PyNWB

Release 1.1.2.post.dev14

Jan 07, 2020

Getting Started

PyNWB is a Python package for working with NWB files. It provides a high-level API for efficiently working with Neurodata stored in the NWB format.

Neurodata Without Borders: Neurophysiology (NWB:N) is a project to develop a unified data format for cellular-based neurophysiology data, focused on the dynamics of groups of neurons measured under a large range of experimental conditions.

The NWB:N team consists of neuroscientists and software developers who recognize that adoption of a unified data format is an important step toward breaking down the barriers to data sharing in neuroscience.

CHAPTER 1

Dependencies

PyNWB has the following minimum requirements, which must be installed before you can get started using PyNWB.

1. Python 3.5, 3.6, or 3.7
2. pip

CHAPTER 2

Installation

2.1 Install release from PyPI

The Python Package Index (PyPI) is a repository of software for the Python programming language.

To install or update PyNWB distribution from PyPI simply run:

```
$ pip install -U pynwb
```

This will automatically install the following required dependencies:

1. hdmf
2. h5py
3. numpy
4. python-dateutil
5. requests
6. ruamel.yaml
7. six
8. chardet

2.2 Install release from Conda-forge

Conda-forge is a community led collection of recipes, build infrastructure and distributions for the conda package manager.

To install or update PyNWB distribution from conda-forge using conda simply run:

```
$ conda install -c conda-forge pynwb
```

2.3 Install latest pre-release

This is useful to try out the latest features and also set up continuous integration of your own project against the latest version of PyNWB.

```
$ pip install -U pynwb --find-links https://github.com/NeurodataWithoutBorders/pynwb/
↪releases/tag/latest  --no-index
```

CHAPTER 3

For developers

3.1 Install from Git repository

For development an editable install is recommended.

```
$ pip install -U virtualenv
$ virtualenv ~/pynwb
$ source ~/pynwb/bin/activate
$ git clone --recurse-submodules git@github.com:NeurodataWithoutBorders/pynwb.git
$ cd pynwb
$ pip install -r requirements.txt
$ pip install -e .
```

3.2 Run tests

For running the tests, it is required to install the development requirements.

```
$ pip install -U virtualenv
$ virtualenv ~/pynwb
$ source ~/pynwb/bin/activate
$ git clone --recurse-submodules git@github.com:NeurodataWithoutBorders/pynwb.git
$ cd pynwb
$ pip install -r requirements.txt -r requirements-dev.txt
$ pip install -e .
$ tox
```

3.3 Following PyNWB Style Guide

Before you create a Pull Request, make sure you are following PyNWB style guide (PEP8). To do that simply run the following command in the project's root directory.

```
$ flake8
```

CHAPTER 4

How to contribute to NWB:N software and documents

4.1 Code of Conduct

This project and everyone participating in it is governed by our code of conduct guidelines. By participating, you are expected to uphold this code. Please report unacceptable behavior.

4.2 Types of Contributions

4.2.1 Did you find a bug? or Do you intend to add a new feature or change an existing one?

- **Identify the appropriate repository** for the change you are suggesting:
  - Use nwb-schema for any changes to the NWB:N format schema, schema language, storage, and other NWB:N related documents
  - Use PyNWB for any changes regarding the PyNWB API and the corresponding documentation
  - Use MatNWB for any changes regarding the PyNWB API and the corresponding documentation
- **Ensure the feature or change was not already reported** by searching on GitHub under PyNWB Issues and nwb-schema issues, respectively .
- If you are unable to find an open issue addressing the problem then open a new issue on the respective repository. Be sure to include:
  - **brief and descriptive title**
  - **clear description of the problem you are trying to solve***. Describing the use case is often more important than proposing a specific solution. By describing the use case and problem you are trying to solve gives the development team and ultimately the NWB:N community a better understanding for the reasons of changes and enables others to suggest solutions.

  - **context** providing as much relevant information as possible and if available a **code sample** or an **executable test case** demonstrating the expected behavior and/or problem.

- Be sure to select the appropriate labels (see *Issue Labels, Projects, and Milestones*) for your tickets so that they can be processed accordingly.
- NWB:N is currently being developed primarily by staff at scientific research institutions and industry, most of which work on many different research projects. Please be patient, if our development team is not able to respond immediately to your issues. In particular issues that belong to later project milestones may not be reviewed or processed until work on that milestone begins.

4.2.2 Did you write a patch that fixes a bug or implements a new feature?

See the `Contributing Patches and Changes` section below for details.

4.2.3 Do you have questions about NWB:N?

Ask questions on our Slack workspace or sign up for our NWB:N mailing list for updates.

4.2.4 Informal discussions between developers and users?

The https://nwb-users.slack.com slack is currently used mainly for informal discussions between developers and users.

4.3 Contributing Patches and Changes

The `dev` branches of PyNWB and nwb-schema, are protected; you cannot push to them directly. You must upload your changes by pushing a new branch, then submit your changes to the `dev` branch via a Pull Request. This allows us to conduct automated testing of your contribution, and gives us a space for developers to discuss the contribution and request changes. If you decide to tackle an issue, please make yourself an assignee on the issue to communicate this to the team. Don't worry - this does not commit you to solving this issue. It just lets others know who they should talk to about it.

From your local copy directory, use the following commands.

If you have not already, you will need to clone the repo:

```
$ git clone --recurse-submodules https://github.com/NeurodataWithoutBorders/pynwb.git
```

1) First create a new branch to work on

```
$ git checkout -b <new_branch>
```

2) Make your changes.

3) We will automatically run tests to ensure that your contributions didn't break anything and that they follow our style guide. You can speed up the testing cycle by running these tests locally on your own computer using `tox` and `flake8`.

4) Push your feature branch to origin (i.e. github)

```
$ git push origin <new_branch>
```

5) Once you have tested and finalized your changes, create a pull request (PR) targeting `dev` as the base branch:

- Ensure the PR description clearly describes the problem and solution.
- Include the relevant issue number if applicable. TIP: Writing e.g. "fix #613" will automatically close issue #613 when this PR is merged.
- Before submitting, please ensure that the code follows the standard coding style of the respective repository.
- If you would like help with your contribution, or would like to communicate contributions that are not ready to merge, submit a PR where the title begins with "[WIP]."
- **NOTE:** Contributed branches will be removed by the development team after the merge is complete and should, hence, not be used after the pull request is complete.

4.4 Issue Labels, Projects, and Milestones

4.4.1 Labels

Labels are used to describe the general scope of an issue, e.g., whether it describes a bug or feature request etc. Please review and select the appropriate labels for the respective Git repository:

- PyNWB issue labels
- nwb-schema issue labels

4.4.2 Milestones

Milestones are used to define the scope and general timeline for issues. Please review and select the appropriate milestones for the respective Git repository:

- PyNWB milestones
- nwb-schema milestones

4.4.3 Projects

Projects are currently used mainly on the NeurodataWithoutBorders organization level and are only accessible to members of the organization. Projects are used to plan and organize developments across repositories. We currently do not use projects on the individual repository level, although that might change in the future.

4.5 Styleguides

4.5.1 Git Commit Message Styleguide

- Use the present tense ("Add feature" not "Added feature")
- The first line should be short and descriptive.
- Additional details may be included in further paragraphs.
- If a commit fixes an issues, then include "Fix #X" where X is the number of the issue.
- Reference relevant issues and pull requests liberally after the first line.

4.5.2 Documentation Styleguide

All documentations is written in reStructuredText (RST) using Sphinx.

4.5.3 Did you fix whitespace, format code, or make a purely cosmetic patch in source code?

Source code changes that are purely cosmetic in nature and do not add anything substantial to the stability, functionality, or testability will generally not be accepted unless they have been approved beforehand. One of the main reasons is that there are a lot of hidden costs in addition to writing the code itself, and with the limited resources of the project, we need to optimize developer time. E.g,. someone needs to test and review PRs, backporting of bug fixes gets harder, it creates noise and pollutes the git repo and many other cost factors.

4.5.4 Format Specification Styleguide

Coming soon

4.5.5 Python Code Styleguide

Python coding style is checked via `flake8` for automatic checking of PEP8 style during pull requests.

4.6 Endorsement

Please don't take the fact that working with an organization (e.g., during a hackathon or via GitHub) as an endorsement of your work or your organization. It's okay to say e.g., "We worked with XXXXX to advance science" but not e.g., "XXXXX supports our work on NWB"."

CHAPTER 5

License and Copyright

See the license files for details about the copyright and license.

As indicated in the PyNWB license: *"You are under no obligation whatsoever to provide any bug fixes, patches, or upgrades to the features, functionality or performance of the source code ("Enhancements") to anyone; however, if you choose to make your Enhancements available either publicly, or directly to Lawrence Berkeley National Laboratory, without imposing a separate written license agreement for such Enhancements, then you hereby grant the following license: a non-exclusive, royalty-free perpetual license to install, use, modify, prepare derivative works, incorporate into other computer software, distribute, and sublicense such enhancements or derivative works thereof, in binary and source code form."*

Contributors to the NWB code base are expected to use a permissive, non-copyleft open source license. Typically 3-clause BSD is used, but any compatible license is allowed, the MIT and Apache 2.0 licenses being good alternative choices. The GPL and other copyleft licenses are not allowed due to the consternation it generates across many organizations.

Also, make sure that you are permitted to contribute code. Some organizations, even academic organizations, have agreements in place that discuss IP ownership in detail (i.e., address IP rights and ownership that you create while under the employ of the organization). These are typically signed documents that you looked at on your first day of work and then promptly forgot. We don't want contributed code to be yanked later due to IP issues.

CHAPTER 6

Introduction

PyNWB provides a high-level Python API for reading and writing NWB:N formatted files. This section provides a broad overview of the software architecture of PyNWB (see Section *Software Architecture* and the functionality provided for reading and writing neurophysiology data from/to NWB files (see Section *NWB:N File Format*)

CHAPTER 7

Software Architecture

The main goal of PyNWB is to enable users and developers to efficiently interact with the NWB data format, format files, and specifications. The following figures provide an overview of the high-level architecture of PyNWB and functionality of the various components.

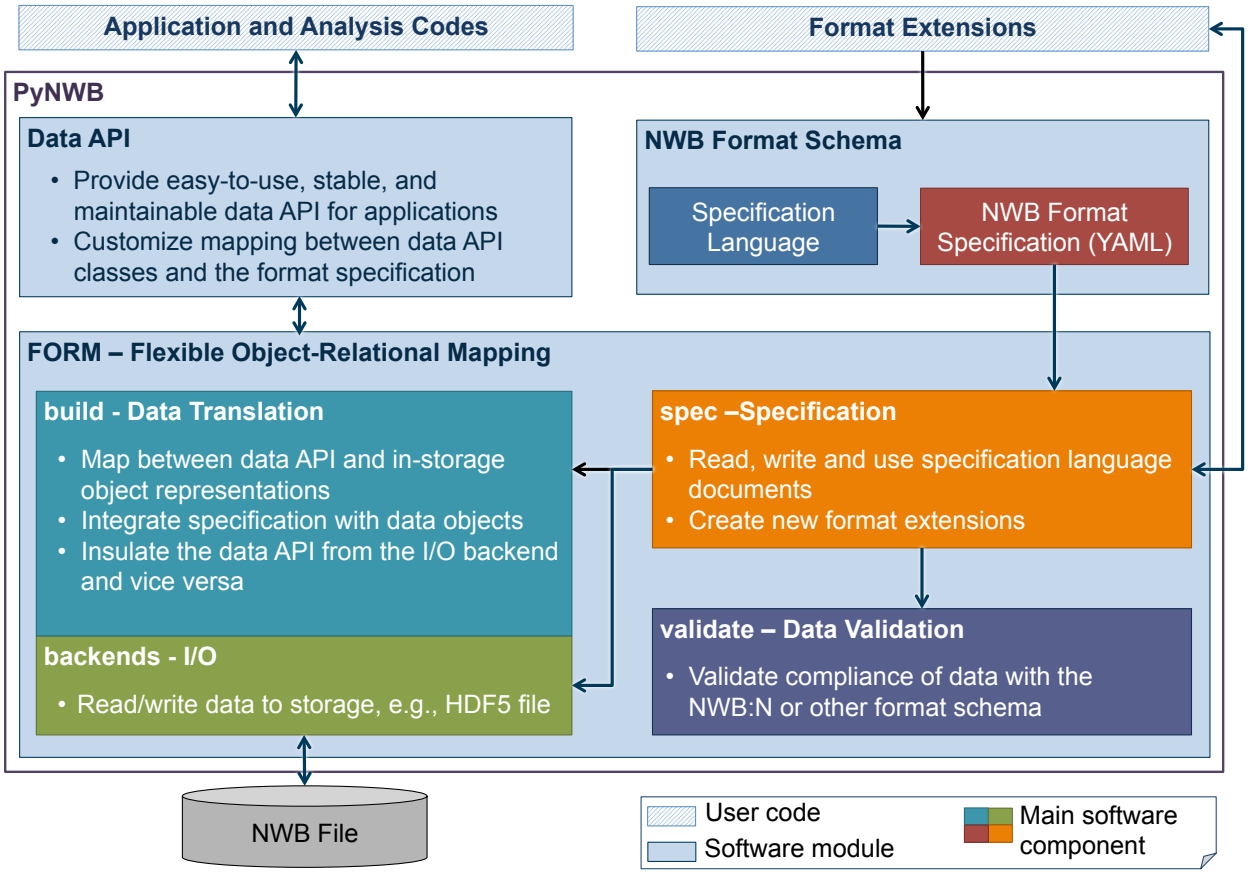

Fig. 1: Overview of the high-level software architecture of PyNWB (click to enlarge).

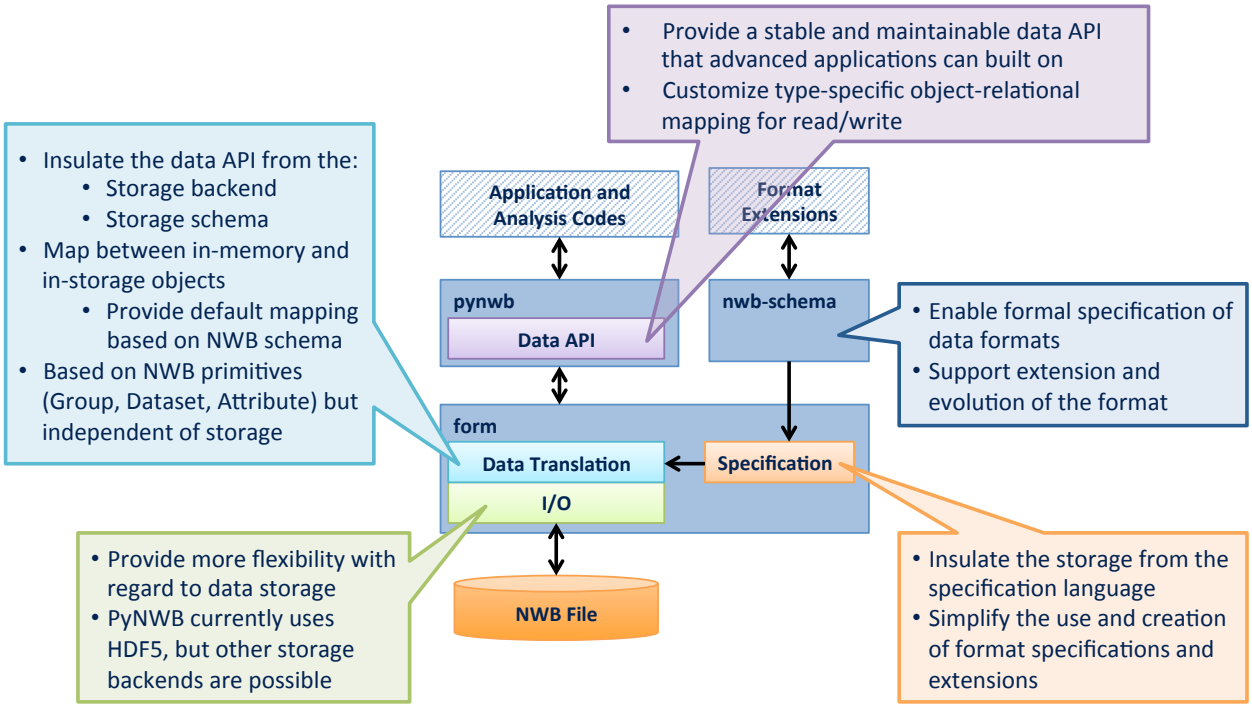

Fig. 2: We choose a modular design for PyNWB to enable flexibility and separate the various aspects of the NWB:N ecosystem (click to enlarge).

7.1 Main Concepts

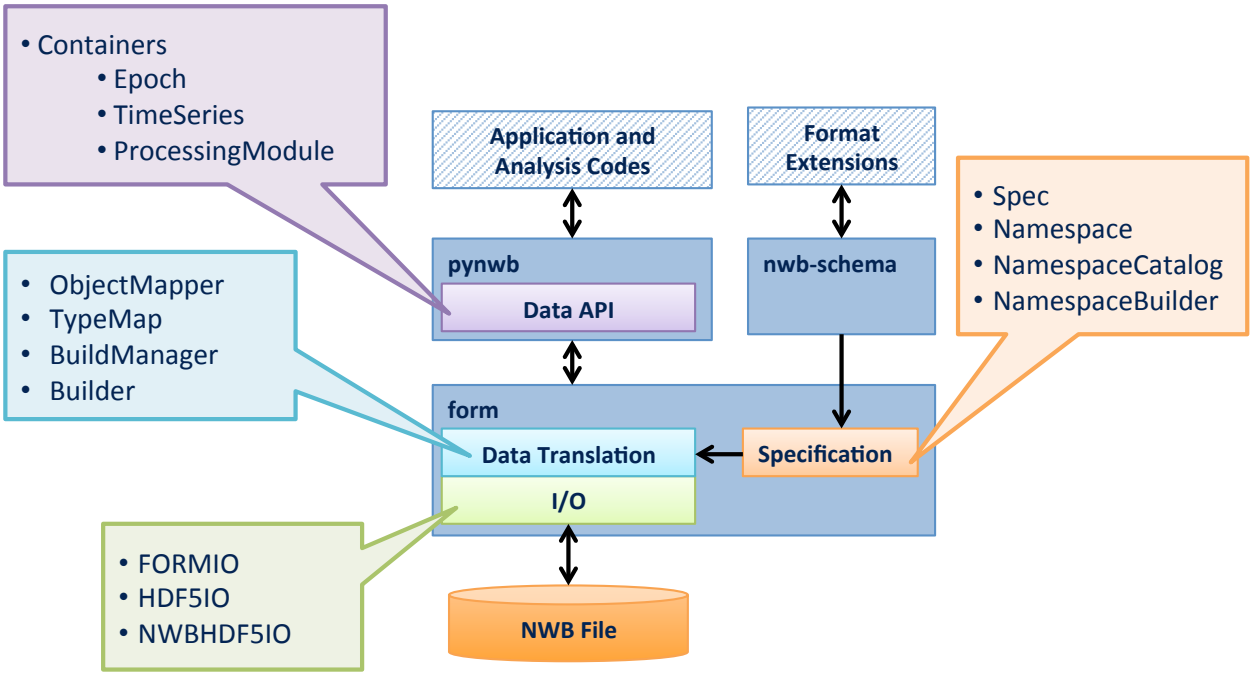

Fig. 3: Overview of the main concepts/classes in PyNWB and their location in the overall software architecture (click to enlarge).

7.1.1 Container

- In memory objects
- Interface for (most) applications
- Like a table row
- PyNWB has many of these – one for each neurodata_type in the NWB schema. PyNWB organizes the containers into a set of modules based on their primary application (e.g., ophys for optophysiology):
  - *pynwb.base*, *pynwb.file*
  - *pynwb.ecephys*
  - *pynwb.ophys*
  - *pynwb.icephys*
  - *pynwb.ogen*
  - *pynwb.behavior*

7.1.2 Builder

- Intermediary objects for I/O
- Interface for I/O

- Backend readers and writers must return and accept these
- There are different kinds of builders for different base types:
  - `GroupBuilder` - represents a collection of objects
  - `DatasetBuilder` - represents data
  - `LinkBuilder` - represents soft-links
  - `RegionBuilder` - represents a slice into data (Subclass of `DatasetBuilder`)
- **Main Module:** `hdmf.build.builders`

7.1.3 Spec

- Interact with format specifications
- Data structures to specify data types and what said types consist of
- Python representation for YAML specifications
- Interface for writing extensions or custom specification
- There are several main specification classes:
  - *`NWBAttributeSpec`* - specification for metadata
  - *`NWBGroupSpec`* - specification for a collection of objects (i.e. subgroups, datasets, link)
  - *`NWBDatasetSpec`* - specification for dataset (like and n-dimensional array). Specifies data type, dimensions, etc.
  - *`NWBLinkSpec`* - specification for link (like a POSIX soft link)
  - `RefSpec` - specification for references (References are like links, but stored as data)
  - *`NWBDtypeSpec`* - specification for compound data types. Used to build complex data type specification, e.g., to define tables (used only in `DatasetSpec` and correspondingly *`NWBDatasetSpec`*)
- **Main Modules:**
  - `hdmf.spec` – General specification classes.
  - *`pynwb.spec`* – NWB specification classes. (Most of these are specializations of the classes from `hdmf.spec`)

Note: A `data_type` (or more specifically a `neurodata_type` in the context of NWB) defines a reusable type in a format specification that can be referenced and used elsewhere in other specifications. The specification of the NWB format is basically a collection of `neurodata_types`, e.g.: `NWBFile` defines a GroupSpec for the top-level group of an NWB format file which includes `TimeSeries`, `ElectrodeGroup`, `ImagingPlane` and many other `neurodata_types` . When creating a specification, two main keys are used to include and define new `neurodata_types`

- `neurodata_type_inc` is used to include an existing type and
- `neurodata_type_def` is used to define a new type

I.e, if both keys are defined then we create a new type that uses/inherits an existing type as a base.

7.1.4 ObjectMapper

- Maintains the mapping between *Container* attributes and *Spec* components
- Provides a way of converting between *Container* and *Builder*
- ObjectMappers are constructed using a *Spec*
- Ideally, one ObjectMapper for each data type
- Things an ObjectMapper should do:
  - Given a *Builder*, return a Container representation
  - Given a *Container*, return a Builder representation
- PyNWB has many of these – one for each type in NWB schema
- **Main Module:** `hdmf.build.map`
  - NWB-specific ObjectMappers are locate in submodules of *`pynwb.io`*

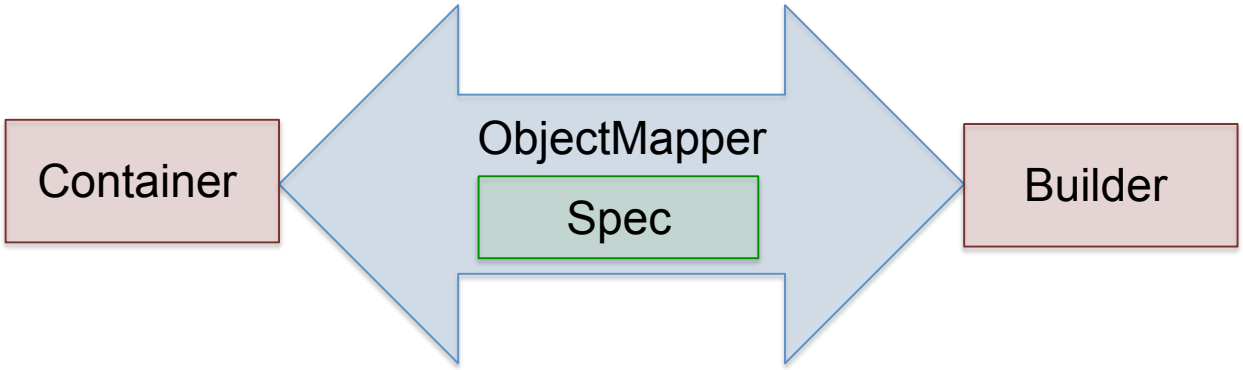

Fig. 4: Relationship between *Container*, *Builder*, *ObjectMapper*, and *Spec*

7.2 Additional Concepts

7.2.1 Namespace, NamespaceCatalog, NamespaceBuilder

- **Namespace**
  - A namespace for specifications
  - Necessary for making extensions
  - Contains basic info about who created extensions
  - Core NWB:N schema has namespace "core"
  - Get from *`pynwb.spec.NWBNamespace`*
    - extension of generic Namespace class that will include core
- `NamespaceCatalog` – A class for managing namespaces
- `NamespaceBuilder` – A utility for building extensions

7.2.2 TypeMap

- Map between data types, Container classes (i.e. a Python class object) and corresponding ObjectMapper classes
- Constructed from a NamespaceCatalog
- Things a TypeMap does:
  - Given an NWB data type, return the associated Container class
  - Given a Container class, return the associated ObjectMapper
- PyNWB has two of these classes:
  - the base class (i.e. `TypeMap`) - handles NWB 2.x
  - *`pynwb.legacy.map.TypeMapLegacy`* - handles NWB 1.x
- PyNWB provides a "global" instance of TypeMap created at runtime
- TypeMaps can be merged, which is useful when combining extensions

7.2.3 BuildManager

- Responsible for memoizing *Builder* and *Container*
- Constructed from a *TypeMap*
- PyNWB only has one of these: `hdmf.build.map.BuildManager`

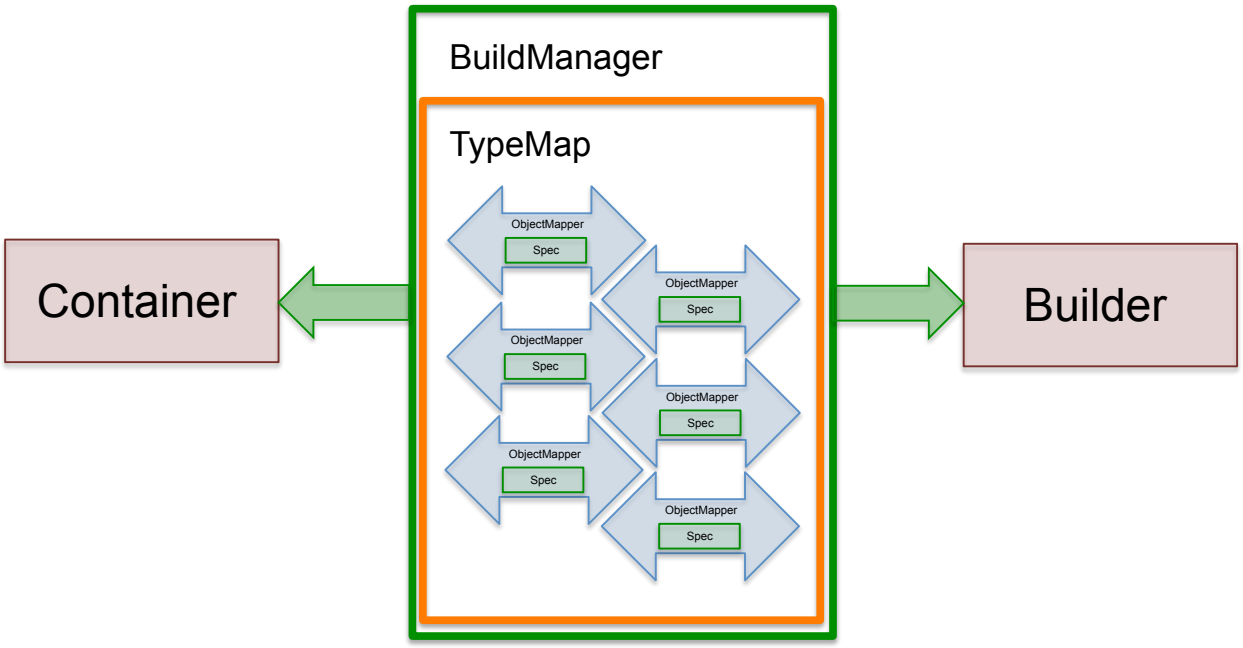

Fig. 5: Overview of *BuildManager* (and *TypeMap*) (click to enlarge).

7.2.4 HDMFIO

- Abstract base class for I/O

- `HDMFIO` has two key abstract methods:
  - `write_builder` – given a builder, write data to storage format
  - `read_builder` – given a handle to storage format, return builder representation
  - Others: `open` and `close`
- Constructed with a *BuildManager*
- Extend this for creating a new I/O backend
- PyNWB has one extension of this:
  - `HDF5IO` - reading and writing HDF5
  - *`NWBHDF5IO`* - wrapper that pulls in core NWB specification

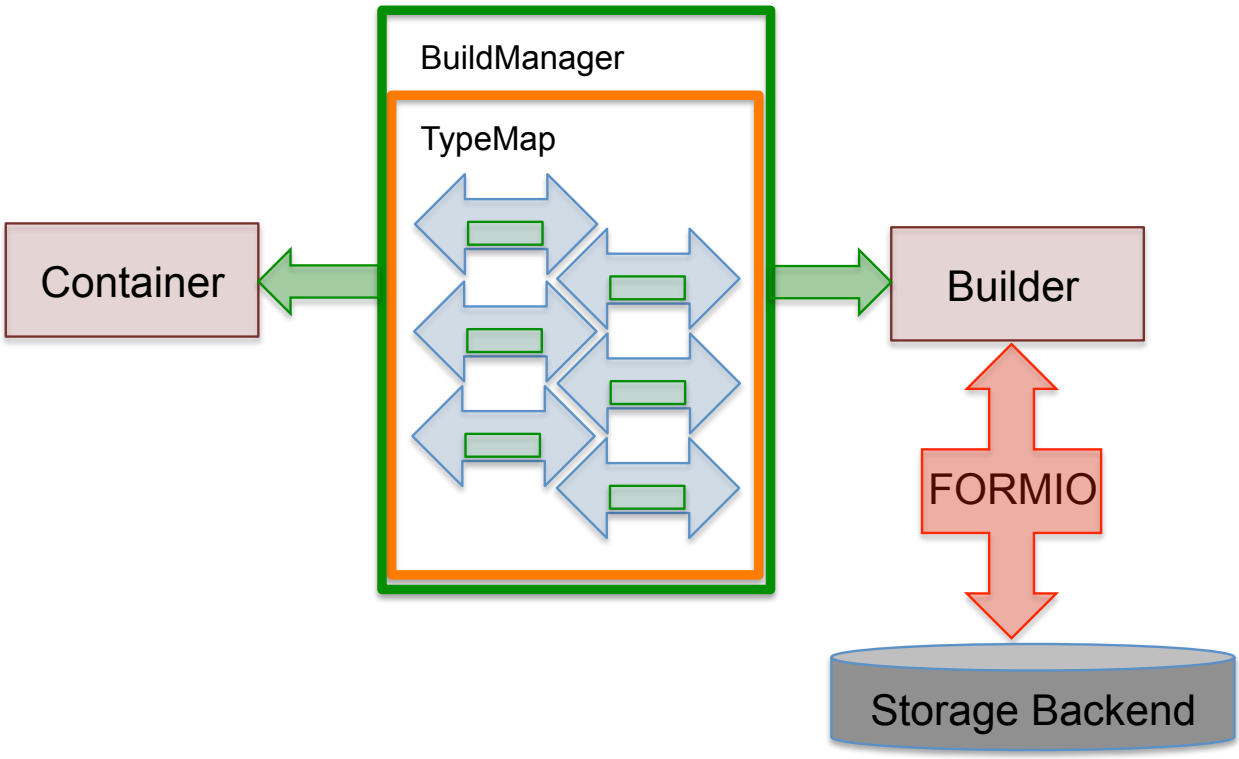

Fig. 6: Overview of *HDMFIO* (click to enlarge).

CHAPTER 8

NWB:N File Format

The NWB Format is built around two concepts: *TimeSeries* and *ProcessingModules*.

TimeSeries are objects for storing time series data, and *Processing Modules* are objects for storing and grouping analyses. The following sections describe these classes in further detail.

8.1 NWBFile

NWB files are represented in PyNWB with *NWBFile* objects. `NWBFile` objects provide functionality for creating *TimeSeries* datasets and *Processing Modules*, as well as functionality for storing experimental metadata and other metadata related to data provenance.

8.2 TimeSeries

TimeSeries objects store time series data. These Python objects correspond to TimeSeries specifications provided by the NWB format specification. Like the NWB specification, TimeSeries Python objects follow an object-oriented inheritance pattern. For example, the class `TimeSeries` serves as the base class for all other TimeSeries types.

The following TimeSeries objects are provided by the API and NWB specification:

- `ElectricalSeries`
  - `SpikeEventSeries`
- `AnnotationSeries`
- `AbstractFeatureSeries`
- `ImageSeries`
  - `ImageMaskSeries`
  - `OpticalSeries`
  - `TwoPhotonSeries`

- *IndexSeries*
- *IntervalSeries*
- *OptogeneticSeries*
- *PatchClampSeries*
  - *CurrentClampSeries*
    - *IZeroClampSeries*
  - *CurrentClampStimulusSeries*
  - *VoltageClampSeries*
  - *VoltageClampStimulusSeries*
- *RoiResponseSeries*
- *SpatialSeries*

8.3 Processing Modules

Processing modules are objects that group together common analyses done during processing of data. Processing module objects are unique collections of analysis results. To standardize the storage of common analyses, NWB provides the concept of an *NWBDataInterface*, where the output of common analyses are represented as objects that extend the *NWBDataInterface* class. In most cases, you will not need to interact with the *NWBDataInterface* class directly. More commonly, you will be creating instances of classes that extend this class.

The following analysis *NWBDataInterface* objects are provided by the API and NWB specification:

- *BehavioralEpochs*
- *BehavioralEvents*
- *BehavioralTimeSeries*
- *CompassDirection*
- *DfOverF*
- *EventDetection*
- *EventWaveform*
- *EyeTracking*
- *FeatureExtraction*
- *FilteredEphys*
- *Fluorescence*
- *ImageSegmentation*
- *ImagingRetinotopy*
- *LFP*
- *MotionCorrection*
- *Position*

Additionally, the *TimeSeries* described *above* are also subclasses of *NWBDataInterface*, and can therefore be used anywhere *NWBDataInterface* is allowed.

Note: In addition to `NWBContainer` which functions as a common base type for Group objects `NWBData` provides a common base for the specification of datasets in the NWB:N format.

CHAPTER 9

Tutorials

9.1 General tutorials

Note: Click *here* to download the full example code

9.1.1 Exploratory data analysis with NWB

This example will focus on the basics of working with an *NWBFile* to do more than storing standardized data for use and exchange. For example, you may want to store results from intermediate analyses or one-off analyses with unknown utility. This functionality is primarily accomplished with linking and scratch space.

Note: The scratch space is explicitly for non-standardized data that is not intended for reuse by others. Standard NWB:N types, and extension if required, should always be used for any data that you intend to share. As such, published data should not include scratch data and a user should be able to ignore any data stored in scratch to use a file.

Raw data

To demonstrate linking and scratch space, lets assume we are starting with some acquired data.

```
from pynwb import NWBFile, TimeSeries, NWBHDF5IO
from datetime import datetime
from dateutil.tz import tzlocal
import numpy as np

# set up the NWBFile
start_time = datetime(2019, 4, 3, 11, tzinfo=tzlocal())
```

(continues on next page)

(continued from previous page)

```
create_date = datetime(2019, 4, 15, 12, tzinfo=tzlocal())

nwb = NWBFile(session_description='demonstrate NWBFile scratch',  # required
              identifier='NWB456',  # required
              session_start_time=start_time,  # required
              file_create_date=create_date)  # optional

# make some fake data
timestamps = np.linspace(0, 100, 1024)
data = np.sin(0.333 * timestamps) + np.cos(0.1 * timestamps) + np.random.
↪randn(len(timestamps))
test_ts = TimeSeries(name='raw_timeseries', data=data, unit='m',␣
↪timestamps=timestamps)

# add it to the NWBFile
nwb.add_acquisition(test_ts)

with NWBHDF5IO('raw_data.nwb', mode='w') as io:
    io.write(nwb)
```

Copying an NWB file

To copy a file, we must first read the file.

```
raw_io = NWBHDF5IO('raw_data.nwb', 'r')
nwb_in = raw_io.read()
```

And then create a shallow copy the file with the *copy* method of *NWBFile*.

```
nwb_proc = nwb_in.copy()
```

Now that we have a copy, lets process some data, and add the results as a *ProcessingModule* to our copy of the file.[1]

```
import scipy.signal as sps

mod = nwb_proc.create_processing_module('filtering_module', "a module to store␣
↪filtering results")

ts1 = nwb_in.acquisition['raw_timeseries']
filt_data = sps.correlate(ts1.data, np.ones(128), mode='same') / 128
ts2 = TimeSeries(name='filtered_timeseries', data=filt_data, unit='m', timestamps=ts1)

mod.add_container(ts2)
```

Now write the copy, which contains the processed data.[2]

[1]

Note: Notice here that we are reusing the timestamps to the original TimeSeries.

