synapse mechanism, e.g., `neuron.h.ExpSyn`

synparams: dict dictionary of parameters for synapse mechanism, keys 'e', 'tau' etc.

weightfun: function function used to draw weights from a numpy.random distribution

weightargs: dict parameters passed to weightfun

minweight: float, minimum weight in units of nS

delayfun: function function used to draw delays from a subclass of scipy.stats.rv_continuous or numpy.random distribution

delayargs: dict parameters passed to `delayfun`

mindelay: float, minimum delay in multiples of dt. Ignored if `delayfun` is an inherited from `scipy.stats.rv_continuous`

multapsefun: function or None function reference, e.g., `scipy.stats.rv_continuous` used to draw a number of synapses for a cell-to-cell connection. If None, draw only one connection

multapseargs: dict arguments passed to multapsefun

syn_pos_args: dict arguments passed to inherited `LFPy.Cell` method `NetworkCell.get_rand_idx_area_and_distribution_norm` to find synapse locations.

save_connections: bool if True (default False), save instantiated connections to HDF5 file `Network.OUTPUTPATH/synapse_connections.h5` as dataset `<pre>:<post>` using a structured ndarray with dtype

```
[('gid_pre'), ('gid', 'i8'), ('weight', 'f8'), ('delay', 'f8'),
('sec', 'U64'), ('sec.x', 'f8'),
('x', 'f8'), ('y', 'f8'), ('z', 'f8')],
```

where `gid_pre` is presynapic cell id, `gid` is postsynaptic cell id, `weight` connection weight, `delay` connection delay, `sec` section name, `sec.x` relative location on section, and `x`, `y`, `z` the corresponding midpoint coordinates of the target compartment.

Returns

list Length 2 list with ndarrays [conncount, syncount] with numbers of instantiated connections and synapses.

Raises

DeprecationWarning if `delayfun` is not a subclass of `scipy.stats.rv_continuous`

`create_population`(*CWD=None*, *CELLPATH=None*, *Cell=<class 'LFPy.network.NetworkCell'>*, *POP_SIZE=4*, *name='L5PC'*, *cell_args=None*, *pop_args=None*, *rotation_args=None*)

Create and append a distributed POP_SIZE-sized population of cells of type Cell with the corresponding name. Cell-object references, gids on this RANK, population size POP_SIZE and names will be added to the lists Network.gids, Network.cells, Network.sizes and Network.names, respectively

Parameters

CWD: path Current working directory

CELLPATH: path Relative path from CWD to source files for cell model (morphology, hoc routines etc.)

Cell: class class defining a Cell-like object, see class NetworkCell

POP_SIZE: int number of cells in population

name: str population name reference

cell_args: dict keys and values for Cell object

pop_args: dict keys and values for Network.draw_rand_pos assigning cell positions

rotation_arg: dict default cell rotations around x and y axis on the form { 'x': np.pi/2, 'y': 0 }. Can only have the keys 'x' and 'y'. Cells are randomly rotated around z-axis using the Cell.set_rotation method.

`enable_extracellular_stimulation`(*electrode*, *t_ext=None*, *n=1*, *model='inf'*)

Enable extracellular stimulation with NEURON's *extracellular* mechanism. Extracellular potentials are computed from electrode currents using the point-source approximation. If `model` is `'inf'` (default), potentials are computed as (r_i is the position of a compartment i, r_n is the position of an electrode n, σ is the conductivity of the medium):

$$V_e(r_i) = \sum_n \frac{I_n}{4\pi\sigma|r_i - r_n|}$$

If `model` is `'semi'`, the method of images is used:

$$V_e(r_i) = \sum_n \frac{I_n}{2\pi\sigma|r_i - r_n|}$$

Parameters

electrode: RecExtElectrode Electrode object with stimulating currents

t_ext: np.ndarray or list Time in ms corresponding to step changes in the provided currents. If None, currents are assumed to have the same time steps as the NEURON simulation.

n: int Points per electrode for spatial averaging

model: str `'inf'` or `'semi'`. If `'inf'` the medium is assumed to be infinite and homogeneous. If `'semi'`, the method of images is used.

Returns

v_ext: dict of np.ndarrays Computed extracellular potentials at cell mid points for each cell of the network's populations. Formatted as `v_ext = {'pop1': np.ndarray[cell, cell_seg,t_ext]}`

`get_connectivity_rand`(*pre='L5PC'*, *post='L5PC'*, *connprob=0.2*)

Dummy function creating a (boolean) cell to cell connectivity matrix between pre and postsynaptic populations.

Connections are drawn randomly between presynaptic cell gids in population 'pre' and postsynaptic cell gids in 'post' on this RANK with a fixed connection probability. self-connections are disabled if presynaptic and postsynaptic populations are the same.

Parameters

pre: str presynaptic population name

post: str postsynaptic population name

connprob: float in [0, 1] connection probability, connections are drawn on random

Returns

ndarray, dtype bool n_pre x n_post array of connections between n_pre presynaptic neurons and n_post postsynaptic neurons on this RANK. Entries with True denotes a connection.

simulate(*probes=None*, *rec_imem=False*, *rec_vmem=False*, *rec_ipas=False*, *rec_icap=False*, *rec_isyn=False*, *rec_vmemsyn=False*, *rec_istim=False*, *rec_pop_contributions=False*, *rec_variables=[]*, *variable_dt=False*, *atol=0.001*, *to_memory=True*, *to_file=False*, *file_name='OUTPUT.h5'*, ***kwargs*)

This is the main function running the simulation of the network model.

Parameters

probes: list of [obj:, optional] None or list of LFPykit.RecExtElectrode like object instances that each have a public method *get_transformation_matrix* returning a matrix that linearly maps each compartments' transmembrane current to corresponding measurement as

$$\mathbf{P} = \mathbf{MI}$$

rec_imem: bool If true, segment membrane currents will be recorded If no electrode argument is given, it is necessary to set rec_imem=True in order to calculate LFP later on. Units of (nA).

rec_vmem: bool record segment membrane voltages (mV)

rec_ipas: bool record passive segment membrane currents (nA)

rec_icap: bool record capacitive segment membrane currents (nA)

rec_isyn: bool record synaptic currents of from Synapse class (nA)

rec_vmemsyn: bool record membrane voltage of segments with Synapse (mV)

rec_istim: bool record currents of StimIntraElectrode (nA)

rec_pop_contributions: bool If True, compute and return single-population contributions to the extracellular potential during simulation time

rec_variables: list of str variables to record, i.e arg=[‘cai’,]

variable_dt: boolean use variable timestep in NEURON. Can not be combimed with *to_file*

atol: float absolute tolerance used with NEURON variable timestep

to_memory: bool Simulate to memory. Only valid with *probes=[<probe>, ...]*, which store measurements to -> <probe>.data

to_file: bool only valid with *probes=[<probe>, ...]*, saves measurement in hdf5 file format.

file_name: str If to_file is True, file which measurements will be written to. The file format is HDF5, default is “OUTPUT.h5”, put in folder Network.OUTPUTPATH

****kwargs: keyword argument dict values passed along to function** *__run_simulation_with_probes()*, containing some or all of the boolean flags: *use_ipas*, *use_icap*, *use_isyn* (defaulting to *False*).

Returns

events Dictionary with keys *times* and *gids*, where values are ndarrays with detected spikes and global neuron identifiers

Raises

Exception if *CVode().use_fast_imem()* method not found

AssertionError if rec_pop_contributions==True and probes==None

1.5.2 class NetworkPopulation

class LFPy.**NetworkPopulation**(*CWD=None*, *CELLPATH=None*, *first_gid=0*, *Cell=<class 'LFPy.network.NetworkCell'>*, *POP_SIZE=4*, *name='L5PC'*, *cell_args=None*, *pop_args=None*, *rotation_args=None*, *OUTPUTPATH='example_parallel_network'*)

Bases: object

NetworkPopulation class representing a group of Cell objects distributed across RANKs.

Parameters

CWD: path or None Current working directory

CELLPATH: path or None Relative path from CWD to source files for cell model (morphology, hoc routines etc.)

first_gid: int The global identifier of the first cell created in this population instance. The first_gid in the first population created should be 0 and cannot exist in previously created NetworkPopulation instances

Cell: class class defining a Cell object, see class NetworkCell above

POP_SIZE: int number of cells in population

name: str population name reference

cell_args: dict keys and values for Cell object

pop_args: dict keys and values for Network.draw_rand_pos assigning cell positions

rotation_arg: dict default cell rotations around x and y axis on the form { 'x': np.pi/2, 'y': 0 }. Can only have the keys 'x' and 'y'. Cells are randomly rotated around z-axis using the Cell.set_rotation() method.

OUTPUTPATH: str path to output file destination

draw_rand_pos(*POP_SIZE*, *radius*, *loc*, *scale*, *cap=None*)

Draw some random location for POP_SIZE cells within radius radius, at mean depth loc and standard deviation scale.

Returned argument is a list of dicts [{'x', 'y', 'z'},].

Parameters

POP_SIZE: int Population size

radius: float Radius of population.

loc: float expected mean depth of somas of population.

scale: float expected standard deviation of depth of somas of population.

cap: None, float or length to list of floats if float, cap distribution between [loc-cap, loc+cap), if list, cap distribution between [loc-cap[0], loc+cap[1]]

Returns

soma_pos: list List of dicts of len POP_SIZE where dict have keys x, y, z specifying xyz-coordinates of cell at list entry *i*.

1.6 Forward models

1.6.1 class CurrentDipoleMoment

class LFPy.**CurrentDipoleMoment**(*cell*)

Bases: lfpykit.models.LinearModel

LinearModel subclass that defines a 2D linear response matrix $\mathbf{M}$ between transmembrane current array $\mathbf{I}$ (nA) of a multicompartment neuron model and the corresponding current dipole moment $\mathbf{P}$ (nA µm) [1] as

$$\mathbf{P} = \mathbf{MI}$$

The current $\mathbf{I}$ is an ndarray of shape (n_seg, n_tsteps) with unit (nA), and the rows of $\mathbf{P}$ represent the x-, y- and z-components of the current diple moment for every time step.

The current dipole moment can be used to compute distal measures of neural activity such as the EEG and MEG using lfpykit.eegmegcalc.FourSphereVolumeConductor or lfpykit.eegmegcalc.MEG, respectively

Parameters

cell: object CellGeometry instance or similar.

