Pynapple: a toolbox for data analysis in neuroscience.

Guillaume Viejo [1], Daniel Levenstein [1,2], Sofia Skromne Carrasco [1], Dhruv Mehrotra [1], Sara Mahallati [1], Gilberto R Vite [1], Henry Denny [1], Lucas Sjulson [3], Francesco P Battaglia [4], Adrien Peyrache [1]

[1] Montreal Neurological Institute and Hospital, McGill University, Montreal, QC, Canada
[2] MILA – Quebec IA Institute
[3] Departments of Psychiatry and Neuroscience, Albert Einstein College of Medicine, Bronx, NY
[4] Donders Institute for Brain, Cognition and Behaviour, Radboud University, 6525AJ Nijmegen, The Netherlands

Correspondence: adrien.peyrache@mcgill.ca

Abstract

Datasets collected in neuroscientific studies are of ever-growing complexity, often combining high dimensional time series data from multiple data acquisition modalities. Handling and manipulating these various data streams in an adequate programming environment is crucial to ensure reliable analysis, and to facilitate sharing of reproducible analysis pipelines. Here, we present Pynapple, a lightweight python package designed to process a broad range of time-resolved data in systems neuroscience. The core feature of this package is a small number of versatile objects that support the manipulation of any data streams and task parameters. The package includes a set of methods to read common data formats and allows users to easily write their own. The resulting code is easy to read and write, avoids low-level data processing and other error-prone steps, and is fully open source. Libraries for higher-level analyses are developed within the Pynapple framework but are contained within in a collaborative repository of specialized and continuously updated analysis routines. This provides flexibility while ensuring long-term stability of the core package. In conclusion, Pynapple provides a common framework for data analysis in neuroscience.

Highlights:

- An open-source framework for data analysis in systems neuroscience.
- Easy-to-use object-oriented programming for data manipulation.
- A lightweight and standalone package ensuring long-term backward compatibility.

Introduction

The increasing size of datasets across scientific disciplines has led to the development of specific tools to store [1,2], analyze [3], and visualize [4] them. While various programming environments such as Matlab and R have long been commonly used in data science, Python has progressively become one of the most popular programming languages [5]. This is due to its open nature, large community-driven development, and versatility of usage. As with virtually all other scientific fields, neuroscience faced the challenges of handling and analyzing large datasets by rapidly developing a wide range of specialized tools to deal with each of these types of data [6–11] and corresponding analyses.

In systems neuroscience, calcium imaging and high-density electrophysiology make it possible to simultaneously monitor the activity of an increasingly large number of neurons [12,13]. Often, this is combined with simultaneous behavioral recordings. As in all other fields, this has required the development of specific pipelines to process [14–25] and store [26,27] the data. Despite this rapid progress, data analysis often relies on custom made, lab-specific code, which is susceptible to error and can be difficult to compare across research groups. While several toolboxes are available to perform neuronal data analysis [8–11,28] (see ref. [29] for review), most of these programs focus on producing high-level analysis from specified types of data, and do not offer the versatility required for rapidly-changing analytical methods and experimental methods.

The key challenge for scientific code is balancing the need for flexibility and stability. This is especially true of science because results should be reproducible (between labs, between the past and the future, and between different experimental setups) while keeping up with rapidly changing requirements (e.g., due to new kinds of data, theories, and analysis methods). To meet these needs, a general toolbox for data analysis must be designed with a few principles in mind.

The first property of such a toolbox is that it should be object-oriented, organizing software around data. This makes the programming environment very efficient for data analysis, particularly in systems neuroscience where data streams can be of very different types. For example, to compute the rate of an event, one can write a function that takes an array of event times and divides the number of elements by the time between the first to the last event. However, this approach neglects to consider that the appropriate epoch in which to calculate the rate could start earlier, or end later, than the first or last event. Addressing these concerns requires another argument, which defines the boundaries of the epoch on which the rate should be computed. Overall, this approach is error prone. The epoch boundaries and event times must be stored in the same time unit and with the same reference (i.e., simultaneous time 0) and the rate function itself can be erroneously called with arrays storing another type of data. In contrast, an object which is specifically designed to represent a series of event times can ameliorate these concerns. For example, it can be created from a specific data loader that ensures proper definition of time units and support epochs (i.e., true beginning and end of the observation time). It will then be immune to the arithmetic operations that can change the values of a generic array (for example an addition that is misplaced in the code). Further, the object can be endowed with a rate property that is specifically written for this object, drastically diminishing the odds of a coding error. While

this approach may discourage users who are not familiar with this type of coding, the benefit far exceeds the effort of learning object-oriented programming, especially if the naming of the methods and properties is explicit.

Another property of an efficient toolbox is that as much data as possible should be captured by only a small number of objects. This ensures that the same code can be used for various datasets and eliminates the need of adapting the structure of the package to handle rare or yet-to-be-developed data types. Then, these objects should then be able to interact via a small number of basic and foundational operations, which are sufficient for most analyses. This allows users to quickly write new code for new use-cases, and easily understand and adapt code written by others, as the same methods can be used for any kind of data.

The toolbox should be able to load common data storage types, and the flexibility to create loaders for future and custom/lab-specific data. It should also support the development of yet-unknown, lab-specific, and specialized analysis methods. In other words, the customization of the package to adapt to any dataset should happen at the input stage and the development of high-level analytical methods should take place outside the core package. The properties listed above ensure the long-term stability of a toolbox, a crucial aspect for maintaining the code repository. Toolboxes built around these principles will be maximally flexible and will have the most general application.

In this paper we introduce the Python Neural Analysis Package (Pynapple), designed with these axioms in mind. The core of Pynapple is five versatile timeseries objects, each with a limited number of methods. We show how Pynapple can be used with most raw neuroscience data types to produce the most common analyses used in contemporary neuroscience. Additionally, we introduce Pynacollada, a collaborative repository for higher level analyses built from the basic functionality provided by Pynapple. A complete neuroscience data analysis pipeline using a common language supports open and reproducible code and provides a foundation upon which novel analyses can be shared and collectively built by the neuroscience community.

Core features of Pynapple

At its core, Pynapple is object-oriented. This approach limits the chances of coding error while providing the user with a simple environment in which all streams of data are by construction expressed in the same time base. Pynapple is built around only five objects that are divided into three categories: two objects represent event timestamps (one or several), two represent time-varying data (one or several time series at the same sampling times), and one represents time epochs. Raw or pre-processed data are loaded into these objects in the coding environment (**Fig. 1**). The data loaders ensure that all loaded objects have the same time base. Hence, once objects are constructed, the user does not have to remember properties of the data such as the sampling frequency or alignment of data indices to clock time. Then, these objects can be manipulated with

their own methods (i.e., object-specific functions). A large majority of data manipulations needed for most users can be achieved with a small number of methods. From there, Pynapple offer some foundational analyses, such as cross-correlation of event times. On top of this, the user may write analytical code that is project specific.