[2]

Note: The `processed_data.nwb` file (i.e., our copy) stores our processing module and contains external links to all data in our original file, i.e., the data from our raw file is being linked to, not copied. This allows us to isolate our processing data in a separate file while still allowing us to

```
with NWBHDF5IO('processed_data.nwb', mode='w', manager=raw_io.manager) as io:
    io.write(nwb_proc)
```

Adding scratch data

You may end up wanting to store results from some one-off analysis, and writing an extension to get your data into an NWBFile is too much over head. This is facilitated by the scratch space in NWB:N.[3]

First, lets read our processed data and then make a copy

```
proc_io = NWBHDF5IO('processed_data.nwb', 'r')
nwb_proc_in = proc_io.read()
```

Now make a copy to put our scratch data into[4]

```
nwb_scratch = nwb_proc_in.copy()
```

Now lets do an analysis for which we do not have a specification, but we would like to store the results for.

```
filt_ts = nwb_scratch.modules['filtering_module']['filtered_timeseries']

fft = np.fft.fft(filt_ts.data)

nwb_scratch.add_scratch(fft, name='dft_filtered', notes='discrete Fourier transform␣
↪from filtered data')
```

Finally, write the results.

```
with NWBHDF5IO('scratch_analysis.nwb', 'w', manager=proc_io.manager) as io:
    io.write(nwb_scratch)
```

To get your results back, you can index into *scratch* or use *get_scratch*:

```
scratch_io = NWBHDF5IO('scratch_analysis.nwb', 'r')
nwb_scratch_in = scratch_io.read()

fft_in = nwb_scratch_in.scratch['dft_filtered']

fft_in = nwb_scratch_in.get_scratch('dft_filtered')
```

Note: Click *here* to download the full example code

access the raw data from our `processed_data.nwb` file, without having to duplicate the data.

3

Note: This scratch space only exists if you add scratch data.

4

Note: We recommend writing scratch data into copies of files only. This will make it easier to isolate and discard scratch data and avoids updating files that store precious data.

9.1.2 Advanced HDF5 I/O

The HDF5 storage backend supports a broad range of advanced dataset I/O options, such as, chunking and compression. Here we demonstrate how to use these features from PyNWB.

Wrapping data arrays with `H5DataIO`

In order to customize the I/O of datasets using the HDF I/O backend we simply need to wrap our datasets using `H5DataIO`. Using H5DataIO allows us to keep the Container classes independent of the I/O backend while still allowing us to customize HDF5-specific I/O features.

Before we get started, lets create an NWBFile for testing so that we can add our data to it.

```
from datetime import datetime
from dateutil.tz import tzlocal
from pynwb import NWBFile

start_time = datetime(2017, 4, 3, 11, tzinfo=tzlocal())
create_date = datetime(2017, 4, 15, 12, tzinfo=tzlocal())

nwbfile = NWBFile(session_description='demonstrate advanced HDF5 I/O features',
                  identifier='NWB123',
                  session_start_time=start_time,
                  file_create_date=create_date)
```

Normally if we create a timeseries we would do

```
from pynwb import TimeSeries
import numpy as np

data = np.arange(100, 200, 10)
timestamps = np.arange(10)
test_ts = TimeSeries(name='test_regular_timeseries',
                     data=data,
                     unit='SIunit',
                     timestamps=timestamps)
nwbfile.add_acquisition(test_ts)
```

Now let's say we want to compress the recorded data values. We now simply need to wrap our data with H5DataIO. Everything else remains the same

```
from hdmf.backends.hdf5.h5_utils import H5DataIO
wrapped_data = H5DataIO(data=data, compression=True)      # <----
test_ts = TimeSeries(name='test_compressed_timeseries',
                     data=wrapped_data,                   # <----
                     unit='SIunit',
                     timestamps=timestamps)
nwbfile.add_acquisition(test_ts)
```

This simple approach gives us access to a broad range of advanced I/O features, such as, chunking and compression. For a complete list of all available settings see `H5DataIO`

Chunking

By default, data arrays are stored *contiguously*. This means that on disk/in memory the elements of a multi-dimensional, such as, `[[1 2] [3 4]]` are actually stored in a one-dimensional buffer `1 2 3 4`. Using

chunking, allows us to break up our array into chunks so that our array will be stored not in one but multiple buffers, e.g., `[1 2] [3 4]`. Using this approach allows optimization of data locality for I/O operations and enables the application of filters (e.g., compression) on a per-chunk basis.

Tip: For an introduction to chunking and compression in HDF5 and h5py in particular see also the online book Python and HDF5 by Andrew Collette.

To use chunking we again, simply need to wrap our dataset via `H5DataIO`. Using chunking then also allows to also create resizable arrays simply by defining the `maxshape` of the array.

```
data = np.arange(10000).reshape((1000, 10))
wrapped_data = H5DataIO(data=data,
                        chunks=True,          # <---- Enable chunking
                        maxshape=(None, 10)   # <---- Make the time dimension␣
↪unlimited and hence resizeable
                        )
test_ts = TimeSeries(name='test_chunked_timeseries',
                     data=wrapped_data,                    # <----
                     unit='SIunit',
                     starting_time=0.0,
                     rate=10.0)
nwbfile.add_acquisition(test_ts)
```

Hint: By also specifying `fillvalue` we can define the value that should be used when reading uninitialized portions of the dataset. If no fill value has been defined, then HDF5 will use a type-appropriate default value.

Note: Chunking can help improve data read/write performance by allowing us to align chunks with common read/write operations. The following blog post provides an example http://geology.beer/2015/02/10/hdf-for-large-arrays/. for this. But you should also know that, with great power comes great responsibility! I.e., if you choose a bad chunk size e.g., too small chunks that don't align with our read/write operations, then chunking can also harm I/O performance.

Compression and Other I/O Filters

HDF5 supports I/O filters, i.e, data transformation (e.g, compression) that are applied transparently on read/write operations. I/O filters operate on a per-chunk basis in HDF5 and as such require the use of chunking. Chunking will be automatically enabled by h5py when compression and other I/O filters are enabled.

To use compression, we can wrap our dataset using `H5DataIO` and define the approbriate opions:

```
wrapped_data = H5DataIO(data=data,
                        compression='gzip',              # <---- Use GZip
                        compression_opts=4,              # <---- Optional GZip␣
↪aggression option
                        )
test_ts = TimeSeries(name='test_gzipped_timeseries',
                     data=wrapped_data,                    # <----
                     unit='SIunit',
                     starting_time=0.0,
                     rate=10.0)
nwbfile.add_acquisition(test_ts)
```

Hint: In addition to `compression`, `H5DataIO` also allows us to enable the `shuffle` and `fletcher32` HDF5 I/O filters.

Note: *h5py* (and *HDF5* more broadly) support a number of different compression algorithms, e.g., *GZIP*, *SZIP*, or *LZF* (or even custom compression filters). However, only *GZIP* is built by default with HDF5, i.e., while data compressed with *GZIP* can be read on all platforms and installation of HDF5, other compressors may not be installed everywhere so that not all users may be able to access those files.

Writing the data

Writing the data now works as usual.

```
from pynwb import NWBHDF5IO

io = NWBHDF5IO('advanced_io_example.nwb', 'w')
io.write(nwbfile)
io.close()
```

Reading the data

Again, nothing has changed for read. All of the above advanced I/O features are handled transparently.

```
io = NWBHDF5IO('advanced_io_example.nwb', 'r')
nwbfile = io.read()
```

Now lets have a look to confirm that all our I/O settings where indeed used.

```
for k, v in nwbfile.acquisition.items():
    print("name=%s, chunks=%s, compression=%s, maxshape=%s" % (k,
                                                              v.data.chunks,
                                                              v.data.compression,
                                                              v.data.maxshape))
```

```
name=test_regular_timeseries, chunks=None, compression=None, maxshape=(10,)
name=test_compressed_timeseries, chunks=(10,), compression=gzip, maxshape=(10,)
name=test_chunked_timeseries, chunks=(250, 5), compression=None, maxshape=(None, 10)
name=test_gzipped_timeseries, chunks=(250, 5), compression=gzip, maxshape=(1000, 10)
```

As we can see, the datasets have been chunked and compressed correctly. Alos, as expected, chunking was automatically enabled for the compressed datasets.

Wrapping `h5py.Datasets` with `H5DataIO`

Just for completeness, `H5DataIO` also allows us to customize how `h5py.Dataset` objects should be handled on write by the PyNWBs HDF5 backend via the `link_data` parameter. If `link_data` is set to `True` then a `SoftLink` or `ExternalLink` will be created to point to the HDF5 dataset On the other hand, if `link_data` is set to `False` then the dataset be copied using h5py.Group.copy **without copying attributes** and **without expanding soft links, external links, or references**.

Note: When wrapping an `h5py.Dataset` object using H5DataIO, then all settings except `link_data` will be ignored as the h5py.Dataset will either be linked to or copied as on write.

Parallel I/O using MPI

The HDF5 storage backend supports parallel I/O using the Message Passing Interface (MPI). Using this feature requires that you install `hdf5` and `h5py` against an MPI driver, and you install `mpi4py`. The basic installation of pynwb will not work. Setup can be tricky, and is outside the scope of this tutorial (for now), and the following assumes that you have HDF5 installed in a MPI configuration. Here we:

1. **Instantiate a dataset for parallel write**: We create TimeSeries with 4 timestamps that we will write in parallel

2. **Write to that file in parallel using MPI**: Here we assume 4 MPI ranks while each rank writes the data for a different timestamp.

3. **Read from the file in parallel using MPI**: Here each of the 4 MPI ranks reads one time step from the file

```
from mpi4py import MPI
import numpy as np
from dateutil import tz
from pynwb import NWBHDF5IO, NWBFile, TimeSeries
from datetime import datetime
from hdmf.data_utils import DataChunkIterator

start_time = datetime(2018, 4, 25, 2, 30, 3, tzinfo=tz.gettz('US/Pacific'))
fname = 'test_parallel_pynwb.nwb'
rank = MPI.COMM_WORLD.rank  # The process ID (integer 0-3 for 4-process run)

# Create file on one rank. Here we only instantiate the dataset we want to
# write in parallel but we do not write any data
if rank == 0:
    nwbfile = NWBFile('aa', 'aa', start_time)
    data = DataChunkIterator(data=None, maxshape=(4,), dtype=np.dtype('int'))

    nwbfile.add_acquisition(TimeSeries('ts_name', description='desc', data=data,
                                       rate=100., unit='m'))
    with NWBHDF5IO(fname, 'w') as io:
        io.write(nwbfile)

# write to dataset in parallel
with NWBHDF5IO(fname, 'a', comm=MPI.COMM_WORLD) as io:
    nwbfile = io.read()
    print(rank)
    nwbfile.acquisition['ts_name'].data[rank] = rank

# read from dataset in parallel
with NWBHDF5IO(fname, 'r', comm=MPI.COMM_WORLD) as io:
    print(io.read().acquisition['ts_name'].data[rank])
```

To specify details about chunking, compression and other HDF5-specific I/O options, we can wrap data via `H5DataIO`, e.g,

```
data = H5DataIO(DataChunkIterator(data=None, maxshape=(100000, 100),
                                  dtype=np.dtype('float')),
                                  chunks=(10, 10), maxshape=(None, None))
```

would initialize your dataset with a shape of (100000, 100) and maxshape of (None, None) and your own custom chunking of (10, 10).

Disclaimer

External links included in the tutorial are being provided as a convenience and for informational purposes only; they do not constitute an endorsement or an approval by the authors of any of the products, services or opinions of the corporation or organization or individual. The authors bear no responsibility for the accuracy, legality or content of the external site or for that of subsequent links. Contact the external site for answers to questions regarding its content.

Note: Click *here* to download the full example code

9.1.3 Modular Data Storage using External Files

PyNWB supports linking between files using external links.

Example Use Case: Integrating data from multiple files

NBWContainer classes (e.g., `TimeSeries`) support the integration of data stored in external HDF5 files with NWB data files via external links. To make things more concrete, let's look at the following use case. We want to simultaneously record multiple data streams during data acquisition. Using the concept of external links allows us to save each data stream to an external HDF5 files during data acquisition and to afterwards link the data into a single NWB:N file. In this case, each recording becomes represented by a separate file-system object that can be set as read-only once the experiment is done. In the following we are using `TimeSeries` as an example, but the same approach works for other NWBContainers as well.

Tip: The same strategies we use here for creating External Links also apply to Soft Links. The main difference between soft and external links is that soft links point to other objects within the same file while external links point to objects in external files.

Tip: In the case of `TimeSeries`, the uncorrected timestamps generated by the acquisition system can be stored (or linked) in the *sync* group. In the NWB:N format, hardware-recorded time data must then be corrected to a common time base (e.g., timestamps from all hardware sources aligned) before it can be included in the *timestamps* of the *TimeSeries*. This means, in the case of `TimeSeries` we need to be careful that we are not including data with incompatible timestamps in the same file when using external links.

Warning: External links can become stale/break. Since external links are pointing to data in other files external links may become invalid any time files are modified on the file system, e.g., renamed, moved or access permissions are changed.

Creating test data

In the following we are creating two `TimeSeries` each written to a separate file. We then show how we can integrate these files into a single NWBFile.

```
from datetime import datetime
from dateutil.tz import tzlocal
from pynwb import NWBFile
from pynwb import TimeSeries
from pynwb import NWBHDF5IO
import numpy as np

# Create the base data
start_time = datetime(2017, 4, 3, 11, tzinfo=tzlocal())
create_date = datetime(2017, 4, 15, 12, tzinfo=tzlocal())
data = np.arange(1000).reshape((100, 10))
timestamps = np.arange(100)
filename1 = 'external1_example.nwb'
filename2 = 'external2_example.nwb'
filename3 = 'external_linkcontainer_example.nwb'
filename4 = 'external_linkdataset_example.nwb'

# Create the first file
nwbfile1 = NWBFile(session_description='demonstrate external files',
                   identifier='NWBE1',
                   session_start_time=start_time,
                   file_create_date=create_date)
# Create the second file
test_ts1 = TimeSeries(name='test_timeseries1',
                      data=data,
                      unit='SIunit',
                      timestamps=timestamps)
nwbfile1.add_acquisition(test_ts1)
# Write the first file
io = NWBHDF5IO(filename1, 'w')
io.write(nwbfile1)
io.close()

# Create the second file
nwbfile2 = NWBFile(session_description='demonstrate external files',
                   identifier='NWBE2',
                   session_start_time=start_time,
                   file_create_date=create_date)
# Create the second file
test_ts2 = TimeSeries(name='test_timeseries2',
                      data=data,
                      unit='SIunit',
                      timestamps=timestamps)
nwbfile2.add_acquisition(test_ts2)
# Write the second file
io = NWBHDF5IO(filename2, 'w')
io.write(nwbfile2)
io.close()
```

Linking to select datasets

Step 1: Create the new NWBFile

```
# Create the first file
nwbfile4 = NWBFile(session_description='demonstrate external files',
```

(continues on next page)

(continued from previous page)

```
                    identifier='NWBE4',
                    session_start_time=start_time,
                    file_create_date=create_date)
```

Step 2: Get the dataset you want to link to

Now let's open our test files and retrieve our timeseries.

```
# Get the first timeseries
io1 = NWBHDF5IO(filename1, 'r')
nwbfile1 = io1.read()
timeseries_1 = nwbfile1.get_acquisition('test_timeseries1')
timeseries_1_data = timeseries_1.data
```

Step 3: Create the object you want to link to the data

To link to the dataset we can simply assign the data object (here ‘‘ timeseries_1.data‘‘) to a new `TimeSeries`

```
# Create a new timeseries that links to our data
test_ts4 = TimeSeries(name='test_timeseries4',
                      data=timeseries_1_data,    # <-------
                      unit='SIunit',
                      timestamps=timestamps)
nwbfile4.add_acquisition(test_ts4)
```

In the above case we did not make it explicit how we want to handle the data from our TimeSeries, this means that *NWBHDF5IO* will need to determine on write how to treat the dataset. We can make this explicit and customize this behavior on a per-dataset basis by wrapping our dataset using `H5DataIO`

```
from hdmf.backends.hdf5.h5_utils import H5DataIO

# Create another timeseries that links to the same data
test_ts5 = TimeSeries(name='test_timeseries5',
                      data=H5DataIO(data=timeseries_1_data,      # <-------
                                    link_data=True),             # <-------
                      unit='SIunit',
                      timestamps=timestamps)
nwbfile4.add_acquisition(test_ts5)
```

Step 4: Write the data

```
from pynwb import NWBHDF5IO

io4 = NWBHDF5IO(filename4, 'w')
io4.write(nwbfile4,
          link_data=True)      # <-------- Specify default behavior to link rather␣
↪than copy data
io4.close()
```

Note: In the case of TimeSeries one advantage of linking to just the main dataset is that we can now use our own timestamps in case the timestamps in the original file are not aligned with the clock of the NWBFile we are creating. In this way we can use the linking to "re-align" different TimeSeries without having to copy the main data.

Linking to whole Containers

Appending to files and linking is made possible by passing around the same `BuildManager`. You can get a manager to pass around using the *`get_manager`* function.

```
from pynwb import get_manager

manager = get_manager()
```

Tip: You can pass in extensions to *`get_manager`* using the *extensions* argument.

Step 1: Get the container object you want to link to

Now let's open our test files and retrieve our timeseries.

```
# Get the first timeseries
io1 = NWBHDF5IO(filename1, 'r', manager=manager)
nwbfile1 = io1.read()
timeseries_1 = nwbfile1.get_acquisition('test_timeseries1')

# Get the second timeseries
io2 = NWBHDF5IO(filename2, 'r', manager=manager)
nwbfile2 = io2.read()
timeseries_2 = nwbfile2.get_acquisition('test_timeseries2')
```

Step 2: Add the container to another NWBFile

To intergrate both *`TimeSeries`* into a single file we simply create a new *`NWBFile`* and our existing *`TimeSeries`* to it. PyNWB's *`NWBHDF5IO`* backend then automatically detects that the TimeSeries have already been written to another file and will create external links for us.

```
# Create a new NWBFile that links to the external timeseries
nwbfile3 = NWBFile(session_description='demonstrate external files',
                   identifier='NWBE3',
                   session_start_time=start_time,
                   file_create_date=create_date)
nwbfile3.add_acquisition(timeseries_1)              # <--------
nwbfile3.add_acquisition(timeseries_2)              # <--------

# Write our third file that includes our two timeseries as external links
io3 = NWBHDF5IO(filename3, 'w', manager=manager)
io3.write(nwbfile3)
io3.close()
```

Copying an NWBFile for linking

Using the *copy* method allows us to easily create a shallow copy of a whole NWB:N file with links to all data in the original file. For example, we may want to store processed data in a new file separate from the raw data, while still being able to access the raw data. See the *Exploratory data analysis with NWB* tutorial for a detailed example.

Creating a single file for sharing

External links are convenient but to share data we may want to hand a single file with all the data to our collaborator rather than having to collect all relevant files. To do this, `HDF5IO` (and in turn *NWBHDF5IO*) provide the convenience function `copy_file`, which copies an HDF5 file and resolves all external links.

Note: Click *here* to download the full example code

9.1.4 NWB basics

This example will focus on the basics of working with an *NWBFile* object, including writing and reading of an NWB file.

The NWB file

```
from datetime import datetime
from dateutil.tz import tzlocal
from pynwb import NWBFile
import numpy as np

start_time = datetime(2017, 4, 3, 11, tzinfo=tzlocal())
create_date = datetime(2017, 4, 15, 12, tzinfo=tzlocal())

nwbfile = NWBFile(session_description='demonstrate NWBFile basics',  # required
                  identifier='NWB123',  # required
                  session_start_time=start_time,  # required
                  file_create_date=create_date)  # optional
```

Time series data

PyNWB stores time series data using the *TimeSeries* class and its subclasses. The main components of a *TimeSeries* are the *data* and the *timestamps*. You will also need to supply the name and unit of measurement for *data*.

```
from pynwb import TimeSeries

data = list(range(100, 200, 10))
timestamps = list(range(10))
test_ts = TimeSeries(name='test_timeseries', data=data, unit='m',␣
↪timestamps=timestamps)
```

Alternatively, if your recordings are sampled at a uniform rate, you can supply *starting_time* and *rate*.

```
rate_ts = TimeSeries(name='test_timeseries', data=data, unit='m', starting_time=0.0,␣
↪rate=1.0)
```

Using this scheme says that this *TimeSeries* started recording 0 seconds after *start_time* stored in the *NWBFile* and sampled every second.

TimeSeries objects can be added directly to your *NWBFile* using the methods *add_acquisition*, *add_stimulus* and *add_stimulus_template*. Which method you use depends on the source of the data: use *add_acquisition* to indicated *acquisition* data, *add_stimulus* to indicate *stimulus* data, and *add_stimulus_template* to store stimulus templates.

```
nwbfile.add_acquisition(test_ts)
```

Access the *TimeSeries* object *'test_timeseries'* from *acquisition* using

```
nwbfile.acquisition['test_timeseries']
```

or

```
nwbfile.get_acquisition('test_timeseries')
```

Writing an NWB file

NWB I/O is carried out using the *NWBHDF5IO* class[1]. This class is responsible for mapping an *NWBFile* object into HDF5 according to the NWB schema.

To write an *NWBFile*, use the `write` method.

```
from pynwb import NWBHDF5IO

io = NWBHDF5IO('example_file_path.nwb', mode='w')
io.write(nwbfile)
io.close()
```

You can also use *NWBHDF5IO* as a context manager:

```
with NWBHDF5IO('example_file_path.nwb', 'w') as io:
    io.write(nwbfile)
```

Reading an NWB file

As with writing, reading is also carried out using the *NWBHDF5IO* class. To read the NWB file we just wrote, use another *NWBHDF5IO* object, and use the `read` method to retrieve an *NWBFile* object.

```
io = NWBHDF5IO('example_file_path.nwb', 'r')
nwbfile_in = io.read()
```

Retrieving data from an NWB file

```
test_timeseries_in = nwbfile_in.acquisition['test_timeseries']
print(test_timeseries_in)
```

[1] HDF5 is currently the only backend supported by NWB.

```
test_timeseries <class 'pynwb.base.TimeSeries'>
Fields:
  comments: no comments
  conversion: 1.0
  data: <HDF5 dataset "data": shape (10,), type "<i8">
  description: no description
  interval: 1
  resolution: 0.0
  timestamps: <HDF5 dataset "timestamps": shape (10,), type "<f8">
  timestamps_unit: Seconds
  unit: SIunit
```

Accessing the data field, you will notice that it does not return the data values, but instead an HDF5 dataset.

```
print(test_timeseries_in.data)
```

```
<HDF5 dataset "data": shape (10,), type "<i8">
```

This object lets you only read in a section of the dataset without reading the entire thing.

```
print(test_timeseries_in.data[:2])
```

```
[100 110]
```

To load the entire dataset, use *[:]*.

```
print(test_timeseries_in.data[:])
io.close()
```

```
[100 110 120 130 140 150 160 170 180 190]
```

If you use *NWBHDF5IO* as a context manager during read, be aware that the *NWBHDF5IO* gets closed and when the context completes and the data will not be available outside of the context manager[2].

Adding More Data

The following illustrates basic data organizational structures that are used throughout NWB:N.

Reusing timestamps

When working with multi-modal data, it can be convenient and efficient to store timestamps once and associate multiple data with the single timestamps instance. PyNWB enables this by letting you reuse timestamps across *TimeSeries* objects. To reuse a *TimeSeries* timestamps in a new *TimeSeries*, pass the existing *TimeSeries* as the new *TimeSeries* timestamps:

```
data = list(range(101, 201, 10))
reuse_ts = TimeSeries('reusing_timeseries', data, 'SIunit', timestamps=test_ts)
```

[2] Neurodata sets can be *very* large, so individual components of the dataset are only loaded into memory when you request them. This functionality is only possible if an open file handle is kept around until users want to load data.

Data interfaces

NWB provides the concept of a *data interface*–an object for a standard storage location of specific types of data–through the *NWBDataInterface* class. For example, *Position* provides a container that holds one or more *SpatialSeries* objects. *SpatialSeries* is a subtype of *TimeSeries* that represents the spatial position of an animal over time. By putting your position data into a *Position* container, downstream users and tools know where to look to retrieve position data. For a comprehensive list of available data interfaces, see the *overview page*. Here is how to create a *Position* object named '*Position*'[3].

```
from pynwb.behavior import Position

position = Position()
```

You can add objects to a data interface as a method of the data interface:

```
position.create_spatial_series(name='position1',
                               data=np.linspace(0, 1, 20),
                               rate=50.,
                               reference_frame='starting gate')
```

or you can add pre-existing objects:

```
from pynwb.behavior import SpatialSeries

spatial_series = SpatialSeries(name='position2',
                               data=np.linspace(0, 1, 20),
                               rate=50.,
                               reference_frame='starting gate')

position.add_spatial_series(spatial_series)
```

or include the object during construction:

```
spatial_series = SpatialSeries(name='position2',
                               data=np.linspace(0, 1, 20),
                               rate=50.,
                               reference_frame='starting gate')

position = Position(spatial_series=spatial_series)
```

Each data interface stores its own type of data. We suggest you read the documentation for the data interface of interest in the *API documentation* to figure out what data the data interface allows and/or requires and what methods you will need to call to add this data.

Processing modules

Processing modules are used for storing a set of data interfaces that are related to a particular processing workflow. For example, if you want to store the intermediate results of a spike sorting workflow, you could create a *ProcessingModule* that contains data interfaces that represent the common first steps in spike sorting e.g. *EventDetection*, *EventWaveform*, *FeatureExtraction*. The final results of the sorting could then be stored in the top-level *Units* table (see below). Derived preprocessed data should go in a processing module, which you can create using *create_processing_module*:

[3] Some data interface objects have a default name. This default name is the type of the data interface. For example, the default name for *ImageSegmentation* is "ImageSegmentation" and the default name for *EventWaveform* is "EventWaveform".

```
behavior_module = nwbfile.create_processing_module(name='behavior',
                                                   description='preprocessed␣
↪behavioral data')
```

or by directly calling the constructor and adding to the *NWBFile* using *add_processing_module*:

```
from pynwb import ProcessingModule

ecephys_module = ProcessingModule(name='ecephys',
                                  description='preprocessed extracellular␣
↪electrophysiology')
nwbfile.add_processing_module(ecephys_module)
```

Best practice is to use the NWB schema module names as processing module names where appropriate. These are: 'behavior', 'ecephys', 'icephys', 'ophys', 'ogen', 'retinotopy', and 'misc'. You may also create a processing module with a custom name. Once these processing modules are added, access them with

```
nwbfile.processing
```

which returns a *dict*:

```
{'behavior':
 behavior <class 'pynwb.base.ProcessingModule'>
 Fields:
   data_interfaces: { Position <class 'pynwb.behavior.Position'> }
   description: preprocessed behavioral data, 'ecephys':
 ecephys <class 'pynwb.base.ProcessingModule'>
 Fields:
   data_interfaces: { }
   description: preprocessed extracellular electrophysiology}
```

NWBDataInterface objects can be added to the behavior *ProcessingModule*.

```
nwbfile.processing['behavior'].add(position)
```

Epochs

Epochs can be added to an NWB file using the method *add_epoch*. The first and second arguments are the start time and stop times, respectively. The third argument is one or more tags for labelling the epoch, and the fifth argument is a list of all the *TimeSeries* that the epoch applies to.

```
nwbfile.add_epoch(2.0, 4.0, ['first', 'example'], [test_ts, ])
nwbfile.add_epoch(6.0, 8.0, ['second', 'example'], [test_ts, ])
```

Trials

Trials can be added to an NWB file using the methods *add_trial* and *add_trial_column*. Together, these methods maintains a table-like structure that can define arbitrary columns without having to go through the extension process.

By default, NWBFile only requires trial start time and trial end time. Additional columns can be added using *add_trial_column*. This method takes a name for the column and a description of what the column stores. You do not need to supply data type, as this will inferred. Once all columns have been added, trial data can be populated using *add_trial*.

Lets add an additional column and some trial data.

```
nwbfile.add_trial_column(name='stim', description='the visual stimuli during the trial
↪')

nwbfile.add_trial(start_time=0.0, stop_time=2.0, stim='person')
nwbfile.add_trial(start_time=3.0, stop_time=5.0, stim='ocean')
nwbfile.add_trial(start_time=6.0, stop_time=8.0, stim='desert')
```

Tabular data such as trials can be converted to a *pandas.DataFrame*.

```
print(nwbfile.trials.to_dataframe())
```

```
    start_time  stop_time    stim
id
0          0.0        2.0  person
1          3.0        5.0   ocean
2          6.0        8.0  desert
```

Units

Units are putative cells in your analysis. Unit metadata can be added to an NWB file using the methods *add_unit* and *add_unit_column*. These methods work like the methods for adding trials described *above*

A unit is only required to contain a unique integer identifier in the 'id' column (this will be automatically assigned if not provided). Additional optional values for each unit include: *spike_times*, *electrodes*, *electrode_group*, *obs_intervals*, *waveform_mean*, and *waveform_sd*. Additional user-defined columns can be added using *add_unit_column*. Like *add_trial_column*, this method also takes a name for the column, a description of what the column stores and does not need a data type. Once all columns have been added, unit data can be populated using *add_unit*.

When providing *spike_times*, you may also wish to specify the time intervals during which the unit was being observed, so that it is possible to distinguish times when the unit was silent from times when the unit was not being recorded (and thus correctly compute firing rates, for example). This information should be provided as a list of [start, end] time pairs in the *obs_intervals* field. If *obs_intervals* is provided, then all entries in *spike_times* should occur within one of the listed intervals. In the example below, all 3 units are observed during the time period from 1 to 10 s and fired spikes during that period. Units 2 and 3 were also observed during the time period from 20-30s; but only unit 2 fired spikes in that period.

Lets specify some unit metadata and then add some units:

```
nwbfile.add_unit_column('location', 'the anatomical location of this unit')
nwbfile.add_unit_column('quality', 'the quality for the inference of this unit')

nwbfile.add_unit(id=1, spike_times=[2.2, 3.0, 4.5],
                 obs_intervals=[[1, 10]], location='CA1', quality=0.95)
nwbfile.add_unit(id=2, spike_times=[2.2, 3.0, 25.0, 26.0],
                 obs_intervals=[[1, 10], [20, 30]], location='CA3', quality=0.85)
nwbfile.add_unit(id=3, spike_times=[1.2, 2.3, 3.3, 4.5],
                 obs_intervals=[[1, 10], [20, 30]], location='CA1', quality=0.90)
```

Now we overwrite the file with all of the data

```
with NWBHDF5IO('example_file_path.nwb', 'w') as io:
    io.write(nwbfile)
```

Note: The Units table has some predefined optional columns. Please review the documentation for *add_unit* before adding custom columns.

Appending to an NWB file

Using functionality discussed above, NWB allows appending to files. To append to a file, you must read the file, add new components, and then write the file. Reading and writing is carried out using *NWBHDF5IO*. When reading the NWBFile, you must specify that you intend to modify it by setting the *mode* argument in the *NWBHDF5IO* constructor to `'a'`. After you have read the file, you can add[4] new data to it using the standard write/add functionality demonstrated above.

Let's see how this works by adding another *TimeSeries* to the BehavioralTimeSeries interface we created above.

First, read the file and get the interface object.

```
io = NWBHDF5IO('example_file_path.nwb', mode='a')
nwbfile = io.read()
position = nwbfile.processing['behavior'].data_interfaces['Position']
```

Next, add a new *SpatialSeries*.

```
data = list(range(300, 400, 10))
timestamps = list(range(10))
test_spatial_series = SpatialSeries('test_spatialseries2', data,
                                    reference_frame='starting_gate',
                                    timestamps=timestamps)
position.add_spatial_series(test_spatial_series)
```

Finally, write the changes back to the file and close it.

```
io.write(nwbfile)
io.close()
```

Note: Click *here* to download the full example code

9.1.5 Extensions

The NWB:N format was designed to be easily extendable. Here we will demonstrate how to extend NWB using the PyNWB API.

Note: A more in-depth discussion of the components and steps for creating and using extensions is available as part of the docs at *Extending NWB*.

Defining extensions

Extensions should be defined separately from the code that uses the extensions. This design decision is based on the assumption that the extension will be written once, and read or used multiple times. Here, we provide an example of

[4] NWB only supports *adding* to files. Removal and modifying of existing data is not allowed.

how to create an extension for subsequent use. (For more information on the available tools for creating extensions, see *Extending NWB*).

The following block of code demonstrates how to create a new namespace, and then add a new *neurodata_type* to this namespace. Finally, it calls `export` to save the extensions to disk for downstream use.

```
from pynwb.spec import NWBNamespaceBuilder, NWBGroupSpec, NWBAttributeSpec

ns_path = "mylab.namespace.yaml"
ext_source = "mylab.extensions.yaml"

ns_builder = NWBNamespaceBuilder('Extension for use in my Lab', "mylab")

ns_builder.include_type('ElectricalSeries', namespace='core')

ext = NWBGroupSpec('A custom ElectricalSeries for my lab',
                   attributes=[NWBAttributeSpec('trode_id', 'the tetrode id', 'int')],
                   neurodata_type_inc='ElectricalSeries',
                   neurodata_type_def='TetrodeSeries')

ns_builder.add_spec(ext_source, ext)
ns_builder.export(ns_path)
```

Running this block will produce two YAML files.

The first file, mylab.namespace.yaml, contains the specification of the namespace.

```
namespaces:
- doc: Extension for use in my Lab
  name: mylab
  schema:
  - namespace: core
    neurodata_type:
    - ElectricalSeries
  - source: mylab.extensions.yaml
```

The second file, mylab.extensions.yaml, contains the details on newly defined types.

```
groups:
- attributes:
  - doc: the tetrode id
    dtype: int
    name: trode_id
  doc: A custom ElectricalSeries for my lab
  neurodata_type_def: TetrodeSeries
  neurodata_type_inc: ElectricalSeries
```

Tip: Detailed documentation of all components and *neurodata_types* that are part of the core schema of NWB:N are available in the schema docs at http://nwb-schema.readthedocs.io . Before creating a new type from scratch, please have a look at the schema docs to see if using or extending an existing type may solve your problem. Also, the schema docs are helpful when extending an existing type to better understand the design and structure of the neurodata_type you are using.

Using extensions

After an extension has been created, it can be used by downstream codes for reading and writing data. There are two main mechanisms for reading and writing extension data with PyNWB. The first involves defining new *NWBContainer* classes that are then mapped to the neurodata types in the extension.

```
from pynwb import register_class, load_namespaces
from pynwb.ecephys import ElectricalSeries
from hdmf.utils import docval, call_docval_func, getargs, get_docval

ns_path = "mylab.namespace.yaml"
load_namespaces(ns_path)


@register_class('TetrodeSeries', 'mylab')
class TetrodeSeries(ElectricalSeries):

    __nwbfields__ = ('trode_id',)

    @docval(*get_docval(ElectricalSeries.__init__) + (
        {'name': 'trode_id', 'type': int, 'doc': 'the tetrode id'},))
    def __init__(self, **kwargs):
        call_docval_func(super(TetrodeSeries, self).__init__, kwargs)
        self.trode_id = getargs('trode_id', kwargs)
```

Note: See the API docs for more information about `docval` `call_docval_func`, `getargs` and `get_docval`

When extending *NWBContainer* or *NWBContainer* subclasses, you should define the class field `__nwbfields__`. This will tell PyNWB the properties of the *NWBContainer* extension.

If you do not want to write additional code to read your extensions, PyNWB is able to dynamically create an *NWBContainer* subclass for use within the PyNWB API. Dynamically created classes can be inspected using the built-in `inspect` module.

```
from pynwb import get_class, load_namespaces

ns_path = "mylab.namespace.yaml"
load_namespaces(ns_path)

AutoTetrodeSeries = get_class('TetrodeSeries', 'mylab')
```

Note: When defining your own *NWBContainer*, the subclass name does not need to be the same as the extension type name. However, it is encouraged to keep class and extension names the same for the purposes of readibility.

