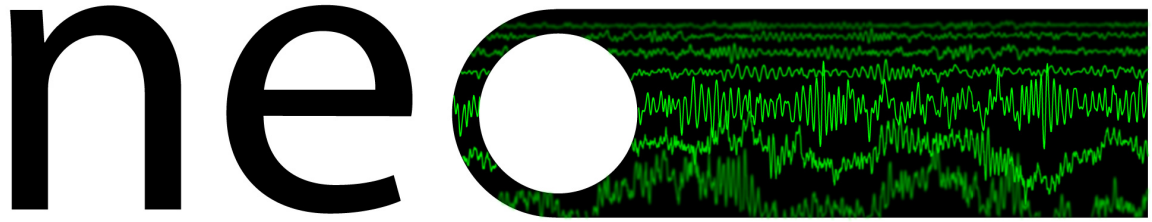
Neo Documentation

Release 0.6.1

Neo authors and contributors <neuralensemble@googlegroups.c

Mar 23, 2018

Contents



Neo is a Python package for working with electrophysiology data in Python, together with support for reading a wide range of neurophysiology file formats, including Spike2, NeuroExplorer, AlphaOmega, Axon, Blackrock, Plexon, Tdt, Igor Pro, and support for writing to a subset of these formats plus non-proprietary formats including Kwik and HDF5.

The goal of Neo is to improve interoperability between Python tools for analyzing, visualizing and generating electrophysiology data, by providing a common, shared object model. In order to be as lightweight a dependency as possible, Neo is deliberately limited to represention of data, with no functions for data analysis or visualization.

Neo is used by a number of other software tools, including SpykeViewer (data analysis and visualization), Elephant (data analysis), the G-node suite (databasing), PyNN (simulations), tridesclous (spike sorting) and ephyviewer (data visualization). OpenElectrophy (data analysis and visualization) used an older version of Neo.

Neo implements a hierarchical data model well adapted to intracellular and extracellular electrophysiology and EEG data with support for multi-electrodes (for example tetrodes). Neo's data objects build on the quantities package, which in turn builds on NumPy by adding support for physical dimensions. Thus Neo objects behave just like normal NumPy arrays, but with additional metadata, checks for dimensional consistency and automatic unit conversion.

A project with similar aims but for neuroimaging file formats is NiBabel.

CHAPTER 1

Installation

Neo is a pure Python package, so it should be easy to get it running on any system.

1.1 Dependencies

- Python >= 2.7
- numpy >= 1.7.1
- quantities >= 0.12.1

For Debian/Ubuntu, you can install these using:

```
$ apt-get install python-numpy python-pip
$ pip install quantities
```

You may need to run these as root. For other operating systems, you can download installers from the links above, or use a scientific Python distribution such as Anaconda.

Certain IO modules have additional dependencies. If these are not satisfied, Neo will still install but the IO module that uses them will fail on loading:

- scipy >= 0.12.0 for NeoMatlabIO
- h5py >= 2.5 for Hdf5IO, KwikIO
- klusta for KwikIO
- igor >= 0.2 for IgorIO
- nixio >= 1.2 for NixIO
- stfio for StimfitIO

1.2 Installing from the Python Package Index

Warning: alpha and beta releases cannot be installed from PyPI.

If you have pip installed:

```
$ pip install neo
```

This will automatically download and install the latest release (again you may need to have administrator privileges on the machine you are installing on).

To download and install manually, download:

https://github.com/NeuralEnsemble/python-neo/archive/neo-0.6.1.zip

Then:

```
$ unzip neo-0.6.1.zip
$ cd neo-0.6.1
$ python setup.py install
```

or:

```
$ python3 setup.py install
```

depending on which version of Python you are using.

1.3 Installing from source

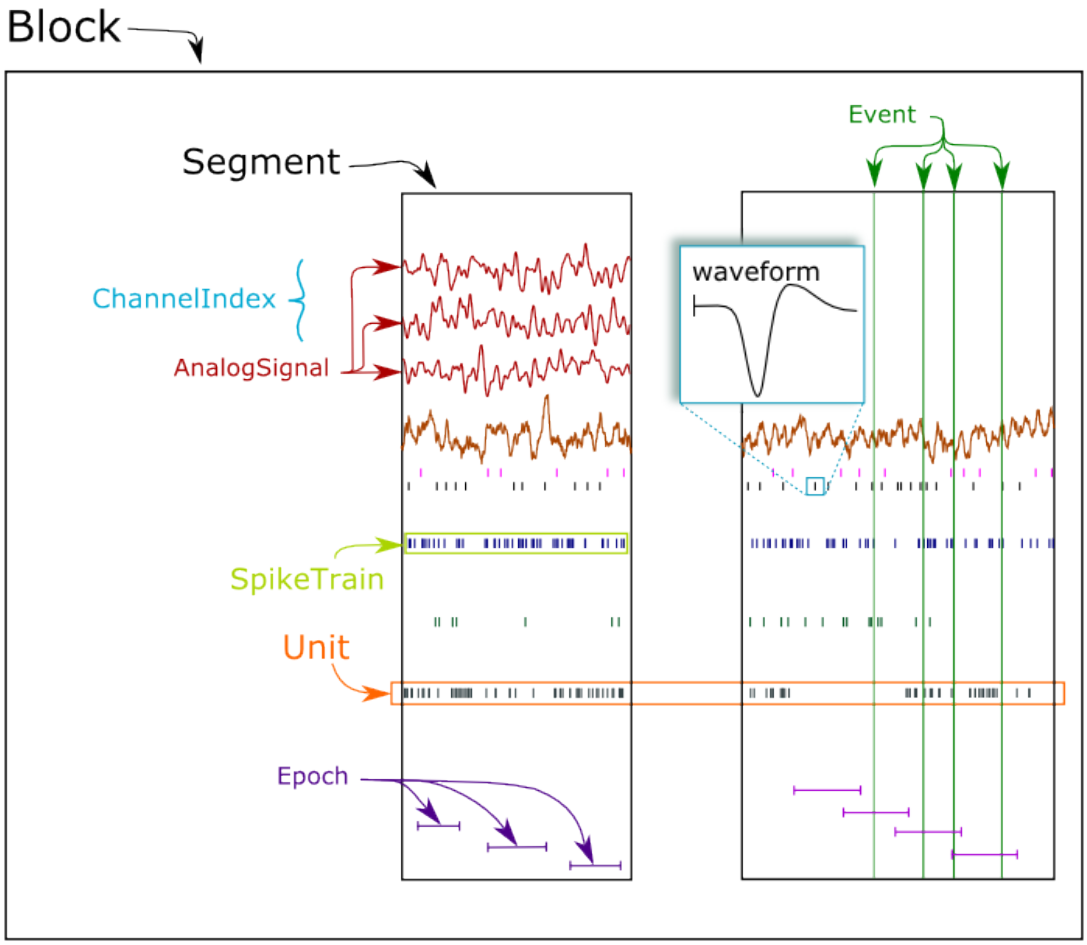
To install the latest version of Neo from the Git repository:

```
$ git clone git://github.com/NeuralEnsemble/python-neo.git
$ cd python-neo
$ python setup.py install
```

CHAPTER 2

Neo core

This figure shows the main data types in Neo:



Neo objects fall into three categories: data objects, container objects and grouping objects.

2.1 Data objects

These objects directly represent data as arrays of numerical values with associated metadata (units, sampling frequency, etc.).

- `AnalogSignal`: A regular sampling of a single- or multi-channel continuous analog signal.
- `IrregularlySampledSignal`: A non-regular sampling of a single- or multi-channel continuous analog signal.
- `SpikeTrain`: A set of action potentials (spikes) emitted by the same unit in a period of time (with optional waveforms).
- `Event`: An array of time points representing one or more events in the data.
- `Epoch`: An array of time intervals representing one or more periods of time in the data.

2.2 Container objects

There is a simple hierarchy of containers:

- *Segment*: A container for heterogeneous discrete or continous data sharing a common clock (time basis) but not necessarily the same sampling rate, start time or end time. A *Segment* can be considered as equivalent to a "trial", "episode", "run", "recording", etc., depending on the experimental context. May contain any of the data objects.
- *Block*: The top-level container gathering all of the data, discrete and continuous, for a given recording session. Contains *Segment*, *Unit* and *ChannelIndex* objects.

2.3 Grouping objects

These objects express the relationships between data items, such as which signals were recorded on which electrodes, which spike trains were obtained from which membrane potential signals, etc. They contain references to data objects that cut across the simple container hierarchy.

- *ChannelIndex*: A set of indices into *AnalogSignal* objects, representing logical and/or physical recording channels. This has two uses:
  1. for linking *AnalogSignal* objects recorded from the same (multi)electrode across several *Segment*s.
  2. for spike sorting of extracellular signals, where spikes may be recorded on more than one recording channel, and the *ChannelIndex* can be used to associate each *Unit* with the group of recording channels from which it was obtained.
- *Unit*: links the *SpikeTrain* objects within a *Block*, possibly across multiple Segments, that were emitted by the same cell. A *Unit* is linked to the *ChannelIndex* object from which the spikes were detected.

2.3.1 NumPy compatibility

Neo data objects inherit from `Quantity`, which in turn inherits from NumPy `ndarray`. This means that a Neo *AnalogSignal* is also a `Quantity` and an array, giving you access to all of the methods available for those objects.

For example, you can pass a *SpikeTrain* directly to the `numpy.histogram()` function, or an *AnalogSignal* directly to the `numpy.std()` function.

If you want to get a numpy.ndarray you use magnitude and rescale from quantities:

```
>>> np_sig = neo_analogsignal.rescale('mV').magnitude
>>> np_times = neo_analogsignal.times.rescale('s').magnitude
```

2.3.2 Relationships between objects

Container objects like *Block* or *Segment* are gateways to access other objects. For example, a *Block* can access a *Segment* with:

```
>>> bl = Block()
>>> bl.segments
# gives a list of segments
```

A *Segment* can access the *AnalogSignal* objects that it contains with:

```
>>> seg = Segment()
>>> seg.analogsignals
# gives a list of AnalogSignals
```

In the *Neo diagram* below, these *one to many* relationships are represented by cyan arrows. In general, an object can access its children with an attribute *childname+s* in lower case, e.g.

- `Block.segments`
- `Segments.analogsignals`
- `Segments.spiketrains`
- `Block.channel_indexes`

These relationships are bi-directional, i.e. a child object can access its parent:

- `Segment.block`
- `AnalogSignal.segment`
- `SpikeTrain.segment`
- `ChannelIndex.block`

Here is an example showing these relationships in use:

```
from neo.io import AxonIO
import urllib
url = "https://portal.g-node.org/neo/axon/File_axon_3.abf"
filename = './test.abf'
urllib.urlretrieve(url, filename)

r = AxonIO(filename=filename)
bl = r.read() # read the entire file > a Block
print(bl)
print(bl.segments) # child access
for seg in bl.segments:
    print(seg)
    print(seg.block) # parent access
```

In some cases, a one-to-many relationship is sufficient. Here is a simple example with tetrodes, in which each tetrode has its own group.:

```
from neo import Block, ChannelIndex
bl = Block()

# the four tetrodes
for i in range(4):
    chx = ChannelIndex(name='Tetrode %d' % i,
                       index=[0, 1, 2, 3])
    bl.channelindexes.append(chx)

# now we load the data and associate it with the created channels
# ...
```

Now consider a more complex example: a 1x4 silicon probe, with a neuron on channels 0,1,2 and another neuron on channels 1,2,3. We create a group for each neuron to hold the *Unit* object associated with this spike sorting group. Each group also contains the channels on which that neuron spiked. The relationship is many-to-many because channels 1 and 2 occur in multiple groups.:

```
bl = Block(name='probe data')

# one group for each neuron
chx0 = ChannelIndex(name='Group 0',
                    index=[0, 1, 2])
bl.channelindexes.append(chx0)

chx1 = ChannelIndex(name='Group 1',
                    index=[1, 2, 3])
bl.channelindexes.append(chx1)

# now we add the spiketrain from Unit 0 to chx0
# and add the spiketrain from Unit 1 to chx1
# ...
```

Note that because neurons are sorted from groups of channels in this situation, it is natural that the *ChannelIndex* contains a reference to the *Unit* object. That unit then contains references to its spiketrains. Also note that recording channels can be identified by names/labels as well as, or instead of, integer indices.

See *Typical use cases* for more examples of how the different objects may be used.

2.3.3 Neo diagram

Object:

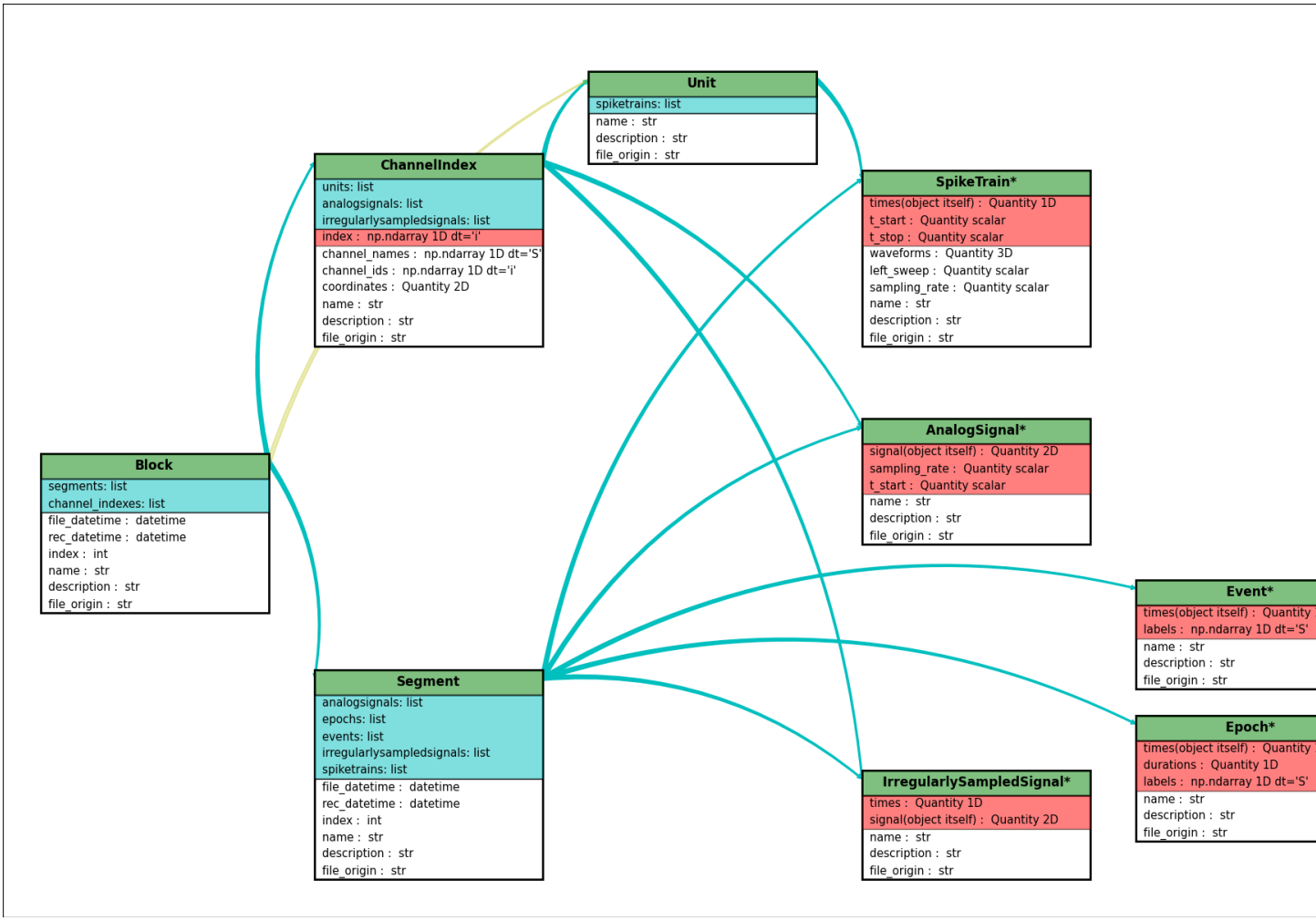
- With a star = inherits from `Quantity`

Attributes:

- In red = required
- In white = recommended

Relationship:

- In cyan = one to many
- In yellow = properties (deduced from other relationships)



`Click here for a better quality SVG diagram`

For more details, see the *API Reference*.

