spykeutils Documentation

Release 0.2.0

Robert Pröpper

December 05, 2012

CONTENTS

Based on the Neo framework, spykeutils is a Python library for analyzing and plotting neurophysiological data. It can be used by itself or in conjunction with Spyke Viewer, a multi-platform GUI application for navigating electrophysiological datasets.

Contents:

CHAPTER

ONE

REQUIREMENTS

Spykeutils is a pure Python package and therefore easy to install. It depends on the following additional packages:

- Python >= 2.7
- neo >= 0.2.2
- scipy
- guiqwt (Optional, for plotting)
- tables (Optional, for analysis results data management. Also known as PyTables.)

Please see the respective websites for instructions on how to install them if they are not present on your computer.

Note: The current version of Neo in the Python Package Index contains some bugs that prevent it from working properly with spykeutils in some situations. Please install the latest version directly from GitHub: https://github.com/rproepp/python-neo

You can download the repository from the GitHub page or clone it using git and then install from the resulting folder:

```
$ python setup.py install
```

CHAPTER

TWO

DOWNLOAD AND INSTALLATION

The easiest way to get spykeutils is from the Python Package Index. If you have pip installed:

```
$ pip install spykeutils
```

Alternatively, if you have setuptools:

```
$ easy_install spykeutils
```

Alternatively, you can get the latest version directly from GitHub at https://github.com/rproepp/spykeutils.

The master branch (selected by default) always contains the current stable version. If you want the latest development version (not recommended unless you need some features that do not exist in the stable version yet), select the develop branch. You can download the repository from the GitHub page or clone it using git and then install from the resulting folder:

```
$ python setup.py install
```

CHAPTER

THREE

USAGE

For the most part, spykeutils is a collection of functions that work on Neo objects. Many functions also take quantities as parameters. Therefore, make sure to get an overview of `neo` and `quantities` before using spykeutils. Once you are familiar with these packages, have a look at the *Examples* or head to the *API reference* to browse the contents of spykeutils.

CHAPTER

FOUR

EXAMPLES

These examples demonstrate the usage of some functions in spykeutils. This includes the creation of a small Neo object hierarchy with toy data.

4.1 Creating the sample data

The functions in spykeutils work on electrophysiological data that is represented in Neo object hierarchies. Usually, you would load these objects from a file, but for the purpose of this demonstration we will manually create an object hierarchy to illustrate their structure. Note that most functions in spykeutils will also work with separate Neo data objects that are not contained in a complete hierarchy. First, we import the modules we will use:

```
>>> import quantities as pq
>>> import neo
>>> import scipy as sp
```

We start with some container objects - two segments that represent trials and two units (representing neurons) that produced the spike trains:

```
>>> segments = []
>>> segments.append(neo.Segment('Trial 1'))
>>> segments.append(neo.Segment('Trial 2'))
>>> units = []
>>> units.append(neo.Unit('Unit 1'))
>>> units.append(neo.Unit('Unit 2'))
```

We create some analog signals from a sine wave with additive Gaussian noise, four signals in each segment:

```
>>> wave = sp.sin(sp.linspace(0, 20*sp.pi, 10000)) * 10
>>> for i in range(8):
...     sig = wave + sp.randn(10000) * 3
...     signal = neo.AnalogSignal(sig*pq.uV, sampling_rate=1*pq.kHz)
...     signal.segment = segments[i%2]
...     segments[i%2].analogsignals.append(signal)
```

And some spike trains from regular intervals or random time points:

```
>>> trains = []
>>> trains.append(neo.SpikeTrain(sp.linspace(0, 10, 40)*pq.s, 10*pq.s))
>>> trains.append(neo.SpikeTrain(sp.linspace(0, 10, 60)*pq.s, 10*pq.s))
>>> trains.append(neo.SpikeTrain(sp.rand(50)*10*pq.s, 10*pq.s))
>>> trains.append(neo.SpikeTrain(sp.rand(70)*10*pq.s, 10*pq.s))
```

Now we create the relationships between the spike trains and container objects. Each unit has two spike trains, one in each segment:

```
>>> segments[0].spiketrains = [trains[0], trains[2]]
>>> segments[1].spiketrains = [trains[1], trains[3]]
>>> units[0].spiketrains = trains[:2]
>>> units[1].spiketrains = trains[2:4]
>>> for s in segments:
...     for st in s.spiketrains:
...         st.segment = s
>>> for u in units:
...     for st in u.spiketrains:
...         st.unit = u
```

Now that our sample data is ready, we will use some of the function from spykeutils to analyze it.

4.2 PSTH

To create a peri stimulus time histogram from our spike trains, we call `spykeutils.rate_estimation.psth()`. This function can create multiple PSTHs and takes a dictionary of lists of spike trains. Since our spike trains were generated by two units, we will create two histograms, one for each unit:

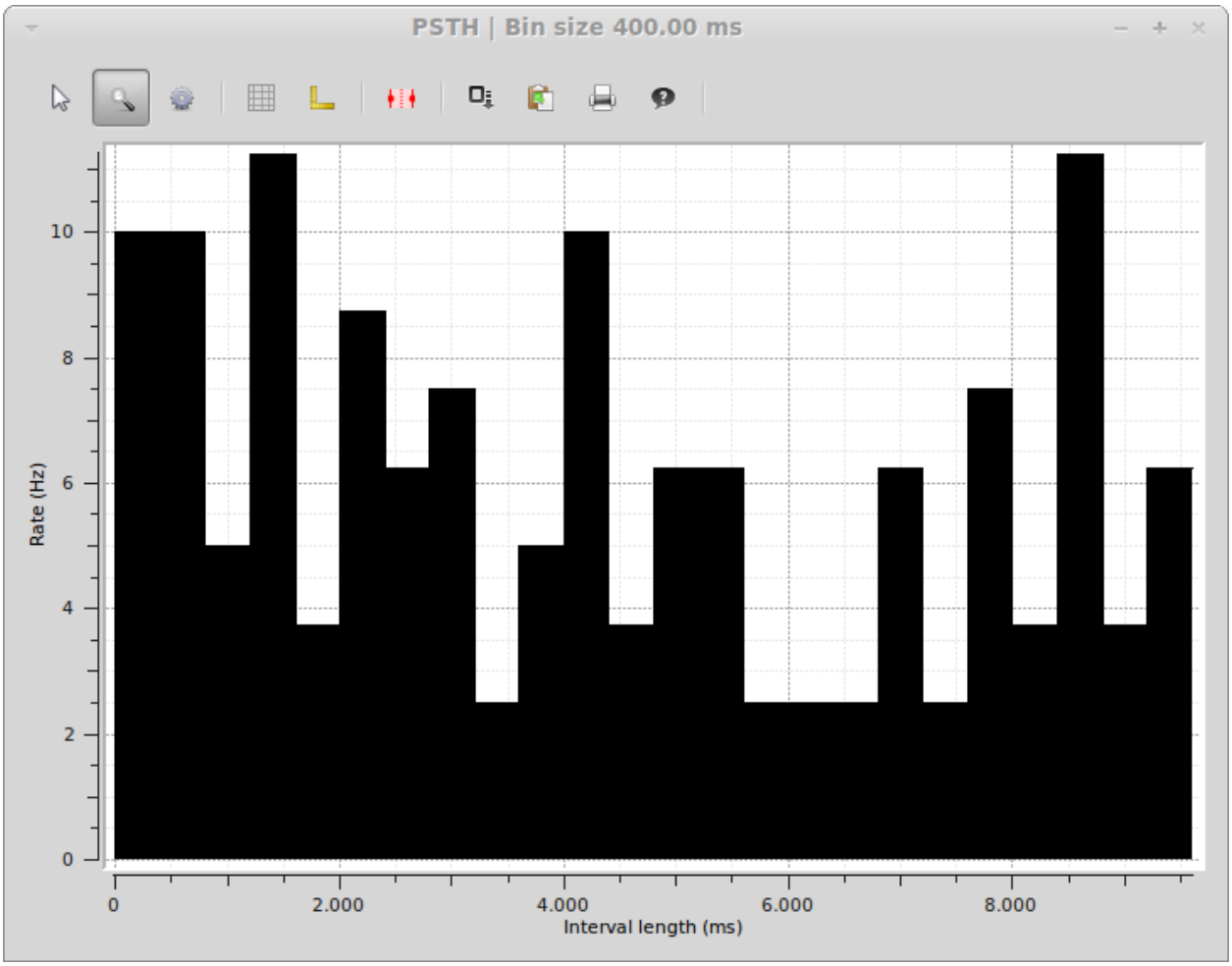
```
>>> import spykeutils.rate_estimation
>>> st_dict = {}
>>> st_dict[units[0]] = units[0].spiketrains
>>> st_dict[units[1]] = units[1].spiketrains
>>> spykeutils.rate_estimation.psth(st_dict, 400*pq.ms)[0]
{<neo.core.unit.Unit object at 0x...>: array([ 6.25,  5.  ,  5.  ,  5.  ,  3.75, ...
```

`spykeutils.rate_estimation.psth()` returns two values: A dictionary with the resulting histograms and a Quantity 1D with the bin edges.

If `guiqwt` is installed, we can also use the `spykeutils.plot` package to create a PSTH plot from our data (in this case we want a bar histogram and therefore only use spike trains from one unit):

```
>>> import spykeutils.plot
>>> spykeutils.plot.psth({units[1]: units[1].spiketrains}, bin_size=400*pq.ms, bar_plot=True)
```

This will open a plot window like the following:



