NeoAnalysis

A Python-Based Toolbox for Quick Electrophysiological Data Processing and Analysis

Documentation V1.0.0

Contents

Overview

NeoAnalysis is a Python-based open-source toolbox that provides the most commonly used functions for electrophysiological data processing and analysis, including spike detection, spike sorting, signal filtering, spike train analysis, LFP analysis, and behavioral data analysis. For each of these analyses, users simply use the graphic user interface (GUI) or specify the parameters in commands and generally will not need to write additional scripts. Specifically, NeoAnalysis has the following features:

I. NeoAnalysis adopts the Neo [1], a powerful open-source module for data input/output (I/O), to import data, which supports most data formats from the major commercial data acquisition systems including Blackrock (Blackrock Microsystems LLC, Utah, USA), Plexon (Plexon Inc., Dallas, TX, USA), and TDT (Tucker-David Technologies; Alachua, FL, USA). All of the input data, including the recorded signal, experimental settings, and the behavioral responses are then converted to HDF5 files for storage or further analysis. The HDF5 format is used because it is a highly efficient format for data I/O, especially for data of a large volume and a complex structure. In addition, it is a unified format that can be used by different operating systems and programming languages [2]. This procedure substantially eliminates the limitation due to data format and substantially eases the work in data storage and sharing.

II. NeoAnalysis integrates the open-source module PyQtGraph [3] to provide user-friendly GUI and data viewing. The PyQtGraph is a Python based graphics and GUI library, which uses less memory and performs much more efficiently than simply using the embedded graphic library ‘matplotlib’ [4]. Furthermore, NeoAnalysis puts a lot of emphasis on user interaction design. In particular, it provides several easy-to-use widgets for offline spike sorting.

III. NeoAnalysis groups all of the experimental information, including the recorded signals, behavioral responses, and the results of preprocessing into a table on a trial-by-trial basis and is capable of easily displaying the data table, which can be further sorted according to given conditions (e.g. experimental conditions). The table is very informative, and NeoAnalysis provides many other functions to run further analysis and to plot results.

IV. NeoAnalysis provides a complete workflow for electrophysiological data analysis, which covers data standardizing, data preprocessing, single unit analysis, data storage, and population data analysis. Throughout the entire data analysis process, users do not have to switch between different programs and toolboxes. More important, NeoAnalysis supports analyzing with automatic condition sorting. Users can obtain sorted results by simply specifying parameters in the commands.

V. NeoAnalysis is capable of processing eye movement information, including calibrating eye position and detecting saccades. During experiments, when recording eye movement trajectories, it is essential to detect the occurrence of saccades and to extract the relevant information. Previous open-source toolboxes generally do not provide such functions.

VI. Due to the incompatibilities between Python 2.7 and Python 3.5, NeoAnalysis provides two slightly different versions for the two releases.

References

1. Garcia S, Guarino D, Jaillet F, Jennings T, Propper R, Rautenberg PL, Rodgers CC, Sobolev A, Wachtler T, Yger P, Davison AP: **Neo: an object model for handling electrophysiology data in multiple formats.** *Front Neuroinform* 2014, **8**:10.
2. Folk M, Cheng A, Yates K: **HDF5: A file format and I/O library for high performance computing applications.** In *Proceedings of Supercomputing*. 1999:5-33.
3. **PyQtGraph - Scientific Graphics and GUI Library for Python** [http://www.pyqtgraph.org/]
4. Hunter J: **Matplotlib: A 2D graphics environment.** *Computing In Science & Engineering* 2007, **9**(3):90-95.

Analysis Modules

As depicted in Fig.1, NeoAnalysis has six major modules:

TransFile

The module for converting recording files from different data acquiring systems to HDF5 format.

SpikeDetection

The module for detecting spikes from the raw signals.

SpikeSorting

The module for offline spike sorting.

AnalogFilter

The module for filtering analog signal.

Graphics

The module for visualizing data and analysis results. It can group data into a table on a trial-by-trial basis according to experimental conditions, and then provide users access to perform analysis like plotting PSTH and other common application.

PopuAnalysis

The module for analyzing data at the population level.

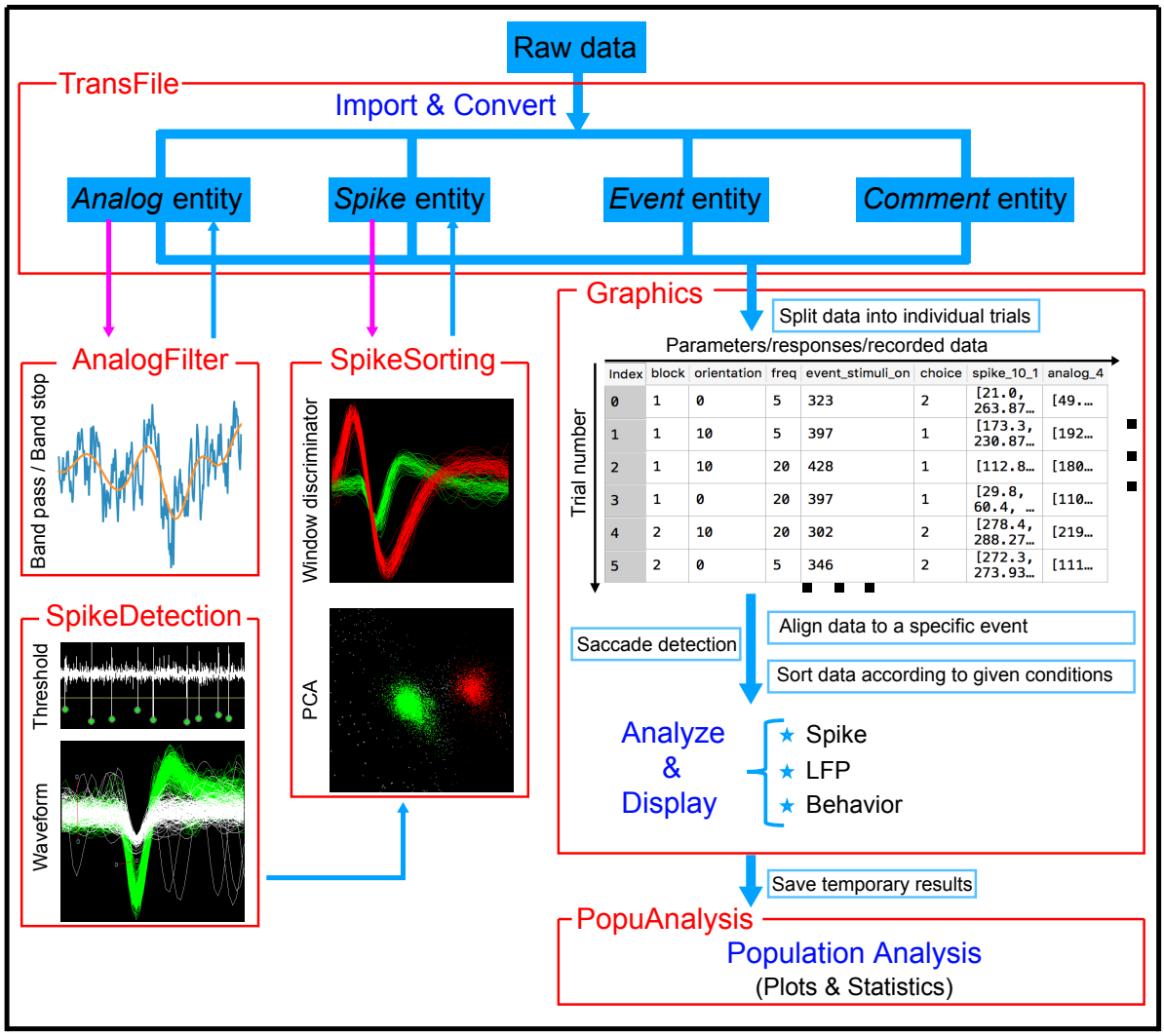

Fig. 1 The major modules and workflow of NeoAnalysis

The Workflow of NeoAnalysis

The procedures of using the six modules of NeoAnalysis are depicted by the arrows in Fig.1. Specifically, they can be described as below:

The TransFile module is used to import raw data from different recording systems and extract the data to *Analog*, *Spike*, *Event* and *Comment* entities, and then convert to a standardized HDF5 format. There is a step-by-step tutorial about how to use this module in the Tutorials chapter.

