Utilizing e-phys data and metadata in the Neo framework

Michael Denker
Inst. of Neurosci. and Medicine (INM-6)
Jülich Research Center, Germany

Neurodata without Borders Hackathon
Janelia Farms, Ashburn, VA
20.-22. Nov 2014
Using Neo to work with e-phys data
Neo: data representations for electrophysiology

- **Neo**: data representations to support *common interface* to consumers of electrophysiological data
- Data representations in Python for the types of data commonly found in an experiment freed from specific semantics (ex: *RecordingChannelGroup*)

Garcia et al. (2014) Front Neuroinform.
• Relationships between data items: i.e., Block → Segment → AnalogSignal
• **Hierarchical** bidirectional relationships (parent ↔ child)
• Can contain references to data objects across container hierarchy
• Metadata as annotations for every Neo core object
• Designed from perspective of *data providers* and *data consumers*
Goals of Neo

• provide *a common set of base classes for neurophysiology data, to improve interoperability* between Python tools for data analysis/visualization/storage/simulation (OpenElectrophy, NeuroTools, elephant, G-Node, Helmholtz, DABBSIP, PyNN...)

• provide tools *to facilitate access* to these base classes from a programmer’s perspective (e.g., filtering the object tree, time slicing objects,...)

• prioritize *simplicity* (e.g. no data analysis methods included), *performance* (based on numpy) and *correctness* (explicit units/dimensionality checking)

• be able to read from/write to *many data formats*
Connecting software resources via Neo

*Neo*: evolving community standard for common to support the interoperability of software tools

Corner stone: tested and efficient *file I/O modules*

Current INM-6 activity: development of a flexible file I/O for one (soon several) of the most desired file formats
Example usage: Loading

To load a file format which is implemented in a MyFormatIO class

```python
>>> from neo.io import MyFormatIO
>>> file = MyFormatIO("myfile.dat")
```

To know what types of objects are supported by this io interface:

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

Not all supported objects can be read directly. For instance, many formats supports AnalogSignal but you can’t access them directly: you must read a Segment and access your AnalogSignal like that:

```python
>>> seg = file.read_segment()
>>> seg.get_analogsignals()
```

To have the list of directly readable objects:

```python
>>> file.readable_objects
[Segment]
```

To read the entire file:

```python
>>> result = file.read()
>>> type(result)
neo.core.Segment
```
class BlackrockIO(BaseIO):
    is_readable = True
    is_writable = False
    supported_objects = [neo.Block, neo.Segment,
                         neo.AnalogSignal, neo.SpikeTrain, neo.EventArray,
                         neo.RecordingChannelGroup, neo.RecordingChannel]
    readable_objects = [neo.Block]
    writeable_objects = []
    has_header = False
    is_streamable = False
    read_params = {}
    write_params = {}
    name = 'Blackrock'
    description = 'This IO reads .nev/.nsX file of the Blackrock (Cerebus) recordings system.'
    extensions = ['ns' + str(_) for _ in range(1, 7)]
    extensions.append('nev')
    mode = 'file'
Example: BlackrockIO readers

def read_segment(self, n_start=None, n_stop=None, load_waveforms=False, nsx_num=None, lazy=False, cascade=True):

    """Reads one Segment.

    The Segment will contain one AnalogSignal for each channel and will go from n_start to n_stop (in samples).

    Arguments:
    n_start : time in samples that the Segment begins
    n_stop : time in samples that the Segment ends

    Python indexing is used, so n_stop is not inclusive.

    Returns a Segment object containing the data.
    """
Example usage: Working with Neo objects

```python
>>> from neo import AnalogSignal, Segment
>>> import numpy
>>> from quantities import uV, Hz

>>> data = numpy.random.uniform(size=100)*uV
>>> signal = AnalogSignal(data, sampling_rate=420*Hz)
>>> isinstance(signal, numpy.ndarray)
True

# Time keeping for analog signals:
>>> signal.t_start
array(0.0) * s
>>> signal.t_stop
array(0.1) * s
>>> signal[20:80].t_stop
array(0.08) * s

# Adding annotations:
>>> signal.annotate(cell_id="20100405a")
```
Example usage: Working with Neo objects

```
# Filtering
>>> signal2 = AnalogSignal(data+1*uV, sampling_rate=420*Hz)
>>> signal2.annotate(cell_id="20100405b")
>>> seg = Segment()
>>> seg.analogsignals.append(signal)
>>> seg.analogsignals.append(signal2)

>>> seg.filter(cell_id="20100405a")
[AnalogSignal in 1.0 uV with 100 float64 values
 annotations: {'cell_id': '20100405a'}
 channel index: None
 sampling rate: 420.0 Hz
 time: 0.0 s to 0.238095238095 s]

# Working with the object:
>>> print
   numpy.mean(seg.filter(cell_id="20100405a")[0].magnitude
) 0.545829319931
```
odML: practical experience with metadata management
Complex meta data in a behavioral experiment

