PyNeuroTrace - Python code for neural activity time series

08 May 2024

Summary

Modern techniques for measuring neuronal activity using fluorescent biosensors and ultra-fast microscopy have allowed neuroscientists unprecedented access to neural information processing *in vivo*. The time series datasets generated from experiments sampling somatic action potentials from populations of neurons, or full-dendritic arbor sampling of populations of synapses, are becoming increasingly larger as new technologies allow for faster acquisition rates and higher temporal resolution of neural signals. Neuronal activities are sourced from an ever-expanding library of fluorescent indicators of distinct measures, including detectors of calcium, membrane voltage, and a range of neurotransmitters and neuromodulators. These biosensors are impacted by their unique molecular kinetics and inherent signal-to-noise properties. The quality of neural signal data sets are also impacted by acquisition instruments, which differ in sensitivity and sampling rate. All of these features, including underlying neural signals, biosensor properties, and microscope capabilities, must be considered during post-imaging signal processing with techniques that can scale to the size of modern neural datasets. To address this problem, here, we describe `pyNeuroTrace`, an open-source Python library developed to aid in processing neuronal signals from large fluorescent biosensor data sets, which allows dynamic control of filtering and signal processing with these unique aspects in mind before analyses of the underlying neuronal activity can be conducted.

Statement of need

Many neuroscience labs using optophysiological methods, such as sampling neural activity using two-photon microscopy or fiber photometry, typically must create and constantly adjust functions and filters to analyze raw recordings. Currently, there is limited standardization for approaches for signal processing, with techniques and algorithms scattered throughout the literature such as [@Friedrich2017], requiring substantial framework overhead [@Giovannucci2019]

or implemented in various programming languages other than Python [@Ortega2024]. Our library, `pyNeuroTrace`, seeks to address these problems by providing an analytic package written purely in Python specifically for neuronal activity data sets. Our package includes a collection of filters and algorithms relevant to optophysiological analysis that are implemented in a generalizable manner for time series data sets in either 1D arrays or a collection of recordings in 2D arrays. Additionally, with the increase in acquisition rates of new imaging techniques, we have implemented a subset of these algorithms using GPU-compatible code to significantly increase processing speeds to accommodate large datasets collected, such as those from large population sampling or ultra-fast kilohertz sampling rates.

Signal Processing

DeltaF/F

There are several methods for calculating the change of intensity of a fluorescent indicator over time [@Grienberger2022]. We implemented the method described by Jia *et al.* for the calculation of $\Delta F/F$, which normalizes the signal to a baseline, helping with bleaching or other changes that occur over time, influencing the detection or magnitude of events in the raw signal [@Jia2010]. This implementation includes several smoothing steps to mitigate shot noise [@Jia2010]. In short, F_0 is calculated by finding the minimum signal in a window of the rolling average of the raw signal. Next, ΔF is calculated as the difference in the raw signal and F_0, which is then divided by F_0 to attain the trace for $\Delta F/F_0$. This $\Delta F/F_0$ signal can be optionally smoothed using an exponentially weighted moving average (EWMA) to remove shot noise. Jia *et al.* defined their rolling average with the following equation:

$$\bar{F} = \left(\frac{1}{\tau_1}\right) \int_{x-\tau_1/2}^{x+\tau_1/2} F(\tau)\, d\tau$$

The variable F_0 is defined using a second time constant, τ_2, this parameter is the length of the rolling window to search for the minimum smoothed signal value to be used as a baseline:

$$F_0(t) = min(\bar{F}(x))|t - \tau_2 < x < t$$

Thus $\Delta F/F$ is calculated using F_0 and F where F is the original raw signal:

$$\Delta F/F = \frac{F(t) - F_0}{F_0}$$

The two time constants, τ_1 and τ_2, can be selected by users. An additional third parameter τ_0 is used to optimize the EWMA functions that smooths the final

$\Delta F/F$ trace. Modifying these parameters will have a dramatic influence on the output signal. `pyNeuroTrace` uses defaults proposed by Jia *et al.* ($\tau_0 = 0.2$ s, $\tau_1 = 0.75$ s, and $\tau_2 = 3$ s), which work well for imaging at 30 Hz, these values should be adjusted for different imaging parameters.

