CalSciPy

Release 0.3.5

Darik A. O'Neil

2023

CONTENTS:

CHAPTER

ONE

INTRODUCTION

CalSciPy contains a variety of useful methods for handling, processing, and visualizing calcium imaging data. It's intended to be a collection of useful, well-documented functions often used in boilerplate code alongside software packages such as Caiman, SIMA, and Suite2P.

1.1 Motivation

I noticed I was often re-writing or copy/pasting a lot of code between environments when working with calcium imaging data. I started this package so I don't have to so you don't have to. No more wasting time writing 6 lines to simply preview your tiff stack, extract a particular channel, or bin some spikes. No more vague exceptions or incomplete documentation when re-using a hastily-made function from 2 months ago. Alongside these time-savers, I've also included some more non-trivial methods that are particularly useful.

1.2 Limitations

The current distribution for the package is incomplete and partially tested. There may be breaking changes between versions.

CHAPTER

TWO

INSTALLATION

2.1 Full Install

Enter `pip install CalSciPy` in your terminal.

2.2 GPU Installation

An installation of CuPy & CUDA are required for gpu-parallelized functions

CHAPTER

THREE

CALSCIPY.BRUKER PACKAGE

3.1 Subpackages

3.1.1 CalSciPy.bruker.xml_mappings package

Submodules

CalSciPy.bruker.xml_mappings.xml_mapping module

CalSciPy.bruker.xml_mappings.xml_mapping.**load_mapping**(*version: str*) → mappingproxy

Load mapping of prairieview xml objects to their respective pyprairieview objects from .json

Parameters
version (*str*) – version of prairieview

Returns
read-only mapping the xml tag and python object

Return type
MappingProxyType

CalSciPy.bruker.xml_mappings.xml_mapping.**write_mapping**(*mapping: dict*, *version: str*) → None

Write mapping of prairieview xml objects to their respective pyprairieview objects to .json

Parameters

- **mapping** (*dict*) – dictionary mapping the xml tag and python object
- **version** (*str*) – version of prairieview

Return type
None

Module contents

3.2 Submodules

3.2.1 CalSciPy.bruker.configuration_values module

3.2.2 CalSciPy.bruker.data module

3.2.3 CalSciPy.bruker.factories module

3.2.4 CalSciPy.bruker.meta_objects module

class CalSciPy.bruker.meta_objects.**GroupMeta**

Bases: object

class CalSciPy.bruker.meta_objects.**PhotostimulationMeta**(*root: ElementTree*, *factory: object*, *width: int = 512*, *height: int = 512*)

Bases: _BrukerMeta

class CalSciPy.bruker.meta_objects.**ROIMeta**(**args*)

Bases: object

ROI Object

generate_coordinates(*width: int*, *height: int*) → Tuple[float, float]

Converts the normalized coordinates to image coordinates

Parameters

- **width** (int) – width of image
- **height** (int) – height of image

Return type
Tuple[float, float]

Returns
x,y coordinates

generate_hull_vertices() → Tuple[Tuple[int, int]]

Identifies the vertices of the Convex-Hull approximation

Return type
Tuple[Tuple[int, int]]

Returns
vertices (Nx2)

generate_mask(*width: int*, *height: int*) → Tuple[Tuple[int, int]]

Converts spiral center & radii to a coordinate mask

Parameters

- **width** (int) – width of image
- **height** (int) – height of image

Return type
Tuple[Tuple[int, int]]

Returns
coordinate mask (y, x)

3.2.5 CalSciPy.bruker.prairie_link module

3.2.6 CalSciPy.bruker.protocols module

3.2.7 CalSciPy.bruker.validation module

exception CalSciPy.bruker.validation.**DingusException**(*message: str*)

Bases: Exception

class CalSciPy.bruker.validation.**DingusLogger**(*exceptions: List[Exception] | None = None*)

Bases: object

add_exception(*other: Exception*) → *CalSciPy.bruker.validation.DingusLogger*

Return type
DingusLogger

raise_exceptions() → *CalSciPy.bruker.validation.DingusLogger*

Return type
DingusLogger

CalSciPy.bruker.validation.**field_validator**(*key: str*, *value: Any*, *var: dataclasses.Field*) → List[Exception]

Return type
List[Exception]

CalSciPy.bruker.validation.**format_fields**(*var: object*) → tuple

Return type
tuple

CalSciPy.bruker.validation.**type_check_nested_types**(*var: Any*, *expected: str*) → bool

Checks type of nested types. WORKS FOR ONLY ONE NEST.

