LFPy Homepage

Release 2.1.1

LFPy-team

Sep 04, 2020

CONTENTS

(Looking for the old LFPy documentation? Follow link)

LFPy is a Python package for calculation of extracellular potentials from multicompartment neuron models and recurrent networks of multicompartment neurons. It relies on the NEURON simulator and uses the Python interface it provides.

Active development of LFPy, as well as issue tracking and revision tracking, relies on GitHub.com and git (git-scm.com). Clone LFPy on GitHub.com: `git clone https://github.com/LFPy/LFPy.git`

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 or NeuroML2 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]

In previous versions (v1.x.x), LFPy was mainly designed for simulation of single neurons, but the forward modeling scheme is in general applicable to neuronal populations. These aspects, and the biophysical assumptions behind LFPy is described in our paper on the package appearing in Frontiers in Neuroinformatics, entitled "LFPy: A tool for biophysical simulation of extracellular potentials generated by detailed model neurons", appearing as part of the research topic "Python in Neuroscience II".

Since v2, LFPy also supports networks of multicompartment neurons, calculations of current-dipole moments, and predictions of both electric potentials and magnetic signals from a series of different volume-conductor models, as described in the bioRXiv preprint "Multimodal modeling of neural network activity: computing LFP, ECoG, EEG and MEG signals with LFPy2.0" by Espen Hagen, Solveig Næss, Torbjørn V Ness, Gaute T Einevoll, found at https://doi.org/10.1101/281717.

Citations:

- 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, at the Norwegian University of Life Sciences , in collaboration with the Laboratory of Neuroinformatics, Nencki Institute of Experimental Biology, Warsaw, Poland, and Center for Integrative Neuroplasticity (CINPLA) at the University of Oslo, Norway. The effort was supported by International Neuroinformatics Coordinating Facility (INCF), The Research Council of Norway (eScience, NevroNor, COBRA) and EU-FP7 (BrainScaleS).

CHAPTER

ONE

TUTORIAL SLIDES ON LFPY

- Slides for OCNS 2018 meeting tutorial T5: Modeling and analysis of extracellular potentials hosted in Seattle, USA on LFPy: CNS2018_LFPy_tutorial.pdf
- Slides for OCNS 2017 meeting tutorial T4: Modeling and analysis of extracellular potentials hosted in Antwerp, Belgium on LFPy and hybridLFPy: CNS2017_LFPy_tutorial.pdf
- Slides from OCNS 2015 meeting tutorial T2: Modeling and analysis of extracellular potentials hosted in Prague, Czech Republic on LFPy and hybridLFPy: CNS2015_LFPy_tutorial.pdf
- Slides from OCNS 2014 meeting tutorial T4: Modeling and analysis of extracellular potentials hosted in Quebec City: hybridlfpy_tutorial_OCNS2014.pdf
- As part of the OCNS 2013 meeting workshop Modeling and interpretation of extracellular potentials, there was also a talk on LFPy. The slides can be found here: lfpy-tutorial_OCNS2013.pdf.

CHAPTER

TWO

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

CHAPTER

THREE

CONTENTS

3.1 Download LFPy

Downlad the latest stable version of LFPy from the Python Package Index: http://pypi.python.org/pypi/LFPy

Or, download the development version of LFPy using git from GitHub.com into a local folder:

```
cd <where to put repositories>
git clone https://github.com/LFPy/LFPy.git
```

The LFPy source code and examples is then found under "LFPy".

The stable versions of LFPy can be accessed by listing and checking out tags, e.g.,

```
cd <path to LFPy>
git tag -l
git checkout <tag name>
```

To browse the documentation and source codes online, see http://lfpy.readthedocs.io/classes.html or https://github.com/LFPy/LFPy

3.2 Developing LFPy

As development of LFPy is now moved onto GitHub (https://github.com/LFPy), one can now fully benefit on working with forks of LFPy, implement new features, and share code improvements through pull requests. We hope that LFPy can be improved continuously through such a collaborative effort.

A list of various LFPy issues, bugs, feature requests etc. is found here. If you want to contribute code fixes or new features in LFPy, send us a pull request.

3.3 Getting started

3.3.1 Dependencies

To install LFPy you will need the following:

1. Python. LFPy's unit-test suite is integrated with and continuously tested using Travis-CI. Tests are run using NEURON >7.6.4 and Python 3.6, 3.7 and 3.8, as well as other Python dependencies listed next. The code build testing status, results of the last test, test coverage test using Coveralls and documentation status can be seen here:

2. Python modules setuptools, numpy, scipy, matplotlib, h5py, mpi4py as well as Cython

   Depending on how LFPy is obtained, missing dependencies will be installed automatically (using `pip`). Manual installation of dependencies can be done using `pip` and the `requirements.txt` file which is supplied with the LFPy source codes (since v2.0). See next section for details.

3. Some example files may rely on additional Python dependencies. If some examples should fail to run due to some `ImportError`, identify the missing dependency name and run

   ```
   pip install <package>
   ```

   Alternatively, use the system default package manager, for example

   ```
   sudo apt install python-<package> # or
   conda install package
   ```

4. NEURON compiled as a Python module, so the following should execute without error in Python console:

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

   If this step fails, see the next section.

   LFPy now requires NEURON 7.6.4 or newer, and should be compiled with MPI in order to support new parallel functionality in LFPy, i.e., execution of networks in parallel.

5. Cython (C-extensions for python) to speed up simulations of extracellular fields. Tested with version > 0.14, and known to fail with version 0.11. LFPy works without Cython, but simulations may run slower and is therefore not recommended.

3.3.2 Installing LFPy

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 source:

```
tar -xzf LFPy-x.x.tar.gz
cd LFPy-x.x
(sudo) pip install -r requirements.txt (--user) # install dependencies
(sudo) python setup.py install (--user)
```

3. From development version in our git (https://git-scm.com) repository:

```
git clone https://github.com/LFPy/LFPy.git
cd LFPy
# git tag -l # list versions
# git checkout <tag name> # choose particular version
(sudo) pip install -r requirements.txt (--user) # install dependencies
(sudo) python setup.py install (--user)
```

4. If you only want to have LFPy in one place, you can also build the LFPy Cython and NEURON NMODL extensions in the source directory. That can be quite useful for active LFPy development.

```
python setup.py develop --user
```

   With any changes in LFPy `*.pyx` source files, rebuild LFPy.

   In case of problems, it may be necessary to remove temporary and compiled files from the git repository before new attempts at building LFPy can be made:

```
git clean -n # list files that will be removed
git clean -fd # remove files
```

In a fresh terminal and python-session you should now be able to issue:

```
import LFPy
```

3.3.3 Uninstalling LFPy

Some times it may be necessary to remove installed versions of LFPy. Depending on how LFPy was installed in the first place, it should under most circumstances suffice to execute

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

If several versions was installed in the past, repeat until no more LFPy files are found.

3.4 Documentation

To generate the html documentation using Sphinx, issue from the LFPy source code directory:

```
sphinx-build -b html <path to LFPy>/doc <path to output>
```

The main html file is in `<path to output>/index.html`. Numpydoc and the ReadTheDocs theme may be needed:

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

3.4.1 Installing NEURON with Python

For most users, and even though NEURON (http://neuron.yale.edu) provides a working Python interpreter, making NEURON work as a Python module may be quite straightforward using pre-built Python distributions such as the Anaconda Scientific Python distribution (http://continuum.io) or Enthought Canopy (https://www.enthought.com/products/canopy/). We here provide some short step-by-step recipes on how to set up a working Python environment using Anaconda with the standard pre-built NEURON binaries on Linux, OSX and Windows.

macOS/linux with Anaconda Scientific Python distribution

Neuron and LFPy is now installable from the conda-forge (https://conda-forge.org), which makes installation a breeze. Download and install Anaconda using the 64-bit installer from https://www.anaconda.com/download. In order to install LFPy in the current (e.g., *base*) environment, issue in the terminal:

```
conda config --add channels conda-forge
conda install lfpy
```

Complete LFPy environments can be created using the *conda_environment_ubuntu.yml* or *conda_environment_macos.yml* files found in the root of the LFPy source code tree:

```
git clone https://github.com/LFPy/LFPy.git
cd LFPy
conda env create -f conda_environment_macos.yml
conda activate lfpy
python setup.py install  # install development version in the current environment
```

Note that other useful packages like `ipython`, `pandas`, `jupyter` etc. must be installed separately, as:

```
conda install <package-name>
```

Ubuntu 18.04.1 LTS 64-bit with Anaconda Scientific Python distribution

Another solution avoiding source code compilation. This recipe was tested on a clean installation of Ubuntu 18.04.1 running in VirtualBox (https://www.virtualbox.org). The VirtualBox guest additions were installed.

