
# PyNWB

***Release 2.6.0***

**unknown**

**Apr 25, 2024**

# 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. If you are new to NWB and would like to learn more, then please also visit the NWB Overview website, which provides an entry point for researchers and developers interested in using NWB.

Neurodata Without Borders (NWB) 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 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

ONE

# INSTALLING PYNWB

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

1. Python 3.8, 3.9, 3.10, or 3.11
2. pip

**Note:** If you are a developer then please see the *Installing PyNWB for Developers* installation instructions instead.

## 1.1 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. pandas
5. python-dateutil

## 1.2 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
```

## 1.3 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

TWO

# TUTORIALS

## 2.1 General tutorials

### 2.1.1 NWB File Basics

This example will focus on the basics of working with an *NWBFile* object, including writing and reading of an NWB file, and giving you an introduction to the basic data types. Before we dive into code showing how to use an *NWBFile*, we first provide a brief overview of the basic concepts of NWB. If you are already familiar with the concepts of *TimeSeries* and *Processing Modules*, then feel free to skip the *Background: Basic concepts* part and go directly to *The NWB file*.

#### Background: Basic concepts

In the NWB Format, each experiment session is typically represented by a separate NWB file. NWB files are represented in PyNWB by *NWBFile* objects which provide functionality for creating and retrieving:

- *TimeSeries* datasets, i.e., objects for storing time series data
- *Processing Modules*, i.e., objects for storing and grouping analyses, and
- experiment metadata and other metadata related to data provenance.

The following sections describe the *TimeSeries* and *ProcessingModule* classes in further detail.

#### TimeSeries

*TimeSeries* objects store time series data and correspond to the *TimeSeries* specifications provided by the NWB Format. Like the NWB specification, *TimeSeries* Python objects follow an object-oriented inheritance pattern, i.e., the class *TimeSeries* serves as the base class for all other *TimeSeries* types, such as, *ElectricalSeries*, which itself may have further subtypes, e.g., *SpikeEventSeries*.

**See also:**

For your reference, NWB defines the following main *TimeSeries* subtypes:

- **Extracellular electrophysiology:** *ElectricalSeries*, *SpikeEventSeries*
- **Intracellular electrophysiology:** *PatchClampSeries* is the base type for all intracellular time series, which is further refined into subtypes depending on the type of recording: *CurrentClampSeries*, *IZeroClampSeries*, *CurrentClampStimulusSeries*, *VoltageClampSeries*, *VoltageClampStimulusSeries*.
- **Optical physiology and imaging:** *ImageSeries* is the base type for image recordings and is further refined by the *ImageMaskSeries*, *OpticalSeries*, and *TwoPhotonSeries* types. Other related time series types are: *IndexSeries* and *RoiResponseSeries*.

- **Others** *OptogeneticSeries*, *SpatialSeries*, *DecompositionSeries*, *AnnotationSeries*, *AbstractFeatureSeries*, and *IntervalSeries*.

### 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.

**See also:**

For your reference, NWB defines the following main analysis *NWBDataInterface* subtypes:

- **Behavior:** *BehavioralEpochs*, *BehavioralEvents*, *BehavioralTimeSeries*, *CompassDirection*, *PupilTracking*, *Position*, *EyeTracking*.
- **Extracellular electrophysiology:** *EventDetection*, *EventWaveform*, *FeatureExtraction*, *FilteredEphys*, *LFP*.
- **Optical physiology:** *DfOverF*, *Fluorescence*, *ImageSegmentation*, *MotionCorrection*.
- **Others:** *Images*.
- **TimeSeries:** Any *TimeSeries* is also a subclass of *NWBDataInterface* and can 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 format.

---

NWB organizes data into different groups depending on the type of data. Groups can be thought of as folders within the file. Here are some of the groups within an *NWBFile* and the types of data they are intended to store:

- **acquisition**: raw, acquired data that should never change
- **processing**: processed data, typically the results of preprocessing algorithms and could change
- **analysis**: results of data analysis
- **stimuli**: stimuli used in the experiment (e.g., images, videos, light pulses)

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 uuid import uuid4

import numpy as np
from dateutil import tz

from pynwb import NWBHDF5IO, NWBFile, TimeSeries
from pynwb.behavior import Position, SpatialSeries
from pynwb.epoch import TimeIntervals
from pynwb.file import Subject
```

### The NWB file

An *NWBFile* represents a single session of an experiment. Each *NWBFile* must have a session description, identifier, and session start time. Importantly, the session start time is the reference time for all timestamps in the file. For instance, an event with a timestamp of 0 in the file means the event occurred exactly at the session start time.

Create an *NWBFile* object with the required fields (*session_description*, *identifier*, *session_start_time*) and additional metadata.

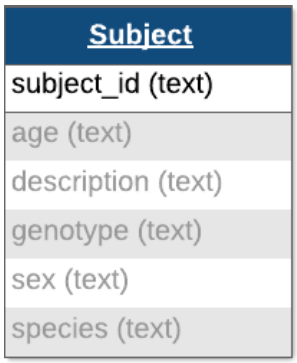
**Note:** Use keyword arguments when constructing *NWBFile* objects.

```
session_start_time = datetime(2018, 4, 25, hour=2, minute=30, second=3, tzinfo=tz.gettz(
↪"US/Pacific"))

nwbfile = NWBFile(
    session_description="Mouse exploring an open field",  # required
    identifier=str(uuid4()),  # required
    session_start_time=session_start_time,  # required
    session_id="session_1234",  # optional
    experimenter=[
        "Baggins, Bilbo",
    ],  # optional
    lab="Bag End Laboratory",  # optional
    institution="University of My Institution",  # optional
    experiment_description="I went on an adventure to reclaim vast treasures.",  #␣
↪optional
    related_publications="DOI:10.1016/j.neuron.2016.12.011",  # optional
)
nwbfile
```

**Note:** See the NWBFile Best Practices for detailed information about the arguments to *NWBFile*

### Subject Information

In the *Subject* object we can store information about the experiment subject, such as `age`, `species`, `genotype`, `sex`, and a `description`.




The fields in the *Subject* object are all free-form text (any format will be valid), however it is recommended to follow particular conventions to help software tools interpret the data:

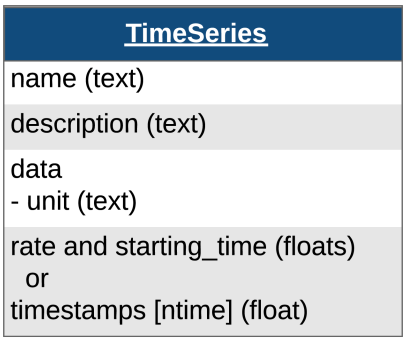
- **age**: ISO 8601 Duration format, e.g., `"P90D"` for 90 days old
- **species**: The formal Latin binomial nomenclature, e.g., `"Mus musculus"`, `"Homo sapiens"`
- **sex**: Single letter abbreviation, e.g., `"F"` (female), `"M"` (male), `"U"` (unknown), and `"O"` (other)

Add the subject information to the *NWBFile* by setting the `subject` field to the new *Subject* object.

```
subject = Subject(
    subject_id="001",
    age="P90D",
    description="mouse 5",
    species="Mus musculus",
    sex="M",
)

nwbfile.subject = subject
subject
```

### Time Series Data

*TimeSeries* is a common base class for measurements sampled over time, and provides fields for `data` and `timestamps` (regularly or irregularly sampled). You will also need to supply the `name` and `unit` of measurement (SI unit).




For instance, we can store a *TimeSeries* data where recording started `0.0` seconds after `start_time` and sampled every second:

```
data = list(range(100, 200, 10))
time_series_with_rate = TimeSeries(
    name="test_timeseries",
    data=data,
    unit="m",
    starting_time=0.0,
    rate=1.0,
)
time_series_with_rate
```

For irregularly sampled recordings, we need to provide the `timestamps` for the `data`:

```
timestamps = list(range(10))
time_series_with_timestamps = TimeSeries(
    name="test_timeseries",
    data=data,
    unit="m",
    timestamps=timestamps,
)
time_series_with_timestamps
```

*TimeSeries* objects can be added directly to *NWBFile* using:

- *NWBFile.add_acquisition* to add *acquisition* data (raw, acquired data that should never change),
- *NWBFile.add_stimulus* to add *stimulus* data, or
- *NWBFile.add_stimulus_template* to store *stimulus templates*.

```
nwbfile.add_acquisition(time_series_with_timestamps)
```

We can access the *TimeSeries* object `'test_timeseries'` in *NWBFile* from `acquisition`:

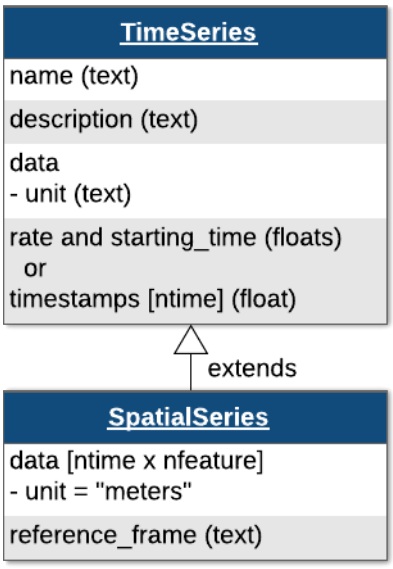
```
nwbfile.acquisition["test_timeseries"]
```

or using the method *NWBFile.get_acquisition*:

```
nwbfile.get_acquisition("test_timeseries")
```

### Spatial Series and Position

*SpatialSeries* is a subclass of *TimeSeries* that represents the spatial position of an animal over time.




Create a *SpatialSeries* object named `"SpatialSeries"` with some fake data.

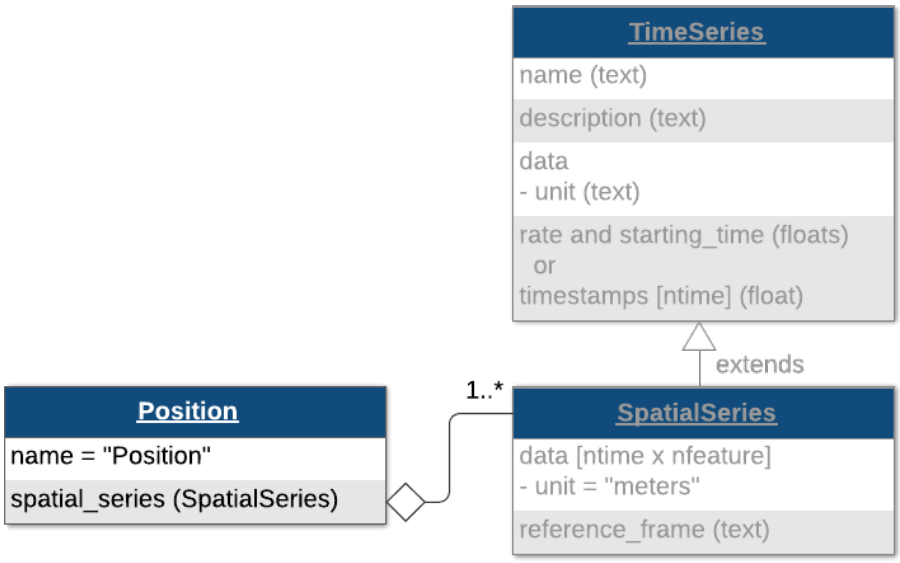
```
# create fake data with shape (50, 2)
# the first dimension should always represent time
position_data = np.array([np.linspace(0, 10, 50), np.linspace(0, 8, 50)]).T
```

(continues on next page)

(continued from previous page)

```
position_timestamps = np.linspace(0, 50).astype(float) / 200

spatial_series_obj = SpatialSeries(
    name="SpatialSeries",
    description="(x,y) position in open field",
    data=position_data,
    timestamps=position_timestamps,
    reference_frame="(0,0) is bottom left corner",
)
spatial_series_obj
```

To help data analysis and visualization tools know that this *SpatialSeries* object represents the position of the subject, store the *SpatialSeries* object inside of a *Position* object, which can hold one or more *SpatialSeries* objects.




Create a *Position* object named "Position"[1].

```
# name is set to "Position" by default
position_obj = Position(spatial_series=spatial_series_obj)
position_obj
```

### Behavior Processing Module

*ProcessingModule* is a container for data interfaces that are related to a particular processing workflow. NWB differentiates between raw, acquired data (*acquisition*), which should never change, and processed data (*processing*), which are the results of preprocessing algorithms and could change. Processing modules can be thought of as folders within the file for storing the related processed data.

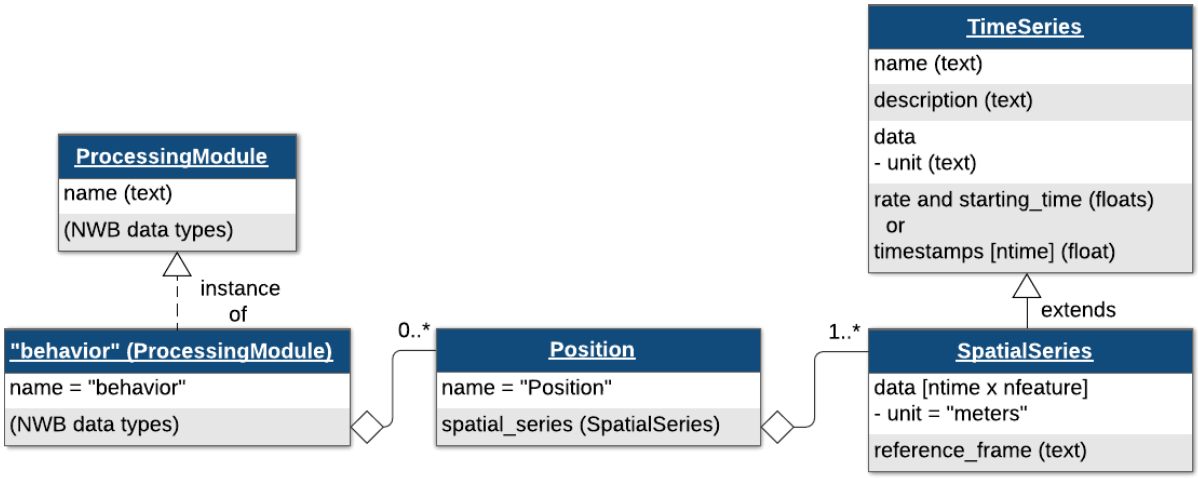
**Tip:** Use the NWB schema module names as processing module names where appropriate. These are: "behavior", "ecephys", "icephys", "ophys", "ogen", and "misc".

[1] 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".

Let's assume that the subject's position was computed from a video tracking algorithm, so it would be classified as processed data.

Create a processing module called `"behavior"` for storing behavioral data in the *NWBFile* and add the *Position* object to the processing module using the method *NWBFile.create_processing_module*:

```
behavior_module = nwbfile.create_processing_module(
    name="behavior", description="processed behavioral data"
)
behavior_module.add(position_obj)
behavior_module
```




Once the behavior processing module is added to the *NWBFile*, you can access it with:

```
nwbfile.processing["behavior"]
```

### Writing an NWB file

NWB I/O is carried out using the *NWBHDF5IO* class[2]. This class is responsible for mapping an *NWBFile* object into HDF5 according to the NWB schema.

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

```
io = NWBHDF5IO("basics_tutorial.nwb", mode="w")
io.write(nwbfile)
io.close()
```

```
/home/docs/checkouts/readthedocs.org/user_builds/pynwb/envs/dev/lib/python3.11/site-
↪packages/hdmf/build/objectmapper.py:260: DtypeConversionWarning: Spec 'TimeSeries/
↪timestamps': Value with data type int64 is being converted to data type float64 as␣
↪specified.
  warnings.warn(full_warning_msg, DtypeConversionWarning)
```

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

```
with NWBHDF5IO("basics_tutorial.nwb", "w") as io:
    io.write(nwbfile)
```

[2] HDF5 is the primary backend supported by NWB.

### 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.

Data arrays are read passively from the file. Accessing the `data` attribute of the *TimeSeries* object does not read the data values, but presents an HDF5 object that can be indexed to read data. You can use the `[:]` operator to read the entire data array into memory.

```
with NWBHDF5IO("basics_tutorial.nwb", "r") as io:
    read_nwbfile = io.read()
    print(read_nwbfile.acquisition["test_timeseries"])
    print(read_nwbfile.acquisition["test_timeseries"].data[:])
```

```
test_timeseries pynwb.base.TimeSeries at 0x139808489841744
Fields:
  comments: no comments
  conversion: 1.0
  data: <HDF5 dataset "data": shape (10,), type "<i8">
  description: no description
  interval: 1
  offset: 0.0
  resolution: -1.0
  timestamps: <HDF5 dataset "timestamps": shape (10,), type "<f8">
  timestamps_unit: seconds
  unit: m

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

It is often preferable to read only a portion of the data. To do this, index or slice into the `data` attribute just like you index or slice a numpy array.

```
with NWBHDF5IO("basics_tutorial.nwb", "r") as io:
    read_nwbfile = io.read()
    print(read_nwbfile.acquisition["test_timeseries"].data[:2])
```

```
[100 110]
```

**Note:** 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[3].

[3] 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.

### Accessing data

We can also access the *SpatialSeries* data by referencing the names of the objects in the hierarchy that contain it. We can access a processing module by indexing `nwbfile.processing` with the name of the processing module, `"behavior"`.

Then, we can access the *Position* object inside of the `"behavior"` processing module by indexing it with the name of the *Position* object, `"Position"`.

Finally, we can access the *SpatialSeries* object inside of the *Position* object by indexing it with the name of the *SpatialSeries* object, `"SpatialSeries"`.

```
with NWBHDF5IO("basics_tutorial.nwb", "r") as io:
    read_nwbfile = io.read()
    print(read_nwbfile.processing["behavior"])
    print(read_nwbfile.processing["behavior"]["Position"])
    print(read_nwbfile.processing["behavior"]["Position"]["SpatialSeries"])
```

```
behavior pynwb.base.ProcessingModule at 0x139808487972624
Fields:
  data_interfaces: {
    Position <class 'pynwb.behavior.Position'>
  }
  description: processed behavioral data

Position pynwb.behavior.Position at 0x139808489070480
Fields:
  spatial_series: {
    SpatialSeries <class 'pynwb.behavior.SpatialSeries'>
  }

SpatialSeries pynwb.behavior.SpatialSeries at 0x139808489068112
Fields:
  comments: no comments
  conversion: 1.0
  data: <HDF5 dataset "data": shape (50, 2), type "<f8">
  description: (x,y) position in open field
  interval: 1
  offset: 0.0
  reference_frame: (0,0) is bottom left corner
  resolution: -1.0
  timestamps: <HDF5 dataset "timestamps": shape (50,), type "<f8">
  timestamps_unit: seconds
  unit: meters
```

### 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*, pass the existing *TimeSeries* as the new *TimeSeries* timestamps:

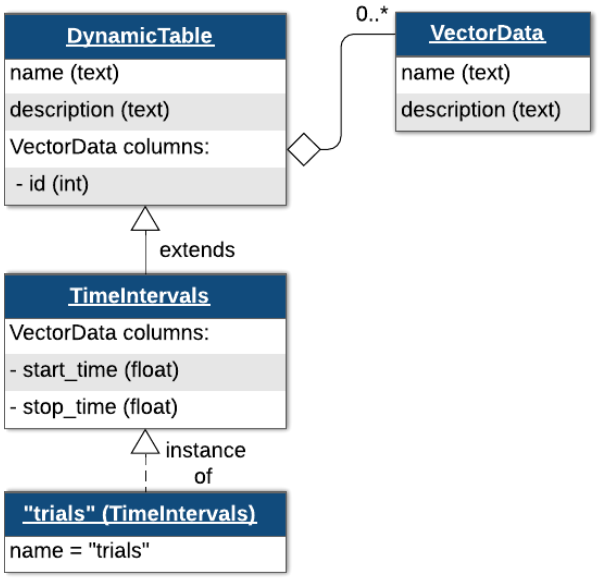
```
data = list(range(101, 201, 10))
reuse_ts = TimeSeries(
    name="reusing_timeseries",
    data=data,
    unit="SIunit",
    timestamps=time_series_with_timestamps,
)
```

### Time Intervals

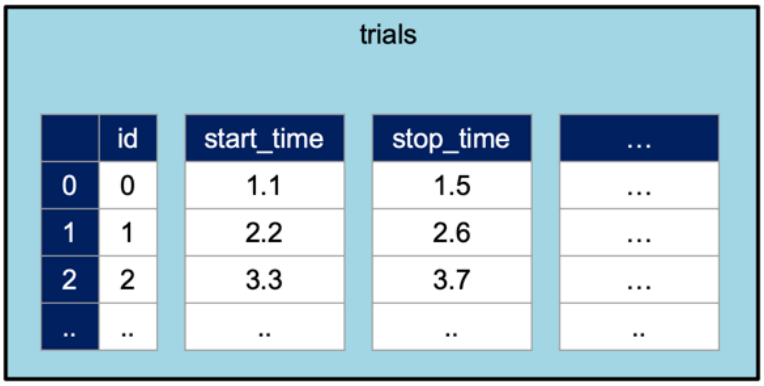
The following provides a brief introduction to managing annotations in time via *TimeIntervals*. See the *Annotating Time Intervals* tutorial for a more detailed introduction to *TimeIntervals*.

### Trials

Trials are stored in *TimeIntervals*, which is a subclass of DynamicTable. DynamicTable is used to store tabular metadata throughout NWB, including trials, electrodes and sorted units. This class offers flexibility for tabular data by allowing required columns, optional columns, and custom columns which are not defined in the standard.




The `trials` *TimeIntervals* class can be thought of as a table with this structure:




By default, *TimeIntervals* objects only require `start_time` and `stop_time` of each trial. Additional columns can be added using the method *NWBFile.add_trial_column*. When all the desired custom columns have been defined, use the *NWBFile.add_trial* method to add each row. In this case, we will add one custom column to the trials table named "correct" which will take a boolean array, then add two trials as rows of the table.

```
nwbfile.add_trial_column(
    name="correct",
    description="whether the trial was correct",
)
nwbfile.add_trial(start_time=1.0, stop_time=5.0, correct=True)
nwbfile.add_trial(start_time=6.0, stop_time=10.0, correct=False)
```

DynamicTable and its subclasses can be converted to a pandas DataFrame for convenient analysis using to_dataframe.

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

### Epochs

Like trials, epochs can be added to an NWB file using the methods *NWBFile.add_epoch_column* and *NWBFile.add_epoch*. The third argument is one or more tags for labeling the epoch, and the fourth argument is a list of all the *TimeSeries* that the epoch applies to.

```
nwbfile.add_epoch(
    start_time=2.0,
    stop_time=4.0,
    tags=["first", "example"],
    timeseries=[time_series_with_timestamps],
)

nwbfile.add_epoch(
    start_time=6.0,
    stop_time=8.0,
    tags=["second", "example"],
    timeseries=[time_series_with_timestamps],
)

nwbfile.epochs.to_dataframe()
```

### Other time intervals

These *TimeIntervals* objects are stored in `NWBFile.intervals`. In addition to the default `epochs` and `trials`, you can also add your own with custom names.

```
sleep_stages = TimeIntervals(
    name="sleep_stages",
    description="intervals for each sleep stage as determined by EEG",
)

sleep_stages.add_column(name="stage", description="stage of sleep")
sleep_stages.add_column(name="confidence", description="confidence in stage (0-1)")

sleep_stages.add_row(start_time=0.3, stop_time=0.5, stage=1, confidence=0.5)
sleep_stages.add_row(start_time=0.7, stop_time=0.9, stage=2, confidence=0.99)
sleep_stages.add_row(start_time=1.3, stop_time=3.0, stage=3, confidence=0.7)

nwbfile.add_time_intervals(sleep_stages)

sleep_stages.to_dataframe()
```

Now we overwrite the file with all of the data

```
with NWBHDF5IO("basics_tutorial.nwb", "w") as io:
    io.write(nwbfile)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/pynwb/envs/dev/lib/python3.11/site-
↪packages/hdmf/build/objectmapper.py:260: DtypeConversionWarning: Spec 'TimeSeries/
↪timestamps': Value with data type int64 is being converted to data type float64 as␣
↪specified.
  warnings.warn(full_warning_msg, DtypeConversionWarning)
```

### Appending to an NWB file

To append to a file, read it with *NWBHDF5IO* and set the `mode` argument 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 acquisition.

```
io = NWBHDF5IO("basics_tutorial.nwb", mode="a")
nwbfile = io.read()

new_time_series = TimeSeries(
    name="new_time_series",
    data=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    timestamps=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    unit="n.a.",
)
nwbfile.add_acquisition(new_time_series)
```

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

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

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

### 2.1.2 Adding/Removing Containers from an NWB File

This tutorial explains how to add and remove containers from an existing NWB file and either write the data back to the same file or export the data to a new file.

#### Adding objects to an NWB file in read/write mode

PyNWB supports adding container objects to an existing NWB file - that is, reading data from an NWB file, adding a container object, such as a new *TimeSeries* object, and writing the modified *NWBFile* back to the same file path on disk. To do so:

1. open the file with an *NWBHDF5IO* object in read/write mode (`mode='r+'` or `mode='a'`)
2. read the *NWBFile*
3. add container objects to the *NWBFile* object
4. write the modified *NWBFile* using the same *NWBHDF5IO* object

For example:

```
import datetime

import numpy as np

from pynwb import NWBHDF5IO, NWBFile, TimeSeries

# first, write a test NWB file
nwbfile = NWBFile(
    session_description="demonstrate adding to an NWB file",
    identifier="NWB123",
    session_start_time=datetime.datetime.now(),
)

filename = "nwbfile.nwb"
with NWBHDF5IO(filename, "w") as io:
    io.write(nwbfile)

# open the NWB file in r+ mode
with NWBHDF5IO(filename, "r+") as io:
    read_nwbfile = io.read()

    # create a TimeSeries and add it to the file under the acquisition group
    data = list(range(100, 200, 10))
    timestamps = np.arange(10, dtype=float)
    test_ts = TimeSeries(
        name="test_timeseries", data=data, unit="m", timestamps=timestamps
    )
    read_nwbfile.add_acquisition(test_ts)
```

(continues on next page)

(continued from previous page)

```
    # write the modified NWB file
    io.write(read_nwbfile)

# confirm the file contains the new TimeSeries in acquisition
with NWBHDF5IO(filename, "r") as io:
    read_nwbfile = io.read()
    print(read_nwbfile)
```

**Note:** You cannot remove objects from an NWB file using the above method.

Modifying an NWB file in this way has limitations. The destination file path must be the same as the source file path, and it is not possible to remove objects from an NWB file. You can use the *NWBHDF5IO.export* method, detailed below, to modify an NWB file in these ways.

### Exporting a written NWB file to a new file path

Use the *NWBHDF5IO.export* method to read data from an existing NWB file, modify the data, and write the modified data to a new file path. Modifications to the data can be additions or removals of objects, such as *TimeSeries* objects. This is especially useful if you have raw data and processed data in the same NWB file and you want to create a new NWB file with all the contents of the original file except for the raw data for sharing with collaborators.

To remove existing containers, use the pop method on any LabelledDict object, such as NWBFile.acquisition, NWBFile.processing, NWBFile.analysis, NWBFile.processing, NWBFile.scratch, NWBFile.devices, NWBFile.stimulus, NWBFile.stimulus_template, NWBFile.electrode_groups, NWBFile.imaging_planes, NWBFile.icephys_electrodes, NWBFile.ogen_sites, NWBFile.lab_meta_data, and *ProcessingModule* objects.

For example:

```
# first, create a test NWB file with a TimeSeries in the acquisition group
nwbfile = NWBFile(
    session_description="demonstrate export of an NWB file",
    identifier="NWB123",
    session_start_time=datetime.datetime.now(),
)
data1 = list(range(100, 200, 10))
timestamps1 = np.arange(10, dtype=float)
test_ts1 = TimeSeries(
    name="test_timeseries1", data=data1, unit="m", timestamps=timestamps1
)
nwbfile.add_acquisition(test_ts1)

# then, create a processing module for processed behavioral data
nwbfile.create_processing_module(
    name="behavior", description="processed behavioral data"
)
data2 = list(range(100, 200, 10))
timestamps2 = np.arange(10, dtype=float)
test_ts2 = TimeSeries(
    name="test_timeseries2", data=data2, unit="m", timestamps=timestamps2
)
```

(continues on next page)

(continued from previous page)

```
nwbfile.processing["behavior"].add(test_ts2)

# write these objects to an NWB file
filename = "nwbfile.nwb"
with NWBHDF5IO(filename, "w") as io:
    io.write(nwbfile)

# read the written file
export_filename = "exported_nwbfile.nwb"
with NWBHDF5IO(filename, mode="r") as read_io:
    read_nwbfile = read_io.read()

    # add a new TimeSeries to the behavior processing module
    data3 = list(range(100, 200, 10))
    timestamps3 = np.arange(10, dtype=float)
    test_ts3 = TimeSeries(
        name="test_timeseries3", data=data3, unit="m", timestamps=timestamps3
    )
    read_nwbfile.processing["behavior"].add(test_ts3)

    # use the pop method to remove the original TimeSeries from the acquisition group
    read_nwbfile.acquisition.pop("test_timeseries1")

    # use the pop method to remove a TimeSeries from a processing module
    read_nwbfile.processing["behavior"].data_interfaces.pop("test_timeseries2")

    # call the export method to write the modified NWBFile instance to a new file path.
    # the original file is not modified
    with NWBHDF5IO(export_filename, mode="w") as export_io:
        export_io.export(src_io=read_io, nwbfile=read_nwbfile)

# confirm the exported file does not contain TimeSeries with names 'test_timeseries1' or
↪'test_timeseries2'
# but does contain a new TimeSeries in processing['behavior'] with name 'test_timeseries3'
with NWBHDF5IO(export_filename, "r") as io:
    read_nwbfile = io.read()
    print(read_nwbfile)
    print(read_nwbfile.processing["behavior"])
```

**Note:** *TimeIntervals* objects, such as `NWBFile.epochs`, `NWBFile.trials`, `NWBFile.invalid_times`, and custom *TimeIntervals* objects cannot be removed (popped) from `NWBFile.intervals`.

**Warning:** Removing an object from an NWBFile may break links and references within the file and across files. This is analogous to having shortcuts/aliases to a file on your filesystem and then deleting the file. Extra caution should be taken when removing heavily referenced items such as *Device* objects, *ElectrodeGroup* objects, the electrodes table, and the *PlaneSegmentation* table.

#### Exporting with new object IDs

When exporting a read NWB file to a new file path, the object IDs within the original NWB file will be copied to the new file. To make the exported NWB file contain a new set of object IDs, call `generate_new_id` on your *NWBFile* object. This will generate a new object ID for the *NWBFile* object and all of the objects within the NWB file.

```
export_filename = "exported_nwbfile.nwb"
with NWBHDF5IO(filename, mode="r") as read_io:
    read_nwbfile = read_io.read()
    read_nwbfile.generate_new_id()

    with NWBHDF5IO(export_filename, mode="w") as export_io:
        export_io.export(src_io=read_io, nwbfile=read_nwbfile)
```

For more information about the export functionality, see *Exporting NWB files* and the PyNWB documentation for *NWBHDF5IO.export*.

For more information about editing a file in place, see *Editing NWB files*.

### 2.1.3 Annotating Time Intervals

Annotating events in time is a common need in neuroscience, e.g. to describes epochs, trials, and invalid times during an experimental session. NWB supports annotation of time intervals via the *TimeIntervals* type. The *TimeIntervals* type is a `DynamicTable` with the following columns:

1. `start_time` and `stop_time` describe the start and stop times of intervals as floating point offsets in seconds relative to the *timestamps_reference_time* of the file. In addition,
2. `tags` is an optional, indexed column used to associate user-defined string tags with intervals (0 or more tags per time interval)
3. `timeseries` is an optional, indexed *TimeSeriesReferenceVectorData* column to map intervals directly to ranges in select, relevant *TimeSeries* (0 or more per time interval)
4. as a `DynamicTable` user may add additional columns to *TimeIntervals* via `add_column`

---

**Hint:** *TimeIntervals* is intended for storing general annotations of time ranges. Depending on the application (e.g., when intervals are generated by data acquisition or automatic data processing), it can be useful to describe intervals (or instantaneous events) in time as *TimeSeries*. NWB provides several types for this purposes, e.g., *IntervalSeries*, *BehavioralEpochs*, *BehavioralEvents*, *EventDetection*, or *SpikeEventSeries*.

---

#### Setup: Creating an example NWB file for the tutorial

```
from datetime import datetime
import numpy as np
from pynwb import NWBFile, TimeSeries
from uuid import uuid4

# create the NWBFile
nwbfile = NWBFile(
    session_description="my first synthetic recording",  # required
    identifier=str(uuid4()),  # required
```

(continues on next page)

(continued from previous page)

```
    session_start_time=datetime(2017, 4, 3, hour=11),  # required
    experimenter="Baggins, Bilbo",  # optional
    lab="Bag End Laboratory",  # optional
    institution="University of Middle Earth at the Shire",  # optional
    experiment_description="I went on an adventure with thirteen dwarves to reclaim vast␣
↪treasures.",  # optional
    session_id="LONELYMTN",  # optional
)
# create some example TimeSeries
test_ts = TimeSeries(
    name="series1",
    data=np.arange(1000),
    unit="m",
    timestamps=np.linspace(0.5, 601, 1000),
)
rate_ts = TimeSeries(
    name="series2", data=np.arange(600), unit="V", starting_time=0.0, rate=1.0
)
# Add the TimeSeries to the file
nwbfile.add_acquisition(test_ts)
nwbfile.add_acquisition(rate_ts)
```

### Adding Time Intervals to a NWBFile

NWB provides a set of pre-defined *TimeIntervals* tables for *epochs*, *trials*, and *invalid_times*.

### Trials

Trials can be added to an NWB file using the methods *add_trial* By default, NWBFile only requires trial `start_time` and `stop_time`. The `tags` and `timeseries` are optional. For `timeseries` we only need to supply the *TimeSeries*. PyNWB automatically calculates the corresponding index range (described by `idx_start` and `count`) for the supplied *TimeSeries* based on the given `start_time` and `stop_time` and the *timestamps* (or *starting_time* and *rate*) of the given *TimeSeries*.

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 with tags and timeseries references.

```
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="dog",
    tags=["animal"],
    timeseries=[test_ts, rate_ts],
)
nwbfile.add_trial(
    start_time=3.0,
```

(continues on next page)

(continued from previous page)

```
    stop_time=5.0,
    stim="mountain",
    tags=["landscape"],
    timeseries=[test_ts, rate_ts],
)
nwbfile.add_trial(
    start_time=6.0,
    stop_time=8.0,
    stim="desert",
    tags=["landscape"],
    timeseries=[test_ts, rate_ts],
)
nwbfile.add_trial(
    start_time=9.0,
    stop_time=11.0,
    stim="tree",
    tags=["landscape", "plant"],
    timeseries=[test_ts, rate_ts],
)
nwbfile.add_trial(
    start_time=12.0,
    stop_time=14.0,
    stim="bird",
    tags=["animal"],
    timeseries=[test_ts, rate_ts],
)
nwbfile.add_trial(
    start_time=15.0,
    stop_time=17.0,
    stim="flower",
    tags=["animal"],
    timeseries=[test_ts, rate_ts],
)
```

### Epochs

Similarly, epochs can be added to an NWB file using the method *add_epoch* and *add_epoch_column*.

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

(continues on next page)

(continued from previous page)

```
        test_ts,
    ],
)
```

### Invalid Times

Similarly, invalid times can be added using the method *add_invalid_time_interval* and *add_invalid_times_column*.

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

### Custom Time Intervals

To define custom, experiment-specific *TimeIntervals* we can add them either: 1) when creating the *NWBFile* by defining the `intervals` constructor argument or 2) via the *add_time_intervals* or *create_time_intervals* after the *NWBFile* has been created.

```
from pynwb.epoch import TimeIntervals

sleep_stages = TimeIntervals(
    name="sleep_stages",
    description="intervals for each sleep stage as determined by EEG",
)

sleep_stages.add_column(name="stage", description="stage of sleep")
sleep_stages.add_column(name="confidence", description="confidence in stage (0-1)")

sleep_stages.add_row(start_time=0.3, stop_time=0.5, stage=1, confidence=0.5)
sleep_stages.add_row(start_time=0.7, stop_time=0.9, stage=2, confidence=0.99)
sleep_stages.add_row(start_time=1.3, stop_time=3.0, stage=3, confidence=0.7)

_ = nwbfile.add_time_intervals(sleep_stages)
```

### Accessing Time Intervals

We can access the predefined *TimeIntervals* tables via the corresponding *epochs*, *trials*, and *invalid_times* properties and for custom *TimeIntervals* via the *get_time_intervals* method. E.g.:

```
_ = nwbfile.intervals
_ = nwbfile.get_time_intervals("sleep_stages")
```

Like any DynamicTable, we can conveniently convert any *TimeIntervals* table to a pandas.DataFrame via to_dataframe, such as:

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

This approach makes it easy to query the data to, e.g., locate all time intervals within a certain time range

```
trials_df = nwbfile.trials.to_dataframe()
trials_df.query("(start_time > 2.0) & (stop_time < 9.0)")
```

### Accessing referenced TimeSeries

As mentioned earlier, the timeseries column is defined by a *TimeSeriesReferenceVectorData* which stores references to the corresponding ranges in *TimeSeries*. Individual references to *TimeSeries* are described via *TimeSeriesReference* tuples with the *idx_start*, *count*, and *timeseries*. Using *TimeSeriesReference* we can easily access the relevant *data* and *timestamps* for the corresponding time range from the *TimeSeries*.

```
# Get a single example TimeSeriesReference from the trials table
example_tsr = nwbfile.trials["timeseries"][0][0]

# Get the data values from the timeseries. This is a shorthand for:
# _ = example_tsr.timeseries.data[example_tsr.idx_start: (example_tsr.idx_start +␣
↪example_tsr.count)]
_ = example_tsr.data

# Get the timestamps. Timestamps are either loaded from the TimeSeries or
# computed from the starting_time and rate
example_tsr.timestamps
```

```
array([0.5      , 1.1011011, 1.7022022])
```

Using *isvalid* we can further check if the reference is valid. A *TimeSeriesReference* is defined as invalid if both *idx_start*, *count* are set to -1. *isvalid* further also checks that the indicated index range and types are valid, raising IndexError and TypeError respectively, if bad *idx_start*, *count* or *timeseries* are found.

```
example_tsr.isvalid()
```

```
True
```

#### Adding TimeSeries references to other tables

Since *TimeSeriesReferenceVectorData* is a regular VectorData type, we can use it to add references to intervals in *TimeSeries* to any DynamicTable. In the *IntracellularRecordingsTable*, e.g., it is used to reference the recording of the stimulus and response associated with a particular intracellular electrophysiology recording.

#### Reading/Writing TimeIntervals to file

Reading and writing the data is as usual:

```
from pynwb import NWBHDF5IO

# write the file
with NWBHDF5IO("example_timeintervals_file.nwb", "w") as io:
    io.write(nwbfile)
# read the file
with NWBHDF5IO("example_timeintervals_file.nwb", "r") as io:
    nwbfile_in = io.read()

    # plot the sleep stages TimeIntervals table
    nwbfile_in.get_time_intervals("sleep_stages").to_dataframe()
```

### 2.1.4 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 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 datetime import datetime
import numpy as np
from pynwb import NWBHDF5IO, NWBFile, TimeSeries

# set up the NWBFile
start_time = datetime(2019, 4, 3, hour=11, minute=0)

nwb = NWBFile(
    session_description="demonstrate NWBFile scratch",  # required
    identifier="NWB456",  # required
    session_start_time=start_time,  # required
)
```

(continues on next page)

(continued from previous page)

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

```
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.[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",
    description="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")
```

[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.

```
# close the IO objects
raw_io.close()
proc_io.close()
scratch_io.close()
```

### 2.1.5 Extending NWB

The NWB format was designed to be easily extendable. Here we discuss some of the basic functionality in PyNWB for creating Neurodata Extensions (NDX).

**See also:**

For a more in-depth, step-by-step guide on how to create, document, and publish NWB extensions, we highly recommend visiting the extension tutorial on the nwb overview website.

#### 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 how to create an extension for subsequent use.

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 NWBAttributeSpec, NWBGroupSpec, NWBNamespaceBuilder

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

ns_builder = NWBNamespaceBuilder(
    "Extension for use in my Lab", "mylab", version="0.1.0"
)

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:
```

(continues on next page)

(continued from previous page)

```
  - 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 code 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 hdmf.utils import docval, get_docval, popargs

from pynwb import load_namespaces, register_class
from pynwb.ecephys import ElectricalSeries

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):
        trode_id = popargs("trode_id", kwargs)
        super().__init__(**kwargs)
        self.trode_id = trode_id
```

**Note:** See the API docs for more information about `docval`, `popargs`, 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 readability.

### 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 simulated data using our extensions.

```
from datetime import datetime
from pynwb import NWBFile
from uuid import uuid4

session_start_time = datetime(2017, 4, 3, hour=11, minute=0)

nwbfile = NWBFile(
    session_description="demonstrate caching",
    identifier=str(uuid4()),
    session_start_time=session_start_time,
)

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,
```

(continues on next page)

(continued from previous page)

```
        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

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, if a namespace is not already loaded, PyNWB loads the namespace cached in the file. To disable this, set `load_namespaces=False` in the *NWBHDF5IO* constructor.

### 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 (*Potato*) and then how to use the new data types. First, we use *pynwb* to define the new data types.

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

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

ns_builder = NWBNamespaceBuilder(name + " extensions", name, version="0.1.0")
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 load_namespaces, register_class
from pynwb.file import MultiContainerInterface, NWBContainer

load_namespaces(ns_path)


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

(continues on next page)

(continued from previous page)

```
    @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().__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 datetime import datetime
from pynwb import NWBHDF5IO, NWBFile

# 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)

session_start_time = datetime(2017, 4, 3, hour=12, minute=0)
nwbfile = NWBFile(
    session_description="a file with metadata",
    identifier=str(uuid4()),
    session_start_time = session_start_time,
)

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
with NWBHDF5IO("test_multicontainerinterface.nwb", "r") as io:
    nwb = io.read()
    print(nwb.get_processing_module()["potato_sack"].get_potato("big_potato").weight)
```

(continues on next page)

(continued from previous page)

```
# 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_potato
```

### 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, since we need to know where each electrode is with respect to the gyri and sulci. 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, NWBGroupSpec, NWBNamespaceBuilder

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
```

(continues on next page)

(continued from previous page)

```
ns_builder = NWBNamespaceBuilder(name + " extensions", name, version="0.1.0")
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
#
#     # 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 datetime import datetime

import numpy as np

from pynwb import NWBHDF5IO, NWBFile, get_class, load_namespaces

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],
```

(continues on next page)

(continued from previous page)

```
        [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)
```

### 2.1.6 Object IDs in NWB

This example focuses on how to access object IDs from NWB container objects and NWB container objects by object ID. Every NWB container object has an object ID that is a UUID string, such as "123e4567-e89b-12d3-a456-426655440000". These IDs have a non-zero probability of being duplicated, but are practically unique and used widely across computing platforms as if they are unique.

The object ID of an NWB container object can be accessed using the `object_id` method.

```
from datetime import datetime
import numpy as np
from pynwb import NWBFile, TimeSeries

# set up the NWBFile
start_time = datetime(2019, 4, 3, hour=11, minute=0)
nwbfile = NWBFile(
    session_description="demonstrate NWB object IDs",
    identifier="NWB456",
    session_start_time=start_time,
)

# make some simulated 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
nwbfile.add_acquisition(test_ts)
```

(continues on next page)

(continued from previous page)

```
# print the object ID of the NWB file
print(nwbfile.object_id)

# print the object ID of the TimeSeries
print(test_ts.object_id)
```

The *NWBFile* class has the *objects* property, which provides a dictionary of all neurodata_type objects in the *NWBFile*, indexed by each object's object ID.

```
print(nwbfile.objects)
```

You can iterate through the *objects* dictionary as with any other Python dictionary.

```
for oid in nwbfile.objects:
    print(nwbfile.objects[oid])

for obj in nwbfile.objects.values():
    print('%s: %s "%s"' % (obj.object_id, obj.neurodata_type, obj.name))
```

If you have stored the object ID of a particular NWB container object, you can use it as a key on *NWBFile.objects* to get the object.

```
ts_id = test_ts.object_id
my_ts = nwbfile.objects[ts_id]  # test_ts == my_ts
```

---

**Note:** It is important to note that the object ID is NOT a unique hash of the data. If the contents of an NWB container change, the object ID remains the same.

---

### 2.1.7 Reading and Exploring an NWB File

This tutorial will demonstrate how to read, explore, and do basic visualizations with an NWB File using different tools.

An *NWBFile* represents a single session of an experiment. It contains all the data of that session and the metadata required to understand the data.

We will demonstrate how to use the DANDI neurophysiology data archive to access the data in two different ways: (1) by downloading it to your computer and (2) streaming it.

We will briefly show tools for exploring NWB Files interactively and refer the reader to the NWB Overview documentation for more details about the available tools.

**See also:**

You can learn more about the *NWBFile* format in the *NWB File Basics* tutorial.

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 matplotlib.pyplot as plt
import numpy as np

from pynwb import NWBHDF5IO
```

We will access NWB data on the DANDI Archive, and demonstrate reading one session of an experiment by Chandravadia et al. (2020). In this study, the authors recorded single neuron activity from the medial temporal lobes of human subjects while they performed a recognition memory task.

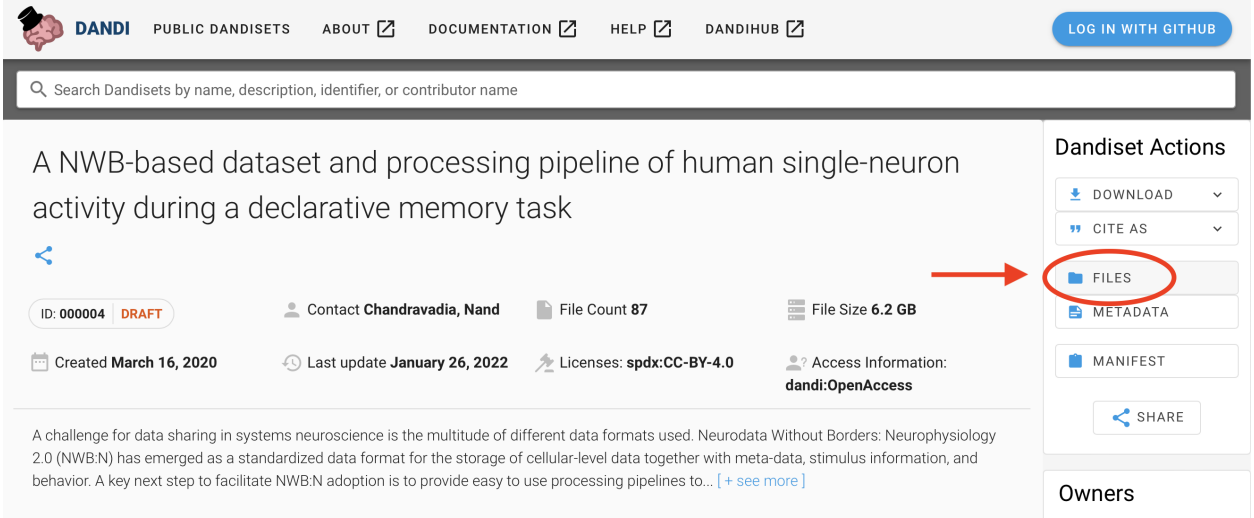
### Download the data

First, we will demonstrate how to download an NWB data file from DANDI to your machine.

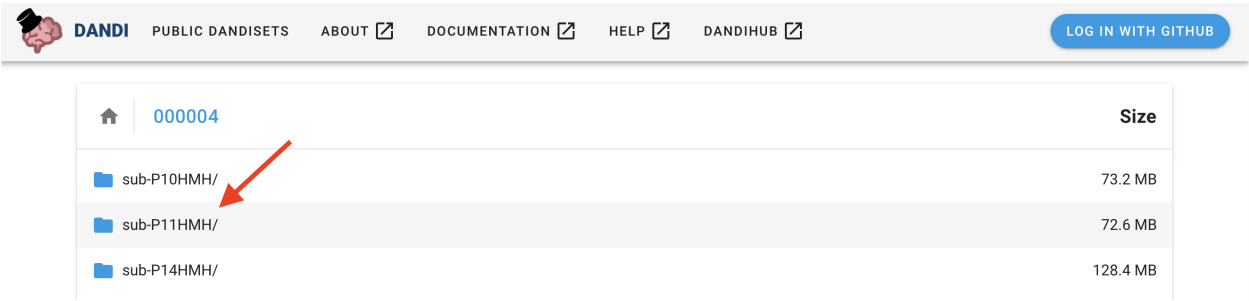
### Download using the DANDI Web UI

You can download files directly from the DANDI website.

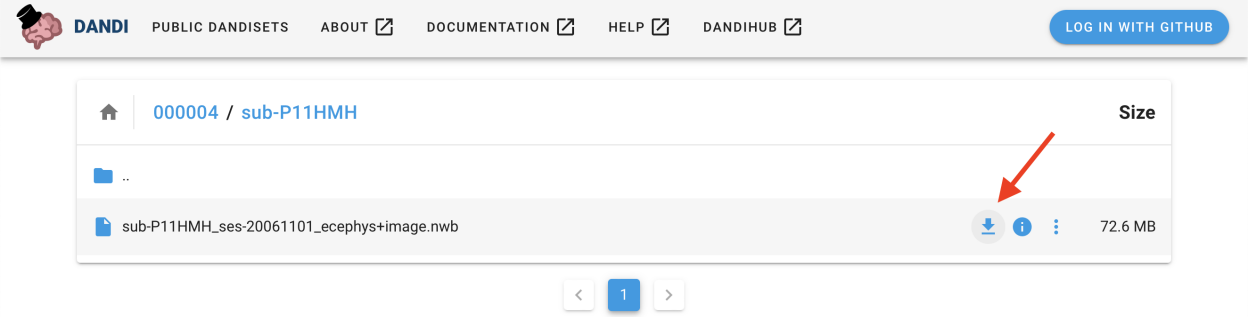
1. Go to the DANDI archive and open this dataset
2. List the files in this dataset by clicking the "Files" button in Dandiset Actions (top right column of the page).




3. Choose the folder "sub-P11MHM" by clicking on its name.




4. Download the NWB data file "sub-P11HMH_ses-20061101_ecephys+image.nwb" to your computer by clicking on the download symbol.




### Downloading data programmatically

Alternatively, you can download data using the *dandi* Python module.

```
from dandi.download import download

download("https://api.dandiarchive.org/api/assets/0f57f0b0-f021-42bb-8eaa-56cd482e2a29/
↪download/", ".")
```

```
PATH                                        SIZE     DONE           DONE% CHECKSUM STATUS  ␣
↪       MESSAGE
sub-P11HMH_ses-20061101_ecephys+image.nwb 72.6 MB 72.6 MB          100%    ok     done
Summary:                                    72.6 MB 72.6 MB                    1 done
                                                    100.00%
```

**See also:**

Learn about all the different ways you can download data from the DANDI Archive here

**See also:**

Streaming data

Instead of downloading data, another approach is to stream data directly from an archive. Streaming data allows you to download only the data you want from a file, so it can be a much better approach when the desired data files contain a lot of data you don't care about. There are several approaches to streaming NWB files, outlined in *Streaming NWB files*.

### Opening an NWB file with NWBHDF5IO

Reading and writing NWB data is carried out using the *NWBHDF5IO* class. *NWBHDF5IO* reads NWB data that is in the HDF5 storage format, a popular, hierarchical format for storing large-scale scientific data.

The first argument to the constructor of *NWBHDF5IO* is the `file_path`. Use the `read` method to read the data into a *NWBFile* object.

```
filepath = "sub-P11HMH_ses-20061101_ecephys+image.nwb"
# Open the file in read mode "r",
io = NWBHDF5IO(filepath, mode="r")
nwbfile = io.read()
nwbfile
```

*NWBHDF5IO* can also be used as a context manager:

```
with NWBHDF5IO(filepath, mode="r") as io2:
    nwbfile2 = io2.read()

    # data accessible here

# data not accessible here
```

The advantage of using a context manager is that the file is closed automatically when the context finishes successfully or if there is an error. Be aware that if you use this method, closing the context (unindenting the code) will automatically close the *NWBHDF5IO* object and the corresponding h5py File object. The data not already read from the NWB file will then be inaccessible, so any code that reads data must be placed within the context.

