Neo Documentation

Release 0.2.1

Neo authors and contributors <neuralensemble@googlegroups.c

November 23, 2012

CONTENTS

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

The goal of Neo is to improve interoperability between Python tools for analyzing, visualizing and generating electrophysiology data (such as OpenElectrophy, NeuroTools, G-node, Helmholtz, PyNN) 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 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

ONE

DOWNLOAD AND INSTALLATION

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

1.1 Dependencies

- Python >= 2.6
- numpy >= 1.3.0 (1.5.0 for Python 3)
- quantities >= 0.9.0

For Debian/Ubuntu, you can install these using:

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

You may need to run these as root. For other operating systems, you can download installers from the links above.

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

- scipy >= 0.8 for NeoMatlabIO
- pytables >= 2.2 for Hdf5IO

For SciPy on Debian testing/Ubuntu, you can install these using:

```
$ apt-get install python-scipy
```

For PyTables version 2.2:

```
$ apt-get install libhdf5-serial-dev python-numexpr cython
$ pip install tables
```

1.2 Installing from the Python Package Index

If you have pip installed:

```
$ pip install neo
```

Alternatively, if you have setuptools:

```
$ easy_install neo
```

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

To download and install manually, download:

http://pypi.python.org/packages/source/n/neo/neo-0.2.1.tar.gz

Then:

```
$ tar xzf neo-0.2.1.tar.gz
$ cd neo-0.2.1
$ python setup.py install
```

or:

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

depending on which version of Python you are using.

1.3 Installing from source

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

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

1.4 Python 3 support

`neo.core` is fully compatible with Python 3, but only some of the IO modules support it, as shown in the table below:

Module	Python 2	Python 3
AlphaOmegaIO	Yes	No
AsciiSignalIO	Yes	Yes
AsciiSpikeTrainIO	Yes	Yes
AxonIO	Yes	No
BlackrockIO	Yes	No
ElanIO	Yes	No
HDF5IO	Yes	No
KlustakwikIO	Yes	No
MicromedIO	Yes	No
NeoMatlabIO	Yes	Yes
NeuroExplorerIO	Yes	No
PickleIO	Yes	Yes
PlexonIO	Yes	No
PyNNIO	Yes	Yes
RawBinarySignalIO	Yes	Yes
Spike2IO	Yes	Yes
TdtIO	Yes	No
WinEdrIO	Yes	Yes
WinWcpIO	Yes	Yes

CHAPTER

TWO

NEO CORE

2.1 Introduction

Objects in Neo represent neural data and collections of data. Neo objects fall into three categories: data objects, container objects and grouping objects.

2.1.1 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 continuous, analog signal.

`AnalogSignalArray`: A regular sampling of a multichannel continuous analog signal. This representation (as a 2D NumPy array) may be more efficient for subsequent analysis than the equivalent list of individual `AnalogSignal` objects.

`Spike`: One action potential characterized by its time and waveform.

`SpikeTrain`: A set of action potentials (spikes) emitted by the same unit in a period of time (with optional waveforms).

`Event` and `EventArray`: A time point representng an event in the data, or an array of such time points.

`Epoch` and `EpochArray`: An interval of time representing a period of time in the data, or an array of such intervals.

2.1.2 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 `RecordingChannelGroup` objects.

2.1.3 Grouping objects

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

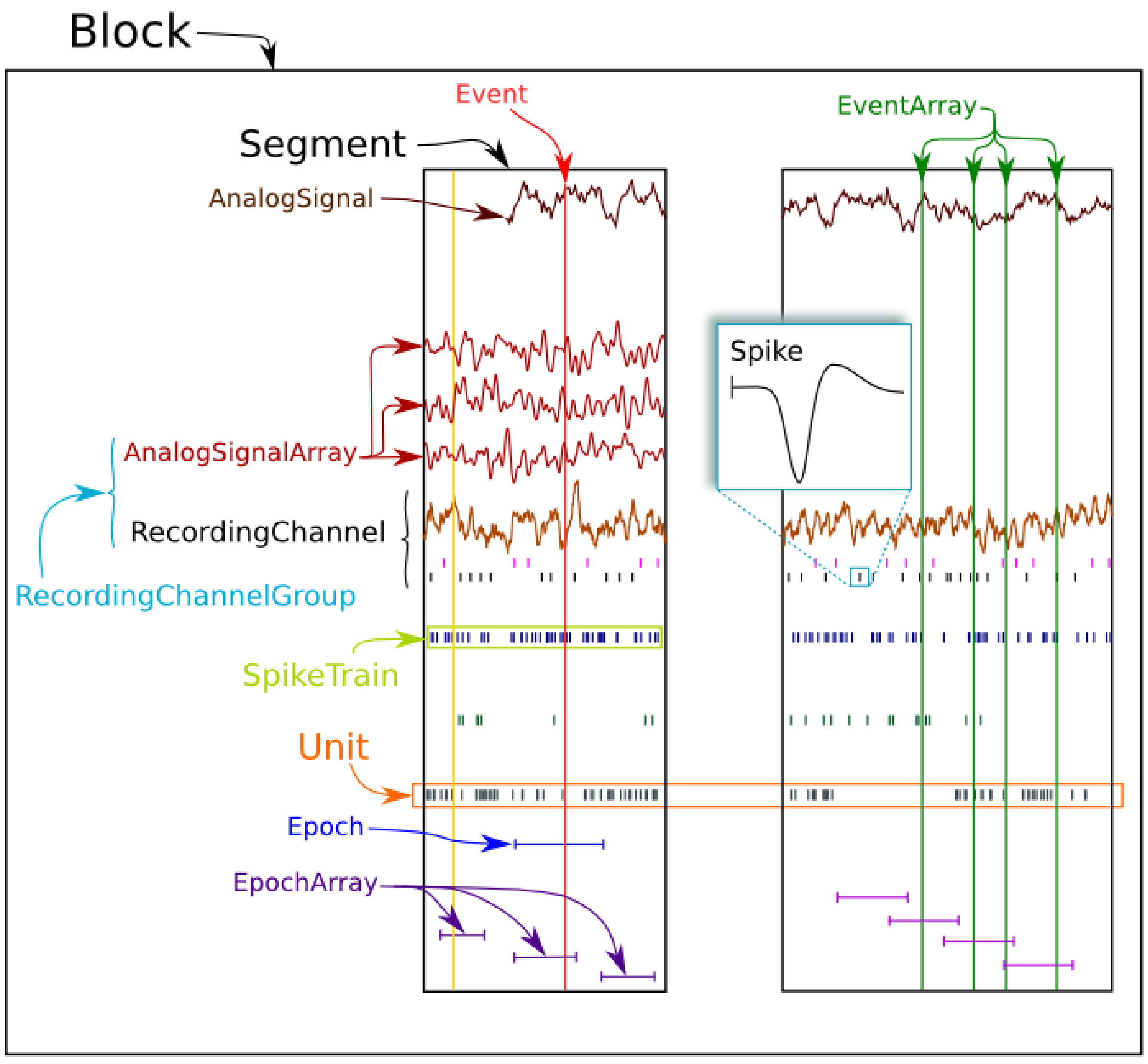
`RecordingChannel:` Links `AnalogSignal`, `SpikeTrain` objects that come from the same logical and/or physical channel inside a `Block`, possibly across several `Segment` objects.

`RecordingChannelGroup:`

A group for associated `RecordingChannel` objects. This has several possible uses:

- for linking several `AnalogSignalArray` objects across several `Segment` objects inside a `Block`.
- for multielectrode arrays, where spikes may be recorded on more than one recording channel, and so the `RecordingChannelGroup` can be used to associate each `Unit` with the group of recording channels from which it was calculated.
- for grouping several `RecordingChannel` objects. There are many use cases for this. For instance, for intracellular recording, it is common to record both membrane potentials and currents at the same time, so each `RecordingChannelGroup` may correspond to the particular property that is being recorded. For multielectrode arrays, `RecordingChannelGroup` is used to gather all `RecordingChannel` objects of the same array.

`Unit:` A Unit gathers all the `SpikeTrain` objects within a common `Block`, possibly across several Segments, that have been emitted by the same cell. A `Unit` is linked to `RecordingChannelGroup` objects from which it was detected. This replaces the `Neuron` class in the previous version of Neo (v0.1).



Neo 0.2 architecture

2.2 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 a 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.recordingchannelgroups`

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

- `Segment.block`
- `AnalogSignal.segment`
- `SpikeTrains.segment`
- `RecordingChannelGroup.block`

Here is an example showing these relationships in use:

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

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

On the *Neo diagram* you can also see a magenta line reflecting the *many-to-many* relationship between `RecordingChannel` and `RecordingChannelGroup`. This means that each group can contain multiple channels, and each channel can belong to multiple groups.