Caching extensions to file

By default, extensions are cached to file so that your NWB file will carry the extensions needed to read the file with it.

To demonstrate this, first we will make some fake data using our extensions.

```
from datetime import datetime
from dateutil.tz import tzlocal
from pynwb import NWBFile
```

(continues on next page)

(continued from previous page)

```
start_time = datetime(2017, 4, 3, 11, tzinfo=tzlocal())
create_date = datetime(2017, 4, 15, 12, tzinfo=tzlocal())

nwbfile = NWBFile('demonstrate caching', 'NWB456', start_time,
                  file_create_date=create_date)

device = nwbfile.create_device(name='trodes_rig123')

electrode_name = 'tetrode1'
description = "an example tetrode"
location = "somewhere in the hippocampus"

electrode_group = nwbfile.create_electrode_group(electrode_name,
                                                 description=description,
                                                 location=location,
                                                 device=device)
for idx in [1, 2, 3, 4]:
    nwbfile.add_electrode(id=idx,
                          x=1.0, y=2.0, z=3.0,
                          imp=float(-idx),
                          location='CA1', filtering='none',
                          group=electrode_group)
electrode_table_region = nwbfile.create_electrode_table_region([0, 2], 'the first and
↪third electrodes')

import numpy as np

rate = 10.0
np.random.seed(1234)
data_len = 1000
data = np.random.rand(data_len * 2).reshape((data_len, 2))
timestamps = np.arange(data_len) / rate

ts = TetrodeSeries('test_ephys_data',
                   data,
                   electrode_table_region,
                   timestamps=timestamps,
                   trode_id=1,
                   # Alternatively, could specify starting_time and rate as follows
                   # starting_time=ephys_timestamps[0],
                   # rate=rate,
                   resolution=0.001,
                   comments="This data was randomly generated with numpy, using 1234
↪as the seed",
                   description="Random numbers generated with numpy.random.rand")
nwbfile.add_acquisition(ts)
```

Note: For more information on writing `ElectricalSeries`, see *Extracellular electrophysiology data*.

Now that we have some data, lets write our file. You can choose not to cache the spec by setting cache_spec=False in `write`

```
from pynwb import NWBHDF5IO

```

(continues on next page)

(continued from previous page)

```
io = NWBHDF5IO('cache_spec_example.nwb', mode='w')
io.write(nwbfile)
io.close()
```

Note: For more information on writing NWB files, see *Writing an NWB file*.

By default, PyNWB does not use the namespaces cached in a file–you must explicitly specify this. This behavior is enabled by the *load_namespaces* argument to the *NWBHDF5IO* constructor.

```
io = NWBHDF5IO('cache_spec_example.nwb', mode='r', load_namespaces=True)
nwbfile = io.read()
```

Creating and using a custom MultiContainerInterface

It is sometimes the case that we need a group to hold zero-or-more or one-or-more of the same object. Here we show how to create an extension that defines a group (*PotatoSack*) that holds multiple objects (*Pototo* es) and then how to use the new data types. First, we use *pynwb* to define the new data types.

```
from pynwb.spec import NWBNamespaceBuilder, NWBGroupSpec, NWBAttributeSpec

name = 'test_multicontainerinterface'
ns_path = name + ".namespace.yaml"
ext_source = name + ".extensions.yaml"

ns_builder = NWBNamespaceBuilder(name + ' extensions', name)
ns_builder.include_type('NWBDataInterface', namespace='core')

potato = NWBGroupSpec(neurodata_type_def='Potato',
                      neurodata_type_inc='NWBDataInterface',
                      doc='A potato', quantity='*',
                      attributes=[
                          NWBAttributeSpec(name='weight',
                                           doc='weight of potato',
                                           dtype='float',
                                           required=True),
                          NWBAttributeSpec(name='age',
                                           doc='age of potato',
                                           dtype='float',
                                           required=False)
                      ])

potato_sack = NWBGroupSpec(neurodata_type_def='PotatoSack',
                           neurodata_type_inc='NWBDataInterface',
                           name='potato_sack',
                           doc='A sack of potatoes', quantity='?',
                           groups=[potato])

ns_builder.add_spec(ext_source, potato_sack)
ns_builder.export(ns_path)
```

Then create Container classes registered to the new data types (this is generally done in a different file)

```
from pynwb import register_class, load_namespaces
from pynwb.file import MultiContainerInterface, NWBContainer

load_namespaces(ns_path)


@register_class('Potato', name)
class Potato(NWBContainer):
    __nwbfields__ = ('name', 'weight', 'age')

    @docval({'name': 'name', 'type': str, 'doc': 'who names a potato?'},
            {'name': 'weight', 'type': float, 'doc': 'weight of potato in grams'},
            {'name': 'age', 'type': float, 'doc': 'age of potato in days'})
    def __init__(self, **kwargs):
        super(Potato, self).__init__(name=kwargs['name'])
        self.weight = kwargs['weight']
        self.age = kwargs['age']


@register_class('PotatoSack', name)
class PotatoSack(MultiContainerInterface):

    __clsconf__ = {
        'attr': 'potatos',
        'type': Potato,
        'add': 'add_potato',
        'get': 'get_potato',
        'create': 'create_potato',
    }
```

Then use the objects (again, this would often be done in a different file).

```
from pynwb import NWBHDF5IO, NWBFile
from datetime import datetime
from dateutil.tz import tzlocal

# You can add potatoes to a potato sack in different ways
potato_sack = PotatoSack(potatos=Potato(name='potato1', age=2.3, weight=3.0))
potato_sack.add_potato(Potato('potato2', 3.0, 4.0))
potato_sack.create_potato('big_potato', 10.0, 20.0)

nwbfile = NWBFile("a file with metadata", "NB123A", datetime(2018, 6, 1,␣
↪tzinfo=tzlocal()))

pmod = nwbfile.create_processing_module('module_name', 'desc')
pmod.add_container(potato_sack)


with NWBHDF5IO('test_multicontainerinterface.nwb', 'w') as io:
    io.write(nwbfile)
```

This is how you read the NWB file (again, this would often be done in a different file).

```
load_namespaces(ns_path)
# from xxx import PotatoSack, Potato
io = NWBHDF5IO('test_multicontainerinterface.nwb', 'r')
nwb = io.read()
```

(continues on next page)

(continued from previous page)

```
print(nwb.get_processing_module()['potato_sack'].get_potato('big_potato').weight)
# note: you can call get_processing_module() with or without the module name as
# an argument. however, if there is more than one module, the name is required.
# here, there is more than one potato, so the name of the potato is required as
# an argument to get get_potato

io.close()
```

Example: Cortical Surface Mesh

Here we show how to create extensions by creating a data class for a cortical surface mesh. This data type is particularly important for ECoG data, we need to know where each electrode is with respect to the gyri and sucli. Surface mesh objects contain two types of data:

1. *vertices*, which is an (n, 3) matrix of floats that represents points in 3D space

2. *faces*, which is an (m, 3) matrix of uints that represents indices of the *vertices* matrix. Each triplet of points defines a triangular face, and the mesh is comprised of a collection of triangular faces.

First, we set up our extension. I am going to use the name *ecog*

```
from pynwb.spec import NWBDatasetSpec, NWBNamespaceBuilder, NWBGroupSpec

name = 'ecog'
ns_path = name + ".namespace.yaml"
ext_source = name + ".extensions.yaml"

# Now we define the data structures. We use `NWBDataInterface` as the base type,
# which is the most primitive type you are likely to use as a base. The name of the
# class is `CorticalSurface`, and it requires two matrices, `vertices` and
# `faces`.

surface = NWBGroupSpec(doc='brain cortical surface',
                       datasets=[
                           NWBDatasetSpec(doc='faces for surface, indexes vertices',␣
↪shape=(None, 3),
                                          name='faces', dtype='uint', dims=('face_
↪number', 'vertex_index')),
                           NWBDatasetSpec(doc='vertices for surface, points in 3D␣
↪space', shape=(None, 3),
                                          name='vertices', dtype='float', dims=(
↪'vertex_number', 'xyz'))],
                       neurodata_type_def='CorticalSurface',
                       neurodata_type_inc='NWBDataInterface')

# Now we set up the builder and add this object

ns_builder = NWBNamespaceBuilder(name + ' extensions', name)
ns_builder.add_spec(ext_source, surface)
ns_builder.export(ns_path)

################
# The above should generate 2 YAML files. `ecog.extensions.yaml`,
# defines the newly defined types
#
# .. code-block:: yaml
```

(continues on next page)

(continued from previous page)

```
#
#     # ecog.namespace.yaml
#       groups:
#       - datasets:
#       - dims:
#           - face_number
#           - vertex_index
#           doc: faces for surface, indexes vertices
#           dtype: uint
#           name: faces
#           shape:
#           - null
#           - 3
#       - dims:
#           - vertex_number
#           - xyz
#           doc: vertices for surface, points in 3D space
#           dtype: float
#           name: vertices
#           shape:
#           - null
#           - 3
#       doc: brain cortical surface
#       neurodata_type_def: CorticalSurface
#       neurodata_type_inc: NWBDataInterface
#
# Finally, we should test the new types to make sure they run as expected

from pynwb import load_namespaces, get_class, NWBHDF5IO, NWBFile
from datetime import datetime
import numpy as np

load_namespaces('ecog.namespace.yaml')
CorticalSurface = get_class('CorticalSurface', 'ecog')

cortical_surface = CorticalSurface(vertices=[[0.0, 1.0, 1.0],
                                             [1.0, 1.0, 2.0],
                                             [2.0, 2.0, 1.0],
                                             [2.0, 1.0, 1.0],
                                             [1.0, 2.0, 1.0]],
                                   faces=np.array([[0, 1, 2], [1, 2, 3]]).astype('uint
↪'),
                                   name='cortex')

nwbfile = NWBFile('my first synthetic recording', 'EXAMPLE_ID', datetime.now())

cortex_module = nwbfile.create_processing_module(name='cortex',
                                                 description='description')
cortex_module.add_container(cortical_surface)


with NWBHDF5IO('test_cortical_surface.nwb', 'w') as io:
    io.write(nwbfile)
```

Note: Click *here* to download the full example code

9.1.6 Iterative Data Write

This example demonstrate how to iteratively write data arrays with applications to writing large arrays without loading all data into memory and streaming data write.

Introduction

What is Iterative Data Write?

In the typical write process, datasets are created and written as a whole. In contrast, iterative data write refers to the writing of the content of a dataset in an incremental, iterative fashion.

Why Iterative Data Write?

The possible applications for iterative data write are broad. Here we list a few typical applications for iterative data write in practice.

- **Large data arrays** A central challenge when dealing with large data arrays is that it is often not feasible to load all of the data into memory. Using an iterative data write process allows us to avoid this problem by writing the data one-subblock-at-a-time, so that we only need to hold a small subset of the array in memory at any given time.
- **Data streaming** In the context of streaming data we are faced with several issues: **1)** data is not available in memory but arrives in subblocks as the stream progresses **2)** caching the data of a stream in-memory is often prohibitively expensive and volatile **3)** the total size of the data is often unknown ahead of time. Iterative data write allows us to address issues 1) and 2) by enabling us to save data to file incrementally as it arrives from the data stream. Issue 3) is addressed in the HDF5 storage backend via support for chunking, enabling the creation of resizable arrays.
  - **Data generators** Data generators are in many ways similar to data streams only that the data is typically being generated locally and programmatically rather than from an external data source.
- **Sparse data arrays** In order to reduce storage size of sparse arrays a challenge is that while the data array (e.g., a matrix) may be large, only few values are set. To avoid storage overhead for storing the full array we can employ (in HDF5) a combination of chunking, compression, and and iterative data write to significantly reduce storage cost for sparse data.

Iterating Over Data Arrays

In PyNWB the process of iterating over large data arrays is implemented via the concept of `DataChunk` and `AbstractDataChunkIterator`.

- `DataChunk` is a simple data structure used to describe a subset of a larger data array (i.e., a data chunk), consisting of:
  - `DataChunk.data` : the array with the data value(s) of the chunk and
  - `DataChunk.selection` : the NumPy index tuple describing the location of the chunk in the whole array.
- `AbstractDataChunkIterator` then defines a class for iterating over large data arrays one-`DataChunk`-at-a-time.
- `DataChunkIterator` is a specific implementation of an `AbstractDataChunkIterator` that accepts any iterable and assumes that we iterate over the first dimension of the data array. `DataChunkIterator` also

supports buffered read, i.e., multiple values from the input iterator can be combined to a single chunk. This is useful for buffered I/O operations, e.g., to improve performance by accumulating data in memory and writing larger blocks at once.

Iterative Data Write: API

On the front end, all a user needs to do is to create or wrap their data in a `AbstractDataChunkIterator`. The I/O backend (e.g., `HDF5IO` or *`NWBHDF5IO`*) then implements the iterative processing of the data chunk iterators. PyNWB also provides with `DataChunkIterator` a specific implementation of a data chunk iterator which we can use to wrap common iterable types (e.g., generators, lists, or numpy arrays). For more advanced use cases we then need to implement our own derived class of `AbstractDataChunkIterator`.

Tip: Currently the HDF5 I/O backend of PyNWB (`HDF5IO`, *`NWBHDF5IO`*) processes itertive data writes one-dataset-at-a-time. This means, that while you may have an arbitrary number of iterative data writes, the write is performed in order. In the future we may use a queing process to enable the simultaneous processing of multiple iterative writes at the same time.

Preparations:

The data write in our examples really does not change. We, therefore, here create a simple helper function first to write a simple NWBFile containing a single timeseries to avoid repition of the same code and to allow us to focus on the important parts of this tutorial.

```
from datetime import datetime
from dateutil.tz import tzlocal
from pynwb import NWBFile, TimeSeries
from pynwb import NWBHDF5IO


def write_test_file(filename, data):
    """
    Simple helper function to write an NWBFile with a single timeseries containing␣
↪data
    :param filename: String with the name of the output file
    :param data: The data of the timeseries
    """

    # Create a test NWBfile
    start_time = datetime(2017, 4, 3, 11, tzinfo=tzlocal())
    create_date = datetime(2017, 4, 15, 12, tzinfo=tzlocal())
    nwbfile = NWBFile('demonstrate NWBFile basics',
                      'NWB123',
                      start_time,
                      file_create_date=create_date)

    # Create our time series
    test_ts = TimeSeries(name='synthetic_timeseries',
                         data=data,                         # <---------
                         unit='SIunit',
                         rate=1.0,
                         starting_time=0.0)
    nwbfile.add_acquisition(test_ts)
```

(continues on next page)

(continued from previous page)

```
    # Write the data to file
    io = NWBHDF5IO(filename, 'w')
    io.write(nwbfile)
    io.close()
```

Example: Write Data from Generators and Streams

Here we use a simple data generator but PyNWB does not make any assumptions about what happens inside the generator. Instead of creating data programmatically, you may hence, e.g., receive data from an acquisition system (or other source). We can, hence, use the same approach to write streaming data.

Step 1: Define the data generator

```
from math import sin, pi
from random import random
import numpy as np


def iter_sin(chunk_length=10, max_chunks=100):
    """
    Generator creating a random number of chunks (but at most max_chunks) of length␣
↪chunk_length containing
    random samples of sin([0, 2pi]).
    """
    x = 0
    num_chunks = 0
    while(x < 0.5 and num_chunks < max_chunks):
        val = np.asarray([sin(random() * 2 * pi) for i in range(chunk_length)])
        x = random()
        num_chunks += 1
        yield val
    return
```

Step 2: Wrap the generator in a DataChunkIterator

```
from hdmf.data_utils import DataChunkIterator

data = DataChunkIterator(data=iter_sin(10))
```

Step 3: Write the data as usual

Here we use our wrapped generator to create the data for a synthetic time series.

```
write_test_file(filename='basic_iterwrite_example.nwb',
                data=data)
```

Discussion

Note, we here actually do not know how long our timeseries will be.

```
print("maxshape=%s, recommended_data_shape=%s, dtype=%s" % (str(data.maxshape),
                                                           str(data.recommended_data_
↪shape()),
                                                           str(data.dtype)))
```

[Out]:

```
maxshape=(None, 10), recommended_data_shape=(1, 10), dtype=float64
```

As we can see `DataChunkIterator` automatically recommends in its `maxshape` that the first dimensions of our array should be unlimited (`None`) and the second dimension be `10` (i.e., the length of our chunk. Since `DataChunkIterator` has no way of knowing the minimum size of the array it automatically recommends the size of the first chunk as the minimum size (i.e, `(1, 10)`) and also infers the data type automatically from the first chunk. To further customize this behavior we may also define the `maxshape`, `dtype`, and `buffer_size` when we create the `DataChunkIterator`.

Tip: We here used `DataChunkIterator` to conveniently wrap our data stream. `DataChunkIterator` assumes that our generators yields in **consecutive order single** complete element along the **first dimension** of our a array (i.e., iterate over the first axis and yield one-element-at-a-time). This behavior is useful in many practical cases. However, if this strategy does not match our needs, then you can alternatively implement our own derived `AbstractDataChunkIterator`. We show an example of this next.

Example: Optimizing Sparse Data Array I/O and Storage

Step 1: Create a data chunk iterator for our sparse matrix

```
from hdmf.data_utils import AbstractDataChunkIterator, DataChunk


class SparseMatrixIterator(AbstractDataChunkIterator):

    def __init__(self, shape, num_chunks, chunk_shape):
        """
        :param shape: 2D tuple with the shape of the matrix
        :param num_chunks: Number of data chunks to be created
        :param chunk_shape: The shape of each chunk to be created
        :return:
        """
        self.shape, self.num_chunks, self.chunk_shape = shape, num_chunks, chunk_shape
        self.__chunks_created = 0

    def __iter__(self):
        return self

    def __next__(self):
        """
        Return in each iteration a fully occupied data chunk of self.chunk_shape␣
↪values at a random
        location within the matrix. Chunks are non-overlapping. REMEMBER: h5py does␣
↪not support all
```

(continues on next page)

(continued from previous page)

```
        fancy indexing that numpy does so we need to make sure our selection can be
        handled by the backend.
        """
        if self.__chunks_created < self.num_chunks:
            data = np.random.rand(np.prod(self.chunk_shape)).reshape(self.chunk_shape)
            xmin = np.random.randint(0, int(self.shape[0] / self.chunk_shape[0]),␣
↪1)[0] * self.chunk_shape[0]
            xmax = xmin + self.chunk_shape[0]
            ymin = np.random.randint(0, int(self.shape[1] / self.chunk_shape[1]),␣
↪1)[0] * self.chunk_shape[1]
            ymax = ymin + self.chunk_shape[1]
            self.__chunks_created += 1
            return DataChunk(data=data,
                             selection=np.s_[xmin:xmax, ymin:ymax])
        else:
            raise StopIteration

    next = __next__

    def recommended_chunk_shape(self):
        # Here we can optionally recommend what a good chunking should be.
        return self.chunk_shape

    def recommended_data_shape(self):
        # We know the full size of the array. In cases where we don't know the full␣
↪size
        # this should be the minimum size.
        return self.shape

    @property
    def dtype(self):
        # The data type of our array
        return np.dtype(float)

    @property
    def maxshape(self):
        # We know the full shape of the array. If we don't know the size of a␣
↪dimension
        # beforehand we can set the dimension to None instead
        return self.shape
```

Step 2: Instantiate our sparse matrix

```
# Setting for our random sparse matrix
xsize = 1000000
ysize = 1000000
num_chunks = 1000
chunk_shape = (10, 10)
num_values = num_chunks * np.prod(chunk_shape)

# Create our sparse matrix data.
data = SparseMatrixIterator(shape=(xsize, ysize),
                            num_chunks=num_chunks,
                            chunk_shape=chunk_shape)
```

In order to also enable compression and other advanced HDF5 dataset I/O featurs we can then also wrap our data via `H5DataIO`.

```
from hdmf.backends.hdf5.h5_utils import H5DataIO
matrix2 = SparseMatrixIterator(shape=(xsize, ysize),
                               num_chunks=num_chunks,
                               chunk_shape=chunk_shape)
data2 = H5DataIO(data=matrix2,
                 compression='gzip',
                 compression_opts=4)
```

We can now also customize the chunking , fillvalue and other settings

```
from hdmf.backends.hdf5.h5_utils import H5DataIO

# Increase the chunk size and add compression
matrix3 = SparseMatrixIterator(shape=(xsize, ysize),
                               num_chunks=num_chunks,
                               chunk_shape=chunk_shape)
data3 = H5DataIO(data=matrix3,
                 chunks=(100, 100),
                 fillvalue=np.nan)

# Increase the chunk size and add compression
matrix4 = SparseMatrixIterator(shape=(xsize, ysize),
                               num_chunks=num_chunks,
                               chunk_shape=chunk_shape)
data4 = H5DataIO(data=matrix4,
                 compression='gzip',
                 compression_opts=4,
                 chunks=(100, 100),
                 fillvalue=np.nan
                 )
```

Step 3: Write the data as usual

Here we simply use our `SparseMatrixIterator` as input for our `TimeSeries`

```
write_test_file(filename='basic_sparse_iterwrite_example.nwb',
                data=data)
write_test_file(filename='basic_sparse_iterwrite_compressed_example.nwb',
                data=data2)
write_test_file(filename='basic_sparse_iterwrite_largechunks_example.nwb',
                data=data3)
write_test_file(filename='basic_sparse_iterwrite_largechunks_compressed_example.nwb',
                data=data4)
```

Check the results

Now lets check out the size of our data file and compare it against the expected full size of our matrix

```
import os

expected_size = xsize * ysize * 8              # This is the full size of our matrix␣
↪in byte
```

(continues on next page)

(continued from previous page)

```
occupied_size = num_values * 8    # Number of non-zero values in out matrix
file_size = os.stat('basic_sparse_iterwrite_example.nwb').st_size  # Real size of the␣
↪file
file_size_compressed = os.stat('basic_sparse_iterwrite_compressed_example.nwb').st_
↪size
file_size_largechunks = os.stat('basic_sparse_iterwrite_largechunks_example.nwb').st_
↪size
file_size_largechunks_compressed = os.stat('basic_sparse_iterwrite_largechunks_
↪compressed_example.nwb').st_size
mbfactor = 1000. * 1000  # Factor used to convert to MegaBytes

print("1) Sparse Matrix Size:")
print("   Expected Size :  %.2f MB" % (expected_size / mbfactor))
print("   Occupied Size :  %.5f MB" % (occupied_size / mbfactor))
print("2) NWB:N HDF5 file (no compression):")
print("   File Size     :  %.2f MB" % (file_size / mbfactor))
print("   Reduction     :  %.2f x" % (expected_size / file_size))
print("3) NWB:N HDF5 file (with GZIP compression):")
print("   File Size     :  %.5f MB" % (file_size_compressed / mbfactor))
print("   Reduction     :  %.2f x" % (expected_size / file_size_compressed))
print("4) NWB:N HDF5 file (large chunks):")
print("   File Size     :  %.5f MB" % (file_size_largechunks / mbfactor))
print("   Reduction     :  %.2f x" % (expected_size / file_size_largechunks))
print("5) NWB:N HDF5 file (large chunks with compression):")
print("   File Size     :  %.5f MB" % (file_size_largechunks_compressed / mbfactor))
print("   Reduction     :  %.2f x" % (expected_size / file_size_largechunks_
↪compressed))
```

[Out]:

```
1) Sparse Matrix Size:
   Expected Size :  8000000.00 MB
   Occupied Size :  0.80000 MB
2) NWB:N HDF5 file (no compression):
   File Size     :  0.89 MB
   Reduction     :  9035219.28 x
3) NWB:N HDF5 file (with GZIP compression):
   File Size     :  0.88847 MB
   Reduction     :  9004283.79 x
4) NWB:N HDF5 file (large chunks):
   File Size     :  80.08531 MB
   Reduction     :  99893.47 x
5) NWB:N HDF5 file (large chunks with compression):
   File Size     :  1.14671 MB
   Reduction     :  6976450.12 x
```

Discussion

- **1) vs 2):** While the full matrix would have a size of `8TB` the HDF5 file is only `0.88MB`. This is roughly the same as the real occupied size of `0.8MB`. When using chunking, HDF5 does not allocate the full dataset but only allocates chunks that actually contain data. In (2) the size of our chunks align perfectly with the occupied chunks of our sparse matrix, hence, only the minimal amount of storage needs to be allocated. A slight overhead (here 0.08MB) is expected because our file contains also the additional objects from the NWBFile, plus some overhead for managing all the HDF5 metadata for all objects.

- **3) vs 2):** Adding compression does not yield any improvement here. This is expected, because, again we selected the chunking here in a way that we already allocated the minimum amount of storage to represent our data and lossless compression of random data is not efficient.
- **4) vs 2):** When we increase our chunk size to `(100,100)` (i.e., `100x` larger than the chunks produced by our matrix generator) we observe an according roughly `100x` increase in file size. This is expected since our chunks now do not align perfectly with the occupied data and each occupied chunk is allocated fully.
- **5) vs 4):** When using compression for the larger chunks we see a significant reduction in file size (`1.14MB` vs. `80MB`). This is because the allocated chunks now contain in addition to the random values large areas of constant fillvalues, which compress easily.

Advantages:

- We only need to hold one `DataChunk` in memory at any given time
- Only the data chunks in the HDF5 file that contain non-default values are ever being allocated
- The overall size of our file is reduced significantly
- Reduced I/O load
- On read users can use the array as usual

Tip: With great power comes great responsibility **!** I/O and storage cost will depend among others on the chunk size, compression options, and the write pattern, i.e., the number and structure of the `DataChunk` objects written. For example, using `(1,1)` chunks and writing them one value at a time would result in poor I/O performance in most practical cases, because of the large number of chunks and large number of small I/O operations required.

Tip:

A word of caution, while this approach helps optimize storage, the in-memory representation on read is still a dense numpy array. This behavior is convenient for many user interactions with the data but can be problematic with regard to performance/memory when accessing large data subsets.

```
io = NWBHDF5IO('basic_sparse_iterwrite_example.nwb', 'r')
nwbfile = io.read()
data = nwbfile.get_acquisition('synthetic_timeseries').data  # <-- PyNWB does lazy␣
↪load; no problem
subset = data[10:100, 10:100]                                # <-- Loading a subset␣
↪is fine too
alldata = data[:]            # <-- !!!! This would load the complete (1000000 x␣
↪1000000) array !!!!
```

Tip: As we have seen here, our data chunk iterator may produce chunks in arbitrary order and locations within the array. In the case of the HDF5 I/O backend we need to take care that the selection we yield can be understood by h5py.

Example: Convert large binary data arrays

When converting large data files, a typical problem is that it is often too expensive to load all the data into memory. This example is very similar to the data generator example only that instead of generating data on-the-fly in memory we are loading data from a file one-chunk-at-a-time in our generator.

Create example data

```
import numpy as np
# Create the test data
datashape = (100, 10)   # OK, this not really large, but we just want to show how it␣
↪works
num_values = np.prod(datashape)
arrdata = np.arange(num_values).reshape(datashape)
# Write the test data to disk
temp = np.memmap('basic_sparse_iterwrite_testdata.npy', dtype='float64', mode='w+',␣
↪shape=datashape)
temp[:] = arrdata
del temp  # Flush to disk
```

Step 1: Create a generator for our array

Note, we here use a generator for simplicity but we could equally well also implement our own `AbstractDataChunkIterator`.

```
def iter_largearray(filename, shape, dtype='float64'):
    """
    Generator reading [chunk_size, :] elements from our array in each iteration.
    """
    for i in range(shape[0]):
        # Open the file and read the next chunk
        newfp = np.memmap(filename, dtype=dtype, mode='r', shape=shape)
        curr_data = newfp[i:(i + 1), ...][0]
        del newfp  # Reopen the file in each iterator to prevent accumulation of data␣
↪in memory
        yield curr_data
    return
```

Step 2: Wrap the generator in a DataChunkIterator

```
from hdmf.data_utils import DataChunkIterator

data = DataChunkIterator(data=iter_largearray(filename='basic_sparse_iterwrite_
↪testdata.npy',
                                              shape=datashape),
                         maxshape=datashape,
                         buffer_size=10)   # Buffer 10 elements into a chunk, i.e.,␣
↪create chunks of shape (10,10)
```

Step 3: Write the data as usual

```
write_test_file(filename='basic_sparse_iterwrite_largearray.nwb',
                data=data)
```

Tip: Again, if we want to explicitly control how our data will be chunked (compressed etc.) in the HDF5 file then we need to wrap our `DataChunkIterator` using `H5DataIO`

Discussion

Let's verify that our data was written correctly

```
# Read the NWB file
from pynwb import NWBHDF5IO  # noqa: F811

io = NWBHDF5IO('basic_sparse_iterwrite_largearray.nwb', 'r')
nwbfile = io.read()
data = nwbfile.get_acquisition('synthetic_timeseries').data
# Compare all the data values of our two arrays
data_match = np.all(arrdata == data[:])   # Don't do this for very large arrays!
# Print result message
if data_match:
    print("Success: All data values match")
else:
    print("ERROR: Mismatch between data")
```

[Out]:

```
Success: All data values match
```

Example: Convert arrays stored in multiple files

In practice, data from recording devices may be distributed across many files, e.g., one file per time range or one file per recording channel. Using iterative data write provides an elegant solution to this problem as it allows us to process large arrays one-subarray-at-a-time. To make things more interesting we'll show this for the case where each recording channel (i.e, the second dimension of our `TimeSeries`) is broken up across files.

Create example data

```
import numpy as np
# Create the test data
num_channels = 10
num_steps = 100
channel_files = ['basic_sparse_iterwrite_testdata_channel_%i.npy' % i for i in␣
↪range(num_channels)]
for f in channel_files:
    temp = np.memmap(f, dtype='float64', mode='w+', shape=(num_steps,))
    temp[:] = np.arange(num_steps, dtype='float64')
    del temp  # Flush to disk
```

Step 1: Create a data chunk iterator for our multifile array

```
from hdmf.data_utils import AbstractDataChunkIterator, DataChunk  # noqa: F811


class MultiFileArrayIterator(AbstractDataChunkIterator):

    def __init__(self, channel_files, num_steps):
        """
        :param channel_files: List of files with the channels
```

(continues on next page)

(continued from previous page)

```
        :param num_steps: Number of timesteps per channel
        :return:
        """
        self.shape = (num_steps, len(channel_files))
        self.channel_files = channel_files
        self.num_steps = num_steps
        self.__curr_index = 0

    def __iter__(self):
        return self

    def __next__(self):
        """
        Return in each iteration the data from a single file
        """
        if self.__curr_index < len(channel_files):
            newfp = np.memmap(channel_files[self.__curr_index],
                              dtype='float64', mode='r', shape=(self.num_steps,))
            curr_data = newfp[:]
            i = self.__curr_index
            self.__curr_index += 1
            del newfp
            return DataChunk(data=curr_data,
                             selection=np.s_[:, i])
        else:
            raise StopIteration

    next = __next__

    def recommended_chunk_shape(self):
        return None   # Use autochunking

    def recommended_data_shape(self):
        return self.shape

    @property
    def dtype(self):
        return np.dtype('float64')

    @property
    def maxshape(self):
        return self.shape
```

Step 2: Instantiate our multi file iterator

```
data = MultiFileArrayIterator(channel_files, num_steps)
```

Step 3: Write the data as usual

```
write_test_file(filename='basic_sparse_iterwrite_multifile.nwb',
                data=data)
```

Discussion

That's it ;-)

Tip: Common mistakes that will result in errors on write:

- The size of a `DataChunk` does not match the selection.
- The selection for the `DataChunk` is not supported by h5py (e.g., unordered lists etc.)

Other common mistakes:

- Choosing inappropriate chunk sizes. This typically means bad performance with regard to I/O and/or storage cost.
- Using auto chunking without supplying a good recommended_data_shape. h5py auto chunking can only make a good guess of what the chunking should be if it (at least roughly) knows what the shape of the array will be.
- Trying to wrap a data generator using the default `DataChunkIterator` when the generator does not comply with the assumptions of the default implementation (i.e., yield individual, complete elements along the first dimension of the array one-at-a-time). Depending on the generator, this may or may not result in an error on write, but the array you are generating will probably end up at least not having the intended shape.
- The shape of the chunks returned by the `DataChunkIterator` do not match the shape of the chunks of the target HDF5 dataset. This can result in slow I/O performance, for example, when each chunk of an HDF5 dataset needs to be updated multiple times on write. For example, when using compression this would mean that HDF5 may have to read, decompress, update, compress, and write a particular chunk each time it is being updated.

Alternative Approach: User-defined dataset write

In the above cases we used the built-in capabilities of PyNWB to perform iterative data write. To gain more fine-grained control of the write process we can alternatively use PyNWB to setup the full structure of our NWB:N file and then update select datasets afterwards. This approach is useful, e.g., in context of parallel write and any time we need to optimize write patterns.

Step 1: Initially allocate the data as empty

```
from hdmf.backends.hdf5.h5_utils import H5DataIO

write_test_file(filename='basic_alternative_custom_write.nwb',
                data=H5DataIO(data=np.empty(shape=(0, 10), dtype='float'),
                              maxshape=(None, 10),  # <-- Make the time dimension␣
↪resizable
                              chunks=(131072, 2),   # <-- Use 2MB chunks
                              compression='gzip',   # <-- Enable GZip compression
                              compression_opts=4,   # <-- GZip aggression
                              shuffle=True,         # <-- Enable shuffle filter
                              fillvalue=np.nan      # <-- Use NAN as fillvalue
                              )
                )
```

Step 2: Get the dataset(s) to be updated

```
from pynwb import NWBHDF5IO    # noqa

io = NWBHDF5IO('basic_alternative_custom_write.nwb', mode='a')
nwbfile = io.read()
data = nwbfile.get_acquisition('synthetic_timeseries').data

# Let's check what the data looks like
print("Shape %s, Chunks: %s, Maxshape=%s" % (str(data.shape), str(data.chunks),␣
↪str(data.maxshape)))
```

[Out]:

```
Shape (0, 10), Chunks: (131072, 2), Maxshape=(None, 10)
```

Step 3: Implement custom write

```
data.resize((8, 10))     # <-- Allocate the space with need
data[0:3, :] = 1         # <-- Write timesteps 0,1,2
data[3:6, :] = 2         # <-- Write timesteps 3,4,5,  Note timesteps 6,7 are not␣
↪being initialized
io.close()               # <-- Close the file
```

Check the results

```
from pynwb import NWBHDF5IO    # noqa

io = NWBHDF5IO('basic_alternative_custom_write.nwb', mode='a')
nwbfile = io.read()
data = nwbfile.get_acquisition('synthetic_timeseries').data
print(data[:])
io.close()
```

[Out]:

```
[[  1.   1.   1.   1.   1.   1.   1.   1.   1.   1.]
 [  1.   1.   1.   1.   1.   1.   1.   1.   1.   1.]
 [  1.   1.   1.   1.   1.   1.   1.   1.   1.   1.]
 [  2.   2.   2.   2.   2.   2.   2.   2.   2.   2.]
 [  2.   2.   2.   2.   2.   2.   2.   2.   2.   2.]
 [  2.   2.   2.   2.   2.   2.   2.   2.   2.   2.]
 [ nan  nan  nan  nan  nan  nan  nan  nan  nan  nan]
 [ nan  nan  nan  nan  nan  nan  nan  nan  nan  nan]]
```

9.2 Domain-specific tutorials

Note: Click *here* to download the full example code

9.2.1 Intracellular electrophysiology data

The following examples will reference variables that may not be defined within the block they are used in. For clarity, we define them here:

Creating and Writing NWB files

When creating a NWB file, the first step is to create the *NWBFile*. The first argument is is a brief description of the dataset.

```
from datetime import datetime
from dateutil.tz import tzlocal
from pynwb import NWBFile
import numpy as np

nwbfile = NWBFile('my first synthetic recording', 'EXAMPLE_ID', datetime.
↪now(tzlocal()),
                  experimenter='Dr. Bilbo Baggins',
                  lab='Bag End Laboratory',
                  institution='University of Middle Earth at the Shire',
                  experiment_description='I went on an adventure with thirteen␣
↪dwarves to reclaim vast treasures.',
                  session_id='LONELYMTN')
```

Device metadata

Device metadata is represented by *Device* objects. To create a device, you can use the *Device* instance method *create_device*.

```
device = nwbfile.create_device(name='Heka ITC-1600')
```

Electrode metadata

Intracellular electrode metadata is represented by *IntracellularElectrode* objects. To create an electrode group, you can use the *NWBFile* instance method *create_ic_electrode*.

```
elec = nwbfile.create_ic_electrode(name="elec0",
                                   description='a mock intracellular electrode',
                                   device=device)
```

Stimulus data

Intracellular stimulus and response data are represented with subclasses of *PatchClampSeries*. There are two classes for representing stimulus data

- *VoltageClampStimulusSeries*
- *CurrentClampStimulusSeries*

and three classes for representing response

- *VoltageClampSeries*

- *CurrentClampSeries*
- *IZeroClampSeries*

Here, we will use *CurrentClampStimulusSeries* to store current clamp stimulus data and then add it to our NWBFile as stimulus data using the *NWBFile* method *add_stimulus*.

```
from pynwb.icephys import CurrentClampStimulusSeries

ccss = CurrentClampStimulusSeries(
    name="ccss", data=[1, 2, 3, 4, 5], starting_time=123.6, rate=10e3, electrode=elec,
↪ gain=0.02, sweep_number=0)

nwbfile.add_stimulus(ccss)
```

We now add another stimulus series but from a different sweep. TimeSeries having the same starting time belong to the same sweep.

```
from pynwb.icephys import VoltageClampStimulusSeries

vcss = VoltageClampStimulusSeries(
    name="vcss", data=[2, 3, 4, 5, 6], starting_time=234.5, rate=10e3, electrode=elec,
↪ gain=0.03, sweep_number=1)

nwbfile.add_stimulus(vcss)
```

Here, we will use *CurrentClampSeries* to store current clamp data and then add it to our NWBFile as acquired data using the *NWBFile* method *add_acquisition*.

```
from pynwb.icephys import CurrentClampSeries

ccs = CurrentClampSeries(
    name="ccs", data=[0.1, 0.2, 0.3, 0.4, 0.5],
    conversion=1e-12, resolution=np.nan, starting_time=123.6, rate=20e3,
    electrode=elec, gain=0.02, bias_current=1e-12, bridge_balance=70e6,
    capacitance_compensation=1e-12, sweep_number=0)

nwbfile.add_acquisition(ccs)
```

And voltage clamp data from the second sweep using *VoltageClampSeries*.

```
from pynwb.icephys import VoltageClampSeries

vcs = VoltageClampSeries(
    name="vcs", data=[0.1, 0.2, 0.3, 0.4, 0.5],
    conversion=1e-12, resolution=np.nan, starting_time=234.5, rate=20e3,
    electrode=elec, gain=0.02, capacitance_slow=100e-12, resistance_comp_
↪correction=70.0,
    sweep_number=1)

nwbfile.add_acquisition(vcs)
```

Once you have finished adding all of your data to the *NWBFile*, write the file with *NWBHDF5IO*.

```
from pynwb import NWBHDF5IO

with NWBHDF5IO('icephys_example.nwb', 'w') as io:
    io.write(nwbfile)
```

For more details on *NWBHDF5IO*, see the *basic tutorial*.