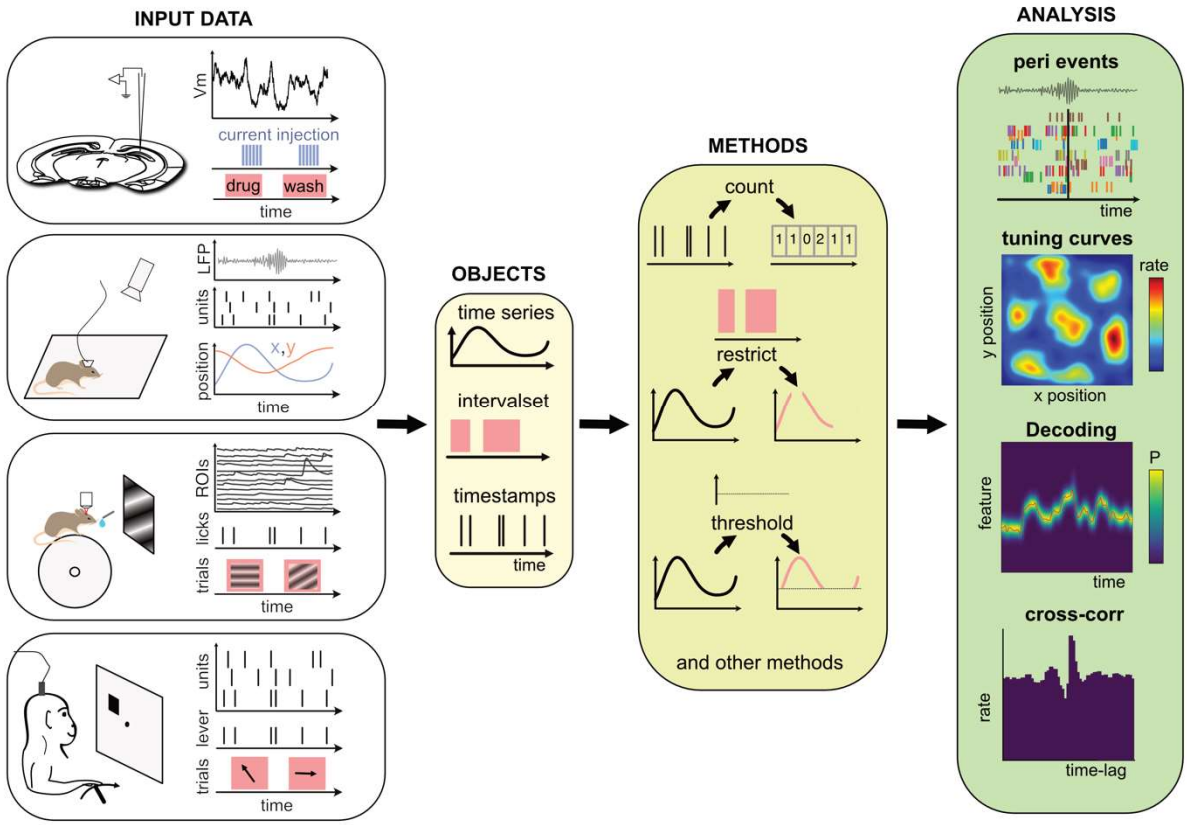

Fig. 1 | Data analysis with the Pynapple package.

Left, any type of input data can be loaded in a small number of core objects. For example (from top to bottom): intracellular recordings in slice during which current is injected and drug is applied to the bath solution; extracellular recordings in freely moving mice whose position is video-tracked; calcium imaging in head-fixed mice during presentation of different visual stimuli and delivery of precisely timed rewards; extracellular recordings in non-human primates during the execution of cognitive tasks.

Middle, object-specific methods allow the user to perform a wide variety of basic operations and to manipulate the data manipulations.

Right, at a higher level, the package contains a set of foundational analysis methods, which do not depend on any external package, such as (from top to bottom) peri-event alignment of the data (top), 1- and 2-D tuning curves, 1- and 2-D decoding; auto- and cross-correlation of event times (e.g., action potentials).

The most basic objects are timestamps (*Ts*), which are typically used for any discrete events, for example spike or lick times. The timestamped data (*Tsd*) object holds timestamps and associated data associated with each timestamp. For example, this object is used to represent an animal's position in its environment, electroencephalogram data, or average calcium fluorescence as a function of time. Two objects were designed to represent arrays of *Ts* and *Tsd*, namely *TsGroup* and *TsdFrame*. The main difference between the two objects is that *TsdFrame* has common timestamps for all the data (and therefore, all data have the same number of samples). *TsGroup* is more generic as each element has its own timestamps. These objects are typically used for ensembles of simultaneously recorded spike trains (*TsGroup*) or simultaneously acquired calcium fluorescence (*TsdFrame*). They are useful when operations need to be performed on a common time basis, for example binning multiple spike trains. Note however that they can be used for many other data types, for example the position of the animal (*TsdFrame*). Last, *IntervalSet* objects represent time epochs, for example the start and end times of intervals in which the animal is running.

Pynapple is built with objects from the Pandas library [5]. As such, Pynapple objects inherit the computational stability and flexibility from this widely used package. Specifically, a *Tsd* object is an extension of (or "inherits" in object-orienting programming) Pandas *Series* object and *TsdFrame* of Pandas *DataFrame* object. A *TsGroup* is a child of *UserDict*, a built-in python object for inheriting dictionaries. Finally, *IntervalSet* inherits Pandas *DataFrame.* Timestamps are by default in units of seconds but can be readily converted to other time units using the `as_units` method in any object.

Pynapple objects have a limited number of core methods (**Fig. 2A**), which form the foundation of further operations. These operations provide a general framework by which users can manipulate the timestamps and their corresponding values as needed for analysis. For example, the time series objects have built-in methods: *`value_from`*, which gets the value from one time series object at the (closest) timestamps from another; *`restrict`*, which "restricts" a time series object; extracting only the data contained within a set of time intervals defined by an *IntervalSet* object; *`count`*, which counts the number of timestamps from a time series object in windows of a given bin size; *`threshold`*, which applies a threshold to the data within a *Ts* or *Tsd* object and returns a *Tsd* containing the data above or below the threshold. All operations can be restricted to a given epoch, specified by an *IntervalSet*.

Furthermore, all objects have a `time_support` property, which keeps track of the time interval over which the data is valid. The time support is an *IntervalSet* object that is attached by default to *Ts*, *Tsd*, *TsdFrame* and *TsGroup* objects. This is a crucial property as, otherwise, it is impossible to know whether periods without data correspond to an epoch during which the underlying event was not observed or because this period has previously been excluded by a `restrict` method.