1. Download and install Anaconda using the 64-bit Linux installer script from https://www.anaconda.com/download (https://repo.anaconda.com/archive/Anaconda3-5.3.1-Linux-x86_64.sh)

   Open Terminal application, then issue:

   ```
   cd $HOME/Downloads
   bash Anaconda3-5.3.1-Linux-x86_64.sh
   ```

   Accept the license and default options. Allow the installer to modify your `.bashrc` file. Installing MS Visual Studio Code (VSCode) is optional.

2. Download and install the 64-bit Debian/Ubuntu .deb file with NEURON from https://www.neuron.yale.edu/neuron/download (https://neuron.yale.edu/ftp/neuron/versions/v7.6/7.6.2/nrn-7.6.2.x86_64-linux-py-36-35-27.deb)

3. The `readline`, `ncurses`, `libtool` library files as well as `make` may be needed by NEURON. Install them from the terminal calling

   ```
   sudo apt install libreadline-dev libncurses-dev libtool make
   ```

4. Edit your `.bashrc` or similar file located in the `$HOME` folder, e.g., by calling in the terminal `gedit $HOME/.bashrc`, to include the lines:

```
# make NEURON python module available to Anaconda python
export PYTHONPATH="/usr/local/nrn/lib/python/:$PYTHONPATH"
```

5. Open a fresh terminal window (or type `source ~/.bashrc` in the terminal)
6. Activate the `base` conda environment:

```
conda activate
```

7. Install LFPy dependencies (not installed by default) using conda

```
conda install mpi4py # numpy matplotlib scipy h5py
```

8. Clone into LFPy using Git (https://git-scm.com), installable by calling `sudo apt install git` in the terminal:

```
git clone https://github.com/LFPy/LFPy.git
cd LFPy
```

9. Build LFPy from source inplace (without moving files)

```
python setup.py develop --user
```

or perform a local installation of LFPy:

```
python setup.py install --user
```

9. Test the installation from the terminal

```
py.test
```

which will run through the LFPy test suite. Hopefully without errors.

Ubuntu 18.04.1 LTS w. system Python 3.6

Another option is to rely on the system Python installation (Python 3.6.7). Make sure that nothing in `$PATH` points to an Anaconda-installed version of Python (may break e.g., `pip3`)

1. Install dependencies through the terminal:

```
sudo apt install libreadline-dev libncurses-dev libtool make
sudo apt install ipython3 cython3 jupyter-notebook python3-pytest \
  python3-numpy python3-scipy python3-matplotlib python3-mpi4py python3-h5py
```

2. Download and install the 64-bit Debian/Ubuntu .deb file with NEURON from https://www.neuron.yale.edu/neuron/download (https://neuron.yale.edu/ftp/neuron/versions/v7.6/nrn-7.6.x86_64-linux.deb)
3. Edit your `.bashrc` or similar file located in the `$HOME` folder, e.g., by calling in the terminal `gedit $HOME/.bashrc`, to include the lines:

```
# make NEURON python module available to Anaconda python
export PYTHONPATH="/usr/local/nrn/lib/python/:$PYTHONPATH"
```

4. Open a fresh terminal window (or type `source ~/.bashrc` in the terminal)

5. Clone into LFPy using Git (https://git-scm.com), installable by calling `sudo apt install git` in the terminal:

```
git clone https://github.com/LFPy/LFPy.git
cd LFPy
```

6. Build LFPy from source inplace (without moving files)

```
python3 setup.py develop --user
```

   or perform a local installation of LFPy:

```
python3 setup.py install --user
```

7. Test the installation from the terminal

```
py.test-3
```

   which will run through the LFPy test suite. Hopefully without errors.

OSX 10.12.x with Anaconda Scientific Python distribution

Another option avoiding source code compilation.

1. Download and install Anaconda using the 64-bit graphical installer from http://continuum.io/downloads
2. Download and install the 64-bit Mac `.pkg` file with NEURON from http://www.neuron.yale.edu/neuron/download. Do not choose to let the NEURON installer edit the `~/.bash_profile` file. The default file to edit is `~/.profile` (see below).
3. Edit your .profile or similar file located in the $HOME folder, e.g., by calling in the Terminal.app `open -t $HOME/.profile`, to include the lines:

```
# make nrniv, mknrnivmodl, etc. available from the command line
export PATH=/Applications/NEURON-7.5/nrn/x86_64/bin/:$PATH

# Append the path to the NEURON python extension module to PYTHONPATH
export PYTHONPATH=/Applications/NEURON-7.5/nrn/lib/python:$PYTHONPATH
```

4. Open a fresh terminal window
5. Install LFPy dependencies (not installed by default) using conda

```
conda install mpi4py
```

6. Clone into LFPy using Git:

```
git clone https://github.com/LFPy/LFPy.git
```

7. Build LFPy from source (without moving files)

```
python setup.py develop
```

8. Test the installation from the terminal

```
python -c "import LFPy"
NEURON -- VERSION 7.5 master (6b4c19f) 2017-09-25
Duke, Yale, and the BlueBrain Project -- Copyright 1984-2016
See http://neuron.yale.edu/neuron/credits
```

If everything worked, you now have a working Python/NEURON/LFPy environment.

Windows with Anaconda Scientific Python distribution

Windows 10 Pro/Education (64-bit) install instructions:

1. Download and install Anaconda Python from https://www.anaconda.com/download.
2. Download and install NEURON from https://www.neuron.yale.edu/neuron/download. Tick the box to "Set DOS environment" (Otherwise Anaconda Python will not find the NEURON python module)
3. Download and install the Visual Studio C++ Build Tools 2015 from: https://www.microsoft.com/en-us/download/details.aspx?id=48159.
4. Download and install Git from https://git-scm.com/downloads
5. Download and install Microsoft MPI from the Official Microsoft Download Center: https://www.microsoft.com/en-us/download/details.aspx?id=55494. Choose the file "MSMpiSetup.exe".
6. Open the Anaconda Prompt under the Anaconda* folder in the start menu
7. Optionally, create a separate conda environment for LFPy:

   ```
   conda create -n LFPy python=3.6 mpi4py numpy scipy matplotlib h5py Cython jupyter
   conda activate LFPy
   ```

   For every future session, the LFPy environment needs to be used by issuing `conda activate LFPy`. From hereon, skip to step 9.
8. Install additional LFPy dependencies listed in `requirements.txt` using `conda` (to avoid package clashes with i.e., `pip install <package_name>`)

   ```
   conda install mpi4py
   ```
9. Clone into LFPy using Git:

   ```
   git clone https://github.com/LFPy/LFPy.git
   ```
10. Build LFPy from source (without moving files)

    ```
    python setup.py develop
    ```
11. NEURON NMODL (.mod) files will not be autocompiled when building LFPy as on MacOS/Linux, as the mknrndll script cannot be run directly in the Anaconda Prompt. To fix this, run the `bash` file in the NEURON program group, change directory within "bash" to the `<LFPy>/LFPy/test` folder, then run `mknrndll`

3.4.2 Installing NEURON with Python from source

Some users have difficulties installing NEURON as a Python module, depending on their platform. We will provide some explanations here, and otherwise direct to the NEURON download pages; https://www.neuron.yale.edu/neuron/download and https://www.neuron.yale.edu/neuron/download/getstd. The NEURON forum (https://www.neuron.yale.edu/phpBB/) is also a useful resource for installation problems.

Dependencies: Ubuntu 18.04 LTS and other Debian-based Linux versions

The instructions below show how to meet all the requirements starting from a clean Ubuntu 18.4 for the installation of NEURON from the development branch.