The Analog module is used to filter analog signals.

The SpikeDetection module is used to detect spikes from the analog signal and save detected spikes in the same HDF5 file. All detected spikes are unclassified and saved with the grouped name 'spikes/spike_channelNumber_0' in the HDF5 file.

The SpikeSorting module is used for offline spike sorting. A GUI window is provided and users can sort spikes using two methods: window discriminator and PCA discriminator.

The Graphics module first organizes all of the relevant data into a data table on a trial-by-trial basis for data visualization, and then provides functions to analyze the data and display the results. This module supports data computation and saccade detection in addition to common analysis of spikes, local field potentials, and other behaviors.

The PopuAnalysis module can retrieve the saved result of each single session and perform population analysis.

Tutorials

In this tutorial, we will give some examples to show how to use NeoAnalysis. These examples include several commonly used functions for electrophysiology data analysis, including offline spike sorting, plotting peristimulus time histogram (PSTH), raster and accumulated spike counts. In particular, since all data processing and analysis functions of NeoAnalysis are based on the converted HDF5 file, we provide a step-by-step tutorial to show how to convert the input data to HDF5 format.

Installation

The NeoAnalysis runs on all platforms including Windows, Linux, and Mac OS X that support Python and OpenGL. Users can freely download the source code and user manual from the website https://github.com/neoanalysis/NeoAnalysis. There are two slightly different versions for Python 2.7 and Python 3, respectively.
NeoAnalysis relies on the following dependent packages:

'numpy>=1.11.3',
'scipy>=0.18.1',
'matplotlib>=2.0.0',
'scikit-learn>=0.18.1',
'quantities>=0.11.1',
'pyopengl>=3.1.0',
'seaborn>=0.7.1',
'pandas>=0.19.2',
'h5py>=2.6.0',
'statsmodels>=0.6.1',

We recommend installing these dependent packages using Anaconda, a famous package and environment manager for Python. First, activate root environment of Anaconda in the command line:

```
>>> source activate root        # for Mac and Linux
>>> activate root               # for Windows
```

Then, install the dependent packages using the command:

```
>>> pip install numpy  scipy  matplotlib  scikit-learn  quantities  pyopengl  seaborn  pandas h5py  statsmodels
```

Finally, activate the *root* environment of Anaconda, unzip the downloaded NeoAnalysis file, enter the folder and install NeoAnalysis:

```
>>> cd /Downloads/NeoAnalysis/NeoAnalysis_Py2.7/        # for Python 2.7.x environment
>>> cd /Downloads/NeoAnalysis/NeoAnalysis_Py3.5/        # for Python 3.x environment
>>> python setup.py install
```

TransFile

Given a typical electrophysiological experiment normally collects spikes, analog signals (e.g. LFP), experimental settings, and behavioral responses (Fig. 2), NeoAnalysis divides the

electrophysiological data into four basic entities: *Spike*, *Analog*, *Event* and *Comment*, as depicted in Fig. 2. All its data processing and analysis functions are based on these four entities.

. A *Spike* entity contains the time points at which the action potentials occur, as well as their waveforms and unit classification.

. An *Analog* entity contains the continuous data that was recorded at a given sampling frequency, such as LFP.

. An *Event* entity contains the time points and the labels defining the occurrence of specific events, such as the onset of a stimulus or the beginning of a trial.

. A *Comment* entity contains the time points and the labels that define the experimental settings, such as the direction of the stimulus in each trial.

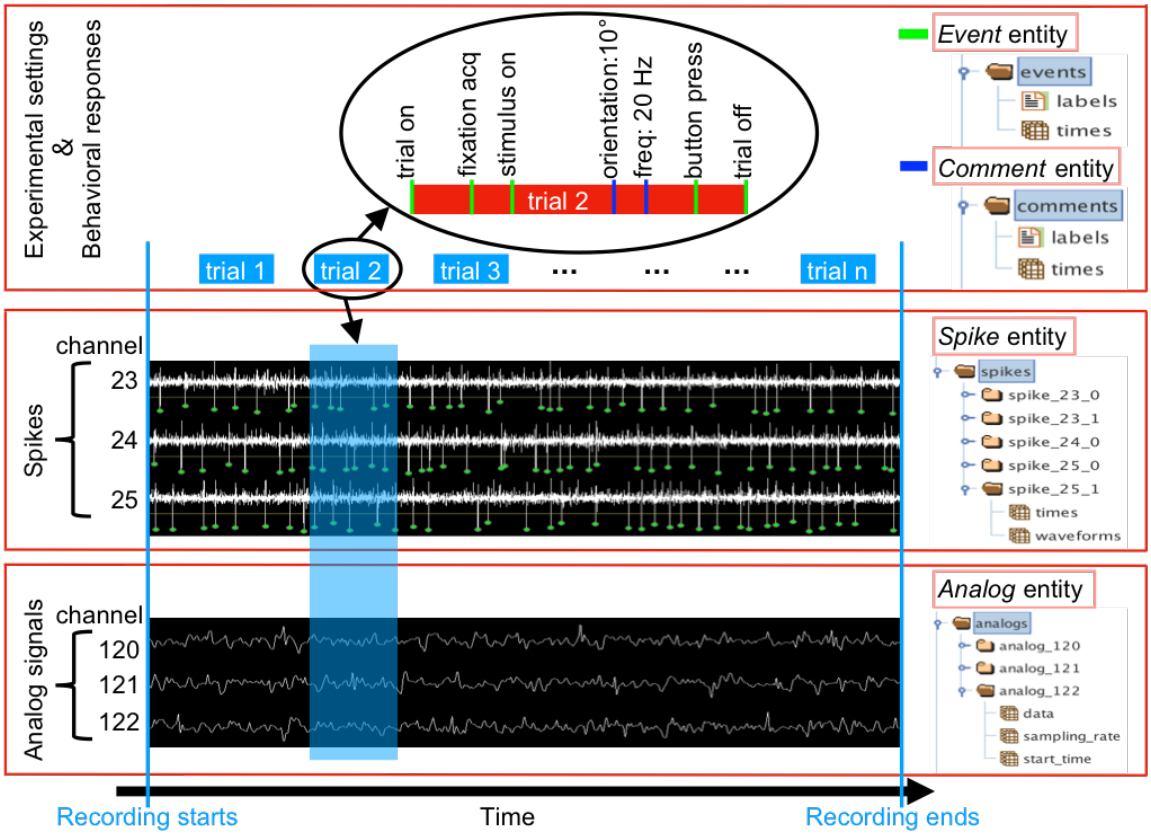

Fig. 2 The principle for data standardization in a typical electrophysiological experiment. The experiment normally runs in a trial-by-trial manner, and the data collected include experimental settings defining the conditions in each trial and the behavioral and neuronal responses. All these data can be divided into four entities: *Event*, *Comment*, *Spike* and *Analog*. The right side shows how these entities are organized in the output HDF5 file.

Converting the raw data to the four basic entities is easy. Users can do this by themselves or with the help of the TransFile module. Currently, the TransFile module has been carefully tested using some raw data recorded from Blackrock and Plexon. Since TransFile module adapts the Neo toolbox for data reading, in theory it can be extended to support all major data

recording systems. We are also happy to assist readers to develop their own interfaces for data importing.

Here, we give a step-by-step tutorial to show how to convert the data recorded by Blackrock and Plexon to the HDF5 file format with a well-defined data structure.