Parameters

- **var** (*Any*) – variable to check
- **expected** (*str*) – expected type

Returns
boolean type comparison

Return type
bool

CalSciPy.bruker.validation.**validate_fields**(*data_class: object*) → bool

Return type
bool

3.2.8 CalSciPy.bruker.xml_objects module

class CalSciPy.bruker.xml_objects.**GalvoPoint**(*, *x: float = 0.0*, *y: float = 0.0*, *name: str = 'Point 0'*, *index: int = 0*, *activity_type: str = 'MarkPoints'*, *uncaging_laser: str = 'Uncaging'*, *uncaging_laser_power: int = 0*, *duration: float = 100.0*, *is_spiral: bool = True*, *spiral_size: float = 0.0*, *spiral_revolutions: float = 0.0*, *z: float = 0.0*) → None

Bases: _BrukerObject

Dataclass for a specific point during galvo-stimulation for a specific marked point in a sequence of photostimulations

Variables
z – relative z-position of the motor + ETL offset (um)

activity_type: str = 'MarkPoints'

duration: float = 100.0

index: int = 0

is_spiral: bool = True

name: str = 'Point 0'

spiral_revolutions: float = 0.0

spiral_size: float = 0.0

uncaging_laser: str = 'Uncaging'

uncaging_laser_power: int = 0

x: float = 0.0

y: float = 0.0

z: float = 0.0

class CalSciPy.bruker.xml_objects.**GalvoPointElement**(*, *initial_delay: int = 0*, *inter_point_delay: float = 0.0*, *duration: float = 0*, *spiral_revolutions: float = 0*, *all_points_at_once: bool = False*, *points: str = 'Point 0'*, *indices: int = 0*) → None

Bases: _BrukerObject

Dataclass for a specific galvo-stimulation for a specific marked point in a sequence of photostimulations

all_points_at_once: bool = False

bool: whether to do all points at once

duration: float = 0

int: duration of stimulation in ms

indices: int = 0

int: index from galvo point list

initial_delay: int = 0

int: initial delay for stimulation

inter_point_delay: float = 0.0

float: inter point delay

points: str = 'Point 0'

str: id from galvo point list

spiral_revolutions: float = 0

int: number of spiral revolutions

class CalSciPy.bruker.xml_objects.**GalvoPointList**(*, *galvo_points: Tuple[object]*) → None

Bases: _BrukerObject

Dataclass for a list of galvo points

galvo_points: Tuple[object]

class CalSciPy.bruker.xml_objects.**MarkPointElement**(*, *points: Tuple[object]*, *repetitions: int = 1*, *uncaging_laser: str = 'Uncaging'*, *uncaging_laser_power: int = 0*, *trigger_frequency: str = 'None'*, *trigger_selection: str = 'PFI0'*, *trigger_count: int = 0*, *async_sync_frequency: str = 'FirstRepetition'*, *voltage_output_category_name: str = 'None'*, *voltage_rec_category_name: str = 'Current'*, *parameter_set: str = 'CurrentSettings'*) → None