Start by installing required packages. We aim to link with the system Python installation, not Anaconda Python. For Anaconda installations, make sure that the correct Python installations is found during NEURON's configure step below.

```
sudo apt install git build-essential autoconf libtool
sudo apt install libxext-dev libncurses-dev zlib1g-dev
sudo apt install bison flex libx11-dev
sudo apt install openmpi-bin libopenmpi-dev
sudo apt install python3-dev python3-numpy python3-scipy python3-matplotlib
sudo apt install ipython3 cython3
```

Linux/Unix installation of NEURON from source

Fetch the source code of NEURON using git

```
cd $HOME
mkdir neuron
cd neuron

git clone https://github.com/neuronsimulator/iv.git
git clone https://github.com/neuronsimulator/nrn.git
```

Set compilers, here using the GNU Compiler Collection (GCC). A compute cluster may have the Intel Compiler binaries (`icc/icpc/mpiicc/mpiicpc`)

```
export CC=gcc
export CXX=g++
export MPICC=mpicc
export MPICXX=mpicxx
```

Optional, compile and install InterViews binaries to the folder `$HOME/.local` folder (which should be in the default `$PATH` on most systems)

```
cd iv
sh build.sh
./configure --prefix=~/.local
make -j4
make install
```

Compile and install NEURON with InterViews and MPI. To disable InterViews, use `--without-iv` during the configuration step.

```
cd ../nrn
sh build.sh
./configure --prefix=~/.local --with-iv=~/.local --with-nrnpython=/usr/bin/python3.6 -
↪-with-mpi=/usr/bin/mpirun --with-paranrn
make -j4
make install
```

You might want to add the folder with NEURON binaries to your `$PATH`, so that you can easily compile NEURON mechanisms using `nrnivmodl` from the terminal. Add the following line to your `$HOME/.bashrc` (or equivalent) file:

```
export PATH=$HOME/.local/x86_64/bin:$PATH
```

Start a new terminal tab or type `source $HOME/.bashrc` to activate.

Install NEURON as a Python module

```
cd src/nrnpython/
python3 setup.py install --user
```

Now you should be able to `import neuron` from Python console and run a small test with success;

```
cd $HOME
ipython3
>>> import neuron
>>> neuron.test()
```

NEURON dependencies and installation on Mac OSX from source

Most of the development work and testing of LFPy has been done on MacOS (10.6-). Our preferred way of building Python has been through MacPorts; http://www.macports.org. Here is an step-by-step explanation on how to compile NEURON agains that installation of Python. Simpler solutions are stipulated above.

To start using MacPorts, follow the instructions on http://www.macports.org/install.php.

Building a python 2.7 environment using MacPorts issue in Terminal:

```
sudo port install python27 py27-ipython py27-numpy py27-matplotlib py27-scipy py27-
↪cython py27-mpi4py py27-h5py
```

Make the installed Python and IPython default:

```
sudo port select --set python python27
sudo port select --set ipython ipython27
```

Install the necessary packages for cloning into repository and compiling NEURON:

```
sudo port install automake autoconf libtool xorg-libXext ncurses mercurial bison flex
```

Install NEURON from the bleeding edge source code. The following recipe assumes a 64 bit build of NEURON and Python on MacOS 10.12, so change "x86_64-apple-darwin16.7.0" throughout to facilitate your system accordingly, as found by running `./config.guess` in the root of the NEURON source code folder;

```
#create a directory in home directory
cd $HOME
mkdir nrn64
```

(continues on next page)

(continued from previous page)

```
cd nrn64

#creating directories
sudo mkdir /Applications/NEURON-7.5
sudo mkdir /Applications/NEURON-7.5/iv
sudo mkdir /Applications/NEURON-7.5/nrn

#Downloading bleeding edge source code
hg clone http://www.neuron.yale.edu/hg/neuron/iv
hg clone http://www.neuron.yale.edu/hg/neuron/nrn
cd iv

#compiling and installing IV under folder /Applications/nrn7.5
sh build.sh
./configure --prefix=/Applications/NEURON-7.5/iv \
        --build=x86_64-apple-darwin16.7.0 --host=x86_64-apple-darwin16.7.0 \
        --x-includes=/usr/X11/include --x-libraries=/usr/X11/lib
make
sudo make install

#Building NEURON with InterViews, you may have to alter the path --with-nrnpython=/
↪python-path
cd $HOME/nrn64/nrn
sh build.sh
./configure --prefix=/Applications/NEURON-7.5/nrn \
        --with-nrnpython=/opt/local/Library/Frameworks/Python.framework/Versions/2.7/
↪Resources/Python.app/Contents/MacOS/Python \
        --host=x86_64-apple-darwin16.7.0 --build=x86_64-apple-darwin16.7.0 \
        --with-paranrn \
        --with-mpi \
        --with-iv=/Applications/NEURON-7.5/iv \
        CFLAGS='-O3 -Wno-return-type -Wno-implicit-function-declaration -Wno-implicit-
↪int -fPIC' \
        CXXFLAGS='-O3 -Wno-return-type -fPIC'
make
sudo make install
sudo make install after_install

#You should now have a working NEURON application under Applications. Small test;
#sudo /Applications/NEURON-7.5/nrn/x86_64/bin/neurondemo

#Final step is to install neuron as a python module
cd src/nrnpython
sudo python setup.py install
```

3.5 LFPy on the Neuroscience Gateway Portal

LFPy is installed on the Neuroscience Gateway Portal (NSG, see http://www.nsgportal.org), and can be used to execute simulations with LFPy both serially and in parallel applications on high-performance computing facilities. The access to the NSG is entirely free, and access to other neuronal simulation software (NEST, NEURON, etc.) is also provided. The procedure for getting started with LFPy on the NSG is quite straightforward through their web-based interface:

1. First, apply for a NSG user account by filling out their application form and sending it by email (follow instructions on http://www.nsgportal.org/portal2)

2. After approval, log in using your credentials, change password if necessary
3. As a first step after log in, create a new folder, e.g., named "LFPyTest" and with some description. This will be the home for your input files and output files, and should contain empty Data and Tasks folders
4. Press the "Data (0)" folder in the left margin. Press the "Upload/Enter Data" button, showing the Upload File interface. Add a label, e.g., "LFPyTest".
5. Next, LFPy simulation files have to be uploaded. As an example, download the example LFPy files https://github.com/espenhgn/LFPy/blob/master/examples/nsg_example/L5_Mainen96_wAxon_LFPy.hoc and https://github.com/espenhgn/LFPy/blob/master/examples/nsg_example/nsg_example.py into a new local folder "nsg_example". Modify as needed.
6. Zip the "nsg_example" folder, upload it to the NSG (cf. step 4) and press "Save"
7. Press "Tasks (0)" in the left margin and "Create New Task"
8. Enter some Description, e.g., "LFPyTest", and "Select Input Data". Hook off "LFPyTest" and press "Select Data"
9. Next, press "Select Tool", and then "Python (2.7.x)"
10. Then, go to the "Set Parameters" tab. This allows for specifying simulation time, main simulation script, and number of parallel threads. Set "Maximum Hours" to 0.1, and "Main Input Python Filename" to "nsg_example.py". Node number and number of cores per node should both be 1. Press "Save Parameters"
11. Everything that is needed has been set up, thus "Save and Run Task" in the Task Summary tab is all that is needed to start the job, but expect some delay for it to start.
12. Once the job is finished, you will be notified by email, or keep refreshing the Task window. The simulation output can be accessed through "View Output". Download the "output.tar.gz" file and unzip it. Among the output files, including stdout.txt and stderr.txt text files and jobscript details, the included folder "nsg_example" will contain the input files and any output files. For this particular example, only a pdf image file is generated, "nsg_example.pdf"

3.6 LFPy Tutorial

This tutorial will guide you through a few first steps with LFPy. It is based on `example1.py` in the `LFPy/examples` folder. In order to obtain all necessary files, please obtain the LFPy source files as described on the Download page

Change directory to the LFPy examples folder, and start the interactive IPython interpreter

```
$ cd <path to LFPy>/examples
$ ipython
```

Let us start by importing `LFPy`, as well as the `numpy` and `os` modules

```
>>> import os
>>> import LFPy
>>> import numpy as np
```

Then we define a dictionary which describes the properties of the cell we want to simulate

```
>>> cell_parameters = {
>>>     'morphology' : os.path.join('morphologies', 'L5_Mainen96_LFPy.hoc'),     #␣
↪Mainen&Sejnowski, Nature, 1996
>>>     'tstart' : 0.,          # start time of simulation, recorders start at t=0
```

(continues on next page)

(continued from previous page)

```
>>>     'tstop' : 20.,          # stop simulation at 20 ms.
>>>     'v_init' : -70.0,       # initial voltage at tstart
>>>     'Ra' : 35.4,            # axial resistivity
>>>     'cm' : 1.0,             # membrane capacitance
>>>     'passive' : True,       # switch on passive leak resistivity
>>>     'passive_parameters' : {'e_pas': -70.0, # passive leak reversal potential
>>>                             'g_pas': 0.001} # passive leak conductivity
>>> }
```

The only mandatory entry is `morphology`, which should point to a `hoc` file specifying the neuron's morphology. Here we also set the start and end times (in milliseconds). Many more options are available (such as adding custom NEURON mechanisms), but for now we leave other parameters at their default values.

