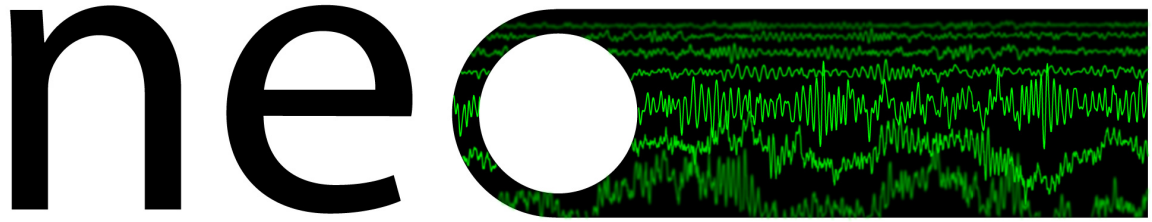
Neo Documentation

Release 0.10.0

Neo authors and contributors <neuralensemble@googlegroups.c

Jul 27, 2021

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 Installing from the Python Package Index

1.1.1 Dependencies

- Python >= 3.7
- numpy >= 1.16.1
- quantities >= 0.12.1

You can install the latest published version of Neo and its dependencies using:

```
$ pip install neo
```

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 >= 1.0.0 for NeoMatlabIO
- h5py >= 2.5 for KwikIO
- klusta for KwikIO
- igor >= 0.2 for IgorIO
- nixio >= 1.5 for NixIO
- stfio for StimfitIO
- pillow for TiffIO

These dependencies can be installed by specifying a comma-separated list with the `pip install` command:

```
$ pip install neo[nixio,tiffio]
```

Or when installing a specific version of neo:

```
$ pip install neo[nixio,tiffio]==0.9.0
```

These additional dependencies for IO modules are available:

```
* igorproio
* kwikio
* neomatlabio
* nixio
* stimfitio
* tiffio
```

To download and install the package manually, download:

https://github.com/NeuralEnsemble/python-neo/archive/neo-0.10.0.zip

Then:

```
$ unzip neo-0.10.0.zip
$ cd neo-0.10.0
$ python setup.py install
```

1.2 Installing from source

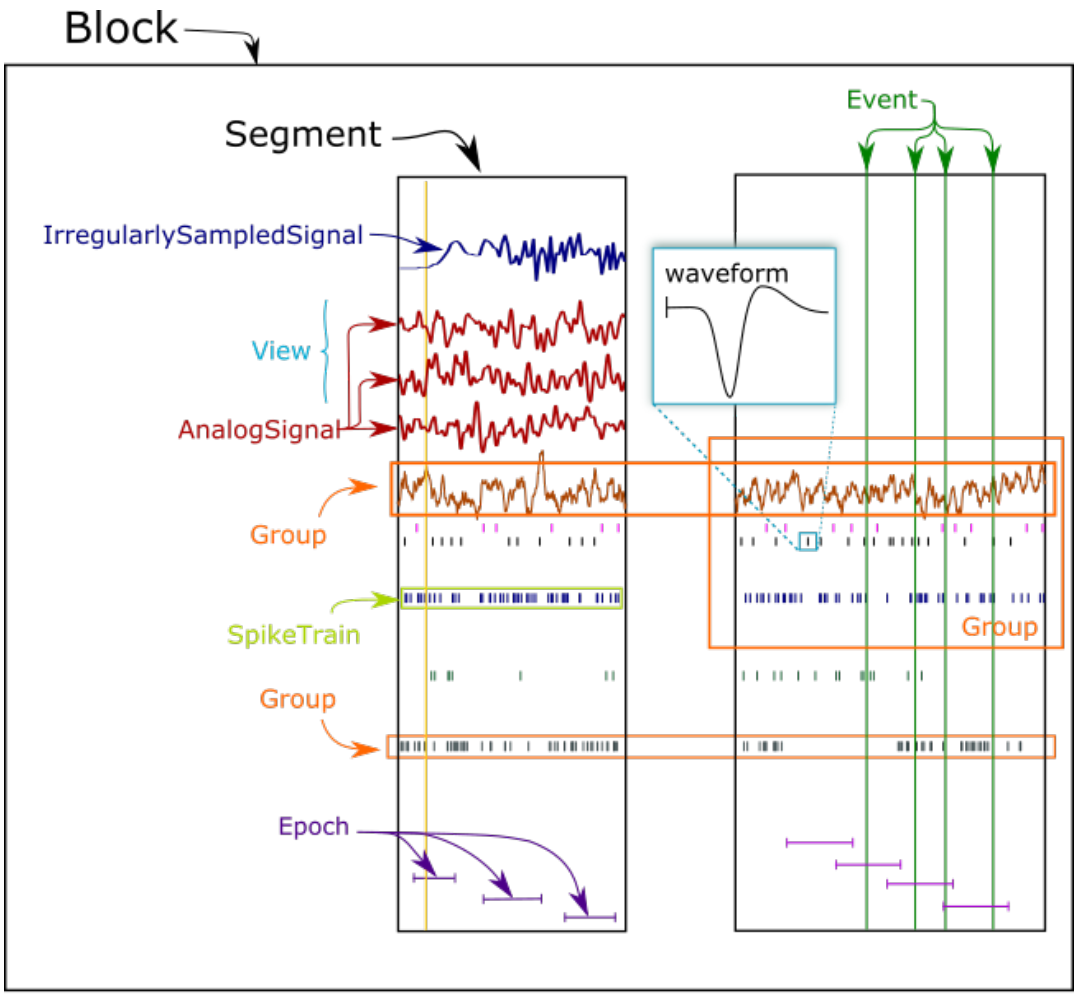
To install the latest version of Neo from the Git repository:

```
$ 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, with the exception of the newly added ImageSequence and RegionOfInterest classes:



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.
- `ImageSequence`: A three dimensional array representing a sequence of images.

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* and *Group* objects.

2.3 Grouping/linking 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.

- *ChannelView*: A set of indices into *AnalogSignal* objects, representing logical and/or physical recording channels. For spike sorting of extracellular signals, where spikes may be recorded on more than one recording channel, the *ChannelView* can be used to reference the group of recording channels from which the spikes were obtained.
- *Group*: Can contain any of the data objects, views, or other groups, outside the hierarchy of the segment and block containers. A common use is to link the *SpikeTrain* objects within a *Block*, possibly across multiple Segments, that were emitted by the same neuron.
- *CircularRegionOfInterest*, *RectangularRegionOfInterest* and *PolygonRegionOfInterest* are three subclasses that link *ImageSequence* objects to signals (*AnalogSignal* objects) extracted from them.

For more details, see grouping.

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.groups`

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

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

Here is an example showing these relationships in use:

```
from neo.io import AxonIO
import urllib.request
url = "https://web.gin.g-node.org/NeuralEnsemble/ephy_testing_data/raw/master/axon/
↪File_axon_3.abf"
filename = './test.abf'
urllib.request.urlretrieve(url, filename)

r = AxonIO(filename=filename)
blocks = r.read() # read the entire file > a list of Blocks
bl = blocks[0]
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, Group
bl = Block()

# the four tetrodes
for i in range(4):
    group = Group(name='Tetrode %d' % i)
    bl.groups.append(group)
```

(continues on next page)

(continued from previous page)

```
# 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 spiketrains for each spike sorting group together with the channels on which that neuron spiked:

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

# one group for each neuron
view0 = ChannelView(recorded_signals, index=[0, 1, 2])
unit0 = Group(view0, name='Group 0')
bl.groups.append(unit0)

view1 = ChannelView(recorded_signals, index=[1, 2, 3])
unit1 = Group(view1, name='Group 1')
bl.groups.append(unit1)

# now we add the spiketrains from Unit 0 to unit0
# and add the spiketrains from Unit 1 to unit1
# ...
```

Now each putative neuron is represented by a *Group* containing the spiktrains of that neuron and a view of the signal selecting only those channels from which the spikes were obtained.

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

2.3.3 Neo diagram

Object:

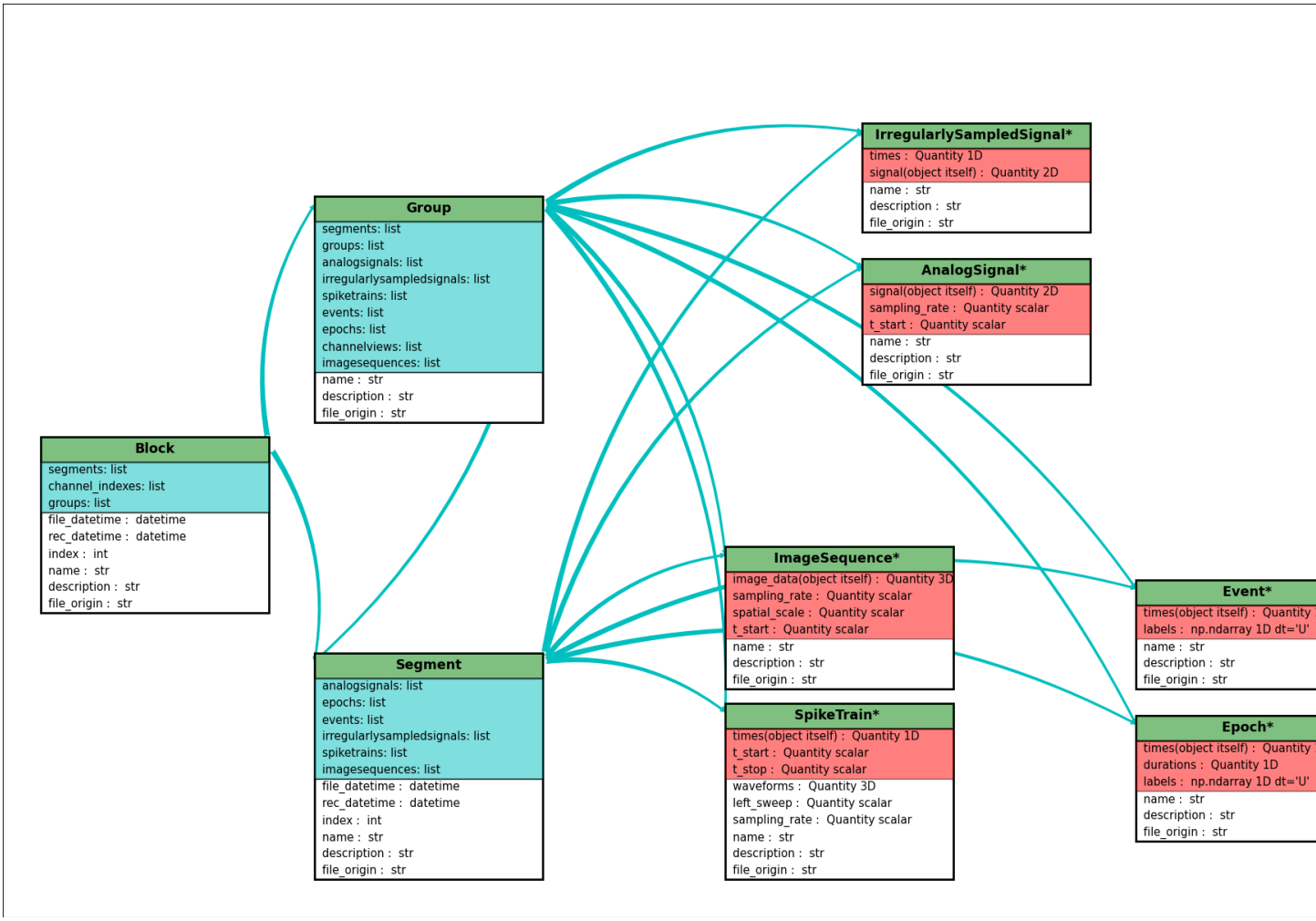
- With a star = inherits from `Quantity`

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`

Note: This figure do not include *`ChannelView`* and `RegionOfInterest`.

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.

2.6 Array Annotations

Next to "regular" annotations there is also a way to annotate arrays of values in order to create annotations with one value per data point. Using this feature, called Array Annotations, the consistency of those annotations with the actual data is ensured. Apart from adding those on object construction, Array Annotations can also be added using the `array_annotate()` method provided by all Neo data objects, e.g.:

```
>>> sptr = SpikeTrain(times=[1, 2, 3]*pq.s, t_stop=3*pq.s)
>>> sptr.array_annotate(index=[0, 1, 2], relevant=[True, False, True])
>>> print(sptr.array_annotations)
{'index': array([0, 1, 2]), 'relevant': array([ True, False,  True])}
```

Since Array Annotations may be written to a file or database, there are some limitations on the data types of arrays: they must be 1-dimensional (i.e. not nested) and contain the same types as annotations:

`integer`, `float`, `complex`, `Quantity`, `string`, `date`, `time` and `datetime`.

CHAPTER 3

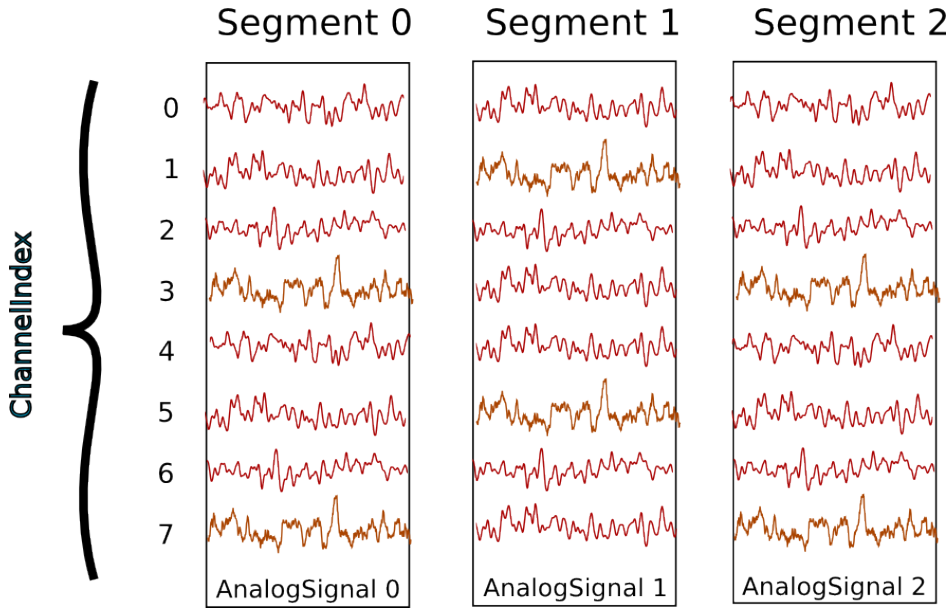
Typical use cases

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 `Group`.



`Segment` and `Group` 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 Group
group = block.groups[0]
avg = np.mean(group.analogsignals, axis=0)

plt.figure()
for index, name in enumerate(group.annotations["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?

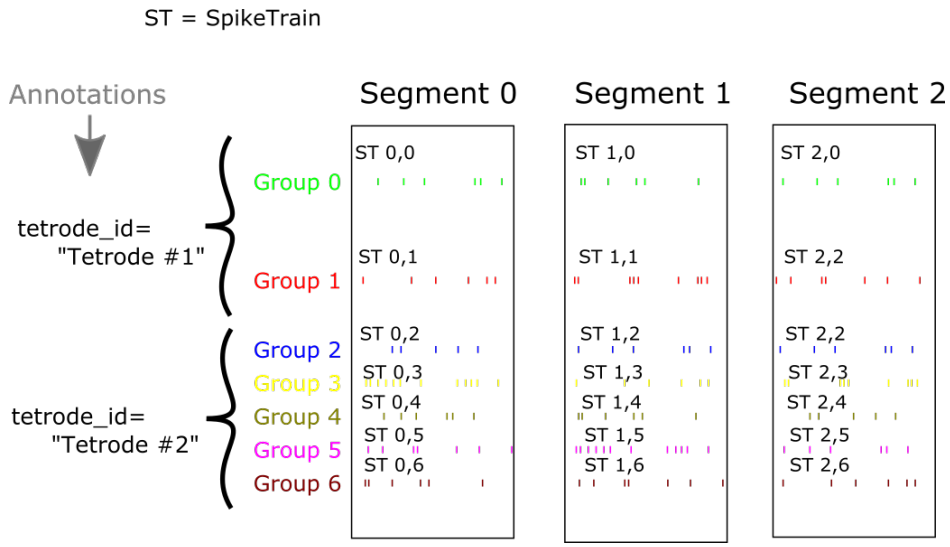
```
index = 5
avg = np.mean([seg.analogsignals[0][:, index] for seg in block.segments[::2]], axis=1)
plt.plot(avg)
```

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.

- 3 `Segments` (one per trial).
- 7 `Groups` (one per neuron), which each contain:
  - 3 `SpikeTrain` objects
  - an annotation showing which tetrode the spiketrains were recorded from

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



Note: In this scenario we have discarded the original signals, perhaps to save space, therefore we use annotations to link the spiketrains to the tetrode they were recorded from. If we wished to include the original extracellular signals, we would add a reference to the three `AnalogSignal` objects for the appropriate tetrode to the `Group` for each neuron.

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

- by trial (`Segment`)
- by neuron (`Group`)
- by tetrode

By trial

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

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

By neuron

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

```
plt.figure()
for i, group in enumerate(block.groups):
    stlist = [st - st.t_start for st in group.spiketrains]
    plt.subplot(len(block.groups), 1, i + 1)
    count, bins = np.histogram(stlist)
```

(continues on next page)

(continued from previous page)

```
    plt.bar(bins[:-1], count, width=bins[1] - bins[0])
    plt.title(f"PSTH of unit {group.name}")
plt.show()
```

By tetrode

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

```
plt.figure()
for i, tetrode_id in enumerate(block.annotations["tetrode_ids"]):
    stlist = []
    for unit in block.filter(objects=Group, tetrode_id=tetrode_id):
        stlist.extend([st - st.t_start for st in unit.spiketrains])
    plt.subplot(2, 1, i + 1)
    count, bins = np.histogram(stlist)
    plt.bar(bins[:-1], count, width=bins[1] - bins[0])
    plt.title(f"PSTH blend of tetrode {tetrode_id}")
plt.show()
```

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 recordings from 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'))

# extract spike trains from all channels
st_list = []
for signal in seg.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 = 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]:
```

(continues on next page)

(continued from previous page)

```
        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 `Group` object, assign all spiketrains to it, and then also assign the `AnalogSignal` on which we detected them.

```
unit = Group()
unit.spiketrains = st_list
unit.analogsignals.extend(seg.analogsignals)
```

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 `Group` for each cluster, assign its spiketrains to it, and still store in each group a reference to the original recording.

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, ...)
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

An IO module 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 Read a time slice of Segment

Some objects support the `time_slice` argument in `read_segment()`. This is useful to read only a subset of a dataset clipped in time. By default `time_slice=None` meaning load eveything.

This reads everything:

```
seg = reader.read_segment(time_slice=None)
```

This reads only the first 5 seconds:

```
seg = reader.read_segment(time_slice=(0*pq.s, 5.*pq.s))
```

4.7 Lazy option and proxy objects

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`. Since neo 0.7, a new lazy sytem have been added for some IO modules (all IO classes that inherit from rawio). To know if a class supports lazy mode use `ClassIO.support_lazy`.

With `lazy=True` all data objects (AnalogSignal/SpikeTrain/Event/Epoch) are replaced by proxy objects (AnalogSignalProxy/SpikeTrainProxy/EventProxy/EpochProxy).

By default (if not specified), `lazy=False`, i.e. all data is loaded.

These proxy objects contain metadata (name, sampling_rate, id, ...) so they can be inspected but they do not contain any array-like data. All proxy objects contain a `load()` method to postpone the real load of array like data.

Further more the `load()` method has a `time_slice` argument to load only a slice from the file. In this way the consumption of memory can be finely controlled.

Here are two examples that read a dataset, extract sections of the signal based on recorded events, and averages the sections.

The first example is without lazy mode, so it consumes more memory:

```
lim0, lim1 = -500 * pq.ms, +1500 * pq.ms
seg = reader.read_segment(lazy=False)
triggers = seg.events[0]
```

(continues on next page)

(continued from previous page)

