
NWB Overview

Release 0.1

Ben Dichter, Lawrence Niu, Ryan Ly, Oliver Ruebel

Apr 19, 2024

FOR USERS

1	Intro to NWB	3
2	Converting neurophysiology data to NWB	9
3	Reading NWB Files	13
4	Extending NWB	21
5	Glossary of Core NWB Tools	43
6	Analysis and Visualization Tools	49
7	Acquisition and Control Tools	77
8	Community Gallery	85
9	Frequently Asked Questions	87
10	Accessing NWB Sources	91
11	NWB Software Analytics	93
12	Indices and tables	153
	Python Module Index	155
	Index	157

This website is an entry point for researchers and developers interested in using [NWB](#). If you are a new NWB user and want to learn about the different tools available to convert your data to NWB, publish your NWB data, and visualize and analyze NWB data, then you are in the right place! These pages will guide you through the main workflow for each of those tasks and point you to the best tools to use for your preferred programming language and types of data.

INTRO TO NWB

Neurodata Without Borders (NWB) provides a common self-describing format ideal for archiving neurophysiology data and sharing it with colleagues. NWB provides a standardized schema – a set of rules and best practices – to help neuroscientists package their data and metadata together so that they are both machine- and human-readable.

Storing your neurophysiology data in the NWB format is beneficial to you and the broader scientific community in several important ways. First, it helps you easily use popular data processing, analysis, and visualization tools. Second, it helps both you and other scientists reuse the data to gain additional scientific insights. Finally, converting your data to NWB is a critical step for getting your data in the [DANDI Archive](#) to share it with collaborators and the public.

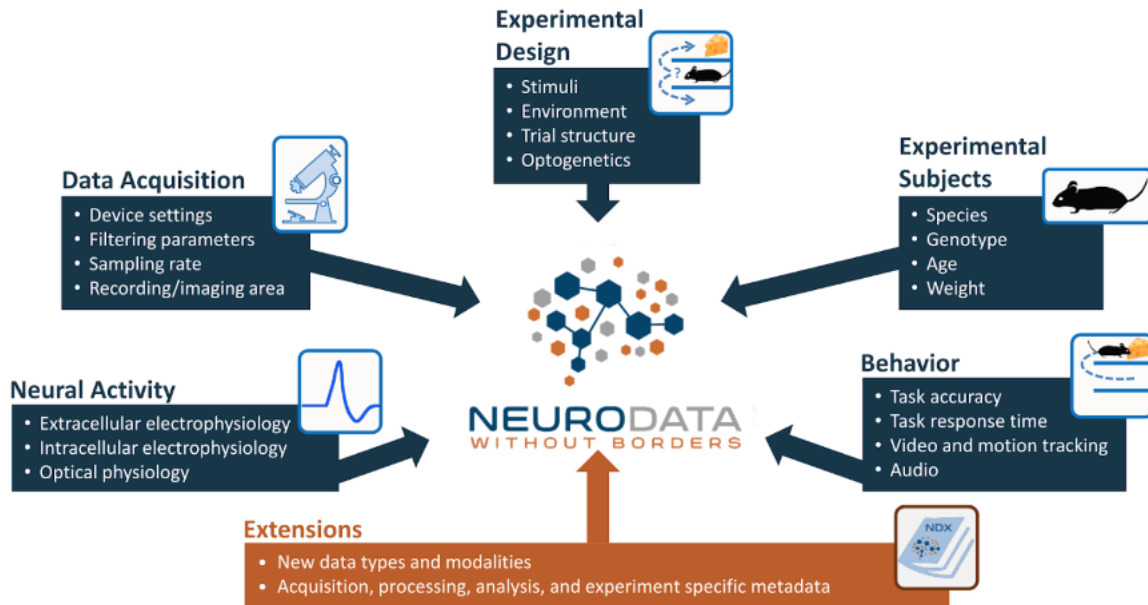
We understand that using a different data format than you are accustomed to using can be daunting. This guide will walk you through what NWB is, how to use NWB tools to convert your data to NWB, and how to read and interact with your data (and any other data) in the NWB format.

1.1 Anatomy of an NWB File

First, we need to understand a few basic concepts of how an NWB file is structured.

The goal of NWB is to package all of the data and metadata of a particular session into a single file in a standardized way. This includes the neurophysiology data itself, but also includes other data such as information about the data acquisition, experiment design, experimental subject, and behavior of that subject. The NWB schema defines data structures for the most common types of neurophysiology data, such as:

- extracellular electrophysiology (e.g., Neuropixel probes)
- intracellular electrophysiology (e.g., patch clamping)
- optical physiology (e.g., two-photon imaging)



An NWB file is an [HDF5](#) file that is structured in a particular way and has the *.nwb* file extension. HDF5 files use a files-and-folders-like structure that allows you to organize your data in a folder hierarchy, as you might do with files on your computer. You can also embed metadata within the file, which allows the file to be self-describing and easy for both humans and machines to understand.

If you inspect an NWB file, you would see something like the following:

```

└─ sub-npI1_ses-20190413_behavior+ecephys.nwb
    └─ acquisition
        └─ ElectricalSeries # raw voltage recording over time
    └─ general
        └─ devices # device metadata
        └─ extracellular_ephys
            └─ Shank1 # electrode group metadata
            └─ electrodes # per-electrode metadata
        └─ institution
        └─ lab
        └─ subject
    └─ intervals
        └─ trials # per-trial metadata
    └─ processing
        └─ behavior
            └─ Position
                └─ position # computed x,y position over time
            └─ licks # timestamps of detected lick events
        └─ ecephys
            └─ LFP
                └─ LFP # computed LFP over time
    └─ session_description
    └─ session_start_time

```

At a glance, you can see that there are objects within the file that represent acquired (raw) voltage data and processed position, lick event, and LFP data. There are also fields that represent metadata about the devices and electrodes used, the institution and lab where the data was collected, subject information, per-trial metadata, a description of the session,

and the session start time.

An NWB file often consists of many different data and metadata so as to provide a complete representation of the data that was collected in a session. This may seem overwhelming at first, but NWB provides tools to make it easy for you to read and write data in the NWB format, as well as inspect and visualize your NWB data. Software tools leverage the standardized structure of an NWB file to find useful data and metadata for processing, analysis, and visualization. And once you understand the structure of an NWB file, you can open any NWB file and quickly make sense of its contents.

1.1.1 File hierarchy

NWB organizes data into different internal groups (i.e., folders) depending on the type of data. Here are some of the groups within an NWB file 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)

Raw data

Raw data are stored in containers placed in the `acquisition` group at the root of the NWB file. Raw data should never change.

Processed data

Processed data, such as the extracted fluorescence traces from calcium imaging ROIs, are stored in containers placed within a modality-specific `ProcessingModule` container in the `processing` group at the root of the NWB file. Any scripts or software that process raw data should store the results within a `ProcessingModule`.

NWB organizes processed data into `ProcessingModule` containers with specific names based on the type of processed data:

Type of processed data	Name of <code>ProcessingModule</code>
extracellular electrophysiology	“ecephys”
intracellular electrophysiology	“icephys”
optical physiology	“ophys”
behavior	“behavior”

See the next section to learn about neurodata types in NWB.

1.2 Neurodata Types

The precise rules for how to store different types of data and the associated metadata are defined using **neurodata types**. Different types of data have different neurodata types. Together, these neurodata type definitions form the **NWB schema**. The NWB schema defines over 60 different neurodata types, which cover the most common data types in extracellular electrophysiology, intracellular electrophysiology, optical physiology, and behavior.

Neurodata types act like classes in Object-Oriented Programming in that they can use inheritance (a neurodata type can be a specialized child of another neurodata type) and composition (a neurodata type can contain other neurodata types).

1.2.1 TimeSeries

The **TimeSeries** neurodata type defines how generic data values that change over time should be stored. In particular, the *TimeSeries* type must contain a description, an N-dimensional “data” dataset (array) where the first dimension is time, and either a 1-dimensional “timestamps” dataset or both a sampling rate and starting time.

The **TimeSeries** neurodata type is the base type for many other neurodata types, such as the **ElectricalSeries** for extracellular electrophysiology. *ElectricalSeries* inherits all of the properties of *TimeSeries* but additionally specifies that the “data” dataset must be 2-dimensional, where the second dimension is electrode, and it must contain an additional “electrodes” dataset with row indices into the “electrodes” table of the NWB file to indicate which electrodes correspond to the second dimension of the “data” dataset.

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**.

All time values in a *TimeSeries* and other neurodata types must be stored in seconds relative to the `timestamps_reference_time` value, a datetime value stored at the root of the NWB file. By default, this is the same as the `session_start_time`, another datetime value stored at the root of the NWB file.

1.2.2 DynamicTable

Tabular (table-like) data are stored in **DynamicTable** objects, which are column-based tables to which you can add custom columns.

Similar to *TimeSeries*, several neurodata types inherit from *DynamicTable* that are specialized for particular types of data and specify particular required or optional columns.

See also:

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

- **TimeIntervals:** stores trials, epochs, and associated metadata.
- **Units:** stores spike times of sorted units and associated metadata.

- [PlaneSegmentation](#): stores regions of interest for optical imaging with associated metadata.
- Several types for storing the structure of intracellular electrophysiology experiments ([IntracellularRecordingsTable](#), [SimultaneousRecordingsTable](#), [SequentialRecordingsTable](#), [RepetitionsTable](#), [ExperimentalConditionsTable](#))

NWB is faced with the challenge of supporting a large variety of different experiment types, so the data types and relationships can get quite complex. For this reason, the NWB development team provides software to help users easily and efficiently read and write NWB files. These software packages are described in the next section.

1.3 Software for reading and writing NWB data

NWB provides two APIs for easily reading and writing NWB data: [PyNWB](#) for Python and [MatNWB](#) for MATLAB.

Each neurodata type defined in the NWB schema has a corresponding class in both PyNWB and MatNWB. Like neurodata types, these classes relate to each other through inheritance and composition. For example, the PyNWB class [TimeSeries](#) serves as the base class (superclass) for more specialized time series classes, such as, [ElectricalSeries](#), and a [LFP](#) object can contain one or more [ElectricalSeries](#) objects, just like how the corresponding neurodata types are defined in the NWB schema.

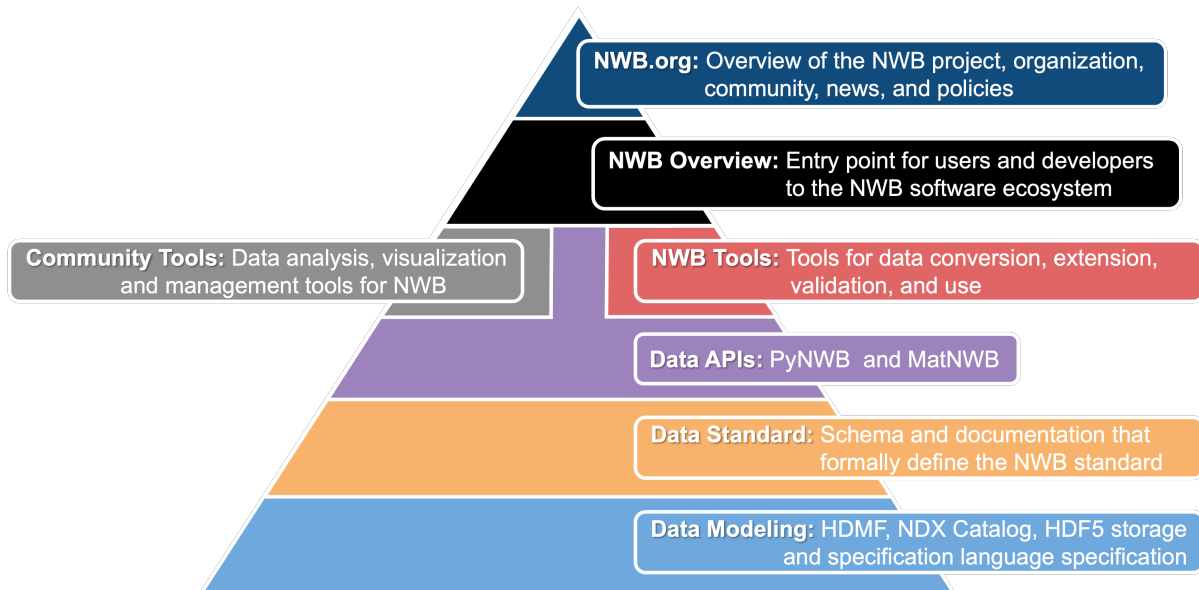
1.4 Next steps

To learn more about:

- how to install and use the APIs, see the [PyNWB](#) and [MatNWB](#) documentation
- how to convert your data to NWB, see [Converting neurophysiology data to NWB](#).
- the broad range of core tools available for NWB see the [Glossary of Core NWB Tools](#)
- community tools available for NWB see the [Analysis and Visualization Tools](#) section.
- the NWB schema, please see the [NWB Schema](#) documentation.

1.4.1 Navigating the NWB Documentation

NWB defines a large software ecosystem and as such there are many different documentation resources available depending on what aspect of NWB and what type of information you want to learn more about. The following sketch provides an overview of the main areas of the NWB documentation. As we navigate from the top of the pyramid to the bottom, the level of detail and depth of the documentation increases. Most end-users will typically only interact with the top half of the pyramid. If you are new to NWB and want to learn more; good news you are at the right place!



At the top (dark blue), we have the [NWB.org](#) website, which focuses on the NWB project at large and provides information about the overall organization, goals, community, and policies of the NWB project and provides a first entry point to NWB.

Next we have the *NWB Overview* (i.e., this website) (black), which serves as an entry point for researchers and developers interested in using NWB and in learning more about the NWB software ecosystem.

A growing collection of *Analysis and Visualization Tools* (gray) developed by the neurophysiology community along with a broad range of *core tools* (red) help users with conversion, extension, validation and use of NWB data files. Commonly, these tools build on the [PyNWB](#) (Python) and [MatNWB](#) (Matlab) reference APIs (lilac) to read and write NWB files. Each of the APIs and tools provides their own in-depth tutorials and documentation to help guide users in adopting and using the software.

Underlying all of this, is the [NWB Format Specification](#), which formally defines and governs the NWB data standard.

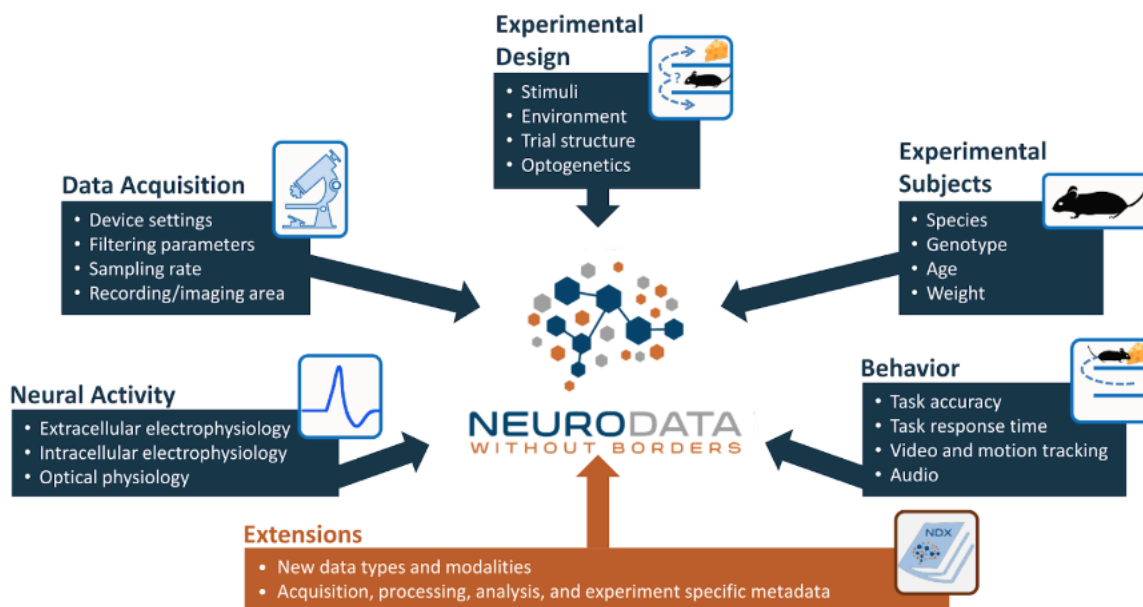
Last but not least, NWB provides and builds on a broad range of data modeling tools and technologies (light blue), e.g., [HDMF](#) and the [HDMF Common Schema](#), the [Neurodata Extension Catalog](#), as well as the [specification language](#) and [data storage](#) specifications.

CONVERTING NEUROPHYSIOLOGY DATA TO NWB

We understand converting to NWB can be daunting. This guide will walk you through the tools available to help you convert your neurophysiology data to NWB. We will start with an overview of NWB and the Python and MATLAB APIs maintained by the core development team, then discuss automated tools for converting formats.

2.1 1. NWB Conversions

The goal of NWB is to package experiment data with the metadata necessary to analyze the data. All of the data in a particular session goes into a single file. This includes the neurophysiology data itself, but also includes other data such as information about the data acquisition, experiment design, experimental subject, and behavior of that subject. The NWB core schema defines data containers for common data objects in neurophysiology data, including: intracellular electrophysiology (e.g. patch clamping), extracellular electrophysiology (e.g. Neuropixel probes), and optical physiology (e.g. two-photon imaging), and behavior.



All of these data types and relationships are defined in the [NWB Schema](#) using the [HDMF specification language](#). NWB is faced with the challenge of supporting a large variety of different experiment types, so the data types and relationships can get quite complex. For this reason the NWB development team provides APIs to help users easily and efficiently read and write NWB files.

The following sections start with the most automated and convenient approaches and proceed to more involved and customizable solutions.

2.2 2. Automatic NWB Conversions using NeuroConv

[NeuroConv](#) is a library for automatic conversions from proprietary formats to NWB. A gallery of all supported formats can be found [here](#). If NeuroConv supports your source data format, this is the recommended approach, because it is easiest to implement and automatically helps you adhere to best practices. For advanced usage of NeuroConv, including creating conversions of ensembles of multiple data streams, see the [NeuroConv User Guide](#).

Although NeuroConv supports many common formats, it may not support every type of source data you need. If your source format is likely to be a common need in the community, for example the output of an acquisition system or a popular data processing package, please use [this form](#) to request support for this format. On the other hand, some data formats can be ad-hoc, particularly data about task and behavior. In this case, proceed to the next section for detailed tutorials on using the PyNWB API to manually add this data to the NWB file.

2.3 3. NWB APIs: PyNWB & MatNWB

The NWB team develops and supports APIs in Python ([PyNWB](#)) and MATLAB ([MatNWB](#)), which guide users in reading and writing NWB files. The features of these two APIs are very similar to each other, so choose whichever is more convenient. Note that most of the high-level conversion tools use PyNWB, but the APIs are interchangeable and there should be no problem with writing data using MatNWB even if you plan to read it using PyNWB later (or vice versa). Below is a table of the available tutorials illustrating the end-to-end conversion of common data types in each imaging modality:

	PyNWB	MatNWB
Reading NWB files	tutorial	tutorial 15 min video
Writing extracellular electrophysiology	tutorial 23 min video	tutorial 46 min video
Writing intracellular electrophysiology	tutorial	tutorial
Writing calcium imaging	tutorial 31 min video	tutorial 39 min video
Advanced I/O	tutorials 26 min video	tutorial 16 min video

These tutorials walk you through the most common data types of each of the modalities. With the tutorials for domain-specific conversion, extensions, advanced I/O, and the documentation for proprietary formats, you have all of the tools to create your own conversion. However, writing a full-fledged conversion script from the ground up with MatNWB and PyNWB is time-intensive, error-prone, and requires detailed knowledge of the source format. That is why, in the next section, we will introduce automated conversion tools that work on a large number of supported proprietary formats.

2.4 4. Using Neurodata Extensions

Neurophysiology is always changing as new technologies are developed. While the core NWB schema supports many of the most common data types in neurophysiology, we need a way to accommodate new technologies and unique metadata needs. Neurodata extensions (NDX) allow us to define new data types. These data types can extend core types, contain core types, or can be entirely new. These extensions are formally defined with a collection of YAML files following the [NWB Specification Language](#).

If the core nwb schema does not meet your needs, you will need to use an extension. Your first step should be to search the [NDX Catalog](#). Using an existing extension is preferred, because it saves you the work of creating a new one, and it improves data standardization across the community. If you do not find an extension that meets your needs, head to the [Extending NWB](#) tutorial. When creating a new extension you should also consider to let the community know by adding a post on the [NWB Help Desk](#).

2.5 5. Validation of NWB files

After you create NWB files, it is a good idea to inspect them to ensure that they are representing your data correctly. There are several complementary tools that can help you ensure you are utilizing NWB optimally.

2.5.1 NWB Validation

Once you create an NWB file, it is a good idea to validate it against the NWB schema. PyNWB comes with a validator that you can run from the command line:

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

see [Validating NWB files](#) for details

2.5.2 NWB Inspector

[NWB Inspector](#) inspects NWB files for compliance with [NWB Best Practices](#) and attempts to find mistakes in conversion. This inspector is meant as a companion to the PyNWB validator, which checks for strict schema compliance. In contrast, this tool attempts to apply some common sense rules to find data components of a file that are technically compliant, but probably incorrect, or suboptimal, or deviate from best practices. I.e., while the PyNWB validator focuses on compliance of the structure of a file with the schema, the inspector focuses on compliance of the actual data with best practices. The NWB Inspector is meant simply as a data review aid. It does not catch all best practice violations, and any warnings it does produce should be checked by a knowledgeable reviewer.

2.5.3 HDFView

[HDFView](#) is a visual tool written by the HDF Group in Java for browsing and editing HDF (HDF5 and HDF4) files. With HDFView, you can open NWB files and inspect their contents manually. HDFView can show you detailed information such as chunking and compression settings ratio achieved for each dataset.

2.5.4 NWB Widgets

[NWB Widgets](#) is a library of widgets for visualization NWB data in a Jupyter notebook (or lab). The widgets allow you to navigate through the hierarchical structure of the NWB file and visualize specific data elements. It is designed to work out-of-the-box with NWB 2.0 files. Using NWB Widgets, you can explore the data in your NWB file and generate simple figures to ensure that your data is represented correctly.

2.6 6. Publishing NWB on DANDI Archive

Once you have completed the conversion of all the data in your dataset, we encourage you to publish this data. We recommend the [DANDI archive](#). DANDI can handle large volumes (TBs) of data and host them for free, and provides a command-line interface (CLI) that is built to handle this volume of data. DANDI also automatically parses NWB files to extract metadata that makes these datasets easier for others to find.

Refer to the [DANDI documentation](#) for how to upload your dataset.

Note: DANDI requires each NWB file to have a *Subject* field with *subject_id* defined. It is possible to create a valid NWB file without this field, but it will not be accepted by DANDI.

If you experience any problems or questions, the [DANDI helpdesk](#) is the best place to ask for help.

READING NWB FILES

This section provides an introduction on how to read NWB files in Python using PyNWB and Matlab using MatNWB. If you are interested in converting your data to NWB, then please see the [User Guide](#) as well as for more in-depth tutorials visit the [PyNWB](#) and [MatNWB](#) tutorials and documentation.

3.1 Reading With PyNWB

- See the PyNWB tutorial on [Reading and Exploring an NWB File](#) for an introduction to how to read, explore, and do basic visualizations with an NWB file in Python.
- The [Query Intracellular Electrophysiology Metadata](#) PyNWB tutorial then focuses specifically on how to read and query metadata related to intracellular electrophysiology data.

3.2 Reading with MatNWB

For most files, MatNWB only requires the `nwbRead` call:

```
nwb = nwbRead('path/to/filename.nwb');
```

This call will read the file, create the necessary NWB schema class files, as well as any extension schemata that is needed for the file itself. This is because both PyNWB and MatNWB embed a copy of the schema environment into the NWB file when it is written.

The returned object above is an `NwbFile` which serves as the root object with which you can use to browse the contents of the file. More detail about the `NwbFile` class can be found here: [Using the NwbFile Class](#).

3.2.1 Using the NwbFile Class

The `NwbFile` class represents the root object for the NWB file and consists of properties and values which map indirectly to the internal HDF5 dataset.

```

Command Window
>> nwb = NwbFile
nwb =
  NwbFile with properties:
      nwb_version: '2.2.2'
      file_create_date: []
      general_source_script_file_name: []
      identifier: []
      session_description: []
      session_start_time: []
      timestamps_reference_time: []
      acquisition: [0x1 types.untyped.Set]
      analysis: [0x1 types.untyped.Set]
      general: [0x1 types.untyped.Set]
      general_data_collection: []
      general_devices: [0x1 types.untyped.Set]
      general_experiment_description: []
      general_experimenter: []
      general_extracellular_ephys: [0x1 types.untyped.Set]
      general_extracellular_ephys_electrodes: []
      general_institution: []
      general_intracellular_ephys: [0x1 types.untyped.Set]
      general_intracellular_ephys_filtering: []
      general_intracellular_ephys_sweep_table: []
      general_keywords: []
      general_lab: []
      general_notes: []
      general_optogenetics: [0x1 types.untyped.Set]
      general_optophysiology: [0x1 types.untyped.Set]
      general_pharmacology: []
      general_protocol: []
      general_related_publications: []
      general_session_id: []
      general_slices: []
      general_source_script: []
      general_stimulus: []
      general_subject: []
      general_surgery: []
      general_virus: []
      intervals: [0x1 types.untyped.Set]
      intervals_epochs: []
      intervals_invalid_times: []
      intervals_trials: []
      processing: [0x1 types.untyped.Set]
      scratch: [0x1 types.untyped.Set]
      stimulus_presentation: [0x1 types.untyped.Set]
      stimulus_templates: [0x1 types.untyped.Set]
      units: []
fx >> |

```

In most cases, the types contained in these files were generated by the embedded NWB schema in the file (or separately if you opted to generate them separately). These types can be traversed using regular MATLAB syntax for accessing

properties and their values.

Aside from the generated Core and Extension types, there are “Untyped” utility Types which are covered in greater detail in *Utility Types in MatNWB*.

Searching by Type

The NwbFile also allows for searching the entire NWB file by type using its `searchFor` method.

You can search for only the class name:

```

Command Window
>> nwb.searchFor('OpticalSeries')
ans =
  Map with properties:
    Count: 1
    KeyType: char
    ValueType: any
>> opticalSeriesMap = nwb.searchFor('OpticalSeries');
>> opticalSeriesMap.keys()
ans =
  1x1 cell array
    {'/stimulus_presentation/StimulusPresentation_encoding'}
>> opticalSeriesMap('/stimulus_presentation/StimulusPresentation_encoding')
ans =
  OpticalSeries with properties:
    distance: 0.7000000000000000
    field_of_view: [1x1 types.untyped.DataStub]
    orientation: 'lower left'
    external_file_starting_frame: 0
    device: []
    dimension: [1x1 types.untyped.DataStub]
    external_file: [1x1 types.untyped.DataStub]
    format: 'external'
    starting_time_unit: 'seconds'
    timestamps_interval: 1
    timestamps_unit: 'seconds'
    data: [1x1 types.untyped.DataStub]
    data_unit: 'meters'
    starting_time_rate: []
    comments: 'no comments'
    control: []
    control_description: []
    data_continuity: []
    data_conversion: 1
    data_resolution: -1
    description: 'no description'
    starting_time: []
    timestamps: [1x1 types.untyped.DataStub]
  
```

Or use the 'includeSubClasses' optional argument to search all subclasses:

```

Command Window
>> tsSubclassMap = nwb.searchFor('TimeSeries', 'includeSubClasses');
>> tsSubclassPaths = tsSubclassMap.keys() .
tsSubclassPaths =
    3×1 cell array
    {'/acquisition/events'
    {'/acquisition/experiment_ids'
    {'/stimulus_presentation/StimulusPresentation_encoding'}
>> nwb.resolve(tsSubclassPaths{3})
ans =
    OpticalSeries with properties:

        distance: 0.7000000000000000
        field_of_view: [1×1 types.untyped.DataStub]
        orientation: 'lower left'
    external_file_starting_frame: 0
        device: []
        dimension: [1×1 types.untyped.DataStub]
        external_file: [1×1 types.untyped.DataStub]
        format: 'external'
    starting_time_unit: 'seconds'
    timestamps_interval: 1
        timestamps_unit: 'seconds'
        data: [1×1 types.untyped.DataStub]
        data_unit: 'meters'
    starting_time_rate: []
        comments: 'no comments'
        control: []
    control_description: []
        data_continuity: []
        data_conversion: 1
        data_resolution: -1
        description: 'no description'
        starting_time: []
        timestamps: [1×1 types.untyped.DataStub]
fx >> |

```

Note: As seen above, the keys to `searchFor` method's returned Map can be paired with the `resolve` method to effectively retrieve an object from any NwbFile. This is also true for internal HDF5 paths.

3.2.2 The Dynamic Table Type

The `Dynamic Table` is a special NWB type composed of multiple `VectorData` objects which act like the columns of a table. The benefits of this type composition is that it allows us to add user-defined columns to a `DynamicTable` without having to extend the NWB data schema and each column can be accessed independently using regular NWB syntax and modified as a regular dataset.

With such an object hierarchy, however, there is no easy way to view the `Dynamic Table` data row by row. This is where `getRow` comes in.

Row-by-row viewing

`getRow` retrieves one or more rows of the dynamic table and produces a MATLAB `table` with a representation of the data. It should be noted that this returned table object is **readonly** and any changes to the returned table will not be reflected back into the NWB file.

By default, you must provide the row index as an argument. This is 1-indexed and follows MATLAB indexing behavior.

```
tableData = dynamicTable.getRow(<tableIndex>);
```

You can also filter your view by specifying what columns you wish to see.

```
filteredData = dynamicTable.getRow(<tableIndex>, 'columns', {'columnName'});
```

In the above example, the `filteredData` table will only have the “columnName” column data loaded.

Finally, if you prefer to select using your custom id column, you can specify by setting the `useId` keyword.

```
tableData = dynamicTable.getRow(<idValue>, 'useId', true);
```

For more information regarding `Dynamic Tables` in `MatNWB` as well as information regarding writing data to them, please see the [MatNWB DynamicTables Tutorial](#).

3.2.3 Utility Types in MatNWB

“Untyped” Utility types are tools which allow for both flexibility as well as limiting certain constraints that are imposed by the NWB schema. These types are commonly stored in the `+types/+untyped/` package directories in your `MatNWB` installation.

Sets and Anons

The `Set` (`types.untyped.Set` or `Constrained Sets`) is used to capture a dynamic number of particular NWB-typed objects. They may contain certain type constraints on what types are allowable to be set. Set keys and values can be set and retrieved using their `set` and `get` methods:

```
value = someSet.get('key name');
```

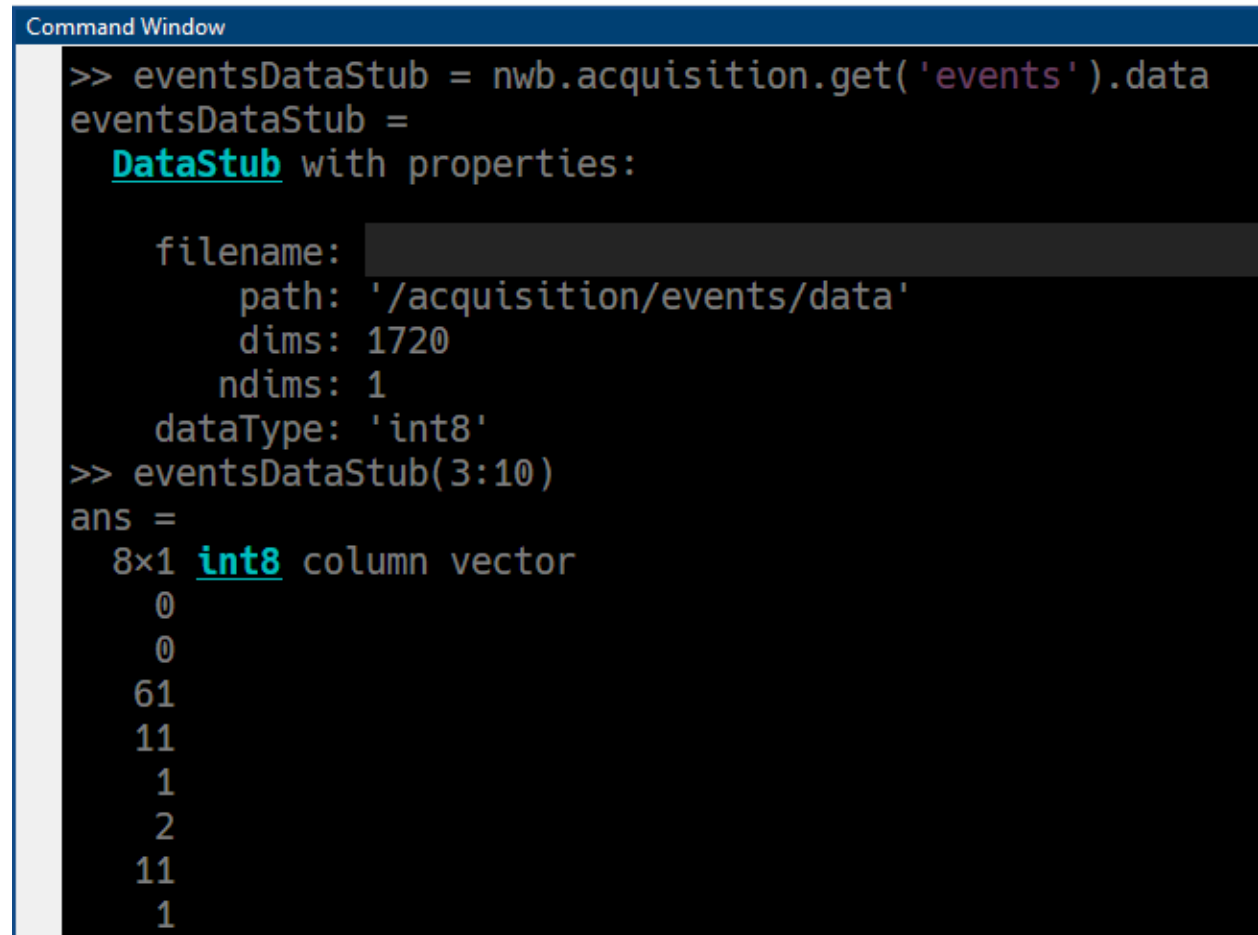
```
someSet.set('key name', value);
```

Note: Sets also borrow `containers.Map`’s `keys` and `values` methods to retrieve cell arrays of either.

The **Anon** type (`types.untyped.Anon`) can be understood as a Set type with only a single key-value entry. This rarer type is only used for cases where the name for the stored object can be set by the user. Anon types may also hold NWB type constraints like Set.

DataStubs and DataPipes

DataStubs serves as a read-only link to your data. It allows for MATLAB-style indexing to retrieve the data stored on disk.



```
Command Window
>> eventsDataStub = nwb.acquisition.get('events').data
eventsDataStub =
  DataStub with properties:

    filename: 
    path: '/acquisition/events/data'
    dims: 1720
    ndims: 1
    dataType: 'int8'
>> eventsDataStub(3:10)
ans =
  8x1 int8 column vector
    0
    0
   61
   11
    1
    2
   11
    1
```

DataPipes are similar to DataStubs in that they allow you to load data from disk; however, they also provide a wide array of features that allow the user to write data to disk, either by streaming parts of data in at a time or by compressing the data before writing. The DataPipe is an advanced type and users looking to leverage DataPipe's capabilities to stream/iteratively write or compress data should read the [Advanced Data Write Tutorial](#).

Links and Views

Links (either `types.untyped.SoftLink` or `types.untyped.ExternalLink`) are views that point to another NWB object, either within the same file or in another external one. *SoftLinks* contain a path into the same NWB file while *ExternalLinks* additionally hold a `filename` field to point to an external NWB file. Both types use their `deref` methods to retrieve the NWB object that they point to though *SoftLinks* require the `NwbFile` object that was read in.

```
referencedObject = softLink.deref(rootNwbFile);
```

```
referencedObject = externalLink.deref();
```

Note: Links are not validated on write by default. It is entirely possible that a link will simply never resolve, either because the path to the NWB object is wrong, or because the external file is simply missing from the NWB distribution.

Views (either `types.untyped.ObjectView` or `types.untyped.RegionView`) are more advanced references which can point to NWB types as well as segments of raw data from a dataset. *ObjectViews* will point to NWB types while *RegionViews* will point to some subset of data. Both types use `refresh` to retrieve their referenced data.

```
referencedObject = objectView.refresh(rootNwbFile);
```

```
dataSubset = regionView.refresh(rootNwbFile);
```

Note: Unlike *Links*, Views cannot point to NWB objects outside of their respective files. Views are also validated on write and will always point to a valid NWB object or raw data if written without errors.

3.2.4 Troubleshooting File Reads in MatNWB

Outlined below are the most common issues reported by users when they read a NWB file as well as common troubleshooting approaches to resolve them.

Schema Version Conflicts

If you run into an error where reading a file appears to expect the wrong properties, you should first check if your MATLAB path is not pointing to other environments with the same packages. MATLAB's internal `which` command to check for unexpected class locations.

Multiple Schema Environments

By default, MatNWB generates all class files in its own installation directory. However, if your work requires you to manipulate multiple different schema versions or extension environments, you may want to generate the class files in a local directory so that MATLAB will default to those classes instead.

To do this, you can use the optional `savdir` keyword argument with `nwbRead` which allows you to specify a directory location within which your generated class files will be saved.

```
nwb = nwbRead('/path/to/matnwb/file.nwb', 'savdir', '.'); % write class files to
↳ current working directory.
```

Note: Other generation functions `generateCore` and `generateExtension` also support the `savendir` option.

Missing Embedded Schemata

Some older NWB files do not have an embedded schema. To read from these files you will need the API generation functions `generateCore` and `generateExtension` to generate the class files before calling `nwbRead` on them. You can also use the utility function `util.getSchemaVersion` to retrieve the correct Core schema for the file you are trying to read:

```
schemaVersion = util.getSchemaVersion('/path/to/matnwb/file.nwb');
generateCore(schemaVersion);
generateExtension(path/to/extension/namespace.yaml);
```

Avoiding Class Regeneration

If you wish to read your file multiple times, you may not want to regenerate your files every time since you know that your current environment is correct. For this case, you can use `nwbRead`'s optional argument `ignorecache` which will ignore the embedded schema and attempt to read your file with the class files accessible from your MATLAB path or current working directory.

```
nwb = nwbRead('path/to/matnwb/file.nwb', 'ignorecache');
```

Bottom of the Barrel

If you're here, you've probably reached your wit's end and wish for more specific help. In such times, you can always contact the NWB team either as a message on our [NWB HelpDesk](#), [Slack Workspace](#) or as an issue on [Github](#).

EXTENDING NWB

Neurophysiology is always changing as new technologies are developed. While the core NWB schema supports many of the most common data types in neurophysiology, we need a way to accommodate new technologies and unique metadata needs. Neurodata extensions (NDX) allow us to define new data types. These data types can extend core types, contain core types, or can be entirely new. These extensions are formally defined with a collection of YAML files following the [NWB Specification Language](#).

4.1 Creating an extension

4.1.1 Using ndx-template

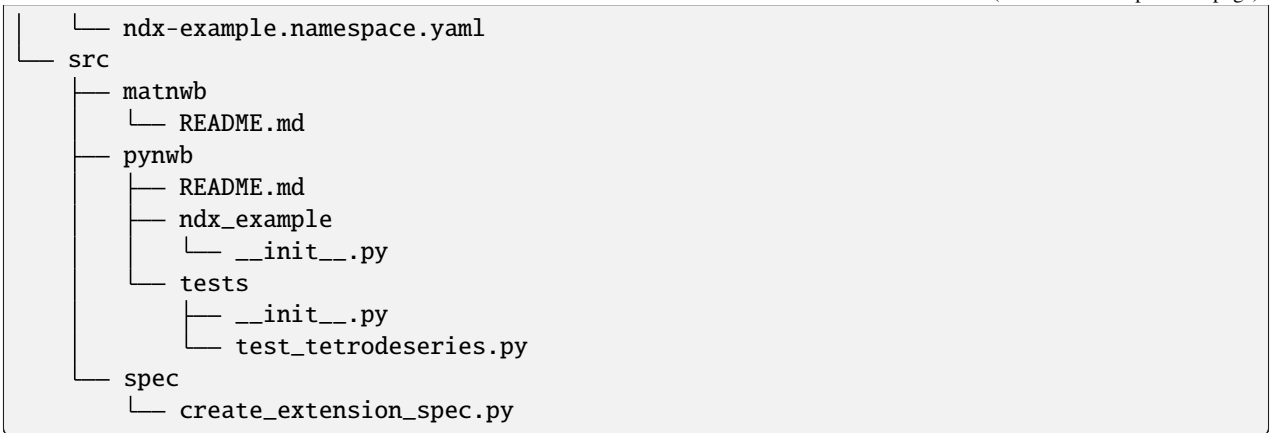
Extensions should be created in their own repository, not alongside data conversion code. This facilitates sharing and editing of the extension separately from the code that uses it. When starting a new extension, we highly recommend using the [ndx-template](#) repository, which automatically generates a repository with the appropriate directory structure.

After you finish the instructions [here](#), you should have a directory structure that looks like this

```
— LICENSE.txt
— MANIFEST.in
— NEXTSTEPS.md
— README.md
— docs
  — Makefile
  — README.md
  — make.bat
  — source
    — _static
      — theme_overrides.css
    — conf.py
    — conf_doc_autogen.py
    — credits.rst
    — description.rst
    — format.rst
    — index.rst
    — release_notes.rst
— requirements.txt
— setup.cfg
— setup.py
— spec
  — ndx-example.extensions.yaml
```

(continues on next page)

(continued from previous page)



At its core, an NWB extension consists of YAML text files, such as those generated in the `spec` folder. While you can write these YAML extension files by hand, PyNWB provides a convenient API via the `spec` module for creating extensions.

Open `src/spec/create_extension_spec.py`. You will be modifying this script to create your own NWB extension. Let's first walk through each piece.

4.1.2 Creating a namespace

NWB organizes types into namespaces. You must define a new namespace before creating any new types. After following the instructions from the `ndx-template`, you should have a file `ndx-my-ext/src/spec/create_extension_spec.py`. The beginning of this file should look like

```

from pynwb.spec import NWBNamespaceBuilder, export_spec, NWBGroupSpec, NWBAttributeSpec
# TODO: import the following spec classes as needed
# from pynwb.spec import NWBDataSetSpec, NWBLinkSpec, NWBDtypeSpec, NWBRefSpec

def main():
    # these arguments were auto-generated from your cookiecutter inputs
    ns_builder = NWBNamespaceBuilder(
        doc="my description",
        name="ndx-my-ext",
        version="0.1.0",
        author="John Doe",
        contact="contact@gmail.com"
    )

```

Here, after the initial imports, we are defining meta-data of the extension. Pay particular attention to `version`. If you make changes to your extension after the initial release, you should increase the numbers in your version number, so that you can keep track of what exact version of the extension was used for each file. We recommend using a [semantic versioning approach](#).

4.1.3 Including types

Next, we need to include types from the core schemas. This is analogous to importing classes in Python. The generated file includes some example imports.

```
ns_builder.include_type('ElectricalSeries', namespace='core')
ns_builder.include_type('TimeSeries', namespace='core')
ns_builder.include_type('NWBDataInterface', namespace='core')
ns_builder.include_type('NWBContainer', namespace='core')
ns_builder.include_type('DynamicTableRegion', namespace='hdmf-common')
ns_builder.include_type('VectorData', namespace='hdmf-common')
ns_builder.include_type('Data', namespace='hdmf-common')
```

Neuroscience-specific data types are defined in the namespace 'core' (which means core NWB). More general organizational data types that are not specific to neuroscience and are relevant across scientific fields are defined in 'hdmf-common'. You can see which types are defined in which namespace by exploring the [NWB schema documentation](https://hdmf-common-schema.readthedocs.io/en/latest/) and hdmf-common schema documentation [<https://hdmf-common-schema.readthedocs.io/en/latest/>](https://hdmf-common-schema.readthedocs.io/en/latest/) `._`.

4.1.4 Defining new neurodata types

Next, the `create_extension_spec.py` file declares an example extension for a new neurodata type called `TetrodeSeries`, which extends the `ElectricalSeries` type. Then it creates a list of all new data types.

```
tetrode_series = NWBGroupSpec(
    neurodata_type_def='TetrodeSeries',
    neurodata_type_inc='ElectricalSeries',
    doc=('An extension of ElectricalSeries to include the tetrode ID for '
        'each time series.'),
    attributes=[
        NWBAttributeSpec(
            name='tetrode_id',
            doc='The tetrode ID.',
            dtype='int32'
        )
    ],
)

# TODO: add all of your new data types to this list
new_data_types = [tetrode_series]
```

The remainder of the file is to generate the YAML files from the spec definition, and should not be changed.

After you make changes to this file, you should run it to re-generate the `ndx-[name].extensions.yaml` and `ndx-[name].namespace.yaml` files. In the next section, we will go into more detail into how to create neurodata types.

4.2 The Spec API

pynwb defines a spec API, which are classes to help generate a valid NWB extension. The [NWB Specification Language](#) defines a structure for data and metadata using Groups, Datasets, Attributes, and Links. These structures are mapped onto [NWBGroupSpec](#), [NWBDatasetSpec](#), [NWBAAttributeSpec](#), and [NWBLinkSpec](#), respectively. Here, we describe in detail each of these classes, and demonstrate how to use them to create custom neurodata types.

4.2.1 Group Specifications

Most neurodata types are Groups, which act like a directory or folder within the NWB file. A Group can have within it Datasets, Attributes, Links, and/or other Groups. Groups are specified with the [NWBGroupSpec](#) class, which provides a python API for specifying the structure for an [NWB Group](#).

```
from pynwb.spec import NWBGroupSpec

spec = NWBGroupSpec(
    neurodata_type_def='MyType',
    neurodata_type_inc='NWBDataInterface',
    doc='A custom NWB type',
    name='quux',
    attributes=[...],
    datasets=[...],
    groups=[...],
    links=[...]
)
```

`neurodata_type_def` and `neurodata_type_inc` define the neurodata type with the following rules:

- `neurodata_type_def` declares the name of the neurodata type.
- `neurodata_type_inc` indicates what data type you are extending (Groups must extend Groups, and Datasets must extend Datasets).
- To define a new neurodata type that does not extend an existing type, use `neurodata_type_inc=NWBContainer` for a group or `neurodata_type_inc=NWBData` for a dataset. `NWBContainer` and `NWBData` are base types for NWB.
- To use a type that has already been defined, use `neurodata_type_inc` and not `neurodata_type_def`.
- You can define a group that is not a neurodata type by omitting both `neurodata_type_def` and `neurodata_type_inc`.

Tip: Although you have the option not to, there are several advantages to defining new groups and neurodata types. Neurodata types can be reused in multiple places in the schema, and can be linked to, while groups that are not neurodata types cannot. You can also have multiple neurodata type groups of the same type in the same group, whereas groups that are not neurodata types are limited to 0 or 1. Most of the time, we would recommend making a group a neurodata type. It is also generally better to extend your neurodata type from an existing type. Look through the [NWB schema](#) to see if a core neurodata type would work as a base for your new type. If no existing type works, consider extending `NWBDataInterface`, which allows you to add the object to a processing module.

Tip: New neurodata types should always be declared at the top level of the schema rather than nesting type definitions. I.e., when creating a new neurodata type it should be placed at the top level of your schema and then included at the

appropriate location via `neurodata_type_inc`. This approach greatly simplifies management of types.

For more information about the options available when specifying a Group, see the [API docs for NWBGroupSpec](#).

4.2.2 Dataset Specifications

All larger blocks of numeric or text data should be stored in Datasets. Specifying datasets is done with `NWBDatasetSpec`.

```
from pynwb.spec import NWBDatasetSpec

spec = NWBDatasetSpec(
    doc='A custom NWB type',
    name='qux',
    shape=(None, None),
    attributes=[...])
```

`neurodata_type_def`, `neurodata_type_inc`, `doc`, `name`, `default_name`, `linkable`, `quantity`, and `attributes` all work the same as they do in `NWBGroupSpec`, described in the previous section.

`dtype` defines the type of the data, which can be a basic type, compound type, or reference type. See a list of [dtype options](#) as part of the specification language docs. Basic types can be defined as string objects and more complex types via `NWBdtypeSpec` and `RefSpec`.

`shape` is a specification defining the allowable shapes for the dataset. See the [shape specification](#) as part of the specification language docs. `None` is mapped to `null`. If no shape is provided, it is assumed that the dataset is only a single element.