2.3.4 Initialization

Neo objects are initialized with "required", "recommended", and "additional" arguments.

- Required arguments MUST be provided at the time of initialization. They are used in the construction of the object.
- Recommended arguments may be provided at the time of initialization. They are accessible as Python attributes. They can also be set or modified after initialization.
- Additional arguments are defined by the user and are not part of the Neo object model. A primary goal of the Neo project is extensibility. These additional arguments are entries in an attribute of the object: a Python dict called `annotations`. Note : Neo annotations are not the same as the *__annotations__* attribute introduced in Python 3.6.

2.4 Example: SpikeTrain

SpikeTrain is a Quantity, which is a NumPy array containing values with physical dimensions. The spike times are a required attribute, because the dimensionality of the spike times determines the way in which the Quantity is constructed.

Here is how you initialize a *SpikeTrain* with required arguments:

```
>>> import neo
>>> st = neo.SpikeTrain([3, 4, 5], units='sec', t_stop=10.0)
>>> print(st)
[ 3.  4.  5.] s
```

You will see the spike times printed in a nice format including the units. Because *st* "is a" Quantity array with units of seconds, it absolutely must have this information at the time of initialization. You can specify the spike times with a keyword argument too:

```
>>> st = neo.SpikeTrain(times=[3, 4, 5], units='sec', t_stop=10.0)
```

The spike times could also be in a NumPy array.

If it is not specified, t_start is assumed to be zero, but another value can easily be specified:

```
>>> st = neo.SpikeTrain(times=[3, 4, 5], units='sec', t_start=1.0, t_stop=10.0)
>>> st.t_start
array(1.0) * s
```

Recommended attributes must be specified as keyword arguments, not positional arguments.

Finally, let's consider "additional arguments". These are the ones you define for your experiment:

```
>>> st = neo.SpikeTrain(times=[3, 4, 5], units='sec', t_stop=10.0, rat_name='Fred')
>>> print(st.annotations)
{'rat_name': 'Fred'}
```

Because rat_name is not part of the Neo object model, it is placed in the dict annotations. This dict can be modified as necessary by your code.

2.5 Annotations

As well as adding annotations as "additional" arguments when an object is constructed, objects may be annotated using the annotate() method possessed by all Neo core objects, e.g.:

```
>>> seg = Segment()
>>> seg.annotate(stimulus="step pulse", amplitude=10*nA)
>>> print(seg.annotations)
{'amplitude': array(10.0) * nA, 'stimulus': 'step pulse'}
```

Since annotations may be written to a file or database, there are some limitations on the data types of annotations: they must be "simple" types or containers (lists, dicts, tuples, NumPy arrays) of simple types, where the simple types are integer, float, complex, Quantity, string, date, time and datetime.

CHAPTER 3

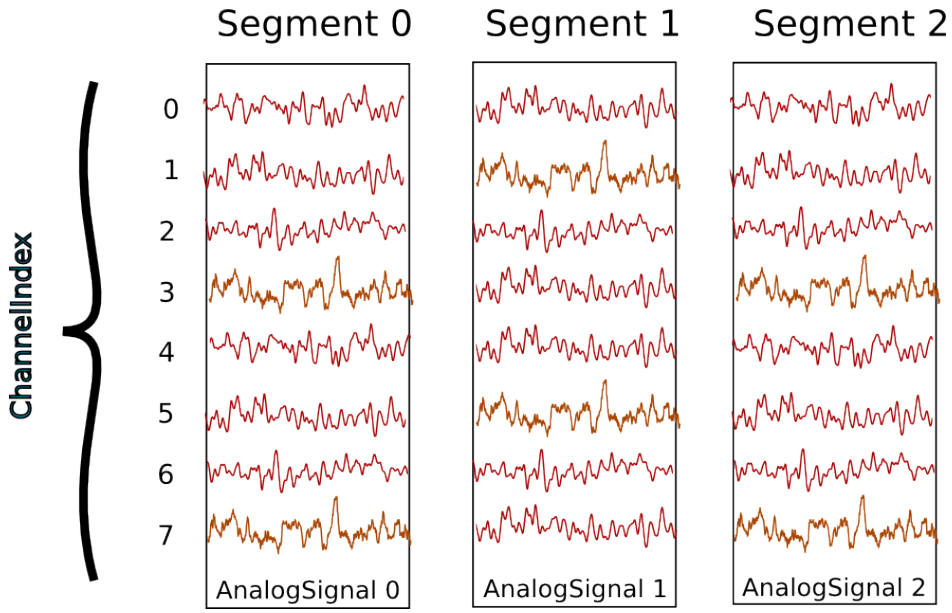
Typical use cases

3.1 Recording multiple trials from multiple channels

In this example we suppose that we have recorded from an 8-channel probe, and that we have recorded three trials/episodes. We therefore have a total of 8 x 3 = 24 signals, grouped into three `AnalogSignal` objects, one per trial.

Our entire dataset is contained in a `Block`, which in turn contains:

- 3 `Segment` objects, each representing data from a single trial,
- 1 `ChannelIndex`.



`Segment` and `ChannelIndex` objects provide two different ways to access the data, corresponding respectively, in this scenario, to access by **time** and by **space**.

Note: Segments do not always represent trials, they can be used for many purposes: segments could represent parallel recordings for different subjects, or different steps in a current clamp protocol.

Temporal (by segment)

In this case you want to go through your data in order, perhaps because you want to correlate the neural response with the stimulus that was delivered in each segment. In this example, we're averaging over the channels.

```
import numpy as np
from matplotlib import pyplot as plt

for seg in block.segments:
    print("Analyzing segment %d" % seg.index)

    avg = np.mean(seg.analogsignals[0], axis=1)

    plt.figure()
    plt.plot(avg)
    plt.title("Peak response in segment %d: %f" % (seg.index, avg.max()))
```

Spatial (by channel)

In this case you want to go through your data by channel location and average over time. Perhaps you want to see which physical location produces the strongest response, and every stimulus was the same:

```
# We assume that our block has only 1 ChannelIndex
chx = block.channelindexes[0]:
siglist = [sig[:, chx.index] for sig in chx.analogsignals]
avg = np.mean(siglist, axis=0)

plt.figure()
for index, name in zip(chx.index, chx.channel_names):
    plt.plot(avg[:, index])
    plt.title("Average response on channels %s: %s' % (index, name)
```

Mixed example

Combining simultaneously the two approaches of descending the hierarchy temporally and spatially can be tricky. Here's an example. Let's say you saw something interesting on the 6th channel (index 5) on even numbered trials during the experiment and you want to follow up. What was the average response?

```
index = chx.index[5]
avg = np.mean([seg.analogsignals[0][:, index] for seg in block.segments[::2]], axis=1)
plt.plot(avg)
```

3.2 Recording spikes from multiple tetrodes

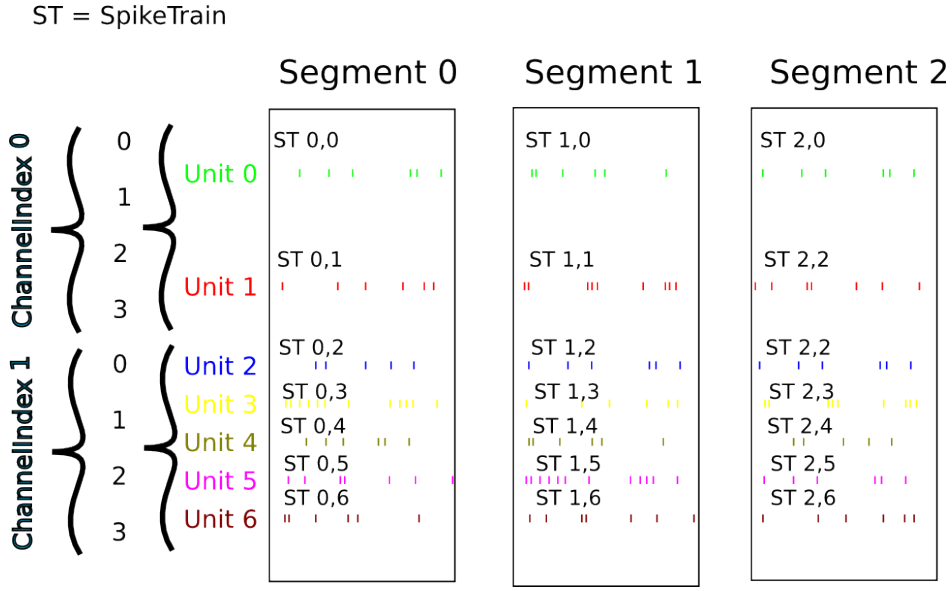
Here is a similar example in which we have recorded with two tetrodes and extracted spikes from the extra-cellular signals. The spike times are contained in `SpikeTrain` objects.

Again, our data set is contained in a `Block`, which contains:

- 3 `Segments` (one per trial).
- 2 `ChannelIndexes` (one per tetrode), which contain:

- 2 `Unit` objects (= 2 neurons) for the first `ChannelIndex`
- 5 `Units` for the second `ChannelIndex`.

In total we have 3 x 7 = 21 `SpikeTrains` in this `Block`.



There are three ways to access the `SpikeTrain` data:

- by `Segment`
- by `RecordingChannel`
- by `Unit`

By Segment

In this example, each `Segment` represents data from one trial, and we want a PSTH for each trial from all units combined:

```
for seg in block.segments:
    print("Analyzing segment %d" % seg.index)
    stlist = [st - st.t_start for st in seg.spiketrains]
    plt.figure()
    count, bins = np.histogram(stlist)
    plt.bar(bins[:-1], count, width=bins[1] - bins[0])
    plt.title("PSTH in segment %d" % seg.index)
```

By Unit

Now we can calculate the PSTH averaged over trials for each unit, using the `block.list_units` property:

```
for unit in block.list_units:
    stlist = [st - st.t_start for st in unit.spiketrains]
    plt.figure()
    count, bins = np.histogram(stlist)
    plt.bar(bins[:-1], count, width=bins[1] - bins[0])
    plt.title("PSTH of unit %s" % unit.name)
```

By ChannelIndex

Here we calculate a PSTH averaged over trials by channel location, blending all units:

```
for chx in block.channelindexes:
    stlist = []
    for unit in chx.units:
        stlist.extend([st - st.t_start for st in unit.spiketrains])
    plt.figure()
    count, bins = np.histogram(stlist)
    plt.bar(bins[:-1], count, width=bins[1] - bins[0])
    plt.title("PSTH blend of tetrode  %s" % chx.name)
```

3.3 Spike sorting

Spike sorting is the process of detecting and classifying high-frequency deflections ("spikes") on a group of physically nearby recording channels.

For example, let's say you have defined a ChannelIndex for a tetrode containing 4 separate channels. Here is an example showing (with fake data) how you could iterate over the contained signals and extract spike times. (Of course in reality you would use a more sophisticated algorithm.)

```
# generate some fake data
seg = Segment()
seg.analogsignals.append(
    AnalogSignal([[0.1, 0.1, 0.1, 0.1],
                  [-2.0, -2.0, -2.0, -2.0],
                  [0.1, 0.1, 0.1, 0.1],
                  [-0.1, -0.1, -0.1, -0.1],
                  [-0.1, -0.1, -0.1, -0.1],
                  [-3.0, -3.0, -3.0, -3.0],
                  [0.1, 0.1, 0.1, 0.1],
                  [0.1, 0.1, 0.1, 0.1]],
                 sampling_rate=1000*Hz, units='V'))
chx = ChannelIndex(channel_indexes=[0, 1, 2, 3])
chx.analogsignals.append(seg.analogsignals[0])


# extract spike trains from each channel
st_list = []
for signal in chx.analogsignals:
    # use a simple threshhold detector
    spike_mask = np.where(np.min(signal.magnitude, axis=1) < -1.0)[0]

    # create a spike train
    spike_times = signal.times[spike_mask]
    st = neo.SpikeTrain(spike_times, t_start=signal.t_start, t_stop=signal.t_stop)

    # remember the spike waveforms
    wf_list = []
    for spike_idx in np.nonzero(spike_mask)[0]:
        wf_list.append(signal[spike_idx-1:spike_idx+2, :])
    st.waveforms = np.array(wf_list)

    st_list.append(st)
```

At this point, we have a list of spiketrain objects. We could simply create a single Unit object, assign all spike trains to it, and then assign the Unit to the group on which we detected it.

```
u = Unit()
u.spiketrains = st_list
chx.units.append(u)
```

Now the recording channel group (tetrode) contains a list of analogsignals, and a single Unit object containing all of the detected spiketrains from those signals.

Further processing could assign each of the detected spikes to an independent source, a putative single neuron. (This processing is outside the scope of Neo. There are many open-source toolboxes to do it, for instance our sister project OpenElectrophy.)

In that case we would create a separate Unit for each cluster, assign its spiketrains to it, and then store all the units in the original recording channel group.

CHAPTER 4

Neo IO

4.1 Preamble

The Neo `io` module aims to provide an exhaustive way of loading and saving several widely used data formats in electrophysiology. The more these heterogeneous formats are supported, the easier it will be to manipulate them as Neo objects in a similar way. Therefore the IO set of classes propose a simple and flexible IO API that fits many format specifications. It is not only file-oriented, it can also read/write objects from a database.

At the moment, there are 3 families of IO modules:

1. for reading closed manufacturers' formats (Spike2, Plexon, AlphaOmega, BlackRock, Axon, ...)
2. for reading(/writing) formats from open source tools (KlustaKwik, Elan, WinEdr, WinWcp, PyNN, ...)
3. for reading/writing Neo structure in neutral formats (HDF5, .mat, ...) but with Neo structure inside (NeoHDF5, NeoMatlab, ...)

Combining **1** for reading and **3** for writing is a good example of use: converting your datasets to a more standard format when you want to share/collaborate.

4.2 Introduction

There is an intrinsic structure in the different Neo objects, that could be seen as a hierachy with cross-links. See *Neo core*. The highest level object is the `Block` object, which is the high level container able to encapsulate all the others.

A `Block` has therefore a list of `Segment` objects, that can, in some file formats, be accessed individually. Depending on the file format, i.e. if it is streamable or not, the whole `Block` may need to be loaded, but sometimes particular `Segment` objects can be accessed individually. Within a `Segment`, the same hierarchical organisation applies. A `Segment` embeds several objects, such as `SpikeTrain`, `AnalogSignal`, `IrregularlySampledSignal`, `Epoch`, `Event` (basically, all the different Neo objects).

Depending on the file format, these objects can sometimes be loaded separately, without the need to load the whole file. If possible, a file IO therefore provides distinct methods allowing to load only particular objects that may be present in the file. The basic idea of each IO file format is to have, as much as possible, read/write methods for the

individual encapsulated objects, and otherwise to provide a read/write method that will return the object at the highest level of hierarchy (by default, a `Block` or a `Segment`).

The *neo.io* API is a balance between full flexibility for the user (all `read_XXX()` methods are enabled) and simple, clean and understandable code for the developer (few `read_XXX()` methods are enabled). This means that not all IOs offer the full flexibility for partial reading of data files.

4.3 One format = one class

The basic syntax is as follows. If you want to load a file format that is implemented in a generic `MyFormatIO` class:

```
>>> from neo.io import MyFormatIO
>>> reader = MyFormatIO(filename="myfile.dat")
```

you can replace `MyFormatIO` by any implemented class, see *List of implemented formats*

4.4 Modes

IO can be based on a single file, a directory containing files, or a database. This is described in the `mode` attribute of the IO class.

```
>>> from neo.io import MyFormatIO
>>> print MyFormatIO.mode
'file'
```

For *file* mode the *filename* keyword argument is necessary. For *directory* mode the *dirname* keyword argument is necessary.

