

---

# hybridLFPy Documentation

*Release 0.2*

**Espen Hagen**

**Nov 19, 2021**



## CONTENTS:

<b>1</b>	<b>hybridLFPy</b>	<b>1</b>
1.1	Project Status . . . . .	1
1.2	Development . . . . .	1
1.3	Citation . . . . .	1
1.4	License . . . . .	2
1.5	Warranty . . . . .	2
1.6	Installation . . . . .	2
1.6.1	examples folder . . . . .	2
1.6.2	docs folder . . . . .	3
1.6.3	Dockerfile . . . . .	3
1.7	Online documentation . . . . .	3
1.8	Notes on performance . . . . .	3
<b>2</b>	<b>Module hybridLFPy</b>	<b>5</b>
2.1	<b>hybridLFPy</b> . . . . .	<b>5</b>
2.1.1	How to use the documentation . . . . .	5
2.1.2	Available classes . . . . .	5
2.1.3	Available utilities . . . . .	6
<b>3</b>	<b>class CachedNetwork</b>	<b>7</b>
<b>4</b>	<b>class CachedNoiseNetwork</b>	<b>11</b>
<b>5</b>	<b>class CachedFixedSpikesNetwork</b>	<b>13</b>
<b>6</b>	<b>class PopulationSuper</b>	<b>15</b>
<b>7</b>	<b>class Population</b>	<b>19</b>
<b>8</b>	<b>class PostProcess</b>	<b>23</b>
<b>9</b>	<b>class GDF</b>	<b>25</b>
<b>10</b>	<b>submodule helpers</b>	<b>29</b>
<b>11</b>	<b>submodulue test</b>	<b>39</b>
<b>12</b>	<b>Indices and tables</b>	<b>41</b>
	<b>Python Module Index</b>	<b>43</b>
	<b>Index</b>	<b>45</b>



## HYBRIDLFPY

Python module implementing a hybrid scheme for predictions of extracellular potentials (local field potentials, LFPs) of spiking neuron network simulations.

### 1.1 Project Status

### 1.2 Development

The module hybridLFPy was mainly developed in the Computational Neuroscience Group (<http://compneuro.umb.no>), Department of Mathematical Sciences and Technology (<http://www.nmbu.no/imt>), at the Norwegian University of Life Sciences (<http://www.nmbu.no>), Aas, Norway, in collaboration with Institute of Neuroscience and Medicine (INM-6) and Institute for Advanced Simulation (IAS-6), Juelich Research Centre and JARA, Juelich, Germany (<http://www.fz-juelich.de/inm/inm-6/EN/>).

### 1.3 Citation

Should you find hybridLFPy useful for your research, please cite the following paper:

Espen Hagen, David Dahmen, Maria L. Stavrinou, Henrik Lindén, Tom Tetzlaff, Sacha J. van Albada, Sonja Grün, Markus Diesmann, Gaute T. Einevoll;  
Hybrid Scheme for Modeling Local Field Potentials from Point-Neuron Networks,  
Cerebral Cortex, Volume 26, Issue 12, 1 December 2016, Pages 4461-4496,  
<https://doi.org/10.1093/cercor/bhw237>

Bibtex source:

```
@article{doi:10.1093/cercor/bhw237,  
author = {Hagen, Espen and Dahmen, David and Stavrinou, Maria L. and Lindén,  
Henrik and Tetzlaff, Tom and van Albada, Sacha J. and Grün, Sonja and
```

(continues on next page)

(continued from previous page)

```
Diesmann, Markus and Einevoll, Gaute T.},
title = {Hybrid Scheme for Modeling Local Field Potentials from
Point-Neuron Networks},
journal = {Cerebral Cortex},
volume = {26},
number = {12},
pages = {4461-4496},
year = {2016},
doi = {10.1093/cercor/bhw237},
URL = { + http://dx.doi.org/10.1093/cercor/bhw237},
eprint = {oup/backfile/content_public/journal/cercor/26/12/10.1093_cercor_bhw237/2/
↪bhw237.pdf}
}
```

## 1.4 License

This software is released under the General Public License (see the [LICENSE](#) file).

## 1.5 Warranty

This software comes without any form of warranty.

## 1.6 Installation

First download all the hybridLFPy source files using git (<http://git-scm.com>). Open a terminal window and type:

```
cd $HOME/where/to/put/hybridLFPy
git clone https://github.com/INM-6/hybridLFPy.git
```

To use hybridLFPy from any working folder without copying files, run:

```
(sudo) pip install -e . (--user)
```

Installing it is also possible, but not recommended as things might change with future pulls from the repository:

```
(sudo) pip install . (--user)
```

### 1.6.1 examples folder

Some example script(s) on how to use this module

## 1.6.2 docs folder

Source files for autogenerated documentation using Sphinx (<https://www.sphinx-doc.org>).

To compile documentation source files in this directory using sphinx, use:

```
sphinx-build -b html docs documentation
```

## 1.6.3 Dockerfile

The provided Dockerfile provides a Docker container recipe for x86\_64 hosts with all dependencies required to run simulation files provided in examples. To build and run the container locally, get Docker from <https://www.docker.com> and issue the following (replace <image-name> with a name of your choosing):

```
docker build -t <image-name> -f Dockerfile
docker run -it -p 5000:5000 <image-name>:latest
```

The --mount option can be used to mount a folder on the host to a target folder as:

```
docker run --mount type=bind,source="$(pwd)",target=/opt/hybridLFPy \
-it -p 5000:5000 <image-name>
```

Then, code examples may be run as:

```
cd /opt/hybridLFPy/examples
nrnivmodl # compile local .mod (NMODL) files
mpirun --allow-run-as-root python3 example_brunel.py
```

## 1.7 Online documentation

The sphinx-generated html documentation can be accessed at <https://hybridLFPy.readthedocs.io>

## 1.8 Notes on performance

The present version of hybridLFPy may facilitate on a trivial parallelism as the contribution of each single-cell LFP can be computed independently. However, this does not imply that the present implementation code is highly optimized for speed. In particular, initializing the multicompartment neuron populations do not as much benefit from increasing the MPI pool size, as exemplified by a benchmark based on the Brunel-network example scaled up to 50,000 neurons and with simplified neuron morphologies.

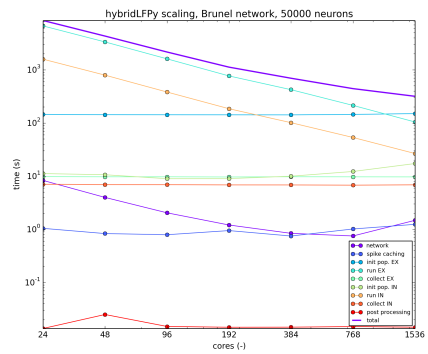


Fig. 1: Scaling example with hybridLFPy based on a Brunel-like network with 50,000 neurons, running on the JU-RECA cluster at the Juelich Supercomputing Centre (JSC), Juelich Research Centre, Germany.