If the dataset is a single element (scalar) that represents meta-data, consider using an `Attribute` (see below) to store the data more efficiently instead. However, note that a `Dataset` can have `Attributes`, whereas an `Attribute` cannot have `Attributes` of its own. `dims` provides labels for each dimension of `shape`.

Using datasets to specify tables

Row-based tables can be specified using `NWBdtypeSpec`. To specify a table, provide a list of `NWBdtypeSpec` objects to the `dtype` argument.

```
from pynwb.spec import NWBDatasetSpec, NWBDtypeSpec

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

Tip: Column-based tables are also possible and more flexible. See the documentation for [DynamicTable](#).

4.2.3 Attribute Specifications

Attributes are small metadata objects describing the nature and/or intended usage of a Group or Dataset. Attributes are defined in the `attributes` field of a `NWBGroupSpec` or `NWBDatasetSpec`. `attributes` takes a list of `NWBAttributeSpec` objects.

```
from pynwb.spec import NWBAttributeSpec

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

`NWBAttributeSpec` has arguments very similar to `NWBDatasetSpec`. A key difference is that an attribute cannot be a neurodata type, i.e., the `neurodata_type_def` and `neurodata_type_inc` keys are not allowed. The only way to match an object with a spec is through the name of the attribute so `name` is required. You cannot have multiple attributes on a single group/dataset that correspond to the same `NWBAttributeSpec`, since these would have to have the same name. Therefore, instead of specifying number of quantity, you have a `required` field which takes a boolean value. Another key difference between datasets and attributes is that attributes cannot have attributes of their own.

Tip: Dataset or Attribute? It is often possible to store data as either a Dataset or an Attribute. Our best advice is to keep Attributes small. In HDF5 the typical size limit for attributes is 64Kbytes. If an attribute is going to store more than 64Kbyte, then make it a Dataset. Attributes are also more efficient for storing very small data, such as scalars. However, attributes cannot have attributes of their own, and in HDF5, I/O filters, such as compression and chunking, cannot apply to attributes.

4.2.4 Link Specifications

You can store an object in one place and reference that object in another without copying the object using [Links](#), which can be defined using `NWBLinkSpec` objects.

```
from pynwb.spec import NWBLinkSpec

spec = NWBLinkSpec(
    doc='my link',
    target_type='ElectricalSeries',
    quantity='?'
)
```

`doc`, `quantity`, and `name` work similarly to `NWBDatasetSpec`.

`target_type` indicates the neurodata type that can be referenced.

Tip: In case you need to store large collections of links, it can be more efficient to create a dataset for storing the links via object references. In NWB, this is used, e.g., in `py:class:~pynwb.epoch.TimeIntervals` to store collections of

references to TimeSeries objects.

Using these functions in `create_extension_spec.py` and then running that file will generate YAML files that define your extension. If you are a MATLAB user, you are now ready to switch over to MATLAB. Just run `generateExtension('path/to/ndx_name.extension.yaml')` and the extension will be automatically generated for you. If you are a Python user, you need to do a little more work to make a Python API that allows you to read and write data according to this extension. The next two sections will teach you how to create this Python API.

4.3 Generating an API for an extension

4.3.1 Generating a MatNWB API

In MatNWB, simply call `generateExtension("path/to/extension/namespace.yaml")`; The class files will be generated under the `+types/+<extension>` module and can be accessed via standard MATLAB class semantics:

```
ts = types.ndx_example.TetrodeSeries(<arguments>);
```

Note: As seen above, MatNWB will convert namespace names if they are not valid identifiers in MATLAB. See [Variable Names](#) for more information. In most cases, the conversion conforms with MATLAB's approach with `matlab.lang.makeValidName()`

4.3.2 Generating a PyNWB API

Now that we have created the extension specification, we need to create the Python interface. These classes will be used just like the PyNWB API to read and write NWB files using Python. There are two ways to do this: you can automatically generate the API classes based on the schema, or you can manually create the API classes. Here, we will show you how to automatically generate the API. In the next section we will discuss why and how to create custom API classes.

Open up `ndx-example/src/pynwb/ndx_example/__init__.py`, and notice the last line:

```
TetrodeSeries = get_class('TetrodeSeries', 'ndx-example')
```

`get_class()` is a function that automatically creates a Python API object by parsing the extension YAML. If you create more neurodata types, simply go down the line creating each one. This is the same object that is created when you use the `load_namespaces` flag on `__init__()`.

4.3.3 Customizing automatically generated APIs

Once these classes are generated, you can customize them by dynamically adding or replacing attributes/methods (a.k.a., monkey patching).

A typical example is adding methods. Let's say you wanted a method that could return data from only the first channel. You could add that method like this:

```
def data_from_first_chan(self):
    return self.data[:, 0]

TetrodeSeries.data_from_first_chan = data_from_first_chan
```

You can also alter existing methods by overwriting them. Lets suppose you wanted to ensure that the `trode_id` field is never less than 0 for the `TetrodeSeries` constructor. You can do this by creating a new `__init__` function and assigning it to the class.

```
from hdmf.utils import docval, get_docval
from hdmf.common.table import DynamicTableRegion

@docval(get_docval(TetrodeSeries.__init__))
def new_init(self, **kwargs):
    assert kwargs['trode_id'] >= 0, f"`trode_id` must be greater than or equal to 0."
    TetrodeSeries.__init__(self, **kwargs)

TetrodeSeries.__init__ = new_init
```

The above code creates a `new_init` method that runs a validation step and then calls the original `__init__`. Then the class `__init__` is overwritten by the new method. Here we also use `docval`, which is described in the next section.

Tip: This approach is easy, but note your API will be locked to your specification. If you make changes to your specification there will be corresponding changes to the API, and this is likely to break existing code. Also, monkey patches can be very confusing to someone who is not aware of them. Differences between the installed module and the actual behavior of the source code can lead to frustrated developers. As such, this approach should be used with great care. In the next section we will show you how to create your own custom API that is more robust.

4.4 Building a custom Python API for an extension

Creating custom extensions is recommended if you want a stable API that can remain the same even as you make changes to the internal data organization. The `pynwb.core` module has various tools to make it easier to write classes that behave like the rest of the PyNWB API.

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

4.4.1 Docval

`docval` is a library within PyNWB and HDMF that performs input validation and automatic documentation generation. Using the `docval` decorator is recommended for methods of custom API classes.

This decorator takes a list of dictionaries that specify the method parameters. These dictionaries are used for enforcing type and building a Sphinx docstring. The first arguments are dictionaries that specify the positional arguments and keyword arguments of the decorated function. These dictionaries must contain the following keys: `'name'`, `'type'`, and `'doc'`. This will define a positional argument. To define a keyword argument, specify a default value using the key `'default'`. To validate the dimensions of an input array add the optional `'shape'` parameter.

The decorated method must take `self` and `**kwargs` as arguments.

When using this decorator, the functions `getargs()` and `popargs()` can be used for easily extracting arguments from `kwargs`.

The following code example demonstrates the use of this decorator:

```
@docval({'name': 'arg1:', 'type': str, 'doc': 'this is the first positional_
↪argument'},
        {'name': 'arg2:', 'type': int, 'doc': 'this is the second positional_
↪argument'},
        {'name': 'kwarg1:', 'type': (list, tuple), 'doc': 'this is a keyword argument',
↪'default': list()}),
        returns='foo object', rtype='Foo')
def foo(self, **kwargs):
    arg1, arg2, kwarg1 = getargs('arg1', 'arg2', 'kwarg1', **kwargs)
    ...
```

The 'shape' parameter is a tuple that follows the same logic as the `shape` parameter in the specification language. It can take the form of a tuple with integers or `None` in each dimension. `None` indicates that this dimension can take any value. For instance, `(3, None)` means the data must be a 2D matrix with a length of 3 and any width. 'shape' can also take a value that is a tuple of tuples, in which case any one of those tuples can match the spec. For instance, "shape": `((3, 3), (4, 4, 4))` would indicate that the shape of this data could either be 3x3 or 4x4x4.

The 'type' argument can take a class or a tuple of classes. We also define special strings that are macros which encompass a number of similar types, and can be used in place of a class, on its own, or within a tuple. 'array_data' allows the data to be of type `np.ndarray`, `list`, `tuple`, or `h5py.Dataset`; and 'scalar_data' allows the data to be `str`, `int`, `float`, `bytes`, or `bool`.

4.4.2 Registering classes

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

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

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

class MyContainer(NWBContainer):
    ...

register_class('MyContainer', 'my_ns', MyContainer)
```

Alternatively, you can use `register_class()` as a decorator.

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

@register_class('MyContainer', 'my_ns')
class MyContainer(NWBContainer):
    ...
```

`register_class()` is used with `NWBData` the same way it is used with `NWBContainer`.

4.4.3 Nwbfields

When creating a new neurodata type, you need to define the new properties on your class, which is done by defining them in the `__nwbfields__` class property. This class property should be a tuple of strings that name the new properties. Adding a property using this functionality will create a property than can be set *only once*. Any new properties of the class should be defined here.

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

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

class MyContainer(NWBContainer):

    __nwbfields__ = ('foo', 'bar')

    ...
```

4.4.4 NWBContainer

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

NWBDataInterface

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

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

MultiContainerInterface

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

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

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

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

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

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

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

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

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

4.4.5 NWBData

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

NWBTable

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

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

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

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

    __columns__ = [
```

(continues on next page)

(continued from previous page)

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

...
```

NWBTableRegion

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

4.4.6 Custom data checks on `__init__`

When creating new instances of an API class, we commonly need to check that input parameters are valid. As a common practice the individual checks are typically implemented as separate functions named `_check_...` on the class and then called in `__init__`.

To support access to older file version (which may not have followed some new requirements) while at the same time preventing the creation of new data that is invalid, PyNWB allows us to detect in `__init__` whether the object is being constructed by the `ObjectMapper` on read or directly by the user, simply by checking if `self._in_construct_mode` is `True/False`. For convenience, PyNWB provides the `_error_on_new_warn_on_construct()` method, which makes it easy to raise warnings on read and errors when creating new data.

4.4.7 ObjectMapper

Customizing the mapping between NWBContainer and the Spec

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

`ObjectMapper` extensions are registered with the decorator `register_map()`.

```
from pynwb import register_map
from hdmf.build import ObjectMapper

@register_map(MyExtensionContainer)
class MyExtensionMapper(ObjectMapper)
    ...
```

`register_map()` can also be used as a function.

```
from pynwb import register_map
from hdmf.build import ObjectMapper

class MyExtensionMapper(ObjectMapper)
    ...

register_map(MyExtensionContainer, MyExtensionMapper)
```

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

4.5 Documenting Extensions

Using the same tools used to generate the documentation for the [NWB core format](#), one can easily generate documentation in HTML, PDF, ePub and many other formats for extensions.

If you used [ndx-template](#), then your repository is already pre-configured to automatically generate documentation for your extension using the [HDMF DocUtils](#) and [Sphinx](#). The docs directory structure should look like this.

```
ndx-my-extension/
  docs/
    source/
      credits.rst
      description.rst
      release_notes.rst
      ...
```

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

```
cd docs/source
make html
```

The generated documentation will be available in `build/html`. To view, open `build/html/index.html` in your browser. These pages contain diagrams of your extension. Note that there are several places where information needs to be added. For instance, the Overview section says:

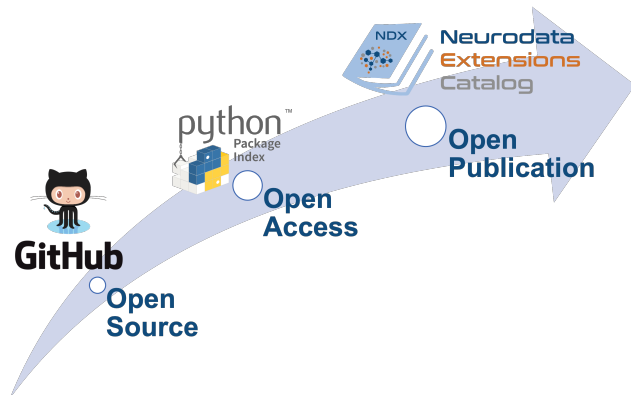
Note: Add the description of your extension here

Within `docs/source`, edit `credits.rst`, `description.rst`, and `release_notes.rst`, then rerun `make html`.

Now that you have created documentation for your extension, it is time to learn how to publish in the NDX catalog.

See [published extensions](#) and learn how to share your extension on the [NDX Catalog website](#).

4.6 Publishing extensions



Neurodata extensions can be shared with the community using the [NDX Catalog](#). As illustrated in the figure, the publication process is divided into three main steps: 1) open release of the sources to the community using GitHub, 2) open access of versioned releases via PyPI, and 3) open publication of the extension to the community via the [NDX Catalog](#).

4.6.1 Open Source: Releasing your extension Git repository

1. Before publishing your extension online you should add a license file. Permissive licenses should be used if possible. A *BSD license* <<https://opensource.org/licenses/BSD-3-Clause>> is recommended.
2. Modify `README.md` at the root directory of your extension repo to describe the extension for interested developers and users.
3. The first step to publishing your extension then is to make your Git repository accessible online via GitHub, or any other public Git hosting service of your choice. To publish your extension on GitHub you will need a [GitHub account](#) and follow the following [instructions](#) to add an existing project to GitHub
4. Make a release for the extension on GitHub with the version number specified. e.g. if the version is `0.1.0`, then this page should exist: https://github.com/<my_username>/<my_extension>/releases/tag/0.1.0. See the [creating a release guide](#) on GitHub for instructions on how to make a release. See the [NWB Versioning Guidelines](#) for details on how to version extensions.

Note: We here focus on GitHub because it is the services that is currently most commonly used for extensions repositories. However, users may chose to use other services (e.g., GitLab or Bitbucket) to share their sources.

4.6.2 Open Access: Releasing your extension on PyPI

To make your extension installable for users via pip and manage public releases NWB uses the [Python Package Index \(PyPI\)](#) index.

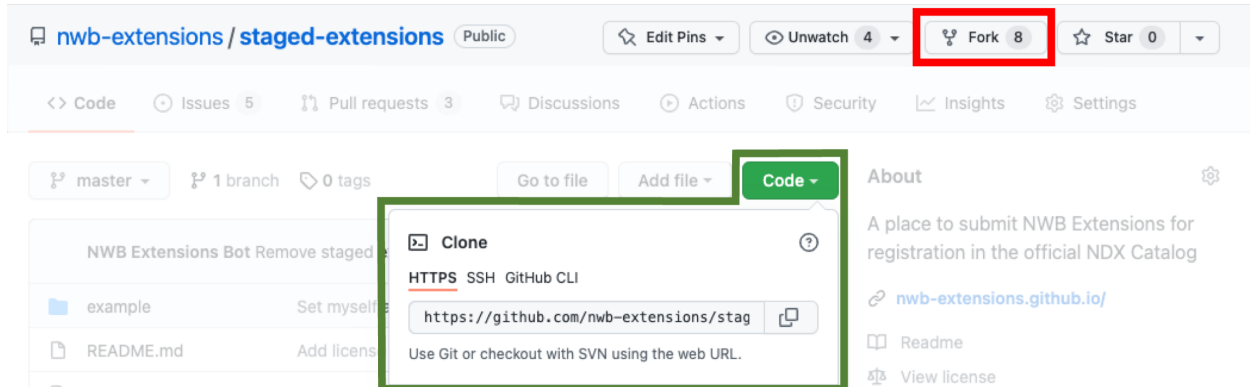
1. Follow [these directions](#) to package your project. You may need to modify `setup.py`. If your extension version is `0.1.0`, then this page should exist: <https://pypi.org/project/<myextension>/0.1.0>
2. Once your GitHub release and `setup.py` are ready, publishing on PyPI:

```
python setup.py sdist bdist_wheel
twine upload dist/*
```

4.6.3 Open Publication: Publishing your extension on the NDX Catalog

The [NDX Catalog](#) serves as a central, community-led catalog for extensions to the NWb data standard. The NDX Catalog manages basic metadata about extensions while ownership of the source repositories for the extensions remain with the developers. To publish your extension on the catalog:

1. Fork the [staged-extensions](#) repository, which is used to submit new extension to the catalog via pull requests.
2. Clone your fork of the *staged-extensions* onto your computer, e.g., via `git clone <my_fork_url>`



3. Copy the directory `staged-extensions/example` to a new directory with the name of your extension, e.g., via `cp -r staged-extensions/example staged-extensions/<my_extension>`
4. Edit `staged-extensions/<my_extension>/ndx-meta.yaml` with information on where to find your NWb extension. The `NEXTSTEPS.md` file in the `ndx-template` includes an autogenerated template `ndx-meta.yaml` file that you may copy and modify. The YAML file **MUST** contain a dict with the following keys:
 - **name:** extension namespace name
 - **version:** extension version
 - **src:** URL for the main page of the public repository (e.g. on GitHub, BitBucket, GitLab) that contains the sources of the extension
 - **pip:** URL for the main page of the extension on PyPI
 - **license:** name of the license of the extension
 - **maintainers:** list of GitHub usernames of those who will reliably maintain the extension

You may copy and modify the following YAML that was auto-generated:

5. Edit `staged-extensions/<my_extension>/README.md` to add information about your extension. Usually, you can here just copy the `README.md` from your extension repo `cp <my_extension>/README.md staged-extensions/<my_extension>/README.md`

6. Add and commit your changes to Git and push your changes to GitHub:

```
cd staged-extensions
git add <my_extension>
git commit -m "Add new catalog entry for <my_extension>"
git push
```

7. Open a pull request. See the [creating a pull request from a fork](#) website for step-by-step instructions on to create a pull request on GitHub.
8. Once the PR has been created, building of your extension will be tested on Windows, Mac, and Linux. The technical team will review your extension shortly after and provide feedback and request changes, if any. Once the technical team has approved and merged your pull request, a new repository, called `<my_extension>-record`

will be created in the [nwb-extensions GitHub organization](#) and you will be added as a maintainer for that repository.

4.6.4 Updating your published extension

Once you have published your extension you can update and publish new version as follows:

1. Update your `<my_extension>` GitHub repository
2. Publish your updated extension on PyPI.
3. Fork the `<my_extension>-record` repository from the [nwb-extensions GitHub organization](#) and update your `ndx-meta.yaml`, `README.md` and other relevant record data
4. Open a pull request to test the changes automatically.
5. The technical team will review your changes shortly after and provide feedback and request changes, if any.
6. Your updated extension is ready once your PR has been approved and merged.

4.6.5 Policies: Neurodata Extension (NDX) rules and guidelines

- [Sharing Guidelines](#): requirements and strategy for sharing format extensions for NWB
- [Sharing Strategies](#): standard practices and strategies for sharing format extensions for NWB
- [Proposal Review Process](#): process by which extensions to the NWB core standard are proposed, evaluated, reviewed, and accepted
- [Versioning Guidelines](#): requirements and strategy for versioning namespaces for the NWB core schema and extensions

4.7 Examples

4.7.1 Extensions for lab-specific metadata: Extending LabMetaData

Use case

Here we address the use case of adding lab-specific metadata to a file, e.g., lab-specific information about experimental protocols, lab-specific identifiers and so on. This approach is intended for usually small metadata. [Extension source](#)

Approach

To include lab-specific metadata, NWB provides `pynwb.file.LabMetaData` as a convenient base type, which makes it easy to add your data to an `pynwb.file.NWBFile` without having to modify the `pynwb.file.NWBFile` type itself (since adding of `pynwb.file.LabMetaData` is already implemented).

Note: NWB uses dynamically extensible table structures based on [DynamicTable](#) to describe metadata and derived results, e.g., `TimeIntervals` for epochs or trials or `ElectrodeTable` to describe extracellular electrodes. Depending on the type of metadata, use of these existing dynamic table structures can help avoid the need for custom extensions by including the data as additional, custom columns in the appropriate existing tables.

Creating the extension

1. Create a new repository for the extension using the `ndx-template`:

```
cookiecutter gh:nwb-extensions/ndx-template
```

2. Answer a few simple questions of the cookiecutter template. We can respond to many questions with Enter to accept the default response (e.g., to start with `version=0.1.0`):

```
namespace [ndx-my-namespace]: ndx-labmetadata-example
description [My NWB extension]: Example extension to illustrate how to extend
↳ LabMetaData for adding lab-specific metadata
author [My Name]: Oliver Ruebel
email [my_email@example.com]: oruebel@lbl.gov
github_username [myname]: oruebel
copyright [2021, Oliver Ruebel]:
version [0.1.0]:
release [alpha]:
Select license:
1 - BSD-3
2 - MIT
3 - Apache Software License 2.0
4 - Other
Choose from 1, 2, 3, 4 [1]: 1
py_pkg_name [ndx_labmetadata_example]:
```

3. Edit `ndx-my-brainlabsrc/spec/create_extension_spec.py` that was generated for you to define the schema of your extension. See [The Spec API](#) section for details on how to use the specification API.

- Add `LabMetaData` as an include type

```
ns_builder.include_type('LabMetaData', namespace='core')
```

- Define your new `LabMetaData` type for your lab

```
labmetadata_ext = NWBGroupSpec(
    name='custom_lab_metadata',
    doc='Example extension type for storing lab metadata',
    neurodata_type_def='LabMetaDataExtensionExample',
    neurodata_type_inc='LabMetaData',
)
```

- Add the Groups, Datasets, and Attributes with the metadata specific to our lab to our `LabMetaData` schema

```
labmetadata_ext.add_dataset(
    name="tissue_preparation",
    doc="Lab-specific description of the preparation of the tissue",
    dtype='text',
    quantity='?'
)
```

- Add our new type definitions to the extension

```
new_data_types = [labmetadata_ext]
```

4. Generate the schema for the extension by running the `create_extension_spec.py` script

```
cd ndx-labmetadata-example
python src/spec/create_extension_spec.py
```

5. Edit `src/pynwb/__init__.py` to define Python API classes for our new extension data types via `pynwb.get_class()`.

```
LabMetaDataExtensionExample = get_class('LabMetaDataExtensionExample', 'ndx-labmetadata-
↳example')
```

6. Define unit tests for the extension. The `ndx-template` created an example test module `src/pynwb/tests/test_tetrodeseries.py` to illustrate how to implement tests. Here we simply remove this file and replace it with our own tests `test_labmetadata_example.py`. More details below in [Creating unit tests](#).

7. To make sure our extension schema and source code files are version controlled, we now add all the files we just created to the Git repo:

```
git add .
git commit -m "Added API classes, tests, and schema files"
```

8. Install your extension (Python only)(Optional)

```
pip install .
```

Now our extension is ready to use!

Creating custom Python API classes

We skip this step here, since this extension of `LabMetaData` is simple enough that the autogenerated class is sufficient. If the autogenerated class from `pynwb.get_class()` for an extension data types is not sufficient, then we can either customize the autogenerated class as described in [Generating a PyNWB API](#) (recommended only for basic changes) or define our own custom API class as described in [Building a custom Python API for an extension](#) (recommended for full customization).

Creating unit tests

Python

MATLAB

Unit test

Roundtrip test (read/write)

Running Python unit tests

```
from pynwb.testing.mock.file import mock_NWBFile
from pynwb.testing import TestCase
from ndx_labmetadata_example import LabMetaDataExtensionExample

class TestLabMetaDataExtensionExample(TestCase):
    """Test basic functionality of LabMetaDataExtensionExample without read/write"""

    def setUp(self):
```

(continues on next page)

(continued from previous page)

```

        """Set up an NWB file. Necessary because TetrodeSeries requires references to
        ↪electrodes."""
        self.nwbfile = mock_NWBFile()

    def test_constructor(self):
        """Test that the constructor for TetrodeSeries sets values as expected."""
        tissue_preparation = "Example tissue preparation"
        lmdee_object = LabMetaDataExtensionExample(tissue_preparation=tissue_preparation)
        self.assertEqual(lmdee_object.tissue_preparation, tissue_preparation)

```

```

from pynwb.testing.mock.file import mock_NWBFile
from pynwb.testing import TestCase
from pynwb.testing.testh5io import NWBH5IOMixin
from ndx_labmetadata_example import LabMetaDataExtensionExample

class TestLabMetaDataExtensionExampleRoundtrip(NWBH5IOMixin, TestCase):
    """
    Roundtrip test for LabMetaDataExtensionExample to test read/write

    This test class writes the LabMetaDataExtensionExample to an NWBFile, then
    reads the data back from the file, and compares that the data read from file
    is consistent with the original data. Using the pynwb.testing infrastructure
    simplifies this complex test greatly by allowing to simply define how to
    create the container, add to a file, and retrieve it from a file. The
    task of writing, reading, and comparing the data is then taken care of
    automatically by the NWBH5IOMixin.
    """

    def setUpContainer(self):
        """set up example LabMetaDataExtensionExample object"""
        self.lab_meta_data = LabMetaDataExtensionExample(tissue_preparation="Example_
        ↪tissue preparation")
        return self.lab_meta_data

    def addContainer(self, nwbfile):
        """Add the test LabMetaDataExtensionExample to the given NWBFile."""
        nwbfile.add_lab_meta_data(lab_meta_data=self.lab_meta_data)

    def getContainer(self, nwbfile):
        """Get the LabMetaDataExtensionExample object from the given NWBFile."""
        return nwbfile.get_lab_meta_data(self.lab_meta_data.name)

```

```

cd ndx-labmetadata-example
pytest

```

Unit test

Roundtrip test (read/write)

Running MATLAB unit tests

Coming soon ...

Coming soon ...

Coming soon ...

Documenting the extension

- **README.md:** Add instructions to the README.md file. This typically includes information on how to install the extension and an example on how to use the extension
- **Schema and user documentation:**
 - Install the latest release of hdmf-docutils: `python -m pip install hdmf-docutils`
 - Generate the documentation for your extension based on the YAML schema files via:

```
cd docs/
make html
```

- To view the docs, simply open docs/build/html/index.html in your browser
- See the docs/README.md for instructions on how to customize the documentation for your extension.

See *Documenting Extensions* for more details.

Writing data using the extension

Python

MATLAB

```
from pynwb.file import NWBFile, Subject
from ndx_labmetadata_example import LabMetaDataExtensionExample
from pynwb import NWBHDF5IO
from uuid import uuid4
from datetime import datetime

# create an example NWBFile
nwbfile = NWBFile(
    session_description="test session description",
    identifier=str(uuid4()),
    session_start_time=datetime(1970, 1, 1),
    subject=Subject(
        age="P50D",
        description="example mouse",
        sex="F",
        subject_id="test_id")
)

# create our custom lab metadata
lab_meta_data = LabMetaDataExtensionExample(tissue_preparation="Example tissue_
↳ preparation")

# Add the test LabMetaDataExtensionExample to the NWBFile
```

(continues on next page)

(continued from previous page)

```
nwbfile.add_lab_meta_data(lab_meta_data=lab_meta_data)

# Write the file to disk
filename = "testfile.nwb"
with NWBHDF5IO(path=filename, mode="a") as io:
    io.write(nwbfile)
```

Coming soon ...

Reading an NWB file that uses the extension

Python

Python (without extension installed)

MATLAB

```
from pynwb import NWBHDF5IO
from ndx_labmetadata_example import LabMetaDataExtensionExample

# Read the file from disk
io = NWBHDF5IO(path=filename, mode="r")
nwbfile = io.read()
# Get the custom lab metadata object
lab_meta_data = nwbfile.get_lab_meta_data(name="custom_lab_metadata")
```

```
from pynwb import NWBHDF5IO

# Read the file from disk. Load the namespace from file to
# autogenerate classes from the schema
io = NWBHDF5IO(path=filename, mode="r", load_namespaces=True)
nwbfile = io.read()
# Get the custom lab metadata object
lab_meta_data = nwbfile.get_lab_meta_data(name="custom_lab_metadata")
```

Coming soon ...

Publishing the extension

The steps to publish an extension are the same for all extensions. We, therefore, here only briefly describe the main steps for publishing our extension. For a more in-depth guide, see the page [Publishing extensions](#)

- **GitHub (Open Source):** To make the sources of your extension openly accessible, publish the extension on GitHub by following the instructions on [Open Source: Releasing your extension Git repository](#).
- **PyPI (Open Access):** Publish your extension on PyPI to make it easy for users to install it and to create a persistent release of the extension following the [Open Access: Releasing your extension on PyPI](#) guide.
- **NDX Catalog (Open Publication):** The [NDX Catalog](#) serves as a central, community-led catalog for extensions to the NWB data standard. The NDX Catalog manages basic metadata about extensions while ownership of the source repositories for the extensions remain with the developers. For a step-by-step guide the [Open Publication: Publishing your extension on the NDX Catalog](#) guide.

GLOSSARY OF CORE NWB TOOLS

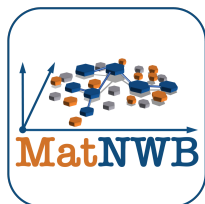
The glossary shown here provides a quick overview of the key software packages of the core NWB software stack. For a more general discussion of the overall organization of the core NWB software stack see the [NWB Software Ecosystem](#) page on the main NWB website.

5.1 Read/Write NWB File APIs

The NWB reference APIs provide full support for reading and writing all components of the NWB standard, including support for extensions. The APIs are interoperable, i.e., files created with PyNWB can be read in MatNWB and vice versa. Both PyNWB and MatNWB support advanced read/write for efficient interaction with very large data files (i.e., data too large for main memory), via lazy data loading, iterative data write, and data compression among others.



PyNWB is the Python reference API for NWB. [Docs](#) [Tutorials](#) [Source](#)



MatNWB is a MATLAB library for reading and writing NWB files. [Docs](#) [Tutorials](#) [Source](#)

5.2 Converting Data to NWB



The [NeuroConv](#) is a Python library for automatic conversion from proprietary data formats to NWB. [Docs](#) [Source](#)



The [NWB GUIDE](#) is a desktop app that provides a no-code **G**raphical **U**ser **I**nterface for **D**ata **E**ntry for converting neurophysiology data to NWB using [NeuroConv](#). **Note:** NWB GUIDE is in pre-release status and under active development. [Source](#)

5.3 Validating NWB Files

NWB provides tools to check that files comply with the [NWB standard schema](#) as well as to check whether the data complies with [NWB Best Practices](#). Validating compliance with the NWB schema ensures that files are structurally correct and can be read by NWB APIs. Validating compliance with best practices helps improve data quality and (re-)usability.



[NWB Inspector](#) is a python library and command-line tool for inspecting NWB files for adherence to [NWB best practices](#). By default, the Inspector also runs the PyNWB validator to check for compliance with the NWB schema. The Inspector can also be easily extended to integrate custom data checks and to configure checks. [Docs](#) [Source](#)



The [PyNWB](#) reference Python API includes classes and command line tools for validating compliance of files with the core NWB schema and the schema of NWB Neurodata Extensions (NDX). [Validation Docs](#)

Hint: In practice, most user should use the [NWB Inspector](#) to validate NWB files, as it helps to check for compliance with both the schema and best practices and provides greater flexibility. Direct use of [PyNWB's validator](#) is primarily useful for use case where schema compliance and performance are of primary concern, for example, during development of extensions or as part of automated test environments.

5.4 Extending NWB

Neurodata Extensions (NDX) are used to extend the NWB data standard, e.g., to integrate new data types with NWB or define standard for lab- or project-specific metadata. The collection of tools listed here are used to create, document, publish extensions. To learn more about how create extensions see the [Extending NWB](#) section.



The [Neurodata Extensions Catalog \(NDX Catalog\)](#) is a community-led catalog of Neurodata Extensions (NDX) to the NWB data standard. The [NDX Catalog](#) provides a central portal to search, publish, and review of NDX. [Catalog Source](#)



The [NDX Template](#) provides a template for creating Neurodata Extensions (NDX) for the NWB data standard. When creating a new extension, the NDX-template will create a detailed NEXTSTEPS.md file describing how to create an extension and how to submit it to the NDX catalog. [Source](#)



The [staged-extensions GitHub repository](#) is used to register new extensions for publication in the [Neurodata Extensions Catalog \(NDX Catalog\)](#). [Source](#)



The [HDMF Documentation Utilities \(hdmf-docuils\)](#) provide utility tools for creating documentation for extension schema defined using the [NWB Schema Language](#). The [NDX Template](#) automatically sets up the documentation for

extensions via the [hdmf-docuils](#) and as such are part of most NDX code repositories without having to interact with the tool directly. [Source](#)



The [HDMF Specification Language](#) defines formal structures for describing the organization of complex data using basic concepts, e.g., Groups, Datasets, Attributes, and Links. The HDMF specification language is defined by the [Hierarchical Data Modeling Framework \(HDMF\)](#). The [NWB Specification Language](#) then is a derivative of the [HDMF Specification Language](#) with minor modifications for NWB (e.g., to use the term *neurodata_type*). [Source](#) [Docs](#).



The NWB data standard is governed by the [NWB Format Specification](#) (a.k.a., the NWB Schema) described using the [NWB Specification Language](#). When creating new extensions we typically build on and reuse existing *neurodata_types* already available in NWB. The [NWB Format Specification](#) provides a reference definition for all types available in NWB. The NWB schema itself builds on the [HDMF Common Schema](#). [Docs](#) [Source](#)



The [HDMF Common Schema](#) defines the schema of common, general data structures, which are used throughout the [NWB Standard Schema](#) but which are not specific to neurophysiology. Example types defined in the HDMF common schema include, e.g., all types related to [DynamicTable](#) for defining data tables. [Docs](#) [Source](#)

5.5 Core Development

Understanding core development tools (e.g., HDMF) is useful for developers in particular when we need to dive deeper into the core data infrastructure for NWB, e.g., when changing or creating new storage methods or when developing features for common data types (e.g., [DynamicTable](#)) that are defined in HDMF and used in NWB.



The [Hierarchical Data Modeling Framework \(HDMF\)](#) is a python package for working with hierarchical data. It provides APIs for specifying data models, reading and writing data to different storage backends, and representing data with Python object. HDMF builds the foundation for the [PyNWB](#) Python API for NWB. [Docs](#) [Source](#)



The [HDMF Zarr \(HDMF-Z\)](#) library implements a Zarr backend for HDMF. HDMF-Z also provides convenience classes for integrating Zarr with the [PyNWB](#) Python API for NWB to support writing of NWB files to Zarr. [Docs](#) [Source](#)

ANALYSIS AND VISUALIZATION TOOLS

6.1 NWB Widgets

NWB Widgets is a library of widgets for visualization NWB data in a Jupyter notebook (or lab). The widgets make it easy to navigate through the hierarchical structure of the NWB file and visualize specific data elements. It is designed to work out-of-the-box with NWB 2.0 files and to be easy to extend. [Source Docs](#)

6.1.1 Demo

The screenshot shows a Jupyter notebook interface with four code cells and a widget visualization. The code cells execute the following commands:

```
In [1]: from pynwb import NWBHDF5IO
        from nwbwidgets import nwb2widget

In [2]: file_name = '/Users/bendichter/Desktop/Buzsaki/SenzaiBuzsaki2017/YutaMouse41/YutaMouse41-150903

In [3]: nwb_io = NWBHDF5IO(file_name, mode='r')
        nwb = nwb_io.read()

In [4]: nwb2widget(nwb)
```

The widget visualization displays the NWB file's metadata and a hierarchical tree of data elements. The metadata fields are:

- experimenter: "Yuta Senzai"
- session_id: "YutaMouse41-150903"
- lab: "Buzsaki"
- institution: "NYU"
- related_publications: "DOI:10.1016/j.neuron.2016.12.01"

The data elements tree is as follows:

- acquisition
- stimulus
- modules
- devices
- electrode_groups
- epochs: experimental epochs
- trials: experimental trials
- units: Autogenerated by NWBFile

6.1.2 Installation

```
pip install nwbwidgets
```

6.1.3 Usage

```
from pynwb import NWBHDF5IO
from nwbwidgets import nwb2widget

io = NWBHDF5IO('path/to/file.nwb', mode='r')
nwb = io.read()

nwb2widget(nwb)
```

6.1.4 How it works

All visualizations are controlled by the dictionary `neurodata_vis_spec`. The keys of this dictionary are pynwb neurodata types, and the values are functions that take as input that `neurodata_type` and output a visualization. The visualizations may be of type `Widget` or `matplotlib.Figure`. When you enter a `neurodata_type` instance into `nwb2widget`, it searches the `neurodata_vis_spec` for that instance's `neurodata_type`, progressing backwards through the parent classes of the `neurodata_type` to find the most specific `neurodata_type` in `neurodata_vis_spec`. Some of these types are containers for other types, and create accordian UI elements for its contents, which are then passed into the `neurodata_vis_spec` and rendered accordingly.

Instead of supplying a function for the value of the `neurodata_vis_spec` dict, you may provide a dict or `OrderedDict` with string keys and function values. In this case, a tab structure is rendered, with each of the key/value pairs as an individual tab. All accordian and tab structures are rendered lazily- they are only called with that tab is selected. As a result, you can provide many tabs for a single data type without a worry. They will only be run if they are selected.

6.1.5 Extending

To extend `NWBWidgets`, all you need to a function that takes as input an instance of a specific `neurodata_type` class, and outputs a `matplotlib` figure or a `jupyter` widget.

6.2 Neurosift

Neurosift provides interactive neuroscience visualizations in the browser. *Neurosift* caters to both individual users through its local mode, allowing the visualization of views directly from your device, as well as a remote access function for presenting your findings to other users on different machines. [Source](#)

With *Neurosift*, you have the ability to construct a multitude of synchronized visuals such as spike raster plots, audio spectrograms, videos, video annotations, position decode fields, timeseries graphs, and more.

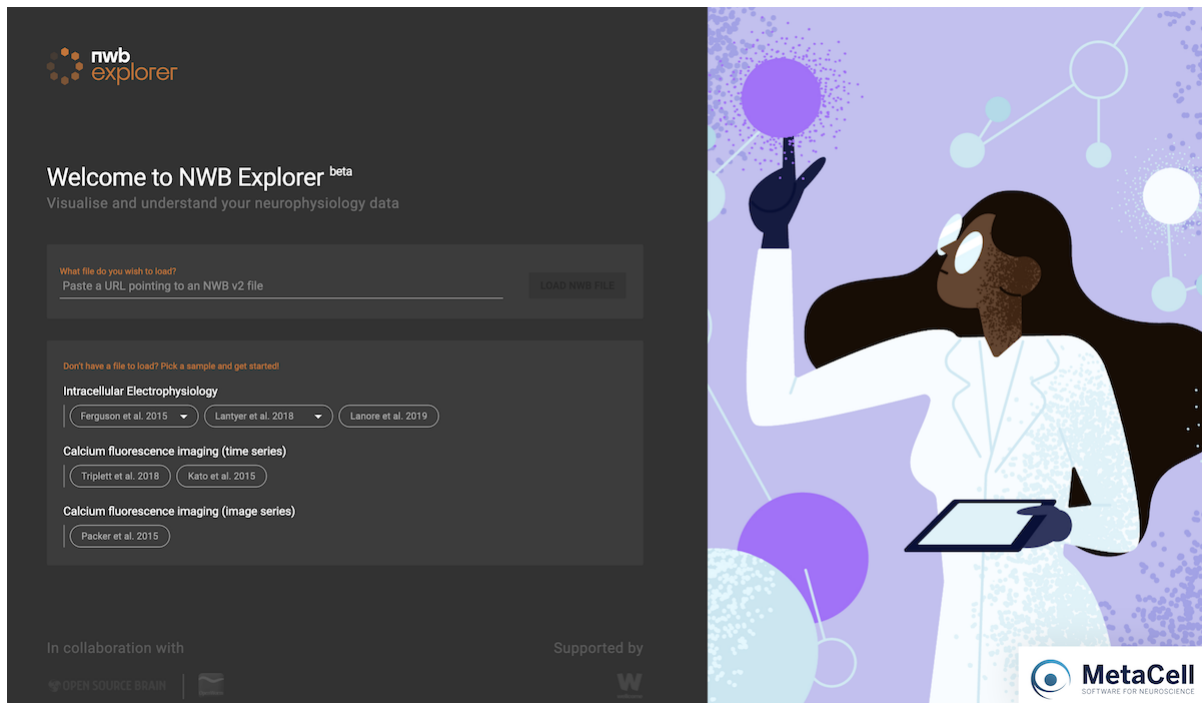
Neurosift is also integrated with DANDI Archive for visualizing NWB files directly in your browser. You can browse the contents of a dandiset directly in your browser, and visualize the items using *Neurosift* views. Here is an example:

```
https://flatironinstitute.github.io/neurosift/#/nwb?url=https://dandiarchive.s3.
amazonaws.com/blobs/c86/cdf/c86cdfba-e1af-45a7-8dfd-d243adc20ced
```

Replace the url query parameter with the appropriate URL for your NWB file on DANDI or anywhere else on the web.

6.3 NWB Explorer

NWB Explorer is a web application developed by MetaCell for reading, visualizing and exploring the content of NWB 2 files. The portal comes with built-in visualization for many data types, and provides an interface to a jupyter notebook for custom analyses and open exploration. [Online](#)

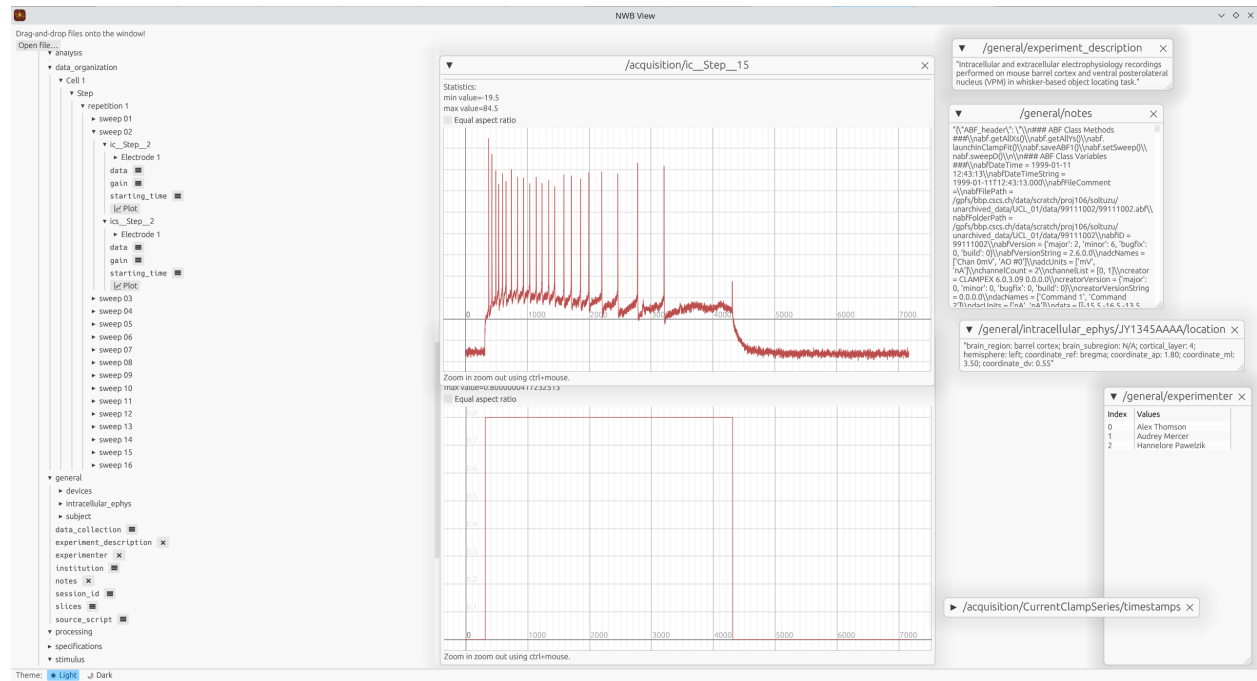


6.4 Nwbview



Nwbview is a cross-platform software with a graphical user interface to display the contents of the binary NWB file format. It is written in Rust for high-performance, memory safety and ease of deployment. Its features include the ability to display the contents of the NWB file in a tree structure. It displays voltage and current recordings data in interactive plots. The tabular data or the text data present in the NWB can be displayed in a scalable window. [Docs Source](#).

The screenshot below shows the *nwbview* GUI. The left panel shows the contents of the NWB file in a tree structure. The right panel shows the details of the selected item, which can be a plot, a table or a text.



nwbview uses the [egui](#) Rust GUI library for rendering.

6.4.1 To install and run using cargo

Cargo is the package manager for Rust, and *nwbview* is listed as a package there. You can find it on [crates.io](#).

Note: HDF5 needs to be installed in your system as cargo will try to locate the HDF5 root directory and/or headers.

First install the *cargo* package manager and then run the following command to install *nwbview*.

```
cargo install nwbview
```

Once you completed the installation, simply type *nwbview* on the console to run it.

```
nwbview
```

6.4.2 To build and run from the source code

The Rust library dependencies are provided in the *cargo.toml* file.

Note that the Rust libraries depend on the following system packages that need to be provided.

- *libgtk-3-dev*
- *librust-atk-dev*
- *libhdf5-serial-dev*

The exact names of the packages may differ between systems.

Once all the dependencies are satisfied, go to the directory containing *cargo.toml* and run the following command.