Ex:

```
>>> reader = io.PlexonIO(filename='File_plexon_1.plx')
>>> reader = io.TdtIO(dirname='aep_05')
```

4.5 Supported objects/readable objects

To know what types of object are supported by a given IO interface:

```
>>> MyFormatIO.supported_objects
[Segment , AnalogSignal , SpikeTrain, Event, Spike]
```

Supported objects does not mean objects that you can read directly. For instance, many formats support `AnalogSignal` but don't allow them to be loaded directly, rather to access the `AnalogSignal` objects, you must read a `Segment`:

```
>>> seg = reader.read_segment()
>>> print(seg.analogsignals)
>>> print(seg.analogsignals[0])
```

To get a list of directly readable objects

```
>>> MyFormatIO.readable_objects
[Segment]
```

The first element of the previous list is the highest level for reading the file. This mean that the IO has a `read_segment()` method:

```
>>> seg = reader.read_segment()
>>> type(seg)
neo.core.Segment
```

All IOs have a read() method that returns a list of `Block` objects (representing the whole content of the file):

```
>>> bl = reader.read()
>>> print bl[0].segments[0]
neo.core.Segment
```

4.6 Lazy option (deprecated)

In some cases you may not want to load everything in memory because it could be too big. For this scenario, some IOs implement `lazy=True/False`. With `lazy=True` all arrays will have a size of zero, but all the metadata will be loaded. The *lazy_shape* attribute is added to all array-like objects (AnalogSignal, IrregularlySampledSignal, SpikeTrain, Epoch, Event). In this case, *lazy_shape* is a tuple that has the same value as *shape* with `lazy=False`. To know if a class supports lazy mode use `ClassIO.support_lazy`. By default (if not specified), `lazy=False`, i.e. all data is loaded. The lazy option will be removed in future Neo versions. Similar functionality will be implemented using proxy objects.

Example of lazy loading:

```
>>> seg = reader.read_segment(lazy=False)
>>> print(seg.analogsignals[0].shape)  # this is (N, M)
>>> seg = reader.read_segment(lazy=True)
>>> print(seg.analogsignals[0].shape)  # this is 0, the AnalogSignal is empty
>>> print(seg.analogsignals[0].lazy_shape)  # this is (N, M)
```

4.7 Details of API

The *neo.io* API is designed to be simple and intuitive:

- each file format has an IO class (for example for Spike2 files you have a `Spike2IO` class).
- each IO class inherits from the `BaseIO` class.
- each IO class can read or write directly one or several Neo objects (for example `Segment`, `Block`, ...): see the `readable_objects` and `writable_objects` attributes of the IO class.
- each IO class supports part of the *neo.core* hierachy, though not necessarily all of it (see `supported_objects`).
- each IO class has a `read()` method that returns a list of `Block` objects. If the IO only supports `Segment` reading, the list will contain one block with all segments from the file.
- each IO class that supports writing has a `write()` method that takes as a parameter a list of blocks, a single block or a single segment, depending on the IO's `writable_objects`.

- each IO is able to do a *lazy* load: all metadata (e.g. `sampling_rate`) are read, but not the actual numerical data. lazy_shape attribute is added to provide information on real size.
- each IO is able to save and load all required attributes (metadata) of the objects it supports.
- each IO can freely add user-defined or manufacturer-defined metadata to the `annotations` attribute of an object.

4.8 If you want to develop your own IO

See *IO developers' guide* for information on how to implement a new IO.

4.9 List of implemented formats

neo.io provides classes for reading and/or writing electrophysiological data files.

Note that if the package dependency is not satisfied for one io, it does not raise an error but a warning.

`neo.io.iolist` provides a list of successfully imported io classes.

Functions:

`neo.io.`**`get_io`**(*filename*, **args*, ***kwargs*)
Return a Neo IO instance, guessing the type based on the filename suffix.

Classes:

class `neo.io.`**`AlphaOmegaIO`**(*filename=None*)
Class for reading data from Alpha Omega .map files (experimental)

This class is an experimental reader with important limitations. See the source code for details of the limitations. The code of this reader is of alpha quality and received very limited testing.

Usage:

```
>>> from neo import io
>>> r = io.AlphaOmegaIO( filename = 'File_AlphaOmega_1.map')
>>> blck = r.read_block()
>>> print blck.segments[0].analogsignals
```

class `neo.io.`**`AsciiSignalIO`**(*filename=None*)
Class for reading signal in generic ascii format. Columns respresents signals. They all share the same sampling rate. The sampling rate is externally known or the first columns could hold the time vector.

Usage:

```
>>> from neo import io
>>> r = io.AsciiSignalIO(filename='File_asciisignal_2.txt')
>>> seg = r.read_segment()
>>> print seg.analogsignals
[<AnalogSignal(array([ 39.0625    ,   0.        ,   0.        , ..., -26.
↪85546875 ...
```

class `neo.io.`**`AsciiSpikeTrainIO`**(*filename=None*)
Class for reading/writing SpikeTrains in a text file. Each Spiketrain is a line.

Usage:

```
>>> from neo import io
>>> r = io.AsciiSpikeTrainIO( filename = 'File_ascii_spiketrain_1.txt')
>>> seg = r.read_segment()
>>> print seg.spiketrains
[<SpikeTrain(array([ 3.89981604,  4.73258781,  0.608428  ,  4.60246277,  1.
↪23805797,
...
```

class neo.io.**AxonIO**(*filename*)

Class for reading data from pCLAMP and AxoScope files (.abf version 1 and 2), developed by Molecular device/Axon technologies.

- abf = Axon binary file
- atf is a text file based format from axon that could be read by AsciiIO (but this file is less efficient.)

class neo.io.**BCI2000IO**(*filename*)

Class for reading data from a BCI2000 .dat file, either version 1.0 or 1.1

class neo.io.**BlackrockIO**(*filename*, *nsx_to_load=None*, ***kargs*)

This IO reads .nev/.nsX files of the Blackrock (Cerebus) recording system.

class neo.io.**BrainVisionIO**(*filename*)

Class for reading data from the BrainVision product.

class neo.io.**BrainwareDamIO**(*filename=None*)

Class for reading Brainware raw data files with the extension '.dam'.

The read_block method returns the first Block of the file. It will automatically close the file after reading. The read method is the same as read_block.

Note:

The file format does not contain a sampling rate. The sampling rate is set to 1 Hz, but this is arbitrary. If you have a corresponding .src or .f32 file, you can get the sampling rate from that. It may also be possible to infer it from the attributes, such as "sweep length", if present.

Usage:

```
>>> from neo.io.brainwaredamio import BrainwareDamIO
>>> damfile = BrainwareDamIO(filename='multi_500ms_mulitrep_ch1.dam')
>>> blk1 = damfile.read()
>>> blk2 = damfile.read_block()
>>> print blk1.segments
>>> print blk1.segments[0].analogsignals
>>> print blk1.units
>>> print blk1.units[0].name
>>> print blk2
>>> print blk2[0].segments
```

class neo.io.**BrainwareF32IO**(*filename=None*)

Class for reading Brainware Spike ReCord files with the extension '.f32'

The read_block method returns the first Block of the file. It will automatically close the file after reading. The read method is the same as read_block.

The read_all_blocks method automatically reads all Blocks. It will automatically close the file after reading.

The read_next_block method will return one Block each time it is called. It will automatically close the file and reset to the first Block after reading the last block. Call the close method to close the file and reset this method back to the first Block.

The isopen property tells whether the file is currently open and reading or closed.

Note 1: There is always only one ChannelIndex. BrainWare stores the equivalent of ChannelIndexes in separate files.

Usage:

```
>>> from neo.io.brainwaref32io import BrainwareF32IO
>>> f32file = BrainwareF32IO(filename='multi_500ms_mulitrep_ch1.f32')
>>> blk1 = f32file.read()
>>> blk2 = f32file.read_block()
>>> print blk1.segments
>>> print blk1.segments[0].spiketrains
>>> print blk1.units
>>> print blk1.units[0].name
>>> print blk2
>>> print blk2[0].segments
```

class neo.io.**BrainwareSrcIO**(*filename=None*)

Class for reading Brainware Spike ReCord files with the extension '.src'

The read_block method returns the first Block of the file. It will automatically close the file after reading. The read method is the same as read_block.

The read_all_blocks method automatically reads all Blocks. It will automatically close the file after reading.

The read_next_block method will return one Block each time it is called. It will automatically close the file and reset to the first Block after reading the last block. Call the close method to close the file and reset this method back to the first Block.

The _isopen property tells whether the file is currently open and reading or closed.

Note 1: The first Unit in each ChannelIndex is always UnassignedSpikes, which has a SpikeTrain for each Segment containing all the spikes not assigned to any Unit in that Segment.

Note 2: The first Segment in each Block is always Comments, which stores all comments as an Event object.

Note 3: The parameters from the BrainWare table for each condition are stored in the Segment annotations. If there are multiple repetitions of a condition, each repetition is stored as a separate Segment.

Note 4: There is always only one ChannelIndex. BrainWare stores the equivalent of ChannelIndexes in separate files.

Usage:

```
>>> from neo.io.brainwaresrcio import BrainwareSrcIO
>>> srcfile = BrainwareSrcIO(filename='multi_500ms_mulitrep_ch1.src')
>>> blk1 = srcfile.read()
>>> blk2 = srcfile.read_block()
>>> blks = srcfile.read_all_blocks()
>>> print blk1.segments
>>> print blk1.segments[0].spiketrains
>>> print blk1.units
>>> print blk1.units[0].name
>>> print blk2
>>> print blk2[0].segments
>>> print blks
>>> print blks[0].segments
```

class neo.io.**ElanIO**(*filename*)

Class for reading data from Elan.

Elan is software for studying time-frequency maps of EEG data.

Elan is developed in Lyon, France, at INSERM U821

https://elan.lyon.inserm.fr

class neo.io.**IgorIO**(*filename=None*, *parse_notes=None*)
Class for reading Igor Binary Waves (.ibw) or Packed Experiment (.pxp) files written by WaveMetrics' IGOR Pro software.

It requires the *igor* Python package by W. Trevor King.

Usage:

```
>>> from neo import io
>>> r = io.IgorIO(filename='...ibw')
```

class neo.io.**KlustaKwikIO**(*filename*, *sampling_rate=30000.0*)
Reading and writing from KlustaKwik-format files.

class neo.io.**KwikIO**(*filename*)
Class for "reading" experimental data from a .kwik file.

Generates a `Segment` with a `AnalogSignal`

class neo.io.**MicromedIO**(*filename*)
Class for reading/writing data from Micromed files (.trc).

class neo.io.**NeoHdf5IO**(*filename*)
Class for reading HDF5 format files created by Neo version 0.4 or earlier.

Writing to HDF5 is not supported by this IO; we recommend using NixIO for this.

class neo.io.**NeoMatlabIO**(*filename=None*)
Class for reading/writing Neo objects in MATLAB format (.mat) versions 5 to 7.2.

This module is a bridge for MATLAB users who want to adopt the Neo object representation. The nomenclature is the same but using Matlab structs and cell arrays. With this module MATLAB users can use neo.io to read a format and convert it to .mat.

Rules of conversion:

- Neo classes are converted to MATLAB structs. e.g., a Block is a struct with attributes "name", "file_datetime", ...
- Neo one_to_many relationships are cellarrays in MATLAB. e.g., `seg.analogsignals[2]` in Python Neo will be `seg.analogsignals{3}` in MATLAB.
- Quantity attributes are represented by 2 fields in MATLAB. e.g., `anasig.t_start = 1.5 * s` in Python will be `anasig.t_start = 1.5` and `anasig.t_start_unit = 's'` in MATLAB.
- classes that inherit from Quantity (AnalogSignal, SpikeTrain, ...) in Python will have 2 fields (array and units) in the MATLAB struct. e.g.: `AnalogSignal( [1., 2., 3.], 'V')` in Python will be `anasig.array = [1. 2. 3]` and `anasig.units = 'V'` in MATLAB.

1 - **Scenario 1: create data in MATLAB and read them in Python**

This MATLAB code generates a block:

```
block = struct();
block.segments = { };
block.name = 'my block with matlab';
for s = 1:3
```

```
    seg = struct();
    seg.name = strcat('segment ',num2str(s));

    seg.analogsignals = { };
    for a = 1:5
        anasig = struct();
        anasig.signal = rand(100,1);
        anasig.signal_units = 'mV';
        anasig.t_start = 0;
        anasig.t_start_units = 's';
        anasig.sampling_rate = 100;
        anasig.sampling_rate_units = 'Hz';
        seg.analogsignals{a} = anasig;
    end

    seg.spiketrains = { };
    for t = 1:7
        sptr = struct();
        sptr.times = rand(30,1)*10;
        sptr.times_units = 'ms';
        sptr.t_start = 0;
        sptr.t_start_units = 'ms';
        sptr.t_stop = 10;
        sptr.t_stop_units = 'ms';
        seg.spiketrains{t} = sptr;
    end

    event = struct();
    event.times = [0, 10, 30];
    event.times_units = 'ms';
    event.labels = ['trig0'; 'trig1'; 'trig2'];
    seg.events{1} = event;

    epoch = struct();
    epoch.times = [10, 20];
    epoch.times_units = 'ms';
    epoch.durations = [4, 10];
    epoch.durations_units = 'ms';
    epoch.labels = ['a0'; 'a1'];
    seg.epochs{1} = epoch;

    block.segments{s} = seg;

end

save 'myblock.mat' block -V7
```

This code reads it in Python:

```
import neo
r = neo.io.NeoMatlabIO(filename='myblock.mat')
bl = r.read_block()
print bl.segments[1].analogsignals[2]
print bl.segments[1].spiketrains[4]
```

2 - **Scenario 2: create data in Python and read them in MATLAB**

This Python code generates the same block as in the previous scenario:

```
import neo
import quantities as pq
from scipy import rand, array

bl = neo.Block(name='my block with neo')
for s in range(3):
    seg = neo.Segment(name='segment' + str(s))
    bl.segments.append(seg)
    for a in range(5):
        anasig = neo.AnalogSignal(rand(100)*pq.mV, t_start=0*pq.s,␣
↪sampling_rate=100*pq.Hz)
        seg.analogsignals.append(anasig)
    for t in range(7):
        sptr = neo.SpikeTrain(rand(40)*pq.ms, t_start=0*pq.ms, t_
↪stop=10*pq.ms)
        seg.spiketrains.append(sptr)
    ev = neo.Event([0, 10, 30]*pq.ms, labels=array(['trig0', 'trig1',
↪'trig2']))
    ep = neo.Epoch([10, 20]*pq.ms, durations=[4, 10]*pq.ms, labels=array([
↪'a0', 'a1']))
    seg.events.append(ev)
    seg.epochs.append(ep)

from neo.io.neomatlabio import NeoMatlabIO
w = NeoMatlabIO(filename='myblock.mat')
w.write_block(bl)
```

This MATLAB code reads it:

```
load 'myblock.mat'
block.name
block.segments{2}.analogsignals{3}.signal
block.segments{2}.analogsignals{3}.signal_units
block.segments{2}.analogsignals{3}.t_start
block.segments{2}.analogsignals{3}.t_start_units
```

3 - **Scenario 3: conversion**

This Python code converts a Spike2 file to MATLAB:

```
from neo import Block
from neo.io import Spike2IO, NeoMatlabIO

r = Spike2IO(filename='spike2.smr')
w = NeoMatlabIO(filename='convertedfile.mat')
blocks = r.read()
w.write(blocks[0])
```

class neo.io.**NestIO**(*filenames=None*)

Class for reading NEST output files. GDF files for the spike data and DAT files for analog signals are possible.

Usage:

```
>>> from neo.io.nestio import NestIO
```

```
>>> files = ['membrane_voltages-1261-0.dat',
        'spikes-1258-0.gdf']
>>> r = NestIO(filenames=files)
```

```
>>> seg = r.read_segment(gid_list=[], t_start=400 * pq.ms,
                         t_stop=600 * pq.ms,
                         id_column_gdf=0, time_column_gdf=1,
                         id_column_dat=0, time_column_dat=1,
                         value_columns_dat=2)
```

class neo.io.**NeuralynxIO**(*dirname*, *use_cache=False*, *cache_path='same_as_resource'*)
Class for reading data from Neuralynx files. This IO supports NCS, NEV, NSE and NTT file formats.