Reading electrophysiology data

Now that you have written some intracellular electrophysiology data, you can read it back in.

```
io = NWBHDF5IO('icephys_example.nwb', 'r')
nwbfile = io.read()
```

For details on retrieving data from an *NWBFile*, we refer the reader to the *basic tutorial*. For this tutorial, we will just get back our the *CurrentClampStimulusSeries* object we added above.

First, get the *CurrentClampStimulusSeries* we added as stimulus data.

```
ccss = nwbfile.get_stimulus('ccss')
```

Grabbing acquisition data can be done via *get_acquisition*

```
vcs = nwbfile.get_acquisition('vcs')
```

We can also get back the electrode we added.

```
elec = nwbfile.get_ic_electrode('elec0')
```

Alternatively, we can also get this electrode from the *CurrentClampStimulusSeries* we retrieved above. This is exposed via the *electrode* attribute.

```
elec = ccss.electrode
```

And the device name via *get_device*

```
device = nwbfile.get_device('Heka ITC-1600')
```

If you have data from multiple electrodes and multiple sweeps, it can be tedious and expensive to search all *PatchClampSeries* for the *TimeSeries* with a given sweep.

Fortunately you don't have to do that manually, instead you can just query the *SweepTable* which stores the mapping between the PatchClampSeries which belongs to a certain sweep number via *get_series*.

The following call will return the voltage clamp data, two timeseries consisting of acquisition and stimulus, from sweep 1.

```
series = nwbfile.sweep_table.get_series(1)
```

Note: Click *here* to download the full example code

9.2.2 Extracellular electrophysiology data

The following examples will reference variables that may not be defined within the block they are used in. For clarity, we define them here:

```
import numpy as np
```

Creating and Writing NWB files

When creating a NWB file, the first step is to create the *NWBFile*. The first argument is the name of the NWB file, and the second argument is a brief description of the dataset.

```
from datetime import datetime
from dateutil.tz import tzlocal
from pynwb import NWBFile

nwbfile = NWBFile('my first synthetic recording', 'EXAMPLE_ID', datetime.
↪now(tzlocal()),
                  experimenter='Dr. Bilbo Baggins',
                  lab='Bag End Laboratory',
                  institution='University of Middle Earth at the Shire',
                  experiment_description='I went on an adventure with thirteen␣
↪dwarves to reclaim vast treasures.',
                  session_id='LONELYMTN')
```

Electrode metadata

Electrode groups (i.e. experimentally relevant groupings of channels) are represented by *ElectrodeGroup* objects. To create an electrode group, you can use the *NWBFile* instance method *create_electrode_group*.

Before creating an *ElectrodeGroup*, you need to provide some information about the device that was used to record from the electrode. This is done by creating a *Device* object using the instance method *create_device*.

```
device = nwbfile.create_device(name='trodes_rig123')
```

Once you have created the *Device*, you can create an *ElectrodeGroup*.

```
electrode_name = 'tetrode1'
description = "an example tetrode"
location = "somewhere in the hippocampus"

electrode_group = nwbfile.create_electrode_group(electrode_name,
                                                 description=description,
                                                 location=location,
                                                 device=device)
```

After setting up electrode group metadata, you should add metadata about the individual electrodes comprising each electrode group. This is done with *add_electrode*.

The first argument to *add_electrode* is a unique identifier that the user should assign. For details on the rest of the arguments, please see the *API documentation*.

```
for idx in [1, 2, 3, 4]:
    nwbfile.add_electrode(id=idx,
                          x=1.0, y=2.0, z=3.0,
                          imp=float(-idx),
                          location='CA1', filtering='none',
                          group=electrode_group)
```

Extracellular recordings

The main classes for storing extracellular recordings are *ElectricalSeries* and *SpikeEventSeries*. *ElectricalSeries* should be used for storing raw voltage traces, local-field potential and filtered voltage traces and *SpikeEventSeries* is meant for storing spike waveforms (typically in preparation for clustering). The results of spike clustering (e.g. per-unit metadata and spike times) should be stored in the top-level *Units* table.

In addition to the *data* and *timestamps* fields inherited from *TimeSeries* class, these two classs will require metadata about the elctrodes from which *data* was generated. This is done by providing an `DynamicTableRegion`, which you can create using the *create_electrode_table_region*

The first argument to *create_electrode_table_region* a list of the indices of the electrodes you want in the region..

```
electrode_table_region = nwbfile.create_electrode_table_region([0, 2], 'the first and␣
↪third electrodes')
```

Now that we have a `DynamicTableRegion`, we can create an *ElectricalSeries* and add it to our *NWBFile*.

```
from pynwb.ecephys import ElectricalSeries

rate = 10.0
np.random.seed(1234)
data_len = 1000
ephys_data = np.random.rand(data_len * 2).reshape((data_len, 2))
ephys_timestamps = np.arange(data_len) / rate

ephys_ts = ElectricalSeries('test_ephys_data',
                            ephys_data,
                            electrode_table_region,
                            timestamps=ephys_timestamps,
                            # Alternatively, could specify starting_time and rate as␣
↪follows
                            # starting_time=ephys_timestamps[0],
                            # rate=rate,
                            resolution=0.001,
                            comments="This data was randomly generated with numpy,␣
↪using 1234 as the seed",
                            description="Random numbers generated with numpy.random.
↪rand")
nwbfile.add_acquisition(ephys_ts)
```

Associate electrodes with units

The *PyNWB Basics tutorial* demonstrates how to add data about units and specifying custom metadata about units. As mentioned *here*, there are some optional fields for units, one of these is *electrodes*. This field takes a list of indices into the electrode table for the electrodes that the unit corresponds to. For example, if two units were inferred from the first electrode (*id* = 1, index = 0), you would specify that like so:

```
nwbfile.add_unit(id=1, electrodes=[0])
nwbfile.add_unit(id=2, electrodes=[0])
```

Designating electrophysiology data

As mentioned above, *ElectricalSeries* and *SpikeEventSeries* are meant for storing specific types of extracellular recordings. In addition to these two *TimeSeries* classes, NWB provides some *data interfaces* for designating the type of data you are storing. We will briefly discuss them here, and refer the reader to *API documentation* and *PyNWB Basics tutorial* for more details on using these objects.

For storing spike data, there are two options. Which one you choose depends on what data you have available. If you need to store the complete, continuous raw voltage traces, you should store your the traces with *ElectricalSeries* objects as *acquisition* data, and use the *EventDetection* class for identifying the spike events in your raw traces. If you do not want to store the raw voltage traces and only the waveform 'snippets' surrounding spike events, you should use the *EventWaveform* class, which can store one or more *SpikeEventSeries* objects.

The results of spike sorting (or clustering) should be stored in the top-level *Units* table. Note that it is not required to store spike waveforms in order to store spike events or waveforms–if you only want to store the spike times of clustered units you can use only the Units table.

For local field potential data, there are two options. Again, which one you choose depends on what data you have available. With both options, you should store your traces with *ElectricalSeries* objects. If you are storing unfiltered local field potential data, you should store the *ElectricalSeries* objects in *LFP* data interface object(s). If you have filtered LFP data, you should store the *ElectricalSeries* objects in *FilteredEphys* data interface object(s).

Once you have finished adding all of your data to the *NWBFile*, write the file with *NWBHDF5IO*.

```
from pynwb import NWBHDF5IO

with NWBHDF5IO('ecephys_example.nwb', 'w') as io:
    io.write(nwbfile)
```

For more details on *NWBHDF5IO*, see the *basic tutorial*.

Reading electrophysiology data

Now that you have written some electrophysiology data, you can read it back in.

```
io = NWBHDF5IO('ecephys_example.nwb', 'r')
nwbfile = io.read()
```

For details on retrieving data from an *NWBFile*, we refer the reader to the *basic tutorial*. For this tutorial, we will just get back our the *ElectricalSeries* object we added above.

First, get the *ElectricalSeries*.

```
ephys_ts = nwbfile.acquisition['test_ephys_data']
```

The second dimension of the *data* attribute should be the electrodes the data was recorded with. We can get the electrodes for each column in *data* from the *electrodes* attribute. For example, information about the electrode in the second index can be retrieved like so:

```
elec2 = ephys_ts.electrodes[1]
```

Note: Click *here* to download the full example code

9.2.3 Calcium imaging data

This tutorial will demonstrate how to write calcium imaging data. The workflow demonstrated here involves three main steps:

1. Acquiring two-photon images
2. Image segmentation
3. Fluorescence and dF/F response

This tutorial assumes that transforming data between these three states is done by users–PyNWB does not provide analysis functionality.

The following examples will reference variables that may not be defined within the block they are used in. For clarity, we define them here:

```
from datetime import datetime
from dateutil.tz import tzlocal

import numpy as np

from pynwb import NWBFile
from pynwb.ophys import TwoPhotonSeries, OpticalChannel, ImageSegmentation,␣
↪Fluorescence
from pynwb.device import Device
```

Creating and Writing NWB files

When creating a NWB file, the first step is to create the *NWBFile*.

```
nwbfile = NWBFile('my first synthetic recording', 'EXAMPLE_ID', datetime.
↪now(tzlocal()),
                  experimenter='Dr. Bilbo Baggins',
                  lab='Bag End Laboratory',
                  institution='University of Middle Earth at the Shire',
                  experiment_description=('I went on an adventure with thirteen '
                                          'dwarves to reclaim vast treasures.'),
                  session_id='LONELYMTN')
```

Adding metadata about acquisition

Before you can add your data, you will need to provide some information about how that data was generated. This amounts describing the device, imaging plane and the optical channel used.

```
device = Device('imaging_device_1')
nwbfile.add_device(device)
optical_channel = OpticalChannel('my_optchan', 'description', 500.)
imaging_plane = nwbfile.create_imaging_plane('my_imgpln', optical_channel, 'a very␣
↪interesting part of the brain',
                                             device, 600., 300., 'GFP', 'my favorite␣
↪brain location',
                                             np.ones((5, 5, 3)), 4.0, 'manifold unit',
↪ 'A frame to refer to')
```

Adding two-photon image data

Now that you have your *ImagingPlane*, you can create a *TwoPhotonSeries* - the class representing two photon imaging data.

From here you have two options. The first option is to supply the image data to PyNWB, using the *data* argument. The other option is the provide a path the images. These two options have trade-offs, so it is worth spending time considering how you want to store this data[1].

```
image_series = TwoPhotonSeries(name='test_iS', dimension=[2],
                               external_file=['images.tiff'], imaging_plane=imaging_
↪plane,
                               starting_frame=[0], format='tiff', starting_time=0.0,␣
↪rate=1.0)
nwbfile.add_acquisition(image_series)
```

Storing image segmentation output

Now that the raw data is stored, you can add the image segmentation results. This is done with the *ImageSegmentation* data interface. This class has the ability to store segmentation from one or more imaging planes; hence the *PlaneSegmentation* class.

```
mod = nwbfile.create_processing_module('ophys', 'contains optical physiology␣
↪processed data')
img_seg = ImageSegmentation()
mod.add(img_seg)
ps = img_seg.create_plane_segmentation('output from segmenting my favorite imaging␣
↪plane',
                                       imaging_plane, 'my_planeseg', image_series)
```

Now that you have your *PlaneSegmentation* object, you can add the resulting ROIs. This is done using the method *add_roi*. The first argument to this method is the *pixel_mask* and the second method is the *image_mask*. The NWB schema allows for either argument to be provided.

```
w, h = 3, 3
pix_mask1 = [(0, 0, 1.1), (1, 1, 1.2), (2, 2, 1.3)]
img_mask1 = [[0.0 for x in range(w)] for y in range(h)]
img_mask1[0][0] = 1.1
img_mask1[1][1] = 1.2
img_mask1[2][2] = 1.3
ps.add_roi(pixel_mask=pix_mask1, image_mask=img_mask1)

pix_mask2 = [(0, 0, 2.1), (1, 1, 2.2)]
img_mask2 = [[0.0 for x in range(w)] for y in range(h)]
img_mask2[0][0] = 2.1
img_mask2[1][1] = 2.2
ps.add_roi(pixel_mask=pix_mask2, image_mask=img_mask2)
```

[1] If you pass in the image data directly, you will not need to worry about distributing the image files with your NWB file. However, we recognize that common image formats have tools built around them, so working with the original file formats can make one's life much simpler. NWB currently does not have the ability to read and parse native image formats. It is up to downstream users to read these file formats.

Storing fluorescence measurements

Now that ROIs are stored, you can store fluorescence (or dF/F[2]) data for these regions of interest. This type of data is stored using the *RoiResponseSeries* class. You will not need to instantiate this class directly to create objects of this type, but it is worth noting that this is the class you will work with after you read data back in.

First, create a data interface to store this data in

```
fl = Fluorescence()
mod.add(fl)
```

Because this data stores information about specific ROIs, you will need to provide a reference to the ROIs that you will be storing data for. This is done using a DynamicTableRegion, which can be created with *create_roi_table_region*.

```
rt_region = ps.create_roi_table_region('the first of two ROIs', region=[0])
```

Now that you have an DynamicTableRegion, you can create your an *RoiResponseSeries*.

```
data = [0., 1., 2., 3., 4., 5., 6., 7., 8., 9.]
timestamps = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
rrs = fl.create_roi_response_series('my_rrs', data, rt_region, unit='lumens',␣
↪timestamps=timestamps)
```

Note: You can also store more than one *RoiResponseSeries* by calling *create_roi_response_series* again.

Once we have finished adding all of our data to our *NWBFile*, make sure to write the file. Writing (and reading) is carried out using *NWBHDF5IO*.

```
from pynwb import NWBHDF5IO

with NWBHDF5IO('ophys_example.nwb', 'w') as io:
    io.write(nwbfile)
```

Reading an NWBFile

Reading is carried out using the *NWBHDF5IO* class. Unlike with writing, using *NWBHDF5IO* as a context manager is not supported and will raise an exception[3].

```
io = NWBHDF5IO('ophys_example.nwb', 'r')
nwbfile = io.read()
```

Getting your data out

After you read the NWB file, you can access individual components of your data file. To get the *ProcessingModule* back, you can index into the *processing* attribute with the name of the *ProcessingModule*.

[2] You can also store dF/F data using the *DfOverF* class.

[3] Neurodata sets can be *very* large, so individual components of the dataset are only loaded into memory when you requst them. This functionality is only possible if closing of the *NWBHDF5IO* object is handled by the user.

```
mod = nwbfile.processing['ophys']
```

Now you can retrieve the *ImageSegmentation* object by indexing into the *ProcessingModule* with the name of the *ImageSegmentation* container. In our case, this is just "ImageSegmentation", since we did not provide a name and kept the default name.

```
ps = mod['ImageSegmentation'].get_plane_segmentation()
```

Once you have the original *PlaneSegmentation* object, you can retrieve your image masks and pixel masks using standard indexing.

```
img_mask1 = ps['image_mask'][0]
pix_mask1 = ps['pixel_mask'][0]
img_mask2 = ps['image_mask'][1]
pix_mask2 = ps['pixel_mask'][1]
```

To get back the fluorescence time series data, first access the *Fluorescence* object we added (like we did above with *ImageSegmentation*) and then retrieve the *RoiResponseSeries* using *create_roi_response_series*[4].

```
rrs = mod['Fluorescence'].get_roi_response_series()


# get the data...
rrs_data = rrs.data
rrs_timestamps = rrs.timestamps
rrs_rois = rrs.rois
# and now do something cool!
```

Note: Click *here* to download the full example code

9.2.4 Allen Brain Observatory

Create an nwb file from Allen Brain Observatory data.

This example demostrates the basic functionality of several parts of the pynwb write API, centered around the optical physiology submodule (pynwb.ophys). We will use the allensdk as a read API, while leveraging the pynwb data model and write api to transform and write the data back to disk.

```
from datetime import datetime
from dateutil.tz import tzlocal

from allensdk.core.brain_observatory_cache import BrainObservatoryCache
import allensdk.brain_observatory.stimulus_info as si

from pynwb import NWBFile, NWBHDF5IO, TimeSeries
from pynwb.ophys import OpticalChannel, DfOverF, ImageSegmentation
from pynwb.image import ImageSeries, IndexSeries
from pynwb.device import Device
```

(continues on next page)

[4] If you added more than one *RoiResponseSeries*, you will need to provide the name of the *RoiResponseSeries* you want to retrieve to *create_roi_response_series*. This same behavior is exhibited with *ImageSegmentation* and various other objects through the PyNWB API.

(continued from previous page)

```
# Settings:
ophys_experiment_id = 562095852
save_file_name = 'brain_observatory.nwb'
```

Let's begin by downloading an Allen Institute Brain Observatory file. After we cache this file locally (approx. 450 MB), we can open data assets we wish to write into our NWB:N file. These include stimulus, acquisition, and processing data, as well as time "epochs" (intervals of interest)".

```
boc = BrainObservatoryCache(manifest_file='manifest.json')
dataset = boc.get_ophys_experiment_data(ophys_experiment_id)
metadata = dataset.get_metadata()
cell_specimen_ids = dataset.get_cell_specimen_ids()
timestamps, dFF = dataset.get_dff_traces()
stimulus_list = [s for s in si.SESSION_STIMULUS_MAP[metadata['session_type']]
                 if s != 'spontaneous']
running_data, _ = dataset.get_running_speed()
trial_table = dataset.get_stimulus_table('master')
trial_table['start'] = timestamps[trial_table['start'].values]
trial_table['end'] = timestamps[trial_table['end'].values]
epoch_table = dataset.get_stimulus_epoch_table()
epoch_table['start'] = timestamps[epoch_table['start'].values]
epoch_table['end'] = timestamps[epoch_table['end'].values]
```

1) First, lets create a top-level "file" container object. All the other NWB:N data components will be stored hierarchically, relative to this container. The data won't actually be written to the file system until the end of the script.

```
nwbfile = NWBFile(
    session_description='Allen Brain Observatory dataset',
    identifier=str(metadata['ophys_experiment_id']),
    session_start_time=metadata['session_start_time'],
    file_create_date=datetime.now(tzlocal())
)
```

2) Next, we add stimuli templates (one for each type of stimulus), and a data series that indexes these templates to describe what stimulus was being shown during the experiment.

```
for stimulus in stimulus_list:
    visual_stimulus_images = ImageSeries(
        name=stimulus,
        data=dataset.get_stimulus_template(stimulus),
        unit='NA',
        format='raw',
        timestamps=[0.0])
    image_index = IndexSeries(
        name=stimulus,
        data=dataset.get_stimulus_table(stimulus).frame.values,
        unit='NA',
        indexed_timeseries=visual_stimulus_images,
        timestamps=timestamps[dataset.get_stimulus_table(stimulus).start.values])
    nwbfile.add_stimulus_template(visual_stimulus_images)
    nwbfile.add_stimulus(image_index)
```

3) Besides the two-photon calcium image stack, the running speed of the animal was also recordered in this experiment. We can store this data as a TimeSeries, in the acquisition portion of the file.

```
running_speed = TimeSeries(
    name='running_speed',
    data=running_data,
    timestamps=timestamps,
    unit='cm/s')

nwbfile.add_acquisition(running_speed)
```

4) In NWB:N, an "epoch" is an interval of experiment time that can slice into a timeseries (for example running_speed, the one we just added). PyNWB uses an object-oriented approach to create links into these timeseries, so that data is not copied multiple times. Here, we extract the stimulus epochs (both fine and coarse-grained) from the Brain Observatory experiment using the allensdk.

```
for _, row in trial_table.iterrows():
    nwbfile.add_epoch(start_time=row.start,
                      stop_time=row.end,
                      timeseries=[running_speed],
                      tags='trials')

for _, row in epoch_table.iterrows():
    nwbfile.add_epoch(start_time=row.start,
                      stop_time=row.end,
                      timeseries=[running_speed],
                      tags='stimulus')
```

5) In the brain observatory, a two-photon microscope is used to acquire images of the calcium activity of neurons expressing a fluorescent protien indicator. Essentially the microscope captures picture (30 times a second) at a single depth in the visual cortex (the imaging plane). Let's use pynwb to store the metadata associated with this hardware and experimental setup:

```
optical_channel = OpticalChannel(
    name='optical_channel',
    description='2P Optical Channel',
    emission_lambda=520.,
)

device = Device(metadata['device'])
nwbfile.add_device(device)

imaging_plane = nwbfile.create_imaging_plane(
    name='imaging_plane',
    optical_channel=optical_channel,
    description='Imaging plane ',
    device=device,
    excitation_lambda=float(metadata['excitation_lambda'].split(' ')[0]),
    imaging_rate=30.,
    indicator='GCaMP6f',
    location=metadata['targeted_structure'],
    conversion=1.0,
    unit='unknown',
    reference_frame='unknown',
)
```

The Allen Insitute does not include the raw imaging signal, as this data would make the file too large. Instead, these data are preprocessed, and a dF/F flourescence signal extracted for each region-of-interest (ROI). To store the chain of computations necessary to describe this data processing pipeline, pynwb provides a "processing module" with interfaces that simplify and standarize the process of adding the steps in this provenance chain to the file:

```
ophys_module = nwbfile.create_processing_module(
    name='ophys_module',
    description='Processing module for 2P calcium responses',
)
```

6) First, we add an image segmentation interface to the module. This interface implements a pre-defined schema and API that facilitates writing segmentation masks for ROI's:

```
image_segmentation_interface = ImageSegmentation(
    name='image_segmentation')

ophys_module.add(image_segmentation_interface)

plane_segmentation = image_segmentation_interface.create_plane_segmentation(
    name='plane_segmentation',
    description='Segmentation for imaging plane',
    imaging_plane=imaging_plane)

for cell_specimen_id in cell_specimen_ids:
    curr_name = cell_specimen_id
    curr_image_mask = dataset.get_roi_mask_array([cell_specimen_id])[0]
    plane_segmentation.add_roi(id=curr_name, image_mask=curr_image_mask)
```

7) Next, we add a dF/F interface to the module. This allows us to write the dF/F timeseries data associated with each ROI.

```
dff_interface = DfOverF(name='dff_interface')
ophys_module.add(dff_interface)

rt_region = plane_segmentation.create_roi_table_region(
    description='segmented cells with cell_specimen_ids',
)

dFF_series = dff_interface.create_roi_response_series(
    name='df_over_f',
    data=dFF,
    unit='NA',
    rois=rt_region,
    timestamps=timestamps,
)
```

Now that we have created the data set, we can write the file to disk:

```
with NWBHDF5IO(save_file_name, mode='w') as io:
    io.write(nwbfile)
```

For good measure, lets read the data back in and see if everything went as planned:

```
with NWBHDF5IO(save_file_name, mode='r') as io:
    nwbfile_in = io.read()
```

CHAPTER 10

Extending NWB

The following page will discuss how to extend NWB using PyNWB.

Note: A simple example demonstrating the creation and use of a custom extension is available as part of the tutorial *Extensions*.

10.1 Creating new Extensions

The NWB specification is designed to be extended. Extension for the NWB format can be done so using classes provided in the *pynwb.spec* module. The classes *NWBGroupSpec*, *NWBDatasetSpec*, *NWBAttributeSpec*, and *NWBLinkSpec* can be used to define custom types.

10.1.1 Attribute Specifications

Specifying attributes is done with *NWBAttributeSpec*.

```
from pynwb.spec import NWBAttributeSpec

spec = NWBAttributeSpec('bar', 'a value for bar', 'float')
```

10.1.2 Dataset Specifications

Specifying datasets is done with *NWBDatasetSpec*.

```
from pynwb.spec import NWBDatasetSpec

spec = NWBDatasetSpec('A custom NWB type',
                      name='qux',
```

(continues on next page)

(continued from previous page)

```
                    attribute=[
                        NWBAttributeSpec('baz', 'a value for baz', 'str'),
                    ],
                    shape=(None, None))
```

Using datasets to specify tables

Tables can be specified using *NWBDtypeSpec*. To specify a table, provide a list of *NWBDtypeSpec* objects to the *dtype* argument.

```
from pynwb.spec import NWBDatasetSpec, NWBDtypeSpec

spec = NWBDatasetSpec('A custom NWB type',
                    name='qux',
                    attribute=[
                        NWBAttributeSpec('baz', 'a value for baz', 'str'),
                    ],
                    dtype=[
                        NWBDtypeSpec('foo', 'column for foo', 'int'),
                        NWBDtypeSpec('bar', 'a column for bar', 'float')
                    ])
```

10.1.3 Group Specifications

Specifying groups is done with the *NWBGroupSpec* class.

```
from pynwb.spec import NWBGroupSpec

spec = NWBGroupSpec('A custom NWB type',
                    name='quux',
                    attributes=[...],
                    datasets=[...],
                    groups=[...])
```

10.1.4 Neurodata Type Specifications

NWBGroupSpec and *NWBDatasetSpec* use the arguments *neurodata_type_inc* and *neurodata_type_def* for declaring new types and extending existing types. New types are specified by setting the argument *neurodata_type_def*. New types can extend an existing type by specifying the argument *neurodata_type_inc*.

Create a new type

```
from pynwb.spec import NWBGroupSpec

# A list of NWBAttributeSpec objects to specify new attributes
addl_attributes = [...]
# A list of NWBDatasetSpec objects to specify new datasets
addl_datasets = [...]
# A list of NWBDatasetSpec objects to specify new groups
addl_groups = [...]
spec = NWBGroupSpec('A custom NWB type',
                    attributes=addl_attributes,
```

(continues on next page)

(continued from previous page)

```
                    datasets=addl_datasets,
                    groups=addl_groups,
                    neurodata_type_def='MyNewNWBType')
```

Extend an existing type

```
from pynwb.spec import NWBGroupSpec

# A list of NWBAttributeSpec objects to specify additional attributes or attributes␣
↪to be overridden
addl_attributes = [...]
# A list of NWBDatasetSpec objects to specify additional datasets or datasets to be␣
↪overridden
addl_datasets = [...]
# A list of NWBGroupSpec objects to specify additional groups or groups to be␣
↪overridden
addl_groups = [...]
spec = NWBGroupSpec('An extended NWB type',
                    attributes=addl_attributes,
                    datasets=addl_datasets,
                    groups=addl_groups,
                    neurodata_type_inc='SpikeEventSeries',
                    neurodata_type_def='MyExtendedSpikeEventSeries')
```

Existing types can be instantiated by specifying *neurodata_type_inc* alone.

```
from pynwb.spec import NWBGroupSpec

# use another NWBGroupSpec object to specify that a group of type
# ElectricalSeries should be present in the new type defined below
addl_groups = [ NWBGroupSpec('An included ElectricalSeries instance',
                             neurodata_type_inc='ElectricalSeries') ]

spec = NWBGroupSpec('An extended NWB type',
                    groups=addl_groups,
                    neurodata_type_inc='SpikeEventSeries',
                    neurodata_type_def='MyExtendedSpikeEventSeries')
```

Datasets can be extended in the same manner (with regard to *neurodata_type_inc* and *neurodata_type_def*, by using the class *NWBDatasetSpec*.

10.2 Saving Extensions

Extensions are used by including them in a loaded namespace. Namespaces and extensions need to be saved to file for downstream use. The class *NWBNamespaceBuilder* can be used to create new namespace and specification files.

Note: When using *NWBNamespaceBuilder*, the core NWB namespace is automatically included

Create a new namespace with extensions

```
from pynwb.spec import NWBGroupSpec, NWBNamespaceBuilder

# create a builder for the namespace
```

(continues on next page)

(continued from previous page)

```
ns_builder = NWBNamespaceBuilder("Extension for use in my laboratory", "mylab", ...)

# create extensions
ext1 = NWBGroupSpec('A custom SpikeEventSeries interface',
                    attributes=[...],
                    datasets=[...],
                    groups=[...],
                    neurodata_type_inc='SpikeEventSeries',
                    neurodata_type_def='MyExtendedSpikeEventSeries')

ext2 = NWBGroupSpec('A custom EventDetection interface',
                    attributes=[...],
                    datasets=[...],
                    groups=[...],
                    neurodata_type_inc='EventDetection',
                    neurodata_type_def='MyExtendedEventDetection')


# add the extension
ext_source = 'mylab.specs.yaml'
ns_builder.add_spec(ext_source, ext1)
ns_builder.add_spec(ext_source, ext2)

# include an existing namespace - this will include all specifications in that␣
↪namespace
ns_builder.include_namespace('collab_ns')

# save the namespace and extensions
ns_path = 'mylab.namespace.yaml'
ns_builder.export(ns_path)
```

Tip: Using the API to generate extensions (rather than writing YAML sources directly) helps avoid errors in the specification (e.g., due to missing required keys or invalid values) and ensure compliance of the extension definition with the NWB specification language. It also helps with maintenance of extensions, e.g., if extensions have to be ported to newer versions of the specification language in the future.

10.3 Incorporating extensions

The NWB file format supports extending existing data types (See *Extending NWB* for more details on creating extensions). Extensions must be registered with PyNWB to be used for reading and writing of custom neurodata types.

The following code demonstrates how to load custom namespaces.

```
from pynwb import load_namespaces
namespace_path = 'my_namespace.yaml'
load_namespaces(namespace_path)
```

Note: This will register all namespaces defined in the file `'my_namespace.yaml'`.

10.3.1 NWBContainer : Representing custom data

To read and write custom data, corresponding *NWBContainer* classes must be associated with their respective specifications. *NWBContainer* classes are associated with their respective specification using the decorator *register_class*.

The following code demonstrates how to associate a specification with the *NWBContainer* class that represents it.

```
from pynwb import register_class
@register_class('MyExtension', 'my_namespace')
class MyExtensionContainer(NWBContainer):
    ...
```

register_class can also be used as a function.

```
from pynwb import register_class
class MyExtensionContainer(NWBContainer):
    ...
register_class('my_namespace', 'MyExtension', MyExtensionContainer)
```

If you do not have an *NWBContainer* subclass to associate with your extension specification, a dynamically created class is created by default.

To use the dynamic class, you will need to retrieve the class object using the function *get_class*. Once you have retrieved the class object, you can use it just like you would a statically defined class.

```
from pynwb import get_class
MyExtensionContainer = get_class('my_namespace', 'MyExtension')
my_ext_inst = MyExtensionContainer(...)
```

If using iPython, you can access documentation for the class's constructor using the help command.

10.3.2 ObjectMapper : Customizing the mapping between NWBContainer and the Spec

If your *NWBContainer* extension requires custom mapping of the *NWBContainer* class for reading and writing, you will need to implement and register a custom `ObjectMapper`.

`ObjectMapper` extensions are registered with the decorator *register_map*.

```
from pynwb import register_map
from form import ObjectMapper
@register_map(MyExtensionContainer)
class MyExtensionMapper(ObjectMapper)
    ...
```

register_map can also be used as a function.

```
from pynwb import register_map
from form import ObjectMapper
class MyExtensionMapper(ObjectMapper)
    ...
register_map(MyExtensionContainer, MyExtensionMapper)
```

Tip: ObjectMappers allow you to customize how objects in the spec are mapped to attributes of your NWBContainer in Python. This is useful, e.g., in cases where you want ot customize the default mapping. For example in TimeSeries

the attribute `unit` which is defined on the dataset `data` (i.e., `data.unit`) would by default be mapped to the attribute `data_unit` on *`TimeSeries`*. The ObjectMapper *`TimeSeriesMap`* then changes this mapping to map `data.unit` to the attribute `unit` on *`TimeSeries`* . ObjectMappers also allow you to customize how constructor arguments for your `NWBContainer` are constructed. E.g., in TimeSeries instead of explicit `timestamps` we may only have a `starting_time` and `rate`. In the ObjectMapper we could then construct `timestamps` from this data on data load to always have `timestamps` available for the user. For an overview of the concepts of containers, spec, builders, object mappers in PyNWB see also *Software Architecture*

10.4 Documenting Extensions

Using the same tools used to generate the documentation for the NWB-N core format one can easily generate documentation in HTML, PDF, ePub and many other format for extensions as well.

Code to generate this documentation is maintained in a separate repo: https://github.com/NeurodataWithoutBorders/nwb-docutils. To use these utilities, install the package with pip:

```
pip install nwb-docutils
```

For the purpose of this example, we assume that our current directory has the following structure.

```
- my_extension/
  - my_extension_source/
      - mylab.namespace.yaml
      - mylab.specs.yaml
      - ...
      - docs/  (Optional)
          - mylab_description.rst
          - mylab_release_notes.rst
```

In addition to Python 3.x, you will also need `sphinx` (including the `sphinx-quickstart` tool) installed. Sphinx is available here http://www.sphinx-doc.org/en/stable/install.html .

We can now create the sources of our documentation as follows:

```
python3 nwb_init_sphinx_extension_doc  \
             --project test \
             --author "Dr. Master Expert" \
             --version "1.2.3" \
             --release alpha \
             --output my_extension_docs \
             --spec_dir my_extension_source \
             --namespace_filename mylab.namespace.yaml \
             --default_namespace mylab
             --external_description my_extension_source/docs/mylab_description.rst \ ␣
↪(Optional)
             --external_release_notes my_extension_source/docs/mylab_release_notes.
↪rst \  (Optional)
```

To automatically generate the RST documentation files from the YAML (or JSON) sources of the extension simply run:

```
cd my_extension_docs
make apidoc
```

Finally, to generate the HTML version of the docs run:

```
make html
```

Tip: Additional instructions for how to use and customize the extension documentations are also available in the `Readme.md` file that `init_sphinx_extension_doc.py` automatically adds to the docs.