### Access stimulus data

Data representing stimuli that were presented to the experimental subject are stored in *stimulus* within the *NWBFile* object.

```
nwbfile.stimulus
```

```
{'StimulusPresentation': StimulusPresentation pynwb.image.OpticalSeries at␣
↪0x139808441838096
Fields:
  comments: no comments
  conversion: 1.0
  data: <HDF5 dataset "data": shape (200, 400, 300, 3), type "|u1">
  description: no description
  dimension: <HDF5 dataset "dimension": shape (3,), type "<i4">
  distance: 0.7
  field_of_view: <HDF5 dataset "field_of_view": shape (3,), type "<f8">
  format: raw
  interval: 1
  offset: 0.0
  orientation: lower left
  resolution: -1.0
  timestamps: <HDF5 dataset "timestamps": shape (200,), type "<f8">
  timestamps_unit: seconds
  unit: meters
}
```

`NWBFile.stimulus` is a dictionary that can contain PyNWB objects representing different types of data, such as images (grayscale, RGB) or time series of images. In this file, `NWBFile.stimulus` contains a single key "StimulusPresentation" with an *OpticalSeries* object representing what images were shown to the subject and at what times.

```
nwbfile.stimulus["StimulusPresentation"]
```

### Lazy loading of datasets

Data arrays are read passively from the NWB file. Accessing the `data` attribute of the *OpticalSeries* object does not read the data values, but presents an `h5py.Dataset` object that can be indexed to read data. You can use the `[:]` operator to read the entire data array into memory.

```
stimulus_presentation = nwbfile.stimulus["StimulusPresentation"]
all_stimulus_data = stimulus_presentation.data[:]
```

Images may be 3D or 4D (grayscale or RGB), where the first dimension must be time (frame). The second and third dimensions represent x and y. The fourth dimension represents the RGB value (length of 3) for color images.

```
stimulus_presentation.data.shape
```

```
(200, 400, 300, 3)
```

This *OpticalSeries* data contains 200 images of size 400x300 pixels with three channels (red, green, and blue).

### Slicing datasets

It is often preferable to read only a portion of the data. To do this, index or slice into the `data` attribute just like if you were indexing or slicing a numpy array.

```
frame_index = 31
image = stimulus_presentation.data[frame_index]
# Reverse the last dimension because the data were stored in BGR instead of RGB
image = image[..., ::-1]
plt.imshow(image, aspect="auto")
```


```
<matplotlib.image.AxesImage object at 0x7f27b083d910>
```

### Access single unit data

Data and metadata about sorted single units are stored in *Units* object. It stores metadata about each single unit in a tabular form, where each row represents a unit with spike times and additional metadata.

**See also:**

You can learn more about units in the *Extracellular Electrophysiology Data* tutorial.

```
units = nwbfile.units
```

We can view the single unit data as a DataFrame.

```
units_df = units.to_dataframe()
units_df.head()
```

To access the spike times of the first single unit, index *units* with the column name "spike_times" and then the row index, 0. All times in NWB are stored in seconds relative to the session start time.

```
units["spike_times"][0]
```

```
array([5932.811644, 6081.077044, 6091.982364, 6093.127644, 6095.068204,
       6097.438244, 6116.694804, 6129.827604, 6134.825004, 6142.583924,
       6148.385364, 6149.993804, 6150.397044, 6155.302324, 6160.684004,
       6164.865244, 6165.967804, 6166.810564, 6169.882924, 6173.715884,
       6178.882244, 6179.994244, 6190.154284, 6197.473884, 6201.784204,
       6204.267124, 6205.795604, 6209.183204, 6214.079844, 6216.054844,
       6216.622204, 6220.794284, 6223.041564, 6227.578284, 6241.826004,
       6243.708444, 6248.290124, 6249.827244, 6251.844244, 6252.321324,
       6255.445964, 6255.450764, 6256.071404, 6262.130524, 6263.449684,
       6271.980484, 6273.345364, 6274.503964, 6278.871164, 6282.031884,
       6293.636604, 6294.736004, 6298.655764, 6309.551284, 6316.313844,
       6317.823484, 6321.783684, 6324.364244, 6326.245564, 6327.291284,
       6327.506404, 6343.618684, 6348.224124, 6356.779644, 6811.910324,
       6831.062924, 6835.395244, 6837.133324, 6842.687684, 6844.716764,
       6852.494204, 6852.676004, 6861.838364, 6867.722964, 6868.506684,
       6870.520564, 6870.696084, 6870.992244, 6874.586124, 6875.521284,
       6875.526764, 6880.573684, 6884.808964, 6885.198524, 6887.827804,
       6891.872644, 6893.842164, 6895.660884, 6905.411844, 6905.945964,
       6908.227684, 6909.327524, 6910.216444, 6913.853124, 6920.003524,
       6920.500964, 6923.282284, 6923.790284, 6923.815324, 6924.482084,
       6929.252164, 6933.061564, 6934.280164, 6937.846484, 6940.652804,
       6943.622284, 6950.313084, 6950.363204, 6954.785244, 6954.933924,
       6959.337924, 6962.573244, 6963.663164, 6966.080444, 6966.146044,
       6970.827564, 6970.979524, 6973.283724, 6987.875764, 6992.403004,
       6993.954124, 6996.946764, 6997.249284, 6999.875764, 7004.600924,
       7008.001244, 7009.398204, 7011.068564, 7017.730804, 7019.278004,
       7024.695844, 7027.928084, 7041.186004, 7043.951164, 7044.394164,
       7052.429444, 7053.377284, 7054.072164, 7072.272564, 7072.836364,
       7073.433204, 7073.844604, 7073.901084, 7079.004124, 7080.598844,
       7083.965564, 7084.016084, 7086.730804, 7090.336364, 7101.254244,
       7114.549764, 7116.230604, 7119.653564, 7119.685684, 7122.680484,
       7127.814804, 7129.421884, 7141.764244, 7143.759004, 7147.602804,
       7149.140324, 7151.777524, 7157.066044, 7157.118404, 7158.532644,
       7334.693084, 7335.015444, 7336.416404, 7340.052364, 7345.172164,
       7363.078724, 7365.825364, 7375.025684, 7381.381804, 7382.057444,
       7382.416484, 7382.997684, 7383.204804, 7384.305124, 7386.312724,
       7386.694364, 7389.302964, 7396.554924, 7397.925444, 7402.733404,
       7406.523084, 7407.317684, 7409.281964, 7411.153164, 7412.606124,
       7413.468804, 7420.820204, 7424.900604, 7425.235964, 7425.489244,
       7426.975844, 7431.840764, 7434.457964, 7436.035524, 7436.086484,
       7438.281564, 7444.681404, 7446.768444, 7452.015684, 7455.865644,
       7459.695124, 7469.229324, 7469.447164, 7470.021844, 7475.205524,
       7480.513124, 7485.459604, 7487.198044, 7491.487884, 7491.529244,
       7503.131844, 7509.601444, 7518.232884, 7520.176244, 7522.629044,
       7526.786084, 7527.225084, 7527.668244, 7531.142964, 7539.957964,
       7541.722804, 7543.518964, 7547.003564, 7549.018484, 7549.069804,
       7550.374044, 7551.872924, 7553.427244, 7563.422044, 7566.144044,
       7567.238124, 7568.862244, 7569.073564, 7575.387404, 7583.933364,
       7584.624484, 7594.007284, 7594.054964, 7595.263164, 7596.982804,
       7602.387084, 7605.198924, 7605.552324, 7606.064884, 7606.100604,
       7606.471644, 7608.873244, 7611.354804, 7611.386804, 7615.139004,
       7620.247884, 7621.979604, 7629.935284, 7632.076484, 7635.188044,
       7635.195444, 7650.052684, 7954.638503, 7954.781023, 7955.781503,
```

(continues on next page)

(continued from previous page)

```
       7957.472423, 8737.861503, 8738.114583, 8745.994143, 8751.983023,
       8752.956743, 8759.633223, 8763.513143, 8767.843183, 8769.437743,
       8769.722423, 8770.277983, 8771.478103, 8774.213063, 8775.969463,
       8788.423623, 8788.552143, 8788.986663, 8794.429063, 8796.650303,
       8797.612063, 8799.980783, 8800.796863, 8803.618063, 8803.886663,
       8803.947783, 8806.479783, 8809.613743, 8811.981903, 8821.248783,
       8821.956583, 8823.353583, 8832.448903, 8833.632423, 8835.896503,
       8846.307303, 8853.548143, 8854.569343, 8855.609823, 8858.938063,
       8859.377623, 8861.211783, 8861.520783, 8863.375583, 8874.494623,
       8875.107303, 8877.309463, 8890.033823, 8893.205503, 8893.727743,
       8902.555583, 8908.088743, 8909.480663, 8909.599623, 8917.999063,
       8918.456263, 8922.668823, 8924.168263, 8925.681703, 8934.790023,
       8941.341583, 8941.729463, 8946.041383, 8953.742663, 8955.295183,
       8955.777943, 8958.401863, 8959.338423, 8968.056023, 8968.512463,
       8970.277183, 8971.752023, 8974.239503, 8983.962903, 8985.244303,
       8988.592263, 8991.135303, 8991.409423, 8992.908263, 8995.101423,
       8997.635343, 8999.058143, 9002.087743, 9003.022143, 9003.163143,
       9003.484943, 9003.741023, 9008.767263, 9008.915183, 9010.942143,
       9010.976183, 9013.915423, 9015.557943, 9018.411263, 9023.921183,
       9030.917263, 9043.395623, 9049.802143, 9053.791543, 9065.276343,
       9074.427783, 9077.736063, 9081.744663, 9082.581383, 9088.609223,
       9089.031503, 9089.147943, 9098.463543])
```

### Visualize spiking activity relative to stimulus onset

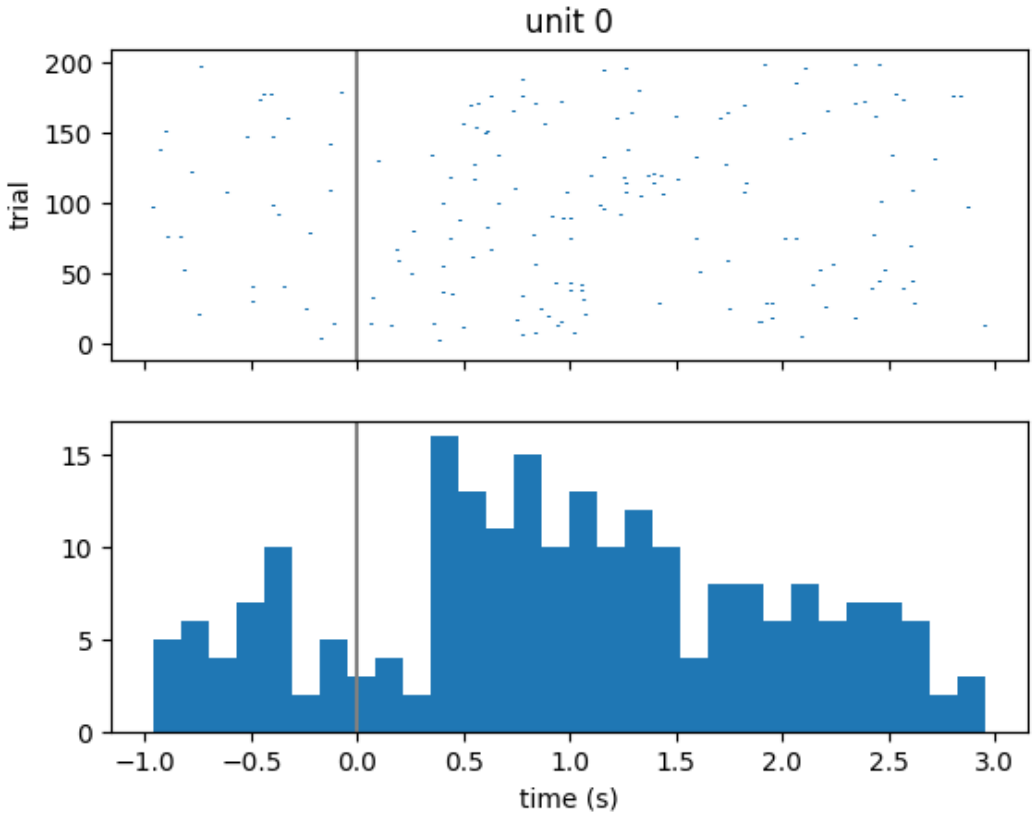
We can look at when these single units spike relative to when image stimuli were presented to the subject. We will iterate over the first 3 units and get their spike times. Then for each unit, we will iterate over each stimulus onset time and compute the spike times relative to stimulus onset. Finally, we will create a raster plot and histogram of these aligned spike times.

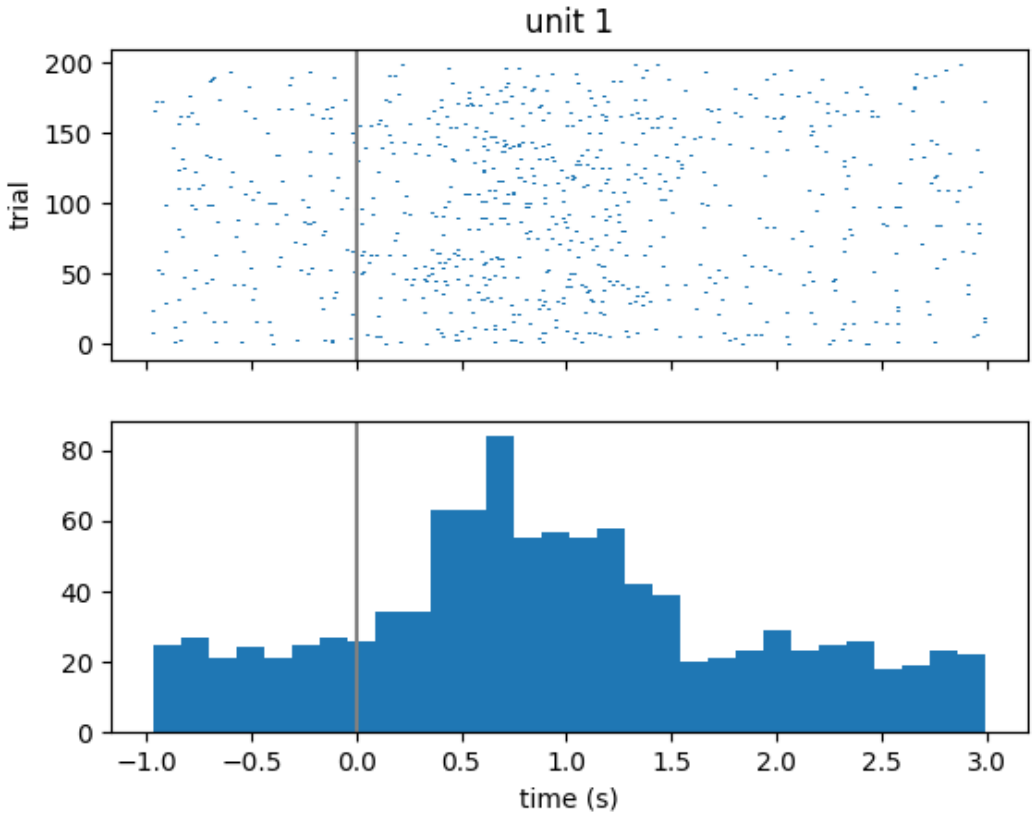
```
before = 1.0  # in seconds
after = 3.0

# Get the stimulus times for all stimuli
# get_timestamps() works whether the time is stored as an array of timestamps or as
# starting time and sampling rate.
stim_on_times = stimulus_presentation.get_timestamps()

for unit in range(3):
    unit_spike_times = nwbfile.units["spike_times"][unit]
    trial_spikes = []
    for time in stim_on_times:
        # Compute spike times relative to stimulus onset
        aligned_spikes = unit_spike_times - time
        # Keep only spike times in a given time window around the stimulus onset
        aligned_spikes = aligned_spikes[
            (-before < aligned_spikes) & (aligned_spikes < after)
        ]
        trial_spikes.append(aligned_spikes)
    fig, axs = plt.subplots(2, 1, sharex="all")
    plt.xlabel("time (s)")
```

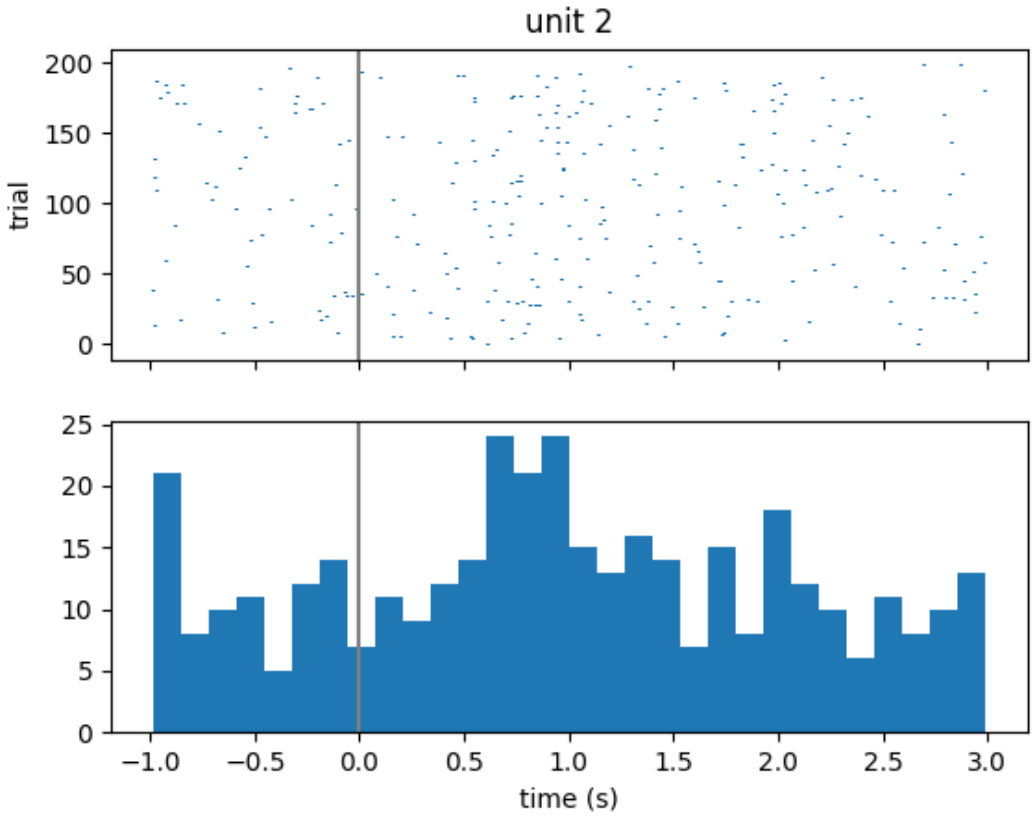
(continues on next page)

(continued from previous page)

```
    axs[0].eventplot(trial_spikes)

    axs[0].set_ylabel("trial")
    axs[0].set_title("unit {}".format(unit))
    axs[0].axvline(0, color=[0.5, 0.5, 0.5])

    axs[1].hist(np.hstack(trial_spikes), 30)
    axs[1].axvline(0, color=[0.5, 0.5, 0.5])
```




unit 1
trial
200
150
100
50
0
80
60
40
20
0
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
time (s)




- 

### Access Trials

Trials are stored as *TimeIntervals* object which is a subclass of DynamicTable. DynamicTable objects are used to store metadata about each trial in a tabular form, where each row represents a trial and has a start time, stop time, and additional metadata.

**See also:**

You can learn more about trials in the *Annotating Time Intervals* tutorial.

Similarly to *Units*, we can view trials as a pandas.DataFrame.

```
trials_df = nwbfile.trials.to_dataframe()
trials_df.head()
```

The stimulus can be mapped one-to-one to each row (trial) of *trials* based on the stim_on_time column.

```
assert np.all(stimulus_presentation.timestamps[:] == trials_df.stim_on_time[:])
```

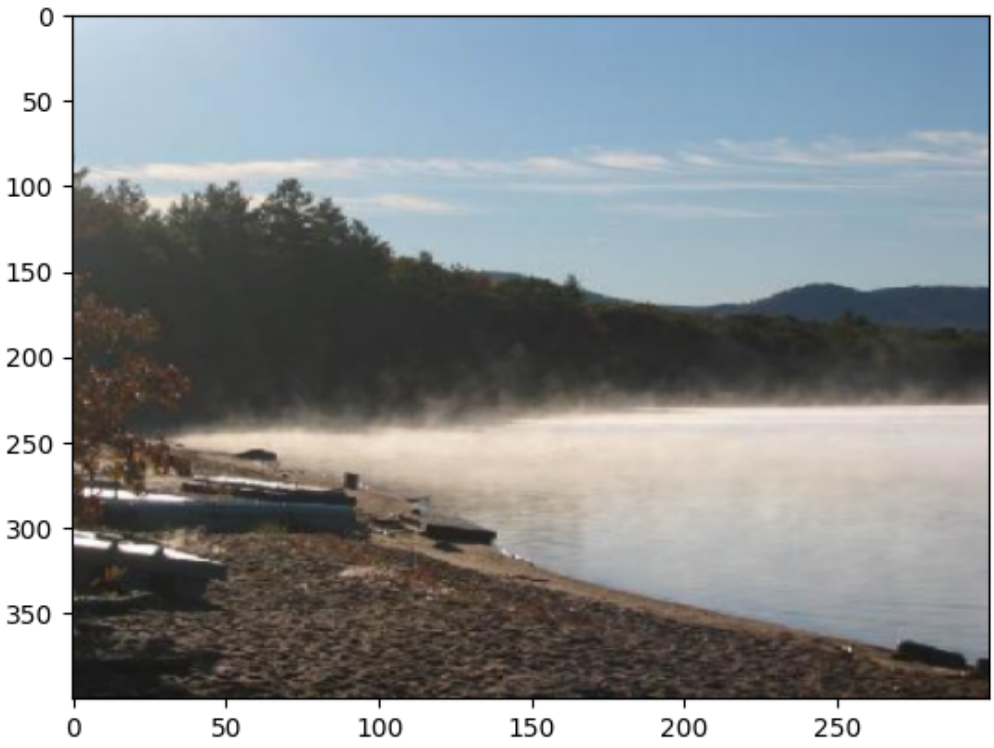
Visualize the first 3 images that were categorized as landscapes in the session:

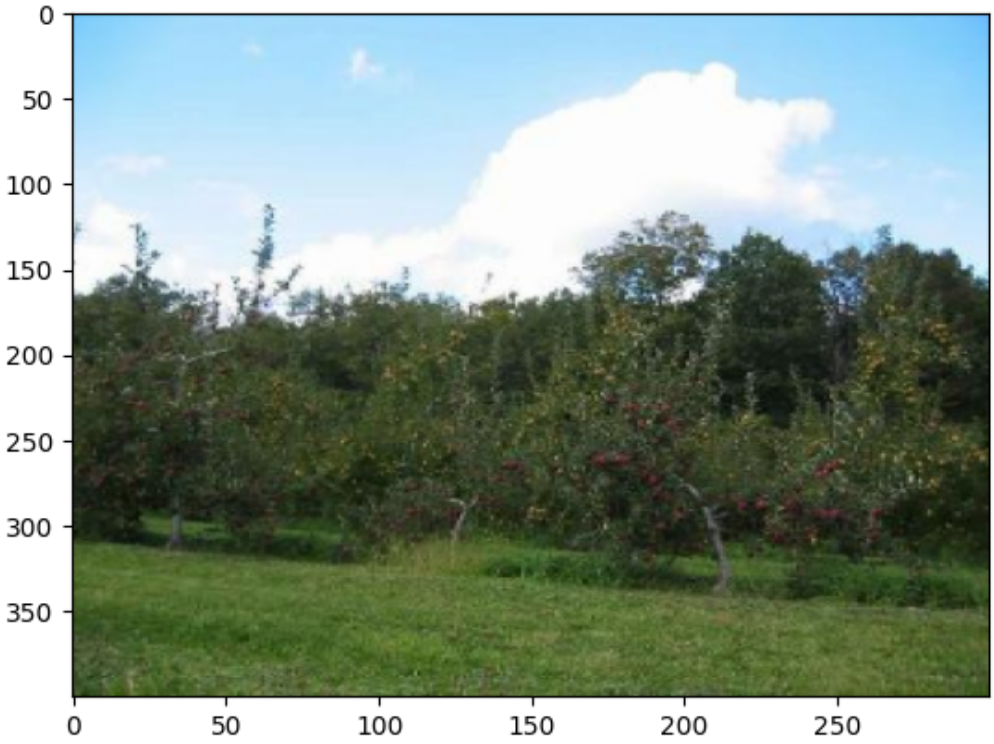
```
stim_on_times_landscapes = trials_df[
    trials_df.category_name == "landscapes"
].stim_on_time
for time in stim_on_times_landscapes.iloc[:3]:
    img = np.squeeze(
```

(continues on next page)

(continued from previous page)

```
        stimulus_presentation.data[
            np.where(stimulus_presentation.timestamps[:] == time)
        ]
    )
    # Reverse the last dimension because the data were stored in BGR instead of RGB
    img = img[..., ::-1]
    plt.figure()
    plt.imshow(img, aspect="auto")
```




0
50
100
150
200
250
300
350
0
50
100
150
200
250


- 

### Exploring the NWB file

So far we have explored the NWB file by printing the *NWBFile* object and accessing its attributes, but it may be useful to explore the data in a more interactive, visual way. See Exploring NWB Files for an updated list of programs for exploring NWB files.

### Close the open NWB file

It is good practice, especially on Windows, to close any files that you have opened.

```
io.close()
```

## 2.2 Domain-specific tutorials

### 2.2.1 Extracellular Electrophysiology Data

This tutorial describes storage of extracellular electrophysiology data in NWB in four main steps:

1. Create the electrodes table
2. Add acquired raw voltage data
3. Add LFP data

4. Add spike data

It is recommended to cover *NWB File Basics* before this tutorial.

**Note:** It is recommended to check if your source data is supported by NeuroConv Extracellular Electrophysiology Gallery. If it is supported, it is recommended to use NeuroConv to convert your 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:

```
from datetime import datetime
from uuid import uuid4

import numpy as np
from dateutil.tz import tzlocal

from pynwb import NWBHDF5IO, NWBFile
from pynwb.ecephys import LFP, ElectricalSeries
```

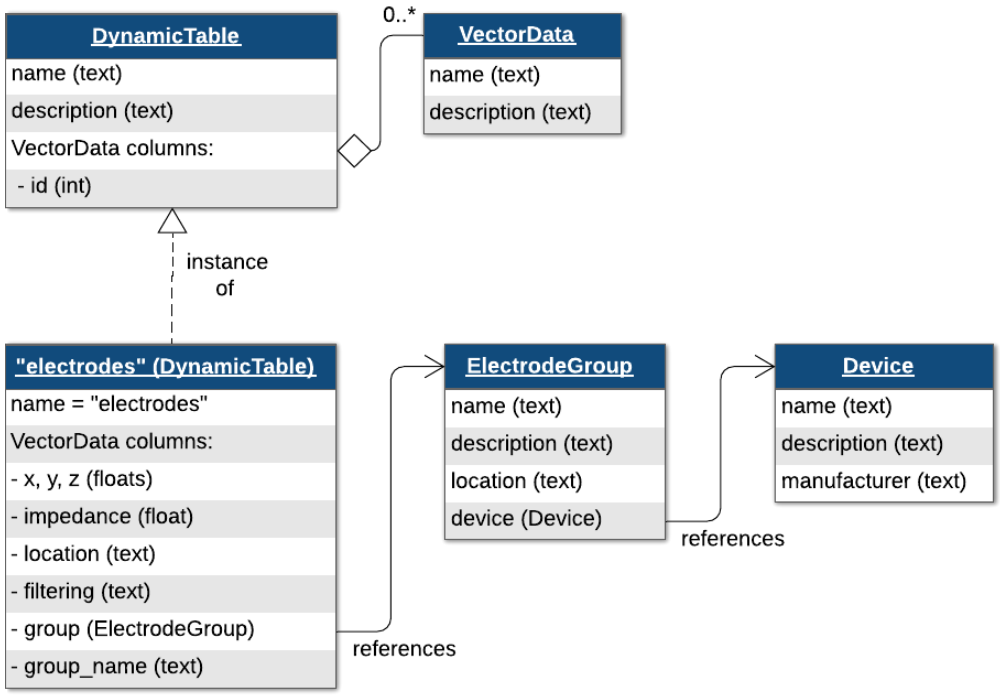
### Creating and Writing NWB files

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

```
nwbfile = NWBFile(
    session_description="my first synthetic recording",
    identifier=str(uuid4()),
    session_start_time=datetime.now(tzlocal()),
    experimenter=[
        "Baggins, Bilbo",
    ],
    lab="Bag End Laboratory",
    institution="University of Middle Earth at the Shire",
    experiment_description="I went on an adventure to reclaim vast treasures.",
    session_id="LONELYMTN001",
)
```

### Electrodes Table

To store extracellular electrophysiology data, you first must create an electrodes table describing the electrodes that generated this data. Extracellular electrodes are stored in an `"electrodes"` table, which is a `DynamicTable`.




The electrodes table references a required *ElectrodeGroup*, which is used to represent a group of electrodes. Before creating an *ElectrodeGroup*, you must define a *Device* object using the method *NWBFile.create_device*.

```
device = nwbfile.create_device(
    name="array", description="the best array", manufacturer="Probe Company 9000"
)
```

Once you have created the *Device*, you can create an *ElectrodeGroup*. Then you can add electrodes one-at-a-time with *NWBFile.add_electrode*. *NWBFile.add_electrode* has two required arguments, `group`, which takes an *ElectrodeGroup*, and `location`, which takes a string. It also has a number of optional metadata fields for electrode features (e.g, `x`, `y`, `z`, `imp`, and `filtering`). Since this table is a `DynamicTable`, we can add additional user-specified metadata as custom columns of the table. We will be adding a `"label"` column to the table. Use the following code to add electrodes for an array with 4 shanks and 3 channels per shank.

```
nwbfile.add_electrode_column(name="label", description="label of electrode")

nshanks = 4
nchannels_per_shank = 3
electrode_counter = 0

for ishank in range(nshanks):
    # create an electrode group for this shank
    electrode_group = nwbfile.create_electrode_group(
        name="shank{}".format(ishank),
        description="electrode group for shank {}".format(ishank),
        device=device,
```

(continues on next page)

(continued from previous page)

```
        location="brain area",
    )
    # add electrodes to the electrode table
    for ielec in range(nchannels_per_shank):
        nwbfile.add_electrode(
            group=electrode_group,
            label="shank{}elec{}".format(ishank, ielec),
            location="brain area",
        )
        electrode_counter += 1
```

Similarly to the `trials` table, we can view the `electrodes` table in tabular form by converting it to a pandas `DataFrame`.

```
nwbfile.electrodes.to_dataframe()
```

**Note:** When we added an electrode with the *add_electrode* method, we passed in the *ElectrodeGroup* object for the `"group"` argument. This creates a reference from the `"electrodes"` table to the individual *ElectrodeGroup* objects, one per row (electrode).

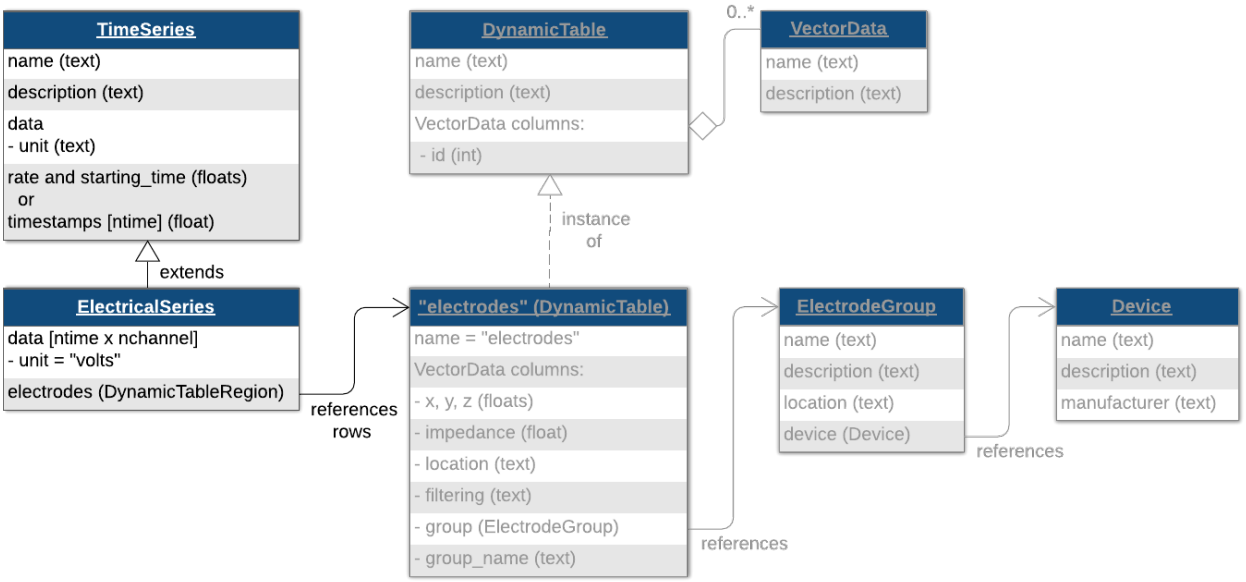
### Extracellular recordings

Raw voltage traces and local-field potential (LFP) data are stored in *ElectricalSeries* objects. *ElectricalSeries* is a subclass of *TimeSeries* specialized for voltage data. To create the *ElectricalSeries* objects, we need to reference a set of rows in the `"electrodes"` table to indicate which electrodes were recorded. We will do this by creating a `DynamicTableRegion`, which is a type of link that allows you to reference rows of a `DynamicTable`. *NWBFile.create_electrode_table_region* is a convenience function that creates a `DynamicTableRegion` which references the `"electrodes"` table.

```
all_table_region = nwbfile.create_electrode_table_region(
    region=list(range(electrode_counter)),  # reference row indices 0 to N-1
    description="all electrodes",
)
```

### Raw voltage data

Now create an *ElectricalSeries* object to store raw data collected during the experiment, passing in this "all_table_region" DynamicTableRegion reference to all rows of the electrodes table.




```
raw_data = np.random.randn(50, 12)
raw_electrical_series = ElectricalSeries(
    name="ElectricalSeries",
    data=raw_data,
    electrodes=all_table_region,
    starting_time=0.0,  # timestamp of the first sample in seconds relative to the␣
↪session start time
    rate=20000.0,  # in Hz
)
```

Since this *ElectricalSeries* represents raw data from the data acquisition system, add it to the acquisition group of the *NWBFile*.

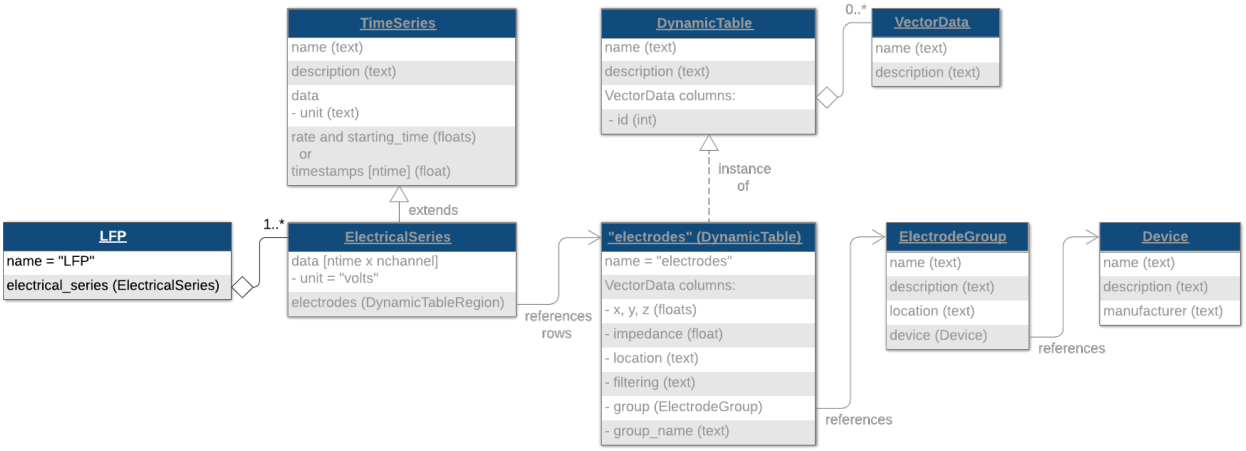
```
nwbfile.add_acquisition(raw_electrical_series)
```

### LFP

Now create an *ElectricalSeries* object to store LFP data collected during the experiment, again passing in the DynamicTableRegion reference to all rows of the "electrodes" table.

```
lfp_data = np.random.randn(50, 12)
lfp_electrical_series = ElectricalSeries(
    name="ElectricalSeries",
    data=lfp_data,
    electrodes=all_table_region,
    starting_time=0.0,
    rate=200.0,
)
```

To help data analysis and visualization tools know that this *ElectricalSeries* object represents LFP data, store the *ElectricalSeries* object inside of an *LFP* object. This is analogous to how we can store the *SpatialSeries* object inside of a *Position* object.




```
lfp = LFP(electrical_series=lfp_electrical_series)
```

Unlike the raw data, which we put into the acquisition group of the *NWBFile*, LFP data is typically considered processed data because the raw data was filtered and downsampled to generate the LFP.

Create a processing module named "ecephys" and add the *LFP* object to it. This is analogous to how we can store the *Position* object in a processing module created with the method *NWBFile.create_processing_module*.

```
ecephys_module = nwbfile.create_processing_module(
    name="ecephys", description="processed extracellular electrophysiology data"
)
ecephys_module.add(lfp)
```

### Spike Times

Spike times are stored in the *Units* table, which is a subclass of DynamicTable. Adding columns to the *Units* table is analogous to how we can add columns to the "electrodes" and "trials" tables.

Generate some random spike data and populate the *Units* table using the method *NWBFile.add_unit*.

```
nwbfile.add_unit_column(name="quality", description="sorting quality")

firing_rate = 20
n_units = 10
res = 1000
duration = 20
for n_units_per_shank in range(n_units):
    spike_times = np.where(np.random.rand((res * duration)) < (firing_rate / res))[0] /␣
↪res
    nwbfile.add_unit(spike_times=spike_times, quality="good")
```

The *Units* table can also be converted to a pandas DataFrame.

```
nwbfile.units.to_dataframe()
```

### Designating electrophysiology data

As mentioned above, *ElectricalSeries* objects are meant for storing specific types of extracellular recordings. In addition to this *TimeSeries* class, NWB provides some *Processing Modules* for designating the type of data you are storing. We will briefly discuss them here, and refer the reader to *API documentation* and *NWB File Basics* 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 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 mean 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).

### Writing electrophysiology data

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

```
with NWBHDF5IO("ecephys_tutorial.nwb", "w") as io:
    io.write(nwbfile)
```

For more details on *NWBHDF5IO*, see the *Writing an NWB file* tutorial.

### Reading electrophysiology data

Access the raw data by indexing *acquisition* with the name of the *ElectricalSeries*, which we named "ElectricalSeries". We can also access the LFP data by indexing *processing* with the name of the processing module "ecephys". Then, we can access the *LFP* object inside the "ecephys" processing module by indexing it with the name of the *LFP* object. The default name of *LFP* objects is "LFP". Finally, we can access the *ElectricalSeries* object inside the *LFP* object by indexing it with the name of the *ElectricalSeries* object, which we named "ElectricalSeries".

```
with NWBHDF5IO("ecephys_tutorial.nwb", "r") as io:
    read_nwbfile = io.read()
    print(read_nwbfile.acquisition["ElectricalSeries"])
    print(read_nwbfile.processing["ecephys"])
    print(read_nwbfile.processing["ecephys"]["LFP"])
    print(read_nwbfile.processing["ecephys"]["LFP"]["ElectricalSeries"])
```

#### Accessing your data

Data arrays are read passively from the file. Calling the data attribute on a *TimeSeries* such as a *ElectricalSeries* does not read the data values, but presents an `h5py.Dataset` object that can be indexed to read data. You can use the `[:]` operator to read the entire data array into memory.

Load and print all the data values of the *ElectricalSeries* object representing the LFP data.

```
with NWBHDF5IO("ecephys_tutorial.nwb", "r") as io:
    read_nwbfile = io.read()
    print(read_nwbfile.processing["ecephys"]["LFP"]["ElectricalSeries"].data[:])
```

#### Accessing data regions

It is often preferable to read only a portion of the data. To do this, index or slice into the `data` attribute just like if you index or slice a `numpy.ndarray`.

The following code prints elements `0:10` in the first dimension (time) and `0:3` in the second dimension (electrodes) from the LFP data we have written. It also demonstrates how to access the spike times of the 0th unit.

```
with NWBHDF5IO("ecephys_tutorial.nwb", "r") as io:
    read_nwbfile = io.read()

    print("section of LFP:")
    print(read_nwbfile.processing["ecephys"]["LFP"]["ElectricalSeries"].data[:10, :3])
    print("")
    print("spike times from 0th unit:")
    print(read_nwbfile.units["spike_times"][0])
```

### 2.2.2 Calcium Imaging Data

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

1. Create imaging plane
2. Add acquired two-photon images
3. Add motion correction (optional)
4. Add image segmentation
5. Add fluorescence and dF/F responses

It is recommended to cover *NWB File Basics* before this tutorial.

---

**Note:** It is recommended to check if your source data is supported by NeuroConv Optical Physiology Gallery. If it is supported, it is recommended to use NeuroConv to convert your 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:

```
from datetime import datetime
from uuid import uuid4

import matplotlib.pyplot as plt
import numpy as np
from dateutil.tz import tzlocal

from pynwb import NWBHDF5IO, NWBFile, TimeSeries
from pynwb.image import ImageSeries
from pynwb.ophys import (
    CorrectedImageStack,
    Fluorescence,
    ImageSegmentation,
    MotionCorrection,
    OnePhotonSeries,
    OpticalChannel,
    RoiResponseSeries,
    TwoPhotonSeries,
)
```

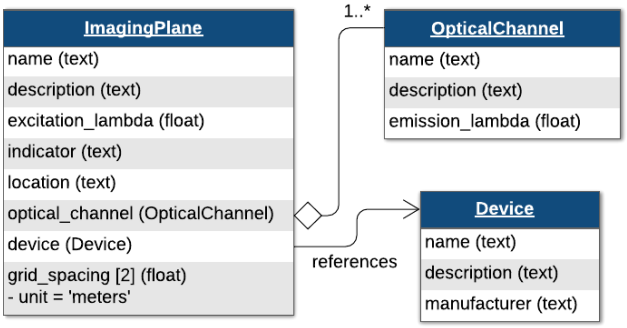
### Creating the NWB file

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

```
nwbfile = NWBFile(
    session_description="my first synthetic recording",
    identifier=str(uuid4()),
    session_start_time=datetime.now(tzlocal()),
    experimenter=[
        "Baggins, Bilbo",
    ],
    lab="Bag End Laboratory",
    institution="University of Middle Earth at the Shire",
    experiment_description="I went on an adventure to reclaim vast treasures.",
    session_id="LONELYMTN001",
)
```

### Imaging Plane

First, we must create an *ImagingPlane* object, which will hold information about the area and method used to collect the optical imaging data. This first requires creation of a *Device* object for the microscope and an *OpticalChannel* object.




Create a *Device* named "Microscope" in the *NWBFile* object. Then create an *OpticalChannel* named "OpticalChannel".

```
device = nwbfile.create_device(
    name="Microscope",
    description="My two-photon microscope",
    manufacturer="The best microscope manufacturer",
)
optical_channel = OpticalChannel(
    name="OpticalChannel",
    description="an optical channel",
    emission_lambda=500.0,
)
```

Now, create a *ImagingPlane* named "ImagingPlane", passing in the *OpticalChannel* object and the *Device* object.

```
imaging_plane = nwbfile.create_imaging_plane(
    name="ImagingPlane",
    optical_channel=optical_channel,
    imaging_rate=30.0,
    description="a very interesting part of the brain",
    device=device,
    excitation_lambda=600.0,
    indicator="GFP",
    location="V1",
    grid_spacing=[0.01, 0.01],
    grid_spacing_unit="meters",
    origin_coords=[1.0, 2.0, 3.0],
    origin_coords_unit="meters",
)
```

### One-photon Series

Now that we have our *ImagingPlane*, we can create a *OnePhotonSeries* object to store raw one-photon imaging data. Here, we have two options. The first option is to supply the raw image data to PyNWB, using the data argument. The second option is to provide a path to the image files. These two options have trade-offs, so it is worth considering how you want to store this data.

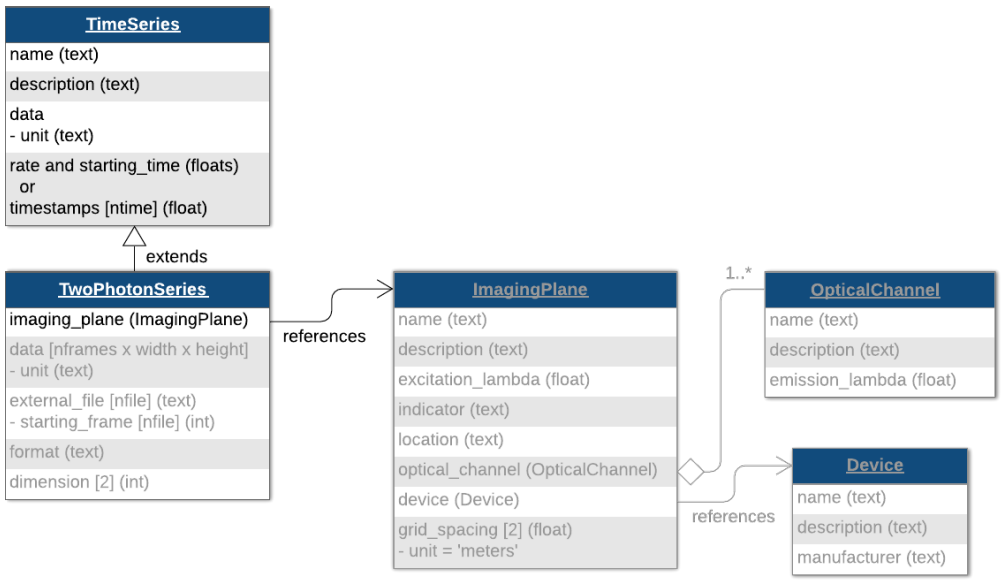
```
# using internal data. this data will be stored inside the NWB file
one_p_series1 = OnePhotonSeries(
    name="OnePhotonSeries_internal",
    data=np.ones((1000, 100, 100)),
    imaging_plane=imaging_plane,
    rate=1.0,
    unit="normalized amplitude",
)

# using external data. only the file paths will be stored inside the NWB file
one_p_series2 = OnePhotonSeries(
    name="OnePhotonSeries_external",
    dimension=[100, 100],
    external_file=["images.tiff"],
    imaging_plane=imaging_plane,
    starting_frame=[0],
    format="external",
    starting_time=0.0,
    rate=1.0,
)
```

Since these one-photon data are acquired data, we will add the *OnePhotonSeries* objects to the *NWBFile* as acquired data.

```
nwbfile.add_acquisition(one_p_series1)
nwbfile.add_acquisition(one_p_series2)
```

### Two-photon Series

*TwoPhotonSeries* objects store acquired two-photon imaging data. This class behaves similarly to *OnePhotonSeries*.




```
# using internal data. this data will be stored inside the NWB file
two_p_series1 = TwoPhotonSeries(
    name="TwoPhotonSeries1",
    data=np.ones((1000, 100, 100)),
    imaging_plane=imaging_plane,
    rate=1.0,
    unit="normalized amplitude",
)

# using external data. only the file paths will be stored inside the NWB file
two_p_series2 = TwoPhotonSeries(
    name="TwoPhotonSeries2",
    dimension=[100, 100],
    external_file=["images.tiff"],
    imaging_plane=imaging_plane,
    starting_frame=[0],
    format="external",
    starting_time=0.0,
    rate=1.0,
)

nwbfile.add_acquisition(two_p_series1)
nwbfile.add_acquisition(two_p_series2)
```

### Motion Correction (optional)

You can also store the result of motion correction using a *MotionCorrection* object, which is a *MultiContainerInterface* (similar to *Position*) which holds 1 or more *CorrectedImageStack* objects.

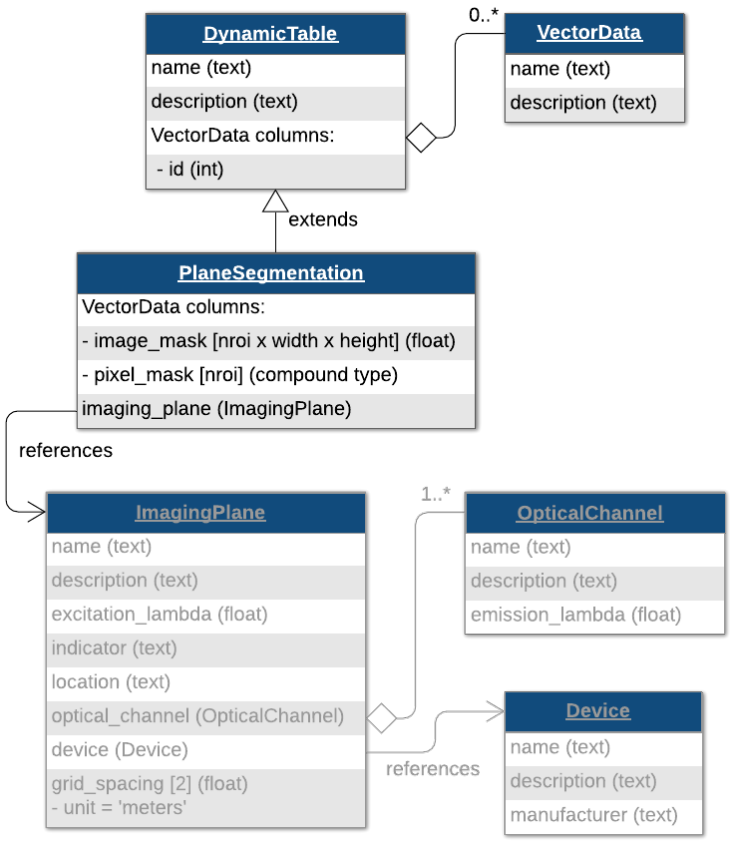
```
corrected = ImageSeries(
    name="corrected",  # this must be named "corrected"
    data=np.ones((1000, 100, 100)),
    unit="na",
    format="raw",
    starting_time=0.0,
    rate=1.0,
)

xy_translation = TimeSeries(
    name="xy_translation",
    data=np.ones((1000, 2)),
    unit="pixels",
    starting_time=0.0,
    rate=1.0,
)

corrected_image_stack = CorrectedImageStack(
    corrected=corrected,
    original=one_p_series1,
    xy_translation=xy_translation,
)

motion_correction = MotionCorrection(corrected_image_stacks=[corrected_image_stack])
```

We will create a *ProcessingModule* named "ophys" to store optical physiology data and add the motion correction data to the *NWBFile*.

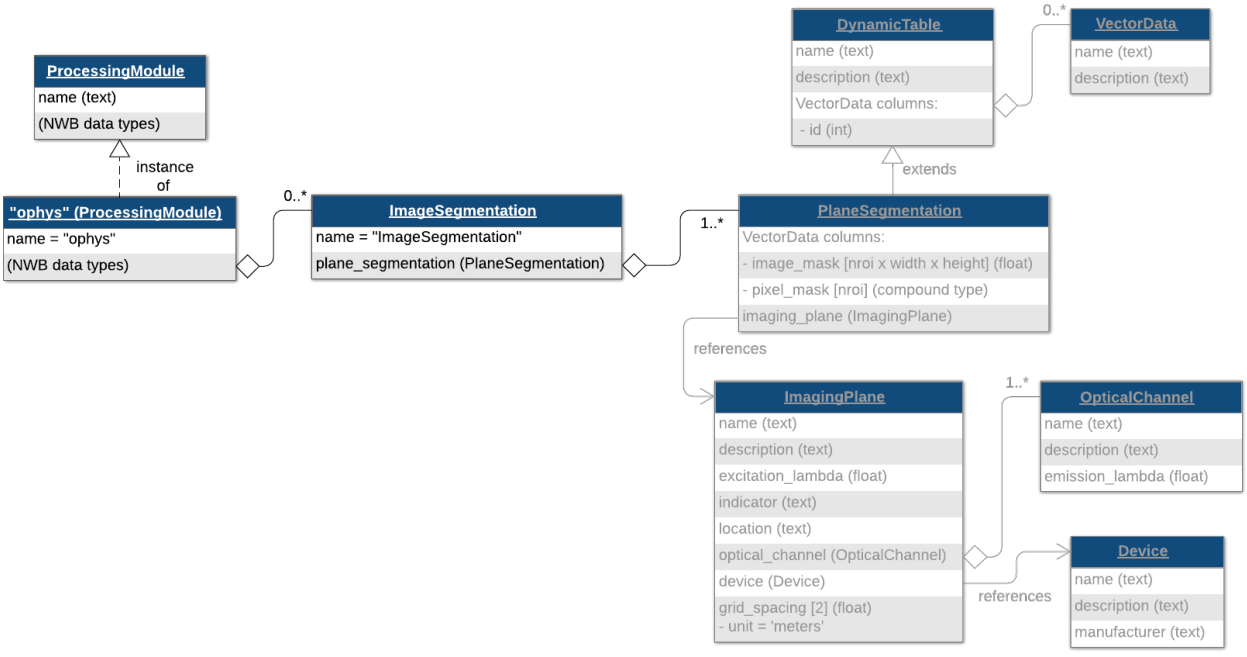
```
ophys_module = nwbfile.create_processing_module(
    name="ophys", description="optical physiology processed data"
)

ophys_module.add(motion_correction)
```

### Plane Segmentation

The *PlaneSegmentation* class stores the detected regions of interest in the *TwoPhotonSeries* data. *PlaneSegmentation* is a subclass of DynamicTable, where each row represents a single region of interest (ROI).