NCS contains signals for one channel NEV contains events NSE contains spikes and waveforms for mono electrodes NTT contains spikes and waveforms for tetrodes

class neo.io.**NeuroExplorerIO**(*filename*)
Class for reading data from NeuroExplorer (.nex)

class neo.io.**NeuroScopeIO**(*filename*)
Reading from Neuroscope format files.

Ref: http://neuroscope.sourceforge.net/

neo.io.**NeuroshareIO**
alias of NeurosharectypesIO

class neo.io.**NixIO**(*filename*, *mode='rw'*)
Class for reading and writing NIX files.

class neo.io.**NSDFIO**(*filename=None*)
Class for reading and writing files in NSDF Format.

It supports reading and writing: Block, Segment, AnalogSignal, ChannelIndex, with all relationships and metadata.

class neo.io.**PickleIO**(*filename=None*, ***kargs*)
A class for reading and writing Neo data from/to the Python "pickle" format.

Note that files in this format may not be readable if using a different version of Neo to that used to create the file. It should therefore not be used for long-term storage, but rather for intermediate results in a pipeline.

class neo.io.**PlexonIO**(*filename*)
Class for reading the old data format from Plexon acquisition system (.plx)

Note that Plexon now use a new format PL2 which is NOT supported by this IO.

Compatible with versions 100 to 106. Other versions have not been tested.

class neo.io.**PyNNNumpyIO**(*filename=None*, ***kargs*)
(DEPRECATED) Reads/writes data from/to PyNN NumpyBinaryFile format

class neo.io.**PyNNTextIO**(*filename=None*, ***kargs*)
(DEPRECATED) Reads/writes data from/to PyNN StandardTextFile format

class neo.io.**RawBinarySignalIO**(*filename*, *dtype='int16'*, *sampling_rate=10000.0*, *nb_channel=2*, *signal_gain=1.0*, *signal_offset=0.0*, *byte_offset=0*)
Class for reading/writing data in a raw binary interleaved compact file.

Important release note

Since the version neo 0.6.0 and the neo.rawio API, argmuents of the IO (dtype, nb_channel, sampling_rate) must be given at the __init__ and not at read_segment() because there is no read_segment() in neo.rawio classes.

So now the usage is:

>>>>r = io.RawBinarySignalIO(filename='file.raw', dtype='int16', nb_channel=16, sampling_rate=10000.)

class neo.io.**StimfitIO**(*filename=None*)

Class for converting a stfio Recording to a Neo object. Provides a standardized representation of the data as defined by the neo project; this is useful to explore the data with an increasing number of electrophysiology software tools that rely on the Neo standard.

stfio is a standalone file i/o Python module that ships with the Stimfit program (http://www.stimfit.org). It is a Python wrapper around Stimfit's file i/o library (libstfio) that natively provides support for the following file types:

- ABF (Axon binary file format; pClamp 6–9)
- ABF2 (Axon binary file format 2; pClamp 10+)
- ATF (Axon text file format)
- AXGX/AXGD (Axograph X file format)
- CFS (Cambridge electronic devices filing system)
- HEKA (HEKA binary file format)
- HDF5 (Hierarchical data format 5; only hdf5 files written by Stimfit or stfio are supported)

In addition, libstfio can use the biosig file i/o library as an additional file handling backend (http://biosig.sourceforge.net/), extending support to more than 30 additional file formats (http://pub.ist.ac.at/~schloegl/biosig/TESTED).

Example usage:

```
>>> import neo
>>> neo_obj = neo.io.StimfitIO("file.abf")
or
>>> import stfio
>>> stfio_obj = stfio.read("file.abf")
>>> neo_obj = neo.io.StimfitIO(stfio_obj)
```

class neo.io.**TdtIO**(*dirname*)

Class for reading data from from Tucker Davis TTank format.

Terminology: TDT holds data with tanks (actually a directory). And tanks hold sub blocks (sub directories). Tanks correspond to Neo Blocks and TDT blocks correspond to Neo Segments.

class neo.io.**WinEdrIO**(*filename*)

Class for reading data from WinEdr, a software tool written by John Dempster.

WinEdr is free: http://spider.science.strath.ac.uk/sipbs/software.htm

class neo.io.**WinWcpIO**(*filename*)

Class for reading data from WinWCP, a software tool written by John Dempster.

WinWCP is free: http://spider.science.strath.ac.uk/sipbs/software.htm

4.10 Logging

neo uses the standard Python `logging` module for logging. All *neo.io* classes have logging set up by default, although not all classes produce log messages. The logger name is the same as the full qualified class name, e.g. `neo.io.hdf5io.NeoHdf5IO`. By default, only log messages that are critically important for users are displayed,

so users should not disable log messages unless they are sure they know what they are doing. However, if you wish to disable the messages, you can do so:

```
>>> import logging
>>>
>>> logger = logging.getLogger('neo')
>>> logger.setLevel(100)
```

Some io classes provide additional information that might be interesting to advanced users. To enable these messages, do the following:

```
>>> import logging
>>>
>>> logger = logging.getLogger('neo')
>>> logger.setLevel(logging.INFO)
```

It is also possible to log to a file in addition to the terminal:

```
>>> import logging
>>>
>>> logger = logging.getLogger('neo')
>>> handler = logging.FileHandler('filename.log')
>>> logger.addHandler(handler)
```

To only log to the terminal:

```
>>> import logging
>>> from neo import logging_handler
>>>
>>> logger = logging.getLogger('neo')
>>> handler = logging.FileHandler('filename.log')
>>> logger.addHandler(handler)
>>>
>>> logging_handler.setLevel(100)
```

This can also be done for individual IO classes:

```
>>> import logging
>>>
>>> logger = logging.getLogger('neo.io.hdf5io.NeoHdf5IO')
>>> handler = logging.FileHandler('filename.log')
>>> logger.addHandler(handler)
```

Individual IO classes can have their loggers disabled as well:

```
>>> import logging
>>>
>>> logger = logging.getLogger('neo.io.hdf5io.NeoHdf5IO')
>>> logger.setLevel(100)
```

And more detailed logging messages can be enabled for individual IO classes:

```
>>> import logging
>>>
>>> logger = logging.getLogger('neo.io.hdf5io.NeoHdf5IO')
>>> logger.setLevel(logging.INFO)
```

The default handler, which is used to print logs to the command line, is stored in `neo.logging_handler`. This example changes how the log text is displayed:

```
>>> import logging
>>> from neo import logging_handler
>>>
>>> formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s -
↪%(message)s')
>>> logging_handler.setFormatter(formatter)
```

For more complex logging, please see the documentation for the logging module.

Note: If you wish to implement more advanced logging as describe in the documentation for the logging module or elsewhere on the internet, please do so before calling any *neo* functions or initializing any *neo* classes. This is because the default handler is created when *neo* is imported, but it is not attached to the *neo* logger until a class that uses logging is initialized or a function that uses logging is called. Further, the handler is only attached if there are no handlers already attached to the root logger or the *neo* logger, so adding your own logger will override the default one. Additional functions and/or classes may get logging during bugfix releases, so code relying on particular modules not having logging may break at any time without warning.

CHAPTER 5

Neo RawIO

For performance and memory consumption reasons a new layer has been added to Neo.

In brief:

- **neo.io** is the user-oriented read/write layer. Reading consists of getting a tree of Neo objects from a data source (file, url, or directory). When reading, all Neo objects are correctly scaled to the correct units. Writing consists of making a set of Neo objects persistent in a file format.
- **neo.rawio** is a low-level layer for reading data only. Reading consists of getting NumPy buffers (often int16/int64) of signals/spikes/events. Scaling to real values (microV, times, ...) is done in a second step. Here the underlying objects must be consistent across Blocks and Segments for a given data source.

The neo.rawio API has been added for developers. The neo.rawio is close to what could be a C API for reading data but in Python/NumPy.

Not all IOs are implemented in `neo.rawio` but all classes implemented in `neo.rawio` are also available in *neo.io*.

Possible uses of the `neo.rawio` API are:

- fast reading chunks of signals in int16 and do the scaling of units (uV) on a GPU while scaling the zoom. This should improve bandwith HD to RAM and RAM to GPU memory.
- load only some small chunk of data for heavy computations. For instance the spike sorting module tridesclous does this.

The `neo.rawio` API is less flexible than *neo.io* and has some limitations:

- read-only
- AnalogSignals must have the same characteristcs across all Blocks and Segments: `sampling_rate`, `shape[1]`, `dtype`
- AnalogSignals should all have the same value of `sampling_rate`, otherwise they won't be read at the same time.
- Units must have SpikeTrain event if empty across all Block and Segment
- Epoch and Event are processed the same way (with `durations=None` for Event).

For an intuitive comparison of *neo.io* and neo.rawio see:

- example/read_file_neo_io.py
- example/read_file_neo_rawio.py

One speculative benefit of the neo.rawio API should be that a developer should be able to code a new RawIO class with little knowledge of the Neo tree of objects or of the quantities package.

5.1 Basic usage

First create a reader from a class:

```
>>> from neo.rawio import PlexonRawIO
>>> reader = PlexonRawIO(filename='File_plexon_3.plx')
```

Then browse the internal header and display information:

```
>>> reader.parse_header()
>>> print(reader)
PlexonRawIO: File_plexon_3.plx
nb_block: 1
nb_segment:  [1]
signal_channels: [V1]
unit_channels: [Wspk1u, Wspk2u, Wspk4u, Wspk5u ... Wspk29u Wspk30u Wspk31u Wspk32u]
event_channels: []
```

You get the number of blocks and segments per block. You have information about channels: **signal_channels**, **unit_channels**, **event_channels**.

All this information is internally available in the *header* dict:

```
>>> for k, v in reader.header.items():
...     print(k, v)
signal_channels [('V1', 0,  1000., 'int16', '',  2.44140625,  0., 0)]
event_channels []
nb_segment [1]
nb_block 1
unit_channels [('Wspk1u', 'ch1#0', '',  0.00146484,  0., 0,  30000.)
('Wspk2u', 'ch2#0', '',  0.00146484,  0., 0,  30000.)
...
```

Read signal chunks of data and scale them:

```
>>> channel_indexes = None  #could be channel_indexes = [0]
>>> raw_sigs = reader.get_analogsignal_chunk(block_index=0, seg_index=0,
                    i_start=1024, i_stop=2048, channel_indexes=channel_indexes)
>>> float_sigs = reader.rescale_signal_raw_to_float(raw_sigs, dtype='float64')
>>> sampling_rate = reader.get_signal_sampling_rate()
>>> t_start = reader.get_signal_t_start(block_index=0, seg_index=0)
>>> units =reader.header['signal_channels'][0]['units']
>>> print(raw_sigs.shape, raw_sigs.dtype)
>>> print(float_sigs.shape, float_sigs.dtype)
>>> print(sampling_rate, t_start, units)
(1024, 1) int16
(1024, 1) float64
1000.0 0.0 V
```

There are 3 ways to select a subset of channels: by index (0 based), by id or by name. By index is not ambiguous 0 to n-1 (included), for some IOs channel_names (and sometimes channel_ids) have no guarantees to be unique, in such cases it would raise an error.

Example with BlackrockRawIO for the file FileSpec2.3001:

```
>>> raw_sigs = reader.get_analogsignal_chunk(channel_indexes=None) #Take all channels
>>> raw_sigs1 = reader.get_analogsignal_chunk(channel_indexes=[0,  2, 4])) #Take 0 2␣
↪and 4
>>> raw_sigs2 = reader.get_analogsignal_chunk(channel_ids=[1, 3, 5]) # Same but with␣
↪there id (1 based)
>>> raw_sigs3 = reader.get_analogsignal_chunk(channel_names=['chan1', 'chan3', 'chan5
↪'])) # Same but with there name
print(raw_sigs1.shape[1], raw_sigs2.shape[1], raw_sigs3.shape[1])
3, 3, 3
```

Inspect units channel. Each channel gives a SpikeTrain for each Segment. Note that for many formats a physical channel can have several units after spike sorting. So the nb_unit could be more than physical channel or signal channels.

```
>>> nb_unit = reader.unit_channels_count()
>>> print('nb_unit', nb_unit)
nb_unit 30
>>> for unit_index in range(nb_unit):
...     nb_spike = reader.spike_count(block_index=0, seg_index=0, unit_index=unit_
↪index)
...     print('unit_index', unit_index, 'nb_spike', nb_spike)
unit_index 0 nb_spike 701
unit_index 1 nb_spike 716
unit_index 2 nb_spike 69
unit_index 3 nb_spike 12
unit_index 4 nb_spike 95
unit_index 5 nb_spike 37
unit_index 6 nb_spike 25
unit_index 7 nb_spike 15
unit_index 8 nb_spike 33
...
```

Get spike timestamps only between 0 and 10 seconds and convert them to spike times:

```
>>> spike_timestamps = reader.spike_timestamps(block_index=0, seg_index=0, unit_
↪index=0,
                    t_start=0., t_stop=10.)
>>> print(spike_timestamps.shape, spike_timestamps.dtype, spike_timestamps[:5])
(424,) int64 [  90  420  708 1020 1310]
>>> spike_times =  reader.rescale_spike_timestamp( spike_timestamps, dtype='float64')
>>> print(spike_times.shape, spike_times.dtype, spike_times[:5])
(424,) float64 [ 0.003       0.014       0.0236      0.034       0.04366667]
```

Get spike waveforms between 0 and 10 s:

```
>>> raw_waveforms = reader.spike_raw_waveforms(  block_index=0, seg_index=0, unit_
↪index=0,
                    t_start=0., t_stop=10.)
>>> print(raw_waveforms.shape, raw_waveforms.dtype, raw_waveforms[0,0,:4])
(424, 1, 64) int16 [-449 -206   34   40]
>>> float_waveforms = reader.rescale_waveforms_to_float(raw_waveforms, dtype='float32
↪', unit_index=0)
>>> print(float_waveforms.shape, float_waveforms.dtype, float_waveforms[0,0,:4])
```

```
(424, 1, 64) float32 [-0.65771484 -0.30175781  0.04980469  0.05859375]
```

Count events per channel:

```
>>> reader = PlexonRawIO(filename='File_plexon_2.plx')
>>> reader.parse_header()
>>> nb_event_channel = reader.event_channels_count()
nb_event_channel 28
>>> print('nb_event_channel', nb_event_channel)
>>> for chan_index in range(nb_event_channel):
...     nb_event = reader.event_count(block_index=0, seg_index=0, event_channel_
↪index=chan_index)
...     print('chan_index',chan_index, 'nb_event', nb_event)
chan_index 0 nb_event 1
chan_index 1 nb_event 0
chan_index 2 nb_event 0
chan_index 3 nb_event 0
...
```

Read event timestamps and times for chanindex=0 and with time limits (t_start=None, t_stop=None):

```
>>> ev_timestamps, ev_durations, ev_labels = reader.event_timestamps(block_index=0,␣
↪seg_index=0, event_channel_index=0,
                    t_start=None, t_stop=None)
>>> print(ev_timestamps, ev_durations, ev_labels)
[1268] None ['0']
>>> ev_times = reader.rescale_event_timestamp(ev_timestamps, dtype='float64')
>>> print(ev_times)
[ 0.0317]
```

CHAPTER 6

Examples

6.1 Introduction

A set of examples in `neo/examples/` illustrates the use of Neo classes.

```
# -*- coding: utf-8 -*-
"""
This is an example for plotting a Neo object with matplotlib.
"""

import urllib

import numpy as np
import quantities as pq
from matplotlib import pyplot

import neo

url = 'https://portal.g-node.org/neo/'
# distantfile = url + 'neuroexplorer/File_neuroexplorer_2.nex'
# localfile = 'File_neuroexplorer_2.nex'

distantfile = 'https://portal.g-node.org/neo/plexon/File_plexon_3.plx'
localfile = './File_plexon_3.plx'

urllib.request.urlretrieve(distantfile, localfile)

# reader = neo.io.NeuroExplorerIO(filename='File_neuroexplorer_2.nex')
reader = neo.io.PlexonIO(filename='File_plexon_3.plx')

bl = reader.read(lazy=False)[0]
for seg in bl.segments:
    print("SEG: " + str(seg.file_origin))
    fig = pyplot.figure()
    ax1 = fig.add_subplot(2, 1, 1)
```

```
    ax2 = fig.add_subplot(2, 1, 2)
    ax1.set_title(seg.file_origin)
    ax1.set_ylabel('arbitrary units')
    mint = 0 * pq.s
    maxt = np.inf * pq.s
    for i, asig in enumerate(seg.analogsignals):
        times = asig.times.rescale('s').magnitude
        asig = asig.magnitude
        ax1.plot(times, asig)

    trains = [st.rescale('s').magnitude for st in seg.spiketrains]
    colors = pyplot.cm.jet(np.linspace(0, 1, len(seg.spiketrains)))
    ax2.eventplot(trains, colors=colors)

pyplot.show()
```

CHAPTER 7

API Reference

neo.core provides classes for storing common electrophysiological data types. Some of these classes contain raw data, such as spike trains or analog signals, while others are containers to organize other classes (including both data classes and other container classes).