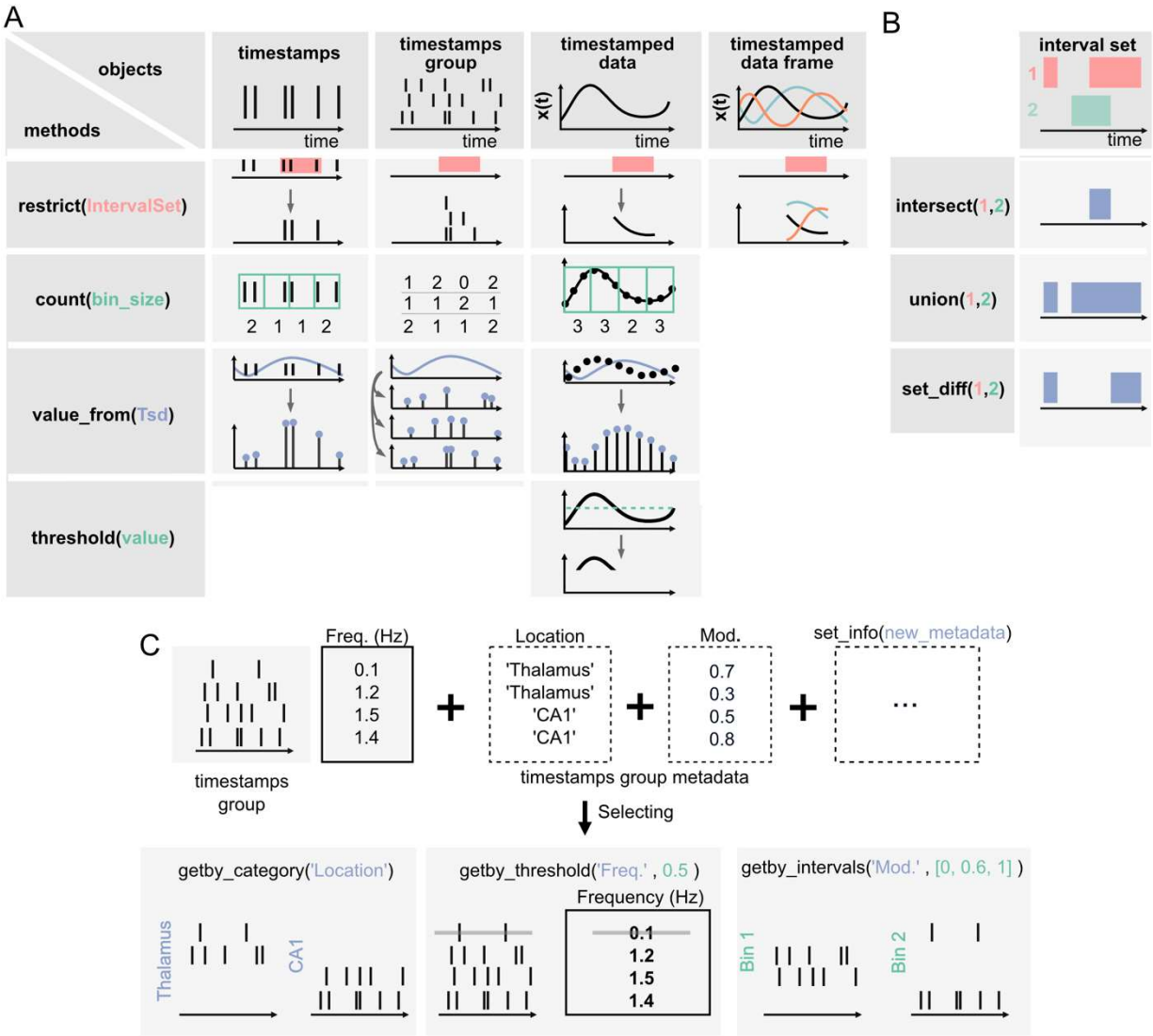

Fig. 2 | Core methods of the Pynapple objects.

A) Methods of Timestamps (*Ts)* and Timestamped Data (*Tsd)* objects. The same methods can be called for different objects, leading to qualitatively similar results. For example, `object.restrict(intervalset)` returns an object now defined on the intersection of its original time support and the input *IntervalSet*. Objects can be any of the timestamps and timestamped data objects. These methods can be called with only one argument, as shown here, since the default parameters are typically the same for most analyses. Yet the methods include additional arguments for more specific operations.

B) Logical operations on pairs of *IntervalSet* objects to compute (from top to bottom) the intersection, union, and difference between epochs. These operations are commonly used to analyze data during specific epochs in a combinatorial manner, such as “exploration period AND running speed is above 5cm/s NOT left arm”.

C) Methods of *TsGroup* objects. Each timestamp is associated by default with its occurrence

rate. Additional custom metadata such as recording location can be added. These metadata can then be used to select and filter timestamps using `getby_category` for discrete labels, `getby_threshold` or `getby_intervals` for numerical values.

In addition to the ability to restrict methods of time series objects, the *IntervalSet* object has methods for logical operations on combinations of *IntervalSets*, all returning other *IntervalSets* (**Fig. 2B**): *`intersect`*, which returns the set intersection of two *IntervalSet* objects; *`union`*, which returns the set union of two *IntervalSets*; *`set_diff`*, which returns the set difference of two IntervalSet; *`drop_short_intervals`*, *`drop_long_intervals`*, which eliminate interval subsets that are shorter or longer than a desired duration; and *`merge_close_intervals`*, which merge intervals that are closer in time than a given duration.

Many experiments in neuroscience are based on trials, each associated with different conditions. *IntervalSets* are perfectly suited for this, as one *IntervalSet* can represent all start and end times of trials. The nature of each trial (e.g., left/right, correct/error) can be stored as a third column within the *IntervalSet* dataframe object. Thus, subsets of trials can be easily selected to restrict data of interest on the corresponding epochs. An alternative approach is to store different *IntervalSets* for different types of trials.

In addition to the ability to apply any methods of the *Ts* object to its members, *TsGroup* has a set of methods to calculate and store metadata about the elements of the group (**Fig. 2c**). For example, one can store and retrieve the anatomical structure from which a neuron was recorded, or the result from downstream analysis, perform operations on each element, and filter by various properties. These methods allow the user to, for example, calculate, store, and compare the properties of multiple neurons in a population. Additional methods for all objects are extensively documented in the documentation, and examples for usage are given in the tutorials.

While these objects and methods are relatively few, they are the foundation of almost any analysis in systems neuroscience. However, if not implemented correctly, they can be both computationally intensive and highly susceptible to user error. The implementation of core features in Pynapple addresses this concern. Crucially, all units are indexed by seconds across the entire package, which limits the need for users to account for indexing and alignment between different streams of data at different sampling rates. For example, a user can simply use `spikes.value_from(position)` to get the animal's position at each spike time, rather than costly and error-prone routines in which a user needs to identify matching indices for the corresponding timestamps across arrays containing spikes and behavioral information. Another common issue in data analysis is to analyze two timeseries that are not recorded at the same sampling rate. Once data are loaded in the same time base (i.e., the same time 0), they can keep their original sampling times. Using the function `value_from` from one object with the other object as argument will provide two time series with the same number of samples and the same

sampling times, which will simplify further analyses. However, this means it is essential that all objects are loaded in the same time base for these methods to function correctly. Pynapple anticipates this by providing a customizable data loader, ensuring time bases are always loaded correctly.