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 *
bl = Block()

# creating individual channel
all_rc= [ ]
for i in range(16):
    rc = RecordingChannel( index= i, name ='rc %d' %i)
    all_rc.append(rc)

# the four tetrodes
for i in range(4):
    rcg = RecordingChannelGroup( name = 'Tetrode %d' % i )
    for rc in all_rc[i*4:(i+1)*4]:
        rcg.recordingchannels.append(rc)
        rc.recordingchannelgroups.append(rcg)
    bl.recordingchannelgroups.append(rcg)

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

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

```
from neo import *
bl = Block(name='probe data')

# create individual channels
all_rc = []
for i in range(4):
    rc = RecordingChannel(index=i, name='channel %d' % i)
    all_rc.append(rc)

# one group for each neuron
rcg0 = RecordingChannelGroup(name='Group 0', index=0)
for i in [0, 1, 2]:
    rcg1.recordingchannels.append(all_rc[i])
    rc[i].recordingchannelgroups.append(rcg0)
bl.recordingchannelgroups.append(rcg0)

rcg1 = RecordingChannelGroup(name='Group 1', index=1)
for i in [1, 2, 3]:
    rcg1.recordingchannels.append(all_rc[i])
    rc[i].recordingchannelgroups.append(rcg1)
bl.recordingchannelgroups.append(rcg1)

# now we add the spiketrain from Unit 0 to rcg0
# and add the spiketrain from Unit 1 to rcg1
# ...
```

Note that because neurons are sorted from groups of channels in this situation, it is natural that the `RecordingChannelGroup` contains the `Unit` object. That unit then contains its spiketrains.

There are some shortcuts for IO writers to automatically create this structure based on 'channel_indexes' entries in the annotations for each spiketrain.

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

2.3 Neo diagram

Object:

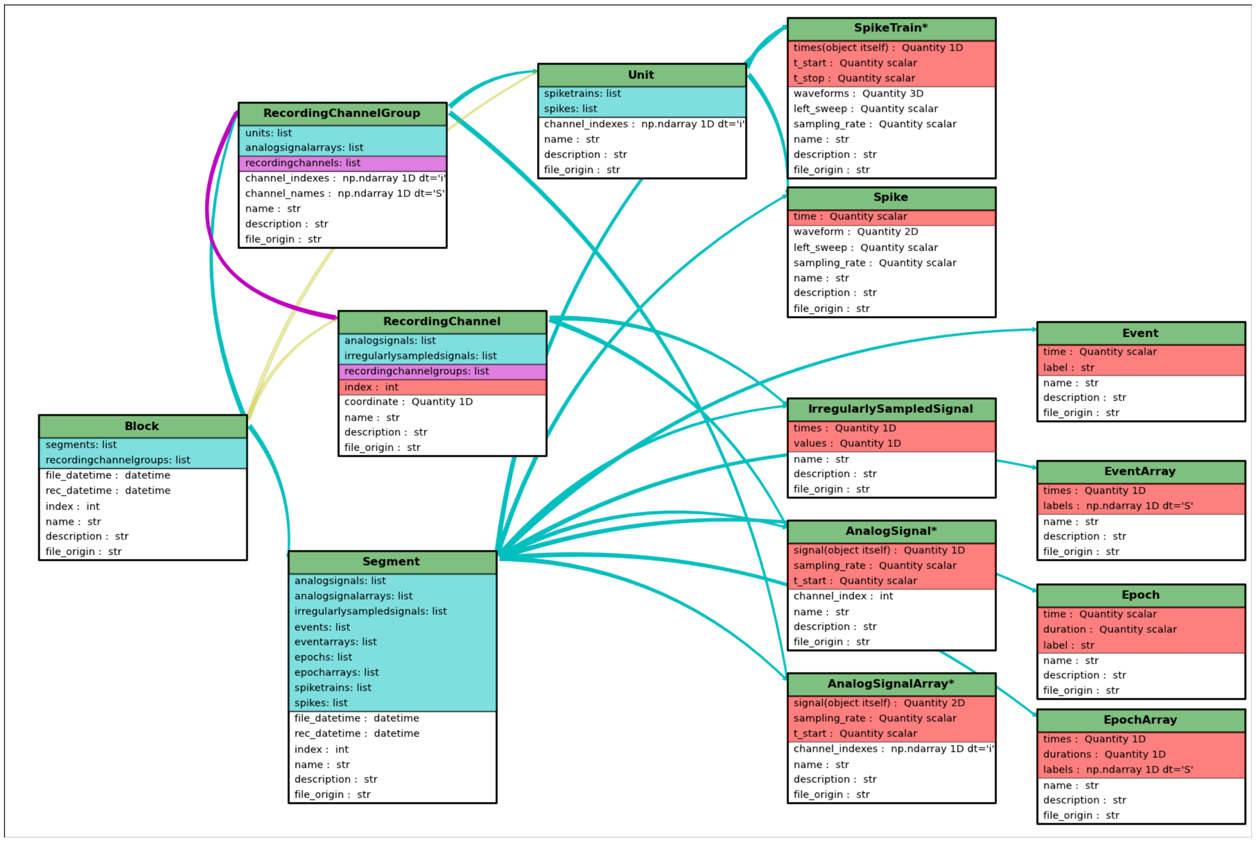
- With a star = inherits from `Quantity`

Attributes:

- In red = required
- In white = recommended

Relationship:

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



`Click here for a better quality SVG diagram`

For more details, see the *API Reference*.

2.4 Inheritance

Some Neo objects (`AnalogSignal`, `SpikeTrain`, `AnalogSignalArray`) inherit from `Quantity`, which in turn inherits from NumPy `ndarray`. This means that a Neo `AnalogSignal` actually 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.

2.5 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`.

2.5.1 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.

2.5.2 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, NumPy arrays) of simple types, where the simple types are `integer`, `float`, `complex`, `Quantity`, `string`, `date`, `time` and `datetime`.

See *specific_annotations*

CHAPTER

THREE

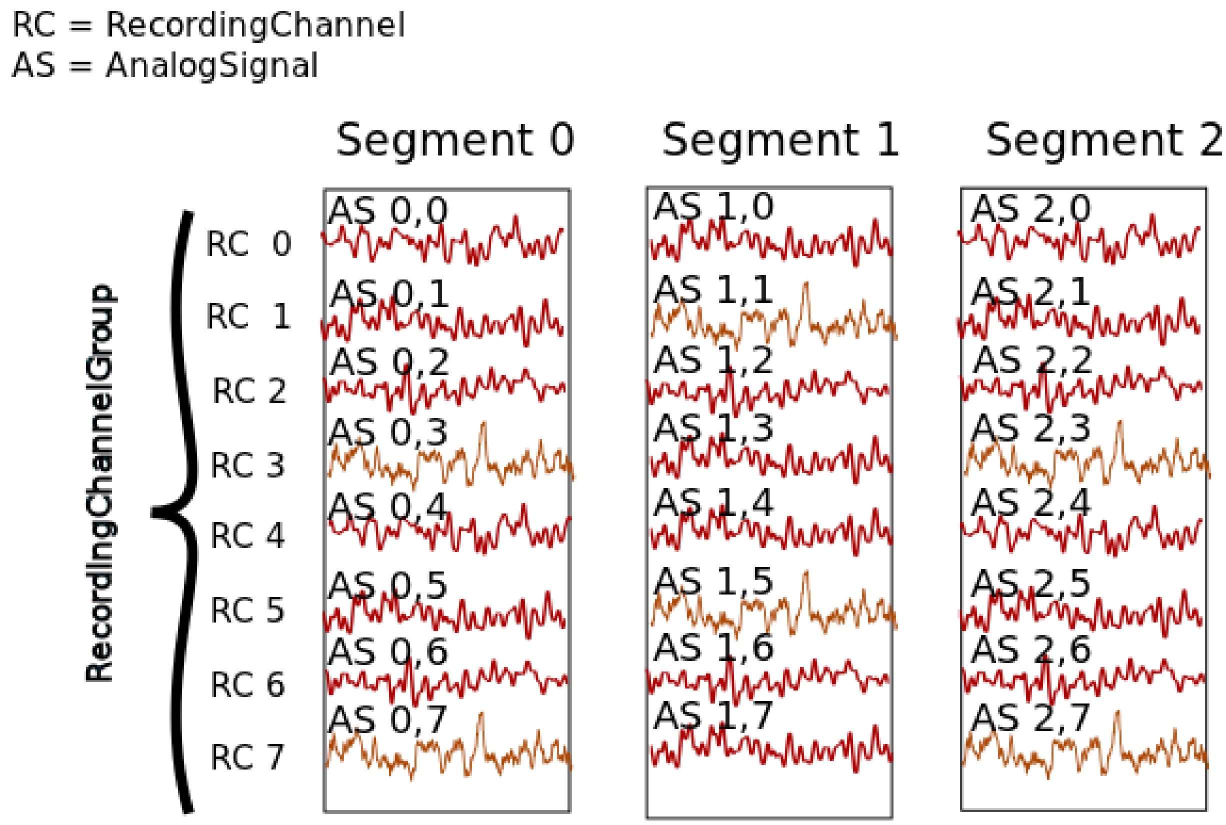
TYPICAL USE CASES

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, each represented by an `AnalogSignal` object.