```
cargo run --release
```

The release flag builds the artifacts with optimizations. Do not specify it when you need to debug.

6.5 HDF Tools

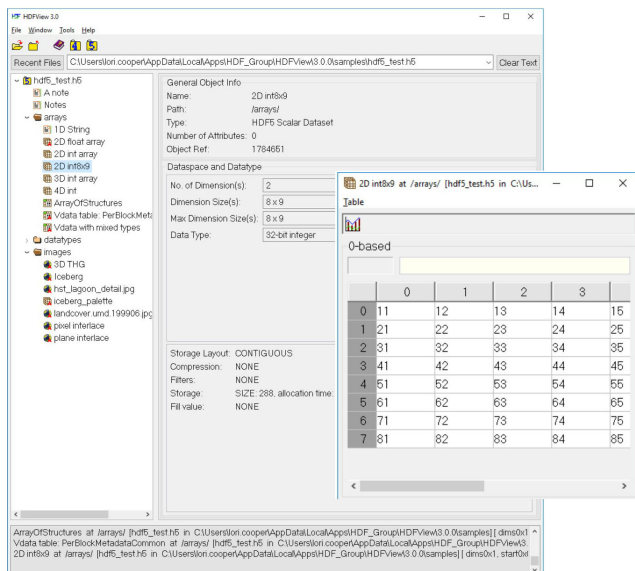
HDF Tools: There are a broad range of useful tools for inspecting and browsing HDF5 files. For example, HDFView and HDF5 plugins for Jupyter or VSCode provide general visual tools for browsing HDF5 files. In addition, the HDF5 library ships with a range of command line tools that can be useful for developers (e.g., *h5ls* and *h5dump* to introspect, *h5diff* to compare, or *h5copy* and *h5repack* to copy HDF5 files). While these tools do not provide NWB-specific functionality, they are useful (mainly for developers) to debug and browse NWB HDF5 files. [HDFView](#) [HDF5 CLI tools](#) [vscode-h5web](#) [h5glance](#) [jupyterlab-h5web](#) [jupyterlab-hdf5](#)

Note: Compatibility with NWB: The tools listed here are generic HDF5 tools and as such are not aware of the NWB schema. Modifying NWB files using generic HDF5 tools can result in invalid NWB files. Use of these tools is, hence, primarily useful for developers, e.g., for debugging and visual inspection of NWB files.

6.5.1 Examples

- [HDFView](#)
- [HDF5 Command-line Tools](#)
- [Jupyter HDF5 plugins](#)
- [VSCode HDF5 plugin](#)

HDFView



HDFView is a visual tool written in Java for browsing and editing HDF (HDF5 and HDF4) files. Using HDFView, you can: i) view a file hierarchy in a tree structure ii) create new files, add or delete groups and datasets, iii) view and modify the content of a dataset, iv) add, delete and modify attributes. HDFView uses the Java HDF Object Package, which implements HDF4 and HDF5 data objects in an object-oriented form. [Download Source](#)

HDF5 Command-line Tools

The HDF5 software distribution ships with a broad range of [HDF5 command-line utilities](#) that cover a broad range of tasks useful for debugging and inspecting HDF5 files, for example:

- **CLI tools for inspecting HDF5 files:**

- `h5ls` lists selected information about file objects in the specified format

Tip: `h5ls` can also be used with remote files on S3 via `h5ls --vfd=ros3 -r <s3path>` or on Windows via `h5ls --vfd=ros3 --s3-cred="(<,>)" <s3path>`

- `h5dump` enables the user to examine the contents of an HDF5 file and dump those contents to an ASCII file.
- `h5diff` compares two HDF5 files and reports the differences. `h5diff` is for serial use while `ph5diff` is for use in parallel environments.
- `h5check` verifies that an HDF5 file is encoded according to the HDF5 specification.
- `h5stat` displays object and metadata information for an HDF5 file.
- `h5watch` Outputs new records appended to a dataset as the dataset grows similar to the Unix user command `tail`.
- `h5debug` debugs an existing HDF5 file at a low level.

- **CLI tools to copy, clean, and edit HDF5 files:**

- `h5repack` copies an HDF5 file to a new file with or without compression/chunking and is typically used to apply HDF5 filters to an input file and saving the output in a new output file.
- `h5copy` copies an HDF5 object (a dataset, named datatype, or group) from an input HDF5 file to an output HDF5 file.
- `h5repart` repartitions a file or family of files, e.g., to join a family of files into a single file or to copy one family of files to another while changing the size of the family members.
- `h5clear` clears superblock status_flags field, removes metadata cache image, prints EOA and EOF, or sets EOA of a file.

These are just a few select tools most relevant to debugging NWB files. See the [HDF5 command-line utilities page](#) for a more detailed overview.

Jupyter HDF5 plugins

[JupyterLab](#) is a popular web-based interactive development environment for notebooks, code, and data. Its flexible interface allows users to configure and arrange workflows in data science, scientific computing, computational journalism, and machine learning. There are several extensions available to facilitate browsing and visualization of HDF5 in JupyterLab and Python notebooks. [h5glance](#) [jupyterlab-h5web](#) [jupyterlab-hdf5](#)

The different libraries each offer slightly different approaches towards visualizaing HDF5 files in Jupyter. [h5glance](#) displays an interactive hierarchy of the HDF5 inline in a Python code notebook. [jupyterlab-hdf5](#) shows the tree in the left sidebar of the JupyterLab UI with additional visualizations shown as separate tabs in the main window. [jupyterlab-h5web](#) shows both the tree and additional visualization in a single additional tab in the main Jupyter window.

Tip: Some extensions expect the file extensions `.h5` to identify HDF5 files. To allow opening of NWB HDF5 files with these tools may require creating custom file associations or renaming NWB files to use the `.h5` extension.

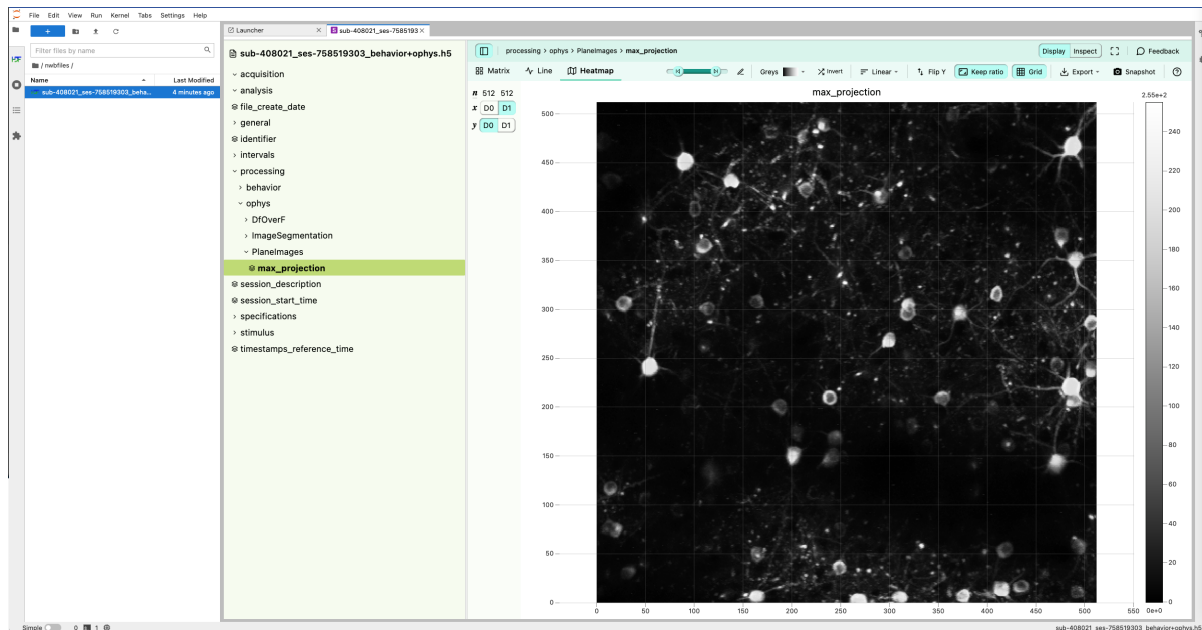


Fig. 1: *jupyterlab-h5web* visualization of an example NWB file.

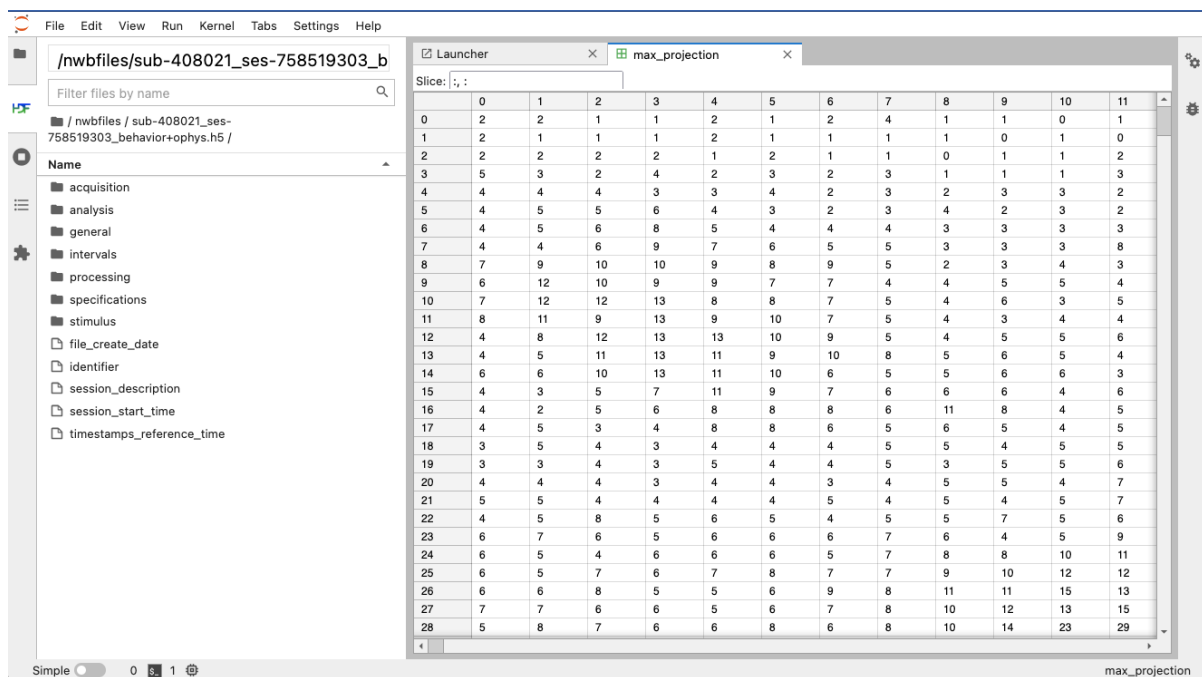


Fig. 2: *jupyterlab-hdf5* visualization of an example NWB file.

```
from h5glance import H5Glance
from pynwb import NWBHDF5IO

nwbio = NWBHDF5IO('nwbfiles/sub-408021_ses-758519303_behavior+ophys.nwb', 'r')
H5Glance(nwbio._file)
```

```

└─ nwbfiles/sub-408021_ses-758519303_behavior+ophys.nwb
    └─ acquisition
    └─ analysis
    └─ general
    └─ intervals
    └─ processing
        └─ behavior
        └─ ophys
            └─ DfOverF
            └─ ImageSegmentation
            └─ PlaneImages
                max_projection [ ]: 512 x 512 entries, dtype: uint8
    └─ specifications
    └─ stimulus
        file_create_date [ ]: 1 entries, dtype: ASCII string
        identifier [ ]: scalar entries, dtype: UTF-8 string
        session_description [ ]: scalar entries, dtype: UTF-8 string
        session_start_time [ ]: scalar entries, dtype: ASCII string
        timestamps_reference_time [ ]: scalar entries, dtype: ASCII string
```

```
# Alternatively, use h5py directly to glance the file with H5Glance
import h5py
h5file = h5py.File('nwbfiles/sub-408021_ses-758519303_behavior+ophys.nwb', 'r')
H5Glance(h5file)
```

```

└─ nwbfiles/sub-408021_ses-758519303_behavior+ophys.nwb
    └─ acquisition
```

Fig. 3: *h5glance* visualization of an example NWB file.

VSCode HDF5 plugin

Much like the [jupyterlab-h5web](#) plugin, the [H5Web Vscode plugin](#) provides an H5Web Viewer to browse HDF5 files in VSCode. [vscode-h5web](#)

Note: NWB typically uses the *.nwb file extension, which is not associated by default with the H5Web plugin. To open an NWB file you can either use right click -> Open with... -> H5Web (any extension) or you can use VS Code's `workbench.editorAssociations` setting to set H5Web as the default editor for additional extensions:

```
"workbench.editorAssociations": {
  "*.nwb": "h5web.viewer",
},
```

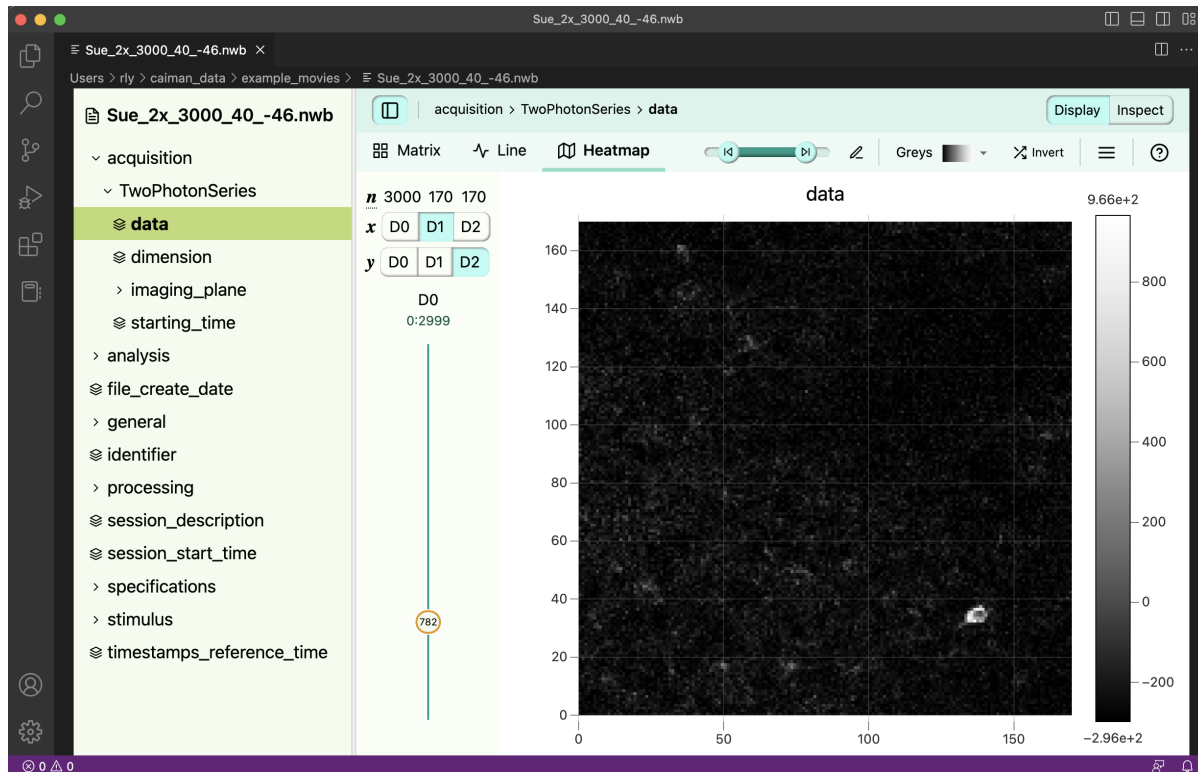


Fig. 4: `vscode-h5web` visualization of an example NWB file.

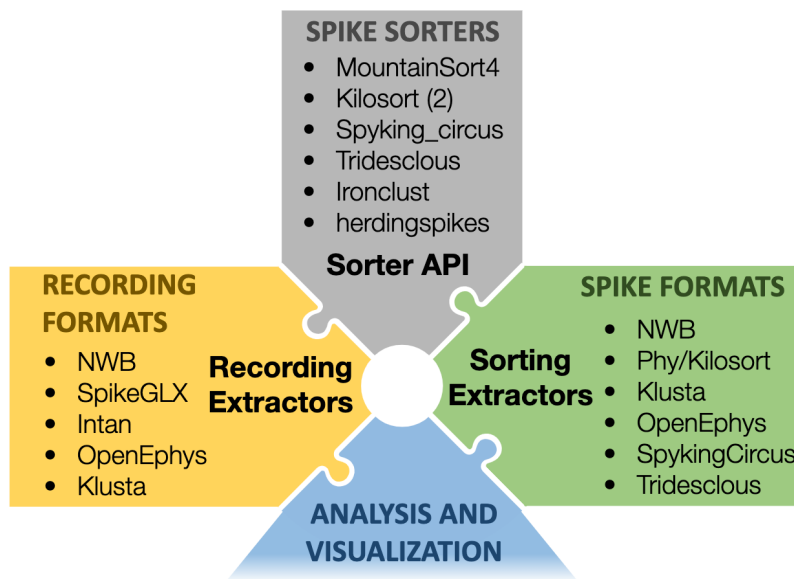
6.6 SpikeInterface

SpikeInterface is a collection of Python modules designed to improve the accessibility, reliability, and reproducibility of spike sorting and all its associated computations. [Video tutorial](#) [Demo](#) [Notebook](#) [Docs](#) [Source](#)

With SpikeInterface, users can:

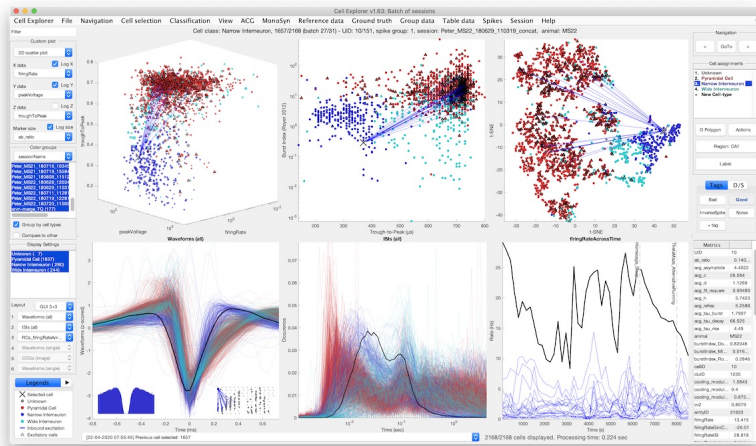
- read/write many extracellular file formats.
- pre-process extracellular recordings.

- run many popular, semi-automatic spike sorters.
- post-process sorted datasets.
- compare and benchmark spike sorting outputs.
- compute quality metrics to validate and curate spike sorting outputs.
- visualize recordings and spike sorting outputs.
- export report and export toPhy
- offer a powerful Qt-based viewer in separate package spikeinterface-gui
- have some powerful sorting components to build your own sorter.



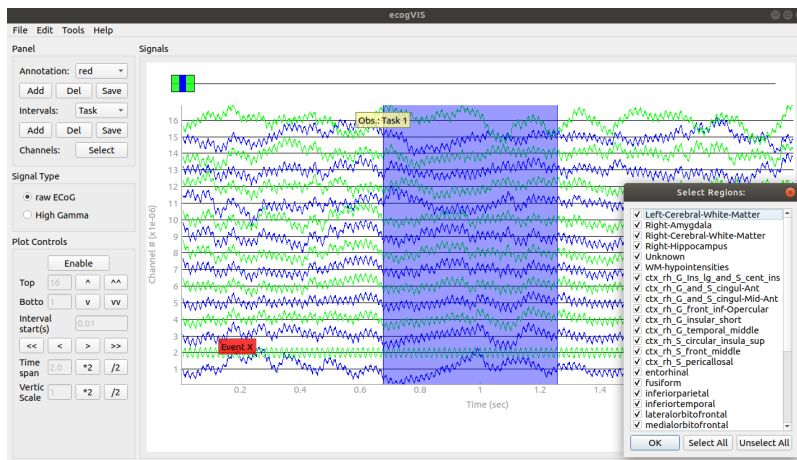
6.7 CellExplorer

CellExplorer is a graphical user interface (GUI), a standardized processing module and data structure for exploring and classifying single cells acquired using extracellular electrodes. [NWB Tutorial Intro Video](#) [Video Tutorial Docs](#) [Source](#).



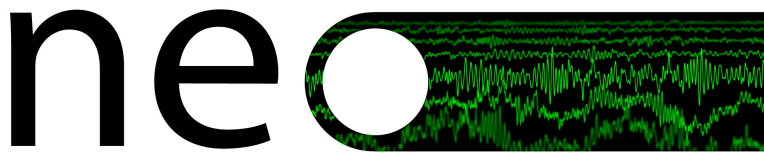
6.8 EcogVIS

EcogVIS is a Python-based, time series visualizer for Electrocorticography (ECoG) signals stored in NWB files. EcogVIS makes it intuitive and simple to visualize ECoG signals from selected channels, brain regions, make annotations and mark intervals of interest. Signal processing and analysis tools will soon be added. [Source](#).



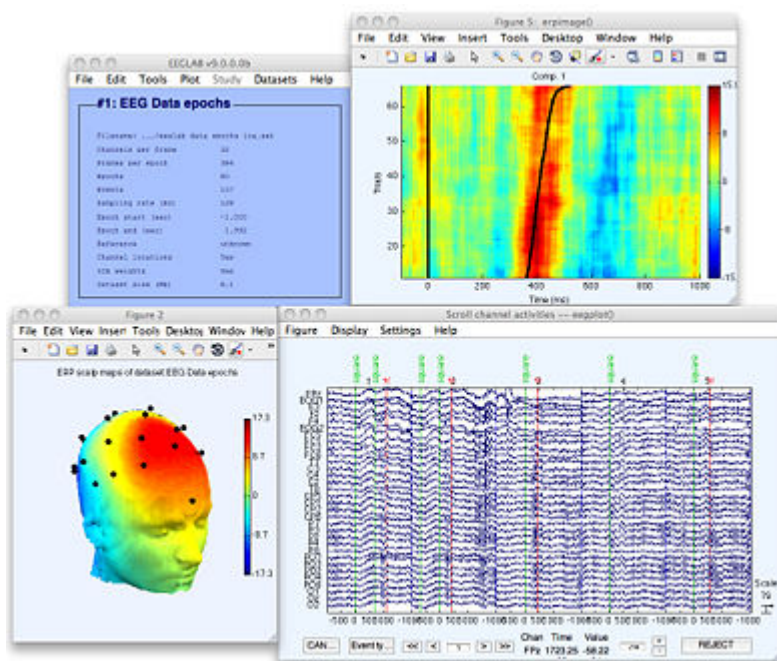
6.9 Neo

Neo Neo is a Python package for working with electrophysiology data in Python, together with support for reading and writing a wide range of neurophysiology file formats (see the [list of supported formats](#)). The goal of Neo is to improve interoperability between Python tools for analyzing, visualizing and generating electrophysiology data, by providing a common, shared object model. In order to be as lightweight a dependency as possible, Neo is deliberately limited to representation of data, with no functions for data analysis or visualization. [Docs](#) [Neo NWBIO](#) [Example Website](#) [Source](#).



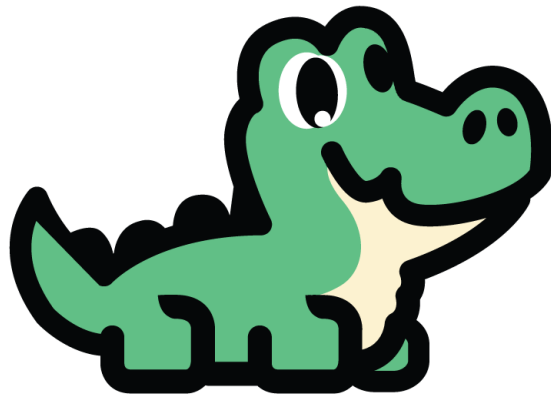
6.10 EEGLAB

EEGLAB is a MATLAB-based electrophysiology (ECoG/iEEG/EEG) software to process and visualize data stored in NWB files. After installing the NWB-IO EEGLAB plugins, time series, channel information, spike timing, and event information may be imported, processed, and visualized using the EEGLAB graphical interface or scripts. MEF3, EDF, and Brain Vision Exchange format may also be converted to NWB files (and vice versa). [Docs](#) [Source](#) [NWB-io EEGLAB plugin](#)



6.11 CalmAn

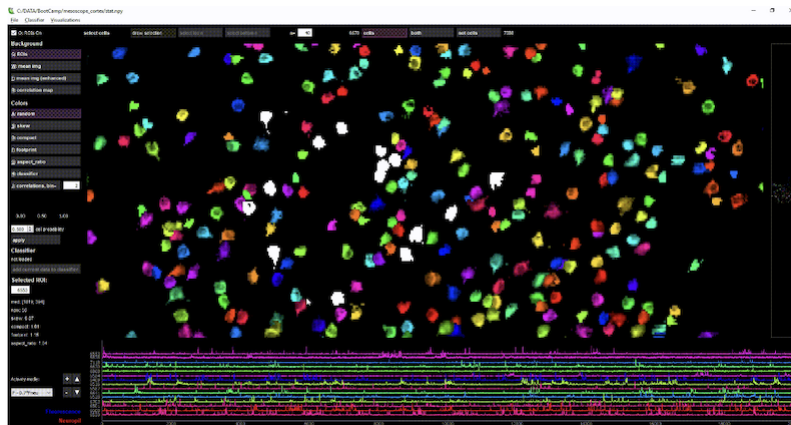
CalmAn is a computational toolbox for large scale Calcium Imaging Analysis, including movie handling, motion correction, source extraction, spike deconvolution and result visualization. CalmAn now supports reading and writing data in NWB 2.0. [NWB Demo](#) [Second Demo Video](#) [Tutorial](#) [Docs](#) [Source](#).



Caiman

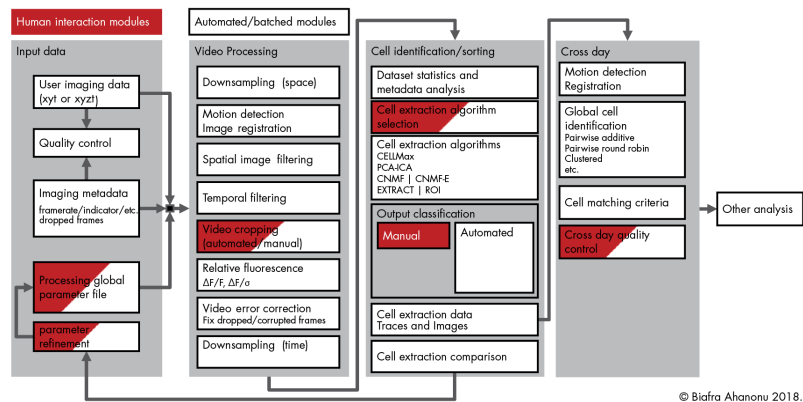
6.12 suite2p

suite2p is an imaging processing pipeline written in Python . *suite2p* includes modules for 1) Registration, 2) Cell detection, 3) Spike detection, and a 4) Visualization GUI. [Video Tutorial Docs Source](#).



6.13 CIAtah

CIAtah (pronounced cheetah; formerly calciumImagingAnalysis) is a Matlab software package for analyzing one- and two-photon calcium imaging datasets. [Video tutorial](#) [Docs](#) [Source](#).

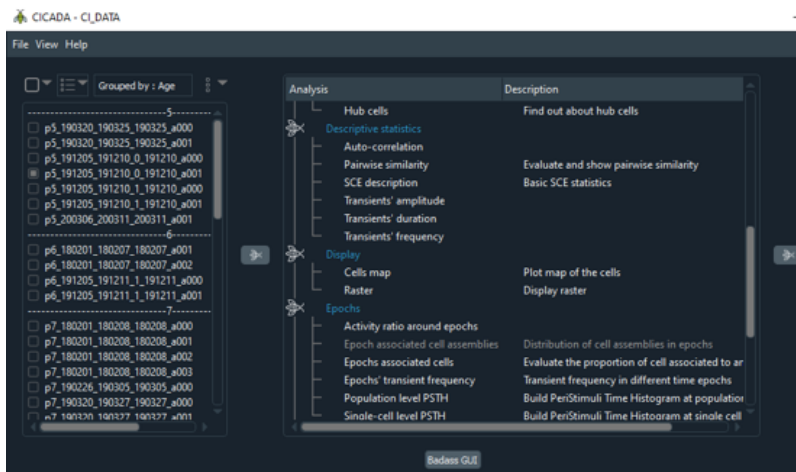


6.14 EXTRACT

EXTRACT is a Tractable and Robust Automated Cell extraction Tool for calcium imaging, which *extracts* the activities of cells as time series from both one-photon and two-photon Ca^{2+} imaging movies. EXTRACT makes minimal assumptions about the data, which is the main reason behind its high robustness and superior performance. [Source](#) [NWB tutorials](#) [Publication](#)

6.15 CICADA

CICADA is a Python pipeline providing a graphical user interface (GUI) that allows the user to visualize, explore and analyze Calcium Imaging data contained in NWB files. [Docs](#) [Source](#) [Video Demo](#) [Cite](#)



6.16 OptiNiSt

OptiNiSt (Optical Neuroimage Studio) is a GUI based workflow pipeline tools for processing two-photon calcium imaging data. [Source Docs](#)

OptiNiSt helps researchers try multiple data analysis methods, visualize the results, and construct the data analysis pipelines easily and quickly on GUI. OptiNiSt's data-saving format follows NWB standards.

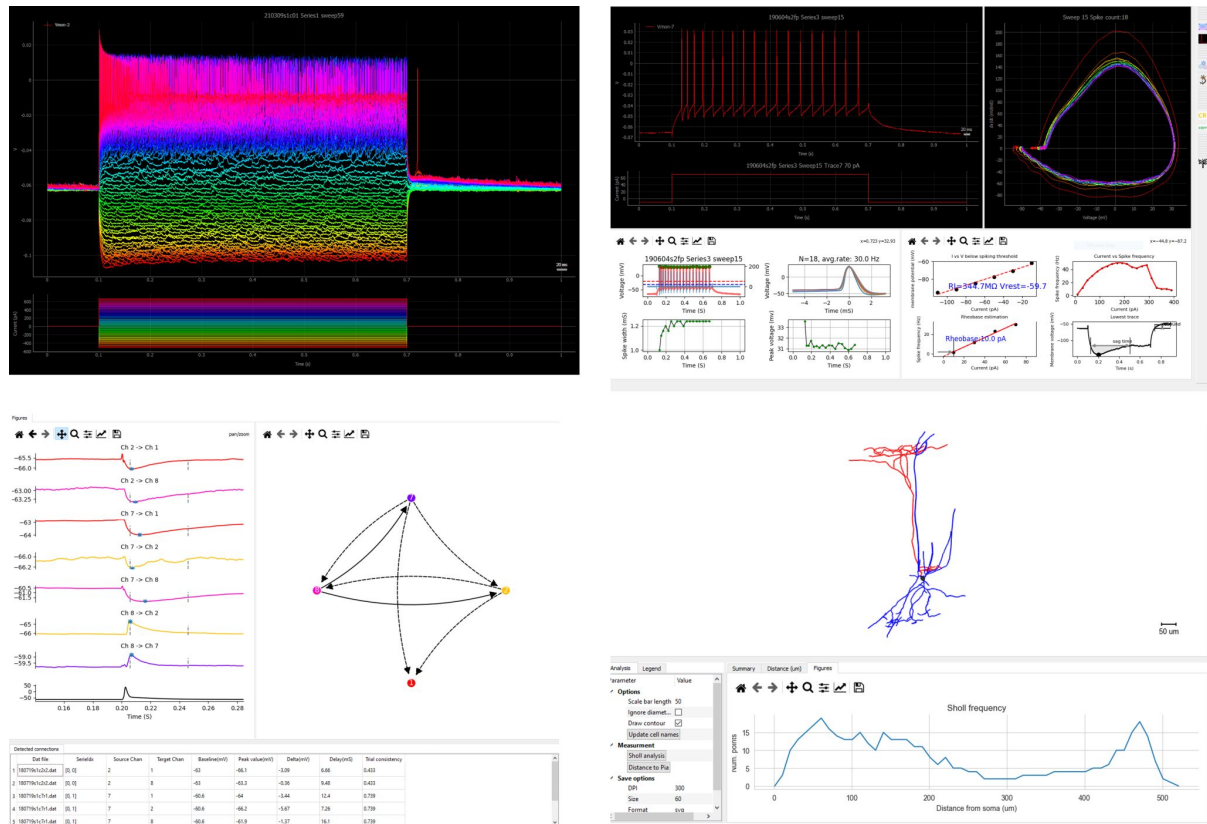
OptiNiSt also supports reproducibility of scientific research, standardization of analysis protocols, and developments of novel analysis tools as plug-in.

6.17 GraFT

GraFT (Graph-Filtered Temporal) is a signal extraction method for spatio-temporal data. GraFT uses a diffusion map to learn a graph over spatial pixels that enables for stochastic filtering of learned sparse representations over each pixel's time-trace. The sparse representations are modeled as in a hierarchical dictionary learning framework with correlated decompositions over the graph. [GitHub repo](#) [NWB tutorials](#) [Cite](#)

6.18 PatchView

PatchView is a GUI tool to perform data analysis and visualization on multi channel whole-cell recording (multi-patch) data, including firing pattern analysis, mini-event analysis, synaptic connection detection, morphological analysis and more. [Documentation](#) [Source](#) [Publication](#)



6.18.1 NWB (NeurodataWithoutBorders) APIs

PatchView reads Heka or Axon format files for patch-clamp data, and uses [PyNWB](#) to export to NWB files.

The code snippet below shows how to convert data from an original Heka .dat file to an extended *NWBFile* object, query the content, and generate a visualization.

```
from patchview.utilitis.PVdat2NWB import dat2NWB

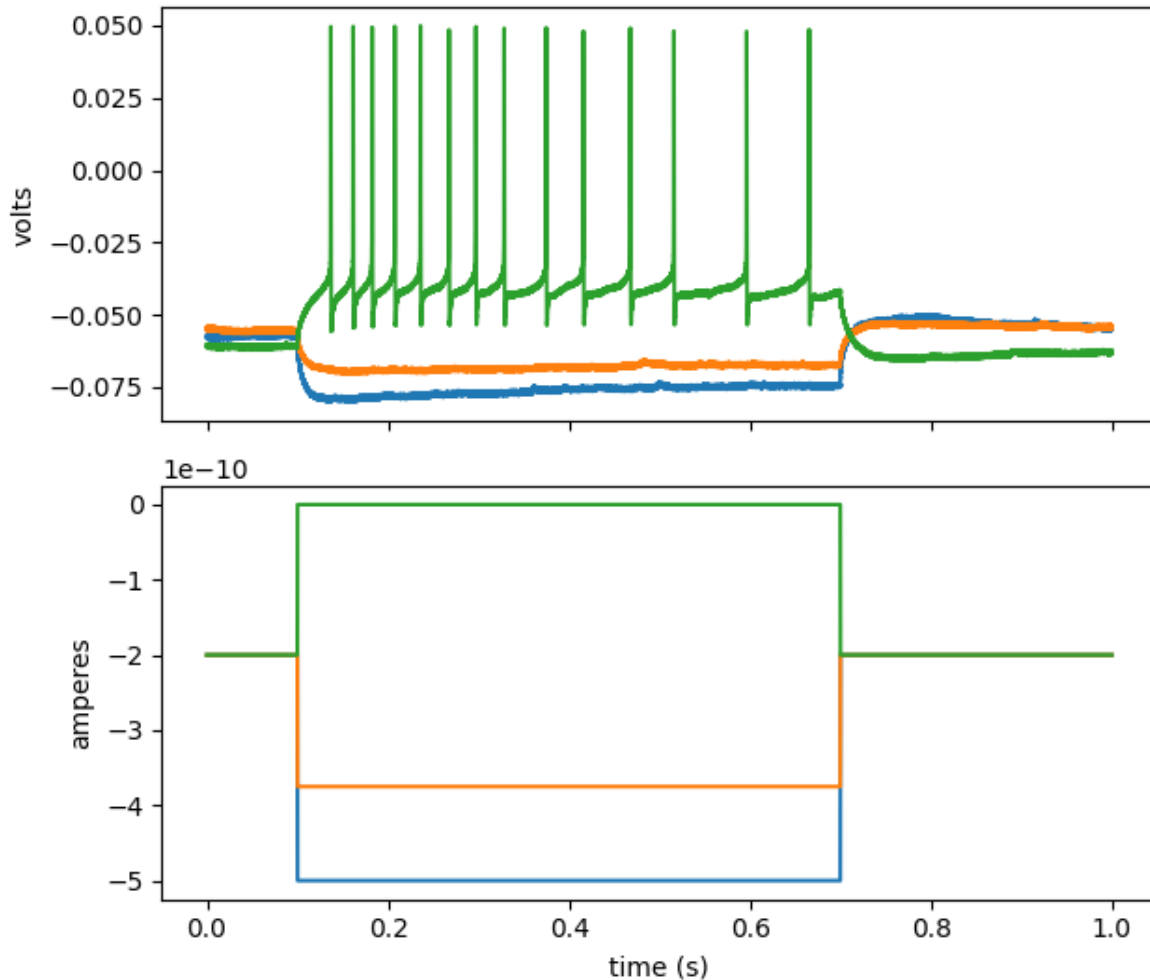
nwbData = dat2NWB('test_singleFP.dat', [0,0]) # extract group 0, series 0 from the file
print(nwbData.fields.keys())