## MODULE HYBRIDLFPY

### 2.1 hybridLFPy

Provides methods for estimating extracellular potentials of simplified spiking neuron network models.

#### 2.1.1 How to use the documentation

Documentation is available in two forms:

1. Docstrings provided with the code, e.g., as `hybridLFPy?` within IPython
2. Autogenerated sphinx-built output, compiled issuing:

```
sphinx-build -b html docs documentation
```

in the root folder of the package sources

#### 2.1.2 Available classes

**CachedNetwork** Offline interface between network spike events and used by class `Population`

**CachedNoiseNetwork** Creation of Poisson spiketrains of a putative network model, interfaces class `Population`

**CachedFixedSpikesNetwork** Creation of identical spiketrains per population of a putative network model, interface to class `Population`

**GDF** Class using sqlite to efficiently store and enquire large amounts of spike output data, used by `Cached*Network`

**PopulationSuper** Parent class setting up a base population of multicompartment neurons

**Population** Daughter of `PopulationSuper`, using `CachedNetwork` spike events as synapse activation times with layer and cell type connection specificity

**PostProcess** Methods for processing output of multiple instances of class `Population`

### 2.1.3 Available utilities

**helpers** Various methods used throughout simulations

**setup\_file\_dest** Setup destination folders of simulation output files

**test** Run a series of unit tests

## CLASS CACHEDNETWORK

```
class hybridLFPy.CachedNetwork(simtime=1000.0, dt=0.1, spike_output_path='spike_output_path',  
                                label='spikes', ext='gdf', GIDs={'EX': [1, 400], 'IN': [401, 100]}, X=['EX',  
                                'IN'], autocollect=True, skiprows=0, cmap='Dark2')
```

Bases: object

Offline processing and storing of network spike events, used by other class objects in the package hybridLFPy.

### Parameters

**simtime** [float] Simulation duration.

**dt** [float,] Simulation timestep size.

**spike\_output\_path** [str] Path to gdf/dat-files with spikes.

**label** [str] Prefix of spiking gdf/dat-files.

**ext** [str] File extension of gdf/dat-files.

**GIDs** [dict] dictionary keys are population names and value a list of length 2 with first GID in population and population size

**X** [list] names of each network population

**autocollect** [bool] If True, class init will process gdf/dat files.

**cmap** [str] Name of colormap, must be in *dir(plt.cm)*.

### Returns

*hybridLFPy.cachednetworks.CachedNetwork* object

See also:

[\*CachedFixedSpikesNetwork\*](#), [\*CachedNoiseNetwork\*](#)

### collect\_gdf()

Collect the gdf-files from network sim in folder *spike\_output\_path* into sqlite database, using the GDF-class.

### Parameters

None

### Returns

None

### get\_xy(xlim, fraction=1.0)

Get pairs of node units and spike trains on specific time interval.

### Parameters

**xlim** [list of floats] Spike time interval, e.g., [0., 1000.].

**fraction** [float in [0, 1.]] If less than one, sample a fraction of nodes in random order.

### Returns

**x** [dict] In *x* key-value entries are population name and neuron spike times.

**y** [dict] Where in *y* key-value entries are population name and neuron gid number.

**plot\_f\_rate**(*ax, X, i, xlim, x, y, binsize=1, yscale='linear', plottype='fill\_between', show\_label=False, rasterized=False*)

Plot network firing rate plot in subplot object.

### Parameters

**ax** [*matplotlib.axes.AxesSubplot* object.]

**X** [str] Population name.

**i** [int] Population index in class attribute *X*.

**xlim** [list of floats] Spike time interval, e.g., [0., 1000.].

**x** [dict] Key-value entries are population name and neuron spike times.

**y** [dict] Key-value entries are population name and neuron gid number.

**yscale** ['str'] Linear, log, or symlog y-axes in rate plot.

**plottype** [str] plot type string in [*'fill\_between', 'bar'*]

**show\_label** [bool] whether or not to show labels

### Returns

**None**

**plot\_raster**(*ax, xlim, x, y, pop\_names=False, markersize=20.0, alpha=1.0, legend=True, marker='o', rasterized=True*)

Plot network raster plot in subplot object.

### Parameters

**ax** [*matplotlib.axes.AxesSubplot* object] plot axes

**xlim** [list] List of floats. Spike time interval, e.g., [0., 1000.].

**x** [dict] Key-value entries are population name and neuron spike times.

**y** [dict] Key-value entries are population name and neuron gid number.

**pop\_names: bool** If True, show population names on yaxis instead of gid number.

**markersize** [float] raster plot marker size

**alpha** [float in [0, 1]] transparency of marker

**legend** [bool] Switch on axes legends.

**marker** [str] marker symbol for matplotlib.pyplot.plot

**rasterized** [bool] if True, the scatter plot will be treated as a bitmap embedded in pdf file output

### Returns

**None**

**raster\_plots**(*xlim*=[0, 1000], *markersize*=1, *alpha*=1.0, *marker*='o')

Pretty plot of the spiking output of each population as raster and rate.

**Parameters**

**xlim** [list] List of floats. Spike time interval, e.g., [0., 1000.].

**markersize** [float] marker size for plot, see *matplotlib.pyplot.plot*

**alpha** [float] transparency for markers, see *matplotlib.pyplot.plot*

**marker** [A valid marker style]

**Returns**

**fig** [*matplotlib.figure.Figure* object]



## CLASS CACHEDNOISENETWORK

**class** `hybridLFPy.CachedNoiseNetwork`(*frate*={'EX': 5.0, 'IN': 10.0}, *autocollect*=False, *\*\*kwargs*)

Bases: [\*hybridLFPy.cachednetworks.CachedNetwork\*](#)

Subclass of `CachedNetwork`.

Use Nest to generate N\_X poisson-generators each with rate *frate*, and record every vector, and create database with spikes.

### Parameters

**frate** [list] Rate of each layer, may be tuple (onset, rate, offset)

**autocollect** [bool] whether or not to automatically gather gdf file output

**\*\*kwargs** [see parent class [\*hybridLFPy.cachednetworks.CachedNetwork\*](#)]

### Returns

[\*hybridLFPy.cachednetworks.CachedNoiseNetwork\*](#) object

See also:

[\*CachedNetwork\*](#), [\*CachedFixedSpikesNetwork\*](#)





## CLASS CACHEDFIXEDSPIKESNETWORK

**class** `hybridLFPy.CachedFixedSpikesNetwork`(*activationtimes*=[200, 300, 400, 500, 600, 700, 800, 900, 1000], *autocollect*=False, **\*\*kwargs**)

Bases: `hybridLFPy.cachednetworks.CachedNetwork`

Subclass of `CachedNetwork`.