```
sig = seg.analogsignals[0]  # here sig contain the whole recording in memory
all_sig_chunks = []
for t in triggers.times:
    t0, t1 = (t + lim0), (t + lim1)
    sig_chunk = sig.time_slice(t0, t1)
    all_sig_chunks.append(sig_chunk)
apply_my_fancy_average(all_sig_chunks)
```

The second example uses lazy mode, so it consumes less memory:

```
lim0, lim1 = -500*pq.ms, +1500*pq.ms
seg = reader.read_segment(lazy=True)
triggers = seg.events[0].load(time_slice=None)  # this loads all triggers in memory
sigproxy = seg.analogsignals[0]  # this is a proxy
all_sig_chunks = []
for t in triggers.times:
    t0, t1 = (t + lim0), (t + lim1)
    sig_chunk = sigproxy.load(time_slice=(t0, t1))  # here real data are loaded
    all_sig_chunks.append(sig_chunk)
apply_my_fancy_average(all_sig_chunks)
```

In addition to `time_slice`, AnalogSignalProxy supports the `channel_indexes` argument. This allows loading only a subset of channels. This is useful where the channel count is very high.

In this example, we read only three selected channels:

```
seg = reader.read_segment(lazy=True)
anasig = seg.analogsignals[0].load(time_slice=None, channel_indexes=[0, 2, 18])
```

4.8 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`.
- some IO are able to do a *lazy* load: all metadata (e.g. `sampling_rate`) are read, but not the actual numerical data.
- 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.

4.9 If you want to develop your own IO

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

4.10 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:

- *AlphaOmegaIO*
- *AsciiImageIO*
- *AsciiSignalIO*
- *AsciiSpikeTrainIO*
- *AxographIO*
- *AxonaIO*
- *AxonIO*
- *BCI2000IO*
- *BlackrockIO*
- *BlkIO*
- *BrainVisionIO*
- *BrainwareDamIO*
- *BrainwareF32IO*
- *BrainwareSrcIO*
- *CedIO*
- *ElanIO*
- *IgorIO*
- *IntanIO*
- *MEArecIO*
- *KlustaKwikIO*
- *KwikIO*
- *MaxwellIO*
- *MicromedIO*
- *NeoMatlabIO*

- *NestIO*
- *NeuralynxIO*
- *NeuroExplorerIO*
- *NeuroScopeIO*
- *NeuroshareIO*
- *NixIO*
- *NWBIO*
- *OpenEphysIO*
- *OpenEphysBinaryIO*
- *PhyIO*
- *PickleIO*
- *PlexonIO*
- *RawBinarySignalIO*
- *RawMCSIO*
- *Spike2IO*
- *SpikeGadgetsIO*
- *SpikeGLXIO*
- *StimfitIO*
- *TdtIO*
- *TiffIO*
- *WinEdrIO*
- *WinWcpIO*

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
```

extensions = ['map']

class neo.io.**AsciiImageIO**(*file_name=None*, *nb_frame=None*, *nb_row=None*, *nb_column=None*, *units=None*, *sampling_rate=None*, *spatial_scale=None*, ***kwargs*)

IO class for reading ImageSequence in a text file

***Usage*:**

```
>>> from neo import io
>>> import quantities as pq
>>> r = io.AsciiImageIO(file_name='File_asciiimage_1.txt',nb_frame=511, nb_
↪row=100,
...                     nb_column=100,units='mm', sampling_rate=1.0*pq.Hz,
...                     spatial_scale=1.0*pq.mm)
>>> block = r.read_block()
read block
creating segment
returning block
>>> block
Block with 1 segments
file_origin: 'File_asciiimage_1.txt
# segments (N=1)
0: Segment with 1 imagesequences # analogsignals (N=0)
```

extensions = []

class neo.io.**AsciiSignalIO**(*filename=None*, *delimiter='t'*, *usecols=None*, *skiprows=0*, *timecolumn=None*, *sampling_rate=array(1.) * Hz*, *t_start=array(0.) * s*, *units=UnitQuantity('volt'*, *1.0 * J/C*, *'V')*, *time_units=UnitTime('second'*, *'s')*, *method='genfromtxt'*, *signal_group_mode='split-all'*, *metadata_filename=None*)

Class for reading signals in generic ascii format. Columns represent signals. They all share the same sampling rate. The sampling rate is externally known or the first column 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 ...
```

Arguments relevant for reading and writing:

delimiter: column delimiter in file, e.g. ' ', one space, two spaces, ',', ';'

timecolumn: None or a valid integer that identifies which column contains the time vector (counting from zero)

units: units of AnalogSignal can be a str or directly a Quantity

time_units: where timecolumn is specified, the time units must be specified as a string or Quantity

metadata_filename: the path to a JSON file containing metadata

Arguments relevant only for reading:

usecols: if None take all columns otherwise a list for selected columns (counting from zero)

skiprows: skip n first lines in case they contains header informations

sampling_rate: the sampling rate of signals. Ignored if timecolumn is not None

t_start: time of the first sample (Quantity). Ignored if timecolumn is not None

signal_group_mode: if 'all-in-one', load data as a single, multi-channel AnalogSignal, if 'split-all' (default for backwards compatibility) load data as separate, single-channel AnalogSignals

method: 'genfromtxt', 'csv', 'homemade' or a user-defined function which takes a filename and usecolumns as argument and returns a 2D NumPy array.

If specifying both usecols and timecolumn, the latter should identify the column index _after_ removing the unused columns.

The methods are as follows:

- 'genfromtxt' use numpy.genfromtxt
- 'csv' use csv module
- 'homemade' use an intuitive, more robust but slow method

If *metadata_filename* is provided, the parameters for reading/writing the file ("delimiter", "timecolumn", "units", etc.) will be read from that file. IF a metadata filename is not provided, the IO will look for a JSON file in the same directory with a matching filename, e.g. if the datafile was named "foo.txt" then the IO would automatically look for a file called "foo_about.json" If parameters are specified both in the metadata file and as arguments to the IO constructor, the former will take precedence.

Example metadata file:

```
{
    "filename": "foo.txt",
    "delimiter": " ",
    "timecolumn": 0,
    "units": "pA",
    "time_units": "ms",
    "sampling_rate": {
        "value": 1.0,
        "units": "kHz"
    },
    "method": "genfromtxt",
    "signal_group_mode": 'all-in-one'
}
```

extensions = ['txt', 'asc', 'csv', 'tsv']

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     # doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
[<SpikeTrain(array([ 3.89981604,  4.73258781,  0.608428  ,  4.60246277,  1.
↪23805797,
...
```

extensions = ['txt']

class neo.io.**AxographIO**(*filename=''*, *force_single_segment=False*)

IO class for reading AxoGraph files (.axgd, .axgx)

Args:

filename (string): File name of the AxoGraph file to read.

force_single_segment (bool): Episodic files are normally read as multi-Segment Neo objects. This parameter can force AxographIO to put all signals into a single Segment. Default: False.

Example:

```
>>> import neo
>>> r = neo.io.AxographIO(filename=filename)
>>> blk = r.read_block(signal_group_mode='split-all')
>>> display(blk)
```

```
>>> # get signals
>>> seg_index = 0  # episode number
>>> sigs = [sig for sig in blk.segments[seg_index].analogsignals
...         if sig.name in channel_names]
>>> display(sigs)
```

```
>>> # get event markers (same for all segments/episodes)
>>> ev = blk.segments[0].events[0]
>>> print([ev for ev in zip(ev.times, ev.labels)])
```

```
>>> # get interval bars (same for all segments/episodes)
>>> ep = blk.segments[0].epochs[0]
>>> print([ep for ep in zip(ep.times, ep.durations, ep.labels)])
```

```
>>> # get notes
>>> print(blk.annotations['notes'])
```

extensions = ['axgd', 'axgx']

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

extensions = ['bin', 'set', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '

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.)

Here an important note from erikli@github for user who want to get the : With Axon ABF2 files, the information that you need to recapitulate the original stimulus waveform (both digital and analog) is contained in multiple places.

- *AxonIO._axon_info['protocol']* – things like number of samples in episode
- *AxonIO.axon_info['section']['ADCSection']* | *AxonIO.axon_info['section']['DACSection']* – things about the number of channels and channel properties
- *AxonIO._axon_info['protocol']['nActiveDACChannel']* – bitmask specifying which DACs are actually active
- *AxonIO._axon_info['protocol']['nDigitalEnable']* – bitmask specifying which set of Epoch timings should be used to specify the duration of digital outputs
- *AxonIO._axon_info['dictEpochInfoPerDAC']* – dict of dict. First index is DAC channel and second index is Epoch number (i.e. information about Epoch A in Channel 2 would be in *AxonIO._axon_info['dictEpochInfoPerDAC'][2][0]*)
- *AxonIO._axon_info['EpochInfo']* – list of dicts containing information about each Epoch's digital out pattern. Digital out is a bitmask with least significant bit corresponding to Digital Out 0

- *AxonIO._axon_info['listDACInfo']* – information about DAC name, scale factor, holding level, etc
- *AxonIO._t_starts* – start time of each sweep in a unified time basis
- *AxonIO._sampling_rate*

The current AxonIO.read_protocol() method utilizes a subset of these. In particular I know it doesn't consider *nDigitalEnable*, *EpochInfo*, or *nActiveDACChannel* and it doesn't account for different types of Epochs offered by Clampex/pClamp other than discrete steps (such as ramp, pulse train, etc and encoded by *nEpochType* in the EpochInfoPerDAC section). I'm currently parsing a superset of the properties used by read_protocol() in my analysis scripts, but that code still doesn't parse the full information and isn't in a state where it could be committed and I can't currently prioritize putting together all the code that would parse the full set of data. The *AxonIO._axon_info['EpochInfo']* section doesn't currently exist.

extensions = ['abf']

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

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

extensions = ['dat']

class neo.io.**BlackrockIO**(*filename*, ***kargs*)

Supplementary class for reading BlackRock data using only a single nsx file.