## get number of sweeps
print(f'Number of sweeps: {nwbData.getNumberOfSweeps()}')

## nwbData is an extended pynwb NWB object
stim, resp = nwbData.getSweep(0) # get one sweep's stimulus and responses

## with additional plotting functions.
nwbData.plotSweep([0, 5, 20]) # plot sweep number 0, 5, 20
```

The last command in the block above should give you a figure like this:



6.19 DeepLabCut

DeepLabCut is an efficient method for 2D and 3D markerless pose estimation based on transfer learning with deep neural networks that achieves excellent results (i.e. you can match human labeling accuracy) with minimal training data (typically 50-200 frames). We demonstrate the versatility of this framework by tracking various body parts in multiple species across a broad collection of behaviors. [Documentation](#)

DeepLabCut has developed [DLC2NWB](#), a Python package for converting from their native output format to NWB. This library uses the NWB extension [ndx-pose](#), which aims to provide a standardized format for storing pose estimation data in NWB. [ndx-pose](#) was developed initially to store the output of DeepLabCut in NWB, but is also designed to store the output of general pose estimation tools.

6.20 SLEAP

SLEAP is an open source deep-learning based framework for multi-animal pose tracking. It can be used to track any type or number of animals and includes an advanced labeling/training GUI for active learning and proofreading. [Documentation](#)

SLEAP has developed [NDXPoseAdaptor](#), an adaptor class within SLEAP for importing and exporting NWB files. Users can either use the SLEAP GUI to import/export NWB files or use the high-level API `Labels.export_nwb` to programmatically export to the NWB format. This adaptor uses the NWB extension `ndx-pose`, which aims to provide a standardized format for storing pose estimation data in NWB.

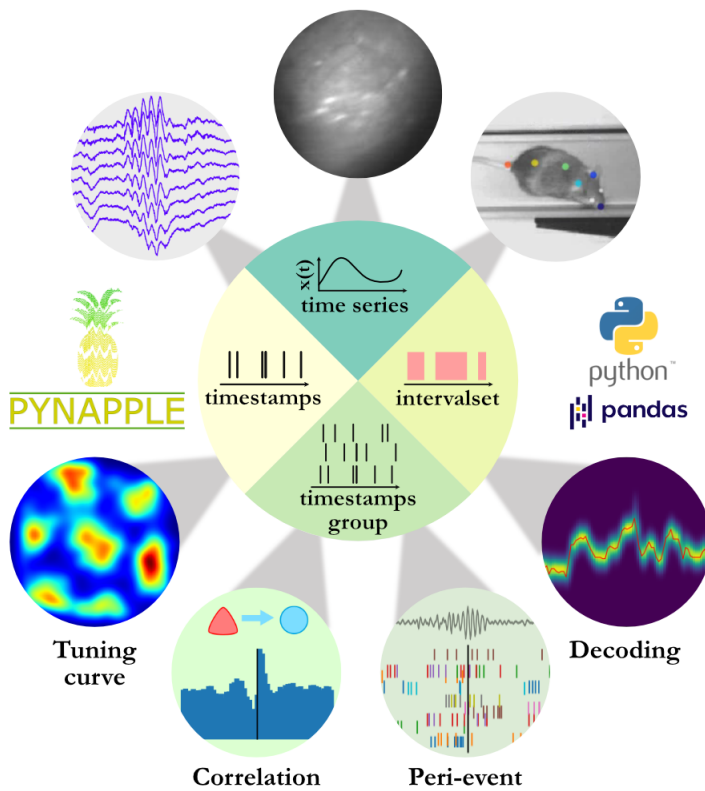
6.21 CEBRA



CEBRA is a machine-learning method that can be used to compress time series in a way that reveals otherwise hidden structures in the variability of the data. It excels on behavioural and neural data recorded simultaneously, and it can decode activity from the visual cortex of the mouse brain to reconstruct a viewed video. [website](#) [paper](#) [GitHub repo](#) [Docs](#) [NWB tutorial](#).

6.22 pynapple

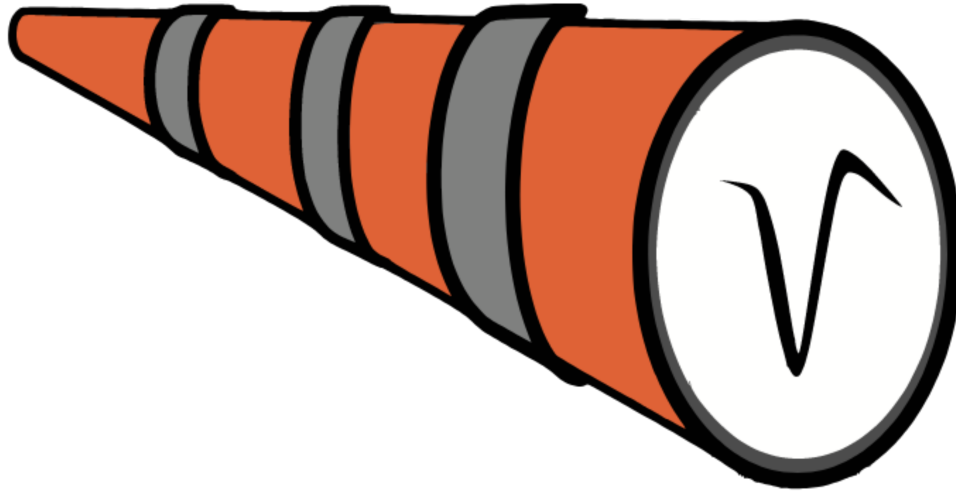
pynapple is a unified toolbox for integrated analysis of multiple data sources. Designed to be “plug & play”, users define and import their own time-relevant variables. Supported data sources include, but are not limited to, electrophysiology, calcium imaging, and motion capture data. Pynapple contains integrated functions for common neuroscience analyses, including cross-correlograms, tuning curves, decoding and perievent time histogram. [Docs](#) [DANDI Demo](#) [Source](#) [Twitter](#)



6.23 Spyglass

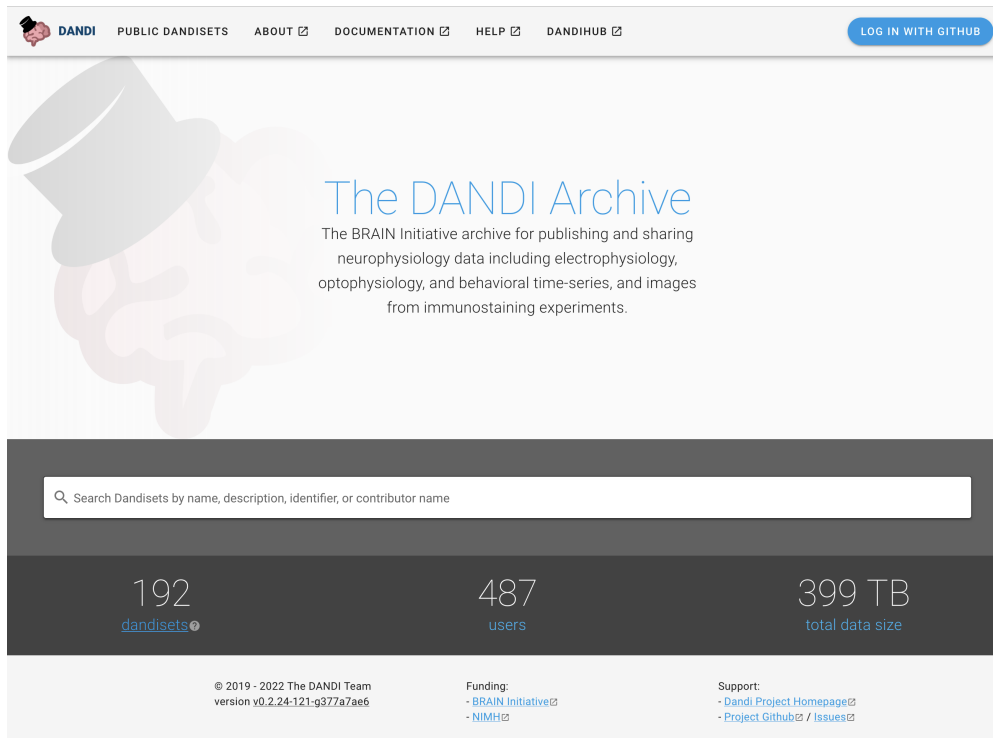
Spyglass is a framework for building your analysis workflows with a focus on reproducibility and sharing. *Spyglass* uses community-developed open-source projects such as NWB and DataJoint to manage data in a shareable format, build and run pipelines and store their inputs and outputs in a relational database, and generate visualizations of analysis results for sharing with the community. It comes with ready-to-use pipelines for spike sorting, LFP analysis, position processing, and fitting of state space models for decoding variables of interest from neural data. [Docs](#) [Source](#).

Spyglass



6.24 DANDI

DANDI (Distributed Archives for Neurophysiology Data Integration) is the NIH BRAIN Initiative archive for publishing and sharing neurophysiology data including electrophysiology, optophysiology, and behavioral time-series, and images from immunostaining experiments. [Online Archive Docs Source](#).



DANDI is:

- An open data archive to submit neurophysiology data for electrophysiology, optophysiology, and behavioral time-series, and images from immunostaining experiments.
- A persistent, versioned, and growing collection of standardized datasets.
- A place to house data to collaborate across research sites.
- Supported by the BRAIN Initiative and the AWS Public dataset programs.

DANDI provides significant benefits:

- A FAIR (Findable, Accessible, Interoperable, Reusable) data archive to house standardized neurophysiology and associated data.
- Rich metadata to support search across data.
- Consistent and transparent data standards to simplify data reuse and software development. We use the Neurodata Without Borders, Brain Imaging Data Structure, Neuroimaging Data Model (NIDM), and other BRAIN Initiative standards to organize and search the data.
- The data can be accessed programmatically allowing for software to work directly with data in the cloud.
- The infrastructure is built on a software stack of open source products, thus enriching the ecosystem.

6.25 DataJoint

DataJoint is an open-source project for defining and operating computational data pipelines—sequences of steps for data acquisition, processing, and transformation. Some [DataJoint Elements](#) support automatic conversion to NWB [Export element_array_ephys to NWB](#)

This page is a collection of tools we are cataloging as a convenience reference for NWB users. This is not a comprehensive list of NWB tools. Many of these tools are built and supported by other groups, and are in active development. If you would like to contribute a tool, please see the instructions [here](#).

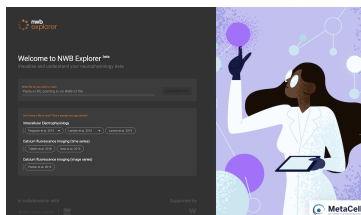
6.26 Exploring NWB Files



NWB Widgets is a library of widgets for visualization NWB data in a Jupyter notebook (or lab). The widgets make it easy to navigate through the hierarchical structure of the NWB file and visualize specific data elements. It is designed to work out-of-the-box with NWB 2.0 files and to be easy to extend. [Source Docs](#)



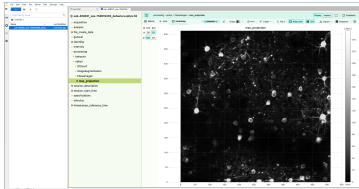
Neurosift provides interactive neuroscience visualizations in the browser. Neurosift caters to both individual users through its local mode, allowing the visualization of views directly from your device, as well as a remote access function for presenting your findings to other users on different machines. [Source](#)



NWB Explorer is a web application developed by MetaCell for reading, visualizing and exploring the content of NWB 2 files. The portal comes with built-in visualization for many data types, and provides an interface to a jupyter notebook for custom analyses and open exploration. [Online](#)

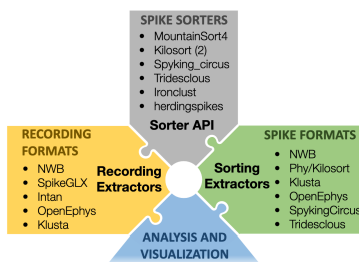


Nwbview is a cross-platform software with a graphical user interface to display the contents of the binary NWB file format. It is written in Rust for high-performance, memory safety and ease of deployment. Its features include the ability to display the contents of the NWB file in a tree structure. It displays voltage and current recordings data in interactive plots. The tabular data or the text data present in the NWB can be displayed in a scalable window. [Docs](#) [Source](#).

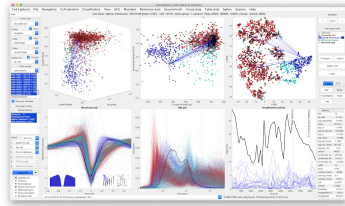


HDF Tools: There are a broad range of useful tools for inspecting and browsing HDF5 files. For example, HDFView and HDF5 plugins for Jupyter or VSCode provide general visual tools for browsing HDF5 files. In addition, the HDF5 library ships with a range of command line tools that can be useful for developers (e.g., *h5ls* and *h5dump* to introspect, *h5diff* to compare, or *h5copy* and *h5repack* to copy HDF5 files). While these tools do not provide NWB-specific functionality, they are useful (mainly for developers) to debug and browse NWB HDF5 files. [HDFView](#) [HDF5 CLI tools](#) [vscode-h5web](#) [h5glance](#) [jupyterlab-h5web](#) [jupyterlab-hdf5](#)

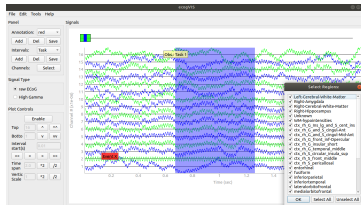
6.27 Extracellular Electrophysiology Tools



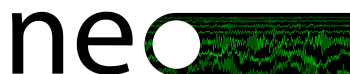
SpikeInterface is a collection of Python modules designed to improve the accessibility, reliability, and reproducibility of spike sorting and all its associated computations. [Video tutorial](#) [Demo Notebook](#) [Docs](#) [Source](#)



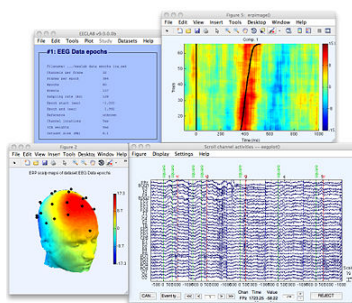
CellExplorer is a graphical user interface (GUI), a standardized processing module and data structure for exploring and classifying single cells acquired using extracellular electrodes. [NWB Tutorial Intro Video](#) [Video Tutorial Docs](#) [Source](#).



EcogVIS is a Python-based, time series visualizer for Electrocorticography (ECoG) signals stored in NWB files. EcogVIS makes it intuitive and simple to visualize ECoG signals from selected channels, brain regions, make annotations and mark intervals of interest. Signal processing and analysis tools will soon be added. [Source](#).



Neo Neo is a Python package for working with electrophysiology data in Python, together with support for reading and writing a wide range of neurophysiology file formats (see the [list of supported formats](#)). The goal of Neo is to improve interoperability between Python tools for analyzing, visualizing and generating electrophysiology data, by providing a common, shared object model. In order to be as lightweight a dependency as possible, Neo is deliberately limited to representation of data, with no functions for data analysis or visualization. [Docs](#) [Neo NWBIO Example](#) [Website](#) [Source](#).

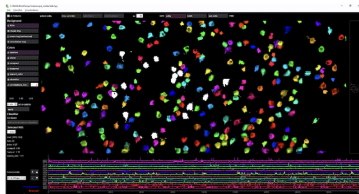


EEGLAB is a MATLAB-based electrophysiology (ECoG/iEEG/EEG) software to process and visualize data stored in NWB files. After installing the NWB-IO EEGLAB plugins, time series, channel information, spike timing, and event information may be imported, processed, and visualized using the EEGLAB graphical interface or scripts. MEF3, EDF, and Brain Vision Exchange format may also be converted to NWB files (and vice versa). [Docs](#) [Source](#) [NWB-io EEGLAB plugin](#)

6.28 Optical Physiology Tools



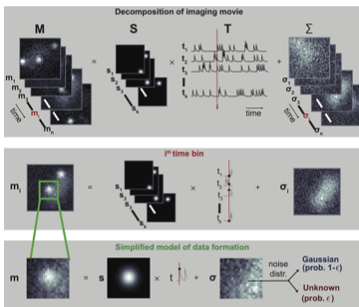
CalMan is a computational toolbox for large scale Calcium Imaging Analysis, including movie handling, motion correction, source extraction, spike deconvolution and result visualization. CaImAn now supports reading and writing data in NWB 2.0. [NWB Demo](#) [Second Demo](#) [Video](#) [Tutorial](#) [Docs](#) [Source](#).



suite2p is an imaging processing pipeline written in Python . suite2p includes modules for 1) Registration, 2) Cell detection, 3) Spike detection, and a 4) Visualization GUI. [Video](#) [Tutorial](#) [Docs](#) [Source](#).



CIAtah (pronounced cheetah; formerly calciumImagingAnalysis) is a Matlab software package for analyzing one- and two-photon calcium imaging datasets. [Video](#) [tutorial](#) [Docs](#) [Source](#).



EXTRACT is a Tractable and Robust Automated Cell extraction Tool for calcium imaging, which *extracts* the activities of cells as time series from both one-photon and two-photon Ca²⁺ imaging movies. EXTRACT makes minimal assumptions about the data, which is the main reason behind its high robustness and superior performance. [Source](#) [NWB tutorials](#) [Publication](#)

6.30 Behavior

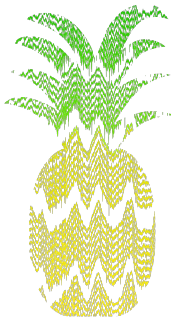
DeepLabCut is an efficient method for 2D and 3D markerless pose estimation based on transfer learning with deep neural networks that achieves excellent results (i.e. you can match human labeling accuracy) with minimal training data (typically 50-200 frames). We demonstrate the versatility of this framework by tracking various body parts in multiple species across a broad collection of behaviors. [Documentation](#)

SLEAP is an open source deep-learning based framework for multi-animal pose tracking. It can be used to track any type or number of animals and includes an advanced labeling/training GUI for active learning and proofreading. [Documentation](#)

6.31 Data Analysis Toolbox

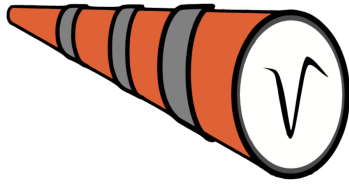


CEBRA is a machine-learning method that can be used to compress time series in a way that reveals otherwise hidden structures in the variability of the data. It excels on behavioural and neural data recorded simultaneously, and it can decode activity from the visual cortex of the mouse brain to reconstruct a viewed video. [website](#) [paper](#) [GitHub repo](#) [Docs](#) [NWB tutorial](#).



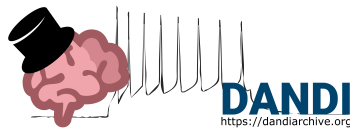
pynapple is a unified toolbox for integrated analysis of multiple data sources. Designed to be “plug & play”, users define and import their own time-relevant variables. Supported data sources include, but are not limited to, electrophysiology, calcium imaging, and motion capture data. Pynapple contains integrated functions for common neuroscience analyses, including cross-correlograms, tuning curves, decoding and perievent time histogram. [Docs](#) [DANDI Demo](#) [Source](#) [Twitter](#)

Spyglass



Spyglass is a framework for building your analysis workflows with a focus on reproducibility and sharing. *Spyglass* uses community-developed open-source projects such as NWB and DataJoint to manage data in a shareable format, build and run pipelines and store their inputs and outputs in a relational database, and generate visualizations of analysis results for sharing with the community. It comes with ready-to-use pipelines for spike sorting, LFP analysis, position processing, and fitting of state space models for decoding variables of interest from neural data. [Docs Source](#).

6.32 Data Archive, Publication, and Management



DANDI (Distributed Archives for Neurophysiology Data Integration) is the NIH BRAIN Initiative archive for publishing and sharing neurophysiology data including electrophysiology, optophysiology, and behavioral time-series, and images from immunostaining experiments. [Online Archive Docs Source](#).



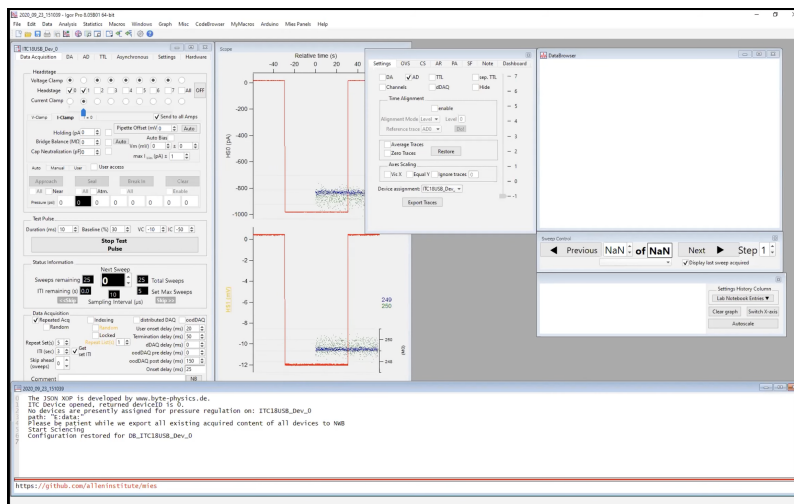
DataJoint is an open-source project for defining and operating computational data pipelines—sequences of steps for data acquisition, processing, and transformation. Some [DataJoint Elements](#) support automatic conversion to NWB [Export element_array_ephys to NWB](#)

Note: Disclaimer: Reference herein to any specific product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement, recommendation, or favoring by the NWB development team, United States Government or any agency thereof, or The Regents of the University of California. Use of the NeurodataWithoutBorders name for endorsements is prohibited.

ACQUISITION AND CONTROL TOOLS

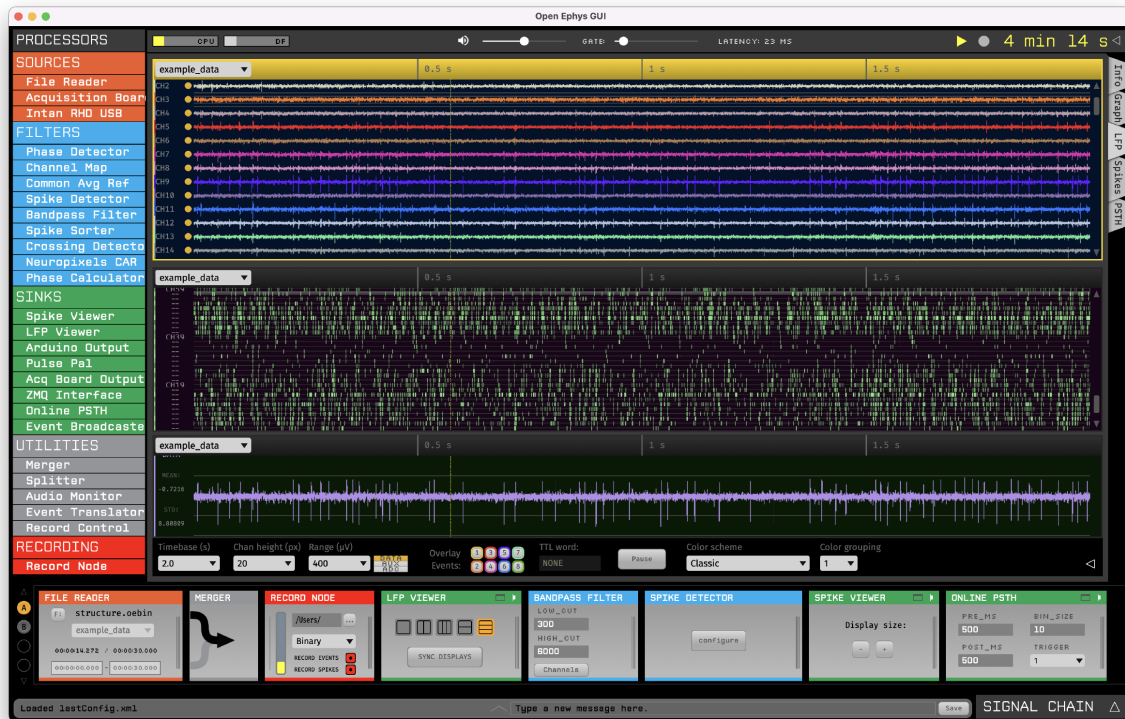
7.1 MIES

MIES is a sweep based data acquisition tool written in Igor Pro. *MIES* has three primary user interfaces: 1) the *WaveBuilder* to generate stimulus sets 2) the *DA_Ephys* GUI to control and observe data acquisition in real time, and 3) the *DataBrowser* to browse acquired data. All three interfaces are intended to be operated in parallel. [Video tutorial](#) [MIES NWB Module Docs](#) [Source](#).



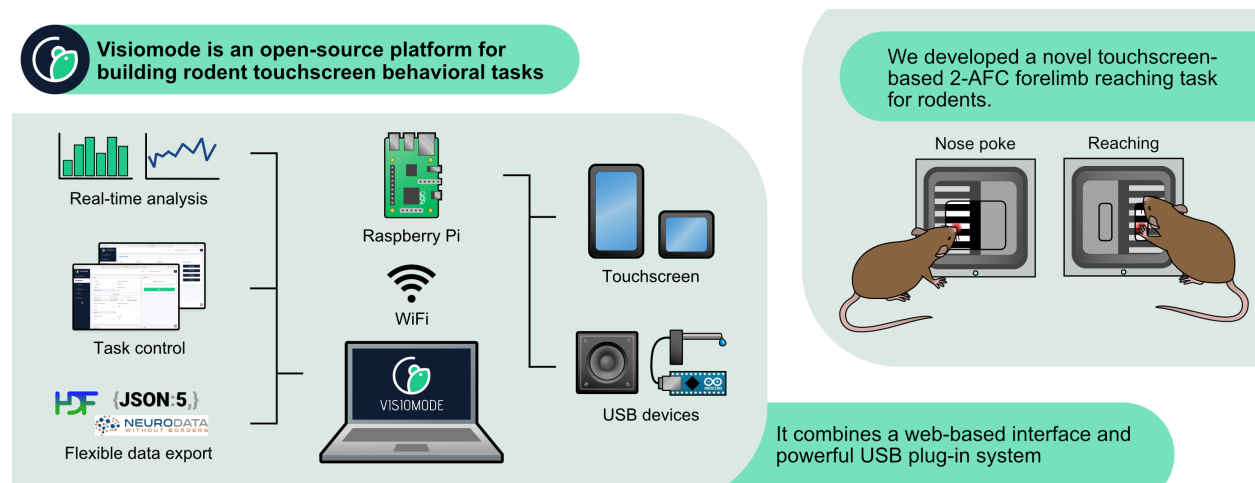
7.2 OpenEphys GUI

OpenEphys GUI is a modular, open-source software for extracellular electrophysiology. Built by neuroscientists, for neuroscientists, *OpenEphys GUI* has all the features needed to acquire and visualize electrophysiology data, while making it easy to add new modules written in C++. The GUI allows the user to configure processing pipelines by mixing and matching modules. Using the NWB format plugin, users can record data directly in NWB format via the *OpenEphys GUI*. — [OpenEphys GUI: Docs Website](#) [Source](#) — [NWB Plugin for OpenEphys: Docs](#) [Source](#)



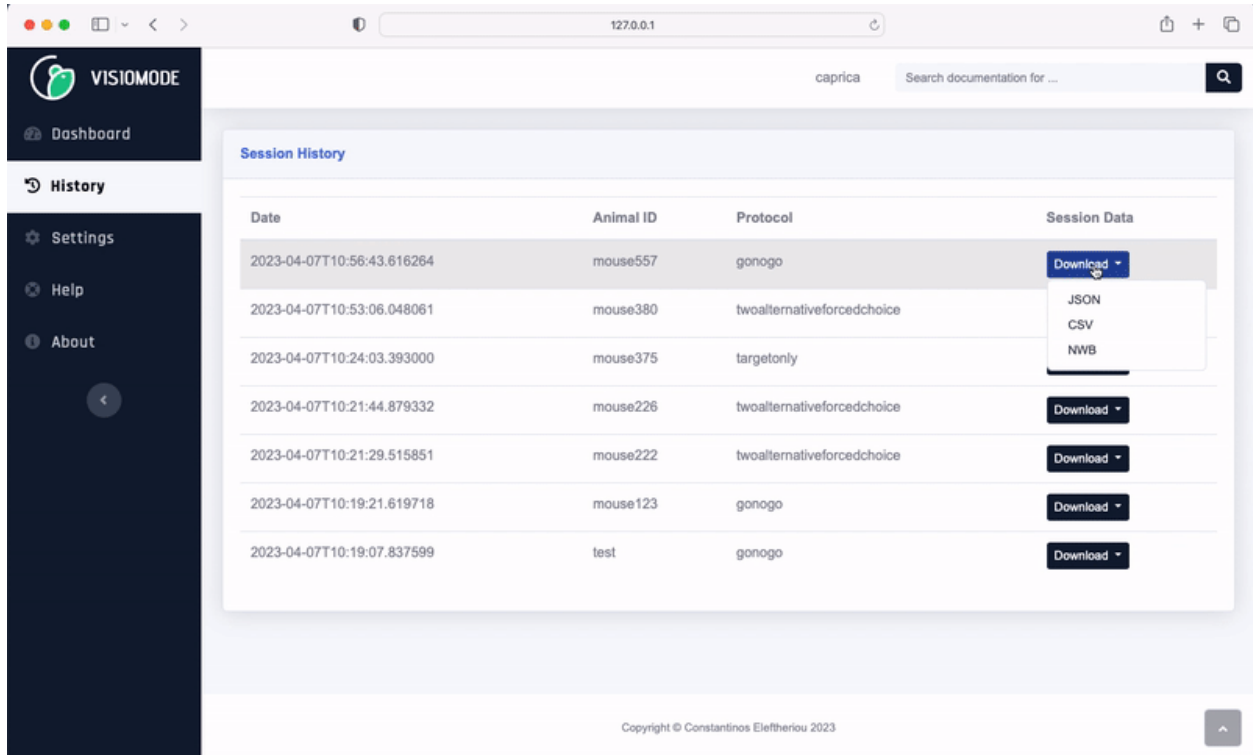
7.3 Visiocode

Visiocode is an open-source platform for building touchscreen-based behavioral tasks for rodents. It leverages the inherent flexibility of touchscreens to offer a simple yet adaptable software and hardware platform. Visiocode is built on the Raspberry Pi computer combining a web-based interface and powerful plug-in system with an operant chamber that can be adapted to generate a wide range of behavioral tasks. [Docs](#) [Source](#) [Publication](#)



7.3.1 Compatibility with NWB

Visiocode session data can be exported to the NWB format directly from the web interface. Navigate to the “History” tab, choose the session you wish to export and select “NWB” from the “Download” dropdown menu. The NWB file will be downloaded to your computer.



Visiocode stores behavioural data under **trials** (docs). The **trials** table contains the following columns:

- **start_time**: the time at which the trial started
- **stop_time**: the time at which the trial ended
- **stimulus**: identifier of the stimulus presented during the trial
- **cue_onset**: the time at which the cue was presented
- **response**: type of response (e.g. touch on left or right side of the screen)
- **response_time**: the time of the response (i.e. the time at which the animal touched the screen)
- **pos_x**: the x-coordinate of the touch
- **pos_y**: the y-coordinate of the touch
- **dist_x**: the touch drag distance in the x-direction
- **dist_y**: the touch drag distance in the y-direction
- **outcome**: the outcome of the trial (e.g. correct or incorrect)
- **correction**: whether the trial was a correction trial (if using)
- **sdt_type**: signal detection theory classification of trial in visual discrimination tasks (if using)

The exported NWB file can then be combined with neurophysiological recordings by linking recording data from different files [as described in the NWB documentation](#). Please take care to synchronize the timestamps of the behavioural

and neurophysiological data before linking them, by recalculating the timestamps relative to the reference time of the behaviour file. For example:

```
from pynwb import NWBHDF5IO, TimeSeries

# Load the Visiocode NWB file
io_behavior = NWBHDF5IO("/path/to/visiocode-behavior.nwb", "r")
nwbfile_behavior = io_behavior.read()

# Load an NWB file with neurophysiological data
io_neurophys = NWBHDF5IO("/path/to/neurophys.nwb", "r")
nwbfile_neurophys = io_neurophys.read()

# Recalculate the timestamps of the neurophysiological data relative
# to the reference start time in the behavior file
timestamp_offset = (
    nwbfile_neurophys.session_start_time - nwbfile_behavior.session_start_time
).total_seconds()

recalc_timestamps = [
    timestamp - timestamp_offset
    for timestamp in nwbfile_neurophys.acquisition["DataTimeSeries"].timestamps
]

# Link the neurophysiological data to the behaviour file
neurophys_timeseries = TimeSeries(
    name="DataTimeSeries",
    data=nwbfile_neurophys.acquisition["DataTimeSeries"].data, # Link to original data
    timestamps=recalc_timestamps, # Remember to add the recalculated timestamps!
    description="Neurophysiological data",
    ...
)

nwbfile_behavior.add_acquisition(neurophys_timeseries)

# Export data to a new "linker" NWB file
io_linker = NWBHDF5IO("/path/to/linker-behavior+phys.nwb", "w")
io_linker.write(nwbfile_behavior, link_data=True)