Bases: _BrukerObject

Dataclass for a specific marked point in a sequence of photostimulations

async_sync_frequency: str = 'FirstRepetition'

str: sync

parameter_set: str = 'CurrentSettings'

str: id of parameter set

points: Tuple[object]

object: Tuple of galvo point elements

repetitions: int = 1

int: repetitions of this stimulation event

trigger_count: int = 0

int: number of triggers

trigger_frequency: str = 'None'

str: trigger frequency

trigger_selection: str = 'PFI0'

uncaging_laser: str = 'Uncaging'

str: identity of uncaging laser

uncaging_laser_power: int = 0

int: uncaging laser power

voltage_output_category_name: str = 'None'

str: name of voltage output experiment

voltage_rec_category_name: str = 'Current'

str: name of voltage recording experiment

class CalSciPy.bruker.xml_objects.**MarkPointSeriesElements**(*, *marks: Tuple[object]*, *iterations: int = 1*, *iteration_delay: float = 0.0*, *calc_funct_map: bool = False*) → None

Bases: _BrukerObject

Dataclass for a sequence of photostimulations

calc_funct_map: bool = False

bool: whether to calculate functional map

iteration_delay: float = 0.0

float: delay between each series iteration (ms)

iterations: int = 1

int: number of times this series is iterated

marks: Tuple[object]

Tuple[object]: series of mark point elements

class CalSciPy.bruker.xml_objects.**Point**(*, *index: int = 0*, *x: float = 0.0*, *y: float = 0.0*, *is_spiral: bool = True*, *spiral_width: float = 0.0*, *spiral_height: float = 0.0*, *spiral_size_in_microns: float = 0.0*) → None

Bases: _BrukerObject

Dataclass for a specific point during galvo-stimulation for a specific marked point in a sequence of photostimulations

index: int = 0

str: 1-order index in galvo point list

is_spiral: bool = True

bool: boolean indicating whether point is spiral

spiral_height: float = 0.0

float: height of spiral

spiral_size_in_microns: float = 0.0

float: size of spiral in microns

spiral_width: float = 0.0

float: width of spiral

x: float = 0.0

float: normalized x position

y: float = 0.0

float: normalized y position

3.3 Module contents

CHAPTER

FOUR

CALSCIPY.EVENT_PROCESSING MODULE

CalSciPy.event_processing.**bin_data**(*data: pandas.core.frame.DataFrame | numpy.ndarray | Iterable*, *bin_length: int*, *fun: Callable*) → pandas.core.frame.DataFrame | numpy.ndarray

Return type
Union[DataFrame, ndarray]

CalSciPy.event_processing.**calculate_firing_rates**(*spike_probability_matrix: numpy.ndarray*, *frame_rate: float = 30.0*, *in_place: bool = False*) → numpy.ndarray

Calculate firing rates

Parameters

- **spike_probability_matrix** (ndarray) – matrix of n neuron x m samples where each element is the probability of a spike
- **frame_rate** (float, default: 30.0) – frame rate of dataset
- **in_place** (bool, default: False) – boolean indicating whether to perform calculation in-place

Return type
ndarray

Returns
firing matrix of n neurons x m samples where each element is a binary indicating presence of spike event

CalSciPy.event_processing.**calculate_mean_firing_rates**(*firing_matrix: numpy.ndarray*) → numpy.ndarray

Calculate mean firing rate

Parameters
firing_matrix (ndarray) – matrix of n neuron x m samples where each element is either a spike or an

instantaneous firing rate

Return type
ndarray

Returns
1-D vector of mean firing rates

CalSciPy.event_processing.**collect_waveforms**(*traces: numpy.ndarray*, *event_indices: Iterable[Iterable[int]]*, *pre: int = 150*, *post: int = 450*) → Tuple[numpy.ndarray]

Collect waveforms for each event

Parameters

- **traces** (ndarray) – a matrix of M neurons x N samples
- **event_indices** (Iterable[Iterable[int]]) – a list of events
- **pre** (int, default: 150) – number of pre-event frames
- **post** (int, default: 450) – number of post-event frames

Return type
Tuple[ndarray]

Returns
a matrix of M events x N samples

CalSciPy.event_processing.**convert_tau**(*tau: float*, *dt: float*) → float

Converts a discrete tau to a continuous tau

Parameters

- **tau** (float) – decay constant
- **dt** (float) – time step (s)

Return type
float

Returns
continuous tau (s)