Suppose the experiment studies the difference in orientation tuning using stimuli of different spatial frequencies. There are two spatial frequencies, 5 Hz, 20 Hz; and 4 orientations, 0°, 90°, 180°, 270°. During the experiment execution period, when a trial starts, a digital number 67 is sent to the recording system. When the stimulus turns on, a digital number 68 is sent to the recording system. For the experimental conditions, we use digital number 100, 101, 102, 103, 104, 105, 106, 107 to represent [5 Hz, 0°], [5 Hz, 90°], [5 Hz, 180°], [5 Hz, 270°], [20 Hz, 0°], [5 Hz, 90°], [20 Hz, 180°], [20 Hz, 270°]. For example, when the experimental condition for an executing trial is [5 Hz, 270°], a digital number 103 is sent to the recording system at some point during the trial executing period. Meanwhile, the spikes obtained using online sorting are recorded in channel 23; three analog signals are recorded in channel 100, 101 and 102, accordingly. Suppose the sampling frequencies of the analog signals recorded in these channels are 1000 Hz, 1000 Hz and 10000 Hz, respectively.

Blackrock

For the Blackrock system, the spikes and events entities are stored in the .nev file, and the Analog entity is sotred in the .ns2 and .ns4 files. The codes for converting these entities are:

```
>>> from NeoAnalysis import TransFile                                    # line 1
>>> TransFile.transfile(filename = 'test_raw_data', machine_type = 'blackrock',
                        replace = True, nsx_to_load = [2, 4])            # line 2
>>> TransFile.generate_comments(filename = 'test_raw_data.h5', method = 'map',
                                replace = True,
                        mapping = {'digital_input_port/100':['frequency:5', 'orientation:0'],
                                   'digital_input_port/101':['frequency:5', 'orientation:90'],
                                   'digital_input_port/102':['frequency:5', 'orientation:180'],
                                   'digital_input_port/103':['frequency:5', 'orientation:270'],
                                   'digital_input_port/104':['frequency:20', 'orientation:0'],
                                   'digital_input_port/105':['frequency:20', 'orientation:90'],
                                  'digital_input_port/106':['frequency:20', 'orientation:180'],
                                 'digital_input_port/107':['frequency:20', 'orientation:270']})
                                                                         # line 3
>>> TransFile.generate_events(filename = 'test_raw_data.h5', method = 'map',
                              replace = True,
                              mapping = {'digital_input_port/67':'trial_on',
                                         'digital_input_port/68':'stimuli_on'})
                                                                                  # line 4
```

In line 2, the parameter *filename* is the file name of the raw recording file (without extension). This command generates a new file, named 'test_raw_data.h5'. Setting the parameter *replace* to be True means clearing the content in the 'test_raw_data.h5' if it already exists. The parameter nsx_to_load determines which .nsx files to be loaded. Here, [2, 4] means loading .ns2 and .ns4. Fig. 3 depicts the data structure in this file.

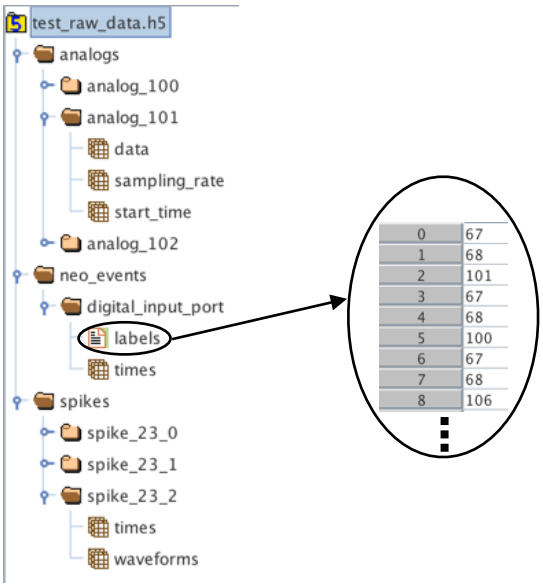

Fig. 3 The data structure after using the TransFile.*transfile* function

In the converted file, the *Analog* entity contains the analog signals recorded in channel100, 101 and 102.The *Spike* entity contains spikes recorded in channel 23, which are sorted online to unit 0,1 and 2. The *neo_events* contains the data about experimental settings and responses that need to be converted to the *Comment* entity and the *Event* entity.
The *Comment* entity and the *Event* entity can be generated using the *generate_comments* and the *generate_events* functions, respectively (line 3 and line 4). In line 3 and 4, the parameter *mapping* is a dictionary that translates the recorded digital numbers from the *neo_events* to their real meanings in the experiment. The final data structure is shown in Fig. 4.

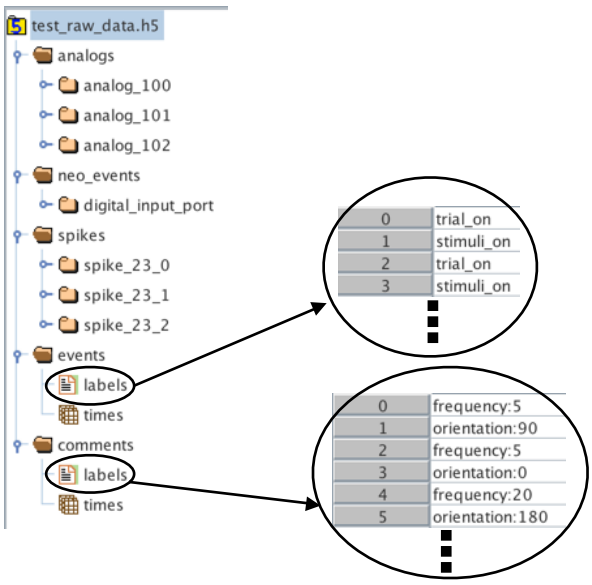

Fig. 4 The data structure in the HDF5 file after using the *generate_comments* and the *generate_events* functions

Plexon

Suppose the trial beginning and the stimulus onset are recorded in *Event001* and *Event002* of Plexon; the digital numbers representing the experimental conditions, including 100, 101, 102, 103, 104, 105, 106, are recorded in the *Strobed* of Plexon. Use the following codes to convert data:

```
>>> from NeoAnalysis import TransFile                                    # line 1
>>> TransFile.transfile(filename = 'test_raw_data', machine_type = 'plexon',
                        replace = True, nsx_to_load = [2, 4])            # line 2
>>> TransFile.generate_comments(filename = 'test_raw_data.h5', method = 'map',
                                replace = True,
                        mapping = {'Strobed/100':['frequency:5', 'orientation:0'],
                                   'Strobed/101':['frequency:5', 'orientation:90'],
                                   'Strobed/102':['frequency:5', 'orientation:180'],
                                   'Strobed/103':['frequency:5', 'orientation:270'],
                                   'Strobed/104':['frequency:20', 'orientation:0'],
                                   'Strobed/105':['frequency:20', 'orientation:90'],
                                   'Strobed/106':['frequency:20', 'orientation:180'],
                                  'Strobed/107':['frequency:20', 'orientation:270']})
                                                                         # line 3
>>> TransFile.generate_events(filename = 'test_raw_data.h5', method = 'map',
                              replace = True,
                              mapping = {'Event001/0':'trial_on',
                                         'Event002/0':'stimuli_on'})
                                                                    # line 4
```

In line 2, the parameter *filename* is the file name of the raw recording file (without extension). Currently, the TransFile module only supports .plx but not .pl2. This command generates a new file, named 'test_raw_data.h5'. Setting the parameter *replace* to be True means clearing the content of the 'test_raw_data.h5' if it already exists. Its data structure is shown in Fig 5.