4.3 Spike Density Estimation

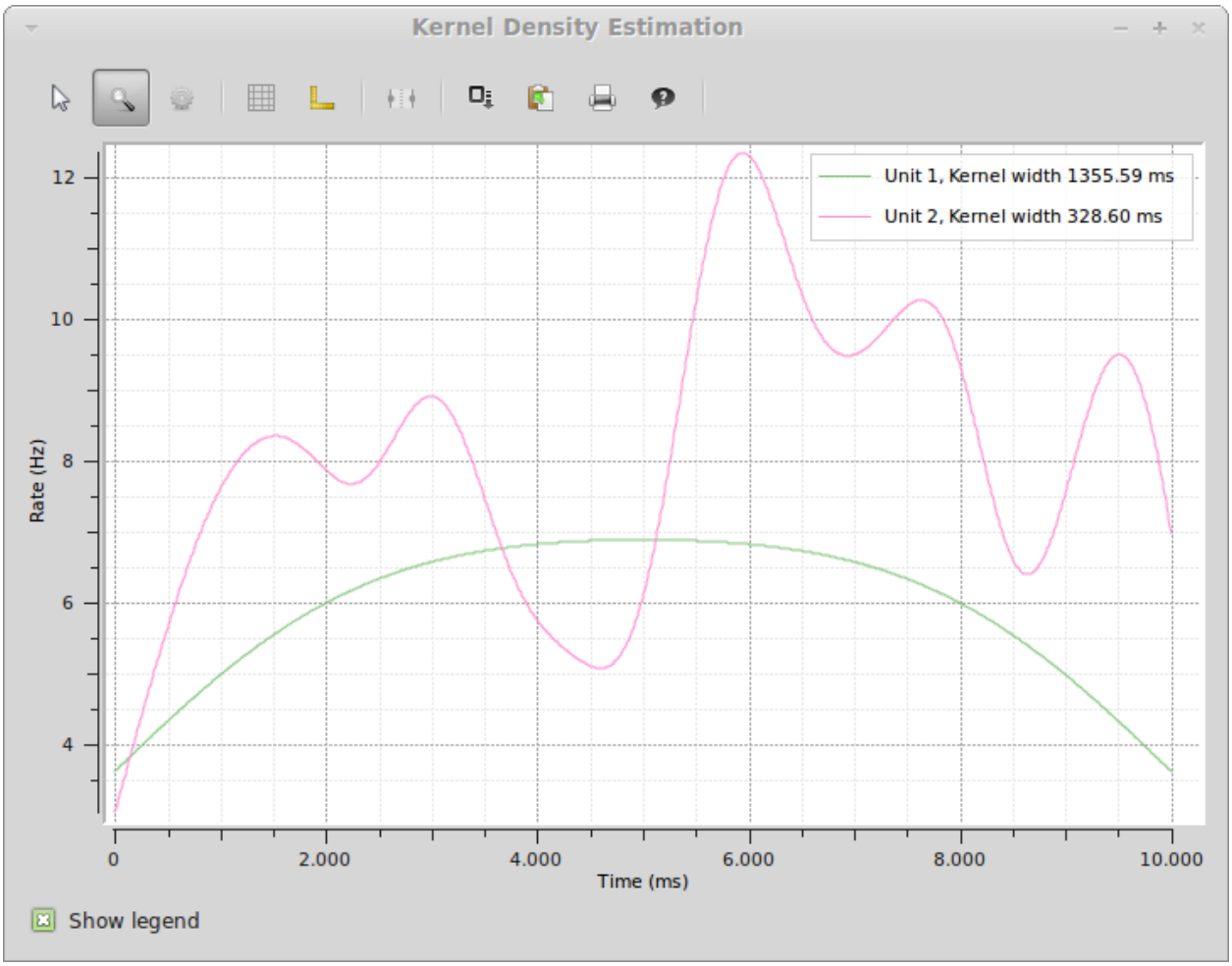
Similar to a PSTH, a spike density estimation gives an esimate of the instantaneous firing rate. Instead of binning, it is based on a kernel convolution which results in a smoother estimate. Creating and SDE with spykeutils works very similar to creating a PSTH. Instead of manually choosing the size of the Gaussian kernel, `spykeutils.rate_estimation.spike_density_estimation()` also supports finding the optimal kernel size automatically for each unit:

```
>>> kernel_sizes = sp.logspace(2,3.3,100) * pq.ms
>>> spykeutils.rate_estimation.spike_density_estimation(st_dict, optimize_steps=kernel_sizes)[0]
{<neo.core.unit.Unit object at 0x...>: array([ 3.61293378,  3.62744654,  3.64195481,  3.65645819, ...
```

As with the PSTH, there is also a plot function for creating a spike density estimation. Here, we use both units because the function produces a line plot where both units can be shown at the same time:

```
>>> spykeutils.plot.sde(st_dict, maximum_kernel=1500*pq.ms, optimize_steps=100)
```

The resulting plot will look like the following:

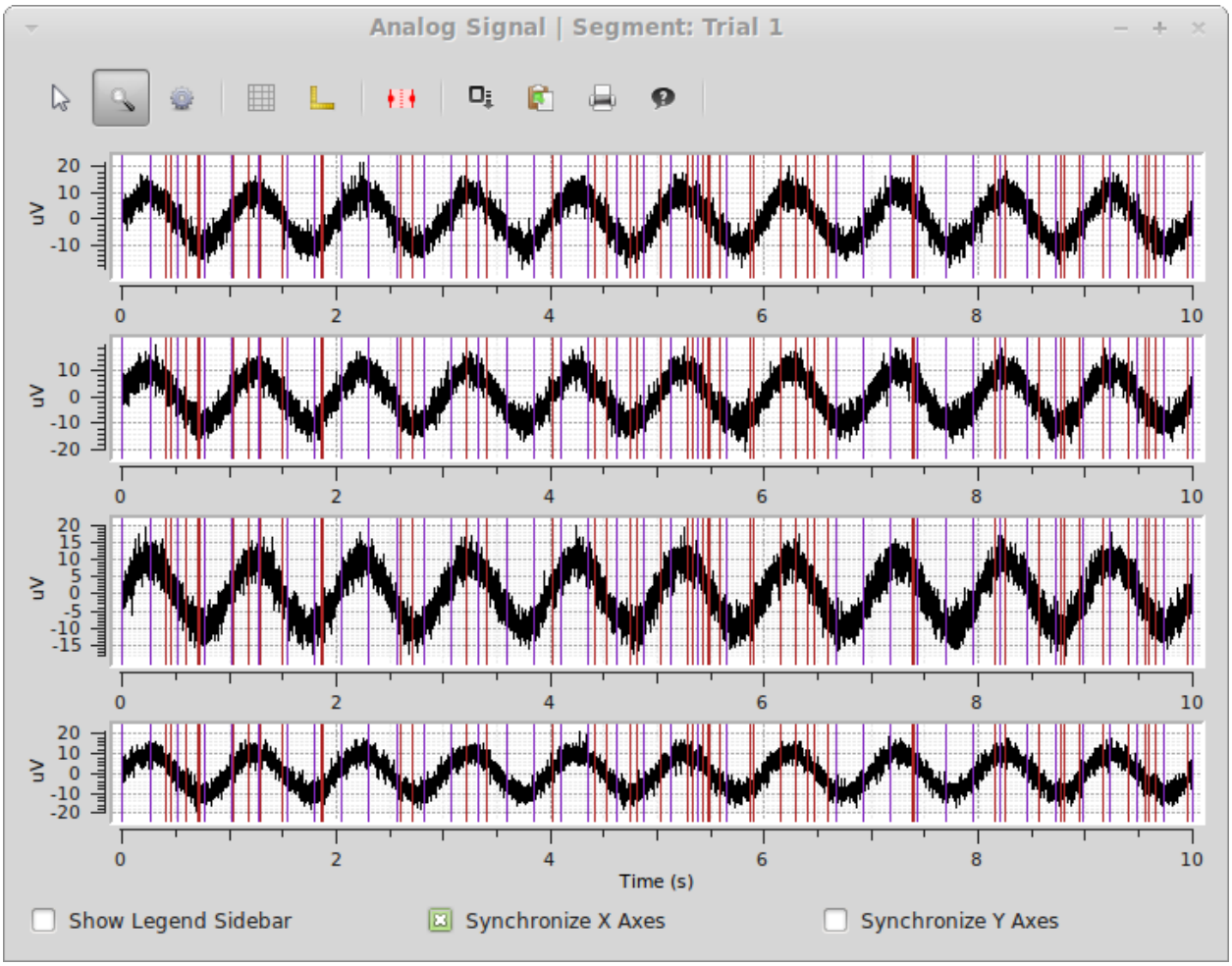


While spike density estimations are preferable to PSTHs in many cases, the picture also shows an important weakness: The estimation will generally be too low on margins. The areas where this happens become larger with kernel size, which is clearly visible from the rounded shape of Unit 1 (which really has a flat rate) with its very large kernel size.