Fake nest output, where each cell in a subpopulation spike simultaneously, and each subpopulation is activated at times given in kwarg *activationtimes*.

### Parameters

**activationtimes** [list of floats] Each entry set spike times of all cells in each population

**autocollect** [bool] whether or not to automatically gather gdf file output

**\*\*kwargs** [see parent class `hybridLFPy.cachednetworks.CachedNetwork`]

### Returns

*hybridLFPy.cachednetworks.CachedFixedSpikesNetwork* object

See also:

`CachedNetwork`, `CachedNoiseNetwork`



## CLASS POPULATIONSUPER

```
class hybridLFPy.PopulationSuper(cellParams={'Ra': 150, 'cm': 1.0, 'dt': 0.1, 'e_pas': 0.0, 'lambda_f': 100,
                                             'morphology': 'morphologies/ex.hoc', 'nsegs_method': 'lambda_f', 'rm':
                                             20000.0, 'tstart': 0, 'tstop': 1000.0, 'v_init': 0.0, 'verbose': False},
                                rand_rot_axis=[], simulationParams={},
                                populationParams={'min_cell_interdist': 1.0, 'number': 400, 'r_z': [[-
1e+199, 1e+99], [10, 10]], 'radius': 100, 'z_max': -350, 'z_min': -450},
                                y='EX', layerBoundaries=[[0.0, -300], [-300, -500]], probes=[],
                                savelist=['somapos'], savefolder='simulation_output_example_brunel',
                                dt_output=1.0, recordSingleContribFrac=0,
                                POPULATIONSEED=123456, verbose=False,
                                output_file='{ }_population_{ }')
```

Bases: object

Main population class object, let one set up simulations, execute, and compile the results. This class is suitable for subclassing for custom cell simulation procedures, inherit things like gathering of data written to disk.

Note that *PopulationSuper.cellsim* do not have any stimuli, its main purpose is to gather common methods for inherited Population objects.

## Parameters

**cellParams** [dict] Parameters for class *LFPy.Cell*

**rand\_rot\_axis** [list] Axis of which to randomly rotate morphs.

**simulationParams** [dict] Additional args for *LFPy.Cell.simulate()*.

**populationParams** [dict] Constraints for population and cell number.

**y** [str] Population identifier string.

**layerBoundaries** [list of lists] Each element is a list setting upper and lower layer boundary (floats)

**probes: list** list of *LFPykit.models.\** like instances for misc. forward-model predictions

**savelist** [list] *LFPy.Cell* arguments to save for each single-cell simulation.

**savefolder** [str] path to where simulation results are stored.

**dt\_output** [float] Time resolution of output, e.g., LFP, CSD etc.

**recordSingleContribFrac** [float] fraction in [0, 1] of individual neurons in population which output will be stored

**POPULATIONSEED** [int/float] Random seed for population, for positions etc.

**verbose** [bool] Verbosity flag.

**output\_file** [str] formattable string for population output, e.g., '{ }\_population\_{ }'

**Returns**

**hybridLFPy.population.PopulationSuper** object

See also:

[\*Population\*](#), `LFPy.Cell`, `LFPy.RecExtElectrode`

**calc\_min\_cell\_interdist**(*x*, *y*, *z*)

Calculate cell interdistance from input coordinates.

**Parameters**

**x, y, z** [numpy.ndarray] xyz-coordinates of each cell-body.

**Returns**

**min\_cell\_interdist** [np.ndarray] For each cell-body center, the distance to nearest neighboring cell

**calc\_signal\_sum**(*measure*='LFP')

Superimpose each cell's contribution to the compound population signal, i.e., the population CSD or LFP or some other lfpkit.<instance>

**Parameters**

**measure** [str]

**Returns**

**numpy.ndarray** The populations-specific compound signal.

**cellsim**(*cellindex*, *return\_just\_cell*=False)

Single-cell *LFPy.Cell* simulation without any stimulus, mostly for reference, as no stimulus is added

**Parameters**

**cellindex** [int] cell index between 0 and POPULATION\_SIZE-1.

**return\_just\_cell** [bool] If True, return only the *LFPy.Cell* object if False, run full simulation, return None.

**Returns**

**None** if *return\_just\_cell* is False

**cell** [*LFPy.Cell* instance] if *return\_just\_cell* is True

See also:

`LFPy.Cell`, `LFPy.Synapse`, `LFPy.RecExtElectrode`

**collectSingleContribs**(*measure*='LFP')

Collect single cell data and save them to HDF5 file. The function will also return signals generated by all cells

**Parameters**

**measure** [str] Either 'LFP', 'CSD' or 'current\_dipole\_moment'

**Returns**

**numpy.ndarray** output of all neurons in population, axis 0 correspond to neuron index

**collect\_data()**

Collect LFPs, CSDs and soma traces from each simulated population, and save to file.

**Parameters**

None

**Returns**

None

**collect\_savelist()**

collect cell attribute data to RANK 0 before dumping data to file

**draw\_rand\_pos(radius, z\_min, z\_max, min\_r=array([0]), min\_cell\_interdist=10.0, \*\*args)**

Draw some random location within radius, z\_min, z\_max, and constrained by min\_r and the minimum cell interdistance. Returned argument is a list of dicts with keys ['x', 'y', 'z'].

**Parameters**

**radius** [float] Radius of population.

**z\_min** [float] Lower z-boundary of population.

**z\_max** [float] Upper z-boundary of population.

**min\_r** [numpy.ndarray] Minimum distance to center axis as function of z.

**min\_cell\_interdist** [float] Minimum cell to cell interdistance.

**\*\*args** [keyword arguments] Additional inputs that is being ignored.

**Returns**

**soma\_pos** [list] List of dicts of len population size where dict have keys x, y, z specifying xyz-coordinates of cell at list entry *i*.

**See also:**

*[PopulationSuper.calc\\_min\\_cell\\_interdist](#)*

**run()**

Distribute individual cell simulations across ranks.

This method takes no keyword arguments.

**Parameters**

None

**Returns**

None

**set\_pop\_soma\_pos()**

Set *pop\_soma\_pos* using *draw\_rand\_pos()*.

This method takes no keyword arguments.