Okada Filter

We also provide a Python implementation of the Okada Filter [@Okada2016]. This filter is designed to filter shot noise from traces in low-signal to noise paradigms, which is common for calcium imaging with two-photon microscopy where the collected photon count is low, and noise from PMT detectors can be nontrivial. This filter is defined by Okada *et al.* as:

$$x_t \leftarrow x_t + \frac{x_{t-1} + x_{t+1} - 2x_t}{2(1 + e^{-\alpha(x_t - x_{t-1})(x_t - x_{t+1})})}$$

In this equation, x_t is the value in the neural activity trace at time t. The value for α, which is a coefficient, should be selected so that the product of $x_t - x_{t-1}$ and $x_t - x_{t+1}$ causes a sufficiently steep sigmoid curve which functions a binary filter in the equation. This function is equivalent to the following conditional states from Okada *et al.*:

$$\text{If } (x_t - x_{t-1})(x_t - x_{t+1}) \leq 0$$
$$x_t \leftarrow x_t$$
$$\text{If } (x_t - x_{t-1})(x_t - x_{t+1}) > 0$$
$$x_t \leftarrow \frac{x_{t-1} + x_{t+1}}{2}$$

Essentially, in each trace the Okada filter replaces the point x_t with the average of adjacent values when the product of the differences in adjacent values is greater than zero. One useful characteristic of this smoothing algorithm is that it does not move the start position of events like other algorithms do [@Okada2016]

Nonnegative Deconvolution

When the kinetics of biosensors are known, nonnegative deconvolution (NND) can be useful to deconvolve raw or $\Delta F/F$ traces back into the most likely discrete triggers of increases in the sensors. `pyNeuroTrace` includes an implementation of an algorithm that can efficiently compute NND in linear time [@Podgorski2012]. This can also be useful for denoising, as well particularly good at finding smaller magnitude events in fluorescent imaging that are often obfuscated by machine noise [@Podgorski2012].

Event Detection

The event detection module uses several strategies to identify neuronal activity events in time series datasets. These methodologies have been previously discussed and compared by Sakaki *et al.* [@Sakaki2018]. These include two generalizable methods and one that requires prior knowledge of recorded event shape. The generalizable methods include filtering the signal through an exponentially weighted moving average (ewma) or cumulative sum of movement above the mean (cusum). The final filter is a matched filter that finds the probability of the trace matching a previously defined shape, such as one described by an exponential rise and decay of calcium signal generated by a GECI. Computationally efficient algorithms were selected, allowing for the possibility of applying them in real-time during experiments.

Visualization

`pyNeuroTrace` has several built-in visualization tools depending on the format of the data. 2D arrays of neuronal timeseries can be displayed as heat maps Figure 1 or as individual traces Figure 2. The heatmap is a useful visualization tool for inspecting many traces at once; additionally, at the bottom of the plot, the stimuli timing is displayed if provided Figure 1. This functionality allows for quick visual inspection of from a population of neurons or signals sampled across a neuronal structure, such as a dendritic arbor.

One of the in-built visualizations is specific to the data structure generated by a custom acousto-optic random access two-photon microscope [@Sakaki2020] Figure 1. This microscope uses acousto-optic deflectors (AODs) to perform inertia-free scanning between preselected points of interest, allowing for extremely fast acquisition rates for sampling neuronal activity throughout complex 3D neural structures. The scan engine of the microscope allows for random-access sampling for imaging activity across the entire dendritic arbor morphology of a single neuron [@Sakaki2020]. This type of imaging does not generate a traditional image. The microscope instead links acquired neuronal traces to points of interest organized into a hierarchical tree structure representing the neuronal morphology in a complex data file. `pyNeuroTrace` will use additional information from this microscope to link structure and function by color-coding where neuronal activity comes from in a sampled neuron.

For individual or small numbers of activity traces, `pyNeuroTrace` has a line plot feature Figure 2. This is an ideal option for inspecting the shape of events, which may be difficult to appreciate from the colormaps in the heatmap visualization. Dotted lines are plotted vertically across the traces of neural activity to indicate when stimulus presentation occurred during an experiment.