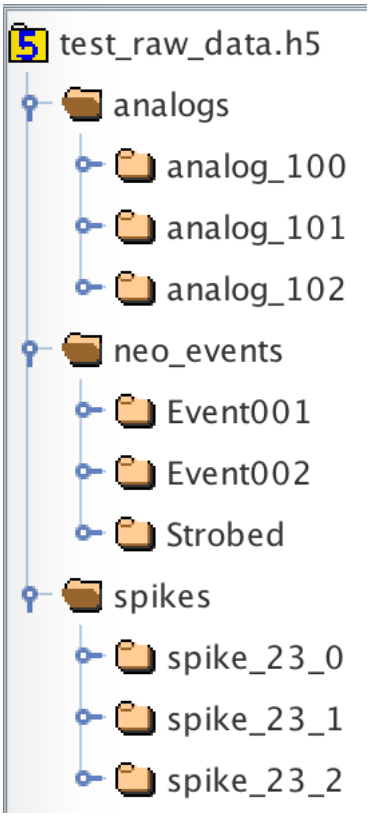

Fig. 5 data structure after using the *transfile* function

In the converted file, the *Analog* entity contains the analog signals recorded in channel 100, 101 and 102.The *Spike* entity contains spikes recorded in channel 23, which are sorted online to unit 0,1 and 2. The *Comment* entity and the *Event* entity can be generated using the *generate_comments* and the *generate_events* functions, respectively (line 3 and line 4). In line 3 and 4, the parameter *mapping* is a dictionary that translates the recorded digital numbers to their real meaning in the experiment. The final data structure is shown Fig. 6.

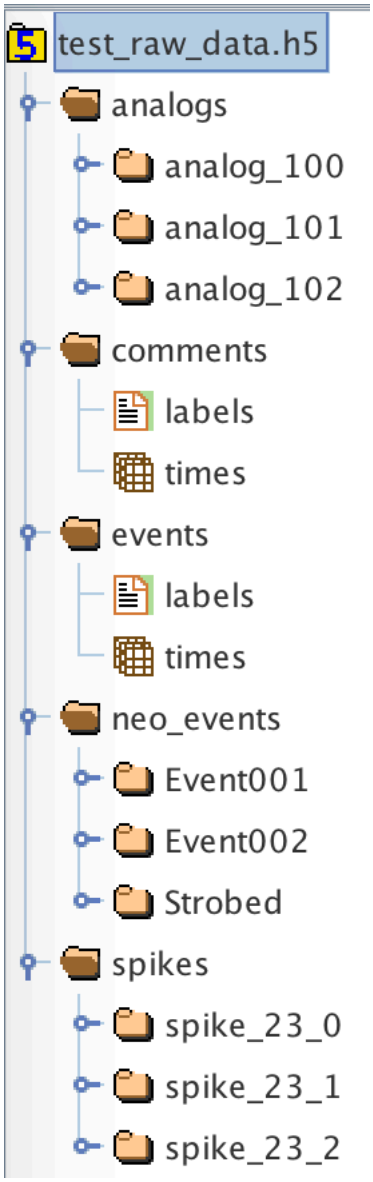

Fig. 6 The data structure in the HDF5 file after using the *generate_comments* and the *generate_events* functions

Offline spike sorting

```
>>> from NeoAnalysis import SpikeSorting
>>> SpikeSorting(3d_pca = True)
```

An interface with several buttons and panels with be displayed. Users can load data and perform offline spike sorting using the window discriminator and/or the principal components analysis discriminator. Fig. 7 shows the graphic user interface (GUI) for offline spike sorting.

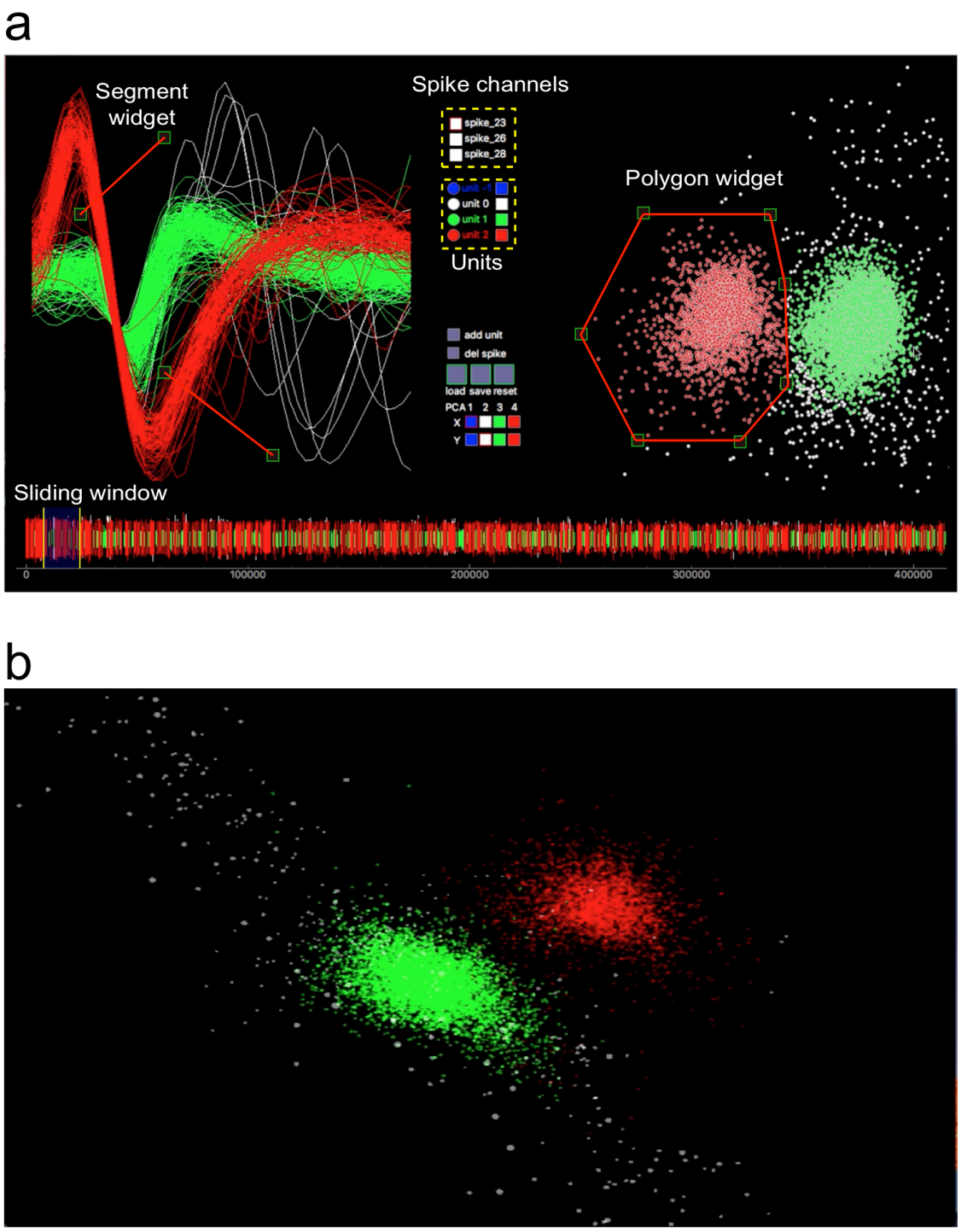

Fig. 7 The graphic user interfaces for offline spike sorting. (a) The main interface, in which the center panel shows the spike channels and the units included in the selected channel; the bottom panel shows all spikes in the selected channel with a sliding window to select a portion of spikes; the left panel shows the waveforms of the selected spikes, and the right panel shows the principal components of all spikes in the selected channel. Users can use the segment widget (two red lines with square ends) to select waveforms or use the polygon widget (red polygon with square nodes) to select data points for re-sorting. (b) A 3D view to display the first three principal components of all spikes in the selected channel.

