spykeutils Documentation

Release 0.4.1

Robert Pröpper

February 04, 2014

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.

A mailinglist for discussion and support is available at https://groups.google.com/d/forum/spyke-viewer

Contents:

CHAPTER 1

Requirements

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

- Python >= 2.7
- neo >= 0.2.1
- scipy
- guiqwt >= 2.1.4 (Optional, for plotting)
- tables (Optional, for analysis results data management. Also known as PyTables.)
- scikit-learn (Optional, for spike sorting quality analysis using Gaussian cluster overlap.)

Please see the respective websites for instructions on how to install them if they are not present on your computer. If you use Linux, you might not have access rights to your Python package installation directory, depending on your configuration. In this case, you will have to execute all shell commands in this section with administrator privileges, e.g. by using `sudo`.

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
```

Users of NeuroDebian or its repositories (available for Debian and Ubuntu) can also install spykeutils using the package manager instead of pip:

```
$ sudo apt-get install python-spykeutils
```

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

The master branch always contains the current stable version. If you want the latest development version, use the develop branch (selected by default). 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 3

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 4

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
>>> import spykeutils.spike_train_generation as stg
```

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

```
>>> segments = [neo.Segment('Trial 1'), neo.Segment('Trial 2')]
>>> units = []
>>> units.append(neo.Unit('Regular intervals'))
>>> units.append(neo.Unit('Homogeneous Poisson'))
>>> units.append(neo.Unit('Modulated Poisson'))
```

We create some spike trains from regular intervals, a homogeneous Poisson process and a modulated Poisson process:

```
>>> 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(stg.gen_homogeneous_poisson(5 * pq.Hz, t_stop=10 * pq.s))
>>> trains.append(stg.gen_homogeneous_poisson(7 * pq.Hz, t_stop=10 * pq.s))
>>> modulation = lambda t: sp.sin(3 * sp.pi * t / 10.0 / pq.s) / 2.0 + 0.5
>>> trains.append(stg.gen_inhomogeneous_poisson(modulation, 10 * pq.Hz, t_stop=10*pq.s))
>>> trains.append(stg.gen_inhomogeneous_poisson(modulation, 10 * pq.Hz, t_stop=10*pq.s))
```

Next, we create analog signals using the spike trains. First, we convolve all spike times with a mock spike waveform.

```
>>> spike = sp.sin(-sp.linspace(0, 2 * sp.pi, 16))
>>> binned_trains = (sp.histogram(trains[0], bins=160000, range=(0,10))[0] +
...                  sp.histogram(trains[2], bins=160000, range=(0,10))[0] +
...                  sp.histogram(trains[4], bins=160000, range=(0,10))[0])
```

```
>>> train_waves = [sp.convolve(binned_trains, spike)]
>>> binned_trains = (sp.histogram(trains[1], bins=160000, range=(0,10))[0] +
...                  sp.histogram(trains[3], bins=160000, range=(0,10))[0] +
...                  sp.histogram(trains[5], bins=160000, range=(0,10))[0])
>>> train_waves.append(sp.convolve(binned_trains, spike))
```

Now we add Gaussian noise and create four signals in each segment:

```
>>> for i in range(8):
...     sig = train_waves[i%2] + 0.2 * sp.randn(train_waves[i%2].shape[0])
...     signal = neo.AnalogSignal(sig * pq.uV, sampling_rate=16 * pq.kHz)
...     signal.segment = segments[i%2]
...     segments[i%2].analogsignals.append(signal)
```

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], trains[4]]
>>> segments[1].spiketrains = [trains[1], trains[3], trains[5]]
>>> units[0].spiketrains = trains[:2]
>>> units[1].spiketrains = trains[2:4]
>>> units[2].spiketrains = trains[4:6]
>>> 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 and ISI

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 dicionary of lists of spike trains. Since our spike trains were generated by three units, we will create three 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
>>> st_dict[units[2]] = units[2].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.

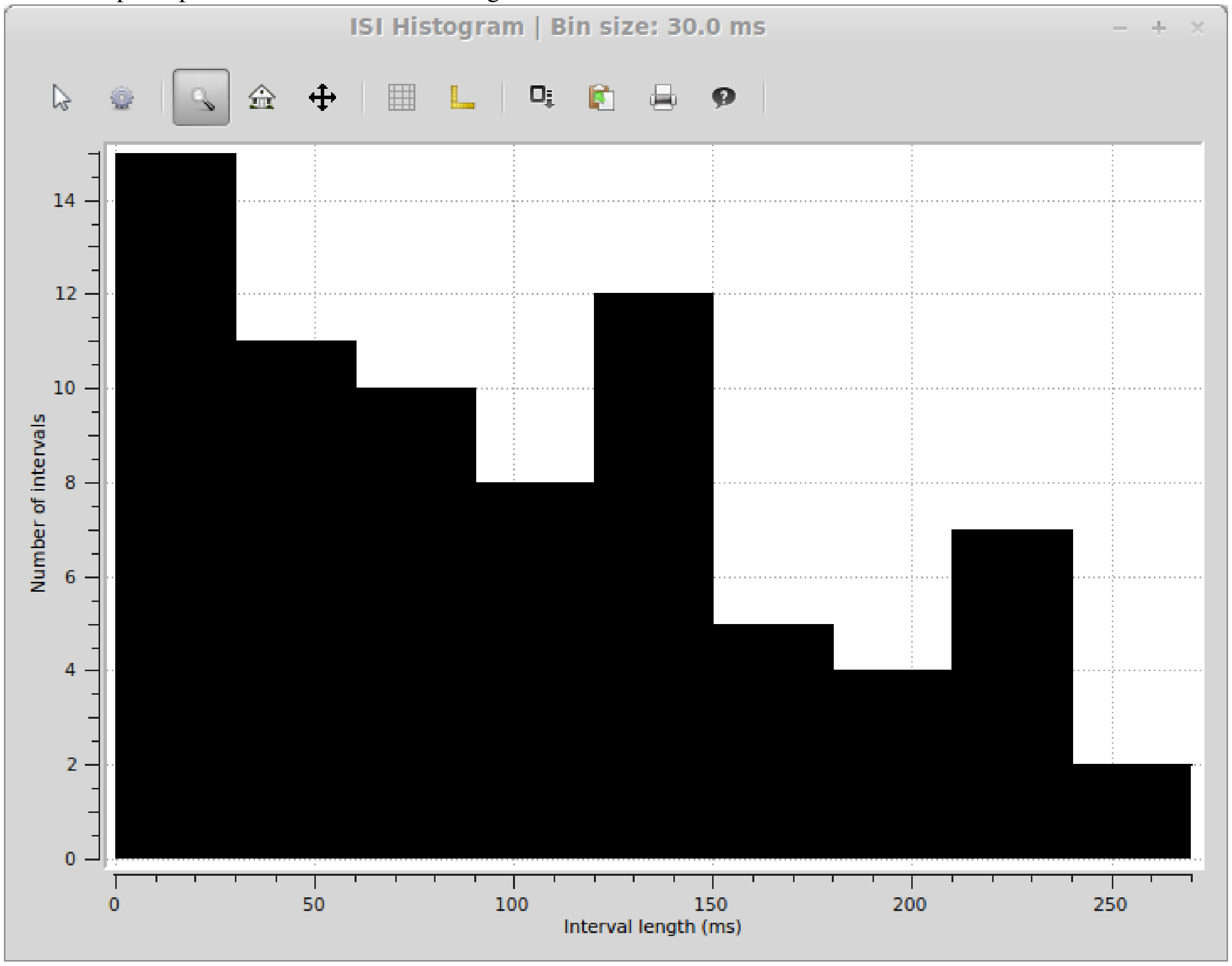
If `guiqwt` is installed, we can also use the `spykeutils.plot` package to create a PSTH plot from our data (in this case we want a bar histogram and therefore only use spike trains from one unit):

```
>>> import spykeutils.plot
>>> spykeutils.plot.psth({units[2]: units[2].spiketrains}, bin_size=400 * pq.ms, bar_plot=True)
```