See also:

LinearModel

eegmegcalc.FourSphereVolumeConductor

eegmegcalc.MEG

eegmegcalc.NYHeadModel

References

[1]

Examples

Compute the current dipole moment of a 3-compartment neuron model:

```
>>> import numpy as np
>>> from lfpykit import CellGeometry, CurrentDipoleMoment
>>> n_seg = 3
>>> cell = CellGeometry(x=np.array([[0.]*2]*n_seg),
                        y=np.array([[0.]*2]*n_seg),
                        z=np.array([[1.*x, 1.*(x+1)]
                                    for x in range(n_seg)]),
                        d=np.array([1.]*n_seg))
>>> cdm = CurrentDipoleMoment(cell)
>>> M = cdm.get_transformation_matrix()
>>> imem = np.array([[-1., 1.],
                     [0., 0.],
                     [1., -1.]])
>>> P = M@imem
>>> P
```

(continues on next page)

(continued from previous page)

```
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 2., -2.]])
```

get_transformation_matrix()

Get linear response matrix

Returns

response_matrix: ndarray shape (3, n_seg) ndarray

Raises

AttributeError if `cell is None`

1.6.2 class PointSourcePotential

class LFPy.**PointSourcePotential**(*cell*, *x*, *y*, *z*, *sigma=0.3*)

Bases: `lfpykit.models.LinearModel`

`LinearModel` subclass that defines a 2D linear response matrix $\mathbf{M}$ between transmembrane current array $\mathbf{I}$ (nA) of a multicompartment neuron model and the corresponding extracellular electric potential $\mathbf{V}_{ex}$ (mV) as

$$\mathbf{V}_{ex} = \mathbf{M}\mathbf{I}$$

The current $\mathbf{I}$ is an ndarray of shape (n_seg, n_tsteps) with unit (nA), and each row indexed by j of $\mathbf{V}_{ex}$ represents the electric potential at each measurement site for every time step.

The elements of $\mathbf{M}$ are computed as

$$M_{ji} = 1/(4\pi\sigma|\mathbf{r}_i - \mathbf{r}_j|)$$

where σ is the electric conductivity of the extracellular medium, $\mathbf{r}_i$ the midpoint coordinate of segment i and $\mathbf{r}_j$ the coordinate of measurement site j [1], [2].

Assumptions:

- the extracellular conductivity σ is infinite, homogeneous, frequency independent (linear) and isotropic.
- each segment is treated as a point source located at the midpoint between its start and end point coordinate.
- each measurement site $\mathbf{r}_j = (x_j, y_j, z_j)$ is treated as a point.
- $|\mathbf{r}_i - \mathbf{r}_j| >= d_i/2$, where d_i is the segment diameter.

Parameters

cell: object `CellGeometry` instance or similar.

x: ndarray of floats x-position of measurement sites (µm)

y: ndarray of floats y-position of measurement sites (µm)

z: ndarray of floats z-position of measurement sites (µm)

sigma: float > 0 scalar extracellular conductivity (S/m)

See also:

`LinearModel`

LineSourcePotential

RecExtElectrode

References

[1], [2]

Examples

Compute the current dipole moment of a 3-compartment neuron model:

```
>>> import numpy as np
>>> from lfpykit import CellGeometry, PointSourcePotential
>>> n_seg = 3
>>> cell = CellGeometry(x=np.array([[0.]*2]*n_seg),
                        y=np.array([[0.]*2]*n_seg),
                        z=np.array([[10.*x, 10.*(x+1)]
                                    for x in range(n_seg)]),
                        d=np.array([1.]*n_seg))
>>> psp = PointSourcePotential(cell,
                               x=np.ones(10)*10,
                               y=np.zeros(10),
                               z=np.arange(10)*10,
                               sigma=0.3)
>>> M = psp.get_transformation_matrix()
>>> imem = np.array([[-1., 1.],
                     [0., 0.],
                     [1., -1.]])
>>> V_ex = M @ imem
>>> V_ex
array([[-0.01387397,  0.01387397],
       [-0.00901154,  0.00901154],
       [ 0.00901154, -0.00901154],
       [ 0.01387397, -0.01387397],
       [ 0.00742668, -0.00742668],
       [ 0.00409718, -0.00409718],
       [ 0.00254212, -0.00254212],
       [ 0.00172082, -0.00172082],
       [ 0.00123933, -0.00123933],
       [ 0.00093413, -0.00093413]])
```

get_transformation_matrix()

Get linear response matrix

Returns

response_matrix: ndarray shape (n_coords, n_seg) ndarray

Raises

AttributeError if `cell is None`

1.6.3 class LineSourcePotential

class LFPy.**LineSourcePotential**(*cell*, *x*, *y*, *z*, *sigma=0.3*)

Bases: lfpykit.models.LinearModel

LinearModel subclass that defines a 2D linear response matrix $\mathbf{M}$ between transmembrane current array $\mathbf{I}$ (nA) of a multicompartment neuron model and the corresponding extracellular electric potential $\mathbf{V}_{ex}$ (mV) as

$$\mathbf{V}_{ex} = \mathbf{M}\mathbf{I}$$

The current $\mathbf{I}$ is an ndarray of shape (n_seg, n_tsteps) with unit (nA), and each row indexed by j of $\mathbf{V}_{ex}$ represents the electric potential at each measurement site for every time step.

The elements of $\mathbf{M}$ are computed as

$$M_{ji} = \frac{1}{4\pi\sigma L_i}\log\left|\frac{\sqrt{h_{ji}^2 + r_{ji}^2} - h_{ji}}{\sqrt{l_{ji}^2 + r_{ji}^2} - l_{ji}}\right|$$

Segment length is denoted L_i, perpendicular distance from the electrode point contact to the axis of the line segment is denoted r_{ji}, longitudinal distance measured from the start of the segment is denoted h_{ji}, and longitudinal distance from the other end of the segment is denoted $l_{ji} = L_i + h_{ji}$ [1], [2].

Assumptions:

- the extracellular conductivity σ is infinite, homogeneous, frequency independent (linear) and isotropic
- each segment is treated as a straight line source with homogeneous current density between its start and end point coordinate
- each measurement site $\mathbf{r}_j = (x_j, y_j, z_j)$ is treated as a point
- The minimum distance to a line source is set equal to segment radius.

Parameters

cell: object CellGeometry instance or similar.

x: ndarray of floats x-position of measurement sites (µm)

y: ndarray of floats y-position of measurement sites (µm)

z: ndarray of floats z-position of measurement sites (µm)

sigma: float > 0 scalar extracellular conductivity (S/m)

See also:

LinearModel

PointSourcePotential

RecExtElectrode

References

[1], [2]

Examples

Compute the current dipole moment of a 3-compartment neuron model:

```
>>> import numpy as np
>>> from lfpykit import CellGeometry, LineSourcePotential
>>> n_seg = 3
>>> cell = CellGeometry(x=np.array([[0.]*2]*n_seg),
                        y=np.array([[0.]*2]*n_seg),
                        z=np.array([[10.*x, 10.*(x+1)]
                                    for x in range(n_seg)]),
                        d=np.array([1.]*n_seg))
>>> lsp = LineSourcePotential(cell,
                              x=np.ones(10)*10,
                              y=np.zeros(10),
                              z=np.arange(10)*10,
                              sigma=0.3)
>>> M = lsp.get_transformation_matrix()
>>> imem = np.array([[-1., 1.],
                     [0., 0.],
                     [1., -1.]])
>>> V_ex = M @ imem
>>> V_ex
array([[-0.01343699,  0.01343699],
       [-0.0084647 ,  0.0084647 ],
       [ 0.0084647 , -0.0084647 ],
       [ 0.01343699, -0.01343699],
       [ 0.00758627, -0.00758627],
       [ 0.00416681, -0.00416681],
       [ 0.002571  , -0.002571  ],
       [ 0.00173439, -0.00173439],
       [ 0.00124645, -0.00124645],
       [ 0.0009382 , -0.0009382 ]])
```

get_transformation_matrix()

Get linear response matrix

Returns

response_matrix: ndarray shape (n_coords, n_seg) ndarray

Raises

AttributeError if `cell is None`

1.6.4 class RecExtElectrode

class LFPy.**RecExtElectrode**(*cell*, *sigma=0.3*, *probe=None*, *x=None*, *y=None*, *z=None*, *N=None*, *r=None*, *n=None*, *contact_shape='circle'*, *method='linesource'*, *verbose=False*, *seedvalue=None*, ***kwargs*)

Bases: lfpykit.models.LinearModel

class RecExtElectrode

Main class that represents an extracellular electric recording devices such as a laminar probe.

This class is a LinearModel subclass that defines a 2D linear response matrix $\mathbf{M}$ between transmembrane current array $\mathbf{I}$ (nA) of a multicompartment neuron model and the corresponding extracellular electric potential $\mathbf{V}_{ex}$ (mV) as

$$\mathbf{V}_{ex} = \mathbf{M}\mathbf{I}$$

The current $\mathbf{I}$ is an ndarray of shape (n_seg, n_tsteps) with unit (nA), and each row indexed by j of $\mathbf{V}_{ex}$ represents the electric potential at each measurement site for every time step.

The class differ from PointSourcePotential and LineSourcePotential by:

- supporting anisotropic volume conductors [1]
- supporting probe geometry specifications using MEAutility (https://meautility.readthedocs.io/en/latest/, https://github.com/alejoe91/MEAutility).
- supporting electrode contact points with finite extents [2], [3]
- switching between point- and linesources, and a combined method that assumes that the root element at segment index 0 is spherical.

Parameters

cell: object CellGeometry instance or similar.

sigma: float or list/ndarray of floats extracellular conductivity in units of (S/m). A scalar value implies an isotropic extracellular conductivity. If a length 3 list or array of floats is provided, these values corresponds to an anisotropic conductor with conductivities $[\sigma_x, \sigma_y, \sigma_z]$.

probe: MEAutility MEA object or None MEAutility probe object

x, y, z: ndarray coordinates or same length arrays of coordinates in units of (µm).

N: None or list of lists Normal vectors [x, y, z] of each circular electrode contact surface, default None

r: float radius of each contact surface, default None (µm)

n: int if N is not None and r > 0, the number of discrete points used to compute the n-point average potential on each circular contact point.

contact_shape: str 'circle'/'square' (default 'circle') defines the contact point shape If 'circle' r is the radius, if 'square' r is the side length

method: str switch between the assumption of 'linesource', 'pointsource', 'root_as_point' to represent each compartment when computing extracellular potentials

verbose: bool Flag for verbose output, i.e., print more information

seedvalue: int random seed when finding random position on contact with r > 0

****kwargs:** Additional keyword arguments parsed to *RecExtElectrode.lfp_method()* which is determined by *method* parameter.