Plot PSTH and raster

In the command window, run the following codes:

```
>>> from NeoAnalysis import Graphics                                    # line 1
>>> filename = '/data_folder/graphics_data.h5'                          # line 2
>>> sg = Graphics(filename = filename, trial_start_mark = '64715',
                  comment_expr = 'key:value')                          # line 3
>>> sg.to_numeric(['patch_direction'])                                  # line 4
>>> firingRate = sg.plot_spike(channel = 'spike_26_1', sort_by = ['patch_direction'],
                  align_to = 'event_64721', pre_time = -300, post_time = 2000, bin_size = 30,
                  overlap = 10, filter_nan = ['event_64721', 'event_64722'], fig_column = 4,
                  fig_marker = [0])  # line 5
```

In line 3, class *Graphics* is initiated. The *trial_start_mark* is the marker representing the start of a trial, which is used to separate the raw data into different trials. The *comment_expr* tells the program how experimental conditions and parameters are stored in the data. Line 4 converts data in *patch_direction* column from string type to numeric type. Line 5 plots raster and PSTH. The *channel* defines the spike channel and the unit order. The *sort_by* defines which experimental conditions are used to sort the data. The *align_to* defines which event marker is used to align the data. The *pre_time* and *post_time* represent the time range (relative to *align_to*) selected for analysis. The *bin_size* and *overlap* represent the bin width for computing PSTH and the overlapping time between two adjacent bins. Fig. 8 is the output of line 5.

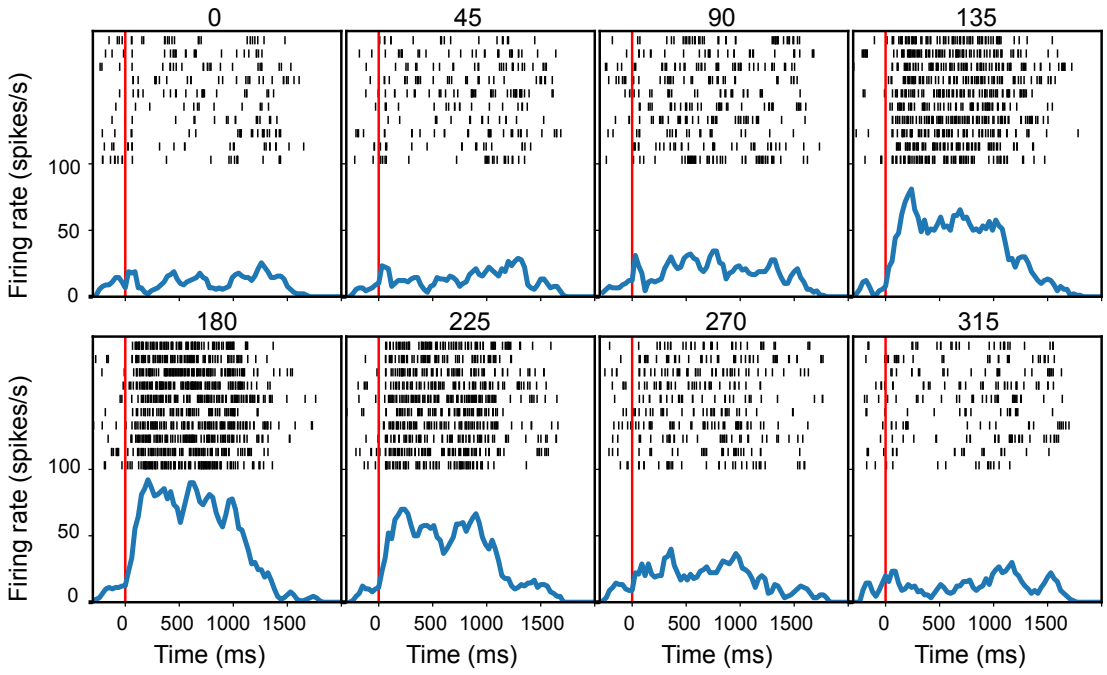

Fig. 8 The raster and PSTH plots for the sample data. Each panel represents the response to one condition defined by the setting *sort_by*.

Plot spike counts

```
>>> spk_count = sg.plot_spike_count(channel='spike_26_1', sort_by = ['patch_direction'],
                  align_to = 'event_64721', timebin = [0, 1000])
```

The above command plots the spike counts during the period defined by the parameter *timebin* (relative to *align_to*). The output of this command is Fig. 9.

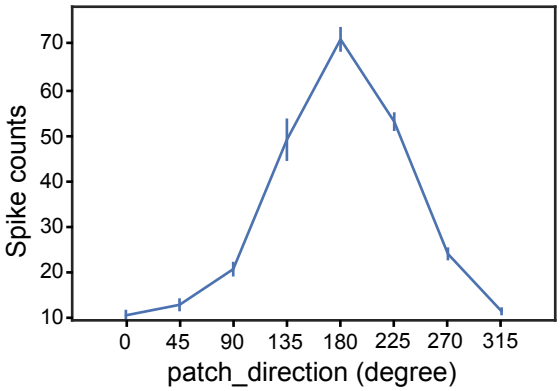

Fig. 9 The line plot of the spike counts for the sample data. Each point represents the spike count within a given period for one experimental condition.

API Reference

TransFile

TransFile is a module for converting files recorded from different data acquisition systems to HDF5 format in a well-defined data structure.

Class transfile

The transfile is a class for converting different recording files to HDF5 format in a well-defined data structure. This class uses the open-source package Neo as data I/O from different data acquiring systems.

Data structure in the converted HDF5 file:

spike_channel (Number)_unit (Number): contain spike timestamps and waveforms

analog_channel (Number): contain analog data values, sampling rate and the recording start time.

neo_events: contain data values and timestamps, which is equal to events in Neo. Neo_events can be extracted as comments and events using additional functions: generate_comments and generate_events.

Args

filename (string):

File name (without extension)

machine_type (string):

Defines which data acquisition system is used to record the file, e.g. 'blackrock'.

Currently, we have strictly tested data from Blackrock and Plexon. For other data acquisition systems, this function needs additional test as we cannot obtain sufficient demo data. We would be happy to help users test these functions if they would like to provide demo files.

Please contact us through email bozhang23@outlook.com

replace (Boolean):

If True, data already existed in the converted file will be overwritten.

**arg (Dict):

Extra arguments needed for specific machine_type.

For 'blackrock':

nsx_to_load: specify which .nsx files are needed to be translated.

channels: specify which spikes channels are needed to be translated.

units: specify which spikes units are needed to be translated.

Returns

-

Examples

>>> TransFile('myfile', 'blackrock', True, nsx_to_load=[2,4])

Translate 'myfile', recorded by 'blackrock' machine, to HDF5 format. Both .ns2 and .ns4 file will be translated (.nev will always be translated).

Class generate_comments
This class extracts comment entities from neo_events. A comment entity contains time points and the labels that define the experimental settings, such as the direction of the stimulus in each trial.
Args
Filename (string):
File name (with extension)
method (string):
if ‘move’, move data from neo_events to comments
if ‘map’, map data in neo_events to comments
replace (Boolean):
if True, data already in file will be rewritten.
**arg:
if method = ‘move’, **arg need to specify which key in neo_events should be moved.
if method = ‘map’, **arg need to specify the mapping dictionary.
Returns
-
Examples

```
>>> generate_comments(filename = ‘test1.h5’, method=’move’, replace=True,
                      key = ‘cond’)
```

This command moves ‘cond’, which contains information about experimental condition, from the neo_events to the comments entity.

```
>>> generate_comments(filename = ‘test2.h5’, method=’map’, replace=True,
                      mapping={‘Strobed/101’:[‘direction:0’,’frequency:5’],
                               ‘Strobed/102’:[‘direction:0’,’frequency:10’],
                               ‘Strobed/103’:[‘direction:0’,’frequency:30’],
                               ‘Strobed/104’:[‘direction:90’,’frequency:5’],
                               ‘Strobed/105’:[‘direction:90’,’frequency:10’],
                               ‘Strobed/106’:[‘direction:90’,’frequency:30’],
                               ‘Strobed/107’:[‘direction:180’,’frequency:5’],
                               ‘Strobed/108’:[‘direction:180’,’frequency:10’],
                               ‘Strobed/109’:[‘direction:180’,’frequency:30’]})
```

The command above defines the labels 101, 102, 103 … 108, 109 in Strobed (one key of the neo_entity) to represent different experimental conditions. This command extracts these experimental conditions together with their timestamps from the neo_events and then generates the comments entity.