The *ImageSegmentation* class can contain multiple *PlaneSegmentation* tables, so that we can store results of different segmentation algorithms or different segmentation classes.




First, we create an *ImageSegmentation* object, then from that object we create a *PlaneSegmentation* table with a link to the *ImagingPlane* created earlier. Then we will add the *ImageSegmentation* object to the previously created

*ProcessingModule*.

```
img_seg = ImageSegmentation()

ps = img_seg.create_plane_segmentation(
    name="PlaneSegmentation",
    description="output from segmenting my favorite imaging plane",
    imaging_plane=imaging_plane,
    reference_images=one_p_series1,  # optional
)

ophys_module.add(img_seg)
```

### Regions Of Interest (ROIs)

#### Image masks

You can add ROIs to the *PlaneSegmentation* table using an image mask or a pixel mask. An image mask is an array that is the same size as a single frame of the *TwoPhotonSeries* that indicates the mask weight of each pixel in the image. Image mask values (weights) may be boolean or continuous between 0 and 1.

```
for _ in range(30):
    image_mask = np.zeros((100, 100))

    # randomly generate example image masks
    x = np.random.randint(0, 95)
    y = np.random.randint(0, 95)
    image_mask[x:x + 5, y:y + 5] = 1

    # add image mask to plane segmentation
    ps.add_roi(image_mask=image_mask)

# show one of the image masks
plt.imshow(image_mask)
```

#### Pixel masks

Alternatively, you could define ROIs using a pixel mask, which is an array of triplets (x, y, weight) that have a non-zero weight. All undefined pixels are assumed to be 0.

```
ps2 = img_seg.create_plane_segmentation(
    name="PlaneSegmentation2",
    description="output from segmenting my favorite imaging plane",
    imaging_plane=imaging_plane,
    reference_images=one_p_series1,  # optional
)

for _ in range(30):
    # randomly generate example starting points for region
    x = np.random.randint(0, 95)
    y = np.random.randint(0, 95)
```

(continues on next page)

(continued from previous page)

```
    # define an example 4 x 3 region of pixels of weight '1'
    pixel_mask = []
    for ix in range(x, x + 4):
        for iy in range(y, y + 3):
            pixel_mask.append((ix, iy, 1))

    # add pixel mask to plane segmentation
    ps2.add_roi(pixel_mask=pixel_mask)
```

### Voxel masks

When storing the segmentation of volumetric imaging, you can use 3D imaging masks. Alternatively, you could define ROIs using a voxel mask, which is an array of triplets (x, y, z, weight) that have a non-zero weight. All undefined voxels are assumed to be 0.

---

**Note:** You need to be consistent within a *PlaneSegmentation* table. You can add ROIs either using image masks, pixel masks, or voxel masks.

---

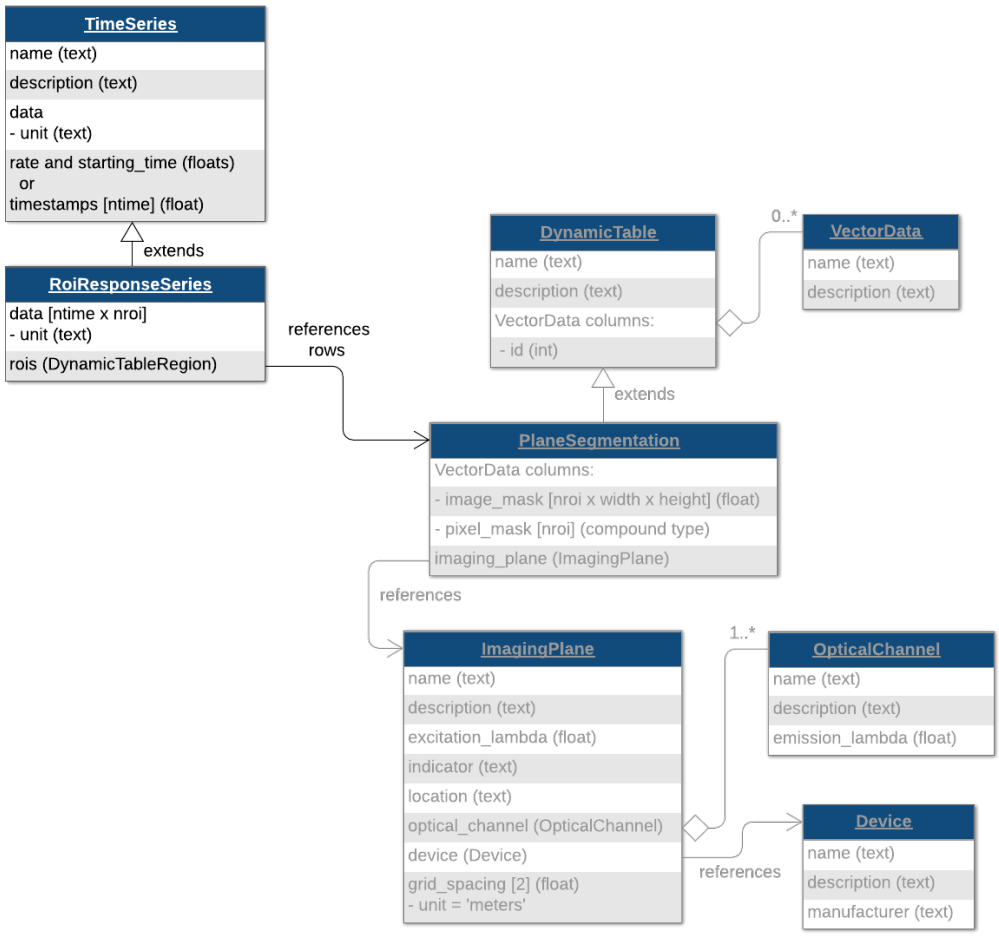
```
ps3 = img_seg.create_plane_segmentation(
    name="PlaneSegmentation3",
    description="output from segmenting my favorite imaging plane",
    imaging_plane=imaging_plane,
    reference_images=one_p_series1,  # optional
)

from itertools import product

for _ in range(30):
    # randomly generate example starting points for region
    x = np.random.randint(0, 95)
    y = np.random.randint(0, 95)
    z = np.random.randint(0, 15)

    # define an example 4 x 3 x 2 voxel region of weight '0.5'
    voxel_mask = []
    for ix, iy, iz in product(
        range(x, x + 4),
        range(y, y + 3),
        range(z, z + 2)
    ):
        voxel_mask.append((ix, iy, iz, 0.5))

    # add voxel mask to plane segmentation
    ps3.add_roi(voxel_mask=voxel_mask)
```

We can view the *PlaneSegmentation* table with pixel masks in tabular form by converting it to a `DataFrame`.

```
ps2.to_dataframe()
```

### Storing Fluorescence Measurements

Now that the regions of interest are stored, you can store fluorescence data for these ROIs. This type of data is stored using the *RoiResponseSeries* classes.




To create a *RoiResponseSeries* object, we will need to reference a set of rows from a *PlaneSegmentation* table to indicate which ROIs correspond to which rows of your recorded data matrix. This is done using a DynamicTableRegion, which is a type of link that allows you to reference specific rows of a DynamicTable, such as a *PlaneSegmentation* table by row indices.

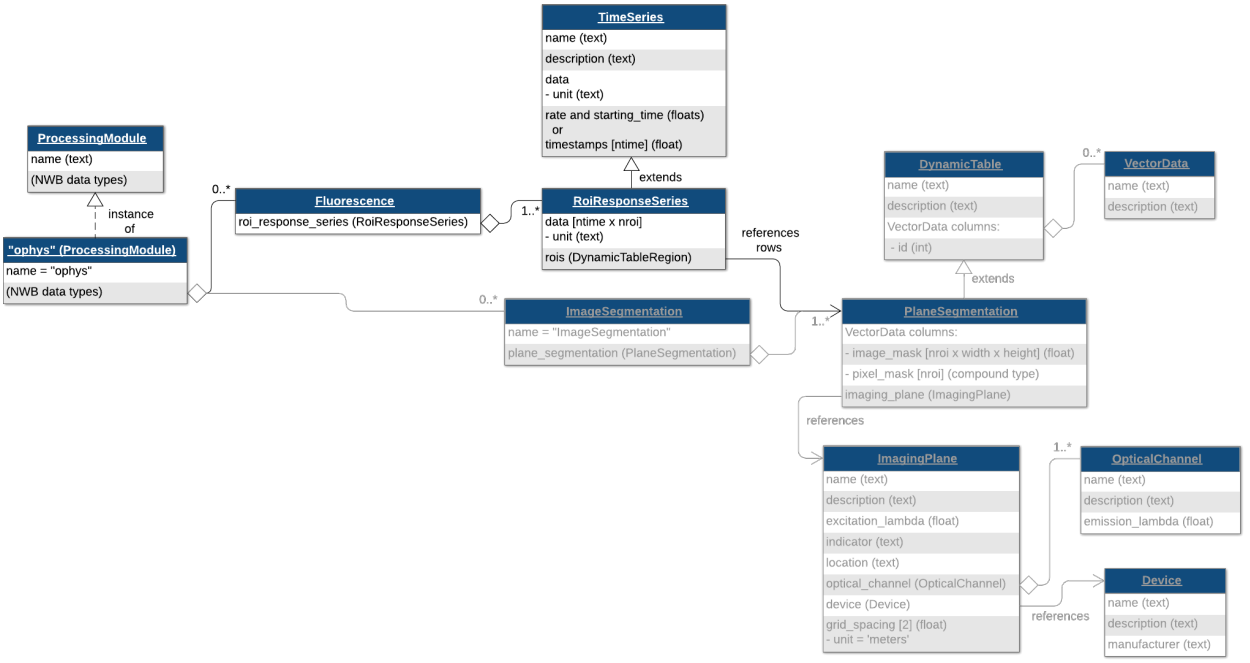
First, we create a DynamicTableRegion that references the first two ROIs of the *PlaneSegmentation* table.

```
rt_region = ps.create_roi_table_region(
    region=[0, 1], description="the first of two ROIs"
)
```

Then we create a *RoiResponseSeries* object to store fluorescence data for those two ROIs.

```
roi_resp_series = RoiResponseSeries(
    name="RoiResponseSeries",
    data=np.ones((50, 2)),  # 50 samples, 2 ROIs
    rois=rt_region,
    unit="lumens",
    rate=30.0,
)
```

To help data analysis and visualization tools know that this *RoiResponseSeries* object represents fluorescence data, we will store the *RoiResponseSeries* object inside of a *Fluorescence* object. Then add the *Fluorescence* object into the same *ProcessingModule* named "ophys" that we created earlier.




```
fl = Fluorescence(roi_response_series=roi_resp_series)
ophys_module.add(fl)
```

**Tip:** If you want to store dF/F data instead of fluorescence data, then store the *RoiResponseSeries* object in a *DfOverF* object, which works the same way as the *Fluorescence* class.

### Write the file

Once we have finished adding all of our data to our *NWBFile*, make sure to write the file. IO operations are carried out using *NWBHDF5IO*.

```
with NWBHDF5IO("ophys_tutorial.nwb", "w") as io:
    io.write(nwbfile)
```

### Read the NWBFile

We can access the raw data by indexing `nwbfile.acquisition` with a name of the *TwoPhotonSeries*, e.g., "TwoPhotonSeries1".

We can also access the fluorescence responses by indexing `nwbfile.processing` with the name of the processing module, "ophys". Then, we can access the *Fluorescence* object inside of the "ophys" processing module by indexing it with the name of the *Fluorescence* object. The default name of *Fluorescence* objects is "Fluorescence". Finally, we can access the *RoiResponseSeries* object inside the *Fluorescence* object by indexing it with the name of the *RoiResponseSeries* object, which we named "RoiResponseSeries".

```
with NWBHDF5IO("ophys_tutorial.nwb", "r") as io:
    read_nwbfile = io.read()
    print(read_nwbfile.acquisition["TwoPhotonSeries1"])
    print(read_nwbfile.processing["ophys"])
    print(read_nwbfile.processing["ophys"]["Fluorescence"])
    print(read_nwbfile.processing["ophys"]["Fluorescence"]["RoiResponseSeries"])
```

#### Accessing your data

Data arrays are read passively from the file. Calling the data attribute on a *TimeSeries* such as a *RoiResponseSeries* does not read the data values, but presents an `h5py` object that can be indexed to read data. You can use the `[:]` operator to read the entire data array into memory. Load and print all the data values of the *RoiResponseSeries* object representing the fluorescence data.

```
with NWBHDF5IO("ophys_tutorial.nwb", "r") as io:
    read_nwbfile = io.read()

    print(read_nwbfile.acquisition["TwoPhotonSeries1"])
    print(read_nwbfile.processing["ophys"]["Fluorescence"]["RoiResponseSeries"].data[:])
```

#### Accessing data regions

It is often preferable to read only a portion of the data. To do this, index or slice into the data attribute just like if you were indexing or slicing a `numpy` array.

The following code prints elements `0:10` in the first dimension (time) and `0:3` (ROIs) in the second dimension from the fluorescence data we have written.

```
with NWBHDF5IO("ophys_tutorial.nwb", "r") as io:
    read_nwbfile = io.read()

    roi_resp_series = read_nwbfile.processing["ophys"]["Fluorescence"][
        "RoiResponseSeries"
    ]

    print("section of fluorescence responses:")
    print(roi_resp_series.data[0:10, 0:3])
```

### 2.2.3 Intracellular Electrophysiology

The following tutorial describes storage of intracellular electrophysiology data in NWB. NWB supports storage of the time series describing the stimulus and response, information about the electrode and device used, as well as metadata about the organization of the experiment.

---

**Note:** For a video tutorial on intracellular electrophysiology in NWB see also the Intracellular electrophysiology basics in NWB and Intracellular ephys metadata tutorials as part of the NWB Course at the INCF Training Space.

---

Recordings of intracellular electrophysiology stimuli and responses are represented with subclasses of *PatchClampSeries* using the *IntracellularElectrode* and *Device* type to describe the electrode and device used.

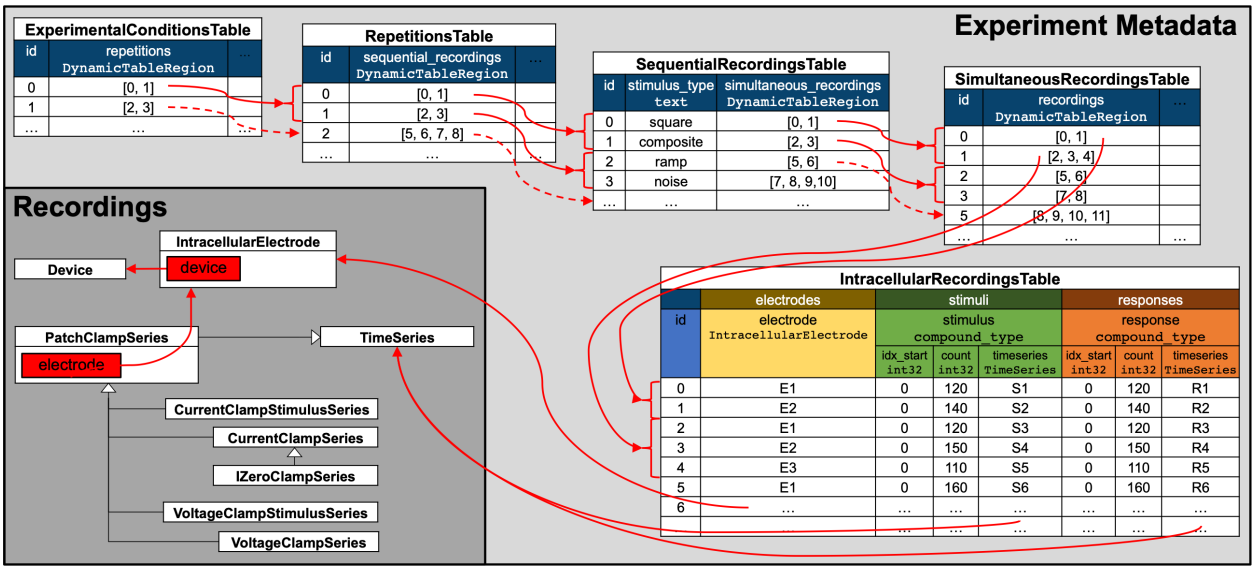



Fig. 1: Illustration of the hierarchy of metadata tables used to describe the organization of intracellular electrophysiology experiments.

To describe the organization of intracellular experiments, the metadata is organized hierarchically in a sequence of tables. All of the tables are so-called DynamicTables enabling users to add columns for custom metadata.

- *IntracellularRecordingsTable* relates electrode, stimulus and response pairs and describes metadata specific to individual recordings.
- *SimultaneousRecordingsTable* groups intracellular recordings from the *IntracellularRecordingsTable* together that were recorded simultaneously from different electrodes and/or cells and describes metadata that is constant across the simultaneous recordings. In practice a simultaneous recording is often also referred to as a sweep.
- *SequentialRecordingsTable* groups simultaneously recorded intracellular recordings from the *SimultaneousRecordingsTable* together and describes metadata that is constant across the simultaneous recordings. In practice a sequential recording is often also referred to as a sweep sequence. A common use of sequential recordings is to group together simultaneous recordings where a sequence of stimuli of the same type with varying parameters have been presented in a sequence (e.g., a sequence of square waveforms with varying amplitude).
- *RepetitionsTable* groups sequential recordings from the *SequentialRecordingsTable*. In practice a repetition is often also referred to a run. A typical use of the *RepetitionsTable* is to group sets of different stimuli that are applied in sequence that may be repeated.
- *ExperimentalConditionsTable* groups repetitions of intracellular recording from the *RepetitionsTable* together that belong to the same experimental conditions.

Storing data in hierarchical tables has the advantage that it allows us to avoid duplication of metadata. E.g., for a single experiment we only need to describe the metadata that is constant across an experimental condition as a single row in the *ExperimentalConditionsTable* without having to replicate the same information across all repetitions and sequential-, simultaneous-, and individual intracellular recordings. For analysis, this means that we can easily focus on individual aspects of an experiment while still being able to easily access information about information from related tables.

**Note:** All of the above mentioned metadata tables are optional and are created automatically by the *NWBFile* class the first time data is being added to a table via the corresponding add functions. However, as tables at higher levels

of the hierarchy link to the other tables that are lower in the hierarchy, we may only exclude tables from the top of the hierarchy. This means, for example, a file containing a *SimultaneousRecordingsTable* then must also always contain a corresponding *IntracellularRecordingsTable*.

### Imports used in the tutorial

```
# Standard Python imports
from datetime import datetime
from uuid import uuid4

import numpy as np
import pandas
from dateutil.tz import tzlocal

# Set pandas rendering option to avoid very wide tables in the html docs
pandas.set_option("display.max_colwidth", 30)
pandas.set_option("display.max_rows", 10)

# Import I/O class used for reading and writing NWB files
# Import main NWB file class
from pynwb import NWBHDF5IO, NWBFile

# Import additional core datatypes used in the example
from pynwb.core import DynamicTable, VectorData
from pynwb.base import TimeSeriesReference, TimeSeriesReferenceVectorData

# Import icephys TimeSeries types used
from pynwb.icephys import VoltageClampSeries, VoltageClampStimulusSeries
```

### A brief example

The following brief example provides a quick overview of the main steps to create an NWBFile for intracelluar electrophysiology data. We then discuss the individual steps in more detail afterwards.

```
# Create an ICEphysFile
ex_nwbfile = NWBFile(
    session_description="my first synthetic recording",
    identifier=str(uuid4()),
    session_start_time=datetime.now(tzlocal()),
    experimenter="Baggins, Bilbo",
    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",
)

# Add a device
ex_device = ex_nwbfile.create_device(name="Heka ITC-1600")
```

(continues on next page)

(continued from previous page)

```
# Add an intracellular electrode
ex_electrode = ex_nwbfile.create_icephys_electrode(
    name="elec0", description="a mock intracellular electrode", device=ex_device
)

# Create an ic-ephys stimulus
ex_stimulus = VoltageClampStimulusSeries(
    name="stimulus",
    data=[1, 2, 3, 4, 5],
    starting_time=123.6,
    rate=10e3,
    electrode=ex_electrode,
    gain=0.02,
)

# Create an ic-response
ex_response = VoltageClampSeries(
    name="response",
    data=[0.1, 0.2, 0.3, 0.4, 0.5],
    conversion=1e-12,
    resolution=np.nan,
    starting_time=123.6,
    rate=20e3,
    electrode=ex_electrode,
    gain=0.02,
    capacitance_slow=100e-12,
    resistance_comp_correction=70.0,
)

# (A) Add an intracellular recording to the file
#     NOTE: We can optionally define time-ranges for the stimulus/response via
#     the corresponding optional _start_index and _index_count parameters.
#     NOTE: It is allowed to add a recording with just a stimulus or a response
#     NOTE: We can  add custom columns to any of our tables in steps (A)-(E)
ex_ir_index = ex_nwbfile.add_intracellular_recording(
    electrode=ex_electrode, stimulus=ex_stimulus, response=ex_response
)

# (B) Add a list of sweeps to the simultaneous recordings table
ex_sweep_index = ex_nwbfile.add_icephys_simultaneous_recording(
    recordings=[
        ex_ir_index,
    ]
)

# (C) Add a list of simultaneous recordings table indices as a sequential recording
ex_sequence_index = ex_nwbfile.add_icephys_sequential_recording(
    simultaneous_recordings=[
        ex_sweep_index,
    ],
    stimulus_type="square",
)
```

(continues on next page)

(continued from previous page)

```
# (D) Add a list of sequential recordings table indices as a repetition
run_index = ex_nwbfile.add_icephys_repetition(
    sequential_recordings=[
        ex_sequence_index,
    ]
)

# (E) Add a list of repetition table indices as a experimental condition
ex_nwbfile.add_icephys_experimental_condition(
    repetitions=[
        run_index,
    ]
)

# Write our test file
ex_testpath = "ex_test_icephys_file.nwb"
with NWBHDF5IO(ex_testpath, "w") as io:
    io.write(ex_nwbfile)

# Read the data back in
with NWBHDF5IO(ex_testpath, "r") as io:
    infile = io.read()

# Optionally plot the organization of our example NWB file
try:
    import matplotlib.pyplot as plt
    from hdmf_docutils.doctools.render import (
        HierarchyDescription,
        NXGraphHierarchyDescription,
    )

    ex_file_hierarchy = HierarchyDescription.from_hdf5(ex_testpath)
    ex_file_graph = NXGraphHierarchyDescription(ex_file_hierarchy)
    ex_fig = ex_file_graph.draw(
        show_plot=False,
        figsize=(12, 16),
        label_offset=(0.0, 0.0065),
        label_font_size=10,
    )
    plt.show()
except ImportError:  # ignore in case hdmf_docutils is not installed
    pass
```

Now that we have seen a brief example, we are going to start from the beginning and go through each of the steps in more detail in the following sections.

### Creating an NWB file for Intracellular electrophysiology

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

```
# Create the file
nwbfile = NWBFile(
    session_description="my first synthetic recording",
    identifier=str(uuid4()),
    session_start_time=datetime.now(tzlocal()),
    experimenter=[
        "Baggins, Bilbo",
    ],
    lab="Bag End Laboratory",
    institution="University of Middle Earth at the Shire",
    experiment_description="I went on an adventure to reclaim vast treasures.",
    session_id="LONELYMTN001",
)
```

### Device metadata

Device metadata is represented by *Device* objects. To create a device, you can use the *NWBFile* 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_icephys_electrode*.

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

### Stimulus and response data

Intracellular stimulus and response data are represented with subclasses of *PatchClampSeries*. A stimulus is described by a time series representing voltage or current stimulation with a particular set of parameters. There are two classes for representing stimulus data:

- *VoltageClampStimulusSeries*
- *CurrentClampStimulusSeries*

The response is then described by a time series representing voltage or current recorded from a single cell using a single intracellular electrode via one of the following classes:

- *VoltageClampSeries*
- *CurrentClampSeries*
- *IZeroClampSeries*

Below we create a simple example stimulus/response recording data pair.

```
# Create an example icephys stimulus.
stimulus = VoltageClampStimulusSeries(
    name="ccss",
    data=[1, 2, 3, 4, 5],
    starting_time=123.6,
    rate=10e3,
    electrode=electrode,
    gain=0.02,
    sweep_number=np.uint64(15),
)

# Create and icephys response
response = VoltageClampSeries(
    name="vcs",
    data=[0.1, 0.2, 0.3, 0.4, 0.5],
    conversion=1e-12,
    resolution=np.nan,
    starting_time=123.6,
    rate=20e3,
    electrode=electrode,
    gain=0.02,
    capacitance_slow=100e-12,
    resistance_comp_correction=70.0,
    sweep_number=np.uint64(15),
)
```

### Adding an intracellular recording

As mentioned earlier, intracellular recordings are organized in the *IntracellularRecordingsTable* which relates electrode, stimulus and response pairs and describes metadata specific to individual recordings.

We can add an intracellular recording to the file via *add_intracellular_recording*. The function will record the data in the *IntracellularRecordingsTable* and add the given electrode, stimulus, or response to the NWBFile object if necessary. Any time we add a row to one of our tables, the corresponding add function (here *add_intracellular_recording*) returns the integer index of the newly created row. The `rowindex` is used in subsequent tables that reference rows in our table.

```
rowindex = nwbfile.add_intracellular_recording(
    electrode=electrode, stimulus=stimulus, response=response, id=10
)
```

**Note:** Since *add_intracellular_recording* can automatically add the objects to the NWBFile we do not need to separately call *add_stimulus* and *add_acquisition* to add our stimulus and response, but it is still fine to do so.

**Note:** The `id` parameter in the call is optional and if the `id` is omitted then PyNWB will automatically number recordings in sequences (i.e., id is the same as the rowindex)

**Note:** The IntracellularRecordigns, SimultaneousRecordings, SequentialRecordingsTable, RepetitionsTable and Ex-

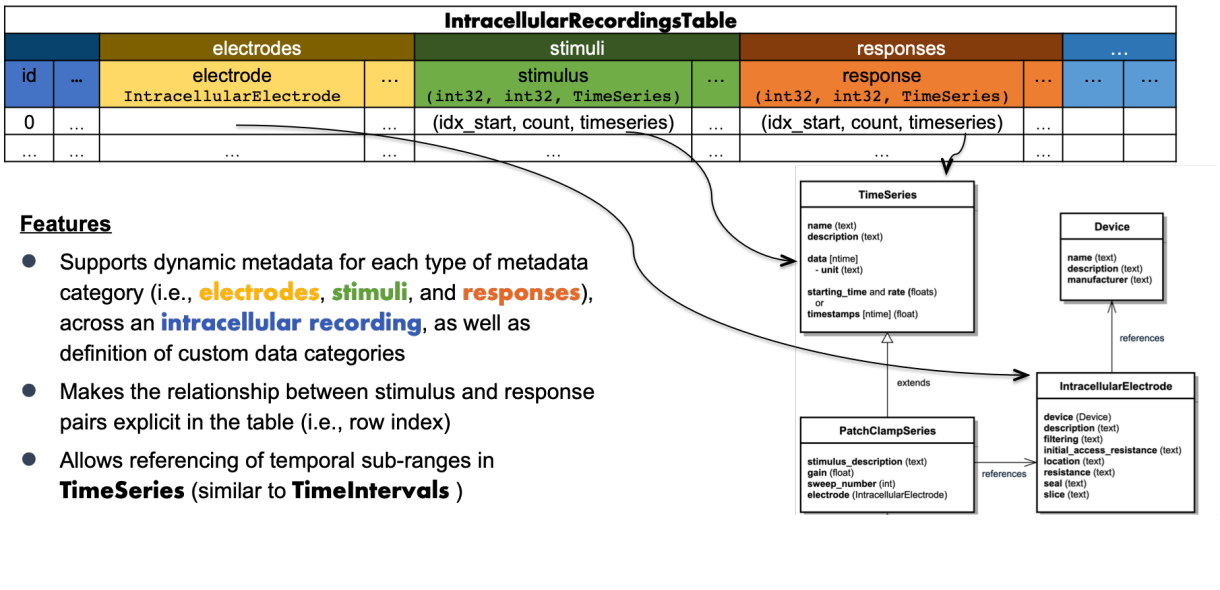



Fig. 2: Illustration of the structure of the IntracellularRecordingsTable

perimentalConditionsTable tables all enforce unique ids when adding rows. I.e., adding an intracellular recording with the same id twice results in a ValueError.

**Note:** We may optionally also specify the relevant time range for a stimulus and/or response as part of the intracellular_recording. This is useful, e.g., in case where the recording of the stimulus and response do not align (e.g., in case the recording of the response started before the recording of the stimulus).

```
rowindex2 = nwbfile.add_intracellular_recording(
    electrode=electrode,
    stimulus=stimulus,
    stimulus_start_index=1,
    stimulus_index_count=3,
    response=response,
    response_start_index=2,
    response_index_count=3,
    id=11,
)
```

**Note:** A recording may optionally also consist of just an electrode and stimulus or electrode and response, but at least one of stimulus or response is required. If either stimulus or response is missing, then the stimulus and response are internally set to the same TimeSeries and the start_index and index_count for the missing parameter are set to -1. When retrieving data from the *TimeSeriesReferenceVectorData*, the missing values will be represented via masked numpy arrays, i.e., as masked values in a `numpy.ma.masked_array` or as a `np.ma.core.MaskedConstant`.

```
rowindex3 = nwbfile.add_intracellular_recording(
    electrode=electrode, response=response, id=12
)
```

**Warning:** For brevity we reused in the above example the same response and stimulus in all rows of the intracellular_recordings. While this is allowed, in most practical cases the stimulus and response will change between intracellular_recordings.

### Adding custom columns to the intracellular recordings table

We can add a column to the main intracellular recordings table as follows.

```
nwbfile.intracellular_recordings.add_column(
    name="recording_tag",
    data=["A1", "A2", "A3"],
    description="String with a recording tag",
)
```

The *IntracellularRecordingsTable* table is not just a DynamicTable but an AlignedDynamicTable. The AlignedDynamicTable type is itself a DynamicTable that may contain an arbitrary number of additional DynamicTable, each of which defines a "category". This is similar to a table with "sub-headings". In the case of the *IntracellularRecordingsTable*, we have three predefined categories, i.e., electrodes, stimuli, and responses. We can also dynamically add new categories to the table. As each category corresponds to a DynamicTable, this means we have to create a new DynamicTable and add it to our table.

```
# Create a new DynamicTable for our category that contains a location column of type␣
↪VectorData
location_column = VectorData(
    name="location",
    data=["Mordor", "Gondor", "Rohan"],
    description="Recording location in Middle Earth",
)

lab_category = DynamicTable(
    name="recording_lab_data",
    description="category table for lab-specific recording metadata",
    colnames=[
        "location",
    ],
    columns=[
        location_column,
    ],
)
# Add the table as a new category to our intracellular_recordings
nwbfile.intracellular_recordings.add_category(category=lab_category)
# Note, the name of the category is name of the table, i.e., 'recording_lab_data'
```

**Note:** In an AlignedDynamicTable all category tables MUST align with the main table, i.e., all tables must have the same number of rows and rows are expected to correspond to each other by index

We can also add custom columns to any of the subcategory tables, i.e., the electrodes, stimuli, and responses tables, and any custom subcategory tables. All we need to do is indicate the name of the category we want to add the column to.

```
nwbfile.intracellular_recordings.add_column(
    name="voltage_threshold",
    data=[0.1, 0.12, 0.13],
    description="Just an example column on the electrodes category table",
    category="electrodes",
)
```

### Adding stimulus templates

One predefined subcategory column is the `stimulus_template` column in the stimuli table. This column is used to store template waveforms of stimuli in addition to the actual recorded stimulus that is stored in the `stimulus` column. The `stimulus_template` column contains an idealized version of the template waveform used as the stimulus. This can be useful as a noiseless version of the stimulus for data analysis or to validate that the recorded stimulus matches the expected waveform of the template. Similar to the `stimulus` and `response` columns, we can specify a relevant time range.

```
stimulus_template = VoltageClampStimulusSeries(
    name="ccst",
    data=[0, 1, 2, 3, 4],
    starting_time=0.0,
    rate=10e3,
    electrode=electrode,
    gain=0.02,
)
nwbfile.add_stimulus_template(stimulus_template)

nwbfile.intracellular_recordings.add_column(
    name="stimulus_template",
    data=[TimeSeriesReference(0, 5, stimulus_template),  # (start_index, index_count,␣
↪stimulus_template)
          TimeSeriesReference(1, 3, stimulus_template),
          TimeSeriesReference.empty(stimulus_template)],  # if there was no data for␣
↪that recording, use empty reference
    description="Column storing the reference to the stimulus template for the recording␣
↪(rows).",
    category="stimuli",
    col_cls=TimeSeriesReferenceVectorData
)

# we can also add stimulus template data as follows
rowindex = nwbfile.add_intracellular_recording(
    electrode=electrode,
    stimulus=stimulus,
    stimulus_template=stimulus_template,  # the full time range of the stimulus template␣
↪will be used unless specified
    recording_tag='A4',
    recording_lab_data={'location': 'Isengard'},
    electrode_metadata={'voltage_threshold': 0.14},
    id=13,
)
```

**Note:** If a stimulus template column exists but there is no stimulus template data for that recording, then *add_intracellular_recording* will internally set the stimulus template to the provided stimulus or response TimeSeries and the start_index and index_count for the missing parameter are set to -1. The missing values will be represented via masked numpy arrays.

**Note:** Since stimulus templates are often reused across many recordings, the timestamps in the templates are not usually aligned with the recording nor with the reference time of the file. The timestamps often start at 0 and are relative to the time of the application of the stimulus.

### Add a simultaneous recording

Before adding a simultaneous recording, we will take a brief discourse to illustrate how we can add custom columns to tables before and after we have populated the table with data

### Define a custom column for a simultaneous recording before populating the table

Before we add a simultaneous recording, let's create a custom data column in our *SimultaneousRecordingsTable*. We can create columns at the beginning (i.e., before we populate the table with rows/data) or we can add columns after we have already populated the table with rows. Here we will show the former. For this, we first need to get access to our table.

```
print(nwbfile.icephys_simultaneous_recordings)
```

```
None
```

The *SimultaneousRecordingsTable* is optional, and since we have not populated it with any data yet, we can see that the table does not actually exist yet. In order to make sure the table is being created we can use *get_icephys_simultaneous_recordings*, which ensures that the table is being created if it does not exist yet.

```
icephys_simultaneous_recordings = nwbfile.get_icephys_simultaneous_recordings()
icephys_simultaneous_recordings.add_column(
    name="simultaneous_recording_tag",
    description="A custom tag for simultaneous_recordings",
)
print(icephys_simultaneous_recordings.colnames)
```

```
('recordings', 'simultaneous_recording_tag')
```

As we can see, we now have successfully created a new custom column.

**Note:** The same process applies to all our other tables as well. We can use the corresponding *get_intracellular_recordings*, *get_icephys_sequential_recordings*, *get_icephys_repetitions* functions instead. In general, we can always use the get functions instead of accessing the property of the file.

### Add a simultaneous recording

Add a single simultaneous recording consisting of a set of intracellular recordings. Again, setting the id for a simultaneous recording is optional. The recordings argument of the *add_icephys_simultaneous_recording* function here is simply a list of ints with the indices of the corresponding rows in the *IntracellularRecordingsTable*

---

**Note:** Since we created our custom `simultaneous_recording_tag column` earlier, we now also need to populate this custom field for every row we add to the *SimultaneousRecordingsTable*.

---

```
rowindex = nwbfile.add_icephys_simultaneous_recording(
    recordings=[rowindex, rowindex2, rowindex3],
    id=12,
    simultaneous_recording_tag="LabTag1",
)
```

---

**Note:** The `recordings` argument is the list of indices of the rows in the *IntracellularRecordingsTable* that we want to reference. The indices are determined by the order in which we added the elements to the table. If we don't know the row indices, but only the ids of the relevant intracellular recordings, then we can search for them as follows:

---

```
temp_row_indices = nwbfile.intracellular_recordings.id == [10, 11]
print(temp_row_indices)
```

```
[0 1]
```

---

**Note:** The same is true for our other tables as well, i.e., referencing is done always by indices of rows (NOT ids). If we only know ids then we can search for them in the same manner on the other tables as well, e.g,. via `nwbfile.simultaneous_recordings.id == 15`. In the search we can use a list of integer ids or a single int.

---

### Define a custom column for a simultaneous recording after adding rows

Depending on the lab workflow, it may be useful to add complete columns to a table after we have already populated the table with rows. We can do this the same way as before, only now we need to provide a data array to populate the values for the existing rows. E.g.:

```
nwbfile.icephys_simultaneous_recordings.add_column(
    name="simultaneous_recording_type",
    description="Description of the type of simultaneous_recording",
    data=[
        "SimultaneousRecordingType1",
    ],
)
```

### Add a sequential recording

Add a single sequential recording consisting of a set of simultaneous recordings. Again, setting the id for a sequential recording is optional. Also this table is optional and will be created automatically by NWBFile. The `simultaneous_recordings` argument of the *add_icephys_sequential_recording* function here is simply a list of ints with the indices of the corresponding rows in the *SimultaneousRecordingsTable*.

```
rowindex = nwbfile.add_icephys_sequential_recording(
    simultaneous_recordings=[0], stimulus_type="square", id=15
)
```

### Add a repetition

Add a single repetition consisting of a set of sequential recordings. Again, setting the id for a repetition is optional. Also this table is optional and will be created automatically by NWBFile. The `sequential_recordings argument` of the *add_icephys_repetition* function here is simply a list of ints with the indices of the corresponding rows in the *SequentialRecordingsTable*.

```
rowindex = nwbfile.add_icephys_repetition(sequential_recordings=[0], id=17)
```

### Add an experimental condition

Add a single experimental condition consisting of a set of repetitions. Again, setting the id for a condition is optional. Also this table is optional and will be created automatically by NWBFile. The `repetitions` argument of the *add_icephys_experimental_condition* function again is simply a list of ints with the indices of the correspondingto rows in the *RepetitionsTable*.

```
rowindex = nwbfile.add_icephys_experimental_condition(repetitions=[0], id=19)
```

As mentioned earlier, to add additional columns to any of the tables, we can use the `.add_column` function on the corresponding table after they have been created.

```
nwbfile.icephys_experimental_conditions.add_column(
    name="tag",
    data=np.arange(1),
    description="integer tag for a experimental condition",
)
```

When we add new items, then we now also need to set the values for the new column, e.g.:

```
rowindex = nwbfile.add_icephys_experimental_condition(repetitions=[0], id=21, tag=3)
```

### Read/write the NWBFile

```
# Write our test file
testpath = "test_icephys_file.nwb"
with NWBHDF5IO(testpath, "w") as io:
    io.write(nwbfile)

# Read the data back in
with NWBHDF5IO(testpath, "r") as io:
    infile = io.read()
```

### Accessing the tables

All of the icephys metadata tables are available as attributes on the NWBFile object. For display purposes, we convert the tables to pandas DataFrames to show their content. For a more in-depth discussion of how to access and use the tables, see the tutorial on *Query Intracellular Electrophysiology Metadata*.

```
pandas.set_option("display.max_columns", 6)  # avoid oversize table in the html docs
nwbfile.intracellular_recordings.to_dataframe()
```

```
# optionally we can ignore the id columns of the category subtables
pandas.set_option("display.max_columns", 5)  # avoid oversize table in the html docs
nwbfile.intracellular_recordings.to_dataframe(ignore_category_ids=True)
```

```
nwbfile.icephys_simultaneous_recordings.to_dataframe()
```

```
nwbfile.icephys_sequential_recordings.to_dataframe()
```

```
nwbfile.icephys_repetitions.to_dataframe()
```

```
nwbfile.icephys_experimental_conditions.to_dataframe()
```

### Validate data

This section is for internal testing purposes only to validate that the roundtrip of the data (i.e., generate –> write –> read) produces the correct results.

```
# Read the data back in
with NWBHDF5IO(testpath, "r") as io:
    infile = io.read()

    # assert intracellular_recordings
    assert np.all(
        infile.intracellular_recordings.id[:] == nwbfile.intracellular_recordings.id[:]
    )
```

(continues on next page)

(continued from previous page)

```
    # Assert that the ids and the VectorData, VectorIndex, and table target of the
    # recordings column of the Sweeps table are correct
    assert np.all(
        infile.icephys_simultaneous_recordings.id[:]
        == nwbfile.icephys_simultaneous_recordings.id[:]
    )
    assert np.all(
        infile.icephys_simultaneous_recordings["recordings"].target.data[:]
        == nwbfile.icephys_simultaneous_recordings["recordings"].target.data[:]
    )
    assert np.all(
        infile.icephys_simultaneous_recordings["recordings"].data[:]
        == nwbfile.icephys_simultaneous_recordings["recordings"].data[:]
    )
    assert (
        infile.icephys_simultaneous_recordings["recordings"].target.table.name
        == nwbfile.icephys_simultaneous_recordings["recordings"].target.table.name
    )

    # Assert that the ids and the VectorData, VectorIndex, and table target of the␣
↪simultaneous
    #  recordings column of the SweepSequences table are correct
    assert np.all(
        infile.icephys_sequential_recordings.id[:]
        == nwbfile.icephys_sequential_recordings.id[:]
    )
    assert np.all(
        infile.icephys_sequential_recordings["simultaneous_recordings"].target.data[:]
        == nwbfile.icephys_sequential_recordings["simultaneous_recordings"].target.data[
            :
        ]
    )
    assert np.all(
        infile.icephys_sequential_recordings["simultaneous_recordings"].data[:]
        == nwbfile.icephys_sequential_recordings["simultaneous_recordings"].data[:]
    )
    assert (
        infile.icephys_sequential_recordings[
            "simultaneous_recordings"
        ].target.table.name
        == nwbfile.icephys_sequential_recordings[
            "simultaneous_recordings"
        ].target.table.name
    )

    # Assert that the ids and the VectorData, VectorIndex, and table target of the
    # sequential_recordings column of the Repetitions table are correct
    assert np.all(infile.icephys_repetitions.id[:] == nwbfile.icephys_repetitions.id[:])
    assert np.all(
        infile.icephys_repetitions["sequential_recordings"].target.data[:]
        == nwbfile.icephys_repetitions["sequential_recordings"].target.data[:]
```

(continues on next page)

(continued from previous page)

```
)
assert np.all(
    infile.icephys_repetitions["sequential_recordings"].data[:]
    == nwbfile.icephys_repetitions["sequential_recordings"].data[:]
)
assert (
    infile.icephys_repetitions["sequential_recordings"].target.table.name
    == nwbfile.icephys_repetitions["sequential_recordings"].target.table.name
)

# Assert that the ids and the VectorData, VectorIndex, and table target of the
# repetitions column of the Conditions table are correct
assert np.all(
    infile.icephys_experimental_conditions.id[:]
    == nwbfile.icephys_experimental_conditions.id[:]
)
assert np.all(
    infile.icephys_experimental_conditions["repetitions"].target.data[:]
    == nwbfile.icephys_experimental_conditions["repetitions"].target.data[:]
)
assert np.all(
    infile.icephys_experimental_conditions["repetitions"].data[:]
    == nwbfile.icephys_experimental_conditions["repetitions"].data[:]
)
assert (
    infile.icephys_experimental_conditions["repetitions"].target.table.name
    == nwbfile.icephys_experimental_conditions["repetitions"].target.table.name
)
assert np.all(
    infile.icephys_experimental_conditions["tag"][:]
    == nwbfile.icephys_experimental_conditions["tag"][:]
)
```

### 2.2.4 Query Intracellular Electrophysiology Metadata

This tutorial focuses on using pandas to query experiment metadata for intracellular electrophysiology experiments using the metadata tables from the `icephys` module. See the *Intracellular Electrophysiology* tutorial for an introduction to the intracellular electrophysiology metadata tables and how to create an NWBFile for intracellular electrophysiology data.

---

**Note:** To enhance display of large pandas DataFrames, we save and render large tables as images in this tutorial. Simply click on the rendered table to view the full-size image.

---

### Imports used in the tutorial

```
import os
```

Settings for improving rendering of tables in the online tutorial

```
import dataframe_image

# Standard Python imports
import numpy as np
import pandas

# Get the path to the this tutorial
try:
    tutorial_path = os.path.abspath(__file__)  # when running as a .py
except NameError:
    tutorial_path = os.path.abspath("__file__")  # when running as a script or notebook
# directory to save rendered dataframe images for display
df_basedir = os.path.abspath(
    os.path.join(
        os.path.dirname(tutorial_path), "../../source/tutorials/domain/images/"
    )
)
# Create the image directory. This is necessary only for gallery tests on GitHub
# but not for normal doc builds the output path already exists
os.makedirs(df_basedir, exist_ok=True)
# Set rendering options for tables
pandas.set_option("display.max_colwidth", 30)
pandas.set_option("display.max_rows", 10)
pandas.set_option("display.max_columns", 6)
pandas.set_option("display.colheader_justify", "right")
dfi_fontsize = 7  # Fontsize to use when rendering with dataframe_image
```

### Example setup

Generate a simple example NWBFile with dummy intracellular electrophysiology data. This example uses a utility function *create_icephys_testfile* to create a dummy NWB file with random icephys data.

```
from pynwb.testing.icephys_testutils import create_icephys_testfile

test_filename = "icephys_pandas_testfile.nwb"
nwbfile = create_icephys_testfile(
    filename=test_filename,  # Write the file to disk for testing
    add_custom_columns=True,  # Add a custom column to each metadata table
    randomize_data=True,  # Randomize the data in the simulus and response
    with_missing_stimulus=True,  # Don't include the stimulus for row 0 and 10
)
```

### Accessing the ICEphys metadata tables

#### Get the parent metadata table

The intracellular electrophysiology metadata consists of a hierarchy of DynamicTables, i.e., *ExperimentalConditionsTable* –> *RepetitionsTable* –> *SequentialRecordingsTable* –> *SimultaneousRecordingsTable* –> *IntracellularRecordingsTable*. However, in a given *NWBFile*, not all tables may exist - a user may choose to exclude tables from the top of the hierarchy (e.g., a file may only contain *SimultaneousRecordingsTable* and *IntracellularRecordingsTable* while omitting all of the other tables that are higher in the hierarchy). To provide a consistent interface for users, PyNWB allows us to easily locate the table that defines the root of the table hierarchy via the function *get_icephys_meta_parent_table*.

```
root_table = nwbfile.get_icephys_meta_parent_table()
print(root_table.neurodata_type)
```

```
ExperimentalConditionsTable
```

#### Getting a specific ICEphys metadata table

We can retrieve any of the ICEphys metadata tables via the corresponding properties of NWBFile, i.e., *intracellular_recordings*, *icephys_simultaneous_recordings*, *icephys_sequential_recordings*, *icephys_repetitions*, *icephys_experimental_conditions*. The property will be `None` if the file does not contain the corresponding table. As such we can also easily check if a NWBFile contains a particular ICEphys metadata table via, e.g.:

```
nwbfile.icephys_sequential_recordings is not None
```

```
True
```

**Warning:** Always use the *NWBFile* properties rather than the corresponding get methods if you only want to retrieve the ICEphys metadata tables. The get methods (e.g., *get_icephys_simultaneous_recordings*) are designed to always return a corresponding ICEphys metadata table for the file and will automatically add the missing table (and all required tables that are lower in the hierarchy) to the file. This behavior is to ease populating the ICEphys metadata tables when creating or updating an *NWBFile*.