Then, the Cell object is created issuing

```
>>> cell = LFPy.Cell(**cell_parameters)
```

Let us now add a synapse to the cell. Again, we define the synapse parameters first

```
>>> synapse_parameters = {
>>>     'idx' : cell.get_closest_idx(x=0., y=0., z=800.), # segment index for synapse
>>>     'e' : 0.,                   # reversal potential
>>>     'syntype' : 'ExpSyn',       # synapse type
>>>     'tau' : 2.,                 # syn. time constant
>>>     'weight' : .1,              # syn. weight
>>>     'record_current' : True           # record syn. current
>>> }
```

and create a Synapse object connected to our cell

```
>>> synapse = LFPy.Synapse(cell, **synapse_parameters)
```

Let us assume we want the synapse to be activated by a single spike at t = 5 ms. We have to pass the spike time to the synapse using

```
>>> synapse.set_spike_times(np.array([5.]))
```

We now have a cell with a synapse, and we can run the simulation

```
>>> cell.simulate(rec_imem=True)
```

Note the `rec_imem=True` argument, this means that the transmembrane currents will be saved - we need the currents to calculate the extracellular potential. An array with all transmembrane currents for the total number of segments `cell.totnsegs` is now accessible as `cell.imem`, the somatic voltage as `cell.tvec`, all with time stamps `cell.tvec`.

```
>>> print(cell.tvec.shape, cell.somav.shape, cell.totnsegs, cell.imem.shape)
```

The final element is the extracellular recording electrode. Again, we start by defining the parameters

```
>>> electrode_parameters = {
>>>     'sigma' : 0.3,   # extracellular conductivity
>>>     'x' : np.array([0]),
>>>     'y' : np.array([0]),
>>>     'z' : np.array([50])
>>> }
```

Here we define a single electrode contact at x = 0, y = 0, z = 50 µm, but a whole array of electrodes can be specified by passing array arguments.

The electrode (recording from `cell`) is created using

```
>>> electrode = LFPy.RecExtElectrode(cell, **electrode_parameters)
```

Finally, we calculate the extracellular potential at the specified electrode location

```
>>> electrode.calc_lfp()
```

The resulting LFP is stored in `electrode.LFP`.

```
>>> print(electrode.LFP.shape)
```

Finally, the cell morphology and synapse location can be plotted

```
>>> from matplotlib.collections import PolyCollection
>>> import matplotlib.pyplot as plt
>>> zips = []
>>> for x, z in cell.get_idx_polygons(projection=('x', 'z')):
>>>     zips.append(zip(x, z))
>>> polycol = PolyCollection(zips,
>>>                          edgecolors='none',
>>>                          facecolors='gray')
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.add_collection(polycol)
>>> ax.plot(cell.xmid[synapse.idx], cell.zmid[synapse.idx], 'ro')
>>> ax.axis(ax.axis('equal'))
>>> plt.show()
```

As well as the simulated output

```
>>> fig = plt.figure()
>>> plt.subplot(311)
>>> plt.plot(cell.tvec, synapse.i)
>>> plt.subplot(312)
>>> plt.plot(cell.tvec, cell.somav)
>>> plt.subplot(313)
>>> plt.plot(cell.tvec, electrode.LFP.T)
>>> plt.show()
```

3.6.1 More examples

More examples of LFPy usage are provided in the `trunk/examples` folder in the source code release, displaying different usages of LFPy.

The examples rely on files present inside the examples folder, such as morphology files (.hoc) and NEURON NMODL (.mod) files.

The easiest way of accessing all of these files is cloning the examples directory using git (https://git-scm.com):

```
$ git clone https://github.com/LFPy/LFPy.git
$ cd LFPy/examples
```

The files provided are

- example1.py
- example2.py
- example3.py
- example4.py
- example5.py
- example6.py
- example7.py
- example8.py
- example_mpi.py
- example_EPFL_neurons.py
- example_LFPyCellTemplate.py
- example_MEA.py
- example_anisotropy.py
- example_loadL5bPCmodelsEH.py
- example_network/example_Network.py
- example_EEG.py

3.7 Notes on LFPy

3.7.1 Morphology files

Cell morphologies can be specified manually in a `.hoc` file. For a simple example, see `examples/morphologies/example_morphology.hoc`. Note the following conventions:

- Sections should be named according to the following scheme:
  - `soma*[]` for somatic sections, `*` is optional
  - `dend*[]` for dendritic sections
  - `apic*[]` for apical dendrite sections
  - `axon*[]` for axonal sections
- Sections must be defined as types Section or SectionList (as `soma[1]` or `soma`)

Also the morphologies exported from the NEURON simulator (for example using Cell Builder -> Export) should work with LFPy, but some times `create soma` may have to be corrected to `create soma[1]` directly in the files. Multi-sectioned somas may occurr e.g., due to faulty conversion from NeuroLucida or SWC format, however, we recommend that these files are corrected. It may not affect the predictions of intracellular voltages, but have implications for predictions of extracellular potentials. We usually assume that the soma is a single compartment and that it is the root section for all other sections.

NEURON convention for creating morphology files in `hoc`:

```
/* -------------------------------------------------
example_morphology.hoc

This hoc file creates a neuron of the following shape:

            \
             \
              \    /
               \  /
                V
                |
                |
                |
                O

Note the conventions:
 - use soma for the soma compartment,
 - use a name starting with dend or apic for the dendrites.
-------------------------------------------------------*/


create soma[1]
create dend[3]

soma[0] {
    pt3dadd(0, 0, 0, 25)
    pt3dadd(0, 0, 35, 25)
}

dend[0] {
    pt3dadd(0, 0, 35, 5)
    pt3dadd(0, 0, 150, 5)
}

dend[1] {
    pt3dadd(0, 0, 150, 2)
    pt3dadd(-50, 20, 200, 1)
}

dend[2] {
    pt3dadd(0, 0, 150, 2)
    pt3dadd(30, 0, 160, 2)
}

connect dend[0](0), soma[0](1)
connect dend[1](0), dend[0](1)
connect dend[2](0), dend[0](1)
```

Other file formats

Support for SWC, NeuroLucida3 and NeuroML morphology file formats is added in LFPy, but consider this an experimental feature as the functionality is not properly tested. If there is something wrong with the files, strange behaviour may occur or LFPy may even fail to load the desired morphology.

3.7.2 Using NEURON NMODL mechanisms

Custom NEURON mechanisms can be loaded from external `.hoc`- or `.py`-files - see `example2.py` and `example3.py`. Python function definitions with arguments can also be given as input to the `Cell`-class, specifying model specific conductances etc. Remeber to compile any `mod` files (if needed) using `nrnivmodl` (or `mknrdll` on Windows).

These model specific declarations is typically run after the `Cell`-class try to read the morphology file, and before running the `_set_nsegs()` and `_collect_geometry()` procedures. Hence, code that modifies the segmentation of the morphology will affect the properties of the instantiated `LFPy.Cell` object.