Class generate_events
This class extracts event entities from the neo_events. Event entities contain time points and the labels defining the occurrence of specific events, such as the onset of a stimulus or the beginning of a trial.
Args

Filename (string):

File name (with extension)

method (string):

if ‘move’, move data from neo_events to events

if ‘map’, map data in neo_events to events

replace (Boolean):

if True, data already in file will be rewritten.

**arg:

if method = ‘move’, **arg need to specify which key in neo_events should be moved.

if method = ‘map’, **arg need to specify the mapping dictionary.

Returns

-

Examples

>>> generate_events(filename = ‘test1.h5’, method=’move’, replace=True, key = ‘digital_input_port’)

The command above moves ‘digital_input_port’, which contains markers representing the happening of certain events, from neo_events to events entity.

>>> generate_comments(filename = ‘test2.h5’, method=’map’, replace=True, key = {‘Event001/0’:’event_001’, ‘Event004/0’:’event_002’, ‘Event005/0’:’event_003’})

The command above defines label 0 in Event001, label 0 in Event004 and label 0 in Event005 as event_001, event_002 and event_003, respectively. Alternatively, they can be renamed in a more readable way. For example, name the event as ‘trial on’, ‘stimulus on’ and ‘stimulus off’, etc. The output of this command generates a event entity.

SpikeDetection

SpikeDetection is a module for offline spike sorting.

Class SpikeDetection

The class for detecting spikes from the analog signals in the converted hdf5 file.

Users can choose spikes based on thresholds and/or waveforms. This class provides a GUI for users to adjust threshold and select waveforms. The detected spikes will be saved as *spike entity* in the same hdf5 file.

Args

-

Returns

-

Examples

>>>SpikeDetection()

Open a GUI window for spike detection.

SpikeSorting

SpikeSorting is a module for offline spike sorting.

Class SpikeSorting
The class for offline spike sorting.
Two methods are provided for spike sorting, window discriminator and principal components analysis. A line widget and a polygon widget are provided for window discriminator and PCA discriminator respectively. The graphic user interface design and data plotting tools are based on an open-source project, PyQtGraph.
Args

pca_3d (Boolean):

If True, a 3D view of the first three principal components of spikes will be shown.

Default: False

reclaim_space (Boolean):

If True, the storage space that the file occupied will be reallocated, which will release extra storage space.

Default: False

Returns

-

Examples

>>> SpikeSorting(True)

Open the offline spike sorting window with a 3D view for the first three principal components of spikes.

AnalogFilter

This is a module for filtering analog signal. The filter methods include band-pass and band-stop filtering. This module can be used with or without a GUI window.

Class AnalogFilter
This is a Class for filtering analog signal. Two filtering methods are provided: band-pass filtering and band-stop filtering. This class provides an optional GUI window to show the filtering results instantaneously.
Args

gui (Boolean):

If True, a GUI window will be displayed, in which users can select the analog channels and filtering methods to process the data.

Default: True

reclaim_space (Boolean):

If True, the storage space that the file occupied will be reallocated, which will release extra storage space.

Default: False

filename (string):
The filename (including the file path) for analysis.
When gui = False, users need to set the filename (with extension).
channels (str or list):
The channel for filtering.
When gui = False, users need to set the channel.
bandpass (list):
When gui = False, users need to set the bandpass.
Set band-pass value for signal filtering.
bandstop (list):
When gui = False, users need to set the bandstop.
Set band-stop value for signal filtering.

Returns

-

Examples

>>> AnalogFilter(False, False, 'myfile.h5', ['analog_23','analog_26'], [4,100], [59,61])
Use setting band-pass (4-100 Hz) and band-stop (59-61 Hz) to filter signal in channels analog_23 and analog_26 in the file 'myfile.h5'

Graphics

This module groups data into a table on a trial-by-trial basis according to experimental conditions, and then provides access for users to perform analysis like plotting PSTH and other common application.

Class Graphics

The Class for analyzing data according to experimental conditions. It can analyze spike train, local field potential and behavioral data (e.g. saccade, reaction time) using different displaying methods.

Args

filename (string):
File name (with or without extension)
trial_start_mark (string):
Define the event marker that represents the start of a trial, which is used to separate the raw data into different trials.
comment_expr (string):
This parameter tells the program how experimental condition and parameters are stored in the data.
For example, an experimental condition, patch direction, is stored in the way 'patch_direction:degree'. By setting the comment_expr as "key:value", the program decodes the key as 'patch_direction', and the value for a particular trial is the degree of that trial.
spike_to_load (string or list):

Define the spike channels and units.
If 'all', spikes in all channels and all units will be loaded.
If 'none', spike data will not be loaded.
If set to be a string like 'spike_26_1', the spike of unit 1 in channel 26 will be loaded.
If set to be a list like ['spike_26_1','spike_23_2'], the spike of unit 1 in channel 26 and spike of unit 2 in channel 23 will be loaded.
Default: 'all'

analog_to_load (string or list):
Define the analog signal channels.
if 'all', analog signals in all channels will be loaded.
if 'none', analog signals will not be loaded.
if set to be a string like 'analog_25', analog signals in channel 25 will be loaded.
if set to be a list like ['analog_25','analog_20'], analog signals in channel 25 and channel 20 will be loaded.
Default: 'none'

Returns

-

Examples

>>> gh = Graphics('myfile.h5', '64715',' key:value')

In this example, event marker '64715' is used to separate the raw data into different trials. 'key:value' is used to extract experimental condition information.

This command initiates the Graphics class, and groups all data into a table, wherein each row represents a trial, and each column represents a specific data, e.g. stimulus onset time, offset time, reaction time, spike, LFP, etc. Use the command 'gh.data_df' to display the data table.

def plot_spike(self, channel, sort_by, align_to, pre_time, post_time, bin_size=30, overlap=0, Mean=3, Sigma=10, limit=False, filter_nan=False, fig_marker=[0], fig_size=[12,7], fig_column=4, fig_pad=0.5, fig_wspace=0.02, fig_hspace=0.15, figure=True):

Group data by experimental conditions and plot PSTH and raster of each condition.

Args

channel (string):
Define the spike channel and unit with a dash in between. Example:
chanel_unit

sort_by (list):
Define the conditions used to sort data

align_to (string):
Define the event marker used to align each trial's spikes

pre_time (int):
Set the time(msec) before the align_to to be covered

post_time (int):
Set the Time(msec) after the align_to to be covered

bin_size (int):
Set the bin size (msec) used to calculate PSTH
Default: 30
overlap (int):
Set the overlap (msec) between adjacent bins
Default: 0
Mean (float):
Set the mean of the Gaussian kernel used to smooth the PSTH
Default: 3
Sigma (float):
Set the sigma of the Gaussian kernel used to smooth the PSTH
Default: 10
limit (string):
An expression used to filter the data by certain conditions.
Default: False
filter_nan (list):
Trials with the NaN value in the listed columns will be excluded
Default: False
fig_marker (list):
Define the positions of the reference vertical lines by setting some time points in the list.
Default: [0]
fig_size (list):
Define the size of the figure
Default: [12,7]
fig_column (int):
Define the number of sub-plots in each row
Default: 4
fig_pad (float):
Set the space of padding of the figure
Default: 0.5
fig_wspace (float):
Set the width reserved for blank space between subplots
Default: 0.02
fig_hspace (float):
Set the height reserved for white space between subplots
Default: 0.15
figure (Boolean):
if True, the figure will be displayed.
Default: True

Returns

{'data':{condition_1:PSTH,
condition_2:PSTH,
.

.
.},
'time': firing rate time}

Examples

>>> firingRate = gh.plot_spike(channel = ‘spike_26_1’, sort_by = [‘patch_direction'], align_to = ’dig_64721’, pre_time = -300, post_time = 2000, bin_size = 30, overlap = 10, filter_nan = [‘dig_64721’,’dig_64722'], fig_column = 4, fig_marker = [0])

def plot_spike_count(self, channel, sort_by, align_to, timebin, limit=False, filter_nan=False, figure=True):

sort data by experimental conditions and plot the spike count during a given period.