See also:

`LinearModel`

PointSourcePotential

LineSourcePotential

References

[1], [2], [3]

Examples

Mock cell geometry and transmembrane currents:

```
>>> import numpy as np
>>> from lfpykit import CellGeometry, RecExtElectrode
>>> # cell geometry with three segments (µm)
>>> cell = CellGeometry(x=np.array([[0, 0], [0, 0], [0, 0]]),
>>>                     y=np.array([[0, 0], [0, 0], [0, 0]]),
>>>                     z=np.array([[0, 10], [10, 20], [20, 30]]),
>>>                     d=np.array([1, 1, 1]))
>>> # transmembrane currents, three time steps (nA)
>>> I_m = np.array([[0., -1., 1.], [-1., 1., 0.], [1., 0., -1.]])
>>> # electrode locations (µm)
>>> r = np.array([[28.24653166, 8.97563241, 18.9492774, 3.47296614,
>>>                1.20517729, 9.59849603, 21.91956616, 29.84686727,
>>>                4.41045505, 3.61146625],
>>>               [24.4954352, 24.04977922, 22.41262238, 10.09702942,
>>>                3.28610789, 23.50277637, 8.14044367, 4.46909208,
>>>                10.93270117, 24.94698813],
>>>               [19.16644585, 15.20196335, 18.08924828, 24.22864702,
>>>                5.85216751, 14.8231048, 24.72666694, 17.77573431,
>>>                29.34508292, 9.28381892]])
>>> # instantiate electrode, get linear response matrix
>>> el = RecExtElectrode(cell=cell, x=r[0, ], y=r[1, ], z=r[2, ],
>>>                      sigma=0.3,
>>>                      method='pointsource')
>>> M = el.get_transformation_matrix()
>>> # compute extracellular potential
>>> M @ I_m
array([[-4.11657148e-05,  4.16621950e-04, -3.75456235e-04],
       [-6.79014892e-04,  7.30256301e-04, -5.12414088e-05],
       [-1.90930536e-04,  7.34007655e-04, -5.43077119e-04],
       [ 5.98270144e-03,  6.73490846e-03, -1.27176099e-02],
       [-1.34547752e-02, -4.65520036e-02,  6.00067788e-02],
       [-7.49957880e-04,  7.03763787e-04,  4.61940938e-05],
       [ 8.69330232e-04,  1.80346156e-03, -2.67279180e-03],
       [-2.04546513e-04,  6.58419628e-04, -4.53873115e-04],
       [ 6.82640209e-03,  4.47953560e-03, -1.13059377e-02],
       [-1.33289553e-03, -1.11818140e-04,  1.44471367e-03]])
```

Compute extracellular potentials after simulating and storage of transmembrane currents with the LFPy.Cell class:

```
>>> import os
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import LFPy
>>> from lfpykit import CellGeometry, RecExtElectrode
>>>
>>> cellParameters = {
>>>     'morphology': os.path.join(LFPy.__path__[0], 'test',
>>>                                'ball_and_sticks.hoc'),
>>>     'v_init': -65,                          # initial voltage
>>>     'cm': 1.0,                              # membrane capacitance
>>>     'Ra': 150,                              # axial resistivity
>>>     'passive': True,                        # insert passive channels
>>>     'passive_parameters': {"g_pas":1./3E4,
>>>                            "e_pas":-65}, # passive params
>>>     'dt': 2**-4,                           # simulation time res
>>>     'tstart': 0.,                          # start t of simulation
>>>     'tstop': 50.,                          # end t of simulation
>>> }
>>> cell = LFPy.Cell(**cellParameters)
>>>
>>> synapseParameters = {
>>>     'idx': cell.get_closest_idx(x=0, y=0, z=800), # segment
>>>     'e': 0,                                 # reversal potential
>>>     'syntype': 'ExpSyn',                    # synapse type
>>>     'tau': 2,                               # syn. time constant
>>>     'weight': 0.01,                         # syn. weight
>>>     'record_current': True                  # syn. current record
>>> }
>>> synapse = LFPy.Synapse(cell, **synapseParameters)
>>> synapse.set_spike_times(np.array([10., 15., 20., 25.]))
>>>
>>> cell.simulate(rec_imem=True)
>>>
>>> N = np.empty((16, 3))
>>> for i in range(N.shape[0]): N[i,] = [1, 0, 0] # normal vectors
>>> electrodeParameters = {          # parameters for RecExtElectrode class
>>>     'sigma': 0.3,               # Extracellular potential
>>>     'x': np.zeros(16)+25,       # Coordinates of electrode contacts
>>>     'y': np.zeros(16),
>>>     'z': np.linspace(-500,1000,16),
>>>     'n': 20,
>>>     'r': 10,
>>>     'N': N,
>>> }
>>> electrode = RecExtElectrode(cell, **electrodeParameters)
>>> M = electrode.get_transformation_matrix()
>>> V_ex = M @ cell.imem
>>> plt.matshow(V_ex)
>>> plt.colorbar()
```

(continues on next page)

(continued from previous page)

```
>>> plt.axis('tight')
>>> plt.show()
```

Compute extracellular potentials during simulation:

```
>>> import os
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import LFPy
>>> from lfpykit import CellGeometry, RecExtElectrode
>>>
>>> cellParameters = {
>>>     'morphology': os.path.join(LFPy.__path__[0], 'test',
>>>                                'ball_and_sticks.hoc'),
>>>     'v_init': -65,                          # initial voltage
>>>     'cm': 1.0,                              # membrane capacitance
>>>     'Ra': 150,                              # axial resistivity
>>>     'passive': True,                        # insert passive channels
>>>     'passive_parameters': {"g_pas":1./3E4,
>>>                            "e_pas":-65}, # passive params
>>>     'dt': 2**-4,                           # simulation time res
>>>     'tstart': 0.,                          # start t of simulation
>>>     'tstop': 50.,                          # end t of simulation
>>> }
>>> cell = LFPy.Cell(**cellParameters)
>>>
>>> synapseParameters = {
>>>     'idx': cell.get_closest_idx(x=0, y=0, z=800), # compartment
>>>     'e': 0,                                 # reversal potential
>>>     'syntype': 'ExpSyn',                    # synapse type
>>>     'tau': 2,                               # syn. time constant
>>>     'weight': 0.01,                         # syn. weight
>>>     'record_current': True                  # syn. current record
>>> }
>>> synapse = LFPy.Synapse(cell, **synapseParameters)
>>> synapse.set_spike_times(np.array([10., 15., 20., 25.]))
>>>
>>> N = np.empty((16, 3))
>>> for i in range(N.shape[0]): N[i,] = [1, 0, 0] #normal vec. of contacts
>>> electrodeParameters = {          # parameters for RecExtElectrode class
>>>     'sigma': 0.3,               # Extracellular potential
>>>     'x': np.zeros(16)+25,       # Coordinates of electrode contacts
>>>     'y': np.zeros(16),
>>>     'z': np.linspace(-500,1000,16),
>>>     'n': 20,
>>>     'r': 10,
>>>     'N': N,
>>> }
>>> electrode = RecExtElectrode(cell, **electrodeParameters)
>>> cell.simulate(probes=[electrode])
>>> plt.matshow(electrode.data)
>>> plt.colorbar()
```

(continues on next page)

(continued from previous page)

```
>>> plt.axis('tight')
>>> plt.show()
```

Use MEAutility to to handle probes

```
>>> import os
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import MEAutility as mu
>>> import LFPy
>>> from lfpykit import CellGeometry, RecExtElectrode
>>>
>>> cellParameters = {
>>>     'morphology': os.path.join(LFPy.__path__[0], 'test',
>>>                                'ball_and_sticks.hoc'),
>>>     'v_init': -65,                          # initial voltage
>>>     'cm': 1.0,                              # membrane capacitance
>>>     'Ra': 150,                              # axial resistivity
>>>     'passive': True,                        # insert passive channels
>>>     'passive_parameters': {"g_pas":1./3E4,
>>>                            "e_pas":-65}, # passive params
>>>     'dt': 2**-4,                           # simulation time res
>>>     'tstart': 0.,                          # start t of simulation
>>>     'tstop': 50.,                          # end t of simulation
>>> }
>>> cell = LFPy.Cell(**cellParameters)
>>>
>>> synapseParameters = {
>>>     'idx': cell.get_closest_idx(x=0, y=0, z=800), # compartment
>>>     'e': 0,                                 # reversal potential
>>>     'syntype': 'ExpSyn',                    # synapse type
>>>     'tau': 2,                               # syn. time constant
>>>     'weight': 0.01,                         # syn. weight
>>>     'record_current': True                  # syn. current record
>>> }
>>> synapse = LFPy.Synapse(cell, **synapseParameters)
>>> synapse.set_spike_times(np.array([10., 15., 20., 25.]))
>>>
>>> cell.simulate(rec_imem=True)
>>>
>>> probe = mu.return_mea('Neuropixels-128')
>>> electrode = RecExtElectrode(cell, probe=probe)
>>> V_ex = electrode.get_transformation_matrix() @ cell.imem
>>> mu.plot_mea_recording(V_ex, probe)
>>> plt.axis('tight')
>>> plt.show()
```

get_transformation_matrix()

Get linear response matrix

Returns

response_matrix: ndarray shape (n_contacts, n_seg) ndarray

Raises

AttributeError if `cell is None`

1.6.5 class RecMEAElectrode

class LFPy.**RecMEAElectrode**(*cell*, *sigma_T=0.3*, *sigma_S=1.5*, *sigma_G=0.0*, *h=300.0*, *z_shift=0.0*, *steps=20*, *probe=None*, *x=array([0])*, *y=array([0])*, *z=array([0])*, *N=None*, *r=None*, *n=None*, *method='linesource'*, *verbose=False*, *seedvalue=None*, *squeeze_cell_factor=None*, ***kwargs*)

Bases: *lfpykit.models.RecExtElectrode*

class RecMEAElectrode

Electrode class that represents an extracellular in vitro slice recording as a Microelectrode Array (MEA). Inherits RecExtElectrode class

Illustration:

```
          Above neural tissue (Saline) -> sigma_S
<-------------------------------------------------> z = z_shift + h

          Neural Tissue -> sigma_T

               o -> source_pos = [x',y',z']

<-----------*-----------------------------------> z = z_shift + 0
             \-> elec_pos = [x,y,z]

          Below neural tissue (MEA Glass plate) -> sigma_G
```

For further details, see reference [1].

Parameters

cell: object GeometryCell instance or similar.

sigma_T: float extracellular conductivity of neural tissue in unit (S/m)

sigma_S: float conductivity of saline bath that the neural slice is immersed in [1.5] (S/m)

sigma_G: float conductivity of MEA glass electrode plate. Most commonly assumed non-conducting [0.0] (S/m)

h: float, int Thickness in um of neural tissue layer containing current the current sources (i.e., in vitro slice or cortex)

z_shift: float, int Height in um of neural tissue layer bottom. If e.g., top of neural tissue layer should be z=0, use z_shift=-h. Defaults to z_shift = 0, so that the neural tissue layer extends from z=0 to z=h.

squeeze_cell_factor: float or None Factor to squeeze the cell in the z-direction. This is needed for large cells that are thicker than the slice, since no part of the cell is allowed to be outside the slice. The squeeze is done after the neural simulation, and therefore does not affect neuronal simulation, only calculation of extracellular potentials.

probe: MEAutility MEA object or None MEAutility probe object

x, y, z: np.ndarray coordinates or arrays of coordinates in units of (um). Must be same length

N: None or list of lists Normal vectors [x, y, z] of each circular electrode contact surface, default None

r: float radius of each contact surface, default None

n: int if N is not None and r > 0, the number of discrete points used to compute the n-point average potential on each circular contact point.

contact_shape: str 'circle'/'square' (default 'circle') defines the contact point shape If 'circle' r is the radius, if 'square' r is the side length

method: str switch between the assumption of 'linesource', 'pointsource', 'root_as_point' to represent each compartment when computing extracellular potentials

verbose: bool Flag for verbose output, i.e., print more information

seedvalue: int random seed when finding random position on contact with r > 0

See also:

`LinearModel`

PointSourcePotential

LineSourcePotential

RecExtElectrode

References

[1]