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

- 3 `Segment` objects, each representing data from a single trial,
- 1 `RecordingChannelGroup`, composed of 8 `RecordingChannel` objects.



`Segment` and `RecordingChannel` 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)

    siglist = seg.analogsignals
    avg = np.mean(siglist, axis=0)

    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 RecordingChannelGroup
rcg = block.recordingchannelgroups[0]:
for rc in rcg.recordingchannels:
    print("Analyzing channel %d: %s", (rc.index, rc.name))

    siglist = rc.analogsignals
    avg = np.mean(siglist, axis=0)

    plt.figure()
    plt.plot(avg)
    plt.title("Average response on channel %d: %s' % (rc.index, rc.name)
```

Note that `Block.list_recordingchannels` is a property that gives direct access to all `RecordingChannels`, so the two first lines:

```
rcg = block.recordingchannelgroups[0]:
for rc in rcg.recordingchannels:
```

could be written as:

```
for rc in block.list_recordingchannels:
```

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 channel 5 on even numbered trials during the experiment and you want to follow up. What was the average response?

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

Here we have assumed that segment are temporally ordered in a `block.segments` and that signals are ordered by channel number in `seg.analogsignals`. It would be safer, however, to avoid assumptions by explicitly testing the `index` attribute of the `RecordingChannel` and `Segment` objects. One way to do this is to loop over the recording channels and access the segments through the signals (each `AnalogSignal` contains a reference to the `Segment` it is contained in).

```
siglist = []
rcg = block.recordingchannelgroups[0]:
for rc in rcg.recordingchannels:
    if rc.index == 5:
        for anasig in rc.analogsignals:
            if anasig.segment.index % 2 == 0:
                siglist.append(anasig)
avg = np.mean(siglist)
```

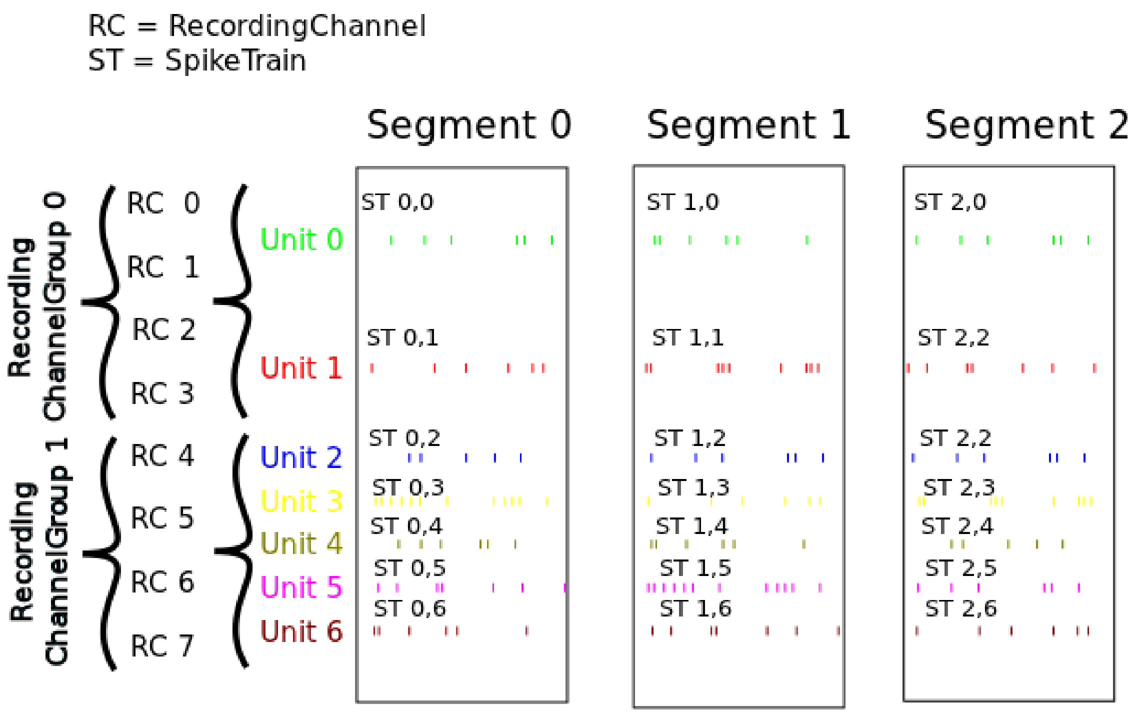
3.2 Recording spikes from multiple tetrodes

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

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

- 3 `Segments` (one per trial).
- 2 `RecordingChannelGroups` (one per tetrode), which contain:
  - 4 `RecordingChannels` each
  - 2 `Unit` objects (= 2 neurons) for the first `RecordingChannelGroup`
  - 5 `Units` for the second `RecordingChannelGroup`.

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



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

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

By Segment

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

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

By Unit

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

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

By RecordingChannelGroup

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

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

3.3 Spike sorting

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

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

```
# generate some fake data
rcg = RecordingChannelGroup()
for n in range(4):
    rcg.recordingchannels.append(neo.RecordingChannel())
    rcg.recordingchannels[n].analogsignals.append(
        AnalogSignal([.1, -2.0, .1, -.1, -.1, -3.0, .1, .1],
            sampling_rate=1000*Hz, units='V'))

# extract spike trains from each channel
st_list = []
for n in range(len(rcg.recordingchannels[0].analogsignals)):
    sigarray = np.array(
        [rcg.recordingchannels[m].analogsignals[n] for m in range(4)])
    # use a simple threshhold detector
    spike_mask = np.where(np.min(sigarray, axis=0) < -1.0 * pq.V)[0]

    # create a spike train
    anasig = rcg.recordingchannels[m].analogsignals[n]
    spike_times = anasig.times[spike_mask]
    st = neo.SpikeTrain(spike_times, t_start=anasig.t_start,
        anasig.t_stop)

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

    st_list.append(st)
```

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

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

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

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

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

CHAPTER

FOUR

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.

`neo.io` can be seen as a *pure-Python* and open-source Neuroshare replacement.

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

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

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

4.2 Introduction

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

A `Block` is 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`, `AnaloSignalArray`, `EpochArray`, `EventArray` (basically, all the different Neo objects).

Depending on the file format, these objects can sometimes be loaded separately, without the need to load the whole file. If possible, a file IO therefore provides distinct methods allowing to load only particular objects that may be present in the file. The basic idea of each IO file format is to have, as much as possible, read/write methods for the individual encapsulated objects, and otherwise to provide a read/write method that will return the object at the highest level of hierarchy (by default, a `Block` or a `Segment`).

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

4.3 One format = one class

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

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

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

4.4 Modes

IO can be based on file, directory, database or fake This is describe in 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 `Block` object:

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

4.6 Lazy and cascade options

In some cases you may not want to load everything in memory because it could be too big. For this scenario, two options are available:

- `lazy=True/False`. With `lazy=True` all arrays will have a size of zero, but all the metadata will be loaded. lazy_shape attribute is added to all object that inheritate Quantitities or numpy.ndarray (AnalogSignal, AnalogSignalArray, SpikeTrain) and to object that have array like attributes (EpochArray, EventArray) In that cases, lazy_shape is a tuple that have the same shape with lazy=False.
- `cascade=True/False`. With `cascade=False` only one object is read (and *one_to_many* and *many_to_many* relationship are not read).

By default (if they are not specified), `lazy=False` and `cascade=True`, i.e. all data is loaded.

Example cascade::