extensions = ['ns1', 'ns2', 'ns3', 'ns4', 'ns5', 'ns6', 'nev']

class neo.io.**BlkIO**(*file_name=None*, *units=None*, *sampling_rate=None*, *spatial_scale=None*, ***kwargs*)

Neo IO module for optical imaging data stored as BLK file

***Usage*:**

```
>>> from neo import io
>>> import quantities as pq
>>> r = io.BlkIO("file_blk_1.BLK",units='V',sampling_rate=1.0*pq.Hz,
...              spatial_scale=1.0*pq.Hz)
>>> block = r.read_block()
reading the header
reading block
returning block
>>> block
Block with 6 segments
file_origin: 'file_blk_1.BLK'
# segments (N=6)
0: Segment with 1 imagesequences description: 'stim nb:0' # analogsignals␣
↪(N=0)
1: Segment with 1 imagesequences description: 'stim nb:1' # analogsignals␣
↪(N=0)
2: Segment with 1 imagesequences description: 'stim nb:2' # analogsignals␣
↪(N=0)
3: Segment with 1 imagesequences description: 'stim nb:3' # analogsignals␣
↪(N=0)
4: Segment with 1 imagesequences description: 'stim nb:4' # analogsignals␣
↪(N=0)
5: Segment with 1 imagesequences description: 'stim nb:5' # analogsignals␣
↪(N=0)
```

Many thanks to Thomas Deneux for the MATLAB code on which this was based.

extensions = []

class neo.io.**BrainVisionIO**(*filename*)
Class for reading data from the BrainVision product.

extensions = ['vhdr']

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
```

extensions = ['dam']

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 Group.

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
```

extensions = ['f32']

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 Group 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 Group.

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.groups
>>> print blk1.groups[0].name
>>> print blk2
>>> print blk2[0].segments
>>> print blks
>>> print blks[0].segments
```

extensions = ['src']

class neo.io.**CedIO**(*filename*, *entfile=None*, *posfile=None*)

Class for reading data from CED (Cambridge Electronic Design) spike2. This internally uses the sonpy package which is closed source.

This IO reads smr and smrx files

extensions = ['smr', 'smrx']

class neo.io.**ElanIO**(*filename*, *entfile=None*, *posfile=None*)

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

Args:

filename (string) : Full path to the .eeg file

entfile (string) : Full path to the .ent file (optional). If None, the path to the ent file is inferred from the filename by adding the ".ent" extension to it

posfile (string) : Full path to the .pos file (optional). If None, the path to the pos file is inferred from the filename by adding the ".pos" extension to it

extensions = ['eeg']

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')
```

extensions = ['ibw', 'pxp']

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

extensions = ['rhd', 'rhs']

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

extensions = ['fet', 'clu', 'res', 'spk']

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

Generates a `Segment` with a `AnalogSignal`

extensions = ['kwik']

class neo.io.**MEArecIO**(*filename*)
Class for "reading" fake data from a MEArec file.

Usage:

```
>>> import neo.rawio
>>> r = neo.rawio.MEArecRawIO(filename='mearec.h5')
>>> 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')
```

extensions = ['h5']

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

extensions = ['h5']

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

extensions = ['trc', 'TRC']

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];
```

(continues on next page)

(continued from previous page)

```
    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])
```

extensions = ['mat']

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)
```

extensions = ['gdf', 'dat']

class neo.io.**NeuralynxIO**(*dirname*, *use_cache=False*, *cache_path='same_as_resource'*, *keep_original_times=False*)

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

extensions = ['nse', 'ncs', 'nev', 'ntt']

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

Class for reading data from NeuroExplorer (.nex)

extensions = ['nex']

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

Reading from Neuroscope format files.

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

extensions = ['xml', 'dat']

neo.io.**NeuroshareIO**

alias of neo.io.neurosharectypesio.NeurosharectypesIO

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

extensions = ['h5', 'nix']

class neo.io.**NWBIO**(*filename*, *mode='r'*)
Class for "reading" experimental data from a .nwb file, and "writing" a .nwb file from Neo

extensions = ['nwb']

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

extensions = []

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

extensions = []

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

extensions = []

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.

extensions = ['pkl', 'pickle']

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.

extensions = ['plx']

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.)

extensions = ['raw', '*']

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

extensions = ['raw']

class neo.io.**Spike2IO**(*filename*, ***kargs*)

extensions = ['smr']

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

extensions = ['rec']

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

Class for reading data from a SpikeGLX system

extensions = []

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)
```

extensions = ['abf', 'dat', 'axgx', 'axgd', 'cfs']

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.

extensions = []

class neo.io.**TiffIO**(*directory_path=None*, *units=None*, *sampling_rate=None*, *spatial_scale=None*, ***kwargs*)

Neo IO module for optical imaging data stored as a folder of TIFF images.

***Usage*:**

```
>>> from neo import io
>>> import quantities as pq
>>> r = io.TiffIO("dir_tiff",spatial_scale=1.0*pq.mm, units='V',
...               sampling_rate=1.0*pq.Hz)
>>> block = r.read_block()
read block
creating segment
returning block
>>> block
Block with 1 segments
file_origin: 'test'
# segments (N=1)
0: Segment with 1 imagesequences
    annotations: {'tiff_file_names': ['file_tif_1_.tiff',
        'file_tif_2.tiff',
        'file_tif_3.tiff',
        'file_tif_4.tiff',
        'file_tif_5.tiff',
        'file_tif_6.tiff',
        'file_tif_7.tiff',
        'file_tif_8.tiff',
        'file_tif_9.tiff',
        'file_tif_10.tiff',
        'file_tif_11.tiff',
        'file_tif_12.tiff',
        'file_tif_13.tiff',
        'file_tif_14.tiff']}
    # analogsignals (N=0)
```

extensions = []

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

extensions = ['EDR', 'edr']

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

extensions = ['wcp']

4.11 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.nixio.NixIO`. 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.nixio.NixIO')
>>> 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.nixio.NixIO')
>>> logger.setLevel(100)
```

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

```
>>> import logging
>>>
>>> logger = logging.getLogger('neo.io.nixio.NixIO')
>>> 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

5.1 Neo RawIO API

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.

5.2 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]
spike_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**, **spike_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
spike_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)
```

(continues on next page)

(continued from previous page)

```
(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 unambiguous 0 to n-1 (included), whereas for some IOs channel_names (and sometimes channel_ids) have no guarantees to be unique. In such cases, using names or ids may raise an error.

A selected subset of channels which is passed to get_analog_signal_chunk, get_analog_signal_size, or get_analog_signal_t_start has the additional restriction that all such channels must have the same t_start and signal_size.

Such subsets of channels may be available in specific RawIOs by using the get_group_signal_channel_indexes method, if the RawIO has defined separate group_ids for each group with those common characteristics.

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.spike_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')
```

(continues on next page)

(continued from previous page)

```
>>> 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]
```

5.3 List of implemented formats

neo.rawio provides classes for reading with low level API electrophysiological data files.

`neo.rawio.rawiolist` provides a list of successfully imported rawio classes.

Functions:

`neo.rawio.`**`get_rawio_class`**(*filename_or_dirname*)
Return a neo.rawio class guess from file extention.

Classes:

- *AxographRawIO*
- *AxonaRawIO*
- *AxonRawIO*
- *BlackrockRawIO*
- *BrainVisionRawIO*
- *CedRawIO*
- *ElanRawIO*
- *IntanRawIO*
- *MaxwellRawIO*
- *MEArecRawIO*
- *MicromedRawIO*
- *NeuralynxRawIO*
- *NeuroExplorerRawIO*
- *NeuroScopeRawIO*
- *NIXRawIO*
- *OpenEphysRawIO*
- *OpenEphysBinaryRawIO*
- :attr:'PhyRawIO'
- *PlexonRawIO*
- *RawBinarySignalRawIO*
- *RawMCSRawIO*
- *Spike2RawIO*
- *SpikeGadgetsRawIO*
- *SpikeGLXRawIO*
- *TdtRawIO*
- *WinEdrRawIO*
- *WinWcpRawIO*

class neo.rawio.**AxographRawIO**(*filename*, *force_single_segment=False*)

RawIO class for reading AxoGraph files (.axgd, .axgx)

Args:

filename (string): File name of the AxoGraph file to read.

force_single_segment (bool): Episodic files are normally read as multi-Segment Neo objects. This parameter can force AxographRawIO to put all signals into a single Segment. Default: False.

Example:

```
>>> import neo
>>> r = neo.rawio.AxographRawIO(filename=filename)
>>> r.parse_header()
>>> print(r)
```

```
>>> # get signals
>>> raw_chunk = r.get_analogsignal_chunk(
...     block_index=0, seg_index=0,
...     i_start=0, i_stop=1024,
...     channel_names=channel_names)
>>> float_chunk = r.rescale_signal_raw_to_float(
...     raw_chunk,
...     dtype='float64',
...     channel_names=channel_names)
>>> print(float_chunk)
```

```
>>> # get event markers
>>> ev_raw_times, _, ev_labels = r.get_event_timestamps(
...     event_channel_index=0)
>>> ev_times = r.rescale_event_timestamp(
...     ev_raw_times, dtype='float64')
>>> print([ev for ev in zip(ev_times, ev_labels)])
```

```
>>> # get interval bars
>>> ep_raw_times, ep_raw_durations, ep_labels = r.get_event_timestamps(
...     event_channel_index=1)
>>> ep_times = r.rescale_event_timestamp(
...     ep_raw_times, dtype='float64')
>>> ep_durations = r.rescale_epoch_duration(
...     ep_raw_durations, dtype='float64')
>>> print([ep for ep in zip(ep_times, ep_durations, ep_labels)])
```

```
>>> # get notes
>>> print(r.info['notes'])
```

```
>>> # get other miscellaneous info
>>> print(r.info)
```

extensions = ['axgd', 'axgx']

class neo.rawio.**AxonaRawIO**(*filename*)

Class for reading raw, continuous data from the Axona dacqUSB system: http://space-memory-navigation.org/DacqUSBFileFormats.pdf

The raw data is saved in .bin binary files with an accompanying .set file about the recording setup (see the above manual for details).