CalSciPy.event_processing.**get_event_onset_intensities**(*traces: numpy.ndarray*, *event_indices: Iterable[Iterable[int]]*) → Tuple[numpy.ndarray]

Retrieve the signal intensity at event onset for each neuron in the event indices

Parameters

- **traces** (ndarray) – An M neuron by N sample matrix
- **event_indices** (Iterable[Iterable[int]]) – An iterable of length M containing a sequence with a duration for each event

Return type
Tuple[ndarray]

Returns
An iterable of length M neurons containing the onset intensities for each event in the sequence

CalSciPy.event_processing.**get_inter_event_intervals**(*event_indices: Iterable[Iterable[int]]*, *frame_rate: float = 30.0*) → Tuple[numpy.ndarray]

Calculate the inter event intervals for each neuron in the event indices

Parameters

- **event_indices** (Iterable[Iterable[int]]) – An iterable of length M containing a sequence with a duration for each event

- **frame_rate** (float, default: 30.0) – frame_rate for trace matrix

Return type
Tuple[ndarray]

Returns
An iterable of length M neurons containing the inter-event intervals for each event in the sequence

CalSciPy.event_processing.**get_num_events**(*event_indices: Iterable[Iterable[int]]*) → numpy.ndarray

Determines the number of events for each neuron in the event indices

Parameters
event_indices (Iterable[Iterable[int]]) – An iterable of length M neurons containing a sequence with a duration for each event

Return type
ndarray

Returns
A 1-D vector of length M neurons containing the number of events for each neuron

CalSciPy.event_processing.**identify_events**(*traces: numpy.ndarray*, *timeout: int = 15*, *frame_rate: float = 30.0*, *smooth: bool = True*, *force_nonneg: bool = True*) → Tuple[List[int]]

Identify event onset for each neuron using the smoothed, non-negative first-time derivative. The threshold for noise is considered 1/2th the standard deviation of the derivative.

Parameters

- **traces** (ndarray) – An M neuron by N sample matrix
- **timeout** (int, default: 15) – timeout distance for peak finding (frames)
- **frame_rate** (float, default: 30.0) – frame rate / time step for trace matrix
- **smooth** (bool, default: True) – boolean indicating whether to smooth first-time derivative
- **force_nonneg** (bool, default: True) – boolean indicating whether to enforce non-negativity constraint on first-time derivative

Return type
Tuple[List[int]]

Returns
An iterable where each element contains a sequence of frames identified as event onsets

CalSciPy.event_processing.**normalize_firing_rates**(*firing_matrix: numpy.ndarray*, *in_place: bool = False*) → numpy.ndarray

Normalize firing rates by scaling to a max of 1.0. Non-negativity constrained.

Parameters

- **firing_matrix** (ndarray) – matrix of n neuron x m samples where each element is either a spike or an instantaneous firing rate
- **in_place** (bool, default: False) – boolean indicating whether to perform calculation in-place

Return type
ndarray

Returns
normalized firing rate matrix of n neurons x m samples

CalSciPy.event_processing.**scale_waveforms**(*waveforms: typing.Iterable[numpy.ndarray], scaler: typing.Callable = <class 'sklearn.preprocessing._data.StandardScaler'>*) → numpy.ndarray

Scale waveforms for cross-neuron comparisons

Parameters

- **waveforms** (Iterable[ndarray]) – An Iterable of M events by N samples matrices of waveforms
- **scaler** (Callable, default: <class 'sklearn.preprocessing._data.StandardScaler'>) – sklearn preprocessing object

Return type
ndarray

Returns
An Iterable of M event by N samples scaled matrices of waveforms

CHAPTER

FIVE

CALSCIPY.IMAGE_PROCESSING MODULE

CalSciPy.image_processing.**gaussian_filter**(*images: np.ndarray*, *sigma: Number | np.ndarry = 1.0*, *block_size: int = None*, *block_buffer: int = 0*, *in_place: bool = False*) → np.ndarray

GPU-parallelized multidimensional gaussian filter. Optional arguments for in-place calculation. Can be calculated blockwise with overlapping or non-overlapping blocks.