4.4 Signal Plot

As a final example, we will again use the `spykeutils.plot` package to create a plot of the signals we created. This plot will also display the timings of our spike trains.

```
>>> spykeutils.plot.signals(segments[0].analogsignals, spike_trains=segments[0].spiketrains)
```



The plot shows all four signals from the first segments as well as the spike times of both spike trains in the same segment.

CHAPTER

FIVE

API REFERENCE

5.1 spykeutils package

class SpykeException
Exception thrown when a function in spykeutils encounters a problem that is not covered by standard exceptions.

When using Spyke Viewer, these exceptions will be caught and shown in the GUI, while general exceptions will not be caught (and therefore be visible in the console) for easier debugging.

5.1.1 conversions Module

analog_signal_array_to_analog_signals(*signal_array*)
Return a list of analog signals for an analog signal array.

If `signal_array` is attached to a recording channel group with exactly is many channels as there are channels in `signal_array`, each created signal will be assigned the corresponding channel. If the attached recording channel group has only one recording channel, all created signals will be assigned to this channel. In all other cases, the created signal will not have a reference to a recording channel.

Note that while the created signals may have references to a segment and channels, the relationships in the other direction are not automatically created (the signals are not attached to the recording channel or segment). Other properties like annotations are not copied or referenced in the created analog signals.

Parameters **signal_array** (`neo.core.AnalogSignalArray`) – An analog signal array from which the AnalogSignal objects are constructed.

Returns A list of analog signals, one for every channel in `signal_array`.

Return type list

epoch_array_to_epochs(*epoch_array*)
Return a list of epochs for an epoch array.

Note that while the created epochs may have references to a segment, the relationships in the other direction are not automatically created (the events are not attached to the segment). Other properties like annotations are not copied or referenced in the created epochs.

Parameters **epoch_array** (`neo.core.EpochArray`) – A period array from which the Epoch objects are constructed.

Returns A list of events, one for of the events in `epoch_array`.

Return type list

event_array_to_events (*event_array*)
Return a list of events for an event array.

Note that while the created events may have references to a segment, the relationships in the other direction are not automatically created (the events are not attached to the segment). Other properties like annotations are not copied or referenced in the created events.

Parameters **event_array** (`neo.core.EventArray`) – An event array from which the Event objects are constructed.

Returns A list of events, one for of the events in `event_array`.

Return type list

spike_train_to_spikes (*spike_train*, *include_waveforms=True*)
Return a list of spikes for a spike train.

Note that while the created spikes have references to the same segment and unit as the spike train, the relationships in the other direction are not automatically created (the spikes are not attached to the unit or segment). Other properties like annotations are not copied or referenced in the created spikes.

Parameters

- **spike_train** (*SpikeTrain*) – A spike train from which the Spike objects are constructed.
- **include_waveforms** (*bool*) – Determines if the `waveforms` property is converted to the spike waveforms. If `waveforms` is None, this parameter has no effect.

Returns A list of Spike objects, one for every spike in `spike_train`.

Return type list

spikes_to_spike_train (*spikes*, *include_waveforms=True*)
Return a spike train for a list of spikes.

All spikes must have an identical left sweep, the same unit and the same segment, otherwise a `SpykeException` is raised.

Note that while the created spike train has references to the same segment and unit as the spikes, the relationships in the other direction are not automatically created (the spike train is not attached to the unit or segment). Other properties like annotations are not copied or referenced in the created spike train.

Parameters

- **spikes** – A list of spike objects from which the SpikeTrain object is constructed.
- **include_waveforms** (*bool*) – Determines if the waveforms from the Spike objects are used to fill the `waveforms` property of the resulting spike train. If `True`, all spikes need a `waveform` property with the same shape or a `SpykeException` is raised (or the `waveform` property needs to be `None` for all spikes).

Returns A SpikeTrain object including all elements of `spikes`.

Return type `neo.core.SpikeTrain`

5.1.2 `correlations` Module

correlogram (*trains*, *bin_size*, *max_lag=500 ms*, *border_correction=True*, *unit=ms*, *progress=None*)
Return (cross-)correlograms from a dictionary of SpikeTrain lists for different units.

Parameters

- **trains** (*dict*) – Dictionary of SpikeTrain lists.

- **bin_size** (*Quantity scalar*) – Bin size (time).
- **max_lag** (*Quantity scalar*) – Cut off (end time of calculated correlogram).
- **border_correction** (*bool*) – Apply correction for less data at higher timelags. Not perfect for bin_size != 1*‘‘unit‘‘, especially with large `max_lag` compared to length of spike trains.
- **unit** (*Quantity*) – Unit of X-Axis.
- **progress** (`spykeutils.progress_indicator.ProgressIndicator`) – A ProgressIndicator object for the operation.

Returns

Two values:

- An ordered dictionary indexed with the indices of `trains` of ordered dictionaries indexed with the same indices. Entries of the inner dictionaries are the resulting (cross-)correlograms as numpy arrays. All crosscorrelograms can be indexed in two different ways: `c[index1][index2]` and `c[index2][index1]`.
- The bins used for the correlogram calculation.