```
>>> seg = reader.read_segment( cascade=True)
>>> print(len(seg.analogsignals)) # this is N
>>> seg = reader.read_segment(cascade=False)
>>> print(len(seg.analogsignals)) # this is zero
```

Example lazy::

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

4.7 Details of API

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

- each file format has an IO class (for example for Spike2 files you have a `Spike2IO` class).
- each IO class inherits from the `BaseIO` class.
- each IO class can read or write directly one or several Neo objects (for example `Segment`, `Block`, ...): see the `readable_objects` and `writable_objects` attributes of the IO class.
- each IO class supports part of the `neo.core` hierachy, though not necessarily all of it (see `supported_objects`).
- each IO class has a `read()` method that returns a `Block` even if there is only one `Segment` and one `AnalogSignal` inside.
- each IO is able to do a *lazy* load: all metadata (e.g. `sampling_rate`) are read, but not the actual numerical data. lazy_shape attribute is added to provide information on real size.
- each IO is able to do a *cascade* load: if `True` (default) all child objects are loaded, otherwise only the top level object is loaded.

- each IO is able to save and load all required attributes (metadata) of the objects it supports.
- each IO can freely add user-defined or manufacturer-defined metadata to the `annotations` attribute of an object.

4.8 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 the classes list of succesfully imported io.

class `neo.io.`**`PlexonIO`**(*filename=None*)
Class for reading plx file.

Usage:

```
>>> from neo import io
>>> r = io.PlexonIO(filename='File_plexon_1.plx')
>>> seg = r.read_segment(lazy=False, cascade=True)
>>> print seg.analogsignals
[]
>>> print seg.spiketrains
[<SpikeTrain(array([  2.75000000e-02,   5.68250000e-02,   8.52500000e-02, ...,
...
>>> print seg.eventarrays
[]
```

class `neo.io.`**`NeuroExplorerIO`**(*filename=None*)
Class for reading nex file.

Usage:

```
>>> from neo import io
>>> r = io.NeuroExplorerIO(filename='File_neuroexplorer_1.nex')
>>> seg = r.read_segment(lazy=False, cascade=True)
>>> print seg.analogsignals
[<AnalogSignal(array([ 39.0625    ,   0.        ,   0.        , ..., -26.85546875, ...
>>> print seg.spiketrains
[<SpikeTrain(array([  2.29499992e-02,   6.79249987e-02,   1.13399997e-01, ...
>>> print seg.eventarrays
[<EventArray: @21.1967754364 s, @21.2993755341 s, @21.350725174 s, @21.5048999786 s, ...
>>> print seg.epocharrays
[<neo.core.epocharray.EpochArray object at 0x10561ba90>, <neo.core.epocharray.EpochArray obj
```

class `neo.io.`**`AxonIO`**(*filename=None*)
Class for reading abf (axon binary file) file.

Usage:

```
>>> from neo import io
>>> r = io.AxonIO(filename='File_axon_1.abf')
>>> bl = r.read_block(lazy=False, cascade=True)
>>> print bl.segments
[<neo.core.segment.Segment object at 0x105516fd0>]
>>> print bl.segments[0].analogsignals
[<AnalogSignal(array([ 2.18811035,  2.19726562,  2.21252441, ...,  1.33056641,
        1.3458252 ,  1.3671875 ], dtype=float32) * pA, [0.0 s, 191.2832 s], sampling rate: 1
```

```
>>> print bl.segments[0].eventarrays
[]
```

class `neo.io.`**`TdtIO`**(*dirname=None*)
Class for reading data from from Tucker Davis TTank format.

Usage:

```
>>> from neo import io
>>> r = io.TdtIO(dirname='aep_05')
>>> bl = r.read_block(lazy=False, cascade=True)
>>> print bl.segments
[<neo.core.segment.Segment object at 0x1060a4d10>]
>>> print bl.segments[0].analogsignals
[<AnalogSignal(array([ 2.18811035,  2.19726562,  2.21252441, ...,  1.33056641,
        1.3458252 ,  1.3671875 ], dtype=float32) * pA, [0.0 s, 191.2832 s], sampling rate: 1
>>> print bl.segments[0].eventarrays
[]
```

class `neo.io.`**`WinEdrIO`**(*filename=None*)
Class for reading data from WinEDR.

Usage:

```
>>> from neo import io
>>> r = io.WinEdrIO(filename='File_WinEDR_1.EDR')
>>> seg = r.read_segment(lazy=False, cascade=True,)
>>> print seg.analogsignals
[<AnalogSignal(array([ 89.21203613,  88.83666992,  87.21008301, ...,  64.56298828,
        67.94128418,  68.44177246], dtype=float32) * pA, [0.0 s, 101.5808 s], sampling rate:
```

class `neo.io.`**`WinWcpIO`**(*filename=None*)
Class for reading from a WinWCP file.

Usage:

```
>>> from neo import io
>>> r = io.WinWcpIO( filename = 'File_winwcp_1.wcp')
>>> bl = r.read_block(lazy = False, cascade = True,)
>>> print bl.segments
[<neo.core.segment.Segment object at 0x1057bd350>, <neo.core.segment.Segment object at 0x105
...
>>> print bl.segments[0].analogsignals
[<AnalogSignal(array([-2438.73388672, -2428.96801758, -2425.61083984, ..., -2695.39453125,
...
```

class `neo.io.`**`ElanIO`**(*filename=None*)
Classe for reading/writing data from Elan.

Usage:

```
>>> from neo import io
>>> r = io.ElanIO( filename = 'File_elan_1.eeg')
>>> seg = r.read_segment(lazy = False, cascade = True,)
>>> print seg.analogsignals
[<AnalogSignal(array([ 89.21203613,  88.83666992,  87.21008301, ...,  64.56298828,
    67.94128418,  68.44177246], dtype=float32) * pA, [0.0 s, 101.5808 s], sampling rate: 100
>>> print seg.spiketrains
[]
>>> print seg.eventarrays
[]
```

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

Usage:

```
>>> from neo import io
>>> r = io.AsciiSignalIO(filename='File_asciisignal_2.txt')
>>> seg = r.read_segment(lazy=False, cascade=True)
>>> print seg.analogsignals
[<AnalogSignal(array([ 39.0625    ,   0.        ,   0.        , ..., -26.85546875 ...
```

class `neo.io.AsciiSpikeTrainIO`(*filename=None*)
Classe for reading/writing SpikeTrain 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(lazy = False, cascade = True,)
>>> print seg.spiketrains
[<SpikeTrain(array([ 3.89981604,  4.73258781,  0.608428  ,  4.60246277,  1.23805797,
...
```

class `neo.io.RawBinarySignalIO`(*filename=None*)
Class for reading/writing data in a raw binary interleaved compact file.

Usage:

```
>>> from neo import io
>>> r = io.RawBinarySignalIO( filename = 'File_ascii_signal_2.txt')
>>> seg = r.read_segment(lazy = False, cascade = True,)
>>> print seg.analogsignals
...
```

class `neo.io.MicromedIO`(*filename=None*)
Class for reading data from micromed (.trc).

Usage:

```
>>> from neo import io
>>> r = io.MicromedIO(filename='File_micromed_1.TRC')
>>> seg = r.read_segment(lazy=False, cascade=True)
>>> print seg.analogsignals
[<AnalogSignal(array([ -1.77246094e+02,  -2.24707031e+02,  -2.66015625e+02,
...
```

class `neo.io.NeuroshareIO`(*filename='', dllname=''*)
Class for reading file trougth neuroshare API. The user need the DLLs in the path of the file format.

Usage:

```
>>> from neo import io
>>> r = io.NeuroshareIO(filename='a_file', dllname=the_name_of_dll)
>>> seg = r.read_segment(lazy=False, cascade=True, import_neuroshare_segment=True)
>>> print seg.analogsignals
[<AnalogSignal(array([ -1.77246094e+02,  -2.24707031e+02,  -2.66015625e+02,
...
>>> print seg.spiketrains
[]
>>> print seg.eventarrays
[<EventArray: 1@1.12890625 s, 1@2.02734375 s, 1@3.82421875 s>]
```

Note: neuroshare.ns_ENTITY_EVENT: are converted to neo.EventArray neuroshare.ns_ENTITY_ANALOG: are converted to neo.AnalogSignal neuroshare.ns_ENTITY_NEURALEVENT: are converted to neo.SpikeTrain

neuroshare.ns_ENTITY_SEGMENT: is something between serie of small AnalogSignal and Spike-train with associated waveforms. It is arbitrarily converted as SpikeTrain.

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

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

class `neo.io.`**`BlackrockIO`**(*filename*, *full_range=array(8192.0) * mV*)
Class for reading/writing data in a BlackRock Neuroshare ns5 files.