3.7.3 Units

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

Quantity	Unit
Potential	[mV]
Current	[nA]
Conductance	[S/cm2]
Extracellular conductivity	[S/m]
Capacitance	[F/cm2]
Dimension	[m]
Synaptic weight	[µS]
Current dipole moment	[nA µm]
Magnetic field (H)	[nA/µm]
Permeability (µ)	[T m/A]

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.

3.7.4 Contributors

LFPy was developed by (as per commit):

- Henrik Lindén https://lindenh.wordpress.com
- Espen Hagen http://www.mn.uio.no/fysikk/english/?vrtx=person-view&uid=espehage
- Szymon Łęski
- Torbjørn V. Ness
- Solveig Næss
- Alessio Buccino
- Eivind Norheim

- Klas H. Pettersen http://www.med.uio.no/imb/english/?vrtx=person-view&uid=klashp
- Gaute T. Einevoll https://www.nmbu.no/ans/gaute.einevoll

3.7.5 Contact

If you want to contact us with questions, bugs and comments, please create an issue on GitHub.com/LFPy/LFPy/issues. We are of course happy to receive feedback of any kind.

3.8 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 - The pythonic neuron object itself laying on top of NEURON representing cells
- TemplateCell - Similar to Cell, but for models using cell templates
- Synapse - Convenience class for inserting synapses onto Cell objects
- StimIntElectrode - Convenience class for inserting electrodes onto Cell objects
- PointProcess - Parent class of Synapse and StimIntElectrode
- RecExtElectrode - Class for performing simulations of extracellular potentials
- RecMEAElectrode - Class for performing simulations of in vitro (slice) extracellular potentials
- Network - Class for creating distributed populations of cells and handling connections between cells in populations
- NetworkCell - Similar to *TemplateCell* with some attributes and methods for spike communication between parallel RANKs
- NetworkPopulation - Class representing group of Cell objects distributed across MPI RANKs
- OneSphereVolumeConductor - For computing extracellular potentials within and outside a homogeneous sphere
- FourSphereVolumeConductor - For computing extracellular potentials in 4-sphere 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 - Functions used by RecExtElectrode class
- tools - Some convenient functions
- inputgenerators - Functions for synaptic input time generation
- eegcalc - Functions for calculating current dipole moment vector P and P_tot from currents and distances.
- run_simulations - Functions to run NEURON simulations

3.8.1 class Cell

class LFPy.**Cell**(*morphology*, *v_init=-70.0*, *Ra=None*, *cm=None*, *passive=False*, *passive_parameters={'e_pas': -70.0, 'g_pas': 0.001}*, *extracellular=False*, *tstart=0.0*, *tstop=100.0*, *dt=0.0625*, *nsegs_method='lambda100'*, *lambda_f=100*, *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 [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

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

To save a cell, use: >>> cell.cellpickler('cell.cpickle') To load this cell again in another session: >>> import cPickle >>> f = file('cell.cpickle', 'rb') >>> cell = cPickle.load(f) >>> f.close() alternatively: >>> import LFPy >>> cell = LFPy.tools.load('cell.cpickle')

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 'extracellular' mechanism. Extracellular potentials are computed from the electrode currents using the pointsource approximation. If 'model' is 'inf' (default), potentials are computed as (r_i is the position of a comparment i, r_e is the position of an elextrode e, 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 im ms corrisponding to step changes in the provided currents. If None, currents are assumed to have the same time steps as NEURON simulation.

n: int Points per electrode to compute 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.

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 ((cell.totnsegs-1)*2, 3) 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'.

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

```
>>> 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]"

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

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

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]"

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')

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

The most efficient way of using this would be something like >>> 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(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

Will return a float in unit of micrometers.

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 vector [x, y, z], where x = self.xmid[idx1] - self.xmid[idx0] etc. Parameters ———- idx0 : int idx1 : int

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

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.

multi_dipoles [ndarray, dtype = float] Shape (n_axial_currents, n_timepoints, 3) 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).

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() Returns ——- list

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

```
>>> 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(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 depth filter

z_max: float depth filter

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)

```
>>> 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

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

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

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

```
>>> 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

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. rotation_order should be a string with 3 elements containing x, y, and z e.g. 'xyz', 'zyx' Input should be angles in radians. using rotation matrices, takes dict with rot. angles, where x, y, z are the rotation angles around respective axes. All rotation angles are optional. 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

simulate(*electrode=None*, *rec_imem=False*, *rec_vmem=False*, *rec_ipas=False*, *rec_icap=False*, *rec_current_dipole_moment=False*, *rec_variables=[]*, *variable_dt=False*, *atol=0.001*, *rtol=0.0*, *to_memory=True*, *to_file=False*, *file_name=None*, *dotprodcoeffs=None*, ***kwargs*)
This is the main function running the simulation of the NEURON model. Start NEURON simulation and record variables specified by arguments. Parameters ———- electrode : :obj: or list, optional

Either an LFPy.RecExtElectrode object or a list of such. If supplied, LFPs will be calculated at every time step and accessible as *electrode.LFP*. If a list of objects is given, accessible as *electrode[0].LFP* etc.

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_current_dipole_moment [bool] If True, compute and record current-dipole moment from transmembrane currents as in Linden et al. (2010) J Comput Neurosci, DOI: 10.1007/s10827-010-0245-4. Will set the *LFPy.Cell* attribute *current_dipole_moment* as n_timesteps x 3 *np.ndarray* where the last dimension contains the x,y,z components of the dipole moment.

rec_variables [list] List of variables to record, i.e arg=['cai',]

variable_dt [bool] Use variable timestep in NEURON

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 electrode, store lfp in -> electrode.LFP

to_file [bool] Only valid with electrode, save LFPs in hdf5 file format

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

dotprodcoeffs [list] List of N x Nseg ndarray. These arrays will at every timestep be multiplied by the membrane currents. Presumably useful for memory efficient csd or lfp calcs

strip_hoc_objects()
Destroy any NEURON hoc objects in the cell object

3.8.2 class TemplateCell

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

Bases: LFPy.cell.Cell

class LFPy.TemplateCell

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

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()
```

3.8.3 class NetworkCell

class LFPy.**NetworkCell**(***args*)

Bases: LFPy.templatecell.TemplateCell

class NetworkCell

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

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,
>>> }
```

(continues on next page)

(continued from previous page)

```
>>> cell = LFPy.NetworkCell(**cellParameters)
>>> cell.simulate()
```

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 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=<MagicMock name='mock.ExpSyn' id='140584091254608'>*, *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

3.8.4 class PointProcess

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

Bases: object

Superclass on top of Synapse, StimIntElectrode, just to import and set some shared variables and extracts Cartesian coordinates of a segment

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]

update_pos(*cell*)
Extract coordinates of point-process

3.8.5 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.

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. Built-in examples: ExpSyn, Exp2Syn

record_current [bool] Decides if current is recorded

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

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

collect_potential(*cell*)
Collect membrane potential of segment with synapse

set_spike_times(*sptimes=array([], dtype=float64)*)
Set the spike times explicitly using numpy arrays

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

3.8.6 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*

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[0]' : -70,
>>>         'dur[0]' : 10,
>>>         'amp[1]' : 0,
>>>         'dur[1]' : 20,
>>>         'amp[2]' : -70,
>>>         'dur[2]' : 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
>>> 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()
>>>      pl.subplot(211)
>>>      pl.plot(cell.tvec, stimulus.i, label=pointprocess['pptype'])
>>>      pl.legend(loc='best')
>>>      pl.title('Stimulus currents (nA)')
>>>      pl.subplot(212)
```

(continues on next page)

(continued from previous page)

```
>>>     pl.plot(cell.tvec, cell.somav, label=pointprocess['pptype'])
>>>     pl.legend(loc='best')
>>>     pl.title('Somatic potential (mV)')
```

collect_current(*cell*)
Fetch electrode current from recorder list

collect_potential(*cell*)
Collect membrane potential of segment with PointProcess

3.8.7 class RecExtElectrode

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

Bases: object

class RecExtElectrode

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

Parameters

cell [None or object] If not None, instantiation of LFPy.Cell, LFPy.TemplateCell 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] accordingly.

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', 'soma_as_point' to represent each compartment when computing extracellular potentials

from_file [bool] if True, load cell object from file

cellfile [str] path to cell pickle

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

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

Examples

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

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import LFPy
>>>
>>> cellParameters = {
>>>     'morphology' : 'examples/morphologies/L5_Mainen96_LFPy.hoc',  #␣
↪morphology file
>>>     '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)
>>>
>>> N = np.empty((16, 3))
>>> for i in xrange(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 = LFPy.RecExtElectrode(cell, **electrodeParameters)
>>> electrode.calc_lfp()
>>> plt.matshow(electrode.LFP)
>>> plt.colorbar()
>>> plt.axis('tight')
>>> plt.show()
```

