Neural Data Science

Lecturer: Jan Lause, Prof. Dr. Philipp Berens

Tutors: Jonas Beck, Rita González Márquez, Fabio Seel

Summer term 2024

Student name: Carlotta Trottenberg, Jungmin Lee, Ralf Krüger

Coding Lab 3

- **Data**: Download the data file `nds_cl_3_*.csv` from ILIAS and save it in a subfolder `../data/`.
- **Dependencies**: You don't have to use the exact versions of all the dependencies in this notebook, as long as they are new enough. But if you run "Run All" in Jupyter and the boilerplate code breaks, you probably need to upgrade them.

Two-photon imaging is widely used to study computations in populations of neurons. In this exercise sheet we will study properties of different indicators and work on methods to infer spikes from calcium traces. All data is provided at a sampling rate of 100 Hz. For analysis, please resample it to 25 Hz using `scipy.signal.decimate`.

In [ ]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from scipy import signal
from scipy.io import loadmat
from __future__ import annotations
import wget

#%matplotlib inline

#%load_ext jupyter_black

#%load_ext watermark
#%watermark --time --date --timezone --updated --python --iversions --watermark
```

In [ ]:
```
#plt.style.use("../matplotlib_style.txt")
```

Load data

In [ ]:
```
# ogb dataset from Theis et al. 2016 Neuron
ogb_calcium = pd.read_csv("data/nds_cl_3_ogb_calcium.csv", header=0)
ogb_spikes = pd.read_csv("data/nds_cl_3_ogb_spikes.csv", header=0)

# gcamp dataset from Chen et al. 2013 Nature
gcamp_calcium = pd.read_csv("data/nds_cl_3_gcamp2_calcium.csv", header=0)
gcamp_spikes = pd.read_csv("data/nds_cl_3_gcamp2_spikes.csv", header=0)
```

In [ ]:
```
ogb_calcium.shape, ogb_spikes.shape, gcamp_calcium.shape, gcamp_spikes.shape
```

Out[ ]:
```
((71986, 11), (71986, 11), (23973, 37), (23973, 37))
```

In [ ]:
```
ogb_spikes.head()
```

Out[ ]:

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0.0	0.0	0	0	0	0.0	0	0.0	0
1	0	0	0.0	0.0	0	1	0	0.0	0	0.0	0
2	0	0	0.0	0.0	0	0	0	0.0	0	0.0	0
3	0	0	0.0	0.0	0	1	0	0.0	0	0.0	0
4	0	0	0.0	0.0	0	0	0	0.0	0	0.0	0

In [ ]:
```
# set resampling factor 100 /25 = 4
q = 4

print(
    "shapes before resampling:",
    ogb_calcium.shape,
    ogb_spikes.shape,
    gcamp_calcium.shape,
    gcamp_spikes.shape,
)


def resample(x: pd.DataFrame, q: int, use_scipy=True) -> pd.DataFrame:
    """
    Resamples the input DataFrame by using SciPy's decimate function or summing

    Parameters:
    - x: pd.DataFrame
        The input DataFrame with spike signals in each column.
    - q: int
        The downsampling factor. The number of rows in the output will be the or
    - use_scipy: bool, default=True
        If True, uses SciPy's decimate function to downsample the DataFrame. If

    Returns:
    - pd.DataFrame
        The resampled DataFrame.
    """
    if use_scipy:
        return pd.DataFrame(
            signal.decimate(x, q=q, axis=0),
            columns=np.arange(0, x.shape[1]).astype("str"),
        )
    else:
        # trim to a multiple of the factor q
        n = x.shape[0]
        m = n // q
        x.drop(x.tail(n - (m * q)).index, inplace=True)

        # put every entry into non-overlapping bins
        x["bin"] = np.arange(0, m).repeat(q)
```

```
        # sum over the bins
        resampled_x = x.groupby(["bin"]).sum()

        return resampled_x


# resample the calcium signal with scipy.signal.decimate
ogb_calcium_25 = resample(ogb_calcium, q=q)
gcamp_calcium_25 = resample(gcamp_calcium, q=q)

# resample the spike signal, that does not work with the scipy function, because
ogb_spikes_25 = resample(ogb_spikes, q=q, use_scipy=False)
gcamp_spikes_25 = resample(gcamp_spikes, q=q, use_scipy=False)

print(
    "shapes after resampling:",
    ogb_calcium_25.shape,
    ogb_spikes_25.shape,
    gcamp_calcium_25.shape,
    gcamp_spikes_25.shape,
)
```