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(lazy = False, cascade = True)
>>> print blck.segments[0].analogsignals
```

class `neo.io.`**`PickleIO`**(*filename=None*, ***kargs*)

class `neo.io.`**`BrainVisionIO`**(*filename=None*)
Class for reading/writing data from BrainVision product (brainAmp, brain analyser...)

Usage:

```
>>> from neo import io
>>> r = io.BrainVisionIO( filename = 'File_brainvision_1.eeg')
>>> seg = r.read_segment(lazy = False, cascade = True,)
```

class `neo.io.`**`ElphyIO`**(*filename=None*)
Class for reading from and writing to an Elphy file.

It enables reading: `- Block - Segment - RecordingChannel - RecordingChannelGroup - EventArray - SpikeTrain`

Usage:

```
>>> from neo import io
>>> r = io.ElphyIO(filename='ElphyExample.DAT')
>>> seg = r.read_block(lazy=False, cascade=True)
>>> print(seg.analogsignals)
>>> print(seg.spiketrains)
>>> print(seg.eventarrays)
>>> print(anasig._data_description)
>>> anasig = r.read_analogsignal(lazy=False, cascade=False)

>>> bl = Block()
>>> ... creating segments, their contents and append to bl
>>> r.write_block( bl )
```

4.9 If you want to develop your own IO

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

CHAPTER

FIVE

EXAMPLES

5.1 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 neo
import urllib




# Plexon files
distantfile = 'https://portal.g-node.org/neo/plexon/File_plexon_3.plx'
localfile = './File_plexon_3.plx'
urllib.urlretrieve(distantfile, localfile)

#create a reader
reader = neo.io.PlexonIO(filename = 'File_plexon_3.plx')
# read the block
bl = reader.read(cascade = True, lazy = False)
print bl
# acces to segments
for seg in bl.segments:
    print seg
    for asig in seg.analogsignals:
        print asig
    for st in seg.spiketrains:
        print st


# CED Spike2 files
distantfile = 'https://portal.g-node.org/neo/spike2/File_spike2_1.smr'
localfile = './File_spike2_1.smr'
urllib.urlretrieve(distantfile, localfile)

#create a reader
reader = neo.io.Spike2IO(filename = 'File_spike2_1.smr')
# read the block
bl = reader.read(cascade = True, lazy = False)
print bl
```

```
# acces 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 plotting neo object with maplotlib.
"""

import neo
import urllib
from matplotlib import pyplot
import numpy

distantfile = 'https://portal.g-node.org/neo/neuroexplorer/File_neuroexplorer_2.nex'
localfile = 'File_neuroexplorer_2.nex'
urllib.urlretrieve(distantfile, localfile)


reader = neo.io.NeuroExplorerIO(filename = 'File_neuroexplorer_2.nex')
bl = reader.read(cascade = True, lazy = False)
for seg in bl.segments:
    fig = pyplot.figure()
    ax1 = fig.add_subplot(2,1,1)
    ax2 = fig.add_subplot(2,1,2, sharex = ax1)
    ax1.set_title(seg.file_origin)
    for asig in seg.analogsignals:
        ax1.plot(asig.times, asig)
    for s,st in enumerate(seg.spiketrains):
        print st.units
        ax2.plot(st, s*numpy.ones(st.size), linestyle = 'None',
                    marker = '|', color = 'k')
pyplot.show()
```

CHAPTER

SIX

API REFERENCE

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:

TODO

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

***Recommended attributes/properties*:**

name A label for the dataset

description text description

file_origin filesystem path or URL of the original data file.

file_datetime the creation date and time of the original data file.

rec_datetime the date and time of the original recording

index integer. You can use this to define an ordering of your Block. It is not used by Neo in any way.

***Container of*:** `Segment` `RecordingChannelGroup`

Properties

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

list_recordingchannels: descends through hierarchy and returns a list of `RecordingChannel` existing in the block.

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

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:

TODO

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

***Recommended attributes/properties*:**

name A label for the dataset

description text description

file_origin filesystem path or URL of the original data file.

file_datetime the creation date and time of the original data file.

rec_datetime the date and time of the original recording

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

***Container of*:** `Epoch` `EpochArray` `Event` `EventArray` `AnalogSignal` `AnalogSignalArray` `IrregularlySampledSignal` `Spike` `SpikeTrain`

class `neo.core.`**`RecordingChannelGroup`**(*channel_names=None*, *channel_indexes=None*, *name=None*, *description=None*, *file_origin=None*, ***annotations*)

This container have sereval purpose:

- Grouping all `AnalogSignalArray` inside a `Block` across `Segment`
- Grouping `RecordingChannel` inside a `block`. This case is *many to many* relation. It mean that a `RecordingChannel` can belong to several group. A typical use case is tetrode (4 X `RecordingChannel` inside a `RecordingChannelGroup`).
- Container of `Unit`. A neuron decharge (`Unit`) can be seen by several electrodes (4 in tetrode case).

Usage 1 multi segment recording with 2 electrode array:

```
bl = Block()
# create a block with 3 Segment and 2 RecordingChannelGroup
for s in range(3):
    seg = Segment(name = 'segment %d' %s, index = s)
    bl.segments.append(seg)

for r in range(2):
    rcg = RecordingChannelGroup('Array probe %d' % r, channel_indexes =
                                arange(64) )
    bl.recordingchannelgroups.append(rcg)

# populating AnalogSignalArray
for s in range(3):
    for r in range(2):
        a = AnalogSignalArray( randn(10000, 64), sampling_rate =
                              10*pq.kHz )
        bl.recordingchannelgroups[r].append(a)
        bl.segments[s].append(a)
```

Usage 2 grouping channel:

```
bl = Block()
# Create a new RecordingChannelGroup and add to current block
rcg = RecordingChannelGroup(channel_names=['ch0', 'ch1', 'ch2'])
rcg.channel_indexes = [0, 1, 2]
bl.recordingchannelgroups.append(rcg)

for i in range(3):
    rc = RecordingChannel(index=i)
    rcg.recordingchannels.append(rc) # <- many to many relationship
    rc.recordingchannelgroups.append(rcg) # <- many to many
                                             relationship
```

Usage 3 dealing with Units:

```
bl = Block()
rcg = RecordingChannelGroup( name = 'octotrode A')
bl.recordingchannelgroups.append(rcg)

# create several Units
for i in range(5):
    u = Unit(name = 'unit %d' % i, description =
             'after a long and hard spike sorting')
    rcg.append(u)
```

***Required attributes*:** None

***Recommended attributes*:**

channel_names List of strings naming each channel

channel_indexes List of integer indexes of each channel

name string

description string

file_origin string

***Container of*:** RecordingChannel AnalogSignalArray Unit

class neo.core.**RecordingChannel**(*index=0*, *coordinate=None*, *name=None*, *description=None*, *file_origin=None*, ***annotations*)

A RecordingChannel is a container for AnalogSignal objects that come from the same logical and/or physical channel inside a Block.

Note that a RecordingChannel can belong to several RecordingChannelGroup.

Usage one Block with 3 Segment and 16 RecordingChannel and 48 AnalogSignal:

```
bl = Block()
# Create a new RecordingChannelGroup and add to current block
rcg = RecordingChannelGroup(name = 'all channels)
bl.recordingchannelgroups.append(rcg)

for c in range(16):
    rc = RecordingChannel(index=c)
    rcg.recordingchannels.append(rc) # <- many to many relationship
    rc.recordingchannelgroups.append(rcg) # <- many to many
                                             relationship

for s in range(3):
    seg = Segment(name = 'segment %d' %s, index = s)
    bl.segments.append(seg)

for c in range(16):
    for s in range(3):
        anasig = AnalogSignal( np.rand(100000), sampling_rate =
                               20*pq.Hz)
        bl.segments[s].analogsignals.append(anasig)
        rcg.recordingchannels[c].analogsignals.append(anasig)
```

***Required attributes/properties*:**

index (int) Index of the channel

***Recommended attributes/properties*:**

coordinate (Quantity) x, y, z

name string

description string

file_origin string

***Container of*:** AnalogSignal IrregularlySampledSignal

class neo.core.**AnalogSignal**(*signal*, *units=None*, *dtype=None*, *copy=True*, *t_start=array(0.0) * s*, *sampling_rate=None*, *sampling_period=None*, *name=None*, *file_origin=None*, *description=None*, *channel_index=None*, ***annotations*)

A representation of a continuous, analog signal acquired at time `t_start` at a certain sampling rate.

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

Usage:

```
>>> from quantities import ms, kHz, nA, uV
>>> import numpy as np
>>> a = AnalogSignal([1,2,3], sampling_rate=0.42*kHz, units='mV')
>>> b = AnalogSignal([4,5,6]*nA, sampling_period=42*ms)
>>> c = AnalogSignal(np.array([1.0, 2.0, 3.0]), t_start=42*ms,
...                  sampling_rate=0.42*kHz, units=uV)
```

***Required attributes/properties*:**

signal the data itself, as a `Quantity` array, NumPy array or list

units required if the signal is a list or NumPy array, not if it is a `Quantity`

sampling_rate *or* :sampling_period: Quantity, number of samples per unit time or interval between two samples. If both are specified, they are checked for consistency.

