CalSciPy

Release 0.3.1

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 MODULE

CalSciPy.bruker.**align_data**(*analog_data: pandas.core.frame.DataFrame*, *frame_times: pandas.core.frame.DataFrame*, *fill: bool = False*) → pandas.core.frame.DataFrame

Synchronizes analog data & imaging frames using the timestamp of each frame. Option to generate a second column in which the frame index is interpolated such that each analog sample matches with an associated frame.

Parameters

- **analog_data** (`DataFrame`) – analog data
- **frame_times** (`DataFrame`) – frame timestamps
- **fill** (`bool`, default: `False`) – whether to include an interpolated nearest-neighbor column so each sample has an associated frame

Return type

`DataFrame`

Returns

a dataframe containing time (index, ms) with aligned columns of voltage recordings/analog data and imaging frame

CalSciPy.bruker.**determine_imaging_content**(*folder: str | pathlib.Path*) → Tuple[int, int, int, int, int]

This function determines the number of channels and planes within a folder containing .tif files exported by Bruker's Prairieview software. It also determines the size of the images (frames, y-pixels, x-pixels). It's a quick / fast alternative to parsing its respective xml. However, note that the function is dependent on the naming conventions of PrairieView and will not work on arbitrary folders.

Parameters

folder (`Union[str, Path]`) – folder containing bruker imaging data

Return type

`Tuple[int, int, int, int, int]`

Returns

channels, planes, frames, height, width

CalSciPy.bruker.**extract_frame_times**(*filename: str | pathlib.Path*) → pandas.core.frame.DataFrame

Function to extract the relative frame times from a PrairieView imaging session's primary .xml file

Param

filename

Return type

`DataFrame`

Returns
dataframe containing time (index, ms) x imaging frame (*zero-indexed*)

`CalSciPy.bruker.`**`generate_bruker_naming_convention`**(*channel: int*, *plane: int*, *num_channels: int = 1*, *num_planes: int = 1*) → str

Generates the expected bruker naming convention for images collected with an arbitrary number of cycles & channels

This function expects that the naming convention is _Cycle00000_Ch0_000000.ome.tiff where the channel is one-indexed. The 5-digit cycle id represents the frame if using multiplane imaging and the 6-digit tag represents the plane. Otherwise, the 5-digit tag is static and the 6-digit tag represents the frame.

Please note that the parameters channel and plane are *zero-indexed*.

Parameters

- **`channel`** (`int`) – channel to produce name for
- **`plane`** (`int`) – plane to produce name for
- **`num_channels`** (`int`, default: 1) – number of channels
- **`num_planes`** (`int`, default: 1) – number of planes

Return type
`str`

Returns
proper naming convention

`CalSciPy.bruker.`**`load_bruker_tifs`**(*folder: str | pathlib.Path*, *channel: int | None = None*, *plane: int | None = None*) → Tuple[numpy.ndarray]

This function loads images collected and converted to .tif files by Bruker's Prairieview software. If multiple channels or multiple planes exist, each channel and plane combination is loaded to a separate numpy array. Identification of multiple channels / planes is dependent on *`determine_imaging_content()`*. Images are loaded as unsigned 16-bit (`numpy.uint16`), though note that raw bruker files are natively 12 or 13-bit.

Parameters

- **`folder`** (`Union[str, Path]`) – folder containing a sequence of single frame tiff files
- **`channel`** (`Optional[int]`, default: `None`) – specific channel to load from dataset (zero-indexed)
- **`plane`** (`Optional[int]`, default: `None`) – specific plane to load from dataset (zero-indexed)

Return type
`Tuple[ndarray]`

Returns
a tuple of numpy arrays (frames, y-pixels, x-pixels, `numpy.uint16`)

`CalSciPy.bruker.`**`load_voltage_recording`**(*path: str | pathlib.Path*) → pandas.core.frame.DataFrame

Import bruker analog data from an imaging folder or individual file. By PrairieView naming conventions, these | files contain "VoltageRecording" in the name.

Parameters
`path` (`Union[str, Path]`) – folder or filename containing analog data

Return type
`DataFrame`

Returns
dataframe containing time (index, ms) x channel data

`CalSciPy.bruker.`**`repackage_bruker_tifs`**(*input_folder: str | pathlib.Path, output_folder: str | pathlib.Path, channel: int = 0, plane: int = 0*) → None

This function repackages a folder containing .tif files exported by Bruker's Prairieview software into a sequence of <4 GB .tif stacks. Note that parameters channel and plane are **zero-indexed**.