Return type dict, Quantity 1D

5.1.3 `progress_indicator` Module

exception `CancelException`
Bases: `exceptions.Exception`

This is raised when a user cancels a progress process. It is used by `ProgressIndicator` and its descendants.

class `ProgressIndicator`
Bases: `object`

Base class for classes indicating progress of a long operation.

This class does not implement any of the methods and can be used as a dummy if no progress indication is needed.

`begin`(*title=’‘*)
Signal that the operation starts.

Parameters **title** (*string*) – The name of the whole operation.

`done`()
Signal that the operation is done.

`set_status`(*new_status*)
Set status description.

Parameters **new_status** (*string*) – A description of the current status.

`set_ticks`(*ticks*)
Set the required number of ticks before the operation is done.

Parameters **ticks** (*int*) – The number of steps that the operation will take.

`step`(*num_steps=1*)
Signal that one or more steps of the operation were completed.

Parameters **num_steps** (*int*) – The number of steps that have been completed.

ignores_cancel (*function*)
Decorator for functions that should ignore a raised `CancelException` and just return nothing in this case

5.1.4 rate_estimation Module

binned_spike_trains (*trains*, *bin_size*, *start=0 ms*, *stop=None*)
Return dictionary of binned rates for a dictionary of SpikeTrain lists.

Parameters

- **trains** (*dict*) – A sequence of *SpikeTrain* lists.
- **bin_size** (*Quantity scalar*) – The desired bin size (as a time quantity).
- **stop** (*Quantity scalar*) – The desired time for the end of the last bin. It will be recalculated if there are spike trains which end earlier than this time.

Returns A dictionary (with the same indices as `trains`) of lists of spike train counts and the bin borders.

Return type dict, Quantity 1D

psth (*trains*, *bin_size*, *rate_correction=True*, *start=0 ms*, *stop=None*)
Return dictionary of peri stimulus time histograms for a dictionary of SpikeTrain lists.

Parameters

- **trains** (*dict*) – A dictionary of lists of SpikeTrain objects.
- **bin_size** (*Quantity scalar*) – The desired bin size (as a time quantity).
- **rate_correction** (*bool*) – Determines if a rates (`True`) or counts (`False`) are returned.
- **start** (*Quantity scalar*) – The desired time for the start of the first bin. It will be recalculated if there are spike trains which start later than this time.
- **stop** (*Quantity scalar*) – The desired time for the end of the last bin. It will be recalculated if there are spike trains which end earlier than this time.

Returns A dictionary (with the same indices as `trains`) of arrays containing counts (or rates if `rate_correction` is `True`) and the bin borders.

Return type dict, Quantity 1D

spike_density_estimation (*trains*, *start=0 ms*, *stop=None*, *evaluation_points=None*, *kernel=gauss_kernel*, *kernel_size=100 ms*, *optimize_steps=None*, *progress=None*)
Create a spike density estimation from a dictionary of lists of SpikeTrain objects. The spike density estimations give an estimate of the instantaneous rate. Optionally finds optimal kernel size for given data.

Parameters

- **trains** (*dict*) – A dictionary of SpikeTrain lists.
- **start** (*Quantity scalar*) – The desired time for the start of the first bin. It will be recalculated if there are spike trains which start later than this time. This parameter can be negative (which could be useful when aligning on events).
- **stop** (*Quantity scalar*) – The desired time for the end of the last bin. It will be recalculated if there are spike trains which end earlier than this time.
- **evaluation_points** (*Quantity 1D*) – An array of time points at which the density estimation is evaluated to produce the data. If this is None, 1000 equally spaced points covering the range of the input spike trains will be used.

- **kernel** (*func*) – The kernel function to use, should accept two parameters: A ndarray of distances and a kernel size. The total area under the kernel function sould be 1. Default: Gaussian kernel
- **kernel_size** (*Quantity scalar*) – A uniform kernel size for all spike trains. Only used if optimization of kernel sizes is not used.
- **optimize_steps** (*Quantity 1D*) – An array of time lengths that will be considered in the kernel width optimization. Note that the optimization assumes a Gaussian kernel and will most likely not give the optimal kernel size if another kernel is used. If None, `kernel_size` will be used.
- **progress** (`spykeutils.progress_indicator.ProgressIndicator`) – Set this parameter to report progress.

Returns

Three values:

- A dictionary of the spike density estimations (Quantity 1D in Hz). Indexed the same as `trains`.
- A dictionary of kernel sizes (Quantity scalars). Indexed the same as `trains`.
- The used evaluation points.

Return type dict, dict, Quantity 1D

aligned_spike_trains (*trains*, *events*, *copy=True*)

Return a list of spike trains aligned to an event (the event will be time 0 on the returned trains).

Parameters

- **trains** (*dict*) – A list of SpikeTrain objects.
- **events** (*dict*) – A dictionary of Event objects, indexed by segment. These events (in case of lists, always the first element in the list) will be used to align the spike trains and will be at time 0 for the aligned spike trains.
- **copy** (*bool*) – Determines if aligned copies of the original spike trains will be returned. If not, every spike train needs exactly one corresponding event, otherwise a `ValueError` will be raised. Otherwise, entries with no event will be ignored.

collapsed_spike_trains (*trains*)

Return a superposition of a list of spike trains.

Parameters **trains** (*iterable*) – A list of SpikeTrain objects

Returns A SpikeTrain object containing all spikes of the given SpikeTrain objects.

minimum_spike_train_interval (*trains*)

Computes the minimum starting time and maximum end time that all given spike trains share.

Parameters **trains** (*dict*) – A dictionary of sequences of SpikeTrain objects.

Returns Maximum shared start time and minimum shared stop time.

Return type Quantity scalar, Quantity scalar

optimal_gauss_kernel_size (*train*, *optimize_steps*, *progress=None*)

Return the optimal kernel size for a spike density estimation of a SpikeTrain for a gaussian kernel. This function takes a single spike train, which can be a superposition of multiple spike trains (created with `collapsed_spike_trains()`) that should be included in a spike density estimation. See (Shimazaki, Shinomoto. Journal of Computational Neuroscience. 2010).