Similiarily, we can create an interspike interval histogram plot with:

```
>>> spykeutils.plot.isi({units[2]: units[2].spiketrains}, bin_size=30 * pq.ms, cut_off=300 * pq.ms, k
```

This will open a plot window like the following:



4.3 Spike Density Estimation

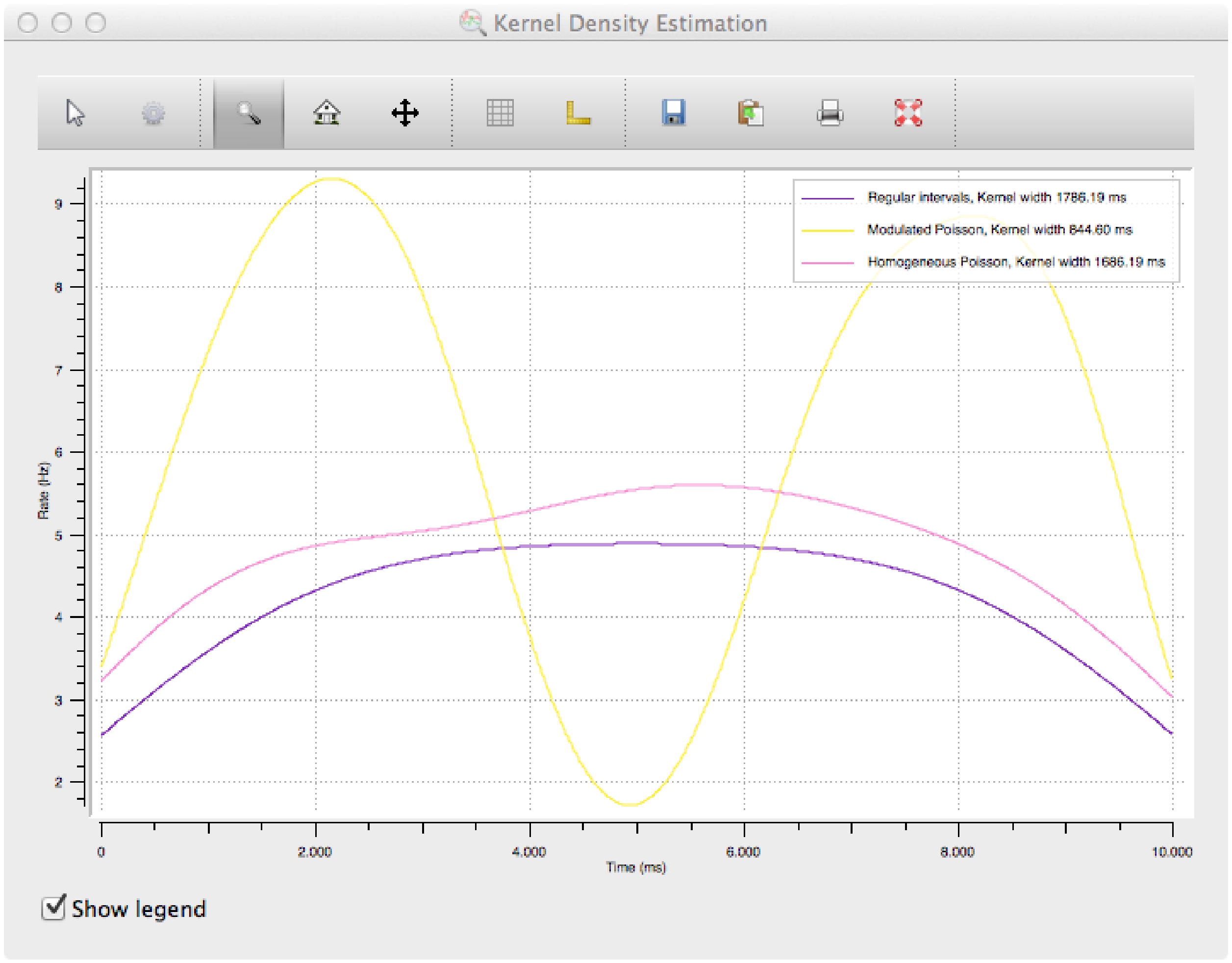
Similar to a PSTH, a spike density estimation gives an esimate of the instantaneous firing rate. Instead of binning, it is based on a kernel convolution which results in a smoother estimate. Creating and SDE with spykeutils works very similar to creating a PSTH. Instead of manually choosing the size 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([ ...
```

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=3000*pq.ms, optimize_steps=100)
```

The resulting plot will look like the following:

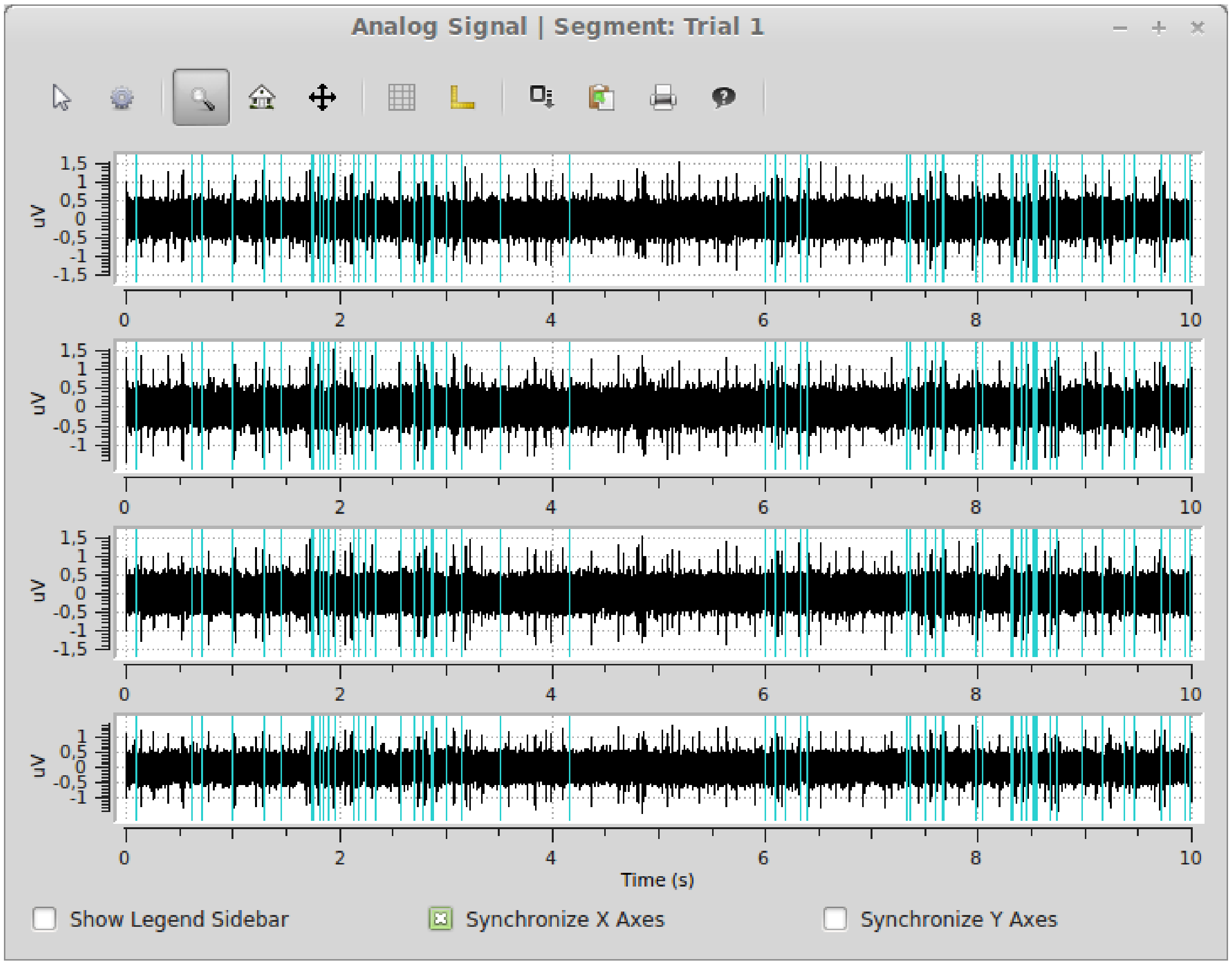


While spike density estimations are preferable to PSTHs in many cases, the picture also shows an important weakness: The estimation will generally be too low on margins. The areas where this happens become larger with kernel size, which is clearly visible from the rounded shape of the purple and pink curves (which should be flat because of the constant rate of the spike trains) with their 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 spike times from one of our spike trains.

```
>>> spykeutils.plot.signals(segments[0].analogsignals, spike_trains=[segments[0].spiketrains[2]], sho
```



The plot shows all four signals from the first segments as well as the spike times of the inhomogeneous poisson process in the same segment.