**Parameters**

None

**Returns**

**numpy.ndarray** (x,y,z) coordinates of each neuron in the population

**See also:**

*PopulationSuper.draw\_rand\_pos*

**set\_rotations()**

Append random z-axis rotations for each cell in population.

This method takes no keyword arguments

**Parameters**

**None**

**Returns**

**numpy.ndarray** Rotation angle around axis *Population.rand\_rot\_axis* of each neuron in the population

## CLASS POPULATION

```
class hybridLFPy.Population(X=['EX', 'IN'], networkSim='hybridLFPy.cachednetworks.CachedNetwork',
                             k_yXL=[[20, 0], [20, 10]], synParams={'EX': {'section': ['apic', 'dend'],
                             'syntype': 'AlphaISyn'}, 'IN': {'section': ['dend'], 'syntype': 'AlphaISyn'}},
                             synDelayLoc=[1.5, 1.5], synDelayScale=[None, None],
                             J_yX=[0.20680155243678455, - 1.2408093146207075], tau_yX=[0.5, 0.5],
                             **kwargs)
```

Bases: [hybridLFPy.population.PopulationSuper](#)

Class *hybridLFPy.Population*, inherited from class *PopulationSuper*.

This class rely on spiking times recorded in a network simulation, layer-resolved indegrees, synapse parameters, delay parameters, all per presynaptic population.

### Parameters

**X** [list of str] Each element denote name of presynaptic populations.

**networkSim** [*hybridLFPy.cachednetworks.CachedNetwork* object] Container of network spike events resolved per population

**k\_yXL** [numpy.ndarray] Num layers x num presynapse populations array specifying the number of incoming connections per layer and per population type.

**synParams** [dict of dicts] Synapse parameters (cf. *LFPy.Synapse* class). Each toplevel key denote each presynaptic population, bottom-level dicts are parameters passed to *LFPy.Synapse*.

**synDelayLoc** [list] Average synapse delay for each presynapse connection.

**synDelayScale** [list] Synapse delay std for each presynapse connection.

**J\_yX** [list of floats] Synapse weights for connections of each presynaptic population, see class *LFPy.Synapse*

### Returns

*hybridLFPy.population.Population* object

See also:

[PopulationSuper](#), [CachedNetwork](#), [CachedFixedSpikesNetwork](#)

[CachedNoiseNetwork](#), [LFPy.Cell](#), [LFPy.RecExtElectrode](#)

**cellsim**(*cellindex*, *return\_just\_cell=False*)

Do the actual simulations of LFP, using synaptic spike times from network simulation.

### Parameters

**cellindex** [int] cell index between 0 and population size-1.

**return\_just\_cell** [bool] If True, return only the *LFPy.Cell* object if False, run full simulation, return None.

#### Returns

None or *LFPy.Cell* object

See also:

**hybridLFPy.csd**, **LFPy.Cell**, **LFPy.Synapse**, **LFPy.RecExtElectrode**

#### **fetchSpCells**(*nodes*, *numSyn*)

For N (nodes count) nestSim-cells draw POPULATION\_SIZE x NTIMES random cell indexes in the population in nodes and broadcast these as *SpCell*.

The returned argument is a list with len = numSyn.size of np.arrays, assumes *numSyn* is a list

#### Parameters

**nodes** [numpy.ndarray, dtype=int] Node # of valid presynaptic neurons.

**numSyn** [numpy.ndarray, dtype=int] # of synapses per connection.

#### Returns

**SpCells** [list] presynaptic network-neuron indices

See also:

**Population.fetch\_all\_SpCells**

#### **fetchSynIdxCell**(*cell*, *nidx*, *synParams*)

Find possible synaptic placements for each cell As synapses are placed within layers with bounds determined by self.layerBoundaries, it will check this matrix accordingly, and use the probabilities from self.connProbLayer to distribute.

For each layer, the synapses are placed with probability normalized by membrane area of each compartment

#### Parameters

**cell** [*LFPy.Cell* instance]

**nidx** [numpy.ndarray] Numbers of synapses per presynaptic population X.

**synParams** [which *LFPy.Synapse* parameters to use.]

#### Returns

**syn\_idx** [list] List of arrays of synapse placements per connection.

See also:

**Population.get\_all\_synIdx**, **Population.get\_synIdx**, **LFPy.Synapse**

#### **get\_all\_SpCells**()

For each postsynaptic cell existing on this RANK, load or compute the presynaptic cell index for each synaptic connection

This function takes no kwargs.

#### Parameters

None

#### Returns



**SpCells** [dict] *output[cellindex][populationname][layerindex]*, np.array of presynaptic cell indices.

See also:

*Population.fetchSpCells*

#### **get\_all\_synDelays()**

Create and load arrays of connection delays per connection on this rank

Get random normally distributed synaptic delays, returns dict of nested list of same shape as SpCells.

Delays are rounded to dt.

This function takes no kwargs.

##### **Parameters**

**None**

##### **Returns**

**dict** *output[cellindex][populationname][layerindex]*, np.array of delays per connection.

See also:

`numpy.random.normal`

#### **get\_all\_synIdx()**

Auxilliary function to set up class attributes containing synapse locations given as LFPy.Cell compartment indices

This function takes no inputs.

##### **Parameters**

**None**

##### **Returns**

**synIdx** [dict] *output[cellindex][populationindex][layerindex]* numpy.ndarray of compartment indices.

See also:

*Population.get\_synidx, Population.fetchSynIdxCell*

#### **get\_synidx(cellindex)**

Local function, draw and return synapse locations corresponding to a single cell, using a random seed set as *POPULATIONSEED + cellindex*.

##### **Parameters**

**cellindex** [int] Index of cell object.

##### **Returns**

**synidx** [dict] *LFPy.Cell* compartment indices

See also:

*Population.get\_all\_synIdx, Population.fetchSynIdxCell*

**insert\_all\_synapses**(*cellindex*, *cell*)

Insert all synaptic events from all presynaptic layers on cell object with index *cellindex*.