```
shapes before resampling: (71986, 11) (71986, 11) (23973, 37) (23973, 37)
shapes after resampling: (17997, 11) (17996, 11) (5994, 37) (5993, 37)
```

Task 1: Visualization of calcium and spike recordings

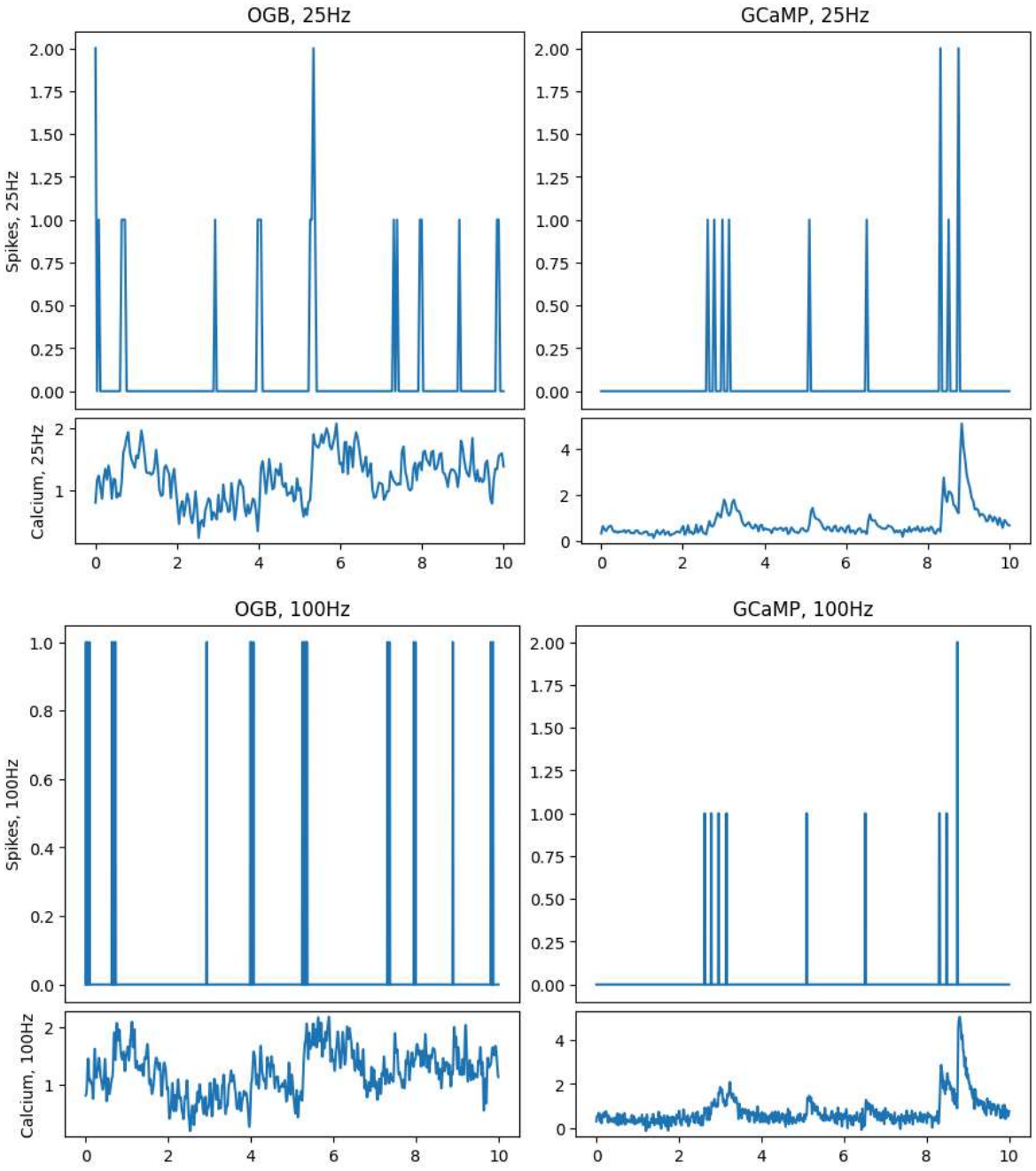
We start again by plotting the raw data - calcium and spike traces in this case. One dataset has been recorded using the synthetic calcium indicator OGB-1 at population imaging zoom (~100 cells in a field of view) and the other one using the genetically encoded indicator GCamp6f zooming in on individual cells. Plot the traces of an example cell from each dataset to show how spikes and calcium signals are related. A good example cell for the OGB-dataset is cell 5. For the CGamp-dataset a good example is cell 6. Zoom in on a small segment of tens of seconds and offset the traces such that a valid comparison is possible.

Grading: 2 pts

In [ ]:
```
# ----------------------------
# Plot raw calcium data (1 pt)
# ----------------------------

# -------------------------
# Plot raw spike data (1 pt)
# -------------------------
SHOWPRERESAMPLED = True

fig, axs = plt.subplots(
    2, 2, figsize=(9, 5), height_ratios=[3, 1], layout="constrained"
)

# set sampling rate in Hz
fs = 25
```

```
# set time-window in s
start = 0
duration = 10
end = start + duration

# convert timewindow to samples
start_s = start * fs
duration_s = duration * fs
end_s = end * fs

# create time linspace
time = np.linspace(start, end, duration_s)

# plot raw ogb data of Cell 5
axs[0, 0].plot(time, ogb_spikes_25["5"][start_s:end_s])
axs[1, 0].plot(time, ogb_calcium_25["5"][start_s:end_s])

# plot raw gcamp data of cell 6
axs[0, 1].plot(time, gcamp_spikes_25["6"][start_s:end_s])
axs[1, 1].plot(time, gcamp_calcium_25["6"][start_s:end_s])

# first row does not need x-axis
axs[0, 0].set_xticks([])
axs[0, 1].set_xticks([])

# set the column names for the subplots
axs[0, 0].set_title("OGB, 25Hz")
axs[0, 1].set_title("GCaMP, 25Hz")

# set the row names for the subplots
axs[0, 0].set_ylabel("Spikes, 25Hz")
axs[1, 0].set_ylabel("Calcium, 25Hz")

plt.show()

if SHOWPRERESAMPLED:
    fig, axs = plt.subplots(
        2, 2, figsize=(9, 5), height_ratios=[3, 1], layout="constrained"
    )

    # set sampling rate in Hz
    fs = 100

    # set time-window in s
    start = 0
    duration = 10
    end = start + duration

    # convert timewindow to samples
    start_s = start * fs
    duration_s = duration * fs
    end_s = end * fs

    # create time linspace
    time = np.linspace(start, end, duration_s)

    # plot raw ogb data of Cell 5
    axs[0, 0].plot(time, ogb_spikes["5"][start_s:end_s])
    axs[1, 0].plot(time, ogb_calcium["5"][start_s:end_s])
```

```
    # plot raw gcamp data of cell 6
    axs[0, 1].plot(time, gcamp_spikes["6"][start_s:end_s])
    axs[1, 1].plot(time, gcamp_calcium["6"][start_s:end_s])