# Clean up
io_behavior.close()
io_neurophys.close()
io_linker.close()
```


7.4 ArControl

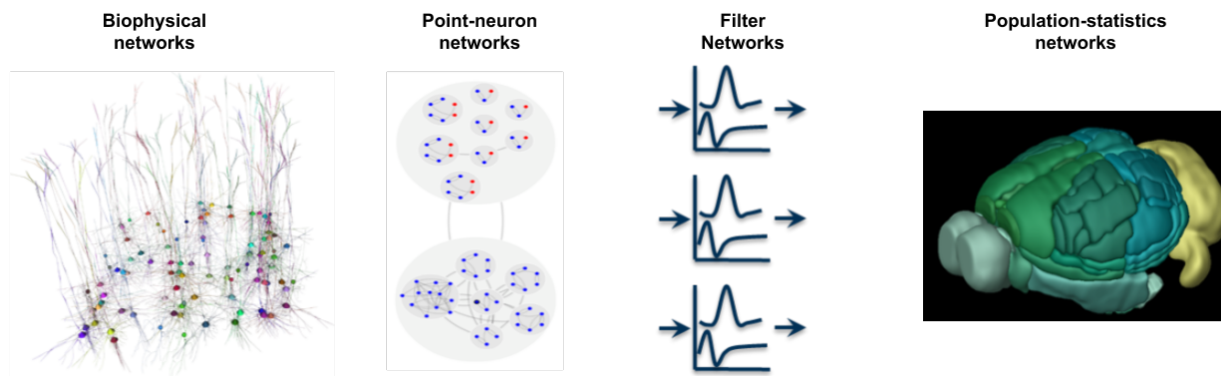
ArControl is a Arduino based digital signals control system. A special application for ArControl is to establish a animal behavioral platform (as Skinner box), which control devices to deliver stimulation and monitor behavioral response. ArControl is also useful to generate Optogenetic TTL pulses. While ArControl does currently not support recording data directly in NWB, it provides tools for converting ArControl data to NWB. [NWB conversion tool Demo ArControl Source](#)



ArControl

7.5 Brain Modeling Toolkit (BMTK)

Brain Modeling Toolkit (BMTK), SONATA, and Visual Neuronal Dynamics (VND) are mutually integrated software tools that are particularly suited to support large-scale bio-realistic brain modeling, but are applicable to a variety of neuronal modeling applications. BMTK is a suite for building and simulating network models at multiple levels of resolution, from biophysically-detailed, to point-neuron, to population-statistics approaches. The modular design of BMTK allows users to easily work across different scales of resolution and different simulation engines using the same code interface. The model architecture and parameters, as well as simulation configuration, input, and output are stored together in the SONATA data format. Models and their output activity can then be visualized with the powerful rendering capabilities of VND. [Docs Tutorial NWB Spike Stimulus Source](#)

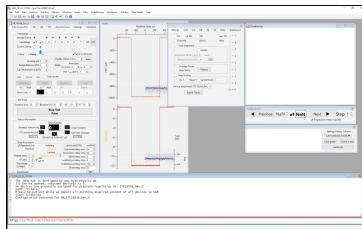


Reference:

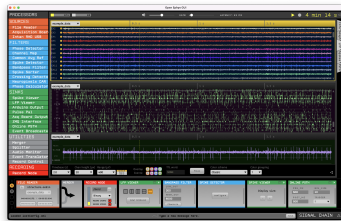
Dai et al. Brain Modeling Toolkit: An open-source software suite for multiscale modeling of brain circuits. PLoS Comput Biol 16(11): e1008386. <https://doi.org/10.1371/journal.pcbi.1008386>

This page is a collection of tools we are cataloging as a convenience reference for NWB users. This is not a comprehensive list of NWB tools. Many of these tools are built and supported by other groups, and are in active development. If you would like to contribute a tool, please see the instructions [here](#).

7.6 Acquiring Electrophysiology Data



MIES is a sweep based data acquisition tool written in Igor Pro. *MIES* has three primary user interfaces: 1) the WaveBuilder to generate stimulus sets 2) the DA_Ephys GUI to control and observe data acquisition in real time, and 3) the DataBrowser to browse acquired data. All three interfaces are intended to be operated in parallel. [Video tutorial](#) [MIES NWB Module Docs](#) [Source](#).



OpenEphys GUI is a modular, open-source software for extracellular electrophysiology. Built by neuroscientists, for neuroscientists, OpenEphys GUI has all the features needed to acquire and visualize electrophysiology data, while making it easy to add new modules written in C++. The GUI allows the user to configure processing pipelines by mixing and matching modules. Using the NWB format plugin, users can record data directly in NWB format via the OpenEphys GUI. — **OpenEphys GUI:** [Docs](#) [Website](#) [Source](#) — **NWB Plugin for OpenEphys:** [Docs](#) [Source](#)

7.7 Controlling and Recoding Behavioral Tasks



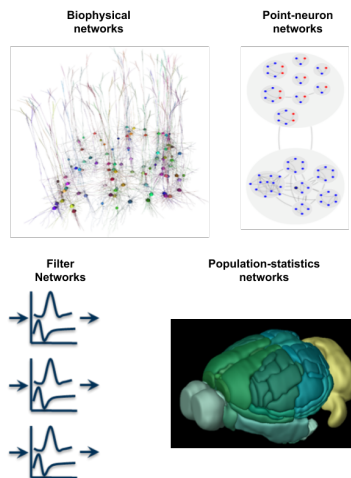
Visiomode is an open-source platform for building touchscreen-based behavioral tasks for rodents. It leverages the inherent flexibility of touchscreens to offer a simple yet adaptable software and hardware platform. Visiomode is built

on the Raspberry Pi computer combining a web-based interface and powerful plug-in system with an operant chamber that can be adapted to generate a wide range of behavioral tasks. [Docs](#) [Source](#) [Publication](#)



ArControl is a Arduino based digital signals control system. A special application for ArControl is to establish a animal behavioral platform (as Skinner box), which control devices to deliver stimulation and monitor behavioral response. ArControl is also useful to generate Optogenetic TTL pulses. While ArControl does currently not support recording data directly in NWB, it provides tools for converting ArControl data to NWB. [NWB conversion tool](#) [Demo](#) [ArControl Source](#)

7.8 Brain Data Modeling and Simulation



Brain Modeling Toolkit (BMTK), SONATA, and Visual Neuronal Dynamics (VND) are mutually integrated software tools that are particularly suited to support large-scale bio-realistic brain modeling, but are applicable to a variety of neuronal modeling applications. BMTK is a suite for building and simulating network models at multiple levels of resolution, from biophysically-detailed, to point-neuron, to population-statistics approaches. The modular design of BMTK allows users to easily work across different scales of resolution and different simulation engines using the same code interface. The model architecture and parameters, as well as simulation configuration, input, and output are stored together in the SONATA data format. Models and their output activity can then be visualized with the powerful rendering capabilities of VND. [Docs](#) [Tutorial](#) [NWB Spike Stimulus](#) [Source](#)

Note: Disclaimer: Reference herein to any specific product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement, recommendation, or favoring by the NWB development team, United States Government or any agency thereof, or The Regents of the University of California. Use of the NeurodataWithoutBorders name for endorsements is prohibited.

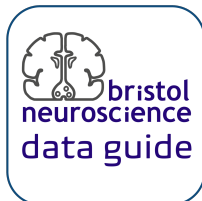
COMMUNITY GALLERY

This page is a collection of community conversion and analysis projects cataloged here as a convenient reference for NWB users. This list is not comprehensive and many of the projects and resources are built and supported by other groups, and are in active development. If you would like to contribute a project or resources, please see the instructions [here](#).

8.1 Data Conversion

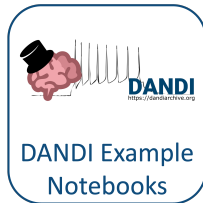


[NeuroConv Catalog](#) is a collection of real-world examples of labs using [NeuroConv](#) to convert their data to NWB files. Each project listed contains a description and a link to an open GitHub repository. Many of the projects listed use advanced customization features beyond what is demonstrated in the core [NeuroConv conversion gallery](#). [NeuroConv Catalog](#)



The Bristol Neuroscience Data Guide includes tutorials for converting data to NWB for [extracellular electrophysiology data](#) and [optical physiology data](#) using both the [PyNWB](#) and [MatNWB](#) APIs for NWB.

8.2 Data Analysis and Reuse



DANDI (Distributed Archives for Neurophysiology Data Integration) maintains a [collection of example notebooks](#) associated with datasets, conference tools, or more generally notebooks that illustrate the use of data on DANDI. [Source](#)

In addition, the neuroscience community is creating examples demonstrating the reuse of NWB data published on DANDI. For example:

- The [INCF working group on NWB](#) has created a [library of MATLAB examples](#) using DANDI datasets authored as MATLAB live scripts. [Source](#)
- [Neuromatch-AJILE12](#) is a package for exploratory analysis of long-term naturalistic human intracranial neural recordings and pose data as part of [Dandiset 000055](#). [Notebook](#) [Source](#) [Paper](#)



The [OpenScope Databook](#) provides scripts and documentation used for brain data analysis and visualization, primarily working with NWB files and the [DANDI](#) archive. Through [Jupyter Book](#), this code is structured as a series of notebooks intended to explain and educate users on how to work with brain data. This resource is provided by the Allen Institute's [OpenScope Project](#), an endeavor of The Allen Institute [Mindscope Program](#). OpenScope is a platform for high-throughput and reproducible neurophysiology open to external scientists to test theories of brain function. [Databook](#) [Source](#)



The [International Brain Laboratory \(IBL\)](#) released a Brainwide Map of neural activity during decision-making, consisting of 547 Neuropixel recordings of 32,784 neurons across 194 regions of the mouse brain. At Cosyne 2023, the IBL team presented an [Introduction to IBL and the Brain-wide map dataset](#) and tutorials on [Using IBL data with NWB](#) and [Using IBL data with ONE](#).

Note: Disclaimer: Reference herein to any specific product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement, recommendation, or favoring by the NWB development team, United States Government or any agency thereof, or The Regents of the University of California. Use of the NeurodataWithoutBorders name for endorsements is prohibited.

FREQUENTLY ASKED QUESTIONS

9.1 Using NWB

9.1.1 Is NWB 2 stable?

Yes! NWB 2.0 was officially released in January 2019, and the schema is stable. A key goal of the NWB endeavour is to ensure that NWB 2 remains accessible. As NWB evolves we strive to ensure that any changes we make do not break backwards compatibility.

9.1.2 I would like to use NWB. How do I get started?

See the *Converting neurophysiology data to NWB* page for more information.

9.1.3 How do I cite NWB 2 in my research?

Oliver Rübel, Andrew Tritt, Ryan Ly, Benjamin K. Dichter, Satrajit Ghosh, Lawrence Niu, Pamela Baker, Ivan Soltesz, Lydia Ng, Karel Svoboda, Loren Frank, Kristofer E. Bouchard, The Neurodata Without Borders ecosystem for neurophysiological data science, Oct. 2022, eLife 11:e78362. <https://doi.org/10.7554/eLife.78362>

9.1.4 How do I install PyNWB?

See the *Installing PyNWB* for details.

9.1.5 How do I install MatNWB?

See the *MatNWB documentation* for details.

9.1.6 What is the difference between PyNWB and nwb-schema?

PyNWB is the Python reference read/write API for the current NWB 2.x format. The *nwb-schema-repo* is used to manage development of the data standard schema. End-users who want to use NWB typically do not need to worry about the nwb-schema repo as the current schema is always installed with the corresponding API (whether it is PyNWB for Python or MatNWB for Matlab).

9.1.7 How do I read NWB files in different programming languages?

For Python and Matlab we recommend using the [PyNWB](#) and [MatNWB](#) reference APIs. To get started see also the [Reading NWB Files](#) page.

If you are using other programming languages (such as R, C/C++, Julia, Java, or Javascript) you can use the standard HDF5 readers that are available for these languages. In contrast to the NWB native API (PyNWB, MatNWB), the HDF5 readers are not aware of NWB schema details. This can make writing valid NWB files in other languages (without PyNWB and MatNWB) tricky, but for read they nonetheless provide access to the full data. For write, applications (e.g., MIES written in Igor) often chose to implement only the parts of the NWB standard that are relevant to the particular application.

9.1.8 Where should I publish my NWB files?

You can publish NWB files in many different archives. Funding or publishing requirements may require you to publish your data in a particular archive. Many such archives already support NWB. If not, please let us know and we will be happy to assist you and the archive developers with supporting the NWB standard.

If you are free to publish data wherever, we would recommend [DANDI](#). DANDI has built-in support for NWB that validates NWB files, automatically extracts key metadata to enable search, and provides tools for interactively exploring and analyzing NWB files. Furthermore, it provides an efficient interface for publishing neuroscience datasets on the TB scale, and can do so for free.

9.1.9 Who can I contact for questions?

- **General questions:** For general questions, use the [NWB Helpdesk](#).
- **Bugs and issues:** To contribute, or to report a bug, create an issue on the appropriate GitHub repository. To find relevant repositories see the [Glossary of Core NWB Tools](#) and [Accessing NWB Sources](#) pages.
- **Stay tuned:** To receive updates about NWB at large, sign up for the [NWB mailing list](#).

For details, please also review our [Contributing Guidelines](#).

9.2 Alternative data standards and formats

9.2.1 How does NWB 2.0 compare to other standards?

See page: [comparison-to-other-standards](#)

9.2.2 Why use HDF5 as the primary backend for NWB?

See page: [why_hdf5](#)

Are you aware of the Rossant blog posts about moving away from HDF5?

Yes. See above for our motivations for using HDF5. Many of the technical issues raised in the blog post have been addressed and in our experience HDF5 is reliable and is performing well for NWB users.

Why not just use HDF5 on its own?

The goal of NWB is to package neurophysiology data with metadata sufficient for reuse and reanalysis of the data by other researchers. HDF5 enables users to provide very rich metadata, sufficient for describing neuroscience data for this purpose. The problem with HDF5 on its own is that it is just too flexible. Without a schema, files could be missing key metadata like the sampling rate of a time series. Furthermore, different labs that use HDF5 would use completely different methods for organizing and annotating experiment data. It would be quite difficult to aggregate data across labs or build common tools without imposing structure on the HDF5 file. This is the purpose of the NWB schema. The NWB schema formalizes requirements that ensure reusability of the data and provides a common structure that enables interoperability across the global neurophysiology community. Users can use extensions to build from schema and describe new types of neurophysiology data.

Why is it discouraged to write videos from lossy formats (mpg, mp4) to internal NWB datasets?

The NWB team strongly encourages that users do NOT package videos of natural behavior or other videos that are stored in lossy compressed formats, such as MP4, in the NWB file. Instead, these data can be included in the NWB file as an `ImageSeries` that has an external file reference to the relative path of the MP4 file. An MP4 file is significantly smaller in file size compared to both the uncompressed frame-by-frame video data (often by about 10X) and such data compressed using algorithms available in HDF5 (e.g., `gzip`, `blosc`). Users *could* store the binary data read from an MP4 file in the data array of an `ImageSeries`, but this data cannot be read as a video directly from the HDF5 file. The binary data can only be read as a video by first writing the data into a new MP4 file and then using a software tool like `FFmpeg` to read the MP4 file. This creates a burden on the data user to have enough space on their filesystem to write the MP4 file and have an appropriate decompression tool installed to decode and read the MP4 file. As a result, putting compressed video data inside an HDF5 file reduces the accessibility of that data and limits its reuse.

9.3 NWB 1 vs 2

9.3.1 What has changed between NWB 1 and 2?

See the [release notes of the NWB format schema](#) for details about changes to the format schema. For details about changes to the specification language see the specification language release notes. With regard to software, NWB 2 marks a full reboot and introduced with `PyNWB`, `MatNWB`, `HDMF docutils`, `nwb-schema` etc. several new packages and repositories while tools, e.g., `api-python`, that were created for NWB:N 1.x have been deprecated.

9.3.2 Does PyNWB support NWB:N 1.0.x files?

`PyNWB` includes the `pynwb/legacy` module which supports reading of NWB:N 1.0.x files from popular data repositories, such as the [Allen Cell Types Atlas](#). For NWB:N 1.0.x files from other sources the millage may vary in particular when files are not fully format compliant, e.g., include arbitrary custom data or are missing required data fields.

9.3.3 What is the difference between NWB and NWB:N?

Neurodata Without Borders (NWB) started as a project by the Kavli Foundation with the goal to enhance accessibility of neuroscience data across the community. The intent was to have a broad range of projects under the NWB umbrella. The Neurodata Without Borders: Neurophysiology (NWB:N) data standard was intended to be the first among many such projects. As NWB:N is currently the only project under the NWB umbrella, the terms “NWB” and “NWB:N” are often used interchangeably.

9.3.4 What is the difference between PyNWB and api-python?

[PyNWB](#) is the Python reference read/write API for the current NWB 2.x format. [api-python](#) is a deprecated write-only API designed for NWB:N 1.0.x files. [PyNWB](#) also provides support for reading some NWB:N 1.0.x files from popular data repositories, such as the [Allen Cell Types Atlas](#) via the `pynwb/legacy` module.

ACCESSING NWB SOURCES

All NWB software is available open source via the following GitHub organizations:

- [Neurodata Without Borders](#): All software resources published by the NWB organization are available online as part of our GitHub organization
- [HDMF](#): The HDMF GitHub organization is used to publish all software related to the Hierarchical Data Modeling Framework (HDMF). HDMF has been developed as part of the NWB project and builds the foundation for PyNWB Python API for NWB.
- [NWB Extensions](#): The NWB Extension Catalog Github organization is used to manage all source repositories for the NDX catalog.

NWB SOFTWARE ANALYTICS

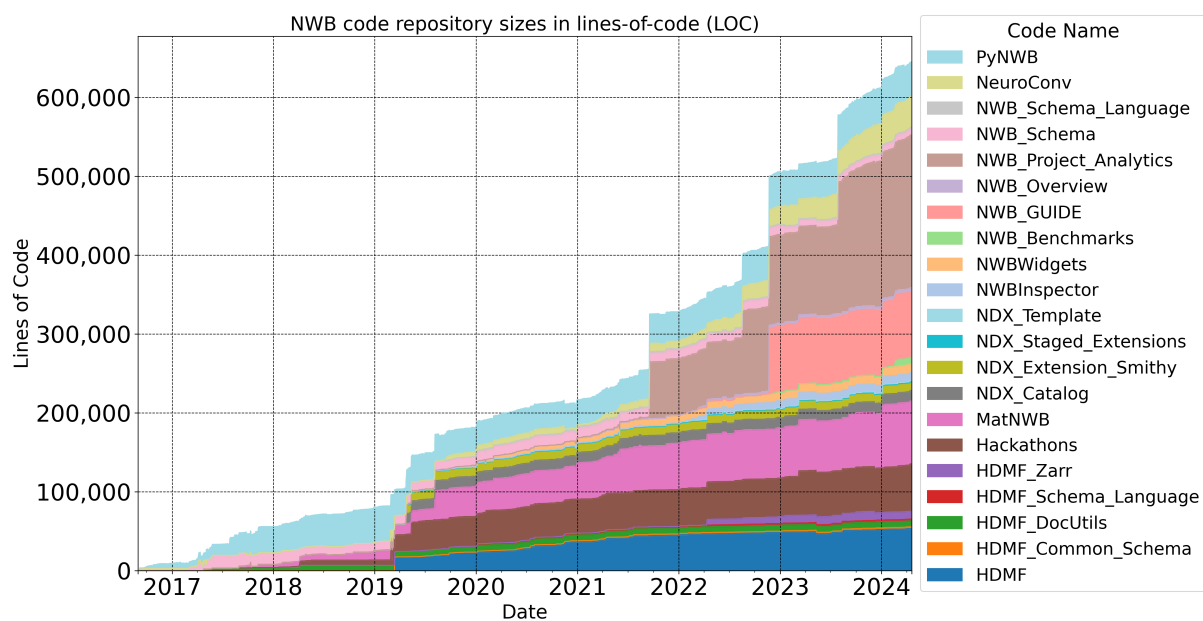
11.1 NWB Code Health Status Board

11.1.1 Social Media

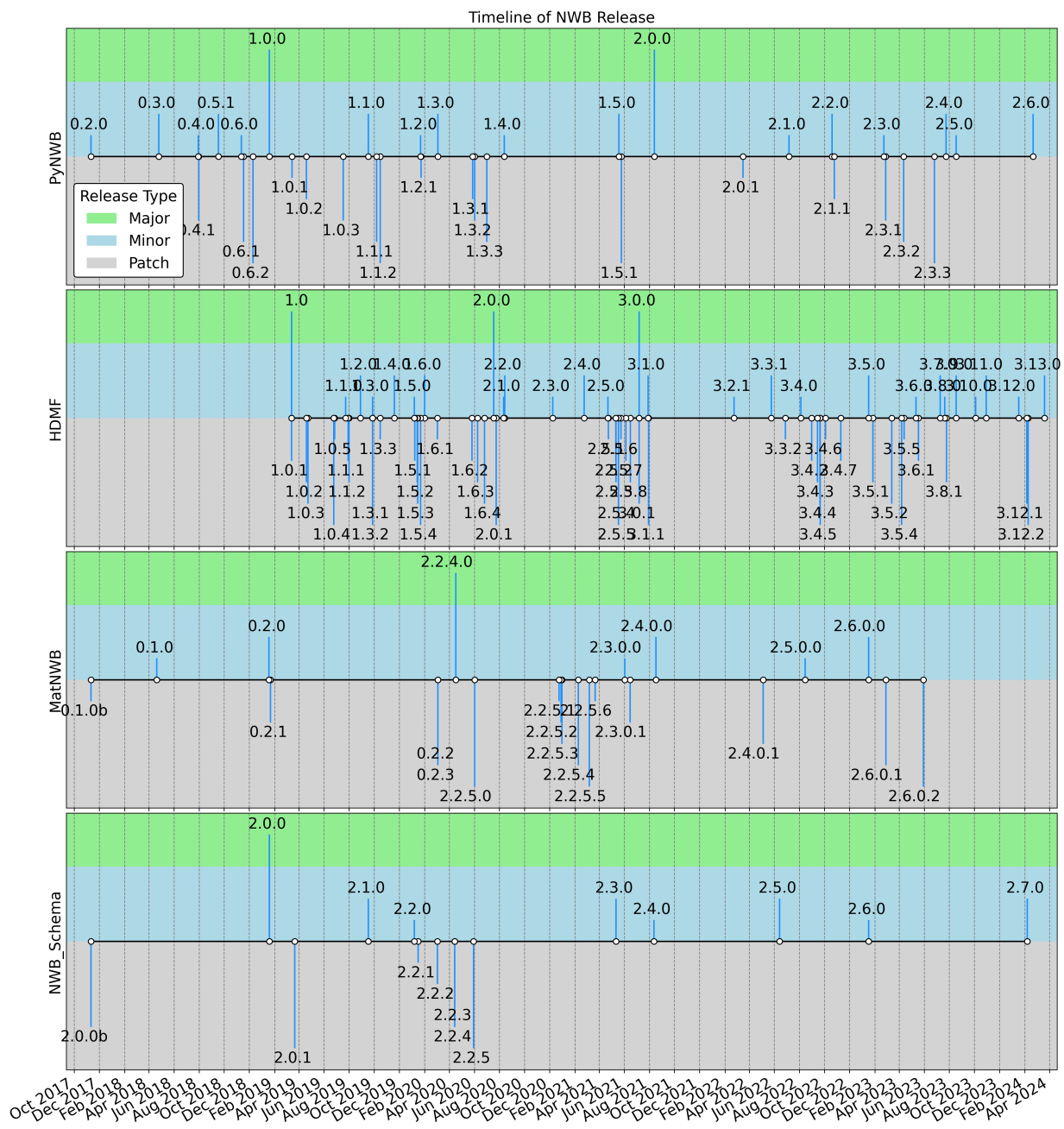
Note: Software health board not included in PDF version. See the HTML version for more details.

11.2 Code Statistics: NWB Core

11.2.1 Lines of Code: All NWB Codes



11.2.2 Release Timeline: NWB APIs and Schema



11.2.3 Contributors

For a listing of all contributors to the various NWB Git repositories see the [contributors.tsv](#) file as part of the [nwb-project-analytics](#) Git repository.

- **Blood Raven** (*a.k.a. codycbakerphd, CodyCBakerPhD, Cody Baker*) : PyNWB: 13, MatNWB: 1, NWBWidgets: 72, NWBInspector: 1618, NWB_GUIDE: 1158, Hackathons: 50, NWB_Benchmarks: 89, NWB_Schema: 14, HDMF: 8, HDMF_Zarr: 1, NeuroConv: 3911
- **!git for-each-ref --format='%c(refname:short)' `git symbolic-ref HEAD`** (*a.k.a. Ben Dichter, bendichter*) : PyNWB: 431, MatNWB: 186, NWBWidgets: 345, NWBInspector: 274, NWB_GUIDE: 7, Hackathons: 252, NWB_Overview: 114, NWB_Schema: 85, NWB_Schema_Language: 9, NWB_Project_Analytics: 4, HDMF: 81, HDMF_Zarr: 4, HDMF_Common_Schema: 3, HDMF_DocUtils: 23, HDMF_Schema_Language: 12, NDX_Template: 9, NDX_Staged_Extensions: 4, NDX_Catalog: 2, NeuroConv: 928
- **Andrew Tritt** : PyNWB: 1958, NWBWidgets: 1, Hackathons: 19, NWB_Schema: 136, HDMF: 88, HDMF_Common_Schema: 43, NDX_Staged_Extensions: 1