Parameters

- **train** (*SpikeTrain*) – The spike train for which the kernel size should be optimized.
- **optimize_steps** (*Quantity 1D*) – Array of kernel sizes to try (the best of these sizes will be returned).
- **progress** (`spykeutils.progress_indicator.ProgressIndicator`) – Set this parameter to report progress. Will be advanced by len(*optimize_steps*) steps.

Returns Best of the given kernel sizes

Return type Quantity scalar

5.1.5 `sorting_quality_assesment` Module

Functions for estimating the quality of spike sorting results. These functions estimate false positive and false negative fractions.

`calculate_overlap_fp_fn`(*means*, *spikes*)

Return a dict of tuples (False positive rate, false negative rate) indexed by unit.

Details for the calculation can be found in (Hill et al. The Journal of Neuroscience. 2011). This function works on prewhitened data, which means it assumes that all clusters have a uniform normal distribution. Data can be prewhitened using the noise covariance matrix.

The calculation for total false positive and false negative rates does not follow (Hill et al. The Journal of Neuroscience. 2011), where a simple addition of pairwise probabilities is proposed. Instead, the total error probabilities are estimated using all clusters at once.

Parameters

- **means** (*dict*) – Dictionary of prewhitened cluster means (e.g. unit templates) indexed by unit as Spike objects or numpy arrays for all units.
- **spikes** (*dict*) – Dictionary, indexed by unit, of lists of prewhitened spike waveforms as Spike objects or numpy arrays for all units.

Returns

Two values:

- A dictionary (indexed by unit) of total (false positives, false negatives) tuples.
- A dictionary of dictionaries, both indexed by units, of pairwise (false positives, false negatives) tuples.

Return type dict, dict

`calculate_refperiod_fp`(*num_spikes*, *refperiod*, *violations*, *total_time*)

Return the rate of false positives calculated from refractory period calculations for each unit. The equation used is described in (Hill et al. The Journal of Neuroscience. 2011).

Parameters

- **num_spikes** (*dict*) – Dictionary of total number of spikes, indexed by unit.
- **refperiod** (*Quantity scalar*) – The refractory period (time). If the spike sorting algorithm includes a censored period (a time after a spike during which no new spikes can be found), subtract it from the refractory period before passing it to this function.
- **violations** (*dict*) – Dictionary of total number of violations, indexed the same as num_spikes.

- **total_time** (*Quantity scalar*) – The total time in which violations could have occured.

Returns A dictionary of false positive rates indexed by unit. Note that values above 0.5 can not be directly interpreted as a false positive rate! These very high values can e.g. indicate that the chosen refractory period was too large.

get_refperiod_violations(*spike_trains*, *refperiod*, *progress=None*)

Return the refractory period violations in the given spike trains for the specified refractory period.

Parameters

- **spike_trains** (*dict*) – Dictionary of lists of SpikeTrain objects.
- **refperiod** (*Quantity scalar*) – The refractory period (time).
- **progress** (`spykeutils.progress_indicator.ProgressIndicator`) – Set this parameter to report progress.

Returns

Two values:

- The total number of violations.
- A dictionary (with the same indices as `spike_trains`) of arrays with violation times (Quantity 1D with the same unit as `refperiod`) for each spike train.

Return type int, dict

5.1.6 `staionarity` Module

spike_amplitude_histogram(*trains*, *num_bins*, *uniform_y_scale=True*, *unit=uV*, *progress=None*)

Return a spike amplitude histogram.

The resulting is useful to assess the drift in spike amplitude over a longer recording. It shows histograms (one for each `trains` entry, e.g. segment) of maximum and minimum spike amplitudes.

Parameters

- **trains** (*list*) – A list of lists of SpikeTrain objects. Each entry of the outer list will be one point on the x-axis (they could correspond to segments), all amplitude occurences of spikes contained in the inner list will be added up.
- **num_bins** (*int*) – Number of bins for the histograms.
- **uniform_y_scale** (*bool*) – If True, the histogram for each channel will use the same bins. Otherwise, the minimum bin range is computed separately for each channel.
- **unit** (*Quantity*) – Unit of Y-Axis.
- **progress** (`spykeutils.progress_indicator.ProgressIndicator`) – Set this parameter to report progress.

Returns

A tuple with three values:

- A three-dimensional histogram matrix, where the first dimension corresponds to bins, the second dimension to the entries of `trains` (e.g. segments) and the third dimension to channels.
- A list of the minimum amplitude value for each channel (all values will be equal if `uniform_y_scale` is true).

- A list of the maximum amplitude value for each channel (all values will be equal if `uniform_y_scale` is true).

Return type (ndarray, list, list)

5.2 Subpackages

5.2.1 spykeutils.plot package

5.2.2 plugin Package

This package provides support for writing plugins for Spyke Viewer. It belongs to *spykeutils* so that plugins can be executed in an evironment where the *spykeviewer* package and its dependencies are not installed (e.g. servers).

`analysis_plugin` Module

class `AnalysisPlugin`

Bases: `spykeutils.plugin.gui_data.DataSet`

Base class for Analysis plugins. Inherit this class to create a plugin.

The two most important methods are `get_name()` and `start()`. Both should be overridden by every plugin. The class also has functionality for GUI configuration and saving/restoring analysis results.

The GUI configuration uses `guidata`. Because *AnalysisPlugin* inherits from *DataSet*, configuration options can easily be added directly to the class definition. For example, the following code creates an analysis that has two configuration options which are used in the start() method to print to the console:

```
from spykeutils.plugin.analysis_plugin import AnalysisPlugin

class ExampleAnalysis(AnalysisPlugin):
    some_time = di.FloatItem('Some time', default=2.0, unit='ms')
    print_more = di.BoolItem('Print additional info', default=True)

    def start(self, current, selections):
        print 'The selected time is', some_time, 'milliseconds.'
        if print_more:
            print 'This is important additional information!'
```

The class attribute `data_dir` contains a base directory for saving and loading data. It is set by Spyke Viewer to the directory specified in the settings. When using an AnalysisPlugin without Spyke Viewer, the default value is an empty string (so the current directory will be used) and the attribute can be set to an arbitrary directory.