Tip: See `make help` for a list of available options for building the documentation in many different output formats (e.g., PDF, ePub, LaTeX, etc.).

Tip: See `python3 init_sphinx_extension_doc.py --help` for a complete list of option to customize the documentation directly during initialization.

Tip: The above example included additional description and release note docs as part of the specification. These are included in the docs via `.. include` commands so that changes in those files are automatically picked up when rebuilding to docs. Alternatively, we can also add custom documentation directly to the docs. In this case the options `--custom_description format_description.rst` and `--custom_release_notes format_release_notes.rst` of the `init_sphinx_extension_doc.py` script are useful to automatically generate the basic setup for those files so that one can easily start to add content directly without having to worry about the additional setup.

10.5 Further Reading

- **Using Extensions:** See *Extending NWB* for an example on how to use extensions during read and write.
- **Specification Language:** For a detailed overview of the specification language itself see https://schema-language.readthedocs.io/en/latest/

CHAPTER 11

Building API classes

After you have written an extension, you will need a Pythonic way to interact with the data model. To do this, you will need to write some classes that represent the data you defined in your specificiation extensions. The *pynwb.core* module has various tools to make it easier to write classes that behave like the rest of the PyNWB API.

The *pynwb.core* defines two base classes that represent the primitive structures supported by the schema. *NWBData* represents datasets and *NWBContainer* represents groups. Additionally, *pynwb.core* offers subclasses of these two classes for writing classes that come with more functionality.

11.1 register_class

When defining a class that represents a *neurodata_type* (i.e. anything that has a *neurodata_type_def*) from your extension, you can tell PyNWB which *neurodata_type* it represents using the function *register_class*. This class can be called on its own, or used as a class decorator. The first argument should be the *neurodata_type* and the second argument should be the *namespace* name.

The following example demonstrates how to register a class as the Python class reprsentation of the *neurodata_type* "MyContainer" from the *namespace* "my_ns".

```
from pynwb import register_class
from pynwb.core import NWBContainer

class MyContainer(NWBContainer):
    ...

regitser_class('MyContainer', 'my_ns', MyContainer)
```

Alternatively, you can use *register_class* as a decorator.

```
from pynwb import register_class
from pynwb.core import NWBContainer

@regitser_class('MyContainer', 'my_ns')
```

(continues on next page)

(continued from previous page)

```
class MyContainer(NWBContainer):
    ...
```

register_class is used with *NWBData* the same way it is used with *NWBContainer*.

11.2 __nwbfields__

When subclassing *NWBData* or *NWBContainer*, you might want to define some properties on your class. This can be done using the __nwbfields__ class property. This class property should be a tuple of strings that name the properties. Adding a property using this functionality will create a property than can be set *only once*.

For example, the following class definition will create the `MyContainer` class that has the properties `foo` and `bar`.

```
from pynwb import register_class
from pynwb.core import NWBContainer


class MyContainer(NWBContainer):

    __nwbfields__ = ('foo', 'bar')

    ...
```

11.3 NWBData

***NWBData* should be used to represent datasets with a *neurodata_type_def*. This section** will discuss the available *NWBData* subclasses for representing common dataset specifications.

11.3.1 NWBTable

If your specification extension contains a table definition i.e. a dataset with a compound data type, you should use the *NWBTable* class to represent this specification. Since *NWBTable* subclasses *NWBData* you can still use __nwbfields__. In addition, you can use the __columns__ class property to specify the columns of the table. __columns__ should be a list or a tuple of `docval`-like dictionaries.

The following example demonstrates how to define a table with the columns `foo` and `bar` that are of type str and int, respectively. We also register the class as the reppresentation of the *neurodata_type* "MyTable" from the *namespace* "my_ns".

```
from pynwb import register_class
from pynwb.core import NWBTable


@register_class('MyTable', 'my_ns')
class MyTable(NWBTable):

    __columns__ = [
        {'name': 'foo', 'type': str, 'doc': 'the foo column'},
        {'name': 'bar', 'type': int, 'doc': 'the bar column'},
    ]
```

(continues on next page)

(continued from previous page)

```
...
```

11.3.2 NWBTableRegion

NWBTableRegion should be used to represent datasets that store a region reference. The constructor for *NWBTableRegion*. When subclassing this class, make sure you provide a way to pass in the required arguments for the *NWBTableRegion* constructor–the *name* of the dataset, the *table* that the region applies to, and the *region* itself.

11.4 NWBContainer

NWBContainer should be used to represent groups with a *neurodata_type_def*. This section will discuss the available *NWBContainer* subclasses for representing common group specifications.

11.4.1 NWBDataInterface

The NWB schema users the neurodata type *NWBDataInterface* for specifying containers that contain data that is not considered metadata. For example, *NWBDataInterface* is a parent neurodata type to *ElectricalSeries* data, but not a parent to *ElectrodeGroup*.

There are no requirements for using *NWBDataInterface* in addition to those inherited from *NWBContainer*.

11.4.2 MultiContainerInterface

Throughout the NWB schema, there are multiple *NWBDataInterface* specifications that include one or more or zero or more of a certain neurodata type. For example, the *LFP* neurodata type contains one or more *ElectricalSeries*. If your extension follows this pattern, you can use *MultiContainerInterface* for defining the representative class.

MultiContainerInterface provides a way of automatically generating setters, getters, and properties for your class. These methods are autogenerated based on a configuration provided using the class property `__clsconf__`. `__clsconf__` should be a dict or a list of dicts. A single dict should be used if your specification contains a single neurodata type. A list of dicts should be used if your specification contains multiple neurodata types that will exist as one or more or zero or more. The contents of the dict are described in the following table.

Key	Attribute	Required?
`type`	the type of the Container	Yes
`attr`	the property name that holds the Containers	Yes
`add`	the name of the method for adding a Container	Yes
`create`	the name of the method for creating a Container	No
`get`	the name of the method for getting a Container by name	Yes

The `type` key provides a way for the setters to check for type. The property under the name given by the. `attr` key will be a *LabelledDict*. If your class uses a single dict, a `__getitem__` method will be autogenerated for indexing into this *LabelledDict*. Finally, a constructor will also be autogenerated if you do not provide one in the class definition.

The following code block demonstrates using *MultiContainerInterface* to build a class that represents the neurodata type "MyDataInterface" from the namespace "my_ns". It contains one or more containers with neurodata type "MyContainer".

```
from pynwb import register_class
from pynwb.core import MultiContainerInterface


@register_class("MyDataInterface", "my_ns")
class MyDataInterface(MultiContainerInterface):

    __clsconf__ = {
        'type': MyContainer,
        'attr': 'containers',
        'add': 'add_container',
        'create': 'create_container',
        'get': 'get_container',
    }
    ...
```

This class will have the methods `add_container`, `create_container`, and `get_container`. It will also have the property `containers`. The `add_container` method will check to make sure that either an object of type `MyContainer` or a list/dict/tuple of objects of type `MyContainer` is passed in. `create_container` will accept the exact same arguments that the `MyContainer` class constructor accepts.

CHAPTER 12

Validating NWB files

Validating NWB files is handled by a command-line tool available in *pynwb*. The validator can be invoked like so:

```
python -m pynwb.validate test.nwb
```

This will validate the file `test.nwb` against the *core* NWB specification. Validating against other specifications i.e. extensions can be done using the `-p` and `-n` flags. For example, the following command will validate against the specifications referenced in the namespace file `mylab.namespace.yaml` in addition to the core specification.

```
python -m pynwb.validate -p mylab.namespace.yaml test.nwb
```

CHAPTER 13

API Documentation

13.1 pynwb.file module

class pynwb.file.**LabMetaData**(*name*)

Bases: *pynwb.core.NWBContainer*

Parameters **name** (str) – name of metadata

namespace = 'core'

neurodata_type = 'LabMetaData'

class pynwb.file.**Subject**(*age=None*, *description=None*, *genotype=None*, *sex=None*, *species=None*, *subject_id=None*, *weight=None*, *date_of_birth=None*)

Bases: *pynwb.core.NWBContainer*

Parameters

- **age** (str) – the age of the subject
- **description** (str) – a description of the subject
- **genotype** (str) – the genotype of the subject
- **sex** (str) – the sex of the subject
- **species** (str) – the species of the subject
- **subject_id** (str) – a unique identifier for the subject
- **weight** (str) – the weight of the subject
- **date_of_birth** (datetime) – datetime of date of birth. May be supplied instead of age.

age

the age of the subject

description

a description of the subject

genotype
the genotype of the subject

sex
the sex of the subject

species
the species of the subject

subject_id
a unique identifier for the subject

weight
the weight of the subject

date_of_birth
datetime of date of birth. May be supplied instead of age.

namespace = 'core'

neurodata_type = 'Subject'

class pynwb.file.**NWBFile**(*session_description*, *identifier*, *session_start_time*, *file_create_date=None*, *timestamps_reference_time=None*, *experimenter=None*, *experiment_description=None*, *session_id=None*, *institution=None*, *keywords=None*, *notes=None*, *pharmacology=None*, *protocol=None*, *related_publications=None*, *slices=None*, *source_script=None*, *source_script_file_name=None*, *data_collection=None*, *surgery=None*, *virus=None*, *stimulus_notes=None*, *lab=None*, *acquisition=None*, *analysis=None*, *stimulus=None*, *stimulus_template=None*, *epochs=None*, *epoch_tags=set()*, *trials=None*, *invalid_times=None*, *intervals=None*, *units=None*, *processing=None*, *lab_meta_data=None*, *electrodes=None*, *electrode_groups=None*, *ic_electrodes=None*, *sweep_table=None*, *imaging_planes=None*, *ogen_sites=None*, *devices=None*, *subject=None*, *scratch=None*)

Bases: *pynwb.core.MultiContainerInterface*

A representation of an NWB file.

Parameters

- **session_description** (str) – a description of the session where this data was generated
- **identifier** (str) – a unique text identifier for the file
- **session_start_time** (datetime) – the start date and time of the recording session
- **file_create_date** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or datetime) – the date and time the file was created and subsequent modifications made
- **timestamps_reference_time** (datetime) – date and time corresponding to time zero of all timestamps; defaults to value of session_start_time
- **experimenter** (tuple or list or str) – name of person who performed experiment
- **experiment_description** (str) – general description of the experiment
- **session_id** (str) – lab-specific ID for the session
- **institution** (str) – institution(s) where experiment is performed

- **keywords** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – Terms to search over
- **notes** (str) – Notes about the experiment.
- **pharmacology** (str) – Description of drugs used, including how and when they were administered. Anesthesia(s), painkiller(s), etc., plus dosage, concentration, etc.
- **protocol** (str) – Experimental protocol, if applicable. E.g., include IACUC protocol
- **related_publications** (tuple or list or str) – Publication information.PMID, DOI, URL, etc. If multiple, concatenate together and describe which is which. such as PMID, DOI, URL, etc
- **slices** (str) – Description of slices, including information about preparation thickness, orientation, temperature and bath solution
- **source_script** (str) – Script file used to create this NWB file.
- **source_script_file_name** (str) – Name of the source_script file
- **data_collection** (str) – Notes about data collection and analysis.
- **surgery** (str) – Narrative description about surgery/surgeries, including date(s) and who performed surgery.
- **virus** (str) – Information about virus(es) used in experiments, including virus ID, source, date made, injection location, volume, etc.
- **stimulus_notes** (str) – Notes about stimuli, such as how and where presented.
- **lab** (str) – lab where experiment was performed
- **acquisition** (list or tuple) – Raw TimeSeries objects belonging to this NWBFile
- **analysis** (list or tuple) – result of analysis
- **stimulus** (list or tuple) – Stimulus TimeSeries objects belonging to this NWBFile
- **stimulus_template** (list or tuple) – Stimulus template TimeSeries objects belonging to this NWBFile
- **epochs** (*TimeIntervals*) – Epoch objects belonging to this NWBFile
- **epoch_tags** (tuple or list or set) – A sorted list of tags used across all epochs
- **trials** (*TimeIntervals*) – A table containing trial data
- **invalid_times** (*TimeIntervals*) – A table containing times to be omitted from analysis
- **intervals** (list or tuple) – any TimeIntervals tables storing time intervals
- **units** (*Units*) – A table containing unit metadata
- **processing** (list or tuple) – ProcessingModule objects belonging to this NWBFile
- **lab_meta_data** (list or tuple) – an extension that contains lab-specific meta-data
- **electrodes** (DynamicTable) – the ElectrodeTable that belongs to this NWBFile
- **electrode_groups** (Iterable) – the ElectrodeGroups that belong to this NWBFile
- **ic_electrodes** (list or tuple) – IntracellularElectrodes that belong to this NWBFile
- **sweep_table** (*SweepTable*) – the SweepTable that belong to this NWBFile

- **imaging_planes** (list or tuple) – ImagingPlanes that belong to this NWBFile
- **ogen_sites** (list or tuple) – OptogeneticStimulusSites that belong to this NWBFile
- **devices** (list or tuple) – Device objects belonging to this NWBFile
- **subject** (*Subject*) – subject metadata
- **scratch** (list or tuple) – scratch data

all_children()

objects

modules

ec_electrode_groups

ec_electrodes

add_epoch_column(*name*, *description*, *data=[]*, *table=False*, *index=False*)

Add a column to the electrode table. See add_column for more details

Parameters

- **name** (str) – the name of this VectorData
- **description** (str) – a description for this column
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – a dataset where the first dimension is a concatenation of multiple vectors
- **table** (bool or DynamicTable) – whether or not this is a table region or the table the region applies to
- **index** (bool or VectorIndex or ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – whether or not this column should be indexed

add_epoch_metadata_column(**args*, ***kwargs*)

This method is deprecated and will be removed in future versions. Please use *add_epoch_column* instead

add_epoch(*start_time*, *stop_time*, *tags=None*, *timeseries=None*)

Creates a new Epoch object. Epochs are used to track intervals in an experiment, such as exposure to a certain type of stimuli (an interval where orientation gratings are shown, or of sparse noise) or a different paradigm (a rat exploring an enclosure versus sleeping between explorations)

Parameters

- **start_time** (*float*) – Start time of epoch, in seconds
- **stop_time** (*float*) – Stop time of epoch, in seconds
- **tags** (str or list or tuple) – user-defined tags used throughout time intervals
- **timeseries** (list or tuple or *TimeSeries*) – the TimeSeries this epoch applies to

add_electrode_column(*name*, *description*, *data=[]*, *table=False*, *index=False*)

Add a column to the electrode table. See add_column for more details

Parameters

- **name** (str) – the name of this VectorData
- **description** (str) – a description for this column
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – a dataset where the first dimension is a concatenation of multiple vectors
- **table** (bool or DynamicTable) – whether or not this is a table region or the table the region applies to
- **index** (bool or VectorIndex or ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – whether or not this column should be indexed

add_electrode(*x*, *y*, *z*, *imp*, *location*, *filtering*, *group*, *id=None*)

Add a unit to the unit table. See add_row for more details.

Required fields are *x*, *y*, *z*, *imp*, *location*, *filtering*, *group* and any columns that have been added (through calls to *add_electrode_columns*).

Parameters

- **x** (*float*) – the x coordinate of the position
- **y** (*float*) – the y coordinate of the position
- **z** (*float*) – the z coordinate of the position
- **imp** (*float*) – the impedance of the electrode
- **location** (str) – the location of electrode within the subject e.g. brain region
- **filtering** (str) – description of hardware filtering
- **group** (*ElectrodeGroup*) – the ElectrodeGroup object to add to this NWBFile
- **id** (int) – a unique identifier for the electrode

create_electrode_table_region(*region*, *description*, *name='electrodes'*)

Parameters

- **region** (slice or list or tuple) – the indices of the table
- **description** (str) – a brief description of what this electrode is
- **name** (str) – the name of this container

add_unit_column(*name*, *description*, *data=[]*, *table=False*, *index=False*)

Add a column to the unit table. See add_column for more details

Parameters

- **name** (str) – the name of this VectorData
- **description** (str) – a description for this column
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – a dataset where the first dimension is a concatenation of multiple vectors

- **table** (bool or DynamicTable) – whether or not this is a table region or the table the region applies to
- **index** (bool or VectorIndex or ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – whether or not this column should be indexed

add_unit(*spike_times=None*, *obs_intervals=None*, *electrodes=None*, *electrode_group=None*, *waveform_mean=None*, *waveform_sd=None*, *id=None*)

Add a unit to the unit table. See add_row for more details.

Parameters

- **spike_times** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – the spike times for each unit
- **obs_intervals** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – the observation intervals (valid times) for each unit. All spike_times for a given unit should fall within these intervals. [[start1, end1], [start2, end2], ...]
- **electrodes** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – the electrodes that each unit came from
- **electrode_group** (*ElectrodeGroup*) – the electrode group that each unit came from
- **waveform_mean** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – the spike waveform mean for each unit. Shape is (time,) or (time, electrodes)
- **waveform_sd** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – the spike waveform standard deviation for each unit. Shape is (time,) or (time, electrodes)
- **id** (int) – the id for each unit

add_trial_column(*name*, *description*, *data=[]*, *table=False*, *index=False*)

Add a column to the trial table. See add_column for more details

Parameters

- **name** (str) – the name of this VectorData
- **description** (str) – a description for this column
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – a dataset where the first dimension is a concatenation of multiple vectors
- **table** (bool or DynamicTable) – whether or not this is a table region or the table the region applies to
- **index** (bool or VectorIndex or ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – whether or not this column should be indexed

add_trial(*start_time*, *stop_time*, *tags=None*, *timeseries=None*)

Add a trial to the trial table. See `add_interval` for more details.

Required fields are *start_time*, *stop_time*, and any columns that have been added (through calls to *add_trial_columns*).

Parameters

- **start_time** (*float*) – Start time of epoch, in seconds
- **stop_time** (*float*) – Stop time of epoch, in seconds
- **tags** (`str` or `list` or `tuple`) – user-defined tags used throughout time intervals
- **timeseries** (`list` or `tuple` or *TimeSeries*) – the TimeSeries this epoch applies to

add_invalid_times_column(*name*, *description*, *data=[]*, *table=False*, *index=False*)

Add a column to the trial table. See `add_column` for more details

Parameters

- **name** (`str`) – the name of this VectorData
- **description** (`str`) – a description for this column
- **data** (`ndarray` or `list` or `tuple` or `Dataset` or `HDMFDataset` or `AbstractDataChunkIterator` or `DataIO`) – a dataset where the first dimension is a concatenation of multiple vectors
- **table** (`bool` or DynamicTable) – whether or not this is a table region or the table the region applies to
- **index** (`bool` or `VectorIndex` or `ndarray` or `list` or `tuple` or `Dataset` or `HDMFDataset` or `AbstractDataChunkIterator`) – whether or not this column should be indexed

add_invalid_time_interval(***kwargs*)

Add a trial to the trial table. See `add_row` for more details.

Required fields are *start_time*, *stop_time*, and any columns that have been added (through calls to *add_invalid_times_columns*).

set_electrode_table(*electrode_table*)

Set the electrode table of this NWBFile to an existing ElectrodeTable

Parameters **electrode_table** (`DynamicTable`) – the ElectrodeTable for this file

add_acquisition(*nwbdata*)

Parameters **nwbdata** (*NWBDataInterface* or `DynamicTable`) – None

add_stimulus(*timeseries*)

Parameters **timeseries** (*TimeSeries*) – None

add_stimulus_template(*timeseries*)

Parameters **timeseries** (*TimeSeries*) – None

add_scratch(*data*, *name=None*, *notes=None*, *table_description=''*)

Add data to the scratch space

Parameters

- **data** (ndarray or list or tuple or DataFrame or DynamicTable or *NWBContainer* or *ScratchData*) – None
- **name** (str) – None
- **notes** (str) – None
- **table_description** (str) – None

get_scratch(*name*, *convert=True*)
Get data from the scratch space

Parameters

- **name** (str) – None
- **convert** (bool) – None

copy()
Shallow copy of an NWB file. Useful for linking across files.

acquisition
a dictionary containing the None in this NWBFile container

add_analysis(*analysis*)
Add a NWBContainer to this NWBFile

Parameters **analysis** (list or tuple or dict or *NWBContainer* or DynamicTable) – the None to add

add_device(*devices*)
Add a Device to this NWBFile

Parameters **devices** (list or tuple or dict or *Device*) – the Device to add

add_electrode_group(*electrode_groups*)
Add an ElectrodeGroup to this NWBFile

Parameters **electrode_groups** (list or tuple or dict or *ElectrodeGroup*) – the ElectrodeGroup to add

add_ic_electrode(*ic_electrodes*)
Add an IntracellularElectrode to this NWBFile

Parameters **ic_electrodes** (list or tuple or dict or *IntracellularElectrode*) – the IntracellularElectrode to add

add_imaging_plane(*imaging_planes*)
Add an ImagingPlane to this NWBFile

Parameters **imaging_planes** (list or tuple or dict or *ImagingPlane*) – the ImagingPlane to add

add_lab_meta_data(*lab_meta_data*)
Add a LabMetaData to this NWBFile

Parameters **lab_meta_data** (list or tuple or dict or *LabMetaData*) – the LabMetaData to add

add_ogen_site(*ogen_sites*)
Add an OptogeneticStimulusSite to this NWBFile

Parameters **ogen_sites** (list or tuple or dict or *OptogeneticStimulusSite*) – the OptogeneticStimulusSite to add

add_processing_module(*processing*)
Add a ProcessingModule to this NWBFile

Parameters **processing** (list or tuple or dict or *ProcessingModule*) – the ProcessingModule to add

add_time_intervals(*intervals*)
Add a TimeIntervals to this NWBFile

Parameters **intervals** (list or tuple or dict or *TimeIntervals*) – the TimeIntervals to add

analysis
a dictionary containing the None in this NWBFile container

create_device(*name*)
Create a Device and add it to this NWBFile

Parameters **name** (str) – the name of this device

Returns the Device object that was created

Return type Device

create_electrode_group(*name*, *description*, *location*, *device*)
Create an ElectrodeGroup and add it to this NWBFile

Parameters

- **name** (str) – the name of this electrode
- **description** (str) – description of this electrode group
- **location** (str) – description of location of this electrode group
- **device** (*Device*) – the device that was used to record from this electrode group

Returns the ElectrodeGroup object that was created

Return type ElectrodeGroup

create_ic_electrode(*name*, *device*, *description*, *slice=None*, *seal=None*, *location=None*, *resistance=None*, *filtering=None*, *initial_access_resistance=None*)
Create an IntracellularElectrode and add it to this NWBFile

Parameters

- **name** (str) – the name of this electrode
- **device** (*Device*) – the device that was used to record from this electrode
- **description** (str) – Recording description, description of electrode (e.g., whole-cell, sharp, etc) COMMENT: Free-form text (can be from Methods)
- **slice** (str) – Information about slice used for recording.
- **seal** (str) – Information about seal used for recording.
- **location** (str) – Area, layer, comments on estimation, stereotaxis coordinates (if in vivo, etc).
- **resistance** (str) – Electrode resistance COMMENT: unit: Ohm.
- **filtering** (str) – Electrode specific filtering.
- **initial_access_resistance** (str) – Initial access resistance.

Returns the IntracellularElectrode object that was created

Return type IntracellularElectrode

create_imaging_plane(*name*, *optical_channel*, *description*, *device*, *excitation_lambda*, *imaging_rate*, *indicator*, *location*, *manifold=None*, *conversion=1.0*, *unit='meters'*, *reference_frame=None*)

Create an ImagingPlane and add it to this NWBFile

Parameters

- **name** (str) – the name of this container
- **optical_channel** (list or *OpticalChannel*) – One of possibly many groups storing channelspecific data.
- **description** (str) – Description of this ImagingPlane.
- **device** (*Device*) – the device that was used to record
- **excitation_lambda** (*float*) – Excitation wavelength in nm.
- **imaging_rate** (*float*) – Rate images are acquired, in Hz.
- **indicator** (str) – Calcium indicator
- **location** (str) – Location of image plane.
- **manifold** (Iterable) – Physical position of each pixel. size=("height", "width", "xyz").
- **conversion** (*float*) – Multiplier to get from stored values to specified unit (e.g., 1e-3 for millimeters)
- **unit** (str) – Base unit that coordinates are stored in (e.g., Meters).
- **reference_frame** (str) – Describes position and reference frame of manifold based on position of first element in manifold.

Returns the ImagingPlane object that was created

Return type ImagingPlane

create_lab_meta_data(*name*)

Create a LabMetaData and add it to this NWBFile

Parameters **name** (str) – name of metadata

Returns the LabMetaData object that was created

Return type LabMetaData

create_ogen_site(*name*, *device*, *description*, *excitation_lambda*, *location*)

Create an OptogeneticStimulusSite and add it to this NWBFile

Parameters

- **name** (str) – The name of this stimulus site
- **device** (*Device*) – the device that was used
- **description** (str) – Description of site.
- **excitation_lambda** (*float*) – Excitation wavelength in nm.
- **location** (str) – Location of stimulation site.

Returns the OptogeneticStimulusSite object that was created

Return type OptogeneticStimulusSite

create_processing_module(*name*, *description*, *data_interfaces=None*)
Create a ProcessingModule and add it to this NWBFile

Parameters

- **name** (str) – The name of this processing module
- **description** (str) – Description of this processing module
- **data_interfaces** (list or tuple or dict) – NWBDataInterfacess that belong to this ProcessingModule

Returns the ProcessingModule object that was created

Return type ProcessingModule

create_time_intervals(*name*, *description='experimental intervals'*, *id=None*, *columns=None*, *colnames=None*)
Create a TimeIntervals and add it to this NWBFile

Parameters

- **name** (str) – name of this TimeIntervals
- **description** (str) – Description of this TimeIntervals
- **id** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or ElementIdentifiers) – the identifiers for this table
- **columns** (tuple or list) – the columns in this table
- **colnames** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – the names of the columns in this table

Returns the TimeIntervals object that was created

Return type TimeIntervals

data_collection
Notes about data collection and analysis.

devices
a dictionary containing the Device in this NWBFile container

electrode_groups
a dictionary containing the ElectrodeGroup in this NWBFile container

electrodes
the ElectrodeTable that belongs to this NWBFile

epoch_tags
A sorted list of tags used across all epochs

epochs
Epoch objects belonging to this NWBFile

experiment_description
general description of the experiment

experimenter
name of person who performed experiment

file_create_date
the date and time the file was created and subsequent modifications made

get_acquisition(*name=None*)
Get a NWBDataInterface from this NWBFile

Parameters **name** (str) – the name of the None

Returns the None with the given name

Return type (<class 'pynwb.core.NWBDataInterface'>, <class 'hdmf.common.table.DynamicTable'>)

get_analysis(*name=None*)
Get a NWBContainer from this NWBFile

Parameters **name** (str) – the name of the None

Returns the None with the given name

Return type (<class 'pynwb.core.NWBContainer'>, <class 'hdmf.common.table.DynamicTable'>)

get_device(*name=None*)
Get a Device from this NWBFile

Parameters **name** (str) – the name of the Device

Returns the Device with the given name

Return type Device

get_electrode_group(*name=None*)
Get an ElectrodeGroup from this NWBFile

Parameters **name** (str) – the name of the ElectrodeGroup

Returns the ElectrodeGroup with the given name

Return type ElectrodeGroup

get_ic_electrode(*name=None*)
Get an IntracellularElectrode from this NWBFile

Parameters **name** (str) – the name of the IntracellularElectrode

Returns the IntracellularElectrode with the given name

Return type IntracellularElectrode

get_imaging_plane(*name=None*)
Get an ImagingPlane from this NWBFile

Parameters **name** (str) – the name of the ImagingPlane

Returns the ImagingPlane with the given name

Return type ImagingPlane

get_lab_meta_data(*name=None*)
Get a LabMetaData from this NWBFile

Parameters **name** (str) – the name of the LabMetaData

Returns the LabMetaData with the given name

Return type LabMetaData

get_ogen_site(*name=None*)
Get an OptogeneticStimulusSite from this NWBFile

Parameters **name** (str) – the name of the OptogeneticStimulusSite

Returns the OptogeneticStimulusSite with the given name

Return type OptogeneticStimulusSite

get_processing_module(*name=None*)
Get a ProcessingModule from this NWBFile

Parameters **name** (str) – the name of the ProcessingModule

Returns the ProcessingModule with the given name

Return type ProcessingModule

get_stimulus(*name=None*)
Get a TimeSeries from this NWBFile

Parameters **name** (str) – the name of the TimeSeries

Returns the TimeSeries with the given name

Return type TimeSeries

get_stimulus_template(*name=None*)
Get a TimeSeries from this NWBFile

Parameters **name** (str) – the name of the TimeSeries

Returns the TimeSeries with the given name

Return type TimeSeries

get_time_intervals(*name=None*)
Get a TimeIntervals from this NWBFile

Parameters **name** (str) – the name of the TimeIntervals

Returns the TimeIntervals with the given name

Return type TimeIntervals

ic_electrodes
a dictionary containing the IntracellularElectrode in this NWBFile container

identifier
a unique text identifier for the file

imaging_planes
a dictionary containing the ImagingPlane in this NWBFile container

institution
institution(s) where experiment is performed

intervals
a dictionary containing the TimeIntervals in this NWBFile container

invalid_times
A table containing times to be omitted from analysis

keywords
Terms to search over

lab
lab where experiment was performed

lab_meta_data
a dictionary containing the LabMetaData in this NWBFile container

namespace = 'core'

neurodata_type = 'NWBFile'

notes
Notes about the experiment.

ogen_sites
a dictionary containing the OptogeneticStimulusSite in this NWBFile container

pharmacology
Description of drugs used, including how and when they were administered. Anesthesia(s), painkiller(s), etc., plus dosage, concentration, etc.

processing
a dictionary containing the ProcessingModule in this NWBFile container

protocol
Experimental protocol, if applicable. E.g., include IACUC protocol

related_publications
Publication information.PMID, DOI, URL, etc. If multiple, concatenate together and describe which is which. such as PMID, DOI, URL, etc

scratch
a dictionary containing the None in this NWBFile container

session_description
a description of the session where this data was generated

session_id
lab-specific ID for the session

session_start_time
the start date and time of the recording session

slices
Description of slices, including information about preparation thickness, orientation, temperature and bath solution

source_script
Script file used to create this NWB file.

source_script_file_name
Name of the source_script file

stimulus
a dictionary containing the TimeSeries in this NWBFile container

stimulus_notes
Notes about stimuli, such as how and where presented.

stimulus_template
a dictionary containing the TimeSeries in this NWBFile container

subject
subject metadata

surgery
Narrative description about surgery/surgeries, including date(s) and who performed surgery.

sweep_table
the SweepTable that belong to this NWBFile

timestamps_reference_time
date and time corresponding to time zero of all timestamps; defaults to value of session_start_time

trials
A table containing trial data

units
A table containing unit metadata

virus
Information about virus(es) used in experiments, including virus ID, source, date made, injection location, volume, etc.

pynwb.file.**ElectrodeTable**(*name='electrodes'*, *description='metadata about extracellular electrodes'*)

pynwb.file.**TrialTable**(*name='trials'*, *description='metadata about experimental trials'*)

pynwb.file.**InvalidTimesTable**(*name='invalid_times'*, *description='time intervals to be removed from analysis'*)

13.2 pynwb.ecephys module

class pynwb.ecephys.**ElectrodeGroup**(*name*, *description*, *location*, *device*)
Bases: *pynwb.core.NWBContainer*

Parameters

- **name** (str) – the name of this electrode
- **description** (str) – description of this electrode group
- **location** (str) – description of location of this electrode group
- **device** (*Device*) – the device that was used to record from this electrode group

description
description of this electrode group

location
description of location of this electrode group

device
the device that was used to record from this electrode group

namespace = 'core'

neurodata_type = 'ElectrodeGroup'

class pynwb.ecephys.**ElectricalSeries**(*name*, *data*, *electrodes*, *channel_conversion=None*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)
Bases: *pynwb.base.TimeSeries*

Stores acquired voltage data from extracellular recordings. The data field of an ElectricalSeries is an int or float array storing data in Volts. TimeSeries::data array structure: [num times] [num channels] (or [num_times] for single electrode).

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **electrodes** (DynamicTableRegion) – the table region corresponding to the electrodes from which this series was recorded
- **channel_conversion** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – Channel-specific conversion factor. Multiply the data in the 'data' dataset by these values along the channel axis (as indicated by axis attribute) AND by the global conversion factor in the 'conversion' attribute of 'data' to get the data values in Volts, i.e, data in Volts = data * data.conversion * channel_conversion. This approach allows for both global and per-channel data conversion factors needed to support the storage of electrical recordings as native values generated by data acquisition systems. If this dataset is not present, then there is no channel-specific conversion factor, i.e. it is 1 for all channels.
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

electrodes
the electrodes that generated this electrical series

channel_conversion
Channel-specific conversion factor. Multiply the data in the 'data' dataset by these values along the channel axis (as indicated by axis attribute) AND by the global conversion factor in the 'conversion' attribute of 'data' to get the data values in Volts, i.e, data in Volts = data * data.conversion * channel_conversion. This approach allows for both global and per-channel data conversion factors needed to support the storage of electrical recordings as native values generated by data acquisition systems. If this dataset is not present, then there is no channel-specific conversion factor, i.e. it is 1 for all channels.

namespace = 'core'

neurodata_type = 'ElectricalSeries'

class pynwb.ecephys.**SpikeEventSeries**(*name*, *data*, *timestamps*, *electrodes*, *resolution=-1.0*, *conversion=1.0*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)

Bases: *pynwb.ecephys.ElectricalSeries*

Stores "snapshots" of spike events (i.e., threshold crossings) in data. This may also be raw data, as reported by ephys hardware. If so, the TimeSeries::description field should describing how events were detected. All SpikeEventSeries should reside in a module (under EventWaveform interface) even if the spikes were reported and stored by hardware. All events span the same recording channels and store snapshots of equal duration. TimeSeries::data array structure: [num events] [num channels] [num samples] (or [num events] [num samples] for single electrode).

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – Timestamps for samples stored in data
- **electrodes** (DynamicTableRegion) – the table region corresponding to the electrodes from which this series was recorded
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

namespace = 'core'

neurodata_type = 'SpikeEventSeries'

class pynwb.ecephys.**EventDetection**(*detection_method*, *source_electricalseries*, *source_idx*, *times*, *name='EventDetection'*)

Bases: *pynwb.core.NWBDataInterface*

Detected spike events from voltage trace(s).

Parameters

- **detection_method** (str) – Description of how events were detected, such as voltage threshold, or dV/dT threshold, as well as relevant values.
- **source_electricalseries** (*ElectricalSeries*) – The source electrophysiology data
- **source_idx** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – Indices (zero-based) into source ElectricalSeries::data array corresponding to time of event. Module description should define what is meant by time of event (e.g., .25msec before action potential peak, zero-crossing time, etc). The index points to each event from the raw data
- **times** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – Timestamps of events, in Seconds
- **name** (str) – the name of this container

detection_method
Description of how events were detected, such as voltage threshold, or dV/dT threshold, as well as relevant values.

source_electricalseries
The source electrophysiology data

source_idx
Indices (zero-based) into source ElectricalSeries::data array corresponding to time of event. Module description should define what is meant by time of event (e.g., .25msec before action potential peak, zero-crossing time, etc). The index points to each event from the raw data

times
Timestamps of events, in Seconds

namespace = 'core'

neurodata_type = 'EventDetection'

class pynwb.ecephys.**EventWaveform**(*spike_event_series={}*, *name='EventWaveform'*)
Bases: *pynwb.core.MultiContainerInterface*

Spike data for spike events detected in raw data stored in this NWBFile, or events detect at acquisition

Parameters

- **spike_event_series** (list or tuple or dict or *SpikeEventSeries*) – SpikeEventSeries to store in this interface
- **name** (str) – the name of this container

__getitem__(*name=None*)
Get a SpikeEventSeries from this EventWaveform

Parameters **name** (str) – the name of the SpikeEventSeries

Returns the SpikeEventSeries with the given name

Return type SpikeEventSeries

add_spike_event_series(*spike_event_series*)
Add a SpikeEventSeries to this EventWaveform

Parameters **spike_event_series** (list or tuple or dict or *SpikeEventSeries*) – the SpikeEventSeries to add

create_spike_event_series(*name*, *data*, *timestamps*, *electrodes*, *resolution=-1.0*, *conversion=1.0*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)
Create a SpikeEventSeries and add it to this EventWaveform

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – Timestamps for samples stored in data
- **electrodes** (DynamicTableRegion) – the table region corresponding to the electrodes from which this series was recorded

- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

Returns the SpikeEventSeries object that was created

Return type SpikeEventSeries

get_spike_event_series(*name=None*)
Get a SpikeEventSeries from this EventWaveform

Parameters **name** (str) – the name of the SpikeEventSeries

Returns the SpikeEventSeries with the given name

Return type SpikeEventSeries

namespace = 'core'

neurodata_type = 'EventWaveform'

spike_event_series
a dictionary containing the SpikeEventSeries in this EventWaveform container

class pynwb.ecephys.**Clustering**(*description*, *num*, *peak_over_rms*, *times*, *name='Clustering'*)
Bases: *pynwb.core.NWBDataInterface*

DEPRECATED in favor of *Units*. Specifies cluster event times and cluster metric for maximum ratio of waveform peak to RMS on any channel in cluster.