Classes from *neo.io* return nested data structures containing one or more class from this module.

Classes:

class neo.core.**Block**(*name=None*, *description=None*, *file_origin=None*, *file_datetime=None*, *rec_datetime=None*, *index=None*, ***annotations*)

Main container gathering all the data, whether discrete or continous, for a given recording session.

A block is not necessarily temporally homogeneous, in contrast to *Segment*.

Usage:

```
>>> from neo.core import (Block, Segment, ChannelIndex,
...                       AnalogSignal)
>>> from quantities import nA, kHz
>>> import numpy as np
>>>
>>> # create a Block with 3 Segment and 2 ChannelIndex objects
,,, blk = Block()
>>> for ind in range(3):
...     seg = Segment(name='segment %d' % ind, index=ind)
...     blk.segments.append(seg)
...
>>> for ind in range(2):
...     chx = ChannelIndex(name='Array probe %d' % ind,
...                        index=np.arange(64))
...     blk.channel_indexes.append(chx)
...
>>> # Populate the Block with AnalogSignal objects
... for seg in blk.segments:
...     for chx in blk.channel_indexes:
...         a = AnalogSignal(np.random.randn(10000, 64)*nA,
...                          sampling_rate=10*kHz)
```

```
...         chx.analogsignals.append(a)
...         seg.analogsignals.append(a)
```

***Required attributes/properties*:** None

***Recommended attributes/properties*:**

name (str) A label for the dataset.

description (str) Text description.

file_origin (str) Filesystem path or URL of the original data file.

file_datetime (datetime) The creation date and time of the original data file.

rec_datetime (datetime) The date and time of the original recording.

***Properties available on this object*:**

list_units descends through hierarchy and returns a list of *Unit* objects existing in the block. This shortcut exists because a common analysis case is analyzing all neurons that you recorded in a session.

Note: Any other additional arguments are assumed to be user-specific metadata and stored in `annotations`.

***Container of*:** *Segment ChannelIndex*

class neo.core.**Segment**(*name=None*, *description=None*, *file_origin=None*, *file_datetime=None*, *rec_datetime=None*, *index=None*, ***annotations*)

A container for data sharing a common time basis.

A *Segment* is a heterogeneous container for discrete or continous data sharing a common clock (time basis) but not necessary the same sampling rate, start or end time.

***Usage*::**

```
>>> from neo.core import Segment, SpikeTrain, AnalogSignal
>>> from quantities import Hz, s
>>>
>>> seg = Segment(index=5)
>>>
>>> train0 = SpikeTrain(times=[.01, 3.3, 9.3], units='sec', t_stop=10)
>>> seg.spiketrains.append(train0)
>>>
>>> train1 = SpikeTrain(times=[100.01, 103.3, 109.3], units='sec',
...                     t_stop=110)
>>> seg.spiketrains.append(train1)
>>>
>>> sig0 = AnalogSignal(signal=[.01, 3.3, 9.3], units='uV',
...                     sampling_rate=1*Hz)
>>> seg.analogsignals.append(sig0)
>>>
>>> sig1 = AnalogSignal(signal=[100.01, 103.3, 109.3], units='nA',
...                     sampling_period=.1*s)
>>> seg.analogsignals.append(sig1)
```

***Required attributes/properties*:** None

***Recommended attributes/properties*:**

name (str) A label for the dataset.

description (str) Text description.

file_origin (str) Filesystem path or URL of the original data file.

file_datetime (datetime) The creation date and time of the original data file.

rec_datetime (datetime) The date and time of the original recording

index (int) You can use this to define a temporal ordering of your Segment. For instance you could use this for trial numbers.

Note: Any other additional arguments are assumed to be user-specific metadata and stored in `annotations`.

***Properties available on this object*:**

all_data (list) A list of all child objects in the *Segment*.

***Container of*:** *Epoch Event AnalogSignal IrregularlySampledSignal SpikeTrain*

class neo.core.**ChannelIndex**(*index*, *channel_names=None*, *channel_ids=None*, *name=None*, *description=None*, *file_origin=None*, *coordinates=None*, ***annotations*)

A container for indexing/grouping data channels.

This container has several purposes:

- Grouping all *AnalogSignal*s and *IrregularlySampledSignal*s inside a *Block* across *Segment*s;
- Indexing a subset of the channels within an *AnalogSignal* and *IrregularlySampledSignal*s;
- Container of *Unit*s. Discharges of multiple neurons (*Unit*'s) can be seen on the same channel.

***Usage 1* providing channel IDs across multiple *Segment*::**

- Recording with 2 electrode arrays across 3 segments
- Each array has 64 channels and is data is represented in a single *AnalogSignal* object per electrode array
- channel ids range from 0 to 127 with the first half covering electrode 0 and second half covering electrode 1

```
>>> from neo.core import (Block, Segment, ChannelIndex,
...                       AnalogSignal)
>>> from quantities import nA, kHz
>>> import numpy as np
...
>>> # create a Block with 3 Segment and 2 ChannelIndex objects
>>> blk = Block()
>>> for ind in range(3):
...     seg = Segment(name='segment %d' % ind, index=ind)
...     blk.segments.append(seg)
...
>>> for ind in range(2):
...     channel_ids=np.arange(64)+ind
...     chx = ChannelIndex(name='Array probe %d' % ind,
...                        index=np.arange(64),
...                        channel_ids=channel_ids,
...                        channel_names=['Channel %i' % chid
...                                       for chid in channel_ids])
...     blk.channel_indexes.append(chx)
...
>>> # Populate the Block with AnalogSignal objects
>>> for seg in blk.segments:
...     for chx in blk.channel_indexes:
```

```
...         a = AnalogSignal(np.random.randn(10000, 64)*nA,
...                          sampling_rate=10*kHz)
...         # link AnalogSignal and ID providing channel_index
...         a.channel_index = chx
...         chx.analogsignals.append(a)
...         seg.analogsignals.append(a)
```

Usage 2 **grouping channels::**

- Recording with a single probe with 8 channels, 4 of which belong to a Tetrode
- Global channel IDs range from 0 to 8
- An additional ChannelIndex is used to group subset of Tetrode channels

```
>>> from neo.core import Block, ChannelIndex
>>> import numpy as np
>>> from quantities import mV, kHz
...
>>> # Create a Block
>>> blk = Block()
>>> blk.segments.append(Segment())
...
>>> # Create a signal with 8 channels and a ChannelIndex handling the
>>> # channel IDs (see usage case 1)
>>> sig = AnalogSignal(np.random.randn(1000, 8)*mV, sampling_rate=10*kHz)
>>> chx = ChannelIndex(name='Probe 0', index=range(8),
...                    channel_ids=range(8),
...                    channel_names=['Channel %i' % chid
...                                   for chid in range(8)])
>>> chx.analogsignals.append(sig)
>>> sig.channel_index=chx
>>> blk.segments[0].analogsignals.append(sig)
...
>>> # Create a new ChannelIndex which groups four channels from the
>>> # analogsignal and provides a second ID scheme
>>> chx = ChannelIndex(name='Tetrode 0',
...                    channel_names=np.array(['Tetrode ch1',
...                                            'Tetrode ch4',
...                                            'Tetrode ch6',
...                                            'Tetrode ch7']),
...                    index=np.array([0, 3, 5, 6]))
>>> # Attach the ChannelIndex to the the Block,
>>> # but not the to the AnalogSignal, since sig.channel_index is
>>> # already linked to the global ChannelIndex of Probe 0 created above
>>> chx.analogsignals.append(sig)
>>> blk.channel_indexes.append(chx)
```

Usage 3 **dealing with *`Unit`* objects::**

- Group 5 unit objects in a single *`ChannelIndex`* object

```
>>> from neo.core import Block, ChannelIndex, Unit
...
>>> # Create a Block
>>> blk = Block()
...
>>> # Create a new ChannelIndex and add it to the Block
>>> chx = ChannelIndex(index=None, name='octotrode A')
```

```
>>> blk.channel_indexes.append(chx)
...
>>> # create several Unit objects and add them to the
>>> # ChannelIndex
>>> for ind in range(5):
...     unit = Unit(name = 'unit %d' % ind,
...                 description='after a long and hard spike sorting')
...     chx.units.append(unit)
```

***Required attributes/properties*:**

index (numpy.array 1D dtype='i') Index of each channel in the attached signals (AnalogSignals and IrregularlySampledSignals). The order of the channel IDs needs to be consistent across attached signals.

***Recommended attributes/properties*:**

name (str) A label for the dataset.

description (str) Text description.

file_origin (str) Filesystem path or URL of the original data file.

channel_names (numpy.array 1D dtype='S') Names for each recording channel.

channel_ids (numpy.array 1D dtype='int') IDs of the corresponding channels referenced by 'index'.

coordinates (quantity array 2D (x, y, z)) Physical or logical coordinates of all channels.

Note: Any other additional arguments are assumed to be user-specific metadata and stored in `annotations`.

***Container of*:** *AnalogSignal IrregularlySampledSignal Unit*

class neo.core.**Unit**(*name=None*, *description=None*, *file_origin=None*, ***annotations*)

A container of *SpikeTrain* objects from a unit.

A *Unit* regroups all the *SpikeTrain* objects that were emitted by a single spike source during a *Block*. A spike source is often a single neuron but doesn't have to be. The spikes may come from different *Segment* objects within the *Block*, so this object is not contained in the usual *Block*/ *Segment*/*SpikeTrain* hierarchy.

A *Unit* is linked to *ChannelIndex* objects from which it was detected. With tetrodes, for instance, multiple channels may record the same *Unit*.

Usage:

```
>>> from neo.core import Unit, SpikeTrain
>>>
>>> unit = Unit(name='pyramidal neuron')
>>>
>>> train0 = SpikeTrain(times=[.01, 3.3, 9.3], units='sec', t_stop=10)
>>> unit.spiketrains.append(train0)
>>>
>>> train1 = SpikeTrain(times=[100.01, 103.3, 109.3], units='sec',
...                     t_stop=110)
>>> unit.spiketrains.append(train1)
```

***Required attributes/properties*:** None

***Recommended attributes/properties*:**

name (str) A label for the dataset.

description (str) Text description.

file_origin (str) Filesystem path or URL of the original data file.

Note: Any other additional arguments are assumed to be user-specific metadata and stored in `annotations`.

***Container of*:** *SpikeTrain*

class neo.core.**AnalogSignal**(*signal*, *units=None*, *dtype=None*, *copy=True*, *t_start=array(0.) * s*, *sampling_rate=None*, *sampling_period=None*, *name=None*, *file_origin=None*, *description=None*, ***annotations*)

Array of one or more continuous analog signals.

A representation of several continuous, analog signals that have the same duration, sampling rate and start time. Basically, it is a 2D array: dim 0 is time, dim 1 is channel index

Inherits from `quantities.Quantity`, which in turn inherits from `numpy.ndarray`.

Usage:

```
>>> from neo.core import AnalogSignal
>>> import quantities as pq
>>>
>>> sigarr = AnalogSignal([[1, 2, 3], [4, 5, 6]], units='V',
...                       sampling_rate=1*pq.Hz)
>>>
>>> sigarr
<AnalogSignal(array([[1, 2, 3],
      [4, 5, 6]]) * mV, [0.0 s, 2.0 s], sampling rate: 1.0 Hz)>
>>> sigarr[:,1]
<AnalogSignal(array([2, 5]) * V, [0.0 s, 2.0 s],
    sampling rate: 1.0 Hz)>
>>> sigarr[1, 1]
array(5) * V
```

***Required attributes/properties*:**

signal (quantity array 2D, numpy array 2D, or list (data, channel)) The data itself.

units (quantity units) Required if the signal is a list or NumPy array, not if it is a `Quantity`

t_start (quantity scalar) Time when signal begins

sampling_rate *or* **sampling_period** (quantity scalar) Number of samples per unit time or interval between two samples. If both are specified, they are checked for consistency.

***Recommended attributes/properties*:**

name (str) A label for the dataset.

description (str) Text description.

file_origin (str) Filesystem path or URL of the original data file.

***Optional attributes/properties*:**

dtype (numpy dtype or str) Override the dtype of the signal array.

copy (bool) True by default.

Note: Any other additional arguments are assumed to be user-specific metadata and stored in `annotations`.

***Properties available on this object*:**

sampling_rate (quantity scalar) Number of samples per unit time. (1/`sampling_period`)

sampling_period (quantity scalar) Interval between two samples. (1/`quantity scalar`)

duration (Quantity) Signal duration, read-only. (size * `sampling_period`)

t_stop (quantity scalar) Time when signal ends, read-only. (`t_start` + `duration`)

times (quantity 1D) The time points of each sample of the signal, read-only. (`t_start` + arange(`shape`[0])/:attr:`sampling_rate`)

channel_index access to the channel_index attribute of the principal ChannelIndex associated with this signal.

***Slicing*:** *AnalogSignal* objects can be sliced. When taking a single column (dimension 0, e.g. [0, :]) or a single element, a `Quantity` is returned. Otherwise an *AnalogSignal* (actually a view) is returned, with the same metadata, except that `t_start` is changed if the start index along dimension 1 is greater than 1. Note that slicing an *AnalogSignal* may give a different result to slicing the underlying NumPy array since signals are always two-dimensional.

***Operations available on this object*:** == != + * /

class `neo.core.`**IrregularlySampledSignal**(*times*, *signal*, *units=None*, *time_units=None*, *dtype=None*, *copy=True*, *name=None*, *file_origin=None*, *description=None*, ***annotations*)

An array of one or more analog signals with samples taken at arbitrary time points.

A representation of one or more continuous, analog signals acquired at time `t_start` with a varying sampling interval. Each channel is sampled at the same time points.

Inherits from `quantities.Quantity`, which in turn inherits from `numpy.ndarray`.

Usage:

```
>>> from neo.core import IrregularlySampledSignal
>>> from quantities import s, nA
>>>
>>> irsig0 = IrregularlySampledSignal([0.0, 1.23, 6.78], [1, 2, 3],
...                                   units='mV', time_units='ms')
>>> irsig1 = IrregularlySampledSignal([0.01, 0.03, 0.12]*s,
...                                   [[4, 5], [5, 4], [6, 3]]*nA)
```

***Required attributes/properties*:**

times (quantity array 1D, numpy array 1D, or list) The time of each data point. Must have the same size as `signal`.

signal (quantity array 2D, numpy array 2D, or list (data, channel)) The data itself.

units (quantity units) Required if the signal is a list or NumPy array, not if it is a `Quantity`.

time_units (quantity units) Required if `times` is a list or NumPy array, not if it is a `Quantity`.

***Recommended attributes/properties*:.**

name (str) A label for the dataset

description (str) Text description.

file_origin (str) Filesystem path or URL of the original data file.

***Optional attributes/properties*:**

dtype (numpy dtype or str) Override the dtype of the signal array. (times are always floats).

copy (bool) True by default.

Note: Any other additional arguments are assumed to be user-specific metadata and stored in `annotations`.