    # first row does not need x-axis
    axs[0, 0].set_xticks([])
    axs[0, 1].set_xticks([])

    # set the column names for the subplots
    axs[0, 0].set_title("OGB, 100Hz")
    axs[0, 1].set_title("GCaMP, 100Hz")

    # set the row names for the subplots
    axs[0, 0].set_ylabel("Spikes, 100Hz")
    axs[1, 0].set_ylabel("Calcium, 100Hz")

    plt.show()
```



Task 2: Simple deconvolution

It is clear from the above plots that the calcium events happen in relationship to the spikes. As a first simple algorithm implement a deconvolution approach like presented in the lecture in the function `deconv_ca`. Assume an exponential kernel where the decay constant depends on the indicator ($\tau_{OGB} = 0.5s$, $\tau_{GCaMP} = 0.1s$). As we know that there can be no negative rates, apply a heavyside function to the output. Plot the kernel as well as an example cell with true and deconvolved spike rates. Scale the signals such as to facilitate comparisons. You can use functions from `scipy` for this.

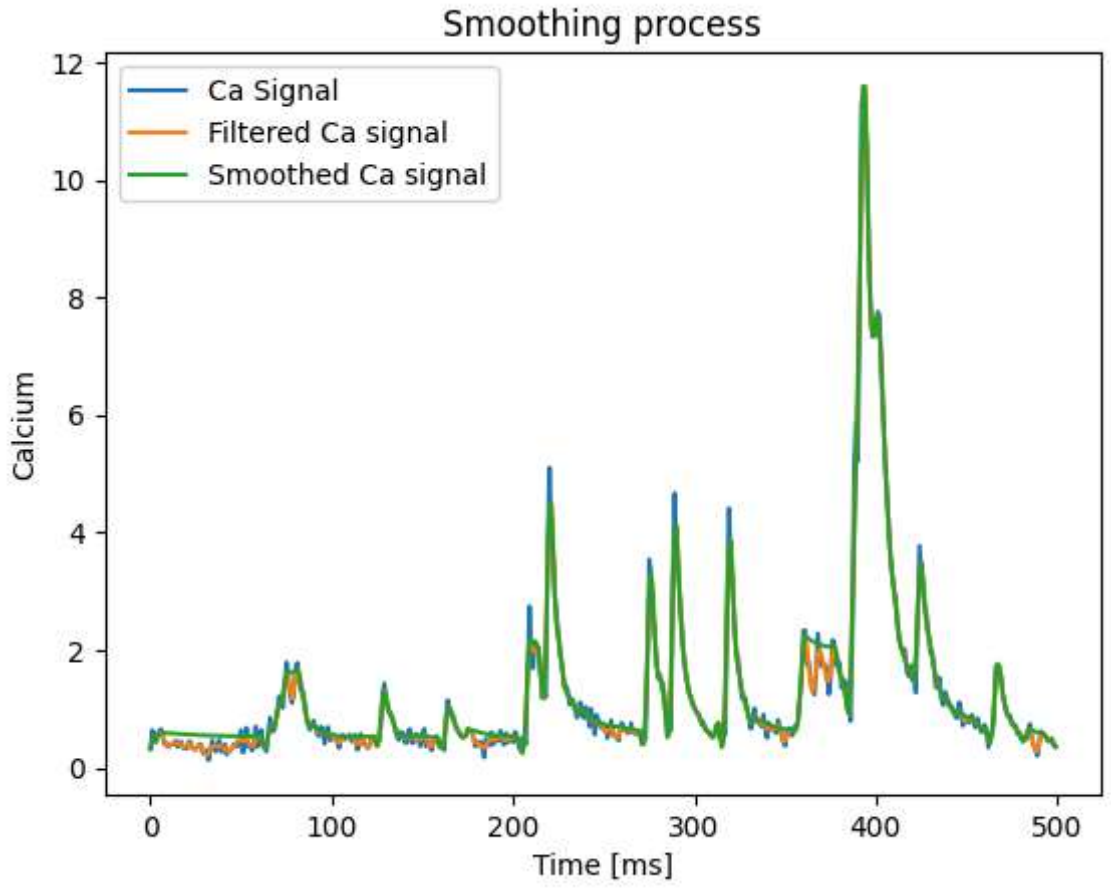
Grading: 3 pts

In [ ]:
```python
def butterworth_lowpass_filter(
    data: np.ndarray, cutoff: float, fs: float, order: int = 4
) -> np.ndarray:
    """
    Apply a 4-pole Butterworth lowpass filter to the input data with a specified

    Parameters
    ----------
    data : np.ndarray, (n_points,)
        The input signal to be filtered.

    cutoff : float
        The cutoff frequency of the filter in Hz.

    fs : float
        The sampling frequency of the input signal in Hz.

    order : int, optional
        The order of the filter. Default is 4 (4-pole).

    Returns
    -------
    np.ndarray
        The filtered signal.
    """
    nyquist = 0.5 * fs
    normal_cutoff = cutoff / nyquist

    sos = signal.butter(order, normal_cutoff, btype="low", analog=False, output=
    filtered_data = signal.sosfiltfilt(sos, data)

    return filtered_data


def iterative_smoothing(data, thrnoise=1, max_iter=5000):
    """
    Apply iterative smoothing to a signal.

    This function applies an iterative smoothing procedure to a 1-dimensional si
    It identifies peaks in the signal and smooths the segments between adjacent
    with amplitudes smaller than the specified threshold. The smoothing is repea
    iteratively until a stopping condition is met.