Parameters

- **description** (str) – Description of clusters or clustering, (e.g. cluster 0 is noise, clusters curated using Klusters, etc).
- **num** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – Cluster number of each event.
- **peak_over_rms** (Iterable) – Maximum ratio of waveform peak to RMS on any channel in the cluster(provides a basic clustering metric).
- **times** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – Times of clustered events, in seconds.
- **name** (str) – the name of this container

description
Description of clusters or clustering, (e.g. cluster 0 is noise, clusters curated using Klusters, etc).

num
Cluster number of each event.

peak_over_rms
Maximum ratio of waveform peak to RMS on any channel in the cluster(provides a basic clustering metric).

times
Times of clustered events, in seconds.

namespace = 'core'

neurodata_type = 'Clustering'

class pynwb.ecephys.**ClusterWaveforms**(*clustering_interface*, *waveform_filtering*, *waveform_mean*, *waveform_sd*, *name='ClusterWaveforms'*)

Bases: *pynwb.core.NWBDataInterface*

DEPRECATED. *ClusterWaveforms* was deprecated in Oct 27, 2018 and will be removed in a future release. Please use the *Units* table to store waveform mean and standard deviation e.g. *NWBFile.units.add_unit(..., waveform_mean=..., waveform_sd=...)*

Describe cluster waveforms by mean and standard deviation for at each sample.

Parameters

- **clustering_interface** (*Clustering*) – the clustered spike data used as input for computing waveforms
- **waveform_filtering** (str) – filter applied to data before calculating mean and standard deviation
- **waveform_mean** (Iterable) – the mean waveform for each cluster
- **waveform_sd** (Iterable) – the standard deviations of waveforms for each cluster
- **name** (str) – the name of this container

clustering_interface
the clustered spike data used as input for computing waveforms

waveform_filtering
filter applied to data before calculating mean and standard deviation

waveform_mean
the mean waveform for each cluster

waveform_sd
the standard deviations of waveforms for each cluster

namespace = 'core'

neurodata_type = 'ClusterWaveforms'

class pynwb.ecephys.**LFP**(*electrical_series={}*, *name='LFP'*)

Bases: *pynwb.core.MultiContainerInterface*

LFP data from one or more channels. The electrode map in each published ElectricalSeries will identify which channels are providing LFP data. Filter properties should be noted in the ElectricalSeries description or comments field.

Parameters

- **electrical_series** (list or tuple or dict or *ElectricalSeries*) – ElectricalSeries to store in this interface
- **name** (str) – the name of this container

__getitem__(*name=None*)
Get an ElectricalSeries from this LFP

Parameters **name** (str) – the name of the ElectricalSeries

Returns the ElectricalSeries with the given name

Return type ElectricalSeries

add_electrical_series(*electrical_series*)
Add an ElectricalSeries to this LFP

Parameters **electrical_series** (list or tuple or dict or *ElectricalSeries*) – the ElectricalSeries to add

create_electrical_series(*name*, *data*, *electrodes*, *channel_conversion=None*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)
Create an ElectricalSeries and add it to this LFP

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **electrodes** (DynamicTableRegion) – the table region corresponding to the electrodes from which this series was recorded
- **channel_conversion** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – Channel-specific conversion factor. Multiply the data in the 'data' dataset by these values along the channel axis (as indicated by axis attribute) AND by the global conversion factor in the 'conversion' attribute of 'data' to get the data values in Volts, i.e, data in Volts = data * data.conversion * channel_conversion. This approach allows for both global and per-channel data conversion factors needed to support the storage of electrical recordings as native values generated by data acquisition systems. If this dataset is not present, then there is no channel-specific conversion factor, i.e. it is 1 for all channels.
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

Returns the ElectricalSeries object that was created

Return type ElectricalSeries

electrical_series
a dictionary containing the ElectricalSeries in this LFP container

get_electrical_series(*name=None*)
Get an ElectricalSeries from this LFP

Parameters **name** (str) – the name of the ElectricalSeries

Returns the ElectricalSeries with the given name

Return type ElectricalSeries

namespace = 'core'

neurodata_type = 'LFP'

class pynwb.ecephys.**FilteredEphys**(*electrical_series={}*, *name='FilteredEphys'*)
Bases: *pynwb.core.MultiContainerInterface*

Ephys data from one or more channels that has been subjected to filtering. Examples of filtered data include Theta and Gamma (LFP has its own interface). FilteredEphys modules publish an ElectricalSeries for each filtered channel or set of channels. The name of each ElectricalSeries is arbitrary but should be informative. The source of the filtered data, whether this is from analysis of another time series or as acquired by hardware, should be noted in each's TimeSeries::description field. There is no assumed 1::1 correspondence between filtered ephys signals and electrodes, as a single signal can apply to many nearby electrodes, and one electrode may have different filtered (e.g., theta and/or gamma) signals represented.

Parameters

- **electrical_series** (list or tuple or dict or *ElectricalSeries*) – ElectricalSeries to store in this interface
- **name** (str) – the name of this container

__getitem__(*name=None*)
Get an ElectricalSeries from this FilteredEphys

Parameters **name** (str) – the name of the ElectricalSeries

Returns the ElectricalSeries with the given name

Return type ElectricalSeries

add_electrical_series(*electrical_series*)
Add an ElectricalSeries to this FilteredEphys

Parameters **electrical_series** (list or tuple or dict or *ElectricalSeries*) – the ElectricalSeries to add

create_electrical_series(*name*, *data*, *electrodes*, *channel_conversion=None*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)
Create an ElectricalSeries and add it to this FilteredEphys

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **electrodes** (DynamicTableRegion) – the table region corresponding to the electrodes from which this series was recorded
- **channel_conversion** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – Channel-specific conversion factor. Multiply the data in the 'data' dataset by these values along the channel axis (as indicated by axis attribute) AND by the global conversion factor in the 'conversion' attribute of 'data' to get the data values in Volts, i.e, data in Volts =

data * data.conversion * channel_conversion. This approach allows for both global and per-channel data conversion factors needed to support the storage of electrical recordings as native values generated by data acquisition systems. If this dataset is not present, then there is no channel-specific conversion factor, i.e. it is 1 for all channels.

- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

Returns the ElectricalSeries object that was created

Return type ElectricalSeries

electrical_series
a dictionary containing the ElectricalSeries in this FilteredEphys container

get_electrical_series(*name=None*)
Get an ElectricalSeries from this FilteredEphys

Parameters **name** (str) – the name of the ElectricalSeries

Returns the ElectricalSeries with the given name

Return type ElectricalSeries

namespace = 'core'

neurodata_type = 'FilteredEphys'

class pynwb.ecephys.**FeatureExtraction**(*electrodes*, *description*, *times*, *features*, *name='FeatureExtraction'*)
Bases: *pynwb.core.NWBDataInterface*

Features, such as PC1 and PC2, that are extracted from signals stored in a SpikeEvent TimeSeries or other source.

Parameters

- **electrodes** (DynamicTableRegion) – the table region corresponding to the electrodes from which this series was recorded
- **description** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – A description for each feature extracted
- **times** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – The times of events that features correspond to

- **features** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – Features for each channel
- **name** (str) – the name of this container

electrodes
the table region corresponding to the electrodes from which this series was recorded

description
A description for each feature extracted

times
The times of events that features correspond to

features
Features for each channel

namespace = 'core'

neurodata_type = 'FeatureExtraction'

13.3 pynwb.icephys module

class pynwb.icephys.**IntracellularElectrode**(*name*, *device*, *description*, *slice=None*, *seal=None*, *location=None*, *resistance=None*, *filtering=None*, *initial_access_resistance=None*)

Bases: *pynwb.core.NWBContainer*

Parameters

- **name** (str) – the name of this electrode
- **device** (*Device*) – the device that was used to record from this electrode
- **description** (str) – Recording description, description of electrode (e.g., whole-cell, sharp, etc) COMMENT: Free-form text (can be from Methods)
- **slice** (str) – Information about slice used for recording.
- **seal** (str) – Information about seal used for recording.
- **location** (str) – Area, layer, comments on estimation, stereotaxis coordinates (if in vivo, etc).
- **resistance** (str) – Electrode resistance COMMENT: unit: Ohm.
- **filtering** (str) – Electrode specific filtering.
- **initial_access_resistance** (str) – Initial access resistance.

slice
Information about slice used for recording.

seal
Information about seal used for recording.

description
Free-form text (can be from Methods)

Type Recording description, description of electrode (e.g., whole-cell, sharp, etc) COMMENT

location
Area, layer, comments on estimation, stereotaxis coordinates (if in vivo, etc).

resistance
Ohm.

Type Electrode resistance COMMENT

Type unit

filtering
Electrode specific filtering.

initial_access_resistance
Initial access resistance.

device
the device that was used to record from this electrode

namespace = 'core'

neurodata_type = 'IntracellularElectrode'

class pynwb.icephys.**PatchClampSeries**(*name*, *data*, *unit*, *electrode*, *gain*, *stimulus_description='NA'*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*, *sweep_number=None*)

Bases: *pynwb.base.TimeSeries*

Stores stimulus or response current or voltage. Superclass definition for patch-clamp data (this class should not be instantiated directly).

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **unit** (str) – The base unit of measurement (should be SI unit)
- **electrode** (*IntracellularElectrode*) – IntracellularElectrode group that describes the electrode that was used to apply or record this data.
- **gain** (*float*) – Units: Volt/Amp (v-clamp) or Volt/Volt (c-clamp)
- **stimulus_description** (str) – the stimulus name/protocol
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset

- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value
- **sweep_number** (int or uint64) – Sweep number, allows for grouping different PatchClampSeries together via the sweep_table

electrode
IntracellularElectrode group that describes the electrode that was used to apply or record this data.

gain
Volt/Amp (v-clamp) or Volt/Volt (c-clamp)

Type Units

stimulus_description
the stimulus name/protocol

sweep_number
Sweep number, allows for grouping different PatchClampSeries together via the sweep_table

namespace = 'core'

neurodata_type = 'PatchClampSeries'

class pynwb.icephys.**CurrentClampSeries**(*name*, *data*, *electrode*, *gain*, *stimulus_description='NA'*, *bias_current=None*, *bridge_balance=None*, *capacitance_compensation=None*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*, *sweep_number=None*)

Bases: *pynwb.icephys.PatchClampSeries*

Stores voltage data recorded from intracellular current-clamp recordings. A corresponding CurrentClampStimulusSeries (stored separately as a stimulus) is used to store the current injected.

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **electrode** (*IntracellularElectrode*) – IntracellularElectrode group that describes the electrode that was used to apply or record this data.
- **gain** (*float*) – Units: Volt/Volt
- **stimulus_description** (str) – the stimulus name/protocol
- **bias_current** (*float*) – Unit: Amp
- **bridge_balance** (*float*) – Unit: Ohm
- **capacitance_compensation** (*float*) – Unit: Farad
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit

- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value
- **sweep_number** (int or uint64) – Sweep number, allows for grouping different PatchClampSeries together via the sweep_table

bias_current
Amp

Type Unit

bridge_balance
Ohm

Type Unit

capacitance_compensation
Farad

Type Unit

namespace = 'core'

neurodata_type = 'CurrentClampSeries'

class pynwb.icephys.**IZeroClampSeries**(*name*, *data*, *electrode*, *gain*, *stimulus_description='NA'*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*, *sweep_number=None*)

Bases: *pynwb.icephys.CurrentClampSeries*

Stores recorded voltage data from intracellular recordings when all current and amplifier settings are off (i.e., CurrentClampSeries fields will be zero). There is no CurrentClampStimulusSeries associated with an IZero series because the amplifier is disconnected and no stimulus can reach the cell.

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **electrode** (*IntracellularElectrode*) – IntracellularElectrode group that describes the electrode that was used to apply or record this data.
- **gain** (*float*) – Units: Volt/Volt
- **stimulus_description** (str) – the stimulus name/protocol

- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value
- **sweep_number** (int or uint64) – Sweep number, allows for grouping different PatchClampSeries together via the sweep_table

namespace = 'core'

neurodata_type = 'IZeroClampSeries'

class pynwb.icephys.**CurrentClampStimulusSeries**(*name*, *data*, *electrode*, *gain*, *stimulus_description='NA'*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*, *sweep_number=None*)

Bases: *pynwb.icephys.PatchClampSeries*

Alias to standard PatchClampSeries. Its functionality is to better tag PatchClampSeries for machine (and human) readability of the file.

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **electrode** (*IntracellularElectrode*) – IntracellularElectrode group that describes the electrode that was used to apply or record this data.
- **gain** (*float*) – Units: Volt/Amp (v-clamp) or Volt/Volt (c-clamp)
- **stimulus_description** (str) – the stimulus name/protocol
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit

- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value
- **sweep_number** (int or uint64) – Sweep number, allows for grouping different PatchClampSeries together via the sweep_table

namespace = 'core'

neurodata_type = 'CurrentClampStimulusSeries'

class pynwb.icephys.**VoltageClampSeries**(*name*, *data*, *electrode*, *gain*, *stimulus_description='NA'*, *capacitance_fast=None*, *capacitance_slow=None*, *resistance_comp_bandwidth=None*, *resistance_comp_correction=None*, *resistance_comp_prediction=None*, *whole_cell_capacitance_comp=None*, *whole_cell_series_resistance_comp=None*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*, *sweep_number=None*)

Bases: *pynwb.icephys.PatchClampSeries*

Stores current data recorded from intracellular voltage-clamp recordings. A corresponding VoltageClampStimulusSeries (stored separately as a stimulus) is used to store the voltage injected.

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **electrode** (*IntracellularElectrode*) – IntracellularElectrode group that describes the electrode that was used to apply or record this data.
- **gain** (*float*) – Units: Volt/Amp
- **stimulus_description** (str) – the stimulus name/protocol
- **capacitance_fast** (*float*) – Unit: Farad
- **capacitance_slow** (*float*) – Unit: Farad
- **resistance_comp_bandwidth** (*float*) – Unit: Hz
- **resistance_comp_correction** (*float*) – Unit: percent
- **resistance_comp_prediction** (*float*) – Unit: percent

- **whole_cell_capacitance_comp** (*float*) – Unit: Farad
- **whole_cell_series_resistance_comp** (*float*) – Unit: Ohm
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value
- **sweep_number** (int or uint64) – Sweep number, allows for grouping different PatchClampSeries together via the sweep_table

capacitance_fast
Farad

Type Unit

capacitance_slow
Farad

Type Unit

resistance_comp_bandwidth
Hz

Type Unit

resistance_comp_correction
percent

Type Unit

resistance_comp_prediction
percent

Type Unit

whole_cell_capacitance_comp
Farad

Type Unit

whole_cell_series_resistance_comp
Ohm

Type Unit

namespace = 'core'

neurodata_type = 'VoltageClampSeries'

class pynwb.icephys.**VoltageClampStimulusSeries**(*name*, *data*, *electrode*, *gain*, *stimulus_description='NA'*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*, *sweep_number=None*)

Bases: *pynwb.icephys.PatchClampSeries*

Alias to standard PatchClampSeries. Its functionality is to better tag PatchClampSeries for machine (and human) readability of the file.

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **electrode** (*IntracellularElectrode*) – IntracellularElectrode group that describes the electrode that was used to apply or record this data.
- **gain** (*float*) – Units: Volt/Amp (v-clamp) or Volt/Volt (c-clamp)
- **stimulus_description** (str) – the stimulus name/protocol
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value
- **sweep_number** (int or uint64) – Sweep number, allows for grouping different PatchClampSeries together via the sweep_table

namespace = 'core'

neurodata_type = 'VoltageClampStimulusSeries'

class pynwb.icephys.**SweepTable**(*name='sweep_table'*, *description='A sweep table groups different PatchClampSeries together.'*, *id=None*, *columns=None*, *colnames=None*)

Bases: hdmf.common.table.DynamicTable

A SweepTable allows to group PatchClampSeries together which stem from the same sweep. A sweep is a group of PatchClampSeries which have the same starting point in time.

Parameters

- **name** (str) – name of this SweepTable
- **description** (str) – Description of this SweepTable
- **id** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or ElementIdentifiers) – the identifiers for this table
- **columns** (tuple or list) – the columns in this table
- **colnames** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – the names of the columns in this table

add_entry(*pcs*)
Add the passed PatchClampSeries to the sweep table.

Parameters **pcs** (*PatchClampSeries*) – PatchClampSeries to add to the table must have a valid sweep_number

get_series(*sweep_number*)
Return a list of PatchClampSeries for the given sweep number.

namespace = 'core'

neurodata_type = 'SweepTable'

13.4 pynwb.ophys module

class pynwb.ophys.**OpticalChannel**(*name*, *description*, *emission_lambda*)
Bases: *pynwb.core.NWBContainer*

An optical channel used to record from an imaging plane.

Parameters

- **name** (str) – the name of this electrode
- **description** (str) – Any notes or comments about the channel.
- **emission_lambda** (*float*) – Emission lambda for channel.

description
Any notes or comments about the channel.

emission_lambda
Emission lambda for channel.

namespace = 'core'

neurodata_type = 'OpticalChannel'

class pynwb.ophys.**ImagingPlane**(*name*, *optical_channel*, *description*, *device*, *excitation_lambda*, *imaging_rate*, *indicator*, *location*, *manifold=None*, *conversion=1.0*, *unit='meters'*, *reference_frame=None*)
Bases: *pynwb.core.NWBContainer*

An imaging plane and its metadata.

Parameters

- **name** (str) – the name of this container

- **optical_channel** (list or *OpticalChannel*) – One of possibly many groups storing channelspecific data.
- **description** (str) – Description of this ImagingPlane.
- **device** (*Device*) – the device that was used to record
- **excitation_lambda** (*float*) – Excitation wavelength in nm.
- **imaging_rate** (*float*) – Rate images are acquired, in Hz.
- **indicator** (str) – Calcium indicator
- **location** (str) – Location of image plane.
- **manifold** (Iterable) – Physical position of each pixel. size=("height", "width", "xyz").
- **conversion** (*float*) – Multiplier to get from stored values to specified unit (e.g., 1e-3 for millimeters)
- **unit** (str) – Base unit that coordinates are stored in (e.g., Meters).
- **reference_frame** (str) – Describes position and reference frame of manifold based on position of first element in manifold.

optical_channel
One of possibly many groups storing channelspecific data.

description
Description of this ImagingPlane.

device
the device that was used to record

excitation_lambda
Excitation wavelength in nm.

imaging_rate
Rate images are acquired, in Hz.

indicator
Calcium indicator

location
Location of image plane.

manifold
Physical position of each pixel. size=("height", "width", "xyz").

conversion
Multiplier to get from stored values to specified unit (e.g., 1e-3 for millimeters)

unit
Base unit that coordinates are stored in (e.g., Meters).

reference_frame
Describes position and reference frame of manifold based on position of first element in manifold.

namespace = 'core'

neurodata_type = 'ImagingPlane'

class pynwb.ophys.**TwoPhotonSeries**(*name*, *imaging_plane*, *data=None*, *unit=None*, *format=None*, *field_of_view=None*, *pmt_gain=None*, *scan_line_rate=None*, *external_file=None*, *starting_frame=None*, *bits_per_pixel=None*, *dimension=None*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)

Bases: *pynwb.image.ImageSeries*

Image stack recorded over time from 2-photon microscope.

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **imaging_plane** (*ImagingPlane*) – Imaging plane class/pointer.
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **unit** (str) – The base unit of measurement (should be SI unit)
- **format** (str) – Format of image. Three types: 1) Image format; tiff, png, jpg, etc. 2) external 3) raw.
- **field_of_view** (Iterable or *TimeSeries*) – Width, height and depth of image, or imaged area (meters).
- **pmt_gain** (*float*) – Photomultiplier gain.
- **scan_line_rate** (*float*) – Lines imaged per second. This is also stored in /general/optophysiology but is kept here as it is useful information for analysis, and so good to be stored w/ the actual data.
- **external_file** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – Path or URL to one or more external file(s). Field only present if format=external. Either external_file or data must be specified, but not both.
- **starting_frame** (Iterable) – Each entry is the frame number in the corresponding external_file variable. This serves as an index to what frames each file contains.
- **bits_per_pixel** (int) – DEPRECATED: Number of bits per image pixel
- **dimension** (Iterable) – Number of pixels on x, y, (and z) axes.
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset

- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

field_of_view
Width, height and depth of image, or imaged area (meters).

imaging_plane
Imaging plane class/pointer.

pmt_gain
Photomultiplier gain.

scan_line_rate
Lines imaged per second. This is also stored in /general/optophysiology but is kept here as it is useful information for analysis, and so good to be stored w/ the actual data.

namespace = 'core'

neurodata_type = 'TwoPhotonSeries'

class pynwb.ophys.**CorrectedImageStack**(*corrected*, *original*, *xy_translation*, *name='CorrectedImageStack'*)
Bases: *pynwb.core.NWBDataInterface*

An image stack where all frames are shifted (registered) to a common coordinate system, to account for movement and drift between frames. Note: each frame at each point in time is assumed to be 2-D (has only x & y dimensions).

Parameters

- **corrected** (*ImageSeries*) – Image stack with frames shifted to the common coordinates.
- **original** (*ImageSeries*) – Link to image series that is being registered.
- **xy_translation** (*TimeSeries*) – Stores the x,y delta necessary to align each frame to the common coordinates, for example, to align each frame to a reference image.
- **name** (str) – The name of this CorrectedImageStack container

corrected
Image stack with frames shifted to the common coordinates.

original
Link to image series that is being registered.

xy_translation
Stores the x,y delta necessary to align each frame to the common coordinates, for example, to align each frame to a reference image.

namespace = 'core'

neurodata_type = 'CorrectedImageStack'

class pynwb.ophys.**MotionCorrection**(*corrected_images_stacks={}*, *name='MotionCorrection'*)
Bases: *pynwb.core.MultiContainerInterface*

A collection of corrected images stacks.

Parameters

- **corrected_images_stacks** (list or tuple or dict or *CorrectedImageStack*) – CorrectedImageStack to store in this interface
- **name** (str) – the name of this container

__getitem__(*name=None*)
Get a CorrectedImageStack from this MotionCorrection

Parameters **name** (str) – the name of the CorrectedImageStack

Returns the CorrectedImageStack with the given name

Return type CorrectedImageStack

add_corrected_image_stack(*corrected_images_stacks*)
Add a CorrectedImageStack to this MotionCorrection

Parameters **corrected_images_stacks** (list or tuple or dict or *CorrectedImageStack*) – the CorrectedImageStack to add

corrected_images_stacks
a dictionary containing the CorrectedImageStack in this MotionCorrection container

create_corrected_image_stack(*corrected*, *original*, *xy_translation*, *name='CorrectedImageStack'*)
Create a CorrectedImageStack and add it to this MotionCorrection

Parameters

- **corrected** (*ImageSeries*) – Image stack with frames shifted to the common coordinates.
- **original** (*ImageSeries*) – Link to image series that is being registered.
- **xy_translation** (*TimeSeries*) – Stores the x,y delta necessary to align each frame to the common coordinates, for example, to align each frame to a reference image.
- **name** (str) – The name of this CorrectedImageStack container

Returns the CorrectedImageStack object that was created

Return type CorrectedImageStack

get_corrected_image_stack(*name=None*)
Get a CorrectedImageStack from this MotionCorrection

Parameters **name** (str) – the name of the CorrectedImageStack

Returns the CorrectedImageStack with the given name

Return type CorrectedImageStack

namespace = 'core'

neurodata_type = 'MotionCorrection'

class pynwb.ophys.**PlaneSegmentation**(*description*, *imaging_plane*, *name=None*, *reference_images=None*, *id=None*, *columns=None*, *colnames=None*)
Bases: hdmf.common.table.DynamicTable

Stores pixels in an image that represent different regions of interest (ROIs) or masks. All segmentation for a given imaging plane is stored together, with storage for multiple imaging planes (masks) supported. Each ROI is stored in its own subgroup, with the ROI group containing both a 2D mask and a list of pixels that make up this mask. Segments can also be used for masking neuropil. If segmentation is allowed to change with time, a new imaging plane (or module) is required and ROI names should remain consistent between them.

Parameters

- **description** (str) – Description of image plane, recording wavelength, depth, etc.
- **imaging_plane** (*ImagingPlane*) – the ImagingPlane this ROI applies to

- **name** (str) – name of PlaneSegmentation.
- **reference_images** (*ImageSeries* or list or dict or tuple) – One or more image stacks that the masks apply to (can be oneelement stack).
- **id** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or ElementIdentifiers) – the identifiers for this table
- **columns** (tuple or list) – the columns in this table
- **colnames** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – the names of the columns in this table

imaging_plane
the ImagingPlane this ROI applies to

reference_images
One or more image stacks that the masks apply to (can be oneelement stack).

add_roi(*pixel_mask=None*, *voxel_mask=None*, *image_mask=None*, *id=None*)
Add a Region Of Interest (ROI) data to this

Parameters

- **pixel_mask** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – pixel mask for 2D ROIs: [(x1, y1, weight1), (x2, y2, weight2), . . .]
- **voxel_mask** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – voxel mask for 3D ROIs: [(x1, y1, z1, weight1), (x2, y2, z2, weight2), . . .]
- **image_mask** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – image with the same size of image where positive values mark this ROI
- **id** (int) – the ID for the ROI

static pixel_to_image(*pixel_mask*)
Converts a 2D pixel_mask of a ROI into an image_mask.

static image_to_pixel(*image_mask*)
Converts an image_mask of a ROI into a pixel_mask

create_roi_table_region(*description*, *region=slice(None*, *None*, *None)*, *name='rois'*)

Parameters

- **description** (str) – a brief description of what the region is
- **region** (slice or list or tuple) – the indices of the table
- **name** (str) – the name of the ROITableRegion

namespace = 'core'

neurodata_type = 'PlaneSegmentation'

class pynwb.ophys.**ImageSegmentation**(*plane_segmentations={}*, *name='ImageSegmentation'*)
Bases: *pynwb.core.MultiContainerInterface*

Stores pixels in an image that represent different regions of interest (ROIs) or masks. All segmentation for a given imaging plane is stored together, with storage for multiple imaging planes (masks) supported. Each ROI is stored in its own subgroup, with the ROI group containing both a 2D mask and a list of pixels that make up

this mask. Segments can also be used for masking neuropil. If segmentation is allowed to change with time, a new imaging plane (or module) is required and ROI names should remain consistent between them.

Parameters

- **plane_segmentations** (list or tuple or dict or *PlaneSegmentation*) – PlaneSegmentation to store in this interface
- **name** (str) – the name of this container

add_segmentation(*imaging_plane*, *description=None*, *name=None*)

Parameters

- **imaging_plane** (*ImagingPlane*) – the ImagingPlane this ROI applies to
- **description** (str) – Description of image plane, recording wavelength, depth, etc.
- **name** (str) – name of PlaneSegmentation.

__getitem__(*name=None*)

Get a PlaneSegmentation from this ImageSegmentation

Parameters **name** (str) – the name of the PlaneSegmentation

Returns the PlaneSegmentation with the given name

Return type PlaneSegmentation

add_plane_segmentation(*plane_segmentations*)

Add a PlaneSegmentation to this ImageSegmentation

Parameters **plane_segmentations** (list or tuple or dict or *PlaneSegmentation*) – the PlaneSegmentation to add

create_plane_segmentation(*description*, *imaging_plane*, *name=None*, *reference_images=None*, *id=None*, *columns=None*, *colnames=None*)

Create a PlaneSegmentation and add it to this ImageSegmentation

Parameters

- **description** (str) – Description of image plane, recording wavelength, depth, etc.
- **imaging_plane** (*ImagingPlane*) – the ImagingPlane this ROI applies to
- **name** (str) – name of PlaneSegmentation.
- **reference_images** (*ImageSeries* or list or dict or tuple) – One or more image stacks that the masks apply to (can be oneelement stack).
- **id** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or ElementIdentifiers) – the identifiers for this table
- **columns** (tuple or list) – the columns in this table
- **colnames** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – the names of the columns in this table

Returns the PlaneSegmentation object that was created

Return type PlaneSegmentation

get_plane_segmentation(*name=None*)

Get a PlaneSegmentation from this ImageSegmentation

Parameters **name** (str) – the name of the PlaneSegmentation

Returns the PlaneSegmentation with the given name

Return type PlaneSegmentation

namespace = 'core'

neurodata_type = 'ImageSegmentation'

plane_segmentations
a dictionary containing the PlaneSegmentation in this ImageSegmentation container

class pynwb.ophys.**RoiResponseSeries**(*name*, *data*, *rois*, *unit=None*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)

Bases: *pynwb.base.TimeSeries*

ROI responses over an imaging plane. Each column in data should correspond to the signal from one ROI.

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **rois** (DynamicTableRegion) – a table region corresponding to the ROIs that were used to generate this data
- **unit** (str) – The base unit of measurement (should be SI unit)
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

rois
a table region corresponding to the ROIs that were used to generate this data

namespace = 'core'

neurodata_type = 'RoiResponseSeries'

class pynwb.ophys.**DfOverF**(*roi_response_series={}*, *name='DfOverF'*)

Bases: *pynwb.core.MultiContainerInterface*

dF/F information about a region of interest (ROI). Storage hierarchy of dF/F should be the same as for segmentation (ie, same names for ROIs and for image planes).

Parameters

- **roi_response_series** (list or tuple or dict or *RoiResponseSeries*) – RoiResponseSeries to store in this interface
- **name** (str) – the name of this container

__getitem__(*name=None*)
Get a RoiResponseSeries from this DfOverF

Parameters **name** (str) – the name of the RoiResponseSeries

Returns the RoiResponseSeries with the given name

Return type RoiResponseSeries

add_roi_response_series(*roi_response_series*)
Add a RoiResponseSeries to this DfOverF

Parameters **roi_response_series** (list or tuple or dict or *RoiResponseSeries*) – the RoiResponseSeries to add

create_roi_response_series(*name*, *data*, *rois*, *unit=None*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)
Create a RoiResponseSeries and add it to this DfOverF

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **rois** (DynamicTableRegion) – a table region corresponding to the ROIs that were used to generate this data
- **unit** (str) – The base unit of measurement (should be SI unit)
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

Returns the RoiResponseSeries object that was created

Return type RoiResponseSeries

get_roi_response_series(*name=None*)
Get a RoiResponseSeries from this DfOverF

Parameters **name** (str) – the name of the RoiResponseSeries

Returns the RoiResponseSeries with the given name

Return type RoiResponseSeries

namespace = 'core'

neurodata_type = 'DfOverF'

roi_response_series
a dictionary containing the RoiResponseSeries in this DfOverF container

class pynwb.ophys.**Fluorescence**(*roi_response_series={}*, *name='Fluorescence'*)
Bases: *pynwb.core.MultiContainerInterface*

Fluorescence information about a region of interest (ROI). Storage hierarchy of fluorescence should be the same as for segmentation (ie, same names for ROIs and for image planes).

Parameters

- **roi_response_series** (list or tuple or dict or *RoiResponseSeries*) – RoiResponseSeries to store in this interface
- **name** (str) – the name of this container

__getitem__(*name=None*)
Get a RoiResponseSeries from this Fluorescence

Parameters **name** (str) – the name of the RoiResponseSeries

Returns the RoiResponseSeries with the given name

Return type RoiResponseSeries

add_roi_response_series(*roi_response_series*)
Add a RoiResponseSeries to this Fluorescence

Parameters **roi_response_series** (list or tuple or dict or *RoiResponseSeries*) – the RoiResponseSeries to add

create_roi_response_series(*name*, *data*, *rois*, *unit=None*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)
Create a RoiResponseSeries and add it to this Fluorescence

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **rois** (DynamicTableRegion) – a table region corresponding to the ROIs that were used to generate this data
- **unit** (str) – The base unit of measurement (should be SI unit)
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data

- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

Returns the RoiResponseSeries object that was created

Return type RoiResponseSeries

get_roi_response_series(*name=None*)
Get a RoiResponseSeries from this Fluorescence

Parameters **name** (str) – the name of the RoiResponseSeries

Returns the RoiResponseSeries with the given name

Return type RoiResponseSeries

namespace = 'core'

neurodata_type = 'Fluorescence'

roi_response_series
a dictionary containing the RoiResponseSeries in this Fluorescence container

13.5 pynwb.ogen module

class pynwb.ogen.**OptogeneticStimulusSite**(*name*, *device*, *description*, *excitation_lambda*, *location*)

Bases: *pynwb.core.NWBContainer*

Parameters

- **name** (str) – The name of this stimulus site
- **device** (*Device*) – the device that was used
- **description** (str) – Description of site.
- **excitation_lambda** (*float*) – Excitation wavelength in nm.
- **location** (str) – Location of stimulation site.

device
the device that was used

description
Description of site.

excitation_lambda
Excitation wavelength in nm.

location
Location of stimulation site.

namespace = 'core'

neurodata_type = 'OptogeneticStimulusSite'

class pynwb.ogen.**OptogeneticSeries**(*name*, *data*, *site*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)

Bases: *pynwb.base.TimeSeries*

Optogenetic stimulus. The data field is in unit of watts.

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **site** (*OptogeneticStimulusSite*) – The site to which this stimulus was applied.
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

site
The site to which this stimulus was applied.

namespace = 'core'

neurodata_type = 'OptogeneticSeries'

13.6 pynwb.retinotopy module

class pynwb.retinotopy.**AImage**(*name*, *data*, *bits_per_pixel*, *dimension*, *format*, *field_of_view*, *focal_depth*)

Bases: *pynwb.core.NWBContainer*

Parameters

- **name** (str) – the name of this axis map

- **data** (Iterable) – Data field.
- **bits_per_pixel** (int) – Number of bits used to represent each value. This is necessary to determine maximum (white) pixel value.
- **dimension** (Iterable) – Number of rows and columns in the image.
- **format** (Iterable) – Format of image. Right now only "raw" supported.
- **field_of_view** (Iterable) – Size of viewing area, in meters.
- **focal_depth** (*float*) – Focal depth offset, in meters.

data
Data field.

bits_per_pixel
Number of bits used to represent each value. This is necessary to determine maximum (white) pixel value.

dimension
Number of rows and columns in the image.

field_of_view
Size of viewing area, in meters.

focal_depth
Focal depth offset, in meters.

format
Format of image. Right now only "raw" supported.

class pynwb.retinotopy.**AxisMap**(*name*, *data*, *field_of_view*, *unit*, *dimension*)
Bases: *pynwb.core.NWBContainer*

Parameters

- **name** (str) – the name of this axis map
- **data** (Iterable) – data field.
- **field_of_view** (Iterable) – Size of viewing area, in meters.
- **unit** (str) – Unit that axis data is stored in (e.g., degrees)
- **dimension** (Iterable) – Number of rows and columns in the image

data
data field.

field_of_view
Size of viewing area, in meters.

unit
Unit that axis data is stored in (e.g., degrees)

dimension
Number of rows and columns in the image

class pynwb.retinotopy.**ImagingRetinotopy**(*sign_map*, *axis_1_phase_map*, *axis_1_power_map*, *axis_2_phase_map*, *axis_2_power_map*, *axis_descriptions*, *focal_depth_image*, *vasculature_image*, *name='ImagingRetinotopy'*)
Bases: *pynwb.core.NWBDataInterface*

Intrinsic signal optical imaging or widefield imaging for measuring retinotopy. Stores orthogonal maps (e.g., altitude/azimuth; radius/theta) of responses to specific stimuli and a combined polarity map from which to identify visual areas. Note: for data consistency, all images and arrays are stored in the format [row][column] and [row, col], which equates to [y][x]. Field of view and dimension arrays may appear backward (i.e., y before x).