CHAPTER 5

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 `neo.core.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** (`neo.core.SpikeTrain`) – A spike train from which the `neo.core.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 `neo.core.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** (*sequence*) – A sequence of `neo.core.Spike` objects from which the spike train 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 All elements of `spikes` as spike train.

Return type `neo.core.SpikeTrain`

5.1.2 `correlations` Module

`correlogram`(*trains*, *bin_size*, *max_lag=array(500.0) * ms*, *border_correction=True*, *per_second=True*, *unit=UnitTime('millisecond', 0.001 * s, 'ms')*, *progress=None*)

Return (cross-)correlograms from a dictionary of spike train lists for different units.

Parameters

- **trains** (*dict*) – Dictionary of `neo.core.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.
- **per_second** (*bool*) – If `True`, counts returned are per second. Otherwise, counts per spike train are returned.
- **unit** (*Quantity*) – Unit of X-Axis.
- **progress** (`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

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** (*list*) – A list of `neo.core.SpikeTrain` objects.
- **events** (*dict*) – A dictionary of Event objects, indexed by segment. These events 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 `neo.core.SpikeTrain` objects

Returns A spike train object containing all spikes of the given spike trains.

Return type `neo.core.SpikeTrain`

optimal_gauss_kernel_size (*train*, *optimize_steps*, *progress=None*)
Return the optimal kernel size for a spike density estimation of a spike train 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.

Implements the algorithm from (Shimazaki, Shinomoto. Journal of Computational Neuroscience. 2010).

Parameters

- **train** (`neo.core.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** (`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

psth (*trains*, *bin_size*, *rate_correction=True*, *start=array(0.0) * ms*, *stop=array(inf) * s*)
Return dictionary of peri stimulus time histograms for a dictionary of spike train lists.

Parameters

- **trains** (*dict*) – A dictionary of lists of `neo.core.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=array(0.0) * ms*, *stop=None*, *kernel=None*, *kernel_size=array(100.0) * ms*, *optimize_steps=None*, *progress=None*)

Create a spike density estimation from a dictionary of lists of spike trains.

The spike density estimations give an estimate of the instantaneous rate. The density estimation is evaluated at 1024 equally spaced points covering the range of the input spike trains. Optionally finds optimal kernel size for given data using the algorithm from (Shimazaki, Shinomoto. Journal of Computational Neuroscience. 2010).

Parameters

- **trains** (*dict*) – A dictionary of `neo.core.SpikeTrain` lists.
- **start** (*Quantity scalar*) – The desired time for the start of the estimation. 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 estimation. It will be recalculated if there are spike trains which end earlier than this time.
- **kernel** (func or `signal_processing.Kernel`) – The kernel function or instance to use, should accept two parameters: A ndarray of distances and a kernel size. The total area under the kernel function should be 1. Automatic optimization assumes a Gaussian kernel and will likely not produce optimal results for different kernels. 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** (`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

5.1.5 `signal_processing` Module

class `CausalDecayingExpKernel`(*kernel_size=array(1.0) * s*, *normalize=True*)

Bases: `spykeutils.signal_processing.Kernel`

Unnormalized: $K(t) = \exp(-\frac{t}{\tau})\Theta(t)$ with $\Theta(t) = \left\{ \begin{array}{ll} 0, & x < 0 \\ 1, & x \geq 0 \end{array} \right.$ and kernel size τ.

Normalized to unit area: $K'(t) = \frac{1}{\tau}K(t)$

boundary_enclosing_at_least (*fraction*)

static evaluate (*t*, *kernel_size*)

normalization_factor (*kernel_size*)

class GaussianKernel (*kernel_size=array(1.0) * s*, *normalize=True*)

Bases: `spykeutils.signal_processing.SymmetricKernel`

Unnormalized: $K(t) = \exp(-\frac{t^2}{2\sigma^2})$ with kernel size σ (corresponds to the standard deviation of a Gaussian distribution).

Normalized to unit area: $K'(t) = \frac{1}{\sigma\sqrt{2\pi}}K(t)$

boundary_enclosing_at_least (*fraction*)

static evaluate (*t*, *kernel_size*)

normalization_factor (*kernel_size*)

class Kernel (*kernel_size*, *normalize*)

Bases: `object`

Base class for kernels.

boundary_enclosing_at_least (*fraction*)

Calculates the boundary b so that the integral from $-b$ to b encloses at least a certain fraction of the integral over the complete kernel.

Parameters **fraction** (*float*) – Fraction of the whole area which at least has to be enclosed.

Returns boundary

Return type Quantity scalar

is_symmetric ()

Should return *True* if the kernel is symmetric.

normalization_factor (*kernel_size*)

Returns the factor needed to normalize the kernel to unit area.

Parameters **kernel_size** (*Quantity scalar*) – Controls the width of the kernel.

Returns Factor to normalize the kernel to unit width.

Return type Quantity scalar

summed_dist_matrix (*vectors*, *presorted=False*)

Calculates the sum of all element pair distances for each pair of vectors.

If $(a_1, \dots, a_n)$ and $(b_1, \dots, b_m)$ are the u-th and v-th vector from *vectors* and K the kernel, the resulting entry in the 2D array will be $D_{uv} = \sum_{i=1}^{n} \sum_{j=1}^{m} K(a_i - b_j)$.

Parameters

- **vectors** (*sequence*) – A sequence of Quantity 1D to calculate the summed distances for each pair. The required units depend on the kernel. Usually it will be the inverse unit of the kernel size.
- **presorted** (*bool*) – Some optimized specializations of this function may need sorted vectors. Set *presorted* to *True* if you know that the passed vectors are already sorted to skip the sorting and thus increase performance.

Return type Quantity 2D

class **`KernelFromFunction`**(*kernel_func*, *kernel_size*)
Bases: `spykeutils.signal_processing.Kernel`

Creates a kernel form a function. Please note, that not all methods for such a kernel are implemented.

`is_symmetric`()

class **`LaplacianKernel`**(*kernel_size=array(1.0) * s*, *normalize=True*)
Bases: `spykeutils.signal_processing.SymmetricKernel`

Unnormalized: $K(t) = \exp(-|\frac{t}{\tau}|)$ with kernel size τ.

Normalized to unit area: $K'(t) = \frac{1}{2\tau}K(t)$

`boundary_enclosing_at_least`(*fraction*)

static **`evaluate`**(*t*, *kernel_size*)

`normalization_factor`(*kernel_size*)

`summed_dist_matrix`(*vectors*, *presorted=False*)

class **`RectangularKernel`**(*half_width=array(1.0) * s*, *normalize=True*)
Bases: `spykeutils.signal_processing.SymmetricKernel`

Unnormalized: $K(t) = \left\{ \begin{array}{ll} 1, & |t| < \tau \\ 0, & |t| \geq \tau \end{array} \right.$ with kernel size τ corresponding to the half width.