    Parameters
    ----------
    data : array_like
        The 1-dimensional input signal.
```

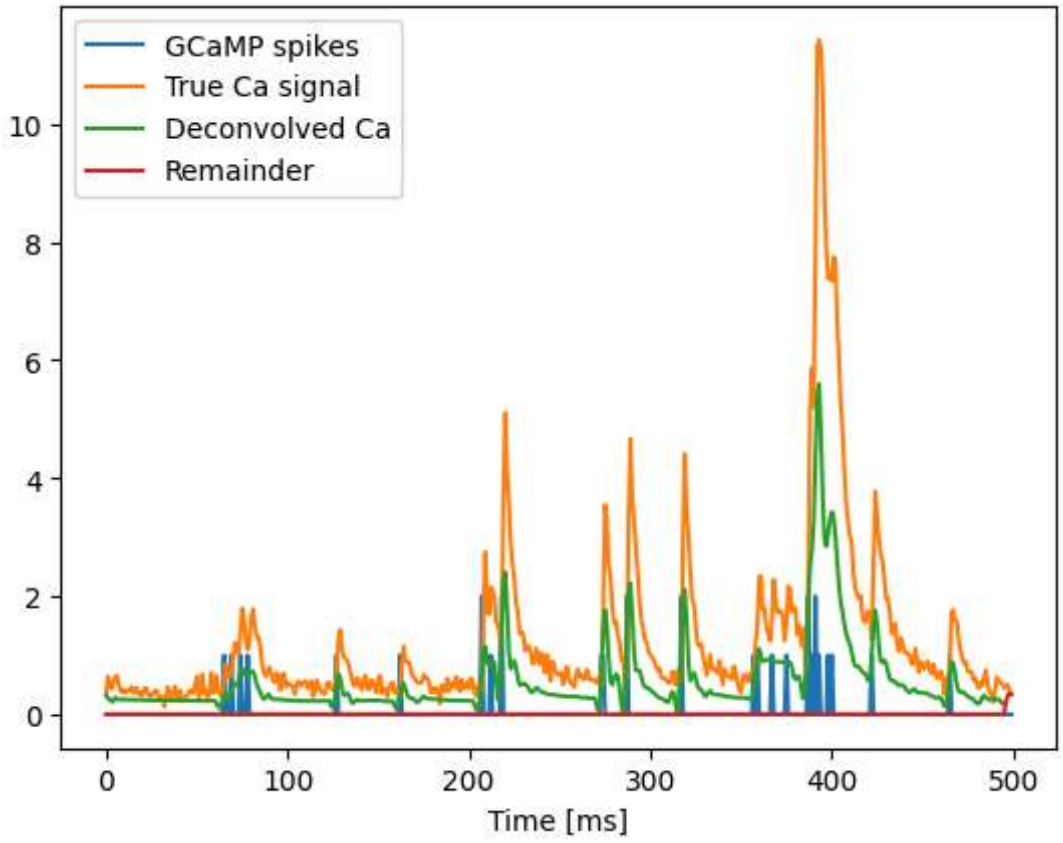
```
    thrnoise : float, optional
        The threshold amplitude below which smoothing is applied. Defaults to 1.
    max_iter : int, optional
        The maximum number of iterations. Defaults to 5000.

    Returns
    -------
    array_like
        The smoothed signal.

    Notes
    -----
    - This function does not modify the input signal.
    - The iterative smoothing continues until all peaks with amplitudes less tha
      the threshold are removed, or until the maximum number of iterations is re
    """
    # Make a copy of the input signal to avoid modifying it in place
    smoothed_signal = data.copy()

    for iteration in range(max_iter):
        # Find all peaks in the signal
        peaks, _ = signal.find_peaks(smoothed_signal)

        # Check if there are enough peaks to continue
        if len(peaks) < 2:
            break

        # Calculate peak-to-peak amplitudes
        peak_amplitudes = np.abs(np.diff(smoothed_signal[peaks]))

        # Identify the smallest peak amplitude
        if len(peak_amplitudes) == 0:
            break
        pmin_idx = np.argmin(peak_amplitudes)
        pmin = peaks[pmin_idx]

        # Check if the smallest peak amplitude is below the threshold
        if peak_amplitudes[pmin_idx] < thrnoise:
            # Define the segment between the adjacent peaks of pmin
            segment_start = peaks[pmin_idx]
            segment_end = peaks[pmin_idx + 1]
            segment = smoothed_signal[segment_start : segment_end + 1]

            # Apply smoothing three times
            for _ in range(3):
                # Calculate the average of each point with its neighbors
                smoothed_segment = segment.copy()
                for i in range(1, len(segment) - 1):
                    smoothed_segment[i] = np.sum(segment[i - 1 : i + 2]) / 3
                segment = smoothed_segment

            # Replace the original segment with the smoothed segment
            smoothed_signal[segment_start : segment_end + 1] = segment
        else:
            # Stop if no peak satisfies the threshold condition
            break

    return smoothed_signal
```

```python
def kernel(t: np.ndarray, tau: float) -> np.ndarray:
    """
    Compute the exponential decay kernel.

    This function takes a numpy array of time values and a decay constant,
    and returns the exponential decay kernel for those time values.

    Parameters
    ----------
    t : np.ndarray
        A numpy array of time values.

    tau : float
        The decay constant.

    Returns
    -------
    np.ndarray
        A numpy array containing the computed exponential decay values.

    Raises
    ------
    ValueError
        If tau is zero.
    """
    if tau == 0:
        raise ValueError("tau must be non-zero")

    return np.exp(-t / tau)


def deconv_ca(ca: np.ndarray, tau: float, dt: float, plot_smoothing: bool) -> np
    """
    Compute the deconvolution of the calcium signal.

    Parameters
    ----------

    ca: np.array, (n_points,)
        Calcium trace

    tau: float
        decay constant of conv kernel

    dt: float
        sampling interval.

    plot_smoothing: bool
        plot smoothing process or not

    Return
    ------

    deconvolved_ca: np.array (n_points,)
        spiking rate of the signal
    """