- **Oliver Ruebel** : PyNWB: 324, MatNWB: 4, NWBWidgets: 1, NWBInspector: 5, Hackathons: 429, NWB_Benchmarks: 18, NWB_Overview: 122, NWB_Schema: 439, NWB_Project_Analytics: 195, HDMF: 80, HDMF_Zarr: 72, HDMF_Common_Schema: 6, HDMF_DocUtils: 97, NDX_Template: 6, NDX_Staged_Extensions: 1, NDX_Catalog: 17
- **Ryan Ly** (*a.k.a. rly*) : PyNWB: 230, MatNWB: 22, NWBWidgets: 2, NWBInspector: 43, NWB_GUIDE: 81, Hackathons: 346, NWB_Overview: 43, NWB_Schema: 106, NWB_Schema_Language: 11, NWB_Project_Analytics: 7, HDMF: 332, HDMF_Zarr: 14, HDMF_Common_Schema: 44, HDMF_DocUtils: 41, HDMF_Schema_Language: 12, NDX_Template: 195, NDX_Staged_Extensions: 52, NDX_Catalog: 24, NDX_Extension_Smithy: 42, NeuroConv: 4
- **Heberto Mayorquin** (*a.k.a. Heberto, h-mayorquin*) : PyNWB: 6, NWBWidgets: 6, NWBInspector: 8, NWB_Schema: 1, NeuroConv: 1248
- **pre-commit-ci[bot]** : NWBWidgets: 16, NWBInspector: 154, NWB_GUIDE: 589, NWB_Benchmarks: 10, HDMF: 17, NeuroConv: 347
- **Garrett** : NWB_GUIDE: 1074, NeuroConv: 4
- **Lawrence Niu** (*a.k.a. Lawrence, lawrence*) : MatNWB: 916, NWB_Overview: 20
- **Szonja Weigl** (*a.k.a. weiglszonja*) : PyNWB: 10, NWBInspector: 13, Hackathons: 6, NeuroConv: 623
- **Luiz Tauffer** (*a.k.a. luiztauffer, luiz*) : NWBWidgets: 128, NDX_Staged_Extensions: 4, NeuroConv: 491
- **Garrett Michael Flynn** : NWB_GUIDE: 389, Hackathons: 1, NeuroConv: 23
- **Jean-Christophe Fillion-Robin** : PyNWB: 105, MatNWB: 3, Hackathons: 76, NWB_Schema: 1, HDMF_DocUtils: 47, NDX_Template: 37
- **NWB Bot** : NWB_Project_Analytics: 257
- **Saksham Sharda** : NWBWidgets: 14, NeuroConv: 224
- **Matthew Avaylon** (*a.k.a. mavaylon1*) : PyNWB: 27, MatNWB: 2, Hackathons: 2, NWB_Overview: 1, NWB_Schema: 4, HDMF: 70, HDMF_Zarr: 36, HDMF_Common_Schema: 21, HDMF_DocUtils: 5, HDMF_Schema_Language: 7
- **In-vidrio** : MatNWB: 128
- **Doruk Ozturk** (*a.k.a. dorukozturk*) : PyNWB: 108, Hackathons: 1, NWB_Schema: 1
- **Thomas Braun** : PyNWB: 61, Hackathons: 20, NWB_Schema: 3, HDMF: 8, HDMF_Common_Schema: 1, NDX_Template: 3, NWB_1.x_Python: 1

- **nicain** (*a.k.a. nicholasc*) : PyNWB: 69, NWBWidgets: 1, Hackathons: 12, NWB_Schema: 1, HDMF_DocUtils: 4
- **Alessio Buccino** : NWBWidgets: 3, NWB_GUIDE: 1, HDMF_Zarr: 2, NeuroConv: 76
- **Jeff Teeters** (*a.k.a. jeff Teeters, jeffteeters*) : PyNWB: 1, NWB_1.x_Matlab: 4, NWB_1.x_Python: 64
- **nile graddis** (*a.k.a. NileGraddis, nilegraddis*) : PyNWB: 47, NWBWidgets: 3, Hackathons: 14, HDMF: 1, HDMF_DocUtils: 2
- **Isuru Fernando** : NDX_Extension_Smithy: 56
- **Armin Najarpour Foroushani** : NWBWidgets: 46, Hackathons: 1
- **Anthony Scopatz** : NDX_Extension_Smithy: 43
- **Julia Sprenger** : NeuroConv: 37
- **Yaroslav Halchenko** : PyNWB: 7, NWBInspector: 1, Hackathons: 8, NWB_Schema: 13, HDMF: 2, HDMF_Common_Schema: 1, NeuroConv: 4
- **Michael Grauer** (*a.k.a. Mike Grauer, mgrauer*) : PyNWB: 4, Hackathons: 31
- **Tom Davidson** : PyNWB: 21, Hackathons: 11, NWB_Schema: 2
- **sbuergers** : NeuroConv: 33
- **Marius van Niekerk** (*a.k.a. mariusvniekerk*) : NDX_Extension_Smithy: 31
- **Jeremy Magland** : PyNWB: 1, NWBWidgets: 3, NeuroConv: 24
- **Tom Donoghue** (*a.k.a. Tom*) : PyNWB: 5, NWBWidgets: 18, NWBInspector: 4
- **Matteo Cantarelli** : Hackathons: 24
- **NWB Extensions Bot** : NDX_Staged_Extensions: 21, NDX_Extension_Smithy: 2
- **Steph Prince** : PyNWB: 7, Hackathons: 10, NWB_Overview: 1, NWB_Schema: 1, NWB_Schema_Language: 1, HDMF: 3
- **cechava** : MatNWB: 22
- **dependabot[bot]** : PyNWB: 2, NWBInspector: 2, HDMF: 12, HDMF_Zarr: 1, NeuroConv: 5
- **Christopher J. Wright** : NDX_Extension_Smithy: 19
- **Matt McCormick** : NWBWidgets: 9, Hackathons: 6, NDX_Extension_Smithy: 4
- **Mark Cafaro** : MatNWB: 17
- **felixp8** : NeuroConv: 16
- **Justin Kiggins** : PyNWB: 2, Hackathons: 13, NWB_1.x_Python: 1
- **Pamela Baker** : Hackathons: 15
- **yangll0620** : Hackathons: 14
- **Alessandra Trapani** : NWBInspector: 11, NeuroConv: 3
- **Ali Mohebi** : Hackathons: 13
- **David Camp** : PyNWB: 13
- **Darin Erat Sleiter** : PyNWB: 2, HDMF: 10, NDX_Staged_Extensions: 1
- **Lydia Ng** : Hackathons: 13
- **Josh Reichardt** : NDX_Extension_Smithy: 12

- **Nathan Clack** : MatNWB: 12
- **Jonathan Cooper** : NWB_1.x_Python: 11
- **Felix Pei** : NeuroConv: 10
- **Josh Siegle** : Hackathons: 10
- **refraction-ray** : NDX_Extension_Smithy: 10
- **Tuan Pham** (*a.k.a. tuanpham96*) : NDX_Staged_Extensions: 2, NeuroConv: 7
- **kevinalexbrown** (*a.k.a. ls, Kevin*) : PyNWB: 4, Hackathons: 5
- **Filipe Fernandes** (*a.k.a. Filipe*) : NDX_Extension_Smithy: 9
- **Kael Dai** : Hackathons: 9
- **lynnebecker13** : Hackathons: 8
- **Matthias Kastner** : PyNWB: 8
- **Jay R Bolton** : PyNWB: 8
- **Simon Ball** : NeuroConv: 7
- **Steffen Burgers** : NeuroConv: 7
- **Jeremy Delahanty** : Hackathons: 7
- **Jerome Lecoq** : Hackathons: 7
- **Sylvain Takerkart** : Hackathons: 6
- **Ariel Rokem** (*a.k.a. arokem*) : PyNWB: 3, Hackathons: 3
- **fairdataihub-bot** : NWB_GUIDE: 6
- **efinkel** : Hackathons: 6
- **friedsam** (*a.k.a. Claudia*) : NWB_1.x_Matlab: 6
- **Liviu S** : Hackathons: 6
- **Arnaud Delorme** : NWB_Overview: 6
- **Padraig Gleeson** : Hackathons: 6
- **ajgranger** : Hackathons: 5
- **John Yearsley** : PyNWB: 4, NWB_Schema: 1
- **Eric Denovellis** : Hackathons: 4, HDMF: 1
- **Luke Campagnola** : Hackathons: 5
- **Shreejoy Tripathy** : Hackathons: 5
- **Mario Melara** (*a.k.a. Mario*) : PyNWB: 5
- **Jed Perkins** : PyNWB: 5
- **Nand Chandravadia** : Hackathons: 5
- **Vijay Iyer** : MatNWB: 5
- **bergjim** : Hackathons: 4
- **Aaron D. Milstein** : Hackathons: 4
- **kcasimo** : Hackathons: 4

- **Thinh Nguyen** : Hackathons: 4
- **Eric Miller** : PyNWB: 4
- **Xiaoxuan Jia** : Hackathons: 4
- **d-sot** : NWBWidgets: 2, HDMF: 2
- **ehennestad** : MatNWB: 1, Hackathons: 3
- **Abby Dichter** : NWBWidgets: 4
- **Akshay Jaggi** : Hackathons: 4
- **nicthib** : Hackathons: 4
- **zcbo** : Hackathons: 3
- **Jeremy Forest** : Hackathons: 3
- **Marcel Bargull** : NDX_Extension_Smithy: 3
- **matthias** : PyNWB: 3
- **Jens K** : Hackathons: 3
- **Konstantinos** : Hackathons: 3
- **Michael Scheid** : NWBWidgets: 2, Hackathons: 1
- **DSegebarth** : Hackathons: 3
- **Dipterix Wang** : Hackathons: 3
- **neuroelf** : Hackathons: 3
- **Ben Hardcastle** : NWBWidgets: 3
- **Sumner L Norman** : Hackathons: 3
- **John T. Wodder II** : NWBInspector: 3
- **sebiRolotti** : Hackathons: 3
- **shenshan** : Hackathons: 3
- **Borde Sandor** : Hackathons: 3
- **Vincent Prevosto** : MatNWB: 2, NDX_Staged_Extensions: 1
- **David Tingley** : Hackathons: 3
- **Mikkel Elle Lepperød** : Hackathons: 3
- **Maksim Rakitin** : NDX_Extension_Smithy: 3
- **Paul Adkisson** : NeuroConv: 2
- **Marike Reimer** (*a.k.a. MarikeReimer*) : Hackathons: 2
- **Matthew Sit** : Hackathons: 2
- **Roni Choudhury** : Hackathons: 2
- **Eric Thomson** : Hackathons: 2
- **Lowell Umayam** : PyNWB: 2
- **Uwe L. Korn** : NDX_Extension_Smithy: 2
- **vijayi** : MatNWB: 2

- **wuffi** : MatNWB: 1, NeuroConv: 1
- **Nicholas Bollweg** : NDX_Extension_Smithy: 2
- **Zach McKenzie** : PyNWB: 2
- **Vadim Frolov** : NWB_1.x_Matlab: 2
- **jakirkham** : NDX_Extension_Smithy: 2
- **smestern** : Hackathons: 2
- **Min RK** : NDX_Extension_Smithy: 2
- **hajapy** : NDX_Extension_Smithy: 2
- **Biafra Ahanonu** : MatNWB: 2
- **Jason Furmanek** : NDX_Extension_Smithy: 2
- **buijennifer** : NDX_Extension_Smithy: 2
- **ap-** : NDX_Extension_Smithy: 2
- **Jessie Liu** : Hackathons: 2
- **Kyu Hyun Lee** : NWB_Overview: 2
- **Alex Estrada** : MatNWB: 2
- **Henry Schreiner** : NDX_Extension_Smithy: 2
- **Ben Beasley** : HDMF: 1
- **rhuszar** : Hackathons: 1
- **charles** : PyNWB: 1
- **jeylau** : Hackathons: 1
- **cshaley** : NDX_Extension_Smithy: 1
- **IvanSmal** : MatNWB: 1
- **Jonny Saunders** : HDMF_Zarr: 1
- **colleenjg** : Hackathons: 1
- **Satrajit Ghosh** : Hackathons: 1
- **Isla Brooks** : NWB_Schema: 1
- **Brian H. Hu** : NWBWidgets: 1
- **Sylvia Schröder** : Hackathons: 1
- **Dan Millman** : Hackathons: 1
- **Tom Gillespie** : Hackathons: 1
- **Kenneth Dyson** : Hackathons: 1
- **Tomáš Hrnčiar** : HDMF: 1
- **Anil Tuncel** : NWB_Overview: 1
- **atlandau** : Hackathons: 1
- **Vyassa Baratham** : Hackathons: 1
- **Zeta** : NWB_Overview: 1

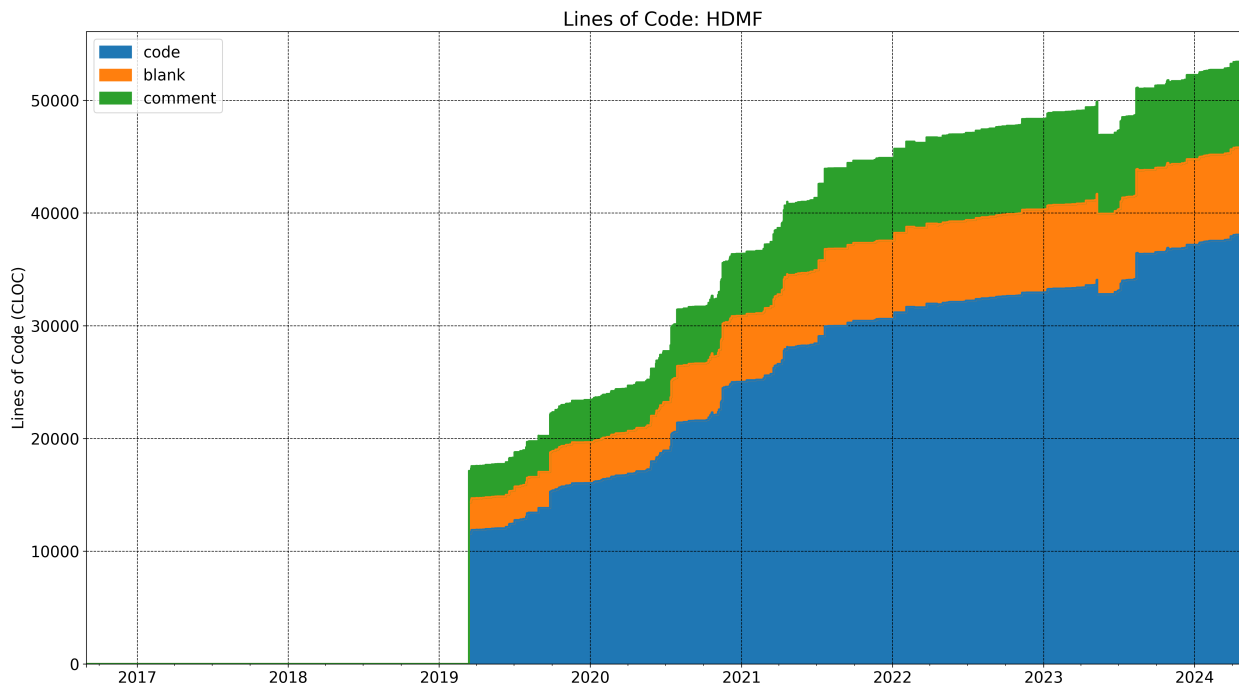
- **Constantinos Eleftheriou** : NWB_Overview: 1
- **actionpotential** : Hackathons: 1
- **Paul LaFosse** : NDX_Staged_Extensions: 1
- **Dav Clark** : NWBWidgets: 1
- **Daniel Aharoni** : Hackathons: 1
- **karnitas** : Hackathons: 1
- **owoolnough** : Hackathons: 1
- **Chris Brozdowski** : NWB_Overview: 1
- **Jonathan Helmus** : NDX_Extension_Smithy: 1
- **NeurekaAI** : NDX_Staged_Extensions: 1
- **Daniel Sprague** : NDX_Staged_Extensions: 1
- **github-actions[bot]** : NWB_GUIDE: 1
- **Liezl Maree** : NWB_Overview: 1
- **Matt Earnshaw** : Hackathons: 1
- **Chenggang Chen** : Hackathons: 1
- **Pierre Le Merre** : Hackathons: 1
- **Iouri Khramtsov** : PyNWB: 1
- **Igoirandlopez** : Hackathons: 1
- **Michael Wulf** : Hackathons: 1
- **Guillaume Viejo** : NWB_Overview: 1
- **Mike Sarahan** : NDX_Extension_Smithy: 1
- **Hannah Choi** : Hackathons: 1
- **Laurelrr** : Hackathons: 1
- **nwb-schema Upstream** : MatNWB: 1
- **Evan Lyall** : Hackathons: 1
- **Derek** : Hackathons: 1
- **Deepti Mittal** : Hackathons: 1
- **nileg** : Hackathons: 1
- **Nicholas Nadeau, P.Eng., AVS** : PyNWB: 1
- **Hamidreza-Alimohammadi** : NDX_Staged_Extensions: 1
- **dcamp_lbl** : PyNWB: 1

11.3 Code Statistics: NWB Tools

Select a tool or code repository below to view the corresponding code statistics:

11.3.1 HDMF

Lines of Code



Release History

Additional Information

- Source: <https://github.com/hdmf-dev/hdmf.git> (main branch = dev)
- Docs: <https://hdmf.readthedocs.io>
- Logo: https://raw.githubusercontent.com/hdmf-dev/hdmf/dev/docs/source/hdmf_logo.png

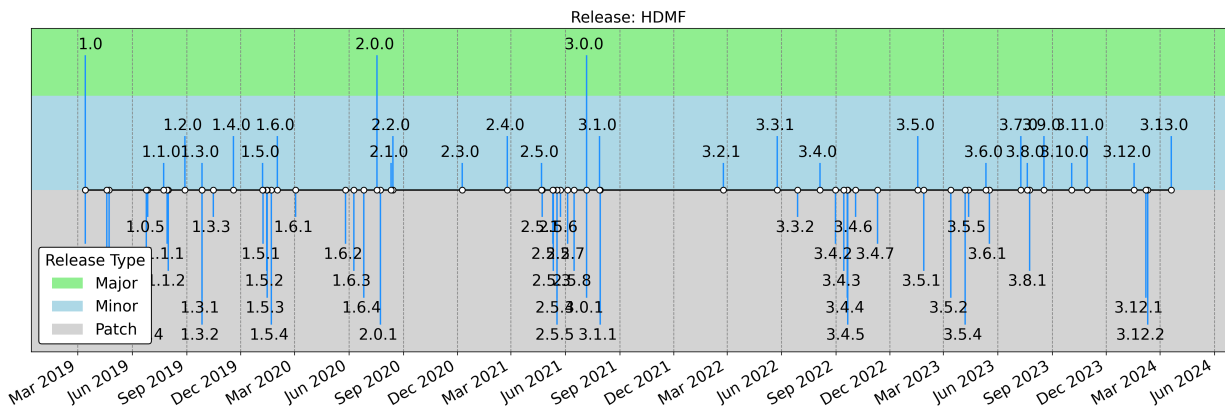
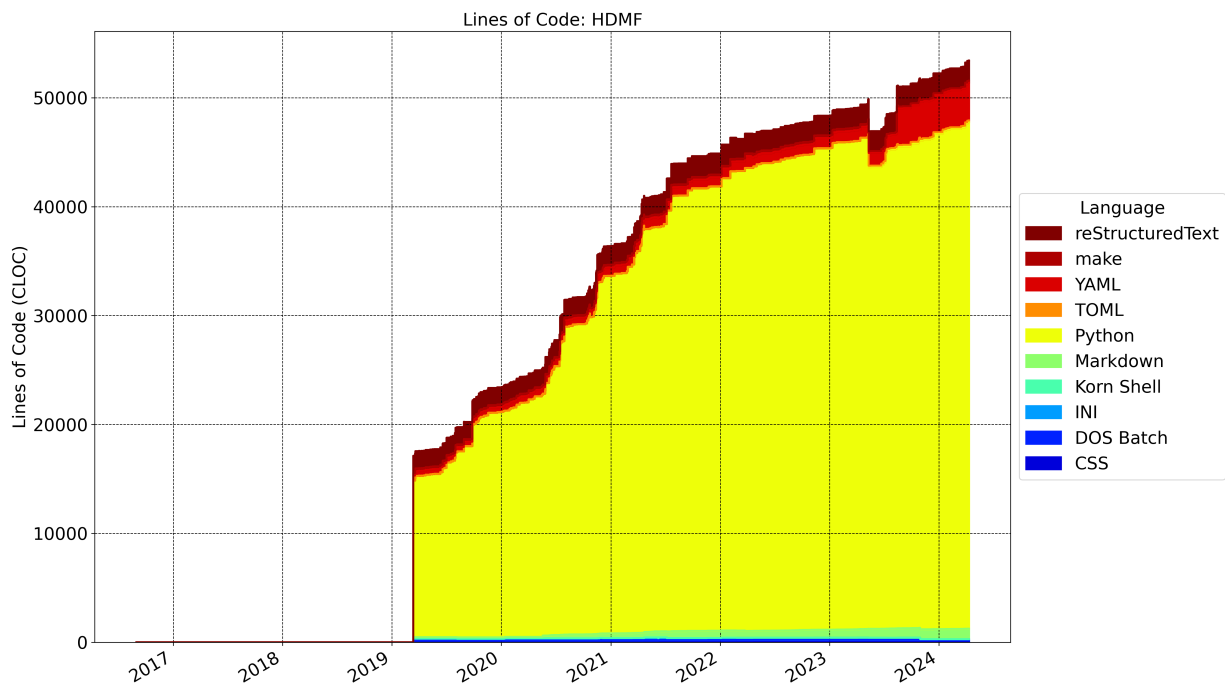
11.3.2 HDMF_Common_Schema

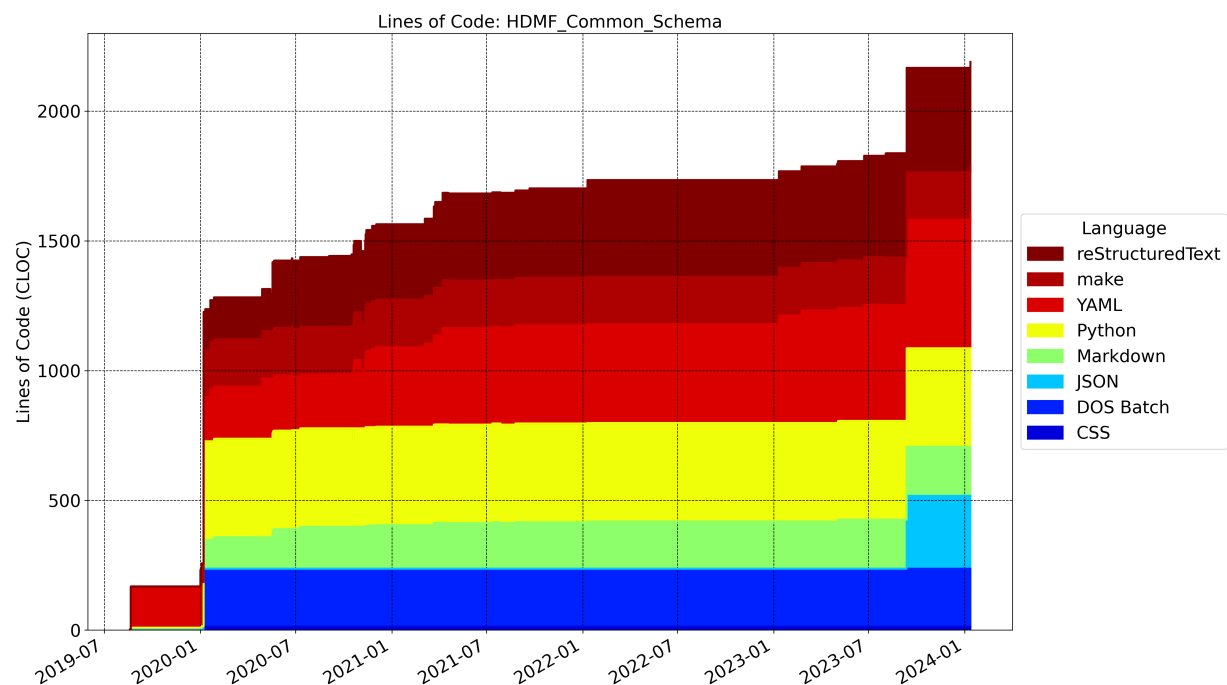
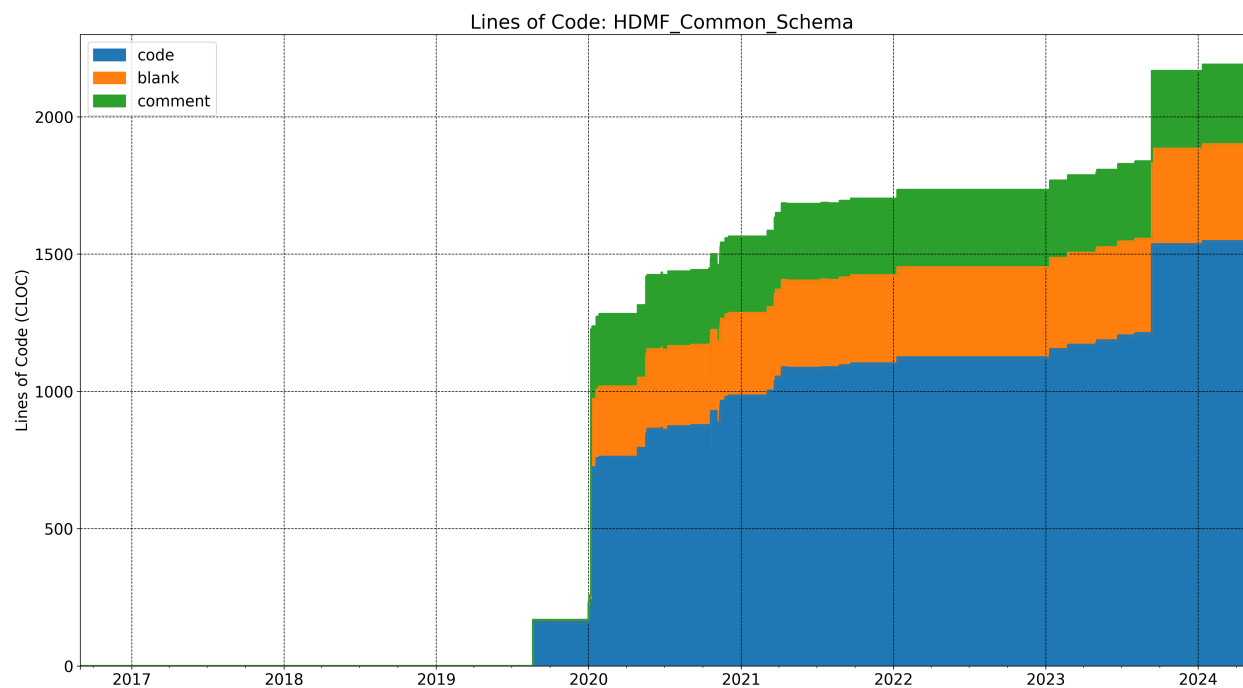
Lines of Code

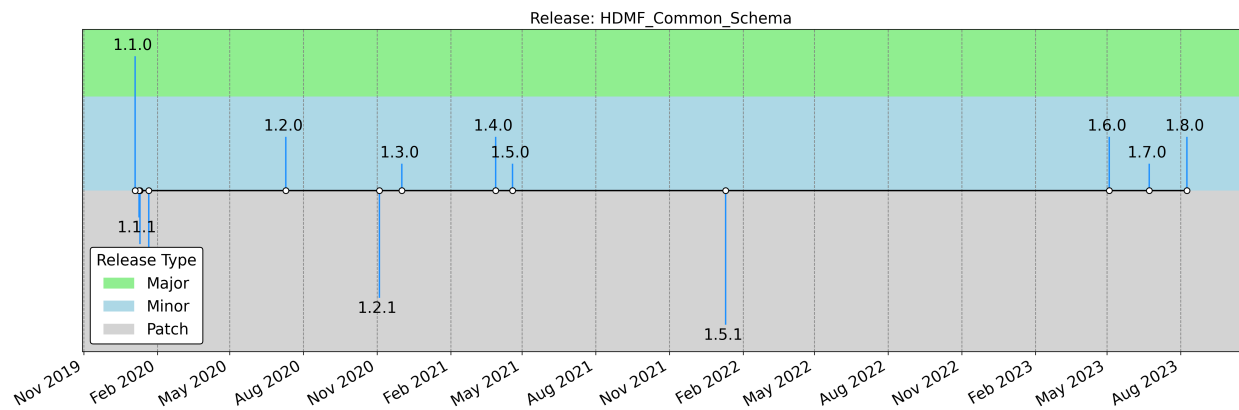
Release History

Additional Information

- Source: <https://github.com/hdmf-dev/hdmf-common-schema.git> (main branch = main)



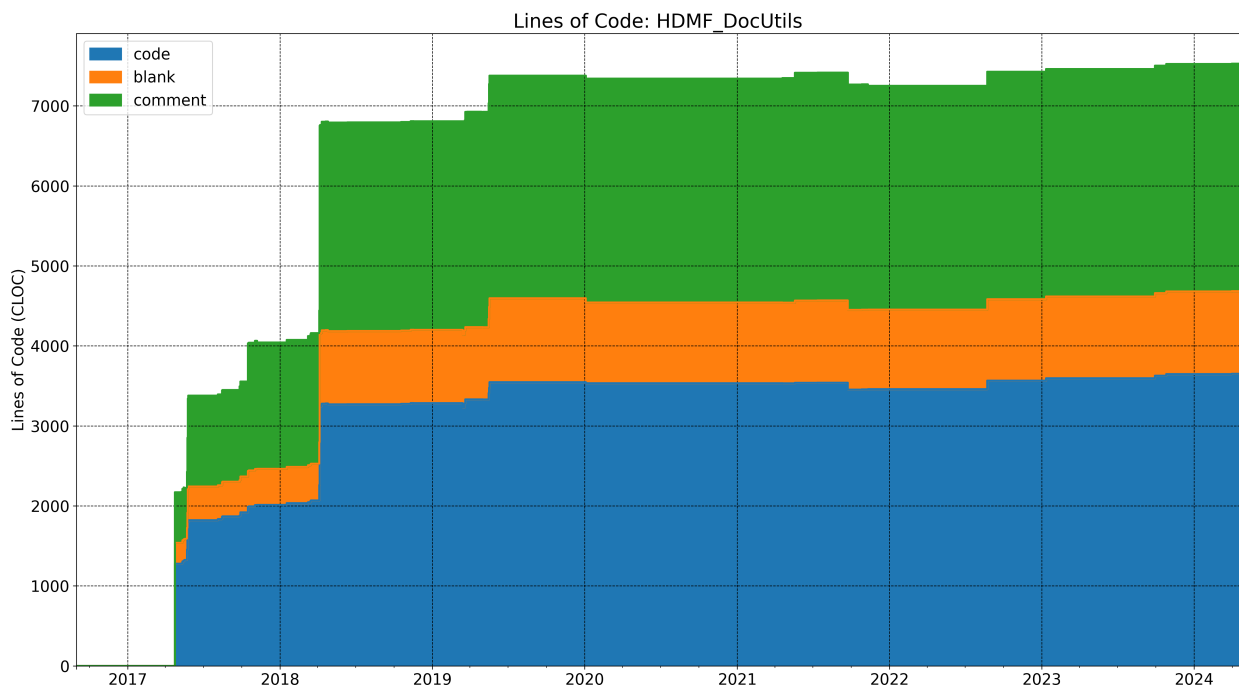


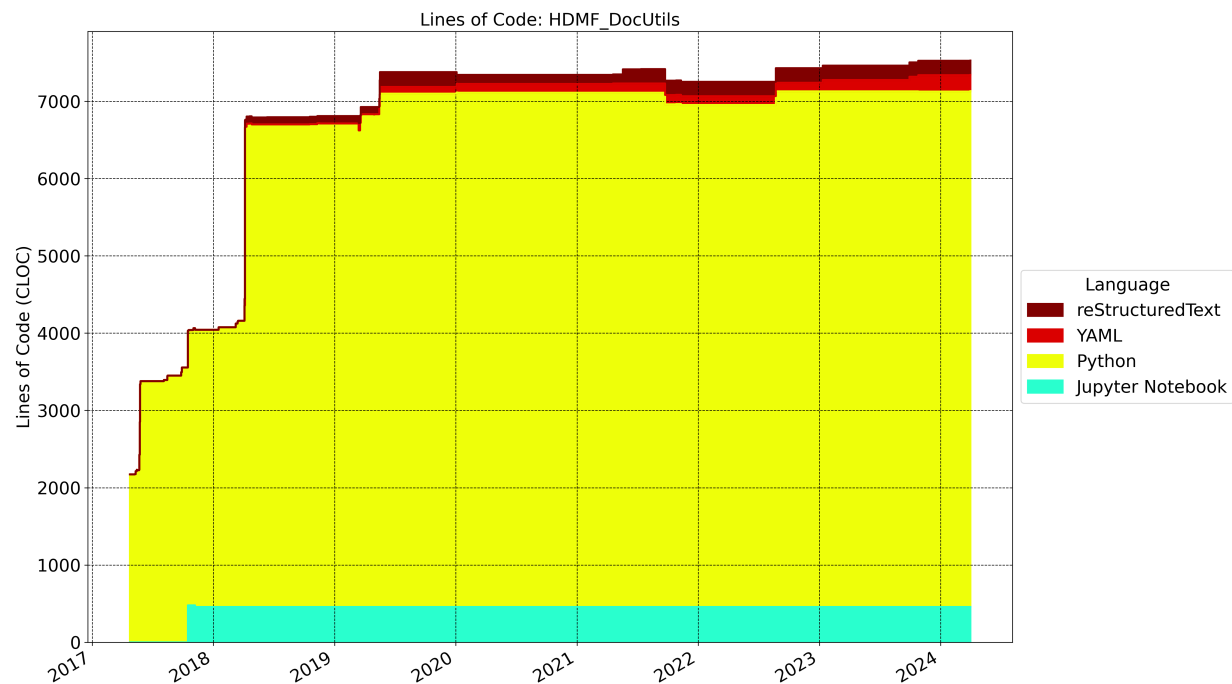


- Docs: <https://hdmf-common-schema.readthedocs.io>

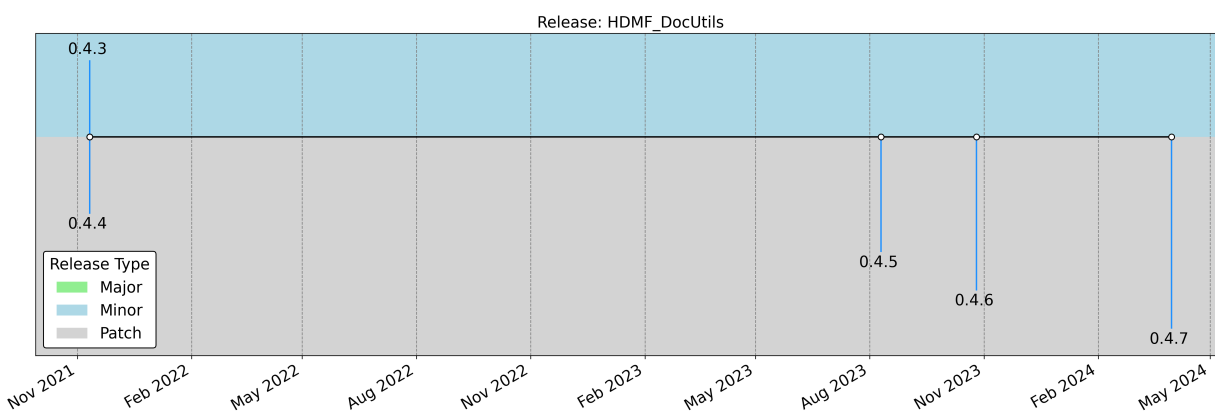
11.3.3 HDMF_DocUtils

Lines of Code





Release History

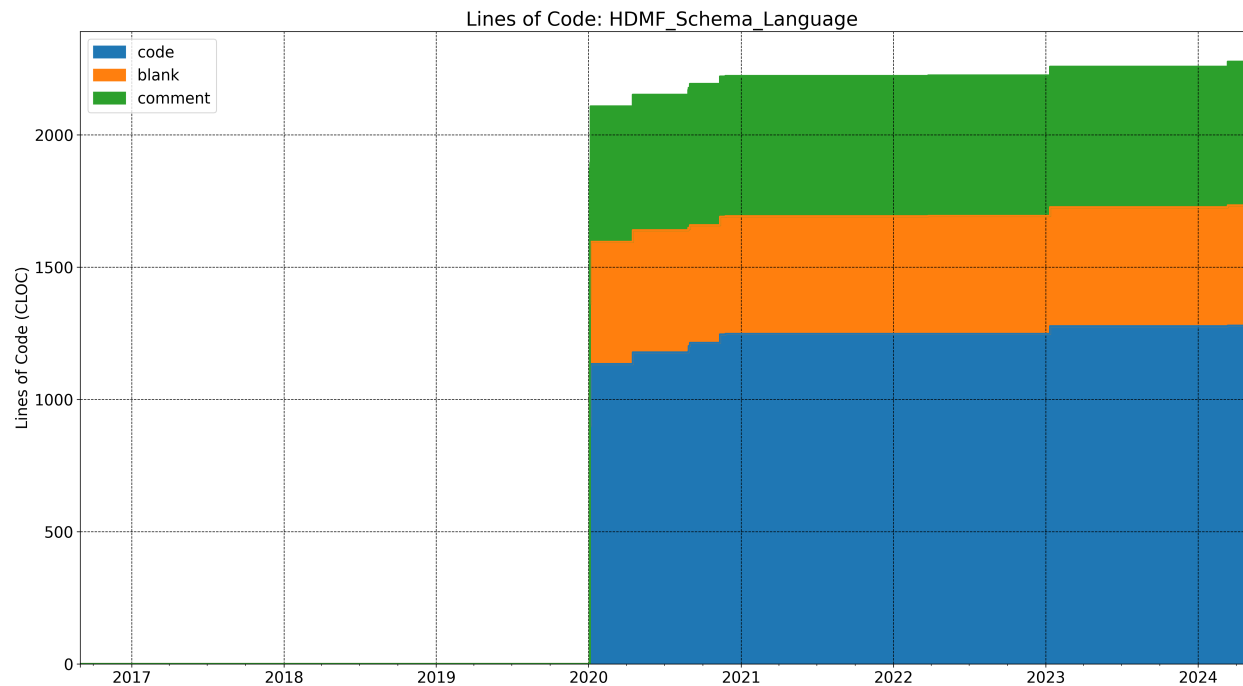


Additional Information

- Source: <https://github.com/hdmf-dev/hdmf-docutils.git> (main branch = main)

11.3.4 HDMF_Schema_Language

Lines of Code



Additional Information

- Source: <https://github.com/hdmf-dev/hdmf-schema-language.git> (main branch = main)
- Docs: <https://hdmf-schema-language.readthedocs.io/>

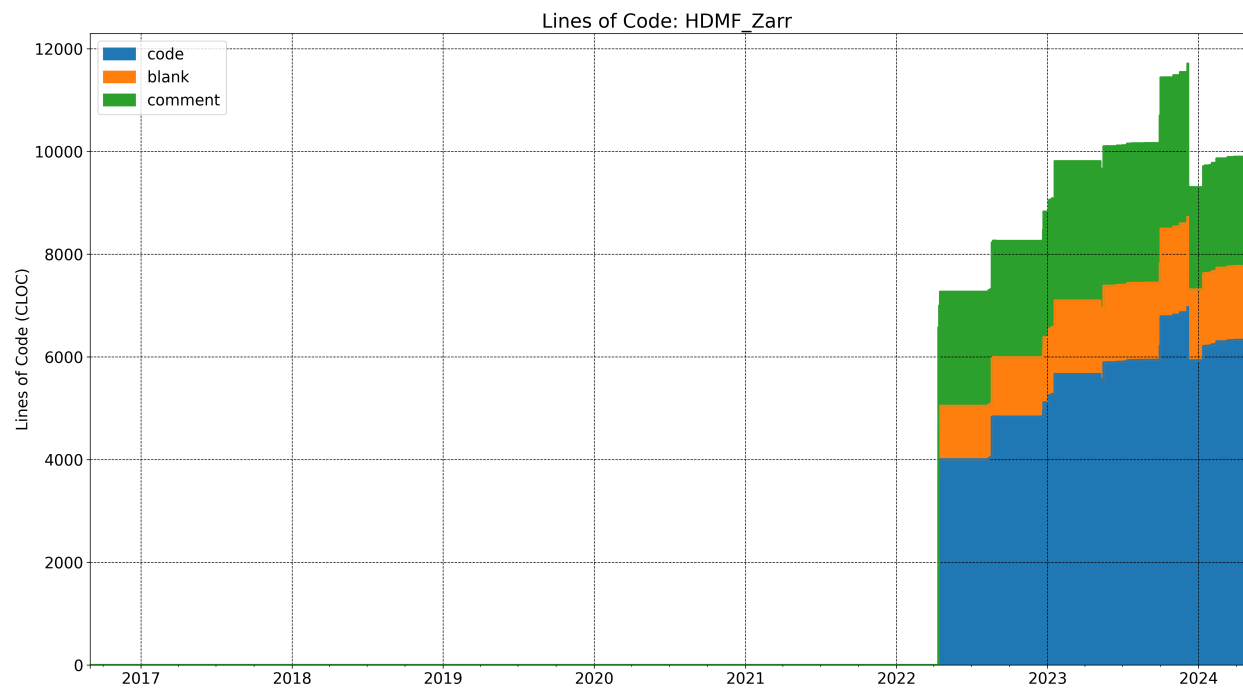
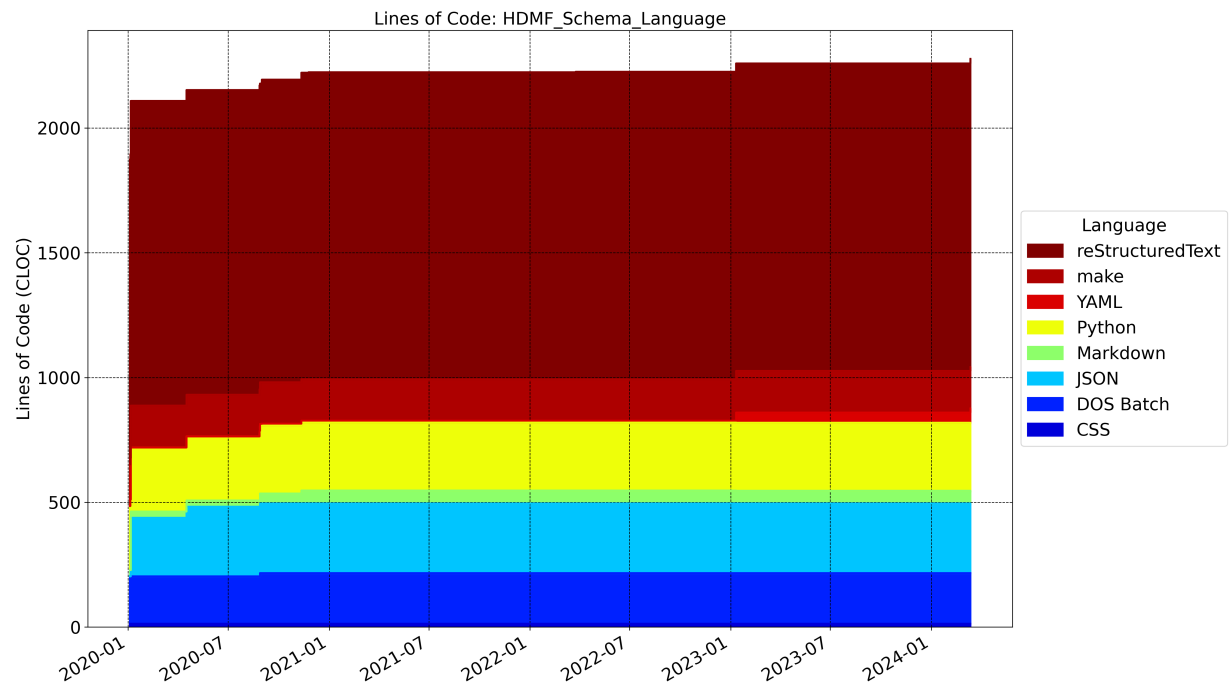
11.3.5 HDMF_Zarr

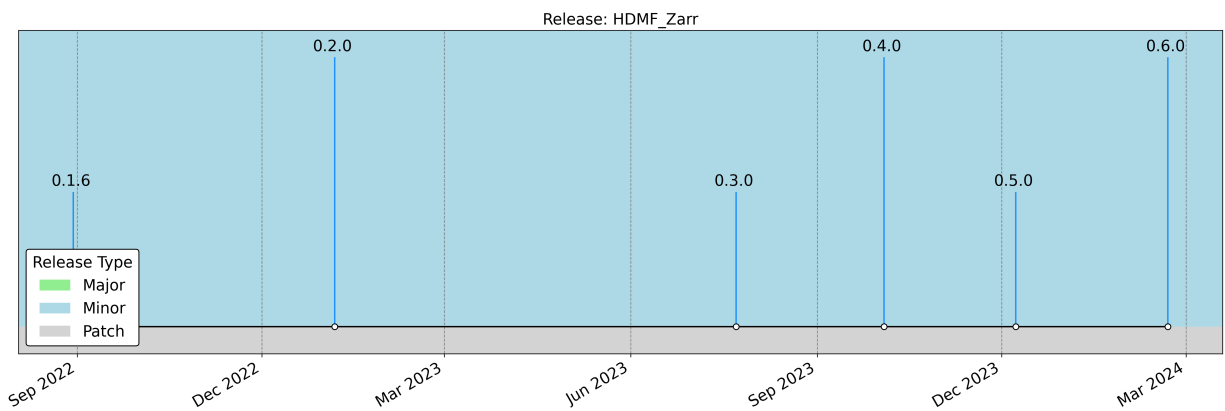
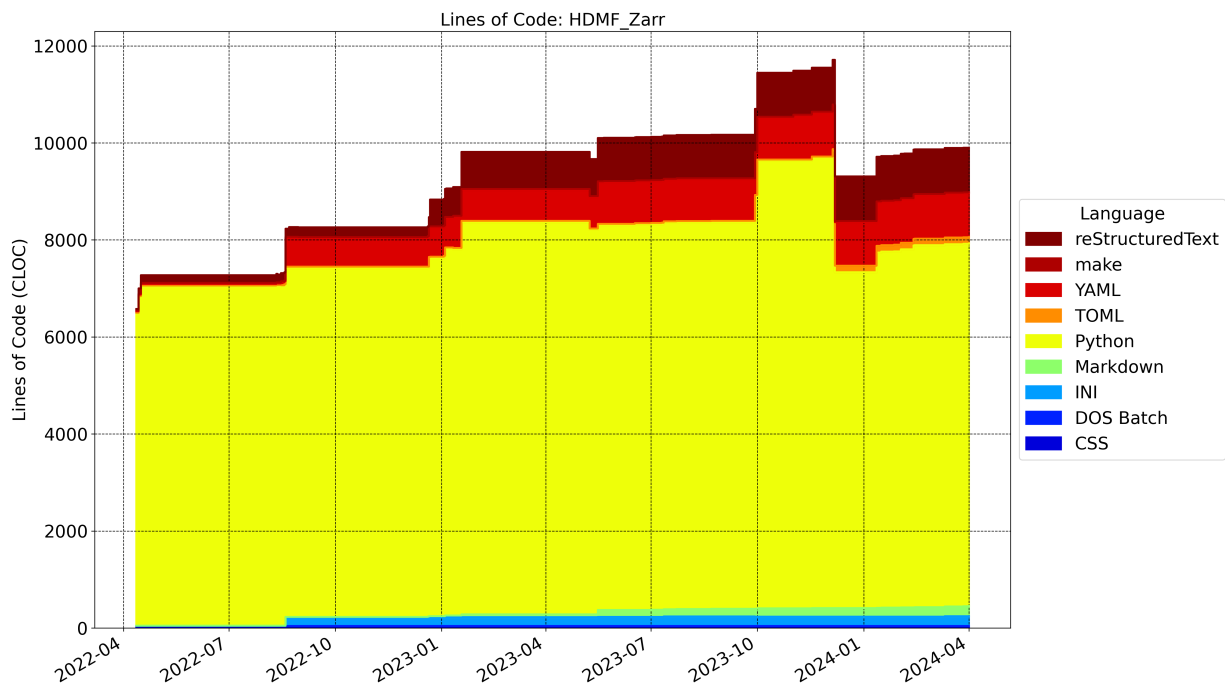
Lines of Code

Release History

Additional Information

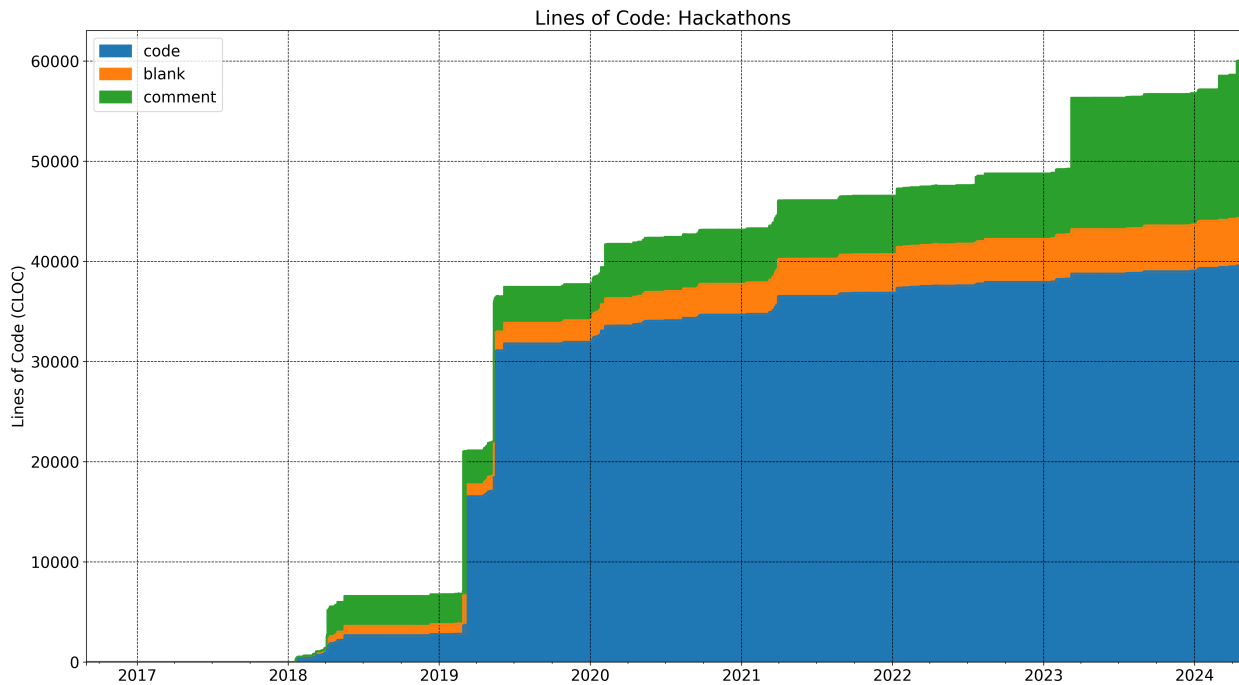
- Source: <https://github.com/hdmf-dev/hdmf-zarr.git> (main branch = dev)
- Docs: <https://hdmf-zarr.readthedocs.io>
- Logo: https://raw.githubusercontent.com/hdmf-dev/hdmf-zarr/dev/docs/source/figures/logo_hdmf_zarr.png





11.3.6 Hackathons

Lines of Code



Additional Information

- Source: https://github.com/NeurodataWithoutBorders/nwb_hackathons.git (main branch = main)
- Docs: https://neurodatawithoutborders.github.io/nwb_hackathons/

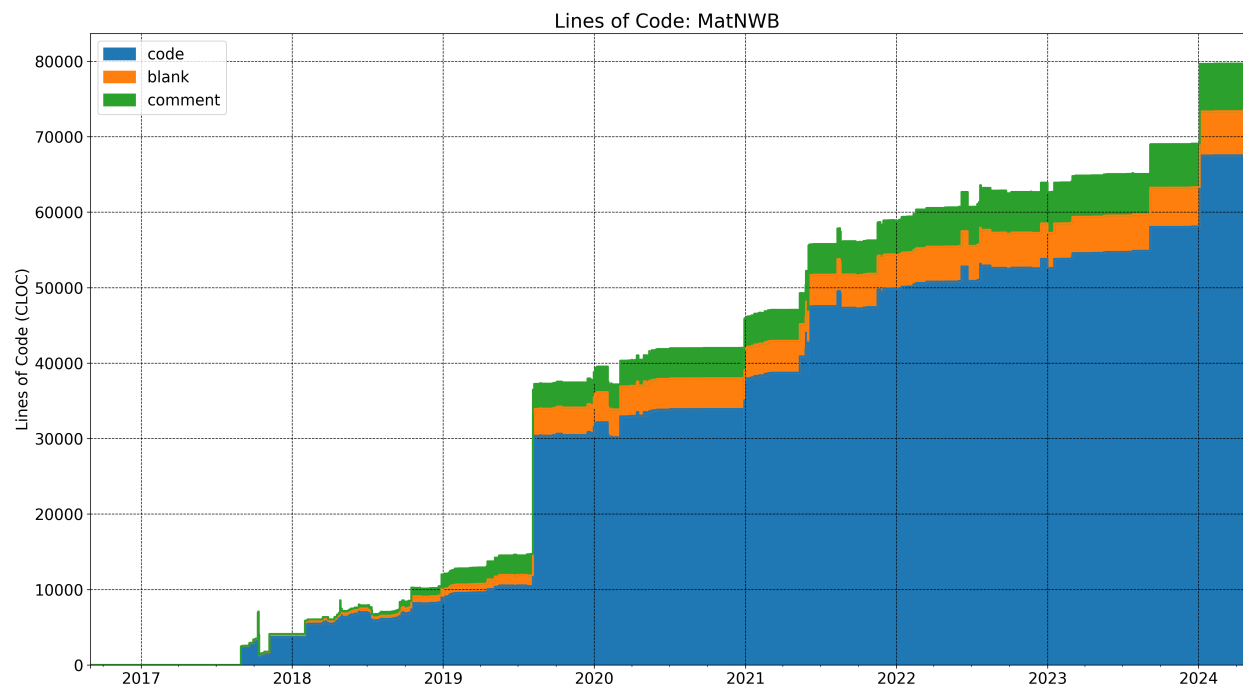
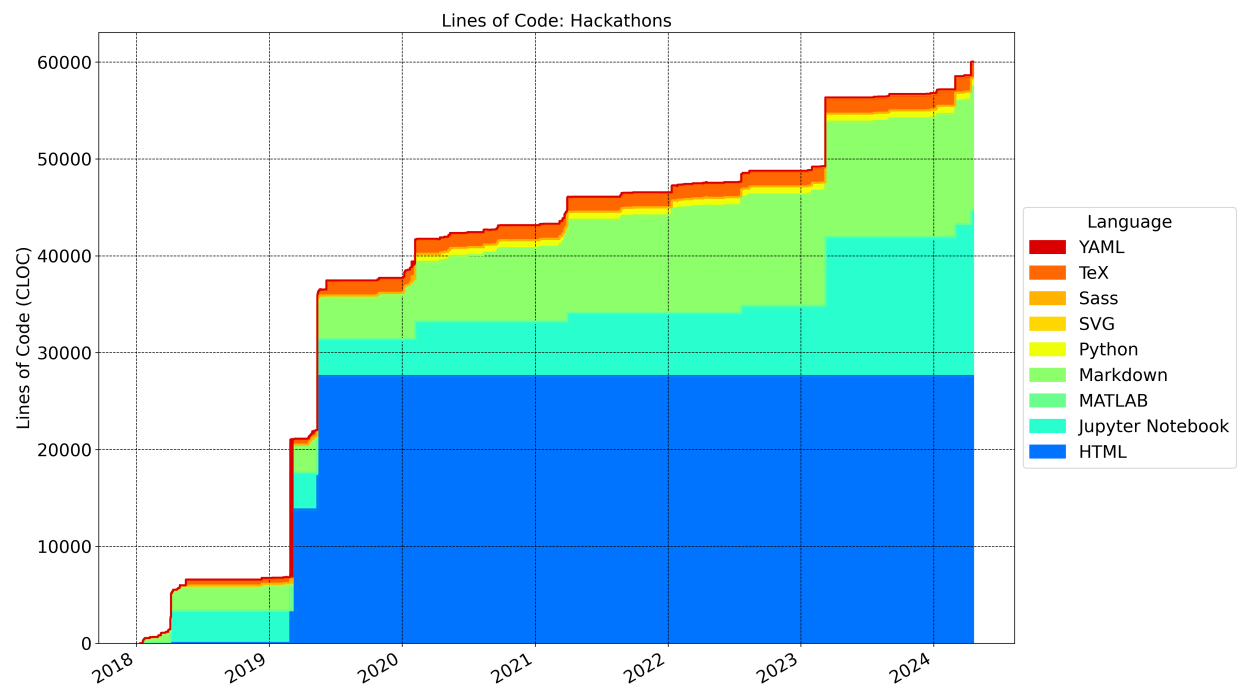
11.3.7 MatNWB

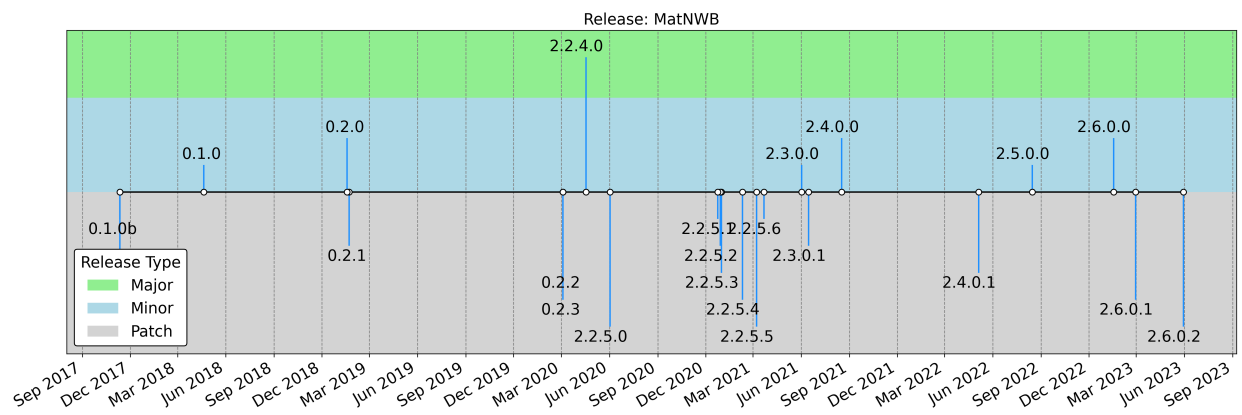
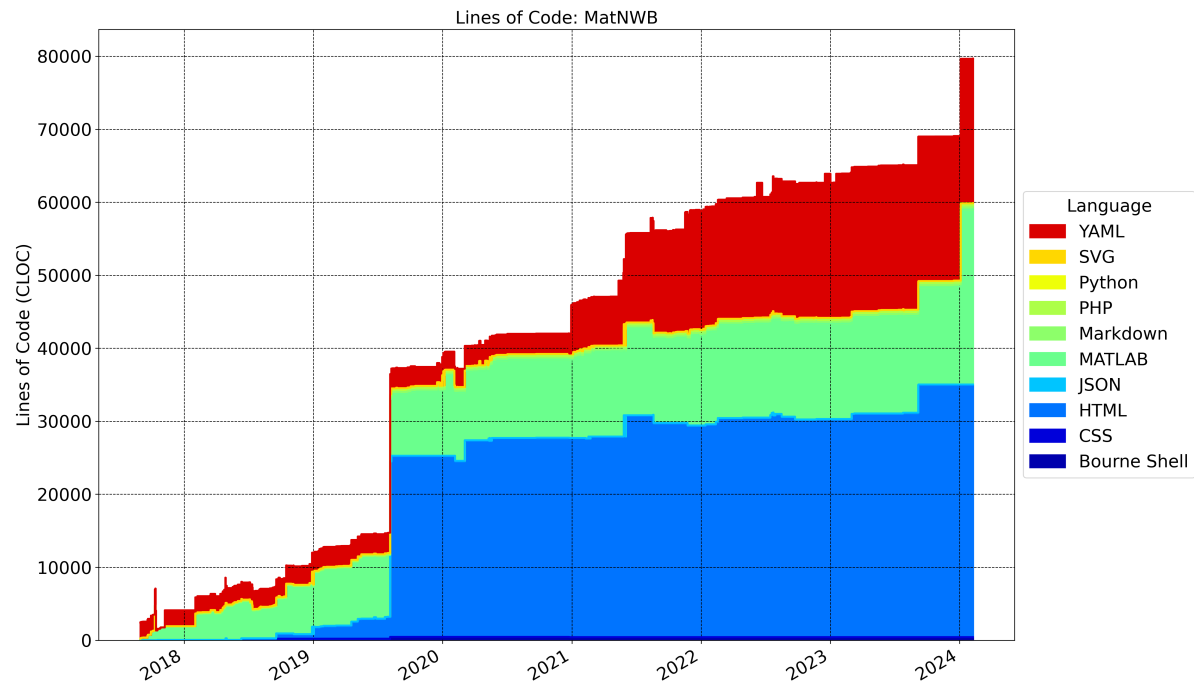
Lines of Code

Release History

Additional Information

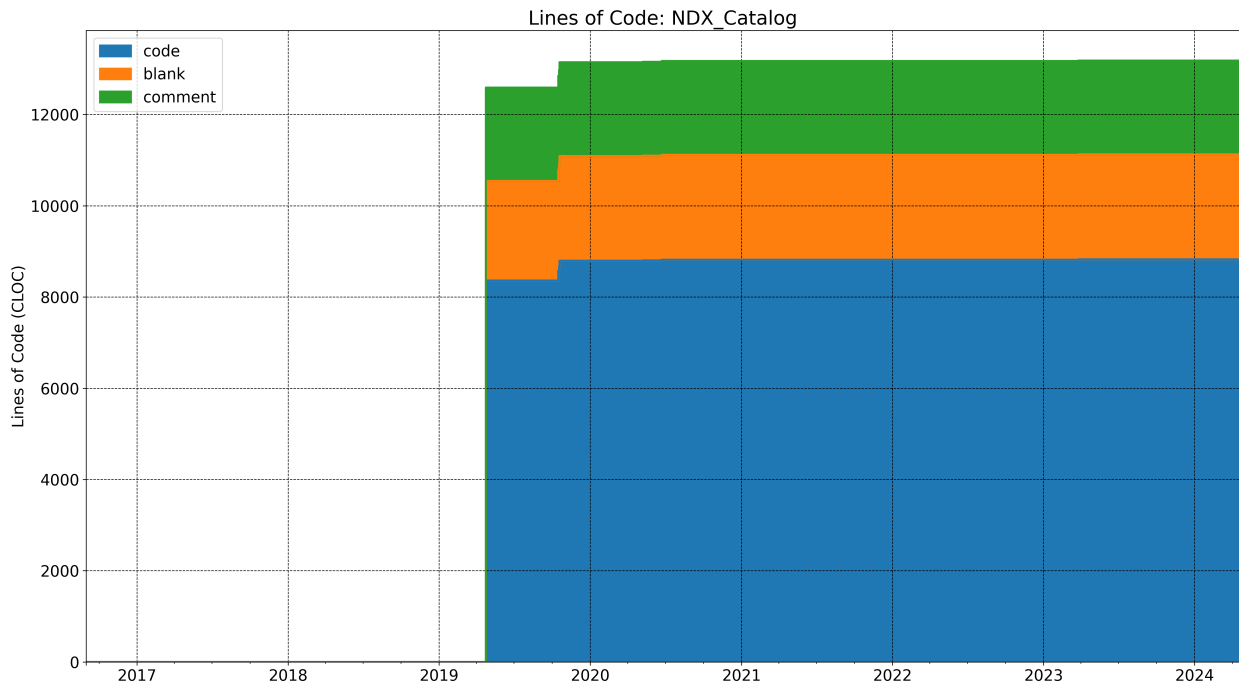
- Source: <https://github.com/NeurodataWithoutBorders/matnwb.git> (main branch = master)
- Docs: <https://neurodatawithoutborders.github.io/matnwb/>
- Logo: https://raw.githubusercontent.com/NeurodataWithoutBorders/matnwb/master/logo/logo_matnwb.png





11.3.8 NDX_Catalog

Lines of Code



Release History

Additional Information

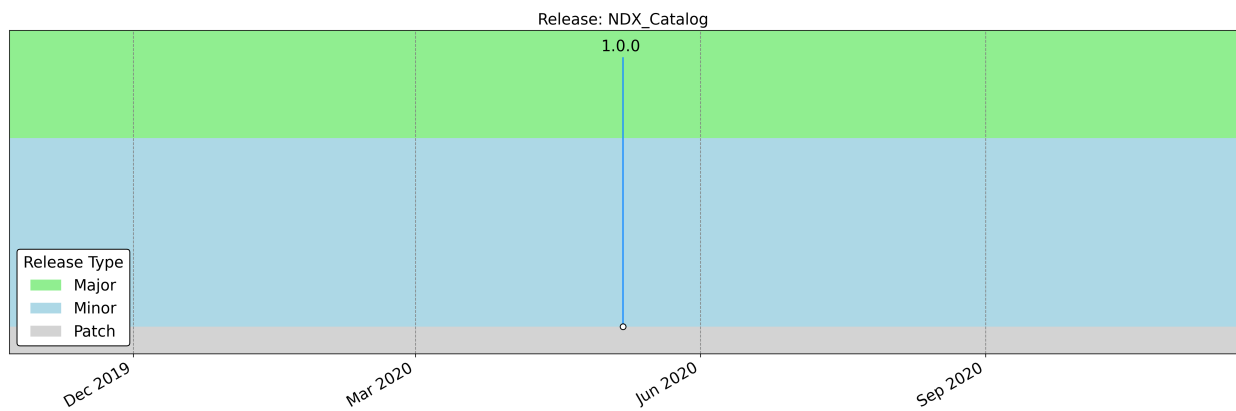
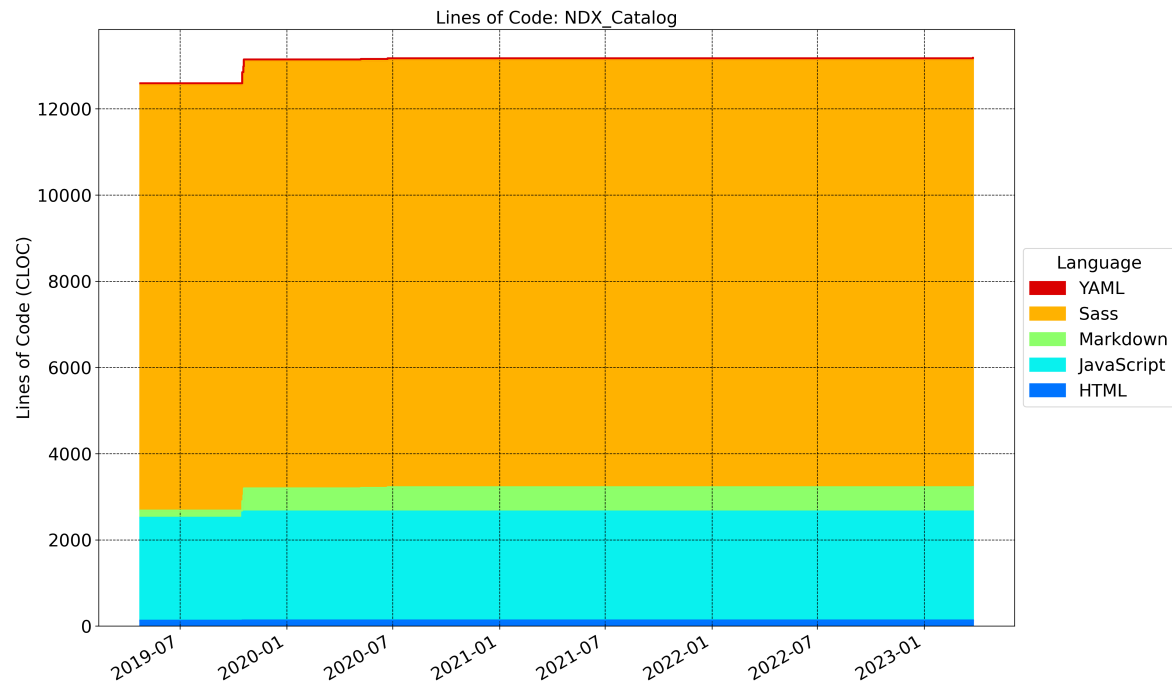
- Source: <https://github.com/nwb-extensions/nwb-extensions.github.io.git> (main branch = main)
- Docs: <https://nwb-extensions.github.io/>
- Logo: <https://github.com/nwb-extensions/nwb-extensions.github.io/blob/main/images/ndx-logo-text.png>

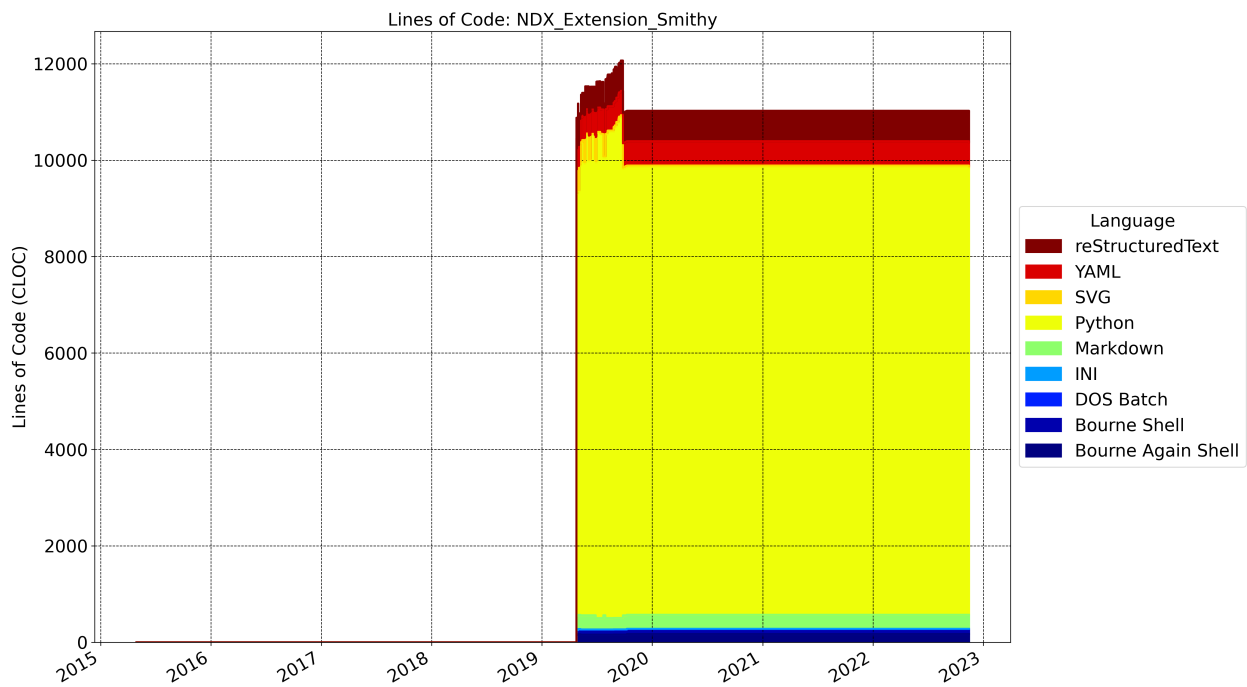
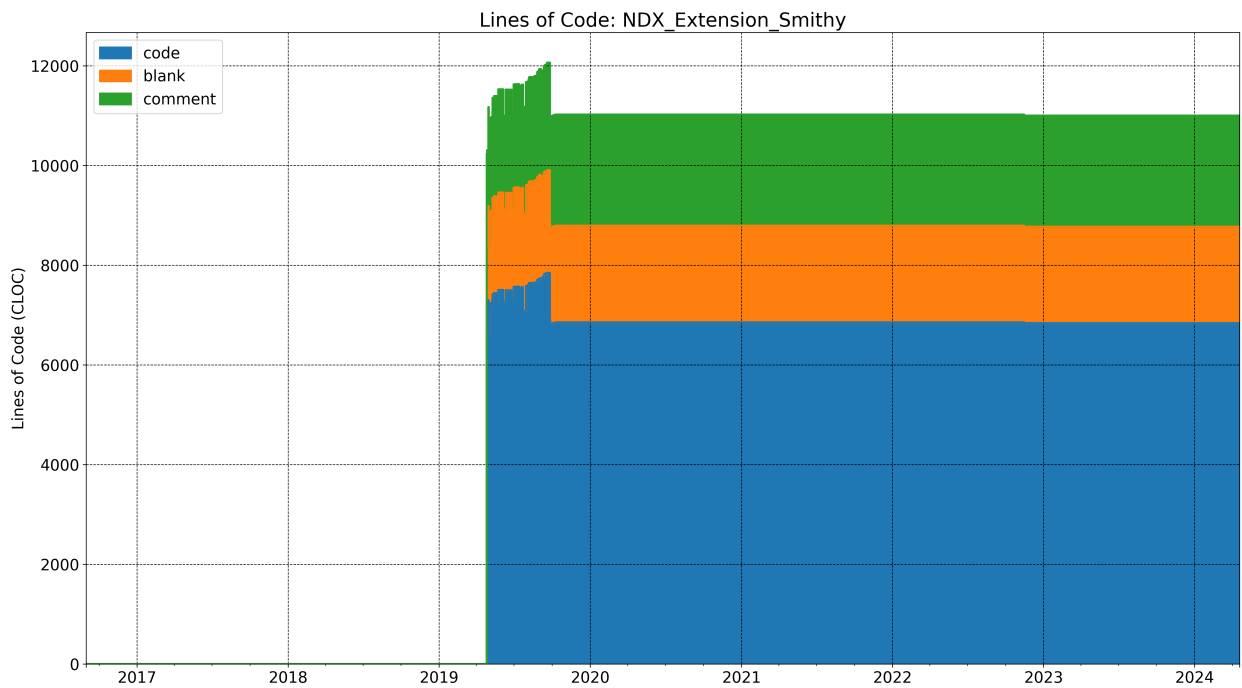
11.3.9 NDX_Extension_Smithy

Lines of Code

Additional Information

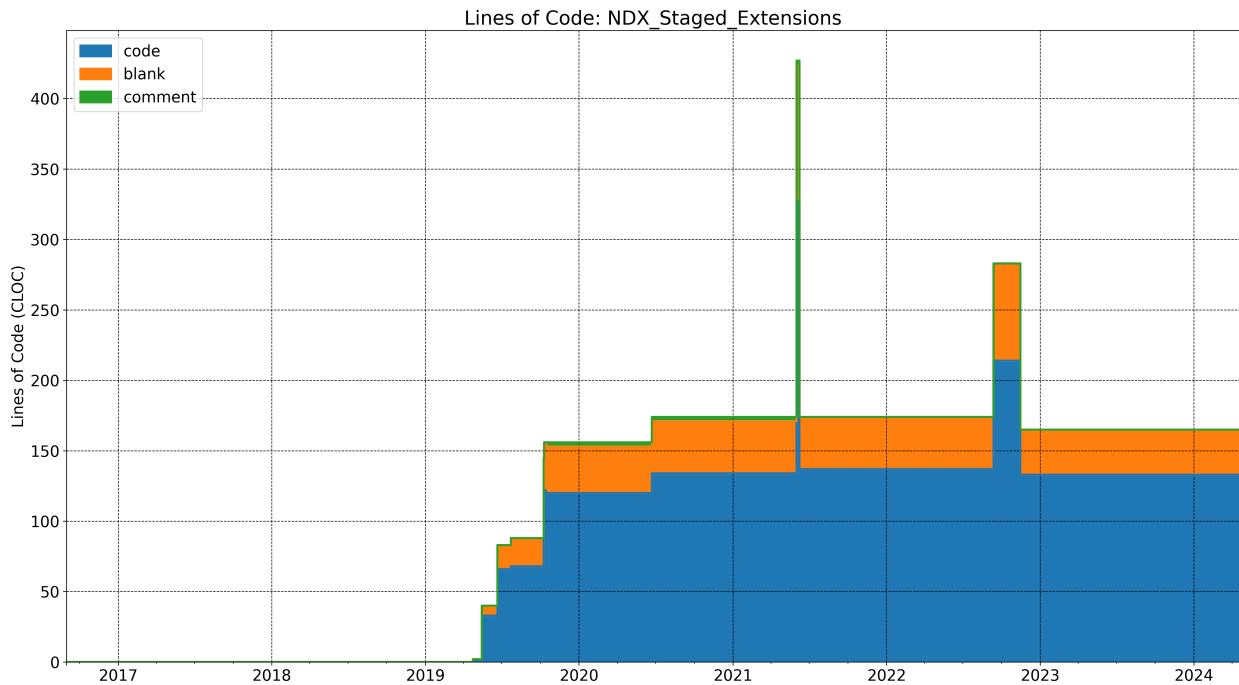
- Source: <https://github.com/nwb-extensions/nwb-extensions-smithy.git> (main branch = master)





11.3.10 NDX_Staged_Extensions

Lines of Code



Additional Information

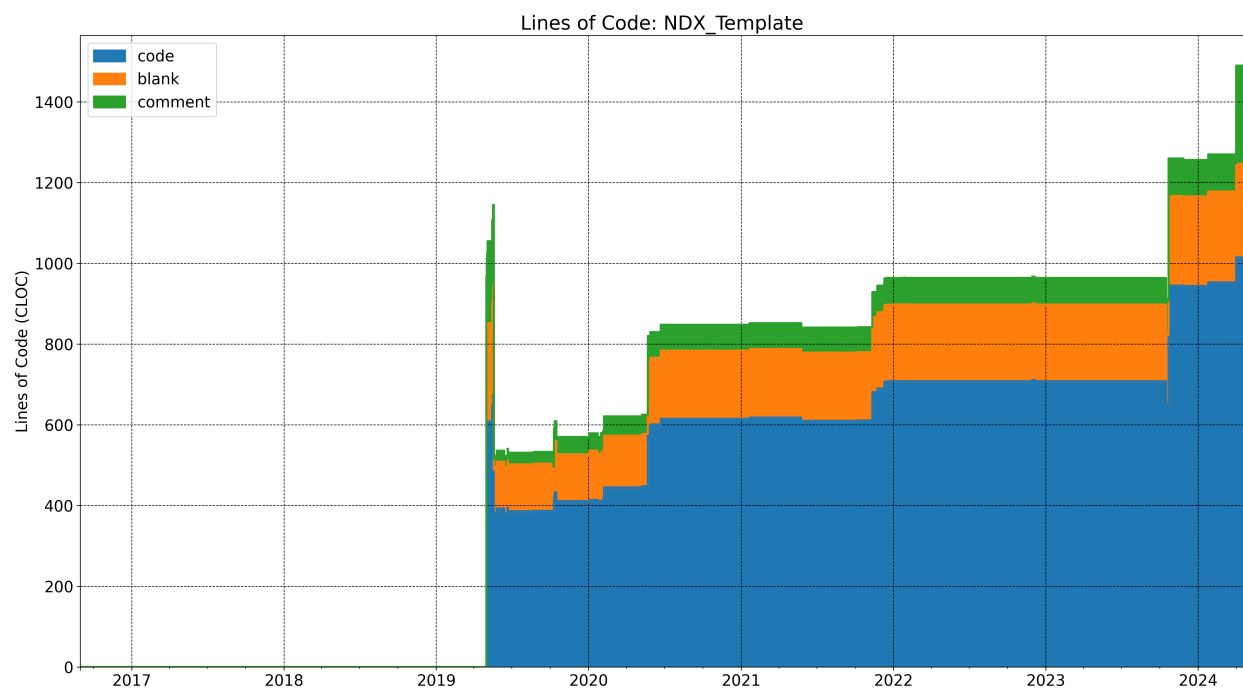
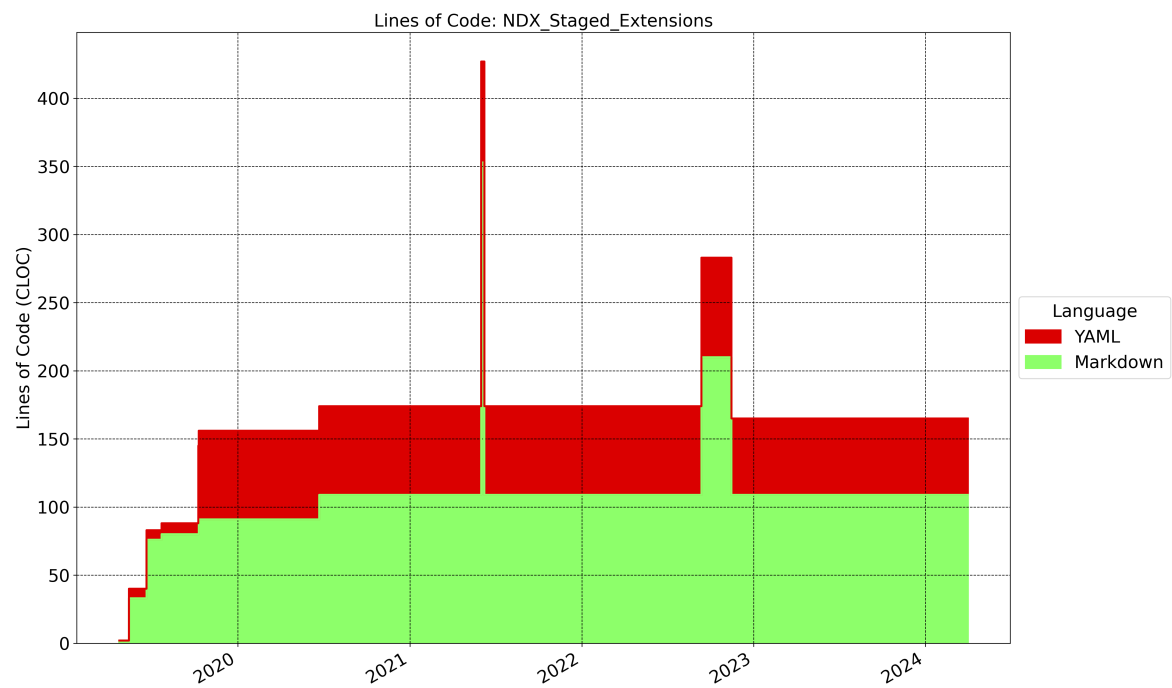
- Source: <https://github.com/nwb-extensions/staged-extensions.git> (main branch = master)

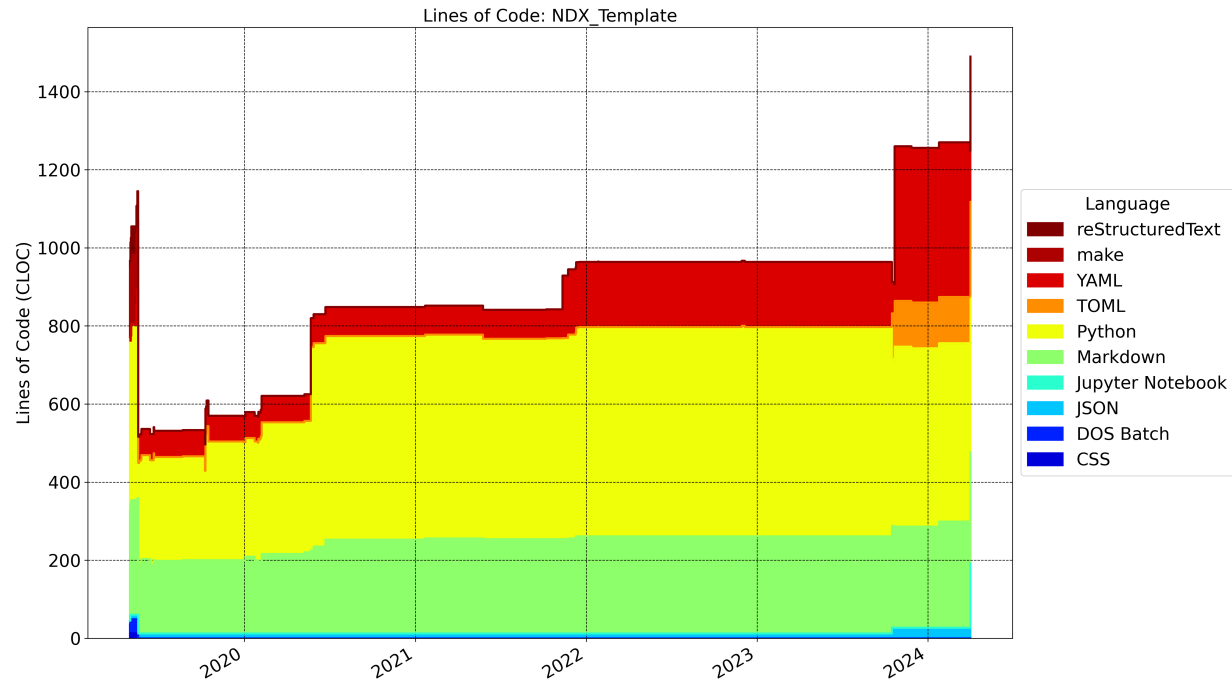
11.3.11 NDX_Template

Lines of Code

Additional Information

- Source: <https://github.com/nwb-extensions/ndx-template.git> (main branch = main)
- Docs: https://nwb-overview.readthedocs.io/en/latest/extensions_tutorial/2_create_extension_spec_walkthrough.html





11.3.12 NWBInspector

Lines of Code

Release History

Additional Information

- Source: <https://github.com/NeurodataWithoutBorders/nwbinspector.git> (main branch = dev)
- Docs: <https://nwbinspector.readthedocs.io>
- Logo: <https://raw.githubusercontent.com/NeurodataWithoutBorders/nwbinspector/dev/docs/logo/logo.png>

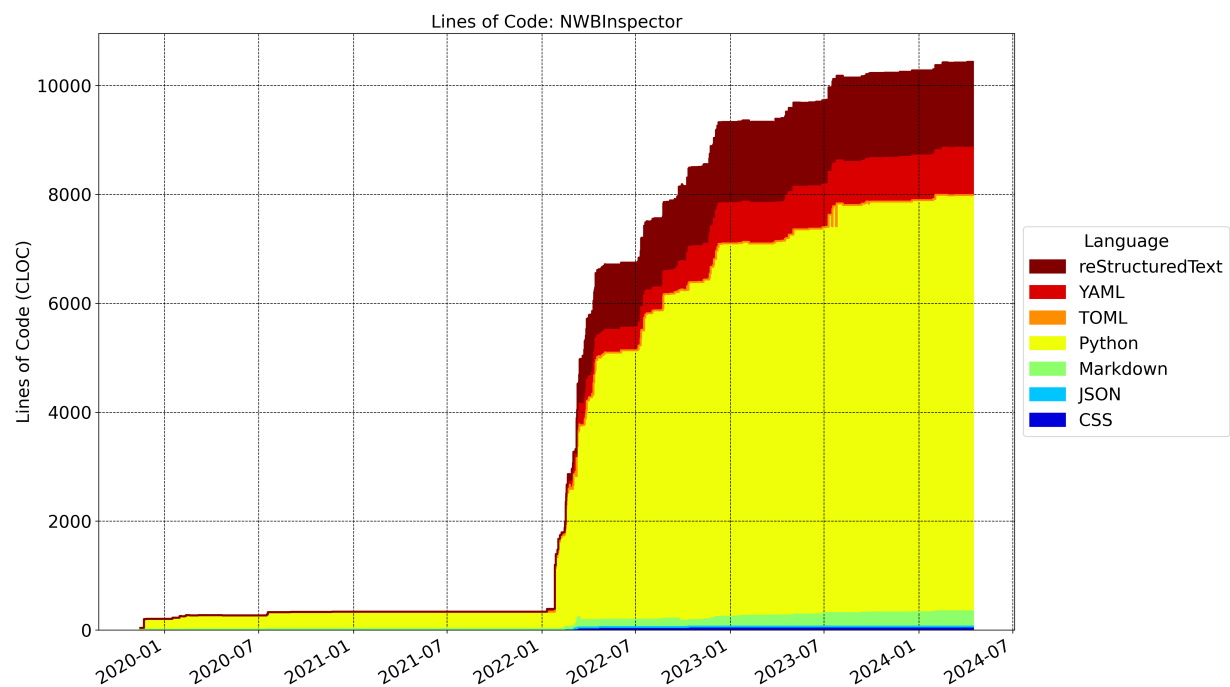
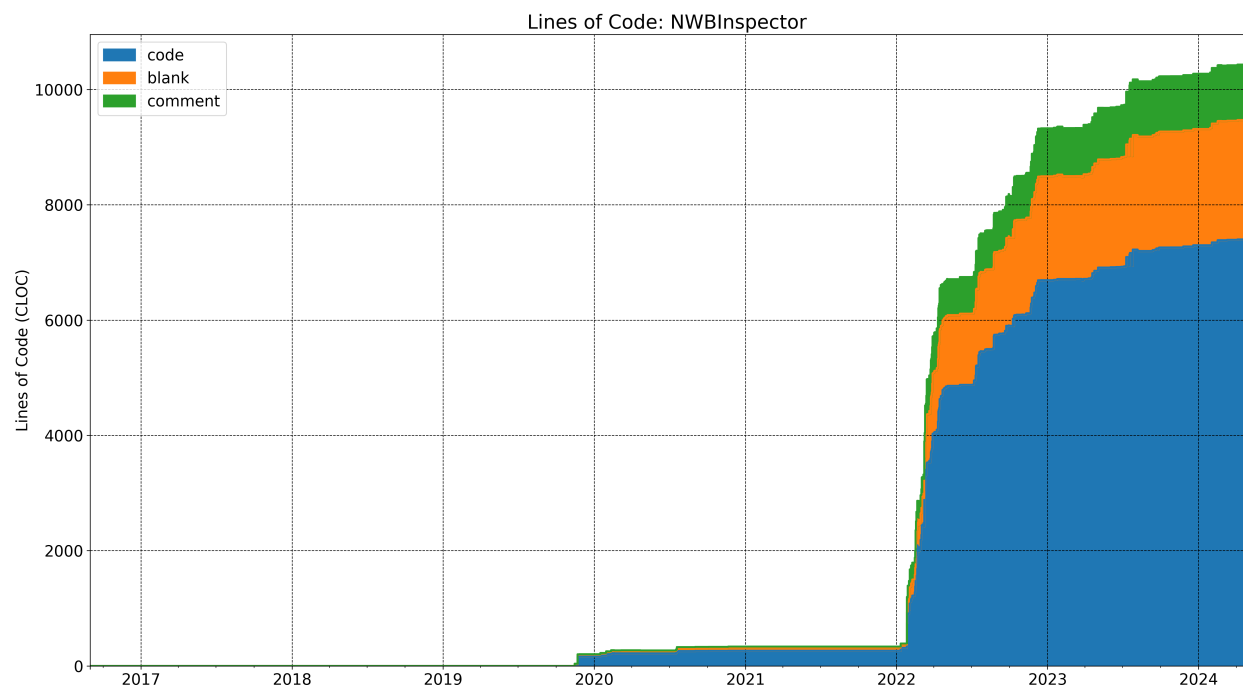
11.3.13 NWBWidgets

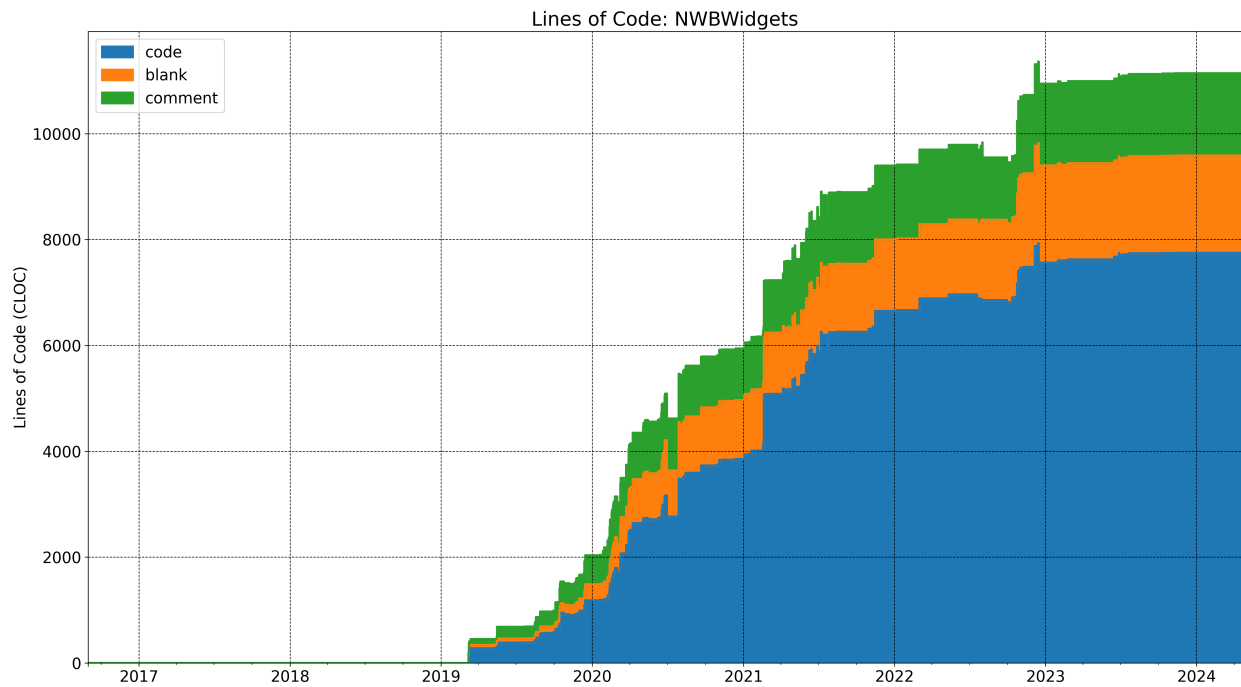
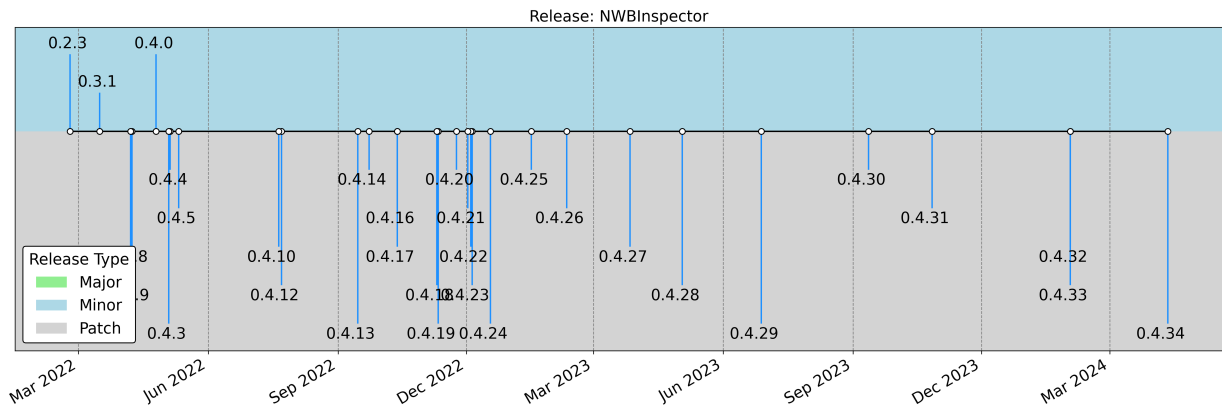
Lines of Code

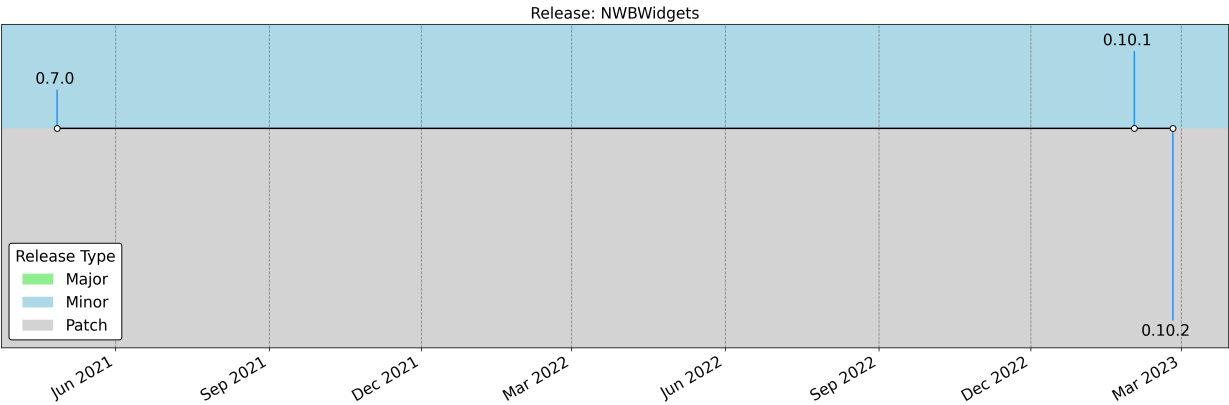
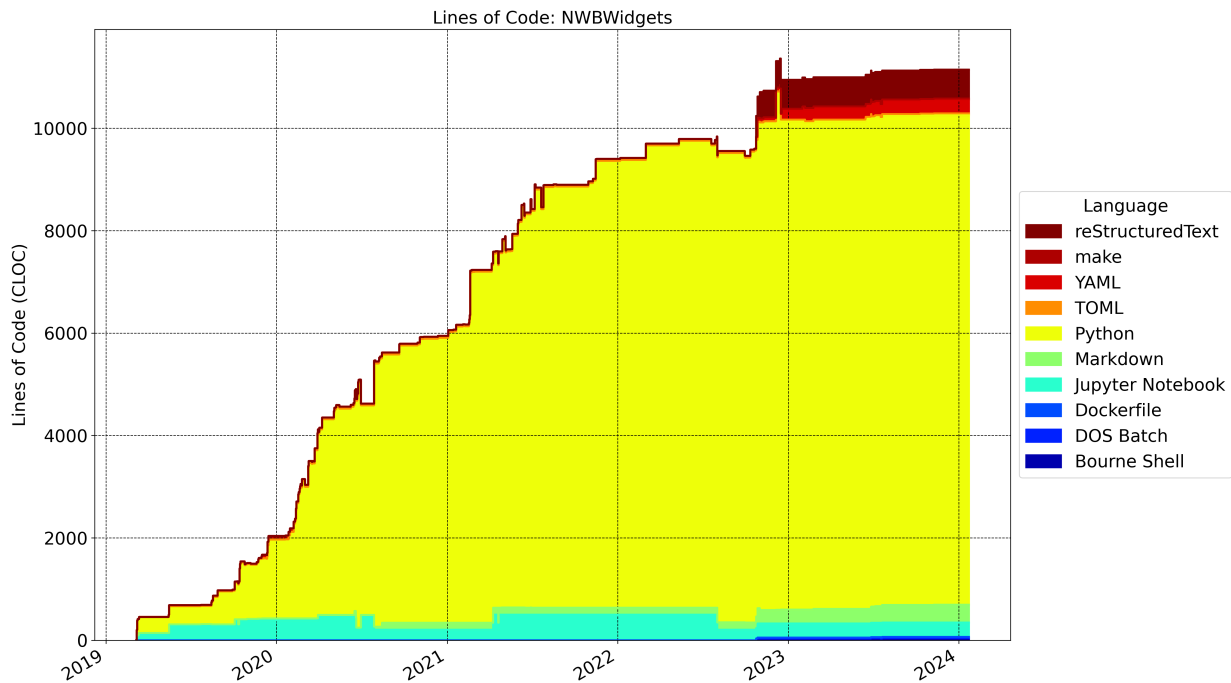
Release History

Additional Information

- Source: <https://github.com/NeurodataWithoutBorders/nwb-jupyter-widgets.git> (main branch = master)
- Logo: <https://user-images.githubusercontent.com/844306/254117081-f20b8c26-79c7-4c1c-a3b5-b49ecf8cce5d.png>

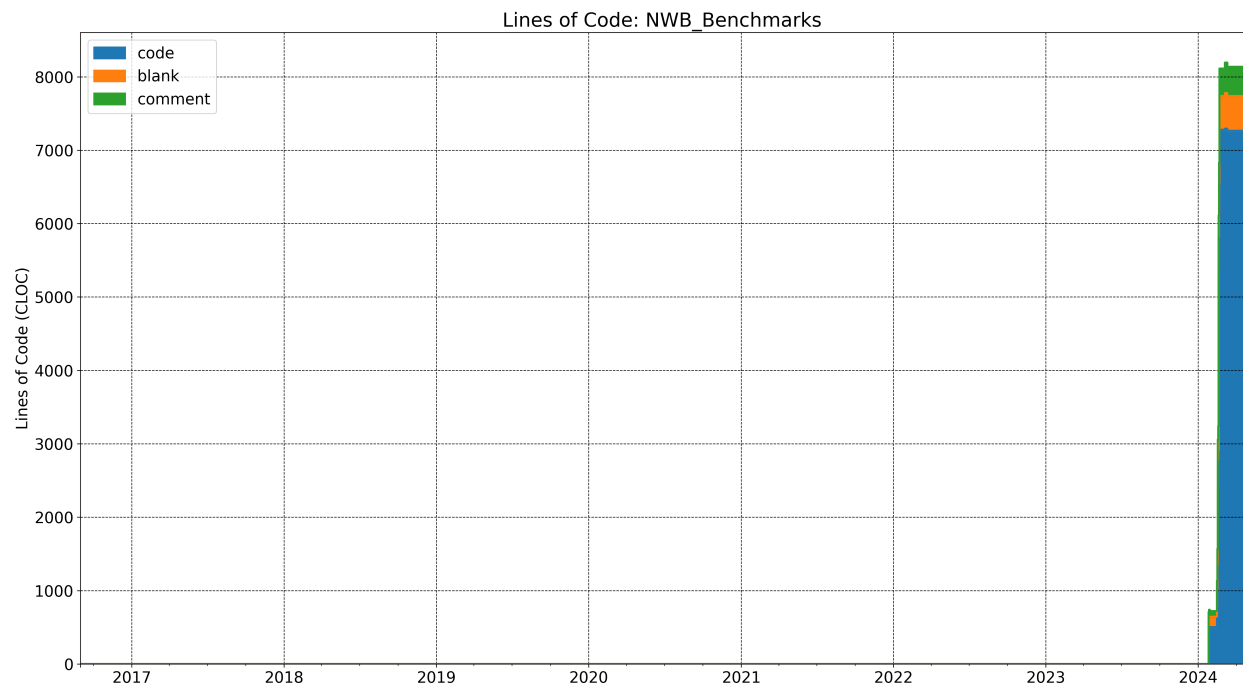






11.3.14 NWB_Benchmarks

Lines of Code



Additional Information

- Source: https://github.com/NeurodataWithoutBorders/nwb_benchmarks.git (main branch = main)

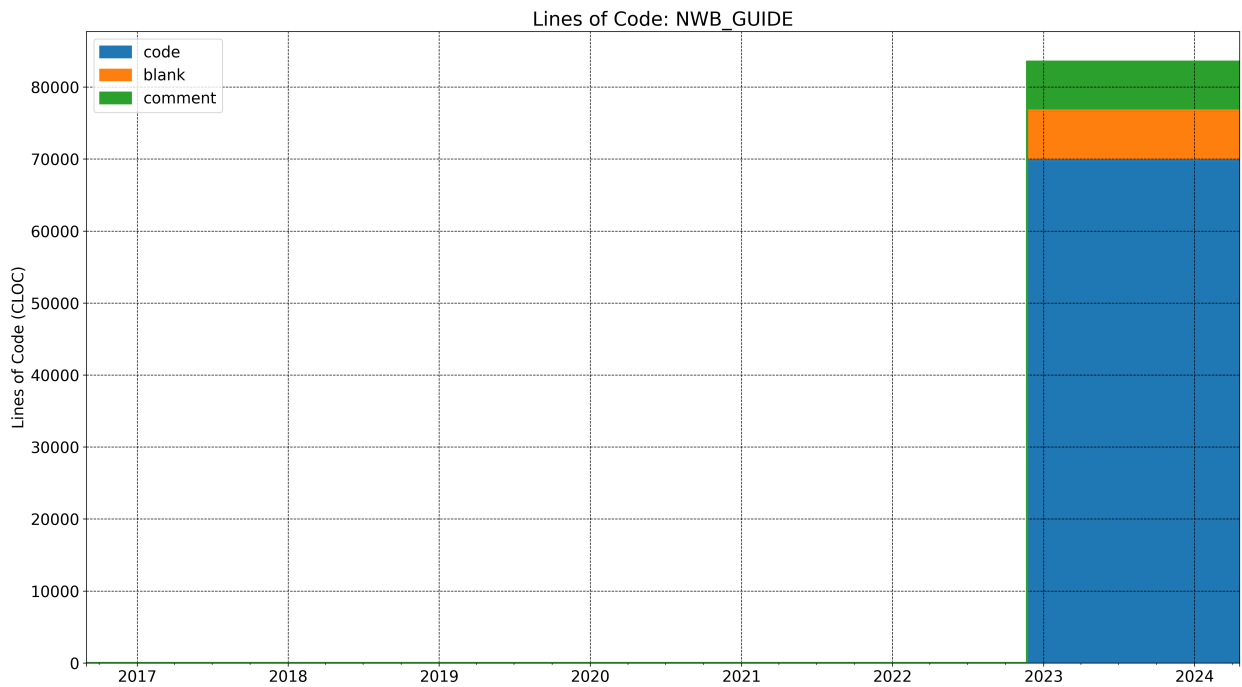
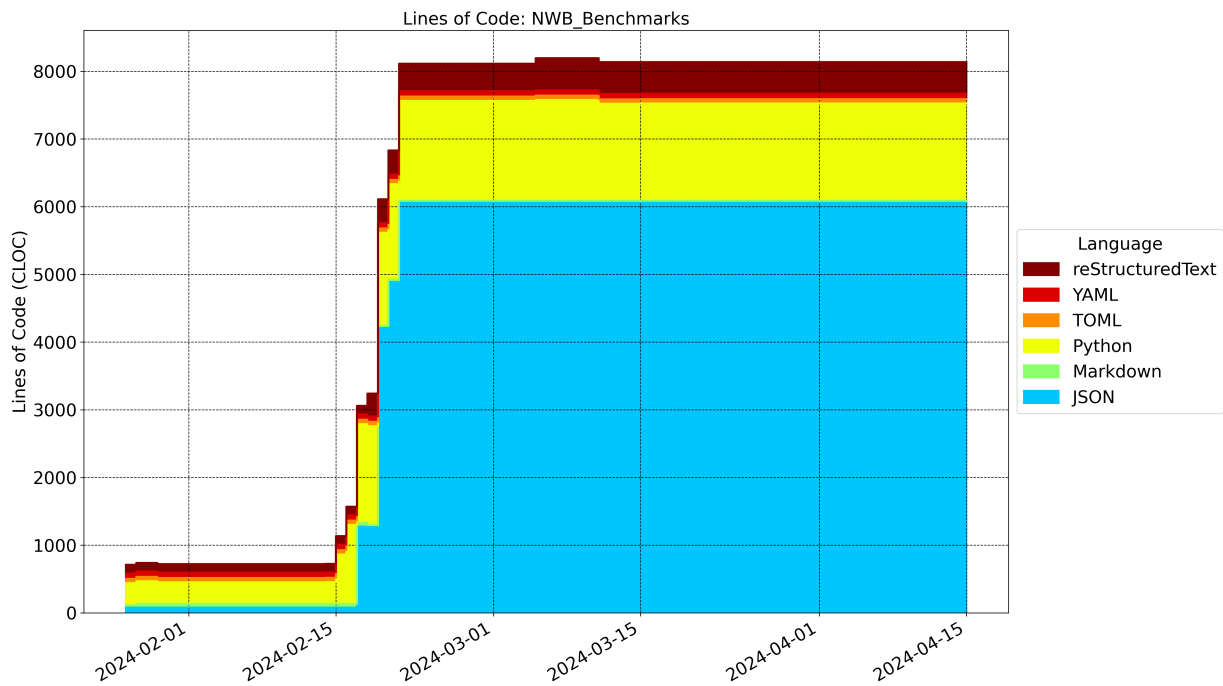
11.3.15 NWB_GUIDE

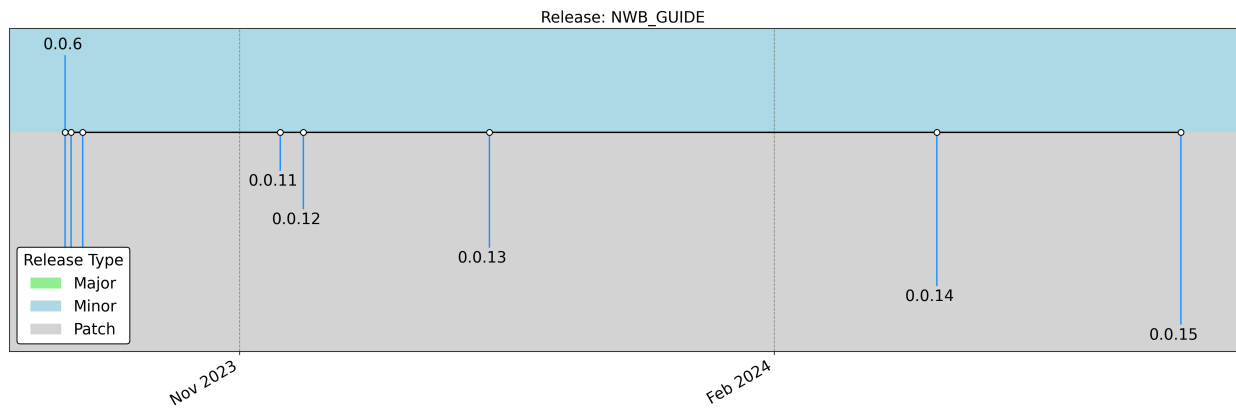
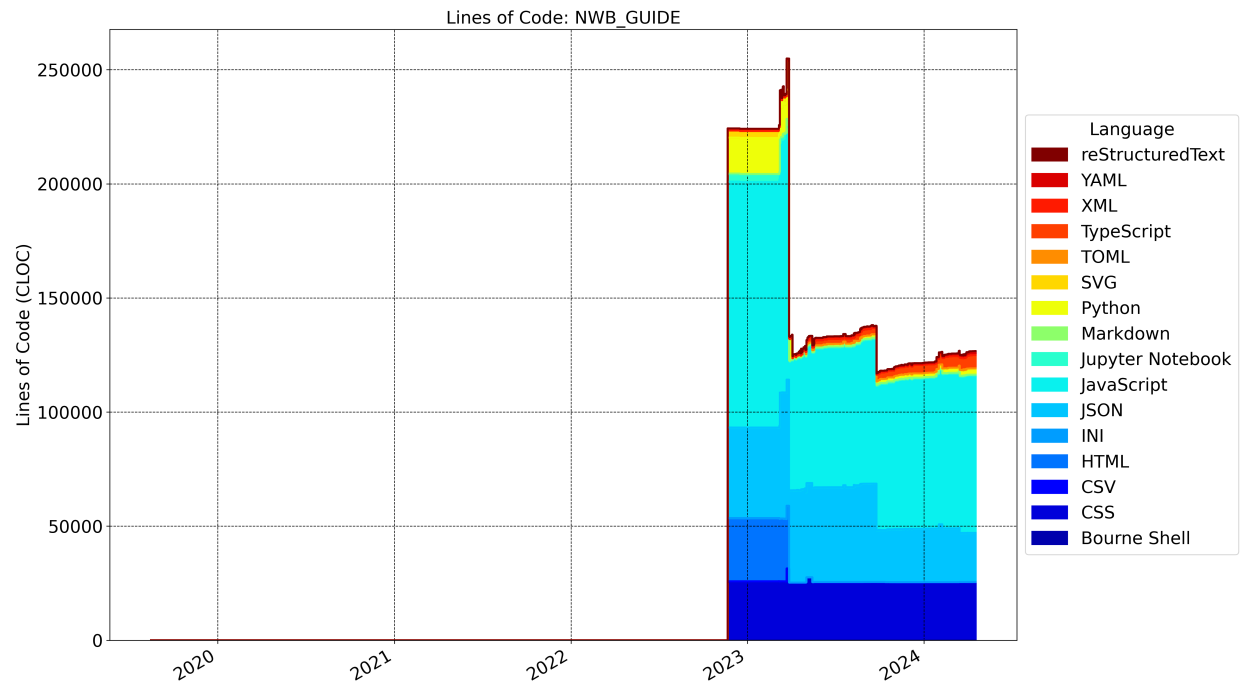
Lines of Code

Release History

Additional Information

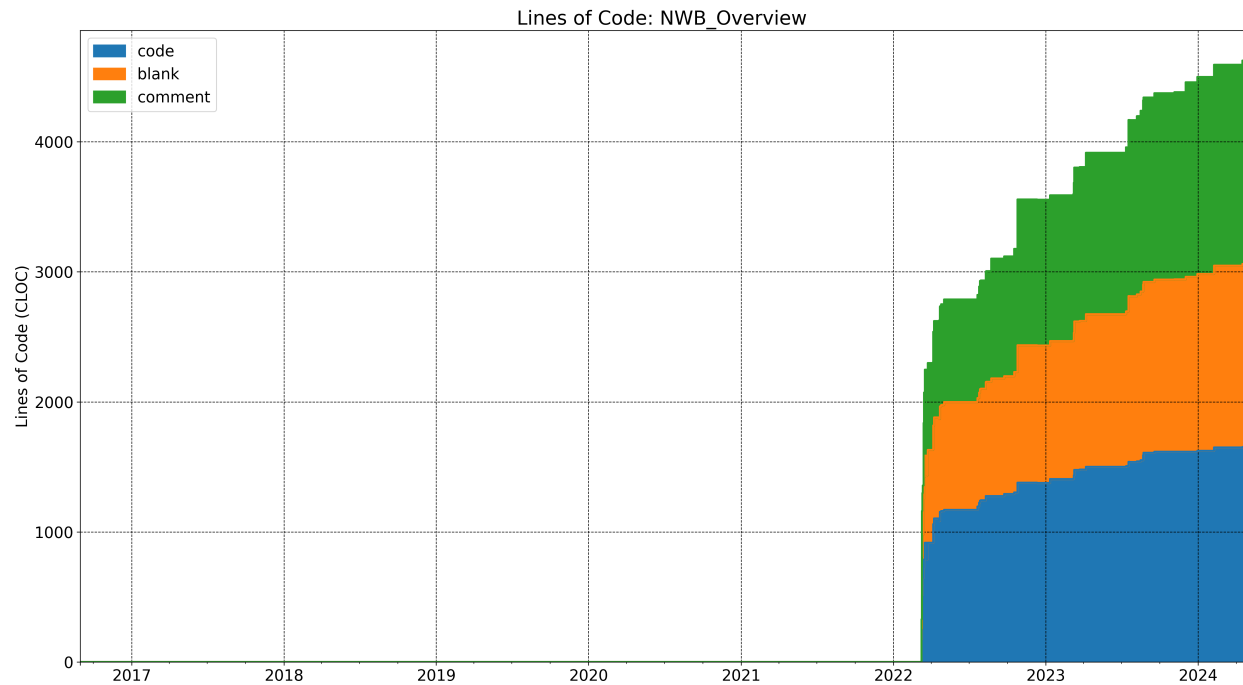
- Source: <https://github.com/NeurodataWithoutBorders/nwb-guide.git> (main branch = main)
- Docs: <https://github.com/NeurodataWithoutBorders/nwb-guide>
- Logo: <https://raw.githubusercontent.com/NeurodataWithoutBorders/nwb-guide/main/src/renderer/assets/img/logo-guide-draft-transparent-tight.png>





11.3.16 NWB_Overview

Lines of Code



Additional Information

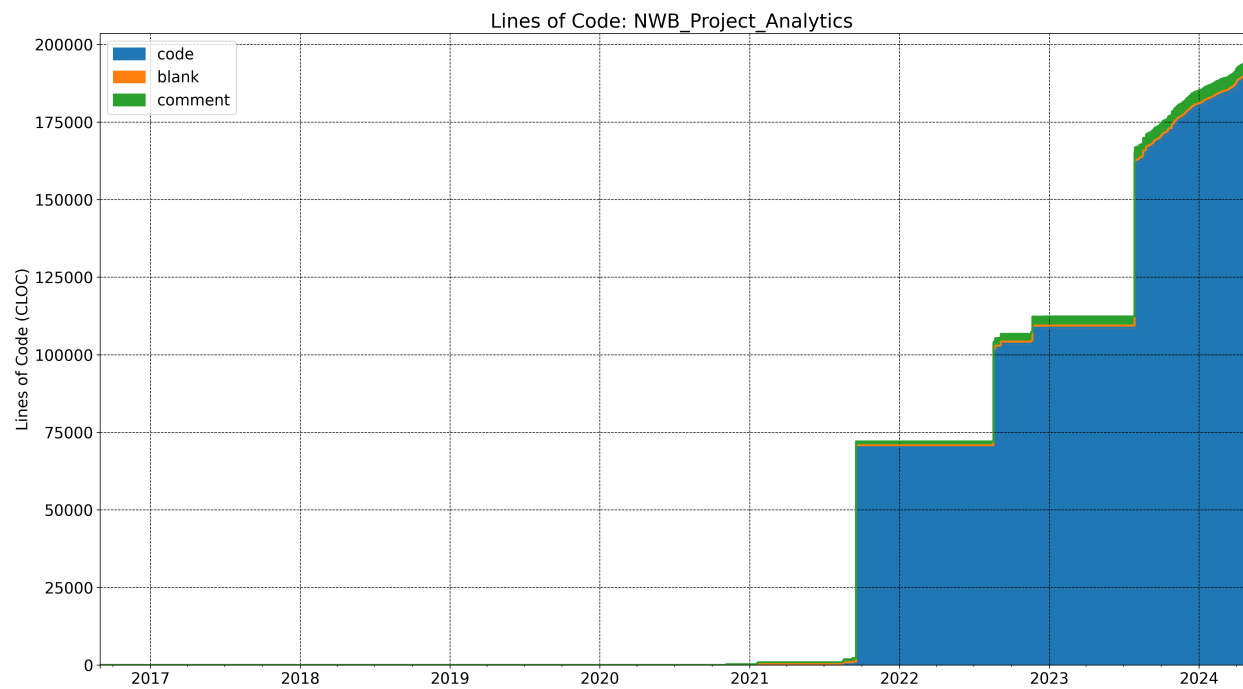
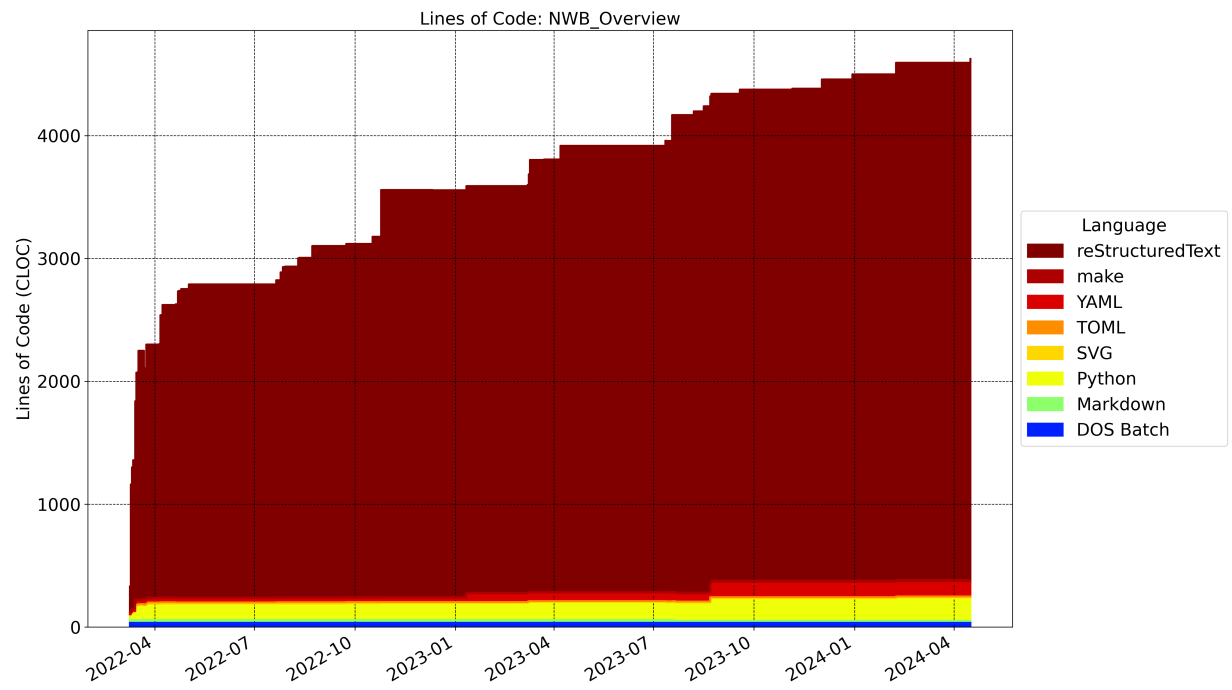
- Source: <https://github.com/NeurodataWithoutBorders/nwb-overview.git> (main branch = main)
- Docs: <https://nwb-overview.readthedocs.io>

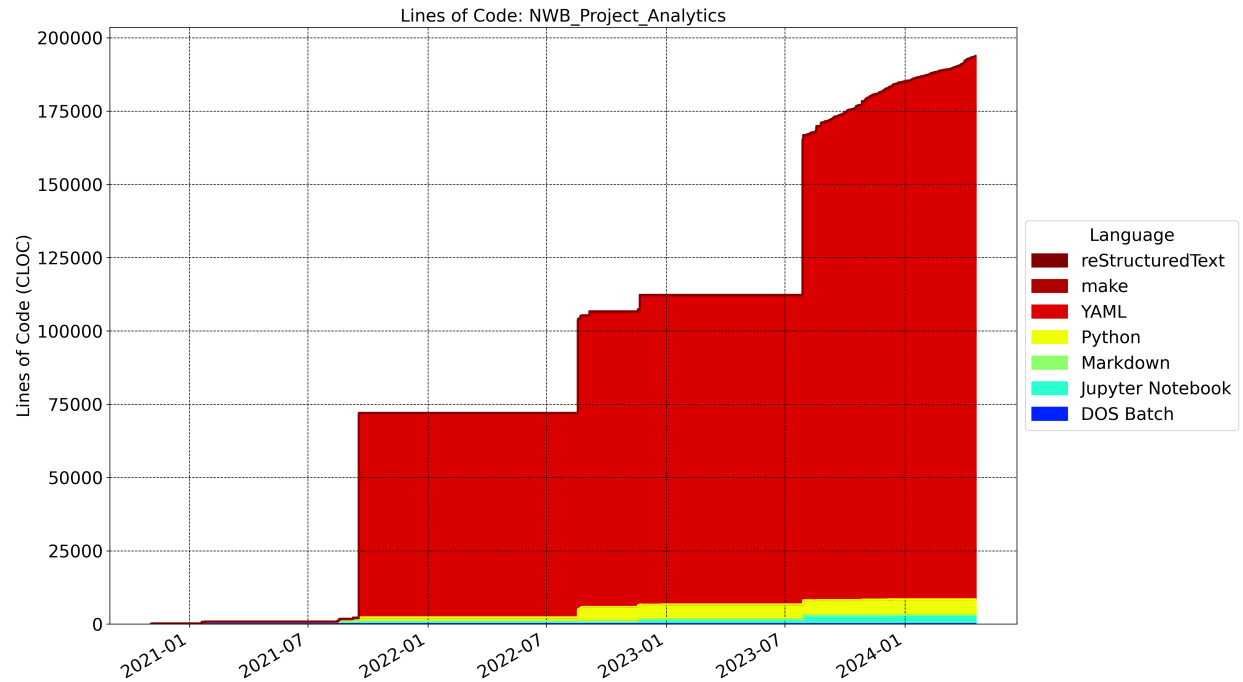
11.3.17 NWB_Project_Analytics

Lines of Code

Additional Information

- Source: <https://github.com/NeurodataWithoutBorders/nwb-project-analytics.git> (main branch = main)
- Docs: <https://github.com/NeurodataWithoutBorders/nwb-project-analytics>





11.3.18 NWB_Schema

Lines of Code

Release History

Additional Information

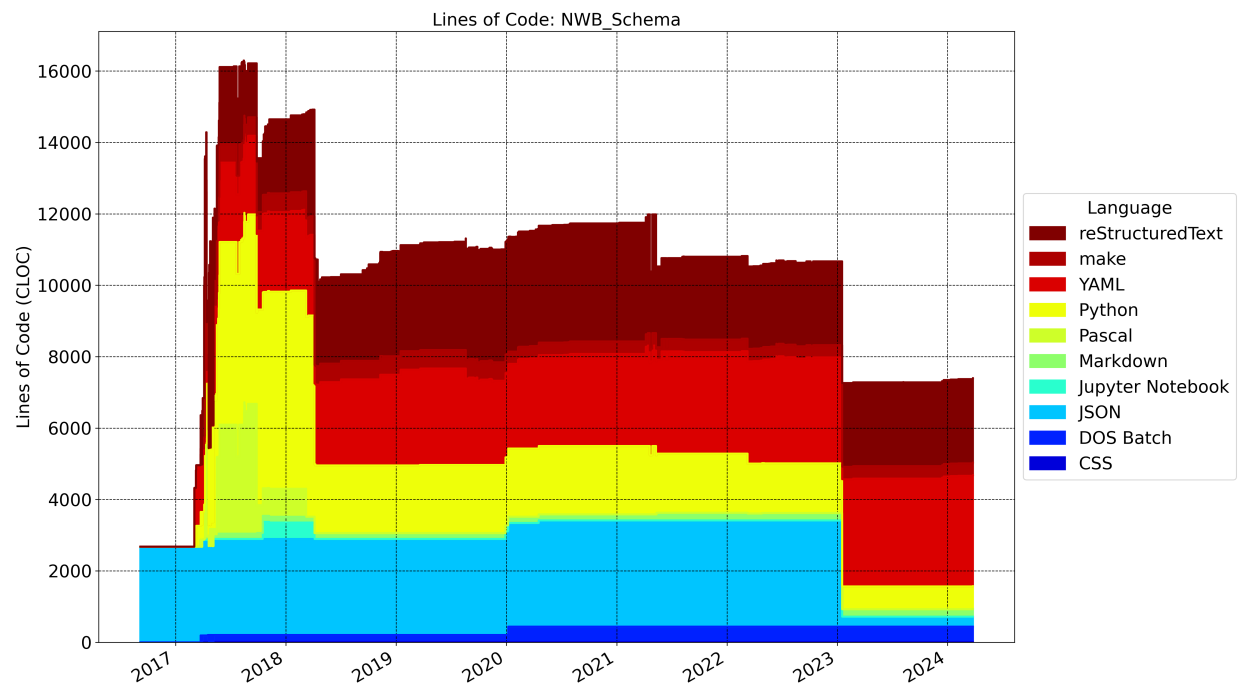
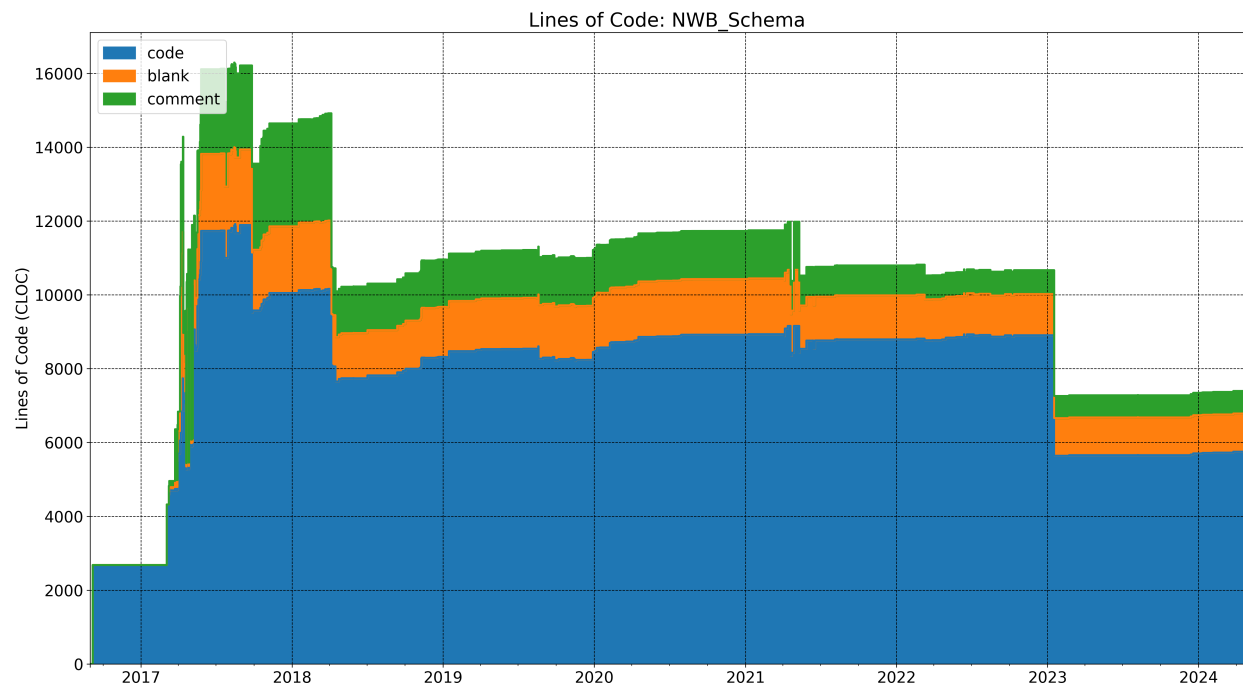
- Source: <https://github.com/NeurodataWithoutBorders/nwb-schema.git> (main branch = dev)
- Docs: <https://nwb-schema.readthedocs.io>

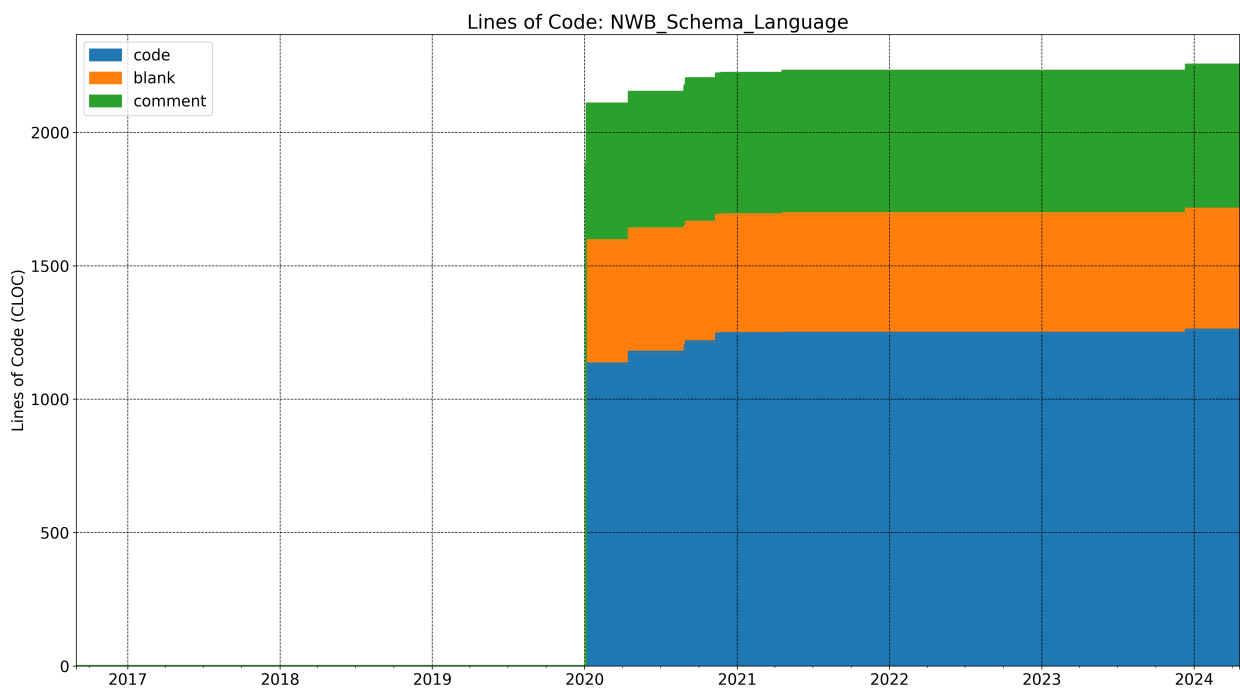
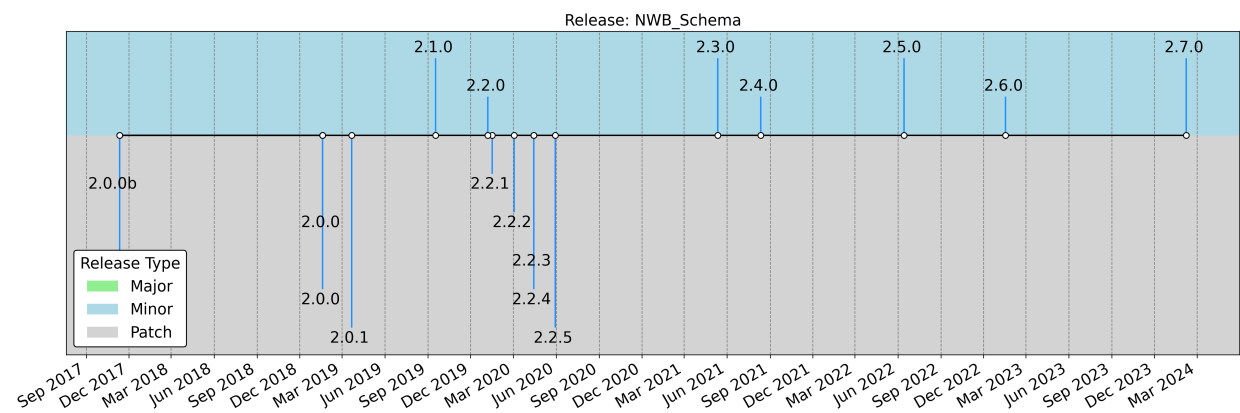
11.3.19 NWB_Schema_Language

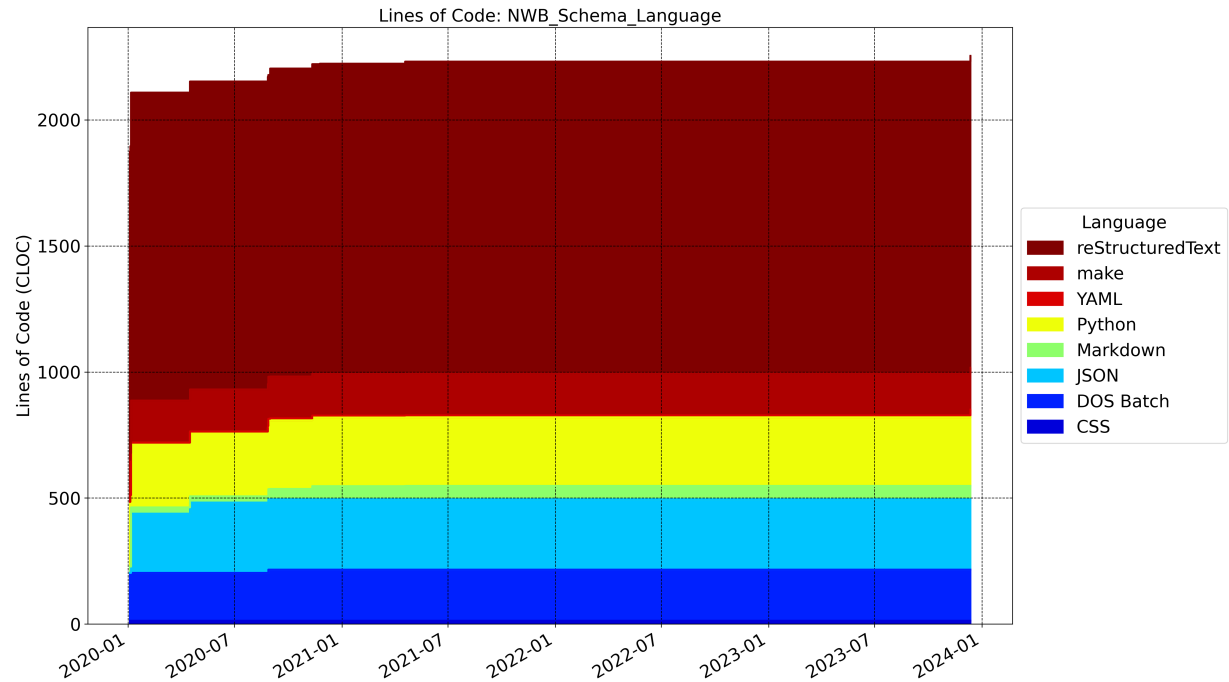
Lines of Code

Additional Information

- Source: <https://github.com/NeurodataWithoutBorders/nwb-schema-language.git> (main branch = main)
- Docs: <https://schema-language.readthedocs.io>







11.3.20 NeuroConv

Lines of Code

Release History

Additional Information

- Source: <https://github.com/catalystneuro/neuroconv.git> (main branch = main)
- Docs: <https://neuroconv.readthedocs.io>
- Logo: https://github.com/catalystneuro/neuroconv/blob/main/docs/img/neuroconv_logo.png

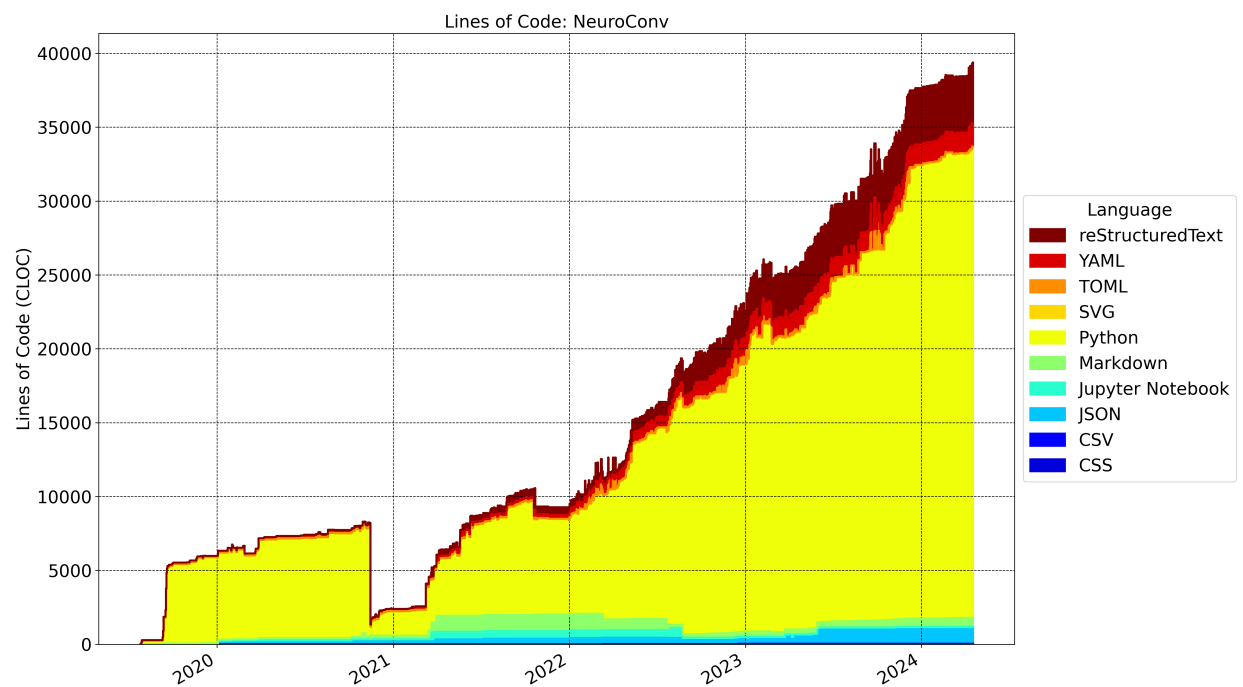
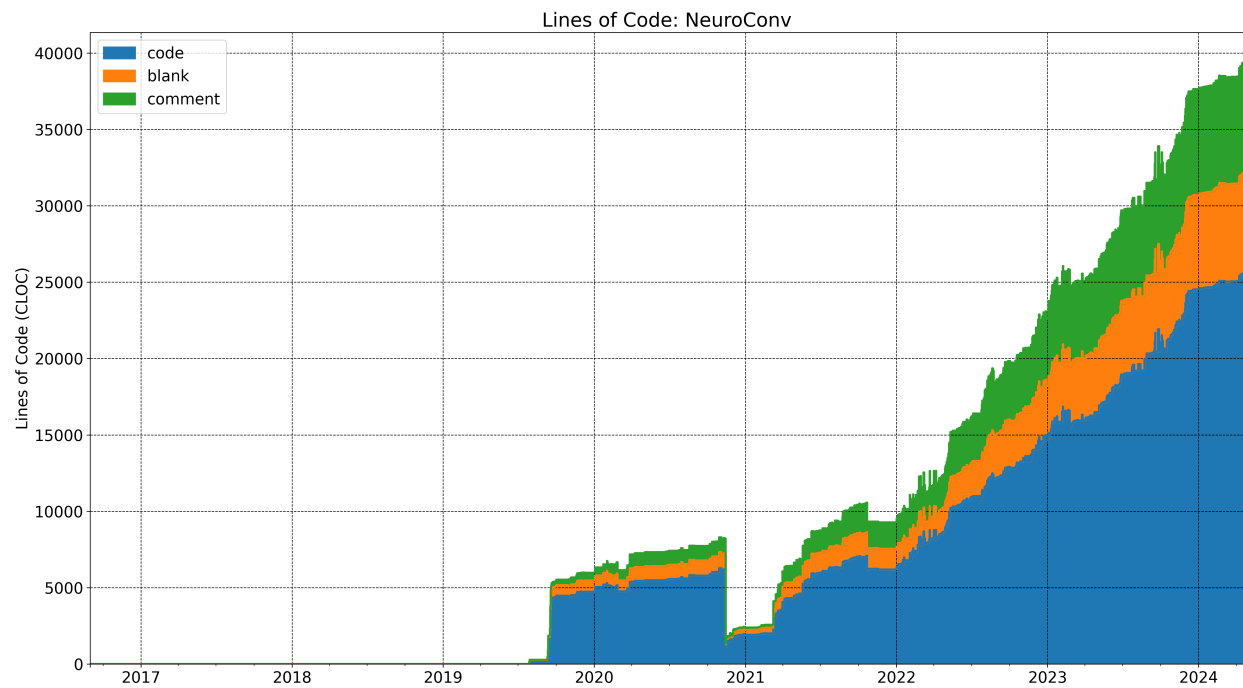
11.3.21 PyNWB

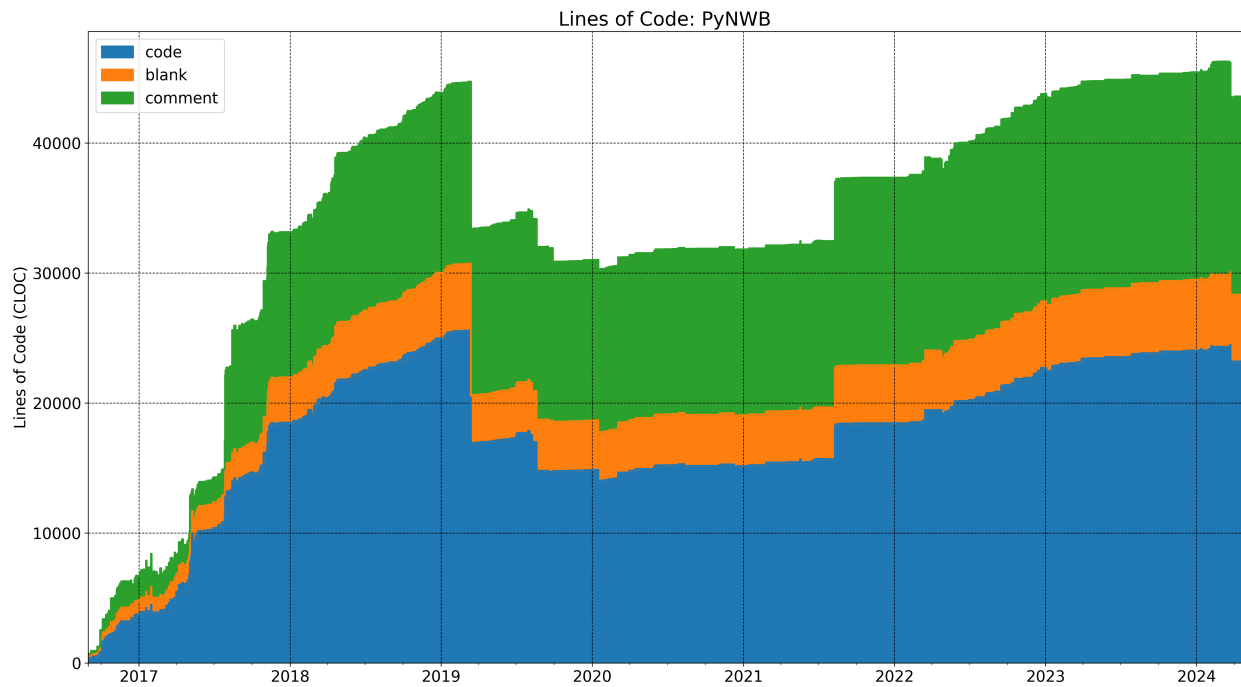
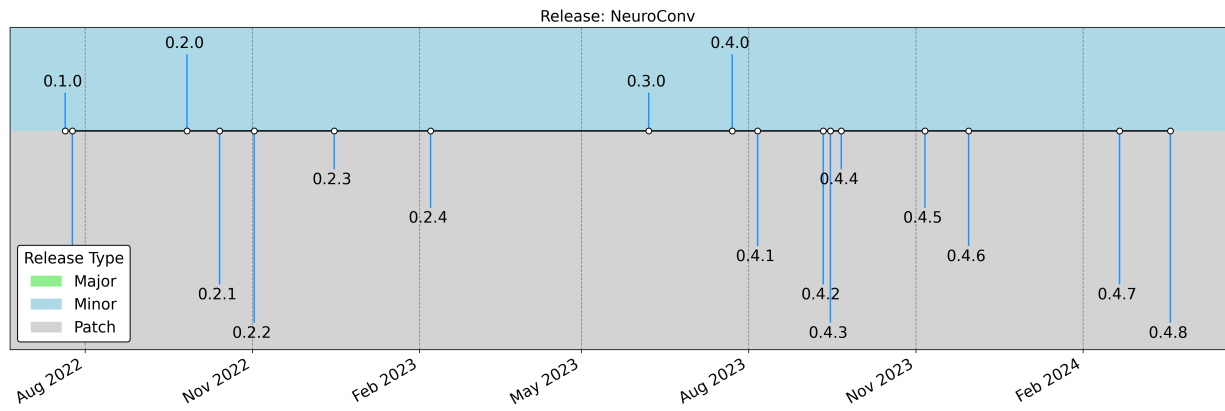
Lines of Code

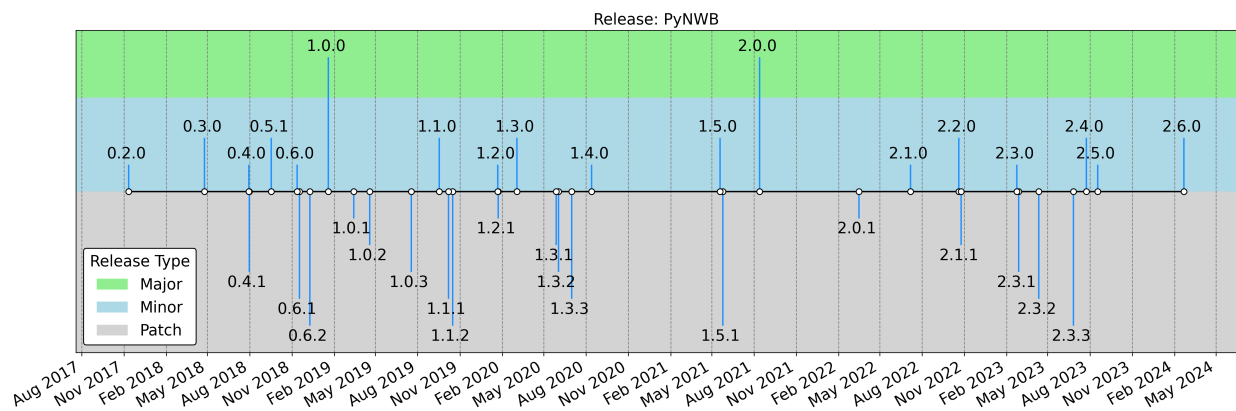
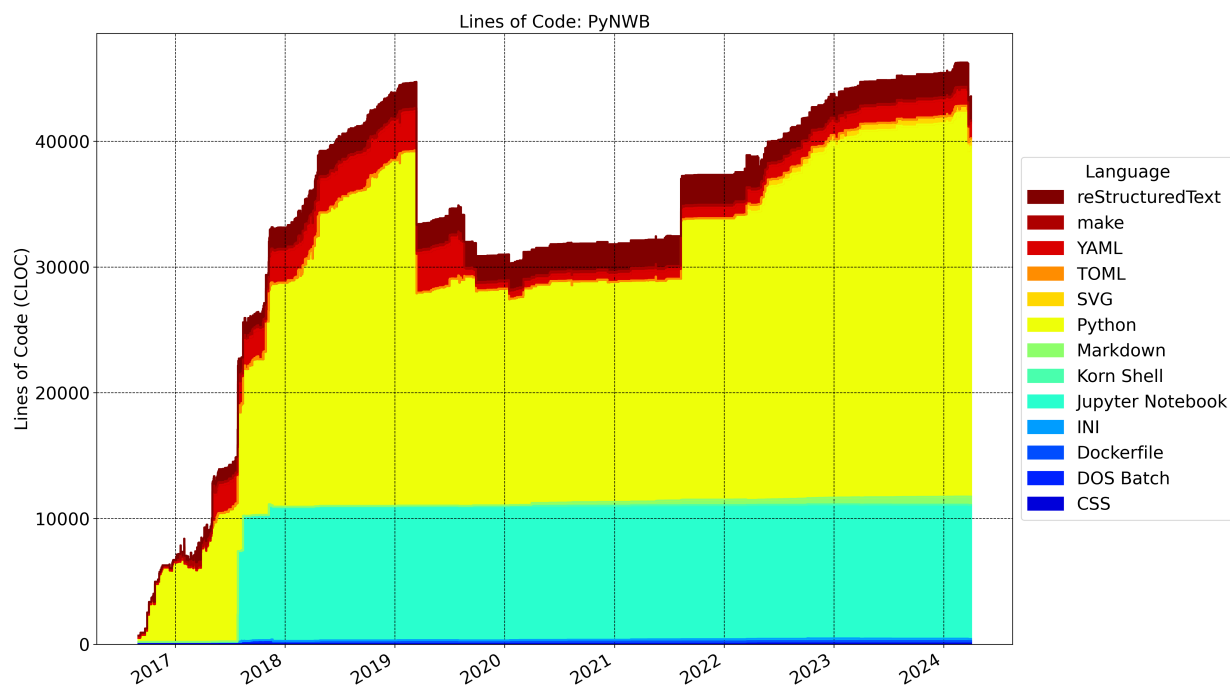
Release History

Additional Information

- Source: <https://github.com/NeurodataWithoutBorders/pynwb.git> (main branch = dev)
- Docs: <https://pynwb.readthedocs.io>
- Logo: https://raw.githubusercontent.com/NeurodataWithoutBorders/pynwb/dev/docs/source/figures/logo_pynwb.png







11.4 nwb_project_analytics package

11.4.1 Submodules

nwb_project_analytics.codecovstats module

Module for getting data from Codecov.io

class nwb_project_analytics.codecovstats.CodecovInfo

Bases: `object`

Helper class for interacting with Codecov.io

static `get_pulls_or_commits`(gitrepo, page_max=100, state='merged', key=None, branch=None)

Get all infos from pull request from Codecov.io

Parameters

- **gitrepo** (`GitRepo`) – `GitRepo` object with the owner and repo info
- **page_max** – Integer with the maximum number of pages to request. Set to `None` to indicate unlimited. (default=100)
- **state** – Filter list by state. One of all, open, closed, merged. Default: merged
- **key** – One of 'pulls' or 'commits'
- **branch** – Branch for which the stats should be retrieved. (Default=None)

Returns

List of dicts (one per pull request) in order of appearance on the page

static `get_time_and_coverage`(pulls_or_commits, filter_zeros=True)

Get the timestamps and total coverage information for all given pull requests.

Parameters

- **pulls** – List of dicts (one per pull request usually generated via the `CodecovInfo.get_pulls`
- **filter_zeros** – Boolean indicating whether coverage values of 0 should be removed

Returns

Tuple of three numpy arrays 1) Sorted array of datetime objects with the timestamps 2) Array of floats with the percent coverage data corresponding to the timestamps 3) Array of pulls missing coverage data

nwb_project_analytics.codestats module

Module for computing code statistics using CLOC

class nwb_project_analytics.codestats.GitCodeStats(output_dir: str, git_paths: dict | None = None)

Bases: `object`

Class with functions to compute code statistics for repos stored on git

The typical use is to