***Recommended attributes/properties*:**

t_start Quantity, time when signal begins. Default: 0.0 seconds

Note that the length of the signal array and the sampling rate are used to calculate `t_stop` and `duration`.

name string

description string

file_origin string

***Optional arguments*:**

dtype

copy (bool) True by default

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

```
>>> a = AnalogSignal([1,2,3], day='Monday')
>>> print a.annotations['day']
Monday
```

***Properties available on this object*:** `sampling_rate`, `sampling_period`, `t_stop`, `duration`

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

class `neo.core.`**`AnalogSignalArray`**(*signal*, *units=None*, *dtype=None*, *copy=True*, *t_start=array(0.0) * s*, *sampling_rate=None*, *sampling_period=None*, *name=None*, *file_origin=None*, *description=None*, *channel_index=None*, ***annotations*)

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

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

***Usage*:** TODO

***Required attributes/properties*:**

t_start time when signal begins

sampling_rate *or* :sampling_period: Quantity, number of samples per unit time or interval between two samples. If both are specified, they are checked for consistency.

***Properties*:**

sampling_period interval between two samples (1/sampling_rate)

duration signal duration (size * sampling_period)

t_stop time when signal ends (t_start + duration)

***Recommended attributes/properties*:**

name

description

file_origin

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

A representation of a continuous, analog signal acquired at time `t_start` with a varying sampling interval.

Usage:

```
>>> from quantities import ms, nA, uV
>>> import numpy as np
>>> a = IrregularlySampledSignal([0.0, 1.23, 6.78], [1,2,3], units='mV', time_units='ms')
>>> b = IrregularlySampledSignal([0.01, 0.03, 0.12]*s, [4,5,6]*nA)
```

***Required attributes/properties*:**

times NumPy array, Quantity array or list

signal Numpy array, Quantity array or list of the same size as times

units required if the signal is a list or NumPy array, not if it is a `Quantity`

time_units required if *times* is a list or NumPy array, not if it is a `Quantity`

***Optional arguments*:**

> **dtype** Data type of the signal (times are always floats)

***Recommended attributes/properties*:**

> **name**
>
> **description**
>
> **file_origin**

class `neo.core.`**`Event`**(*time*, *label*, *name=None*, *description=None*, *file_origin=None*, ***annotations*)

Object to represent an event occurring at a particular time. Useful for managing trigger, stimulus, ...

Usage:

***Required attributes/properties*:**

> **time** (quantity):
>
> **label** (str):

***Recommended attributes/properties*:**

> **name**
>
> **description**
>
> **file_origin**

`EventArray(times=array([], dtype=float64) * s, labels=array([], dtype='|S1'), name=None, description=None, file_origin=None, **annotations)`

Array of events. Introduced for performance reasons. An `EventArray` is prefered to a list of `Event` objects.

***Usage*:** TODO

***Required attributes/properties*:**

> **times** (quantity array 1D)
>
> **labels** (numpy.array 1D dtype='S')

***Recommended attributes/properties*:**

> **name**
>
> **description**
>
> **file_origin**

class `neo.core.`**`Epoch`**(*time*, *duration*, *label*, *name=None*, *description=None*, *file_origin=None*, ***annotations*)

Similar to `Event` but with a duration. Useful for describing a period, the state of a subject, ...

***Usage*:** TODO

***Required attributes/properties*:**

> **time** (quantity)
>
> **duration** (quantity)
>
> **label** (str)

Recommended attributes/properties:

> **name**
>
> **description**
>
> **file_origin**

EpochArray(times=array([], dtype=float64) * s, durations=array([], dtype=float64) * s, labe
dtype='|S1'), name=None, description=None, file_origin=None, **annotations)

Array of epochs. Introduced for performance reason. An `EpochArray` is prefered to a list of `Epoch` objects.

***Usage*:** TODO

***Required attributes/properties*:**

> **times** (quantity array 1D)
>
> **durations** (quantity array 1D)
>
> **labels** (numpy.array 1D dtype='S'))

***Recommended attributes/properties*:**

> **name**
>
> **description**
>
> **file_origin**

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

A `Unit` regroups all the `SpikeTrain` objects that were emitted by a neuron during a `Block`. The spikes may come from different `Segment` objects within the `Block`, so this object is not contained in the usual `Block`/`Segment` /`SpikeTrain` hierarchy.

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

This replaces the `Neuron` class in the previous version of Neo.

Usage:

```
# Store the spike times from a pyramidal neuron recorded on channel 0
u = neo.Unit(name='pyramidal neuron')

# first segment
st1 = neo.SpikeTrain(times=[.01, 3.3, 9.3], units='sec')
u.spiketrains.append(st1)

# second segment
st2 = neo.SpikeTrain(times=[100.01, 103.3, 109.3], units='sec')
u.spiketrains.append(st2)
```

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

***Recommended attributes/properties*:**

> **name**
>
> **description**
>
> **file_origin**

***Container of*:** `SpikeTrain` `Spike`

class `neo.core.`**`Spike`**(*time=array(0.0) * s*, *waveform=None*, *sampling_rate=None*, *left_sweep=None*, *name=None*, *description=None*, *file_origin=None*, ***annotations*)

Object to represent one spike emitted by a `Unit` and represented by its time occurence and optional waveform.

***Usage*:** TODO

***Required attributes/properties*:**

time (quantity)

***Recommended attributes/properties*:**

waveform (quantity 2D (channel_index X time))

sampling_rate

left_sweep

name

description

file_origin

***Properties*:**

right_sweep

duration

class `neo.core.`**`SpikeTrain`**(*times*, *t_stop*, *units=None*, *dtype=<type 'float'>*, *copy=True*, *sampling_rate=array(1.0) * Hz*, *t_start=array(0.0) * s*, *waveforms=None*, *left_sweep=None*, *name=None*, *file_origin=None*, *description=None*, ***annotations*)

SpikeTrain is a `Quantity` array of spike times.

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

***Required arguments*:**

times a list, 1d numpy array, or quantity array, containing the times of each spike.

t_stop time at which SpikeTrain ends. This will be converted to the same units as the data. 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.

Your spike times must have units. Preferably, *times* is a Quantity array with units of time. Otherwise, you must specify the keyword argument *units*.

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

***Recommended arguments*:**

t_start time at which SpikeTrain began. This will be converted to the same units as the data. Default is zero seconds.

waveforms the waveforms of each spike

sampling_rate the sampling rate of the waveforms

left_sweep Quantity, in units of time. Time from the beginning of the waveform to the trigger time of the spike.

sort if True, the spike train will be sorted

name string

description string

file_origin string

Any other keyword arguments are stored in the `self.annotations` dict.

Other arguments relating to implementation `dtype` : data type (float32, float64, etc) `copy` : boolean, whether to copy the data or use a view.

These arguments, as well as *units*, are simply passed to the Quantity constructor.

Note that *copy* must be True when you request a change of units or dtype.

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. Note that t_start and t_stop are not changed automatically, though you can still manually change them.

Example::

```
>>> st = SpikeTrain([3,4,5] * pq.s, t_stop=10.0)
>>> st2 = st[1:3]
>>> st.t_start
array(0.0) * s
>>> st2
<SpikeTrain(array([ 4.,  5.]) * s, [0.0 s, 10.0 s])>
```

CHAPTER

SEVEN

RELEASE NOTES

7.1 What's new in version 0.2.1?

- assorted bug fixes
- added `time_slice()` method to the `SpikeTrain` and `AnalogSignalArray` classes.
- improvements to annotation data type handling
- added PickleIO, allowing saving Neo objects in the Python pickle format.
- added ElphyIO (see http://www.unic.cnrs-gif.fr/software.html)
- added BrainVisionIO (see http://www.brainvision.com/)
- improvements to PlexonIO
- added `merge()` method to the `Block` and `Segment` classes
- development was mostly moved to GitHub, although the issue tracker is still at neuralensemble.org/neo

7.2 What's new in version 0.2?

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

EIGHT

DEVELOPERS' GUIDE

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

8.1 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.

8.2 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 follow the RSS feed.

8.3 Requirements

- Python 2.6, 2.7, 3.1 or 3.2
- numpy >= 1.3.0
- quantities >= 0.9.0
- nose >= 0.11.1
- if using Python 2.6 or 3.1, unittest2 >= 0.5.1
- Distribute >= 0.6
- Sphinx >= 0.6.4
- (optional) tox >= 0.9 (makes it easier to test with multiple Python versions)
- (optional) coverage >= 2.85 (for measuring test coverage)

8.4 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 https://github.com/NeuralEnsemble/python-neo (see http://help.github.com/fork-a-repo/).