    # insert your code here

    # ----------------------------------------
```

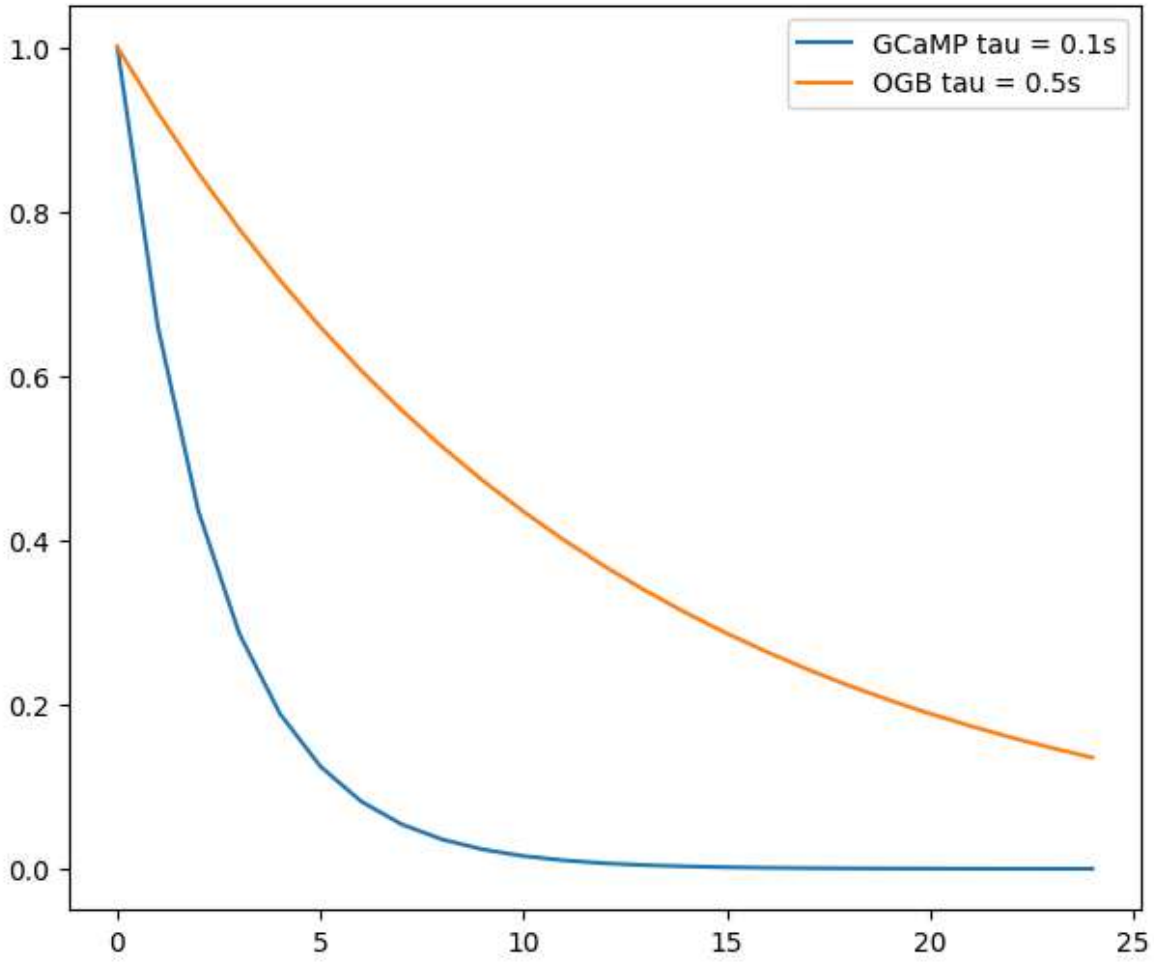
```
    # apply devonvolution to calcium signal (1 pt)
    # -----------------------------------------

    # procedure follows Yaksi & Friedrich 2006
    # 1. low pass filter
    # set cutoff frequency to 0.2*25 = 5 as in the paper
    cutoff_freq = 5  # Hz
    filtered_ca = butterworth_lowpass_filter(ca[:dt], cutoff_freq, fs=25, order=

    # 2. Apply iterative average smoothing
    smoothed_ca = iterative_smoothing(filtered_ca, thrnoise=0.3, max_iter=100000

    if plot_smoothing == True:
        plt.plot(ca[:dt], label='Ca Signal')
        plt.plot(filtered_ca, label='Filtered Ca signal')
        plt.plot(smoothed_ca, label='Smoothed Ca signal')
        plt.title('Smoothing process')
        plt.xlabel('Time [ms]')
        plt.ylabel('Calcium')
        plt.legend()
        plt.show("Smoothing process")

    # 3. deconvolve
    # create kernel
    kernel_ca = kernel(np.linspace(0, 2 * tau, int(50 * tau)), tau)

    deconvolved_ca, remainder = signal.deconvolve(smoothed_ca, kernel_ca)
    # apply heavyside function
    deconvolved_ca[deconvolved_ca < 0] = 0

    return deconvolved_ca, remainder


deconvolved_ca, remainder = deconv_ca(gcamp_calcium_25["6"], 0.1, 500, True)
```



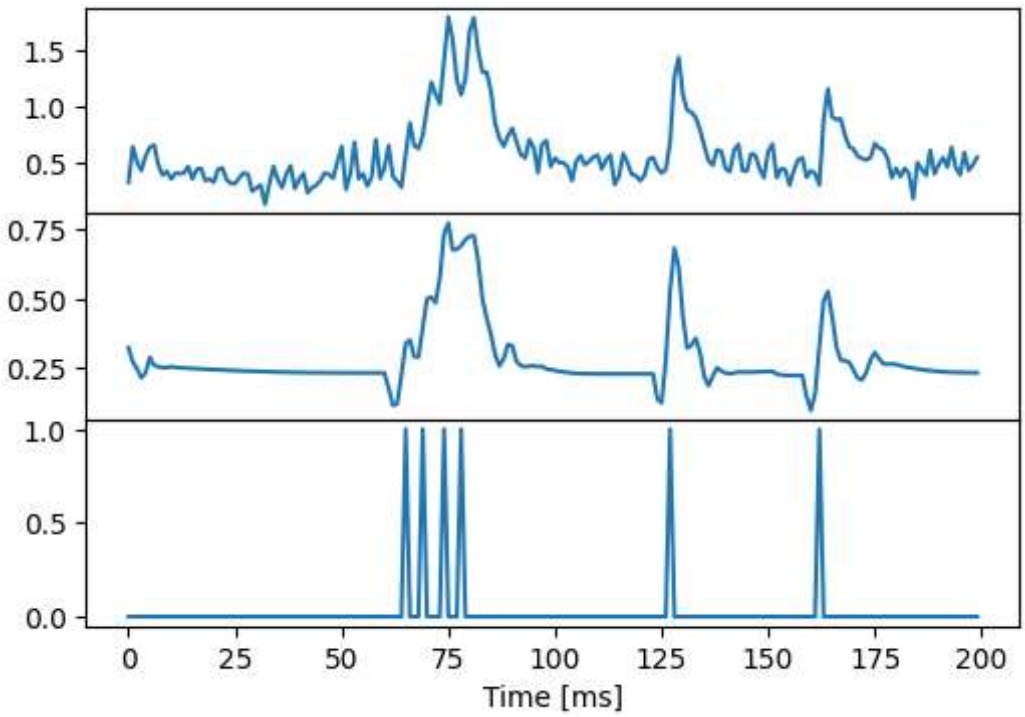
In [ ]:
```
plt.plot(gcamp_spikes_25["6"][:500], label='GCaMP spikes')
plt.plot(gcamp_calcium_25["6"][:500], label="True Ca signal" )
plt.plot(deconvolved_ca, label='Deconvolved Ca')
plt.plot(remainder, label='Remainder')
plt.xlabel('Time [ms]')
plt.legend()
plt.show()
```



In [ ]:
```
fig, ax = plt.subplots(figsize=(6, 5), layout="constrained")