Parameters

- **sign_map** (*AxisMap*) – Sine of the angle between the direction of the gradient in axis_1 and axis_2.
- **axis_1_phase_map** (*AxisMap*) – Phase response to stimulus on the first measured axis.
- **axis_1_power_map** (*AxisMap*) – Power response on the first measured axis. Response is scaled so 0.0 is no power in the response and 1.0 is maximum relative power.
- **axis_2_phase_map** (*AxisMap*) – Phase response to stimulus on the second measured axis.
- **axis_2_power_map** (*AxisMap*) – Power response on the second measured axis. Response is scaled so 0.0 is no power in the response and 1.0 is maximum relative power.
- **axis_descriptions** (Iterable) – Two-element array describing the contents of the two response axis fields. Description should be something like ["altitude", "azimuth"] or ["radius", "theta"].
- **focal_depth_image** (*AImage*) – Gray-scale image taken with same settings/parameters (e.g., focal depth, wavelength) as data collection. Array format: [rows][columns].
- **vasculature_image** (*AImage*) – Gray-scale anatomical image of cortical surface. Array structure: [rows][columns].
- **name** (str) – the name of this container

axis_1_phase_map
Phase response to stimulus on the first measured axis.

axis_1_power_map
Power response on the first measured axis. Response is scaled so 0.0 is no power in the response and 1.0 is maximum relative power.

axis_2_phase_map
Phase response to stimulus on the second measured axis.

axis_2_power_map
Power response on the second measured axis. Response is scaled so 0.0 is no power in the response and 1.0 is maximum relative power.

axis_descriptions
Two-element array describing the contents of the two response axis fields. Description should be something like ["altitude", "azimuth"] or ["radius", "theta"].

focal_depth_image
[rows][columns].

Type Gray-scale image taken with same settings/parameters (e.g., focal depth, wavelength) as data collection. Array format

sign_map
Sine of the angle between the direction of the gradient in axis_1 and axis_2.

vasculature_image
[rows][columns].

Type Gray-scale anatomical image of cortical surface. Array structure

namespace = 'core'

neurodata_type = 'ImagingRetinotopy'

13.7 pynwb.image module

class pynwb.image.**ImageSeries**(*name*, *data=None*, *unit=None*, *format=None*, *external_file=None*, *starting_frame=None*, *bits_per_pixel=None*, *dimension=None*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)

Bases: *pynwb.base.TimeSeries*

General image data that is common between acquisition and stimulus time series. The image data can be stored in the HDF5 file or it will be stored as an external image file.

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **unit** (str) – The base unit of measurement (should be SI unit)
- **format** (str) – Format of image. Three types: 1) Image format; tiff, png, jpg, etc. 2) external 3) raw.
- **external_file** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – Path or URL to one or more external file(s). Field only present if format=external. Either external_file or data must be specified, but not both.
- **starting_frame** (Iterable) – Each entry is the frame number in the corresponding external_file variable. This serves as an index to what frames each file contains.
- **bits_per_pixel** (int) – DEPRECATED: Number of bits per image pixel
- **dimension** (Iterable) – Number of pixels on x, y, (and z) axes.
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset

- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

dimension
Number of pixels on x, y, (and z) axes.

external_file
Path or URL to one or more external file(s). Field only present if format=external. Either external_file or data must be specified, but not both.

starting_frame
Each entry is the frame number in the corresponding external_file variable. This serves as an index to what frames each file contains.

format

1) Image format; tiff, png, jpg, etc. 2) external 3) raw.

Type Format of image. Three types

bits_per_pixel

namespace = 'core'

neurodata_type = 'ImageSeries'

class pynwb.image.**IndexSeries**(*name*, *data*, *indexed_timeseries*, *unit=None*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)

Bases: *pynwb.base.TimeSeries*

Stores indices to image frames stored in an ImageSeries. The purpose of the ImageIndexSeries is to allow a static image stack to be stored somewhere, and the images in the stack to be referenced out-of-order. This can be for the display of individual images, or of movie segments (as a movie is simply a series of images). The data field stores the index of the frame in the referenced ImageSeries, and the timestamps array indicates when that image was displayed.

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **indexed_timeseries** (*TimeSeries*) – HDF5 link to TimeSeries containing images that are indexed.
- **unit** (str) – The base unit of measurement (should be SI unit)
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data

- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

indexed_timeseries
HDF5 link to TimeSeries containing images that are indexed.

namespace = 'core'

neurodata_type = 'IndexSeries'

class pynwb.image.**ImageMaskSeries**(*name*, *data*, *masked_imageseries*, *unit=None*, *format=None*, *external_file=None*, *starting_frame=None*, *bits_per_pixel=None*, *dimension=None*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)

Bases: *pynwb.image.ImageSeries*

An alpha mask that is applied to a presented visual stimulus. The data[] array contains an array of mask values that are applied to the displayed image. Mask values are stored as RGBA. Mask can vary with time. The timestamps array indicates the starting time of a mask, and that mask pattern continues until it's explicitly changed.

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **masked_imageseries** (*ImageSeries*) – Link to ImageSeries that mask is applied to.
- **unit** (str) – The base unit of measurement (should be SI unit)
- **format** (str) – Format of image. Three types: 1) Image format; tiff, png, jpg, etc. 2) external 3) raw.
- **external_file** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – Path or URL to one or more external file(s). Field only present if format=external. Either external_file or data must be specified, but not both.
- **starting_frame** (Iterable) – Each entry is the frame number in the corresponding external_file variable. This serves as an index to what frames each file contains.
- **bits_per_pixel** (int) – DEPRECATED: Number of bits per image pixel
- **dimension** (Iterable) – Number of pixels on x, y, (and z) axes.
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit

- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

masked_imageseries
Link to ImageSeries that mask is applied to.

namespace = 'core'

neurodata_type = 'ImageMaskSeries'

class pynwb.image.**OpticalSeries**(*name*, *distance*, *field_of_view*, *orientation*, *data=None*, *unit=None*, *format=None*, *external_file=None*, *starting_frame=None*, *bits_per_pixel=None*, *dimension=None*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)

Bases: *pynwb.image.ImageSeries*

Image data that is presented or recorded. A stimulus template movie will be stored only as an image. When the image is presented as stimulus, additional data is required, such as field of view (eg, how much of the visual field the image covers, or how what is the area of the target being imaged). If the OpticalSeries represents acquired imaging data, orientation is also important.

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **distance** (*float*) – Distance from camera/monitor to target/eye.
- **field_of_view** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Width, height and depth of image, or imaged area (meters).
- **orientation** (str) – Description of image relative to some reference frame (e.g., which way is up). Must also specify frame of reference.
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **unit** (str) – The base unit of measurement (should be SI unit)
- **format** (str) – Format of image. Three types: 1) Image format; tiff, png, jpg, etc. 2) external 3) raw.
- **external_file** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – Path or URL to one or more external file(s). Field only present if format=external. Either external_file or data must be specified, but not both.

- **starting_frame** (Iterable) – Each entry is the frame number in the corresponding external_file variable. This serves as an index to what frames each file contains.
- **bits_per_pixel** (int) – DEPRECATED: Number of bits per image pixel
- **dimension** (Iterable) – Number of pixels on x, y, (and z) axes.
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

distance
Distance from camera/monitor to target/eye.

field_of_view
Width, height and depth of image, or imaged area (meters).

orientation
Description of image relative to some reference frame (e.g., which way is up). Must also specify frame of reference.

namespace = 'core'

neurodata_type = 'OpticalSeries'

class pynwb.image.**GrayscaleImage**(*name*, *data*, *resolution=None*, *description=None*)
Bases: *pynwb.base.Image*

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – data of image
- **resolution** (*float*) – pixels / cm
- **description** (str) – description of image

namespace = 'core'

neurodata_type = 'GrayscaleImage'

class pynwb.image.**RGBImage**(*name*, *data*, *resolution=None*, *description=None*)
Bases: *pynwb.base.Image*

Parameters

- **name** (str) – The name of this TimeSeries dataset

- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – data of image
- **resolution** (*float*) – pixels / cm
- **description** (str) – description of image

namespace = 'core'

neurodata_type = 'RGBImage'

class pynwb.image.**RGBAImage**(*name*, *data*, *resolution=None*, *description=None*)

Bases: *pynwb.base.Image*

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – data of image
- **resolution** (*float*) – pixels / cm
- **description** (str) – description of image

namespace = 'core'

neurodata_type = 'RGBAImage'

13.8 pynwb.behavior module

class pynwb.behavior.**SpatialSeries**(*name*, *data*, *reference_frame*, *conversion=1.0*, *resolution=-1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)

Bases: *pynwb.base.TimeSeries*

Direction, e.g., of gaze or travel, or position. The TimeSeries::data field is a 2D array storing position or direction relative to some reference frame. Array structure: [num measurements] [num dimensions]. Each SpatialSeries has a text dataset reference_frame that indicates the zero-position, or the zero-axes for direction. For example, if representing gaze direction, "straight-ahead" might be a specific pixel on the monitor, or some other point in space. For position data, the 0,0 point might be the top-left corner of an enclosure, as viewed from the tracking camera. The unit of data will indicate how to interpret SpatialSeries values.

Create a SpatialSeries TimeSeries dataset

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **reference_frame** (str) – description defining what the zero-position is
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data

- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

reference_frame
description defining what the zero-position is

namespace = 'core'

neurodata_type = 'SpatialSeries'

class pynwb.behavior.**BehavioralEpochs**(*interval_series={}*, *name='BehavioralEpochs'*)
Bases: *pynwb.core.MultiContainerInterface*

TimeSeries for storing behavoioral epochs. The objective of this and the other two Behavioral interfaces (e.g. BehavioralEvents and BehavioralTimeSeries) is to provide generic hooks for software tools/scripts. This allows a tool/script to take the output one specific interface (e.g., UnitTimes) and plot that data relative to another data modality (e.g., behavioral events) without having to define all possible modalities in advance. Declaring one of these interfaces means that one or more TimeSeries of the specified type is published. These TimeSeries should reside in a group having the same name as the interface. For example, if a BehavioralTimeSeries interface is declared, the module will have one or more TimeSeries defined in the module sub-group "BehavioralTimeSeries". BehavioralEpochs should use IntervalSeries. BehavioralEvents is used for irregular events. BehavioralTimeSeries is for continuous data.

Parameters

- **interval_series** (list or tuple or dict or *IntervalSeries*) – IntervalSeries to store in this interface
- **name** (str) – the name of this container

__getitem__(*name=None*)
Get an IntervalSeries from this BehavioralEpochs

Parameters **name** (str) – the name of the IntervalSeries

Returns the IntervalSeries with the given name

Return type IntervalSeries

add_interval_series(*interval_series*)
Add an IntervalSeries to this BehavioralEpochs

Parameters **interval_series** (list or tuple or dict or *IntervalSeries*) – the IntervalSeries to add

create_interval_series(*name*, *data=[]*, *timestamps=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)
Create an IntervalSeries and add it to this BehavioralEpochs

Parameters

- **name** (str) – The name of this TimeSeries dataset

- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – >0 if interval started, <0 if interval ended.
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

Returns the IntervalSeries object that was created

Return type IntervalSeries

get_interval_series(*name=None*)
Get an IntervalSeries from this BehavioralEpochs

Parameters **name** (str) – the name of the IntervalSeries

Returns the IntervalSeries with the given name

Return type IntervalSeries

interval_series
a dictionary containing the IntervalSeries in this BehavioralEpochs container

namespace = 'core'

neurodata_type = 'BehavioralEpochs'

class pynwb.behavior.**BehavioralEvents**(*time_series={}*, *name='BehavioralEvents'*)
Bases: *pynwb.core.MultiContainerInterface*

TimeSeries for storing behavioral events. See description of BehavioralEpochs for more details.

Parameters

- **time_series** (list or tuple or dict or *TimeSeries*) – TimeSeries to store in this interface
- **name** (str) – the name of this container

__getitem__(*name=None*)
Get a TimeSeries from this BehavioralEvents

Parameters **name** (str) – the name of the TimeSeries

Returns the TimeSeries with the given name

Return type TimeSeries

add_timeseries(*time_series*)
Add a TimeSeries to this BehavioralEvents

Parameters **time_series** (list or tuple or dict or *TimeSeries*) – the TimeSeries to add

create_timeseries(*name*, *data=None*, *unit=None*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)
Create a TimeSeries and add it to this BehavioralEvents

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **unit** (str) – The base unit of measurement (should be SI unit)
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

Returns the TimeSeries object that was created

Return type TimeSeries

get_timeseries(*name=None*)
Get a TimeSeries from this BehavioralEvents

Parameters **name** (str) – the name of the TimeSeries

Returns the TimeSeries with the given name

Return type TimeSeries

namespace = 'core'

neurodata_type = 'BehavioralEvents'

time_series
a dictionary containing the TimeSeries in this BehavioralEvents container

class pynwb.behavior.**BehavioralTimeSeries**(*time_series={}*, *name='BehavioralTimeSeries'*)
Bases: *pynwb.core.MultiContainerInterface*

TimeSeries for storing Behavoioral time series data. See description of BehavioralEpochs for more details.

Parameters

- **time_series** (list or tuple or dict or *TimeSeries*) – TimeSeries to store in this interface
- **name** (str) – the name of this container

__getitem__(*name=None*)
Get a TimeSeries from this BehavioralTimeSeries

Parameters **name** (str) – the name of the TimeSeries

Returns the TimeSeries with the given name

Return type TimeSeries

add_timeseries(*time_series*)
Add a TimeSeries to this BehavioralTimeSeries

Parameters **time_series** (list or tuple or dict or *TimeSeries*) – the TimeSeries to add

create_timeseries(*name*, *data=None*, *unit=None*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)
Create a TimeSeries and add it to this BehavioralTimeSeries

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **unit** (str) – The base unit of measurement (should be SI unit)
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

Returns the TimeSeries object that was created

Return type TimeSeries

get_timeseries(*name=None*)
Get a TimeSeries from this BehavioralTimeSeries

Parameters **name** (str) – the name of the TimeSeries

Returns the TimeSeries with the given name

Return type TimeSeries

namespace = 'core'

neurodata_type = 'BehavioralTimeSeries'

time_series
a dictionary containing the TimeSeries in this BehavioralTimeSeries container

class pynwb.behavior.**PupilTracking**(*time_series={}*, *name='PupilTracking'*)

Bases: *pynwb.core.MultiContainerInterface*

Eye-tracking data, representing pupil size.

Parameters

- **time_series** (list or tuple or dict or *TimeSeries*) – TimeSeries to store in this interface
- **name** (str) – the name of this container

__getitem__(*name=None*)

Get a TimeSeries from this PupilTracking

Parameters **name** (str) – the name of the TimeSeries

Returns the TimeSeries with the given name

Return type TimeSeries

add_timeseries(*time_series*)

Add a TimeSeries to this PupilTracking

Parameters **time_series** (list or tuple or dict or *TimeSeries*) – the TimeSeries to add

create_timeseries(*name*, *data=None*, *unit=None*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)

Create a TimeSeries and add it to this PupilTracking

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **unit** (str) – The base unit of measurement (should be SI unit)
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

Returns the TimeSeries object that was created

Return type TimeSeries

get_timeseries(*name=None*)
Get a TimeSeries from this PupilTracking

Parameters **name** (str) – the name of the TimeSeries

Returns the TimeSeries with the given name

Return type TimeSeries

namespace = 'core'

neurodata_type = 'PupilTracking'

time_series
a dictionary containing the TimeSeries in this PupilTracking container

class pynwb.behavior.**EyeTracking**(*spatial_series={}*, *name='EyeTracking'*)
Bases: *pynwb.core.MultiContainerInterface*

Eye-tracking data, representing direction of gaze.

Parameters

- **spatial_series** (list or tuple or dict or *SpatialSeries*) – SpatialSeries to store in this interface
- **name** (str) – the name of this container

__getitem__(*name=None*)
Get a SpatialSeries from this EyeTracking

Parameters **name** (str) – the name of the SpatialSeries

Returns the SpatialSeries with the given name

Return type SpatialSeries

add_spatial_series(*spatial_series*)
Add a SpatialSeries to this EyeTracking

Parameters **spatial_series** (list or tuple or dict or *SpatialSeries*) – the SpatialSeries to add

create_spatial_series(*name*, *data*, *reference_frame*, *conversion=1.0*, *resolution=-1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)
Create a SpatialSeries and add it to this EyeTracking

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **reference_frame** (str) – description defining what the zero-position is
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data

- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

Returns the SpatialSeries object that was created

Return type SpatialSeries

get_spatial_series(*name=None*)
Get a SpatialSeries from this EyeTracking

Parameters **name** (str) – the name of the SpatialSeries

Returns the SpatialSeries with the given name

Return type SpatialSeries

namespace = 'core'

neurodata_type = 'EyeTracking'

spatial_series
a dictionary containing the SpatialSeries in this EyeTracking container

class pynwb.behavior.**CompassDirection**(*spatial_series={}*, *name='CompassDirection'*)
Bases: *pynwb.core.MultiContainerInterface*

With a CompassDirection interface, a module publishes a SpatialSeries object representing a floating point value for theta. The SpatialSeries::reference_frame field should indicate what direction corresponds to 0 and which is the direction of rotation (this should be clockwise). The si_unit for the SpatialSeries should be radians or degrees.

Parameters

- **spatial_series** (list or tuple or dict or *SpatialSeries*) – SpatialSeries to store in this interface
- **name** (str) – the name of this container

__getitem__(*name=None*)
Get a SpatialSeries from this CompassDirection

Parameters **name** (str) – the name of the SpatialSeries

Returns the SpatialSeries with the given name

Return type SpatialSeries

add_spatial_series(*spatial_series*)
Add a SpatialSeries to this CompassDirection

Parameters **spatial_series** (list or tuple or dict or *SpatialSeries*) – the SpatialSeries to add

create_spatial_series(*name*, *data*, *reference_frame*, *conversion=1.0*, *resolution=-1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)

Create a SpatialSeries and add it to this CompassDirection

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **reference_frame** (str) – description defining what the zero-position is
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

Returns the SpatialSeries object that was created

Return type SpatialSeries

get_spatial_series(*name=None*)

Get a SpatialSeries from this CompassDirection

Parameters **name** (str) – the name of the SpatialSeries

Returns the SpatialSeries with the given name

Return type SpatialSeries

namespace = 'core'

neurodata_type = 'CompassDirection'

spatial_series

a dictionary containing the SpatialSeries in this CompassDirection container

class pynwb.behavior.**Position**(*spatial_series={}*, *name='Position'*)

Bases: *pynwb.core.MultiContainerInterface*

Position data, whether along the x, x/y or x/y/z axis.

Parameters

- **spatial_series** (list or tuple or dict or *SpatialSeries*) – SpatialSeries to store in this interface

- **name** (str) – the name of this container

__getitem__(*name=None*)
Get a SpatialSeries from this Position

Parameters **name** (str) – the name of the SpatialSeries

Returns the SpatialSeries with the given name

Return type SpatialSeries

add_spatial_series(*spatial_series*)
Add a SpatialSeries to this Position

Parameters **spatial_series** (list or tuple or dict or *SpatialSeries*) – the SpatialSeries to add

create_spatial_series(*name*, *data*, *reference_frame*, *conversion=1.0*, *resolution=-1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)
Create a SpatialSeries and add it to this Position

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **reference_frame** (str) – description defining what the zero-position is
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

Returns the SpatialSeries object that was created

Return type SpatialSeries

get_spatial_series(*name=None*)
Get a SpatialSeries from this Position

Parameters **name** (str) – the name of the SpatialSeries

Returns the SpatialSeries with the given name

Return type SpatialSeries

namespace = 'core'

neurodata_type = 'Position'

spatial_series
a dictionary containing the SpatialSeries in this Position container

13.9 pynwb.base module

class pynwb.base.**ProcessingModule**(*name*, *description*, *data_interfaces=None*)
Bases: *pynwb.core.MultiContainerInterface*

Processing module. This is a container for one or more containers that provide data at intermediate levels of analysis

ProcessingModules should be created through calls to NWB.create_module(). They should not be instantiated directly

Parameters

- **name** (str) – The name of this processing module
- **description** (str) – Description of this processing module
- **data_interfaces** (list or tuple or dict) – NWBDataInterfacess that belong to this ProcessingModule

description
Description of this processing module

data_interfaces
a dictionary containing the None in this ProcessingModule container

containers

__getitem__(*name=None*)
Get a NWBDataInterface from this ProcessingModule

Parameters **name** (str) – the name of the None

Returns the None with the given name

Return type (<class 'pynwb.core.NWBDataInterface'>, <class 'hdmf.common.table.DynamicTable'>)

add_container(*container*)
Add an NWBContainer to this ProcessingModule

Parameters **container** (*NWBDataInterface* or DynamicTable) – the NWBDataInterface to add to this Module

get_container(*container_name*)
Retrieve an NWBContainer from this ProcessingModule

Parameters **container_name** (str) – the name of the NWBContainer to retrieve

add_data_interface(*NWBDataInterface*)

Parameters **NWBDataInterface** (*NWBDataInterface* or DynamicTable) – the NWBDataInterface to add to this Module

get_data_interface(*data_interface_name*)

Parameters **data_interface_name** (str) – the name of the NWBContainer to retrieve

add(*data_interfaces*)
Add a NWBDataInterface to this ProcessingModule

Parameters **data_interfaces** (list or tuple or dict or *NWBDataInterface* or DynamicTable) – the None to add

get(*name=None*)
Get a NWBDataInterface from this ProcessingModule

Parameters **name** (str) – the name of the None

Returns the None with the given name

Return type (<class 'pynwb.core.NWBDataInterface'>, <class 'hdmf.common.table.DynamicTable'>)

namespace = 'core'

neurodata_type = 'ProcessingModule'

class pynwb.base.**TimeSeries**(*name*, *data=None*, *unit=None*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)
Bases: *pynwb.core.NWBDataInterface*

A generic base class for time series data

Create a TimeSeries object

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **unit** (str) – The base unit of measurement (should be SI unit)
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

timestamps_unit

interval

rate
Sampling rate in Hz

starting_time_unit

starting_time
The timestamp of the first sample

num_samples
Tries to return the number of data samples. If this cannot be assessed, returns None.

data

data_link

timestamps

timestamp_link

time_unit

comments
Human-readable comments about this TimeSeries dataset

control
Numerical labels that apply to each element in data

control_description
Description of each control value

conversion
Scalar to multiply each element in data to convert it to the specified unit

description
Description of this TimeSeries dataset

namespace = 'core'

neurodata_type = 'TimeSeries'

resolution
The smallest meaningful difference (in specified unit) between values in data

unit
The base unit of measurement (should be SI unit)

class pynwb.base.**Image**(*name*, *data*, *resolution=None*, *description=None*)
Bases: *pynwb.core.NWBData*

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – data of image
- **resolution** (*float*) – pixels / cm
- **description** (str) – description of image

namespace = 'core'

neurodata_type = 'Image'

class pynwb.base.**Images**(*name*, *images=None*, *description='no description'*)
Bases: *pynwb.core.MultiContainerInterface*

Parameters

- **name** (str) – The name of this set of images
- **images** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – image objects
- **description** (str) – description of images

description
description of images

__getitem__(*name=None*)
Get an Image from this Images

Parameters **name** (str) – the name of the Image

Returns the Image with the given name

Return type Image

add_image(*images*)
Add an Image to this Images

Parameters **images** (list or tuple or dict or *Image*) – the Image to add

create_image(*name*, *data*, *resolution=None*, *description=None*)
Create an Image and add it to this Images

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – data of image
- **resolution** (*float*) – pixels / cm
- **description** (str) – description of image

Returns the Image object that was created

Return type Image

get_image(*name=None*)
Get an Image from this Images

Parameters **name** (str) – the name of the Image

Returns the Image with the given name

Return type Image

images
a dictionary containing the Image in this Images container

namespace = 'core'

neurodata_type = 'Images'

13.10 pynwb.misc module

class pynwb.misc.**AnnotationSeries**(*name*, *data=[]*, *timestamps=None*, *comments='no comments'*, *description='no description'*)
Bases: *pynwb.base.TimeSeries*

Stores text-based records about the experiment. To use the AnnotationSeries, add records individually through add_annotation() and then call finalize(). Alternatively, if all annotations are already stored in a list, use set_data() and set_timestamps()

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset

add_annotation(*time*, *annotation*)

Add an annotation

Parameters

- **time** (*float*) – The time for the annotation
- **annotation** (str) – the annotation

namespace = 'core'

neurodata_type = 'AnnotationSeries'

class pynwb.misc.**AbstractFeatureSeries**(*name*, *feature_units*, *features*, *data=[]*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)

Bases: *pynwb.base.TimeSeries*

Represents the salient features of a data stream. Typically this will be used for things like a visual grating stimulus, where the bulk of data (each frame sent to the graphics card) is bulky and not of high value, while the salient characteristics (eg, orientation, spatial frequency, contrast, etc) are what important and are what are used for analysis

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **feature_units** (Iterable) – The unit of each feature
- **features** (Iterable) – Description of each feature
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data this TimeSeries dataset stores. Can also store binary data e.g. image frames
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit

- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

features
Description of each feature

feature_units
The unit of each feature

add_features(*time*, *features*)

Parameters

- **time** (*float*) – the time point of this feature
- **features** (list or ndarray) – the feature values for this time point

namespace = 'core'

neurodata_type = 'AbstractFeatureSeries'

class pynwb.misc.**IntervalSeries**(*name*, *data=[]*, *timestamps=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)

Bases: *pynwb.base.TimeSeries*

Stores intervals of data. The timestamps field stores the beginning and end of intervals. The data field stores whether the interval just started (>0 value) or ended (<0 value). Different interval types can be represented in the same series by using multiple key values (eg, 1 for feature A, 2 for feature B, 3 for feature C, etc). The field data stores an 8-bit integer. This is largely an alias of a standard TimeSeries but that is identifiable as representing time intervals in a machine-readable way.

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – >0 if interval started, <0 if interval ended.
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **description** (str) – Description of this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

add_interval(*start*, *stop*)

Parameters

- **start** (*float*) – The name of this TimeSeries dataset
- **stop** (*float*) – The name of this TimeSeries dataset

data

timestamps

namespace = 'core'

neurodata_type = 'IntervalSeries'

class pynwb.misc.**Units**(*name='Units'*, *id=None*, *columns=None*, *colnames=None*, *description=None*, *electrode_table=None*)

Bases: hdmf.common.table.DynamicTable

Event times of observed units (e.g. cell, synapse, etc.).

Parameters

- **name** (str) – Name of this Units interface
- **id** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or ElementIdentifiers) – the identifiers for this table
- **columns** (tuple or list) – the columns in this table
- **colnames** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – the names of the columns in this table
- **description** (str) – a description of what is in this table
- **electrode_table** (DynamicTable) – the table that the *electrodes* column indexes

add_unit(*spike_times=None*, *obs_intervals=None*, *electrodes=None*, *electrode_group=None*, *waveform_mean=None*, *waveform_sd=None*, *id=None*)

Add a unit to this table

Parameters

- **spike_times** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – the spike times for each unit
- **obs_intervals** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – the observation intervals (valid times) for each unit. All spike_times for a given unit should fall within these intervals. [[start1, end1], [start2, end2], ...]
- **electrodes** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – the electrodes that each unit came from
- **electrode_group** (*ElectrodeGroup*) – the electrode group that each unit came from
- **waveform_mean** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – the spike waveform mean for each unit. Shape is (time,) or (time, electrodes)
- **waveform_sd** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – the spike waveform standard deviation for each unit. Shape is (time,) or (time, electrodes)
- **id** (int) – the id for each unit

get_unit_spike_times(*index*, *in_interval=None*)

Parameters

- **index** (int or list or tuple or ndarray) – the index of the unit in unit_ids to retrieve spike times for
- **in_interval** (tuple or list) – only return values within this interval

get_unit_obs_intervals(*index*)

Parameters **index** (int) – the index of the unit in unit_ids to retrieve observation intervals for

namespace = 'core'

neurodata_type = 'Units'

class pynwb.misc.**DecompositionSeries**(*name*, *data*, *metric*, *description='no description'*, *unit='no unit'*, *bands=None*, *source_timeseries=None*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *control=None*, *control_description=None*)

Bases: *pynwb.base.TimeSeries*

Stores product of spectral analysis

Parameters

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – dims: num_times * num_channels * num_bands
- **metric** (str) – metric of analysis. recommended: 'phase', 'amplitude', 'power'
- **description** (str) – Description of this TimeSeries dataset
- **unit** (str) – SI unit of measurement
- **bands** (DynamicTable) – a table for describing the frequency bands that the signal was decomposed into
- **source_timeseries** (*TimeSeries*) – the input TimeSeries from this analysis
- **resolution** (str or float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (str or float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – Timestamps for samples stored in data
- **starting_time** (*float*) – The timestamp of the first sample
- **rate** (*float*) – Sampling rate in Hz
- **comments** (str) – Human-readable comments about this TimeSeries dataset
- **control** (Iterable) – Numerical labels that apply to each element in data
- **control_description** (Iterable) – Description of each control value

source_timeseries
the input TimeSeries from this analysis

metric
'phase', 'amplitude', 'power'

Type metric of analysis. recommended

bands
the bands that the signal is decomposed into

add_band(*band_name=None*, *band_limits=None*, *band_mean=None*, *band_stdev=None*)
Add ROI data to this

Parameters

- **band_name** (str) – the name of the frequency band
- **band_limits** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – low and high frequencies of band-pass filter in Hz
- **band_mean** (*float*) – the mean of Gaussian filters in Hz
- **band_stdev** (*float*) – the standard deviation of Gaussian filters in Hz

namespace = 'core'

neurodata_type = 'DecompositionSeries'

13.11 pynwb.epoch module

class pynwb.epoch.**TimeIntervals**(*name*, *description='experimental intervals'*, *id=None*, *columns=None*, *colnames=None*)
Bases: hdmf.common.table.DynamicTable

Table for storing Epoch data

Parameters

- **name** (str) – name of this TimeIntervals
- **description** (str) – Description of this TimeIntervals
- **id** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or ElementIdentifiers) – the identifiers for this table
- **columns** (tuple or list) – the columns in this table
- **colnames** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator) – the names of the columns in this table

add_interval(*start_time*, *stop_time*, *tags=None*, *timeseries=None*)

Parameters

- **start_time** (*float*) – Start time of epoch, in seconds
- **stop_time** (*float*) – Stop time of epoch, in seconds
- **tags** (str or list or tuple) – user-defined tags used throughout time intervals
- **timeseries** (list or tuple or *TimeSeries*) – the TimeSeries this epoch applies to

namespace = 'core'

neurodata_type = 'TimeIntervals'

13.12 pynwb package

13.12.1 Subpackages

pynwb.io package

Submodules

pynwb.io.base module

class pynwb.io.base.**ModuleMap**(*spec*)
Bases: *pynwb.io.core.NWBContainerMapper*

name(*builder*, *manager*)

constructor_args = {'name': <function ModuleMap.name>}

obj_attrs = {}

class pynwb.io.base.**TimeSeriesMap**(*spec*)
Bases: *pynwb.io.core.NWBContainerMapper*

name(*builder*, *manager*)

timestamps_attr(*container*, *manager*)

timestamps_carg(*builder*, *manager*)

constructor_args = {'name': <function TimeSeriesMap.name>, 'timestamps': <function T

obj_attrs = {'timestamps': <function TimeSeriesMap.timestamps_attr>}

pynwb.io.behavior module

pynwb.io.core module

class pynwb.io.core.**NWBBaseTypeMapper**(*spec*)
Bases: hdmf.build.map.ObjectMapper

Create a map from AbstractContainer attributes to specifications

Parameters **spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

static get_nwb_file(*container*)

constructor_args = {'name': <function ObjectMapper.get_container_name>}

obj_attrs = {}

class pynwb.io.core.**NWBContainerMapper**(*spec*)
Bases: *pynwb.io.core.NWBBaseTypeMapper*

Create a map from AbstractContainer attributes to specifications

Parameters **spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

constructor_args = {'name': <function ObjectMapper.get_container_name>}

obj_attrs = {}

class pynwb.io.core.**NWBDataMap**(*spec*)
Bases: *pynwb.io.core.NWBBaseTypeMapper*

Create a map from AbstractContainer attributes to specifications

Parameters **spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

carg_name(*builder*, *manager*)

carg_data(*builder*, *manager*)

constructor_args = {'data': <function NWBDataMap.carg_data>, 'name': <function NWBDa

obj_attrs = {}

class pynwb.io.core.**NWBTableRegionMap**(*spec*)
Bases: *pynwb.io.core.NWBDataMap*

Create a map from AbstractContainer attributes to specifications

Parameters **spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

carg_table(*builder*, *manager*)

carg_region(*builder*, *manager*)

constructor_args = {'data': <function NWBDataMap.carg_data>, 'name': <function NWBDa

obj_attrs = {}

pynwb.io.ecephys module

pynwb.io.epoch module

class pynwb.io.epoch.**TimeIntervalsMap**(*spec*)
Bases: hdmf.common.io.table.DynamicTableMap

constructor_args = {'name': <function ObjectMapper.get_container_name>}

obj_attrs = {'colnames': <function DynamicTableMap.attr_columns>}

pynwb.io.file module

class pynwb.io.file.**NWBFileMap**(*spec*)
Bases: hdmf.build.map.ObjectMapper

scratch_datas(*container*, *manager*)

scratch_containers(*container*, *manager*)

scratch(*builder*, *manager*)

dateconversion(*builder*, *manager*)

dateconversion_trt (*builder*, *manager*)

dateconversion_list (*builder*, *manager*)

name (*builder*, *manager*)

experimenter_carg (*builder*, *manager*)

experimenter_obj_attr (*container*, *manager*)

publications_carg (*builder*, *manager*)

publication_obj_attr (*container*, *manager*)

constructor_args = {'experimenter': <function NWBFileMap.experimenter_carg>, 'file_cr

obj_attrs = {'experimenter': <function NWBFileMap.experimenter_obj_attr>, 'related_pu

class pynwb.io.file.**SubjectMap**(*spec*)
Bases: hdmf.build.map.ObjectMapper

Create a map from AbstractContainer attributes to specifications

Parameters **spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

dateconversion (*builder*, *manager*)

constructor_args = {'date_of_birth': <function SubjectMap.dateconversion>, 'name': <

obj_attrs = {}

pynwb.io.icephys module

class pynwb.io.icephys.**SweepTableMap**(*spec*)
Bases: hdmf.common.io.table.DynamicTableMap

constructor_args = {'name': <function ObjectMapper.get_container_name>}

obj_attrs = {'colnames': <function DynamicTableMap.attr_columns>}

class pynwb.io.icephys.**VoltageClampSeriesMap**(*spec*)
Bases: *pynwb.io.base.TimeSeriesMap*

constructor_args = {'name': <function TimeSeriesMap.name>, 'timestamps': <function T

obj_attrs = {'timestamps': <function TimeSeriesMap.timestamps_attr>}

pynwb.io.image module

class pynwb.io.image.**ImageSeriesMap**(*spec*)
Bases: *pynwb.io.base.TimeSeriesMap*

constructor_args = {'name': <function TimeSeriesMap.name>, 'timestamps': <function T

obj_attrs = {'timestamps': <function TimeSeriesMap.timestamps_attr>}

pynwb.io.misc module

class pynwb.io.misc.**UnitsMap**(*spec*)
Bases: hdmf.common.io.table.DynamicTableMap

electrodes_column(*container*, *manager*)

constructor_args = {'name': <function ObjectMapper.get_container_name>}

obj_attrs = {'colnames': <function DynamicTableMap.attr_columns>, 'electrodes': <fun

pynwb.io.ogen module

pynwb.io.ophys module

class pynwb.io.ophys.**PlaneSegmentationMap**(*spec*)
Bases: hdmf.common.io.table.DynamicTableMap

constructor_args = {'name': <function ObjectMapper.get_container_name>}

obj_attrs = {'colnames': <function DynamicTableMap.attr_columns>}

class pynwb.io.ophys.**ImagingPlaneMap**(*spec*)
Bases: *pynwb.io.core.NWBContainerMapper*

constructor_args = {'name': <function ObjectMapper.get_container_name>}

obj_attrs = {}

pynwb.io.retinotopy module

Module contents

pynwb.legacy package

Subpackages

pynwb.legacy.io package

Submodules

pynwb.legacy.io.base module

class pynwb.legacy.io.base.**ModuleMap**(*spec*)
Bases: *pynwb.legacy.map.ObjectMapperLegacy*

name(**args*)

carg_description(**args*)

constructor_args = {'description': <function ModuleMap.carg_description>, 'name': <f

obj_attrs = {}

class pynwb.legacy.io.base.**TimeSeriesMap**(*spec*)
Bases: *pynwb.legacy.map.ObjectMapperLegacy*

carg_name(**args*)

carg_starting_time(**args*)

carg_rate(**args*)

constructor_args = {'name': <function TimeSeriesMap.carg_name>, 'rate': <function Ti

obj_attrs = {}

pynwb.legacy.io.behavior module

class pynwb.legacy.io.behavior.**BehavioralTimeSeriesMap**(*spec*)
Bases: *pynwb.legacy.map.ObjectMapperLegacy*