Designed for use on arrays larger than the available memory capacity.

Footprint is of the form np.ones((frames, y pixels, x pixels)) with the origin in the center

Parameters

- **images** – images stack to be filtered
- **sigma** (default: `1.0`) – sigma for gaussian filter
- **block_size** (default: `None`) – the size of each block. Must fit within memory
- **block_buffer** (default: `0`) – the size of the overlapping region between block
- **in_place** (default: `False`) – whether to calculate in-place

Returns

images: numpy array (frames, y pixels, x pixels)

CalSciPy.image_processing.**median_filter**(*images: numpy.ndarray*, *mask: numpy.ndarray = array([[[1., 1., 1.], [1., 1., 1.], [1., 1., 1.]], [[1., 1., 1.], [1., 1., 1.], [1., 1., 1.]], [[1., 1., 1.], [1., 1., 1.], [1., 1., 1.]]])*, *block_size: int | None = None*, *block_buffer: int = 0*, *in_place: bool = False*) → numpy.ndarray

GPU-parallelized multidimensional median filter. Optional arguments for in-place calculation. Can be calculated blockwise with overlapping or non-overlapping blocks.

Designed for use on arrays larger than the available memory capacity.

Footprint is of the form np.ones((frames, y pixels, x pixels)) with the origin in the center

Parameters

- **images** (ndarray) – images stack to be filtered
- **mask** (ndarray, default:

  [[1., 1., 1.], [1., 1., 1.], [1., 1., 1.]],

  [[1., 1., 1.], [1., 1., 1.], [1., 1., 1.]])) – mask of the median filter
- **block_size** (Optional[int], default: `None`) – the size of each block. Must fit within memory

- **block_buffer** (int, default: 0) – the size of the overlapping region between block
- **in_place** (bool, default: False) – whether to calculate in-place

Return type
ndarray

Returns
images: numpy array (frames, y pixels, x pixels)

CHAPTER

SIX

CALSCIPY.INTERACTIVE_VISUALS MODULE

CalSciPy.interactive_visuals.**plot_spikes**(*spike_prob: numpy.ndarray | None = None*, *spike_times: numpy.ndarray | None = None*, *traces: numpy.ndarray | None = None*, *frame_rate: float | None = None*, *y_label: str = 'f/f0'*) → None

Function to interactively visualize spike inference

Parameters

- **spike_prob** (Optional[ndarray], default: None) –
- **spike_times** (Optional[ndarray], default: None) –
- **traces** (Optional[ndarray], default: None) –
- **frame_rate** (Optional[float], default: None) –
- **y_label** (str, default: 'f/f0') –

Return type
None

CalSciPy.interactive_visuals.**plot_traces**(*traces: numpy.ndarray*, *frame_rate: float | None = None*, *y_label: str = 'f/f0'*, *mode: str = 'overlay'*) → None

Parameters

- **traces** (ndarray) –
- **frame_rate** (Optional[float], default: None) –
- **y_label** (str, default: 'f/f0') –
- **mode** (str, default: 'overlay') –

Return type
None

Returns

CalSciPy.interactive_visuals.**plot_trials**(*data: numpy.ndarray*, *trials: numpy.ndarray*, *trial_conditions: None*, *bin_duration: float | None = None*, *y_label: str = 'Firing Rate (Hz)'*) → None

Parameters

- **data** (ndarray) –
- **trials** (ndarray) –
- **trial_conditions** (None) –

- **bin_duration** (Optional[float], default: None) –
- **y_label** (str, default: 'Firing Rate (Hz)') –

Return type
None

Returns

CHAPTER

SEVEN

CALSCIPY.IO_TOOLS MODULE

CalSciPy.io_tools.**load_binary**(*path: str | pathlib.Path*, *mapped: bool = False*, *mode: str = 'r+'*, *missing_metadata: Mapping | None = None*) → numpy.ndarray | numpy.memmap