#### Inspecting the table hierarchy

For any given table we can further check if and which columns are foreign `DynamicTableRegion` columns pointing to other tables via the the `has_foreign_columns` and `get_foreign_columns`, respectively.

```
print("Has Foreign Columns:", root_table.has_foreign_columns())
print("Foreign Columns:", root_table.get_foreign_columns())
```

```
Has Foreign Columns: True
Foreign Columns: ['repetitions']
```

Using `get_linked_tables` we can then also look at all links defined directly or indirectly from a given table to other tables. The result is a `list` of `typing.NamedTuple` objects containing, for each found link, the:

- *"source_table"* DynamicTable object,
- *"source_column"* DynamicTableRegion column from the source table, and
- *"target_table"* DynamicTable (which is the same as *source_column.table*).

```
linked_tables = root_table.get_linked_tables()

# Print the links
for i, link in enumerate(linked_tables):
    print(
        "%s (%s, %s) ----> %s"
        % (
            "    " * i,
            link.source_table.name,
            link.source_column.name,
            link.target_table.name,
        )
    )
```

```
(experimental_conditions, repetitions) ----> repetitions
    (repetitions, sequential_recordings) ----> sequential_recordings
        (sequential_recordings, simultaneous_recordings) ----> simultaneous_recordings
            (simultaneous_recordings, recordings) ----> intracellular_recordings
```

### Converting ICEphys metadata tables to pandas DataFrames

#### Using nested DataFrames

Using the to_dataframe method we can easily convert tables to pandas DataFrames.

```
exp_cond_df = root_table.to_dataframe()
exp_cond_df
```

By default, the method will resolve DynamicTableRegion references and include the rows that are referenced in related tables as DataFrame objects, resulting in a hierarchically nested DataFrame. For example, looking at a single cell of the repetitions column of our *ExperimentalConditionsTable* table, we get the corresponding subset of repetitions from the py:class:*~pynwb.icephys.RepetitionsTable*.

```
exp_cond_df.iloc[0]["repetitions"]
```

In contrast to the other ICEphys metadata tables, the *IntracellularRecordingsTable* does not contain any DynamicTableRegion columns, but it is a AlignedDynamicTable which contains sub-tables for electrodes, stimuli, and responses. For convenience, the *to_dataframe* of the *IntracellularRecordingsTable* provides a few additional optional parameters to ignore the ids of the category tables (via ignore_category_ids=True) or to convert the electrode, stimulus, and response references to ObjectIds. For example:

```
ir_df = nwbfile.intracellular_recordings.to_dataframe(
    ignore_category_ids=True,
    electrode_refs_as_objectids=True,
    stimulus_refs_as_objectids=True,
    response_refs_as_objectids=True,
)
```

(continues on next page)

(continued from previous page)

```
# save the table as image to display in the docs
dataframe_image.export(
    obj=ir_df,
    filename=os.path.join(df_basedir, "intracellular_recordings_dataframe.png"),
    table_conversion="matplotlib",
    fontsize=dfi_fontsize,
)
```

| | intracellular_recordings | electrodes | stimuli | responses |
|---|---|---|---|---|
| | recording_tags | electrode | stimulus | response |
| (intracellular_recordings, id) | | | | |
| 0 | A1 | fa8da911-936c-4854-918b-df... | (None, None, None) | (0, 10, ef218ab8-fdbe-4b64... |
| 1 | A2 | e63ba930-a0f3-44ef-a50b-39... | (0, 10, 1264998e-6206-4950... | (0, 10, 67ebd51f-eae2-4cfc... |
| 2 | B1 | fa8da911-936c-4854-918b-df... | (0, 10, 11eeb86b-d1a3-463c... | (0, 10, 9e12ff0b-5565-4a0a... |
| 3 | B2 | e63ba930-a0f3-44ef-a50b-39... | (0, 10, 7367c177-1abb-4ebe... | (0, 10, e675e4ef-2361-4cfd... |
| 4 | C1 | fa8da911-936c-4854-918b-df... | (0, 10, 1fc2faa6-311e-4fae... | (0, 10, e7144784-3817-4d5f... |
| 5 | C2 | e63ba930-a0f3-44ef-a50b-39... | (0, 10, 8a470610-b50e-432f... | (0, 10, 053bc1f2-5383-47ea... |
| 6 | C3 | fa8da911-936c-4854-918b-df... | (0, 10, 8f87dedf-ecb3-4053... | (0, 10, 1338e017-6313-4dea... |
| 7 | D1 | e63ba930-a0f3-44ef-a50b-39... | (0, 10, ed410c72-c89e-432d... | (0, 10, 22b4d3c9-33cd-4257... |
| 8 | D2 | fa8da911-936c-4854-918b-df... | (0, 10, 3f4eb534-0ee8-4922... | (0, 10, f51c0e64-d758-4f61... |
| 9 | D3 | e63ba930-a0f3-44ef-a50b-39... | (0, 10, 6cdee884-1adf-4d46... | (0, 10, 8cbf8ee1-cd77-47eb... |
| 10 | A1 | fa8da911-936c-4854-918b-df... | (None, None, None) | (0, 10, 89d92762-0ed3-4de0... |
| 11 | A2 | e63ba930-a0f3-44ef-a50b-39... | (0, 10, 37300bcc-cba9-4d70... | (0, 10, 338bee58-cde7-4325... |
| 12 | B1 | fa8da911-936c-4854-918b-df... | (0, 10, 33b28ac8-0934-43dd... | (0, 10, 1a4a3136-7d9e-49db... |
| 13 | B2 | e63ba930-a0f3-44ef-a50b-39... | (0, 10, 46bae0da-22b6-4207... | (0, 10, b3ed4550-9dfd-4dbb... |
| 14 | C1 | fa8da911-936c-4854-918b-df... | (0, 10, e896c7a6-e930-4d83... | (0, 10, c2b9df9e-bfe1-41d2... |
| 15 | C2 | e63ba930-a0f3-44ef-a50b-39... | (0, 10, e03f45cb-6f9a-41a0... | (0, 10, d0f25710-4e56-4c9b... |
| 16 | C3 | fa8da911-936c-4854-918b-df... | (0, 10, 59e61255-ac41-4a5e... | (0, 10, 4c0e856c-6607-4eda... |
| 17 | D1 | e63ba930-a0f3-44ef-a50b-39... | (0, 10, 6aab5980-39d8-4470... | (0, 10, e1532a2e-a888-4bf0... |
| 18 | D2 | fa8da911-936c-4854-918b-df... | (0, 10, 617bc140-681d-4602... | (0, 10, 60713287-2a58-45de... |
| 19 | D3 | e63ba930-a0f3-44ef-a50b-39... | (0, 10, c39a42ba-683b-48be... | (0, 10, a00efc12-f366-4a79... |

### Using indexed DataFrames

Depending on the particular analysis, we may be interested only in a particular table and do not want to recursively load and resolve all the linked tables. By setting `index=True` when converting the table `to_dataframe` the `DynamicTableRegion` links will be represented as lists of integers indicating the rows in the target table (without loading data from the referenced table).

```
root_table.to_dataframe(index=True)
```

To resolve links related to a set of rows, we can then simply use the corresponding `DynamicTableRegion` column from our original table, e.g.:

```
root_table["repetitions"][
    0
]  # Look-up the repetitions for the first experimental condition
```

We can also naturally resolve links ourselves by looking up the relevant table and then accessing elements of the table directly.

```
# All DynamicTableRegion columns in the ICEphys table are indexed so we first need to
# follow the ".target" to the VectorData and then look up the table via ".table"
target_table = root_table["repetitions"].target.table
target_table[[0, 1]]
```

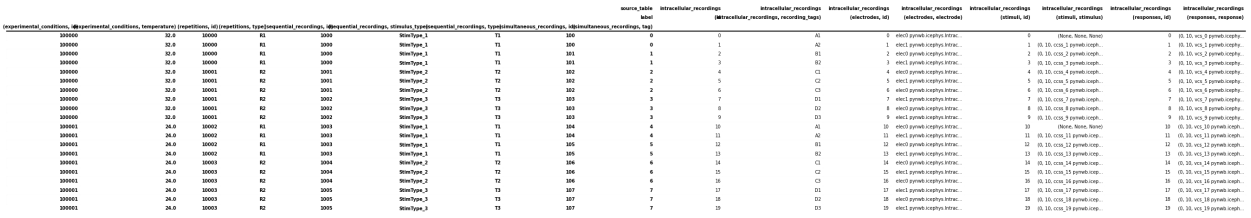
**Note:** We can also explicitly exclude the DynamicTableRegion columns (or any other column) from the DataFrame using e.g., `root_table.to_dataframe(exclude={'repetitions', })`.

### Using a single, hierarchical DataFrame

To gain a more direct overview of all metadata at once and avoid iterating across levels of nested DataFrames during analysis, it can be useful to flatten (or unnest) nested DataFrames, expanding the nested DataFrames by adding their columns to the main table, and expanding the corresponding rows in the parent table by duplicating the data from the existing columns across the new rows. For example, an experimental condition represented by a single row in the *ExperimentalConditionsTable* containing 5 repetitions would be expanded to 5 rows, each containing a copy of the metadata from the experimental condition along with the metadata of one of the repetitions. Repeating this process recursively, a single row in the *ExperimentalConditionsTable* will then ultimately expand to the total number of intracellular recordings from the *IntracellularRecordingsTable* that belong to the experimental conditions table.

HDMF povides several convenience functions to help with this process. Using the to_hierarchical_dataframe method, we can transform our hierarchical table into a single pandas DataFrame. To avoid duplication of data in the display, the hierarchy is represented as a pandas MultiIndex on the rows so that only the data from the last table in our hierarchy (i.e. here the *IntracellularRecordingsTable*) is represented as columns.

```
from hdmf.common.hierarchicaltable import to_hierarchical_dataframe

icephys_meta_df = to_hierarchical_dataframe(root_table)

# save table as image to display in the docs
dataframe_image.export(
    obj=icephys_meta_df,
    filename=os.path.join(df_basedir, "icephys_meta_dataframe.png"),
    table_conversion="matplotlib",
    fontsize=dfi_fontsize,
)
```



Depending on the analysis, it can be useful to further process our DataFrame. Using the standard reset_index function, we can turn the data from the MultiIndex to columns of the table itself, effectively denormalizing the display by repeating all data across rows. HDMF then also provides: 1) drop_id_columns to remove all "id" columns and 2) flatten_column_index to turn the MultiIndex on the columns of the table into a regular Index of tuples.

**Note:** Dropping `id` columns is often useful for visualization purposes while for query and analysis it is often useful to maintain the `id` columns to facilitate lookups and correlation of information.

```
from hdmf.common.hierarchicaltable import drop_id_columns, flatten_column_index

# Reset the index of the dataframe and turn the values into columns instead
icephys_meta_df.reset_index(inplace=True)
# Flatten the column-index, turning the pandas.MultiIndex into a pandas.Index of tuples
flatten_column_index(dataframe=icephys_meta_df, max_levels=2, inplace=True)
# Remove the id columns. By setting inplace=False allows us to visualize the result of␣
↪this
# action while keeping the id columns in our main icephys_meta_df table
drid_icephys_meta_df = drop_id_columns(dataframe=icephys_meta_df, inplace=False)

# save the table as image to display in the docs
dataframe_image.export(
    obj=drid_icephys_meta_df,
    filename=os.path.join(df_basedir, "icephys_meta_dataframe_drop_id.png"),
    table_conversion="matplotlib",
    fontsize=dfi_fontsize,
)
```

| | (experimental_conditions, temperature) | (repetitions, type) | (sequential_recordings, stimulus_type) | (sequential_recordings, type) | (simultaneous_recordings, tag) | (intracellular_recordings, recording_tags) | (electrodes, electrode) | (stimuli, stimulus) | (responses, response) |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 32.0 | R1 | StimType_1 | T1 | 0 | A1 | elec0 pynwb.icephys.Intrac... | (None, None, None) | (0, 10, vcs_0 pynwb.icephy... |
| 1 | 32.0 | R1 | StimType_1 | T1 | 0 | A2 | elec1 pynwb.icephys.Intrac... | (0, 10, ccss_1 pynwb.iceph... | (0, 10, vcs_1 pynwb.icephy... |
| 2 | 32.0 | R1 | StimType_1 | T1 | 1 | B1 | elec0 pynwb.icephys.Intrac... | (0, 10, ccss_2 pynwb.iceph... | (0, 10, vcs_2 pynwb.icephy... |
| 3 | 32.0 | R1 | StimType_1 | T1 | 1 | B2 | elec1 pynwb.icephys.Intrac... | (0, 10, ccss_3 pynwb.iceph... | (0, 10, vcs_3 pynwb.icephy... |
| 4 | 32.0 | R2 | StimType_2 | T2 | 2 | C1 | elec0 pynwb.icephys.Intrac... | (0, 10, ccss_4 pynwb.iceph... | (0, 10, vcs_4 pynwb.icephy... |
| 5 | 32.0 | R2 | StimType_2 | T2 | 2 | C2 | elec1 pynwb.icephys.Intrac... | (0, 10, ccss_5 pynwb.iceph... | (0, 10, vcs_5 pynwb.icephy... |
| 6 | 32.0 | R2 | StimType_2 | T2 | 2 | C3 | elec0 pynwb.icephys.Intrac... | (0, 10, ccss_6 pynwb.iceph... | (0, 10, vcs_6 pynwb.icephy... |
| 7 | 32.0 | R2 | StimType_3 | T3 | 3 | D1 | elec1 pynwb.icephys.Intrac... | (0, 10, ccss_7 pynwb.iceph... | (0, 10, vcs_7 pynwb.icephy... |
| 8 | 32.0 | R2 | StimType_3 | T3 | 3 | D2 | elec0 pynwb.icephys.Intrac... | (0, 10, ccss_8 pynwb.iceph... | (0, 10, vcs_8 pynwb.icephy... |
| 9 | 32.0 | R2 | StimType_3 | T3 | 3 | D3 | elec1 pynwb.icephys.Intrac... | (0, 10, ccss_9 pynwb.iceph... | (0, 10, vcs_9 pynwb.icephy... |
| 10 | 24.0 | R1 | StimType_1 | T1 | 4 | A1 | elec0 pynwb.icephys.Intrac... | (None, None, None) | (0, 10, vcs_10 pynwb.iceph... |
| 11 | 24.0 | R1 | StimType_1 | T1 | 4 | A2 | elec1 pynwb.icephys.Intrac... | (0, 10, ccss_11 pynwb.icep... | (0, 10, vcs_11 pynwb.iceph... |
| 12 | 24.0 | R1 | StimType_1 | T1 | 5 | B1 | elec0 pynwb.icephys.Intrac... | (0, 10, ccss_12 pynwb.icep... | (0, 10, vcs_12 pynwb.iceph... |
| 13 | 24.0 | R1 | StimType_1 | T1 | 5 | B2 | elec1 pynwb.icephys.Intrac... | (0, 10, ccss_13 pynwb.icep... | (0, 10, vcs_13 pynwb.iceph... |
| 14 | 24.0 | R2 | StimType_2 | T2 | 6 | C1 | elec0 pynwb.icephys.Intrac... | (0, 10, ccss_14 pynwb.icep... | (0, 10, vcs_14 pynwb.iceph... |
| 15 | 24.0 | R2 | StimType_2 | T2 | 6 | C2 | elec1 pynwb.icephys.Intrac... | (0, 10, ccss_15 pynwb.icep... | (0, 10, vcs_15 pynwb.iceph... |
| 16 | 24.0 | R2 | StimType_2 | T2 | 6 | C3 | elec0 pynwb.icephys.Intrac... | (0, 10, ccss_16 pynwb.icep... | (0, 10, vcs_16 pynwb.iceph... |
| 17 | 24.0 | R2 | StimType_3 | T3 | 7 | D1 | elec1 pynwb.icephys.Intrac... | (0, 10, ccss_17 pynwb.icep... | (0, 10, vcs_17 pynwb.iceph... |
| 18 | 24.0 | R2 | StimType_3 | T3 | 7 | D2 | elec0 pynwb.icephys.Intrac... | (0, 10, ccss_18 pynwb.icep... | (0, 10, vcs_18 pynwb.iceph... |
| 19 | 24.0 | R2 | StimType_3 | T3 | 7 | D3 | elec1 pynwb.icephys.Intrac... | (0, 10, ccss_19 pynwb.icep... | (0, 10, vcs_19 pynwb.iceph... |

### Useful additional data preparations

### Expanding TimeSeriesReference columns

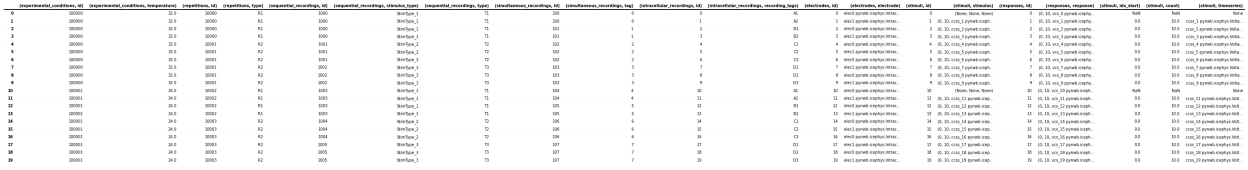
For query purposes it can be useful to expand the stimulus and response columns to separate the `(start, count, timeseries)` values in separate columns. This is primarily useful if we want to perform queries on these components directly, otherwise it is usually best to keep the stimulus/response references around as *:py:class:`~pynwb.base.TimeSeriesReference*, which provides additional features to inspect and validate the references and load data. We, therefore, here keep the data in both forms in the table

```
# Expand the ('stimuli', 'stimulus') to a DataFrame with 3 columns
stimulus_df = pandas.DataFrame(
    icephys_meta_df[("stimuli", "stimulus")].tolist(),
    columns=[("stimuli", "idx_start"), ("stimuli", "count"), ("stimuli", "timeseries")],
    index=icephys_meta_df.index,
)
# If we want to remove the original ('stimuli', 'stimulus') from the dataframe we can call
# icephys_meta_df.drop(labels=[('stimuli', 'stimulus'), ], axis=1, inplace=True)
# Add our expanded columns to the icephys_meta_df dataframe
icephys_meta_df = pandas.concat([icephys_meta_df, stimulus_df], axis=1)

# save the table as image to display in the docs
```

(continues on next page)

(continued from previous page)

```
dataframe_image.export(
    obj=icephys_meta_df,
    filename=os.path.join(df_basedir, "icephys_meta_dataframe_expand_tsr.png"),
    table_conversion="matplotlib",
    fontsize=dfi_fontsize,
)
```



We can then easily expand also the `(responses, response)` column in the same way

```
response_df = pandas.DataFrame(
    icephys_meta_df[("responses", "response")].tolist(),
    columns=[
        ("responses", "idx_start"),
        ("responses", "count"),
        ("responses", "timeseries"),
    ],
    index=icephys_meta_df.index,
)
icephys_meta_df = pandas.concat([icephys_meta_df, response_df], axis=1)
```

### Adding Stimulus/Response Metadata

With all TimeSeries stimuli and responses listed in the table, we can easily iterate over the TimeSeries to expand our table with additional columns with information from the TimeSeries, e.g., the `neurodata_type` or `name` or any other properties we may wish to extract from our stimulus and response TimeSeries (e.g., `rate`, `starting_time`, `gain` etc.). Here we show a few examples.

```
# Add a column with the name of the stimulus TimeSeries object.
# Note: We use getattr here to easily deal with missing values,
#       i.e., here the cases where no stimulus is present
col = ("stimuli", "name")
icephys_meta_df[col] = [
    getattr(s, "name", None) for s in icephys_meta_df[("stimuli", "timeseries")]
]

# Often we can easily do this in a bulk-fashion by specifying
# the collection of fields of interest
for field in ["neurodata_type", "gain", "rate", "starting_time", "object_id"]:
    col = ("stimuli", field)
    icephys_meta_df[col] = [
        getattr(s, field, None) for s in icephys_meta_df[("stimuli", "timeseries")]
    ]
# save the table as image to display in the docs
dataframe_image.export(
    obj=icephys_meta_df,
```

(continues on next page)

(continued from previous page)

```
    filename=os.path.join(df_basedir, "icephys_meta_dataframe_add_stimres.png"),
    table_conversion="matplotlib",
    max_cols=10,
    fontsize=dfi_fontsize,
)
```

| | (experimental_conditions, id) | (experimental_conditions, temperature) | (repetitions, id) | (repetitions, type) | (sequential_recordings, id) | ... | (stimuli, neurodata_type) | (stimuli, gain) | (stimuli, rate) | (stimuli, starting_time) | (stimuli, object_id) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 100000 | 32.0 | 10000 | R1 | 1000 | ... | None | NaN | NaN | NaN | None |
| 1 | 100000 | 32.0 | 10000 | R1 | 1000 | ... | VoltageClampStimulusSeries | 0.038010 | 8000.0 | 68.275953 | 1264998e-6206-4950-9859-ea... |
| 2 | 100000 | 32.0 | 10000 | R1 | 1000 | ... | VoltageClampStimulusSeries | 0.930204 | 5000.0 | 78.429495 | 11eeb86b-d1a3-463c-b82e-f9... |
| 3 | 100000 | 32.0 | 10000 | R1 | 1000 | ... | VoltageClampStimulusSeries | 0.737919 | 3000.0 | 34.893178 | 7367c177-1abb-4ebe-8c97-e1... |
| 4 | 100000 | 32.0 | 10001 | R2 | 1001 | ... | VoltageClampStimulusSeries | 0.183045 | 4000.0 | 8.907989 | 1fc2faa6-311e-4fae-801a-79... |
| 5 | 100000 | 32.0 | 10001 | R2 | 1001 | ... | VoltageClampStimulusSeries | 0.545212 | 9000.0 | 43.327463 | 8a470610-b50e-432f-8e54-48... |
| 6 | 100000 | 32.0 | 10001 | R2 | 1001 | ... | VoltageClampStimulusSeries | 0.929765 | 2000.0 | 97.551425 | 8f87dedf-ecb3-4053-b852-7c... |
| 7 | 100000 | 32.0 | 10001 | R2 | 1002 | ... | VoltageClampStimulusSeries | 0.319169 | 5000.0 | 1.602690 | ed410c72-c89e-432d-a20e-f1... |
| 8 | 100000 | 32.0 | 10001 | R2 | 1002 | ... | VoltageClampStimulusSeries | 0.328358 | 2000.0 | 53.697790 | 3f4eb534-0ee8-4922-8815-2f... |
| 9 | 100000 | 32.0 | 10001 | R2 | 1002 | ... | VoltageClampStimulusSeries | 0.679388 | 6000.0 | 21.905109 | 6cdee884-1adf-4d46-99e7-3d... |
| 10 | 100001 | 24.0 | 10002 | R1 | 1003 | ... | None | NaN | NaN | NaN | None |
| 11 | 100001 | 24.0 | 10002 | R1 | 1003 | ... | VoltageClampStimulusSeries | 0.270597 | 10000.0 | 49.079180 | 37300bcc-cba9-4d70-ad4e-f6... |
| 12 | 100001 | 24.0 | 10002 | R1 | 1003 | ... | VoltageClampStimulusSeries | 0.141152 | 9000.0 | 49.013644 | 33b28ac8-0934-43dd-a17f-27... |
| 13 | 100001 | 24.0 | 10002 | R1 | 1003 | ... | VoltageClampStimulusSeries | 0.541628 | 7000.0 | 44.366272 | 46bae0da-22b6-4207-a9dc-d9... |
| 14 | 100001 | 24.0 | 10003 | R2 | 1004 | ... | VoltageClampStimulusSeries | 0.466215 | 4000.0 | 43.860254 | e896c7a6-e930-4d83-acd8-32... |
| 15 | 100001 | 24.0 | 10003 | R2 | 1004 | ... | VoltageClampStimulusSeries | 0.995851 | 4000.0 | 71.971481 | e03f45cb-6f9a-41a0-8740-15... |
| 16 | 100001 | 24.0 | 10003 | R2 | 1004 | ... | VoltageClampStimulusSeries | 0.338194 | 1000.0 | 56.039399 | 59e61255-ac41-4a5e-85bd-b5... |
| 17 | 100001 | 24.0 | 10003 | R2 | 1005 | ... | VoltageClampStimulusSeries | 0.734886 | 6000.0 | 65.202506 | 6aab5980-39d8-4470-b45e-9b... |
| 18 | 100001 | 24.0 | 10003 | R2 | 1005 | ... | VoltageClampStimulusSeries | 0.712377 | 9000.0 | 72.016684 | 617bc140-681d-4602-863c-b4... |
| 19 | 100001 | 24.0 | 10003 | R2 | 1005 | ... | VoltageClampStimulusSeries | 0.638423 | 6000.0 | 85.082817 | c39a42ba-683b-48be-9009-5c... |

Naturally we can again do the same also for our response columns

```
for field in ["name", "neurodata_type", "gain", "rate", "starting_time", "object_id"]:
    col = ("responses", field)
    icephys_meta_df[col] = [
        getattr(s, field, None) for s in icephys_meta_df[("responses", "timeseries")]
    ]
```

And we can use the same process to also gather additional metadata about the *IntracellularElectrode*, *Device* and others

```
for field in ["name", "device", "object_id"]:
    col = ("electrodes", field)
    icephys_meta_df[col] = [
        getattr(s, field, None) for s in icephys_meta_df[("electrodes", "electrode")]
    ]
```

This basic approach allows us to easily collect all data needed for query in a convenient spreadsheet for display, query, and analysis.

### Performing common metadata queries

With regard to the experiment metadata tables, many of the queries we identified based on feedback from the community follow the model of: *"Given X return Y"*, e.g.:

- **Given a particular stimulus return:**
  - the corresponding response
  - the corresponding electrode
  - the stimulus type
  - all stimuli/responses recorded at the same time (i.e., during the same simultaneous recording)
  - all stimuli/responses recorded during the same sequential recording
- **Given a particular response return:**

  - the corresponding stimulus
  - the corresponding electrode
  - all stimuli/responses recorded at the same time (i.e., during the same simultaneous recording)
  - all stimuli/responses recorded during the same sequential recording
- **Given an electrode return:**
  - all responses (and stimuli) related to the electrode
  - all sequential recordings (a.k.a., sweeps) recorded with the electrode
- **Given a stimulus type return:**
  - all related stimulus/response recordings
  - all the repetitions in which it is present
- **Given a stimulus type and a repetition return:**
  - all the responses
- **Given a simultaneous recording (a.k.a., sweep) return:**
  - the repetition/condition/sequential recording it belongs to
  - all other simultaneous recordings that are part of the same repetition
  - the experimental condition the simultaneous recording is part of
- **Given a repetition return:**
  - the experimental condition the simultaneous recording is part of
  - all sequential- and/or simultaneous recordings within that repetition
- **Given an experimental condition return:**
  - All corresponding repetitions or sequential/simultaneous/intracellular recordings
- Get the list of all stimulus types

More complex analytics will then commonly combine multiple such query constraints to further process the corresponding data, e.g.,

- Given a stimulus and a condition, return all simultaneous recordings (a.k.a., sweeps) across repetitions and average the responses

Generally, many of the queries involve looking up a piece of information in on table (e.g., finding a stimulus type in *SequentialRecordingsTable*) and then querying for related information in child tables (by following the DynamicTableRegion links included in the corresponding rows) to look up more specific information (e.g., all recordings related to the stimulus type) or alternatively querying for related information in parent tables (by finding rows in the parent table that link to our rows) and then looking up more general information (e.g., information about the experimental condition). Using this approach, we can resolve the above queries using the individual DynamicTable objects directly, while loading only the data that is absolutely necessary into memory.

With the bulk data stored usually in some form of *PatchClampSeries*, the ICEphys metadata tables will usually be comparatively small (in terms of total memory). Once we have created our integrated DataFrame as shown above, performing the queries described above becomes quite simple as all links between tables have already been resolved and all data has been expanded across all rows. In general, resolving queries on our "denormalized" table amounts to evaluating one or more conditions on one or more columns and then retrieving the rows that match our conditions form the table.

Once we have all metadata in a single table, we can also easily sort the rows of our table based on a flexible set of conditions or even cluster rows to compute more advanced groupings of intracellular recordings.

Below we show just a few simple examples:

### Given a response, get the stimulus

```
# Get a response 'vcs_9' from the file
response = nwbfile.get_acquisition("vcs_9")
# Return all data related to that response, including the stimulus
# as part of ('stimuli', 'stimulus') column
icephys_meta_df[icephys_meta_df[("responses", "object_id")] == response.object_id]
```

### Given a response load the associated data

References to timeseries are stored in the *IntracellularRecordingsTable* via *TimeSeriesReferenceVectorData* columns which return the references to the stimulus/response via *TimeSeriesReference* objects. Using *TimeSeriesReference* we can easily inspect the selected data.

```
ref = icephys_meta_df[("responses", "response")][0]  # Get the TimeSeriesReference
_ = ref.isvalid()  # Is the reference valid
_ = ref.idx_start  # Get the start index
_ = ref.count  # Get the count
_ = ref.timeseries.name  # Get the timeseries
_ = ref.timestamps  # Get the selected timestamps
ref_data = ref.data  # Get the selected recorded response data values
# Print the data values just as an example
print("data = " + str(ref_data))
```

```
data = [0.25010232 0.91367506 0.89496367 0.60720485 0.61132515 0.93673802
 0.54867843 0.35846793 0.7088874  0.62949005]
```

### Get a list of all stimulus types

```
unique_stimulus_types = np.unique(
    icephys_meta_df[("sequential_recordings", "stimulus_type")]
)
print(unique_stimulus_types)
```

```
['StimType_1' 'StimType_2' 'StimType_3']
```

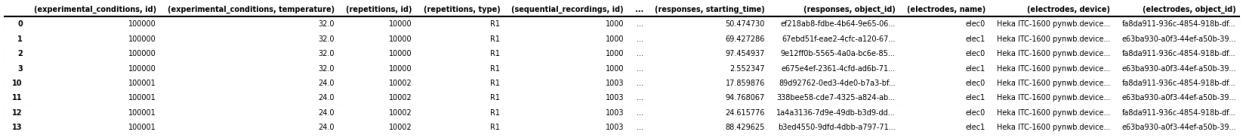
### Given a stimulus type, get all corresponding intracellular recordings

```
query_res_df = icephys_meta_df[
    icephys_meta_df[("sequential_recordings", "stimulus_type")] == "StimType_1"
]

# save the table as image to display in the docs
dataframe_image.export(
```

(continues on next page)

(continued from previous page)

```
    obj=query_res_df,
    filename=os.path.join(df_basedir, "icephys_meta_query_result_dataframe.png"),
    table_conversion="matplotlib",
    max_cols=10,
    fontsize=dfi_fontsize,
)
```



| | (experimental_conditions, id) | (experimental_conditions, temperature) | (repetitions, id) | (repetitions, type) | (sequential_recordings, id) | ... | (responses, starting_time) | (responses, object_id) | (electrodes, name) | (electrodes, device) | (electrodes, object_id) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 100000 | 32.0 | 10000 | R1 | 1000 | ... | 50.474730 | ef218ab8-fdbe-4b64-9e65-06... | elec0 | Heka ITC-1600 pynwb.device... | fa8da911-936c-4854-918b-df... |
| 1 | 100000 | 32.0 | 10000 | R1 | 1000 | ... | 69.427286 | 67ebd51f-eae2-4cfc-a120-67... | elec1 | Heka ITC-1600 pynwb.device... | e63ba930-a0f3-44ef-a50b-39... |
| 2 | 100000 | 32.0 | 10000 | R1 | 1000 | ... | 97.454937 | 9e12ff0b-5565-4a0a-bc6e-85... | elec0 | Heka ITC-1600 pynwb.device... | fa8da911-936c-4854-918b-df... |
| 3 | 100000 | 32.0 | 10000 | R1 | 1000 | ... | 2.552347 | e675e4ef-2361-4cfd-ad6b-71... | elec1 | Heka ITC-1600 pynwb.device... | e63ba930-a0f3-44ef-a50b-39... |
| 10 | 100001 | 24.0 | 10002 | R1 | 1003 | ... | 17.859876 | 89d92762-0ed3-4de0-b7a3-bf... | elec0 | Heka ITC-1600 pynwb.device... | fa8da911-936c-4854-918b-df... |
| 11 | 100001 | 24.0 | 10002 | R1 | 1003 | ... | 94.768067 | 338bee58-cde7-4325-a824-ab... | elec1 | Heka ITC-1600 pynwb.device... | e63ba930-a0f3-44ef-a50b-39... |
| 12 | 100001 | 24.0 | 10002 | R1 | 1003 | ... | 24.615776 | 1a4a3136-7d9e-49db-b3d9-dd... | elec0 | Heka ITC-1600 pynwb.device... | fa8da911-936c-4854-918b-df... |
| 13 | 100001 | 24.0 | 10002 | R1 | 1003 | ... | 88.429625 | b3ed4550-9dfd-4dbb-a797-71... | elec1 | Heka ITC-1600 pynwb.device... | e63ba930-a0f3-44ef-a50b-39... |

### 2.2.5 Intracellular Electrophysiology Data using SweepTable

The following tutorial describes storage of intracellular electrophysiology data in NWB using the SweepTable to manage recordings.

> **Warning:** The use of SweepTable has been deprecated as of PyNWB >v2.0 in favor of the new hierarchical intracellular electrophysiology metadata tables to allow for a more complete description of intracellular electrophysiology experiments. See the *Intracellular electrophysiology* tutorial for details.

#### 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 uuid import uuid4

import numpy as np
from dateutil.tz import tzlocal

from pynwb import NWBFile

nwbfile = NWBFile(
    session_description="my first synthetic recording",
    identifier=str(uuid4()),
    session_start_time=datetime.now(tzlocal()),
    experimenter=[
        "Baggins, Bilbo",
    ],
    lab="Bag End Laboratory",
    institution="University of Middle Earth at the Shire",
    experiment_description="I went on an adventure to reclaim vast treasures.",
    session_id="LONELYMTN001",
)
```

### 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_icephys_electrode*.

```
elec = nwbfile.create_icephys_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, use_sweep_table=True)
```

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, use_sweep_table=True)
```

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, use_sweep_table=True)
```

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,
)
```

(continues on next page)

(continued from previous page)

```
nwbfile.add_acquisition(vcs, use_sweep_table=True)
```

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_icephys_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 of two timeseries consisting of acquisition and stimulus, from sweep 1.

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

(continues on next page)

(continued from previous page)

```
# close the file
io.close()
```

### 2.2.6 Behavior Data

This tutorial will demonstrate the usage of the *pynwb.behavior* module for adding behavioral data to an *NWBFile*.

**See also:**

You can learn more about the *NWBFile* format in the *NWB File Basics* tutorial.

The examples below follow this general workflow for adding behavior data to an *NWBFile*:

- create an object:
  - *TimeSeries* for continuous time series data,
  - *SpatialSeries* for continuous spatial data (e.g. position, direction relative to some reference frame),
  - *IntervalSeries* or *TimeIntervals* for time intervals
- store that object inside a behavior interface object:
  - *Position* for position measured over time
  - *CompassDirection* for view angle measured over time
  - *BehavioralTimeSeries* for continuous time series data
  - *BehavioralEvents* for behavioral events (e.g. reward amount)
  - *BehavioralEpochs* for behavioral intervals (e.g. sleep intervals)
  - *PupilTracking* for eye-tracking data of pupil size
  - *EyeTracking* for eye-tracking data of gaze direction
- create a behavior processing module for the *NWBFile* and add the interface object(s) to it

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 uuid import uuid4

import numpy as np
from dateutil.tz import tzlocal

from pynwb import NWBHDF5IO, NWBFile, TimeSeries
from pynwb.behavior import (
    BehavioralEpochs,
    BehavioralEvents,
    BehavioralTimeSeries,
    CompassDirection,
    EyeTracking,
    Position,
    PupilTracking,
    SpatialSeries,
)
```

(continues on next page)

(continued from previous page)

```
from pynwb.epoch import TimeIntervals
from pynwb.misc import IntervalSeries
```

### Create an NWB File

Create an *NWBFile* object with the required fields (`session_description`, `identifier`, `session_start_time`) and additional metadata.

```
nwbfile = NWBFile(
    session_description="my first synthetic recording",
    identifier=str(uuid4()),
    session_start_time=datetime.now(tzlocal()),
    experimenter=[
        "Baggins, Bilbo",
    ],
    lab="Bag End Laboratory",
    institution="University of Middle Earth at the Shire",
    experiment_description="I went on an adventure to reclaim vast treasures.",
    session_id="LONELYMTN001",
)

nwbfile
```

### SpatialSeries: Storing continuous spatial data

*SpatialSeries* is a subclass of *TimeSeries* that represents data in space, such as the spatial direction, e.g., of gaze or travel, or position of an animal over time.

Create data that corresponds to x, y position over time.

```
position_data = np.array([np.linspace(0, 10, 50), np.linspace(0, 8, 50)]).T

position_data.shape
```

```
(50, 2)
```

In *SpatialSeries* `data`, the first dimension is always time (in seconds), the second dimension represents the x, y position (in meters).

---

**Note:** *SpatialSeries* data should be stored as one continuous stream, as it is acquired, not by trial as is often reshaped for analysis. Data can be trial-aligned on-the-fly using the trials table. See the *Annotating Time Intervals* tutorial for further information.

---

For position data `reference_frame` indicates the zero-position, e.g. the 0,0 point might be the bottom-left corner of an enclosure, as viewed from the tracking camera.

```
timestamps = np.linspace(0, 50) / 200

position_spatial_series = SpatialSeries(
```

(continues on next page)

(continued from previous page)

```
    name="SpatialSeries",
    description="Position (x, y) in an open field.",
    data=position_data,
    timestamps=timestamps,
    reference_frame="(0,0) is bottom left corner",
)

position_spatial_series
```

### Position: Storing position measured over time

To help data analysis and visualization tools know that this *SpatialSeries* object represents the position of the subject, store the *SpatialSeries* object inside a *Position* object, which can hold one or more *SpatialSeries* objects.

```
position = Position(spatial_series=position_spatial_series)
```

**See also:**

You can learn more about the *SpatialSeries* data type and *Position* interface in the *Spatial Series and Position* tutorial.

**See also:**

You can learn more about best practices that can be applied to *SpatialSeries* at NWB Best Practices.

### Create a Behavior Processing Module

Create a processing module called "behavior" for storing behavioral data in the *NWBFile* using the *create_processing_module* method, and then add the *Position* object to the processing module.

```
behavior_module = nwbfile.create_processing_module(
    name="behavior", description="Processed behavioral data"
)

behavior_module.add(position)
```

**See also:**

You can read more about the basic concepts of processing modules in the *Processing Modules* tutorial.

### CompassDirection: Storing view angle measured over time

Analogous to how position can be stored, we can create a *SpatialSeries* object for representing the view angle of the subject.

For direction data `reference_frame` indicates the zero-axes, for instance “straight-ahead” might be a specific pixel on the monitor, or some other point in space. The unit of measurement for the *SpatialSeries* object should be radians or degrees.

```
view_angle_data = np.linspace(0, 4, 50)
```

(continues on next page)

(continued from previous page)

```
direction_spatial_series = SpatialSeries(
    name="SpatialSeries",
    description="View angle of the subject measured in radians.",
    data=view_angle_data,
    timestamps=timestamps,
    reference_frame="straight ahead",
    unit="radians",
)

direction = CompassDirection(
    spatial_series=direction_spatial_series, name="CompassDirection"
)
```

We can add a *CompassDirection* object to the `behavior` processing module the same way as we have added the position data:

```
behavior_module.add(direction)
```

### BehavioralTimeSeries: Storing continuous behavior data

*BehavioralTimeSeries* is an interface for storing continuous behavior data, such as the speed of a subject.

```
speed_data = np.linspace(0, 0.4, 50)

speed_time_series = TimeSeries(
    name="speed",
    data=speed_data,
    timestamps=timestamps,
    description="The speed of the subject measured over time.",
    unit="m/s",
)

behavioral_time_series = BehavioralTimeSeries(
    time_series=speed_time_series,
    name="BehavioralTimeSeries",
)

behavior_module.add(behavioral_time_series)
```

### BehavioralEvents: Storing behavioral events

*BehavioralEvents* is an interface for storing behavioral events. We can use it for storing the timing and amount of rewards (e.g. water amount) or lever press times.

```
reward_amount = [1.0, 1.5, 1.0, 1.5]
events_timestamps = [1.0, 2.0, 5.0, 6.0]

time_series = TimeSeries(
    name="lever_presses",
    data=reward_amount,
```

(continues on next page)

(continued from previous page)

```
    timestamps=events_timestamps,
    description="The water amount the subject received as a reward.",
    unit="ml",
)

behavioral_events = BehavioralEvents(time_series=time_series, name="BehavioralEvents")

behavior_module.add(behavioral_events)
```

Storing only the timestamps of the events is possible with the ndx-events NWB extension. You can also add labels associated with the events with this extension. You can find information about installation and example usage here.

**See also:**

You can learn more about using extensions in the *Extending NWB* tutorial.

### BehavioralEpochs: Storing intervals of behavior data

*BehavioralEpochs* is for storing intervals of behavior data. *BehavioralEpochs* uses *IntervalSeries* to represent behavioral epochs.

Create *IntervalSeries* object that represents the time intervals when the animal was running. *IntervalSeries* uses 1 to indicate the beginning of an interval and -1 to indicate the end.

```
run_intervals = IntervalSeries(
    name="running",
    description="Intervals when the animal was running.",
    data=[1, -1, 1, -1, 1, -1],
    timestamps=[0.5, 1.5, 3.5, 4.0, 7.0, 7.3],
)

behavioral_epochs = BehavioralEpochs(name="BehavioralEpochs")

behavioral_epochs.add_interval_series(run_intervals)
```

you can add more than one *IntervalSeries* to a *BehavioralEpochs*

```
sleep_intervals = IntervalSeries(
    name="sleeping",
    description="Intervals when the animal was sleeping.",
    data=[1, -1, 1, -1],
    timestamps=[15.0, 30.0, 60.0, 95.0],
)
behavioral_epochs.add_interval_series(sleep_intervals)

behavior_module.add(behavioral_epochs)
```

Using *TimeIntervals* representing time intervals is often preferred over *BehavioralEpochs* and *IntervalSeries*. *TimeIntervals* is a subclass of DynamicTable which offers flexibility for tabular data by allowing the addition of optional columns which are not defined in the standard.

Create a *TimeIntervals* object that represents the time intervals when the animal was sleeping.

```
sleep_intervals = TimeIntervals(
    name="sleep_intervals",
    description="Intervals when the animal was sleeping.",
)

sleep_intervals.add_column(name="stage", description="The stage of sleep.")

sleep_intervals.add_row(start_time=0.3, stop_time=0.35, stage=1)
sleep_intervals.add_row(start_time=0.7, stop_time=0.9, stage=2)
sleep_intervals.add_row(start_time=1.3, stop_time=3.0, stage=3)

nwbfile.add_time_intervals(sleep_intervals)
```

### PupilTracking: Storing continuous eye-tracking data of pupil size

*PupilTracking* is for storing eye-tracking data which represents pupil size. *PupilTracking* holds one or more *TimeSeries* objects that can represent different features such as the dilation of the pupil measured over time by a pupil tracking algorithm.

```
pupil_diameter = TimeSeries(
    name="pupil_diameter",
    description="Pupil diameter extracted from the video of the right eye.",
    data=np.linspace(0.001, 0.002, 50),
    timestamps=timestamps,
    unit="meters",
)

pupil_tracking = PupilTracking(time_series=pupil_diameter, name="PupilTracking")

behavior_module.add(pupil_tracking)
```

### EyeTracking: Storing continuous eye-tracking data of gaze direction

*EyeTracking* is for storing eye-tracking data which represents direction of gaze as measured by an eye tracking algorithm. An *EyeTracking* object holds one or more *SpatialSeries* objects that represents the vertical and horizontal gaze positions extracted from a video.

```
right_eye_position = np.linspace(-20, 30, 50)

right_eye_positions = SpatialSeries(
    name="right_eye_position",
    description="The position of the right eye measured in degrees.",
    data=right_eye_position,
    timestamps=timestamps,
    reference_frame="bottom left",
    unit="degrees",
)

eye_tracking = EyeTracking(name="EyeTracking", spatial_series=right_eye_positions)
```

We can add another *SpatialSeries* representing the position of the left eye in degrees.

```
left_eye_position = np.linspace(-2, 20, 50)

left_eye_positions = SpatialSeries(
    name="left_eye_position",
    description="The position of the left eye measured in degrees.",
    data=left_eye_position,
    timestamps=timestamps,
    reference_frame="bottom left",
    unit="degrees",
)

eye_tracking.add_spatial_series(spatial_series=left_eye_positions)

behavior_module.add(eye_tracking)
```

### Writing the behavior data to an NWB file

As demonstrated in the *Writing an NWB file* tutorial, we will use *NWBHDF5IO* to write the file.

```
with NWBHDF5IO("behavioral_tutorial.nwb", "w") as io:
    io.write(nwbfile)
```

### Reading and accessing the behavior data

To read the NWB file we just wrote, use another *NWBHDF5IO* object, and use the *read* method to retrieve an *NWBFile* object.

We can access the behavior processing module by indexing `nwbfile.processing` with the name of the processing module `"behavior"`. We can also inspect the objects hierarchy within this processing module with the `.children` attribute.

```
with NWBHDF5IO("behavioral_tutorial.nwb", "r") as io:
    read_nwbfile = io.read()
    print(read_nwbfile.processing["behavior"].children)
```

```
(BehavioralEpochs pynwb.behavior.BehavioralEpochs at 0x139808110651408
Fields:
  interval_series: {
    running <class 'pynwb.misc.IntervalSeries'>,
    sleeping <class 'pynwb.misc.IntervalSeries'>
  }
, BehavioralEvents pynwb.behavior.BehavioralEvents at 0x139808109166032
Fields:
  time_series: {
    lever_presses <class 'pynwb.base.TimeSeries'>
  }
, BehavioralTimeSeries pynwb.behavior.BehavioralTimeSeries at 0x139808111135376
Fields:
  time_series: {
    speed <class 'pynwb.base.TimeSeries'>
  }
```

(continues on next page)

(continued from previous page)

```
, CompassDirection pynwb.behavior.CompassDirection at 0x139808110683856
Fields:
  spatial_series: {
    SpatialSeries <class 'pynwb.behavior.SpatialSeries'>
  }
, EyeTracking pynwb.behavior.EyeTracking at 0x139808110488144
Fields:
  spatial_series: {
    left_eye_position <class 'pynwb.behavior.SpatialSeries'>,
    right_eye_position <class 'pynwb.behavior.SpatialSeries'>
  }
, Position pynwb.behavior.Position at 0x139808110954832
Fields:
  spatial_series: {
    SpatialSeries <class 'pynwb.behavior.SpatialSeries'>
  }
, PupilTracking pynwb.behavior.PupilTracking at 0x139808111144464
Fields:
  time_series: {
    pupil_diameter <class 'pynwb.base.TimeSeries'>
  }
)
```

For instance, we can access the *SpatialSeries* data by referencing the names of the objects in the hierarchy that contain it. We can access the *Position* object inside the `behavior` processing module by indexing it with the name of the *Position* object, `"Position"`. Then, we can access the *SpatialSeries* object inside the *Position* object by indexing it with the name of the *SpatialSeries* object, `"SpatialSeries"`.

```
with NWBHDF5IO("behavioral_tutorial.nwb", "r") as io:
    read_nwbfile = io.read()
    print(read_nwbfile.processing["behavior"]["Position"]["SpatialSeries"])
```

```
SpatialSeries pynwb.behavior.SpatialSeries at 0x139808111661584
Fields:
  comments: no comments
  conversion: 1.0
  data: <HDF5 dataset "data": shape (50, 2), type "<f8">
  description: Position (x, y) in an open field.
  interval: 1
  offset: 0.0
  reference_frame: (0,0) is bottom left corner
  resolution: -1.0
  timestamps: <HDF5 dataset "timestamps": shape (50,), type "<f8">
  timestamps_unit: seconds
  unit: meters
```

Data arrays are read passively from the file. Accessing the `data` attribute of the *SpatialSeries* object does not read the data values, but presents an HDF5 object that can be indexed to read data. You can use the `[:]` operator to read the entire data array into memory.

```
with NWBHDF5IO("behavioral_tutorial.nwb", "r") as io:
    read_nwbfile = io.read()
```

(continues on next page)

(continued from previous page)

```
print(read_nwbfile.processing["behavior"]["Position"]["SpatialSeries"].data[:])
```

```
[[ 0.          0.        ]
 [ 0.20408163  0.16326531]
 [ 0.40816327  0.32653061]
 [ 0.6122449   0.48979592]
 [ 0.81632653  0.65306122]
 [ 1.02040816  0.81632653]
 [ 1.2244898   0.97959184]
 [ 1.42857143  1.14285714]
 [ 1.63265306  1.30612245]
 [ 1.83673469  1.46938776]
 [ 2.04081633  1.63265306]
 [ 2.24489796  1.79591837]
 [ 2.44897959  1.95918367]
 [ 2.65306122  2.12244898]
 [ 2.85714286  2.28571429]
 [ 3.06122449  2.44897959]
 [ 3.26530612  2.6122449 ]
 [ 3.46938776  2.7755102 ]
 [ 3.67346939  2.93877551]
 [ 3.87755102  3.10204082]
 [ 4.08163265  3.26530612]
 [ 4.28571429  3.42857143]
 [ 4.48979592  3.59183673]
 [ 4.69387755  3.75510204]
 [ 4.89795918  3.91836735]
 [ 5.10204082  4.08163265]
 [ 5.30612245  4.24489796]
 [ 5.51020408  4.40816327]
 [ 5.71428571  4.57142857]
 [ 5.91836735  4.73469388]
 [ 6.12244898  4.89795918]
 [ 6.32653061  5.06122449]
 [ 6.53061224  5.2244898 ]
 [ 6.73469388  5.3877551 ]
 [ 6.93877551  5.55102041]
 [ 7.14285714  5.71428571]
 [ 7.34693878  5.87755102]
 [ 7.55102041  6.04081633]
 [ 7.75510204  6.20408163]
 [ 7.95918367  6.36734694]
 [ 8.16326531  6.53061224]
 [ 8.36734694  6.69387755]
 [ 8.57142857  6.85714286]
 [ 8.7755102   7.02040816]
 [ 8.97959184  7.18367347]
 [ 9.18367347  7.34693878]
 [ 9.3877551   7.51020408]
 [ 9.59183673  7.67346939]
 [ 9.79591837  7.83673469]
 [10.          8.        ]]
```

Alternatively, you can read only a portion of the data by indexing or slicing into the `data` attribute just like if you were indexing or slicing a numpy array.

```
with NWBHDF5IO("behavioral_tutorial.nwb", "r") as io:
    read_nwbfile = io.read()
    print(read_nwbfile.processing["behavior"]["Position"]["SpatialSeries"].data[:2])
```

```
[[0.         0.        ]
 [0.20408163 0.16326531]]
```

### 2.2.7 Storing Image Data in NWB

This tutorial will demonstrate the usage of the *pynwb.image* module for adding images to an *NWBFile*.

Image data can be a collection of individual images or movie segments (as a movie is simply a series of images), about the subject, the environment, the presented stimuli, or other parts related to the experiment. This tutorial focuses in particular on the usage of:

- *OpticalSeries* for series of images that were presented as stimulus
- *ImageSeries*, for series of images (movie segments);
- *GrayscaleImage*, *RGBImage*, *RGBAImage*, for static images;

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
import numpy as np
import os
from PIL import Image
from pynwb import NWBHDF5IO, NWBFile
from pynwb.base import Images
from pynwb.image import GrayscaleImage, ImageSeries, OpticalSeries, RGBAImage, RGBImage
from uuid import uuid4

# Define file paths used in the tutorial
nwbfile_path = os.path.abspath("images_tutorial.nwb")
moviefiles_path = [
    os.path.abspath("image/file_1.tiff"),
    os.path.abspath("image/file_2.tiff"),
    os.path.abspath("image/file_3.tiff"),
]
```

#### Create an NWB File

Create an *NWBFile* object with the required fields (`session_description`, `identifier`, `session_start_time`) and additional metadata.

```
session_start_time = datetime(2018, 4, 25, hour=2, minute=30)

nwbfile = NWBFile(
    session_description="my first synthetic recording",
```

(continues on next page)

(continued from previous page)

```
    identifier=str(uuid4()),
    session_start_time=session_start_time,
    experimenter=[
        "Baggins, Bilbo",
    ],
    lab="Bag End Laboratory",
    institution="University of Middle Earth at the Shire",
    experiment_description="I went on an adventure to reclaim vast treasures.",
    session_id="LONELYMTN001",
)

nwbfile
```

**See also:**

You can learn more about the *NWBFile* format in the *NWB File Basics* tutorial.

### OpticalSeries: Storing series of images as stimuli

*OpticalSeries* is for time series of images that were presented to the subject as stimuli. We will create an *OpticalSeries* object with the name "StimulusPresentation" representing what images were shown to the subject and at what times.

Image data can be stored either in the HDF5 file or as an external image file. For this tutorial, we will use fake image data with shape of ('time', 'x', 'y', 'RGB') = (200, 50, 50, 3). As in all *TimeSeries*, the first dimension is time. The second and third dimensions represent x and y. The fourth dimension represents the RGB value (length of 3) for color images.

NWB differentiates between acquired data and data that was presented as stimulus. We can add it to the *NWBFile* object as stimulus data using the *add_stimulus* method.

```
image_data = np.random.randint(low=0, high=255, size=(200, 50, 50, 3), dtype=np.uint8)
optical_series = OpticalSeries(
    name="StimulusPresentation",  # required
    distance=0.7,  # required
    field_of_view=[0.2, 0.3, 0.7],  # required
    orientation="lower left",  # required
    data=image_data,
    unit="n.a.",
    format="raw",
    starting_frame=[0.0],
    rate=1.0,
    comments="no comments",
    description="The images presented to the subject as stimuli",
)

nwbfile.add_stimulus(stimulus=optical_series)
```

### ImageSeries: Storing series of images as acquisition

*ImageSeries* is a general container for time series of images acquired during the experiment. Image data can be stored either in the HDF5 file or as an external image file. When color images are stored in the HDF5 file the color channel order is expected to be RGB.

We can add raw data to the *NWBFile* object as *acquisition* using the *add_acquisition* method.

```
image_data = np.random.randint(low=0, high=255, size=(200, 50, 50, 3), dtype=np.uint8)
behavior_images = ImageSeries(
    name="ImageSeries",
    data=image_data,
    description="Image data of an animal moving in environment.",
    unit="n.a.",
    format="raw",
    rate=1.0,
    starting_time=0.0,
)

nwbfile.add_acquisition(behavior_images)
```

### External Files

External files (e.g. video files of the behaving animal) can be added to the *NWBFile* by creating an *ImageSeries* object using the *external_file* attribute that specifies the path to the external file(s) on disk. The file(s) path must be relative to the path of the NWB file. Either `external_file` or `data` must be specified, but not both.

If the sampling rate is constant, use *rate* and *starting_time* to specify time. For irregularly sampled recordings, use *timestamps* to specify time for each sample image.

Each external image may contain one or more consecutive frames of the full *ImageSeries*. The *starting_frame* attribute serves as an index to indicate which frame each file contains. For example, if the `external_file` dataset has three paths to files and the first and the second file have 2 frames, and the third file has 3 frames, then this attribute will have values *[0, 2, 4]*.

```
external_file = [
    os.path.relpath(movie_path, nwbfile_path) for movie_path in moviefiles_path
]
# We have 3 movie files each containing multiple frames. We here need to specify the␣
↪timestamp for each frame.
timestamps = [0.0, 0.04, 0.07, 0.1, 0.14, 0.16, 0.21]
behavior_external_file = ImageSeries(
    name="ExternalFiles",
    description="Behavior video of animal moving in environment.",
    unit="n.a.",
    external_file=external_file,
    format="external",
    starting_frame=[0, 2, 4],
    timestamps=timestamps,
)

nwbfile.add_acquisition(behavior_external_file)
```

**Note:** See the External Links in NWB and DANDI guidelines of the DANDI data archive for best practices on how to organize external files, e.g., movies and images.

### Static images

Static images can be stored in an *NWBFile* object by creating an *RGBAImage*, *RGBImage* or *GrayscaleImage* object with the image data.

**Note:** All basic image types *RGBAImage*, *RGBImage*, and *GrayscaleImage* provide the optional: 1) `description` parameter to include a text description about the image and 2) `resolution` parameter to specify the *pixels / cm* resolution of the image.