**Parameters**

**cellindex** [int] cell index in the population.

**cell** [*LFPy.Cell* instance] Postsynaptic target cell.

**Returns**

None

See also:

**Population.insert\_synapse**

**insert\_synapses**(*cell*, *cellindex*, *synParams*, *idx=array([], dtype=float64)*, *X='EX'*, *SpCell=array([], dtype=float64)*, *synDelays=None*)

Insert synapse with *parameters* = *synparams* on cell=*cell*, with segment indexes given by *idx*. *SpCell* and *SpTimes* picked from Brunel network simulation

**Parameters**

**cell** [*LFPy.Cell* instance] Postsynaptic target cell.

**cellindex** [int] Index of cell in population.

**synParams** [dict] Parameters passed to *LFPy.Synapse*.

**idx** [numpy.ndarray] Postsynaptic compartment indices.

**X** [str] presynaptic population name

**SpCell** [numpy.ndarray] Presynaptic spiking cells.

**synDelays** [numpy.ndarray] Per connection specific delays.

**Returns**

None

See also:

*Population.insert\_all\_synapses*

## CLASS POSTPROCESS

```
class hybridLFPy.PostProcess(y=['EX', 'IN'], dt_output=1.0, mapping_Yy=[('EX', 'EX'), ('IN', 'IN')],
                             savelist=['somapos'], probes=[],
                             savefolder='simulation_output_example_brunel', cells_subfolder='cells',
                             populations_subfolder='populations', figures_subfolder='figures',
                             output_file='{}_population_{}.h5', compound_file='{}_sum.h5')
```

Bases: object

class *PostProcess*: Methods to deal with the contributions of every postsynaptic sub-population.

### Parameters

- y** [list] Postsynaptic cell-type or population-names.
- dt\_output** [float] Time resolution of output data.
- savelist** [list] List of strings, each corresponding to LFPy.Cell attributes
- probes** [list] list of LFPykit.models.\* like instances
- savefolder** [str] Path to main output folder.
- mapping\_Yy** [list] List of tuples, each tuple pairing population with cell type, e.g., [(‘L4E’, ‘p4’), (‘L4E’, ‘ss4’)].
- cells\_subfolder** [str] Folder under *savefolder* containing cell output.
- populations\_subfolder** [str] Folder under *savefolder* containing population specific output.
- figures\_subfolder** [str] Folder under *savefolder* containing figs.

**calc\_measure**(*measure*='LFP')

Sum all the measure contributions from every cell type.

### Parameters

**measure: str** ‘LFP’, ‘CSD’ or ‘current\_dipole\_moment’

### Returns

**measure\_dict: dict of ndarray** Contributions by each cell type y

**measure\_sum: ndarray** Summed contributions of all cell types

**calc\_measure\_layer**(*datadict*, *measure*='LFP')

Calculate the measure from concatenated subpopulations residing in a certain layer, e.g all L4E pops are summed, according to the *mapping\_Yy* attribute of the *hybridLFPy.Population* objects.

### Parameters

**datadict: dict**

**measure:** str

**Returns**

**measure\_dict:** dict of ndarray Contributions by each subpopulation Y

**create\_tar\_archive()**

Create a tar archive of the main simulation outputs.

**run()**

Perform the postprocessing steps, computing compound signals from cell-specific output files.

## CLASS GDF

```
class hybridLFPy.GDF(dbname, bsize=1000000, new_db=True, debug=False)
```

Bases: object

1. Read from gdf files.
2. Create sqlite db of (neuron, spike time).
3. Query spike times for neurons.

**Parameters**

**dbname** [str] Filename of sqlite database, see *sqlite3.connect*

**bsize** [int] Number of spike times to insert.

**new\_db** [bool] New database with name dbname, will overwrite at a time, determines memory usage.

**Returns**

*hybridLFPy.gdf.GDF* object

See also:

**sqlite3**, **sqlite3.connect**, **sqlite3.connect.cursor**

**close()**

Close *sqlite3.connect.cursor* and *sqlite3.connect* objects

**Parameters**

**None**

**Returns**

**None**

See also:

**sqlite3.connect.cursor**, **sqlite3.connect**

```
create(re='brunel-py-ex-*.gdf', index=True, skiprows=0)
```

Create db from list of gdf file glob

**Parameters**

**re** [str] File glob to load.

**index** [bool] Create index on neurons for speed.

**skiprows** [int] Number of skipped first lines

**Returns**

None

See also:

**sqlite3.connect.cursor, sqlite3.connect**

**create\_from\_list**(*re=[]*, *index=True*)

Create db from list of arrays.

**Parameters**

**re** [list] Index of element is cell index, and element *i* an array of spike times in ms.

**index** [bool] Create index on neurons for speed.

**Returns**

None

See also:

**sqlite3.connect.cursor, sqlite3.connect**

**interval**(*T=[0, 1000]*)

Get all spikes in a time interval T.

**Parameters**

**T** [list] Time interval.

**Returns**

**s** [list] Nested list with spike times.

See also:

**sqlite3.connect.cursor**

**neurons**()

Return list of neuron indices.

**Parameters**

None

**Returns**

**list** list of neuron indices

See also:

**sqlite3.connect.cursor**

**num\_spikes**()

Return total number of spikes.

**Parameters**

None

**Returns**

**list**

**plotstuff**(*T*=[0, 1000])

Create a scatter plot of the contents of the database, with entries on the interval T.

**Parameters**

**T** [list] Time interval.

**Returns**

**None**

**See also:**

[\*GDF.select\\_neurons\\_interval\*](#)

**select**(*neurons*)

Select spike trains.

**Parameters**

**neurons** [numpy.ndarray or list] Array of list of neurons.

**Returns**

**list** List of numpy.ndarray objects containing spike times.

**See also:**

**sqlite3.connect.cursor**

**select\_neurons\_interval**(*neurons*, *T*=[0, 1000])

Get all spikes from neurons in a time interval T.

**Parameters**

**neurons** [list] network neuron indices

**T** [list] Time interval.

**Returns**

**s** [list] Nested list with spike times.

**See also:**

**sqlite3.connect.cursor**





## SUBMODULE HELPERS

Documentation:

This is a script containing general helper functions.

`hybridLFPy.helpers.autocorrfunc(freq, power)`

Calculate autocorrelation function(s) for given power spectrum/spectra.

### Parameters

**freq** [numpy.ndarray] 1 dimensional array of frequencies.

**power** [numpy.ndarray] 2 dimensional power spectra, 1st axis units, 2nd axis frequencies.

### Returns

**time** [tuple] 1 dim numpy.ndarray of times.

**autof** [tuple] 2 dim numpy.ndarray; autocorrelation functions, 1st axis units, 2nd axis times.

`hybridLFPy.helpers.calculate_fft(data, tbin)`

Function to calculate the Fourier transform of data.

### Parameters

**data** [numpy.ndarray] 1D or 2D array containing time series.

**tbin** [float] Bin size of time series (in ms).

### Returns

**freqs** [numpy.ndarray] Frequency axis of signal in Fourier space.

**fft** [numpy.ndarray] Signal in Fourier space.

`hybridLFPy.helpers.centralize(data, time=False, units=False)`

Function to subtract the mean across time and/or across units from data

### Parameters

**data** [numpy.ndarray] 1D or 2D array containing time series, 1st index: unit, 2nd index: time

**time** [bool] True: subtract mean across time.

**units** [bool] True: subtract mean across units.

### Returns

**numpy.ndarray** 1D or 0D array of centralized signal.

`hybridLFPy.helpers.coherence(freq, power, cross)`

Calculate frequency resolved coherence for given power- and crossspectra.