Importing data from common and custom pipelines

The proliferation of experimental methods has come with a proliferation of data formats, as well as the need to rapidly develop new formats that meet new experimental needs. Usually, these data formats are dependent on the software that was used to preprocess the raw data, making them difficult to load for further analysis. Additionally, an experimental setup can generate multiple streams of data that are saved within multiple files of various types. Thus, a universal toolbox should be able to load popular data formats into a common framework and offer the user the ability to write functions to load their own data types.

To ease the process of loading and synchronizing data from various streams, Pynapple includes an I/O layer that allows the user to load multiple types of datasets and write them to a common format for further analysis and sharing. The primary way by which a user interacts with the I/O layer is an object that represents an experimental session, with the properties of the object being the various time series. This I/O object is created by calling the function `load_session`, which will load all data associated with that session (**Fig. 3A**). For example, calling `load_session` for an *in vivo* electrophysiology recording would return an object called `data`, which will have properties `data.spikes`, `data.position`, and `data.epochs` which respectively store a *TsGroup,* containing the spike times, a *TsdFrame* containing the position of the animal, and an *IntervalSet* containing the times when the animal is on the track. With this object-oriented I/O method, the user can interact with the various data streams associated with a given experimental session and load multiple sessions at once without the risk of mixing data as each time series is attached to only one I/O object.

Data synchronization is the crux of any analysis pipeline. The `load_session` function is thus a crucial step in using the package. For unsupported data types, it is the responsibility of the users to design the preprocessing scripts that align the data streams in the same absolute time base. The data loading and synchronizing functions already included in the package for supported data types is a good starting point for any user writing a custom loading function (details this process are provided later).

While data types are usually specific to a recording modality (i.e., calcium imaging and electrophysiology), there are several pieces of metadata that are common to many experiments, such as the strain of the animal, age, sex, and name of the experimenter. When loading a session for the first time, the I/O process starts with a graphical user interface (GUI) in which the user can

quickly and easily input the general information as well as any session epoch and behavioral tracking data (Fig. 3b). This information is saved in a *BaseLoader* class.

General session information is common across experimental sessions, however specialized data streams are usually specific to recording modalities. To cover the variety of preprocessing analysis pipelines currently used in systems neuroscience, the Pynapple I/O can load data formats from popular preprocessing pipelines (e.g., CNMF-E, Phy, NeuroSuite or Suite2P). This is implemented via a set of specialized object subclasses of the *BaseLoader* class. To avoid code redundancy, these I/O classes inherit the properties of the *BaseLoader* class. This is a core aspect of object-oriented programing, and it means that these specialized I/O classes have all the methods and properties of the parent *BaseLoader* objects. This ensures compatibility across various loading functions. However, once generated, these specialized I/O classes are unique and independent from each other, ensuring stability. Like the *BaseLoader* class, a specialized GUI for electrophysiology and calcium imaging is provided, with relevant metadata fields, for example electrode position in electrophysiology and type of fluorescence indicator in calcium imaging (**Fig. 3B**).

To avoid repeating the process of inputting session information and synchronization of multiple data streams, Pynapple saves all synchronized data into a unique file and can accommodate a wide range of neuroscientific data types. Recently, Neurodata Without Borders (NWB) [26,27] has emerged as a flexible data format used for public data sharing and large databases such as those collected by the Allen Institute. Thus, we chose to use the NWB format for fast and universal data loading and saving with Pynapple. The *BaseLoader* is responsible for initializing the NWB file within the session folder (i.e. it creates a new NWB file if none is present) (**Fig. 3C**). Converting user's data to NWB format encourages standardization and can facilitate sharing both data and analysis pipelines written with Pynapple.

Many other preprocessing pipelines exist and can often be unique to a lab or even to an individual project. To accommodate present and future needs for these specific pipelines, the documentation of Pynapple provides an easy-to-follow recipe for creating a custom I/O class that inherits the *BaseLoader* and can interact with a pre-existing NWB file. There are multiple benefits of the inheritance approach of data loading classes within the I/O layer of Pynapple. First, future development of new I/O classes will not affect the core and processing layers of Pynapple. This ensures long-term stability of the package. Second, users can develop their own custom I/O using chain inheritance of pre-existing classes. Using preexisting code to load data in a specific manner instead of rewriting already existing functions avoids preprocessing errors. Third, users can still use Pynapple without using the I/O layer of Pynapple. Last, in order to apply previous analyses, or analyses developed in another lab, to new data or data types a user only needs to develop a new I/O class for their data. This will import the data to the common Pynapple core from which the same analysis pipeline can be used.

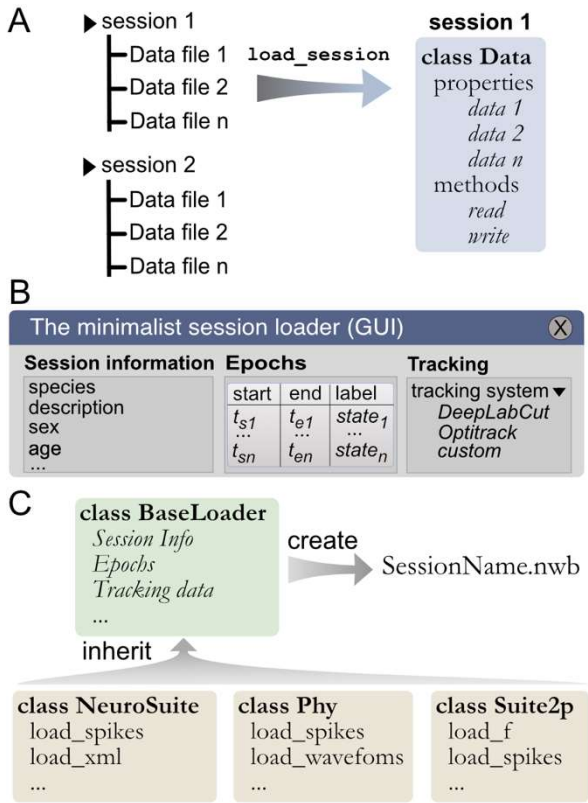

Fig. 3 | Built-in and customizable loading function for Pynapple.

A) Data is originally organized as separate files in a folder. A built-in or custom-made load_session function is called to load the data into a Data class.

B) Data can be loaded through a customizable GUI to enter all relevant information regarding the experiment, for example animal strain, among others. The main epochs of the recording (e.g., behavioral states, stimuli category, etc.) can be loaded from standard tabular data files (such as CSV). Behavioral tracking data extracted from various common systems and saved as a CSV file can also be loaded.

C) Pynapple offers various built-in loaders for commonly used data formats, as well as a template to easily design a customizable loader to adapt to any other format or specific task design.