# -------------------------
# Plot the 2 kernels (1 pt)
# -------------------------
ax.plot(kernel(np.linspace(0, 1, 25), 0.1), label='GCaMP tau = 0.1s')
ax.plot(kernel(np.linspace(0, 1, 25), 0.5), label='OGB tau = 0.5s')
plt.legend()
plt.show()
```



In [ ]:
```
# ------------------------------------------------------------------------
# Compare true and deconvolved spikes rates for the OGB or GCamP Cell (1 pt)
# ------------------------------------------------------------------------

fig, axs = plt.subplots(
    3, 1, figsize=(6, 4), height_ratios=[1, 1, 1], gridspec_kw=dict(hspace=0)
)

axs[0].plot(gcamp_calcium_25["6"][:200])
axs[1].plot(deconvolved_ca[:200])
axs[2].plot(gcamp_spikes_25["6"][:200])
plt.xlabel('Time [ms]')
plt.show()
```



Task 3: Run more complex algorithm

As reviewed in the lecture, a number of more complex algorithms for inferring spikes from calcium traces have been developed. Run an implemented algorithm on the data and plot the result. There is a choice of algorithms available, for example:

- Vogelstein: oopsi
- Theis: c2s
- Friedrich: OASIS

Grading: 2 pts

In [ ]:
```
# run this cell to download the oopsi.py file if you haven't already mannually d
# and put it in the same folder as this notebook
#!wget https://raw.githubusercontent.com/liubenyuan/py-oopsi/master/oopsi.py
```

In [ ]:
```
import oopsi
```

In [ ]:
```
# ------------------------------------------------------------
# Apply one of the advanced algorithms on the OGB Cell (0.5 pts)
# ------------------------------------------------------------
thing = oopsi.fast(F=ogb_calcium_25["5"], iter_max=10)
```

In [ ]:
```
ogb_calcium_25
```

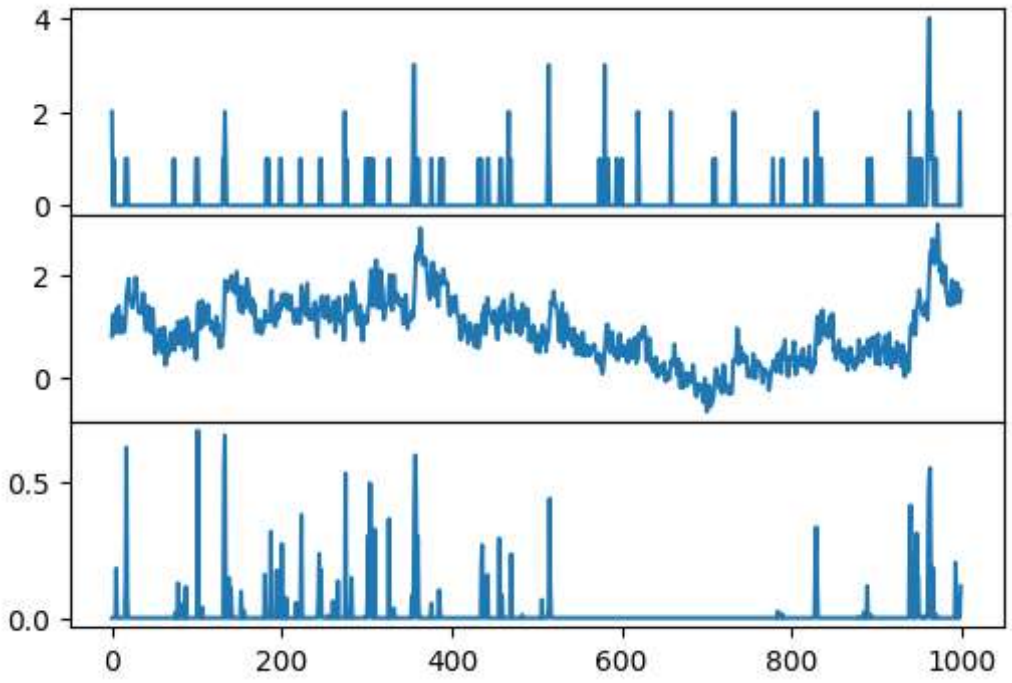
Out[ ]:

	0	1	2	3	4	5	6	7	8
0	-0.225512	0.682813	NaN	NaN	-0.014691	0.809080	2.409345	NaN	0.223117
1	-0.038297	0.791697	NaN	NaN	0.014081	1.169652	2.813134	NaN	0.447265
2	0.325830	1.403091	NaN	NaN	0.179510	1.236355	2.124315	NaN	0.527059
3	0.511742	1.628543	NaN	NaN	0.577951	1.061192	2.468787	NaN	0.484921
4	0.424323	1.426286	NaN	NaN	0.321181	0.870407	3.078764	NaN	0.652461
...	...	...	...	...	...	...	...	...	...
17992	0.214454	0.489878	NaN	NaN	-0.135411	1.217638	0.487426	NaN	0.118117
17993	0.298484	0.605883	NaN	NaN	-0.191092	1.205213	0.513444	NaN	-0.017239
17994	0.413962	0.452771	NaN	NaN	-0.159158	1.183842	0.742174	NaN	0.290038
17995	0.623706	0.224505	NaN	NaN	-0.164893	1.268332	0.400869	NaN	0.631918
17996	0.203266	0.341060	NaN	NaN	-0.116066	1.098661	1.139374	NaN	0.488987

17997 rows × 11 columns

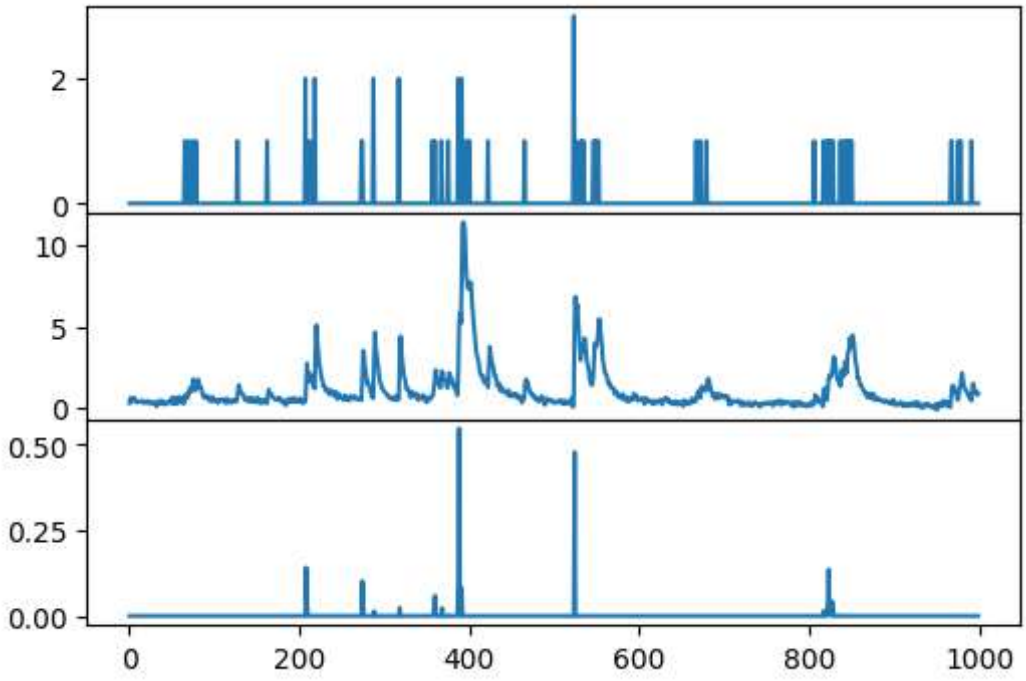
In [ ]:

```
# ----------------------------------------
# Plot the results for the OGB Cell (0.5 pts)
# ----------------------------------------

fig, axs = plt.subplots(
    3, 1, figsize=(6, 4), height_ratios=[1, 1, 1], gridspec_kw=dict(hspace=0)
)

# OGB Cell
axs[0].plot(ogb_spikes_25["5"][:1000])
axs[1].plot(ogb_calcium_25["5"][:1000])
axs[2].plot(thing[0][:1000])
plt.show()
```



In [ ]:
```
# ----------------------------------------------------------------
# Apply one of the advanced algorithms on the GCamP Cell (0.5 pts)
# ----------------------------------------------------------------
thing = oopsi.fast(F=gcamp_calcium_25["6"], iter_max=10)
```

In [ ]:
```
# ------------------------------------------
# Plot the results for the GCamp Cell (0.5 pts)
# ------------------------------------------

fig, axs = plt.subplots(
    3, 1, figsize=(6, 4), height_ratios=[1, 1, 1], gridspec_kw=dict(hspace=0)
)


# GCamP Cell
axs[0].plot(gcamp_spikes_25["6"][:1000])
axs[1].plot(gcamp_calcium_25["6"][:1000])
axs[2].plot(thing[0][:1000])
plt.show()
```



Task 4: Evaluation of algorithms

To formally evaluate the algorithms on the two datasets run the deconvolution algorithm and the more complex one on all cells and compute the correlation between true and inferred spike trains. `DataFrames` from the `pandas` package are a useful tool for aggregating data and later plotting it. Create a dataframe with columns

- algorithm
- correlation
- indicator

and enter each cell. Plot the results using `stripplot` and/or `boxplot` in the `seaborn` package.

Grading: 3 pts

Evaluate on OGB data

In [ ]:
```
# ------------------------------------------------
# Create dataframe for OGB Cell as described (1 pt)
# ------------------------------------------------
# columns: algorithm, correlation, indicator
# algorithms: deconvolution, oopsi
# indicators: GCamp, OGB
# rows = cells

# each cell takes up 2 rows?? one per algorithm

indicator = 'OGB'

# init lists
algo = []
c = []
indi = []
```

```
# loop thru all cells
# for each cell, compute 1 correlation per algorithm
for cell in ogb_calcium_25:
    raw_trace = ogb_calcium_25[cell]

    # get length
    length = len(raw_trace)

    # only run algorithms if the column is not nans:
    if (raw_trace).isna().sum() < length:
        oopsi_trace = oopsi.fast(F=raw_trace, iter_max=10)[0]
        decon_trace, remainder = deconv_ca(raw_trace, 0.5, 1000, False)
        oopsi_corr = np.corrcoef(raw_trace, oopsi_trace)[0,1]
        decon_corr = np.corrcoef(raw_trace[:len(decon_trace)], decon_trace)[0,1]
    else:
        oopsi_corr = np.nan
        decon_corr = np.nan