Args

channel (string):
Define the spike channel and unit with a dash in between. Example: chanel_unit

sort_by (list):
Set the experimental conditions used to sort data

align_to (string):
Set the event marker used to align each trial's spikes

timebin (list):
Define the period for calculating spike counts.

limit (string):
an expression used to filter the data by certain conditions.
Default: False

filter_nan (list):
trials with the NaN value in the listed columns will be excluded
Default: False

figure (Boolean):
if True, the figure will be displayed.
Default: True

Returns

{condition_1: {'mean':value,
'sem':value}
condition_2: {'mean':value,
'sem':value}
'
'
'
}

Examples

>>>spk_count = gh.plot_spike_count(channel = ’spike_26_1’, sort_by = [‘patch_direction’], align_to =’dig_64721’, timebin=[0,700])

def plot_line(self, target, sort_by, limit=False, filter_nan=False):

sort data by experimental conditions and plot scalar data in lineplot (e.g. reaction time)
Args
target (string):
the name of the scalar data to be analyzed
sort_by (list):
experimental conditions used to sort data
limit (string):
an expression used to filter the data by certain conditions.
Default: False
filter_nan (list):
trials with the NaN value in the listed columns will be excluded
Default: False
Returns
{condition_1: {'mean':value,
'sem':value,
'num':value}
condition_2: {'mean':value,
'sem':value,
'num':value}
'
'
'
}
Examples
Reaction_time=gh.plot_line('Reaction_time', sort_by=['a','A'],
limit='Reaction_time<500')

def plot_bar(self, target, sort_by, limit=False, filter_nan=False, ci=95, kind='bar'):
sort data by experimental conditions and plot scalar data in barplot (e.g. reaction time)
Args
target (string):
the name of the scalar data to be analyzed
sort_by (list):
experimental conditions used to sort data
limit (string):
an expression used to filter the data by certain conditions.
Default: False
filter_nan (list):
trials with the NaN value in the listed columns will be excluded
Default: False
ci (float):
confidence interval
Default: 95
kind (str):

Define the plot type, options include ‘bar’, ‘point’
Default: ‘bar’

Returns

{condition_1: {'mean':value,
'sem':value,
'num':value}
condition_2: {'mean':value,
'sem':value,
'num':value}
'
'
'
}

Examples

Reaction_time=gh.plot_line('Reaction_time', sort_by=['a','A'],
limit='Reaction_time<500')

def plot_analog(self, channel, sort_by, align_to, pre_time, post_time, limit=False, filter_nan=False, normalize=True, fig_marker=[0], fig_size=[12,7], fig_column=4):
sort data by experimental conditions and plot analog signals (e.g. LFP)

Args

channel (string):
define the analog channel
sort_by (list):
Define the experimental conditions used to sort data
align_to (string):
Define the event marker used to align each trial's signals
pre_time (int):
Set the time(msec) before the align_to to be covered
post_time (int):
Set the time(msec) after the align_to to be covered
limit (string):
An expression used to filter the data by certain conditions.
Default: False
filter_nan (list):
Trials with the NaN value in the listed columns will be excluded
Default: False
fig_marker (list):
Defines the positions of the reference vertical lines by setting some time points in the list.
Default: [0]
fig_size (list):
The size of the figure
Default: [12,7]

fig_column (int):
Number of sub-plots in one row
Default: 4

Returns
{'time': analog signal time
'data': {'condition_1': signal data,
'condition_2': signal data,
.
.
.
}
}

def plot_spectral(self, channel, sort_by, align_to, pre_time, post_time, limit=False, filter_nan=False, x_lim=[1,100], y_lim=False, log=False, fig_size=[12,7]):
sort data by experimental conditions and plot the spectrum of the analog signals (e.g. LFP)
Args
channel (string):
define the analog channel
sort_by (list):
experimental conditions used to sort data
align_to (string):
event marker used to align each trial's signals
pre_time (int):
set the time(msec) before the align_to to be covered
post_time (int):
set the time(msec) after the align_to to be covered
limit (string):
an expression used to filter the data by certain conditions.
Default: False
filter_nan (list):
trials with NaN value in the listed columns will be excluded
Default: False
x_lim (list):
set limits of x-axis
Default: [0,100]
y_lim (list):
set limits of the y-axis
Default: False
fig_size (list):
the size of the figure
Default: [12,7]
Returns
{'frequency': frequency

'data': {'condition_1':signal data,
'condition_2':signal data,
.
.
.
}
}

def plot_spectrogram(self, channel, sort_by, align_to, pre_time, post_time, limit=False, filter_nan=False, y_lim=[0,100], normalize=True, fig_mark=[0], fig_size=[12,7], fig_column=4):
sort data by experimental conditions and plot spectrogram for analog signals (e.g. LFP)
Args
channel (string):
define the analog channel
sort_by (list):
experimental conditions used to sort data
align_to (string):
event marker used to align each trial' signals
pre_time (int):
set the time(msec) before the align_to to be covered
post_time (int):
set the time(msec) after the align_to to be covered
limit (string):
an expression used to filter the data by certain conditions.
Default: False
filter_nan (list):
trials with NaN value in the listed columns will be excluded
Default: False
y_lim (list):
set limits of y-axis
Default: [0, 100]
fig_mark (list):
draw vertical lines at the time points set in the list.
Default: [0]
fig_size (list):
the size of the figure
Default: [12,7]
fig_column (int):
number of sub-plots in one row
Default: 4
Returns
{'frequency': frequency,
'time': analog signal time,

'data':{
'condition_1': spectrogram value,
'condition_2': spectrogram value,
.
.
.
}
}

def analog_filter (self,channel, band_pass=None, band_stop=None):
filter analog signals
Args
channel (string):
define the analog channel
band_pass (list):
set the frequency range for band pass filtering.
Default: None
band_stop (list):
set the frequency for band-stop filtering.
Default: None
Returns
-

def calibrate_eye(self, eye_channel, realign_mark, realign_timebin, eye_medfilt_win=21, eye_gausfilt_sigma=3):
smooth eye movement trajectory and realign eye position to a relatively stable period of time, e.g. during fixation.
Args
eye_channel (list):
the first element is the channel name for the horizontal eye position
the second element is the channel name for the vertical eye position
realign_mark (string):
event marker used to align eye positions
realign_timebin (list):
a period of time relative to the realign_mark, e.g. [0,100]
eye_medfilt_win (int):
The parameter for the median filter to smooth the eye movement trajectory
eye_gausfilt_sigma (int):
sigma of the Gaussian kernel to smooth the eye movement trajectory
Returns:
-

def find_saccade(self, eye_channel, eye_speed_win=5, sac_speed_threshold=100, sac_duration_threshold=10, sac_displacement_threshold=2):

find all saccades for all trials
Args
    eye_channel (list):
        the first element is the channel name for the horizontal eye position
        the second element is the channel name for the vertical eye position
    eye_speed_wind (int):
        number of points to calculate eye movement speed
    sac_speed_threshold (int):
        set the speed threshold for a valid saccade
        Default: 100 (deg/s)
    sac_duration_threshold (int):
        set the (minimum) duration threshold for a valid saccade.
        Default: 10 (msec)
    sac_displacement_threshold (int):
        set the minimum saccade amplitude for a valid saccade
        Default: 2 (deg)
Returns
  -

def choose_saccade(self, align_to, timebin, ampbin=False):
choose saccades in each trial that happened within a certain period and of certain amplitude
Args
    align_to (string):
        Define the event marker for the zero time point
    timebin (list):
        time period relative to the align_to timestamp
        select saccades happened within the set period
    ampbin (list):
        amplitude range
        select saccades of set amplitude
        Default: False
Returns
  -

def reclaim_space(self,file_name):
reallocate the storage space that the occupied by the file, then release extra storage space.
Args
    file_name (string):
        the name of the work space
Returns
    -

def save_data(self, space_name, data, key, replace=False):
save analysis results to the workspace for population level analysis