To get a local copy of the repository:

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

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

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

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

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

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

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

To update to the latest version from the repository:

```
$ git pull
```

8.5 Running the test suite

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

```
$ cd neo/test
```

With Python 2.7 or 3.2:

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

If you have nose installed:

```
$ nosetests
```

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

To run tests from an individual file:

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

8.6 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
```

8.7 Working on the documentation

The documentation is written in reStructuredText, using the Sphinx documentation system. To build the documentation:

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

Then open *some/directory/neo_trunk/doc/build/html/index.html* in your browser.

8.8 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**. Then you can commit them to your local repository:

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

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

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

```
$ git push
```

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

8.9 Python 3

Neo core should work with both recent versions of Python 2 (versions 2.6 and 2.7) and Python 3. Neo IO modules should ideally work with both Python 2 and 3, but certain modules may only work with one or the other (see :doc:install).

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

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

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

8.10 Coding standards and style

All code should conform as much as possible to PEP 8, and should run with Python 2.6, 2.7, 3.1 and 3.2.

8.11 Making a release

First check that the version string (in `neo/version.py`, `setup.py` and `doc/conf.py`) is correct.

To build a source package:

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

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

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

Finally, tag the release in the Git repository:

```
$ git tag <version>
```

8.12 If you want to develop your own IO module

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

CHAPTER

NINE

IO DEVELOPERS' GUIDE

9.1 Guidelines for IO implementation

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

1. Fully understand the object model. See *Neo core*. If in doubt ask the mailing list.
2. Fully understand `neo.io.exampleio`, It is a fake IO to explain the API. If in doubt ask the list.
3. Copy/paste `exampleio.py` and choose clear file and class names for your IO.
4. Decide which **supported objects** and **readable objects** your IO will deal with. This is the crucial point.
5. Implement all methods `read_XXX()` related to **readable objects**.
6. Do not forget all : lasy and cascade combination.
7. Write good docstrings. List dependencies, including minimum version numbers.
8. Add your class to `neo.io.__init__`. Keep the import inside try/except for dependency reasons.
9. Contact the Neo maintainers to put sample files for testing on the G-Node server (write access is not public).
10. Write tests in `neo/test/io/test_xxxxxio.py`. You must at least pass the standard tests (inherited from `BaseTestIO`).
11. Commit or send a patch only if all tests pass.

9.2 Miscellaneous

Notes:

- if your IO supports several version of a format (like ABF1, ABF2), upload to G-node test file repository all file version possible. (for utest coverage).
- `neo.io.tools.create_many_to_one_relationship()` offers a utility to complete the hierachy when all one-to-many relationships have been created.
- `neo.io.tools.populate_RecordingChannel()` offers a utility to create inside a `Block` all `RecordingChannel` objects and links to `AnalogSignal`, `SpikeTrain`, ...
- 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.

9.3 Tests

`neo.test.io.commun_io_test.BaseTestIO` provide standard tests. To use these you need to upload some sample data files at the G-Node portal. They will be publicly accessible for testing Neo. These tests:

- check the compliance with the schema: hierachy, attribute types, ...
- check if the IO respects the *lazy* and *cascade* keywords.
- For IO 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 the G-Node portal and store them locally in `neo/test/io/files_for_tests/`. Subsequent test runs use the previously downloaded files, rather than trying to download them each time.

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

```python
# encoding: utf-8
"""
Tests of io.axonio
"""
from __future__ import absolute_import

try:
    import unittest2 as unittest
except ImportError:
    import unittest

from ...io import AxonIO
from .common_io_test import BaseTestIO


class TestAxonIO(BaseTestIO, unittest.TestCase):
    files_to_test = ['File_axon_1.abf',
                     'File_axon_2.abf',
                     'File_axon_3.abf',
                     'File_axon_4.abf',
                     'File_axon_5.abf',
                     'File_axon_6.abf',


                        ]
    files_to_download = files_to_test
    ioclass = AxonIO


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

9.4 ExampleIO

class `neo.io.`**`ExampleIO`**(*filename=None*)

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

For the user, it generates a `Segment` or a `Block` with a sinusoidal `AnalogSignal`, a `SpikeTrain` and an `EventArray`.

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 IO API (*Details of API*)
- Follow *Guidelines for IO implementation*

Usage:

```
>>> from neo import io
>>> r = io.ExampleIO(filename='itisafake.nof')
>>> seg = r.read_segment(lazy=False, cascade=True)
>>> print(seg.analogsignals)
[<AnalogSignal(array([ 0.19151945,  0.62399373,  0.44149764, ...,  0.96678374,
...
>>> print(seg.spiketrains)
 [<SpikeTrain(array([ -0.83799524,   6.24017951,   7.76366686,   4.45573701,
    12.60644415,  10.68328994,   8.07765735,   4.89967804,
...
>>> print(seg.eventarrays)
[<EventArray: TriggerB@9.6976 s, TriggerA@10.2612 s, TriggerB@2.2777 s, TriggerA@6.8607 s, .
>>> anasig = r.read_analogsignal(lazy=True, cascade=False)
>>> print(anasig._data_description)
{'shape': (150000,)}
>>> anasig = r.read_analogsignal(lazy=False, cascade=False)
```