```
>> git_code_stats = GitCodeStats(...) >> git_code_stats.compute_code_stats(...) >>
git_code_stats.compute_summary_stats(...)
```

If results have been computed previously and cached then do:

```
>> git_code_stats = GitCodeStats.from_cache(...) >> git_code_stats.compute_summary_stats(...)
```

We can check if valid cached files exist via `GitCodeStats.cached()` or the `from_cache` function will raise a `ValueError` if the cache does not exist.

Variables

- **git_paths** – Dict of strings with the keys being the name of the tool and the values being the git URL, e.g., `'https://github.com/NeurodataWithoutBorders/pynwb.git'`.
- **output_dir** – Path to the directory where outputs are being stored
- **source_dir** – Path where the sources of repos are being checked out to. (`self.output_dir/src`)
- **cache_file_cloc** – Path to the YAML file for storing cloc statistics (may not exist if results are not cached)
- **cache_file_commits** – Path to the YAML file with the commit stats (may not exist if results are not cached)
- **cloc_stats** – Dict with the CLOC statistics
- **commit_stats** – Dict with the commit statistics.
- **summary_stats** – Dict with time-aligned summary statistics for all repos. The values of the dict are `pandas.DataFrame` objects and the keys are strings with the statistic type, i.e., `'sizes'`, `'blank'`, `'codes'`, `'comment'`, `'nfiles'`
- **contributors** – Pandas dataframe with contributors to the various repos determined via `get_contributors` and `merge_contributors`. NOTE: During calculation this will include the `'email'` column with the emails corresponding to the `'name'` of the user. However, when loading from cache the email column may not be available as a user may choose to not cache the email column, e.g., due to privacy concerns (even though the information is usually compiled from Git logs of public code repositories)

static `cached(output_dir)`

Check if a complete cached version of this class exists at `output_dir`

static `clean_outdirs(output_dir, source_dir)`

Delete the output directory and all its contents and create a new clean directory. Create a new `source_dir`.

Parameters

- **output_dir** – Output directory for caching results
- **source_dir** – Directory for storing repos checked out from git

Returns

A tuple of two strings with the `output_dir` and `source_dir` for git sources

static `clone_repos(repos, source_dir)`

Clone all of the given repositories.

Parameters

- **repos** – Dict where the keys are the names of the repos and the values are the git source path to clone
- **source_dir** – Directory where all the git repos should be cloned to. Each repo will be cloned into a subdirectory in `source_dir` that is named after the corresponding key in the `repos` dict.

Returns

Dict where the keys are the same as in repos but the values are instances of `git.repo.base.Repo` pointing to the corresponding git repository.

compute_code_stats(*cloc_path*: *str*, *clean_source_dir*: *bool* = *False*, *contributor_params*: *dict* | *None* = *None*)

Compute code statistics using CLOC.

NOTE: Repos will be checked out from GitHub and CLOC computed for all

commits, i.e., the repo will be checked out at all commits of the repo and CLOC will be run. This process can be very expensive. Using the cache is recommended when possible.

WARNING: This function calls `self.clean_outdirs`. Any previously cached results will be lost!

Parameters

- **cloc_path** – Path to the cloc command for running cloc stats
- **clean_source_dir** – Bool indicating whether to remove `self.source_dir` when finished
- **contributor_params** – dict of string indicating additional command line parameters to pass to *git shortlog*. E.g., `–since="3 years"`. Similarly we may specify `–after`, `–before` and `–until`.

Returns

None. The function initializes `self.commit_stats`, `self.cloc_stats`, and `self.contributors`

compute_language_stats(*ignore_lang*=*None*)

Compute for each code the breakdown in lines-of-code per language (including blank, comment, and code lines for each language).

The index of the resulting dataframe will typically be different for each code as changes occurred on different dates. The index reflects dates on which code changes occurred.

Parameters

ignore_lang – List of languages to ignore. Usually ['SUM', 'header'] are useful to ignore.

Returns

Dictionary of `pandas.DataFrame` objects with the language stats for the different repos

compute_summary_stats(*date_range*)

Compile summary of line-of-code (LOC) across repos by categories: sizes, blanks, codes, comments, and nfiles

The goal is to align and expand results from all repos so that we can plot them together. Here we create a continuous date range and expand the results from all repos to align with our common time axis. For dates where no new CLOC stats are recorded for a repo, the statistics from the previous time are carried forward to fill in the gaps.

Parameters

date_range (*pandas.date_range*) – Pandas datarange object for which the stats should be computed

Returns

Dict where the values are Pandas `DataFrame` objects with summary statistics and the keys are strings with the statistic type, i.e., 'sizes', 'blank', 'codes', 'comment', 'nfiles'

static from_cache(*output_dir*)

Create a `GitCodeStats` object from cached results :param *output_dir*: The output directory where the cache files are stored :return: A new `GitCodeStats` object with the results loaded from the cache