Create a map from AbstractContainer attributes to specifications

Parameters **spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

carg_time_series(**args*)

constructor_args = {'name': <function ObjectMapper.get_container_name>, 'source': <f

obj_attrs = {}

class pynwb.legacy.io.behavior.**PupilTrackingMap**(*spec*)
Bases: *pynwb.legacy.map.ObjectMapperLegacy*

Create a map from AbstractContainer attributes to specifications

Parameters **spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

carg_time_series(**args*)

constructor_args = {'name': <function ObjectMapper.get_container_name>, 'source': <f

obj_attrs = {}

pynwb.legacy.io.ecephys module

pynwb.legacy.io.epoch module

class pynwb.legacy.io.epoch.**EpochMap**(*spec*)
Bases: *pynwb.legacy.map.ObjectMapperLegacy*

name(*builder*)

constructor_args = {'name': <function EpochMap.name>, 'source': <function ObjectMapp

obj_attrs = {}

class pynwb.legacy.io.epoch.**EpochTimeSeriesMap**(*spec*)
Bases: *pynwb.legacy.map.ObjectMapperLegacy*

constructor_args = {'name': <function ObjectMapper.get_container_name>, 'source': <f

obj_attrs = {}

pynwb.legacy.io.file module

class pynwb.legacy.io.file.**NWBFileMap**(*spec*)
Bases: *pynwb.legacy.map.ObjectMapperLegacy*

name(*builder*)

constructor_args = {'file_name': <function NWBFileMap.name>, 'name': <function Objec

obj_attrs = {}

pynwb.legacy.io.icephys module

class pynwb.legacy.io.icephys.**PatchClampSeriesMap**(*spec*)
Bases: *pynwb.legacy.map.ObjectMapperLegacy*

Create a map from AbstractContainer attributes to specifications

Parameters **spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

carg_electrode(**args*)

constructor_args = {'electrode': <function PatchClampSeriesMap.carg_electrode>, 'name

obj_attrs = {}

pynwb.legacy.io.image module

class pynwb.legacy.io.image.**ImageSeriesMap**(*spec*)
Bases: *pynwb.legacy.map.ObjectMapperLegacy*

Create a map from AbstractContainer attributes to specifications

Parameters **spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

carg_data(**args*)

constructor_args = {'data': <function ImageSeriesMap.carg_data>, 'name': <function O

obj_attrs = {}

pynwb.legacy.io.misc module

class pynwb.legacy.io.misc.**AbstractFeatureSeriesMap**(*spec*)
Bases: *pynwb.legacy.map.ObjectMapperLegacy*

Create a map from AbstractContainer attributes to specifications

Parameters **spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

carg_feature_units(**args*)

constructor_args = {'feature_units': <function AbstractFeatureSeriesMap.carg_feature_

obj_attrs = {}

pynwb.legacy.io.ogen module

class pynwb.legacy.io.ogen.**OptogeneticSeriesMap**(*spec*)
Bases: *pynwb.legacy.map.ObjectMapperLegacy*

Create a map from AbstractContainer attributes to specifications

Parameters **spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

carg_site(**args*)

constructor_args = {'name': <function ObjectMapper.get_container_name>, 'site': <fun

obj_attrs = {}

pynwb.legacy.io.ophys module

class pynwb.legacy.io.ophys.**PlaneSegmentationMap**(*spec*)
Bases: *pynwb.legacy.map.ObjectMapperLegacy*

carg_imaging_plane(**args*)

constructor_args = {'imaging_plane': <function PlaneSegmentationMap.carg_imaging_plan

obj_attrs = {}

class pynwb.legacy.io.ophys.**TwoPhotonSeriesMap**(*spec*)
Bases: *pynwb.legacy.map.ObjectMapperLegacy*

Create a map from AbstractContainer attributes to specifications

Parameters **spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

carg_data(**args*)

carg_unit(**args*)

carg_imaging_plane(**args*)

constructor_args = {'data': <function TwoPhotonSeriesMap.carg_data>, 'imaging_plane':

obj_attrs = {}

pynwb.legacy.io.retinotopy module

Module contents

Submodules

pynwb.legacy.map module

pynwb.legacy.map.**decode**(*val*)

class pynwb.legacy.map.**ObjectMapperLegacy**(*spec*)
Bases: hdmf.build.map.ObjectMapper

Create a map from AbstractContainer attributes to specifications

Parameters **spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

source_gettr(*builder*, *manager*)

constructor_args = {'name': <function ObjectMapper.get_container_name>, 'source': <f

obj_attrs = {}

class pynwb.legacy.map.**TypeMapLegacy**(*namespaces=None*, *mapper_cls=<class 'hdmf.build.map.ObjectMapper'>*)

Bases: hdmf.build.map.TypeMap

Parameters

- **namespaces** (NamespaceCatalog) – the NamespaceCatalog to use
- **mapper_cls** (type) – the ObjectMapper class to use

get_builder_dt(*builder*)

For a given builder, return the neurodata_type. In this legacy TypeMap, the builder may have out-of-spec neurodata_type; this function coerces this to a 2.0 compatible version.

get_builder_ns(*builder*)

Get the namespace of a builder

Parameters **builder** (DatasetBuilder or GroupBuilder or LinkBuilder) – the builder to get the sub-specification for

Module contents

pynwb.legacy.**get_type_map**(***kwargs*)

Get a TypeMap to use for I/O for Allen Institute Brain Observatory files (NWB v1.0.6)

pynwb.legacy.**register_map**(*container_cls*, *mapper_cls=None*)

Register an ObjectMapper to use for a Container class type If mapper_cls is not specified, returns a decorator for registering an ObjectMapper class as the mapper for container_cls. If mapper_cls specified, register the class as the mapper for container_cls

Parameters

- **container_cls** (type) – the Container class for which the given ObjectMapper class gets used for
- **mapper_cls** (type) – the ObjectMapper class to use to map

pynwb.testing package

Module contents

pynwb.testing.**remove_test_file**(*path*)

A helper function for removing intermediate test files

This checks if the environment variable CLEAN_NWB has been set to False before removing the file. If CLEAN_NWB is set to False, it does not remove the file.

13.12.2 Submodules

pynwb.core module

pynwb.core.**prepend_string**(*string*, *prepend=' '*)

class pynwb.core.**NWBMixin**(*name*)

Bases: hdmf.container.AbstractContainer

Parameters **name** (str) – the name of this container

get_ancestor(*neurodata_type=None*)
Traverse parent hierarchy and return first instance of the specified data_type

Parameters **neurodata_type** (str) – the data_type to search for

class pynwb.core.**NWBContainer**(*name*)
Bases: *pynwb.core.NWBMixin*, hdmf.container.Container

Parameters **name** (str) – the name of this container

namespace = 'core'

neurodata_type = 'NWBContainer'

class pynwb.core.**NWBDataInterface**(*name*)
Bases: *pynwb.core.NWBContainer*

Parameters **name** (str) – the name of this container

namespace = 'core'

neurodata_type = 'NWBDataInterface'

class pynwb.core.**NWBData**(*name*, *data*)
Bases: *pynwb.core.NWBMixin*, hdmf.container.Data

Parameters

- **name** (str) – the name of this container
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or Data) – the source of the data

data

__getitem__(*args*)

append(*arg*)

extend(*arg*)

namespace = 'core'

neurodata_type = 'NWBData'

class pynwb.core.**ScratchData**(*name*, *data*, *notes=''*)
Bases: *pynwb.core.NWBData*

Parameters

- **name** (str) – the name of this container
- **data** (ndarray or list or tuple or Dataset or HDMFDataset or AbstractDataChunkIterator or DataIO or Data) – the source of the data
- **notes** (str) – notes about the data

namespace = 'core'

neurodata_type = 'ScratchData'

class pynwb.core.**NWBTable**(*columns*, *name*, *data=[]*)
Bases: *pynwb.core.NWBData*

Subclasses should specify the class attribute __columns__.

This should be a list of dictionaries with the following keys:

- name the column name

- `type` the type of data in this column
- `doc` a brief description of what gets stored in this column

For reference, this list of dictionaries will be used with docval to autogenerate the `add_row` method for adding data to this table.

If __columns__ is not specified, no custom `add_row` method will be added.

The class attribute __defaultname__ can also be set to specify a default name for the table class. If __defaultname__ is not specified, then `name` will need to be specified when the class is instantiated.

Parameters

- **columns** (`list` or `tuple`) – a list of the columns in this table
- **name** (`str`) – the name of this container
- **data** (`ndarray` or `list` or `tuple` or `Dataset` or `HDMFDataset` or `AbstractDataChunkIterator` or `DataIO`) – the source of the data

columns

add_row(*values*)

Parameters **values** (`dict`) – the values for each column

which(***kwargs*)
Query a table

__getitem__(*args*)

to_dataframe()
Produce a pandas DataFrame containing this table's data.

classmethod from_dataframe(*df*, *name=None*, *extra_ok=False*)

Construct an instance of NWBTable (or a subclass) from a pandas DataFrame. The columns of the dataframe should match the columns defined on the NWBTable subclass.

Parameters

- **df** (`DataFrame`) – input data
- **name** (`str`) – the name of this container
- **extra_ok** (`bool`) – accept (and ignore) unexpected columns on the input dataframe

class pynwb.core.**NWBTableRegion**(*name*, *table*, *region*)
Bases: *pynwb.core.NWBData*, `hdmf.container.DataRegion`

A class for representing regions i.e. slices or indices into an NWBTable

Parameters

- **name** (`str`) – the name of this container
- **table** (*NWBTable*) – the ElectrodeTable this region applies to
- **region** (`slice` or `list` or `tuple` or `RegionReference`) – the indices of the table

table
The ElectrodeTable this region applies to

region
The indices into table

__getitem__(*idx*)

class pynwb.core.**MultiContainerInterface**(*name*)
Bases: *pynwb.core.NWBDataInterface*

A class for dynamically defining a API classes that represent NWBDataInterfaces that contain multiple Containers of the same type

To use, extend this class, and create a dictionary as a class attribute with the following keys:

- 'add' to name the method for adding Container instances
- 'create' to name the method fo creating Container instances
- 'get' to name the method for getting Container instances
- 'attr' to name the attribute that stores the Container instances
- 'type' to provide the Container object type

See LFP or Position for an example of how to use this.

Parameters **name** (str) – the name of this container

class pynwb.core.**LabelledDict**(*label*, *def_key_name='name'*)
Bases: dict

A dict wrapper class for aggregating Timeseries from the standard locations

Parameters

- **label** (str) – the label on this dictionary
- **def_key_name** (str) – the default key name

label

__getitem__(*args*)
x.__getitem__(y) <==> x[y]

add(*container*)
Add a container to this LabelledDict

Parameters **container** (*NWBData* or *NWBContainer*) – the container to add to this LabelledDict

pynwb.device module

class pynwb.device.**Device**(*name*)
Bases: *pynwb.core.NWBContainer*

Parameters **name** (str) – the name of this device

namespace = 'core'

neurodata_type = 'Device'

pynwb.spec module

class pynwb.spec.**NWBRefSpec**(*target_type*, *reftype*)
Bases: hdmf.spec.spec.RefSpec

Parameters

- **target_type** (str) – the target type GroupSpec or DatasetSpec

- **reftype** (str) – the type of references this is i.e. region or object

class pynwb.spec.**NWBAttributeSpec**(*name*, *doc*, *dtype*, *shape=None*, *dims=None*, *required=True*, *parent=None*, *value=None*, *default_value=None*)

Bases: hdmf.spec.spec.AttributeSpec

Parameters

- **name** (str) – The name of this attribute
- **doc** (str) – a description about what this specification represents
- **dtype** (str or RefSpec) – The data type of this attribute
- **shape** (list or tuple) – the shape of this dataset
- **dims** (list or tuple) – the dimensions of this dataset
- **required** (bool) – whether or not this attribute is required. ignored when "value" is specified
- **parent** (*BaseStorageSpec*) – the parent of this spec
- **value** (*None*) – a constant value for this attribute
- **default_value** (*None*) – a default value for this attribute

class pynwb.spec.**NWBLinkSpec**(*doc*, *target_type*, *quantity=1*, *name=None*)

Bases: hdmf.spec.spec.LinkSpec

Parameters

- **doc** (str) – a description about what this link represents
- **target_type** (str) – the target type GroupSpec or DatasetSpec
- **quantity** (str or int) – the required number of allowed instance
- **name** (str) – the name of this link

neurodata_type_inc
The neurodata type of target specification

class pynwb.spec.**BaseStorageOverride**

Bases: object

This class is used for the purpose of overriding BaseStorageSpec classmethods, without creating diamond inheritance hierarchies.

classmethod type_key()
Get the key used to store data type on an instance

classmethod inc_key()
Get the key used to define a data_type include.

classmethod def_key()
Get the key used to define a data_type definition.

neurodata_type_inc

neurodata_type_def

classmethod build_const_args(*spec_dict*)
Extend base functionality to remap data_type_def and data_type_inc keys

class pynwb.spec.**NWBDtypeSpec**(*name*, *doc*, *dtype*)

Bases: hdmf.spec.spec.DtypeSpec

Parameters

- **name** (str) – the name of this column
- **doc** (str) – a description about what this data type is
- **dtype** (str or list or RefSpec) – the data type of this column

class pynwb.spec.**NWBDatasetSpec**(*doc*, *dtype=None*, *name=None*, *default_name=None*, *shape=None*, *dims=None*, *attributes=[]*, *linkable=True*, *quantity=1*, *default_value=None*, *neurodata_type_def=None*, *neurodata_type_inc=None*)

Bases: *pynwb.spec.BaseStorageOverride*, hdmf.spec.spec.DatasetSpec

The Spec class to use for NWB specifications

Parameters

- **doc** (str) – a description about what this specification represents
- **dtype** (str or list or RefSpec) – The data type of this attribute. Use a list of DtypeSpecs to specify a compound data type.
- **name** (str) – The name of this dataset
- **default_name** (str) – The default name of this dataset
- **shape** (list or tuple) – the shape of this dataset
- **dims** (list or tuple) – the dimensions of this dataset
- **attributes** (list) – the attributes on this group
- **linkable** (bool) – whether or not this group can be linked
- **quantity** (str or int) – the required number of allowed instance
- **default_value** (*None*) – a default value for this dataset
- **neurodata_type_def** (str) – the NWB data type this spec defines
- **neurodata_type_inc** (NWBDatasetSpec or str) – the NWB data type this spec includes

class pynwb.spec.**NWBGroupSpec**(*doc*, *name=None*, *default_name=None*, *groups=[]*, *datasets=[]*, *attributes=[]*, *links=[]*, *linkable=True*, *quantity=1*, *neurodata_type_def=None*, *neurodata_type_inc=None*)

Bases: *pynwb.spec.BaseStorageOverride*, hdmf.spec.spec.GroupSpec

The Spec class to use for NWB specifications

Parameters

- **doc** (str) – a description about what this specification represents
- **name** (str) – the name of this group
- **default_name** (str) – The default name of this group
- **groups** (list) – the subgroups in this group
- **datasets** (list) – the datasets in this group
- **attributes** (list) – the attributes on this group
- **links** (list) – the links in this group
- **linkable** (bool) – whether or not this group can be linked

- **quantity** (str or int) – the required number of allowed instance
- **neurodata_type_def** (str) – the NWB data type this spec defines
- **neurodata_type_inc** (NWBGroupSpec or str) – the NWB data type this spec includes

classmethod dataset_spec_cls()
The class to use when constructing DatasetSpec objects

Override this if extending to use a class other than DatasetSpec to build dataset specifications

get_neurodata_type(*neurodata_type*)
Get a specification by "data_type"

Parameters **neurodata_type** (str) – the neurodata_type to retrieve

add_group(*doc*, *name=None*, *default_name=None*, *groups=[]*, *datasets=[]*, *attributes=[]*, *links=[]*, *linkable=True*, *quantity=1*, *neurodata_type_def=None*, *neurodata_type_inc=None*)
Add a new specification for a subgroup to this group specification

Parameters

- **doc** (str) – a description about what this specification represents
- **name** (str) – the name of this group
- **default_name** (str) – The default name of this group
- **groups** (list) – the subgroups in this group
- **datasets** (list) – the datasets in this group
- **attributes** (list) – the attributes on this group
- **links** (list) – the links in this group
- **linkable** (bool) – whether or not this group can be linked
- **quantity** (str or int) – the required number of allowed instance
- **neurodata_type_def** (str) – the NWB data type this spec defines
- **neurodata_type_inc** (NWBGroupSpec or str) – the NWB data type this spec includes

add_dataset(*doc*, *dtype=None*, *name=None*, *default_name=None*, *shape=None*, *dims=None*, *attributes=[]*, *linkable=True*, *quantity=1*, *default_value=None*, *neurodata_type_def=None*, *neurodata_type_inc=None*)
Add a new specification for a subgroup to this group specification

Parameters

- **doc** (str) – a description about what this specification represents
- **dtype** (str or list or RefSpec) – The data type of this attribute. Use a list of DtypeSpecs to specify a compound data type.
- **name** (str) – The name of this dataset
- **default_name** (str) – The default name of this dataset
- **shape** (list or tuple) – the shape of this dataset
- **dims** (list or tuple) – the dimensions of this dataset
- **attributes** (list) – the attributes on this group
- **linkable** (bool) – whether or not this group can be linked

- **quantity** (str or int) – the required number of allowed instance
- **default_value** (*None*) – a default value for this dataset
- **neurodata_type_def** (str) – the NWB data type this spec defines
- **neurodata_type_inc** (NWBDatasetSpec or str) – the NWB data type this spec includes

class pynwb.spec.**NWBNamespace**(*doc*, *name*, *schema*, *full_name=None*, *version=None*, *date=None*, *author=None*, *contact=None*, *catalog=None*)

Bases: hdmf.spec.namespace.SpecNamespace

A Namespace class for NWB

Parameters

- **doc** (str) – a description about what this namespace represents
- **name** (str) – the name of this namespace
- **schema** (list) – location of schema specification files or other Namespaces
- **full_name** (str) – extended full name of this namespace
- **version** (str or tuple or list) – Version number of the namespace
- **date** (datetime or str) – Date last modified or released. Formatting is %Y-%m-%d %H:%M:%S, e.g, 2017-04-25 17:14:13
- **author** (str or list) – Author or list of authors.
- **contact** (str or list) – List of emails. Ordering should be the same as for author
- **catalog** (SpecCatalog) – The SpecCatalog object for this SpecNamespace

classmethod types_key()

Get the key used for specifying types to include from a file or namespace

Override this method to use a different name for 'data_types'

class pynwb.spec.**NWBNamespaceBuilder**(*doc*, *name*, *full_name=None*, *version=None*, *author=None*, *contact=None*)

Bases: hdmf.spec.write.NamespaceBuilder

A class for writing namespace and spec files for extensions of types in the NWB core namespace

Create a NWBNamespaceBuilder

Parameters

- **doc** (str) – a description about what this namespace represents
- **name** (str) – the name of this namespace
- **full_name** (str) – extended full name of this namespace
- **version** (str or tuple or list) – Version number of the namespace
- **author** (str or list) – Author or list of authors.
- **contact** (str or list) – List of emails. Ordering should be the same as for author

pynwb.validate module

pynwb.validate.**main**()

13.12.3 Module contents

This package will contain functions, classes, and objects for reading and writing data in NWB format

pynwb.**get_type_map**(*extensions=None*)

Get a BuildManager to use for I/O using the given extensions. If no extensions are provided, return a BuildManager that uses the core namespace

Parameters **extensions** (str or TypeMap or list) – a path to a namespace, a TypeMap, or a list consisting of paths to namespaces and TypeMaps

Returns the namespaces loaded from the given file

Return type tuple

pynwb.**get_manager**(*extensions=None*)

Get a BuildManager to use for I/O using the given extensions. If no extensions are provided, return a BuildManager that uses the core namespace

Parameters **extensions** (str or TypeMap or list) – a path to a namespace, a TypeMap, or a list consisting of paths to namespaces and TypeMaps

Returns the namespaces loaded from the given file

Return type tuple

pynwb.**load_namespaces**(*namespace_path*)

Load namespaces from file

Parameters **namespace_path** (str) – the path to the YAML with the namespace definition

Returns the namespaces loaded from the given file

Return type tuple

pynwb.**available_namespaces**()

Returns all namespaces registered in the namespace catalog

pynwb.**register_class**(*neurodata_type*, *namespace*, *container_cls=None*)

Register an NWBContainer class to use for reading and writing a neurodata_type from a specification If container_cls is not specified, returns a decorator for registering an NWBContainer subclass as the class for neurodata_type in namespace.

Parameters

- **neurodata_type** (str) – the neurodata_type to get the spec for
- **namespace** (str) – the name of the namespace
- **container_cls** (type) – the class to map to the specified neurodata_type

pynwb.**register_map**(*container_cls*, *mapper_cls=None*)

Register an ObjectMapper to use for a Container class type If mapper_cls is not specified, returns a decorator for registering an ObjectMapper class as the mapper for container_cls. If mapper_cls is specified, register the class as the mapper for container_cls

Parameters

- **container_cls** (type) – the Container class for which the given ObjectMapper class gets used
- **mapper_cls** (type) – the ObjectMapper class to use to map

pynwb.**get_class**(*neurodata_type*, *namespace*)
Get the class object of the NWBContainer subclass corresponding to a given neurodata_type.

Parameters

- **neurodata_type** (str) – the neurodata_type to get the NWBContainer class for
- **namespace** (str) – the namespace the neurodata_type is defined in

pynwb.**validate**(*io*, *namespace='core'*)
Validate an NWB file against a namespace

Parameters

- **io** (HDMFIO) – the HDMFIO object to read from
- **namespace** (str) – the namespace to validate against

Returns errors in the file

Return type list

class pynwb.**NWBHDF5IO**(*path*, *mode*, *load_namespaces=False*, *manager=None*, *extensions=None*, *file=None*, *comm=None*)
Bases: hdmf.backends.hdf5.h5tools.HDF5IO

Parameters

- **path** (str) – the path to the HDF5 file
- **mode** (str) – the mode to open the HDF5 file with, one of ("w", "r", "r+", "a", "w-", "x")
- **load_namespaces** (bool) – whether or not to load cached namespaces from given path - not applicable in write mode
- **manager** (BuildManager) – the BuildManager to use for I/O
- **extensions** (str or TypeMap or list) – a path to a namespace, a TypeMap, or a list consisting paths to namespaces and TypeMaps
- **file** (File) – a pre-existing h5py.File object
- **comm** (*Intracomm*) – the MPI communicator to use for parallel I/O

modindex

CHAPTER 14

Software Process

14.1 Continuous Integration

PyNWB is tested against Ubuntu, macOS and Windows operating systems. The project has both unit and integration tests.

- CircleCI runs all PyNWB tests on Ubuntu
- Appveyor runs all PyNWB tests on Windows
- Travis runs all PyNWB tests on macOS

Each time a PR is published or updated, the project is built, packaged and tested on all support operating systems and python distributions. That way, as a contributor you know if you introduced regressions or coding style inconsistencies.

There are badges in the README file which shows the current condition of the dev branch.

14.2 Coverage

Coverage is computed and reported using the coverage tool. There is a badge in the README file which shows percentage coverage. A detailed report can be found on codecov which shows line by line which lines are covered by the tests.

14.3 Requirement Specifications

There are 2 kinds of requirements specification in PyNWB.

14.3.1 Setup.py Dependencies

The first one is the dependencies in the *setup.py* file which lists the abstract dependencies for the PyNWB project. Note that there should not be specific versions of packages in the *setup.py* file.

14.3.2 Requirements.txt Dependencies

The second one is *requirements.txt* which contain a list of pinned (concrete) dependencies to reproduce an entire development environment to work with PyNWB.

In order to check the status of the required packages requires.io is used to create a badge on the project README. If all the required packages are up to date, a green badge appears.

If some of the packages are outdated, see *How to Update Requirements Files*.

14.4 Versioning and Releasing

PyNWB uses versioneer for versioning source and wheel distributions. Versioneer creates a semi-unique release name for the wheels that are created. It requires a version control system (git in PyNWB's case) to generate a release name. After all the tests pass, CircleCI creates both wheels(*.whl) and source distribution(*.tgz) for Python 3 and uploads them back to GitHub as a release. Versioneer makes it possible to get the source distribution from GitHub and create wheels directly without having to use a version control system because it hardcodes versions in the source distribution.

CHAPTER 15

How to Make a Roundtrip Test

The PyNWB test suite has tools for easily doing round-trip tests of container classes. These tools exist in the integration test suite in `tests/integration/ui_write/base.py` for this reason and for the sake of keeping the repository organized, we recommend you write your tests in the `tests/integration/ui_write` subdirectory of the Git repository.

For executing your new tests, we recommend using the *test.py* script in the top of the Git repository. Roundtrip tests will get executed as part of the integration test suite, which can be executed with the following command:

```
$ python test.py -i
```

The roundtrip test will generate a new NWB file with the name `test_<CLASS_NAME>.nwb` where `CLASS_NAME` is the class name of the `Container` class you are roundtripping. The test will write an NWB file with an instance of the container to disk, read this instance back in, and compare it to the instance that was used for writing to disk. Once the test is complete, the NWB file will be deleted. You can keep the NWB file around after the test completes by setting the environment variable `CLEAN_NWB` to `0`, `false`, `False`, or `FALSE`. Setting `CLEAN_NWB` to any value not listed here will cause the roundtrip NWB file to be deleted once the test has completed

Before writing tests, we also suggest you familiarize yourself with the *software architecture* of PyNWB.

15.1 `TestMapRoundTrip`

To write a roundtrip test, you will need to subclass the `TestMapRoundTrip` class and override some of its instance methods.

`TestMapRoundTrip` provides four methods for testing the process of going from in-memory Python object to data stored on disk and back. Three of these methods–`setUpContainer`, `addContainer`, and `getContainer`–are required for carrying out the roundtrip test. The fourth method is required for testing the conversion from the container to the `builder`–the intermediate data structure that gets used by `HDMFIO` implementations for writing to disk.

If you do not want to test step of the process, you can just implement `setUpContainer`, `addContainer`, and `getContainer`.

15.1.1 setUpContainer

The first thing (and possibly the *only* thing – see *TestDataInterfaceIO*) you need to do is override is the `setUpContainer` method. This method should take no arguments, and return an instance of the container class you are testing.

Here is an example using a generic *TimeSeries*:

```
from . base import TestMapRoundTrip

class TimeSeriesRoundTrip(TestMapRoundTrip):

    def setUpContainer(self):
        return TimeSeries('test_timeseries', 'example_source', list(range(100, 200,␣
↪10)),
                          'SIunit', timestamps=list(range(10)), resolution=0.1)
```

15.1.2 addContainer

The next thing is to tell the `TestMapRoundTrip` how to add the container to an NWBFile. This method takes a single argument–the *NWBFile* instance that will be used to write your container.

This method is required because different container types are allowed in different parts of an NWBFile. This method is also where you can add additonial containers that your container of interest depends on. For example, for the *ElectricalSeries* roundtrip test, `addContainer` handles adding the *ElectrodeGroup*, *ElectrodeTable*, and *Device* dependencies.

Continuing from our example above, we will add the method for adding a generic *TimeSeries* instance:

```
class TimeSeriesRoundTrip(TestMapRoundTrip):

    def addContainer(self, nwbfile):
        nwbfile.add_acquisition(self.container)
```

15.1.3 getContainer

Finally, you need to tell `TestMapRoundTrip` how to get back the container we added. As with `addContainer`, this method takes an *NWBFile* as its single argument. The only difference is that this *NWBFile* instance is what was read back in.

Again, since not all containers go in the same place, we need to tell the test harness how to get back our container of interest.

To finish off example from above, we will add the method for getting back our generic *TimeSeries* instance:

```
class TimeSeriesRoundTrip(TestMapRoundTrip):

    def getContainer(self, nwbfile):
        return nwbfile.get_acquisition(self.container.name)
```

15.1.4 setUpBuilder

As mentioned above, there is an optional method to override. This method will add two additional tests. First, it will add a test for converting your container into a builder to make sure the intermerdiate data structure gets built

appropriately. Second it will add a test for constructing your container from the builder returned by your overridden `setUpBuilder` method. This method takes no arguments, and should return the builder representation of your container class instance.

This method is not required, but can serve as an additional check to make sure your containers are getting converted to the expected structure as described in your specification.

Continuing from the *TimeSeries* example, lets add `setUpBuilder`:

```
from hdmf.build import GroupBuilder

class TimeSeriesRoundTrip(TestMapRoundTrip):

    def setUpBuilder(self):
        return GroupBuilder('test_timeseries',
                            attributes={'source': 'example_source',
                                        'namespace': base.CORE_NAMESPACE,
                                        'neurodata_type': 'TimeSeries',
                                        'description': 'no description',
                                        'comments': 'no comments'},
                            datasets={'data': DatasetBuilder('data', list(range(100,␣
↪200, 10)),
                                                             attributes={'unit':
↪'SIunit',
                                                                         'conversion
↪': 1.0,
                                                                         'resolution
↪': 0.1}),
                                      'timestamps': DatasetBuilder('timestamps',␣
↪list(range(10)),
                                                                   attributes={'unit
↪': 'Seconds', 'interval': 1})})
```

15.2 TestDataInterfaceIO

If you are testing something that can go in *acquisition*, you can avoid writing `addContainer` and `getContainer` by extending `TestDataInterfaceIO`. This class has already overridden these methods to add your container object to acquisition.

Even if your container can go in acquisition, you may still need to override `addContainer` if your container depends other containers that you need to add to the *NWBFile* that will be written.

CHAPTER 16

How to Make a Release

A core developer should use the following steps to create a release *X.Y.Z* of **pynwb**.

Note: Since the pynwb wheels do not include compiled code, they are considered *pure* and could be generated on any supported platform.

That said, considering the instructions below have been tested on a Linux system, they may have to be adapted to work on macOS or Windows.

16.1 Prerequisites

- All CI tests are passing on CircleCI and Azure Pipelines.
- You have a GPG signing key.

16.2 Documentation conventions

The commands reported below should be evaluated in the same terminal session.

Commands to evaluate starts with a dollar sign. For example:

```
$ echo "Hello"
Hello
```

means that `echo "Hello"` should be copied and evaluated in the terminal.

16.3 Setting up environment

1. First, register for an account on PyPI.

2. If not already the case, ask to be added as a `Package Index Maintainer`.
3. Create a `~/.pypirc` file with your login credentials:

```
[distutils]
index-servers =
  pypi
  pypitest

[pypi]
username=<your-username>
password=<your-password>

[pypitest]
repository=https://test.pypi.org/legacy/
username=<your-username>
password=<your-password>
```

where `<your-username>` and `<your-password>` correspond to your PyPI account.

16.4 PyPI: Step-by-step

1. Make sure that all CI tests are passing on CircleCI and Azure Pipelines.
2. List all tags sorted by version

```
$ git tag -l | sort -V
```

3. Choose the next release version number

```
$ release=X.Y.Z
```

> **Warning:** To ensure the packages are uploaded on PyPI, tags must match this regular expression: `^[0-9]+(\.[0-9]+)*(\.post[0-9]+)?$`.

4. Download latest sources

```
$ cd /tmp && git clone --recurse-submodules git@github.
↪com:NeurodataWithoutBorders/pynwb && cd pynwb
```

5. Tag the release

```
$ git tag --sign -m "pynwb ${release}" ${release} origin/dev
```

> **Warning:** This step requires a GPG signing key.

6. Publish the release tag

```
$ git push origin ${release}
```

Important: This will trigger builds on each CI services and automatically upload the wheels and source distribution on PyPI.

---

7. Check the status of the builds on CircleCI and Azure Pipelines.
8. Once the builds are completed, check that the distributions are available on PyPI and that a new GitHub release was created.
9. Create a clean testing environment to test the installation

```
$ mkvirtualenv pynwb-${release}-install-test && \
  pip install pynwb && \
  python -c "import pynwb; print(pynwb.__version__)"
```

Note: If the `mkvirtualenv` command is not available, this means you do not have virtualenvwrapper installed, in that case, you could either install it or directly use virtualenv or venv.

10. Cleanup

```
$ deactivate  && \
  rm -rf dist/* && \
  rmvirtualenv pynwb-${release}-install-test
```

16.5 Conda: Step-by-step

Warning: Publishing on conda requires you to have corresponding package version uploaded on PyPI. So you have to do the PypI and Github release before you do the conda release.

In order to release a new version on conda-forge, follow the steps below:

1. Choose the next release version number (that matches with the pypi version that you already published)

```
$ release=X.Y.Z
```

2. Fork pynwb-feedstock

   First step is to fork pynwb-feedstock repository. This is the recommended best practice by conda.

3. Clone forked feedstock

   Fill the YOURGITHUBUSER part.

```
$ cd /tmp && git clone https://github.com/YOURGITHUBUSER/pynwb-feedstock.git
```

4. Download corresponding source for the release version

```
$ cd /tmp && \
  wget https://github.com/NeurodataWithoutBorders/pynwb/releases/download/
↪$release/pynwb-$release.tar.gz
```

5. Create a new branch

```
$ cd pynwb-feedstock && \
  git checkout -b $release
```

6. Modify `meta.yaml`

   Update the version string and sha256.

   We have to modify the sha and the version string in the `meta.yaml` file.

   For linux flavors:

```
$ sed -i "2s/.*/{% set version = \"$release\" %}/" recipe/meta.yaml
$ sha=$(openssl sha256 /tmp/pynwb-$release.tar.gz | awk '{print $2}')
$ sed -i "3s/.*/{$ set sha256 = \"$sha\" %}/" recipe/meta.yaml
```

   For macOS:

```
$ sed -i -- "2s/.*/{% set version = \"$release\" %}/" recipe/meta.yaml
$ sha=$(openssl sha256 /tmp/pynwb-$release.tar.gz | awk '{print $2}')
$ sed -i -- "3s/.*/{$ set sha256 = \"$sha\" %}/" recipe/meta.yaml
```

7. Push the changes

```
$ git push origin $release
```

8. Create a Pull Request

   Create a pull request against the main repository. If the tests are passed a new release will be published on Anaconda cloud.

CHAPTER 17

How to Update Requirements Files

The different requirements files introduced in *Software Process* section are the following:

- requirements.txt
- requirements-dev.txt
- requirements-doc.txt

17.1 requirements.txt

Requirements.txt of the project can be created or updated and then captured using the following script:

```
mkvirtualenv pynwb-requirements

cd pynwb
pip install .
pip check # check for package conflicts
pip freeze > requirements.txt

deactivate
rmvirtualenv pynwb-requirements
```

17.2 requirements-(dev|doc).txt

Any of these requirements files can be updated using the following scripts:

```
cd pynwb

# Set the requirements file to update
target_requirements=requirements-dev.txt

```

(continues on next page)

(continued from previous page)

```
mkvirtualenv pynwb-requirements

# Install updated requirements
pip install -U -r $target_requirements

# If relevant, you could pip install new requirements now
# pip install -U <name-of-new-requirement>

# Check for any conflicts in installed packages
pip check

# Update list of pinned requirements
pip freeze > $target_requirements

deactivate
rmvirtualenv pynwb-requirements
```

CHAPTER 18

Copyright

CHAPTER 19

License

CHAPTER 20

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

G

I

K

L

M

N

O

P

R

S

T

U

V

W

X