Here is the entire file:

```
# encoding: utf-8
"""
Class for "reading" fake data from an imaginary file.

For the user, it generates a :class:`Segment` or a :class:`Block` with a
sinusoidal :class:`AnalogSignal`, a :class:`SpikeTrain` and an
:class:`EventArray`.

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

Depends on: scipy

Supported: Read

Author: sgarcia

"""
from __future__ import absolute_import

# I need to subclass BaseIO
from .baseio import BaseIO

# to import from core
from ..core import Block, Segment, AnalogSignal, SpikeTrain, EventArray

# some tools to finalize the hierachy
from .tools import create_many_to_one_relationship

# note neo.core needs only numpy and quantities
import numpy as np
import quantities as pq
```

```
# but my specific IO can depend on many other packages
from numpy import pi, newaxis
import datetime
try:
    have_scipy = True
    from scipy import stats
    from scipy import randn, rand
    from scipy.signal import resample
except ImportError:
    have_scipy = False


# I need to subclass BaseIO
class ExampleIO(BaseIO):
    """
    Class for "reading" fake data from an imaginary file.

    For the user, it generates a :class:`Segment` or a :class:`Block` with a
    sinusoidal :class:`AnalogSignal`, a :class:`SpikeTrain` and an
    :class:`EventArray`.

    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 IO API (:ref:`neo_io_API`)
      * Follow :ref:`io_guiline`

    Usage:
        >>> from neo import io
        >>> r = io.ExampleIO(filename='itisafake.nof')
        >>> seg = r.read_segment(lazy=False, cascade=True)
        >>> print(seg.analogsignals)  # doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
        [<AnalogSignal(array([ 0.19151945,  0.62399373,  0.44149764, ...,  0.96678374,
        ...
        >>> print(seg.spiketrains)    # doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
         [<SpikeTrain(array([ -0.83799524,   6.24017951,   7.76366686,   4.45573701,
            12.60644415,  10.68328994,   8.07765735,   4.89967804,
        ...
        >>> print(seg.eventarrays)    # doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
        [<EventArray: TriggerB@9.6976 s, TriggerA@10.2612 s, TriggerB@2.2777 s, TriggerA@6.8607 s, .
        >>> anasig = r.read_analogsignal(lazy=True, cascade=False)
        >>> print(anasig._data_description)
        {'shape': (150000,)}
        >>> anasig = r.read_analogsignal(lazy=False, cascade=False)

    """

    is_readable = True # This class can only read data
    is_writable = False # write is not supported

    # This class is able to directly or indirectly handle the following objects
    # You can notice that this greatly simplifies the full Neo object hierarchy
    supported_objects  = [ Segment , AnalogSignal, SpikeTrain, EventArray ]

    # This class can return either a Block or a Segment
    # The first one is the default ( self.read )
    # These lists should go from highest object to lowest object because
    # common_io_test assumes it.
    readable_objects    = [ Segment , AnalogSignal, SpikeTrain ]
    # This class is not able to write objects
    writeable_objects   = [ ]

    has_header         = False
    is_streameable     = False

    # This is for GUI stuff : a definition for parameters when reading.
    # This dict should be keyed by object ('Block'). Each entry is a list
    # of tuple. The first entry in each tuple is the parameter name. The
    # second entry is a dict with keys 'value' (for default value),
    # and 'label' (for a descriptive name).
    # Note that if the highest-level object requires parameters,
    # common_io_test will be skipped.
    read_params = {
        Segment : [
            ('segment_duration',
                {'value' : 15., 'label' : 'Segment size (s.)'}),
            ('num_analogsignal',
                {'value' : 8, 'label' : 'Number of recording points'}),
            ('num_spiketrain_by_channel',
                {'value' : 3, 'label' : 'Num of spiketrains'}),
            ],
        }

    # do not supported write so no GUI stuff
    write_params       = None

    name               = 'example'

    extensions          = [ 'nof' ]

    # mode can be 'file' or 'dir' or 'fake' or 'database'
    # the main case is 'file' but some reader are base on a directory or a database
    # this info is for GUI stuff also
    mode = 'fake'



    def __init__(self , filename = None) :
        """


        Arguments:
            filename : the filename

        Note:
            - filename is here just for exampe because it will not be take in account
            - if mode=='dir' the argument should be dirname (See TdtIO)

        """
        BaseIO.__init__(self)
        self.filename = filename
        # Seed so all instances can return the same values
        np.random.seed(1234)
    # Segment reading is supported so I define this :
    def read_segment(self,
                     # the 2 first keyword arguments are imposed by neo.io API
                     lazy = False,
                     cascade = True,
                     # all following arguments are decied by this IO and are free
                     segment_duration = 15.,
                     num_analogsignal = 4,
                     num_spiketrain_by_channel = 3,
                    ):
        """
        Return a fake Segment.

        The self.filename does not matter.

        In this IO read by default a Segment.

        This is just a example to be adapted to each ClassIO.
        In this case these 3 paramters are  taken in account because this function
        return a generated segment with fake AnalogSignal and fake SpikeTrain.

        Parameters:
            segment_duration :is the size in secend of the segment.
            num_analogsignal : number of AnalogSignal in this segment
            num_spiketrain : number of SpikeTrain in this segment

        """

        sampling_rate = 10000. #Hz
        t_start = -1.


        #time vector for generated signal
        timevect = np.arange(t_start, t_start+ segment_duration , 1./sampling_rate)

        # create an empty segment
        seg = Segment( name = 'it is a seg from exampleio')

        if cascade:
            # read nested analosignal
            for i in range(num_analogsignal):
                ana = self.read_analogsignal( lazy = lazy , cascade = cascade ,
                                            channel_index = i ,segment_duration = segment_duration, t
                seg.analogsignals += [ ana ]

            # read nested spiketrain
            for i in range(num_analogsignal):
                for j in range(num_spiketrain_by_channel):
                    sptr = self.read_spiketrain(lazy = lazy , cascade = cascade ,
                                                    segment_duration = segment_duration, t_st
                    seg.spiketrains += [ sptr ]


            # create an EventArray that mimic triggers.
            # note that ExampleIO  do not allow to acess directly to EventArray
            # for that you need read_segment(cascade = True)
            eva = EventArray()
            if lazy:
                # in lazy case no data are readed
                # eva is empty
                pass
            else:
                # otherwise it really contain data
                n = 1000

                # neo.io support quantities my vector use second for unit
                eva.times = timevect[(rand(n)*timevect.size).astype('i')]* pq.s
                # all duration are the same
                eva.durations = np.ones(n)*500*pq.ms
                # label
                l = [ ]
                for i in range(n):
                    if rand()>.6: l.append( 'TriggerA' )
                    else : l.append( 'TriggerB' )
                eva.labels = np.array( l )

            seg.eventarrays += [ eva ]

        create_many_to_one_relationship(seg)
        return seg


    def read_analogsignal(self ,
                            # the 2 first key arguments are imposed by neo.io API
                            lazy = False,
                            cascade = True,
                            channel_index = 0,
                            segment_duration = 15.,
                            t_start = -1,
                            ):
        """
        With this IO AnalogSignal can e acces directly with its channel number

        """
        sr = 10000.
        sinus_freq = 3. # Hz
        #time vector for generated signal:
        tvect = np.arange(t_start, t_start+ segment_duration , 1./sr)


        if lazy:
            anasig = AnalogSignal([ ], units = 'V', sampling_rate=sr*pq.Hz, t_start=t_start*pq.s)
            # we add the attribute lazy_shape with the size if loaded
            anasig.lazy_shape = tvect.shape
        else:
            # create analogsignal (sinus of 3 Hz)
            sig = np.sin(2*pi*tvect*sinus_freq + channel_index/5.*2*pi)+rand(tvect.size)
            anasig = AnalogSignal(sig, units= 'V' ,  sampling_rate = sr * pq.Hz , t_start = t_start*p

        # for attributes out of neo you can annotate
        anasig.annotate(channel_index = channel_index)
        anasig.annotate(info = 'it is a sinus of %f Hz' %sinus_freq )

        return anasig
```

```
def read_spiketrain(self ,
                                                # the 2 first key arguments are imposed by neo.io API
                                                lazy = False,
                                                cascade = True,

                                                    segment_duration = 15.,
                                                    t_start = -1,
                                                    channel_index = 0,
                                                    ):
    """
    With this IO SpikeTrain can e acces directly with its channel number
    """
    # There are 2 possibles behaviour for a SpikeTrain
    # holding many Spike instance or directly holding spike times
    # we choose here the first :

    num_spike_by_spiketrain = 40
    sr = 10000.

    if lazy:
        times = [ ]
    else:
        times = rand(num_spike_by_spiketrain)*segment_duration+t_start

    # create a spiketrain
    spiketr = SpikeTrain(times, t_start = t_start*pq.s, t_stop = (t_start+segment_duration)*pq.s
                                                units = pq.s,
                                                name = 'it is a spiketrain from exampleio',
                                                )

    if lazy:
        # we add the attribute lazy_shape with the size if loaded
        spiketr.lazy_shape = (num_spike_by_spiketrain,)

    # ours spiketrains also hold the waveforms:

    # 1 generate a fake spike shape (2d array if trodness >1)
    w1 = -stats.nct.pdf(np.arange(11,60,4), 5,20)[::-1]/3.
    w2 = stats.nct.pdf(np.arange(11,60,2), 5,20)
    w = np.r_[ w1 , w2 ]
    w = -w/max(w)

    if not lazy:
        # in the neo API the waveforms attr is 3 D in case tetrode
        # in our case it is mono electrode so dim 1 is size 1
        waveforms  = np.tile( w[newaxis,newaxis,:], ( num_spike_by_spiketrain ,1, 1) )
        waveforms *=  randn(*waveforms.shape)/6+1
        spiketr.waveforms = waveforms*pq.mV
        spiketr.sampling_rate = sr * pq.Hz
        spiketr.left_sweep = 1.5* pq.s

    # for attributes out of neo you can annotate
    spiketr.annotate(channel_index = channel_index)

    return spiketr
```

CHAPTER

TEN

AUTHORS AND CONTRIBUTORS

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

- Samuel Garcia [1]
- Andrew Davison [2]
- Chris Rodgers [3]
- Pierre Yger [2]
- Yann Mahnoun [4]
- Luc Estabanez [2]
- Andrey Sobolev [5]
- Thierry Brizzi [2]
- Florent Jaillet [6]
- Philipp Rautenberg [5]
- Thomas Wachtler [5]
- Cyril Dejean [7]
- Robert Pröpper [8]

1. Centre de Recherche en Neuroscience de Lyon, CNRS UMR5292 - INSERM U1028 - Universite Claude Bernard Lyon 1
2. Unité de Neuroscience, Information et Complexité, CNRS UPR 3293, Gif-sur-Yvette, France
3. University of California, Berkeley
4. Laboratoire de Neurosciences Intégratives et Adaptatives, CNRS UMR 6149 - Université de Provence, Marseille, France
5. G-Node, Ludwig-Maximilians-Universität, Munich, Germany
6. Institut de Neurosciences de la Timone, CNRS UMR 7289 - Université d'Aix-Marseille, Marseille, France
7. Centre de Neurosciences Integratives et Cignitives, UMR 5228 - CNRS - Université Bordeaux I - Université Bordeaux II
8. Neural Information Processing Group, TU Berlin, Germany

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

CHAPTER

ELEVEN

LICENSE

Neo is distributed under a BSD licence.

CHAPTER

TWELVE

CONTRIBUTING

The people behind the project (see *Authors and contributors*) are very open to discussion. Any feedback is gladly received and highly appreciated! Discussion of Neo takes place on the NeuralEnsemble mailing list.

Source code is on GitHub.

The bug tracker is at:

`http://neuralensemble.org/trac/neo`

PYTHON MODULE INDEX