Compute extracellular potentials during simulation (recommended):

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

(continues on next page)

(continued from previous page)

```
>>> import LFPy
>>>
>>> cellParameters = {
>>>     'morphology' : 'examples/morphologies/L5_Mainen96_LFPy.hoc',  #␣
↪morphology file
>>>     '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 xrange(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 = LFPy.RecExtElectrode(**electrodeParameters)
>>>
>>> cell.simulate(electrode=electrode)
>>>
>>> plt.matshow(electrode.LFP)
>>> plt.colorbar()
>>> plt.axis('tight')
>>> plt.show()
```

Use MEAutility to to handle probes

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import MEAutility as mu
>>> import LFPy
>>>
>>> cellParameters = {
>>>     'morphology' : 'examples/morphologies/L5_Mainen96_LFPy.hoc',  #␣
↪morphology file
```

(continues on next page)

(continued from previous page)

```
>>>     '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 = LFPy.RecExtElectrode(cell, probe=probe)
>>> electrode.calc_lfp()
>>> mu.plot_mea_recording(electrode.LFP, probe)
>>> plt.axis('tight')
>>> plt.show()
```

class RecMEAElectrode(*cell=None*, *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*, *perCellLFP=False*, *method='linesource'*, *from_file=False*, *cellfile=None*, *verbose=False*, *seedvalue=None*, *squeeze_cell_factor=None*, ***kwargs*)

Bases: `LFPy.recextelectrode.RecExtElectrode`

class RecMEAElectrode

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

Set-up:

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

Parameters

cell [None or object] If not None, instantiation of LFPy.Cell, LFPy.TemplateCell 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', 'soma_as_point' to represent each compartment when computing extracellular potentials

from_file [bool] if True, load cell object from file

cellfile [str] path to cell pickle

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

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

Examples

See also examples/example_MEA.py

>>> import numpy as np

>>> import matplotlib.pyplot as plt

>>> import LFPy

>>>

>>> cellParameters = {

>>> 'morphology' ['examples/morphologies/L5_Mainen96_LFPy.hoc', # morphology file]

>>> '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), # 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.]))
>>>
>>> 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]
>>> 'x' [np.linspace(0, 1200, 16), # electrode requires 1d vector of positions]
>>> 'y' [np.zeros(16),]
>>> 'z' [np.zeros(16),]
>>> "method": "pointsource",
>>> "h": 300,
>>> "squeeze_cell_factor": 0.5,
>>> }
>>> MEA = LFPy.RecMEAElectrode(cell, **MEA_electrode_parameters)
>>>
>>> cell.simulate(electrode=MEA)
```

>>>

>>> plt.matshow(MEA.LFP)

>>> plt.colorbar()

>>> plt.axis('tight')

>>> plt.show()

RecMEAElectrode.**calc_lfp**(*t_indices=None*, *cell=None*)

Calculate LFP on electrode geometry from all cell instances. Will chose distributed calculated if electrode contain 'n', 'N', and 'r'

Parameters

cell [obj, optional] *LFPy.Cell* or *LFPy.TemplateCell* instance. Must be specified here if it was not specified at the initiation of the *RecExtElectrode* class

t_indices [np.ndarray] Array of timestep indexes where extracellular potential should be calculated.

RecMEAElectrode.**calc_mapping**(*cell*)

Creates a linear mapping of transmembrane currents of each segment of the supplied cell object to contribution to extracellular potential at each electrode contact point of the RexExtElectrode object. Sets the class attribute "mapping", which is a shape (n_contact, n_segs) ndarray, such that the extracellular potential at the contacts phi = np.dot(mapping, I_mem) where I_mem is a shape (n_segs, n_tsteps) ndarray with transmembrane currents for each time step of the simulation.

Parameters

cell [obj] *LFPy.Cell* or *LFPy.TemplateCell* instance.

Returns

None

RecMEAElectrode.**test_cell_extent**()

Test if the cell is confined within the slice. If class argument "squeeze_cell" is True, cell is squeezed to to fit inside slice.

calc_lfp(*t_indices=None*, *cell=None*)

Calculate LFP on electrode geometry from all cell instances. Will chose distributed calculated if electrode contain 'n', 'N', and 'r'

Parameters

cell [obj, optional] *LFPy.Cell* or *LFPy.TemplateCell* instance. Must be specified here if it was not specified at the initiation of the *RecExtElectrode* class

t_indices [np.ndarray] Array of timestep indexes where extracellular potential should be calculated.

calc_mapping(*cell*)

Creates a linear mapping of transmembrane currents of each segment of the supplied cell object to contribution to extracellular potential at each electrode contact point of the RexExtElectrode object. Sets the class attribute "mapping", which is a shape (n_contact, n_segs) ndarray, such that the extracellular potential at the contacts phi = np.dot(mapping, I_mem) where I_mem is a shape (n_segs, n_tsteps) ndarray with transmembrane currents for each time step of the simulation.

Parameters

cell [obj] *LFPy.Cell* or *LFPy.TemplateCell* instance.

Returns

mapping [ndarray] The attribute RecExtElectrode.mapping is returned (optional)

set_cell(*cell*)
Set the supplied cell object as attribute "cell" of the RecExtElectrode object

Parameters

cell [obj] *LFPy.Cell* or *LFPy.TemplateCell* instance.

Returns

None

3.8.8 class Network

class LFPy.**Network**(*dt=0.1*, *tstart=0.0*, *tstop=1000.0*, *v_init=- 65.0*, *celsius=6.3*, *OUTPUT-PATH='example_parallel_network'*, *verbose=False*)
Bases: object

connect(*pre, post, connectivity, syntype=<MagicMock name='mock.ExpSyn' id='140583600958160'>, 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=<built-in method normal of numpy.random.mtrand.RandomState object>, delayargs={'loc': 2, 'scale': 0.2}, mindelay=0.3, multapsefun=<built-in method normal of numpy.random.mtrand.RandomState object>, multapseargs={'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 numpy.random distribution

delayargs [dict] parameters passed to delayfun

mindelay [float,] minimum delay in multiples of dt

multapsefun [function or None] function reference, e.g., numpy.random.normal 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_positions.h5" as dataset "<pre>:<post>" using a structured ndarray with dtype [('gid', 'i8'), ('x', float), ('y', float), ('z', float)] where gid is postsynaptic cell id, and x,y,z the corresponding midpoint coordinates of the target compartment.

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

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*, *seed=None*)

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

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

Parameters

electrode:

Either an LFPy.RecExtElectrode object or a list of such. If supplied, LFPs will be calculated at every time step and accessible as electrode.LFP. If a list of objects is given, accessible as electrode[0].LFP etc.

rec_imem: 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: record segment membrane voltages (mV)

rec_ipas: record passive segment membrane currents (nA)

rec_icap: record capacitive segment membrane currents (nA)

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

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

rec_istim: record currents of StimIntraElectrode (nA)

rec_current_dipole_moment [bool] If True, compute and record current-dipole moment from transmembrane currents as in Linden et al. (2010) J Comput Neurosci, DOI: 10.1007/s10827-010-0245-4. Will set the *LFPy.Cell* attribute *current_dipole_moment* as n_timesteps x 3 *ndarray* where the last dimension contains the x,y,z components of the dipole moment.