This function loads images saved in language-agnostic binary format. Ideal for optimal read/write speeds and highly-robust to corruption. However, the downside is that the images and their metadata are split into two separate files. Images are saved with the *.bin* extension, while metadata is saved with extension *.json*. If for some reason you lose the metadata, you can still load the binary if you know three of the following: number of frames, y-pixels, x-pixels, and the datatype (`numpy.dtype`)

Parameters

- **path** (`Union`[`str`, `Path`]) – folder containing binary file
- **mapped** (`bool`, default: `False`) – boolean indicating whether to load image using memory-mapping
- **mode** (`str`, default: `'r+'`) – indicates the level of access permitted to the original binary
- **missing_metadata** (`Optional`[`Mapping`], default: `None`) – if you have lost the metadata or otherwise wish to manually provide it

Return type
`Union`[`ndarray`, `memmap`]

Returns
image (frames, y-pixels, x-pixels)

CalSciPy.io_tools.**load_images**(*path: str | pathlib.Path*) → numpy.ndarray

Load images into a numpy array. If path is a folder, all .tif files found non-recursively in the directory will be compiled to a single array.

Parameters
path (`Union`[`str`, `Path`]) – a file containing images or a folder containing several imaging stacks

Return type
`ndarray`

Returns
numpy array (frames, y-pixels, x-pixels)

CalSciPy.io_tools.**save_binary**(*path: str | pathlib.Path*, *images: numpy.ndarray*, *name: str = 'binary_video'*) → int

Save images to language-agnostic binary format. Ideal for optimal read/write speeds and highly-robust to corruption. However, the downside is that the images and their metadata are split into two separate files. Images are saved with the *.bin* extension, while metadata is saved with extension *.json*. If for some reason you lose the metadata, you can still load the binary if you know three of the following: number of frames, y-pixels, x-pixels, and the

datatype. The datatype is almost always unsigned 16-bit (`numpy.uint16`) for all modern imaging systems–even if they are collected at 12 or 13-bit.

Parameters
path (Union[str, Path]) – path to save images to. The path stem is considered the filename if it doesn't have any extension. If

no filename is provided then the default filename is *binary_video*.

Parameters

- **images** (ndarray) – images to save (frames, y-pixels, x-pixels)
- **name** (str, default: 'binary_video') – specify filename for produced files

Return type
int

Returns
0 if successful

CalSciPy.io_tools.**save_images**(*path: str | pathlib.Path*, *images: numpy.ndarray*, *name: str = 'images'*, *size_cap: float = 3.9*) → int

Save a numpy array to a single .tif file. If size > 4GB then saved as a series of files. If path is not a file and already exists the default filename will be *images*.

Parameters

- **path** (Union[str, Path]) – filename or absolute path
- **images** (ndarray) – numpy array (frames, y pixels, x pixels)
- **name** (str, default: 'images') – filename for saving images
- **size_cap** (float, default: 3.9) – maximum size per file

Return type
int

Returns
returns 0 if successful

CHAPTER

EIGHT

CALSCIPY.MISC MODULE

class CalSciPy.misc.**PatternMatching**(*value: Any*, *comparison_expressions: Iterable[Any]*)

Bases: object

CalSciPy.misc.**calculate_frames_per_file**(*y_pixels: int*, *x_pixels: int*, *bit_depth: numpy.dtype = <class 'numpy.uint16'>*, *size_cap: numbers.Number = 3.9*) → int

Estimates the number of image frames to allocate to each file given some maximum size.