Examples

Mock cell geometry and transmembrane currents:

```
>>> import numpy as np
>>> from lfpykit import CellGeometry, RecMEAElectrode
>>> # cell geometry with four segments (µm)
>>> cell = CellGeometry(
>>>     x=np.array([[0, 10], [10, 20], [20, 30], [30, 40]]),
>>>     y=np.array([[0, 0], [0, 0], [0, 0], [0, 0]]),
>>>     z=np.array([[0, 0], [0, 0], [0, 0], [0, 0]]) + 10,
>>>     d=np.array([1, 1, 1, 1]))
>>> # transmembrane currents, three time steps (nA)
>>> I_m = np.array([[0.25, -1., 1.],
>>>                 [-1., 1., -0.25],
>>>                 [1., -0.25, -1.],
>>>                 [-0.25, 0.25, 0.25]])
>>> # electrode locations (µm)
>>> r = np.stack([np.arange(10)*4 + 2, np.zeros(10), np.zeros(10)])
>>> # instantiate electrode, get linear response matrix
>>> el = RecMEAElectrode(cell=cell,
>>>                      sigma_T=0.3, sigma_S=1.5, sigma_G=0.0,
>>>                      x=r[0, ], y=r[1, ], z=r[2, ],
>>>                      method='pointsource')
>>> M = el.get_transformation_matrix()
```

(continues on next page)

(continued from previous page)

```
>>> # compute extracellular potential
>>> M @ I_m
array([[-0.00233572, -0.01990957,  0.02542055],
       [-0.00585075, -0.01520865,  0.02254483],
       [-0.01108601, -0.00243107,  0.01108601],
       [-0.01294584,  0.01013595, -0.00374823],
       [-0.00599067,  0.01432711, -0.01709416],
       [ 0.00599067,  0.01194602, -0.0266944 ],
       [ 0.01294584,  0.00953841, -0.02904238],
       [ 0.01108601,  0.00972426, -0.02324134],
       [ 0.00585075,  0.01075236, -0.01511768],
       [ 0.00233572,  0.01038382, -0.00954429]])
```

See also <LFPy>/examples/example_MEA.py

```
>>> import os
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import LFPy
>>> from lfpykit import CellGeometry, RecMEAElectrode
>>>
>>> cellParameters = {
>>>     'morphology': os.path.join(LFPy.__path__[0], 'test',
>>>                                'ball_and_sticks.hoc'),
>>>     'v_init': -65,                          # initial voltage
>>>     'cm': 1.0,                              # membrane capacitance
>>>     'Ra': 150,                              # axial resistivity
>>>     'passive': True,                        # insert passive channels
>>>     'passive_parameters': {"g_pas":1./3E4,
>>>                            "e_pas":-65}, # passive params
>>>     'dt': 2**-4,                            # simulation time res
>>>     'tstart': 0.,                          # start t of simulation
>>>     'tstop': 50.,                          # end t of simulation
>>> }
>>> cell = LFPy.Cell(**cellParameters)
>>> cell.set_rotation(x=np.pi/2, z=np.pi/2)
>>> cell.set_pos(z=100)
>>> synapseParameters = {
>>>     'idx': cell.get_closest_idx(x=800, y=0, z=100), # segment
>>>     'e': 0,                                 # reversal potential
>>>     'syntype': 'ExpSyn',                    # synapse type
>>>     'tau': 2,                               # syn. time constant
>>>     'weight': 0.01,                        # syn. weight
>>>     'record_current': True                  # syn. current record
>>> }
>>> synapse = LFPy.Synapse(cell, **synapseParameters)
>>> synapse.set_spike_times(np.array([10., 15., 20., 25.]))
>>>
>>> MEA_electrode_parameters = {
>>>     'sigma_T': 0.3,      # extracellular conductivity
>>>     'sigma_G': 0.0,      # MEA glass electrode plate conductivity
>>>     'sigma_S': 1.5,      # Saline bath conductivity
```

(continues on next page)

(continued from previous page)

```
>>>     'x': np.linspace(0, 1200, 16),  # 1d vector of positions
>>>     'y': np.zeros(16),
>>>     'z': np.zeros(16),
>>>     "method": "pointsource",
>>>     "h": 300,
>>>     "squeeze_cell_factor": 0.5,
>>> }
>>> cell.simulate(rec_imem=True)
>>>
>>> MEA = RecMEAElectrode(cell, **MEA_electrode_parameters)
>>> V_ext = MEA.get_transformation_matrix() @ lfpy_cell.imem
>>>
>>> plt.matshow(V_ext)
>>> plt.colorbar()
>>> plt.axis('tight')
>>> plt.show()
```

distort_cell_geometry(*axis='z'*, *nu=0.0*)

Distorts cellular morphology with a relative squeeze_cell_factor along a chosen axis preserving Poisson's ratio. A ratio nu=0.5 assumes uncompressible and isotropic media that embeds the cell. A ratio nu=0 will only affect geometry along the chosen axis. A ratio nu=-1 will isometrically scale the neuron geometry along each axis. This method does not affect the underlying cable properties of the cell, only predictions of extracellular measurements (by affecting the relative locations of sources representing the compartments).

Parameters

axis: str which axis to apply compression/stretching. Default is "z".

nu: float Poisson's ratio. Ratio between axial and transversal compression/stretching. Default is 0.

get_transformation_matrix()

Get linear response matrix

Returns

response_matrix: ndarray shape (n_contacts, n_seg) ndarray

Raises

AttributeError if `cell is None`

1.6.6 class OneSphereVolumeConductor

class LFPy.**OneSphereVolumeConductor**(*cell*, *r*, *R=10000.0*, *sigma_i=0.3*, *sigma_o=0.03*)

Bases: `lfpykit.models.LinearModel`

Computes extracellular potentials within and outside a spherical volume- conductor model that assumes homogeneous, isotropic, linear (frequency independent) conductivity in and outside the sphere with a radius R. The conductivity in and outside the sphere must be greater than 0, and the current source(s) must be located within the radius R.

The implementation is based on the description of electric potentials of point charge in an dielectric sphere embedded in dielectric media [1], which is mathematically equivalent to a current source in conductive media.

This class is a `LinearModel` subclass that defines a 2D linear response matrix $\mathbf{M}$ between transmembrane current array $\mathbf{I}$ (nA) of a multicompartment neuron model and the corresponding extracellular electric potential

$\mathbf{V}_{ex}$ (mV) as

$$\mathbf{V}_{ex} = \mathbf{MI}$$

The current $\mathbf{I}$ is an ndarray of shape (n_seg, n_tsteps) with unit (nA), and each row indexed by j of $\mathbf{V}_{ex}$ represents the electric potential at each measurement site for every time step.

Parameters

cell: object or None `CellGeometry` instance or similar.

r: ndarray, dtype=float shape(3, n_points) observation points in space in spherical coordinates (radius, theta, phi) relative to the center of the sphere.

R: float sphere radius (µm)

sigma_i: float electric conductivity for radius r <= R (S/m)

sigma_o: float electric conductivity for radius r > R (S/m)

References

[1]

Examples

Compute the potential for a single monopole along the x-axis:

```
>>> # import modules
>>> from lfpykit import OneSphereVolumeConductor
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> # observation points in spherical coordinates (flattened)
>>> X, Y = np.mgrid[-15000:15100:1000., -15000:15100:1000.]
>>> r = np.array([np.sqrt(X**2 + Y**2).flatten(),
>>>               np.arctan2(Y, X).flatten(),
>>>               np.zeros(X.size)])
>>> # set up class object and compute electric potential in all locations
>>> sphere = OneSphereVolumeConductor(cell=None, r=r, R=10000.,
>>>                                   sigma_i=0.3, sigma_o=0.03)
>>> Phi = sphere.calc_potential(rs=8000, current=1.).reshape(X.shape)
>>> # plot
>>> fig, ax = plt.subplots(1,1)
>>> im=ax.contourf(X, Y, Phi,
>>>                levels=np.linspace(Phi.min(),
>>>                np.median(Phi[np.isfinite(Phi)]) * 4, 30))
>>> circle = plt.Circle(xy=(0,0), radius=sphere.R, fc='none', ec='k')
>>> ax.add_patch(circle)
>>> fig.colorbar(im, ax=ax)
>>> plt.show()
```

calc_potential(*rs*, *current*, *min_distance=1.0*, *n_max=1000*)

Return the electric potential at observation points for source current as function of time.

Parameters

rs: float monopole source location along the horizontal x-axis (µm)

current: float or ndarray, dtype float float or shape (n_tsteps,) array containing source current (nA)

min_distance: None or float minimum distance between source location and observation point (µm) (in order to avoid singularities)

n_max: int Number of elements in polynomial expansion to sum over (see [1]).

Returns

Phi: ndarray shape (n-points,) ndarray of floats if I is float like. If I is an 1D ndarray, and shape (n-points, I.size) ndarray is returned. Unit (mV).

References

[1]

`get_transformation_matrix`(*n_max=1000*)

Compute linear mapping between transmembrane currents of `CellGeometry` like object instance and extracellular potential in and outside of sphere.

Parameters

n_max: int Number of elements in polynomial expansion to sum over (see [1]).

Returns

ndarray Shape (n_points, n_compartments) mapping between individual segments and extracellular potential in extracellular locations

Raises

AttributeError if `cell is None`

Notes

Each segment is treated as a point source in space. The minimum source to measurement site distance will be set to the diameter of each segment

References

[1]

Examples

Compute extracellular potential in one-sphere volume conductor model from `LFPy.Cell` object:

```
>>> # import modules
>>> import LFPy
>>> from lfpykit import CellGeometry,          >>>     OneSphereVolumeConductor
>>> import os
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from matplotlib.collections import PolyCollection
>>> # create cell
>>> cell = LFPy.Cell(morphology=os.path.join(LFPy.__path__[0], 'test',
```

(continues on next page)

(continued from previous page)

```
>>>                                             'ball_and_sticks.hoc'),
>>>                   tstop=10.)
>>> cell.set_pos(z=9800.)
>>> # stimulus
>>> syn = LFPy.Synapse(cell, idx=cell.totnsegs-1, syntype='Exp2Syn',
>>>                    weight=0.01)
>>> syn.set_spike_times(np.array([1.]))
>>> # simulate
>>> cell.simulate(rec_imem=True)
>>> # observation points in spherical coordinates (flattened)
>>> X, Z = np.mgrid[-500:501:10., 9500:10501:10.]
>>> Y = np.zeros(X.shape)
>>> r = np.array([np.sqrt(X**2 + Z**2).flatten(),
>>>               np.arccos(Z / np.sqrt(X**2 + Z**2)).flatten(),
>>>               np.arctan2(Y, X).flatten()])
>>> # set up class object and compute mapping between segment currents
>>> # and electric potential in space
>>> sphere = OneSphereVolumeConductor(cell, r=r, R=10000.,
>>>                                   sigma_i=0.3, sigma_o=0.03)
>>> M = sphere.get_transformation_matrix(n_max=1000)
>>> # pick out some time index for the potential and compute potential
>>> ind = cell.tvec==2.
>>> V_ex = (M @ cell.imem)[:, ind].reshape(X.shape)
>>> # plot potential
>>> fig, ax = plt.subplots(1,1)
>>> zips = []
>>> for x, z in cell.get_idx_polygons(projection=('x', 'z')):
>>>     zips.append(list(zip(x, z)))
>>> polycol = PolyCollection(zips,
>>>                          edgecolors='none',
>>>                          facecolors='gray')
>>> vrange = 1E-3 # limits for color contour plot
>>> im=ax.contour(X, Z, V_ex,
>>>               levels=np.linspace(-vrange, vrange, 41))
>>> circle = plt.Circle(xy=(0,0), radius=sphere.R, fc='none', ec='k')
>>> ax.add_collection(polycol)
>>> ax.add_patch(circle)
>>> ax.axis(ax.axis('equal'))
>>> ax.set_xlim(X.min(), X.max())
>>> ax.set_ylim(Z.min(), Z.max())
>>> fig.colorbar(im, ax=ax)
>>> plt.show()
```