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

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

variable_dt: boolean, using variable timestep in NEURON

atol: absolute tolerance used with NEURON variable timestep

to_memory: only valid with electrode, store lfp in -> electrode.LFP

to_file: only valid with electrode, save LFPs in hdf5 file format

file_name [str] If to_file is True, file which extracellular potentials will be written to. The file format is HDF5, default is "OUTPUT.h5", put in folder Network.OUTPUTPATH

dotprodcoeffs [list of N x Nseg ndarray. These arrays will at] every timestep be multiplied by the membrane currents. Presumably useful for memory efficient csd or lfp calcs

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

Returns

SPIKES [dict] the first returned argument is a dictionary with keys 'gids' and 'times'. Each item is a nested list of len(Npop) times N_X where N_X is the corresponding population size. Each entry is a np.ndarray containing the spike times of each cell in the nested list in item 'gids'

OUTPUT [list of ndarray] if parameters electrode is not None and/or dotprodcoeffs is not None, contains the [electrode.LFP, ..., (dotprodcoeffs[0] dot I)(t), . . .] The first output is a structured array, so OUTPUT[0]['imem'] corresponds to the total LFP and the other the per-population contributions.

P [ndarray] if rec_current_dipole_moment==True, contains the x,y,z-components of current-dipole moment from transmembrane currents summed up over all populations

3.8.9 class NetworkPopulation

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

Bases: object

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*.

3.8.10 class InfiniteVolumeConductor

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

Bases: object

Main class for computing extracellular potentials with current dipole approximation in an infinite 3D volume conductor model that assumes homogeneous, isotropic, linear (frequency independent) conductivity

Parameters

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

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:

```
>>> import LFPy
>>> import numpy as np
>>> inf_model = LFPy.InfiniteVolumeConductor(sigma=0.3)
>>> p = np.array([[10., 10., 10.]])
>>> r = np.array([[1000., 0., 5000.]])
>>> phi_p = inf_model.get_dipole_potential(p, r)
```

get_dipole_potential(*p*, *r*)
Return electric potential from current dipole with current dipole approximation

p [ndarray, dtype=float] Shape (n_timesteps, 3) 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 contaning 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) FourSphereVolumeConductor.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

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 [Cell object from LFPy]

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
>>> 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 = LFPy.InfiniteVolumeConductor(sigma)
>>> phi = MD_INF.get_multi_dipole_potential(cell, electrode_locs,
>>>                                         timepoints = timepoints)
```

3.8.11 class OneSphereVolumeConductor

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

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, which is mathematically equivalent to a current source in conductive media, as published by Deng (2008), Journal of Electrostatics 66:549-560

Parameters

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)

Examples

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

```
>>> # import modules
>>> from LFPy 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(r, R=10000.,
>>>                                   sigma_i=0.3, sigma_o=0.03)
>>> Phi = sphere.calc_potential(rs=8000, I=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_mapping(*cell*, *n_max=1000*)
Compute linear mapping between transmembrane currents of LFPy.Cell like object instantiation and extracellular potential in and outside of sphere. Cell position must be set in space, using the method `Cell.set_pos(**kwargs)`.

Parameters

cell [LFPy.Cell like instance] Instantiation of class LFPy.Cell, TemplateCell or NetworkCell.

n_max [int] Number of elements in polynomial expansion to sum over (see Deng 2008).

Returns

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

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

Examples

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

```
>>> # import modules
>>> import LFPy
>>> 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', '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 = LFPy.OneSphereVolumeConductor(r=r, R=10000.,
>>>                                        sigma_i=0.3, sigma_o=0.03)
>>> mapping = sphere.calc_mapping(cell, n_max=1000)
>>> # pick out some time index for the potential and compute potential
>>> ind = cell.tvec==2.
>>> Phi = np.dot(mapping, 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(zip(x, z))
>>> polycol = PolyCollection(zips,
>>>                          edgecolors='none',
```

(continues on next page)

(continued from previous page)

```
>>>                             facecolors='gray')
>>> vrange = 1E-3 # limits for color contour plot
>>> im=ax.contour(X, Z, Phi,
>>>               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()
```

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

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

Parameters

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

I [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 singular values)

n_max [int] Number of elements in polynomial expansion to sum over (see Deng 2008).

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).

3.8.12 class FourSphereVolumeConductor

class LFPy.**FourSphereVolumeConductor**(*radii*, *sigmas*, *r_electrodes*, *iter_factor=2.0202020202020204e-08*)

Bases: object

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).

Parameters

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.

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]`.

Examples

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

```
>>> import LFPy
>>> import numpy as np
>>> radii = [79000., 80000., 85000., 90000.]
>>> sigmas = [0.3, 1.5, 0.015, 0.3]
>>> r = np.array([[0., 0., 90000.], [0., 85000., 0.]])
>>> rz = np.array([0., 0., 78000.])
>>> sphere_model = LFPy.FourSphereVolumeConductor(radii, sigmas, r)
>>> # current dipole moment
>>> p = np.array([[10., 10., 10.]]*10) # 10 timesteps
>>> # compute potential
>>> potential = sphere_model.calc_potential(p, rz)
```

calc_phi (*p_tan*)
Return azimuthal angle between x-axis and contact point locations(s)

Parameters

p_tan [ndarray, dtype=float] Shape (n_timesteps, 3) array containing tangential component of current dipole moment in units of (nA*µm)

Returns

phi [ndarray, dtype=float] Shape (n_contacts, n_timesteps) array containing azimuthal angle in units of (radians) between x-axis vector(s) and projection of contact point location vector(s) rxyz into xy-plane. z-axis is defined in the direction of rzloc. y-axis is defined in the direction of p_tan (orthogonal to rzloc). x-axis is defined as cross product between p_tan and rzloc (x).

calc_potential (*p*, *rz*)
Return electric potential from current dipole moment p.

Parameters

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

rz [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.r in units of (mV) for all timesteps of current dipole moment p.

calc_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 LFPy
>>> 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)
>>> radii = [200., 300., 400., 500.]
>>> sigmas = [0.3, 1.5, 0.015, 0.3]
>>> electrode_locs = np.array([[50., -50., 250.]])
>>> timepoints = np.array([0,100])
>>> MD_4s = LFPy.FourSphereVolumeConductor(radii,
>>>                                        sigmas,
>>>                                        electrode_locs)
>>> phi = MD_4s.calc_potential_from_multi_dipoles(cell,
>>>                                               timepoints)
```

calc_theta()

Return polar angle(s) between rzloc and contact point location(s)

Returns

theta [ndarray, dtype=float] Shape (n_contacts,) array containing polar angle in units of (radians) between z-axis and n_contacts contact point location vector(s) in FourSphereVolumeConductor.rxyz z-axis is defined in the direction of rzloc and the radial dipole.

3.8.13 class MEG

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

Bases: object

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 (Nunez and Srinivasan, Oxford University Press, 2006):

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

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

Examples

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

```
>>> import LFPy, os, numpy as np, matplotlib.pyplot as plt
>>> 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_current_dipole_moment=True)
```

Compute the dipole location as an average of segment locations weighted by membrane area:

```
>>> dipole_location = (cell.area * np.c_[cell.xmid, cell.ymid, cell.zmid].T /␣
↪cell.area.sum()).sum(axis=1)
```

Instantiate the LFPy.MEG object, compute and plot the magnetic signal in a sensor location:

```
>>> sensor_locations = np.array([[1E4, 0, 0]])
>>> meg = LFPy.MEG(sensor_locations)
>>> H = meg.calculate_H(cell.current_dipole_moment, dipole_location)
>>> 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])
```

calculate_H(*current_dipole_moment*, *dipole_location*)

Compute magnetic field H from single current-dipole moment localized somewhere in space

Parameters

current_dipole_moment [ndarray, dtype=float] shape (n_timesteps x 3) 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 n_timesteps x 3) array with x,y,z-components of the magnetic field **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: Blagoev et al. (2007) Modelling the magnetic signature of neuronal tissue. NeuroImage 37 (2007) 137–148 DOI: 10.1016/j.neuroimage.2007.04.033

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 n_timesteps x 3) array with x,y,z-components of the magnetic field **H** in units of (nA/µm)

Examples

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

```
>>> import LFPy, os, numpy as np, matplotlib.pyplot as plt
>>> 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 LFPy.MEG object, compute and plot the magnetic signal in a sensor location:

```
>>> sensor_locations = np.array([[1E4, 0, 0]])
>>> meg = LFPy.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])
```

3.8.14 submodule eegmegcalc

LFPy.**get_current_dipole_moment**(*dist*, *current*)

Return current dipole moment vector P and P_tot of cell.