Usage: import neo.rawio r = neo.rawio.AxonaRawIO(

filename=os.path.join(dir_name, base_filename)

) 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]

)

extensions = ['bin', 'set', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '

class neo.rawio.**AxonRawIO**(*filename=''*)

extensions = ['abf']

class neo.rawio.**BlackrockRawIO**(*filename=None*, *nsx_override=None*, *nev_override=None*, *nsx_to_load=None*, *verbose=False*)

Class for reading data in from a file set recorded by the Blackrock (Cerebus) recording system.

Upon initialization, the class is linked to the available set of Blackrock files.

Note: This routine will handle files according to specification 2.1, 2.2, and 2.3. Recording pauses that may occur in file specifications 2.2 and 2.3 are automatically extracted and the data set is split into different segments.

The Blackrock data format consists not of a single file, but a set of different files. This constructor associates itself with a set of files that constitute a common data set. By default, all files belonging to the file set have the same base name, but different extensions. However, by using the override parameters, individual filenames can be set.

Args:

filename (string): File name (without extension) of the set of Blackrock files to associate with. Any .nsX or .nev, .sif, or .ccf extensions are ignored when parsing this parameter.

nsx_override (string): File name of the .nsX files (without extension). If None, filename is used. Default: None.

nev_override (string): File name of the .nev file (without extension). If None, filename is used. Default: None.

nsx_to_load (int, list, 'max', 'all' (=None)) default None: IDs of nsX file from which to load data, e.g., if set to 5 only data from the ns5 file are loaded. If 'all', then all nsX will be loaded. Contrary to previsous version of the IO (<0.7), nsx_to_load must be set at the init before parse_header().

Examples:

```
>>> reader = BlackrockRawIO(filename='FileSpec2.3001', nsx_to_load=5)
>>> reader.parse_header()
```

Inspect a set of file consisting of files FileSpec2.3001.ns5 and FileSpec2.3001.nev

```
>>> print(reader)
```

Display all informations about signal channels, units, segment size....

extensions = ['ns1', 'ns2', 'ns3', 'ns4', 'ns5', 'ns6', 'nev']

class neo.rawio.**BrainVisionRawIO**(*filename=''*)

extensions = ['vhdr']

class neo.rawio.**CedRawIO**(*filename=''*, *take_ideal_sampling_rate=False*)

Class for reading data from CED (Cambridge Electronic Design) spike2. This internally uses the sonpy package which is closed source.

This IO reads smr and smrx files

extensions = ['smr', 'smrx']

class neo.rawio.**ElanRawIO**(*filename=None*, *entfile=None*, *posfile=None*)

extensions = ['eeg']

class neo.rawio.**IntanRawIO**(*filename=''*)

extensions = ['rhd', 'rhs']

class neo.rawio.**MaxwellRawIO**(*filename=''*, *rec_name=None*)

Class for reading MaxOne or MaxTwo files.

extensions = ['h5']

class neo.rawio.**MEArecRawIO**(*filename=''*)

Class for "reading" fake data from a MEArec file.

Usage:

```
>>> import neo.rawio
>>> r = neo.rawio.MEArecRawIO(filename='mearec.h5')
>>> 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')
```

extensions = ['h5']

class neo.rawio.**MicromedRawIO**(*filename=''*)

Class for reading data from micromed (.trc).

extensions = ['trc', 'TRC']

class neo.rawio.**NeuralynxRawIO**(*dirname=''*, *filename=''*, *keep_original_times=False*, ***kargs*)

'' Class for reading datasets recorded by Neuralynx.

This version works with rawmode of one-dir for a single directory of files or one-file for a single file.

Examples:

```
>>> reader = NeuralynxRawIO(dirname='Cheetah_v5.5.1/original_data')
>>> reader.parse_header()
```

Inspect all files in the directory.

```
>>> print(reader)
```

Display all information about signal channels, units, segment size. . . .

extensions = ['nse', 'ncs', 'nev', 'ntt']

class neo.rawio.**NeuroExplorerRawIO**(*filename=''*)

extensions = ['nex']

class neo.rawio.**NeuroScopeRawIO**(*filename=''*)

extensions = ['xml', 'dat']

class neo.rawio.**NIXRawIO**(*filename=''*)

extensions = ['nix']

class neo.rawio.**OpenEphysRawIO**(*dirname=''*)

OpenEphys GUI software offers several data formats, see https://open-ephys.atlassian.net/wiki/spaces/OEW/pages/491632/Data+format

This class implements the legacy OpenEphys format here https://open-ephys.atlassian.net/wiki/spaces/OEW/pages/65667092/Open+Ephys+format

The OpenEphys group already proposes some tools here: https://github.com/open-ephys/analysis-tools/blob/master/OpenEphys.py but (i) there is no package at PyPI and (ii) those tools read everything in memory.

The format is directory based with several files:

- .continuous
- .events
- .spikes

This implementation is based on:

- this code https://github.com/open-ephys/analysis-tools/blob/master/Python3/OpenEphys.py written by Dan Denman and Josh Siegle
- a previous PR by Cristian Tatarau at Charité Berlin

In contrast to previous code for reading this format, here all data use memmap so it should be super fast and light compared to legacy code.

When the acquisition is stopped and restarted then files are named `*_2`, `*_3`. In that case this class creates a new Segment. Note that timestamps are reset in this situation.

Limitation :

- Works only if all continuous channels have the same sampling rate, which is a reasonable hypothesis.
- When the recording is stopped and restarted all continuous files will contain gaps. Ideally this would lead to a new Segment but this use case is not implemented due to its complexity. Instead it will raise an error.

Special cases:

- Normaly all continuous files have the same first timestamp and length. In situations where it is not the case all files are clipped to the smallest one so that they are all aligned, and a warning is emitted.

extensions = []

class neo.rawio.**OpenEphysBinaryRawIO**(*dirname=''*)

Handle several Blocks and several Segments.

Correspondencies Neo OpenEphys block[n-1] experiment[n] New device start/stop segment[s-1] recording[s] New recording start/stop

This IO handles several signal streams. Special event (npy) data are represented as array_annotations. The current implementation does not handle spiking data, this will be added upon user request

extensions = []

class neo.rawio.**PhyRawIO**(*dirname=''*)
Class for reading Phy data.

Usage:

```
>>> import neo.rawio
>>> r = neo.rawio.PhyRawIO(dirname='/dir/to/phy/folder')
>>> r.parse_header()
>>> print(r)
>>> spike_timestamp = r.get_spike_timestamps(block_index=0,
... seg_index=0, spike_channel_index=0, t_start=None, t_stop=None)
>>> spike_times = r.rescale_spike_timestamp(spike_timestamp, 'float64')
```

extensions = []

class neo.rawio.**PlexonRawIO**(*filename=''*)

extensions = ['plx']

class neo.rawio.**RawBinarySignalRawIO**(*filename=''*, *dtype='int16'*, *sampling_rate=10000.0*, *nb_channel=2*, *signal_gain=1.0*, *signal_offset=0.0*, *bytesoffset=0*)

extensions = ['raw', '*']

class neo.rawio.**RawMCSRawIO**(*filename=''*)

extensions = ['raw']

class neo.rawio.**Spike2RawIO**(*filename=''*, *take_ideal_sampling_rate=False*, *ced_units=True*, *try_signal_grouping=True*)
This implementation in neo read only old smr files. For smrx files you need to use CedRawIO which is based on sonpy.

extensions = ['smr']

class neo.rawio.**SpikeGadgetsRawIO**(*filename=''*, *selected_streams=None*)

extensions = ['rec']

class neo.rawio.**SpikeGLXRawIO**(*dirname=''*)
Class for reading data from a SpikeGLX system

extensions = []

class neo.rawio.**TdtRawIO**(*dirname=''*, *sortname=''*)

extensions = []

class neo.rawio.**WinEdrRawIO**(*filename=''*)

extensions = ['EDR', 'edr']

class neo.rawio.**WinWcpRawIO**(*filename=''*)

```
    extensions = ['wcp']
```

CHAPTER 6

Examples

6.1 Introduction

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

```
"""
This is an example for reading files with neo.io
"""

import urllib

import neo

url_repo = 'https://web.gin.g-node.org/NeuralEnsemble/ephy_testing_data/raw/master/'

# Plexon files
distantfile = url_repo + 'plexon/File_plexon_3.plx'
localfile = './File_plexon_3.plx'
urllib.request.urlretrieve(distantfile, localfile)

# create a reader
reader = neo.io.PlexonIO(filename='File_plexon_3.plx')
# read the blocks
blks = reader.read(lazy=False)
print(blks)
# access to segments
for blk in blks:
    for seg in blk.segments:
        print(seg)
        for asig in seg.analogsignals:
            print(asig)
        for st in seg.spiketrains:
            print(st)
```

(continues on next page)

(continued from previous page)

```
# CED Spike2 files
distantfile = url_repo + 'spike2/File_spike2_1.smr'
localfile = './File_spike2_1.smr'
urllib.request.urlretrieve(distantfile, localfile)

# create a reader
reader = neo.io.Spike2IO(filename='File_spike2_1.smr')
# read the block
bl = reader.read(lazy=False)[0]
print(bl)
# access to segments
for seg in bl.segments:
    print(seg)
    for asig in seg.analogsignals:
        print(asig)
    for st in seg.spiketrains:
        print(st)
```

```
"""
This is an example for reading files with neo.rawio
compare with read_files_neo_io.py
"""

import urllib
from neo.rawio import PlexonRawIO

url_repo = 'https://web.gin.g-node.org/NeuralEnsemble/ephy_testing_data/raw/master/'

# Get Plexon files
distantfile = url_repo + 'plexon/File_plexon_3.plx'
localfile = './File_plexon_3.plx'
urllib.request.urlretrieve(distantfile, localfile)

# create a reader
reader = PlexonRawIO(filename='File_plexon_3.plx')
reader.parse_header()
print(reader)
print(reader.header)

# Read signal chunks
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)

# Count units and spikes per unit
nb_unit = reader.spike_channels_count()
print('nb_unit', nb_unit)
for unit_index in range(nb_unit):
```

(continues on next page)