Normalized to unit area: $K'(t) = \frac{1}{2\tau}K(t)$

`boundary_enclosing_at_least`(*fraction*)

static **`evaluate`**(*t*, *half_width*)

`normalization_factor`(*half_width*)

class **`SymmetricKernel`**(*kernel_size*, *normalize*)
Bases: `spykeutils.signal_processing.Kernel`

Base class for symmetric kernels.

`is_symmetric`()

`summed_dist_matrix`(*vectors*, *presorted=False*)

class **`TriangularKernel`**(*half_width=array(1.0) * s*, *normalize=True*)
Bases: `spykeutils.signal_processing.SymmetricKernel`

Unnormalized: $K(t) = \left\{ \begin{array}{ll} 1 - \frac{|t|}{\tau}, & |t| < \tau \\ 0, & |t| \geq \tau \end{array} \right.$ with kernel size τ corresponding to the half width.

Normalized to unit area: $K'(t) = \frac{1}{\tau}K(t)$

`boundary_enclosing_at_least`(*fraction*)

static **`evaluate`**(*t*, *half_width*)

`normalization_factor`(*half_width*)

`as_kernel_of_size`(*obj*, *kernel_size*)
Returns a kernel of desired size.

Parameters

- **obj** (*Kernel or func*) – Either an existing kernel or a kernel function. A kernel function takes two arguments. First a *Quantity 1D* of evaluation time points and second a kernel size.

- **kernel_size** (*Quantity 1D*) – Desired size of the kernel.

Returns A `Kernel` with the desired kernel size. If *obj* is already a `Kernel` instance, a shallow copy of this instance with changed kernel size will be returned. If *obj* is a function it will be wrapped in a `Kernel` instance.

Return type `Kernel`

`discretize_kernel`(*kernel*, *sampling_rate*, *area_fraction=0.99999*, *num_bins=None*, *ensure_unit_area=False*)

Discretizes a kernel.

Parameters

- **kernel** (`Kernel` or function) – The kernel or kernel function. If a kernel function is used it should take exactly one 1-D array as argument.
- **area_fraction** (*float*) – Fraction between 0 and 1 (exclusive) of the integral of the kernel which will be at least covered by the discretization. Will be ignored if *num_bins* is not *None*. If *area_fraction* is used, the kernel has to provide a method `boundary_enclosing_at_least()` (see `Kernel.boundary_enclosing_at_least()`).
- **sampling_rate** (*Quantity scalar*) – Sampling rate for the discretization. The unit will typically be a frequency unit.
- **num_bins** (*int*) – Number of bins to use for the discretization.
- **ensure_unit_area** (*bool*) – If *True*, the area of the discretized kernel will be normalized to 1.0.

Return type Quantity 1D

`smooth`(*binned*, *kernel*, *sampling_rate*, *mode='same'*, ***kernel_discretization_params*)

Smoothes a binned representation (e.g. of a spike train) by convolving with a kernel.

Parameters

- **binned** (*1-D array*) – Bin array to smooth.
- **kernel** (`Kernel`) – The kernel instance to convolve with.
- **sampling_rate** (*Quantity scalar*) – The sampling rate which will be used to discretize the kernel. It should be equal to the sampling rate used to obtain *binned*. The unit will typically be a frequency unit.
- **mode** (*{'same', 'full', 'valid'}*) –
  - 'same': The default which returns an array of the same size as *binned*
  - 'full': Returns an array with a bin for each shift where *binned* and the discretized kernel overlap by at least one bin.
  - 'valid': Returns only the discretization bins where the discretized kernel and *binned* completely overlap.

  See also numpy.convolve.
- **kernel_discretization_params** (*dict*) – Additional discretization arguments which will be passed to `discretize_kernel()`.

Returns The smoothed representation of *binned*.

Return type Quantity 1D

st_convolve(*train*, *kernel*, *sampling_rate*, *mode='same'*, *binning_params={}*, *kernel_discretization_params={}*)

Convolves a `neo.core.SpikeTrain` with a kernel.

Parameters

- **train** (`neo.core.SpikeTrain`) – Spike train to convolve.
- **kernel** (`Kernel`) – The kernel instance to convolve with.
- **sampling_rate** (*Quantity scalar*) – The sampling rate which will be used to bin the spike train. The unit will typically be a frequency unit.
- **mode** (*{'same', 'full', 'valid'}*) –
  - 'same': The default which returns an array covering the whole duration of the spike train *train*.
  - 'full': Returns an array with additional discretization bins in the beginning and end so that for each spike the whole discretized kernel is included.
  - 'valid': Returns only the discretization bins where the discretized kernel and spike train completely overlap.

  See also `scipy.signal.convolve()`.
- **binning_params** (*dict*) – Additional discretization arguments which will be passed to `tools.bin_spike_trains()`.
- **kernel_discretization_params** (*dict*) – Additional discretization arguments which will be passed to `discretize_kernel()`.

Returns The convolved spike train, the boundaries of the discretization bins

Return type (Quantity 1D, Quantity 1D with the inverse units of *sampling_rate*)

5.1.6 `spike_train_generation` Module

gen_homogeneous_poisson(*rate*, *t_start=array(0.0) * s*, *t_stop=None*, *max_spikes=None*, *refractory=array(0.0) * s*)

Generate a homogeneous Poisson spike train. The length is controlled with *t_stop* and *max_spikes*. Either one or both of these arguments have to be given.

Parameters

- **rate** (*Quantity scalar*) – Average firing rate of the spike train to generate as frequency scalar.
- **t_start** (*Quantity scalar*) – Time at which the spike train begins as time scalar. The first actual spike will be greater than this time.
- **t_stop** (*Quantity scalar*) – Time at which the spike train ends as time scalar. All generated spikes will be lower or equal than this time. If set to None, the number of generated spikes is controlled by *max_spikes* and *t_stop* will be equal to the last generated spike.
- **max_spikes** – Maximum number of spikes to generate. Fewer spikes might be generated in case *t_stop* is also set.
- **refractory** (*Quantity scalar*) – Absolute refractory period as time scalar. No spike will follow another spike for the given duration. Afterwards the firing rate will instantaneously be set to *rate* again.

Returns The generated spike train.

Return type `neo.core.SpikeTrain`

gen_inhomogeneous_poisson (*modulation*, *max_rate*, *t_start=array(0.0) * s*, *t_stop=None*, *max_spikes=None*, *refractory=array(0.0) * s*)

Generate an inhomogeneous Poisson spike train. The length is controlled with *t_stop* and *max_spikes*. Either one or both of these arguments have to be given.

Parameters

- **modulation** (*function*) – Function $f((t_1, \dots, t_n)) : [\text{t\_start}, \text{t\_end}]^n \to [0, 1]^n$ giving the instantaneous firing rates at times $(t_1, \dots, t_n)$ as proportion of *max_rate*. Thus, a 1-D array will be passed to the function and it should return an array of the same size.
- **max_rate** (*Quantity scalar*) – Maximum firing rate of the spike train to generate as frequency scalar.
- **t_start** (*Quantity scalar*) – Time at which the spike train begins as time scalar. The first actual spike will be greater than this time.
- **t_stop** (*Quantity scalar*) – Time at which the spike train ends as time scalar. All generated spikes will be lower or equal than this time. If set to None, the number of generated spikes is controlled by *max_spikes* and *t_stop* will be equal to the last generated spike.
- **refractory** (*Quantity scalar*) – Absolute refractory period as time scalar. No spike will follow another spike for the given duration. Afterwards the firing rate will instantaneously be set to *rate* again.

Returns The generated spike train.

Return type `neo.core.SpikeTrain`

5.1.7 `spike_train_metrics` Module

cs_dist (*trains*, *smoothing_filter*, *sampling_rate*, *filter_area_fraction=0.99999*)

Calculates the Cauchy-Schwarz distance between two spike trains given a smoothing filter.