### Parameters

**freq** [numpy.ndarray] Frequencies, 1 dim array.

**power** [numpy.ndarray] Power spectra, 1st axis units, 2nd axis frequencies.

**cross** [numpy.ndarray,] Cross spectra, 1st axis units, 2nd axis units, 3rd axis frequencies.

#### Returns

**freq: tuple** 1 dim numpy.ndarray of frequencies.

**coh: tuple** ndim 3 numpy.ndarray of coherences, 1st axis units, 2nd axis units, 3rd axis frequencies.

hybridLFPy.helpers.**compound\_crossspec**(*a\_data*, *tbin*, *Df=None*, *pointProcess=False*)

Calculate cross spectra of compound signals. *a\_data* is a list of datasets (*a\_data* = [data1,data2,...]). For each dataset in *a\_data*, the compound signal is calculated and the crossspectra between these compound signals is computed.

If *pointProcess=True*, power spectra are normalized by the length *T* of the time series.

#### Parameters

**a\_data** [list of numpy.ndarrays] Array: 1st axis unit, 2nd axis time.

**tbin** [float] Binsize in ms.

**Df** [float/None,] Window width of sliding rectangular filter (smoothing), None -> no smoothing.

**pointProcess** [bool] If set to True, crossspectrum is normalized to signal length *T*

#### Returns

**freq** [tuple] numpy.ndarray of frequencies.

**CRO** [tuple] 3 dim numpy.ndarray; 1st axis first compound signal, 2nd axis second compound signal, 3rd axis frequency.

### Examples

```
>>> compound_crossspec([np.array([analog_sig1, analog_sig2]),
                          np.array([analog_sig3, analog_sig4])], tbin, Df=Df)
Out[1]: (freq,CRO)
>>> CRO.shape
Out[2]: (2,2,len(analog_sig1))
```

hybridLFPy.helpers.**compound\_mean**(*data*)

Compute the mean of the compound/sum signal. Data is first summed across units and averaged across time.

#### Parameters

**data** [numpy.ndarray] 1st axis unit, 2nd axis time

#### Returns

**float** time-averaged compound/sum signal

## Examples

```
>>> compound_mean(np.array([[1, 2, 3], [4, 5, 6]]))
7.0
```

hybridLFPy.helpers.**compound\_powerspec**(data, tbin, Df=None, pointProcess=False)

Calculate the power spectrum of the compound/sum signal. data is first summed across units, then the power spectrum is calculated.

If pointProcess=True, power spectra are normalized by the length T of the time series.

### Parameters

**data** [numpy.ndarray,] 1st axis unit, 2nd axis time

**tbin** [float,] binsize in ms

**Df** [float/None,] window width of sliding rectangular filter (smoothing), None -> no smoothing

**pointProcess** [bool,] if set to True, powerspectrum is normalized to signal length T

### Returns

**freq** [tuple] numpy.ndarray of frequencies

**POW** [tuple] 1 dim numpy.ndarray, frequency series

## Examples

```
>>> compound_powerspec(np.array([analog_sig1, analog_sig2]), tbin, Df=Df)
Out[1]: (freq, POW)
>>> POW.shape
Out[2]: (len(analog_sig1),)
```

hybridLFPy.helpers.**compound\_variance**(data)

Compute the variance of the compound/sum signal. Data is first summed across units, then the variance across time is calculated.

### Parameters

**data** [numpy.ndarray] 1st axis unit, 2nd axis time

### Returns

**float** variance across time of compound/sum signal

## Examples

```
>>> compound_variance(np.array([[1, 2, 3], [4, 5, 6]]))
2.6666666666666665
```

hybridLFPy.helpers.**corrcoef**(time, crossf, integration\_window=0.0)

Calculate the correlation coefficient for given auto- and crosscorrelation functions. Standard settings yield the zero lag correlation coefficient. Setting integration\_window > 0 yields the correlation coefficient of integrated auto- and crosscorrelation functions. The correlation coefficient between a zero signal with any other signal is defined as 0.

### Parameters

**time** [numpy.ndarray] 1 dim array of times corresponding to signal.

**crossf** [numpy.ndarray] Crosscorrelation functions, 1st axis first unit, 2nd axis second unit, 3rd axis times.

**integration\_window: float** Size of the integration window.

#### Returns

**cc** [numpy.ndarray] 2 dim array of correlation coefficient between two units.

hybridLFPy.helpers.**crosscorrfunc**(*freq*, *cross*)

Calculate crosscorrelation function(s) for given cross spectra.

#### Parameters

**freq** [numpy.ndarray] 1 dimensional array of frequencies.

**cross** [numpy.ndarray] 2 dimensional array of cross spectra, 1st axis units, 2nd axis units, 3rd axis frequencies.

#### Returns

**time** [tuple] 1 dim numpy.ndarray of times.

**crossf** [tuple] 3 dim numpy.ndarray, crosscorrelation functions, 1st axis first unit, 2nd axis second unit, 3rd axis times.

hybridLFPy.helpers.**crossspec**(*data*, *tbin*, *Df*=None, *units*=False, *pointProcess*=False)

Calculate (smoothed) cross spectra of data. If `units=True`, cross spectra are averaged across units. Note that averaging is done on cross spectra rather than data.

Cross spectra are normalized by the length T of the time series -> no scaling with T.

If `pointProcess=True`, power spectra are normalized by the length T of the time series.

#### Parameters

**data** [numpy.ndarray,] 1st axis unit, 2nd axis time

**tbin** [float,] binsize in ms

**Df** [float/None,] window width of sliding rectangular filter (smoothing), None -> no smoothing

**units** [bool,] average cross spectrum

**pointProcess** [bool,] if set to True, cross spectrum is normalized to signal length T

#### Returns

**freq** [tuple] numpy.ndarray of frequencies

**CRO** [tuple] if `units=True`: 1 dim numpy.ndarray; frequency series if `units=False`: 3 dim numpy.ndarray; 1st axis first unit, 2nd axis second unit, 3rd axis frequency

## Examples

```
>>> crossspec(np.array([analog_sig1, analog_sig2]), tbin, Df=Df)
Out[1]: (freq,CRO)
>>> CRO.shape
Out[2]: (2,2,len(analog_sig1))
```

```
>>> crossspec(np.array([analog_sig1, analog_sig2]), tbin, Df=Df, units=True)
Out[1]: (freq,CRO)
>>> CRO.shape
Out[2]: (len(analog_sig1),)
```

hybridLFPy.helpers.**cv**(data, units=False)

Calculate coefficient of variation (cv) of data. Mean and standard deviation are computed across time.