1.7 Current Dipole Moment forward models

1.7.1 class InfiniteVolumeConductor

class LFPy.**InfiniteVolumeConductor**(*sigma=0.3*)

Bases: lfpykit.eegmegcalc.InfiniteVolumeConductor

Main class for computing extracellular potentials with current dipole moment $\mathbf{P}$ in an infinite 3D volume conductor model that assumes homogeneous, isotropic, linear (frequency independent) conductivity σ. The potential V is computed as [1]:

$$V = \frac{\mathbf{P} \cdot \mathbf{r}}{4\pi\sigma r^3}$$

Parameters

sigma: float Electrical conductivity in extracellular space in units of (S/cm)

See also:

FourSphereVolumeConductor

MEG

References

[1]

Examples

Computing the potential from dipole moment valid in the far field limit. Theta correspond to the dipole alignment angle from the vertical z-axis:

```
>>> from lfpykit.eegmegcalc import InfiniteVolumeConductor
>>> import numpy as np
>>> inf_model = InfiniteVolumeConductor(sigma=0.3)
>>> p = np.array([[10.], [10.], [10.]])  # [nA µm]
>>> r = np.array([[1000., 0., 5000.]])  # [µm]
>>> inf_model.get_dipole_potential(p, r)  # [mV]
array([[1.20049432e-07]])
```

get_dipole_potential(*p*, *r*)

Return electric potential from current dipole moment p in locations r relative to dipole

Parameters

p: ndarray, dtype=float Shape (3, n_timesteps) array containing the x,y,z components of the current dipole moment in units of (nA*µm) for all timesteps

r: ndarray, dtype=float Shape (n_contacts, 3) array containing the displacement vectors from dipole location to measurement location

Returns

potential: ndarray, dtype=float Shape (n_contacts, n_timesteps) array containing the electric potential at contact point(s) r in units of (mV) for all timesteps of current dipole moment p

get_multi_dipole_potential(*cell*, *electrode_locs*, *timepoints=None*)

Return electric potential from multiple current dipoles from cell

The multiple current dipoles corresponds to dipoles computed from all axial currents in a neuron simulation, typically two axial currents per compartment, excluding the root compartment.

Parameters

cell: LFPy.Cell object

electrode_locs: ndarray, dtype=float Shape (n_contacts, 3) array containing n_contacts electrode locations in cartesian coordinates in units of [µm]. All `r_el` in electrode_locs must be placed so that `|r_el|` is less than or equal to scalp radius and larger than the distance between dipole and sphere center: `|rz| < |r_el| <= radii[3]`.

timepoints: ndarray, dtype=int array of timepoints at which you want to compute the electric potential. Defaults to None. If not given, all simulation timesteps will be included.

Returns

potential: ndarray, dtype=float Shape (n_contacts, n_timesteps) array containing the electric potential at contact point(s) electrode_locs in units of [mV] for all timesteps of neuron simulation

Examples

Compute extracellular potential from neuron simulation in four-sphere head model. Instead of simplifying the neural activity to a single dipole, we compute the contribution from every multi dipole from all axial currents in neuron simulation:

```
>>> import LFPy
>>> from lfpykit.eegmegcalc import InfiniteVolumeConductor
>>> import numpy as np
>>> cell = LFPy.Cell('PATH/TO/MORPHOLOGY', extracellular=False)
>>> syn = LFPy.Synapse(cell, idx=cell.get_closest_idx(0,0,100),
>>>                    syntype='ExpSyn', e=0., tau=1., weight=0.001)
>>> syn.set_spike_times(np.mgrid[20:100:20])
>>> cell.simulate(rec_vmem=True, rec_imem=False)
>>> sigma = 0.3
>>> timepoints = np.array([10, 20, 50, 100])
>>> electrode_locs = np.array([[50., -50., 250.]])
>>> MD_INF = InfiniteVolumeConductor(sigma)
>>> phi = MD_INF.get_multi_dipole_potential(cell, electrode_locs,
>>>                                         timepoints = timepoints)
```

get_transformation_matrix(*r*)

Get linear response matrix mapping current dipole moment in (nA µm) to extracellular potential in (mV) at recording sites *r* (µm)

Parameters

r: ndarray, dtype=float Shape (n_contacts, 3) array contaning the displacement vectors from dipole location to measurement location (µm)

Returns

response_matrix: ndarray shape (n_contacts, 3) ndarray

1.7.2 class FourSphereVolumeConductor

class LFPy.**FourSphereVolumeConductor**(*r_electrodes*, *radii=None*, *sigmas=None*, *iter_factor=2.0202020202020204e-08*)

Bases: `lfpykit.eegmegcalc.FourSphereVolumeConductor`

Main class for computing extracellular potentials in a four-sphere volume conductor model that assumes homogeneous, isotropic, linear (frequency independent) conductivity within the inner sphere and outer shells. The conductance outside the outer shell is 0 (air).

This class implements the corrected 4-sphere model described in [1], [2]

Parameters

r_electrodes: ndarray, dtype=float Shape (n_contacts, 3) array containing n_contacts electrode locations in cartesian coordinates in units of (µm). All `r_el` in `r_electrodes` must be less than or equal to scalp radius and larger than the distance between dipole and sphere center: `|rz| < r_el <= radii[3]`.

radii: list, dtype=float Len 4 list with the outer radii in units of (µm) for the 4 concentric shells in the four-sphere model: brain, csf, skull and scalp, respectively.

sigmas: list, dtype=float Len 4 list with the electrical conductivity in units of (S/m) of the four shells in the four-sphere model: brain, csf, skull and scalp, respectively.

iter_factor: float iteration-stop factor

See also:

InfiniteVolumeConductor

MEG

References

[1], [2]

Examples

Compute extracellular potential from current dipole moment in four-sphere head model:

```
>>> from lfpykit.eegmegcalc import FourSphereVolumeConductor
>>> import numpy as np
>>> radii = [79000., 80000., 85000., 90000.]  # (µm)
>>> sigmas = [0.3, 1.5, 0.015, 0.3]  # (S/m)
>>> r_electrodes = np.array([[0., 0., 90000.], [0., 85000., 0.]]) # (µm)
>>> sphere_model = FourSphereVolumeConductor(r_electrodes, radii,
>>>                                          sigmas)
>>> # current dipole moment
>>> p = np.array([[10.]*10, [10.]*10, [10.]*10]) # 10 timesteps (nA µm)
>>> dipole_location = np.array([0., 0., 78000.])  # (µm)
>>> # compute potential
>>> sphere_model.get_dipole_potential(p, dipole_location)  # (mV)
array([[1.06247669e-08, 1.06247669e-08, 1.06247669e-08, 1.06247669e-08,
        1.06247669e-08, 1.06247669e-08, 1.06247669e-08, 1.06247669e-08,
        1.06247669e-08, 1.06247669e-08],
```

(continues on next page)

(continued from previous page)

```
       [2.39290752e-10, 2.39290752e-10, 2.39290752e-10, 2.39290752e-10,
        2.39290752e-10, 2.39290752e-10, 2.39290752e-10, 2.39290752e-10,
        2.39290752e-10, 2.39290752e-10]])
```

`get_dipole_potential`(*p*, *dipole_location*)

Return electric potential from current dipole moment `p` in location `dipole_location` in locations `r_electrodes`

Parameters

p: ndarray, dtype=float Shape (3, n_timesteps) array containing the x,y,z components of the current dipole moment in units of (nA*µm) for all timesteps.

dipole_location: ndarray, dtype=float Shape (3,) array containing the position of the current dipole in cartesian coordinates. Units of (µm).

Returns

potential: ndarray, dtype=float Shape `(n_contacts, n_timesteps)` array containing the electric potential at contact point(s) `FourSphereVolumeConductor.rxyz` in units of (mV) for all timesteps of current dipole moment `p`.

`get_dipole_potential_from_multi_dipoles`(*cell*, *timepoints=None*)

Return electric potential from multiple current dipoles from cell.

By multiple current dipoles we mean the dipoles computed from all axial currents in a neuron simulation, typically two axial currents per compartment, except for the root compartment.

Parameters

cell: LFPy Cell object, LFPy.Cell

timepoints: ndarray, dtype=int array of timepoints at which you want to compute the electric potential. Defaults to None. If not given, all simulation timesteps will be included.

Returns

potential: ndarray, dtype=float Shape (n_contacts, n_timesteps) array containing the electric potential at contact point(s) electrode_locs in units of [mV] for all timesteps of neuron simulation.

Examples

Compute extracellular potential from neuron simulation in four-sphere head model. Instead of simplifying the neural activity to a single dipole, we compute the contribution from every multi dipole from all axial currents in neuron simulation:

```
>>> import os
>>> import LFPy
>>> from LFPy import FourSphereVolumeConductor
>>> import numpy as np
>>> cell = LFPy.Cell(os.path.join(LFPy.__path__[0], 'test',
>>>                               'ball_and_sticks.hoc'),
>>>                  v_init=-65, cm=1., Ra=150,
>>>                  passive=True,
>>>                  passive_parameters=dict(g_pas=1/1E4, e_pas=-65))
>>> syn = LFPy.Synapse(cell, idx=cell.get_closest_idx(0,0,100),
```

(continues on next page)

(continued from previous page)

```
>>>                     syntype='ExpSyn', e=0., tau=1., weight=0.001)
>>> syn.set_spike_times(np.mgrid[20:100:20])
>>> cell.simulate(rec_vmem=True, rec_imem=False)
>>> cell.set_pos(0, 0, 78800)
>>> radii = [79000., 80000., 85000., 90000.]
>>> sigmas = [0.3, 1.5, 0.015, 0.3]
>>> r_electrodes = np.array([[0., 0., 90000.]])
>>> MD_4s = FourSphereVolumeConductor(r_electrodes=r_electrodes,
>>>                                   radii=radii,
>>>                                   sigmas=sigmas)
>>> phi = MD_4s.get_dipole_potential_from_multi_dipoles(cell)
```

get_transformation_matrix(*dipole_location*)

Get linear response matrix mapping current dipole moment in (nA µm) located in location `rz` to extracellular potential in (mV) at recording sites `FourSphereVolumeConductor.rxyz` (µm)

Parameters

dipole_location: ndarray, dtype=float Shape (3,) array containing the position of the current dipole in cartesian coordinates. Units of (µm).