Let $v_a(t)$ and $v_b(t)$ with $t \in \mathcal{T}$ be the spike trains convolved with some smoothing filter and $V(a, b) = \int_{\mathcal{T}} v_a(t) v_b(t) dt$. Then, the Cauchy-Schwarz distance of the spike trains is defined as $d_{CS}(a, b) = \arccos \frac{V(a,b)^2}{V(a,a)V(b,b)}$.

The Cauchy-Schwarz distance is closely related to the Schreiber et al. similarity measure S_S by $d_{CS} = \arccos S_S^2$

This function numerically convolves the spike trains with the smoothing filter which can be quite slow and inaccurate. If the analytical result of the autocorrelation of the smoothing filter is known, one can use `schreiber_similarity()` for a more efficient and precise calculation.

Further information can be found in *Paiva, A. R. C., Park, I., & Principe, J. (2010). Inner products for representation and learning in the spike train domain. Statistical Signal Processing for Neuroscience and Neurotechnology, Academic Press, New York.*

Parameters

- **trains** (*sequence*) – Sequence of `neo.core.SpikeTrain` objects of which the distance will be calculated pairwise.
- **smoothing_filter** (`signal_processing.Kernel`) – Smoothing filter to be convolved with the spike trains.
- **sampling_rate** (*Quantity scalar*) – The sampling rate which will be used to bin the spike trains as inverse time scalar.

- **filter_area_fraction** (*float*) – A value between 0 and 1 which controls the interval over which the smoothing filter will be discretized. At least the given fraction of the complete smoothing filter area will be covered. Higher values can lead to more accurate results (besides the sampling rate).

Returns Matrix containing the Cauchy-Schwarz distance of all pairs of spike trains

Return type 2-D array

`event_synchronization` (*trains*, *tau=None*, *kernel=signal_processing.RectangularKernel(1.0*, *normalize=False)*, *sort=True*)

Calculates the event synchronization.

Let $d(x|y)$ be the count of spikes in y which occur shortly before an event in x with a time difference of less than τ. Moreover, let n_x and n_y be the number of total spikes in the spike trains x and y. The event synchrony is then defined as $Q_T = \frac{d(x|y)+d(y|x)}{\sqrt{n_x n_y}}$.

The time maximum time lag τ can be determined automatically for each pair of spikes t^x_i and t^y_j by the formula $\tau_{ij} = \frac{1}{2} \min\{t^x_{i+1} - t^x_i, t^x_i - t^x_{i-1}, t^y_{j+1} - t^y_j, t^y_j - t^y_{j-1}\}$

Further and more detailed information can be found in *Quiroga, R. Q., Kreuz, T., & Grassberger, P. (2002). Event synchronization: a simple and fast method to measure synchronicity and time delay patterns. Physical Review E, 66(4), 041904.*

Parameters

- **trains** (*sequence*) – Sequence of `neo.core.SpikeTrain` objects of which the van Rossum distance will be calculated pairwise.
- **tau** (*Quantity scalar*) – The maximum time lag for two spikes to be considered coincident or synchronous as time scalar. To have it determined automatically by above formula set it to *None*.
- **kernel** (`signal_processing.Kernel`) – Kernel to use in the calculation of the distance.
- **sort** (*bool*) – Spike trains with sorted spike times are be needed for the calculation. You can set *sort* to *False* if you know that your spike trains are already sorted to decrease calculation time.

Returns Matrix containing the event synchronization for all pairs of spike trains.

Return type 2-D array

`hunter_milton_similarity` (*trains*, *tau=array(1.0) * s*, *kernel=None*)

Calculates the Hunter-Milton similarity measure.

If the kernel function is denoted as $K(t)$, a function $d(x_k) = K(x_k - y_{k'})$ can be defined with $y_{k'}$ being the closest spike in spike train y to the spike x_k in spike train x. With this the Hunter-Milton similarity measure is $S_H = \frac{1}{2}\left(\frac{1}{n_x}\sum_{k=1}^{n_x} d(x_k) + \frac{1}{n_y}\sum_{k'=1}^{n_y} d(y_{k'})\right)$.

This implementation returns 0 if one of the spike trains is empty, but 1 if both are empty.

Further information can be found in

- *Hunter, J. D., & Milton, J. G. (2003). Amplitude and Frequency Dependence of Spike Timing: Implications for Dynamic Regulation. Journal of Neurophysiology.*
- *Dauwels, J., Vialatte, F., Weber, T., & Cichocki, A. (2009). On similarity measures for spike trains. Advances in Neuro-Information Processing, 177-185.*

Parameters

- **trains** (*sequence*) – Sequence of `neo.core.SpikeTrain` objects of which the Hunter-Milton similarity will be calculated pairwise.
- **tau** (*Quantity scalar*) – The time scale for determining the coincidence of two events as time scalar.
- **kernel** (`signal_processing.Kernel`) – Kernel to use in the calculation of the distance. If *None*, a unnormalized Laplacian kernel will be used.

Returns Matrix containing the Hunter-Milton similarity for all pairs of spike trains.

Return type 2-D array

`norm_dist` (*trains*, *smoothing_filter*, *sampling_rate*, *filter_area_fraction=0.99999*)

Calculates the norm distance between spike trains given a smoothing filter.

Let $v_a(t)$ and $v_b(t)$ with $t \in \mathcal{T}$ be the spike trains convolved with some smoothing filter. Then, the norm distance of the spike trains is defined as $d_{ND}(a, b) = \sqrt{\int_{\mathcal{T}} (v_a(t) - v_b(t))^2 dt}$.

Further information can be found in *Paiva, A. R. C., Park, I., & Principe, J. (2010). Inner products for representation and learning in the spike train domain. Statistical Signal Processing for Neuroscience and Neurotechnology, Academic Press, New York.*

Parameters

- **trains** (*sequence*) – Sequence of `neo.core.SpikeTrain` objects of which the distance will be calculated pairwise.
- **smoothing_filter** (`signal_processing.Kernel`) – Smoothing filter to be convolved with the spike trains.
- **sampling_rate** (*Quantity scalar*) – The sampling rate which will be used to bin the spike trains as inverse time scalar.
- **filter_area_fraction** (*float*) – A value between 0 and 1 which controls the interval over which the smoothing filter will be discretized. At least the given fraction of the complete smoothing filter area will be covered. Higher values can lead to more accurate results (besides the sampling rate).

Returns Matrix containing the norm distance of all pairs of spike trains given the smoothing_filter.

Return type Quantity 2D with units depending on the smoothing filter (usually temporal frequency units)

`schreiber_similarity` (*trains*, *kernel*, *sort=True*)

Calculates the Schreiber et al. similarity measure between spike trains given a kernel.

Let $v_a(t)$ and $v_b(t)$ with $t \in \mathcal{T}$ be the spike trains convolved with some smoothing filter and $V(a, b) = \int_{\mathcal{T}} v_a(t) v_b(t) dt$. The autocorrelation of the smoothing filter corresponds to the kernel used to analytically calculate the Schreiber et al. similarity measure. It is defined as $S_S(a, b) = \frac{V(a,b)}{\sqrt{V(a,a)V(b,b)}}$. It is closely related to the Cauchy-Schwarz distance d_{CS} by $S_S = \sqrt{\cos d_{CS}}$.

In opposite to `cs_dist()` which numerically convolves the spike trains with a smoothing filter, this function directly uses the kernel resulting from the smoothing filter's autocorrelation. This allows a more accurate and faster calculation.

Further information can be found in:

- *Dauwels, J., Vialatte, F., Weber, T., & Cichocki, A. (2009). On similarity measures for spike trains. Advances in Neuro-Information Processing, 177-185.*

- *Paiva, A. R. C., Park, I., & Principe, J. C. (2009). A comparison of binless spike train measures. Neural Computing and Applications, 19(3), 405-419. doi:10.1007/s00521-009-0307-6*

Parameters

- **trains** (*sequence*) – Sequence of `neo.core.SpikeTrain` objects of which the distance will be calculated pairwise.
- **kernel** (`signal_processing.Kernel`) – Kernel to use. It corresponds to a smoothing filter by being the autocorrelation of such a filter.
- **sort** (*bool*) – Spike trains with sorted spike times will be needed for the calculation. You can set *sort* to *False* if you know that your spike trains are already sorted to decrease calculation time.