### Parameters

**data** [numpy.ndarray] 1st axis unit, 2nd axis time.

**units** [bool] Average cv.

### Returns

**numpy.ndarray** If units=False, series of unit `cv`s.

**float** If units=True, mean cv across units.

## Examples

```
>>> cv(np.array([[1, 2, 3, 4, 5, 6], [11, 2, 3, 3, 4, 5]]))
array([ 0.48795004, 0.63887656])
```

```
>>> cv(np.array([[1, 2, 3, 4, 5, 6], [11, 2, 3, 3, 4, 5]]), units=True)
0.56341330073710316
```

hybridLFPy.helpers.**dump\_dict\_of\_nested\_lists\_to\_h5**(fname, data)

Take nested list structure and dump it in hdf5 file.

### Parameters

**fname** [str] Filename

**data** [dict(list(numpy.ndarray))] Dict of nested lists with variable len arrays.

### Returns

**None**

hybridLFPy.helpers.**fano**(data, units=False)

Calculate fano factor (FF) of data. Mean and variance are computed across time.

### Parameters

**data** [numpy.ndarray] 1st axis unit, 2nd axis time.

**units** [bool] Average FF.

### Returns

**numpy.ndarray** If units=False, series of unit FFs.

**float** If units=True, mean FF across units.

## Examples

```
>>> fano(np.array([[1, 2, 3, 4, 5, 6], [11, 2, 3, 3, 4, 5]]))
array([ 0.83333333,  1.9047619 ])
```

```
>>> fano(np.array([[1, 2, 3, 4, 5, 6], [11, 2, 3, 3, 4, 5]]), units=True)
1.3690476190476191
```

`hybridLFPy.helpers.load_dict_of_nested_lists_from_h5(fname, toplevelkeys=None)`  
Load nested list structure from hdf5 file

### Parameters

**fname** [str] Filename

**toplevelkeys** [None or iterable,] Load a two(default) or three-layered structure.

### Returns

**dict(list(numpy.ndarray))** dictionary of nested lists with variable length array data.

`hybridLFPy.helpers.load_h5_data(path="", data_type='LFP', y=None, electrode=None, warmup=0.0, scaling=1.0)`

Function loading results from hdf5 file

### Parameters

**path** [str] Path to hdf5-file

**data\_type** [str] Signal types in ['CSD', 'LFP', 'CSDsum', 'LFPsum'].

**y** [None or str] Name of population.

**electrode** [None or int] TODO: update, electrode is NOT USED

**warmup** [float] Lower cutoff of time series to remove possible transients

**scaling** [float,] Scaling factor for population size that determines the amount of loaded single-cell signals

### Returns

**numpy.ndarray** [electrode id, compound signal] if y is None

**numpy.ndarray** [cell id, electrode, single-cell signal] otherwise

`hybridLFPy.helpers.mean(data, units=False, time=False)`

Function to compute mean of data

### Parameters

**data** [numpy.ndarray] 1st axis unit, 2nd axis time

**units** [bool] Average over units

**time** [bool] Average over time

### Returns

**if units=False and time=False:** error

**if units=True:** 1 dim numpy.ndarray; time series

**if time=True:** 1 dim numpy.ndarray; series of unit means across time

**if units=True and time=True:** float; unit and time mean

## Examples

```
>>> mean(np.array([[1, 2, 3], [4, 5, 6]]), units=True)
array([ 2.5,  3.5,  4.5])
```

```
>>> mean(np.array([[1, 2, 3], [4, 5, 6]]), time=True)
array([ 2.,  5.])
```

```
>>> mean(np.array([[1, 2, 3], [4, 5, 6]]), units=True, time=True)
3.5
```

hybridLFPy.helpers.**movav**(y, Dx, dx)

Moving average rectangular window filter: calculate average of signal y by using sliding rectangular window of size Dx using binsize dx

### Parameters

**y** [numpy.ndarray] Signal

**Dx** [float] Window length of filter.

**dx** [float] Bin size of signal sampling.

### Returns

**numpy.ndarray** Filtered signal.

hybridLFPy.helpers.**normalize**(data)

Function to normalize data to have mean 0 and unity standard deviation (also called z-transform)

### Parameters

**data** [numpy.ndarray]

### Returns

**numpy.ndarray** z-transform of input array

hybridLFPy.helpers.**powerspec**(data, tbin, Df=None, units=False, pointProcess=False)

Calculate (smoothed) power spectra of all timeseries in data. If units=True, power spectra are averaged across units. Note that averaging is done on power spectra rather than data.

If pointProcess is True, power spectra are normalized by the length T of the time series.

### Parameters

**data** [numpy.ndarray] 1st axis unit, 2nd axis time.

**tbin** [float] Binsize in ms.

**Df** [float/None,] Window width of sliding rectangular filter (smoothing), None is no smoothing.

**units** [bool] Average power spectrum.

**pointProcess** [bool] If set to True, powerspectrum is normalized to signal length T.

### Returns

**freq** [tuple] numpy.ndarray of frequencies.

**POW** [tuple]

**if units=False:** 2 dim numpy.ndarray; 1st axis unit, 2nd axis frequency

**if units=True:** 1 dim numpy.ndarray; frequency series

### Examples

```
>>> powerspec(np.array([analog_sig1, analog_sig2]), tbin, Df=Df)
Out[1]: (freq, POW)
>>> POW.shape
Out[2]: (2, len(analog_sig1))
```

```
>>> powerspec(np.array([analog_sig1, analog_sig2]), tbin, Df=Df, units=True)
Out[1]: (freq, POW)
>>> POW.shape
Out[2]: (len(analog_sig1),)
```

hybridLFPy.helpers.**read\_gdf**(fname, skiprows=0)

Fast line-by-line gdf-file reader.

#### Parameters

**fname** [str] Path to gdf-file.

**skiprows** [int] number of skipped rows

#### Returns

**numpy.ndarray** ([gid, val0, val1, \*\*]), dtype=object) mixed datatype array

hybridLFPy.helpers.**setup\_file\_dest**(params, clearDestination=True)

Function to set up the file catalog structure for simulation output

#### Parameters

**params** [object] e.g., *cellsim16popsParams.multicompartment\_params()*

**clear\_dest** [bool] Savefolder will be cleared if already existing.

#### Returns

None

hybridLFPy.helpers.**variance**(data, units=False, time=False)

Compute the variance of data across time, units or both.