Returns

response_matrix: ndarray shape (n_contacts, 3) ndarray

1.7.3 class NYHeadModel

class LFPy.**NYHeadModel**(*nyhead_file=None*)

Bases: `object`

Main class for computing EEG signals from current dipole moment $\mathbf{P}$ in New York Head Model [1], [2]

Assumes units of nA * um for current dipole moment, and mV for the EEG

Parameters

nyhead_file: str [optional] Location of file containing New York Head Model. If empty (or None), it will be looked for in the main LFPykit folder. If not present the user is asked if it should be downloaded from https://www.parralab.org/nyhead/sa_nyhead.mat

See also:

FourSphereVolumeConductor

MEG

Notes

The original unit of the New York model current dipole moment is (probably?) mA*m, and the EEG output unit is V. LFPykit's current dipole moments have units nA*um, and EEGs from the NYhead model is here recomputed in units of mV.

References

[1], [2]

Examples

Computing EEG from dipole moment.

```
>>> from lfpykit.eegmegcalc import NYHeadModel
```

```
>>> nyhead = NYHeadModel()
```

```
>>> nyhead.set_dipole_pos('parietal_lobe') # predefined example location
>>> M = nyhead.get_transformation_matrix()
```

```
>>> # Rotate to be along normal vector of cortex
>>> p = nyhead.rotate_dipole_to_surface_normal([[0.], [0.], [1.]])
>>> eeg = M @ p  # (mV)
```

find_closest_electrode()

Returns minimal distance (mm) and closest electrode idx to dipole location specified in self.dipole_pos.

get_transformation_matrix()

Get linear response matrix mapping from current dipole moment (nA µm) to EEG signal (mV) at EEG electrodes (n=231)

Returns

response_matrix: ndarray shape (231, 3) ndarray

return_closest_idx(*pos*)

Returns the index of the closest vertex in the brain to a given position (in mm).

Parameters

pos [array of length (3)] [x, y, z] of a location in the brain, given in mm, and not in um which is the default position unit in LFPy

Returns

——-

idx [int] Index of the vertex in the brain that is closest to the given location

rotate_dipole_to_surface_normal(*p*, *orig_ax_vec=[0, 0, 1]*)

Returns rotated dipole moment, p_rot, oriented along the normal vector of the cortex at the dipole location

Parameters

p [np.ndarray of size (3, num_timesteps)] Current dipole moment from neural simulation, [p_x(t), p_y(t), p_z(t)]. If z-axis is the depth axis of cortex in the original neural simulation p_x(t) and p_y(t) will typically be small, and orig_ax_vec = [0, 0, 1].

orig_ax_vec [np.ndarray or list of length (3)] Original surface vector of cortex in the neural simulation. If depth axis of cortex is the z-axis, orig_ax_vec = [0, 0, 1].

Returns

p_rot [np.ndarray of size (3, num_timesteps)] Rotated current dipole moment, oriented along cortex normal vector at the dipole location

References

See: https://en.wikipedia.org/wiki/Rotation_matrix under "Rotation matrix from axis and angle"

`set_dipole_pos`(*dipole_pos=None*)

Sets the dipole location in the brain

Parameters

dipole_pos: None, str or array of length (3) [x, y, z) (mm) Location of the dipole. If no argument is given (or dipole_pos=None), a location, 'motorsensory_cortex', from self.dipole_pos_dict is used. If dipole_pos is an array of length 3, the closest vertex in the brain will be set as the dipole location.

1.7.4 class `InfiniteHomogeneousVolCondMEG`

`class` LFPy.`InfiniteHomogeneousVolCondMEG`(*sensor_locations*, *mu=1.2566370614359173e-06*)

Bases: `lfpykit.eegmegcalc.InfiniteHomogeneousVolCondMEG`

Basic class for computing magnetic field from current dipole moment. For this purpose we use the Biot-Savart law derived from Maxwell's equations under the assumption of negligible magnetic induction effects [1]:

$$\mathbf{H} = \frac{\mathbf{p} \times \mathbf{R}}{4\pi R^3}$$

where $\mathbf{p}$ is the current dipole moment, $\mathbf{R}$ the vector between dipole source location and measurement location, and $R = |\mathbf{R}|$

Note that the magnetic field $\mathbf{H}$ is related to the magnetic field $\mathbf{B}$ as

$$\mu_0 \mathbf{H} = \mathbf{B} - \mathbf{M}$$

where μ_0 is the permeability of free space (very close to permebility of biological tissues). $\mathbf{M}$ denotes material magnetization (also ignored)

Parameters

sensor_locations: ndarray, dtype=float shape (n_locations x 3) array with x,y,z-locations of measurement devices where magnetic field of current dipole moments is calculated. In unit of [µm]

mu: float Permeability. Default is permeability of vacuum ($\mu_0 = 4 * \pi * 10^{-7}$ T*m/A)

Raises

AssertionError If dimensionality of sensor_locations is wrong

See also:

FourSphereVolumeConductor

InfiniteVolumeConductor

References

[1]

Examples

Define cell object, create synapse, compute current dipole moment:

```
>>> import LFPy, os, numpy as np, matplotlib.pyplot as plt
>>> from LFPy import InfiniteHomogeneousVolCondMEG as MEG
>>> from LFPy import CurrentDipoleMoment
>>> # create LFPy.Cell object
>>> cell = LFPy.Cell(morphology=os.path.join(LFPy.__path__[0], 'test',
>>>                                          'ball_and_sticks.hoc'),
>>>                  passive=True)
>>> cell.set_pos(0., 0., 0.)
>>> # create single synaptic stimuli at soma (idx=0)
>>> syn = LFPy.Synapse(cell, idx=0, syntype='ExpSyn', weight=0.01, tau=5,
>>>                    record_current=True)
>>> syn.set_spike_times_w_netstim()
>>> # simulate, record current dipole moment
>>> cdm = CurrentDipoleMoment(cell=cell)
>>> cell.simulate(probes=[cdm])
>>> # Compute the dipole location as an average of segment locations
>>> # weighted by membrane area:
>>> dipole_location = (cell.area * np.c_[cell.x.mean(axis=1),
>>>                                      cell.y.mean(axis=1),
>>>                                      cell.z.mean(axis=1)].T
>>>                    / cell.area.sum()).sum(axis=1)
>>> # Define sensor site, instantiate MEG object, get transformation matrix
>>> sensor_locations = np.array([[1E4, 0, 0]])
>>> meg = MEG(sensor_locations)
>>> M = meg.get_transformation_matrix(dipole_location)
>>> # compute the magnetic signal in a single sensor location:
>>> H = M @ cdm.data
>>> # plot output
>>> plt.figure(figsize=(12, 8), dpi=120)
>>> plt.subplot(311)
>>> plt.plot(cell.tvec, cell.somav)
>>> plt.ylabel(r'$V_{soma}$ (mV)')
>>> plt.subplot(312)
>>> plt.plot(cell.tvec, syn.i)
>>> plt.ylabel(r'$I_{syn}$ (nA)')
>>> plt.subplot(313)
>>> plt.plot(cell.tvec, H[0].T)
>>> plt.ylabel(r'$H$ (nA/um)')
>>> plt.xlabel('$t$ (ms)')
>>> plt.legend(['$H_x$', '$H_y$', '$H_z$'])
>>> plt.show()
```

`calculate_B`(*p*, *r_p*)

Compute magnetic field B from single current dipole `p` localized somewhere in space at `r_p`.

This function returns the magnetic field $\mathbf{B} = \mathbf{H}$.

Parameters

p: ndarray, dtype=float shape (3, n_timesteps) array with x,y,z-components of current-dipole moment time series data in units of (nA µm)

r_p: ndarray, dtype=float shape (3,) array with x,y,z-location of dipole in units of (µm)

Returns

ndarray, dtype=float shape (n_locations x 3 x n_timesteps) array with x,y,z-components of the magnetic field $\mathbf{B}$ in units of (nA/µm)

`calculate_H`(*current_dipole_moment*, *dipole_location*)

Compute magnetic field H from single current-dipole moment localized in an infinite homogeneous volume conductor.

Parameters

current_dipole_moment: ndarray, dtype=float shape (3, n_timesteps) array with x,y,z-components of current- dipole moment time series data in units of (nA µm)

dipole_location: ndarray, dtype=float shape (3,) array with x,y,z-location of dipole in units of (µm)

Returns

ndarray, dtype=float shape (n_locations x 3 x n_timesteps) array with x,y,z-components of the magnetic field $\mathbf{H}$ in units of (nA/µm)

Raises

AssertionError If dimensionality of current_dipole_moment and/or dipole_location is wrong

`calculate_H_from_iaxial`(*cell*)

Computes the magnetic field in space from axial currents computed from membrane potential values and axial resistances of multicompartment cells.

See [1] for details on the biophysics governing magnetic fields from axial currents.

Parameters

cell: object LFPy.Cell-like object. Must have attribute vmem containing recorded membrane potentials in units of mV

Returns

H: ndarray, dtype=float shape (n_locations x 3 x n_timesteps) array with x,y,z-components of the magnetic field $\mathbf{H}$ in units of (nA/µm)

References

[1]

Examples

Define cell object, create synapse, compute current dipole moment:

```
>>> import LFPy, os, numpy as np, matplotlib.pyplot as plt
>>> from LFPy import InfiniteHomogeneousVolCondMEG as MEG
>>> cell = LFPy.Cell(morphology=os.path.join(LFPy.__path__[0], 'test',
>>>                                          'ball_and_sticks.hoc'),
>>>                  passive=True)
>>> cell.set_pos(0., 0., 0.)
>>> syn = LFPy.Synapse(cell, idx=0, syntype='ExpSyn', weight=0.01,
>>>                    record_current=True)
>>> syn.set_spike_times_w_netstim()
>>> cell.simulate(rec_vmem=True)
>>> # Instantiate the MEG object, compute and plot the magnetic
>>> # signal in a sensor location:
>>> sensor_locations = np.array([[1E4, 0, 0]])
>>> meg = MEG(sensor_locations)
>>> H = meg.calculate_H_from_iaxial(cell)
>>> plt.subplot(311)
>>> plt.plot(cell.tvec, cell.somav)
>>> plt.subplot(312)
>>> plt.plot(cell.tvec, syn.i)
>>> plt.subplot(313)
>>> plt.plot(cell.tvec, H[0, 1, :])  # y-component
>>> plt.show()
```

get_transformation_matrix(*dipole_location*)

Get linear response matrix mapping current dipole moment in (nA µm) located in location `dipole_location` to magnetic field **H** in units of (nA/µm) at `sensor_locations`

Parameters

dipole_location: ndarray, dtype=float shape (3,) array with x,y,z-location of dipole in units of (µm)

Returns

response_matrix: ndarray shape (n_contacts, 3, 3) ndarray

1.7.5 class SphericallySymmetricVolCondMEG

class LFPy.**SphericallySymmetricVolCondMEG**(*r*, *mu=1.2566370614359173e-06*)

Bases: `lfpykit.eegmegcalc.SphericallySymmetricVolCondMEG`

Computes magnetic fields from current dipole in spherically-symmetric volume conductor models.