Returns Matrix containing the Schreiber et al. similarity measure of all pairs of spike trains.

Return type 2-D array

`st_inner`(*a*, *b*, *smoothing_filter*, *sampling_rate*, *filter_area_fraction=0.99999*)

Calculates the inner product of spike trains given a smoothing filter.

Let $v_a(t)$ and $v_b(t)$ with $t \in \mathcal{T}$ be the spike trains convolved with some smoothing filter. Then, the inner product of the spike trains is defined as $\int_\mathcal{T} v_a(t)v_b(t)dt$.

Further information can be found in *Paiva, A. R. C., Park, I., & Principe, J. (2010). Inner products for representation and learning in the spike train domain. Statistical Signal Processing for Neuroscience and Neurotechnology, Academic Press, New York.*

Parameters

- **a** (*sequence*) – Sequence of `neo.core.SpikeTrain` objects.
- **b** (*sequence*) – Sequence of `neo.core.SpikeTrain` objects.
- **smoothing_filter** (`signal_processing.Kernel`) – A smoothing filter to be convolved with the spike trains.
- **sampling_rate** (*Quantity scalar*) – The sampling rate which will be used to bin the spike train as inverse time scalar.
- **filter_area_fraction** (*float*) – A value between 0 and 1 which controls the interval over which the *smoothing_filter* will be discretized. At least the given fraction of the complete *smoothing_filter* area will be covered. Higher values can lead to more accurate results (besides the sampling rate).

Returns Matrix containing the inner product for each pair of spike trains with one spike train from *a* and the other one from *b*.

Return type Quantity 2D with units depending on the smoothing filter (usually temporal frequency units)

`st_norm`(*train*, *smoothing_filter*, *sampling_rate*, *filter_area_fraction=0.99999*)

Calculates the spike train norm given a smoothing filter.

Let $v(t)$ with $t \in \mathcal{T}$ be a spike train convolved with some smoothing filter. Then, the norm of the spike train is defined as $\int_\mathcal{T} v(t)^2 dt$.

Further information can be found in *Paiva, A. R. C., Park, I., & Principe, J. (2010). Inner products for representation and learning in the spike train domain. Statistical Signal Processing for Neuroscience and Neurotechnology, Academic Press, New York.*

Parameters

- **train** (`neo.core.SpikeTrain`) – Spike train of which to calculate the norm.
- **smoothing_filter** (`signal_processing.Kernel`) – Smoothing filter to be convolved with the spike train.
- **sampling_rate** (*Quantity scalar*) – The sampling rate which will be used to bin the spike train as inverse time scalar.
- **filter_area_fraction** (*float*) – A value between 0 and 1 which controls the interval over which the smoothing filter will be discretized. At least the given fraction of the complete smoothing filter area will be covered. Higher values can lead to more accurate results (besides the sampling rate).

Returns The norm of the spike train given the smoothing_filter.

Return type Quantity scalar with units depending on the smoothing filter (usually temporal frequency units)

`van_rossum_dist` (*trains*, *tau=array(1.0) * s*, *kernel=None*, *sort=True*)

Calculates the van Rossum distance.

It is defined as Euclidean distance of the spike trains convolved with a causal decaying exponential smoothing filter. A detailed description can be found in *Rossum, M. C. W. (2001). A novel spike distance. Neural Computation, 13(4), 751-763.* This implementation is normalized to yield a distance of 1.0 for the distance between an empty spike train and a spike train with a single spike. Divide the result by sqrt(2.0) to get the normalization used in the cited paper.

Given N spike trains with n spikes on average the run-time complexity of this function is $O(N^2n^2)$. An implementation in $O(N^2n)$ would be possible but has a high constant factor rendering it slower in practical cases.

Parameters

- **trains** (*sequence*) – Sequence of `neo.core.SpikeTrain` objects of which the van Rossum distance will be calculated pairwise.
- **tau** (*Quantity scalar*) – Decay rate of the exponential function as time scalar. Controls for which time scale the metric will be sensitive. This parameter will be ignored if *kernel* is not *None*. May also be `scipy.inf` which will lead to only measuring differences in spike count.
- **kernel** (`signal_processing.Kernel`) – Kernel to use in the calculation of the distance. This is not the smoothing filter, but its autocorrelation. If *kernel* is *None*, an unnormalized Laplacian kernel with a size of *tau* will be used.
- **sort** (*bool*) – Spike trains with sorted spike times might be needed for the calculation. You can set *sort* to *False* if you know that your spike trains are already sorted to decrease calculation time.

Returns Matrix containing the van Rossum distances for all pairs of spike trains.

Return type 2-D array

`van_rossum_multiunit_dist` (*units*, *weighting*, *tau=array(1.0) * s*, *kernel=None*)

Calculates the van Rossum multi-unit distance.

The single-unit distance is defined as Euclidean distance of the spike trains convolved with a causal decaying exponential smoothing filter. A detailed description can be found in *Rossum, M. C. W. (2001). A novel spike distance. Neural Computation, 13(4), 751-763.* This implementation is normalized to yield a distance of 1.0 for the distance between an empty spike train and a spike train with a single spike. Divide the result by sqrt(2.0) to get the normalization used in the cited paper.

Given the p- and q-th spike train of *a* and respectively *b* let R_{pq} be the squared single-unit distance between these two spike trains. Then the multi-unit distance is $\sqrt{\sum_p (R_{pp} + c \cdot \sum_{q \neq p} R_{pq})}$ with c being equal to *weighting*. The weighting parameter controls the interpolation between a labeled line and a summed population coding.

More information can be found in *Houghton, C., & Kreuz, T. (2012). On the efficient calculation of van Rossum distances. Network: Computation in Neural Systems, 23(1-2), 48-58.*

Given N spike trains in total with n spikes on average the run-time complexity of this function is $O(N^2 n^2)$ and $O(N^2 + Nn^2)$ memory will be needed.

Parameters

- **units** (*dict*) – Dictionary of sequences with each sequence containing the trials of one unit. Each trial should be a `neo.core.SpikeTrain` and all units should have the same number of trials.
- **weighting** (*float*) – Controls the interpolation between a labeled line and a summed population coding.
- **tau** (*Quantity scalar*) – Decay rate of the exponential function as time scalar. Controls for which time scale the metric will be sensitive. This parameter will be ignored if *kernel* is not *None*. May also be `scipy.inf` which will lead to only measuring differences in spike count.
- **kernel** (`signal_processing.Kernel`) – Kernel to use in the calculation of the distance. This is not the smoothing filter, but its autocorrelation. If *kernel* is *None*, an unnormalized Laplacian kernel with a size of *tau* will be used.

Returns A 2D array with the multi-unit distance for each pair of trials.

Return type 2D array

`victor_purpura_dist`(*trains*, *q=array(1.0) * Hz*, *kernel=None*, *sort=True*)

Calculates the Victor-Purpura's (VP) distance. It is often denoted as $D^{\text{spike}}[q]$.

It is defined as the minimal cost of transforming spike train *a* into spike train *b* by using the following operations:

- Inserting or deleting a spike (cost 1.0).
- Shifting a spike from t to t' (cost $q \cdot |t - t'|$).

A detailed description can be found in *Victor, J. D., & Purpura, K. P. (1996). Nature and precision of temporal coding in visual cortex: a metric-space analysis. Journal of Neurophysiology.*

Given the average number of spikes n in a spike train and N spike trains the run-time complexity of this function is $O(N^2 n^2)$ and $O(N^2 + n^2)$ memory will be needed.