### RGBAImage: for color images with transparency

*RGBAImage* is for storing data of color image with transparency. `RGBAImage.data` must be 3D where the first and second dimensions represent x and y. The third dimension has length 4 and represents the RGBA value.

```
img = Image.open("docs/source/figures/logo_pynwb.png")  # an example image

rgba_logo = RGBAImage(
    name="pynwb_RGBA_logo",
    data=np.array(img),
    resolution=70.0,  # in pixels / cm
    description="RGBA version of the PyNWB logo.",
)
```

### RGBImage: for color images

*RGBImage* is for storing data of RGB color image. `RGBImage.data` must be 3D where the first and second dimensions represent x and y. The third dimension has length 3 and represents the RGB value.

```
rgb_logo = RGBImage(
    name="pynwb_RGB_logo",
    data=np.array(img.convert("RGB")),
    resolution=70.0,
    description="RGB version of the PyNWB logo.",
)
```

### GrayscaleImage: for grayscale images

*GrayscaleImage* is for storing grayscale image data. `GrayscaleImage.data` must be 2D where the first and second dimensions represent x and y.

```
gs_logo = GrayscaleImage(
    name="pynwb_Grayscale_logo",
    data=np.array(img.convert("L")),
    description="Grayscale version of the PyNWB logo.",
    resolution=35.433071,
)
```

### Images: a container for images

Add the images to an *Images* container that accepts any of these image types.

```
images = Images(
    name="logo_images",
    images=[rgb_logo, rgba_logo, gs_logo],
    description="A collection of logo images presented to the subject.",
)

nwbfile.add_acquisition(images)
```

### IndexSeries for repeated images

You may want to set up a time series of images where some images are repeated many times. You could create an *ImageSeries* that repeats the data each time the image is shown, but that would be inefficient, because it would store the same data multiple times. A better solution would be to store the unique images once and reference those images. This is how *IndexSeries* works. First, create an *Images* container with the order of images defined using an *ImageReferences*. Then create an *IndexSeries* that indexes into the *Images*.

```
from scipy import misc

from pynwb.base import ImageReferences
from pynwb.image import GrayscaleImage, Images, IndexSeries, RGBImage

gs_face = GrayscaleImage(
    name="gs_face",
    data=misc.face(gray=True),
    description="Grayscale version of a raccoon.",
    resolution=35.433071,
)

rgb_face = RGBImage(
    name="rgb_face",
    data=misc.face(),
    resolution=70.0,
    description="RGB version of a raccoon.",
)
```

(continues on next page)

(continued from previous page)

```
images = Images(
    name="raccoons",
    images=[rgb_face, gs_face],
    description="A collection of raccoons.",
    order_of_images=ImageReferences("order_of_images", [rgb_face, gs_face]),
)

idx_series = IndexSeries(
    name="stimuli",
    data=[0, 1, 0, 1],
    indexed_images=images,
    unit="N/A",
    timestamps=[0.1, 0.2, 0.3, 0.4],
)
```

Here *data* contains the (0-indexed) index of the displayed image as they are ordered in the *ImageReferences*.

#### Writing the images to an NWB File

As demonstrated in the *Writing an NWB file* tutorial, we will use *NWBHDF5IO* to write the file.

```
with NWBHDF5IO(nwbfile_path, "w") as io:
    io.write(nwbfile)
```

#### Reading and accessing data

To read the NWB file, use another *NWBHDF5IO* object, and use the *read* method to retrieve an *NWBFile* object.

We can access the data added as acquisition to the NWB File by indexing `nwbfile.acquisition` with the name of the *ImageSeries* object "ImageSeries".

We can also access *OpticalSeries* data that was added to the NWB File as stimuli by indexing `nwbfile.stimulus` with the name of the *OpticalSeries* object "StimulusPresentation". Data arrays are read passively from the file. Accessing the `data` attribute of the *OpticalSeries* object does not read the data values into memory, but returns an HDF5 object that can be indexed to read data. Use the `[:]` operator to read the entire data array into memory.

```
with NWBHDF5IO(nwbfile_path, "r") as io:
    read_nwbfile = io.read()
    print(read_nwbfile.acquisition["ImageSeries"])
    print(read_nwbfile.stimulus["StimulusPresentation"].data[:])
```

### 2.2.8 Allen Brain Observatory

Create an nwb file from Allen Brain Observatory data.

This example demonstrates 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.

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

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

# 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"],
)
```

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(
```

(continues on next page)

(continued from previous page)

```
        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 recorded 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 protein 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=520.0,
)

device = Device(metadata["device"])
nwbfile.add_device(device)
```

(continues on next page)

(continued from previous page)

```
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=30.0,
    indicator="GCaMP6f",
    location=metadata["targeted_structure"],
    conversion=1.0,
    unit="unknown",
    reference_frame="unknown",
)
```

The Allen Institute 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 fluorescence 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 standardize 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(
```

(continues on next page)

(continued from previous page)

```
    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()
```

### 2.2.9 Optogenetics

This tutorial will demonstrate how to write optogenetics data.

#### Creating an NWBFile object

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

```
from datetime import datetime
from uuid import uuid4

from dateutil.tz import tzlocal

from pynwb import NWBFile

nwbfile = NWBFile(
    session_description="my first synthetic recording",
    identifier=str(uuid4()),
    session_start_time=datetime.now(tzlocal()),
    experimenter="Baggins, Bilbo",
    lab="Bag End Laboratory",
    institution="University of Middle Earth at the Shire",
    experiment_description="I went on an adventure to reclaim vast treasures.",
    session_id="LONELYMTN",
)
```

### Adding optogenetic data

The *ogen* module contains two data types that you will need to write optogenetics data, *OptogeneticStimulusSite*, which contains metadata about the stimulus site, and *OptogeneticSeries*, which contains the power applied by the laser over time, in watts.

First, you need to create a *Device* object linked to the *NWBFile*:

```
device = nwbfile.create_device(
    name="device",
    description="description of device",
    manufacturer="optional but recommended",
)
```

Now, you can create an *OptogeneticStimulusSite*. The easiest way to do this is to use the *create_ogen_site* method.

```
ogen_site = nwbfile.create_ogen_site(
    name="OptogeneticStimulusSite",
    device=device,
    description="This is an example optogenetic site.",
    excitation_lambda=600.0,  # nm
    location="VISrl",
)
```

Another equivalent approach would be to create a *OptogeneticStimulusSite* and then add it to the *NWBFile*:

```
from pynwb.ogen import OptogeneticStimulusSite

ogen_stim_site = OptogeneticStimulusSite(
    name="OptogeneticStimulusSite2",
    device=device,
    description="This is an example optogenetic site.",
    excitation_lambda=600.0,  # nm
    location="VISrl",
)

nwbfile.add_ogen_site(ogen_stim_site)
```

The second approach is necessary if you have an extension of *OptogeneticStimulusSite*.

With the *OptogeneticStimulusSite* added, you can now create a *OptogeneticSeries*. Here, we will generate some random data using numpy and specify the timing using `rate`. By default, the starting time of the time series is the session start time, specified in *NWBFile*. If you have samples at irregular intervals, you should use `timestamps` instead.

```
import numpy as np

from pynwb.ogen import OptogeneticSeries

ogen_series = OptogeneticSeries(
    name="OptogeneticSeries",
    data=np.random.randn(20),  # watts
    site=ogen_site,
    rate=30.0,  # Hz
```

(continues on next page)

(continued from previous page)

```
)

nwbfile.add_stimulus(ogen_series)
```

## 2.3 Advanced I/O

### 2.3.1 Defining HDF5 Dataset I/O Settings (chunking, compression, etc.)

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 pynwb import NWBFile

start_time = datetime(2017, 4, 3, hour=11, minute=0)

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

Normally if we create a *TimeSeries* we would do

```
import numpy as np

from pynwb import TimeSeries

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 chunk and 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

test_ts = TimeSeries(
    name="test_compressed_timeseries",
    data=H5DataIO(data=data, compression=True),  # <----
    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. Here, the settings for chunking and compression are determined automatically.

### 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 resizable
)
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. You can find a discussion on how chunking can help in the Python and HDF5 book. by Andrew Collette.

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 appropriate options:

```
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.

### Writing the data

Writing the data now works as usual.

```
from pynwb import NWBHDF5IO

with NWBHDF5IO("advanced_io_example.nwb", "w") as io:
    io.write(nwbfile)
```

### Reading the data

Nothing has changed for read. All 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"
```

(continues on next page)

(continued from previous page)

```
        % (k, v.data.chunks, v.data.compression, v.data.maxshape)
    )
io.close()
```

```
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. Also, 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.

---

### Dynamically Loaded Filters

HDF5 allows you to install additional filters and use these filters as plugins. Some of these filters may have superior performance to the default *GZIP* when it comes to read speed, write speed, or compression ratio. You can install several of the most popular filter plugins using the hdf5plugin library.

First, use `pip` to install hdf5plugin:

```
pip install hdf5plugin
```

This command automatically installs the filters. Here is an example of how you would use the Z Standard algorithm:

```
import hdf5plugin
from hdmf.backends.hdf5.h5_utils import H5DataIO

from pynwb.file import TimeSeries

wrapped_data = H5DataIO(
    data=data,
    **hdf5plugin.Zstd(clevel=3),  # set the compression and compression_opts parameters
    allow_plugin_filters=True,
)

test_ts = TimeSeries(
    name="test_gzipped_timeseries",
    data=wrapped_data,
    unit="SIunit",
```

(continues on next page)

(continued from previous page)

```
    starting_time=0.0,
    rate=10.0,
)
```

See a list of supported compressors here.

---

**Note:** *h5py* (and *HDF5* more broadly) supports 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, so some users may not be able to access those datasets.

---

#### 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.

### 2.3.2 Editing NWB files

This tutorial demonstrates how to edit NWB files in-place to make small changes to existing containers. To add or remove containers from an NWB file, see *Adding/Removing Containers from an NWB File*. How and whether it is possible to edit an NWB file depends on the storage backend and the type of edit.

**Warning:** Manually editing an existing NWB file can make the file invalid if you are not careful. We highly recommend making a copy before editing and running a validation check on the file after editing it. See *Validating NWB files*.

#### Editing datasets

When reading an HDF5 NWB file, PyNWB exposes `h5py.Dataset` objects, which can be edited in place. For this to work, you must open the file in read/write mode (`"r+"` or `"a"`).

First, let's create an NWB file with data:

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

nwbfile = NWBFile(
    session_description="my first synthetic recording",
    identifier="EXAMPLE_ID",
    session_start_time=datetime.now(tzlocal()),
    session_id="LONELYMTN",
)
```

(continues on next page)

(continued from previous page)

```
nwbfile.add_acquisition(
    TimeSeries(
        name="synthetic_timeseries",
        description="Random values",
        data=np.random.randn(100, 100),
        unit="m",
        rate=10e3,
    )
)

with NWBHDF5IO("test_edit.nwb", "w") as io:
    io.write(nwbfile)
```

Now, let's edit the values of the dataset

```
with NWBHDF5IO("test_edit.nwb", "r+") as io:
    nwbfile = io.read()
    nwbfile.acquisition["synthetic_timeseries"].data[:10] = 0.0
```

You can edit the attributes of that dataset through the `attrs` attribute:

```
with NWBHDF5IO("test_edit.nwb", "r+") as io:
    nwbfile = io.read()
    nwbfile.acquisition["synthetic_timeseries"].data.attrs["unit"] = "volts"
```

### Changing the shape of dataset

Whether it is possible to change the shape of a dataset depends on how the dataset was created. If the dataset was created with a flexible shape, then it is possible to change in-place. Creating a dataset with a flexible shape is done by specifying the `maxshape` argument of the `H5DataIO` class constructor. Using a `None` value for a component of the `maxshape` tuple allows the size of the corresponding dimension to grow, such that is can be be reset arbitrarily long in that dimension. Chunking is required for datasets with flexible shapes. Setting `maxshape`, hence, automatically sets chunking to `True`, if not specified.

First, let's create an NWB file with a dataset with a flexible shape:

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

nwbfile = NWBFile(
    session_description="my first synthetic recording",
    identifier="EXAMPLE_ID",
    session_start_time=datetime.now(tzlocal()),
    session_id="LONELYMTN",
)

data_io = H5DataIO(data=np.random.randn(100, 100), maxshape=(None, 100))

nwbfile.add_acquisition(
    TimeSeries(
        name="synthetic_timeseries",
        description="Random values",
        data=data_io,
```

(continues on next page)

(continued from previous page)

```
        unit="m",
        rate=10e3,
    )
)

with NWBHDF5IO("test_edit2.nwb", "w") as io:
    io.write(nwbfile)
```

The `None` value in the first component of `maxshape` means that the the first dimension of the dataset is unlimited. By setting the second dimension of `maxshape` to `100`, that dimension is fixed to be no larger than `100`. If you do not specify a``maxshape``, then the shape of the dataset will be fixed to the shape that the dataset was created with. Here, you can change the shape of the first dimension of this dataset.

```
with NWBHDF5IO("test_edit2.nwb", "r+") as io:
    nwbfile = io.read()
    nwbfile.acquisition["synthetic_timeseries"].data.resize((200, 100))
```

This will change the shape of the dataset in-place. If you try to change the shape of a dataset with a fixed shape, you will get an error.

---

**Note:** There are several types of dataset edits that cannot be done in-place: changing the shape of a dataset with a fixed shape, or changing the datatype, compression, chunking, max-shape, or fill-value of a dataset. For any of these, we recommend using the *pynwb.NWBHDF5IO.export* method to export the data to a new file. See *Adding/Removing Containers from an NWB File* for more information.

---

### Editing groups

Editing of groups is not yet supported in PyNWB. To edit the attributes of a group, open the file and edit it using `h5py`:

```
import h5py

with h5py.File("test_edit.nwb", "r+") as f:
    f["acquisition"]["synthetic_timeseries"].attrs["description"] = "Random values in␣
↪volts"
```

**Warning:** Be careful not to edit values that will bring the file out of compliance with the NWB specification.

### Renaming groups and datasets

Rename groups and datasets in-place using the `move` method. For example, to rename the `"synthetic_timeseries"` group:

```
with h5py.File("test_edit.nwb", "r+") as f:
    f["acquisition"].move("synthetic_timeseries", "synthetic_timeseries_renamed")
```

You can use this same technique to move a group or dataset to a different location in the file. For example, to move the `"synthetic_timeseries_renamed"` group to the `"analysis"` group:

```
with h5py.File("test_edit.nwb", "r+") as f:
    f["acquisition"].move(
        "synthetic_timeseries_renamed",
        "/analysis/synthetic_timeseries_renamed",
    )
```

### 2.3.3 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 contents 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 a 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 a 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.
- `GenericDataChunkIterator` is a semi-abstract version of a `AbstractDataChunkIterator` that automatically handles the selection of buffer regions and resolves communication of compatible chunk regions. Users specify chunk and buffer shapes or sizes and the iterator will manage how to break the data up for write. For further details, see the GenericDataChunkIterator tutorial.

### 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 iterative 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 queuing 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 repetition of the same code and to allow us to focus on the important parts of this tutorial.

```
from datetime import datetime
from pynwb import NWBHDF5IO, NWBFile, TimeSeries
from uuid import uuid4

def write_test_file(filename, data, close_io=True):
    """

    Simple helper function to write an NWBFile with a single timeseries containing data
```

(continues on next page)

(continued from previous page)

```
    :param filename: String with the name of the output file
    :param data: The data of the timeseries
    :param close_io: Close and destroy the NWBHDF5IO object used for writing␣
↪(default=True)

    :returns: None if close_io==True otherwise return NWBHDF5IO object used for write
    """

    # Create a test NWBfile
    start_time = datetime(2017, 4, 3, hour=11, minute=30)
    nwbfile = NWBFile(
        session_description="demonstrate iterative write",
        identifier=str(uuid4()),
        session_start_time=start_time,
    )

    # Create our time series
    test_ts = TimeSeries(
        name="synthetic_timeseries",
        data=data,
        unit="n/a",
        rate=1.0,
    )
    nwbfile.add_acquisition(test_ts)

    # Write the data to file
    io = NWBHDF5IO(filename, "w")
    io.write(nwbfile)
    if close_io:
        io.close()
        del io
        io = None
    return io
```

### 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 use the same approach to write streaming data.

### Step 1: Define the data generator

```
from math import pi, sin
from random import random

import numpy as np


def iter_sin(chunk_length=10, max_chunks=100):
```

(continues on next page)

(continued from previous page)

```
    """
    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, here we don't actually 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))
)
```

```
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 should 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 generator yields in **consecutive order** a **single** complete element along the **first dimension** of our 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 using `GenericDataChunkIterator` or implementing your own derived `AbstractDataChunkIterator` may be more appropriate. We show an example of how to implement your own `AbstractDataChunkIterator` next. See the GenericDataChunkIterator tutorial as part of the HDMF documentation for details on how to use `GenericDataChunkIterator`.

### 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
        the 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
```

(continues on next page)

(continued from previous page)

```
    next = __next__

    def recommended_chunk_shape(self):
        # Here we can optionally recommend what a good chunking could 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 features 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, fill value, and other settings

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

(continues on next page)

(continued from previous page)

```
# 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 bytes
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 = 1000.0 * 1000  # Factor used to convert to MegaBytes
```

(continues on next page)

(continued from previous page)

```
print("1) Sparse Matrix Size:")
print("   Expected Size :  %.2f MB" % (expected_size / mbfactor))
print("   Occupied Size :  %.5f MB" % (occupied_size / mbfactor))
print("2) NWB HDF5 file (no compression):")
print("   File Size     :  %.2f MB" % (file_size / mbfactor))
print("   Reduction     :  %.2f x" % (expected_size / file_size))
print("3) NWB 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 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 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))
```

```
1) Sparse Matrix Size:
   Expected Size :  8000000.00 MB
   Occupied Size :  0.80000 MB
2) NWB HDF5 file (no compression):
   File Size     :  1.04 MB
   Reduction     :  7672653.89 x
3) NWB HDF5 file (with GZIP compression):
   File Size     :  1.05359 MB
   Reduction     :  7593100.91 x
4) NWB HDF5 file (large chunks):
   File Size     :  80.25313 MB
   Reduction     :  99684.59 x
5) NWB HDF5 file (large chunks with compression):
   File Size     :  1.31608 MB
   Reduction     :  6078634.74 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 accordingly 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 fill values, 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 other factors, 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)  # This is 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` or use `GenericDataChunkIterator`.

```
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,
```

(continues on next page)

(continued from previous page)

```
    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

with NWBHDF5IO("basic_sparse_iterwrite_largearray.nwb", "r") as io:
    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")
```

```
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
        :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",
```

(continues on next page)

(continued from previous page)

```
                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 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

# Use H5DataIO to specify how to setup the dataset in the file
dataio = H5DataIO(
    shape=(0, 10),  # Initial shape. If the shape is known then set to full shape
    dtype=np.dtype("float"),  # dtype of the dataset
    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
)

# Write a test NWB file with our dataset and keep the NWB file (i.e., the NWBHDF5IO␣
↪object) open
io = write_test_file(
    filename="basic_alternative_custom_write.nwb", data=dataio, close_io=False
)
```

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

```
# Let's check what the data looks like before we write
print(
    "Before write: Shape= %s, Chunks= %s, Maxshape=%s"
    % (
        str(dataio.dataset.shape),
        str(dataio.dataset.chunks),
        str(dataio.dataset.maxshape),
    )
)

# Allocate space. Only needed if we didn't set the initial shape large enough
dataio.dataset.resize((8, 10))

# Write 1s in timesteps 0-2
dataio.dataset[0:3, :] = 1

# Write 2s in timesteps 3-5
# NOTE: timesteps 6 and 7 are not being initialized
dataio.dataset[3:6, :] = 2

# Close the file
io.close()
```

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

### Check the results

```
from pynwb import NWBHDF5IO  # noqa

io = NWBHDF5IO("basic_alternative_custom_write.nwb", mode="r")
nwbfile = io.read()
dataset = nwbfile.get_acquisition("synthetic_timeseries").data
print(
    "After write: Shape= %s, Chunks= %s, Maxshape=%s"
    % (str(dataset.shape), str(dataset.chunks), str(dataset.maxshape))
)
print(dataset[:])
io.close()
```

```
After write: Shape= (8, 10), Chunks= (131072, 2), Maxshape=(None, 10)
[[ 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]]
```

We allocated our data to be `shape=(8, 10)` but we only wrote data to the first 6 rows of the array. As expected, we therefore, see our `fillvalue` of `nan` in the last two rows of the data.

**Total running time of the script:** (0 minutes 8.594 seconds)

### 2.3.4 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 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 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
import numpy as np
from pynwb import NWBHDF5IO, NWBFile, TimeSeries
from uuid import uuid4

# Create the base data
start_time = datetime(2017, 4, 3, hour=11, minute=0)
data = np.arange(1000).reshape((100, 10))
```

(continues on next page)

(continued from previous page)

```
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=str(uuid4()),
    session_start_time=start_time,
)
# 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
with NWBHDF5IO(filename1, "w") as io:
    io.write(nwbfile1)

# Create the second file
nwbfile2 = NWBFile(
    session_description="demonstrate external files",
    identifier=str(uuid4()),
    session_start_time=start_time,
)
# 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
with NWBHDF5IO(filename2, "w") as io:
    io.write(nwbfile2)
```

### Linking to select datasets

#### Step 1: Create the new NWBFile

```
# Create the first file
nwbfile4 = NWBFile(
    session_description="demonstrate external files",
    identifier=str(uuid4()),
    session_start_time=start_time,
```

(continues on next page)

(continued from previous page)

```
)
```

### 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

```
with NWBHDF5IO(filename4, "w") as io4:
    # Use link_data=True to specify default behavior to link rather than copy data
    io4.write(nwbfile4, link_data=True)
io1.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 integrate both *TimeSeries* into a single file we simply create a new *NWBFile* and add 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=str(uuid4()),
```

(continues on next page)

(continued from previous page)

```
    session_start_time=start_time,
)
nwbfile3.add_acquisition(timeseries_1)  # <--------
nwbfile3.add_acquisition(timeseries_2)  # <--------

# Write our third file that includes our two timeseries as external links
with NWBHDF5IO(filename3, "w", manager=manager) as io3:
    io3.write(nwbfile3)
io1.close()
io2.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.

### 2.3.5 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.backends.hdf5.h5_utils import H5DataIO

start_time = datetime(2018, 4, 25, hour=2, minute=30, second=3)
fname = "test_parallel_pynwb.nwb"
rank = MPI.COMM_WORLD.rank  # The process ID (integer 0-3 for 4-process run)
```

(continues on next page)

(continued from previous page)

```
# 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 = H5DataIO(shape=(4,), maxshape=(4,), dtype=np.dtype("int"))

    nwbfile.add_acquisition(
        TimeSeries(name="ts_name", description="desc", data=data, rate=100.0, 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])
```

**Note:** Using `hdmf.backends.hdf5.h5_utils.H5DataIO` we can also specify further details about the data layout, e.g., via the chunking and compression parameters.

### 2.3.6 Streaming NWB files

You can read specific sections within individual data files directly from remote stores such as the DANDI Archive. This is especially useful for reading small pieces of data from a large NWB file stored remotely. First, you will need to get the location of the file. The code below illustrates how to do this on DANDI using the dandi API library.

#### Getting the location of the file on DANDI

The `DandiAPIClient` can be used to get the S3 URL of any NWB file stored in the DANDI Archive. If you have not already, install the latest release of the `dandi` package.

```
pip install dandi
```

Now you can get the url of a particular NWB file using the dandiset ID and the path of that file within the dandiset.

**Note:** To learn more about the dandi API see the DANDI Python API docs

```
from dandi.dandiapi import DandiAPIClient

dandiset_id = '000006'  # ephys dataset from the Svoboda Lab
filepath = 'sub-anm372795/sub-anm372795_ses-20170718.nwb'  # 450 kB file
```

(continues on next page)

(continued from previous page)

```
with DandiAPIClient() as client:
    asset = client.get_dandiset(dandiset_id, 'draft').get_asset_by_path(filepath)
    s3_url = asset.get_content_url(follow_redirects=1, strip_query=True)
```

### Streaming Method 1: fsspec

fsspec is another data streaming approach that is quite flexible and has several performance advantages. This library creates a virtual filesystem for remote stores. With this approach, a virtual file is created for the file and the virtual filesystem layer takes care of requesting data from the S3 bucket whenever data is read from the virtual file. Note that this implementation is completely unaware of internals of the HDF5 format and thus can work for **any** file, not only for the purpose of use with H5PY and PyNWB.

First install `fsspec` and the dependencies of the `HTTPFileSystem`:

```
pip install fsspec requests aiohttp
```

Then in Python:

```
import fsspec
import pynwb
import h5py
from fsspec.implementations.cached import CachingFileSystem

# first, create a virtual filesystem based on the http protocol
fs = fsspec.filesystem("http")

# create a cache to save downloaded data to disk (optional)
fs = CachingFileSystem(
    fs=fs,
    cache_storage="nwb-cache",  # Local folder for the cache
)

# next, open the file
with fs.open(s3_url, "rb") as f:
    with h5py.File(f) as file:
        with pynwb.NWBHDF5IO(file=file) as io:
            nwbfile = io.read()
            print(nwbfile.acquisition['lick_times'].time_series['lick_left_times'].
↪data[:])
```

fsspec is a library that can be used to access a variety of different store formats, including (at the time of writing):

```
from fsspec.registry import known_implementations
known_implementations.keys()
```

file, memory, dropbox, http, https, zip, tar, gcs, gs, gdrive, sftp, ssh, ftp, hdfs, arrow_hdfs, webhdfs, s3, s3a, wandb, oci, adl, abfs, az, cached, blockcache, filecache, simplecache, dask, dbfs, github, git, smb, jupyter, jlab, libarchive, reference

The S3 backend, in particular, may provide additional functionality for accessing data on DANDI. See the fsspec documentation on known implementations for a full updated list of supported store formats.

One downside of this fsspec method is that fsspec is not optimized for reading HDF5 files, and so streaming data using this method can be slow. A faster alternative is `remfile` described below.

### Streaming Method 2: ROS3

ROS3 stands for "read only S3" and is a driver created by the HDF5 Group that allows HDF5 to read HDF5 files stored remotely in s3 buckets. Using this method requires that your HDF5 library is installed with the ROS3 driver enabled. With ROS3 support enabled in h5py, we can instantiate a *NWBHDF5IO* object with the S3 URL and specify the driver as "ros3".

```
from pynwb import NWBHDF5IO

with NWBHDF5IO(s3_url, mode='r', driver='ros3') as io:
    nwbfile = io.read()
    print(nwbfile)
    print(nwbfile.acquisition['lick_times'].time_series['lick_left_times'].data[:])
```

This will download metadata about the file from the S3 bucket to memory. The values of datasets are accessed lazily, just like when reading an NWB file stored locally. So, slicing into a dataset will require additional time to download the sliced data (and only the sliced data) to memory.

---

**Note:** Pre-built h5py packages on PyPI do not include this S3 support. If you want this feature, you could use packages from conda-forge, or build h5py from source against an HDF5 build with S3 support. You can install HDF5 with the ROS3 driver from conda-forge using `conda`. You may first need to uninstall a currently installed version of `h5py`.

```
pip uninstall h5py
conda install -c conda-forge "h5py>=3.2"
```

---

Besides the extra burden of installing h5py from a non-PyPI source, one downside of this ROS3 method is that this method does not support automatic retries in case the connection fails.

### Method 3: remfile

`remfile` is another library that enables indexing and streaming of files in s3. remfile is simple and fast, especially for the initial load of the nwb file and for accessing small pieces of data. The caveats of `remfile` are that it is a very new project that has not been tested in a variety of use-cases and caching options are limited compared to `fsspec`. *remfile* is a simple, lightweight dependency with a very small codebase.

You can install `remfile` with pip:

```
pip install remfile
```

```
import h5py
from pynwb import NWBHDF5IO
import remfile

rem_file = remfile.File(s3_url)

with h5py.File(rem_file, "r") as h5py_file:
    with NWBHDF5IO(file=h5py_file, load_namespaces=True) as io:
        nwbfile = io.read()
        print(nwbfile.acquisition["lick_times"].time_series["lick_left_times"].data[:])
```

#### Which streaming method to choose?

From a user perspective, once opened, the *NWBFile* works the same with fsspec, ros3, or remfile. However, in general, we currently recommend using fsspec for streaming NWB files because it is more performant and reliable than ros3 and more widely tested than remfile. However, if you are experiencing long wait times for the initial file load on your network, you may want to try remfile.

Advantages of fsspec include:

1. supports caching, which will dramatically speed up repeated requests for the same region of data,
2. automatically retries when s3 fails to return, which helps avoid errors when accessing data due to intermittent errors in connections with S3 (remfile does this as well),
3. works also with other storage backends (e.g., GoogleDrive or Dropbox, not just S3) and file formats, and
4. in our experience appears to provide faster out-of-the-box performance than the ros3 driver.

### 2.3.7 Zarr IO

Zarr is an alternative backend option for NWB files. It is a Python package that provides an implementation of chunked, compressed, N-dimensional arrays. Zarr is a good option for large datasets because, like HDF5, it is designed to store data on disk and only load the data into memory when needed. Zarr is also a good option for parallel computing because it supports concurrent reads and writes.

Note that the Zarr native storage formats are optimized for storage in cloud storage (e.g., S3). For very large files, Zarr will create many files which can lead to issues for traditional file system (that are not cloud object stores) due to limitations on the number of files per directory (this affects local disk, GDrive, Dropbox etc.).

Zarr read and write is provided by the hdmf-zarr package. First, create an an NWBFile using PyNWB.

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

import numpy as np
from pynwb import NWBFile, TimeSeries

# Create the NWBFile. Substitute your NWBFile generation here.
nwbfile = NWBFile(
    session_description="my first synthetic recording",
    identifier="EXAMPLE_ID",
    session_start_time=datetime.now(tzlocal()),
    session_id="LONELYMTN",
)
```

#### Dataset Configuration

Like HDF5, Zarr provides options to chunk and compress datasets. To leverage these features, replace all `H5DataIO` with the analogous `ZarrDataIO`, which takes compressors specified by the `numcodecs` library. For example, here is an example *TimeSeries* where the `data` Dataset is compressed with a Blosc-zstd compressor:

```
from numcodecs import Blosc
from hdmf_zarr import ZarrDataIO

data_with_zarr_data_io = ZarrDataIO(
```

(continues on next page)

(continued from previous page)

```
    data=np.random.randn(100, 100),
    chunks=(10, 10),
    fillvalue=0,
    compressor=Blosc(cname='zstd', clevel=3, shuffle=Blosc.SHUFFLE)
)
```

Now add it to the *NWBFile*.

```
nwbfile.add_acquisition(
    TimeSeries(
        name="synthetic_timeseries",
        data=data_with_zarr_data_io,
        unit="m",
        rate=10e3,
    )
)
```

### Writing to Zarr

To write NWB files to Zarr, replace the *NWBHDF5IO* with hdmf_zarr.nwb.NWBZarrIO.

```
from hdmf_zarr.nwb import NWBZarrIO
import os

path = "zarr_tutorial.nwb.zarr"
absolute_path = os.path.abspath(path)
with NWBZarrIO(path=path, mode="w") as io:
    io.write(nwbfile)
```

**Note:** The main reason for using the absolute_path here is for testing purposes to ensure links and references work as expected. Otherwise, using the relative path here instead is fine.

### Reading from Zarr

To read NWB files from Zarr, replace the *NWBHDF5IO* with the analogous hdmf_zarr.nwb.NWBZarrIO.

```
with NWBZarrIO(path=absolute_path, mode="r") as io:
    read_nwbfile = io.read()
```

**Note:** For more information, see the hdmf-zarr documentation.

CHAPTER

# THREE

# CITING PYNWB

## 3.1 BibTeX entry

If you use PyNWB in your research, please use the following citation:

```
@article {10.7554/eLife.78362,
      article_type = {journal},
      title = {{The Neurodata Without Borders ecosystem for neurophysiological data␣
↪science}},
      author = {R\"ubel, Oliver and Tritt, Andrew and Ly, Ryan and Dichter, Benjamin K.␣
↪and
                Ghosh, Satrajit and Niu, Lawrence and Baker, Pamela and Soltesz, Ivan and
                Ng, Lydia and Svoboda, Karel and Frank, Loren and Bouchard, Kristofer E.},
      editor = {Colgin, Laura L and Jadhav, Shantanu P},
      volume = {11{,
      year = {2022},
      month = {oct},
      pub_date = {2022-10-04},
      pages = {e78362},
      citation = {eLife 2022;11:e78362},
      doi = {10.7554/eLife.78362},
      url = {https://doi.org/10.7554/eLife.78362},
      keywords = {Neurophysiology, data ecosystem, data language, data standard, FAIR␣
↪data, archive},
      journal = {eLife},
      issn = {2050-084X},
      publisher = {eLife Sciences Publications, Ltd},
}
```

## 3.2 Using RRID

- ResourceID: SCR_017452
- Proper Citation: **(PyNWB, RRID:SCR_017452)**

## 3.3 Using duecredit

Citations can be generated using duecredit. To install duecredit, run `pip install duecredit`.

You can obtain a list of citations for your Python script, e.g., `yourscript.py`, using:

```
cd /path/to/your/module
python -m duecredit yourscript.py
```

Alternatively, you can set the environment variable `DUECREDIT_ENABLE=yes`

```
DUECREDIT-ENABLE=yes python yourscript.py
```

Citations will be saved in a hidden file (`.duecredit.p`) in the current directory. You can then use the duecredit command line tool to export the citations to different formats. For example, you can display your citations in BibTeX format using:

```
duecredit summary --format=bibtex
```

For more information on using duecredit, please consult its homepage.

CHAPTER

FOUR

# 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
```

If the file contains no NWB extensions, then this command will validate the file `test.nwb` against the *core* NWB specification. On success, the output will be:

```
Validating test.nwb against cached namespace information using namespace 'core'.
 - no errors found.
```

and the program exit code is `0`. On error, the program exit code is 1 and the list of errors is outputted.

If the file contains NWB extensions, then the above validation command will validate the file `test.nwb` against all extensions in the file and the core NWB specification.

To validate against only one NWB extension that is cached within the file, use the `-n` flag. For example, the following command will validate against the "ndx-my-extension" namespace that is cached within the `test.nwb` file.

```
python -m pynwb.validate -n ndx-my-extension test.nwb
```

To validate against the version of the **core** NWB specification that is included with the installed version of PyNWB, use the `--no-cached-namespace` flag. This can be useful in validating files against newer or older versions of the **core** NWB specification that are installed with newer or older versions of PyNWB.

```
python -m pynwb.validate --no-cached-namespace test.nwb
```

```
$python -m pynwb.validate --help
usage: validate.py [-h] [-n NS] [-lns] [--cached-namespace | --no-cached-namespace]␣
↪paths [paths ...]

Validate an NWB file

positional arguments:
  paths                 NWB file paths

optional arguments:
  -h, --help            show this help message and exit
  -n NS, --ns NS        the namespace to validate against
  -lns, --list-namespaces
                        List the available namespaces and exit.
  --cached-namespace    Use the cached namespace (default).
  --no-cached-namespace
```

(continues on next page)

(continued from previous page)

```
                        Don't use the cached namespace.

If --ns is not specified, validate against all namespaces in the NWB file.
```

Validation against a namespace that is not cached within the schema is not currently possible but is a planned feature.

CHAPTER

FIVE

# EXPORTING NWB FILES

You can use the export feature of PyNWB to create a modified version of an existing NWB file, while preserving the original file.

To do so, first open the NWB file using *NWBHDF5IO*. Then, read the NWB file into an *NWBFile* object, modify the *NWBFile* object or its child objects, and export the modified *NWBFile* object to a new file path. The modifications will appear in the exported file and not the original file.

These modifications can consist of removals of containers, additions of containers, and changes to container attributes. If container attributes are changed, then `NWBFile.set_modified()` must be called on the *NWBFile* before exporting.

```
with NWBHDF5IO(self.read_path, mode='r') as read_io:
    nwbfile = read_io.read()
    # ...  # modify nwbfile
    nwbfile.set_modified()  # this may be necessary if the modifications are changes to␣
↪attributes

    with NWBHDF5IO(self.export_path, mode='w') as export_io:
        export_io.export(src_io=read_io, nwbfile=nwbfile)
```

**Note:** Modifications to `h5py.Dataset` objects act *directly* on the read file on disk. Changes are applied immediately and do not require exporting or writing the file. If you want to modify a dataset only in the new file, than you should replace the whole object with a new array holding the modified data. To prevent unintentional changes to the source file, the source file should be opened with `mode='r'`.

**Note:** Moving containers within the same file is currently not supported directly via export. See the following discussion on the NWB Help Desk for details.

**Note:** After exporting an *NWBFile*, the object IDs of the *NWBFile* and its child containers will be identical to the object IDs of the read *NWBFile* and its child containers. The object ID of a container uniquely identifies the container within a file, but should *not* be used to distinguish between two different files.

**See also:**

The tutorial *Adding/Removing Containers from an NWB File* provides additional examples of adding and removing containers from an NWB file.

## 5.1 How do I create a copy of an NWB file with different data layouts (e.g., applying compression)?

Use the h5repack command line tool from the HDF5 Group. See also this h5repack tutorial.

## 5.2 How do I create a copy of an NWB file with different controls over how links are treated and whether copies are deep or shallow?

Use the h5copy command line tool from the HDF5 Group. See also this h5copy tutorial.

## 5.3 How do I generate new object IDs for a newly exported NWB file?

Before calling `export`, call the method `generate_new_id` on the *NWBFile* to generate a new set of object IDs for the `NWBFile` and all of its children, recursively. Then export the *NWBFile*. The original NWB file is preserved.

```
with NWBHDF5IO(self.read_path, manager=manager, mode='r') as read_io:
    nwbfile = read_io.read()
    # ...  # modify nwbfile if desired
    nwbfile.generate_new_id()

    with NWBHDF5IO(self.export_path, mode='w') as export_io:
        export_io.export(src_io=read_io, nwbfile=nwbfile)
```

## 5.4 My NWB file contains links to datasets in other HDF5 files. How do I create a new NWB file with copies of the datasets?

Pass the keyword argument `write_args={'link_data': False}` to *NWBHDF5IO.export*. This is similar to passing the keyword argument `link_data=False` to `NWBHDF5IO.write` when writing a file with a copy of externally linked datasets.

For example:

```
with NWBHDF5IO(self.read_path, mode='r') as read_io:
    nwbfile = read_io.read()
    # nwbfile contains a TimeSeries where the TimeSeries data array is a link to an␣
↪external dataset
    # in a different HDF5 file than self.read_path

    with NWBHDF5IO(self.export_path, mode='w') as export_io:
        export_io.export(src_io=read_io, nwbfile=nwbfile, write_args={'link_data': False}
↪)  # copy linked datasets
        # the written file will contain no links to external datasets
```

You can also the h5copy command line tool from the HDF5 Group. See also this h5copy tutorial.

## 5.5 How do I write a newly instantiated NWBFile to two different file paths?

PyNWB does not support writing an *NWBFile* that was not read from a file to two different files. For example, if you instantiate *NWBFile* A and write it to file path 1, you cannot also write it to file path 2. However, you can first write the *NWBFile* to file path 1, read the *NWBFile* from file path 1, and then export it to file path 2.

```
with NWBHDF5IO(self.filepath1, manager=manager, mode='w') as write_io:
    write_io.write(nwbfile)

with NWBHDF5IO(self.filepath1, manager=manager, mode='r') as read_io:
    read_nwbfile = read_io.read()

    with NWBHDF5IO(self.filepath2, mode='w') as export_io:
        export_io.export(src_io=read_io, nwbfile=nwbfile)
```

CHAPTER

SIX

# API DOCUMENTATION

## 6.1 pynwb.file module

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

Bases: *NWBContainer*

Container for storing lab-specific meta-data

The LabMetaData class serves as a base type for defining lab specific meta-data. To define your own lab-specific metadata, create a Neurodata Extension (NDX) for NWB that defines the data to add. Using the LabMetaData container as a base type makes it easy to add your data to an NWBFile without having to modify the NWBFile type itself, since adding of LabMetaData is already implemented. For more details on how to create an extension see the Extending NWB tutorial.

**Parameters**

**name** (str) – name of lab metadata

**namespace = 'core'**

**neurodata_type = 'LabMetaData'**

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

Bases: *NWBContainer*

Subject information and metadata.

**Parameters**

- **age** (str or timedelta) – The age of the subject. The ISO 8601 Duration format is recommended, e.g., "P90D" for 90 days old.A timedelta will automatically be converted to The ISO 8601 Duration format.
- **age__reference** (str) – Age is with reference to this event. Can be 'birth' or 'gestational'. If reference is omitted, then 'birth' is implied. Value can be None when read from an NWB file with schema version 2.0 to 2.5 where age__reference is missing.
- **description** (str) – A description of the subject, e.g., "mouse A10".
- **genotype** (str) – The genotype of the subject, e.g., "Sst-IRES-Cre/wt;Ai32(RCL-ChR2(H134R)_EYFP)/wt".
- **sex** (str) – The sex of the subject. Using "F" (female), "M" (male), "U" (unknown), or "O" (other) is recommended.
- **species** (str) – The species of the subject. The formal latin binomal name is recommended, e.g., "Mus musculus"

- **subject_id** (str) – A unique identifier for the subject, e.g., "A10"
- **weight** (float or str) – The weight of the subject, including units. Using kilograms is recommended. e.g., "0.02 kg". If a float is provided, then the weight will be stored as "[value] kg".
- **date_of_birth** (datetime or date) – The date of birth, which may include time and timezone. May be supplied instead of age.
- **strain** (str) – The strain of the subject, e.g., "C57BL/6J"

**property age**

The age of the subject. The ISO 8601 Duration format is recommended, e.g., "P90D" for 90 days old.A timedelta will automatically be converted to The ISO 8601 Duration format.

**property age__reference**

Age is with reference to this event. Can be 'birth' or 'gestational'. If reference is omitted, then 'birth' is implied. Value can be None when read from an NWB file with schema version 2.0 to 2.5 where age__reference is missing.

**property date_of_birth**

The date of birth, which may include time and timezone. May be supplied instead of age.

**property description**

A description of the subject, e.g., "mouse A10".

**property genotype**

The genotype of the subject, e.g., "Sst-IRES-Cre/wt;Ai32(RCL-ChR2(H134R)_EYFP)/wt".

**namespace = 'core'**

**neurodata_type = 'Subject'**

**property sex**

The sex of the subject. Using "F" (female), "M" (male), "U" (unknown), or "O" (other) is recommended.

**property species**

The species of the subject. The formal latin binomal name is recommended, e.g., "Mus musculus"

**property strain**

The strain of the subject, e.g., "C57BL/6J"

**property subject_id**

A unique identifier for the subject, e.g., "A10"

**property weight**

The weight of the subject, including units. Using kilograms is recommended. e.g., "0.02 kg". If a float is provided, then the weight will be stored as "[value] kg".

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*, *icephys_electrodes=None*, *icephys_filtering=None*, *intracellular_recordings=None*, *icephys_simultaneous_recordings=None*, *icephys_sequential_recordings=None*, *icephys_repetitions=None*, *icephys_experimental_conditions=None*)

Bases: *MultiContainerInterface*, HERDManager

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 or date) – the start date and time of the recording session
- **file_create_date** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or datetime or date) – the date and time the file was created and subsequent modifications made
- **timestamps_reference_time** (datetime or date) – 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 Array or StrDataset 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, DynamicTable, or NWBDataInterface 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) – DEPRECATED use icephys_electrodes parameter instead. 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
- **icephys_electrodes** (list or tuple) – IntracellularElectrodes that belong to this NWBFile.
- **icephys_filtering** (str) – [DEPRECATED] Use IntracellularElectrode.filtering instead. Description of filtering used.
- **intracellular_recordings** (*IntracellularRecordingsTable*) – the IntracellularRecordingsTable table that belongs to this NWBFile
- **icephys_simultaneous_recordings** (*SimultaneousRecordingsTable*) – the SimultaneousRecordingsTable table that belongs to this NWBFile

- **icephys_sequential_recordings** (*SequentialRecordingsTable*) – the SequentialRecordingsTable table that belongs to this NWBFile
- **icephys_repetitions** (*RepetitionsTable*) – the RepetitionsTable table that belongs to this NWBFile
- **icephys_experimental_conditions** (*ExperimentalConditionsTable*) – the ExperimentalConditionsTable table that belongs to this NWBFile

**all_children**()

Get a list of all child objects and their child objects recursively.

If the object has an object_id, the object will be added to "ret" to be returned. If that object has children, they will be added to the "stack" in order to be: 1) Checked to see if has an object_id, if so then add to "ret" 2) Have children that will also be checked

**property objects**

**property modules**

**property ec_electrode_groups**

**property ec_electrodes**

**property ic_electrodes**

**property icephys_filtering**

**add_ic_electrode**(**args*, ***kwargs*)

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

**create_ic_electrode**(**args*, ***kwargs*)

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

**get_ic_electrode**(**args*, ***kwargs*)

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

**add_epoch_column**(*name*, *description*, *data=[]*, *table=False*, *index=False*, *enum=False*, *col_cls=<class 'hdmf.common.table.VectorData'>*)

**Add a column to the epoch 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 Array or StrDataset 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or int) –

- False (default): do not generate a VectorIndex
- True: generate one empty VectorIndex
- VectorIndex: Use the supplied VectorIndex
- array-like of ints: Create a VectorIndex and use these values as the data
- int: Recursively create *n* VectorIndex objects for a multi-ragged array

- **enum** (bool or ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – whether or not this column contains data from a fixed set of elements
- **col_cls** (type) – class to use to represent the column data. If table=True, this field is ignored and a DynamicTableRegion object is used. If enum=True, this field is ignored and a EnumData object is used.

**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*, *enum=False*, *col_cls=<class 'hdmf.common.table.VectorData'>*)

**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 Array or StrDataset 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or int) –
  - False (default): do not generate a VectorIndex
  - True: generate one empty VectorIndex

- VectorIndex: Use the supplied VectorIndex
- array-like of ints: Create a VectorIndex and use these values as the data
- int: Recursively create *n* VectorIndex objects for a multi-ragged array

- **enum** (bool or ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – whether or not this column contains data from a fixed set of elements
- **col_cls** (type) – class to use to represent the column data. If table=True, this field is ignored and a DynamicTableRegion object is used. If enum=True, this field is ignored and a EnumData object is used.

**add_electrode**(*x=None*, *y=None*, *z=None*, *imp=None*, *location=None*, *filtering=None*, *group=None*, *id=None*, *rel_x=None*, *rel_y=None*, *rel_z=None*, *reference=None*, *enforce_unique_id=True*)

**Add an electrode to the electrodes table.**

See add_row for more details.

Required fields are *location* and *group* and any columns that have been added (through calls to *add_electrode_columns*).

**Parameters**

- **x** (float) – the x coordinate of the position (+x is posterior)
- **y** (float) – the y coordinate of the position (+y is inferior)
- **z** (float) – the z coordinate of the position (+z is right)
- **imp** (float) – the impedance of the electrode, in ohms
- **location** (str) – the location of electrode within the subject e.g. brain region. Required.
- **filtering** (str) – description of hardware filtering, including the filter name and frequency cutoffs
- **group** (*ElectrodeGroup*) – the ElectrodeGroup object to add to this NWBFile. Required.
- **id** (int) – a unique identifier for the electrode
- **rel_x** (float) – the x coordinate within the electrode group
- **rel_y** (float) – the y coordinate within the electrode group
- **rel_z** (float) – the z coordinate within the electrode group
- **reference** (str) – Description of the reference electrode and/or reference scheme used for this electrode, e.g.,"stainless steel skull screw" or "online common average referencing".
- **enforce_unique_id** (bool) – enforce that the id in the table must be unique

**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*, *enum=False*, *col_cls=<class 'hdmf.common.table.VectorData'>*)

**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 Array or StrDataset 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or int) –
  - False (default): do not generate a VectorIndex
  - True: generate one empty VectorIndex
  - VectorIndex: Use the supplied VectorIndex
  - array-like of ints: Create a VectorIndex and use these values as the data
  - int: Recursively create *n* VectorIndex objects for a multi-ragged array
- **enum** (bool or ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – whether or not this column contains data from a fixed set of elements
- **col_cls** (type) – class to use to represent the column data. If table=True, this field is ignored and a DynamicTableRegion object is used. If enum=True, this field is ignored and a EnumData object is used.

**add_unit**(*spike_times=None*, *obs_intervals=None*, *electrodes=None*, *electrode_group=None*, *waveform_mean=None*, *waveform_sd=None*, *waveforms=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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the spike times for each unit
- **obs_intervals** (ndarray or list or tuple or Dataset or Array or StrDataset 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 Array or StrDataset 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 Array or StrDataset 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the spike waveform standard deviation for each unit. Shape is (time,) or (time, electrodes)
- **waveforms** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – Individual waveforms for each spike. If the dataset is three-dimensional, the third dimension shows the response from different electrodes that all observe this unit simultaneously. In this case, the *electrodes* column of this Units table should be used to indicate which electrodes are associated with this unit, and the electrodes dimension here should be in the same order as the electrodes referenced in the *electrodes* column of this table.
- **id** (int) – the id for each unit

**add_trial_column**(*name*, *description*, *data=[]*, *table=False*, *index=False*, *enum=False*, *col_cls=<class 'hdmf.common.table.VectorData'>*)

**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 Array or StrDataset 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or int) –
  - False (default): do not generate a VectorIndex
  - True: generate one empty VectorIndex
  - VectorIndex: Use the supplied VectorIndex
  - array-like of ints: Create a VectorIndex and use these values as the data
  - int: Recursively create *n* VectorIndex objects for a multi-ragged array
- **enum** (bool or ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – whether or not this column contains data from a fixed set of elements
- **col_cls** (type) – class to use to represent the column data. If table=True, this field is ignored and a DynamicTableRegion object is used. If enum=True, this field is ignored and a EnumData object is used.

**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*, *enum=False*, *col_cls=<class 'hdmf.common.table.VectorData'>*)

**Add a column to the invalid times 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 Array or StrDataset 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or int) –
  - False (default): do not generate a VectorIndex
  - True: generate one empty VectorIndex
  - VectorIndex: Use the supplied VectorIndex
  - array-like of ints: Create a VectorIndex and use these values as the data
  - int: Recursively create *n* VectorIndex objects for a multi-ragged array
- **enum** (bool or ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – whether or not this column contains data from a fixed set of elements
- **col_cls** (type) – class to use to represent the column data. If table=True, this field is ignored and a DynamicTableRegion object is used. If enum=True, this field is ignored and a EnumData object is used.

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

**Add a time interval to the invalid times 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*).

**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

**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*, *use_sweep_table=False*)

**Parameters**

- **nwbdata** (*NWBDataInterface* or DynamicTable) – None
- **use_sweep_table** (bool) – Use the deprecated SweepTable

**add_stimulus**(*stimulus=None*, *use_sweep_table=False*, *timeseries=None*)

**Parameters**

- **stimulus** (*TimeSeries* or DynamicTable or *NWBDataInterface*) – The stimulus presentation data to add to this NWBFile.
- **use_sweep_table** (bool) – Use the deprecated SweepTable
- **timeseries** (*TimeSeries*) – The "timeseries" keyword argument is deprecated. Use the "nwbdata" argument instead.

**add_stimulus_template**(*timeseries*, *use_sweep_table=False*)

**Parameters**

- **timeseries** (*TimeSeries* or *Images*) – None
- **use_sweep_table** (bool) – Use the deprecated SweepTable

**get_intracellular_recordings**()

Get the NWBFile.intracellular_recordings table.

In contrast to NWBFile.intracellular_recordings, this function will create the IntracellularRecordingsTable table if not yet done, whereas NWBFile.intracellular_recordings will return None if the table is currently not being used.

**Returns**

The NWBFile.intracellular_recordings table

**Return type**

*IntracellularRecordingsTable*

**add_intracellular_recording**(*electrode=None*, *stimulus_start_index=None*, *stimulus_index_count=None*, *stimulus=None*, *stimulus_template_start_index=None*, *stimulus_template_index_count=None*, *stimulus_template=None*, *response_start_index=None*, *response_index_count=None*, *response=None*, *electrode_metadata=None*, *stimulus_metadata=None*, *response_metadata=None*)

**Add a intracellular recording to the intracellular_recordings table. If the**
electrode, stimulus, and/or response do not exist yet in the NWBFile, then they will be added to this NWBFile before adding them to the table.