Foundational data processing

The basic methods that manipulate the core objects in Pynapple allow users to perform common, but powerful, neuroscience analyses (**Fig. 2**). These analyses are powerful because they are able to describe the relationships between time series objects, while requiring the fewest number of parameters to be set by the user. This minimizes complexity, while maximizing generalizability. The operations in Pynapple can recreate neuroscience analyses from a broad number of subdisciplines. These analyses form the foundation of neuroscience data analysis in Pynapple. To illustrate the versatility of Pynapple and how it can be used, we reanalyzed five openly available datasets.

The first foundational analysis is computing neural tuning curves. Tuning curves relate specific stimuli to the firing rate of neurons. To this end, Pynapple computes the firing rate of a neuron (or any other timestamped data) during each epoch in an *IntervalSet* object, for example for discrete conditions such as “ON/OFF" stimuli. Tuning curves can also be computed with respect to a continuous feature. Once computed, Pynapple is able to use tuning curves from a population of neurons to decode stimuli using a Bayesian decoder [30,31] (**Fig. 4A**).

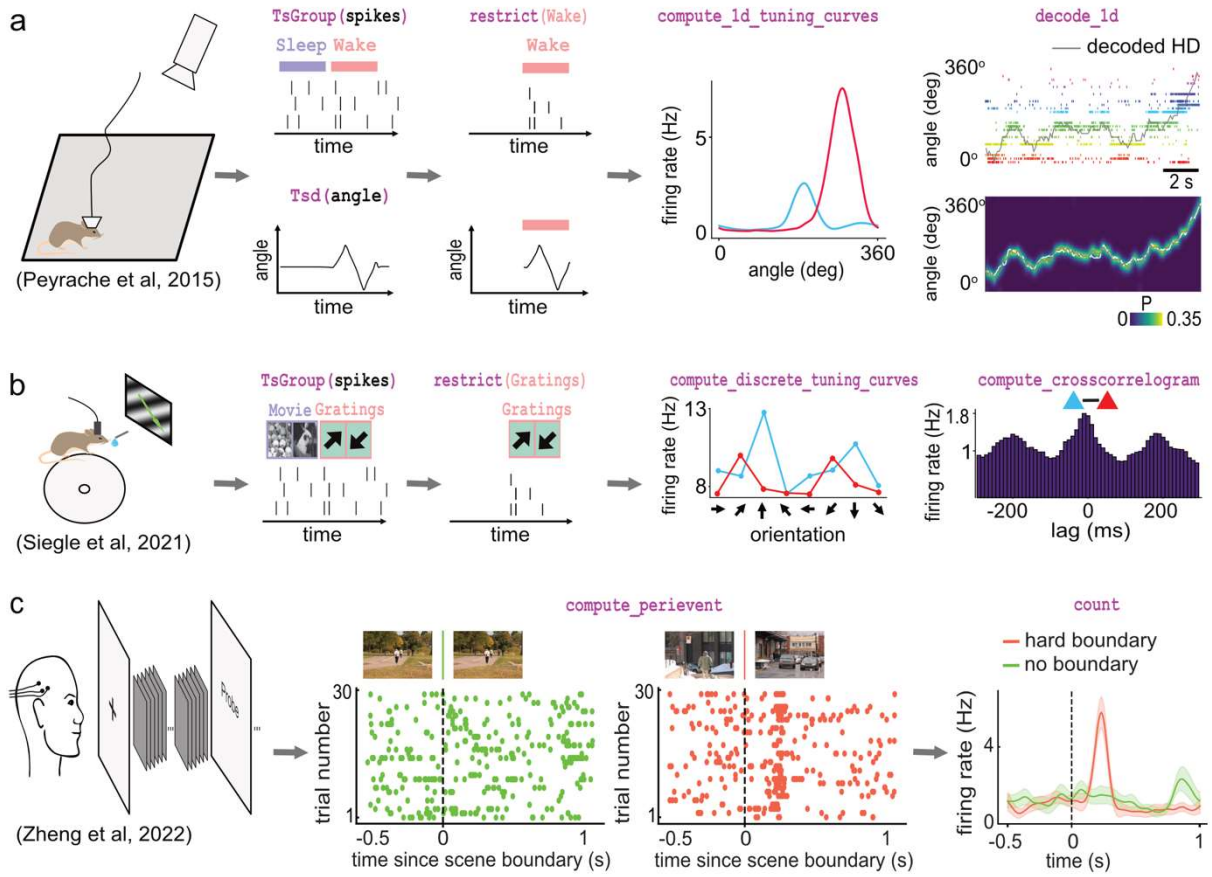

Fig. 4 | Examples of foundational analysis across various electrophysiological datasets using Pynapple.

A) Analysis of an ensemble of head-direction cells. From left to right: data were collected in a freely moving mouse randomly foraging for food; all data are restricted to the wake epoch (i.e., during exploration); the tuning curve of two neurons relative to the animal's head-direction; animal's head-direction is decoded from the neuronal ensemble. Data from refs. [32,33]

B) Analysis of V1 neurons during visual stimulation. From left to right: the mouse was recorded while being head-fixed and presented with drifting gratings; spikes, stimulation and epochs are shown; example tuning curves of two V1 neurons, showing their firing rates for different grating orientations; example cross-correlation between two V1 neurons, showing an oscillatory co-modulation at about 5Hz during visual stimulation. Data from ref. [34]

C) Analysis of medial temporal lobe neurons in human epileptic subjects. From left to right: subjects, implanted with hybrid deep electrodes, were shown a series of short clips; raster plot of a single neuron around continuous movie shot trials (green) and hard boundary trials, which are transitions between two unrelated movies (orange); peri-event neuronal firing rate for both trial types. Data from ref. [35]
Images in panels b and c are from ref. [36]

The second foundational analysis is computing auto- and cross-correlograms of event data. In the most abstract sense, these correlograms show the relationship between previous and future events and a current event at time 0. In Pynapple, cross-correlograms can be generated for any two series of events by computing the event rate for each time bin of a target time series relative to each event of a reference time series. Commonly, this is used to examine the likelihood of an action potential in a neuron relating to a previous or future action potential in the same neuron (auto-correlogram) or in another neuron (cross-correlogram) (**Fig. 4B**). However, Pynapple does not limit this function to spiking data and correlograms may be performed on any event-based data.

The third and final foundational analysis is peri-event alignment. This involves aligning a specified window from *Ts*/*Tsd*/*TsGroup* data to a specific *Ts*, known as “TimeStamp Reference". This allows users to align data to specific points in time, and measure changes in rates around this specified time point (**Fig. 4C**). One example where this function is useful is aligning neuronal spikes to specific stimuli, such as optogenetic illumination, presentation of a tone, or electrical stimulation.

Some of the analyses presented so far are designed for spikes (and discrete events in general) and cannot be applied for continuous traces such as calcium imaging data. Pynapple includes specialized functions that can compute the tuning of a continuous value with respect to a feature, as shown for the modulation of fluorescence in calcium imaging with respect to the speed of the animal (**Fig. 5A**) or of the position of a vertical bar on a screen in the fly’s ellipsoid body (**Fig. 5B**).