Additionally, if a record of stimulus or trigger times is provided `pyNeuroTrace` can plot the average evoked response in a recording. Figure 3 shows the average evoked response to a full-field OFF and ON stimuli from the data in Figure 1.

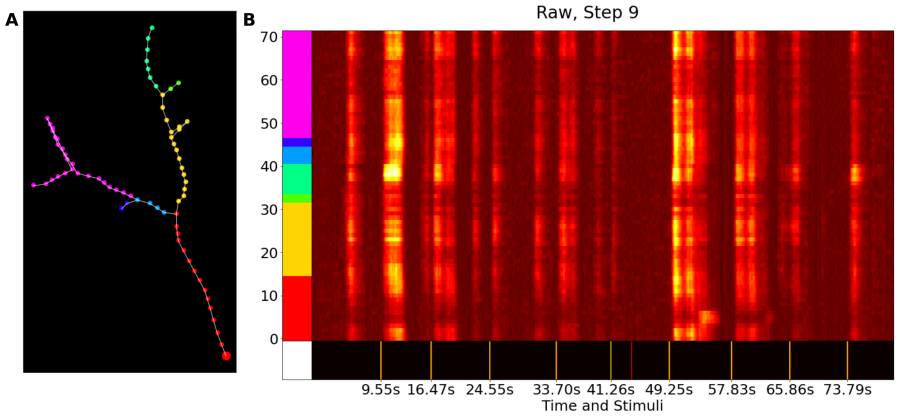

Figure 1: An example of a heatmap generated by pyNeuroTrace. A) An example of a lab-specific visualization of points of interest across a dendritic arbor imaged *in vivo* with an AOD random-access two-photon microscope. Plotted data was recorded from a single dendritic branch *in vivo* using a random access two-photon microscope. The indicator is a GCaMP6m, a plasmid encoding the GECI was single-cell electroporated [@Haas2001] B) Heatmap of visually evoked responses imaged at 148 Hz, showing intensity (yellow being highest), in addition to stim indicators along the X axis, and branch indicator along the Y.

This graph shows the evoked synaptic weights differ for the different visual stimuli presented to the animal.

GPU Acceleration

Several of the filters in `pyNeuroTrace` have been rewritten to be almost entirely vectorized in their calculations. The benefit is more noticeable when comparing the difference in performance while using large data sets, such as those generated using a longer time series or faster acquisition rates. These vectorized implementations gain further speed by being executed on a GPU using the Cupy Python library [@cupy_learningsys2017]. The GPU-accelerated filters can be imported from the `pyneurotrace.gpu.filters` module, and a CUDA-compatible graphics card is required for their execution. This functionality is becoming increasingly crucial as acquisition rates increase for kilohertz imaging of activity [@Zhang2019], which can generate arrays of hundreds of thousands of data points in just a few minutes of recording. Figure 4 shows the difference in calculating arrays of various sizes using either the CPU or vectorized GPU-based approach of the $\Delta F/F$ function. The CPU used in these calculations was an Intel i5-9600K with six 4.600GHz cores; the GPU was an NVIDIA GeForce RTX 4070 with CUDA Version 12.3. When compared, the GPU calculation was on average in the order of 100 times faster on time series up to 100,000 points in