Note: For more complex organization of intracellular recordings you may also be interested in the related SimultaneousRecordingsTable, SequentialRecordingsTable, RepetitionsTable, and ExperimentalConditionsTable tables and the related functions of NWBFile: add_icephys_simultaneous_recording, add_icephys_sequential_recording, add_icephys_repetition, and add_icephys_experimental_condition.

**Parameters**

- **electrode** (*IntracellularElectrode*) – The intracellular electrode used
- **stimulus_start_index** (int) – Start index of the stimulus
- **stimulus_index_count** (int) – Stop index of the stimulus
- **stimulus** (*TimeSeries*) – The TimeSeries (usually a PatchClampSeries) with the stimulus
- **stimulus_template_start_index** (int) – Start index of the stimulus template
- **stimulus_template_index_count** (int) – Stop index of the stimulus template
- **stimulus_template** (*TimeSeries*) – The TimeSeries (usually a PatchClampSeries) with the stimulus template waveforms
- **response_start_index** (int) – Start index of the response
- **response_index_count** (int) – Stop index of the response
- **response** (*TimeSeries*) – The TimeSeries (usually a PatchClampSeries) with the response
- **electrode_metadata** (dict) – Additional electrode metadata to be stored in the electrodes table
- **stimulus_metadata** (dict) – Additional stimulus metadata to be stored in the stimuli table
- **response_metadata** (dict) – Additional resposnse metadata to be stored in the responses table

**Returns**
Integer index of the row that was added to IntracellularRecordingsTable

**Return type**
int

**get_icephys_simultaneous_recordings()**
Get the NWBFile.icephys_simultaneous_recordings table.

In contrast to NWBFile.icephys_simultaneous_recordings, this function will create the SimultaneousRecordingsTable table if not yet done, whereas NWBFile.icephys_simultaneous_recordings will return None if the table is currently not being used.

**Returns**
The NWBFile.icephys_simultaneous_recordings table

**Return type**
*SimultaneousRecordingsTable*

**add_icephys_simultaneous_recording**(*recordings*)

Add a new simultaneous recording to the icephys_simultaneous_recordings table

**Parameters**
**recordings** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the indices of the recordings belonging to this simultaneous recording

**Returns**
Integer index of the row that was added to SimultaneousRecordingsTable

**Return type**
int

**get_icephys_sequential_recordings**()

Get the NWBFile.icephys_sequential_recordings table.

In contrast to NWBFile.icephys_sequential_recordings, this function will create the IntracellularRecordingsTable table if not yet done, whereas NWBFile.icephys_sequential_recordings will return None if the table is currently not being used.

**Returns**
The NWBFile.icephys_sequential_recordings table

**Return type**
*SequentialRecordingsTable*

**add_icephys_sequential_recording**(*stimulus_type*, *simultaneous_recordings*)

Add a new sequential recording to the icephys_sequential_recordings table

**Parameters**

- **stimulus_type** (str) – the type of stimulus used for the sequential recording
- **simultaneous_recordings** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the indices of the simultaneous_recordings belonging to this sequential recording

**Returns**
Integer index of the row that was added to SequentialRecordingsTable

**Return type**
int

**get_icephys_repetitions**()

Get the NWBFile.icephys_repetitions table.

In contrast to NWBFile.icephys_repetitions, this function will create the RepetitionsTable table if not yet done, whereas NWBFile.icephys_repetitions will return None if the table is currently not being used.

**Returns**
The NWBFile.icephys_repetitions table

**Return type**
*RepetitionsTable*

**add_icephys_repetition**(*sequential_recordings=None*)

Add a new repetition to the RepetitionsTable table

**Parameters**
**sequential_recordings** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the indices of the sequential recordings belonging to this repetition

**Returns**
Integer index of the row that was added to RepetitionsTable

**Return type**
int

**get_icephys_experimental_conditions**()

Get the NWBFile.icephys_experimental_conditions table.

In contrast to NWBFile.icephys_experimental_conditions, this function will create the RepetitionsTable table if not yet done, whereas NWBFile.icephys_experimental_conditions will return None if the table is currently not being used.

**Returns**
The NWBFile.icephys_experimental_conditions table

**Return type**
*ExperimentalConditionsTable*

**add_icephys_experimental_condition**(*repetitions=None*)

Add a new condition to the ExperimentalConditionsTable table

**Parameters**
**repetitions** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the indices of the repetitions belonging to this condition

**Returns**
Integer index of the row that was added to ExperimentalConditionsTable

**Return type**
int

**get_icephys_meta_parent_table**()

Get the top-most table in the intracellular ephys metadata table hierarchy that exists in this NWBFile.

The intracellular ephys metadata consists of a hierarchy of DynamicTables, i.e., experimental_conditions –> repetitions –> sequential_recordings –> simultaneous_recordings –> intracellular_recordings etc. In a given NWBFile not all tables may exist. This convenience functions returns the top-most table that exists in this file. E.g., if the file contains only the simultaneous_recordings and intracellular_recordings tables then the function would return the simultaneous_recordings table. Similarly, if the file contains all tables then it will return the experimental_conditions table.

**Returns**
DynamicTable object or None

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

Add data to the scratch space

**Parameters**

- **data** (str or int or float or bytes or bool or ndarray or list or tuple or DataFrame or DynamicTable or *NWBContainer* or *ScratchData*) – The data to add to the scratch space.
- **name** (str) – The name of the data. Required only when passing in a scalar, numpy.ndarray, list, or tuple
- **notes** (str) – Notes to add to the data. Only used when passing in numpy.ndarray, list, or tuple. This argument is not recommended. Use the *description* argument instead.
- **table_description** (str) – Description for the internal DynamicTable used to store a pandas.DataFrame. This argument is not recommended. Use the *description* argument instead.
- **description** (str) – Description of the data. Required only when passing in a scalar, numpy.ndarray, list, tuple, or pandas.DataFrame. Ignored when passing in an NWBContainer, DynamicTable, or ScratchData object.

**get_scratch**(*name*, *convert=True*)

Get data from the scratch space

**Parameters**

- **name** (str) – the name of the object to get
- **convert** (bool) – return the original data, not the NWB object

**copy**()

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

**property acquisition**

a dictionary containing the NWBDataInterface or DynamicTable in this NWBFile

**add_analysis**(*analysis*)

Add one or multiple NWBContainer or DynamicTable objects to this NWBFile

**Parameters**

**analysis** (list or tuple or dict or *NWBContainer* or DynamicTable) – one or multiple NWBContainer or DynamicTable objects to add to this NWBFile

**add_device**(*devices*)

Add one or multiple Device objects to this NWBFile

**Parameters**

**devices** (list or tuple or dict or *Device*) – one or multiple Device objects to add to this NWBFile

**add_electrode_group**(*electrode_groups*)

Add one or multiple ElectrodeGroup objects to this NWBFile

**Parameters**

**electrode_groups** (list or tuple or dict or *ElectrodeGroup*) – one or multiple ElectrodeGroup objects to add to this NWBFile

**add_icephys_electrode**(*icephys_electrodes*)

Add one or multiple IntracellularElectrode objects to this NWBFile

**Parameters**

**icephys_electrodes** (list or tuple or dict or *IntracellularElectrode*) – one or multiple IntracellularElectrode objects to add to this NWBFile

**add_imaging_plane**(*imaging_planes*)

Add one or multiple ImagingPlane objects to this NWBFile

**Parameters**
**imaging_planes** (list or tuple or dict or *ImagingPlane*) – one or multiple ImagingPlane objects to add to this NWBFile

**add_lab_meta_data**(*lab_meta_data*)

Add one or multiple LabMetaData objects to this NWBFile

**Parameters**
**lab_meta_data** (list or tuple or dict or *LabMetaData*) – one or multiple LabMetaData objects to add to this NWBFile

**add_ogen_site**(*ogen_sites*)

Add one or multiple OptogeneticStimulusSite objects to this NWBFile

**Parameters**
**ogen_sites** (list or tuple or dict or *OptogeneticStimulusSite*) – one or multiple OptogeneticStimulusSite objects to add to this NWBFile

**add_processing_module**(*processing*)

Add one or multiple ProcessingModule objects to this NWBFile

**Parameters**
**processing** (list or tuple or dict or *ProcessingModule*) – one or multiple ProcessingModule objects to add to this NWBFile

**add_time_intervals**(*intervals*)

Add one or multiple TimeIntervals objects to this NWBFile

**Parameters**
**intervals** (list or tuple or dict or *TimeIntervals*) – one or multiple TimeIntervals objects to add to this NWBFile

**property analysis**

a dictionary containing the NWBContainer or DynamicTable in this NWBFile

**create_device**(*name*, *description=None*, *manufacturer=None*)

Create a Device object and add it to this NWBFile

**Parameters**

- **name** (str) – the name of this device
- **description** (str) – Description of the device (e.g., model, firmware version, processing software version, etc.)
- **manufacturer** (str) – the name of the manufacturer of this device

**Returns**
the Device object that was created

**Return type**
*Device*

**create_electrode_group**(*name*, *description*, *location*, *device*, *position=None*)

Create an ElectrodeGroup object and add it to this NWBFile

**Parameters**

- **name** (str) – the name of this electrode group

- **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
- **position** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – stereotaxic position of this electrode group (x, y, z)

**Returns**
the ElectrodeGroup object that was created

**Return type**
*ElectrodeGroup*

**create_icephys_electrode**(*name*, *device*, *description*, *slice=None*, *seal=None*, *location=None*, *resistance=None*, *filtering=None*, *initial_access_resistance=None*, *cell_id=None*)

Create an IntracellularElectrode object 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).
- **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, unit - Ohm.
- **filtering** (str) – Electrode specific filtering.
- **initial_access_resistance** (str) – Initial access resistance.
- **cell_id** (str) – Unique ID of cell.

**Returns**
the IntracellularElectrode object that was created

**Return type**
*IntracellularElectrode*

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

Create an ImagingPlane object 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 channel-specific data.
- **description** (str) – Description of this ImagingPlane.

- **device** (*Device*) – the device that was used to record
- **excitation_lambda** (float) – Excitation wavelength in nm.
- **indicator** (str) – Calcium indicator
- **location** (str) – Location of image plane.
- **imaging_rate** (float) – Rate images are acquired, in Hz. If the corresponding TimeSeries is present, the rate should be stored there instead.
- **manifold** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – DEPRECATED - Physical position of each pixel. size=("height", "width", "xyz"). Deprecated in favor of origin_coords and grid_spacing.
- **conversion** (float) – DEPRECATED - Multiplier to get from stored values to specified unit (e.g., 1e-3 for millimeters) Deprecated in favor of origin_coords and grid_spacing.
- **unit** (str) – DEPRECATED - Base unit that coordinates are stored in (e.g., Meters). Deprecated in favor of origin_coords_unit and grid_spacing_unit.
- **reference_frame** (str) – Describes position and reference frame of manifold based on position of first element in manifold.
- **origin_coords** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – Physical location of the first element of the imaging plane (0, 0) for 2-D data or (0, 0, 0) for 3-D data. See also reference_frame for what the physical location is relative to (e.g., bregma).
- **origin_coords_unit** (str) – Measurement units for origin_coords. The default value is 'meters'.
- **grid_spacing** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – Space between pixels in (x, y) or voxels in (x, y, z) directions, in the specified unit. Assumes imaging plane is a regular grid. See also reference_frame to interpret the grid.
- **grid_spacing_unit** (str) – Measurement units for grid_spacing. The default value is 'meters'.

**Returns**
the ImagingPlane object that was created

**Return type**
*ImagingPlane*

**create_lab_meta_data**(*name*)

Create a LabMetaData object and add it to this NWBFile

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

**Returns**
the LabMetaData object that was created

**Return type**
*LabMetaData*

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

Create an OptogeneticStimulusSite object 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 object 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) – NWBDataInterfaces 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 object 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the ordered names of the columns in this table. columns must also be provided.

**Returns**
the TimeIntervals object that was created

**Return type**
*TimeIntervals*

**property data_collection**

Notes about data collection and analysis.

**property devices**

a dictionary containing the Device in this NWBFile

**property electrode_groups**

a dictionary containing the ElectrodeGroup in this NWBFile

**property electrodes**

the ElectrodeTable that belongs to this NWBFile

**property epoch_tags**

A sorted list of tags used across all epochs

**property epochs**

Epoch objects belonging to this NWBFile

**property experiment_description**

general description of the experiment

**property experimenter**

name of person who performed experiment

**property 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 NWBDataInterface or DynamicTable

**Returns**
the NWBDataInterface or DynamicTable with the given name

**Return type**
*NWBDataInterface* or DynamicTable

**get_analysis**(*name=None*)

Get a NWBContainer from this NWBFile

**Parameters**
**name** (str) – the name of the NWBContainer or DynamicTable

**Returns**
the NWBContainer or DynamicTable with the given name

**Return type**
*NWBContainer* or 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_icephys_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 NWBDataInterface from this NWBFile

**Parameters**
**name** (str) – the name of the NWBDataInterface or DynamicTable

**Returns**
the NWBDataInterface or DynamicTable with the given name

**Return type**
*NWBDataInterface* or DynamicTable

**get_stimulus_template**(*name=None*)

Get a TimeSeries from this NWBFile

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

**Returns**
the TimeSeries or Images with the given name

**Return type**
*TimeSeries* or *Images*

**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*

**property icephys_electrodes**

a dictionary containing the IntracellularElectrode in this NWBFile

**property icephys_experimental_conditions**

A table for grouping different intracellular recording repetitions together that belong to the same experimental experimental_conditions.

**property icephys_repetitions**

A table for grouping different intracellular recording sequential recordings together.With each SweepSequence typically representing a particular type of stimulus, the RepetitionsTable table is typically used to group sets of stimuli applied in sequence.

**property icephys_sequential_recordings**

A table for grouping different simultaneous intracellular recording from the SimultaneousRecordingsTable table together. This is typically used to group together simultaneous recordings where the a sequence of stimuli of the same type with varying parameters have been presented in a sequence.

**property icephys_simultaneous_recordings**

SimultaneousRecordingsTable table for grouping different intracellular recordings fromthe IntracellularRecordingsTable table together that were recorded simultaneously from different electrodes

**property identifier**

a unique text identifier for the file

**property imaging_planes**

a dictionary containing the ImagingPlane in this NWBFile

**property institution**

institution(s) where experiment is performed

**property intervals**

a dictionary containing the TimeIntervals in this NWBFile

**property intracellular_recordings**

IntracellularRecordingsTable table to group together a stimulus and response from a single intracellular electrode and a single simultaneous recording.

**property invalid_times**

A table containing times to be omitted from analysis

**property keywords**

Terms to search over

**property lab**

lab where experiment was performed

**property lab_meta_data**

a dictionary containing the LabMetaData in this NWBFile

**namespace = 'core'**

**neurodata_type = 'NWBFile'**

**property notes**

Notes about the experiment.

**property ogen_sites**

a dictionary containing the OptogeneticStimulusSite in this NWBFile

**property pharmacology**

Description of drugs used, including how and when they were administered. Anesthesia(s), painkiller(s), etc., plus dosage, concentration, etc.

**property processing**

a dictionary containing the ProcessingModule in this NWBFile

**property protocol**

Experimental protocol, if applicable. E.g., include IACUC protocol

**property related_publications**

Publication information.PMID, DOI, URL, etc. If multiple, concatenate together and describe which is which. such as PMID, DOI, URL, etc

**property scratch**

a dictionary containing the DynamicTable, NWBContainer, or ScratchData in this NWBFile

**property session_description**

a description of the session where this data was generated

**property session_id**

lab-specific ID for the session

**property session_start_time**

the start date and time of the recording session

**property slices**

Description of slices, including information about preparation thickness, orientation, temperature and bath solution

**property source_script**

Script file used to create this NWB file.

**property source_script_file_name**

Name of the source_script file

**property stimulus**

a dictionary containing the NWBDataInterface or DynamicTable in this NWBFile

**property stimulus_notes**

Notes about stimuli, such as how and where presented.

**property stimulus_template**

a dictionary containing the TimeSeries or Images in this NWBFile

**property subject**

subject metadata

**property surgery**

Narrative description about surgery/surgeries, including date(s) and who performed surgery.

**property sweep_table**

the SweepTable that belong to this NWBFile

**property timestamps_reference_time**

date and time corresponding to time zero of all timestamps; defaults to value of session_start_time

**property trials**

A table containing trial data

**property units**

A table containing unit metadata

**property 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'*)

## 6.2 pynwb.ecephys module

class pynwb.ecephys.**ElectrodeGroup**(*name*, *description*, *location*, *device*, *position=None*)

Bases: *NWBContainer*

Defines a related group of electrodes.

**Parameters**

- **name** (str) – the name of this electrode group
- **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
- **position** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – stereotaxic position of this electrode group (x, y, z)

**property description**

description of this electrode group

**property device**

the device that was used to record from this electrode group

**property location**

description of location of this electrode group

**namespace = 'core'**

**neurodata_type = 'ElectrodeGroup'**

**property position**

stereotaxic position of this electrode group (x, y, z)

class pynwb.ecephys.**ElectricalSeries**(*name*, *data*, *electrodes*, *channel_conversion=None*, *filtering=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*, *offset=0.0*)

Bases: *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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. Can be 1D or 2D. The first dimension must be time. The second dimension represents electrodes/channels.
- **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 Array or StrDataset 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.
- **filtering** (str) – Filtering applied to all channels of the data. For example, if this ElectricalSeries represents high-pass-filtered data (also known as AP Band), then this value could be 'High-pass 4-pole Bessel filter at 500 Hz'. If this ElectricalSeries represents low-pass-filtered LFP data and the type of filter is unknown, then this value could be 'Low-pass filter at 300 Hz'. If a non-standard filter type is used, provide as much detail about the filter properties as possible.
- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.

**property 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.

**property electrodes**

the electrodes that generated this electrical series

**property filtering**

Filtering applied to all channels of the data. For example, if this ElectricalSeries represents high-pass-filtered data (also known as AP Band), then this value could be 'High-pass 4-pole Bessel filter at 500 Hz'. If this ElectricalSeries represents low-pass-filtered LFP data and the type of filter is unknown, then this value could be 'Low-pass filter at 300 Hz'. If a non-standard filter type is used, provide as much detail about the filter properties as possible.

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*, *offset=0.0*)

Bases: *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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. Can be 1D or 2D. The first dimension must be time. The second dimension represents electrodes/channels.
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (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
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.

namespace = 'core'

neurodata_type = 'SpikeEventSeries'

class pynwb.ecephys.**EventDetection**(*detection_method*, *source_electricalseries*, *source_idx*, *times*, *name='EventDetection'*)

Bases: *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 Array or StrDataset 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – Timestamps of events, in Seconds
- **name** (str) – the name of this container

**property detection_method**

Description of how events were detected, such as voltage threshold, or dV/dT threshold, as well as relevant values.

**namespace = 'core'**

**neurodata_type = 'EventDetection'**

**property source_electricalseries**

The source electrophysiology data

**property 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

**property times**

Timestamps of events, in Seconds

**class** pynwb.ecephys.**EventWaveform**(*spike_event_series={}*, *name='EventWaveform'*)

Bases: *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 one or multiple SpikeEventSeries objects to this EventWaveform

**Parameters**
**spike_event_series** (list or tuple or dict or *SpikeEventSeries*) – one or multiple SpikeEventSeries objects to add to this EventWaveform

**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*, *offset=0.0*)

Create a SpikeEventSeries object and add it to this EventWaveform

**Parameters**

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. Can be 1D or 2D. The first dimension must be time. The second dimension represents electrodes/channels.
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (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
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.

**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'**

**property spike_event_series**

a dictionary containing the SpikeEventSeries in this EventWaveform

class pynwb.ecephys.**Clustering**(*description*, *num*, *peak_over_rms*, *times*, *name='Clustering'*)

Bases: *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 Array or StrDataset 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – Times of clustered events, in seconds.
- **name** (str) – the name of this container

**property description**

Description of clusters or clustering, (e.g. cluster 0 is noise, clusters curated using Klusters, etc).

**namespace = 'core'**

**neurodata_type = 'Clustering'**

**property num**

Cluster number of each event.

**property peak_over_rms**

Maximum ratio of waveform peak to RMS on any channel in the cluster(provides a basic clustering metric).

**property times**

Times of clustered events, in seconds.

class pynwb.ecephys.**ClusterWaveforms**(*clustering_interface*, *waveform_filtering*, *waveform_mean*, *waveform_sd*, *name='ClusterWaveforms'*)

Bases: *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

**property clustering_interface**

the clustered spike data used as input for computing waveforms

**namespace = 'core'**

**neurodata_type = 'ClusterWaveforms'**

**property waveform_filtering**

filter applied to data before calculating mean and standard deviation

**property waveform_mean**

the mean waveform for each cluster

**property waveform_sd**

the standard deviations of waveforms for each cluster

**class** pynwb.ecephys.**LFP**(*electrical_series={}*, *name='LFP'*)

Bases: *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 one or multiple ElectricalSeries objects to this LFP

**Parameters**
**electrical_series** (list or tuple or dict or *ElectricalSeries*) – one or multiple ElectricalSeries objects to add to this LFP

**create_electrical_series**(*name*, *data*, *electrodes*, *channel_conversion=None*, *filtering=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*, *offset=0.0*)

Create an ElectricalSeries object and add it to this LFP

**Parameters**

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. Can

be 1D or 2D. The first dimension must be time. The second dimension represents electrodes/channels.

- **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 Array or StrDataset 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.
- **filtering** (str) – Filtering applied to all channels of the data. For example, if this ElectricalSeries represents high-pass-filtered data (also known as AP Band), then this value could be 'High-pass 4-pole Bessel filter at 500 Hz'. If this ElectricalSeries represents low-pass-filtered LFP data and the type of filter is unknown, then this value could be 'Low-pass filter at 300 Hz'. If a non-standard filter type is used, provide as much detail about the filter properties as possible.
- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.

**Returns**
the ElectricalSeries object that was created

**Return type**
*ElectricalSeries*

**property electrical_series**

a dictionary containing the ElectricalSeries in this LFP

**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: *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 one or multiple ElectricalSeries objects to this FilteredEphys

**Parameters**
**electrical_series** (list or tuple or dict or *ElectricalSeries*) – one or multiple ElectricalSeries objects to add to this FilteredEphys

**create_electrical_series**(*name*, *data*, *electrodes*, *channel_conversion=None*, *filtering=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*, *offset=0.0*)

Create an ElectricalSeries object and add it to this FilteredEphys

**Parameters**

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. Can be 1D or 2D. The first dimension must be time. The second dimension represents electrodes/channels.

- **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 Array or StrDataset 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.
- **filtering** (str) – Filtering applied to all channels of the data. For example, if this ElectricalSeries represents high-pass-filtered data (also known as AP Band), then this value could be 'High-pass 4-pole Bessel filter at 500 Hz'. If this ElectricalSeries represents low-pass-filtered LFP data and the type of filter is unknown, then this value could be 'Low-pass filter at 300 Hz'. If a non-standard filter type is used, provide as much detail about the filter properties as possible.
- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.

**Returns**
the ElectricalSeries object that was created

**Return type**
*ElectricalSeries*

**property electrical_series**

a dictionary containing the ElectricalSeries in this FilteredEphys

**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: *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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – A description for each feature extracted
- **times** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – The times of events that features correspond to
- **features** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – Features for each channel
- **name** (str) – the name of this container

**property electrodes**

the table region corresponding to the electrodes from which this series was recorded

**property description**

A description for each feature extracted

**property times**

The times of events that features correspond to

**property features**

Features for each channel

**namespace = 'core'**

**neurodata_type = 'FeatureExtraction'**

## 6.3 pynwb.icephys module

pynwb.icephys.**ensure_unit**(*self*, *name*, *current_unit*, *unit*, *nwb_version*)

A helper to ensure correct unit used.

Issues a warning with details if *current_unit* is to be ignored, and *unit* to be used instead.

**class** pynwb.icephys.**IntracellularElectrode**(*name*, *device*, *description*, *slice=None*, *seal=None*, *location=None*, *resistance=None*, *filtering=None*, *initial_access_resistance=None*, *cell_id=None*)

Bases: *NWBContainer*

Describes an intracellular electrode and associated metadata.

**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).
- **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, unit - Ohm.
- **filtering** (str) – Electrode specific filtering.
- **initial_access_resistance** (str) – Initial access resistance.
- **cell_id** (str) – Unique ID of cell.

**property cell_id**

Unique ID of cell.

**property description**

Recording description, description of electrode (e.g., whole-cell, sharp, etc).

**property device**

the device that was used to record from this electrode

**property filtering**

Electrode specific filtering.

**property initial_access_resistance**

Initial access resistance.

**property location**

Area, layer, comments on estimation, stereotaxis coordinates (if in vivo, etc).

**namespace = 'core'**

**neurodata_type = 'IntracellularElectrode'**

**property resistance**

Electrode resistance, unit - Ohm.

**property seal**

Information about seal used for recording.

**property slice**

Information about slice used for recording.

**class** pynwb.icephys.**PatchClampSeries**(*name*, *data*, *unit*, *electrode*, *gain*, *stimulus_description='N/A'*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*, *offset=0.0*, *sweep_number=None*)

Bases: *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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. The first dimension must be time.
- **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** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.
- **sweep_number** (int or uint32 or uint64) – Sweep number, allows for grouping different PatchClampSeries together via the sweep_table

**property electrode**

IntracellularElectrode group that describes the electrode that was used to apply or record this data.

**property gain**

Volt/Amp (v-clamp) or Volt/Volt (c-clamp)

**Type**
*Units*

**property stimulus_description**

the stimulus name/protocol

**property 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='N/A'*, *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*, *offset=0.0*, *unit='volts'*)

Bases: *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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. The first dimension must be time.
- **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** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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 uint32 or uint64) – Sweep number, allows for grouping different PatchClampSeries together via the sweep_table

- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.
- **unit** (str) – The base unit of measurement (must be 'volts')

**property bias_current**

Unit - Amp

**property bridge_balance**

Unit - Ohm

**property capacitance_compensation**

Unit - Farad

**namespace = 'core'**

**neurodata_type = 'CurrentClampSeries'**

**class** pynwb.icephys.**IZeroClampSeries**(*name*, *data*, *electrode*, *gain*, *stimulus_description='N/A'*, *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*, *offset=0.0*, *unit='volts'*)

Bases: *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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. The first dimension must be time.
- **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. Setting this to a value other than "N/A" is deprecated as of NWB 2.3.0.
- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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 uint32 or uint64) – Sweep number, allows for grouping different PatchClampSeries together via the sweep_table
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.
- **unit** (str) – The base unit of measurement (must be 'volts')

**namespace = 'core'**

**neurodata_type = 'IZeroClampSeries'**

class pynwb.icephys.**CurrentClampStimulusSeries**(*name*, *data*, *electrode*, *gain*, *stimulus_description='N/A'*, *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*, *offset=0.0*, *unit='amperes'*)

Bases: *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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. The first dimension must be time.
- **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** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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 uint32 or uint64) – Sweep number, allows for grouping different PatchClampSeries together via the sweep_table
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.
- **unit** (str) – The base unit of measurement (must be 'amperes')

**namespace = 'core'**

**neurodata_type = 'CurrentClampStimulusSeries'**

**class** pynwb.icephys.**VoltageClampSeries**(*name*, *data*, *electrode*, *gain*, *stimulus_description='N/A'*, *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*, *offset=0.0*, *unit='amperes'*)

Bases: *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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. The first dimension must be time.
- **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** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit

- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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 uint32 or uint64) – Sweep number, allows for grouping different PatchClampSeries together via the sweep_table
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.
- **unit** (str) – The base unit of measurement (must be 'amperes')

**property capacitance_fast**

Unit - Farad

**property capacitance_slow**

Unit - Farad

**property resistance_comp_bandwidth**

Unit - Hz

**property resistance_comp_correction**

Unit - percent

**property resistance_comp_prediction**

Unit - percent

**property whole_cell_capacitance_comp**

Unit - Farad

**property whole_cell_series_resistance_comp**

Unit - Ohm

**namespace = 'core'**

**neurodata_type = 'VoltageClampSeries'**

**class** pynwb.icephys.**VoltageClampStimulusSeries**(*name*, *data*, *electrode*, *gain*, *stimulus_description='N/A'*, *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*, *offset=0.0*, *unit='volts'*)

Bases: *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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. The first dimension must be time.
- **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** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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 uint32 or uint64) – Sweep number, allows for grouping different PatchClampSeries together via the sweep_table
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.
- **unit** (str) – The base unit of measurement (must be 'volts')

**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: 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the ordered names of the columns in this table. columns must also be provided.

**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'**

**class** pynwb.icephys.**IntracellularElectrodesTable**(*id=None*, *columns=None*, *colnames=None*)

Bases: DynamicTable

Table for storing intracellular electrode related metadata'

**Parameters**

- **id** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the ordered names of the columns in this table. columns must also be provided.

**namespace = 'core'**

**neurodata_type = 'IntracellularElectrodesTable'**

**class** pynwb.icephys.**IntracellularStimuliTable**(*id=None*, *columns=None*, *colnames=None*)

Bases: DynamicTable

Table for storing intracellular electrode related metadata'

**Parameters**

- **id** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the ordered names of the columns in this table. columns must also be provided.

**namespace = 'core'**

**neurodata_type = 'IntracellularStimuliTable'**

class pynwb.icephys.**IntracellularResponsesTable**(*id=None*, *columns=None*, *colnames=None*)

Bases: DynamicTable

Table for storing intracellular electrode related metadata'

**Parameters**

- **id** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the ordered names of the columns in this table. columns must also be provided.

**namespace = 'core'**

**neurodata_type = 'IntracellularResponsesTable'**

class pynwb.icephys.**IntracellularRecordingsTable**(*id=None*, *columns=None*, *colnames=None*, *category_tables=None*, *categories=None*)

Bases: AlignedDynamicTable

A table to group together a stimulus and response from a single electrode and a single simultaneous_recording. Each row in the table represents a single recording consisting typically of a stimulus and a corresponding response.

**Parameters**

- **id** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the ordered names of the columns in this table. columns must also be provided.
- **category_tables** (list) – List of DynamicTables to be added to the container. NOTE - Only regular DynamicTables are allowed. Using AlignedDynamicTable as a category for AlignedDynamicTable is currently not supported.
- **categories** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – List of names with the ordering of category tables

**add_recording**(*electrode=None*, *stimulus_start_index=None*, *stimulus_index_count=None*, *stimulus=None*, *stimulus_template_start_index=None*, *stimulus_template_index_count=None*, *stimulus_template=None*, *response_start_index=None*, *response_index_count=None*, *response=None*, *electrode_metadata=None*, *stimulus_metadata=None*, *response_metadata=None*)

Add a single recording to the IntracellularRecordingsTable table.

Typically, both stimulus and response are expected. However, in some cases only a stimulus or a response may be recodred as part of a recording. In this case, None may be given for either stimulus or response, but not both. Internally, this results in both stimulus and response pointing to the same TimeSeries, while the start_index and index_count for the invalid series will both be set to -1.

**Parameters**

- **electrode** (*IntracellularElectrode*) – The intracellular electrode used
- **stimulus_start_index** (int) – Start index of the stimulus
- **stimulus_index_count** (int) – Stop index of the stimulus
- **stimulus** (*TimeSeries*) – The TimeSeries (usually a PatchClampSeries) with the stimulus
- **stimulus_template_start_index** (int) – Start index of the stimulus template
- **stimulus_template_index_count** (int) – Stop index of the stimulus template
- **stimulus_template** (*TimeSeries*) – The TimeSeries (usually a PatchClampSeries) with the stimulus template waveforms
- **response_start_index** (int) – Start index of the response
- **response_index_count** (int) – Stop index of the response
- **response** (*TimeSeries*) – The TimeSeries (usually a PatchClampSeries) with the response
- **electrode_metadata** (dict) – Additional electrode metadata to be stored in the electrodes table
- **stimulus_metadata** (dict) – Additional stimulus metadata to be stored in the stimuli table
- **response_metadata** (dict) – Additional resposnse metadata to be stored in the responses table

**Returns**

Integer index of the row that was added to this table

**Return type**

int

**to_dataframe**(*ignore_category_ids=False*, *electrode_refs_as_objectids=False*, *stimulus_refs_as_objectids=False*, *response_refs_as_objectids=False*)

Convert the collection of tables to a single pandas DataFrame

**Parameters**

- **ignore_category_ids** (bool) – Ignore id columns of sub-category tables
- **electrode_refs_as_objectids** (bool) – replace object references in the electrode column with object_ids
- **stimulus_refs_as_objectids** (bool) – replace object references in the stimulus column with object_ids
- **response_refs_as_objectids** (bool) – replace object references in the response column with object_ids

**namespace = 'core'**

**neurodata_type = 'IntracellularRecordingsTable'**

class pynwb.icephys.**SimultaneousRecordingsTable**(*intracellular_recordings_table=None*, *id=None*, *columns=None*, *colnames=None*)

Bases: DynamicTable

A table for grouping different intracellular recordings from the IntracellularRecordingsTable table together that were recorded simultaneously from different electrodes.

**Parameters**

- **intracellular_recordings_table** (*IntracellularRecordingsTable*) – the IntracellularRecordingsTable table that the recordings column indexes. May be None when reading the Container from file as the table attribute is already populated in this case but otherwise this is required.
- **id** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the ordered names of the columns in this table. columns must also be provided.

**add_simultaneous_recording**(*recordings*)

**Add a single simultaneous recording (i.e., one sweep, or one row) consisting of one or more recordings and associated custom simultaneous recording metadata to the table.**

**Parameters**

**recordings** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the indices of the recordings belonging to this simultaneous recording

**Returns**

Integer index of the row that was added to this table

**Return type**

int

**namespace = 'core'**

**neurodata_type = 'SimultaneousRecordingsTable'**

class pynwb.icephys.**SequentialRecordingsTable**(*simultaneous_recordings_table=None*, *id=None*, *columns=None*, *colnames=None*)

Bases: DynamicTable

A table for grouping different intracellular recording simultaneous_recordings from the SimultaneousRecordingsTable table together. This is typically used to group together simultaneous_recordings where the a sequence of stimuli of the same type with varying parameters have been presented in a sequence.

**Parameters**

- **simultaneous_recordings_table** (*SimultaneousRecordingsTable*) – the SimultaneousRecordingsTable table that the simultaneous_recordings column indexes. May be None when reading the Container from file as the table attribute is already populated in this case but otherwise this is required.

- **id** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the ordered names of the columns in this table. columns must also be provided.

**add_sequential_recording**(*stimulus_type*, *simultaneous_recordings*)

**Add a sequential recording (i.e., one row) consisting of one or more simultaneous recordings** and associated custom sequential recording metadata to the table.

**Parameters**

- **stimulus_type** (str) – the type of stimulus used for the sequential recording
- **simultaneous_recordings** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the indices of the simultaneous_recordings belonging to this sequential recording

**Returns**
Integer index of the row that was added to this table

**Return type**
int

**namespace = 'core'**

**neurodata_type = 'SequentialRecordingsTable'**

**class** pynwb.icephys.**RepetitionsTable**(*sequential_recordings_table=None*, *id=None*, *columns=None*, *colnames=None*)

Bases: DynamicTable

A table for grouping different intracellular recording sequential recordings together. With each SweepSequence typically representing a particular type of stimulus, the RepetitionsTable table is typically used to group sets of stimuli applied in sequence.

**Parameters**

- **sequential_recordings_table** (*SequentialRecordingsTable*) – the SequentialRecordingsTable table that the sequential_recordings column indexes. May be None when reading the Container from file as the table attribute is already populated in this case but otherwise this is required.
- **id** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the ordered names of the columns in this table. columns must also be provided.

**add_repetition**(*sequential_recordings=None*)

**Add a repetition (i.e., one row) consisting of one or more sequential recordings** and associated custom repetition metadata to the table.

**Parameters**
**sequential_recordings** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the indices of the sequential recordings belonging to this repetition

**Returns**
Integer index of the row that was added to this table

**Return type**
int

**namespace = 'core'**

**neurodata_type = 'RepetitionsTable'**

**class** pynwb.icephys.**ExperimentalConditionsTable**(*repetitions_table=None*, *id=None*, *columns=None*, *colnames=None*)

Bases: DynamicTable

A table for grouping different intracellular recording repetitions together that belong to the same experimental conditions.

**Parameters**

- **repetitions_table** (*RepetitionsTable*) – the RepetitionsTable table that the repetitions column indexes
- **id** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the ordered names of the columns in this table. columns must also be provided.

**add_experimental_condition**(*repetitions=None*)

**Add a condition (i.e., one row) consisting of one or more repetitions of sequential recordings** and associated custom experimental_conditions metadata to the table.

**Parameters**
**repetitions** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the indices of the repetitions belonging to this condition

**Returns**
Integer index of the row that was added to this table

**Return type**
int

**namespace = 'core'**

**neurodata_type = 'ExperimentalConditionsTable'**

## 6.4 pynwb.ophys module

class pynwb.ophys.**OpticalChannel**(*name*, *description*, *emission_lambda*)

Bases: *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 wavelength for channel, in nm.

property **description**

Any notes or comments about the channel.

property **emission_lambda**

Emission wavelength for channel, in nm.

**namespace = 'core'**

**neurodata_type = 'OpticalChannel'**

class pynwb.ophys.**ImagingPlane**(*name*, *optical_channel*, *description*, *device*, *excitation_lambda*, *indicator*, *location*, *imaging_rate=None*, *manifold=None*, *conversion=1.0*, *unit='meters'*, *reference_frame=None*, *origin_coords=None*, *origin_coords_unit='meters'*, *grid_spacing=None*, *grid_spacing_unit='meters'*)

Bases: *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 channel-specific data.
- **description** (str) – Description of this ImagingPlane.
- **device** (*Device*) – the device that was used to record
- **excitation_lambda** (float) – Excitation wavelength in nm.
- **indicator** (str) – Calcium indicator
- **location** (str) – Location of image plane.
- **imaging_rate** (float) – Rate images are acquired, in Hz. If the corresponding TimeSeries is present, the rate should be stored there instead.
- **manifold** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – DEPRECATED - Physical position of each pixel. size=(“height”, “width”, “xyz”). Deprecated in favor of origin_coords and grid_spacing.
- **conversion** (float) – DEPRECATED - Multiplier to get from stored values to specified unit (e.g., 1e-3 for millimeters) Deprecated in favor of origin_coords and grid_spacing.

- **unit** (str) – DEPRECATED - Base unit that coordinates are stored in (e.g., Meters). Deprecated in favor of origin_coords_unit and grid_spacing_unit.
- **reference_frame** (str) – Describes position and reference frame of manifold based on position of first element in manifold.
- **origin_coords** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – Physical location of the first element of the imaging plane (0, 0) for 2-D data or (0, 0, 0) for 3-D data. See also reference_frame for what the physical location is relative to (e.g., bregma).
- **origin_coords_unit** (str) – Measurement units for origin_coords. The default value is 'meters'.
- **grid_spacing** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – Space between pixels in (x, y) or voxels in (x, y, z) directions, in the specified unit. Assumes imaging plane is a regular grid. See also reference_frame to interpret the grid.
- **grid_spacing_unit** (str) – Measurement units for grid_spacing. The default value is 'meters'.

**property conversion**

DEPRECATED - Multiplier to get from stored values to specified unit (e.g., 1e-3 for millimeters) Deprecated in favor of origin_coords and grid_spacing.

**property description**

Description of this ImagingPlane.

**property device**

the device that was used to record

**property excitation_lambda**

Excitation wavelength in nm.

**property imaging_rate**

Rate images are acquired, in Hz. If the corresponding TimeSeries is present, the rate should be stored there instead.

**property indicator**

Calcium indicator

**property location**

Location of image plane.

**property manifold**

DEPRECATED - Physical position of each pixel. size=("height", "width", "xyz"). Deprecated in favor of origin_coords and grid_spacing.

**namespace = 'core'**

**neurodata_type = 'ImagingPlane'**

**property optical_channel**

One of possibly many groups storing channel-specific data.

**property reference_frame**

Describes position and reference frame of manifold based on position of first element in manifold.

property unit

DEPRECATED - Base unit that coordinates are stored in (e.g., Meters). Deprecated in favor of origin_coords_unit and grid_spacing_unit.

class pynwb.ophys.**OnePhotonSeries**(*name*, *imaging_plane*, *data=None*, *unit=None*, *format=None*, *pmt_gain=None*, *scan_line_rate=None*, *exposure_time=None*, *binning=None*, *power=None*, *intensity=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*, *device=None*, *offset=0.0*)

Bases: *ImageSeries*

Image stack recorded over time from 1-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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. Can be 3D or 4D. The first dimension must be time (frame). The second and third dimensions represent x and y. The optional fourth dimension represents z. Either data or external_file must be specified (not None), but not both. If data is not specified, data will be set to an empty 3D array.
- **unit** (str) – The unit of measurement of the image data, e.g., values between 0 and 255. Required when data is specified. If unit (and data) are not specified, then unit will be set to "unknown".
- **format** (str) – Format of image. Three types - 1) Image format; tiff, png, jpg, etc. 2) external 3) raw.
- **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.
- **exposure_time** (float) – Exposure time of the sample; often the inverse of the frequency.
- **binning** (int or uint) – Amount of pixels combined into 'bins'; could be 1, 2, 4, 8, etc.
- **power** (float) – Power of the excitation in mW, if known.
- **intensity** (float) – Intensity of the excitation in mW/mm^2, if known.
- **external_file** (ndarray or list or tuple or Dataset or Array or StrDataset 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 (not None), but not both.
- **starting_frame** (Iterable) – Each entry is a frame number that corresponds to the first frame of each file listed in external_file within the full ImageSeries.
- **bits_per_pixel** (int) – DEPRECATED: Number of bits per image pixel
- **dimension** (Iterable) – Number of pixels on x, y, (and z) axes.

- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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
- **device** (*Device*) – Device used to capture the images/video.
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.

**property binning**

Amount of pixels combined into 'bins'; could be 1, 2, 4, 8, etc.

**property exposure_time**

Exposure time of the sample; often the inverse of the frequency.

**property imaging_plane**

Imaging plane class/pointer.

**property intensity**

Intensity of the excitation in mW/mm^2, if known.

**namespace = 'core'**

**neurodata_type = 'OnePhotonSeries'**

**property pmt_gain**

Photomultiplier gain.

**property power**

Power of the excitation in mW, if known.

**property 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.

**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*, *device=None*, *offset=0.0*)

Bases: *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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. Can be 3D or 4D. The first dimension must be time (frame). The second and third dimensions represent x and y. The optional fourth dimension represents z. Either data or external_file must be specified (not None), but not both. If data is not specified, data will be set to an empty 3D array.
- **unit** (str) – The unit of measurement of the image data, e.g., values between 0 and 255. Required when data is specified. If unit (and data) are not specified, then unit will be set to "unknown".
- **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 Array or StrDataset 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 (not None), but not both.
- **starting_frame** (Iterable) – Each entry is a frame number that corresponds to the first frame of each file listed in external_file within the full ImageSeries.
- **bits_per_pixel** (int) – DEPRECATED: Number of bits per image pixel
- **dimension** (Iterable) – Number of pixels on x, y, (and z) axes.
- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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
- **device** (*Device*) – Device used to capture the images/video.
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.

**property field_of_view**

Width, height and depth of image, or imaged area (meters).

**property imaging_plane**

Imaging plane class/pointer.

**namespace = 'core'**

**neurodata_type = 'TwoPhotonSeries'**

**property pmt_gain**

Photomultiplier gain.

**property 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.

**class** pynwb.ophys.**CorrectedImageStack**(*corrected*, *original*, *xy_translation*, *name='CorrectedImageStack'*)

Bases: *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. This must have the name "corrected".
- **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. This must have the name "xy_translation".
- **name** (str) – The name of this CorrectedImageStack container

**property corrected**

Image stack with frames shifted to the common coordinates. This must have the name "corrected".

**property original**

Link to image series that is being registered.

**property 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. This must have the name "xy_translation".

**namespace = 'core'**

**neurodata_type = 'CorrectedImageStack'**

**class** pynwb.ophys.**MotionCorrection**(*corrected_image_stacks={}*, *name='MotionCorrection'*)

Bases: *MultiContainerInterface*

A collection of corrected images stacks.

**Parameters**

- **corrected_image_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_image_stacks*)

Add one or multiple CorrectedImageStack objects to this MotionCorrection

**Parameters**
**corrected_image_stacks** (list or tuple or dict or *CorrectedImageStack*) – one or multiple CorrectedImageStack objects to add to this MotionCorrection

**property corrected_image_stacks**

a dictionary containing the CorrectedImageStack in this MotionCorrection

**create_corrected_image_stack**(*corrected*, *original*, *xy_translation*, *name='CorrectedImageStack'*)

Create a CorrectedImageStack object and add it to this MotionCorrection

**Parameters**

- **corrected** (*ImageSeries*) – Image stack with frames shifted to the common coordinates. This must have the name "corrected".
- **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. This must have the name "xy_translation".
- **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: 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the ordered names of the columns in this table. columns must also be provided.

property imaging_plane

the ImagingPlane this ROI applies to

property 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 Array or StrDataset 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 Array or StrDataset 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 Array or StrDataset 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: *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 one or multiple PlaneSegmentation objects to this ImageSegmentation

**Parameters**
**plane_segmentations** (list or tuple or dict or *PlaneSegmentation*) – one or multiple PlaneSegmentation objects to add to this ImageSegmentation

**create_plane_segmentation**(*description*, *imaging_plane*, *name=None*, *reference_images=None*, *id=None*, *columns=None*, *colnames=None*)

Create a PlaneSegmentation object 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the ordered names of the columns in this table. columns must also be provided.

**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'**

**property plane_segmentations**

a dictionary containing the PlaneSegmentation in this ImageSegmentation

**class** pynwb.ophys.**RoiResponseSeries**(*name*, *data*, *unit*, *rois*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*, *offset=0.0*)

Bases: *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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. May be 1D or 2D. The first dimension must be time. The optional second dimension represents ROIs
- **unit** (str) – The base unit of measurement (should be SI unit)
- **rois** (DynamicTableRegion) – a table region corresponding to the ROIs that were used to generate this data
- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.

**property 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: *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 one or multiple RoiResponseSeries objects to this DfOverF

**Parameters**

**roi_response_series** (list or tuple or dict or *RoiResponseSeries*) – one or multiple RoiResponseSeries objects to add to this DfOverF

**create_roi_response_series**(*name*, *data*, *unit*, *rois*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*, *offset=0.0*)

Create a RoiResponseSeries object and add it to this DfOverF

**Parameters**

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. May be 1D or 2D. The first dimension must be time. The optional second dimension represents ROIs
- **unit** (str) – The base unit of measurement (should be SI unit)
- **rois** (DynamicTableRegion) – a table region corresponding to the ROIs that were used to generate this data
- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.

**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'**

**property roi_response_series**
a dictionary containing the RoiResponseSeries in this DfOverF

**class** pynwb.ophys.**Fluorescence**(*roi_response_series={}*, *name='Fluorescence'*)
Bases: *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 one or multiple RoiResponseSeries objects to this Fluorescence

**Parameters**
**roi_response_series** (list or tuple or dict or *RoiResponseSeries*) – one or multiple RoiResponseSeries objects to add to this Fluorescence

**create_roi_response_series**(*name*, *data*, *unit*, *rois*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*, *offset=0.0*)
Create a RoiResponseSeries object and add it to this Fluorescence

**Parameters**
- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. May be 1D or 2D. The first dimension must be time. The optional second dimension represents ROIs
- **unit** (str) – The base unit of measurement (should be SI unit)

- **rois** (DynamicTableRegion) – a table region corresponding to the ROIs that were used to generate this data
- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.

**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'**

**property roi_response_series**

a dictionary containing the RoiResponseSeries in this Fluorescence

## 6.5 pynwb.ogen module

class pynwb.ogen.**OptogeneticStimulusSite**(*name*, *device*, *description*, *excitation_lambda*, *location*)

Bases: *NWBContainer*

Optogenetic stimulus site.

**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.

**property description**

Description of site.

**property device**

The device that was used.

**property excitation_lambda**

Excitation wavelength in nm.

**property 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*, *offset=0.0*)

Bases: *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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values over time.
- **site** (*OptogeneticStimulusSite*) – The site to which this stimulus was applied.
- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.

**property site**

The site to which this stimulus was applied.

**namespace = 'core'**

**neurodata_type = 'OptogeneticSeries'**

## 6.6 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*, *offset=0.0*, *device=None*)

Bases: *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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. Can be 3D or 4D. The first dimension must be time (frame). The second and third dimensions represent x and y. The optional fourth dimension represents z. Either data or external_file must be specified (not None), but not both. If data is not specified, data will be set to an empty 3D array.
- **unit** (str) – The unit of measurement of the image data, e.g., values between 0 and 255. Required when data is specified. If unit (and data) are not specified, then unit will be set to "unknown".
- **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 Array or StrDataset 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 (not None), but not both.
- **starting_frame** (Iterable) – Each entry is a frame number that corresponds to the first frame of each file listed in external_file within the full ImageSeries.
- **bits_per_pixel** (int) – DEPRECATED: Number of bits per image pixel

- **dimension** (Iterable) – Number of pixels on x, y, (and z) axes.
- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.
- **device** (*Device*) – Device used to capture the images/video.

**DEFAULT_DATA = array([], shape=(0, 0, 0), dtype=uint8)**

**property bits_per_pixel**

**property device**

Device used to capture the images/video.

**property dimension**

Number of pixels on x, y, (and z) axes.

**property 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 (not None), but not both.

**property format**

Format of image. Three types - 1) Image format; tiff, png, jpg, etc. 2) external 3) raw.

**namespace = 'core'**

**neurodata_type = 'ImageSeries'**

**property starting_frame**

Each entry is a frame number that corresponds to the first frame of each file listed in external_file within the full ImageSeries.

class pynwb.image.**IndexSeries**(*name*, *data*, *unit*, *indexed_timeseries=None*, *indexed_images=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*, *offset=0.0*)

Bases: *TimeSeries*

Stores indices to image frames stored in an ImageSeries. The purpose of the IndexSeries 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. Must be 1D, where the first dimension must be time (frame)
- **unit** (str) – The base unit of measurement (should be SI unit)
- **indexed_timeseries** (*TimeSeries*) – Link to TimeSeries containing images that are indexed.
- **indexed_images** (*Images*) – Link to Images object containing an ordered set of images that are indexed. The Images object must contain a 'ordered_images' dataset specifying the order of the images in the Images type.
- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.

**property indexed_timeseries**

Link to TimeSeries containing images that are indexed.

**namespace = 'core'**

**neurodata_type = 'IndexSeries'**

**class** pynwb.image.**ImageMaskSeries**(*name*, *masked_imageseries*, *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*, *offset=0.0*, *device=None*)

Bases: *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
- **masked_imageseries** (*ImageSeries*) – Link to ImageSeries that mask is applied to.
- **data** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. Can be 3D or 4D. The first dimension must be time (frame). The second and third dimensions represent x and y. The optional fourth dimension represents z. Either data or external_file must be specified (not None), but not both. If data is not specified, data will be set to an empty 3D array.
- **unit** (str) – The unit of measurement of the image data, e.g., values between 0 and 255. Required when data is specified. If unit (and data) are not specified, then unit will be set to "unknown".
- **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 Array or StrDataset 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 (not None), but not both.
- **starting_frame** (Iterable) – Each entry is a frame number that corresponds to the first frame of each file listed in external_file within the full ImageSeries.
- **bits_per_pixel** (int) – DEPRECATED: Number of bits per image pixel
- **dimension** (Iterable) – Number of pixels on x, y, (and z) axes.
- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.
- **device** (*Device*) – Device used to capture the mask data. This field will likely not be needed. The device used to capture the masked ImageSeries data should be stored in the ImageSeries.

**property 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*, *device=None*, *offset=0.0*)

Bases: *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 Array or StrDataset 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – Images presented to subject, either grayscale or RGB. May be 3D or 4D. The first dimension must be time (frame). The second and third dimensions represent x and y. The optional fourth dimension must be length 3 and represents the RGB value for color images. Either data or external_file must be specified, but not both.
- **unit** (str) – The unit of measurement of the image data, e.g., values between 0 and 255. Required when data is specified. If unit (and data) are not specified, then unit will be set to "unknown".
- **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 Array or StrDataset 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 (not None), but not both.
- **starting_frame** (Iterable) – Each entry is a frame number that corresponds to the first frame of each file listed in external_file within the full ImageSeries.
- **bits_per_pixel** (int) – DEPRECATED: Number of bits per image pixel
- **dimension** (Iterable) – Number of pixels on x, y, (and z) axes.
- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit

- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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
- **device** (*Device*) – Device used to capture the images/video.
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.

**property distance**

Distance from camera/monitor to target/eye.

**property field_of_view**

Width, height and depth of image, or imaged area (meters).