The examples shown in Figures 4 and 5 show how these core analyses are useful for rapid data screening with just a few lines of code in a Jupyter notebook, for example. Overall, these foundational functions form the building blocks of most other analyses in systems neuroscience. Importantly, they are built-in and thus do not have any external dependencies, ensuring that they can be used stand-alone without relying on any third-party software. All other developments of analysis pipelines take place outside Pynapple, ensuring the core package is only updated rarely and remains lightweight.

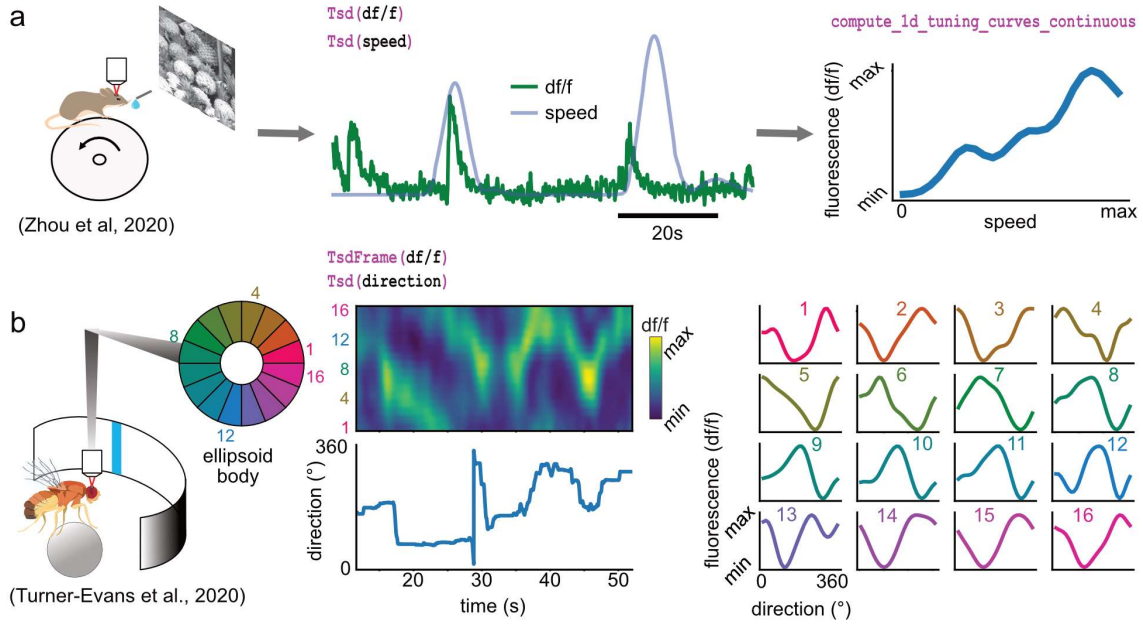

Fig. 5 | Examples of foundational analysis across various calcium imaging datasets using Pynapple.

A) Analysis of a V1 neuron during visual stimulation. From left to right: the mouse was recorded while being head-fixed on a running wheel and presented with natural scene movies; fluorescence traces from a pre-processed region of interest and running speed are loaded; continuous tuning curve is directly obtained from fluorescence and speed. Data from ref. [37]. Image is from ref. [36]

B) Analysis of neuronal activity in the fly central complex. From left to right: a *Drosophila melanogaster* is tethered to a calcium imaging setup while the position of a vertical bar is in closed loop with the fly's movements on a ball; calcium activity in the ellipsoid body is divided into 16 wedges; example fluorescence trace and direction of the fly. Tuning curves are obtained as in **A**, with the direction as feature. Data from ref. [38]

Pynacollada: a collaborative library for specialized and continuously updated data analyses.

Pynapple is designed to be stable in the foreseeable future and its core functionality is not meant to be modified. However, actual data analysis usually requires more than the available core functions. This type of data analysis is "fluid", constantly updated by new software developments and theoretical work. Furthermore, this kind of development is collaborative in nature and the supervision of such projects is less sensitive than that of a stable package. To balance the needs for stability and flexibility, high-level functions were separated from Pynapple and included instead in Pynacollada: the Pynapple Collaborative repository hosted on GitHub.

Complex analyses are added to Pynacollada in the form of libraries. Each library developed for Pynacollada takes the form of a Jupyter notebook (or python scripts) which guides the user through the analysis step-by-step. As such, libraries built for Pynacollada should provide training, promote good practice in programming, and allow users to easily adapt code to their own project. Examples of complex analyses currently handled by Pynacollada are outlined below (**Fig. 6**).

Recent advances in the application of manifold theory to neural data analysis have allowed neuroscientists to project high-dimensional data into three or fewer dimensions [39–41]. The structure of these projections reflects the structure of these higher dimensional processes, allowing us to infer the information encoded by the population. The Pynacollada "neural_manifold" library contains a Jupyter notebook that provides a step-by-step process for recreating a ring manifold using spiking data recorded from a population of head-direction neurons (**Fig. 6**). This code can be adapted by the end-user for analysis of their own data by simply importing their own data and refactoring the parameters to suit their needs.

A second complex analysis handled by Pynacollada is sharp wave-ripple (SWR) detection. Detecting oscillatory events is a routine procedure in electrophysiology, yet usually depends on many arbitrary choices of parameters. In this case, the Jupyter notebook showcases an example of detecting SWRs, a well-characterized oscillation of the hippocampus (**Fig. 6**).

In addition, Pynacollada currently includes libraries for spike waveform processing, EEG analysis, and video tracking, among others. We invite the community to contribute to this repository by improving current libraries or upload new ones. For new libraries, only rapid screening and tests will be performed, but the code will not go through the kind of validation that is in place for Pynapple as an external library will never affect the functioning of the core package. The documentation describes what is expected in each library to simplify readability, sharing, and maintenance and, overall, how libraries should conform to Pynacollada standards. We hope this will be broadly adopted by the community, allowing researchers across labs to easily share their code.

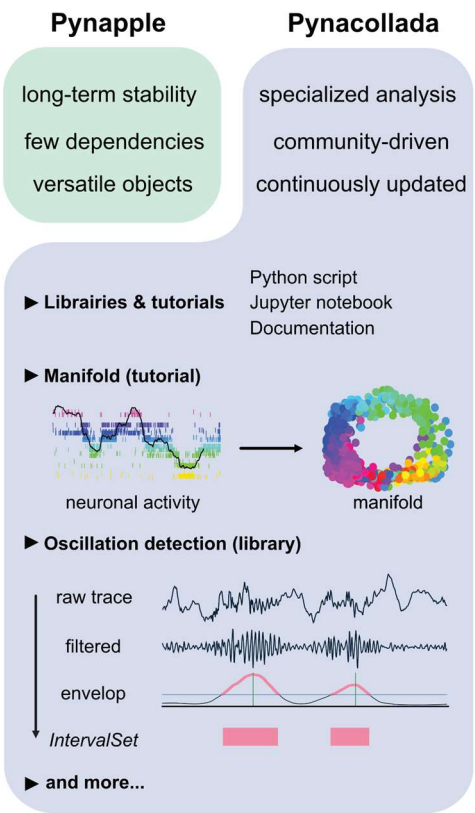

Fig. 6 | The Pynapple collaborative data analysis repository (Pynacollada) environment.