`configure`()

Configure the analysis. Override if a different or additional configuration apart from guidata is needed.

`get_name`()

Return the name of an analysis. Override to specify analysis name.

Returns The name of the plugin.

Return type str

`get_parameters`()

Return a dictionary of the configuration that can be read with `deserialize_parameters()`. Override both if non-guidata attributes need to be serialized or if some guidata parameters should not be serialized (e.g. they only affect the visual presentation).

Returns A dictionary of all configuration parameters.

Return type dict

load(*name*, *selections*, *params=None*, *consider_guiparams=True*)
Return the most recent HDF5 file for a certain parameter configuration. If no such file exists, return None. This function works with the files created by `save()`.

Parameters

- **name** (*str*) – The name of the results to load.
- **selections** (*sequence*) – A list of `DataProvider` objects that are relevant for the analysis results.
- **params** (*dict*) – A dictionary, indexed by strings (which should be valid as python identifiers), with parameters apart from GUI configuration used to obtain the results. All keys have to be integers, floats, strings or lists of these types.
- **consider_guiparams** (*bool*) – Determines if the guidata parameters of the class should be considered if they exist in the HDF5 file. This should be set to False if `save()` is used with `save_guiparams` set to `False`.

Returns An open PyTables file object ready to be used to read data. Afterwards, the file has to be closed by calling the `tables.File.close()` method. If no appropriate file exists, None is returned.

Return type `tables.File`

save(*name*, *selections*, *params=None*, *save_guiparams=True*)
Return a HDF5 file object with parameters already stored. Save analysis results to this file.

Parameters

- **name** (*str*) – The name of the results to save. A folder with this name will be used (and created if necessary) to store the analysis result files.
- **selections** (*sequence*) – A list of `DataProvider` objects that are relevant for the analysis results.
- **params** (*dict*) – A dictionary, indexed by strings (which should be valid as python identifiers), with parameters apart from GUI configuration used to obtain the results. All keys have to be integers, floats, strings or lists of these types.
- **save_guiparams** (*bool*) – Determines if the guidata parameters of the class should be saved in the file.

Returns An open PyTables file object ready to be used to store data. Afterwards, the file has to be closed by calling the `tables.File.close()` method.

Return type `tables.File`

set_parameters(*parameters*)
Load configuration from a dictionary that has been created by `serialize_parameters()`. Override both if non-guidata attributes need to be serialized or if some guidata parameterss hould not be serialized (e.g. they only affect the visual presentation).

Parameters **parameters** (*dict*) – A dictionary of all configuration parameters.

start(*current*, *selections*)
Entry point for processing. Override with analysis code.

Parameters

- **current** (spykeviewer.plugin_framework.data_provider.DataProvider) – This data provider is used if the analysis should be performed on the data currently selected in the GUI.
- **selections** (*list*) – This parameter contains all saved selections. It is used if an analysis needs multiple data sets.

data_provider Module

class DataProvider(*name*, *progress*)

Bases: object

Defines all methods that should be implemented by a selection/data provider class.

A *DataProvider* encapsulates access to a selection of data. It can be used by plugins to acesss data currently selected in the GUI or in saved selections. It also contains an attribute *progress*, a spykeutils.progress_indicator.ProgressIndicator that can be used to report the progress of an operation (and is used by methods of this class if they can lead to processing times of half a second or more).

This class serves as an abstract base class and should not be instantiated.

analog_signal_arrays()

Return a list of AnalogSignalArray objects.

analog_signal_arrays_by_channelgroup()

Return a dictionary (indexed by RecordingChannelGroup) of lists of AnalogSignalArray objects.

If analog signals arrays not attached to a RecordingChannel are selected, their dictionary key will be DataProvider.no_channelgroup.

analog_signal_arrays_by_channelgroup_and_segment()

Return a dictionary (indexed by RecordingChannelGroup) of dictionaries (indexed by Segment) of AnalogSignalArray objects.

If there are multiple analog signals in one RecordingChannel for the same Segment, only the first will be contained in the returned dictionary. If analog signal arrays not attached to a Segment or RecordingChannelGroup are selected, their dictionary key will be DataProvider.no_segment or DataProvider.no_channelgroup, respectively.

analog_signal_arrays_by_segment()

Return a dictionary (indexed by Segment) of lists of AnalogSignalArray objects.

If analog signals arrays not attached to a Segment are selected, their dictionary key will be DataProvider.no_segment.

analog_signal_arrays_by_segment_and_channelgroup()

Return a dictionary (indexed by RecordingChannelGroup) of dictionaries (indexed by Segment) of AnalogSignalArray objects.

If there are multiple analog signals in one RecordingChannel for the same Segment, only the first will be contained in the returned dictionary. If analog signal arrays not attached to a Segment or RecordingChannelGroup are selected, their dictionary key will be DataProvider.no_segment or DataProvider.no_channelgroup, respectively.

analog_signals()

Return a list of AnalogSignal objects.

analog_signals_by_channel()

Return a dictionary (indexed by RecordingChannel) of lists of AnalogSignal objects.

If analog signals not attached to a RecordingChannel are selected, their dictionary key will be `DataProvider.no_channel`.

analog_signals_by_channel_and_segment()
Return a dictionary (indexed by RecordingChannel) of dictionaries (indexed by Segment) of AnalogSignal lists.

If analog signals not attached to a Segment or RecordingChannel are selected, their dictionary key will be `DataProvider.no_segment` or `DataProvider.no_channel`, respectively.

analog_signals_by_segment()
Return a dictionary (indexed by Segment) of lists of AnalogSignal objects.