***Properties available on this object*:**

sampling_intervals (quantity array 1D) Interval between each adjacent pair of samples. (`times[1:] - times[:-1]`)

duration (quantity scalar) Signal duration, read-only. (`times[-1] - times[0]`)

t_start (quantity scalar) Time when signal begins, read-only. (`times[0]`)

t_stop (quantity scalar) Time when signal ends, read-only. (`times[-1]`)

***Slicing*:** *IrregularlySampledSignal* objects can be sliced. When this occurs, a new *IrregularlySampledSignal* (actually a view) is returned, with the same metadata, except that `times` is also sliced in the same way.

***Operations available on this object*:** == != + * /

class neo.core.**Event**(*times=None*, *labels=None*, *units=None*, *name=None*, *description=None*, *file_origin=None*, ***annotations*)

Array of events.

Usage:

```
>>> from neo.core import Event
>>> from quantities import s
>>> import numpy as np
>>>
>>> evt = Event(np.arange(0, 30, 10)*s,
...             labels=np.array(['trig0', 'trig1', 'trig2'],
...                             dtype='S'))
>>>
>>> evt.times
array([  0.,  10.,  20.]) * s
>>> evt.labels
array(['trig0', 'trig1', 'trig2'],
      dtype='|S5')
```

***Required attributes/properties*:**

times (quantity array 1D) The time of the events.

labels (numpy.array 1D dtype='S') Names or labels for the events.

***Recommended attributes/properties*:**

name (str) A label for the dataset.

description (str) Text description.

file_origin (str) Filesystem path or URL of the original data file.

Note: Any other additional arguments are assumed to be user-specific metadata and stored in `annotations`.

class neo.core.**Epoch**(*times=None*, *durations=None*, *labels=None*, *units=None*, *name=None*, *description=None*, *file_origin=None*, ***annotations*)

Array of epochs.

Usage:

```
>>> from neo.core import Epoch
>>> from quantities import s, ms
>>> import numpy as np
>>>
>>> epc = Epoch(times=np.arange(0, 30, 10)*s,
...             durations=[10, 5, 7]*ms,
...             labels=np.array(['btn0', 'btn1', 'btn2'], dtype='S'))
>>>
>>> epc.times
array([  0.,  10.,  20.]) * s
>>> epc.durations
array([ 10.,   5.,   7.]) * ms
>>> epc.labels
array(['btn0', 'btn1', 'btn2'],
      dtype='|S4')
```

***Required attributes/properties*:**

times (quantity array 1D) The starts of the time periods.

durations (quantity array 1D) The length of the time period.

labels (numpy.array 1D dtype='S') Names or labels for the time periods.

***Recommended attributes/properties*:**

name (str) A label for the dataset,

description (str) Text description,

file_origin (str) Filesystem path or URL of the original data file.

Note: Any other additional arguments are assumed to be user-specific metadata and stored in `annotations`,

class neo.core.**SpikeTrain**(*times*, *t_stop*, *units=None*, *dtype=<type 'float'>*, *copy=True*, *sampling_rate=array(1.) * Hz*, *t_start=array(0.) * s*, *waveforms=None*, *left_sweep=None*, *name=None*, *file_origin=None*, *description=None*, ***annotations*)

SpikeTrain is a `Quantity` array of spike times.

It is an ensemble of action potentials (spikes) emitted by the same unit in a period of time.

Usage:

```
>>> from neo.core import SpikeTrain
>>> from quantities import s
>>>
>>> train = SpikeTrain([3, 4, 5]*s, t_stop=10.0)
>>> train2 = train[1:3]
>>>
>>> train.t_start
array(0.0) * s
>>> train.t_stop
array(10.0) * s
>>> train
<SpikeTrain(array([ 3.,  4.,  5.]) * s, [0.0 s, 10.0 s])>
>>> train2
<SpikeTrain(array([ 4.,  5.]) * s, [0.0 s, 10.0 s])>
```

***Required attributes/properties*:**

times (quantity array 1D, numpy array 1D, or list) The times of each spike.

units (quantity units) Required if `times` is a list or `ndarray`, not if it is a `Quantity`.

t_stop (quantity scalar, numpy scalar, or float) Time at which *SpikeTrain* ended. This will be converted to the same units as `times`. This argument is required because it specifies the period of time over which spikes could have occurred. Note that `t_start` is highly recommended for the same reason.

Note: If `times` contains values outside of the range [t_start, t_stop], an Exception is raised.

***Recommended attributes/properties*:**

name (str) A label for the dataset.

description (str) Text description.

file_origin (str) Filesystem path or URL of the original data file.

t_start (quantity scalar, numpy scalar, or float) Time at which *SpikeTrain* began. This will be converted to the same units as `times`. Default: 0.0 seconds.

waveforms (quantity array 3D (spike, channel_index, time)) The waveforms of each spike.

sampling_rate (quantity scalar) Number of samples per unit time for the waveforms.

left_sweep (quantity array 1D) Time from the beginning of the waveform to the trigger time of the spike.

sort (bool) If True, the spike train will be sorted by time.

***Optional attributes/properties*:**

dtype (numpy dtype or str) Override the dtype of the signal array.

copy (bool) Whether to copy the times array. True by default. Must be True when you request a change of units or dtype.

Note: Any other additional arguments are assumed to be user-specific metadata and stored in `annotations`.

***Properties available on this object*:**

sampling_period (quantity scalar) Interval between two samples. (1/`sampling_rate`)

duration (quantity scalar) Duration over which spikes can occur, read-only. (`t_stop` - `t_start`)

spike_duration (quantity scalar) Duration of a waveform, read-only. (`waveform`.shape[2] * `sampling_period`)

right_sweep (quantity scalar) Time from the trigger times of the spikes to the end of the waveforms, read-only. (`left_sweep` + `spike_duration`)

times (quantity array 1D) Returns the *SpikeTrain* as a quantity array.

***Slicing*:** *SpikeTrain* objects can be sliced. When this occurs, a new *SpikeTrain* (actually a view) is returned, with the same metadata, except that `waveforms` is also sliced in the same way (along dimension 0). Note that t_start and t_stop are not changed automatically, although you can still manually change them.

CHAPTER 8

Release notes

8.1 Neo 0.6.0 release notes

23rd March 2018

Major changes:

- Introduced `neo.rawio`: a low-level reader for various data formats
- Added continuous integration for all IOs using CircleCI (previously only *`neo.core`* was tested, using Travis CI)
- Moved the test file repository to https://web.gin.g-node.org/NeuralEnsemble/ephy_testing_data - this makes it easier for people to contribute new files for testing.

Other important changes:

- Added `time_index()` and `splice()` methods to `AnalogSignal`
- IO fixes and improvements: Blackrock, TDT, Axon, Spike2, Brainvision, Neuralynx
- Implemented *__deepcopy__* for all data classes
- New IO: BCI2000
- Lots of PEP8 fixes!
- Implemented *__getitem__* for `Epoch`
- Removed "cascade" support from all IOs
- Removed lazy loading except for IOs based on rawio
- Marked lazy option as deprecated
- Added `time_slice()` in read_segment() for IOs based on rawio
- Made `SpikeTrain.times` return a `Quantity` instead of a `SpikeTrain`
- Raise a `ValueError` if `t_stop` is earlier than `t_start` when creating an empty `SpikeTrain`

- Changed filter behaviour to return all objects if no filter parameters are specified
- Fix pickling/unpickling of `Events`

Deprecated IO classes:

- `KlustaKwikIO` (use `KwikIO` instead)
- `PyNNTextIO`, `PyNNNumpyIO`

(Full list of closed issues)

Thanks to Björn Müller, Andrew Davison, Achilleas Koutsou, Chadwick Boulay, Julia Sprenger, Matthieu Senoville, Michael Denker and especially Samuel Garcia for their contributions to this release.

Note: version 0.6.1 was released immediately following 0.6.0 to fix a minor problem with the documentation.

8.2 Neo 0.5.2 release notes

27th September 2017

- Removed support for Python 2.6
- Pickling `AnalogSignal` and `SpikeTrain` now preserves parent objects
- Added NSDFIO, which reads and writes NSDF files
- Fixes and improvements to PlexonIO, NixIO, BlackrockIO, NeuralynxIO, IgorIO, ElanIO, MicromedIO, TdtIO and others.

Thanks to Michael Denker, Achilleas Koutsou, Mieszko Grodzicki, Samuel Garcia, Julia Sprenger, Andrew Davison, Rohan Shah, Richard C Gerkin, Mieszko Grodzicki, Mikkel Elle Lepperød, Joffrey Gonin, Hélissande Fragnaud, Elodie Legouée and Matthieu Sénoville for their contributions to this release.

(Full list of closed issues)

8.3 Neo 0.5.1 release notes

4th May 2017

- Fixes to `AxonIO` (thanks to @erikli and @cjfraz) and `NeuroExplorerIO` (thanks to Mark Hollenbeck)
- Fixes to pickling of `Epoch` and `Event` objects (thanks to Hélissande Fragnaud)
- Added methods `as_array()` and `as_quantity()` to Neo data objects to simplify the common tasks of turning a Neo data object back into a plain Numpy array
- Added `NeuralynxIO`, which reads standard Neuralynx output files in ncs, nev, nse and ntt format (thanks to Julia Sprenger and Carlos Canova).
- Added the `extras_require` field to setup.py, to clearly document the requirements for different io modules. For example, this allows you to run **`pip install neo[neomatlabio]`** and have the extra dependency needed for the `neomatlabio` module (scipy in this case) be automatically installed.
- Fixed a bug where slicing an `AnalogSignal` did not modify the linked `ChannelIndex`.

(Full list of closed issues)

8.4 Neo 0.5.0 release notes

22nd March 2017

For Neo 0.5, we have taken the opportunity to simplify the Neo object model.

Although this will require an initial time investment for anyone who has written code with an earlier version of Neo, the benefits will be greater simplicity, both in your own code and within the Neo code base, which should allow us to move more quickly in fixing bugs, improving performance and adding new features.

More detail on these changes follows:

8.4.1 Merging of "single-value" and "array" versions of data classes

In previous versions of Neo, we had `AnalogSignal` for one-dimensional (single channel) signals, and `AnalogSignalArray` for two-dimensional (multi-channel) signals. In Neo 0.5.0, these have been merged under the name `AnalogSignal`. `AnalogSignal` has the same behaviour as the old `AnalogSignalArray`.

It is still possible to create an `AnalogSignal` from a one-dimensional array, but this will be converted to an array with shape *(n, 1)*, e.g.:

```
>>> signal = neo.AnalogSignal([0.0, 0.1, 0.2, 0.5, 0.6, 0.5, 0.4, 0.3, 0.0],
...                           sampling_rate=10*kHz,
...                           units=nA)
>>> signal.shape
(9, 1)
```

Multi-channel arrays are created as before, but using `AnalogSignal` instead of `AnalogSignalArray`:

```
>>> signal = neo.AnalogSignal([[0.0, 0.1, 0.2, 0.5, 0.6, 0.5, 0.4, 0.3, 0.0],
...                            [0.0, 0.2, 0.4, 0.7, 0.9, 0.8, 0.7, 0.6, 0.3]],
...                           sampling_rate=10*kHz,
...                           units=nA)
>>> signal.shape
(9, 2)
```

Similarly, the `Epoch` and `EpochArray` classes have been merged into an array-valued class `Epoch`, ditto for `Event` and `EventArray`, and the `Spike` class, whose main function was to contain the waveform data for an individual spike, has been suppressed; waveform data are now available as the `waveforms` attribute of the `SpikeTrain` class.

8.4.2 Recording channels

As a consequence of the removal of "single-value" data classes, information on recording channels and the relationship between analog signals and spike trains is also stored differently.

In Neo 0.5, we have introduced a new class, `ChannelIndex`, which replaces both `RecordingChannel` and `RecordingChannelGroup`.

In older versions of Neo, a `RecordingChannel` object held metadata about a logical recording channel (a name and/or integer index) together with references to one or more `AnalogSignal`s recorded on that channel at different points in time (different `Segment`s); redundantly, the `AnalogSignal` also had a `channel_index` attribute, which could be used in addition to or instead of creating a `RecordingChannel`.

Metadata about `AnalogSignalArrays` could be contained in a `RecordingChannelGroup` in a similar way, i.e. `RecordingChannelGroup` functioned as an array-valued version of `RecordingChannel`, but `RecordingChannelGroup` could also be used to group together individual `RecordingChannel` objects.

With Neo 0.5, information about the channel names and ids of an `AnalogSignal` is contained in a `ChannelIndex`, e.g.:

```
>>> signal = neo.AnalogSignal([[0.0, 0.1, 0.2, 0.5, 0.6, 0.5, 0.4, 0.3, 0.0],
...                            [0.0, 0.2, 0.4, 0.7, 0.9, 0.8, 0.7, 0.6, 0.3]],
...                            [0.0, 0.1, 0.3, 0.6, 0.8, 0.7, 0.6, 0.5, 0.3]],
...                           sampling_rate=10*kHz,
...                           units=nA)
>>> channels = neo.ChannelIndex(index=[0, 1, 2],
...                             channel_names=["chan1", "chan2", "chan3"])
>>> signal.channel_index = channels
```

In this use, it replaces `RecordingChannel`.

`ChannelIndex` may also be used to group together a subset of the channels of a multi-channel signal, for example:

```
>>> channel_group = neo.ChannelIndex(index=[0, 2])
>>> channel_group.analogsignals.append(signal)
>>> unit = neo.Unit()  # will contain the spike train recorded from channels 0 and 2.
>>> unit.channel_index = channel_group
```

8.4.3 Checklist for updating code from 0.3/0.4 to 0.5

To update your code from Neo 0.3/0.4 to 0.5, run through the following checklist:

1. Change all usages of `AnalogSignalArray` to `AnalogSignal`.
2. Change all usages of `EpochArray` to `Epoch`.
3. Change all usages of `EventArray` to `Event`.
4. Where you have a list of (single channel) `AnalogSignals` all of the same length, consider converting them to a single, multi-channel `AnalogSignal`.
5. Replace `RecordingChannel` and `RecordingChannelGroup` with `ChannelIndex`.

Note: in points 1-3, the data structure is still an array, it just has a shorter name.

8.4.4 Other changes

- added `NixIO` (about the NIX format)
- added `IgorIO`
- added `NestIO` (for data files produced by the NEST simulator)
- `NeoHdf5IO` is now read-only. It will read data files produced by earlier versions of Neo, but another HDF5-based IO, e.g. `NixIO`, should be used for writing data.
- many fixes/improvements to existing IO modules. All IO modules should now work with Python 3.

8.5 Version 0.4.0

- added StimfitIO

- added KwikIO
- significant improvements to AxonIO, BlackrockIO, BrainwareSrcIO, NeuroshareIO, PlexonIO, Spike2IO, TdtIO,
- many test suite improvements
- Container base class

8.6 Version 0.3.3

- fix a bug in PlexonIO where some EventArrays only load 1 element.
- fix a bug in BrainwareSrcIo for segments with no spikes.

8.7 Version 0.3.2

- cleanup of io test code, with additional helper functions and methods
- added BrainwareDamIo
- added BrainwareF32Io
- added BrainwareSrcIo

8.8 Version 0.3.1

- lazy/cascading improvement
- load_lazy_olbject() in neo.io added
- added NeuroscopeIO

8.9 Version 0.3.0

- various bug fixes in neo.io
- added ElphyIO
- SpikeTrain performence improved
- An IO class now can return a list of Block (see read_all_blocks in IOs)
- python3 compatibility improved

8.10 Version 0.2.1

- assorted bug fixes
- added `time_slice()` method to the `SpikeTrain` and `AnalogSignalArray` classes.
- improvements to annotation data type handling
- added PickleIO, allowing saving Neo objects in the Python pickle format.