Parameters

- **y_pixels** (int) – number of y_pixels in image
- **x_pixels** (int) – number of x_pixels in image
- **bit_depth** (dtype, default: <class 'numpy.uint16'>) – bit-depth / type of image elements
- **size_cap** (Number, default: 3.9) – maximum file size

Return type
int

Returns
the maximum number of frames to allocate for each file

CalSciPy.misc.**generate_blocks**(*sequence: Iterable*, *block_size: int*, *block_buffer: int = 0*) → Iterator

Returns a generator of some arbitrary iterable sequence that yields m blocks with overlapping regions of size n

Parameters

- **sequence** (Iterable) – Sequence to be split into overlapping blocks
- **block_size** (int) – size of blocks
- **block_buffer** (int, default: 0) – size of overlap between blocks

Return type
Iterator

Returns
generator yielding m blocks with overlapping regions of size n

CalSciPy.misc.**generate_overlapping_blocks**(*sequence: Iterable*, *block_size: int*, *block_buffer: int*) → Iterator

Returns a generator of some arbitrary iterable sequence that yields m blocks with overlapping regions of size n

Parameters

- **sequence** (Iterable) – Sequence to be split into overlapping blocks

- **block_size** (int) – size of blocks
- **block_buffer** (int) – size of overlap between blocks

Return type
Iterator

Returns
generator yielding m blocks with overlapping regions of size n

CalSciPy.misc.**generate_padded_filename**(*output_folder: pathlib.Path*, *index: int*, *base: str = 'images'*, *digits: int = 2*, *ext: str = '.tif'*) → pathlib.Path

Generates a pathlib Path whose name is defined as '{base}_{index}{ext}' where index is zero-padded if it is not equal to the number of digits

Parameters

- **output_folder** (Path) – folder that will contain file
- **index** (int) – index of file
- **base** (str, default: 'images') – base tag of file
- **digits** (int, default: 2) – number of digits for representing index
- **ext** (str, default: '.tif') – file extension

Return type
Path

Returns
generated filename

CalSciPy.misc.**generate_sliding_window**(*sequence: Iterable*, *window_length: int*, *step_size: int = 1*) → numpy.ndarray

Return type
ndarray

CalSciPy.misc.**generate_time_vector**(*num_samples: int*, *sampling_frequency: numbers.Number = 30.0*, *start: numbers.Number = 0.0*, *step: numbers.Number | None = None*) → numpy.ndarray

Generates a time vector for a number of samples collected at either

Parameters

- **num_samples** (int) –
- **sampling_frequency** (Number, default: 30.0) –
- **start** (Number, default: 0.0) –
- **step** (Optional[Number], default: None) –

Return type
ndarray

Returns

CalSciPy.misc.**sliding_window**(*sequence: numpy.ndarray*, *window_length: int*, *function: Callable*, **args*, ***kwargs*) → numpy.ndarray

Return type
ndarray

CalSciPy.misc.**wrap_cupy_block**(*cupy_function: Callable*) → Callable

Wraps a cupy function such that incoming numpy arrays are converting to cupy arrays and swapped back on return

Parameters
cupy_function (Callable) – any cupy function that accepts numpy arrays

Return type
Callable

Returns
wrapped function

CalSciPy.misc.**zero_pad_num_to_string**(*idx: int*, *num_zeros: int*) → str

Return type
str

CHAPTER

NINE

CALSCIPY.REORGANIZATION MODULE

CalSciPy.reorganization.**generate_raster**(*event_frames: Iterable[Iterable[int]]*, *total_frames: int | None = None*) → numpy.ndarray

Generate raster from an iterable of iterables containing the spike or event times for each neuron

Parameters

- **event_frames** (Iterable[Iterable[int]]) – iterable containing an iterable identifying the event frames for each neuron
- **total_frames** (Optional[int], default: None) – total number of frames

Return type
ndarray

Returns
event matrix of neurons x total frames

CalSciPy.reorganization.**generate_tensor**(*traces_as_matrix: numpy.ndarray*, *chunk_size: int*) → numpy.ndarray

Generates a tensor given chunk / trial indices

Parameters

- **traces_as_matrix** (ndarray) – traces in matrix form (neurons x frames)
- **chunk_size** (int) – size of each chunk

Return type
ndarray

Returns
traces as a tensor of trial x neurons x frames

CalSciPy.reorganization.**merge_factorized_matrices**(*factorized_traces: numpy.ndarray*, *components: int | Iterable[int] = 0*) → numpy.ndarray

Concatenate a neuron x chunk or trial array in which each element is a component x frame factorization of the original trace:

Parameters

- **factorized_traces** (ndarray) – neurons x chunks (trial, tif, etc) containing the neuron's trace factorized into several components
- **components** (Union[int, Iterable[int]], default: 0) – specific component to extract

Return type
ndarray

Returns
traces of specific component in matrix form

CalSciPy.reorganization.**merge_tensor**(*traces_as_tensor: numpy.ndarray*) → numpy.ndarray

Concatenate multiple trials or tiffs into single matrix:

Parameters
traces_as_tensor (ndarray) – chunk (trial, tif, etc) x neurons x frames

Return type
ndarray

Returns
traces in matrix form (neurons x frames)

CHAPTER

TEN

CALSCIPY.TRACE_PROCESSING MODULE

CalSciPy.trace_processing.**calculate_dfof**(*traces: numpy.ndarray*, *frame_rate: float = 30.0*, *in_place: bool = False*, *offset: float = 0.0*, *external_reference: numpy.ndarray | None = None*, *method: str = 'baseline'*) → numpy.ndarray

Return type
ndarray

CalSciPy.trace_processing.**calculate_standardized_noise**(*fold_fluorescence_over_baseline: numpy.ndarray*, *frame_rate: float = 30.0*) → numpy.ndarray

Calculates a frame-rate independent standardized noise as defined as:

$$v = \frac{\sigma \frac{\Delta F}{F}}{\sqrt{f}}$$

It is robust against outliers and approximates the standard deviation of f/f0 baseline fluctuations. For comparison, the more exquisite of the Allen Brain Institute's public datasets are approximately 1*%Hz^(-1/2)

Parameters

- **fold_fluorescence_over_baseline** (ndarray) – fold fluorescence over baseline (i.e., f/f0)
- **frame_rate** (float, default: 30.0) – frame rate of dataset

Return type
ndarray

Returns
standardized noise (units are 1*%Hz^(-1/2)) for each neuron

CalSciPy.trace_processing.**detrend_polynomial**(*traces: numpy.ndarray*, *in_place: bool = False*) → numpy.ndarray

Detrend traces using a fourth-order polynomial

Parameters

- **traces** (ndarray) – matrix of traces in the form of neurons x frames
- **in_place** (bool, default: False) – boolean indicating whether to perform calculation in-place

Return type
ndarray

Returns
detrended traces

CalSciPy.trace_processing.**perona_malik_diffusion**(*traces: numpy.ndarray*, *iters: int = 25*, *kappa: float = 0.15*, *gamma: float = 0.25*, *sigma: float = 0*, *in_place: bool = False*) → numpy.ndarray

Edge-preserving smoothing using perona malik diffusion. This is a non-linear smoothing technique that avoids the temporal distortion introduced onto traces by standard gaussian smoothing.

The parameter *kappa* controls the level of smoothing ("diffusion") as a function of the derivative of the trace (or "gradient" in the case of 2D images where this algorithm is often used). This function is known as the diffusion coefficient. When the derivative for some portion of the trace is low, the algorithm will encourage smoothing to reduce noise. If the derivative is large like during a burst of activity, the algorithm will discourage smoothing to maintain its structure. Here, the argument *kappa* is multiplied by the dynamic range to generate the true kappa.

The diffusion coefficient implemented here is e^(-(derivative/kappa)^2).

Perona-Malik diffusion is an iterative process. The parameter *gamma* controls the rate of diffusion, while parameter *iters* sets the number of iterations to perform.

This implementation is currently situated to handle 1-D vectors because it gives us some performance benefits.

Parameters

- **traces** (ndarray) – matrix of M neurons by N samples
- **iters** (int, default: 25) – number of iterations
- **kappa** (float, default: 0.15) – used to calculate the true kappa, where true kappa = kappa * dynamic range. range 0-1
- **gamma** (float, default: 0.25) – rate of diffusion for each iter. range 0-1
- **in_place** (bool, default: False) – whether to calculate in-place

Return type
ndarray

Returns
smoothed traces

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C