This class facilitates calculations according to eq. (34) from [1] (see also [2]) defined as:

$$\mathbf{H} = \frac{1}{4\pi}\frac{F\mathbf{p}\times\mathbf{r}_p - (\mathbf{p}\times\mathbf{r}_p\cdot\mathbf{r})\nabla F}{F^2}, \text{ where}$$
$$F = a(ra + r^2 - \mathbf{r}_p\cdot\mathbf{r}),$$
$$\nabla F = (r^{-1}a^2 + a^{-1}\mathbf{a}\cdot\mathbf{r} + 2a + 2r)\mathbf{r} - (a + 2r + a^{-1}\mathbf{a}\cdot\mathbf{r})\mathbf{r}_p,$$
$$\mathbf{a} = \mathbf{r} - \mathbf{r}_p,$$
$$a = |\mathbf{a}|,$$
$$r = |\mathbf{r}|.$$

Here, $\mathbf{p}$ is the current dipole moment, $\mathbf{r}$ the measurement location(s) and $\mathbf{r}_p$ the current dipole location.

Note that the magnetic field $\mathbf{H}$ is related to the magnetic field $\mathbf{B}$ as

$$\mu_0\mathbf{H} = \mathbf{B} - \mathbf{M},$$

where μ_0 denotes the permeability of free space (very close to permebility of biological tissues). $\mathbf{M}$ denotes material magnetization (which is ignored).

Parameters

r: ndarray sensor locations, shape `(n, 3)` where `n` denotes number of locations, unit [µm]

mu: float Permeability. Default is permeability of vacuum ($\mu_0 = 4\pi 10^{-7}$ Tm/A)

Raises

AssertionError If dimensionality of sensor locations `r` is wrong

See also:

InfiniteHomogeneousVolCondMEG

References

[1], [2]

Examples

Compute the magnetic field from a current dipole located

```
>>> import numpy as np
>>> from LFPy import SphericallySymmetricVolCondMEG
>>> p = np.array([[0, 1, 0]]).T  # tangential current dipole (nAµm)
>>> r_p = np.array([0, 0, 90000])  # dipole location (µm)
>>> r = np.array([[0, 0, 92000]])  # measurement location (µm)
>>> m = SphericallySymmetricVolCondMEG(r=r)
>>> M = m.get_transformation_matrix(r_p=r_p)
>>> H = M @ p
>>> H  # magnetic field (nA/µm)
array([[[9.73094081e-09],
        [0.00000000e+00],
        [0.00000000e+00]]])
```

`calculate_B`(*p*, *r_p*)

Compute magnetic field B from single current dipole `p` localized somewhere in space at `r_p`.

This function returns the magnetic field $\mathbf{B} = \mathbf{H}$.

Parameters

p: ndarray, dtype=float shape (3, n_timesteps) array with x,y,z-components of current-dipole moment time series data in units of (nA µm)

r_p: ndarray, dtype=float shape (3,) array with x,y,z-location of dipole in units of (µm)

Returns

ndarray, dtype=float shape (n_locations x 3 x n_timesteps) array with x,y,z-components of the magnetic field $\mathbf{B}$ in units of (nA/µm)

`calculate_H`(*p*, *r_p*)

Compute magnetic field $\mathbf{H}$ from single current dipole `p` localized somewhere in space at `r_p`

Parameters

p: ndarray, dtype=float shape (3, n_timesteps) array with x,y,z-components of current-dipole moment time series data in units of (nA µm)

r_p: ndarray, dtype=float shape (3,) array with x,y,z-location of dipole in units of (µm)

Returns

ndarray, dtype=float shape (n_locations x 3 x n_timesteps) array with x,y,z-components of the magnetic field $\mathbf{H}$ in units of (nA/µm)

Raises

AssertionError If dimensionality of current_dipole_moment `p` and/or dipole_location `r_p` is wrong

`calculate_H_from_iaxial`(*cell*)

Computes the magnetic field in space from axial currents computed from membrane potential values and axial resistances of multicompartment cells.

See [1] for details on the biophysics governing magnetic fields from axial currents.

Parameters

cell: object LFPy.Cell-like object. Must have attribute vmem containing recorded membrane potentials in units of mV

Returns

H: ndarray, dtype=float shape (n_locations x 3 x n_timesteps) array with x,y,z-components of the magnetic field $\mathbf{H}$ in units of (nA/µm)

References

[1]

Examples

Define cell object, create synapse, compute current dipole moment:

```
>>> import LFPy, os, numpy as np, matplotlib.pyplot as plt
>>> from LFPy import SphericallySymmetricVolCondMEG as MEG
>>> cell = LFPy.Cell(morphology=os.path.join(LFPy.__path__[0], 'test',
>>>                                          'ball_and_sticks.hoc'),
>>>                  passive=True)
>>> cell.set_pos(0., 0., 0.)
>>> syn = LFPy.Synapse(cell, idx=0, syntype='ExpSyn', weight=0.01,
>>>                    record_current=True)
>>> syn.set_spike_times_w_netstim()
>>> cell.simulate(rec_vmem=True)
>>> # Instantiate the MEG object, compute and plot the magnetic
>>> # signal in a sensor location:
>>> r = np.array([[1E4, 0, 0]])
>>> meg = MEG(r)
>>> H = meg.calculate_H_from_iaxial(cell)
>>> plt.subplot(311)
>>> plt.plot(cell.tvec, cell.somav)
>>> plt.subplot(312)
>>> plt.plot(cell.tvec, syn.i)
>>> plt.subplot(313)
>>> plt.plot(cell.tvec, H[0, 1, :])  # y-component
>>> plt.show()
```

`get_transformation_matrix`(*r_p*)

Get linear response matrix mapping current dipole moment in (nA µm) located in location `r_p` to magnetic field **H** in units of (nA/µm) at sensor locations `r`

Parameters

r_p: ndarray, dtype=float shape (3,) array with x,y,z-location of dipole in units of (µm)

Returns

response_matrix: ndarray shape (n_sensors, 3, 3) ndarray

Raises

AssertionError If dipole location `r_p` has the wrong shape or if its radius is greater than radius to any sensor location in `<object>.r`

1.8 Current Source Density (CSD)

1.8.1 class LaminarCurrentSourceDensity

class LFPy.**LaminarCurrentSourceDensity**(*cell*, *z*, *r*)

Bases: lfpykit.models.LinearModel

Facilitates calculations of the ground truth Current Source Density (CSD) in cylindrical volumes aligned with the z-axis based on [1] and [2].

The implementation assumes piecewise linear current sources similar to LineSourcePotential, and accounts for the fraction of each segment's length within each volume, see Eq. 11 in [2].

This class is a LinearModel subclass that defines a 2D linear response matrix $\mathbf{M}$ between transmembrane current array $\mathbf{I}$ (nA) of a multicompartment neuron model and the corresponding CSD $\mathbf{C}$ (nA/µm^3) as

$$\mathbf{C} = \mathbf{M}\mathbf{I}$$

The current $\mathbf{I}$ is an ndarray of shape (n_seg, n_tsteps) with unit (nA), and each row indexed by j of $\mathbf{C}$ represents the CSD in each volume for every time step as the sum of currents divided by the volume.

Parameters

cell: object or None CellGeometry instance or similar.

z: ndarray, dtype=float shape (n_volumes, 2) array of lower and upper edges of each volume along the z-axis in units of (µm). The lower edge value must be below the upper edge value.

r: ndarray, dtype=float shape (n_volumes,) array with assumed radius of each cylindrical volume. Each radius must be greater than zero, and in units of (µm)

Raises

AttributeError inputs z and r must be ndarrays of correct shape etc.

See also:

LinearModel

VolumetricCurrentSourceDensity

References

[1], [2]

Examples

Mock cell geometry and transmembrane currents:

```
>>> import numpy as np
>>> from lfpykit import CellGeometry, LaminarCurrentSourceDensity
>>> # cell geometry with three segments (µm)
>>> cell = CellGeometry(x=np.array([[0, 0], [0, 0], [0, 0]]),
>>>                     y=np.array([[0, 0], [0, 0], [0, 0]]),
>>>                     z=np.array([[0, 10], [10, 20], [20, 30]]),
>>>                     d=np.array([1, 1, 1]))
```

(continues on next page)

(continued from previous page)

```
>>> # transmembrane currents, three time steps (nA)
>>> I_m = np.array([[0., -1., 1.], [-1., 1., 0.], [1., 0., -1.]])
>>> # define geometry (z - upper and lower boundary;  r - radius)
>>> # of cylindrical volumes aligned with the z-axis (µm)
>>> z = np.array([[-10., 0.], [0., 10.], [10., 20.],
>>>               [20., 30.], [30., 40.]])
>>> r = np.array([100., 100., 100., 100., 100.])
>>> # instantiate electrode, get linear response matrix
>>> csd = LaminarCurrentSourceDensity(cell=cell, z=z, r=r)
>>> M = csd.get_transformation_matrix()
>>> # compute current source density [nA/µm3]
>>> M @ I_m
array([[ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 0.00000000e+00, -3.18309886e-06,  3.18309886e-06],
       [-3.18309886e-06,  3.18309886e-06,  0.00000000e+00],
       [ 3.18309886e-06,  0.00000000e+00, -3.18309886e-06],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00]])
```

get_transformation_matrix()

Get linear response matrix

Returns

response_matrix: ndarray shape (n_volumes, n_seg) ndarray

Raises

AttributeError if `cell is None`

1.8.2 class VolumetricCurrentSourceDensity

class LFPy.**VolumetricCurrentSourceDensity**(*cell*, *x=None*, *y=None*, *z=None*, *dl=1.0*)

Bases: `lfpykit.models.LinearModel`

Facilitates calculations of the ground truth Current Source Density (CSD) across 3D volumetric grid with bin edges defined by parameters `x`, `y` and `z`.

The implementation assumes piecewise constant current sources similar to `LineSourcePotential`, and accounts for the fraction of each segment's length within each volume by counting the number of points representing partial segments with max length `dl` divided by the number of partial segments.

This class is a `LinearModel` subclass that defines a 4D linear response matrix $\mathbf{M}$ of shape `(x.size-1, y.size-1, z.size-1, n_seg)` between transmembrane current array $\mathbf{I}$ (nA) of a multicompartment neuron model and the corresponding CSD $\mathbf{C}$ (nA/µm^3) as

$$\mathbf{C} = \mathbf{M}\mathbf{I}$$

The current $\mathbf{I}$ is an ndarray of shape (n_seg, n_tsteps) with unit (nA), and each row indexed by j of $\mathbf{C}$ represents the CSD in each bin for every time step as the sum of currents divided by the volume.

Parameters

cell: object or None `CellGeometry` instance or similar.

x, y, z: ndarray, dtype=float shape (n,) array of bin edges of each volume along each axis in units of (µm). Must be monotonously increasing.

dl: float discretization length of compartments before binning (µm). Default=1. Lower values will result in more accurate estimates as each line source gets split into more points.