```
static from_nwb(cache_dir: str, cloc_path: str, start_date: datetime | None = None, end_date: datetime |
               None = None, read_cache: bool = True, write_cache: bool = True,
               cache_contributor_emails: bool = False, clean_source_dir: bool = True)
```

Convenience function to compute GitCodeStats statistics for all NWB git repositories defined by `GitRepos.merge(NWBGitInfo.GIT_REPOS, NWBGitInfo.NWB1_GIT_REPOS)`

For HDMF and the NDX_Extension_Smithy code statistics before the start date of the repo are set to 0. HDMF was extracted from PyNWB and as such, while there is code history before the official date for HDMF that part is history of PyNWB and so we set those values to 0. Similarly, the NDX_Extension_Smithy also originated from another code.

Parameters

- **cache_dir** – Path to the director where the files with the cached results are stored or should be written to
- **cloc_path** – Path to the cloc shell command
- **start_date** – Start date from which to compute summary statistics from. If set to `None`, then use `NWBGitInfo.NWB2_START_DATE`
- **end_date** – End date until which to compute summary statistics to. If set to `None`, then use `datetime.today()`
- **read_cache** – Bool indicating whether results should be loaded from cache if cached files exists at `cache_dir`. NOTE: If `read_cache` is `True` and files are in the cache then the results will be loaded without checking results (e.g., whether results in the cache are complete and up-to-date).
- **write_cache** – Bool indicating whether to write the results to the cache.
- **cache_contributor_emails** – Save the emails of contributors in the cached TSV file
- **clean_source_dir** – Bool indicating whether to remove `self.source_dir` when finished computing the code stats. This argument only takes effect when code statistics are computed (i.e., not when data is loaded from cache)

Returns

Tuple with the: 1) `GitCodeStats` object with all NWB code statistics and 2) dict with the results from `GitCodeStats.compute_summary_stats` 3) dict with language statistics computed via `GitCodeStats.compute_language_stats` 4) list of all languages used 5) dict with the release date and timeline statistics

```
static get_contributors(repo: Repo | str, contributor_params: str | None = None)
```

Compute list of contributors for the given repo using `git shortlog -summary -numbered -email`

Parameters

- **repo** – The git repository to process
- **contributor_params** – String indicating additional command line parameters to pass to `git shortlog`. E.g., `-since="3 years"`. Similarly we may specify `-since`, `-after`, `-before` and `-until`.

:return Pandas dataframe with the name, email, and number of contributions to the repo

```
get_languages_used(ignore_lang=None)
```

Get the list of languages used in the repos

Parameters

- **ignore_lang** – List of strings with the languages that should be ignored. (Default=`None`)

Returns

array of strings with the unique set of languages used

static `git_repo_stats(repo: Repo, cloc_path: str, output_dir: str)`

Compute cloc statistics for the given repo.

Run cloc only for the last commit on each day to avoid excessive runs

Parameters

- **repo** – The git repository to process
- **cloc_path** – Path to run cloc on the command line
- **output_dir** – Path to the directory where outputs are being stored

Returns

The function returns 2 elements, `commit_stats` and `cloc_stats`. `commit_stats` is a list of dicts with information about all commits. The list is sorted in time from most current [0] to oldest [-1]. `cloc_stats` is a list of dicts with CLOC code statistics. CLOC is run only on the last commit on each day to reduce the number of codecov runs and speed-up computation.

static `merge_contributors(data_frames: dict, merge_duplicates: bool = True)`

Take dict of dataframes generated by `GitCodeStats.get_contributors` and merge them into a single dataframe

Parameters

- **data_frames** – Dict of dataframes where the keys are the names of repos
- **merge_duplicates** – Attempt to detect and merge duplicate contributors by name and email

Returns

Combined pandas dataframe

static `run_cloc(cloc_path, src_dir, out_file)`

Run CLOC on the given srcdir, save the results to outdir, and return the parsed results.

write_to_cache(cache_contributor_emails: bool = False)

Save the stats to YAML and contributors to TSV.

Results will be saved to the `self.cache_file_cloc`, `self.cloc_stats`, `self.cache_file_commits`, `self.cache_git_paths`, and `self.cache_contributors_paths`

Parameters

cache_contributor_emails – Save the emails of contributors in the cached TSV file

nwb_project_analytics.create_codestat_pages module

Script for creating rst pages and figures with NWB code statistics

`nwb_project_analytics.create_codestat_pages.create_codestat_pages(out_dir: str, data_dir: str, cloc_path: str = 'cloc', load_cached_results: bool = True, cache_results: bool = True, cache_contributor_emails: bool = False, start_date: datetime | None = None, end_date: datetime | None = None, print_status: bool = True)`

Main function used to render all pages and figures related to the tool statistics

Parameters

- **out_dir** – Directory where the RST and image files should be saved to
- **data_dir** – Directory where the data for the code statistics should be cached
- **cloc_path** – Path to the cloc tool if not callable directly via “cloc”
- **load_cached_results** – Load code statistics from data_dir if available
- **cache_results** – Save code statistic results to data_dir
- **cache_contributor_emails** – Save the emails of contributors in the cached TSV file
- **start_date** – Datetime object with the start date for plots. If None then NWBGit-Info.NWB2_START_DATE will be used as default.
- **end_date** – Datetime object with the end date for plots. If None then datetime.today() will be used as default.
- **print_status** – Print status of creation (Default=True)

```
nwb_project_analytics.create_codestat_pages.create_toolstat_page(out_dir: str, repo_name: str,
                                                                repo: GitRepo, figures:
                                                                OrderedDict, print_status: bool
                                                                = True)
```

Create a page with the statistics for a particular tool

Parameters

- **out_dir** – Directory where the RST file should be saved to
- **repo_name** – Name of the code repository
- **figures** – OrderedDict of RSTFigure object to render on the page
- **print_status** – Print status of creation (Default=True)

Returns

```
nwb_project_analytics.create_codestat_pages.init_codestat_pages_dir(out_dir)
Delete out_dir and all its contents and create a new clean out_dir :param out_dir: Directory to be removed :return:
```

nwb_project_analytics.gitstats module

Module for querying GitHub repos

```
class nwb_project_analytics.gitstats.GitHubRepoInfo(repo)
```

Bases: `object`

Helper class to get information about a repo from GitHub

Variables

repo – a GitRepo tuple with the owner and name of the repo

```
static collect_all_release_names_and_date(repos: dict, cache_dir: str, read_cache: bool = True,
                                          write_cache: bool = True)
```

get_release_names_and_dates(kwargs)**

Get names and dates of releases :param kwargs: Additional keyword arguments to be passed to self.get_releases

Returns

Tuple with the list of names as strings and the list of dates as datetime objects

get_releases(use_cache=True)

Get the last 100 release for the given repo

NOTE: GitHub uses pagination. Here we set the number of items per page to 100

which should usually fit all releases, but in the future we may need to iterate over pages to get all the releases not just the latest 100. Possible implementation <https://gist.github.com/victorbordo/5581fd8b89ed93bf3eb2b478529b9e38>

Parameters

use_cache – If set to True then return the cached results if computed previously. In this case the per_page parameter will be ignored

Raises

Error if response is not Ok, e.g., if the GitHub request limit is exceeded.

Returns

List of dicts with the release data

static get_version_jump_from_tags(tags)

Assuming semantic versioning release tags get the version jumps from the tags

Returns

OrderedDict

static releases_from_nwb(cache_dir: str, read_cache: bool = True, write_cache: bool = True)

class nwb_project_analytics.gitstats.GitRepo(owner: str, repo: str, mainbranch: str, docs: str | None = None, logo: str | None = None, startdate: datetime | None = None)

Bases: `tuple`

Named tuple with basic information about a GitHub repository

static compute_issue_time_of_first_response(issue)

For a given GitHub issue compute the time to first response based on the the issue's timeline

docs: str

Online documentation for the software

get_commits_as_dataframe(since, github_obj, tqdm)

Get a dataframe for all commits with updates later than the given data

Parameters

- **since** – Datetime object with the date of the oldest issue to retrieve
- **github_obj** – PyGitHub github.Github object to use for retrieving issues
- **tqdm** – Supply the tqdm progress bar class to use

Returns

Pandas DataFrame with the commits data

get_issues_as_dataframe(*since*, *github_obj*, *tqdm=None*)

Get a dataframe for all issues with updates later than the given data

Parameters

- **since** – Datetime object with the date of the oldest issue to retrieve
- **github_obj** – PyGitHub github.Github object to use for retrieving issues
- **tqdm** – Supply the tqdm progress bar class to use

Returns

Pandas DataFrame with the issue data

property github_issues_url

URL for GitHub issues page

property github_path

https path for the git repo

property github_pulls_url

URL for GitHub pull requests page

logo: *str*

URL with the PNG of the logo for the repository

mainbranch: *str*

The main branch of the repository

owner: *str*

Owner of the repo on GitHub

repo: *str*

Name of the repository

startdate: *datetime*

Some repos start from forks so we want to track statistics starting from then rather than the beginning of time

class `nwb_project_analytics.gitstats.GitRepos`(*arg, **kw)

Bases: *OrderedDict*

Dict where the keys are names of codes and the values are GitRepo objects

get_info_objects()

Get an *OrderedDict* of *GitHubRepoInfo* object from the repos

static merge(o1, o2)

Merge two *GitRepo* dicts and return a new *GitRepos* dict with the combined items

class `nwb_project_analytics.gitstats.IssueLabel`(*label: str*, *description: str*, *color: str*)

Bases: *tuple*

Named tuple describing a label for issues on a Git repository.

color: *str*

Hex code of the color for the label

description: *str*

Description of the lable

label: str

Label of the issue, usually consisting <type>: <level>. <type> indicates the general area the label is used for, e.g., to assign a category, priority, or topic to an issue. <level> then indicates importance or sub-category with the given <type>, e.g., critical, high, medium, low level as part of the priority type

property level

Get the level of the issue, indicating the importance or sub-category of the label within the given self.type, e.g., critical, high, medium, low level as part of the priority type.

Returns

str with the level or None in case the label does not have a level (e.g., if the label does not contain a “:” to separate the type and level).

property rgb

Color code converted to RGB

Returns

Tuple of ints with (red, green, blue) color values

property type

Get the type of the issue label indicating the general area the label is used for, e.g., to assign a category, priority, or topic to an issue.

Returns

str with the type or None in case the label does not have a category (i.e., if the label does not contain a “:” to separate the type and level).

class nwb_project_analytics.gitstats.**IssueLabels**(*arg, **kw)

Bases: `OrderedDict`

OrderedDict where the keys are names of issues labels and the values are IssueLabel objects

property colors

Get a list of all color hex codes uses

get_by_type(label_type)

Get a new IssueLabels dict with just the lables with the given category

property levels

Get a list of all level strings used in labels (may include Node)

static merge(o1, o2)

Merger two IssueLabels dicts and return a new IssuesLabels dict with the combined items

property rgbs

Get a list of all rgb color codes used

property types

Get a list of all type strings used in labels (may include None)

class nwb_project_analytics.gitstats.**NWBGitInfo**

Bases: `object`

Class for storing basic information about NWB repositories

class property CORE_API_REPOS

Dictionary with the main NWB git repos related the user APIs.

```
CORE_DEVELOPERS = ['rly', 'bendichter', 'oruebel', 'ajtritt', 'ln-vidrio',  
'mavaylon1', 'CodyCBakerPhD', 'stephprince', 'lawrence-mbf', 'dependabot[bot]',  
'nwb-bot', 'hdmf-bot', 'pre-commit-ci[bot]']
```

List of names of the core developers of NWB overall. These are used, e.g., when analyzing issue stats as core developer issues should not count against user issues.

```

GIT_REPOS = {'HDMF': GitRepo(owner='hdmf-dev', repo='hdmf', mainbranch='dev',
docs='https://hdmf.readthedocs.io', logo='https://raw.githubusercontent.com/
hdmf-dev/hdmf/dev/docs/source/hdmf_logo.png', startdate=datetime.datetime(2019, 3,
13, 0, 0)), 'HDMF_Common_Schema': GitRepo(owner='hdmf-dev',
repo='hdmf-common-schema', mainbranch='main',
docs='https://hdmf-common-schema.readthedocs.io', logo=None, startdate=None),
'HDMF_DocUtils': GitRepo(owner='hdmf-dev', repo='hdmf-docutils', mainbranch='main',
docs=None, logo=None, startdate=None), 'HDMF_Schema_Language':
GitRepo(owner='hdmf-dev', repo='hdmf-schema-language', mainbranch='main',
docs='https://hdmf-schema-language.readthedocs.io/', logo=None, startdate=None),
'HDMF_Zarr': GitRepo(owner='hdmf-dev', repo='hdmf-zarr', mainbranch='dev',
docs='https://hdmf-zarr.readthedocs.io', logo='https://raw.githubusercontent.com/
hdmf-dev/hdmf-zarr/dev/docs/source/figures/logo_hdmf_zarr.png', startdate=None),
'Hackathons': GitRepo(owner='NeurodataWithoutBorders', repo='nwb_hackathons',
mainbranch='main', docs='https://neurodatawithoutborders.github.io/nwb_hackathons/',
logo=None, startdate=None), 'MatNWB': GitRepo(owner='NeurodataWithoutBorders',
repo='matnwb', mainbranch='master',
docs='https://neurodatawithoutborders.github.io/matnwb/', logo='https://raw.
githubusercontent.com/NeurodataWithoutBorders/matnwb/master/logo/logo_matnwb.png',
startdate=None), 'NDX_Catalog': GitRepo(owner='nwb-extensions',
repo='nwb-extensions.github.io', mainbranch='main',
docs='https://nwb-extensions.github.io/', logo='https://github.com/nwb-extensions/
nwb-extensions.github.io/blob/main/images/ndx-logo-text.png', startdate=None),
'NDX_Extension_Smithy': GitRepo(owner='nwb-extensions',
repo='nwb-extensions-smithy', mainbranch='master', docs=None, logo=None,
startdate=datetime.datetime(2019, 4, 25, 0, 0)), 'NDX_Staged_Extensions':
GitRepo(owner='nwb-extensions', repo='staged-extensions', mainbranch='master',
docs=None, logo=None, startdate=None), 'NDX_Template':
GitRepo(owner='nwb-extensions', repo='ndx-template', mainbranch='main',
docs='https://nwb-overview.readthedocs.io/en/latest/extensions_tutorial/
2_create_extension_spec_walkthrough.html', logo=None, startdate=None),
'NWBInspector': GitRepo(owner='NeurodataWithoutBorders', repo='nwbinspector',
mainbranch='dev', docs='https://nwbinspector.readthedocs.io', logo='https://raw.
githubusercontent.com/NeurodataWithoutBorders/nwbinspector/dev/docs/logo/logo.png',
startdate=None), 'NWBWidgets': GitRepo(owner='NeurodataWithoutBorders',
repo='nwb-jupyter-widgets', mainbranch='master', docs=None,
logo='https://user-images.githubusercontent.com/844306/
254117081-f20b8c26-79c7-4c1c-a3b5-b49ecf8cce5d.png', startdate=None),
'NWB_Benchmarks': GitRepo(owner='NeurodataWithoutBorders', repo='nwb_benchmarks',
mainbranch='main', docs=None, logo=None, startdate=None), 'NWB_GUIDE':
GitRepo(owner='NeurodataWithoutBorders', repo='nwb-guide', mainbranch='main',
docs='https://github.com/NeurodataWithoutBorders/nwb-guide',
logo='https://raw.githubusercontent.com/NeurodataWithoutBorders/nwb-guide/main/src/
renderer/assets/img/logo-guide-draft-transparent-tight.png',
startdate=datetime.datetime(2022, 11, 21, 0, 0)), 'NWB_Overview':
GitRepo(owner='NeurodataWithoutBorders', repo='nwb-overview', mainbranch='main',
docs='https://nwb-overview.readthedocs.io', logo=None, startdate=None),
'NWB_Project_Analytics': GitRepo(owner='NeurodataWithoutBorders',
repo='nwb-project-analytics', mainbranch='main',
docs='https://github.com/NeurodataWithoutBorders/nwb-project-analytics', logo=None,
startdate=None), 'NWB_Schema': GitRepo(owner='NeurodataWithoutBorders',
repo='nwb-schema', mainbranch='dev', docs='https://nwb-schema.readthedocs.io',
logo=None, startdate=None), 'NWB_Schema_Language':
GitRepo(owner='NeurodataWithoutBorders', repo='nwb-schema-language',
mainbranch='main', docs='https://schema-language.readthedocs.io', logo=None,
startdate=None), 'NeuroConv': GitRepo(owner='catalystneuro', repo='neuroconv',
docs='https://neuroconv.readthedocs.io', logo='https://github.com/catalystneuro/neuroconv/blob/main/docs/img/neuroconv_logo.png', startdate=None),
'PyNWB': GitRepo(owner='NeurodataWithoutBorders', repo='pynwb', mainbranch='dev',
docs='https://pynwb.readthedocs.io', logo='https://raw.githubusercontent.com/

```

Dictionary with main NWB git repositories. The values are GitRepo tuples with the owner and repo name.

HDMF_START_DATE = datetime.datetime(2019, 3, 13, 0, 0)

HDMF was originally part of PyNWB. As such code statistics before this start date for HDMF reflect stats that include both PyNWB and HDMF and will result in duplicate counting of code stats if PyNWB and HDMF are shown together. For HDMF 2019-03-13 coincides with the removal of HDMF from PyNWB with PR #850 and the release of HDMF 1.0. For the plotting 2019-03-13 is therefore a good date to start considering HDMF stats to avoid duplication of code in statistics, even though the HDMF repo existed on GitHub already since 2019-01-23T23:48:27Z, which could be alternatively considered as the start date. Older dates will include code history carried over from PyNWB to HDMF. Set to None to consider the full history of HDMF but as mentioned, this will lead to some duplicate counting of code before 2019-03-13

MISSING_RELEASE_TAGS = {'MatNWB': [(0.1.0b, datetime.datetime(2017, 11, 11, 0, 0))], 'NWB_Schema': [(2.0.0, datetime.datetime(2019, 1, 19, 0, 0)), (2.0.0b, datetime.datetime(2017, 11, 11, 0, 0))]}

List of early releases that are missing a tag on GitHub

NWB1_DEPRECATION_DATE = datetime.datetime(2016, 8, 1, 0, 0)

Date when to declare the NWB 1.0 APIs as deprecated. The 3rd Hackathon was held on July 31 to August 1, 2017 at Janelia Farm, in Ashburn, Virginia, which marks the date when NWB 2.0 was officially accepted as the follow-up to NWB 1.0. NWB 1.0 as a project ended about 1 year before that.

NWB1_GIT_REPOS = {'NWB_1.x_Matlab': GitRepo(owner='NeurodataWithoutBorders', repo='api-matlab', mainbranch='dev', docs=None, logo=None, startdate=None), 'NWB_1.x_Python': GitRepo(owner='NeurodataWithoutBorders', repo='api-python', mainbranch='dev', docs=None, logo=None, startdate=None)}

Dictionary with main NWB 1.x git repositories. The values are GitRepo tuples with the owner and repo name.

NWB2_BETA_RELEASE = datetime.datetime(2017, 11, 11, 0, 0)

Date of the first official beta release of NWB 2 as part of SfN 2017

NWB2_FIRST_STABLE_RELEASE = datetime.datetime(2019, 1, 19, 0, 0)

Date of the first official stable release of NWB 2.0

NWB2_START_DATE = datetime.datetime(2016, 8, 31, 0, 0)

Date of the first release of PyNWB on the NWB GitHub. While some initial work was ongoing before that date, this was the first public release of code related to NWB 2.x

NWB_EXTENSION_SMITHY_START_DATE = datetime.datetime(2019, 4, 25, 0, 0)

NWB_Extension_Smithy is a fork with changes. We therefore should count only the sizes after the fork data which based on <https://api.github.com/repos/nwb-extensions/nwb-extensions-smithy> is 2019-04-25T20:56:02Z

NWB_GUIDE_START_DATE = datetime.datetime(2022, 11, 21, 0, 0)

NWB GUIDE was forked from SODA so we want to start tracking stats starting from that date


```

STANDARD_ISSUE_LABELS = {'category: bug': IssueLabel(label='category: bug',
description='errors in the code or code behavior', color='#ee0701'), 'category:
enhancement': IssueLabel(label='category: enhancement', description='improvements
of code or code behavior', color='#1D76DB'), 'category: proposal':
IssueLabel(label='category: proposal', description='discussion of proposed
enhancements or new features', color='#dddddd'), 'compatibility: breaking change':
IssueLabel(label='compatibility: breaking change', description='fixes or
enhancements that will break schema or API compatibility', color='#B24AD1'), 'help
wanted: deep dive': IssueLabel(label='help wanted: deep dive',
description='request for community contributions that will involve many parts of the
code base', color='#0E8A16'), 'help wanted: good first issue':
IssueLabel(label='help wanted: good first issue', description='request for
community contributions that are good for new contributors', color='#0E8A16'),
'priority: critical': IssueLabel(label='priority: critical', description='impacts
proper operation or use of core function of NWB or the software', color='#a0140c'),
'priority: high': IssueLabel(label='priority: high', description='impacts proper
operation or use of feature important to most users', color='#D93F0B'), 'priority:
low': IssueLabel(label='priority: low', description='alternative solution already
working and/or relevant to only specific user(s)', color='#FEF2C0'), 'priority:
medium': IssueLabel(label='priority: medium', description='non-critical problem
and/or affecting only a small set of NWB users', color='#FBCA04'), 'priority:
wontfix': IssueLabel(label='priority: wontfix', description='will not be fixed due
to low priority and/or conflict with other feature/priority', color='#ffffff'),
'topic: docs': IssueLabel(label='topic: docs', description='Issues related to
documentation', color='#D4C5F9'), 'topic: testing': IssueLabel(label='topic:
testing', description='Issues related to testing', color='#D4C5F9')}

```

nwb_project_analytics.renderstats module

Module with routines for plotting code and git statistics

```

class nwb_project_analytics.renderstats.PatchedMPLPlot(data, kind=None, by: IndexLabel | None =
None, subplots: bool |
Sequence[Sequence[str]] = False,
sharex=None, sharey: bool = False,
use_index: bool = True, figsize=None,
grid=None, legend: bool | str = True,
rot=None, ax=None, fig=None, title=None,
xlim=None, ylim=None, xticks=None,
yticks=None, xlabel: Hashable | None =
None, ylabel: Hashable | None = None,
fontsize=None, secondary_y: bool | tuple |
list | np.ndarray = False, colormap=None,
table: bool = False, layout=None,
include_bool: bool = False, column:
IndexLabel | None = None, **kws)

```

Bases: `MPLPlot`

axes: `np.ndarray`

```
class nwb_project_analytics.renderstats.RenderClocStats
```

Bases: `object`

Helper class for rendering code line statistics generated using `GitCodeStats`, e.g., via `GitCodeStats.from_nwb`.

```
static plot_cloc_sizes_stacked_area(summary_stats: dict, order: list | None = None, colors: list |  
                                   None = None, title: str | None = None, fontsize: int = 20)
```

Stacked curve plot of code size statistics

Parameters

- **summary_stats** – dict with the results form `GitCodeStats.compute_summary_stats`
- **order** – List of strings selecting the order in which codes should be stacked in the plot. If set to none then all keys in `summary_stats` will be used sorted alphabetically.
- **colors** – List of colors to be used. One per repo. Must be the same length as `order`.

Returns

Matplotlib axis object used for plotting

```
static plot_reposize_code_comment_blank(summary_stats: dict, repo_name: str, title: str | None =  
                                       None)
```

Plot repository size broken down by code, comment, and blank for a particular repo

Parameters

- **summary_stats** – dict with the results form `GitCodeStats.compute_summary_stats`
- **repo_name** – Key in dataframes of `summary_stats` with the name of the code repository to plot.
- **title** – Title of the plot

Returns

Matplotlib axis object used for plotting

```
static plot_reposize_language(per_repo_lang_stats: dict, languages_used_all: list, repo_name: str,  
                             figsize: tuple | None = None, fontsize: int = 18, title: str | None =  
                             None)
```

Plot repository size broken down by language for a particular repo

To compute the language statistics for code repositories we can use

```
git_code_stats, summary_stats = GitCodeStats.from_nwb(...)  
ignore_lang = ['SUM', 'header']  
languages_used_all = git_code_stats.get_languages_used(ignore_lang)  
per_repo_lang_stats = git_code_stats.compute_language_stats(ignore_lang)
```

Parameters

- **per_repo_lang_stats** – Dict with per repository language statistics compute via `GitCodeStats.compute_language_statistics`
- **languages_used_all** – List/array with the languages uses
- **repo_name** – Key in dataframes of `summary_stats` with the name of the code repository to plot.
- **figsize** – Figure size tuple. Default=(18, 10)
- **fontsize** – Fontsize
- **title** – Title of the plot

Returns

Matplotlib axis object used for plotting

class nwb_project_analytics.renderstats.RenderCodecovInfo

Bases: `object`

Helper class for plotting CoedcovInfo data

classmethod `plot_codecov_grid`(*codecovs: dict*, *plot_xlim: tuple | None = None*, *fontsize: int = 16*, *basefilename: str | None = None*)

Plot coverage results for one or more codes as a single figure with one row per code so all codes appear in their own plots but with a shared x-axis for time and creating only a single file

Example for setting for codecovs:

```
codecovs = {r: CodecovInfo.get_pulls_or_commits(NWBGitInfo.GIT_REPOS[r],
                                                key='commits', state='all',
                                                branch=NWBGitInfo.GIT_REPOS[r].
↳mainbranch)
            for r in ['HDMF', 'PyNWB', 'MatNWB']}
```

Parameters

- **codecovs** – Dictionary where the keys are the names of the codes and the values are the output from `CodecovInfo.get_pulls_or_commits` defining the coverage timeline for each code.
- **plot_xlim** – Tuple of datetime objects defining the time-range of the x-axis. E.g., `plot_xlim=(datetime.strptime("2021-01-01", "%Y-%m-%d"), datetime.today())`
- **fontsize** – Fontsize to be used for axes label, tickmarks, and titles. (default=16)
- **basefilename** – Base name of the file(s) where the plots should be saved to. Set to `None` to only show but not save the plots. Figures will be saved as both PDF and PNG. (default=`None`)

classmethod `plot_codecov_individual`(*codecovs: dict*, *plot_xlim: tuple | None = None*, *fontsize: int = 16*, *figsize: tuple | None = None*, *title: str | None = None*)

Plot coverage results for a code as an individual figure

Example for setting for codecovs:

```
codecovs = {r: CodecovInfo.get_pulls_or_commits(NWBGitInfo.GIT_REPOS[r],
                                                key='commits', state='all',
                                                branch=NWBGitInfo.GIT_REPOS[r].
↳mainbranch)
            for r in ['HDMF', 'PyNWB', 'MatNWB']}
```

Parameters

- **codecovs** – Dictionary where the key is the name of the codes and the values are the output from `CodecovInfo.get_pulls_or_commits` defining the coverage timeline for each code.
- **plot_xlim** – Tuple of datetime objects defining the time-range of the x-axis. E.g., `plot_xlim=(datetime.strptime("2021-01-01", "%Y-%m-%d"), datetime.today())`
- **fontsize** – Fontsize to be used for axes label, tickmarks, and titles. (default=16)
- **figsize** – Figure size tuple. Default is (18,6)
- **title** – Optional title for the figure

Returns

Matplotlib figure

```
static plot_codecov_multiline(codecovs: dict, plot_xlim: tuple | None = None, fill_alpha: float = 0.2,  
                             fontsize: int = 16, title: str | None = None, figsize: tuple | None =  
                             None)
```

Plot coverage results for one or more codes as a single figure with each code represented by a line plot with optional filled area.

Example for setting for codecovs:

```
codecovs = {r: CodecovInfo.get_pulls_or_commits(NWBGitInfo.GIT_REPOS[r],  
                                              key='commits', state='all',  
                                              branch=NWBGitInfo.GIT_REPOS[r].  
↳mainbranch)  
          for r in ['HDMF', 'PyNWB', 'MatNWB']}
```

Parameters

- **codecovs** – Dictionary where the keys are the names of the codes and the values are the output from CodecovInfo.get_pulls_or_commits defining the coverage timeline for each code.
- **plot_xlim** – Tuple of datetime objects defining the time-range of the x-axis. E.g., plot_xlim=(datetime.strptime("2021-01-01", "%Y-%m-%d"), datetime.today())
- **fill_alpha** – Alpha value to be used for the area plots. Set to 0 or less to disable area plots (default=0.2)
- **fontsize** – Fontsize to be used for axes label, tickmarks, and titles. (default=16)
- **title** – Optional title for the figure
- **figsize** – Optional tuple of ints with the figure size

Returns

Matplotlib figure created here

```
class nwb_project_analytics.renderstats.RenderCommitStats
```

Bases: `object`

Helper class for rendering commit history for repos

```
COLOR_ADDITIONS = 'darkgreen'
```

```
COLOR_DELETIONS = 'darkred'
```

```
static plot_commit_additions_and_deletions(commits: DataFrame, repo_name: str | None = None,  
                                           xaxis_dates: bool = False, bar_width: float = 0.8,  
                                           color_additions: str = 'darkgreen', color_deletions:  
                                           str = 'darkred', xticks_rotate: int = 90)
```

Plot the number of additions and deletions for commits as a bar plot

Parameters

- **commits** – Pandas DataFrame with the commits generated via GitRepo.get_commits_as_dataframe
- **repo_name** – Name of the Git repository

- **xaxis_dates** – Place bars by date (True) or equally spaced by order of commits (False)
- **bar_width** – Width of the bars. When plotting with xaxis_dates=True using a narrow bar width can help avoid overlap between bars
- **color_additions** – Color to be used for additions
- **color_deletions** – Color to be used for deletions
- **xticks_rotate** – Degrees to rotate x axis labels

Returns

Tuple with the Matplotlib figure and axis used

```
static plot_commit_additions_and_deletions_summary(commits: dict, bar_width: float = 0.8,
                                                    color_additions='darkgreen',
                                                    color_deletions='darkred', xticks_rotate:
                                                    int = 45, start_date: datetime | None =
                                                    None, end_date: datetime | None = None)
```

Plot bar chart with total additions and deletions for a collection of repositories

Parameters

- **commits** – Dict where the keys are the nwb_project_analytics.gitstats.GitRepo objects (or the string name of the repo) and the values are pandas DataFrames with the commits generated via GitRepo.get_commits_as_dataframe
- **bar_width** – Width of the bars
- **color_additions** – Color to be used for additions
- **color_deletions** – Color to be used for deletions
- **xticks_rotate** – Degrees to rotate x axis labels
- **start_date** – Optional start date to be rendered in the title
- **end_date** – Optional end data to be rendered in the title

```
static plot_commit_cumulative_additions_and_deletions(commits: DataFrame, repo_name: str |
                                                         None = None,
                                                         color_additions='darkgreen',
                                                         color_deletions='darkred')
```

Plot the cumulative number of additions and deletions for commits as a stacked area plot

Parameters

- **commits** – Pandas DataFrame with the commits generated via GitRepo.get_commits_as_dataframe()
- **repo_name** – Name of the Git repository
- **color_additions** – Color to be used for additions
- **color_deletions** – Color to be used for deletions

Returns

Tuple with the Matplotlib figure and axis used

```
class nwb_project_analytics.renderstats.RenderReleaseTimeline
```

Bases: `object`

Helper class for rendering GitHubRepoInfo release timelines

```
classmethod plot_multiple_release_timelines(release_timelines: dict, add_releases: dict | None =
                                             None, date_range: tuple | None = None,
                                             month_intervals: int = 2, fontsize: int = 16, title:
                                             str | None = None)
```

Plot multiple aligned timelines of the releases of a collection of GitHubRepoInfo repo objects

Parameters

- **release_timelines** – Dict where the keys are the repo names and the values are tuples with the 1) name of the versions and 2) dates of the versions
- **add_releases** (*Dict where the keys are a subset of the keys of the github_repos dict and the values are lists of tuples with "name: str" and "date: datetime.strptime(d[0:10], "%Y-%m-%d")" of additional releases for the given repo. Usually this is set to NWBGitInfo.MISSING_RELEASE_TAGS*) – Sometimes libraries did not use git tags to mark releases. With this we can add additional releases that are missing from the git tags. If None this is set to NWBGitInfo.MISSING_RELEASE_TAGS by default. Set to empty dict if no additional releases should be added.
- **date_range** – Tuple of datetime objects with the start and stop time to use along the x axis for rendering. If date_range is None it is automatically set to (NWBGitInfo.NWB2_BETA_RELEASE - timedelta(days=60), datetime.today())
- **month_intervals** – Integer with spacing of month along the x axis. (Default=2)
- **fontsize** – Fontsize to use in the plots

Returns

Tuple of matplotlib figure object and list of axes object used for rendering

```
static plot_release_timeline(repo_name: str, dates: list, versions: list, figsize: tuple | None = None,
                             fontsize: int = 14, month_intervals: int = 3, xlim: tuple | None = None,
                             ax=None, title_on_yaxis: bool = False, add_releases: list | None =
                             None)
```

Plot a timeline of the releases for a single GitHubRepoInfo repo

Based on https://matplotlib.org/stable/gallery/lines_bars_and_markers/timeline.html

Parameters

- **repo_info** – The GitHubRepoInfo object to plot a release timeline for
- **figsize** – Tuple with the figure size if a new figure is to be created, i.e., if ax is None
- **fontsize** – Fontsize to use for labels (default=14)
- **month_intervals** – Integer indicating the step size in number of month for the y axis
- **xlim** – Optional tuple of datetime objects with the start and end-date for the x axis
- **ax** – Matplotlib axis object to be used for plotting
- **title_on_yaxis** – Show plot title as name of the y-axis (True) or as the main title (False) (default=False)
- **add_releases** (*List of tuples with "name: str" and "date: datetime.strptime(d[0:10], "%Y-%m-%d")"*) – Sometimes libraries did not use git tags to mark releases. With this we can add additional releases that are missing from the git tags.

Returns

Matplotlib axis object used for plotting

11.4.2 Module contents

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

n

`nwb_project_analytics`, [151](#)
`nwb_project_analytics.codecovstats`, [133](#)
`nwb_project_analytics.codestats`, [133](#)
`nwb_project_analytics.create_codestat_pages`,
 [137](#)
`nwb_project_analytics.gitstats`, [138](#)
`nwb_project_analytics.renderstats`, [145](#)

INDEX

A

`axes` (`nwb_project_analytics.renderstats.PatchedMPLPlot` attribute), 145

`create_toolstat_page()` (in module `nwb_project_analytics.create_codestat_pages`), 138

C

`cached()` (`nwb_project_analytics.codestats.GitCodeStats` static method), 134

`clean_outdirs()` (`nwb_project_analytics.codestats.GitCodeStats` static method), 134

`clone_repos()` (`nwb_project_analytics.codestats.GitCodeStats` static method), 134

`CodecovInfo` (class in `nwb_project_analytics.codecovstats`), 133

`collect_all_release_names_and_date()` (`nwb_project_analytics.gitstats.GitHubRepoInfo` static method), 138

`color` (`nwb_project_analytics.gitstats.IssueLabel` attribute), 140

`COLOR_ADDITIONS` (`nwb_project_analytics.renderstats.RenderCommits` attribute), 148

`COLOR_DELETIONS` (`nwb_project_analytics.renderstats.RenderCommits` attribute), 148

`colors` (`nwb_project_analytics.gitstats.IssueLabels` property), 141

`compute_code_stats()` (`nwb_project_analytics.codestats.GitCodeStats` method), 135

`compute_issue_time_of_first_response()` (`nwb_project_analytics.gitstats.GitRepo` static method), 139

`compute_language_stats()` (`nwb_project_analytics.codestats.GitCodeStats` method), 135

`compute_summary_stats()` (`nwb_project_analytics.codestats.GitCodeStats` method), 135

`CORE_API_REPOS` (`nwb_project_analytics.gitstats.NWBGitInfo` property), 141

`CORE_DEVELOPERS` (`nwb_project_analytics.gitstats.NWBGitInfo` attribute), 141

`create_codestat_pages()` (in module `nwb_project_analytics.create_codestat_pages`),

D

`description` (`nwb_project_analytics.gitstats.IssueLabel` attribute), 140

`docs` (`nwb_project_analytics.gitstats.GitRepo` attribute), 139

F

`from_cache()` (`nwb_project_analytics.codestats.GitCodeStats` static method), 135

`from_nwb()` (`nwb_project_analytics.codestats.GitCodeStats` static method), 135

G

`get_by_type()` (`nwb_project_analytics.gitstats.IssueLabels` method), 141

`get_commits_as_dataframe()` (`nwb_project_analytics.gitstats.GitRepo` method), 139

`get_contributors()` (`nwb_project_analytics.codestats.GitCodeStats` static method), 136

`get_info_objects()` (`nwb_project_analytics.gitstats.GitRepos` method), 140

`get_issues_as_dataframe()` (`nwb_project_analytics.gitstats.GitRepo` method), 139

`get_languages_used()` (`nwb_project_analytics.codestats.GitCodeStats` method), 136

`get_pulls_or_commits()` (`nwb_project_analytics.codecovstats.CodecovInfo` static method), 133

`get_release_names_and_dates()` (`nwb_project_analytics.gitstats.GitHubRepoInfo` method), 138

`get_releases()` (`nwb_project_analytics.gitstats.GitHubRepoInfo` method), 139

`get_time_and_coverage()` (nwb_project_analytics.codecovstats.CodecovInfo static method), 133
`get_version_jump_from_tags()` (nwb_project_analytics.gitstats.GitHubRepoInfo static method), 139
`git_repo_stats()` (nwb_project_analytics.codestats.GitCodeStats static method), 137
`GIT_REPOS` (nwb_project_analytics.gitstats.NWBGitInfo attribute), 142
`GitCodeStats` (class in nwb_project_analytics.codestats), 133
`github_issues_url` (nwb_project_analytics.gitstats.GitRepo property), 140
`github_path` (nwb_project_analytics.gitstats.GitRepo property), 140
`github_pulls_url` (nwb_project_analytics.gitstats.GitRepo property), 140
`GitHubRepoInfo` (class in nwb_project_analytics.gitstats), 138
`GitRepo` (class in nwb_project_analytics.gitstats), 139
`GitRepos` (class in nwb_project_analytics.gitstats), 140
H
`HDMF_START_DATE` (nwb_project_analytics.gitstats.NWBGitInfo attribute), 144
I
`init_codestat_pages_dir()` (in module nwb_project_analytics.create_codestat_pages), 138
`IssueLabel` (class in nwb_project_analytics.gitstats), 140
`IssueLabels` (class in nwb_project_analytics.gitstats), 141
L
`label` (nwb_project_analytics.gitstats.IssueLabel attribute), 140
`level` (nwb_project_analytics.gitstats.IssueLabel property), 141
`levels` (nwb_project_analytics.gitstats.IssueLabels property), 141
`logo` (nwb_project_analytics.gitstats.GitRepo attribute), 140
M
`mainbranch` (nwb_project_analytics.gitstats.GitRepo attribute), 140
`merge()` (nwb_project_analytics.gitstats.GitRepos static method), 140
`merge()` (nwb_project_analytics.gitstats.IssueLabels static method), 141
`merge_contributors()` (nwb_project_analytics.codestats.GitCodeStats static method), 137
`MISSING_RELEASE_TAGS` (nwb_project_analytics.gitstats.NWBGitInfo attribute), 144
`nwb_project_analytics` module, 151
`nwb_project_analytics.codecovstats` module, 133
`nwb_project_analytics.codestats` module, 133
`nwb_project_analytics.create_codestat_pages` module, 137
`nwb_project_analytics.gitstats` module, 138
`nwb_project_analytics.renderstats` module, 145
`NWBGitInfo` (class in nwb_project_analytics.gitstats), 141
`NWB1_DEPRECATION_DATE` (nwb_project_analytics.gitstats.NWBGitInfo attribute), 144
`NWB1_GIT_REPOS` (nwb_project_analytics.gitstats.NWBGitInfo attribute), 144
`NWB2_BETA_RELEASE` (nwb_project_analytics.gitstats.NWBGitInfo attribute), 144
`NWB2_FIRST_STABLE_RELEASE` (nwb_project_analytics.gitstats.NWBGitInfo attribute), 144
`NWB2_START_DATE` (nwb_project_analytics.gitstats.NWBGitInfo attribute), 144
`NWB_EXTENSION_SMITHY_START_DATE` (nwb_project_analytics.gitstats.NWBGitInfo attribute), 144
`NWB_GUIDE_START_DATE` (nwb_project_analytics.gitstats.NWBGitInfo attribute), 144
`owner` (nwb_project_analytics.gitstats.GitRepo attribute), 140