Parameters

- **trains** (*sequence*) – Sequence of `neo.core.SpikeTrain` objects of which the distance will be calculated pairwise.
- **q** (*Quantity scalar*) – Cost factor for spike shifts as inverse time scalar. If *kernel* is not *None*, *q* will be ignored.
- **kernel** (`signal_processing.Kernel`) – Kernel to use in the calculation of the distance. If *kernel* is *None*, an unnormalized triangular kernel with a half width of *2.0/q* will be used.
- **sort** (*bool*) – Spike trains with sorted spike times will be needed for the calculation. You can set *sort* to *False* if you know that your spike trains are already sorted to decrease calculation time.

Returns Matrix containing the VP distance of all pairs of spike trains.

Return type 2-D array

victor_purpura_multiunit_dist(*units*, *reassignment_cost*, *q=array(1.0) * Hz*, *kernel=None*)

Calculates the Victor-Purpura's (VP) multi-unit distance.

It is defined as the minimal cost of transforming the spike trains a into spike trains b by using the following operations:

- Inserting or deleting a spike (cost 1.0).
- Shifting a spike from t to t' (cost $q \cdot |t - t'|$).
- Moving a spike to another spike train (cost *reassignment_cost*).

A detailed description can be found in *Aronov, D. (2003). Fast algorithm for the metric-space analysis of simultaneous responses of multiple single neurons. Journal of Neuroscience Methods.*

Given the average number of spikes N in a spike train and L units with n spike trains each the run-time complexity is $O(n^2 LN^{L+1})$. The space complexity is $O(n^2 + LN^{L+1})$.

For calculating the distance between only two units one should use `victor_purpura_dist()` which is more efficient.

Parameters

- **units** (*dict*) – Dictionary of sequences with each sequence containing the trials of one unit. Each trial should be a `neo.core.SpikeTrain` and all units should have the same number of trials.
- **reassignment_cost** (*float*) – Cost to reassign a spike from one train to another (sometimes denoted with k). Should be between 0 and 2. For 0 spikes can be reassigned without any cost, for 2 and above it is cheaper to delete and reinsert a spike.
- **q** (*Quantity scalar*) – Cost factor for spike shifts as inverse time scalar. If *kernel* is not *None*, *q* will be ignored.
- **kernel** (`signal_processing.Kernel`) – Kernel to use in the calculation of the distance. If *kernel* is *None*, an unnormalized triangular kernel with a half width of *2.0/q* will be used.

Returns A 2D array with the multi-unit distance for each pair of trials.

Return type 2D array

5.1.8 `sorting_quality_assesment` Module

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

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 generating processes are not independent.

`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 `neo.core.SpikeTrain` objects.
- **refperiod** (*Quantity scalar*) – The refractory period (time).
- **progress** (`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

`overlap_fp_fn` (*spikes*, *means=None*, *covariances=None*)

Return dicts of tuples (False positive rate, false negative rate) indexed by unit. This function needs `sklearn` if `covariances` is not set to `'white'`.

This function estimates the pairwise and total false positive and false negative rates for a number of waveform clusters. The results can be interpreted as follows: False positives are the fraction of spikes in a cluster that is estimated to belong to a different cluster (a specific cluster for pairwise results or any other cluster for total results). False negatives are the number spikes from other clusters that are estimated to belong to a given cluster (also expressed as fraction, this number can be larger than 1 in extreme cases).

Details for the calculation can be found in (Hill et al. The Journal of Neuroscience. 2011). The calculation for total false positive and false negative rates does not follow Hill et al., who propose a simple addition of pairwise probabilities. Instead, the total error probabilities are estimated using all clusters at once.

Parameters

- **spikes** (*dict*) – Dictionary, indexed by unit, of lists of spike waveforms as `neo.core.Spike` objects or numpy arrays. If the waveforms have multiple channels, they will be flattened automatically. All waveforms need to have the same number of samples.
- **means** (*dict*) – Dictionary, indexed by unit, of lists of spike waveforms as `neo.core.Spike` objects or numpy arrays. Means for units that are not in this dictionary will be estimated using the spikes. Note that if you pass `'white'` for `covariances` and you want to provide means, they have to be whitened in the same way as the spikes. Default: None, means will be estimated from data.
- **covariances** (*dict or str*) – Dictionary, indexed by unit, of lists of covariance matrices. Covariances for units that are not in this dictionary will be estimated using the spikes. It is useful to give a covariance matrix if few spikes are present - consider using the noise covariance. If you use prewhitened spikes (i.e. all clusters are normal distributed, so their covariance matrix is the identity), you can pass `'white'` here. The calculation will be much faster in this case and the sklearn package is not required. Default: None, covariances will estimated from data.

Returns

Two values:

- A dictionary (indexed by unit) of total (false positive rate, false negative rate) tuples.
- A dictionary of dictionaries, both indexed by units, of pairwise (false positive rate, false negative rate) tuples.

Return type dict, dict

`variance_explained`(*spikes*, *means=None*, *noise=None*)

Returns the fraction of variance in each channel that is explained by the means.

Values below 0 or above 1 for large data sizes indicate that some assumptions were incorrect (e.g. about channel noise) and the results should not be trusted.

Parameters

- **spikes** (*dict*) – Dictionary, indexed by unit, of `neo.core.SpikeTrain` objects (where the `waveforms` member includes the spike waveforms) or lists of `neo.core.Spike` objects.
- **means** (*dict*) – Dictionary, indexed by unit, of lists of spike waveforms as `neo.core.Spike` objects or numpy arrays. Means for units that are not in this dictionary will be estimated using the spikes. Default: None - means will be estimated from given spikes.
- **noise** (*Quantity 1D*) – The known noise levels (as variance) per channel of the original data. This should be estimated from the signal periods that do not contain spikes, otherwise the explained variance could be overestimated. If None, the estimate of explained variance is done without regard for noise. Default: None

Return dict A dictionary of arrays, both indexed by unit. If `noise` is `None`, the dictionary contains the fraction of explained variance per channel without taking noise into account. If `noise` is given, it contains the fraction of variance per channel explained by the means and given noise level together.

5.1.9 `stationarity` Module

`spike_amplitude_histogram`(*trains*, *num_bins*, *uniform_y_scale=True*, *unit=UnitQuantity('microvolt', 1e-06 * V, '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 `neo.core.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** (`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)

5.1.10 `tools` Module

`apply_to_dict` (*fn*, *dictionary*, **args*)

Applies a function to all spike trains in a dictionary of spike train sequences.

Parameters

- **fn** (*function*) – Function to apply. Should take a `neo.core.SpikeTrain` as first argument.
- **dictionary** (*dict*) – Dictionary of sequences of `neo.core.SpikeTrain` objects to apply the function to.
- **args** – Additional arguments which will be passed to `fn`.

Returns A new dictionary with the same keys as `dictionary`.

Return type dict

`bin_spike_trains` (*trains*, *sampling_rate*, *t_start=None*, *t_stop=None*)

Creates binned representations of spike trains.

Parameters

- **trains** (*dict*) – A dictionary of sequences of `neo.core.SpikeTrain` objects.
- **sampling_rate** (*Quantity scalar*) – The sampling rate which will be used to bin the spike trains as inverse time scalar.
- **t_stop** (*Quantity scalar*) – The desired time for the end of the last bin as time scalar. It will be the maximum stop time of all spike trains if `None` is passed.

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

Return type dict, Quantity 1D with time units

`concatenate_spike_trains` (*trains*)

Concatenates spike trains.

Parameters **trains** (*sequence*) – `neo.core.SpikeTrain` objects to concatenate.

Returns A spike train consisting of the concatenated spike trains. The spikes will be in the order of the given spike trains and `t_start` and `t_stop` will be set to the minimum and maximum value.

Return type `neo.core.SpikeTrain`

extract_spikes(*train*, *signals*, *length*, *align_time*)
Extract spikes with waveforms from analog signals using a spike train. Spikes that are too close to the beginning or end of the shortest signal to be fully extracted are ignored.