Unlike Pynapple, which is designed for long-term stability, Pynacollada is a repository of project-oriented libraries. This way, the community can collaborate on constantly evolving data analysis code without affecting the functionality of the core pipeline. Each project should include a script that can be called for specific functions and/or Jupyter notebooks to showcase the use of the code, as well as proper documentation. Pynacollada already includes several libraries and/or tutorials, including but not limited to: (1) a tutorial on manifold analysis, covering how to project neuronal data on low-dimensional subspace using various machine learning techniques; (2) a library for oscillation detection in local field potentials, which takes raw broadband traces as inputs and outputs *IntervalSet* objects corresponding to the start and end times of oscillation bouts.

Discussion

Here we introduced Pynapple, a lightweight and open-source python toolbox for neural data analysis. The goal of this package is to offer a versatile set of tools to study typical neurophysiological and behavioral data, specifically time series (e.g., spike times, behavioral events, and continuous time series) and time intervals (e.g., trials and brain states). It also provides users with generic functions for neuroscience analyses such as tuning curves and cross-correlograms. Finally, Pynapple was designed to rely on a minimum number of dependencies, which are themselves very common and thus highly stable. As such, accessibility is the guiding axiom of Pynapple.

The design of Pynapple is centered around the manipulation of simple, abstract objects that are common to most neurophysiological and behavioral datasets. The core of Pynapple is built around five objects: Timestamps (*Ts*) and group of Timestamps (*TsGroup*), Time Series Data (*Tsd*) and ensemble of co-registered Tsd (*TsdFrame*), as well as *IntervalSets*. These objects can be manipulated with properties that are, in most cases, common to all objects. Building around these fundamental objects and properties means Pynapple is highly flexible and able to handle most neurophysiological and behavioral datasets, making it accessible to most systems neuroscientists.

To simplify analysis for users, Pynapple offers simple wrappers for loading data with popular preprocessing pipelines. However, to be fully accessible, it is not sufficient for a package's core operations to be able to process all data types in theory. Data produced in neuroscience has a wide variety of file types, which are often only loaded by specific analysis software. Data is also largely experiment-specific. To unify these disparate file types and configurations, Pynapple's data loader is customizable. In addition to being able to load current popular data formats, this customizable data-loader means emerging file formats may continue to be loaded in the future, without significant overhauls to the main package. This offers Pynapple long-term stability and means that Pynapple will continue to remain accessible in the foreseeable future.

In further pursuit of accessibility, from these simple objects and properties, Pynapple has several built-in, foundational analyses that are common across the field of systems neuroscience. These foundational analyses include computing neural tuning curves, computing auto-/cross-correlograms, peri-event alignment, and performing Bayesian decoding. From these foundational analyses, higher order analyses can be developed. However, these higher order analyses are more prone to customization, thereby making them relatively more flexible. As such, higher order analyses are stored in the collaborative repository known as Pynacollada. This keeps the core Pynapple package stable, while allowing the user to integrate new advances in neurophysiological and behavioral analysis into their workflow.

Other software provide programming environments which deal with common neuroscientific data and an interface between stored data and analytical methods [10]. However, one problem that arises from this structure is that objects and data structures are rigidly defined, leading to a lack of versatility for new types of data or task design. In contrast, Pynapple offers a more flexible working environment and will remain accessible even as user requirements change.

While Pynapple expands accessibility to data analysis, it has some limitations inherent to its design. The first issue is that Pynapple is currently only available through Python. Thus, some transition is required for those primarily trained in other programming languages commonly used in neuroscience, including MATLAB and Julia. The design of the package around objects is a strength in many regards but could represent a challenge for users who are not accustomed to this programming approach. We have addressed this concern by providing users with detailed documentation, which includes a broad variety of examples. We will also keep on providing training opportunities for all future users. Last, Python code may run slower than similar code written in other languages. Pynapple is based on Pandas, whose methods are already highly optimized. Yet, current development is underway to improve computation speed and these developments are transparent for the users as they won't change the organization of the package.

Soon, Pynapple will be part of an entire suite of plugin libraries that we are developing to further enhance Pynapple. To keep Pynapple robust and stable, we will develop these plugins as standalone packages. These external packages will include an automated datalogger for recapitulating analyses, an on-line visualizer for Pynapple objects, and a package for parallel computing in Pynapple. This will address the speed issue inherent to code written in Python by allowing multiple analyses to be performed simultaneously. These packages will begin to address

the limitations of Pynapple we described previously, enhancing the long-term stability of Pynapple, and streamlining accessibility for its users.

References

1. Folk, M., Heber, G., Koziol, Q., Pourmal, E. & Robinson, D. An overview of the HDF5 technology suite and its applications. in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases* 36–47 (Association for Computing Machinery, 2011). doi:10.1145/1966895.1966900.
2. Wells, D. C. & Greisen, E. W. *FITS - a Flexible Image Transport System*. 445 (1979).
3. Pedregosa, F. *et al.* Scikit-learn: Machine learning in Python. *J. Mach. Learn. Res.* **12**, 2825–2830 (2011).
4. van der Maaten, L. J. P. & Hinton, G. E. Visualizing High-Dimensional Data Using t-SNE. *J. Mach. Learn. Res.* **9**, 2579–2605 (2008).
5. McKinney, W. & others. pandas: a foundational Python library for data analysis and statistics. *Python High Perform. Sci. Comput.* **14**, 1–9 (2011).
6. Abraham, A. *et al.* Machine learning for neuroimaging with scikit-learn. *Front. Neuroinformatics* 14 (2014).
7. Tadel, F., Baillet, S., Mosher, J. C., Pantazis, D. & Leahy, R. M. Brainstorm: A User-Friendly Application for MEG/EEG Analysis. *Comput. Intell. Neurosci.* **2011**, 879716 (2011).
8. Oostenveld, R., Fries, P., Maris, E. & Schoffelen, J.-M. FieldTrip: Open Source Software for Advanced Analysis of MEG, EEG, and Invasive Electrophysiological Data. *Comput. Intell. Neurosci.* **2011**, e156869 (2010).
9. Bokil, H., Andrews, P., Kulkarni, J. E., Mehta, S. & Mitra, P. P. Chronux: A platform for analyzing neural signals. *J. Neurosci. Methods* **192**, 146–151 (2010).
10. Garcia, S. *et al.* Neo: an object model for handling electrophysiology data in multiple formats. *Front. Neuroinformatics* **8**, (2014).
11. Freeman, J. *et al.* Mapping brain activity at scale with cluster computing. *Nat. Methods* **11**, 941–950 (2014).
12. Stevenson, I. H. & Kording, K. P. How advances in neural recording affect data analysis. *Nat. Neurosci.* **14**, 139–142 (2011).
13. Urai, A. E., Doiron, B., Leifer, A. M. & Churchland, A. K. Large-scale neural recordings call for new insights to link brain and behavior. *Nat. Neurosci.* **25**, 11–19 (2022).
14. Pachitariu, M., Steinmetz, N., Kadir, S., Carandini, M. & Harris, K. D. Kilosort: realtime spike-sorting for extracellular electrophysiology with hundreds of channels. *bioRxiv* 061481 (2016) doi:10.1101/061481.
15. Pachitariu, M. *et al.* Suite2p: beyond 10,000 neurons with standard two-photon microscopy. 061507 Preprint at https://doi.org/10.1101/061507 (2017).
16. Hazan, L., Zugaro, M. & Buzsáki, G. Klusters, NeuroScope, NDManager: A free software suite for neurophysiological data processing and visualization. *J. Neurosci. Methods* **155**, 207–216 (2006).