Raises

See also:

`LinearModel`

LaminarCurrentSourceDensity

Notes

The resulting mapping M may be very sparse (i.e, mostly made up by zeros) and can be converted into a sparse array for more efficient multiplication for the same result:

```
>>> import scipy.sparse as ss
>>> M_csc = ss.csc_matrix(M.reshape((-1, M.shape[-1])))
>>> C = M_csc @ I_m
>>> np.all(C.reshape((M.shape[:-1] + (-1,))) == (M @ I_m))
True
```

Examples

Mock cell geometry and transmembrane currents:

```
>>> import numpy as np
>>> from lfpykit import CellGeometry, VolumetricCurrentSourceDensity
>>> # cell geometry with three segments (µm)
>>> cell = CellGeometry(x=np.array([[0, 0], [0, 0], [0, 0]]),
>>>                     y=np.array([[0, 0], [0, 0], [0, 0]]),
>>>                     z=np.array([[0, 10], [10, 20], [20, 30]]),
>>>                     d=np.array([1, 1, 1]))
>>> # transmembrane currents, three time steps (nA)
>>> I_m = np.array([[0., -1., 1.], [-1., 1., 0.], [1., 0., -1.]])
>>> # instantiate probe, get linear response matrix
>>> csd = VolumetricCurrentSourceDensity(cell=cell,
>>>                                      x=np.linspace(-20, 20, 5),
>>>                                      y=np.linspace(-20, 20, 5),
>>>                                      z=np.linspace(-20, 20, 5), dl=1.)
>>> M = csd.get_transformation_matrix()
>>> # compute current source density [nA/µm3]
>>> M @ I_m
array([[[[ 0.,  0.,  0.],
         [ 0.,  0.,  0.],
         [ 0.,  0.,  0.],
         [ 0.,  0.,  0.]],
         ...
```

`get_transformation_matrix()`

Get linear response matrix

Returns

response_matrix: ndarray shape (x.size-1, y.size-1, z.size-1, n_seg) ndarray

Raises

AttributeError if `cell is None`

1.9 Misc.

1.9.1 submodule `lfpcalc`

1.9.2 submodule `tools`

`LFPy.tools.`**`load`**(*filename*)

Generic loading of cPickled objects from file

Parameters

filename: str path to pickle file

`LFPy.tools.`**`noise_brown`**(*ncols*, *nrows=1*, *weight=1.0*, *filter=None*, *filterargs=None*)

Return 1/f^2 noise of shape(nrows, ncols obtained by taking the cumulative sum of gaussian white noise, with rms weight.

If filter is not None, this function will apply the filter coefficients obtained by:

```
>>> b, a = filter(**filterargs)
>>> signal = scipy.signal.lfilter(b, a, signal)
```

Parameters

ncols: int

nrows: int

weight: float

filter: None or function

filterargs: **dict parameters passed to *filter*

1.9.3 submodule alias_method

LFPy.alias_method.**alias_method**(*idx*, *probs*, *nsyn*)

Alias method for drawing random numbers from a discrete probability distribution. See http://www.keithschwarz.com/darts-dice-coins/

Parameters

idx: np.ndarray compartment indices as array of ints

probs: np.ndarray compartment areas as array of floats

nsyn: int number of randomized compartment indices

Returns

out: np.ndarray integer array of randomly drawn compartment indices

LFPy.alias_method.**alias_setup**(*probs*)

Set up function for alias method. See http://www.keithschwarz.com/darts-dice-coins/

Parameters

probs: np.ndarray float array of compartment areas

Returns

J: np.ndarray array of ints

q: np.ndarray array of floats

1.9.4 submodule inputgenerators

Copyright (C) 2012 Computational Neuroscience Group, NMBU.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

LFPy.inputgenerators.**get_activation_times_from_distribution**(*n*, *tstart=0.0*, *tstop=1000000.0*, *distribution=<scipy.stats._continuous_distns.expon_gen object>*, *rvs_args={'loc': 0, 'scale': 1}*, *maxiter=1000000.0*)

Construct a length n list of ndarrays containing continously increasing random numbers on the interval [tstart, tstop], with intervals drawn from a chosen continuous random variable distribution subclassed from scipy.stats.rv_continous, e.g., scipy.stats.expon or scipy.stats.gamma.

The most likely initial first entry is `tstart + method.rvs(size=inf, **rvs_args).mean()`

Parameters

n: int number of ndarrays in list

tstart: float minimum allowed value in ndarrays

tstop: float maximum allowed value in ndarrays

distribution: object subclass of scipy.stats.rv_continous. Distributions producing negative values should be avoided if continously increasing values should be obtained, i.e., the probability density function `(distribution.pdf(**rvs_args))` should be `0` for `x < 0`, which is not explicitly tested for.

rvs_args: dict parameters for method.rvs method. If "size" is in dict, then tstop will be ignored, and each ndarray in output list will be `distribution.rvs(**rvs_args).cumsum() + tstart`. If size is not given in dict, then values up to tstop will be included

maxiter: int maximum number of iterations

Returns

list of ndarrays length n list of arrays containing data

Raises

AssertionError if distribution does not have the 'rvs' attribute

StopIteration if number of while-loop iterations reaches maxiter

Examples

Create n sets of activation times with intervals drawn from the exponential distribution, with rate expectation lambda 10 s^-1 (thus scale=1000 / lambda). Here we assume output in units of ms

```
>>> from LFPy.inputgenerators import get_activation_times_from_distribution
>>> import scipy.stats as st
>>> import matplotlib.pyplot as plt
>>> times = get_activation_times_from_distribution(n=10, tstart=0.,
>>>                                                tstop=1000.,
>>>                                                distribution=st.expon,
>>>                                                rvs_args=dict(loc=0.,
>>>                                                scale=100.))
```

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[1] H. Linden, K. H. Pettersen, G. T. Einevoll (2010). Intrinsic dendritic filtering gives low-pass power spectra of local field potentials. J Comput Neurosci, 29:423–444. DOI: 10.1007/s10827-010-0245-4

[1] Linden H, Hagen E, Leski S, Norheim ES, Pettersen KH, Einevoll GT (2014) LFPy: a tool for biophysical simulation of extracellular potentials generated by detailed model neurons. Front. Neuroinform. 7:41. doi: 10.3389/fninf.2013.00041

[2] Hagen E, Næss S, Ness TV and Einevoll GT (2018) Multimodal Modeling of Neural Network Activity: Computing LFP, ECoG, EEG, and MEG Signals With LFPy 2.0. Front. Neuroinform. 12:92. doi: 10.3389/fninf.2018.00092

[1] Linden H, Hagen E, Leski S, Norheim ES, Pettersen KH, Einevoll GT (2014) LFPy: a tool for biophysical simulation of extracellular potentials generated by detailed model neurons. Front. Neuroinform. 7:41. doi: 10.3389/fninf.2013.00041

[2] Hagen E, Næss S, Ness TV and Einevoll GT (2018) Multimodal Modeling of Neural Network Activity: Computing LFP, ECoG, EEG, and MEG Signals With LFPy 2.0. Front. Neuroinform. 12:92. doi: 10.3389/fninf.2018.00092

[1] Ness, T. V., Chintaluri, C., Potworowski, J., Leski, S., Glabska, H., Wojcik, D. K., et al. (2015). Modelling and analysis of electrical potentials recorded in microelectrode arrays (MEAs). Neuroinformatics 13:403–426. doi: 10.1007/s12021-015-9265-6

[2] Linden H, Hagen E, Leski S, Norheim ES, Pettersen KH, Einevoll GT (2014) LFPy: a tool for biophysical simulation of extracellular potentials generated by detailed model neurons. Front. Neuroinform. 7:41. doi: 10.3389/fninf.2013.00041

[3] Hagen E, Næss S, Ness TV and Einevoll GT (2018) Multimodal Modeling of Neural Network Activity: Computing LFP, ECoG, EEG, and MEG Signals With LFPy 2.0. Front. Neuroinform. 12:92. doi: 10.3389/fninf.2018.00092

[1] Ness, T. V., Chintaluri, C., Potworowski, J., Leski, S., Glabska, H., Wojcik, D. K., et al. (2015). Modelling and analysis of electrical potentials recorded in microelectrode arrays (MEAs). Neuroinformatics 13:403–426. doi: 10.1007/s12021-015-9265-6

[1] Shaozhong Deng (2008), Journal of Electrostatics 66:549-560. DOI: 10.1016/j.elstat.2008.06.003

[1] Shaozhong Deng (2008), Journal of Electrostatics 66:549-560. DOI: 10.1016/j.elstat.2008.06.003

[1] Shaozhong Deng (2008), Journal of Electrostatics 66:549-560. DOI: 10.1016/j.elstat.2008.06.003

[1] Nunez and Srinivasan, Oxford University Press, 2006

[1] Næss S, Chintaluri C, Ness TV, Dale AM, Einevoll GT and Wójcik DK (2017) Corrected Four-sphere Head Model for EEG Signals. Front. Hum. Neurosci. 11:490. doi: 10.3389/fnhum.2017.00490

[2] Hagen E, Næss S, Ness TV and Einevoll GT (2018) Multimodal Modeling of Neural Network Activity: Computing LFP, ECoG, EEG, and MEG Signals With LFPy 2.0. Front. Neuroinform. 12:92. doi: 10.3389/fninf.2018.00092

[1] Huang, Parra, Haufe (2016) The New York Head—A precise standardized volume conductor model for EEG source localization and tES targeting. Neuroimage 140:150–162. doi: 10.1016/j.neuroimage.2015.12.019

[2] Naess et al. (2020) Biophysical modeling of the neural origin of EEG and MEG signals. bioRxiv 2020.07.01.181875. doi: 10.1101/2020.07.01.181875

[1] Nunez and Srinivasan, Oxford University Press, 2006

[1] Blagoev et al. (2007) Modelling the magnetic signature of neuronal tissue. NeuroImage 37 (2007) 137–148 DOI: 10.1016/j.neuroimage.2007.04.033

[1] Hämäläinen M., et al., Reviews of Modern Physics, Vol. 65, No. 2, April 1993.

[2] Sarvas J., Phys.Med. Biol., 1987, Vol. 32, No 1, 11-22.

[1] Blagoev et al. (2007) Modelling the magnetic signature of neuronal tissue. NeuroImage 37 (2007) 137–148 DOI: 10.1016/j.neuroimage.2007.04.033

[1] Pettersen KH, Hagen E, Einevoll GT (2008) Estimation of population firing rates and current source densities from laminar electrode recordings. J Comput Neurosci (2008) 24:291–313. DOI 10.1007/s10827-007-0056-4

[2] Hagen E, Fossum JC, Pettersen KH, Alonso JM, Swadlow HA, Einevoll GT (2017) Journal of Neuroscience, 37(20):5123-5143. DOI: https://doi.org/10.1523/JNEUROSCI.2715-16.2017

PYTHON MODULE INDEX

INDEX

I

L

M

N

O

P

R

S

T

U

V