Parameters

- **train** (`neo.core.SpikeTrain`) – The spike times.
- **signals** (*sequence*) – A sequence of `neo.core.AnalogSignal` objects from which the spikes are extracted. The waveforms of the returned spikes are extracted from these signals in the same order they are given.
- **length** (*Quantity scalar*) – The length of the waveform to extract as time scalar.
- **align_time** (*Quantity scalar*) – The alignment time of the spike times as time scalar. This is the time delta from the start of the extracted waveform to the exact time of the spike.

Returns A list of `neo.core.Spike` objects, one for each time point in `train`. All returned spikes include their `waveform` property.

Return type list

maximum_spike_train_interval(*trains*, *t_start=array(inf) * s*, *t_stop=array(-inf) * s*)
Computes the minimum starting time and maximum end time of all given spike trains. This yields an interval containing the spikes of all spike trains.

Parameters

- **trains** (*dict*) – A dictionary of sequences of `neo.core.SpikeTrain` objects.
- **t_start** (*Quantity scalar*) – Maximum starting time to return.
- **t_stop** (*Quantity scalar*) – Minimum end time to return. If `None`, infinity is used.

Returns Minimum t_start time and maximum t_stop time as time scalars.

Return type Quantity scalar, Quantity scalar

minimum_spike_train_interval(*trains*, *t_start=array(-inf) * s*, *t_stop=array(inf) * s*)
Computes the maximum starting time and minimum end time that all given spike trains share. This yields the shortest interval shared by all spike trains.

Parameters

- **trains** (*dict*) – A dictionary of sequences of `neo.core.SpikeTrain` objects.
- **t_start** (*Quantity scalar*) – Minimal starting time to return.
- **t_stop** (*Quantity scalar*) – Maximum end time to return. If `None`, infinity is used.

Returns Maximum shared t_start time and minimum shared t_stop time as time scalars.

Return type Quantity scalar, Quantity scalar

remove_from_hierarchy(*obj*, *remove_half_orphans=True*)
Removes a Neo object from the hierarchy it is embedded in. Mostly downward links are removed (except for possible links in `neo.core.Spike` or `neo.core.SpikeTrain` objects). For example, when `obj` is a `neo.core.Segment`, the link from its parent `neo.core.Block` will be severed. Also, all links to the segment from its spikes and spike trains will be severed.

Parameters

- **obj** (*Neo object*) – The object to be removed.
- **remove_half_orphans** (*bool*) – When True, `neo.core.Spike` and `neo.core.SpikeTrain` belonging to a `neo.core.Segment` or `neo.core.Unit`

removed by this function will be removed from the hierarchy as well, even if they are still linked from a `neo.core.Unit` or `neo.core.Segment`, respectively. In this case, their links to the hierarchy defined by `obj` will be kept intact.

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).

spykeutils installs a script named "spykeplugin" that can be used to start plugins directly from the command line, supplying selection and plugin parameter information. It is also the default script that Spyke Viewer uses when starting plugins remotely. If you want to implement your own script for starting plugins remotely, e.g. on a server, you should conform to the interface of this script.

`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 import analysis_plugin, gui_data

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

    def start(self, current, selections):
        print 'The selected time is', self.some_time, 'milliseconds.'
        if self.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()`. Parameters that are not part of the guidata attributes of the plugin are ignored. Override if non-guidata attributes need to be serialized.

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 `neo.core.AnalogSignalArray` objects.

analog_signal_arrays_by_channelgroup ()

Return a dictionary (indexed by RecordingChannelGroup) of lists of `neo.core.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 `neo.core.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 `neo.core.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 `neo.core.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(*conversion_mode=1*)

Return a list of `neo.core.AnalogSignal` objects.

Parameters conversion_mode (*int*) – Determines what signals are returned:

1. AnalogSignal objects only
2. AnalogSignal objects extracted from AnalogSignalArrays only
3. Both AnalogSignal objects and extracted AnalogSignalArrays

analog_signals_by_channel(*conversion_mode=1*)

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

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

Parameters conversion_mode (*int*) – Determines what signals are returned:

1. AnalogSignal objects only
2. AnalogSignal objects extracted from AnalogSignalArrays only
3. Both AnalogSignal objects and extracted AnalogSignalArrays

analog_signals_by_channel_and_segment(*conversion_mode=1*)

Return a dictionary (indexed by RecordingChannel) of dictionaries (indexed by Segment) of `neo.core.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.

Parameters conversion_mode (*int*) – Determines what signals are returned:

1. AnalogSignal objects only
2. AnalogSignal objects extracted from AnalogSignalArrays only
3. Both AnalogSignal objects and extracted AnalogSignalArrays

analog_signals_by_segment(*conversion_mode=1*)

Return a dictionary (indexed by Segment) of lists of `neo.core.AnalogSignal` objects.

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

Parameters conversion_mode (*int*) – Determines what signals are returned:

1. AnalogSignal objects only
2. AnalogSignal objects extracted from AnalogSignalArrays only
3. Both AnalogSignal objects and extracted AnalogSignalArrays

analog_signals_by_segment_and_channel(*conversion_mode=1*)

Return a dictionary (indexed by Segment) of dictionaries (indexed by RecordingChannel) of `neo.core.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.

Parameters conversion_mode (*int*) – Determines what signals are returned:

1. AnalogSignal objects only
2. AnalogSignal objects extracted from AnalogSignalArrays only
3. Both AnalogSignal objects and extracted AnalogSignalArrays

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.

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.

refresh_view()
Refresh associated views of the data.

Use this method if when you change the neo hierarchy on which the selection is based (e.g. adding or removing objects). It will ensure that all current views on the data are updated, for example in Spyke Viewer.

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 `neo.core.SpikeTrain` objects.

spike_trains_by_segment()
Return a dictionary (indexed by Segment) of lists of `neo.core.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 `neo.core.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 `neo.core.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 `neo.core.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 `neo.core.Spike` objects.

spikes_by_segment()
Return a dictionary (indexed by Segment) of lists of `neo.core.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 `neo.core.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 `neo.core.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 `neo.core.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 6

Changelog

6.1 Version 0.4.1

- Faster caching for Neo lazy loading.
- Faster correlogram calculation.

6.2 Version 0.4.0

- Correlogram plot supports new square plot matrix mode and count per second in addition to per segment.
- New options in spike waveform plot.
- DataProvider objects support transparent lazy loading for compatible IOs (currently only Hdf5IO).
- DataProvider can be forced to use a certain IO instead of automatically determining it by file extension.
- Load parameters for IOs can be specified in DataProvider.
- IO class, IO parameters and IO plugins are saved in selections and properly used in startplugin.py
- Qt implementation of ProgressBar available in plot.helper (moved from Spyke Viewer).
- Loading support for IO plugins (moved from Spyke Viewer).

6.3 Version 0.3.0

- Added implementations for various spike train metrics.
- Added generation functions for poisson spike trains
- Added tools module with various utility functions, e.g. binning spike trains or removing objects from Neo hierarchies.
- Added explained variance function to spike sorting quality assessment.
- Improved legends for plots involving colored lines.
- Plots now have a minimum size and scroll bars appear if the plots would become too small.
- Renamed plot.ISI to plot.isi for consistency

6.4 Version 0.2.1

- Added “Home” and “Pan” tools for plots (useful when no middle mouse button is available).
- Changed default grid in plots to show only major grid.
- Added a method to DataProvider for refreshing views after object hierarchy changed.
- New parameter for DataProvider AnalogSignal methods: AnalogSignalArrays can be automatically converted and returned.
- Significantly improved speed of spike density estimation and optimal kernel size calculation.
- Spike sorting quality assessment using gaussian clusters is now possible without prewhitening spikes or providing prewhitened means.
- Renamed “spyke-plugin” script to “spykeplugin”

6.5 Version 0.2.0

Initial documented public release.

Acknowledgements

spykeutils was created by Robert Pröpper [1], supported by the Research Training Group GRK 1589/1. Jan Gosmann [2] contributed and tested the spike train metric implementations.

[1] Neural Information Processing Group, TU Berlin
[2] Bernstein Center for Computational Neuroscience, Berlin

CHAPTER 8

Indices and tables

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