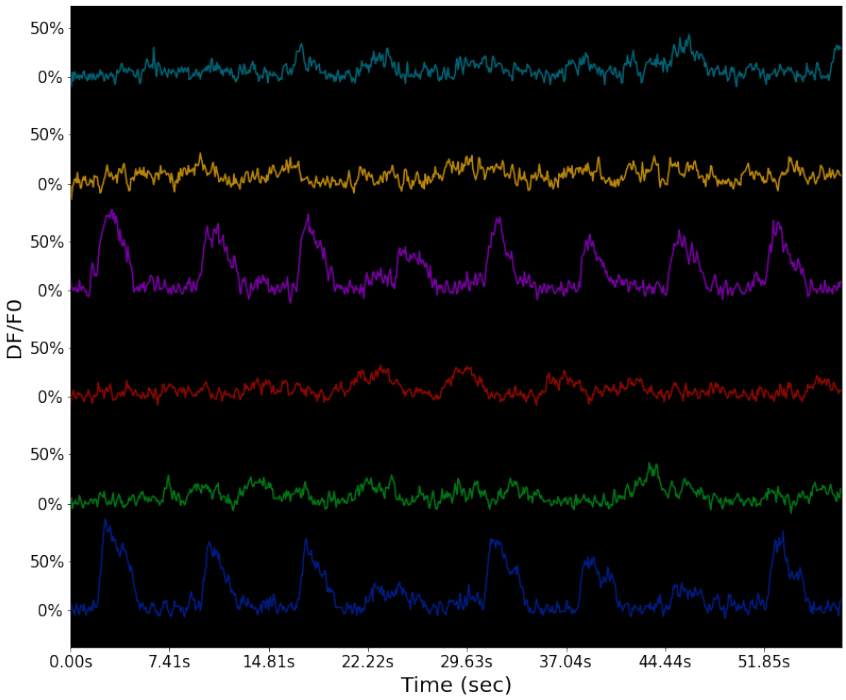

Figure 2: Six traces from neuron cell bodies imaged *in vivo* with a SLAP2 microscope. Tectal cells were bulk loaded with CAL-590-AM, a calcium senstive dye. Plotting individual traces better highlights the event shape

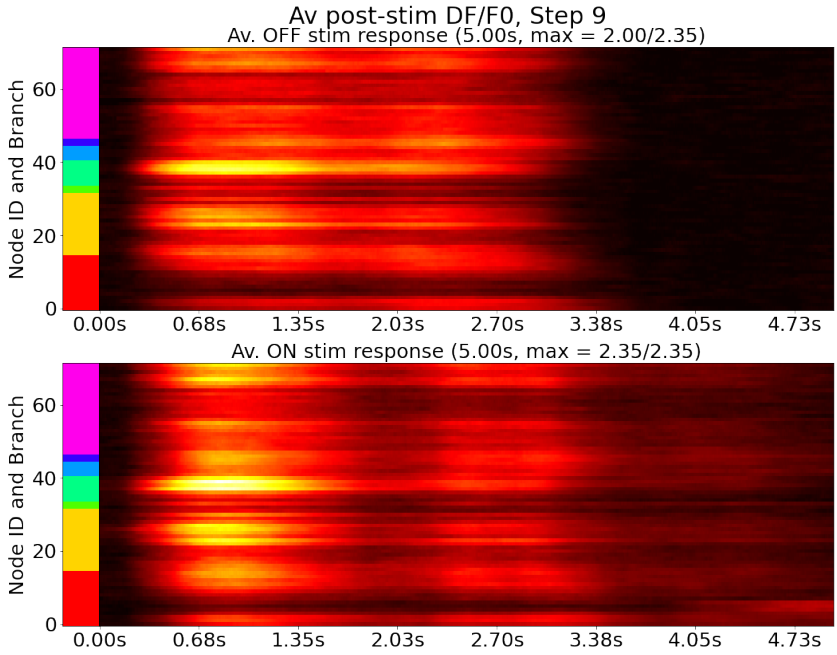

Figure 3: Average stimulus-locked responses from the same *in vivo* imaging experiment of the dendritic branch in Figure 1 . The response for each branch node is plotted for two visual stimuli, full field OFF and ON

size.