17. Fee, M. S., Mitra, P. P. & Kleinfeld, D. Automatic sorting of multiple unit neuronal signals in the presence of anisotropic and non-Gaussian variability. *J. Neurosci. Methods* **69**, 175–188 (1996).
18. Harris, K. D., Henze, D. A., Csicsvari, J., Hirase, H. & Buzsáki, G. Accuracy of tetrode spike separation as determined by simultaneous intracellular and extracellular measurements. *J. Neurophysiol.* **84**, 401–414 (2000).
19. Yger, P. *et al.* A spike sorting toolbox for up to thousands of electrodes validated with ground truth recordings in vitro and in vivo. *eLife* **7**, e34518 (2018).
20. Mathis, A. *et al.* DeepLabCut: markerless pose estimation of user-defined body parts with deep learning. *Nat. Neurosci.* **21**, 1281–1289 (2018).
21. Zhou, P. *et al.* Efficient and accurate extraction of in vivo calcium signals from microendoscopic video data. *eLife* **7**, e28728 (2018).
22. Mukamel, E. A., Nimmerjahn, A. & Schnitzer, M. J. Automated Analysis of Cellular Signals from Large-Scale Calcium Imaging Data. *Neuron* **63**, 747–760 (2009).
23. Romano, S. A. *et al.* An integrated calcium imaging processing toolbox for the analysis of neuronal population dynamics. *PLOS Comput. Biol.* **13**, e1005526 (2017).
24. Kaifosh, P., Zaremba, J. D., Danielson, N. B. & Losonczy, A. SIMA: Python software for analysis of dynamic fluorescence imaging data. *Front. Neuroinformatics* **8**, (2014).
25. Pnevmatikakis, E. A. & Giovannucci, A. NoRMCorre: An online algorithm for piecewise rigid motion correction of calcium imaging data. *J. Neurosci. Methods* **291**, 83–94 (2017).
26. Teeters, J. L. *et al.* Neurodata Without Borders: Creating a Common Data Format for Neurophysiology. *Neuron* **88**, 629–634 (2015).
27. Rübel, O. *et al.* The Neurodata Without Borders ecosystem for neurophysiological data science. *eLife* **11**, e78362 (2022).
28. Nasiotis, K. *et al.* Integrated open-source software for multiscale electrophysiology. *Sci. Data* **6**, 231 (2019).
29. Unakafova, V. A. & Gail, A. Comparing Open-Source Toolboxes for Processing and Analysis of Spike and Local Field Potentials Data. *Front. Neuroinformatics* **13**, (2019).
30. Zhang, K., Ginzburg, I., McNaughton, B. L. & Sejnowski, T. J. Interpreting Neuronal Population Activity by Reconstruction: Unified Framework With Application to Hippocampal Place Cells. *J. Neurophysiol.* **79**, 1017–1044 (1998).
31. Brown, E. N., Frank, L. M., Tang, D., Quirk, M. C. & Wilson, M. A. A statistical paradigm for neural spike train decoding applied to position prediction from ensemble firing patterns of rat hippocampal place cells. *J. Neurosci.* **18**, 7411 (1998).
32. Peyrache, A., Lacroix, M. M., Petersen, P. C. & Buzsáki, G. Internally organized mechanisms of the head direction sense. *Nat. Neurosci.* **18**, 569–575 (2015).
33. Peyrache, A., Petersen, P. C. & Buzsaki, G. Extracellular recordings from multi-site silicon probes in the anterior thalamus and subicular formation of freely moving mice. *CRCNS.org* doi:http://dx.doi.org/10.6080/K0G15XS1.
34. Siegle, J. H. *et al.* Survey of spiking in the mouse visual system reveals functional hierarchy. *Nature* **592**, 86–92 (2021).
35. Zheng, J. *et al.* Neurons detect cognitive boundaries to structure episodic memories in humans. *Nat. Neurosci.* **25**, 358–368 (2022).
36. Olmos, A. & Kingdom, F. A. A. A Biologically Inspired Algorithm for the Recovery of Shading and Reflectance Images. *Perception* **33**, 1463–1473 (2004).

37. Zhou, P. *et al.* EASE: EM-Assisted Source Extraction from calcium imaging data. *bioRxiv* (2020).
38. Turner-Evans, D. B. *et al.* The Neuroanatomical Ultrastructure and Function of a Biological Ring Attractor. *Neuron* **108**, 145-163.e10 (2020).
39. Chaudhuri, R., Gercek, B., Pandey, B., Peyrache, A. & Fiete, I. The intrinsic attractor manifold and population dynamics of a canonical cognitive circuit across waking and sleep. *Nat. Neurosci.* **22**, 1512–1520 (2019).
40. Viejo, G. & Peyrache, A. Precise coupling of the thalamic head-direction system to hippocampal ripples. *Nat. Commun.* **11**, 2524 (2020).
41. Gardner, R. J. *et al.* Toroidal topology of population activity in grid cells. *Nature* **602**, 123–128 (2022).

Data Availability

All data used in this manuscript are publicly available.

Code Availability

All code is publicly available online.
Pynapple: https://github.com/PeyracheLab/pynapple
Pynacollada: https://github.com/PeyracheLab/pynacollada

Acknowledgment

This work was supported by a Canadian Research Chair in Systems Neuroscience, CIHR Project Grant 155957, NSERC Discovery Grant RGPIN-2018-04600, and the Canada-Israel Health Research Initiative, jointly funded by the Canadian Institutes of Health Research, the Israel Science Foundation, the International Development Research Centre, Canada and the Azrieli Foundation 108877-001 (AP).

Author Contributions

G.V., A.P., and F.P.B. designed and developed Pynapple, with contribution from L.S. All authors contributed to the writing of the manuscript and the design of the figures; A.P. supervised the project.

Competing Interests

No competing interests disclosed

Materials & Correspondence

All correspondence should be directed to Adrien Peyrache at adrien.peyrache@mcgill.ca