**property 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: *Image*

**Parameters**

- **name** (str) – The name of this image
- **data** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – Data of grayscale image. Must be 2D where the dimensions represent x and y.
- **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: *Image*

**Parameters**

- **name** (str) – The name of this image
- **data** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – Data of color image. Must be 3D where the first and second dimensions represent x and y. The third dimension has length 3 and represents the RGB value.

- **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: *Image*

**Parameters**

- **name** (str) – The name of this image
- **data** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – Data of color image with transparency. Must be 3D where the first and second dimensions represent x and y. The third dimension has length 4 and represents the RGBA value.
- **resolution** (float) – pixels / cm
- **description** (str) – description of image

**namespace = 'core'**

**neurodata_type = 'RGBAImage'**

## 6.7 pynwb.behavior module

class pynwb.behavior.**SpatialSeries**(*name*, *data*, *reference_frame*, *unit='meters'*, *conversion=1.0*, *resolution=-1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*, *offset=0.0*)

Bases: *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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. Can be 1D or 2D. The first dimension must be time. If 2D, there can be 1, 2, or 3 columns, which represent x, y, and z.
- **reference_frame** (str) – description defining what the zero-position is
- **unit** (str) – The base unit of measurement (should be SI unit)
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit

- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.

**property reference_frame**

description defining what the zero-position is

**namespace = 'core'**

**neurodata_type = 'SpatialSeries'**

**class** pynwb.behavior.**BehavioralEpochs**(*interval_series={}*, *name='BehavioralEpochs'*)

Bases: *MultiContainerInterface*

TimeSeries for storing behavioral 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 of 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 one or multiple IntervalSeries objects to this BehavioralEpochs

**Parameters**

**interval_series** (list or tuple or dict or *IntervalSeries*) – one or multiple IntervalSeries objects to add to this BehavioralEpochs

**create_interval_series**(*name*, *data=[]*, *timestamps=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*)

Create an IntervalSeries object and add it to this BehavioralEpochs

**Parameters**

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. Must be 1D, where the first dimension must be time. Values are >0 if interval started, <0 if interval ended.
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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*

**property interval_series**

a dictionary containing the IntervalSeries in this BehavioralEpochs

**namespace = 'core'**

**neurodata_type = 'BehavioralEpochs'**

**class** pynwb.behavior.**BehavioralEvents**(*time_series={}*, *name='BehavioralEvents'*)

Bases: *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 one or multiple TimeSeries objects to this BehavioralEvents

**Parameters**
**time_series** (list or tuple or dict or *TimeSeries*) – one or multiple TimeSeries objects to add to this BehavioralEvents

**create_timeseries**(*name*, *data*, *unit*, *resolution=-1.0*, *conversion=1.0*, *offset=0.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*, *continuity=None*)

Create a TimeSeries object and add it to this BehavioralEvents

**Parameters**

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – The data values. The first dimension must be time. Can also store binary data, e.g., image frames
- **unit** (str) – The base unit of measurement (should be SI unit)
- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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

- **continuity** (str) – Optionally describe the continuity of the data. Can be "continuous", "instantaneous", or"step". For example, a voltage trace would be "continuous", because samples are recorded from a continuous process. An array of lick times would be "instantaneous", because the data represents distinct moments in time. Times of image presentations would be "step" because the picture remains the same until the next time-point. This field is optional, but is useful in providing information about the underlying data. It may inform the way this data is interpreted, the way it is visualized, and what analysis methods are applicable.

**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'**

**property time_series**

a dictionary containing the TimeSeries in this BehavioralEvents

class pynwb.behavior.**BehavioralTimeSeries**(*time_series={}*, *name='BehavioralTimeSeries'*)

Bases: *MultiContainerInterface*

TimeSeries for storing Behavioral 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 one or multiple TimeSeries objects to this BehavioralTimeSeries

**Parameters**
**time_series** (list or tuple or dict or *TimeSeries*) – one or multiple TimeSeries objects to add to this BehavioralTimeSeries

**create_timeseries**(*name*, *data*, *unit*, *resolution=-1.0*, *conversion=1.0*, *offset=0.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*, *continuity=None*)

Create a TimeSeries object and add it to this BehavioralTimeSeries

**Parameters**

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – The data values. The first dimension must be time. Can also store binary data, e.g., image frames
- **unit** (str) – The base unit of measurement (should be SI unit)
- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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
- **continuity** (str) – Optionally describe the continuity of the data. Can be "continuous", "instantaneous", or"step". For example, a voltage trace would be "continuous", because samples are recorded from a continuous process. An array of lick times would be "instantaneous", because the data represents distinct moments in time. Times of image presentations would be "step" because the picture remains the same until the next time-point. This field is optional, but is useful in providing information about the underlying data. It may inform the way this data is interpreted, the way it is visualized, and what analysis methods are applicable.

**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'**

**property time_series**

a dictionary containing the TimeSeries in this BehavioralTimeSeries

**class** pynwb.behavior.**PupilTracking**(*time_series={}*, *name='PupilTracking'*)

Bases: *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 one or multiple TimeSeries objects to this PupilTracking

**Parameters**
**time_series** (list or tuple or dict or *TimeSeries*) – one or multiple TimeSeries objects to add to this PupilTracking

**create_timeseries**(*name*, *data*, *unit*, *resolution=-1.0*, *conversion=1.0*, *offset=0.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*, *continuity=None*)

Create a TimeSeries object and add it to this PupilTracking

**Parameters**

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – The data values. The first dimension must be time. Can also store binary data, e.g., image frames
- **unit** (str) – The base unit of measurement (should be SI unit)

- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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
- **continuity** (str) – Optionally describe the continuity of the data. Can be "continuous", "instantaneous", or"step". For example, a voltage trace would be "continuous", because samples are recorded from a continuous process. An array of lick times would be "instantaneous", because the data represents distinct moments in time. Times of image presentations would be "step" because the picture remains the same until the next time-point. This field is optional, but is useful in providing information about the underlying data. It may inform the way this data is interpreted, the way it is visualized, and what analysis methods are applicable.

**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'**

**property time_series**

a dictionary containing the TimeSeries in this PupilTracking

**class** pynwb.behavior.**EyeTracking**(*spatial_series={}*, *name='EyeTracking'*)

Bases: *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 one or multiple SpatialSeries objects to this EyeTracking

**Parameters**
**spatial_series** (list or tuple or dict or *SpatialSeries*) – one or multiple SpatialSeries objects to add to this EyeTracking

**create_spatial_series**(*name*, *data*, *reference_frame*, *unit='meters'*, *conversion=1.0*, *resolution=-1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*, *offset=0.0*)

Create a SpatialSeries object and add it to this EyeTracking

**Parameters**

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. Can be 1D or 2D. The first dimension must be time. If 2D, there can be 1, 2, or 3 columns, which represent x, y, and z.
- **reference_frame** (str) – description defining what the zero-position is
- **unit** (str) – The base unit of measurement (should be SI unit)
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.

**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'**

**property spatial_series**

a dictionary containing the SpatialSeries in this EyeTracking

**class** pynwb.behavior.**CompassDirection**(*spatial_series={}*, *name='CompassDirection'*)

Bases: *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 one or multiple SpatialSeries objects to this CompassDirection

**Parameters**
**spatial_series** (list or tuple or dict or *SpatialSeries*) – one or multiple SpatialSeries objects to add to this CompassDirection

**create_spatial_series**(*name*, *data*, *reference_frame*, *unit='meters'*, *conversion=1.0*, *resolution=-1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*, *offset=0.0*)

Create a SpatialSeries object and add it to this CompassDirection

**Parameters**

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. Can be 1D or 2D. The first dimension must be time. If 2D, there can be 1, 2, or 3 columns, which represent x, y, and z.
- **reference_frame** (str) – description defining what the zero-position is
- **unit** (str) – The base unit of measurement (should be SI unit)
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.

**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'**

**property spatial_series**

a dictionary containing the SpatialSeries in this CompassDirection

class pynwb.behavior.**Position**(*spatial_series={}*, *name='Position'*)

Bases: *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 one or multiple SpatialSeries objects to this Position

**Parameters**
**spatial_series** (list or tuple or dict or *SpatialSeries*) – one or multiple SpatialSeries objects to add to this Position

**create_spatial_series**(*name*, *data*, *reference_frame*, *unit='meters'*, *conversion=1.0*, *resolution=-1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*, *offset=0.0*)

Create a SpatialSeries object and add it to this Position

**Parameters**

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. Can be 1D or 2D. The first dimension must be time. If 2D, there can be 1, 2, or 3 columns, which represent x, y, and z.
- **reference_frame** (str) – description defining what the zero-position is
- **unit** (str) – The base unit of measurement (should be SI unit)
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data

- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.

**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'**

**property spatial_series**

a dictionary containing the SpatialSeries in this Position

## 6.8 pynwb.base module

**class** pynwb.base.**ProcessingModule**(*name*, *description*, *data_interfaces=None*)

Bases: *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) – NWBDataInterfaces that belong to this ProcessingModule

property description

Description of this processing module

property data_interfaces

a dictionary containing the NWBDataInterface or DynamicTable in this ProcessingModule

property containers

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

__getitem__(*name=None*)

Get a NWBDataInterface from this ProcessingModule

**Parameters**
**name** (str) – the name of the NWBDataInterface or DynamicTable

**Returns**
the NWBDataInterface or DynamicTable with the given name

**Return type**
*NWBDataInterface* or DynamicTable

add(*data_interfaces*)

Add one or multiple NWBDataInterface or DynamicTable objects to this ProcessingModule

**Parameters**
**data_interfaces** (list or tuple or dict or *NWBDataInterface* or DynamicTable) – one or multiple NWBDataInterface or DynamicTable objects to add to this ProcessingModule

get(*name=None*)

Get a NWBDataInterface from this ProcessingModule

**Parameters**
**name** (str) – the name of the NWBDataInterface or DynamicTable

**Returns**
the NWBDataInterface or DynamicTable with the given name

**Return type**
*NWBDataInterface* or DynamicTable

**namespace = 'core'**

**neurodata_type = 'ProcessingModule'**

class pynwb.base.**TimeSeries**(*name*, *data*, *unit*, *resolution=-1.0*, *conversion=1.0*, *offset=0.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*, *continuity=None*)

Bases: *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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or TimeSeries) – The data values. The first dimension must be time. Can also store binary data, e.g., image frames
- **unit** (str) – The base unit of measurement (should be SI unit)
- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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
- **continuity** (str) – Optionally describe the continuity of the data. Can be "continuous", "instantaneous", or"step". For example, a voltage trace would be "continuous", because samples are recorded from a continuous process. An array of lick times would be "instantaneous", because the data represents distinct moments in time. Times of image presentations would be "step" because the picture remains the same until the next time-point. This field is optional, but is useful in providing information about the underlying data. It may inform the way this data is interpreted, the way it is visualized, and what analysis methods are applicable.

**DEFAULT_DATA = array([], dtype=uint8)**

**DEFAULT_UNIT = 'unknown'**

**DEFAULT_RESOLUTION = -1.0**

**DEFAULT_CONVERSION = 1.0**

**DEFAULT_OFFSET = 0.0**

**property timestamps_unit**

**property interval**

**property starting_time**

The timestamp of the first sample

**property starting_time_unit**

**property num_samples**

Tries to return the number of data samples. If this cannot be assessed, returns None.

**property data**

**property data_link**

**property timestamps**

**property timestamp_link**

**property time_unit**

**get_timestamps()**

Get the timestamps of this TimeSeries. If timestamps are not stored in this TimeSeries, generate timestamps.

**get_data_in_units()**

Get the data of this TimeSeries in the specified unit of measurement, applying the conversion factor and offset:

$$out = data * conversion + offset$$

If the field 'channel_conversion' is present, the conversion factor for each channel is additionally applied to each channel:

$$out_{channel} = data * conversion * conversion_{channel} + offset$$

NOTE: This will read the entire dataset into memory.

**Return type**
numpy.ndarray

**property comments**

Human-readable comments about this TimeSeries dataset

**property continuity**

Optionally describe the continuity of the data. Can be "continuous", "instantaneous", or"step". For example, a voltage trace would be "continuous", because samples are recorded from a continuous process. An array of lick times would be "instantaneous", because the data represents distinct moments in time. Times of image presentations would be "step" because the picture remains the same until the next time-point. This field is optional, but is useful in providing information about the underlying data. It may inform the way this data is interpreted, the way it is visualized, and what analysis methods are applicable.

**property control**

Numerical labels that apply to each element in data

**property control_description**

Description of each control value

**property conversion**

Scalar to multiply each element in data to convert it to the specified unit

**property description**

Description of this TimeSeries dataset

**namespace = 'core'**

**neurodata_type = 'TimeSeries'**

**property offset**

Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.

**property rate**

Sampling rate in Hz

**property resolution**

The smallest meaningful difference (in specified unit) between values in data

**property unit**

The base unit of measurement (should be SI unit)

**class** pynwb.base.**Image**(*name*, *data*, *resolution=None*, *description=None*)

Bases: *NWBData*

Abstract image class. It is recommended to instead use pynwb.image.GrayscaleImage or pynwb.image.RGPImage where appropriate.

**Parameters**

- **name** (str) – The name of this image
- **data** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – data of image. Dimensions: x, y [, r,g,b[,a]]
- **resolution** (float) – pixels / cm
- **description** (str) – description of image

**namespace = 'core'**

**neurodata_type = 'Image'**

**class** pynwb.base.**ImageReferences**(*name*, *data*)

Bases: *NWBData*

Ordered dataset of references to Image objects.

**Parameters**

- **name** (str) – The name of this ImageReferences object.
- **data** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – The images in order.

**namespace = 'core'**

**neurodata_type = 'ImageReferences'**

**class** pynwb.base.**Images**(*name*, *images=None*, *description='no description'*, *order_of_images=None*)

Bases: *MultiContainerInterface*

An collection of images with an optional way to specify the order of the images using the "order_of_images" dataset. An order must be specified if the images are referenced by index, e.g., from an IndexSeries.

**Parameters**

- **name** (str) – The name of this set of images
- **images** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – image objects
- **description** (str) – description of images
- **order_of_images** (*ImageReferences*) – Ordered dataset of references to Image objects stored in the parent group.

**__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 one or multiple Image objects to this Images

**Parameters**
**images** (list or tuple or dict or *Image*) – one or multiple Image objects to add to this Images

**create_image**(*name*, *data*, *resolution=None*, *description=None*)

Create an Image object and add it to this Images

**Parameters**

- **name** (str) – The name of this image
- **data** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – data of image. Dimensions: x, y [, r,g,b[,a]]
- **resolution** (float) – pixels / cm
- **description** (str) – description of image

**Returns**
the Image object that was created

**Return type**
*Image*

**property description**

description of images

**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*

**property images**

a dictionary containing the Image in this Images

**namespace = 'core'**

**neurodata_type = 'Images'**

**property order_of_images**

Ordered dataset of references to Image objects stored in the parent group.

**class** pynwb.base.**TimeSeriesReference**(*idx_start: int*, *count: int*, *timeseries:* TimeSeries)

Bases: NamedTuple

Class used to represent data values of a *TimeSeriesReferenceVectorData* This is a typing.NamedTuple type with predefined tuple components *idx_start*, *count*, and *timeseries*.

**Variables**

- **idx_start**
- **count**
- **timeseries**

Create new instance of TimeSeriesReference(idx_start, count, timeseries)

**idx_start: int**

Start index in time for the timeseries

**count: int**

Number of timesteps to be selected starting from *idx_start*

**timeseries: *TimeSeries***

The *TimeSeries* object the TimeSeriesReference applies to

**check_types**()

Helper function to check correct types for *idx_start*, *count*, and *timeseries*.

This function is usually used in the try/except block to check for *TypeError* raised by the function.

See also *isvalid* to check both validity of types and the reference itself.

**Returns**
True if successful. If unsuccessful *TypeError* is raised.

**Raises**
**TypeError** – If one of the fields does not match the expected type

**isvalid**()

Check whether the reference is valid. Setting *idx_start* and *count* to -1 is used to indicate invalid references. This is useful to allow for missing data in *TimeSeriesReferenceVectorData*

**Returns**
True if the selection is valid. Returns False if both *idx_start* and *count* are negative. Raises *IndexError* in case the indices are bad.

**Raises**

- **IndexError** – If the combination of *idx_start* and *count* are not valid for the given timeseries.
- **TypeError** – If one of the fields does not match the expected type

**property timestamps**

Get the floating point timestamp offsets in seconds from the timeseries that correspond to the array. These are either loaded directly from the *timeseries* timestamps or calculated from the starting time and sampling rate.

**Raises**

- **IndexError** – If the combination of *idx_start* and *count* are not valid for the given timeseries.
- **TypeError** – If one of the fields does not match the expected type

**Returns**
Array with the timestamps.

**property data**

Get the selected data values. This is a convenience function to slice data from the *timeseries* based on the given *idx_start* and *count*

**Raises**

- **IndexError** – If the combination of *idx_start* and *count* are not valid for the given timeseries.
- **TypeError** – If one of the fields does not match the expected type

**Returns**
Result of `self.timeseries.data[self.idx_start: (self.idx_start + self.count)]`. Returns None in case the reference is invalid (i.e., if both *idx_start* and *count* are negative.

**classmethod empty**(*timeseries*)

Creates an empty TimeSeriesReference object to represent missing data.

When missing data needs to be represented, NWB defines `None` for the complex data type `(idx_start, count, TimeSeries)` as (-1, -1, TimeSeries) for storage. The exact timeseries object will technically not matter since the empty reference is a way of indicating a NaN value in a *TimeSeriesReferenceVectorData* column.

An example where this functionality is used is *IntracellularRecordingsTable* where only one of stimulus or response data was recorded. In such cases, the timeseries object for the empty stimulus *TimeSeriesReference* could be set to the response series, or vice versa.

**returns**
Returns *TimeSeriesReference*

**Parameters**
**timeseries** (*TimeSeries*) – the timeseries object to reference.

class pynwb.base.**TimeSeriesReferenceVectorData**(*name='timeseries'*, *description='Column storing references to a TimeSeries (rows). For each TimeSeries this VectorData column stores the start_index and count to indicate the range in time to be selected as well as an object reference to the TimeSeries.'*, *data=[]*)

Bases: VectorData

Column storing references to a TimeSeries (rows). For each TimeSeries this VectorData column stores the start_index and count to indicate the range in time to be selected as well as an object reference to the TimeSeries.

**Representing missing values** In practice we sometimes need to be able to represent missing values, e.g., in the *IntracellularRecordingsTable* we have *TimeSeriesReferenceVectorData* to link to stimulus and response recordings, but a user can specify either only one of them or both. Since there is no `None` value for a complex types like `(idx_start, count, TimeSeries)`, NWB defines `None` as `(-1, -1, TimeSeries)` for storage, i.e., if the `idx_start` (and `count`) is negative then this indicates an invalid link (in practice both `idx_start` and `count` must always either both be positive or both be negative). When selecting data via the *get* or *__getitem__* functions, `(-1, -1, TimeSeries)` values are replaced by the corresponding *TIME_SERIES_REFERENCE_NONE_TYPE* tuple to avoid exposing NWB storage internals to the user and simplifying the use of and checking for missing values. **NOTE:** We can still inspect the raw data values by looking at `self.data` directly instead.

**Variables**
- **TIME_SERIES_REFERENCE_TUPLE**
- **TIME_SERIES_REFERENCE_NONE_TYPE**

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

**TIME_SERIES_REFERENCE_TUPLE**

Return type when calling *get* or *__getitem__*

alias of *TimeSeriesReference*

**TIME_SERIES_REFERENCE_NONE_TYPE = (None, None, None)**

Tuple used to represent None values when calling *get* or *__getitem__*. See also *TIME_SERIES_REFERENCE_TUPLE*

**namespace = 'core'**

**neurodata_type = 'TimeSeriesReferenceVectorData'**

**add_row**(*val*)

Append a data value to this column.

**Parameters**
**val** (*TimeSeriesReference* or tuple) – the value to add to this column. If this is a regular tuple then it must be convertible to a TimeSeriesReference

**append**(*arg*)

Append a data value to this column.

**Parameters**

**arg** (*TimeSeriesReference* or tuple) – the value to append to this column. If this is a regular tuple then it must be convertible to a TimeSeriesReference

**get**(*key*, ***kwargs*)

Retrieve elements from this object.

The function uses *TIME_SERIES_REFERENCE_TUPLE* to describe individual records in the dataset. This allows the code to avoid exposing internal details of the schema to the user and simplifies handling of missing values by explicitly representing missing values via *TIME_SERIES_REFERENCE_NONE_TYPE* rather than the internal representation used for storage of `(-1, -1, TimeSeries)`.

**Parameters**

- **key** – Selection of the elements
- **kwargs** – Ignored

**Returns**

*TIME_SERIES_REFERENCE_TUPLE* if a single element is being selected. Otherwise return a list of *TIME_SERIES_REFERENCE_TUPLE* objects. Missing values are represented by *TIME_SERIES_REFERENCE_NONE_TYPE* in which all values (i.e., idx_start, count, timeseries) are set to None.

## 6.9 pynwb.misc module

**class** pynwb.misc.**AnnotationSeries**(*name*, *data=[]*, *timestamps=None*, *comments='no comments'*, *description='no description'*)

Bases: *TimeSeries*

Stores text-based records about the experiment. To use the AnnotationSeries, add records individually through add_annotation(). Alternatively, if all annotations are already stored in a list or numpy array, set the data and timestamps in the constructor.

**Parameters**

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The annotations over time. Must be 1D.
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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*, *offset=0.0*)

Bases: *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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. May be 1D or 2D. The first dimension must be time. The optional second dimension represents features
- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.

property features

Description of each feature

property 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: *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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. Must be 1D, where the first dimension must be time. Values are >0 if interval started, <0 if interval ended.
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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 start time of the interval
- **stop** (float) – The stop time of the interval

**property data**

**property timestamps**

**namespace = 'core'**

**neurodata_type = 'IntervalSeries'**

**class** pynwb.misc.**Units**(*name='Units'*, *id=None*, *columns=None*, *colnames=None*, *description=None*, *electrode_table=None*, *waveform_rate=None*, *waveform_unit='volts'*, *resolution=None*)

Bases: 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the ordered names of the columns in this table. columns must also be provided.
- **description** (str) – a description of what is in this table
- **electrode_table** (DynamicTable) – the table that the *electrodes* column indexes
- **waveform_rate** (float) – Sampling rate of the waveform means
- **waveform_unit** (str) – Unit of measurement of the waveform means
- **resolution** (float) – The smallest possible difference between two spike times

**waveforms_desc = 'Individual waveforms for each spike. If the dataset is three-dimensional, the third dimension shows the response from different electrodes that all observe this unit simultaneously. In this case, the `electrodes` column of this Units table should be used to indicate which electrodes are associated with this unit, and the electrodes dimension here should be in the same order as the electrodes referenced in the `electrodes` column of this table.'**

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

Add a unit to this table

**Parameters**

- **spike_times** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the spike times for each unit
- **obs_intervals** (ndarray or list or tuple or Dataset or Array or StrDataset 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 Array or StrDataset 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 Array or StrDataset 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the spike waveform standard deviation for each unit. Shape is (time,) or (time, electrodes)

- **waveforms** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – Individual waveforms for each spike. If the dataset is three-dimensional, the third dimension shows the response from different electrodes that all observe this unit simultaneously. In this case, the *electrodes* column of this Units table should be used to indicate which electrodes are associated with this unit, and the electrodes dimension here should be in the same order as the electrodes referenced in the *electrodes* column of this table.
- **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'**

**property resolution**

The smallest possible difference between two spike times

**property waveform_rate**

Sampling rate of the waveform means

**property waveform_unit**

Unit of measurement of the waveform means

**class** pynwb.misc.**DecompositionSeries**(*name*, *data*, *metric*, *description='no description'*, *unit='no unit'*, *bands=None*, *source_timeseries=None*, *source_channels=None*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=None*, *comments='no comments'*, *control=None*, *control_description=None*, *offset=0.0*)

Bases: *TimeSeries*

Stores product of spectral analysis

**Parameters**

- **name** (str) – The name of this TimeSeries dataset
- **data** (ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or *TimeSeries*) – The data values. Must be 3D, where the first dimension must be time, the second dimension must be channels, and the third dimension must be 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
- **source_channels** (DynamicTableRegion) – The channels that provided the source data. In the case of electrical recordings this is typically a DynamicTableRegion pointing to the electrodes table at NWBFile.electrodes, similar to ElectricalSeries.electrodes.
- **resolution** (float) – The smallest meaningful difference (in specified unit) between values in data
- **conversion** (float) – Scalar to multiply each element in data to convert it to the specified unit
- **timestamps** (ndarray or list or tuple or Dataset or Array or StrDataset 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
- **offset** (float) – Scalar to add to each element in the data scaled by 'conversion' to finish converting it to the specified unit.

**DEFAULT_DATA = array([], shape=(0, 0, 0), dtype=uint8)**

**property source_timeseries**

the input TimeSeries from this analysis

**property source_channels**

the channels that provided the source data

**property metric**

metric of analysis. recommended - 'phase', 'amplitude', 'power'

**property 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – low and high frequencies of bandpass 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'**

## 6.10 pynwb.epoch module

**class** pynwb.epoch.**TimeIntervals**(*name*, *description='experimental intervals'*, *id=None*, *columns=None*, *colnames=None*)

Bases: 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO 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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator) – the ordered names of the columns in this table. columns must also be provided.

**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'**

## 6.11 pynwb package

### 6.11.1 Subpackages

#### pynwb.io package

#### Submodules

#### pynwb.io.base module

**class** pynwb.io.base.**ModuleMap**(*spec*)

Bases: *NWBContainerMapper*

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.base.**TimeSeriesMap**(*spec*)

Bases: *NWBContainerMapper*

Create a map from AbstractContainer attributes to specifications

**Parameters**
**spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

**timestamps_attr**(*container*, *manager*)

**timestamps_carg**(*builder*, *manager*)

**data_attr**(*container*, *manager*)

**data_carg**(*builder*, *manager*)

**unit_carg**(*builder*, *manager*)

constructor_args = {'data': <function TimeSeriesMap.data_carg>, 'name': <function ObjectMapper.get_container_name>, 'timestamps': <function TimeSeriesMap.timestamps_carg>, 'unit': <function TimeSeriesMap.unit_carg>}

obj_attrs = {'data': <function TimeSeriesMap.data_attr>, 'timestamps': <function TimeSeriesMap.timestamps_attr>}

### pynwb.io.behavior module

### pynwb.io.core module

class pynwb.io.core.**NWBBaseTypeMapper**(*spec*)

Bases: 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: *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: *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.**ScratchDataMap**(*spec*)

Bases: *NWBContainerMapper*

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.**NWBTableRegionMap**(*spec*)

Bases: *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 = {'name': <function ObjectMapper.get_container_name>, 'region': <function NWBTableRegionMap.carg_region>, 'table': <function NWBTableRegionMap.carg_table>}**

**obj_attrs = {}**

class pynwb.io.core.**VectorDataMap**(*spec*)

Bases: ObjectMapper

Create a map from AbstractContainer attributes to specifications

**Parameters**
**spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

**get_attr_value**(*spec*, *container*, *manager*)

Get the value of the attribute corresponding to this spec from the given container

**Parameters**

- **spec** (AttributeSpec) – the spec to get the attribute value for
- **container** (VectorData) – the container to get the attribute value from
- **manager** (BuildManager) – the BuildManager used for managing this build

**constructor_args = {'name': <function ObjectMapper.get_container_name>}**

**obj_attrs = {}**

### pynwb.io.ecephys module

### pynwb.io.epoch module

class pynwb.io.epoch.**TimeIntervalsMap**(*spec*)

Bases: DynamicTableMap

Create a map from AbstractContainer attributes to specifications

**Parameters**
**spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

**columns_carg**(*builder*, *manager*)

**constructor_args = {'columns': <function TimeIntervalsMap.columns_carg>, 'name': <function ObjectMapper.get_container_name>}**

**obj_attrs = {'colnames': <function DynamicTableMap.attr_columns>}**

### pynwb.io.file module

pynwb.io.file.**parse_datetime**(*datestr*)

Parse an ISO 8601 date string into a datetime object or a date object.

If the date string does not contain a time component, then parse into a date object.

**Parameters**
**datestr** – str

**Returns**
datetime.datetime or datetime.date

class pynwb.io.file.**NWBFileMap**(*spec*)

Bases: ObjectMapper

Create a map from AbstractContainer attributes to specifications

**Parameters**
**spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

**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_create_date': <function NWBFileMap.dateconversion_list>, 'file_name': <function NWBFileMap.name>, 'name': <function ObjectMapper.get_container_name>, 'related_publications': <function NWBFileMap.publications_carg>, 'scratch': <function NWBFileMap.scratch>, 'session_start_time': <function NWBFileMap.dateconversion>, 'timestamps_reference_time': <function NWBFileMap.dateconversion_trt>}**

**obj_attrs = {'experimenter': <function NWBFileMap.experimenter_obj_attr>, 'related_publications': <function NWBFileMap.publication_obj_attr>, 'scratch_containers': <function NWBFileMap.scratch_containers>, 'scratch_datas': <function NWBFileMap.scratch_datas>}**

class pynwb.io.file.**SubjectMap**(*spec*)

Bases: ObjectMapper

Create a map from AbstractContainer attributes to specifications

**Parameters**
**spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

**dateconversion**(*builder*, *manager*)

**age_reference_none**(*builder*, *manager*)

**constructor_args = {'age__reference': <function SubjectMap.age_reference_none>, 'date_of_birth': <function SubjectMap.dateconversion>, 'name': <function ObjectMapper.get_container_name>}**

**obj_attrs = {}**

### pynwb.io.icephys module

class pynwb.io.icephys.**VoltageClampSeriesMap**(*spec*)

Bases: *TimeSeriesMap*

Create a map from AbstractContainer attributes to specifications

**Parameters**
**spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

**constructor_args = {'data': <function TimeSeriesMap.data_carg>, 'name': <function ObjectMapper.get_container_name>, 'timestamps': <function TimeSeriesMap.timestamps_carg>, 'unit': <function TimeSeriesMap.unit_carg>}**

**obj_attrs = {'data': <function TimeSeriesMap.data_attr>, 'timestamps': <function TimeSeriesMap.timestamps_attr>}**

class pynwb.io.icephys.**IntracellularRecordingsTableMap**(*spec*)

Bases: AlignedDynamicTableMap

Customize the mapping for AlignedDynamicTable

Create a map from AbstractContainer attributes to specifications

**Parameters**
**spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

**electrodes**(*container*, *manager*)

**stimuli**(*container*, *manager*)

**responses**(*container*, *manager*)

**constructor_args = {'name': <function ObjectMapper.get_container_name>}**

**obj_attrs = {'colnames': <function DynamicTableMap.attr_columns>, 'electrodes': <function IntracellularRecordingsTableMap.electrodes>, 'responses': <function IntracellularRecordingsTableMap.responses>, 'stimuli': <function IntracellularRecordingsTableMap.stimuli>}**

### pynwb.io.image module

**class** pynwb.io.image.**ImageSeriesMap**(*spec*)

Bases: *TimeSeriesMap*

Create a map from AbstractContainer attributes to specifications

**Parameters**
**spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

**constructor_args = {'data': <function TimeSeriesMap.data_carg>, 'name': <function ObjectMapper.get_container_name>, 'timestamps': <function TimeSeriesMap.timestamps_carg>, 'unit': <function TimeSeriesMap.unit_carg>}**

**obj_attrs = {'data': <function TimeSeriesMap.data_attr>, 'timestamps': <function TimeSeriesMap.timestamps_attr>}**

### pynwb.io.misc module

**class** pynwb.io.misc.**UnitsMap**(*spec*)

Bases: DynamicTableMap

Create a map from AbstractContainer attributes to specifications

**Parameters**
**spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

**resolution_carg**(*builder*, *manager*)

**waveform_rate_carg**(*builder*, *manager*)

**waveform_unit_carg**(*builder*, *manager*)

**electrodes_column**(*container*, *manager*)

**constructor_args = {'name': <function ObjectMapper.get_container_name>, 'resolution': <function UnitsMap.resolution_carg>, 'waveform_rate': <function UnitsMap.waveform_rate_carg>, 'waveform_unit': <function UnitsMap.waveform_unit_carg>}**

**obj_attrs = {'colnames': <function DynamicTableMap.attr_columns>, 'electrodes': <function UnitsMap.electrodes_column>}**

### pynwb.io.ogen module

### pynwb.io.ophys module

**class** pynwb.io.ophys.**PlaneSegmentationMap**(*spec*)

Bases: DynamicTableMap

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 = {'colnames': <function DynamicTableMap.attr_columns>}**

**class** pynwb.io.ophys.**ImagingPlaneMap**(*spec*)

Bases: *NWBContainerMapper*

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 = {}**

### pynwb.io.retinotopy module

### pynwb.io.utils module

pynwb.io.utils.**get_nwb_version**(*builder: Builder*, *include_prerelease=False*) → Tuple[int, ...]

Get the version of the NWB file from the root of the given builder, as a tuple.

If the "nwb_version" attribute on the root builder equals "2.5.1", then (2, 5, 1) is returned. If the "nwb_version" attribute on the root builder equals "2.5.1-alpha" and include_prerelease=False, then (2, 5, 1) is returned. If the "nwb_version" attribute on the root builder equals "2.5.1-alpha" and include_prerelease=True, then (2, 5, 1, "alpha") is returned.

If the "nwb_version" attribute == "2.0b" (the only deviation from semantic versioning in the 2.x series), then if include_prerelease=True, (2, 0, 0, "b") is returned; else, (2, 0, 0) is returned.

**Parameters**

- **builder** (Builder) – Any builder within an NWB file.
- **include_prerelease** (*bool*) – Whether to include prerelease information in the returned tuple.

**Returns**
The version of the NWB file, as a tuple.

**Return type**
tuple

**Raises**
**ValueError** – if the 'nwb_version' attribute is missing from the root of the NWB file.

### Module contents

### pynwb.legacy package

### Subpackages

### pynwb.legacy.io package

### Submodules

### pynwb.legacy.io.base module

**class** pynwb.legacy.io.base.**ModuleMap**(*spec*)

Bases: *ObjectMapperLegacy*

Create a map from AbstractContainer attributes to specifications

**Parameters**
**spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

**name**(**args*)

**carg_description**(**args*)

**constructor_args = {'description': <function ModuleMap.carg_description>, 'name': <function ModuleMap.name>, 'source': <function ObjectMapperLegacy.source_gettr>}**

**obj_attrs = {}**

**class** pynwb.legacy.io.base.**TimeSeriesMap**(*spec*)

Bases: *ObjectMapperLegacy*

Create a map from AbstractContainer attributes to specifications

**Parameters**
**spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

**carg_name**(**args*)

**carg_starting_time**(**args*)

**carg_rate**(**args*)

**constructor_args = {'name': <function TimeSeriesMap.carg_name>, 'rate': <function TimeSeriesMap.carg_rate>, 'source': <function ObjectMapperLegacy.source_gettr>, 'starting_time': <function TimeSeriesMap.carg_starting_time>}**

**obj_attrs = {}**

### pynwb.legacy.io.behavior module

class pynwb.legacy.io.behavior.**BehavioralTimeSeriesMap**(*spec*)

Bases: *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': <function ObjectMapperLegacy.source_gettr>, 'time_series': <function BehavioralTimeSeriesMap.carg_time_series>}**

**obj_attrs = {}**

class pynwb.legacy.io.behavior.**PupilTrackingMap**(*spec*)

Bases: *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': <function ObjectMapperLegacy.source_gettr>, 'time_series': <function PupilTrackingMap.carg_time_series>}**

**obj_attrs = {}**

### pynwb.legacy.io.ecephys module

### pynwb.legacy.io.epoch module

class pynwb.legacy.io.epoch.**EpochMap**(*spec*)

Bases: *ObjectMapperLegacy*

Create a map from AbstractContainer attributes to specifications

**Parameters**
**spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

**name**(*builder*)

**constructor_args = {'name': <function EpochMap.name>, 'source': <function ObjectMapperLegacy.source_gettr>}**

**obj_attrs = {}**

class pynwb.legacy.io.epoch.**EpochTimeSeriesMap**(*spec*)

Bases: *ObjectMapperLegacy*

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>, 'source': <function ObjectMapperLegacy.source_gettr>}**

**obj_attrs = {}**

### pynwb.legacy.io.file module

**class** pynwb.legacy.io.file.**NWBFileMap**(*spec*)

Bases: *ObjectMapperLegacy*

Create a map from AbstractContainer attributes to specifications

**Parameters**
**spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

**name**(*builder*)

**constructor_args = {'file_name': <function NWBFileMap.name>, 'name': <function ObjectMapper.get_container_name>, 'source': <function ObjectMapperLegacy.source_gettr>}**

**obj_attrs = {}**

### pynwb.legacy.io.icephys module

**class** pynwb.legacy.io.icephys.**PatchClampSeriesMap**(*spec*)

Bases: *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': <function ObjectMapper.get_container_name>, 'source': <function ObjectMapperLegacy.source_gettr>}**

**obj_attrs = {}**

### pynwb.legacy.io.image module

**class** pynwb.legacy.io.image.**ImageSeriesMap**(*spec*)

Bases: *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 ObjectMapper.get_container_name>, 'source': <function ObjectMapperLegacy.source_gettr>}

obj_attrs = {}

### pynwb.legacy.io.misc module

class pynwb.legacy.io.misc.**AbstractFeatureSeriesMap**(*spec*)

Bases: *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_units>, 'name': <function ObjectMapper.get_container_name>, 'source': <function ObjectMapperLegacy.source_gettr>}

obj_attrs = {}

### pynwb.legacy.io.ogen module

class pynwb.legacy.io.ogen.**OptogeneticSeriesMap**(*spec*)

Bases: *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': <function OptogeneticSeriesMap.carg_site>, 'source': <function ObjectMapperLegacy.source_gettr>}

obj_attrs = {}

### pynwb.legacy.io.ophys module

class pynwb.legacy.io.ophys.**PlaneSegmentationMap**(*spec*)

Bases: *ObjectMapperLegacy*

Create a map from AbstractContainer attributes to specifications

**Parameters**
**spec** (DatasetSpec or GroupSpec) – The specification for mapping objects to builders

**carg_imaging_plane**(**args*)

constructor_args = {'imaging_plane': <function PlaneSegmentationMap.carg_imaging_plane>, 'name': <function ObjectMapper.get_container_name>, 'source': <function ObjectMapperLegacy.source_gettr>}

obj_attrs = {}

class pynwb.legacy.io.ophys.**TwoPhotonSeriesMap**(*spec*)

Bases: *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': <function TwoPhotonSeriesMap.carg_imaging_plane>, 'name': <function ObjectMapper.get_container_name>, 'source': <function ObjectMapperLegacy.source_gettr>, 'unit': <function TwoPhotonSeriesMap.carg_unit>}

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: 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': <function ObjectMapperLegacy.source_gettr>}

obj_attrs = {}

class pynwb.legacy.map.**TypeMapLegacy**(*namespaces=None*, *mapper_cls=None*)

Bases: 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

### Subpackages

### pynwb.testing.mock package

### Submodules

### pynwb.testing.mock.base module

pynwb.testing.mock.base.**mock_TimeSeries**(*name: str | None = None*, *data=None*, *unit: str = 'volts'*, *resolution: float = -1.0*, *conversion: float = 1.0*, *timestamps=None*, *starting_time: float | None = None*, *rate: float | None = None*, *comments: str = 'no comments'*, *description: str = 'no description'*, *control=None*, *control_description=None*, *continuity=None*, *nwbfile:* NWBFile *| None = None*, *offset=0.0*) → *TimeSeries*

### pynwb.testing.mock.behavior module

pynwb.testing.mock.behavior.**mock_SpatialSeries**(*name: str | None = None*, *data=None*, *reference_frame: str = 'lower left is (0, 0)'*, *unit: str = 'meters'*, *conversion=1.0*, *resolution=-1.0*, *timestamps=None*, *starting_time: float | None = None*, *rate: float | None = 10.0*, *comments: str = 'no comments'*, *description: str = 'no description'*, *control=None*, *control_description=None*, *nwbfile:* NWBFile | *None = None*) → *SpatialSeries*

pynwb.testing.mock.behavior.**mock_Position**(*name: str | None = None*, *spatial_series:* SpatialSeries | *None = None*, *nwbfile:* NWBFile | *None = None*) → *Position*

pynwb.testing.mock.behavior.**mock_PupilTracking**(*name: str | None = None*, *time_series:* TimeSeries | *None = None*, *nwbfile:* NWBFile | *None = None*) → *PupilTracking*

pynwb.testing.mock.behavior.**mock_CompassDirection**(*name: str | None = None*, *spatial_series:* SpatialSeries | *None = None*, *nwbfile:* NWBFile | *None = None*) → *CompassDirection*

### pynwb.testing.mock.device module

pynwb.testing.mock.device.**mock_Device**(*name: str | None = None*, *description: str = 'description'*, *manufacturer: str | None = None*, *nwbfile:* NWBFile | *None = None*) → *Device*

### pynwb.testing.mock.ecephys module

pynwb.testing.mock.ecephys.**mock_ElectrodeGroup**(*name: str | None = None*, *description: str = 'description'*, *location: str = 'location'*, *device:* Device | *None = None*, *position: str | None = None*, *nwbfile:* NWBFile | *None = None*) → *ElectrodeGroup*

pynwb.testing.mock.ecephys.**mock_ElectrodeTable**(*n_rows: int = 5*, *group:* ElectrodeGroup | *None = None*, *nwbfile:* NWBFile | *None = None*) → DynamicTable

pynwb.testing.mock.ecephys.**mock_electrodes**(*n_electrodes: int = 5*, *table: DynamicTable | None = None*, *nwbfile:* NWBFile | *None = None*) → DynamicTableRegion

pynwb.testing.mock.ecephys.**mock_ElectricalSeries**(*name: str | None = None*, *description: str = 'description'*, *data=None*, *rate: float = 30000.0*, *timestamps=None*, *starting_time: float | None = None*, *electrodes: DynamicTableRegion | None = None*, *filtering: str = 'filtering'*, *nwbfile:* NWBFile | *None = None*, *channel_conversion: ndarray | None = None*, *conversion: float = 1.0*, *offset: float = 0.0*) → *ElectricalSeries*

pynwb.testing.mock.ecephys.**mock_SpikeEventSeries**(*name: str | None = None*, *description: str = 'description'*, *data=None*, *timestamps=array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])*, *electrodes: DynamicTableRegion | None = None*, *nwbfile: NWBFile | None = None*) → *SpikeEventSeries*

pynwb.testing.mock.ecephys.**mock_Units**(*num_units: int = 10*, *max_spikes_per_unit: int = 10*, *seed: int = 0*, *nwbfile: NWBFile | None = None*) → *Units*

### pynwb.testing.mock.file module

pynwb.testing.mock.file.**mock_NWBFile**(*session_description: str = 'session_description'*, *identifier: str | None = None*, *session_start_time: datetime = datetime.datetime(1970, 1, 1, 0, 0)*, ***kwargs*)

pynwb.testing.mock.file.**mock_Subject**(*age: str | None = 'P50D'*, *description: str = 'this is a mock mouse.'*, *sex: str | None = 'F'*, *subject_id: str | None = None*, *nwbfile: NWBFile | None = None*)

### pynwb.testing.mock.icephys module

pynwb.testing.mock.icephys.**mock_IntracellularElectrode**(*name: str | None = None*, *description: str = 'description'*, *device: Device | None = None*, *nwbfile: NWBFile | None = None*) → *IntracellularElectrode*

pynwb.testing.mock.icephys.**mock_VoltageClampStimulusSeries**(*name: str | None = None*, *data=None*, *rate: float = 100000.0*, *electrode: IntracellularElectrode | None = None*, *gain: float = 0.02*, *timestamps=None*, *starting_time: float | None = None*, *nwbfile: NWBFile | None = None*) → *VoltageClampStimulusSeries*

pynwb.testing.mock.icephys.**mock_VoltageClampSeries**(*name: str | None = None*, *data=None*, *conversion: float = 1.0*, *resolution: float = nan*, *starting_time: float | None = None*, *rate: float | None = 100000.0*, *electrode: IntracellularElectrode | None = None*, *gain: float = 0.02*, *capacitance_slow: float = 1e-10*, *resistance_comp_correction: float = 70.0*, *nwbfile: NWBFile | None = None*) → *VoltageClampSeries*

pynwb.testing.mock.icephys.**mock_CurrentClampSeries**(*name: str | None = None*, *data=None*, *electrode: IntracellularElectrode | None = None*, *gain: float = 0.02*, *stimulus_description: str = 'N/A'*, *bias_current=None*, *bridge_balance=None*, *capacitance_compensation=None*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time: float | None = None*, *rate: float | None = 100000.0*, *comments: str = 'no comments'*, *description: str = 'no description'*, *control=None*, *control_description=None*, *sweep_number=None*, *offset=0.0*, *unit: str = 'volts'*, *nwbfile: NWBFile | None = None*) → *CurrentClampSeries*

pynwb.testing.mock.icephys.**mock_CurrentClampStimulusSeries**(*name: str | None = None*, *data=None*, *electrode=None*, *gain=0.02*, *stimulus_description='N/A'*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=100000.0*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*, *sweep_number=None*, *offset=0.0*, *unit='amperes'*, *nwbfile: NWBFile | None = None*) → *CurrentClampStimulusSeries*

pynwb.testing.mock.icephys.**mock_IZeroClampSeries**(*name: str | None = None*, *data=None*, *electrode: IntracellularElectrode | None = None*, *gain=0.02*, *stimulus_description='N/A'*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=100000.0*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*, *sweep_number=None*, *offset=0.0*, *unit='volts'*, *nwbfile: NWBFile | None = None*) → *IZeroClampSeries*

pynwb.testing.mock.icephys.**mock_IntracellularRecordingsTable**(*n_rows: int = 5*, *nwbfile: NWBFile | None = None*) → *IntracellularRecordingsTable*

### pynwb.testing.mock.ogen module

pynwb.testing.mock.ogen.**mock_OptogeneticStimulusSite**(*name: str | None = None*, *device: Device | None = None*, *description: str = 'optogenetic stimulus site'*, *excitation_lambda: float = 500.0*, *location: str = 'part of the brain'*, *nwbfile: NWBFile | None = None*) → *OptogeneticStimulusSite*

pynwb.testing.mock.ogen.**mock_OptogeneticSeries**(*name: str | None = None*, *data=None*, *site: OptogeneticStimulusSite | None = None*, *resolution: float = -1.0*, *conversion: float = 1.0*, *timestamps=None*, *starting_time: float | None = None*, *rate: float | None = 10.0*, *comments: str = 'no comments'*, *description: str = 'no description'*, *control=None*, *control_description=None*, *nwbfile: NWBFile | None = None*) → *OptogeneticSeries*

### pynwb.testing.mock.ophys module

pynwb.testing.mock.ophys.**mock_OpticalChannel**(*name: str | None = None*, *description: str = 'description'*, *emission_lambda: float = 500.0*, *nwbfile: NWBFile | None = None*) → *OpticalChannel*

pynwb.testing.mock.ophys.**mock_ImagingPlane**(*name: str | None = None*, *optical_channel: OpticalChannel | None = None*, *description: str = 'description'*, *device: Device | None = None*, *excitation_lambda: float = 500.0*, *indicator: str = 'indicator'*, *location: str = 'unknown'*, *imaging_rate: float = 30.0*, *manifold=None*, *conversion: float = 1.0*, *unit: str = 'meters'*, *reference_frame=None*, *origin_coords=None*, *origin_coords_unit: str = 'meters'*, *grid_spacing=None*, *grid_spacing_unit: str = 'meters'*, *nwbfile: NWBFile | None = None*) → *ImagingPlane*

pynwb.testing.mock.ophys.**mock_OnePhotonSeries**(*name: str | None = None*, *imaging_plane: ImagingPlane | None = None*, *data=None*, *rate: float | None = 50.0*, *unit: str = 'n.a.'*, *exposure_time=None*, *binning=None*, *power=None*, *intensity=None*, *format=None*, *pmt_gain=None*, *scan_line_rate=None*, *external_file=None*, *starting_frame=[0]*, *bits_per_pixel=None*, *dimension=None*, *resolution=-1.0*, *conversion=1.0*, *offset=0.0*, *timestamps=None*, *starting_time=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*, *device=None*, *nwbfile: NWBFile | None = None*) → *OnePhotonSeries*

pynwb.testing.mock.ophys.**mock_TwoPhotonSeries**(*name: str | None = None*, *imaging_plane:* ImagingPlane *| None = None*, *data=None*, *rate=50.0*, *unit='n.a.'*, *format=None*, *field_of_view=None*, *pmt_gain=None*, *scan_line_rate=None*, *external_file=None*, *starting_frame=[0]*, *bits_per_pixel=None*, *dimension=None*, *resolution=-1.0*, *conversion=1.0*, *offset=0.0*, *timestamps=None*, *starting_time=None*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*, *device=None*, *nwbfile:* NWBFile *| None = None*) → *TwoPhotonSeries*

pynwb.testing.mock.ophys.**mock_PlaneSegmentation**(*description: str = 'no description'*, *imaging_plane:* ImagingPlane *| None = None*, *name: str | None = None*, *reference_images=None*, *n_rois: int = 5*, *nwbfile:* NWBFile *| None = None*) → *PlaneSegmentation*

pynwb.testing.mock.ophys.**mock_ImageSegmentation**(*plane_segmentations: Sequence[*PlaneSegmentation*] | None = None*, *name: str | None = None*, *nwbfile:* NWBFile *| None = None*) → *ImageSegmentation*

pynwb.testing.mock.ophys.**mock_RoiResponseSeries**(*name: str | None = None*, *data=None*, *unit: str = 'n.a.'*, *rois=None*, *resolution=-1.0*, *conversion=1.0*, *timestamps=None*, *starting_time=None*, *rate=50.0*, *comments='no comments'*, *description='no description'*, *control=None*, *control_description=None*, *n_rois=None*, *plane_segmentation:* PlaneSegmentation *| None = None*, *nwbfile:* NWBFile *| None = None*) → *RoiResponseSeries*

pynwb.testing.mock.ophys.**mock_DfOverF**(*roi_response_series:* RoiResponseSeries *| None = None*, *name: str | None = None*, *nwbfile:* NWBFile *| None = None*) → *DfOverF*

pynwb.testing.mock.ophys.**mock_Fluorescence**(*roi_response_series: Sequence[*RoiResponseSeries*] | None = None*, *name: str | None = None*, *nwbfile:* NWBFile *| None = None*) → *Fluorescence*

### pynwb.testing.mock.utils module

pynwb.testing.mock.utils.**name_generator**(*name*)

Returns unique names of neurodata types using an incrementing number. The first time you pass "TimeSeries" it returns "TimeSeries". The second time, it returns "TimeSeries2", the third time it returns "TimeSeries3", etc.

**Parameters**
**name** (*str*) – name of neurodata_type, e.g. TimeSeries

**Returns**
**name of neurodata_object**

**Return type**
str

### Module contents

The mock module provides mock instances of common neurodata types which can be used to write tests for downstream libraries. For instance, to write an RoiResponseSeries, you need a PlaneSegmentation, which requires an ImagingPlane, which in turn requires an OpticalChannel and a Device, all of which need to be populated with reasonable mock data. This library streamlines the process of creating mock objects by generating the required prerequisites for this object on-the-fly and automatically using reasonable defaults. Any of these default objects and parameters can be overridden.

### Submodules

### pynwb.testing.icephys_testutils module

Module with helper functions to facilitate testing

pynwb.testing.icephys_testutils.**create_icephys_stimulus_and_response**(*sweep_number*, *electrode*, *randomize_data*)

Internal helper function to construct a dummy stimulus and response pair representing an intracellular recording:

**Parameters**

- **sweep_number** (*int*) – Integer sweep number of the recording
- **electrode** (*pynwb.icephys.IntracellularElectrode*) – Intracellular electrode used
- **randomize_data** (*bool*) – Randomize data values in the stimulus and response

**Returns**

Tuple of VoltageClampStimulusSeries with the stimulus and VoltageClampSeries with the response.

pynwb.testing.icephys_testutils.**create_icephys_testfile**(*filename=None*, *add_custom_columns=True*, *randomize_data=True*, *with_missing_stimulus=True*)

Create a small but relatively complex icephys test file that we can use for testing of queries.

**Parameters**

- **filename** (*str, None*) – The name of the output file to be generated. If set to None then the file is not written but only created in memory
- **add_custom_colums** (*bool*) – Add custom metadata columns to each table
- **randomize_data** (*bool*) – Randomize data values in the stimulus and response

**Returns**

NWBFile object with icephys data created for writing. NOTE: If filename is provided then the file is written to disk, but the function does not read the file back. If you want to use the file from disk then you will need to read it with NWBHDF5IO.