#### Parameters

**data** [numpy.ndarray] 1st axis unit, 2nd axis time.

**units** [bool] Variance across units

**time** [bool] Average over time

#### Returns

**if units=False and time=False:** Exception

**if units=True:** 1 dim numpy.ndarray; time series

**if time=True:** 1 dim numpy.ndarray; series of single unit variances across time

**if units=True and time=True:** float; mean of single unit variances across time



## Examples

```
>>> variance(np.array([[1, 2, 3],[4, 5, 6]]), units=True)
array([ 2.25,  2.25,  2.25])
```

```
>>> variance(np.array([[1, 2, 3], [4, 5, 6]]), time=True)
array([ 0.66666667,  0.66666667])
```

```
>>> variance(np.array([[1, 2, 3], [4, 5, 6]]), units=True, time=True)
0.66666666666666663
```

hybridLFPy.helpers.**write\_gdf**(*gdf*, *fname*)

Fast line-by-line gdf-file write function

### Parameters

**gdf** [numpy.ndarray] Column 0 is gids, columns 1: are values.

**fname** [str] Path to gdf-file.

### Returns

None



## SUBMODULUE TEST

`hybridLFPy.test(verbosity=2)`  
Run unittests for hybridLFPy

**Returns**

**None**



## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### h

`hybridLFPy`, [5](#)

`hybridLFPy.helpers`, [29](#)





## A

`autocorrfunc()` (in module `hybridLFPy.helpers`), 29

## C

`CachedFixedSpikesNetwork` (class in `hybridLFPy`), 13

`CachedNetwork` (class in `hybridLFPy`), 7

`CachedNoiseNetwork` (class in `hybridLFPy`), 11

`calc_measure()` (`hybridLFPy.PostProcess` method), 23

`calc_measure_layer()` (`hybridLFPy.PostProcess` method), 23

`calc_min_cell_interdist()` (`hybridLFPy.PopulationSuper` method), 16

`calc_signal_sum()` (`hybridLFPy.PopulationSuper` method), 16

`calculate_fft()` (in module `hybridLFPy.helpers`), 29

`cellsim()` (`hybridLFPy.Population` method), 19

`cellsim()` (`hybridLFPy.PopulationSuper` method), 16

`centralize()` (in module `hybridLFPy.helpers`), 29

`close()` (`hybridLFPy.GDF` method), 25

`coherence()` (in module `hybridLFPy.helpers`), 29

`collect_data()` (`hybridLFPy.PopulationSuper` method), 16

`collect_gdf()` (`hybridLFPy.CachedNetwork` method), 7

`collect_savelist()` (`hybridLFPy.PopulationSuper` method), 17

`collectSingleContribs()` (`hybridLFPy.PopulationSuper` method), 16

`compound_crossspec()` (in module `hybridLFPy.helpers`), 30

`compound_mean()` (in module `hybridLFPy.helpers`), 30

`compound_powerspec()` (in module `hybridLFPy.helpers`), 31

`compound_variance()` (in module `hybridLFPy.helpers`), 31

`corrcoef()` (in module `hybridLFPy.helpers`), 31

`create()` (`hybridLFPy.GDF` method), 25

`create_from_list()` (`hybridLFPy.GDF` method), 26

`create_tar_archive()` (`hybridLFPy.PostProcess` method), 24

`crosscorrfunc()` (in module `hybridLFPy.helpers`), 32

`crossspec()` (in module `hybridLFPy.helpers`), 32

`cv()` (in module `hybridLFPy.helpers`), 33

## D

`draw_rand_pos()` (`hybridLFPy.PopulationSuper` method), 17

`dump_dict_of_nested_lists_to_h5()` (in module `hybridLFPy.helpers`), 33

## F

`fano()` (in module `hybridLFPy.helpers`), 33

`fetchSpCells()` (`hybridLFPy.Population` method), 20

`fetchSynIdxCell()` (`hybridLFPy.Population` method), 20

## G

`GDF` (class in `hybridLFPy`), 25

`get_all_SpCells()` (`hybridLFPy.Population` method), 20

`get_all_synDelays()` (`hybridLFPy.Population` method), 21

`get_all_synIdx()` (`hybridLFPy.Population` method), 21

`get_synidx()` (`hybridLFPy.Population` method), 21

`get_xy()` (`hybridLFPy.CachedNetwork` method), 7

## H

`hybridLFPy`

module, 5

`hybridLFPy.helpers`

module, 29

## I

`insert_all_synapses()` (`hybridLFPy.Population` method), 21

`insert_synapses()` (`hybridLFPy.Population` method), 22

`interval()` (`hybridLFPy.GDF` method), 26

## L

`load_dict_of_nested_lists_from_h5()` (in module `hybridLFPy.helpers`), 34

`load_h5_data()` (in module *hybridLFPy.helpers*), 34

## M

`mean()` (in module *hybridLFPy.helpers*), 34

module

*hybridLFPy*, 5

*hybridLFPy.helpers*, 29

`movav()` (in module *hybridLFPy.helpers*), 35

## N

`neurons()` (*hybridLFPy.GDF* method), 26

`normalize()` (in module *hybridLFPy.helpers*), 35

`num_spikes()` (*hybridLFPy.GDF* method), 26

## P

`plot_f_rate()` (*hybridLFPy.CachedNetwork* method),  
8

`plot_raster()` (*hybridLFPy.CachedNetwork* method),  
8

`plotstuff()` (*hybridLFPy.GDF* method), 27

`Population` (class in *hybridLFPy*), 19

`PopulationSuper` (class in *hybridLFPy*), 15

`PostProcess` (class in *hybridLFPy*), 23

`powerspec()` (in module *hybridLFPy.helpers*), 35

## R

`raster_plots()` (*hybridLFPy.CachedNetwork* method),  
8

`read_gdf()` (in module *hybridLFPy.helpers*), 36

`run()` (*hybridLFPy.PopulationSuper* method), 17

`run()` (*hybridLFPy.PostProcess* method), 24

## S

`select()` (*hybridLFPy.GDF* method), 27

`select_neurons_interval()` (*hybridLFPy.GDF*  
method), 27

`set_pop_soma_pos()` (*hybridLFPy.PopulationSuper*  
method), 17

`set_rotations()` (*hybridLFPy.PopulationSuper*  
method), 18

`setup_file_dest()` (in module *hybridLFPy.helpers*),  
36

## T

`test()` (in module *hybridLFPy*), 39

## V

`variance()` (in module *hybridLFPy.helpers*), 36

## W

`write_gdf()` (in module *hybridLFPy.helpers*), 37