(continued from previous page)

```
    nb_spike = reader.spike_count(block_index=0, seg_index=0, unit_index=unit_index)
    print('unit_index', unit_index, 'nb_spike', nb_spike)

# Read spike times
spike_timestamps = reader.get_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])
spike_times = reader.rescale_spike_timestamp(spike_timestamps, dtype='float64')
print(spike_times.shape, spike_times.dtype, spike_times[:5])

# Read spike waveforms
raw_waveforms = reader.get_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])
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])

# Read event timestamps and times (take anotehr file)
distantfile = url_repo + 'plexon/File_plexon_2.plx'
localfile = './File_plexon_2.plx'
urllib.request.urlretrieve(distantfile, localfile)

# Count event per channel
reader = PlexonRawIO(filename='File_plexon_2.plx')
reader.parse_header()
nb_event_channel = reader.event_channels_count()
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)

ev_timestamps, ev_durations, ev_labels = reader.get_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)
ev_times = reader.rescale_event_timestamp(ev_timestamps, dtype='float64')
print(ev_times)
```

```
"""
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
```

(continues on next page)

(continued from previous page)

```
distantfile = 'https://web.gin.g-node.org/NeuralEnsemble/ephy_testing_data/raw/master/
↪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, Group, AnalogSignal
>>> from quantities import nA, kHz
>>> import numpy as np
>>>
>>> # create a Block with 3 Segment and 2 Group 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):
...     group = Group(name='Array probe %d' % ind)
...     blk.groups.append(group)
...
>>> # Populate the Block with AnalogSignal objects
... for seg in blk.segments:
...     for group in blk.groups:
...         a = AnalogSignal(np.random.randn(10000, 64)*nA,
...                          sampling_rate=10*kHz)
```

(continues on next page)

(continued from previous page)

```
...         group.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.

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

***Container of*:** *Segment* *Group*

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.**Group**(*objects=None*, *name=None*, *description=None*, *file_origin=None*, *allowed_types=None*, ***annotations*)

Can contain any of the data objects, views, or other groups, outside the hierarchy of the segment and block containers. A common use is to link the *SpikeTrain* objects within a *Block*, possibly across multiple Segments, that were emitted by the same neuron.

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

***Recommended attributes/properties*:**

objects (Neo object) Objects with which to pre-populate the *Group*

name (str) A label for the group.

description (str) Text description.

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

***Optional arguments*:**

allowed_types (list or tuple) Types of Neo object that are allowed to be added to the Group. If not specified, any Neo object can be added.

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

***Container of*:** *AnalogSignal*, *IrregularlySampledSignal*, *SpikeTrain*, *Event*, *Epoch*, *ChannelView*, *Group*

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*, *array_annotations=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]
```

(continues on next page)

(continued from previous page)

```
<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.

array_annotations (dict) Dict mapping strings to numpy arrays containing annotations for all data points

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`)

***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*, *array_annotations=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.

array_annotations (dict) Dict mapping strings to numpy arrays containing annotations for all data points

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.**ChannelView**(*obj*, *index*, *name=None*, *description=None*, *file_origin=None*, *array_annotations=None*, ***annotations*)

A tool for indexing a subset of the channels within an *AnalogSignal* or *IrregularlySampledSignal*;

***Required attributes/properties*:**

obj (AnalogSignal or IrregularlySampledSignal) The signal being indexed.

index (list/1D-array) boolean or integer mask to select the channels of interest.

***Recommended attributes/properties*:**

name (str) A label for the view.

description (str) Text description.

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

array_annotations (dict) Dict mapping strings to numpy arrays containing annotations for all data points

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

class neo.core.**Event**(*times=None*, *labels=None*, *units=None*, *name=None*, *description=None*, *file_origin=None*, *array_annotations=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='U'))
>>>
>>> evt.times
array([  0.,  10.,  20.]) * s
>>> evt.labels
array(['trig0', 'trig1', 'trig2'],
      dtype='<U5')
```

***Required attributes/properties*:**

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

labels (numpy.array 1D dtype='U' or list) 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.

***Optional attributes/properties*:**

array_annotations (dict) Dict mapping strings to numpy arrays containing annotations for all data points

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*, *array_annotations=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='U'))
>>>
>>> epc.times
array([  0.,  10.,  20.]) * s
>>> epc.durations
array([ 10.,   5.,   7.]) * ms
>>> epc.labels
array(['btn0', 'btn1', 'btn2'],
      dtype='<U4')
```

***Required attributes/properties*:**

times (quantity array 1D, numpy array 1D or list) The start times of each time period.

durations (quantity array 1D, numpy array 1D, list, or quantity scalar) The length(s) of each time period. If a scalar, the same value is used for all time periods.

labels (numpy.array 1D dtype='U' or list) 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.

***Optional attributes/properties*:**

array_annotations (dict) Dict mapping strings to numpy arrays containing annotations for all data points

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=None*, *copy=True*, *sampling_rate=array(1.) * Hz*, *t_start=array(0.) * s*, *waveforms=None*, *left_sweep=None*, *name=None*, *file_origin=None*, *description=None*, *array_annotations=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
```

(continues on next page)

(continued from previous page)

```
<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, 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.

array_annotations (dict) Dict mapping strings to numpy arrays containing annotations for all data points

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.

class neo.core.**ImageSequence**(*image_data*, *units=None*, *dtype=None*, *copy=True*, *t_start=array(0.) * s*, *spatial_scale=None*, *frame_duration=None*, *sampling_rate=None*, *name=None*, *description=None*, *file_origin=None*, ***annotations*)

Representation of a sequence of images, as an array of three dimensions organized as [frame][row][column].

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

usage:

```
>>> from neo.core import ImageSequence
>>> import quantities as pq
>>>
>>> img_sequence_array = [[[column for column in range(20)]for row in range(20)]
...                           for frame in range(10)]
>>> image_sequence = ImageSequence(img_sequence_array, units='V',
...                                sampling_rate=1 * pq.Hz,
...                                spatial_scale=1 * pq.micrometer)
>>> image_sequence
ImageSequence 10 frames with width 20 px and height 20 px; units V; datatype int64
sampling rate: 1.0
spatial_scale: 1.0
>>> image_sequence.spatial_scale
array(1.) * um
```

***Required attributes/properties*:**

image_data (3D NumPy array, or a list of 2D arrays) The data itself

units (quantity units)

sampling_rate *or* **frame_duration** (quantity scalar) Number of samples per unit time or duration of a single image frame. If both are specified, they are checked for consistency.

spatial_scale (quantity scalar) size for a pixel.

t_start (quantity scalar) Time when sequence begins. Default 0.

***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/`frame_duration`)

frame_duration (quantity scalar) Duration of each image frame. (1/`sampling_rate`)

spatial_scale Size of a pixel

duration (Quantity) Sequence duration, read-only. (size * `frame_duration`)

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

class neo.core.**RectangularRegionOfInterest**(*x*, *y*, *width*, *height*)

Representation of a rectangular ROI

Usage:

```
>>> roi = RectangularRegionOfInterest(20.0, 20.0, width=5.0, height=5.0)
>>> signal = image_sequence.signal_from_region(roi)
```

***Required attributes/properties*:**

x, y (integers or floats) Pixel coordinates of the centre of the ROI

width (integer or float) Width (x-direction) of the ROI in pixels

height (integer or float) Height (y-direction) of the ROI in pixels

class neo.core.**CircularRegionOfInterest**(*x*, *y*, *radius*)

Representation of a circular ROI

Usage:

```
>>> roi = CircularRegionOfInterest(20.0, 20.0, radius=5.0)
>>> signal = image_sequence.signal_from_region(roi)
```

***Required attributes/properties*:**

x, y (integers or floats) Pixel coordinates of the centre of the ROI

radius (integer or float) Radius of the ROI in pixels

class neo.core.**PolygonRegionOfInterest**(**vertices*)

Representation of a polygonal ROI

Usage:

```
>>> roi = PolygonRegionOfInterest(
...     (20.0, 20.0),
...     (30.0, 20.0),
...     (25.0, 25.0)
... )
>>> signal = image_sequence.signal_from_region(roi)
```

***Required attributes/properties*:**

vertices tuples containing the (x, y) coordinates, as integers or floats, of the vertices of the polygon

CHAPTER 8

Release notes

8.1 Neo 0.10.0 release notes

27th July 2021

8.1.1 New IO modules

- *CedIO* - an alternative to *Spike2IO*
- *AxonaIO*
- *OpenEphysIO* - handle the binary format
- *PhyIO*
- *SpikeGLXIO*
- *NWBIO* - support for a subset of the NWB:N format
- *MaxwellIO*

8.1.2 Bug fixes and improvements in IO modules

- *NeuralynxIO* was refactored and now supports new file versions (neuraview) and single file loading.
- Legacy versions of old IOs were removed for NeuralynxIO (neuralynxio_v1), BlackrockIO, NeoHdf5IO.
- `NixIOfr` now supports array annotations of `AnalogSignal` objects.
- `NSDFIO` was removed because we can no longer maintain it.
- all IOs now accept `pathlib.Path` objects.
- The IO modules of this release have been tested with version 0.1.0 of the ephy_testing_data.

8.1.3 Removal of Unit and ChannelIndex

In version 0.9.0 *Group* and *ChannelView* were introduced, replacing Unit and ChannelIndex, which were deprecated. In this version the deprecated Unit and ChannelIndex are removed and only the new *Group* and *ChannelView* objects are available.

8.1.4 Supported Python and NumPy versions

We no longer support Python 3.6, nor versions of NumPy older than 1.16.

8.1.5 Other new or modified features

- Lists of *SpikeTrain* objects can now also be created from two arrays: one containing spike times and the other unit identities of the times (SpikeTrainList).
- Object identity is now preserved when using utility time_slice() methods.

See all pull requests included in this release and the list of closed issues.

8.1.6 RawIO modules

Internal refactoring of the neo.rawio module regarding channel grouping. Now the concept of a signal stream is used to handle channel groups for signals. This enhances the way the annotation and array_annotation attributes are rendered at neo.io level.

8.1.7 Acknowledgements

Thanks to Samuel Garcia, Julia Sprenger, Peter N. Steinmetz, Andrew Davison, Steffen Bürgers, Regimantas Jurkus, Alessio Buccino, Shashwat Sridhar, Jeffrey Gill, Etienne Combrisson, Ben Dichter and Elodie Legouée for their contributions to this release.

8.2 Neo 0.9.0 release notes

10th November 2020

8.2.1 Group and ChannelView replace Unit and ChannelIndex

Experience with ChannelIndex and Unit has shown that these classes are often confusing and difficult to understand. In particular, ChannelIndex was trying to provide three different functionalities in a single object:

- providing information about individual traces within AnalogSignals like the channel id and the channel name (labelling)
- grouping a subset of traces within an AnalogSignal via the index attribute (masking)
- linking between / grouping AnalogSignals (grouping)

while grouping SpikeTrains required a different class, Unit. For more pointers to the difficulties this created, and some of the limitations of this approach, see this Github issue.

With the aim of making the three functionalities of labelling, masking and grouping both easier to use and more flexible, we have replaced ChannelIndex and Unit with:

- array annotations (*labelling*) - already available since Neo 0.8
- `ChannelView` (*masking*) - defines subsets of channels within an *AnalogSignal* using a mask
- `Group` (*grouping*) - allows any Neo object except :class‘Segment‘ and `Block` to be grouped

For some guidance on migrating from `ChannelIndex`/`Unit` to `Group` and `ChannelView` see ../grouping.

8.2.2 Python 3 only

We have now dropped support for Python 2.7 and Python 3.5, and for versions of NumPy older than 1.13. In future, we plan to follow NEP29 + one year, i.e. we will support Python and NumPy versions for one year longer than recommended in NEP29. This was discussed here.

8.2.3 Change in default behaviour for grouping channels in IO modules

Previously, when reading multiple related signals (same length, same units) from a file, some IO classes would by default create a separate, single-channel `AnalogSignal` per signal, others would combine all related signals into one multi-channel `AnalogSignal`.

From Neo 0.9.0, the default for all IO classes is to create a one multi-channel `AnalogSignal`. To get the “multiple single-channel signals” behaviour, use:

```
io.read(signal_group_mode="split-all")
```

8.2.4 Other new or modified features

- added methods `rectify()`, `downsample()` and `resample()` to `AnalogSignal`
- `SpikeTrain.merge()` can now merge multiple spiketrains
- the utility function `cut_block_by_epochs()` gives a new `Block` now rather than modifying the block in place
- some missing properties such as `t_start` were added to `ImageSequence`, and `sampling_period` was renamed to `frame_duration`
- `AnalogSignal.time_index()` now accepts arrays of times, not just a scalar.

See all pull requests included in this release and the list of closed issues.

8.2.5 Bug fixes and improvements in IO modules

- NeoMatlabIO (support for signal annotations)
- NeuralynxIO (fix handling of empty .nev files)
- AxonIO (support EDR3 header, fix channel events bug)
- Spike2IO (fix rounding problem, fix for v9 SON files)
- MicromedIO (fix label encoding)

8.2.6 Acknowledgements

Thanks to Julia Sprenger, Samuel Garcia, Andrew Davison, Alexander Kleinjohann, Hugo van Kemenade, Achilleas Koutsou, Jeffrey Gill, Corentin Fragnaud, Aitor Morales-Gregorio, Rémi Proville, Robin Gutzen, Marin Manuel, Simon Danner, Michael Denker, Peter N. Steinmetz, Diziet Asahi and Lucien Krapp for their contributions to this release.

8.3 Neo 0.8.0 release notes

30th September 2019

8.3.1 Lazy loading

Neo 0.8 sees a major new feature, the ability to selectively load only parts of a data file (for supported file formats) into memory, for example only a subset of the signals in a segment, a subset of the channels in a signal, or even only a certain time slice of a given signal.

This can lead to major savings in time and memory consumption, or can allow files that are too large to be loaded into memory in their entirety to be processed one section at a time.

Here is an example, loading only certain sections of a signal:

```
lim0, lim1 = -500*pq.ms, +1500*pq.ms
seg = reader.read_segment(lazy=True)     # this loads only the segment structure and␣
↪metadata
                                         #    but all data objects are replaced by␣
↪proxies
triggers = seg.events[0].load()          # this loads all triggers in memory
sigproxy = seg.analogsignals[0]          # this is a proxy object
all_sig_chunks = []
for t in triggers.times:
    t0, t1 = (t + lim0), (t + lim1)
    sig_chunk = sigproxy.load(time_slice=(t0, t1))  # here the actual data are loaded
    all_sig_chunks.append(sig_chunk)
```

Not all IO modules support lazy loading (but many do). To know whether a given IO class supports lazy mode, use `SomeIO.support_lazy`.

For more details, see *Lazy option and proxy objects*.

8.3.2 Image sequence data

Another new feature, although one that is more experimental, is support for image sequence data, coming from calcium imaging of neuronal activity, voltage-sensitive dye imaging, etc.

The new `ImageSequence` object contains a sequence of image frames as a 3D array. As with other Neo data objects, the object also holds metadata such as the sampling rate/frame duration and the spatial scale (physical size represented by one pixel).

Three new IO modules, `TiffIO`, `AsciiImageIO` and `BlkIO`, allow reading such data from file, e.g.:

```
from quantities import Hz, mm, dimensionless
from neo.io import TiffIO

```

(continues on next page)

(continued from previous page)

```
data = TiffIO(data_path).read(units=dimensionless, sampling_rate=25 * Hz,
                              spatial_scale=0.05 * mm)
images = data[0].segments[0].imagesequences[0]
```

`ImageSequence` is a subclass of the NumPy `ndarray`, and so can be manipulated in the same ways, e.g.:

```
images /= images.max()
background = np.mean(images, axis=0)
preprocessed_images = images - background
```

Since a common operation with image sequences is to extract time series from regions of interest, Neo also provides various region-of-interest classes which perform this operation, returning an `AnalogSignal` object:

```
roi = CircularRegionOfInterest(x=50, y=50, radius=10)
signal = preprocessed_images.signal_from_region(roi)[0]
```

8.3.3 Other new features

- new neo.utils module
- Numpy 1.16+ compatibility
- `time_shift()` method for `Epoch`/`Event`/`AnalogSignal`
- `time_slice()` method is now more robust
- dropped support for Python 3.4

See all pull requests included in this release and the list of closed issues.

8.3.4 Bug fixes and improvements in IO modules

- Blackrock
- Neuroshare
- NixIOFr
- NixIO (array annotation + 1d coordinates)
- AsciiSignal (fix + json metadata + IrregularlySampledSignals + write proxy)
- Spike2 (group same sampling rate)
- Brainvision
- NeuralynxIO

Warning: Some IOs (based on rawio) when loading can choose to split each channel into its own 1-channel `AnalogSignal` or to group them in a multi-channel `AnalogSignal`. The default behavior (either `signal_group_mode='split-all'` or `'group-same-units'`) is not the same for all IOs for backwards compatibility reasons. In the next release, all IOs will have the default `signal_group_mode='group-same-units'`

8.3.5 Acknowledgements

Thanks to Achileas Koutsou, Chek Yin Choi, Richard C. Gerkin, Hugo van Kemenade, Alexander Kleinjohann, Björn Müller, Jeffrey Gill, Christian Kothe, Mike Sintsov, @rishidhingra, Michael Denker, Julia Sprenger, Corentin Fragnaud, Andrew Davison and Samuel Garcia for their contributions to this release.

8.4 Neo 0.7.2 release notes

10th July 2019

New RawIO class:

- AxographRawIO

Bug fixes:

- Various CI fixes

Thanks to Andrew Davison, Samuel Garcia, Jeffrey Gill and Julia Sprenger for their contributions to this release.

8.5 Neo 0.7.1 release notes

13th December 2019

- Add alias *duplicate_with_new_array* for *duplicate_with_new_data*, for backwards compatibility
- Update *NeuroshareapiIO* and *NeurosharectypesIO* to Neo 0.6
- Create basic and compatibility test for *nixio_fr*

Thanks to Chek Yin Choi, Andrew Davison and Julia Sprenger for their contributions to this release.

8.6 Neo 0.7.0 release notes

26th November 2018

Main added features:

- array annotations

Other features:

- *Event.to_epoch()*
- Change the behaviour of *SpikeTrain.__add__* and *SpikeTrain.__sub__*
- bug fix for *Epoch.time_slice()*

New IO classes:

- RawMCSRawIO (raw multi channel system file format)
- OpenEphys format
- Intanrawio (both RHD and RHS)
- AxographIO

Many bug fixes and improvements in IO:

- AxonIO
- WinWCPIO
- NixIO
- ElphyIO
- Spike2IO
- NeoMatlab
- NeuralynxIO
- BlackrockIO (V2.3)
- NixIO (rewritten)

Removed:

- PyNNIO

(Full list of closed issues)

Thanks to Achilleas Koutsou, Andrew Davison, Björn Müller, Chadwick Boulay, erikli, Jeffrey Gill, Julia Sprenger, Lucas (lkoelman), Mark Histed, Michael Denker, Mike Sintsov, Samuel Garcia, Scott W Harden and William Hart for their contributions to this release.

8.7 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://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.

8.8 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)

8.9 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)

8.10 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:

8.10.1 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.

8.10.2 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
```

8.10.3 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.

8.10.4 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.

8.11 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

8.12 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.

8.13 Version 0.3.2

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

8.14 Version 0.3.1

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

8.15 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

8.16 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://neuro-psi.cnrs.fr/spip.php?article943)
- added BrainVisionIO (see https://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

8.17 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 macOS, 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/en/articles/watching-and-unwatching-repositories).

9.3 Requirements

- Python 3.5 or later
- numpy >= 1.11.0
- quantities >= 0.12.1

- nose >= 1.1.2 (for running tests)
- Sphinx (for building documentation)
- (optional) coverage >= 2.85 (for measuring test coverage)
- (optional) scipy >= 0.12 (for MatlabIO)
- (optional) h5py >= 2.5 (for KwikIO)
- (optional) nixio (for NixIO)
- (optional) pillow (for TiffIO)

We strongly recommend you develop within a virtual environment (from virtualenv, venv or conda).