Reach to grasp exp.:
- 120 trials / subsession
- ~ 5 subsessions / day
- ~ 70 days / monkey
- 3 monkeys

Neural data in:
- .2 files
- .2 formats

Zehl, Denker, Stoewer, Jaillet, Brochier, Riehle, Wachtler, Grün (in prep.)
Reach to grasp exp.:
- 120 trials / subsession
- ~5 subsessions / day
- ~70 days / monkey
- 3 monkeys

Neural data in:
- 2 files
- 2 formats

Zehl, Denker, Stoewer, Jaillet, Brochier, Riehle, Wachtler, Grün (in prep.)
odML: meta data for electrophysiological experiments

Example: Stimulus
Stimulus duration = 500.0 ms ± 0.0 ms

odML value types:
- integer
- float
- string
- URL
- text
- n-tuple
- date
- time
- boolean
- person

Value:
- data value = 500.0
- uncertainty = 0.0
- unit = ms
- type = float
- definition = duration in ms

Property:
- name = duration
- definition = duration of stimulus

Subsection:
(can contain properties and/or sub-subsections)
- name = types
- type = stimulus
- definition = stimulus types

Section:
(can contain properties and/or subsections)
- name = stimulus
- type = stimulus
- definition = contains all stimulus parameters

Root Section:
(groups all sections)
- author = Lyuba Zehl
- date = 2012-11-16
- version = 01

Grewe et al. (2011) Front Neuroinform.
odML: compiled via scripts and the editor

---

**Code 1: How to generate an odML file in Python**

```python
import odml
import datetime

# Generate an odml-document
odml_document = odml.Document(author = "Lyuba Zehl",
                              date = datetime.date(2014, 6, 1))

# Generate an odml-section
odml_section = odml.Section(name = "ElectrodeArray",
                            definition = "Information on electrodes")

# Generate an odml-subsection
odml_subsection = odml.Section(name = "Electrodes",
                                definition = "Information on electrodes")

# Generate odml-properties with their corresponding odml-value
odml_value_a = odml.Value(data = "file://home/zehl/IDScheme/100.png",
                           dtype = odml.DType.url,
                           definition = "URL to file.")

odml_value_b1 = odml.Value(data = 4.2,
                           dtype = odml.DType.float,
                           unit = "mm",
                           definition = "Array width.")

odml_value_b2 = odml.Value(data = 4.2,
                           dtype = odml.DType.float,
                           unit = "mm",
                           definition = "Array length.")

# Append all odml-objects to odml-tree
odml_section.append(odml_subsection)

# Save the odml-document with all its content
write_to = "/home/zehl/AREADNE_2014/example.odml"
odml.tools.xmlparser.XMLWriter(odml_document).write_file(write_to)
```

---

**Improve python libraries for generation and access of meta data information**

**Tutorial on building successful meta data collections using odML**

**Accessible templates for common odML sections (e.g., common hardware) to be provided**
Usage example

neo.io.blackrockio

odML

data
annotations

experiment_io:

must perform matching
Relationship to the Human Brain Project
Build, Simulate and Validate

- Workbench through web portal
- Integrated workflow
- Access to platforms
- Provenance tracking
Mitglied in der Helmholtz-Gemeinschaft

**Tools for the analysis of functional data**
(work package 5.3)

**Standalone open source Python package**

**ElePhAnT**

- Surrogate generation
- Spike sorting
- Spectral analysis
- Signal processing
- Spike pattern analysis
- Spike train correlation
- Spike train statistics
- Spike-triggered averaging
- Data set filtering and cutting
- Data representations
- Testing

**HBP Unified Portal**

**Functional Data Analysis Toolkit (FDAT)**

- Individual specific analysis tasks
Combining multiple tools in FDAT components

FDAT task
- Load data
- Analyse data (e.g., histogram of CVs across all spike train objects in input)
- Create output files (hdf5) and result figures (png)

NIX under consideration

URI to input file

Simple analysis parameters

Uses

Links to output files
Summary

Neo is a good target platform for implementation of a format because:

- Many file format readers already implemented, leading to easy portability
- Open source, extensible
- Provides well-structured representation of ephys data
- Provides a simple metadata annotation mechanism
- Is the basis for a variety of libraries and programs

It is not (yet) because:

- Neo is python-only
- Integration of structured hierarchical metadata and its link to primary data is not yet fully implemented/defined (e.g., information about trials) → NIX

odML-based hierarchical metadata collections

- work for large collections of metadata…
- …but lack an automatic linking mechanism between data and metadata