**Return type**

*NWBFile*

### pynwb.testing.make_test_files module

### pynwb.testing.testh5io module

**class** pynwb.testing.testh5io.**NWBH5IOMixin**

Bases: object

Mixin class for methods to run a roundtrip test writing an NWB file with an Container and reading the Container from the NWB file. The setUp, test_roundtrip, and tearDown methods will be run by unittest.

The abstract methods setUpContainer, addContainer, and getContainer needs to be implemented by classes that include this mixin.

Example:

```
class TestMyContainerIO(NWBH5IOMixin, TestCase):
    def setUpContainer(self):
        # return a test Container to read/write
    def addContainer(self, nwbfile):
        # add the test Container to an NWB file
    def getContainer(self, nwbfile):
        # return the test Container from an NWB file
```

This code is adapted from hdmf.testing.H5RoundTripMixin.

**setUp**()

**tearDown**()

**abstract setUpContainer**()

Should return the test Container to read/write

**test_roundtrip**()

Test whether the read Container has the same contents as the original Container and validate the file.

**test_roundtrip_export**()

Test whether the test Container read from an exported file has the same contents as the original test Container and validate the file

**roundtripContainer**(*cache_spec=True*)

Add the Container to an NWBFile, write it to file, read the file, and return the Container from the file.

**roundtripExportContainer**(*cache_spec=True*)

Add the test Container to an NWBFile, write it to file, read the file, export the read NWBFile to another file, and return the test Container from the file

**abstract addContainer**(*nwbfile*)

Should add the test Container to the given NWBFile

**abstract getContainer**(*nwbfile*)

Should return the test Container from the given NWBFile

**validate**()

Validate the created files

**class** pynwb.testing.testh5io.**AcquisitionH5IOMixin**

Bases: *NWBH5IOMixin*

Mixin class for methods to run a roundtrip test writing an NWB file with an Container as an acquisition and reading the Container as an acquisition from the NWB file. The setUp, test_roundtrip, and tearDown methods will be run by unittest.

The abstract method setUpContainer needs to be implemented by classes that include this mixin.

Example:

```
class TestMyContainerIO(NWBH5IOMixin, TestCase):
    def setUpContainer(self):
        # return a test Container to read/write
```

This code is adapted from hdmf.testing.H5RoundTripMixin.

**addContainer**(*nwbfile*)

Add an NWBDataInterface object to the file as an acquisition

**getContainer**(*nwbfile*)

Get the NWBDataInterface object from the file

**class** pynwb.testing.testh5io.**NWBH5IOFlexMixin**

Bases: object

Mixin class that includes methods to run a flexible roundtrip test. The setUp, test_roundtrip, and tearDown methods will be run by unittest.

The abstract methods getContainerType, addContainer, and getContainer must be implemented by classes that includethis mixin.

Example:

```
class TestMyContainerIO(NWBH5IOFlexMixin, TestCase):
    def getContainerType(self):
        # return the name of the type of Container being tested, for test ID␣
↪purposes
    def addContainer(self):
        # add the test Container to the test NWB file
    def getContainer(self, nwbfile):
        # return the test Container from an NWB file
```

This class is more flexible than NWBH5IOMixin and should be used for new roundtrip tests.

This code is adapted from hdmf.testing.H5RoundTripMixin.

**setUp**()

**tearDown**()

**get_manager**()

**abstract getContainerType**() → str

Return the name of the type of Container being tested, for test ID purposes.

**abstract addContainer**()

Add the test Container to the NWBFile.

The NWBFile is accessible from self.nwbfile. The Container should be accessible from getContainer(self.nwbfile).

**abstract getContainer**(*nwbfile:* NWBFile)

Return the test Container from the given NWBFile.

**test_roundtrip**()

Test whether the read Container has the same contents as the original Container and validate the file.

**test_roundtrip_export**()

Test whether the test Container read from an exported file has the same contents as the original test Container and validate the file.

**roundtripContainer**(*cache_spec=True*)

Write the file, validate the file, read the file, and return the Container from the file.

**roundtripExportContainer**(*cache_spec=True*)

Roundtrip the container, then export the read NWBFile to a new file, validate the files, and return the test Container from the file.

**validate**()

Validate the created files.

#### pynwb.testing.utils module

pynwb.testing.utils.**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.

#### Module contents

### 6.11.2 Submodules

#### pynwb.core module

pynwb.core.**prepend_string**(*string*, *prepend=' '*)

**class** pynwb.core.**NWBMixin**(*name*)

Bases: 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: *NWBMixin*, Container

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

**namespace = 'core'**

neurodata_type = 'NWBContainer'

class pynwb.core.NWBDataInterface(*name*)

Bases: *NWBContainer*

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

namespace = 'core'

neurodata_type = 'NWBDataInterface'

class pynwb.core.NWBData(*name*, *data*)

Bases: *NWBMixin*, Data

**Parameters**

- **name** (str) – the name of this container
- **data** (str or int or float or bytes or bool or ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or Data) – the source of the data

property data

__getitem__(*args*)

append(*arg*)

extend(*arg*)

The extend_data method adds all the elements of the iterable arg to the end of the data of this Data container.

**Parameters**
**arg** – The iterable to add to the end of this VectorData

namespace = 'core'

neurodata_type = 'NWBData'

class pynwb.core.ScratchData(*name*, *data*, *notes=''*, *description=None*)

Bases: *NWBData*

**Parameters**

- **name** (str) – the name of this container
- **data** (str or int or float or bytes or bool or ndarray or list or tuple or Dataset or Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO or Data) – the source of the data
- **notes** (str) – notes about the data. This argument will be deprecated. Use description instead
- **description** (str) – notes about the data

property notes

namespace = 'core'

neurodata_type = 'ScratchData'

class pynwb.core.**NWBTable**(*columns*, *name*, *data=[]*)

Bases: Table

Defined in PyNWB for API backward compatibility. See HDMF Table for details.

**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 Array or StrDataset or HDMFDataset or AbstractDataChunkIterator or DataIO) – the source of the data

class pynwb.core.**MultiContainerInterface**(*name*)

Bases: *NWBDataInterface*, MultiContainerInterface

Defined in PyNWB for API backward compatibility. See HDMF MultiContainterInterface for details.

**Parameters**

**name** (str) – the name of this container

### pynwb.device module

class pynwb.device.**Device**(*name*, *description=None*, *manufacturer=None*)

Bases: *NWBContainer*

Metadata about a data acquisition device, e.g., recording system, electrode, microscope.

**Parameters**

- **name** (str) – the name of this device
- **description** (str) – Description of the device (e.g., model, firmware version, processing software version, etc.)
- **manufacturer** (str) – the name of the manufacturer of this device

**property description**

Description of the device (e.g., model, firmware version, processing software version, etc.)

**property manufacturer**

the name of the manufacturer of this device

**namespace = 'core'**

**neurodata_type = 'Device'**

### pynwb.resources module

class pynwb.resources.**HERD**(*keys=None*, *files=None*, *entities=None*, *objects=None*, *object_keys=None*, *entity_keys=None*, *type_map=None*)

Bases: HERD

HDMF External Resources Data Structure. A table for mapping user terms (i.e. keys) to resource entities.

**Parameters**

- **keys** (KeyTable) – The table storing user keys for referencing resources.
- **files** (FileTable) – The table for storing file ids used in external resources.

- **entities** (EntityTable) – The table storing entity information.
- **objects** (ObjectTable) – The table storing object information.
- **object_keys** (ObjectKeyTable) – The table storing object-key relationships.
- **entity_keys** (EntityKeyTable) – The table storing entity-key relationships.
- **type_map** (TypeMap) – The type map. If None is provided, the HDMF-common type map will be used.

### pynwb.spec module

class pynwb.spec.**NWBRefSpec**(*target_type*, *reftype*)

Bases: 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: 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: LinkSpec

**Parameters**

- **doc** (str) – a description about what this link represents
- **target_type** (str or BaseStorageSpec) – the target type GroupSpec or DatasetSpec
- **quantity** (str or int) – the required number of allowed instance
- **name** (str) – the name of this link

property **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.

**property neurodata_type_inc**

**property 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: 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: *BaseStorageOverride*, DatasetSpec

The Spec class to use for NWB dataset specifications.

Classes will automatically include NWBData if None is specified.

**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: *BaseStorageOverride*, GroupSpec

The Spec class to use for NWB group specifications.

Classes will automatically include NWBContainer if None is specified.

**Parameters**

- **doc** (str) – a description about what this specification represents
- **name** (str) – the name of the Group that is written to the file. If this argument is omitted, users will be required to enter a `name` field when creating instances of this data type in the API. Another option is to specify `default_name`, in which case this name will be used as the name of the Group if no other name is provided.
- **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 allowable number of instance of this group in a certain location. See table of options here. Note that if youspecify `name`, `quantity` cannot be `'*'`, `'+'`, or an integer greater that 1, because you cannot have more than one group of the same name in the same parent group.
- **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 "neurodata_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 the Group that is written to the file. If this argument is omitted, users will be required to enter a `name` field when creating instances of this data type in the

API. Another option is to specify `default_name`, in which case this name will be used as the name of the Group if no other name is provided.

- **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 allowable number of instance of this group in a certain location. See table of options here. Note that if youspecify `name`, `quantity` cannot be `'*'`, `'+'`, or an integer greater that 1, because you cannot have more than one group of the same name in the same parent group.
- **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: 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: 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

Command line tool to Validate an NWB file against a namespace.

pynwb.validate.**validate**(*io=None*, *namespace=None*, *paths=None*, *use_cached_namespaces=True*, *verbose=False*, *driver=None*)

Validate NWB file(s) against a namespace or its cached namespaces.

NOTE: If an io object is provided and no namespace name is specified, then the file will be validated against the core namespace, even if use_cached_namespaces is True.

**Parameters**

- **io** (HDMFIO) – An open IO to an NWB file.
- **namespace** (str) – A specific namespace to validate against.
- **paths** (list) – List of NWB file paths.
- **use_cached_namespaces** (bool) – Whether to use namespaces cached within the file for validation.
- **verbose** (bool) – Whether or not to print messages to stdout.

- **driver** (str) – Driver for h5py to use when opening the HDF5 file.

**Returns**
Validation errors in the file.

**Return type**
list or list, bool

pynwb.validate.**validate_cli**()
CLI wrapper around pynwb.validate.

### 6.11.3 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 the TypeMap for the given extensions. If no extensions are provided,**
return the TypeMap for 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**
TypeMap loaded for the given extension or NWB core namespace

**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.**get_nwbfile_version**(*h5py_file*)

Get the NWB version of the file if it is an NWB file.

**Returns**

Tuple consisting of: 1) the original version string as stored in the file and 2) a tuple with the parsed components of the version string, consisting of integers and strings, e.g., (2, 5, 1, beta). (None, None) will be returned if the file is not a valid NWB file or the nwb_version is missing, e.g., in the case when no data has been written to the file yet.

**Parameters**

**h5py_file** (File) – An NWB file

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*)

**Parse the YAML file for a given neurodata_type that is a subclass of NWBContainer and automatically generate its**

python API. This will work for most containers, but is known to not work for descendants of MultiContainerInterface and DynamicTable, so these must be defined manually (for now). *get_class* infers the API mapping directly from the specification. If you want to define a custom mapping, you should not use this function and you should define the class manually.

Examples:

Generating and registering an extension is as simple as:

```
MyClass = get_class('MyClass', 'ndx-my-extension')
```

*get_class* defines only the *__init__* for the class. In cases where you want to provide additional methods for querying, plotting, etc. you can still use *get_class* and attach methods to the class after-the-fact, e.g.:

```
def get_sum(self, a, b):
    return self.feat1 + self.feat2

MyClass.get_sum = get_sum
```

**Parameters**

- **neurodata_type** (str) – the neurodata_type to get the NWBContainer class for
- **namespace** (str) – the namespace the neurodata_type is defined in

**class** pynwb.**NWBHDF5IO**(*path=None*, *mode='r'*, *load_namespaces=True*, *manager=None*, *extensions=None*, *file=None*, *comm=None*, *driver=None*, *herd_path=None*)

Bases: HDF5IO

**Parameters**

- **path** (str or Path) – 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 or when *manager* is not None or when *extensions* is not None
- **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 or S3File) – a pre-existing h5py.File object
- **comm** (Intracomm) – the MPI communicator to use for parallel I/O
- **driver** (str) – driver for h5py to use when opening HDF5 file
- **herd_path** (str) – The path to the HERD

**static can_read**(*path: str*)

Determine whether a given path is readable by this class

**property nwb_version**

Get the version of the NWB file opened via this NWBHDF5IO object.

**Returns**

Tuple consisting of: 1) the original version string as stored in the file and 2) a tuple with the parsed components of the version string, consisting of integers and strings, e.g., (2, 5, 1, beta). (None, None) will be returned if the nwb_version is missing, e.g., in the case when no data has been written to the file yet.

**read**(*skip_version_check=False*)

Read the NWB file from the IO source.

**raises TypeError**

If the NWB file version is missing or not supported

**return**

NWBFile container

**Parameters**

**skip_version_check** (bool) – skip checking of NWB version

**export**(*src_io*, *nwbfile=None*, *write_args=None*)

Export an NWB file to a new NWB file using the HDF5 backend.

If `nwbfile` is provided, then the build manager of `src_io` is used to build the container, and the resulting builder will be exported to the new backend. So if `nwbfile` is provided, `src_io` must have a non-None manager property. If `nwbfile` is None, then the contents of `src_io` will be read and exported to the new backend.

Arguments can be passed in for the `write_builder` method using `write_args`. Some arguments may not be supported during export. `{'link_data': False}` can be used to copy any datasets linked to from the original file instead of creating a new link to those datasets in the exported file.

The exported file will not contain any links to the original file. All links, internal and external, will be preserved in the exported file. All references will also be preserved in the exported file.

The exported file will use the latest schema version supported by the version of PyNWB used. For example, if the input file uses the NWB schema version 2.1 and the latest schema version supported by PyNWB is 2.3, then the exported file will use the 2.3 NWB schema.

Example usage:

```
with NWBHDF5IO(self.read_path, mode='r') as read_io:
    nwbfile = read_io.read()
    # ...  # modify nwbfile
    nwbfile.set_modified()  # this may be necessary if the modifications
↪are changes to attributes

    with NWBHDF5IO(self.export_path, mode='w') as export_io:
        export_io.export(src_io=read_io, nwbfile=nwbfile)
```

See *Exporting NWB files* and *Adding/Removing Containers from an NWB File* for more information and examples.

**Parameters**

- **src_io** (HDMFIO) – the HDMFIO object (such as NWBHDF5IO) that was used to read the data to export
- **nwbfile** (NWBFile) – the NWBFile object to export. If None, then the entire contents of src_io will be exported
- **write_args** (dict) – arguments to pass to write_builder

modindex

CHAPTER

SEVEN

# INSTALLING PYNWB FOR DEVELOPERS

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

1. Python 3.8, 3.9, 3.10, or 3.11
2. pip

## 7.1 Set up a virtual environment

For development, we recommend installing PyNWB in a virtual environment in editable mode. You can use the virtualenv tool to create a new virtual environment. Or you can use the conda package and environment management system for managing virtual environments.

### 7.1.1 Option 1: Using virtualenv

First, install the latest version of the `virtualenv` tool and use it to create a new virtual environment. This virtual environment will be stored in the `venv` directory in the current directory.

```
pip install -U virtualenv
virtualenv venv
```

On macOS or Linux, run the following to activate your new virtual environment:

```
source venv/bin/activate
```

On Windows, run the following to activate your new virtual environment:

```
venv\Scripts\activate
```

This virtual environment is a space where you can install Python packages that are isolated from other virtual environments. This is especially useful when working on multiple Python projects that have different package requirements and for testing Python code with different sets of installed packages or versions of Python.

Activate your newly created virtual environment using the above command whenever you want to work on HDMF. You can also deactivate it using the `deactivate` command to return to the base environment.

### 7.1.2 Option 2: Using conda

First, install Anaconda to install the `conda` tool. Then create and activate a new virtual environment called "venv" with Python 3.8 installed.

```
conda create --name venv python=3.8
conda activate venv
```

Similar to a virtual environment created with `virtualenv`, a conda environment is a space where you can install Python packages that are isolated from other virtual environments. In general, you should use `conda install` instead of `pip install` to install packages in a conda environment.

Activate your newly created virtual environment using the above command whenever you want to work on HDMF. You can also deactivate it using the `conda deactivate` command to return to the base environment.

## 7.2 Install from Git repository

After you have created and activated a virtual environment, clone the PyNWB git repository from GitHub, install the package requirements using the pip Python package manager, and install PyNWB in editable mode.

```
git clone --recurse-submodules https://github.com/NeurodataWithoutBorders/pynwb.git
cd pynwb
pip install -r requirements.txt -r requirements-dev.txt
pip install -e .
```

## 7.3 Run tests

For running the tests, it is required to install the development requirements. Again, first activate your virtualenv or conda environment.

```
git clone --recurse-submodules https://github.com/NeurodataWithoutBorders/pynwb.git
cd pynwb
pip install -r requirements.txt -r requirements-dev.txt
pip install -e .
tox
```

For debugging it can be useful to keep the intermediate NWB files created by the tests. To keep these files create the environment variables `CLEAN_NWB`/`CLEAN_HDMF` and set them to 1.

## 7.4 FAQ

1. I am using a git cloned copy of PyNWB and getting the error: `RuntimeError: Unable to load a TypeMap - no namespace file found`

   or the error: `RuntimeError: 'core' is not a registered namespace.`

   - The PyNWB repo uses git submodules that have to be checked out when cloning the repos. Please make sure you are using the `--recurse-submodules` flag when running `git clone`:

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

You can also run the following on your existing cloned repo.

```
git submodule init
git submodule update --checkout --force
```

2. I did a `git pull` but I'm getting errors that some `neurodata_type` does not exist.
   - The PyNWB repo uses git submodules that have to be updated as well. Please make sure you are using the `git pull --recurse-submodules`

CHAPTER

EIGHT

# 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.

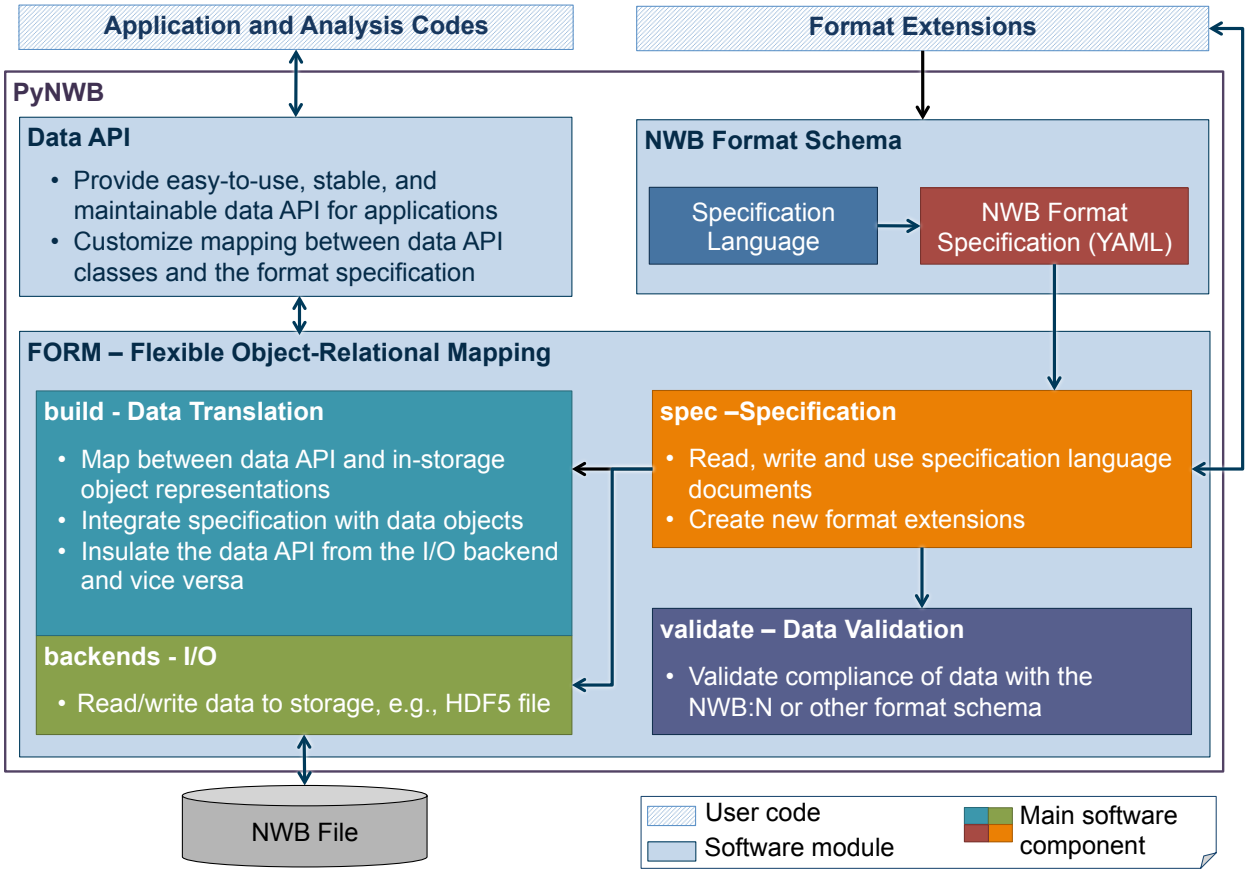



Fig. 1: Overview of the high-level software architecture of PyNWB (click to enlarge).

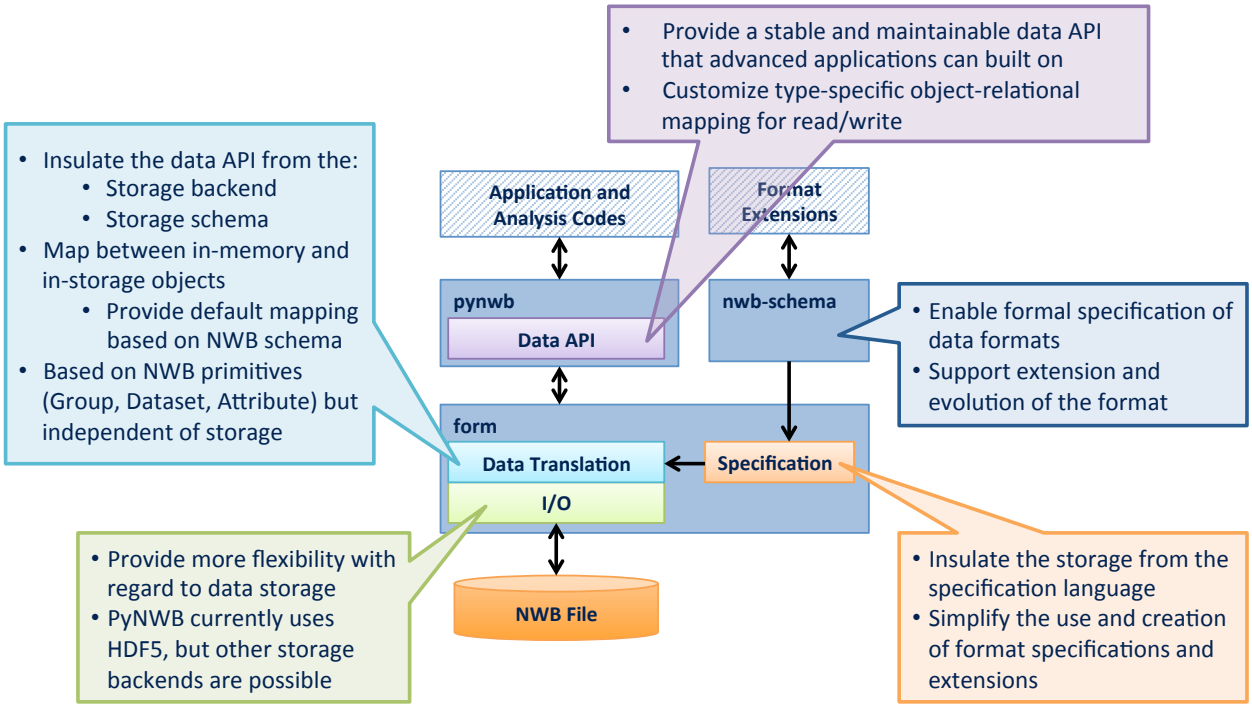



Fig. 2: We choose a modular design for PyNWB to enable flexibility and separate the various aspects of the NWB:N ecosystem (click to enlarge).

## 8.1 Main Concepts

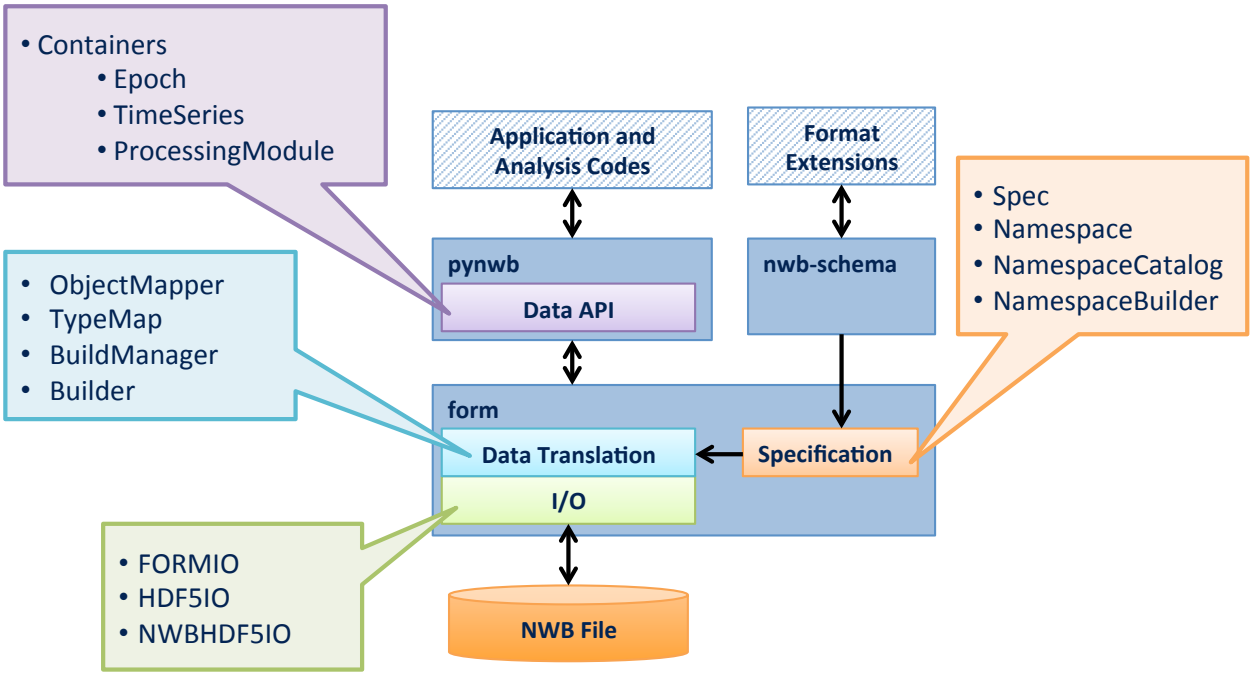



Fig. 3: Overview of the main concepts/classes in PyNWB and their location in the overall software architecture (click to enlarge).

### 8.1.1 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*

### 8.1.2 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`

### 8.1.3 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.

---

### 8.1.4 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.objectmapper`
  - NWB-specific ObjectMappers are locate in submodules of *`pynwb.io`*

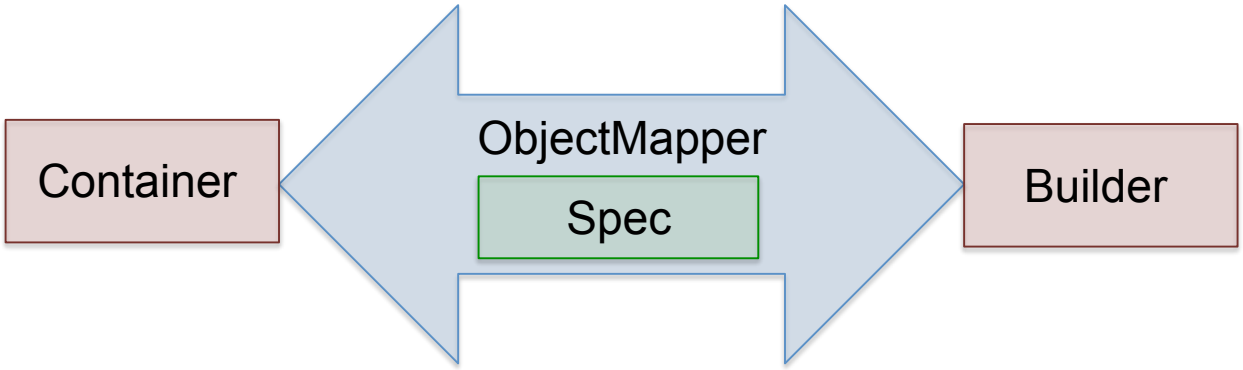



Fig. 4: Relationship between *Container*, *Builder*, *ObjectMapper*, and *Spec*

## 8.2 Additional Concepts

### 8.2.1 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

### 8.2.2 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

### 8.2.3 BuildManager

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

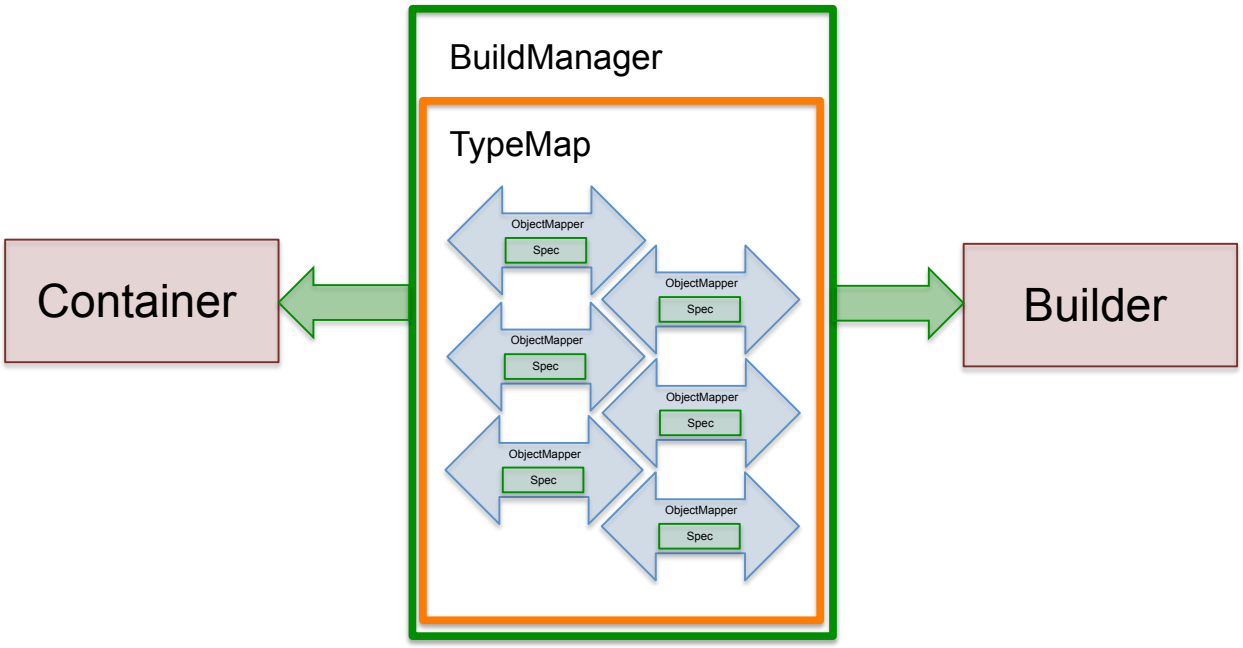



Fig. 5: Overview of *BuildManager* (and *TypeMap*) (click to enlarge).

### 8.2.4 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

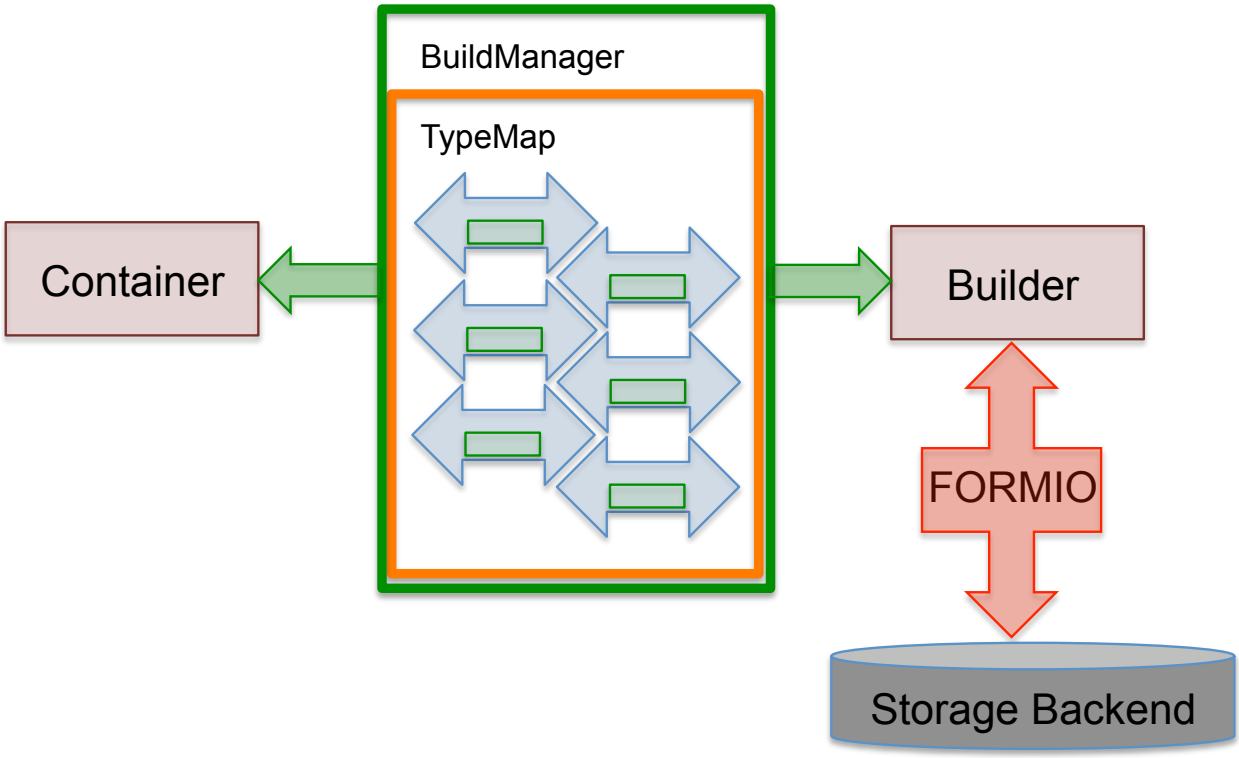



Fig. 6: Overview of *HDMFIO* (click to enlarge).

CHAPTER

NINE

# 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
- requirements-min.txt

## 9.1 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
```

## 9.2 requirements-(dev|doc).txt

Any of these requirements files can be updated using the following scripts:

```
cd pynwb

# Set the requirements file to update: requirements-dev.txt or requirements-doc.txt
target_requirements=requirements-dev.txt

mkvirtualenv pynwb-requirements

# Install updated requirements
pip install -U -r $target_requirements

# If relevant, you could pip install new requirements now
```

(continues on next page)

(continued from previous page)

```
# 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
```

## 9.3 requirements-min.txt

Minimum requirements should be updated manually if a new feature or bug fix is added in a dependency that is required for proper running of PyNWB. Minimum requirements should also be updated if a user requests that PyNWB be installable with an older version of a dependency, all tests pass using the older version, and there is no valid reason for the minimum version to be as high as it is.

CHAPTER

# TEN

# SOFTWARE PROCESS

## 10.1 Continuous Integration

PyNWB is tested against Ubuntu, macOS, and Windows operating systems. The project has both unit and integration tests. Tests run on GitHub Actions.

Each time a PR is published or updated, the project is built, packaged, and tested on all supported 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.

## 10.2 Coverage

Code coverage is computed and reported using the coverage tool. There are two coverage-related badges in the README file. One shows the status of the GitHub Action workflow which runs the coverage tool and uploads the report to codecov, and the other badge shows the percentage coverage reported from codecov. A detailed report can be found on codecov, which shows line by line which lines are covered by the tests.

## 10.3 Installation Requirements

pyproject.toml contains a list of package dependencies and their version ranges allowed for running PyNWB. As a library, upper bound version constraints create more harm than good in the long term (see this blog post) so we avoid setting upper bounds on requirements.

If some of the packages are outdated, see *How to Update Requirements Files*.

## 10.4 Testing Requirements

There are several kinds of requirements files used for testing PyNWB.

The first one is the requirements-min.txt file, which lists the package dependencies and their minimum versions for installing PyNWB.

The second one is requirements.txt, which lists the pinned (concrete) dependencies to reproduce an entire development environment to use PyNWB.

The third one is requirements-dev.txt, which lists the pinned (concrete) dependencies to reproduce an entire development environment to use PyNWB, run PyNWB tests, check code style, compute coverage, and create test environments.

The final one is environment-ros3.yml, which lists the dependencies used to test ROS3 streaming in PyNWB.

## 10.5 Documentation Requirements

requirements-doc.txt lists the dependencies to generate the documentation for PyNWB. Both this file and requirements.txt are used by ReadTheDocs to initialize the local environment for Sphinx to run.

## 10.6 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 a wheel (*.whl) and source distribution (*.tar.gz) for Python 3 and uploads them back to GitHub as a releases. 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.

It is important to note that GitHub automatically generates source code archives in .zip and .tar.gz formats and attaches those files to all releases as an asset. These files currently do not contain the submodules within PyNWB and thus do not serve as a complete installation. For a complete source code archive, use the source distribution generated by CircleCI, typically named *pynwb-{version}.tar.gz*.

## 10.7 Coordinating with nwb-schema Repository and Releases

The default branch is "dev". It is important that all releases of PyNWB contain a released version of nwb-schema. If a release contains an unreleased version of nwb-schema, e.g., from an untagged commit on the "dev" branch, then tracking the identity of the included nwb-schema becomes difficult and the same version string could point to two different versions of the schema.

Whenever the "dev" branch of the nwb-schema repo is updated, a commit should be made to the "schema_x.y.z" branch of PyNWB, where "x.y.z" is the upcoming version of nwb-schema, that updates the nwb-schema submodule to the latest commit of the "dev" branch on nwb-schema. If the update to nwb-schema is the first change after a release, the "schema_x.y.z" branch should be created, the nwb-schema submodule should be updated, and a draft PR should be made for merging the "schema_x.y.z" branch to "dev". This PR provides a useful public view into how the API changes with each change to the schema.

If the change in nwb-schema requires an accompanying change to PyNWB, then a new branch should be made with the corresponding changes, and a new PR should be made for merging the new branch into the "schema_x.y.z" branch. The PR should be merged in GitHub's "squash and merge" mode.

When a new version of nwb-schema x.y.z is released, the "schema_x.y.z" branch of PyNWB should be checked to ensure that the nwb-schema submodule points to the new release-tagged commit of nwb-schema. Then the PR should be merged into dev with GitHub's "merge" mode. Commits should NOT be squashed because they will usually represent independent changes to the API or schema, and the git history should reflect those changes separately.

The "dev" branch should NEVER contain unreleased versions of nwb-schema to prevent cases of users and developers accidentally publishing files with unreleased schema. This problem cannot be completely avoided, however, as users could still publish files generated from the "schema_x.y.z" branch of PyNWB.

The nwb-schema uses hdmf-common-schema. Changes in hdmf-common-schema that affect nwb-schema result in version changes of nwb-schema and as such are managed in the same fashion. One main difference is that updates to hdmf-common-schema may also involve updates to version requirements for HDMF in PyNWB.

CHAPTER

# ELEVEN

# 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.

---

## 11.1 Prerequisites

- All CI tests are passing on GitHub Actions.
- You have a GPG signing key.
- Dependency versions in `requirements.txt`, `requirements-dev.txt`, `requirements-opt.txt`, `requirements-doc.txt`, and `requirements-min.txt` are up-to-date.
- Legal information and copyright dates in `Legal.txt`, `license.txt`, `README.rst`, `docs/source/conf.py`, and any other files are up-to-date.
- Package information in `setup.py` is up-to-date.
- `README.rst` information is up-to-date.
- The `nwb-schema` submodule is up-to-date. The version number should be checked manually in case syncing the git submodule does not work as expected.
- Documentation reflects any new features and changes in PyNWB functionality.
- Documentation builds locally.
- Documentation builds on the ReadTheDocs project on the "dev" build.
- Release notes have been prepared.
- An appropriate new version number has been selected.

## 11.2 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.

## 11.3 Publish release on PyPI: Step-by-step

1. Make sure that all CI tests are passing on GitHub Actions.
2. List all tags sorted by version.

   ```
   $ git tag -l | sort -V
   ```

3. Choose the next release version number and store it in a variable.

   ```
   $ release=X.Y.Z
   ```

   **Warning:** To ensure the packages are uploaded on PyPI, tags must match this regular expression: `^[0-9]+.[0-9]+.[0-9]+$`.

4. Download the 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 the "Deploy release" GitHub Actions workflow which will automatically upload the wheels and source distribution to both the PyNWB PyPI project page and a new GitHub release using the nwb-bot account.

7. Check the status of the builds on GitHub Actions.
8. Once the builds are completed, check that the distributions are available on PyNWB PyPI project page and that a new GitHub release was created.
9. Copy the release notes from `CHANGELOG.md` to the newly created GitHub release.

10. Create a clean testing environment to test the installation.

    On bash/zsh:

```
$ python -m venv pynwb-${release}-install-test && \
  source pynwb-${release}-install-test/bin/activate
```

    On other shells, see the Python instructions for creating a virtual environment.

11. Test the installation:

```
$ pip install pynwb && \
  python -c "import pynwb; print(pynwb.__version__)"
```

10. Cleanup

    On bash/zsh:

```
$ deactivate && \
  rm -rf dist/* && \
  rm -rf pynwb-${release}-install-test
```

## 11.4 Publish release on conda-forge: Step-by-step

**Warning:** Publishing on conda requires you to have the corresponding package version uploaded on PyPI. So you have to do the PyPI and Github release before you do the conda release.

---

**Note:** Conda-forge maintains a bot called "regro-cf-autotick-bot" that regularly monitors PyPI for new releases of packages that are also on conda-forge. When a new release is detected, usually within 24 hours of publishing on PyPI, the bot will create a Pull Request with the correct modifications to the version and sha256 values in `meta.yaml`. If the requirements in `setup.py` have been changed, then you need to modify the requirements/run section in `meta.yaml` manually to reflect these changes. Once tests pass, merge the PR, and a new release will be published on Anaconda cloud. This is the easiest way to update the package version on conda-forge.

---

In order to release a new version on conda-forge manually, follow the steps below:

1. Store the release version string (this should match the PyPI version that you already published).

```
$ release=X.Y.Z
```

2. Fork the pynwb-feedstock repository to your GitHub user account.
3. Clone the forked feedstock to your local filesystem.

   Fill the YOURGITHUBUSER part.

```
$ cd /tmp && git clone https://github.com/YOURGITHUBUSER/pynwb-feedstock.git
```

4. Download the 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 (line 2) and sha256 (line 3).

   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
```

   If the requirements in `setup.py` have been changed, then modify the requirements/run list in the `meta.yaml` file to reflect these changes.

7. Push the changes to your fork.

```
$ git push origin $release
```

8. Create a Pull Request.

   Create a pull request against the main feedstock repository. After the tests pass, merge the PR, and a new release will be published on Anaconda cloud.

CHAPTER

# TWELVE

# TESTING

PyNWB has a goal of 100% test coverage, so it is important that any changes to the code base be covered by tests. Our tests are split into four main categories, all of which are in the `tests/` folder:

- `tests/back_compat` tests to check compatibility of the API with older version of NWB. This is a collection of small NWB files with many different versions and a small testing script that ensures we can still read them.
- `tests/unit`: tests that cover each use-case of each method or function
- `tests/integration`: tests that neurodata types and NWB file can be written and read in various modes and backends including HDF5 and using the ros3 driver.
- `tests/validation`: tests command line usage of the `pynwb.validate` NWB validator.

In addition to the `tests/` folder, there is also a *pynwb.testing* package. This module does not contain any tests but instead contains classes and/or functions that help with the implementation of tests for PyNWB and are used in the testing suite and also be imported and used by downstream libraries, e.g., to implement test suites for extensions to the NWB (NDX).

## 12.1 Mock

When creating tests within PyNWB and in downstream libraries, it is often necessary to create example instances of neurodata objects. However, this can be quite laborious for some types. For instance, creating an *RoiResponseSeries* would require you to make a DynamicTableRegion of a *PlaneSegmentation* table with the appropriate rows. This object in turn requires input of an *ImageSegmentation* object, which in turn requires a *Device* and an *OpticalChannel* object. In the end, creating a single neurodata object in this case requires the creation of 5 other objects. *testing.mock* is a module that creates boilerplate objects with a single line of code that can be used for testing. In this case, you could simply run

```
from pynwb.testing.mock.ophys import mock_RoiResponseSeries

roi_response_series = mock_RoiResponseSeries()
```

This acts much like the standard *RoiResponseSeries* class constructor, except that *all* of the fields have defaults. It auto-magically creates a DynamicTableRegion of a *mock_PlaneSegmentation*, which in turn calls the `mock` version of all the other necessary neurodata types. You can customize any of these fields just as you would normally, overriding these defaults:

```
from pynwb.testing.mock.ophys import mock_RoiResponseSeries

roi_response_series = mock_RoiResponseSeries(data=[[1,2,3], [1,2,3]])
```

If you want to create objects and automatically add them to an *NWBFile*, create an *NWBFile* and pass it into the mock function:

```
from pynwb.testing.mock.file import mock_NWBFile
from pynwb.testing.mock.ophys import mock_RoiResponseSeries

nwbfile = mock_NWBFile()
mock_RoiResponseSeries(nwbfile=nwbfile)
```

Now this NWBFile contains an *RoiResponseSeries* and all the upstream classes:

```
>>> print(nwbfile)

root pynwb.file.NWBFile at 0x4335131760
Fields:
  devices: {
    Device <class 'pynwb.device.Device'>,
    Device2 <class 'pynwb.device.Device'>
  }
  file_create_date: [datetime.datetime(2023, 6, 26, 21, 56, 44, 322249,␣
↪tzinfo=tzlocal())]
  identifier: 3c13e816-a50f-49a9-85ec-93b9944c3e79
  imaging_planes: {
    ImagingPlane <class 'pynwb.ophys.ImagingPlane'>,
    ImagingPlane2 <class 'pynwb.ophys.ImagingPlane'>
  }
  processing: {
    ophys <class 'pynwb.base.ProcessingModule'>
  }
  session_description: session_description
  session_start_time: 1970-01-01 00:00:00-05:00
  timestamps_reference_time: 1970-01-01 00:00:00-05:00
```

### 12.1.1 Name generator

Two neurodata objects stored in the same location within an NWB file must have unique names. This can cause an error if you want to create a few neurodata objects with the same default name. To avoid this issue, each mock neurodata function uses the *name_generator* to generate unique names for each neurodata object. Consecutive neurodata objects of the same type will be named e.g. "TimeSeries", "TimeSeries2", "TimeSeries3", etc.

## 12.2 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.

### 12.2.1 NWBH5IOMixin

To write a roundtrip test, you will need to subclass the *NWBH5IOMixin* class and override some of its instance methods.

*NWBH5IOMixin* 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`.

#### setUpContainer

The first thing (and possibly the *only* thing – see *AcquisitionH5IOMixin*) 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 pynwb.testing import NWBH5IOMixin, TestCase


class TimeSeriesRoundTrip(NWBH5IOMixin, TestCase):

    def setUpContainer(self):
        return TimeSeries(
            "test_timeseries",
            "example_source",
            list(range(100, 200, 10)),
            "SIunit",
            timestamps=list(range(10)),
            resolution=0.1,
        )
```

### addContainer

The next thing is to tell the *NWBH5IOMixin* 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 additional 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(NWBH5IOMixin, TestCase):

    def addContainer(self, nwbfile):
        nwbfile.add_acquisition(self.container)
```

### getContainer

Finally, you need to tell *NWBH5IOMixin* 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(NWBH5IOMixin, TestCase):

    def getContainer(self, nwbfile):
        return nwbfile.get_acquisition(self.container.name)
```

### 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(NWBH5IOMixin, TestCase):

    def setUpBuilder(self):
        return GroupBuilder(
            'test_timeseries',
            attributes={
                'source': 'example_source',
                'namespace': base.CORE_NAMESPACE,
```

(continues on next page)

(continued from previous page)

```
                '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},
                )
            }
        )
```

### 12.2.2 AcquisitionH5IOMixin

If you are testing something that can go in *acquisition*, you can avoid writing `addContainer` and `getContainer` by extending *AcquisitionH5IOMixin*. 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

# THIRTEEN

# HOW TO MAKE A TUTORIAL

Tutorials are define using sphinx-gallery. The sources of tutorials are stored in `docs/gallery` with each subfolder corresponding to a subsection in the tutorial gallery.

## 13.1 Create a new tutorial

1. Add a new python file to the appropriate subfolder of `docs/gallery`.

---

**Hint:** If you want the output of code cells to be rendered in the docs, then simply include the prefix `plot_` as part of the name of tutorial file. Tutorials without the prefix will render just the code cells and text.

---

2. Add the names of any files created by running the python file to the `files_to_remove` variable in `clean_up_tests()` in `test.py`. This function will remove the created output files after the sphinx-gallery tests are completed.
2. **Optional:** To specify an explicit position for your tutorial in the subsection of the gallery, update the `GALLERY_ORDER` variable of the `CustomSphinxGallerySectionSortKey` class defined in `docs/source/conf.py`. If you skip this step, the tutorial will be added in alphabetical order.
3. Check that the docs are building correctly and fix any errors

```
cd docs
make html
```

4. View the docs to make sure your gallery renders correctly

```
open docs/_build/html/index.html
```

5. Make a PR to request addition of your tutorial

## 13.2 Create a new tutorial collection

To add a section to the tutorial gallery

1. Create a new folder in `docs/gallery/<new-section>`
2. Create a `docs/gallery/<new-section>/README.txt` file with the anchor name and title of the section, e.g.:

```
.. _my-new-tutorials:

My new tutorial section
-----------------------
```

3. Add tutorials to the section by adding the tutorial files to the new folder

CHAPTER

FOURTEEN

# HOW TO CONTRIBUTE TO NWB SOFTWARE AND DOCUMENTS

## 14.1 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.

## 14.2 Types of Contributions

### 14.2.1 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 format schema, schema language, storage, and other NWB related documents
  - Use PyNWB for any changes regarding the PyNWB API and the corresponding documentation
  - Use MatNWB for any changes regarding the MatNWB 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 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 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.

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

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

### 14.2.3 Do you have questions about NWB?

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

### 14.2.4 Informal discussions between developers and users?

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

## 14.3 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.

## 14.4 Issue Labels, Projects, and Milestones

### 14.4.1 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

### 14.4.2 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

### 14.4.3 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.

## 14.5 Style Guides

### 14.5.1 Git Commit Message Style Guide

- 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.

### 14.5.2 Documentation Style Guide

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

### 14.5.3 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.

### 14.5.4 Format Specification Style Guide

**Coming soon**

### 14.5.5 Python Code Style Guide

Before you create a Pull Request, make sure you are following the PyNWB style guide. To check whether your code conforms to the PyNWB style guide, simply run the ruff tool in the project's root directory. `ruff` will also sort imports automatically and check against additional code style rules.

We also use `ruff` to sort python imports automatically and double-check that the codebase conforms to PEP8 standards, while using the codespell tool to check spelling.

`ruff` and `codespell` are installed when you follow the developer installation instructions. See *Installing PyNWB for Developers*.

```
$ ruff check .
$ codespell
```

## 14.6 Endorsement

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

## 14.7 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

FIFTEEN

# COPYRIGHT

CHAPTER

SIXTEEN

# LICENSE

CHAPTER

SEVENTEEN

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## p

# INDEX

## Symbols



## A

## B

## C

## D

## E

## F



## G

## H



## I

## K



## L



## M

## N

## O

## P

## R

## S



## T

## U



## V



## W



## X