- added ElphyIO (see http://www.unic.cnrs-gif.fr/software.html)
- added BrainVisionIO (see http://www.brainvision.com/)
- improvements to PlexonIO
- added `merge()` method to the `Block` and `Segment` classes
- development was mostly moved to GitHub, although the issue tracker is still at neuralensemble.org/neo

8.11 Version 0.2.0

New features compared to neo 0.1:

- new schema more consistent.
- new objects: RecordingChannelGroup, EventArray, AnalogSignalArray, EpochArray
- Neuron is now Unit
- use the quantities module for everything that can have units.
- Some objects directly inherit from Quantity: SpikeTrain, AnalogSignal, AnalogSignalArray, instead of having an attribute for data.
- Attributes are classifyed in 3 categories: necessary, recommended, free.
- lazy and cascade keywords are added to all IOs
- Python 3 support
- better tests

CHAPTER 9

Developers' guide

These instructions are for developing on a Unix-like platform, e.g. Linux or Mac OS X, with the bash shell. If you develop on Windows, please get in touch.

9.1 Mailing lists

General discussion of Neo development takes place in the NeuralEnsemble Google group.

Discussion of issues specific to a particular ticket in the issue tracker should take place on the tracker.

9.2 Using the issue tracker

If you find a bug in Neo, please create a new ticket on the issue tracker, setting the type to "defect". Choose a name that is as specific as possible to the problem you've found, and in the description give as much information as you think is necessary to recreate the problem. The best way to do this is to create the shortest possible Python script that demonstrates the problem, and attach the file to the ticket.

If you have an idea for an improvement to Neo, create a ticket with type "enhancement". If you already have an implementation of the idea, create a patch (see below) and attach it to the ticket.

To keep track of changes to the code and to tickets, you can register for a GitHub account and then set to watch the repository at GitHub Repository (see https://help.github.com/articles/watching-repositories/).

9.3 Requirements

- Python 2.7, 3.4 or later
- numpy >= 1.7.1
- quantities >= 0.9.0

- nose >= 0.11.1 (for running tests)
- Sphinx >= 0.6.4 (for building documentation)
- (optional) tox >= 0.9 (makes it easier to test with multiple Python versions)
- (optional) coverage >= 2.85 (for measuring test coverage)
- (optional) scipy >= 0.12 (for MatlabIO)
- (optional) h5py >= 2.5 (for KwikIO, NeoHdf5IO)

We strongly recommend you develop within a virtual environment (from virtualenv, venv or conda). It is best to have at least one virtual environment with Python 2.7 and one with Python 3.x.

9.4 Getting the source code

We use the Git version control system. The best way to contribute is through GitHub. You will first need a GitHub account, and you should then fork the repository at GitHub Repository (see http://help.github.com/fork-a-repo/).

To get a local copy of the repository:

```
$ cd /some/directory
$ git clone git@github.com:<username>/python-neo.git
```

Now you need to make sure that the `neo` package is on your PYTHONPATH. You can do this either by installing Neo:

```
$ cd python-neo
$ python setup.py install
$ python3 setup.py install
```

(if you do this, you will have to re-run `setup.py install` any time you make changes to the code) *or* by creating symbolic links from somewhere on your PYTHONPATH, for example:

```
$ ln -s python-neo/neo
$ export PYTHONPATH=/some/directory:${PYTHONPATH}
```

An alternate solution is to install Neo with the *develop* option, this avoids reinstalling when there are changes in the code:

```
$ sudo python setup.py develop
```

or using the "-e" option to pip:

```
$ pip install -e python-neo
```

To update to the latest version from the repository:

```
$ git pull
```

9.5 Running the test suite

Before you make any changes, run the test suite to make sure all the tests pass on your system:

```
$ cd neo/test
```

With Python 2.7 or 3.x:

```
$ python -m unittest discover
$ python3 -m unittest discover
```

If you have nose installed:

```
$ nosetests
```

At the end, if you see "OK", then all the tests passed (or were skipped because certain dependencies are not installed), otherwise it will report on tests that failed or produced errors.

To run tests from an individual file:

```
$ python test_analogsignal.py
$ python3 test_analogsignal.py
```

9.6 Writing tests

You should try to write automated tests for any new code that you add. If you have found a bug and want to fix it, first write a test that isolates the bug (and that therefore fails with the existing codebase). Then apply your fix and check that the test now passes.

To see how well the tests cover the code base, run:

```
$ nosetests --with-coverage --cover-package=neo --cover-erase
```

9.7 Working on the documentation

All modules, classes, functions, and methods (including private and subclassed builtin methods) should have docstrings. Please see PEP257 for a description of docstring conventions.

Module docstrings should explain briefly what functions or classes are present. Detailed descriptions can be left for the docstrings of the respective functions or classes. Private functions do not need to be explained here.

Class docstrings should include an explanation of the purpose of the class and, when applicable, how it relates to standard neuroscientific data. They should also include at least one example, which should be written so it can be run as-is from a clean newly-started Python interactive session (that means all imports should be included). Finally, they should include a list of all arguments, attributes, and properties, with explanations. Properties that return data calculated from other data should explain what calculation is done. A list of methods is not needed, since documentation will be generated from the method docstrings.

Method and function docstrings should include an explanation for what the method or function does. If this may not be clear, one or more examples may be included. Examples that are only a few lines do not need to include imports or setup, but more complicated examples should have them.

Examples can be tested easily using the iPython *%doctest_mode* magic. This will strip >>> and ... from the beginning of each line of the example, so the example can be copied and pasted as-is.

The documentation is written in reStructuredText, using the Sphinx documentation system. Any mention of another Neo module, class, attribute, method, or function should be properly marked up so automatic links can be generated. The same goes for quantities or numpy.

To build the documentation:

```
$ cd python-neo/doc
$ make html
```

Then open *some/directory/python-neo/doc/build/html/index.html* in your browser.

9.8 Committing your changes

Once you are happy with your changes, **run the test suite again to check that you have not introduced any new bugs**. It is also recommended to check your code with a code checking program, such as pyflakes or flake8. Then you can commit them to your local repository:

```
$ git commit -m 'informative commit message'
```

If this is your first commit to the project, please add your name and affiliation/employer to `doc/source/authors.rst`

You can then push your changes to your online repository on GitHub:

```
$ git push
```

Once you think your changes are ready to be included in the main Neo repository, open a pull request on GitHub (see https://help.github.com/articles/using-pull-requests).

9.9 Python version

Neo core should work with both Python 2.7 and Python 3 (version 3.4 or newer). Neo IO modules should ideally work with both Python 2 and 3, but certain modules may only work with one or the other (see *Installation*).

So far, we have managed to write code that works with both Python 2 and 3. Mainly this involves avoiding the `print` statement (use `logging.info` instead), and putting `from __future__ import division` at the beginning of any file that uses division.

If in doubt, Porting to Python 3 by Lennart Regebro is an excellent resource.

The most important thing to remember is to run tests with at least one version of Python 2 and at least one version of Python 3. There is generally no problem in having multiple versions of Python installed on your computer at once: e.g., on Ubuntu Python 2 is available as *python* and Python 3 as *python3*, while on Arch Linux Python 2 is *python2* and Python 3 *python*. See PEP394 for more on this. Using virtual environments makes this very straightforward.

9.10 Coding standards and style

All code should conform as much as possible to PEP 8, and should run with Python 2.7, and 3.4 or newer.

You can use the pep8 program to check the code for PEP 8 conformity. You can also use flake8, which combines pep8 and pyflakes.

However, the pep8 and flake8 programs do not check for all PEP 8 issues. In particular, they do not check that the import statements are in the correct order.

Also, please do not use `from xyz import *`. This is slow, can lead to conflicts, and makes it difficult for code analysis software.

9.11 Making a release

Add a section in `/doc/source/whatisnew.rst` for the release.

First check that the version string (in `neo/version.py`) is correct.

To build a source package:

```
$ python setup.py sdist
```

To upload the package to PyPI (currently Samuel Garcia and Andrew Davison have the necessary permissions to do this):

```
$ python setup.py sdist upload
$ python setup.py upload_docs --upload-dir=doc/build/html
```

Finally, tag the release in the Git repository and push it:

```
$ git tag <version>
$ git push --tags origin
```

9.12 If you want to develop your own IO module

See *IO developers' guide* for implementation of a new IO.

CHAPTER 10

IO developers' guide

10.1 Guidelines for IO implementation

There are two ways to add a new IO module:

- By directly adding a new IO class in a module within *neo.io*: the reader/writer will deal directly with Neo objects
- By adding a RawIO class in a module within `neo.rawio`: the reader should work with raw buffers from the file and provide some internal headers for the scale/units/name/... You can then generate an IO module simply by inheriting from your RawIO class and from `neo.io.BaseFromRaw`

For read only classes, we encourage you to write a `RawIO` class because it allows slice reading, and is generally much quicker and easier (although only for reading) than implementing a full IO class. For read/write classes you can mix the two levels neo.rawio for reading and neo.io for writing.

Recipe to develop an IO module for a new data format:

1. Fully understand the object model. See *Neo core*. If in doubt ask the mailing list.
2. Fully understand `neo.io.examplerawio`, It is a fake IO to explain the API. If in doubt ask the list.
3. Copy/paste `examplerawio.py` and choose clear file and class names for your IO.
4. implement all methods that **raise(NotImplementedError)** in `neo.rawio.baserawio`. Return None when the object is not supported (spike/waveform)
5. Write good docstrings. List dependencies, including minimum version numbers.
6. Add your class to `neo.rawio.__init__`. Keep imports inside `try/except` for dependency reasons.
7. Create a class in `neo/io/`
8. Add your class to `neo.io.__init__`. Keep imports inside `try/except` for dependency reasons.
9. Create an account at https://gin.g-node.org and deposit files in `NeuralEnsemble/ephy_testing_data`.

10. Write tests in `neo/rawio/test_xxxxxrawio.py`. You must at least pass the standard tests (inherited from `BaseTestRawIO`). See `test_examplerawio.py`
11. Write a similar test in `neo.tests/iotests/test_xxxxxio.py`. See `test_exampleio.py`
12. Make a pull request when all tests pass.

10.2 Miscellaneous

- If your IO supports several versions of a format (like ABF1, ABF2), upload to the gin.g-node.org test file repository all file versions possible. (for test coverage).
- `neo.core.Block.create_many_to_one_relationship()` offers a utility to complete the hierachy when all one-to-many relationships have been created.
- In the docstring, explain where you obtained the file format specification if it is a closed one.
- If your IO is based on a database mapper, keep in mind that the returned object MUST be detached, because this object can be written to another url for copying.

10.3 Tests

`neo.rawio.tests.common_rawio_test.BaseTestRawIO` and `neo.test.io.commun_io_test.BaseTestIO` provide standard tests. To use these you need to upload some sample data files at gin-gnode. They will be publicly accessible for testing Neo. These tests:

- check the compliance with the schema: hierachy, attribute types, ...
- For IO modules able to both write and read data, it compares a generated dataset with the same data after a write/read cycle.

The test scripts download all files from gin-gnode and stores them locally in `/tmp/files_for_tests/`. Subsequent test runs use the previously downloaded files, rather than trying to download them each time.

Each test must have at least one class that inherits `BaseTestRawIO` and that has 3 attributes:

- `rawioclass`: the class
- `entities_to_test`: a list of files (or directories) to be tested one by one
- `files_to_download`: a list of files to download (sometimes bigger than `entities_to_test`)

Here is an example test script taken from the distribution: `test_axonrawio.py`:

```
# -*- coding: utf-8 -*-

# needed for python 3 compatibility
from __future__ import unicode_literals, print_function, division, absolute_import

import unittest

from neo.rawio.axonrawio import AxonRawIO

from neo.rawio.tests.common_rawio_test import BaseTestRawIO


class TestAxonRawIO(BaseTestRawIO, unittest.TestCase, ):
    rawioclass = AxonRawIO
```

```
    entities_to_test = [
        'File_axon_1.abf',  # V2.0
        'File_axon_2.abf',  # V1.8
        'File_axon_3.abf',  # V1.8
        'File_axon_4.abf',  # 2.0
        'File_axon_5.abf',  # V.20
        'File_axon_6.abf',  # V.20
        'File_axon_7.abf',  # V2.6
    ]
    files_to_download = entities_to_test

    def test_read_raw_protocol(self):
        reader = AxonRawIO(filename=self.get_filename_path('File_axon_7.abf'))
        reader.parse_header()

        reader.read_raw_protocol()


if __name__ == "__main__":
    unittest.main()
```

10.4 Logging

All IO classes by default have logging using the standard `logging` module: already set up. The logger name is the same as the fully qualified class name, e.g. `neo.io.hdf5io.NeoHdf5IO`. The `class.logger` attribute holds the logger for easy access.

There are generally 3 types of situations in which an IO class should use a logger

- Recoverable errors with the file that the users need to be notified about. In this case, please use `logger.warning()` or `logger.error()`. If there is an exception associated with the issue, you can use `logger.exception()` in the exception handler to automatically include a backtrace with the log. By default, all users will see messages at this level, so please restrict it only to problems the user absolutely needs to know about.
- Informational messages that advanced users might want to see in order to get some insight into the file. In this case, please use `logger.info()`.
- Messages useful to developers to fix problems with the io class. In this case, please use `logger.debug()`.

A log handler is automatically added to *neo*, so please do not user your own handler. Please use the `class.logger` attribute for accessing the logger inside the class rather than `logging.getLogger()`. Please do not log directly to the root logger (e.g. `logging.warning()`), use the class's logger instead (`class.logger.warning()`). In the tests for the io class, if you intentionally test broken files, please disable logs by setting the logging level to *100*.

10.5 ExampleIO

class neo.io.**ExampleIO**(*filename=''*)