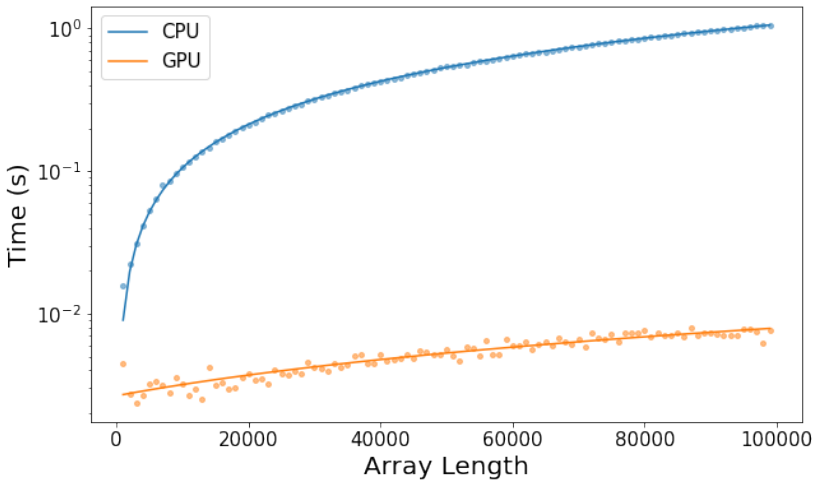

Figure 4: Comparison between $\Delta F/F$ with EWMA calculations for different array sizes using either the CPU (blue) or GPU (orange).

To vectorize the functions several were modified. For example the EMWA used to smooth the $\Delta F/F$ signal as described by Jia *et al.* was changed to an approximation using convolution with an exponential function. The kernel used to perform this is defined as:

$$w[i] = \begin{cases} \alpha \cdot (1-\alpha)^i & \text{for } i = 0, 1, 2, \ldots, N-1 \\ 0 & \text{otherwise} \end{cases}$$

Where α is defined as:

$$\alpha = 1 - e^{-\frac{1}{\tau}}$$

τ is a user-selected time constant in seconds, which is translated into the number of samples using the acquisition rate used to acquire the data. N is a window parameter for the kernel calculated using α:

$$N = \left\lfloor -\frac{\log(10^{-10})}{\alpha} \right\rfloor$$

This filters for smaller values that have a minuscule influence on the weighted average. The kernel needs to be normalized such that the sum of its elements is

1, to produce smoothing with the same approximate value as the non-vectorized implementation:

$$w[i] \leftarrow \frac{w[i]}{\sum_{j=0}^{N-1} w[j]}$$

The normalized kernel is then convolved with the $\Delta F/F$ signal, d:

$$c[k] = \sum_{i=0}^{N-1} w[i] \cdot d[k-i]$$

This convolved signal, c is then normalized to the cumulative sum of the exponential kernel:

$$n[j] = \sum_{i=0}^{j} w[i]$$

$$emwa = \frac{c[i]}{n[i]}$$

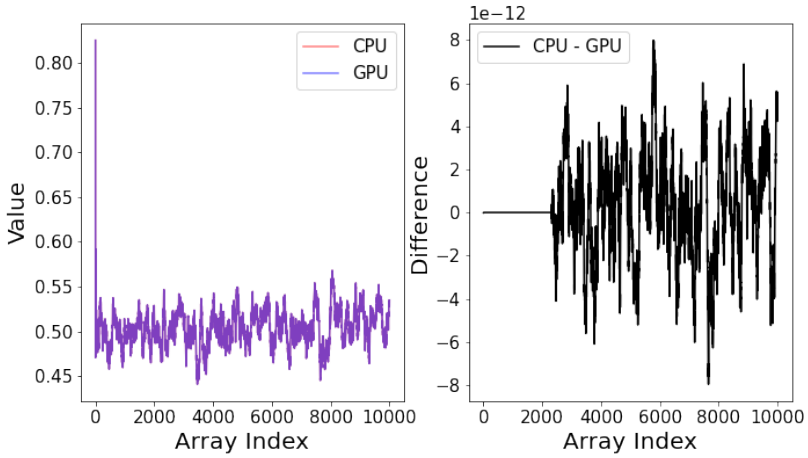

Figure 5: Overlay of the EWMA calculations using the CPU implementation and GPU approximation in red and blue. The difference in values from the output is also plotted. The EWMA traces diverge after the initial window length of the GPU version's kernel, but never by more than a factor of 1e-11.

To demonstrate the differences between the CPU and GPU implementations of the EWMA calculations were performed on an array of random values Figure 5.

These were generated from the same array using the respective decays for either implementation using the time constant of 50 milliseconds and a sampling rate of 2kHz. Depending on user parameters, the difference between the two outputs typically ranges in magnitude from 1e-16 to 1e-12. These discrepancies can also be attributed to differences in floating-point number accuracy between CPU and GPU calculations.

Acknowledgements

The development of this software was supported by funds from the Canadian Institutes of Health Research (CIHR) Foundation Award (FDN-148468).

References