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/en/articles/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
$ 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
$ 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:

```
$ 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/en/articles/about-pull-requests).

9.9 Python version

Neo should work with Python 3.5 or newer. If you need support for Python 2.7, use Neo v0.8.0 or earlier.

9.10 Coding standards and style

All code should conform as much as possible to PEP 8, and should run with Python 3.5 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
```

Tag the release in the Git repository and push it:

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

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

```
$ twine upload dist/neo-0.X.Y.tar.gz
```

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.rawio.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`:

```
import unittest

from neo.rawio.axonrawio import AxonRawIO

from neo.test.rawiotest.common_rawio_test import BaseTestRawIO


class TestAxonRawIO(BaseTestRawIO, unittest.TestCase, ):
    rawioclass = AxonRawIO
    entities_to_test = [
        'axon/File_axon_1.abf',  # V2.0
        'axon/File_axon_2.abf',  # V1.8
        'axon/File_axon_3.abf',  # V1.8
        'axon/File_axon_4.abf',  # 2.0
```

(continues on next page)

(continued from previous page)

```
        'axon/File_axon_5.abf',  # V.20
        'axon/File_axon_6.abf',  # V.20
        'axon/File_axon_7.abf',  # V2.6
        'axon/test_file_edr3.abf',  # EDR3
    ]
    entities_to_download = [
        'axon'
    ]

    def test_read_raw_protocol(self):
        reader = AxonRawIO(filename=self.get_local_path('axon/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.nixio.NixIO`. 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.rawio.**ExampleRawIO**(*filename=''*)

Class for "reading" fake data from an imaginary file.

For the user, it gives access 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*

- Follow the *Guidelines for IO implementation*

This fake IO:

- has 2 blocks
- blocks have 2 and 3 segments
- has 2 signals streams of 8 channel each (sample_rate = 10000) so 16 channels in total
- has 3 spike_channels
- has 2 event channels: one has *type=event*, the other has *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(spike_channel_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)
```

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

Here are the entire files:

```
"""
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 from BaseRawIO
    * copy/paste all methods that need to be implemented.
    * 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_streams'] = signal_streams
            self.header['signal_channels'] = signal_channels
            self.header['spike_channels'] = spike_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
```

(continues on next page)

(continued from previous page)

```
  3. Step 3 : Create the neo.io class with the wrapper
    * Create a file in neo/io/ that ends with "io.py"
    * Create a class that inherits both your 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 .baserawio import (BaseRawIO, _signal_channel_dtype, _signal_stream_dtype,
                _spike_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 gives access 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 :ref:`neo_rawio_API`
      * Follow the :ref:`io_guiline`

    This fake IO:
        * has 2 blocks
        * blocks have 2 and 3 segments
        * has  2 signals streams  of 8 channel each (sample_rate = 10000) so 16␣
↪channels in total
        * has 3 spike_channels
        * has 2 event channels: one has *type=event*, the other has
          *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(spike_channel_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)
```

(continues on next page)

(continued from previous page)

```
    """
    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 part of a RawIO
        # we need to collect from the original format all
        # information required for fast access
        # 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 fake signals stream information
        signal_streams = []
        for c in range(2):
            name = f'stream {c}'
            stream_id = c
            signal_streams.append((name, stream_id))
        signal_streams = np.array(signal_streams, dtype=_signal_stream_dtype)

        # create fake signals channels information
        # This is mandatory!!!!
        # gain/offset/units are really important because
        # the scaling to real value will be done with that
        # The real signal will be evaluated as `(raw_signal * gain + offset) * pq.
↪Quantity(units)`
        signal_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 related 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.
            # stream_id indicates how to group channels
            # channels inside a "stream" share same characteristics
            #  (sampling rate/dtype/t_start/units/...)
            stream_id = str(c // 8)
            signal_channels.append((ch_name, chan_id, sr, dtype, units, gain, offset,␣
↪stream_id))
```

(continues on next page)

(continued from previous page)

```
        signal_channels = np.array(signal_channels, dtype=_signal_channel_dtype)

        # A stream can contain signals with different physical units.
        # Here, the two last channels will have different units (pA)
        # Since AnalogSignals must have consistent units across channels,
        # this stream will be split in 2 parts on the neo.io level and finally 3␣
↪AnalogSignals
        # will be generated per Segment.
        signal_channels[-2:]['units'] = 'pA'

        # create fake 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
        spike_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.
            spike_channels.append((unit_name, unit_id, wf_units, wf_gain,
                                   wf_offset, wf_left_sweep, wf_sampling_rate))
        spike_channels = np.array(spike_channels, dtype=_spike_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_streams'] = signal_streams
        self.header['signal_channels'] = signal_channels
        self.header['spike_channels'] = spike_channels
        self.header['event_channels'] = event_channels

        # insert some annotations/array_annotations at some place
        # at neo.io level. IOs can add annotations
        # to any object. To keep this functionality with the wrapper
        # BaseFromRaw you can add annotations in a nested dict.

        # `_generate_minimal_annotations()` must be called to generate the nested
        # dict of annotations/array_annotations
        self._generate_minimal_annotations()
        # this pprint lines really help for understand the nested (and complicated␣
↪sometimes) dict
```

(continues on next page)

(continued from previous page)

```
        # from pprint import pprint
        # pprint(self.raw_annotations)

        # Until here all mandatory operations for setting up a rawio are implemented.
        # The following lines provide additional, recommended annotations for the
        # final neo objects.
        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(2):
                    sig_an = seg_ann['signals'][c]['nickname'] = \
                        f'This stream {c} is from a subdevice'
                    # add some array annotations (8 channels)
                    sig_an = seg_ann['signals'][c]['__array_annotations__']['impedance
↪'] = \
                        np.random.rand(8) * 10000
                for c in range(3):
                    spiketrain_an = seg_ann['spikes'][c]
                    spiketrain_an['quality'] = 'Good!!'
                    # add some array annotations
                    num_spikes = self.spike_count(block_index, seg_index, c)
                    spiketrain_an['__array_annotations__']['amplitudes'] = \
                        np.random.randn(num_spikes)

                for c in range(2):
                    event_an = seg_ann['events'][c]
                    if c == 0:
                        event_an['nickname'] = 'Miss Event 0'
                        # add some array annotations
                        num_ev = self.event_count(block_index, seg_index, c)
                        event_an['__array_annotations__']['button'] = ['A'] * num_ev
                    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, stream_index):
        # We generate fake data in which the two stream signals have the same shape
        # across all segments (10.0 seconds)
        # This is not the case for real data, instead you should return the signal
```

(continues on next page)

(continued from previous page)

```
        # size depending on the block_index and segment_index
        # 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, stream_index):
        # 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,
                                stream_index, channel_indexes):
        # this must return a signal chunk in a signal stream
        # limited with i_start/i_stop (can be None)
        # channel_indexes can be None (=all channel in the stream) 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.
        # To speed up this call all preparatory calculations should be implemented
        # in _parse_header().

        # 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

        if i_start < 0 or i_stop > 100000:
            # some check
            raise IndexError("I don't like your jokes")

        if channel_indexes is None:
            nb_chan = 8
        elif isinstance(channel_indexes, slice):
            channel_indexes = np.arange(8, dtype='int')[channel_indexes]
            nb_chan = len(channel_indexes)
        else:
            channel_indexes = np.asarray(channel_indexes)
            if any(channel_indexes < 0):
                raise IndexError('bad boy')
            if any(channel_indexes >= 8):
                raise IndexError('big bad wolf')
```

(continues on next page)

(continued from previous page)

```
            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, spike_channel_index):
        # Must return the nb of spikes for given (block_index, seg_index, spike_
↪channel_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, spike_channel_index, t_
↪start, t_stop):
        # In our IO, timestamp 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, spike_channel_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['spike_channels']
        # Here, we have a realistic case: all waveforms are only noise.
        # it is not always the case
```

(continues on next page)

(continued from previous page)

```
        # we 20 spikes with a sweep of 50 (5ms)

        # trick to get how many spike in the slice
        ts = self._get_spike_timestamps(block_index, seg_index,
                                        spike_channel_index, t_start, t_stop)
        nb_spike = ts.size

        np.random.seed(2205)  # a magic number (my birthday)
        waveforms = np.random.randint(low=-2**4, high=2**4, size=nb_spike * 50, dtype=
↪'int16')
        waveforms = waveforms.reshape(nb_spike, 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, event_channel_index):
        # must rescale to second a particular event_timestamps
```

(continues on next page)

(continued from previous page)

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

```
"""
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, 21]
- 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, 6]
- 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]
- William Hart [17]
- erikli(github)
- Jeffrey Gill [18]
- Lucas (lkoelman@github)
- Mark Histed
- Mike Sintsov [19]
- Scott W Harden [20]
- Chek Yin Choi (hkchekc@github)
- Corentin Fragnaud [21]
- Alexander Kleinjohann
- Christian Kothe
- rishidhingra@github
- Hugo van Kemenade
- Aitor Morales-Gregorio [13]
- Peter N Steinmetz [22]
- Shashwat Sridhar
- Alessio Buccino [23]
- Regimantas Jurkus [13]
- Steffen Buergers [24]
- Etienne Combrisson [6]
- Ben Dichter [24]
- Elodie Legouée [21]

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 Cognitives, 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
17. Swinburne University of Technology, Australia
18. Case Western Reserve University (CWRU) · Department of Biology
19. IAL Developmental Neurobiology, Kazan Federal University, Kazan, Russia
20. Harden Technologies, LLC
21. Institut des Neurosciences Paris-Saclay, CNRS UMR 9197 - Université Paris-Sud, Gif-sur-Yvette, France
22. Neurtex Brain Research Institute, Dallas, TX, USAs
23. Bio Engineering Laboratory, DBSSE, ETH, Basel, Switzerland
24. CatalystNeuro

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

11.1 Acknowledgements

Neo was developed in part in the Human Brain Project, funded from the European Union's Horizon 2020 Framework Programme for Research and Innovation under Specific Grant Agreements No. 720270 and No. 785907 (Human Brain Project SGA1 and SGA2).

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{neo14,
    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

G

I

K

M

N

O

P

R

S

T

W