Parameters

- **`input_folder`** (`Union`[`str`, `Path`]) – folder containing a sequence of single frame .tif files exported by Bruker's Prairieview
- **`output_folder`** (`Union`[`str`, `Path`]) – empty folder where .tif stacks will be saved
- **`channel`** (`int`, default: `0`) – specify channel
- **`plane`** (`int`, default: `0`) – specify plane

Return type

`None`

CHAPTER

FOUR

CALSCIPY.COLORING MODULE

class CalSciPy.coloring.**BackgroundImage**(*images: numpy.ndarray*, *style: int = 0*, *cutoffs: Tuple[float, float] = (0.0, 100.0)*)

Bases: `object`

cast() → numpy.ndarray

Return type
`ndarray`

convert() → numpy.ndarray

Return type
`ndarray`

property get: ndarray

rescale() → numpy.ndarray

Return type
`ndarray`

stylize() → numpy.ndarray

Return type
`ndarray`

CalSciPy.coloring.**color_images**(*images: numpy.ndarray*, *rois: numpy.ndarray*) → numpy.ndarray

Return type
`ndarray`

CalSciPy.coloring.**cutoff_images**(*images: numpy.ndarray*, *cutoffs: Tuple[float, float] = (0.0, 100.0)*, *in_place: bool = True*) → numpy.ndarray

Return type
`ndarray`

CalSciPy.coloring.**generate_background_images**(*images: numpy.ndarray*, *style: int = 0*) → numpy.ndarray

Generates a background image

Parameters

- **images** (`ndarray`) –
- **style** (`int`, default: `0`) –

Return type
`ndarray`

Returns

`CalSciPy.coloring.`**`generate_custom_colormap`**(*colors: Tuple[Tuple[float, float, float]]*) → matplotlib.colors.Colormap

Generate a custom linearized colormap from a list of rgb colors Each color must be in the form a tuple of three floats with each float being between 0.0 - 1.0.

Parameters

`colors` (*`list[tuple[float, float, float]]`*) – a list of colors

Returns

a custom linearized colormap

Return type

matplotlib.pyplot.cm.colors.Colormap

`CalSciPy.coloring.`**`rescale_images`**(*images: numpy.ndarray*, *new_range: Tuple[float, float] = (0.0, 255.0)*, *in_place: bool = True*) → numpy.ndarray

Return type

`ndarray`

CHAPTER

FIVE

CALSCIPY.EVENT_PROCESSING MODULE

`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

SIX

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

SEVEN

CALSCIPY.INTERACTIVE_VISUALS MODULE

`CalSciPy.interactive_visuals.`**`interactive_traces`**`(`*traces: numpy.ndarray*, *frame_rate: float*, ***kwargs*`)` → None

Function to interactive compare traces. Press Up/Down to switch neurons

Parameters

- **`traces`** (`ndarray`) – primary traces
- **`frame_rate`** (`float`) – frame rate

Return type
`None`

Returns
interactive figure

`CalSciPy.interactive_visuals.`**`interactive_traces_compare`**`(`*traces: numpy.ndarray*, *traces2: numpy.ndarray*, *frame_rate: float*, ***kwargs*`)` → None

Function to interactively compare two sets of traces. Press Up/Down to switch neurons

Parameters

- **`traces`** (`ndarray`) – primary traces
- **`traces2`** (`ndarray`) – secondary trace
- **`frame_rate`** (`float`) – frame_rate

Return type
`None`

Returns
interactive figure

`CalSciPy.interactive_visuals.`**`interactive_traces_overlay`**`(`*traces: numpy.ndarray*, *traces2: numpy.ndarray*, *frame_rate: float*, ***kwargs*`)` → None

Function to interactive compare traces with an overlay trace (e.g., noise). Press Up/Down to switch neurons

Parameters

- **`traces`** (`ndarray`) – primary traces
- **`traces2`** (`ndarray`) – secondary trace
- **`frame_rate`** (`float`) – frame_rate

Return type
`None`

Returns
interactive figure

CHAPTER

EIGHT

CALSCIPY.IO_TOOLS MODULE

`CalSciPy.io_tools.`**`load_binary`**(*path: str | pathlib.Path*, *mapped: bool = False*) → 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

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*) → 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)

Return type
`int`

Returns
0 if successful

`CalSciPy.io_tools.`**`save_images`**(*path: str | pathlib.Path*, *images: numpy.ndarray*, *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)
- **`size_cap`** (`float`, default: `3.9`) – maximum size per file

Return type
`int`

Returns
returns 0 if successful

CHAPTER

NINE

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.`**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

CHAPTER

TEN

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*, *component: 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
- **component** (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

ELEVEN

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*) → numpy.ndarray

Calculates f/f0 (fold fluorescence over baseline). Baseline is defined as the 5th percentile of the signal after a 1Hz low-pass filter using a Hamming window. Baseline can be calculated using an external reference | using the raw argument or adjusted by using the offset argument. Supports in-place calculation | (off by default).

Parameters

- **`traces`** (`ndarray`) – matrix of traces in the form of neurons x frames
- **`frame_rate`** (`float`, default: `30.0`) – frame rate of dataset
- **`in_place`** (`bool`, default: `False`) – boolean indicating whether to perform calculation in-place
- **`offset`** (`float`, default: `0.0`) – offset added to baseline; useful if traces are non-negative
- **`external_reference`** (`Optional`[`ndarray`], default: `None`) – secondary dataset used to calculate baseline; useful if traces have been factorized

Return type

`ndarray`

Returns

f/f0 matrix of n neurons x m samples

`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*, *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

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

CalSciPy.bruker, ??
CalSciPy.coloring, ??
CalSciPy.event_processing, ??
CalSciPy.image_processing, ??
CalSciPy.interactive_visuals, ??
CalSciPy.io_tools, ??
CalSciPy.misc, ??
CalSciPy.reorganization, ??
CalSciPy.trace_processing, ??