Here is the entire file:

```
# -*- coding: utf-8 -*-
"""
ExampleRawIO is a class of a  fake example.
This is to be used when coding a new RawIO.
```

```
Rules for creating a new class:
  1. Step 1: Create the main class
    * Create a file in **neo/rawio/** that endith with "rawio.py"
    * Create the class that inherits BaseRawIO
    * copy/paste all methods that need to be implemented.
      See the end a neo.rawio.baserawio.BaseRawIO
    * code hard! The main difficulty **is _parse_header()**.
      In short you have a create a mandatory dict than
      contains channel informations::

            self.header = {}
            self.header['nb_block'] = 2
            self.header['nb_segment'] = [2, 3]
            self.header['signal_channels'] = sig_channels
            self.header['unit_channels'] = unit_channels
            self.header['event_channels'] = event_channels

  2. Step 2: RawIO test:
    * create a file in neo/rawio/tests with the same name with "test_" prefix
    * copy paste neo/rawio/tests/test_examplerawio.py and do the same

  3. Step 3 : Create the neo.io class with the wrapper
    * Create a file in neo/io/ that endith with "io.py"
    * Create a that hinerits bot yrou RawIO class and BaseFromRaw class
    * copy/paste from neo/io/exampleio.py

  4.Step 4 : IO test
    * create a file in neo/test/iotest with the same previous name with "test_" prefix
    * copy/paste from neo/test/iotest/test_exampleio.py



"""
from __future__ import unicode_literals, print_function, division, absolute_import

from .baserawio import (BaseRawIO, _signal_channel_dtype, _unit_channel_dtype,
                        _event_channel_dtype)

import numpy as np


class ExampleRawIO(BaseRawIO):
    """
    Class for "reading" fake data from an imaginary file.

    For the user, it give acces to raw data (signals, event, spikes) as they
    are in the (fake) file int16 and int64.

    For a developer, it is just an example showing guidelines for someone who wants
    to develop a new IO module.

    Two rules for developers:
      * Respect the Neo RawIO API (:ref:`_neo_rawio_API`)
      * Follow :ref:`_io_guiline`

    This fake IO:
        * have 2 blocks
```

```
        * blocks have 2 and 3 segments
        * have 16 signal_channel sample_rate = 10000
        * have 3 unit_channel
        * have 2 event channel: one have *type=event*, the other have
          *type=epoch*


    Usage:
        >>> import neo.rawio
        >>> r = neo.rawio.ExampleRawIO(filename='itisafake.nof')
        >>> r.parse_header()
        >>> print(r)
        >>> raw_chunk = r.get_analogsignal_chunk(block_index=0, seg_index=0,
                                i_start=0, i_stop=1024,  channel_names=channel_names)
        >>> float_chunk = reader.rescale_signal_raw_to_float(raw_chunk, dtype='float64
↪',
                                channel_indexes=[0, 3, 6])
        >>> spike_timestamp = reader.spike_timestamps(unit_index=0, t_start=None, t_
↪stop=None)
        >>> spike_times = reader.rescale_spike_timestamp(spike_timestamp, 'float64')
        >>> ev_timestamps, _, ev_labels = reader.event_timestamps(event_channel_
↪index=0)

    """
    extensions = ['fake']
    rawmode = 'one-file'

    def __init__(self, filename=''):
        BaseRawIO.__init__(self)
        # note that this filename is ued in self._source_name
        self.filename = filename

    def _source_name(self):
        # this function is used by __repr__
        # for general cases self.filename is good
        # But for URL you could mask some part of the URL to keep
        # the main part.
        return self.filename

    def _parse_header(self):
        # This is the central of a RawIO
        # we need to collect in the original format all
        # informations needed for further fast acces
        # at any place in the file
        # In short _parse_header can be slow but
        # _get_analogsignal_chunk need to be as fast as possible

        # create signals channels information
        # This is mandatory!!!!
        # gain/offset/units are really important because
        # the scaling to real value will be done with that
        # at the end real_signal = (raw_signal* gain + offset) * pq.Quantity(units)
        sig_channels = []
        for c in range(16):
            ch_name = 'ch{}'.format(c)
            # our channel id is c+1 just for fun
            # Note that chan_id should be realated to
            # original channel id in the file format
```

```
            # so that the end user should not be lost when reading datasets
            chan_id = c + 1
            sr = 10000.  # Hz
            dtype = 'int16'
            units = 'uV'
            gain = 1000. / 2 ** 16
            offset = 0.
            # group_id isonly for special cases when channel have diferents
            # sampling rate for instance. See TdtIO for that.
            # Here this is the general case :all channel have the same characteritics
            group_id = 0
            sig_channels.append((ch_name, chan_id, sr, dtype, units, gain, offset,␣
↪group_id))
        sig_channels = np.array(sig_channels, dtype=_signal_channel_dtype)

        # creating units channels
        # This is mandatory!!!!
        # Note that if there is no waveform at all in the file
        # then wf_units/wf_gain/wf_offset/wf_left_sweep/wf_sampling_rate
        # can be set to any value because _spike_raw_waveforms
        # will return None
        unit_channels = []
        for c in range(3):
            unit_name = 'unit{}'.format(c)
            unit_id = '#{}'.format(c)
            wf_units = 'uV'
            wf_gain = 1000. / 2 ** 16
            wf_offset = 0.
            wf_left_sweep = 20
            wf_sampling_rate = 10000.
            unit_channels.append((unit_name, unit_id, wf_units, wf_gain,
                                  wf_offset, wf_left_sweep, wf_sampling_rate))
        unit_channels = np.array(unit_channels, dtype=_unit_channel_dtype)

        # creating event/epoch channel
        # This is mandatory!!!!
        # In RawIO epoch and event they are dealt the same way.
        event_channels = []
        event_channels.append(('Some events', 'ev_0', 'event'))
        event_channels.append(('Some epochs', 'ep_1', 'epoch'))
        event_channels = np.array(event_channels, dtype=_event_channel_dtype)

        # fille into header dict
        # This is mandatory!!!!!
        self.header = {}
        self.header['nb_block'] = 2
        self.header['nb_segment'] = [2, 3]
        self.header['signal_channels'] = sig_channels
        self.header['unit_channels'] = unit_channels
        self.header['event_channels'] = event_channels

        # insert some annotation at some place
        # at neo.io level IO are free to add some annoations
        # to any object. To keep this functionality with the wrapper
        # BaseFromRaw you can add annoations in a nested dict.
        self._generate_minimal_annotations()
        # If you are a lazy dev you can stop here.
        for block_index in range(2):
```

```
            bl_ann = self.raw_annotations['blocks'][block_index]
            bl_ann['name'] = 'Block #{}'.format(block_index)
            bl_ann['block_extra_info'] = 'This is the block {}'.format(block_index)
            for seg_index in range([2, 3][block_index]):
                seg_ann = bl_ann['segments'][seg_index]
                seg_ann['name'] = 'Seg #{} Block #{}'.format(
                    seg_index, block_index)
                seg_ann['seg_extra_info'] = 'This is the seg {} of block {}'.format(
                    seg_index, block_index)
                for c in range(16):
                    anasig_an = seg_ann['signals'][c]
                    anasig_an['info'] = 'This is a good signals'
                for c in range(3):
                    spiketrain_an = seg_ann['units'][c]
                    spiketrain_an['quality'] = 'Good!!'
                for c in range(2):
                    event_an = seg_ann['events'][c]
                    if c == 0:
                        event_an['nickname'] = 'Miss Event 0'
                    elif c == 1:
                        event_an['nickname'] = 'MrEpoch 1'

    def _segment_t_start(self, block_index, seg_index):
        # this must return an float scale in second
        # this t_start will be shared by all object in the segment
        # except AnalogSignal
        all_starts = [[0., 15.], [0., 20., 60.]]
        return all_starts[block_index][seg_index]

    def _segment_t_stop(self, block_index, seg_index):
        # this must return an float scale in second
        all_stops = [[10., 25.], [10., 30., 70.]]
        return all_stops[block_index][seg_index]

    def _get_signal_size(self, block_index, seg_index, channel_indexes=None):
        # we are lucky: signals in all segment have the same shape!! (10.0 seconds)
        # it is not always the case
        # this must return an int = the number of sample

        # Note that channel_indexes can be ignored for most cases
        # except for several sampling rate.
        return 100000

    def _get_signal_t_start(self, block_index, seg_index, channel_indexes):
        # This give the t_start of signals.
        # Very often this equal to _segment_t_start but not
        # always.
        # this must return an float scale in second

        # Note that channel_indexes can be ignored for most cases
        # except for several sampling rate.

        # Here this is the same.
        # this is not always the case
        return self._segment_t_start(block_index, seg_index)

    def _get_analogsignal_chunk(self, block_index, seg_index, i_start, i_stop,␣
↪channel_indexes):
```

```
        # this must return a signal chunk limited with
        # i_start/i_stop (can be None)
        # channel_indexes can be None (=all channel) or a list or numpy.array
        # This must return a numpy array 2D (even with one channel).
        # This must return the orignal dtype. No conversion here.
        # This must as fast as possible.
        # Everything that can be done in _parse_header() must not be here.

        # Here we are lucky:  our signals is always zeros!!
        # it is not always the case
        # internally signals are int16
        # convertion to real units is done with self.header['signal_channels']

        if i_start is None:
            i_start = 0
        if i_stop is None:
            i_stop = 100000

        assert i_start >= 0, "I don't like your jokes"
        assert i_stop <= 100000, "I don't like your jokes"

        if channel_indexes is None:
            nb_chan = 16
        else:
            nb_chan = len(channel_indexes)
        raw_signals = np.zeros((i_stop - i_start, nb_chan), dtype='int16')
        return raw_signals

    def _spike_count(self, block_index, seg_index, unit_index):
        # Must return the nb of spike for given (block_index, seg_index, unit_index)
        # we are lucky:  our units have all the same nb of spikes!!
        # it is not always the case
        nb_spikes = 20
        return nb_spikes

    def _get_spike_timestamps(self, block_index, seg_index, unit_index, t_start, t_
↪stop):
        # In our IO, timstamp are internally coded 'int64' and they
        # represent the index of the signals 10kHz
        # we are lucky: spikes have the same discharge in all segments!!
        # incredible neuron!! This is not always the case

        # the same clip t_start/t_start must be used in _spike_raw_waveforms()

        ts_start = (self._segment_t_start(block_index, seg_index) * 10000)

        spike_timestamps = np.arange(0, 10000, 500) + ts_start

        if t_start is not None or t_stop is not None:
            # restricte spikes to given limits (in seconds)
            lim0 = int(t_start * 10000)
            lim1 = int(t_stop * 10000)
            mask = (spike_timestamps >= lim0) & (spike_timestamps <= lim1)
            spike_timestamps = spike_timestamps[mask]

        return spike_timestamps

    def _rescale_spike_timestamp(self, spike_timestamps, dtype):
```

```
        # must rescale to second a particular spike_timestamps
        # with a fixed dtype so the user can choose the precisino he want.
        spike_times = spike_timestamps.astype(dtype)
        spike_times /= 10000.  # because 10kHz
        return spike_times

    def _get_spike_raw_waveforms(self, block_index, seg_index, unit_index, t_start, t_
↪stop):
        # this must return a 3D numpy array (nb_spike, nb_channel, nb_sample)
        # in the original dtype
        # this must be as fast as possible.
        # the same clip t_start/t_start must be used in _spike_timestamps()

        # If there there is no waveform supported in the
        # IO them _spike_raw_waveforms must return None

        # In our IO waveforms come from all channels
        # they are int16
        # convertion to real units is done with self.header['unit_channels']
        # Here, we have a realistic case: all waveforms are only noise.
        # it is not always the case
        # we 20 spikes with a sweep of 50 (5ms)

        np.random.seed(2205)  # a magic number (my birthday)
        waveforms = np.random.randint(low=-2 ** 4, high=2 ** 4, size=20 * 50, dtype=
↪'int16')
        waveforms = waveforms.reshape(20, 1, 50)
        return waveforms

    def _event_count(self, block_index, seg_index, event_channel_index):
        # event and spike are very similar
        # we have 2 event channels
        if event_channel_index == 0:
            # event channel
            return 6
        elif event_channel_index == 1:
            # epoch channel
            return 10

    def _get_event_timestamps(self, block_index, seg_index, event_channel_index, t_
↪start, t_stop):
        # the main difference between spike channel and event channel
        # is that for here we have 3 numpy array timestamp, durations, labels
        # durations must be None for 'event'
        # label must a dtype ='U'

        # in our IO event are directly coded in seconds
        seg_t_start = self._segment_t_start(block_index, seg_index)
        if event_channel_index == 0:
            timestamp = np.arange(0, 6, dtype='float64') + seg_t_start
            durations = None
            labels = np.array(['trigger_a', 'trigger_b'] * 3, dtype='U12')
        elif event_channel_index == 1:
            timestamp = np.arange(0, 10, dtype='float64') + .5 + seg_t_start
            durations = np.ones((10), dtype='float64') * .25
            labels = np.array(['zoneX'] * 5 + ['zoneZ'] * 5, dtype='U12')

        if t_start is not None:
```

```
            keep = timestamp >= t_start
            timestamp, labels = timestamp[keep], labels[keep]
            if durations is not None:
                durations = durations[keep]

        if t_stop is not None:
            keep = timestamp <= t_stop
            timestamp, labels = timestamp[keep], labels[keep]
            if durations is not None:
                durations = durations[keep]

        return timestamp, durations, labels

    def _rescale_event_timestamp(self, event_timestamps, dtype):
        # must rescale to second a particular event_timestamps
        # with a fixed dtype so the user can choose the precisino he want.

        # really easy here because in our case it is already seconds
        event_times = event_timestamps.astype(dtype)
        return event_times

    def _rescale_epoch_duration(self, raw_duration, dtype):
        # really easy here because in our case it is already seconds
        durations = raw_duration.astype(dtype)
        return durations
```

```
# -*- coding: utf-8 -*-
"""
neo.io have been split in 2 level API:
  * neo.io: this API give neo object
  * neo.rawio: this API give raw data as they are in files.

Developper are encourage to use neo.rawio.

When this is done the neo.io is done automagically with
this king of following code.

Author: sgarcia

"""

from neo.io.basefromrawio import BaseFromRaw
from neo.rawio.examplerawio import ExampleRawIO


class ExampleIO(ExampleRawIO, BaseFromRaw):
    name = 'example IO'
    description = "Fake IO"

    # This is an inportant choice when there are several channels.
    #   'split-all' :  1 AnalogSignal each 1 channel
    #   'group-by-same-units' : one 2D AnalogSignal for each group of channel with␣
↪same units
    _prefered_signal_group_mode = 'group-by-same-units'

    def __init__(self, filename=''):
        ExampleRawIO.__init__(self, filename=filename)
        BaseFromRaw.__init__(self, filename)
```

CHAPTER 11

Authors and contributors

The following people have contributed code and/or ideas to the current version of Neo. The institutional affiliations are those at the time of the contribution, and may not be the current affiliation of a contributor.

- Samuel Garcia [1]
- Andrew Davison [2]
- Chris Rodgers [3]
- Pierre Yger [2]
- Yann Mahnoun [4]
- Luc Estabanez [2]
- Andrey Sobolev [5]
- Thierry Brizzi [2]
- Florent Jaillet [6]
- Philipp Rautenberg [5]
- Thomas Wachtler [5]
- Cyril Dejean [7]
- Robert Pröpper [8]
- Domenico Guarino [2]
- Achilleas Koutsou [5]
- Erik Li [9]
- Georg Raiser [10]
- Joffrey Gonin [2]
- Kyler Brown [?]
- Mikkel Elle Lepperød [11]

- C Daniel Meliza [12]
- Julia Sprenger [13]
- Maximilian Schmidt [13]
- Johanna Senk [13]
- Carlos Canova [13]
- Hélissande Fragnaud [2]
- Mark Hollenbeck [14]
- Mieszko Grodzicki
- Rick Gerkin [15]
- Matthieu Sénoville [2]
- Chadwick Boulay [16]
- Björn Müller [13]

1. Centre de Recherche en Neuroscience de Lyon, CNRS UMR5292 - INSERM U1028 - Universite Claude Bernard Lyon 1
2. Unité de Neuroscience, Information et Complexité, CNRS UPR 3293, Gif-sur-Yvette, France
3. University of California, Berkeley
4. Laboratoire de Neurosciences Intégratives et Adaptatives, CNRS UMR 6149 - Université de Provence, Marseille, France
5. G-Node, Ludwig-Maximilians-Universität, Munich, Germany
6. Institut de Neurosciences de la Timone, CNRS UMR 7289 - Université d'Aix-Marseille, Marseille, France
7. Centre de Neurosciences Integratives et Cignitives, UMR 5228 - CNRS - Université Bordeaux I - Université Bordeaux II
8. Neural Information Processing Group, TU Berlin, Germany
9. Department of Neurobiology & Anatomy, Drexel University College of Medicine, Philadelphia, PA, USA
10. University of Konstanz, Konstanz, Germany
11. Centre for Integrative Neuroplasticity (CINPLA), University of Oslo, Norway
12. University of Virginia
13. INM-6, Forschungszentrum Jülich, Germany
14. University of Texas at Austin
15. Arizona State University
16. Ottawa Hospital Research Institute, Canada

If we've somehow missed you off the list we're very sorry - please let us know.

CHAPTER 12

License

Neo is free software, distributed under a 3-clause Revised BSD licence (BSD-3-Clause).

CHAPTER 13

Support

If you have problems installing the software or questions about usage, documentation or anything else related to Neo, you can post to the NeuralEnsemble mailing list. If you find a bug, please create a ticket in our issue tracker.

CHAPTER 14

Contributing

Any feedback is gladly received and highly appreciated! Neo is a community project, and all contributions are welcomed - see the *Developers' guide* for more information. Source code is on GitHub.

CHAPTER 15

Citation

To cite Neo in publications, please use:

> Garcia S., Guarino D., Jaillet F., Jennings T.R., Pröpper R., Rautenberg P.L., Rodgers C., Sobolev A.,Wachtler T., Yger P. and Davison A.P. (2014) Neo: an object model for handling electrophysiology data in multiple formats. Frontiers in Neuroinformatics 8:10: doi:10.3389/fninf.2014.00010

A BibTeX entry for LaTeX users is:

```
@article{neo09,
    author = {Garcia S. and Guarino D. and Jaillet F. and Jennings T.R. and Pröpper R.
↪ and
              Rautenberg P.L. and Rodgers C. and Sobolev A. and Wachtler T. and Yger␣
↪P.
              and Davison A.P.},
    doi = {10.3389/fninf.2014.00010},
    full_text = {http://www.frontiersin.org/Journal/10.3389/fninf.2014.00010/abstract}
↪,
    journal = {Frontiers in Neuroinformatics},
    month = {February},
    title = {Neo: an object model for handling electrophysiology data in multiple␣
↪formats},
    volume = {8:10},
    year = {2014}
}
```

Python Module Index

n

Index