Parameters

current [ndarray, dtype=float] Either an array containing all transmembrane currents from all compartments of the cell, or an array of all axial currents between compartments in cell in units of nA

dist [ndarray, dtype=float] When input current is an array of axial currents, dist is the length of each axial current. When current is an array of transmembrane currents, dist is the position vector of each compartment middle. Unit is (µm).

Returns

P [ndarray, dtype=float] Array containing the current dipole moment for all timesteps in the x-, y- and z-direction in units of (nA*µm)

P_tot [ndarray, dtype=float] Array containing the magnitude of the current dipole moment vector for all timesteps in units of (nA*µm)

Examples

Get current dipole moment vector and scalar moment from axial currents computed from membrane potentials:

```
>>> 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)
>>> d_list, i_axial = cell.get_axial_currents()
>>> P_ax, P_ax_tot = LFPy.get_current_dipole_moment(d_list, i_axial)
```

Get current dipole moment vector and scalar moment from transmembrane currents using the extracellular mechanism in NEURON:

```
>>> import LFPy
>>> import numpy as np
>>> cell = LFPy.Cell('PATH/TO/MORPHOLOGY', extracellular=True)
>>> 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=False, rec_imem=True)
>>> P_imem, P_imem_tot = LFPy.get_current_dipole_moment(np.c_[cell.xmid,
>>>                                                           cell.ymid,
>>>                                                           cell.zmid],
>>>                                                     cell.imem)
```

3.8.15 submodule `lfpcalc`

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.lfpcalc.`**`calc_lfp_linesource`**(*cell*, *x*, *y*, *z*, *sigma*, *r_limit*)

Calculate electric field potential using the line-source method, all compartments treated as line sources, including soma.

Parameters

cell: obj LFPy.Cell or LFPy.TemplateCell like instance

x [float] extracellular position, x-axis

y [float] extracellular position, y-axis

z [float] extracellular position, z-axis

sigma [float] extracellular conductivity

r_limit [np.ndarray] minimum distance to source current for each compartment

LFPy.lfpcalc.**calc_lfp_linesource_anisotropic**(*cell*, *x*, *y*, *z*, *sigma*, *r_limit*)

Calculate electric field potential using the line-source method, all compartments treated as line sources, even soma.

Parameters

cell: obj LFPy.Cell or LFPy.TemplateCell instance

x [float] extracellular position, x-axis

y [float] extracellular position, y-axis

z [float] extracellular position, z-axis

sigma [array] extracellular conductivity [sigma_x, sigma_y, sigma_z]

r_limit [np.ndarray] minimum distance to source current for each compartment

LFPy.lfpcalc.**calc_lfp_linesource_moi**(*cell*, *x*, *y*, *z*, *sigma_T*, *sigma_S*, *sigma_G*, *steps*, *h*, *r_limit*, ***kwargs*)

Calculate extracellular potentials using the line-source equation on all compartments for in vitro Microelectrode Array (MEA) slices

Parameters

cell: obj LFPy.Cell or LFPy.TemplateCell like instance

x [float] extracellular position, x-axis

y [float] extracellular position, y-axis

z [float] extracellular position, z-axis

sigma_T [float] extracellular conductivity in tissue slice

sigma_G [float] Conductivity of MEA glass electrode plane. Should normally be zero for MEA set up, and for this method, only zero valued sigma_G is supported.

sigma_S [float] Conductivity of saline bath that tissue slice is immersed in

steps [int] Number of steps to average over the in technically infinite sum

h [float] Slice thickness in um.

r_limit [np.ndarray] minimum distance to source current for each compartment

LFPy.lfpcalc.**calc_lfp_pointsource**(*cell*, *x*, *y*, *z*, *sigma*, *r_limit*)

Calculate extracellular potentials using the point-source equation on all compartments

Parameters

cell: obj LFPy.Cell or LFPy.TemplateCell like instance

x [float] extracellular position, x-axis

y [float] extracellular position, y-axis

z [float] extracellular position, z-axis

sigma [float] extracellular conductivity

r_limit [np.ndarray] minimum distance to source current for each compartment

LFPy.lfpcalc.**calc_lfp_pointsource_anisotropic**(*cell*, *x*, *y*, *z*, *sigma*, *r_limit*)
Calculate extracellular potentials using the anisotropic point-source equation on all compartments

Parameters

cell: obj LFPy.Cell or LFPy.TemplateCell instance

x [float] extracellular position, x-axis

y [float] extracellular position, y-axis

z [float] extracellular position, z-axis

sigma [array] extracellular conductivity in [x,y,z]-direction

r_limit [np.ndarray] minimum distance to source current for each compartment

LFPy.lfpcalc.**calc_lfp_pointsource_moi**(*cell*, *x*, *y*, *z*, *sigma_T*, *sigma_S*, *sigma_G*, *steps*, *h*, *r_limit*, ***kwargs*)
Calculate extracellular potentials using the point-source equation on all compartments for in vitro Microelectrode Array (MEA) slices

Parameters

cell: obj LFPy.Cell or LFPy.TemplateCell like instance

x [float] extracellular position, x-axis

y [float] extracellular position, y-axis

z [float] extracellular position, z-axis

sigma_T [float] extracellular conductivity in tissue slice

sigma_G [float] Conductivity of MEA glass electrode plane. Should normally be zero for MEA set up.

sigma_S [float] Conductivity of saline bath that tissue slice is immersed in

steps [int] Number of steps to average over the in technically infinite sum

h [float] Slice thickness in um.

r_limit [np.ndarray] minimum distance to source current for each compartment

LFPy.lfpcalc.**calc_lfp_soma_as_point**(*cell*, *x*, *y*, *z*, *sigma*, *r_limit*)
Calculate electric field potential using the line-source method, soma is treated as point/sphere source

Parameters

cell: obj *LFPy.Cell* or *LFPy.TemplateCell* like instance

x [float] extracellular position, x-axis

y [float] extracellular position, y-axis

z [float] extracellular position, z-axis

sigma [float] extracellular conductivity in S/m

r_limit [np.ndarray] minimum distance to source current for each compartment.

LFPy.lfpcalc.**calc_lfp_soma_as_point_anisotropic**(*cell*, *x*, *y*, *z*, *sigma*, *r_limit*)
Calculate electric field potential, soma is treated as point source, all compartments except soma are treated as line sources.

Parameters

cell: obj LFPy.Cell or LFPy.TemplateCell instance

x [float] extracellular position, x-axis

y [float] extracellular position, y-axis

z [float] extracellular position, z-axis

sigma [array] extracellular conductivity [sigma_x, sigma_y, sigma_z]

r_limit [np.ndarray] minimum distance to source current for each compartment

LFPy.lfpcalc.**calc_lfp_soma_as_point_moi**(*cell*, *x*, *y*, *z*, *sigma_T*, *sigma_S*, *sigma_G*, *steps*, *h*, *r_limit*, ***kwargs*)
Calculate extracellular potentials for in vitro Microelectrode Array (MEA) slices, where soma (compartment zero) is treated as a point source, and all other compartments as line sources.

Parameters

cell: obj LFPy.Cell or LFPy.TemplateCell like instance

x [float] extracellular position, x-axis

y [float] extracellular position, y-axis

z [float] extracellular position, z-axis

sigma_T [float] extracellular conductivity in tissue slice

sigma_G [float] Conductivity of MEA glass electrode plane. Should normally be zero for MEA set up, and for this method, only zero valued sigma_G is supported.

sigma_S [float] Conductivity of saline bath that tissue slice is immersed in

steps [int] Number of steps to average over the in technically infinite sum

h [float] Slice thickness in um.

r_limit [np.ndarray] minimum distance to source current for each compartment

LFPy.lfpcalc.**return_dist_from_segments**(*xstart*, *ystart*, *zstart*, *xend*, *yend*, *zend*, *p*)
Returns distance and closest point on line segments from point p

3.8.16 submodule tools

LFPy.tools.**load**(*filename*)
Generic loading of cPickled objects from 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)
```

3.8.17 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_* *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`, but this 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

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

L

INDEX

I

L

M

N

O

P

R

S

T

U