Args
space_name (string):
file path of the work space for storing analysis results
data (dict):
analysis results to be stored
key (string):
name the stored results
replace (Boolean):
if True, stored results will be rewritten if their key has already existed.
Returns
-

def add_column(self,name,add_data):
add certain column to the data table
Args
name (string, list):
define the name(s) for the newly added column
add_data (int, float, string, list, pandas.Series, pandas.DataFrame):
if int, float or string, all rows of this new column will be filled with this value
if list, pandas.Series or pandas.DataFrame, their dimensions need to be consistent with the data table
Returns
-

def del_columns(self, columns):
delete certain columns in the data table
Args
columns (string, list):
list the column names to be deleted
Returns
-

def del_trials(self,trials):
delete certain trials in the data table
Args
trials (list):
indices of trials to be deleted
Returns
-

def df_multiply(self, column, multiply_info):
this function multiplies the selected column with certain factor
Args
column (string):

the column name to be played with
multiply_info (string, int, float or pandas.DataFrame):
information to be used for multiplying
Returns
-

def df_division(self, column, division_info):
this function divides the selected column by certain factor
Args
column (string):
the column name to be played with
division_info (string, int, float or pandas.DataFrame):
information to be used for dividing
Returns
-

def df_add(self, column, added_info):
this function performs adding to a given column
Args
column (string):
the column name to be played with
added_info (string, int, float or pandas.DataFrame):
The information to be added to the selected column can be string, int, float, or pandas.DataFrame
Returns
-

def df_minus(self, column, minus_info):
this function performs minus to a given column
Args
column (string):
the column name to be played with
minus_info (string, int, float or pandas.DataFrame):
information to be subtracted from the selected column
Returns
-

def to_numeric(self, columns):
convert data type in certain columns to numeric type
Args
columns (string or list):
column names needed to be converted
Returns
-

def rename(self, names_dict):
rename certain columns
Args
names_dict (dict):
{'old_name_1':'new_name_1',
'old_name_2':'new_name_2',
.
.
.
}
Returns
-

PopuAnalysis

The Module for analyzing data at the population level. This module uses the results stored in the workspace obtained from analyzing single session data.

Class PopuAnalysis
class for analyzing data at the population level.
Args
filename (string):
file name of the workspace (with extension)
Returns
-
Examples
>>> PopuAnalysis('test_workspace.h5')
initiate PopuAnalysis class

def plot_spike(self, store_key, conditions, normalize=False, fig_mark=[0], line_style=False, x_lim=False, y_lim=False, err_style='ci_band', ci=68):
plot average PSTH among neuronal population
Args
store_key (string):
define which data to be analyzed in the workspace
conditions (list):
define spikes of which experimental conditions will be plotted
normalize (Boolean, list):
if True, min-max normalization will be used among all conditions
if False, no normalization
if list, min-max normalization will be used among conditions in the list
Default: False
fig_mark (list):

draw vertical lines at the time points in the list.
Default: [0]

line_style (list):
line style used for plotting.
the length of the line_style list must equal the length of the conditions
Default: False, automatically use line styles for different lines

x_lim (list):
set limits of the x-axis

Default: False

y_lim (list):
set limits of the y-axis
Default: False

err_style (string):
set how to plot the uncertainty across units, select from {ci_band, ci_bars, boot_traces, boot_kde, unit_traces, unit_points}
Default: ci_band

ci (int):
confidence interval.
Default: 68

Returns

{'data': {condition_1:PSTH,
condition_2:PSTH,
.
.
.
},
'time':firing rate time
}

def plot_spectrogram(self,store_key,condition,fig_mark=[0],y_lim=[0,100]):
plot spectrogram of analog signals (e.g. LFP) at population level

Args

store_key (string):
define which data to be analyzed in the workspace

condition (string):
define which conditions will be plotted

fig_mark (list):
draw vertical lines at the time points in the list.
Default: [0]

y_lim (list):
set limits of y-axis
Default: [0,100]

Returns

{'frequency': frequency,

'time': analog signal time,
'data': spectrogram value}

def plot_line(self, store_key, conditions):
plot scalar data (e.g. reaction time) in population level
Args
store_key (string):
define which data to be analyzed in the workspace
conditions (list):
define which conditions will be plotted
Returns
{'mean':{'condition_1':value,
'condition_2':value,
.
.
.
},
'sem':{'condition_1':value,
'condition_2':value,
.
.
.
}
}

def plot_bar(self, store_key, conditions, ci=95, kind='bar'):
plot scalar data (e.g. reaction time) at population level
Args
store_key (string):
define which data to be analyzed in the workspace
conditions (list):
define which conditions to be plotted
ci (fload):
confidence interval
Default: 95
kind (str):
Define the type of of plot. Options: 'bar', 'point'
Default: 'bar'
Returns
{'mean':{'condition_1':value,
'condition_2':value,
.
.
.

},
'sem':{'condition_1':value,
'condition_2':value,
.
.
.
}
}

def plot_analog(self, store_key, conditions, line_style=False, fig_mark=[0], x_lim=False, y_lim=False):
plot analog signals (e.g. LFP) at population level
Args
store_key (string):
define which data to be analyzed in the workspace
conditions (list):
define which conditions will be plotted
line_style (list):
line style used for plotting.
the length of the line_style list must equal the length of conditions
Default: False, automatically use line styles for different lines
fig_mark (list):
draw vertical lines at the time points in the list.
Default: [0]
x_lim (list):
set limits of y-axis
Default: False
y_lim (list):
set limits of y-axis
Default: False
Returns
{'time':time,'data':{'condition_1':mean signal data,
'condition_2':mean signal data,
.
.
.
}
}

def plot_spectral(self, store_key, conditions, line_style=False, x_lim=[0,100], log=False):
plot spectrum of analog signals (e.g. LFP) in population level
Args
store_key (string):
define which data to be analyzed in the workspace

conditions (list):
define which conditions will be plotted
line_style (list):
line style used for plotting.
the length of the line_style list must equal the length of conditions
Default: False, automatically use line styles for different lines
x_lim (list):
set limits of y-axis
Default: [0,100]
log (Boolean):
if True, y-axis will use logarithmic axis
Default: False

Returns

{'frequency':frequency,
'data':{'condition_1':mean signal data,
'condition_2':mean signal data,
.
.
.
}
}

def stats_ttest_rel(self, store_key, cond_1, cond_2):
paired t-test, only used for scalar values

Args

store_key (string):
define which data to be analyzed in the workspace
cond_1 (string), cond_2 (string):
compare these two conditions using paired t-test

Returns

pvalue

def stats_ttest_ind(self, store_key, cond_1, cond_2):
# t-test, only used for scalar values

Args

store_key (string):
define which data to be analyzed in the workspace
cond_1 (string), cond_2 (string):
compare these two conditions using t-test

Returns

pvalue

def stats_ttest_1samp(self, store_key, cond, compare_value):
calculate t-test for the mean of one group of scores, only used for scalar values

Args
store_key (string):
define which data to be analyzed in the workspace
cond (string):
sample observation
compare_value (float):
expected value in the null hypothesis
Returns
pvalue

def stats_desc(self, store_key, cond):
descriptive statistics, only used for scalar values
Args
store_key (string):
define which data to be analyzed in the workspace
cond (string):
sample observation
Returns
descriptive statistics

def stats_anova_oneway(self, store_key, conditions):
one way ANOVA, only used for scalar values
Args
store_key (string):
define which data to be analyzed in the workspace
conditions (list):
list of experimental condition
Returns
pvalue

def stats_anova_twoway(self, store_key, conditions):
two-way ANOVA, only used for scalar values
Args
store_key (string):
define which data to be analyzed in the workspace
conditions (list):
list of experimental condition
Returns
pvalue

def close(self):
close the work space