    # add to my lists
    algo.append('oopsi')
    c.append(oopsi_corr)
    indi.append(indicator)

    algo.append('decon')
    c.append(decon_corr)
    indi.append(indicator)
```

Create OGB dataframe

In [ ]:
```
df_ogb = pd.DataFrame({"algorithm": algo, "correlation": c, "indicator": indi})
df_ogb.head()
```

Out[ ]:

	algorithm	correlation	indicator
0	oopsi	0.146937	OGB
1	decon	0.462753	OGB
2	oopsi	0.105047	OGB
3	decon	0.321657	OGB
4	oopsi	NaN	OGB

Evaluate on GCamp data

In [ ]:
```
# ------------------------------------------------
# Create dataframe for GCamP Cell as described (1 pt)
# ------------------------------------------------

indicator = 'GCamP'

# init lists
algo = []
c = []
indi = []
```

```
# loop thru all cells
# for each cell, compute 1 correlation per algorithm
for cell in ogb_calcium_25:
    raw_trace = ogb_calcium_25[cell]

    # get length
    length = len(raw_trace)

    # only run algorithms if the column is not nans:
    if (raw_trace).isna().sum() < length:
        oopsi_trace = oopsi.fast(F=raw_trace, iter_max=10)[0]
        decon_trace, remainder = deconv_ca(raw_trace, 0.1, 1000, False)
        oopsi_corr = np.corrcoef(raw_trace, oopsi_trace)[0,1]
        decon_corr = np.corrcoef(raw_trace[:len(decon_trace)], decon_trace)[0,1]
    else:
        oopsi_corr = np.nan
        decon_corr = np.nan

    # add to my lists
    algo.append('oopsi')
    c.append(oopsi_corr)
    indi.append(indicator)

    algo.append('decon')
    c.append(decon_corr)
    indi.append(indicator)
```

Create GCamp dataframe

In [ ]:
```
df_gcamp = pd.DataFrame({"algorithm": algo, "correlation": c, "indicator": indic
df_gcamp.head()
```

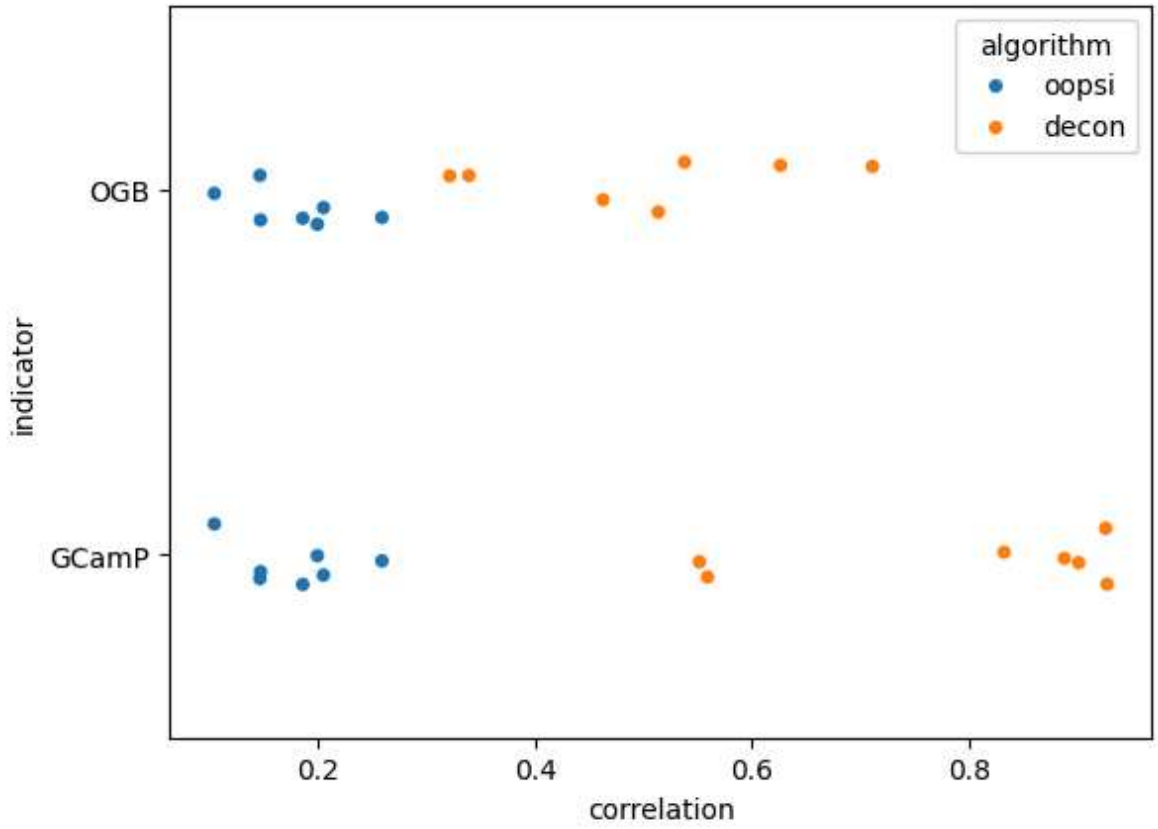
Out[ ]:

	algorithm	correlation	indicator
0	oopsi	0.146937	GCamP
1	decon	0.831866	GCamP
2	oopsi	0.105047	GCamP
3	decon	0.559037	GCamP
4	oopsi	NaN	GCamP

Combine both dataframes and plot

In [ ]:
```
# ------------------------------------------------------------------------
# Create Strip/Boxplot for both cells and algorithms Cell as described (1 pt)
# hint: you can seperate the algorithms by color
# ------------------------------------------------------------------------

combined_df = pd.concat([df_ogb, df_gcamp], ignore_index=True, sort=False)
sns.stripplot(data=combined_df, x="correlation", y="indicator", hue="algorithm")
```

Out[ ]:
```
<Axes: xlabel='correlation', ylabel='indicator'>
```


algorithm
oopsi
decon
OGB
GCamP
indicator
0.2
0.4
0.6
0.8
correlation