If analog signals not attached to a Segment are selected, their dictionary key will be `DataProvider.no_segment`.

analog_signals_by_segment_and_channel()
Return a dictionary (indexed by Segment) of dictionaries (indexed by RecordingChannel) of AnalogSignal lists.

If analog signals not attached to a Segment or RecordingChannel are selected, their dictionary key will be `DataProvider.no_segment` or `DataProvider.no_channel`, respectively.

blocks()
Return a list of selected Block objects.

The returned objects will contain all regular references, not just to selected objects.

data_dict()
Return a dictionary with all information to serialize the object.

epoch_arrays()
Return a dictionary (indexed by Segment) of lists of EpochArray objects.

epochs(*include_array_epochs=True*)
Return a dictionary (indexed by Segment) of lists of Epoch objects.

Parameters **include_array_epochs** (*bool*) – Determines if EpochArray objects should be converted to Epoch objects and included in the returned list.

event_arrays()
Return a dictionary (indexed by Segment) of lists of EventArray objects.

events(*include_array_events=True*)
Return a dictionary (indexed by Segment) of lists of Event objects.

Parameters **include_array_events** (*bool*) – Determines if EventArray objects should be converted to Event objects and included in the returned list.

classmethod from_data(*data*, *progress=None*)
Create a new *DataProvider* object from a dictionary. This method is mostly for internal use.

The respective type of *DataProvider* (e.g. `spykeviewer.plugin_framework.data_provider_neo.DataProv` has to be imported in the environment where this function is called.

Parameters

- **data** (*dict*) – A dictionary containing data from a *DataProvider* object, as returned by `data_dict()`.
- **progress** (*ProgressIndicator*) – The object where loading progress will be indicated.

labeled_epochs(*label*, *include_array_epochs=True*)
Return a dictionary (indexed by Segment) of lists of Epoch objects with the given label.

Parameters

- **label** (*str*) – The name of the Epoch objects to be returnded
- **include_array_epochs** (*bool*) – Determines if EpochArray objects should be converted to Epoch objects and included in the returned list.

labeled_events(*label*, *include_array_events=True*)
Return a dictionary (indexed by Segment) of lists of Event objects with the given label.

Parameters

- **label** (*str*) – The name of the Event objects to be returnded
- **include_array_events** (*bool*) – Determines if EventArray objects should be converted to Event objects and included in the returned list.

num_analog_signal_arrays()
Return the number of AnalogSignalArray objects.

num_analog_signals()
Return the number of AnalogSignal objects.

recording_channel_groups()
Return a list of selected RecordingChannelGroup objects.

The returned objects will contain all regular references, not just to selected objects.

recording_channels()
Return a list of selected RecordingChannel objects.

The returned objects will contain all regular references, not just to selected objects.

segments()
Return a list of selected Segment objects.

The returned objects will contain all regular references, not just to selected objects.

selection_blocks()
Return a list of selected blocks.

The returned blocks will contain references to all other selected elements further down in the object hierarchy, but no references to elements which are not selected. The returned hierarchy is a copy, so changes made to it will not persist. The main purpose of this function is to provide an object hierarchy that can be saved to a neo file. It is not recommended to use it for data processing, the respective functions that return objects lower in the hierarchy are better suited for that purpose.

spike_trains()
Return a list of SpikeTrain objects.

spike_trains_by_segment()
Return a dictionary (indexed by Segment) of lists of SpikeTrain objects.

If spike trains not attached to a Segment are selected, their dictionary key will be `DataProvider.no_segment`.

spike_trains_by_segment_and_unit()
Return a dictionary (indexed by Unit) of dictionaries (indexed by Segment) of SpikeTrain objects.

If there are multiple spike trains in one Segment for the same Unit, only the first will be contained in the returned dictionary. If spike trains not attached to a Unit or Segment are selected, their dictionary key will be `DataProvider.no_unit` or `DataProvider.no_segment`, respectively.

spike_trains_by_unit()
Return a dictionary (indexed by Unit) of lists of SpikeTrain objects.

If spike trains not attached to a Unit are selected, their dicionary key will be `DataProvider.no_unit`.

spike_trains_by_unit_and_segment()
Return a dictionary (indexed by Unit) of dictionaries (indexed by Segment) of SpikeTrain objects.

If there are multiple spike trains in one Segment for the same Unit, only the first will be contained in the returned dictionary. If spike trains not attached to a Unit or Segment are selected, their dictionary key will be `DataProvider.no_unit` or `DataProvider.no_segment`, respectively.

spikes()
Return a list of Spike objects.

spikes_by_segment()
Return a dictionary (indexed by Segment) of lists of Spike objects.

If spikes not attached to a Segment are selected, their dictionary key will be `DataProvider.no_segment`.

spikes_by_segment_and_unit()
Return a dictionary (indexed by Segment) of dictionaries (indexed by Unit) of lists of Spike lists.

If spikes not attached to a Unit or Segment are selected, their dictionary key will be `DataProvider.no_unit` or `DataProvider.no_segment`, respectively.

spikes_by_unit()
Return a dictionary (indexed by Unit) of lists of Spike objects.

If spikes not attached to a Unit are selected, their dicionary key will be `DataProvider.no_unit`.

spikes_by_unit_and_segment()
Return a dictionary (indexed by Unit) of dictionaries (indexed by Segment) of Spike lists.

If there are multiple spikes in one Segment for the same Unit, only the first will be contained in the returned dictionary. If spikes not attached to a Unit or Segment are selected, their dictionary key will be `DataProvider.no_unit` or `DataProvider.no_segment`, respectively.

units()
Return a list of selected Unit objects.

The returned objects will contain all regular references, not just to selected objects.

`gui_data` Module

This module gives access to all members of `guidata.dataset.dataitems` and `guidata.dataset.datatypes`. If `guidata` cannot be imported, the module offers suitable dummy objects instead (e.g. for use on a server).

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

S