Documentation of the ImageAnal function

Balazs B Ujfalussy

January 18, 2023

The ImageAnal python script includes a few python programs we use to analyse our Ca imaging data. This document is a reference for the functionalities that can be used with that program. We will give detailed descriptions of the functionalities you can use, examples for code blocks and figures showing the results.

Contents

1 Preparation

To analyse our data with our scripts, you need the following components:

1. installed version of python, version 3.7 or more. We recommend the use of anaconda and jupyter or jupyterlab.
2. Your data that you would like to analyse. This includes the behavioral log files and the recorded neuronal data (fluorescence traces or membrane potential recordings). All these data has to be in the right format that we will describe later. Imaging data will be typically preprocessed using suite2p.
3. Our python codebase, ABmice. For now, this can be downloaded from our github repository https://github.com/bbujfalussy/ABmice

In the following, we will start by describing typical use of our analysis script (Section 2), and then provide detailed examples for most of the functions that you can use.

2 Simple example to start

Here we give a simple example of how to load the data and start analysing it. The examples we are giving use `jupyter` notebooks, and some of the graphics settings may be different for other environments. In `jupyter` the code can be structured by using `cells` - which we strongly recommend.

You start by importing the analysis functions and setting the graphics:

```
from ImageAnal import *
%matplotlib widget
```

2.1 Organizing data

You need to define where to look for the data. The required file structure is the following:

- All data should be in the `data` folder
- Within the data folder separate subfolders are needed for each mouse. Folder name starts with the **name** of the mouse.
- For each mouse there should be at least two folders: one for the **imaging** data and one for the **behavioral** data.
- The behavioral folder is named as `MouseName_TaskName` - so we need a separate folder for each different task
- The behavioral log files are in separate subfolders named by the experiment's start time within the behavioral folder - e.g. `2021-02-03_10-15-50`
- The imaging folder is named as `MouseName_imaging`
- The suite2p imaging files are also in separate folders for each experiment below the imaging folder.

```
datapath = os.getcwd() + '/' #current working directory - look for data and strings
    here!
date_time = '2020-03-10_11-35-02' # date and time of the imaging session
name = 'rn018' # mouse name
task = 'contingency_learning' # task name

## locate the suite2p folder
suite2p_folder = datapath + 'data/' + name + '_imaging/rn018_TSeries
    -03102020-0939-002/'

## the name and location of the imaging log file
imaging_logfile_name = suite2p_folder + 'rn018_TSeries-03102020-0939-002.xml'

## the name and location of the trigger voltage file
TRIGGER_VOLTAGE_FILENAME = suite2p_folder + 'rn018_TSeries-03102020-0939-002
    _Cycle00001_VoltageRecording_001.csv'
```

2.2 Loading data

Python looks for the data in the specified folders. It loads the behavioral data (position, lick and rewards) as well as the imaging data. It calculates the activity of the cells as a function of position in the different corridors and calculates their spatial tuning measures and corridor selectivity and various other things.

Importantly, all data that belongs to a given session is stored in a single python object. The typo of the object is `ImagingSessionData`, but you can name it according to your choice. Here we name it as `D1` since it is the first dataset.

```
# 3. load all the data - this taks ~20 secs in my computer
D1 = ImagingSessionData(datapath, date_time, name, task, suite2p_folder,
    imaging_logfile_name, TRIGGER_VOLTAGE_FILENAME)
```

When loading the data python provides a lot of useful information about the data. Paying careful attention to this information can help you in understanding your data, the reasons for analysis failures or finding bugs in the code.

2.3 Data loading options

There are several optional arguments that you can provide when loading your data. Most of these have a default value that you can overwrite when you initially create the object. To list all available options, we can check the definition of the `ImagingSessionData` object:

```
def __init__(self, datapath, date_time, name, task, suite2p_folder,
    imaging_logfile_name, TRIGGER_VOLTAGE_FILENAME, sessionID=np.nan, selected_laps=
    None, speed_threshold=5, randseed=123, elfiz=False, reward_zones=None):
```

In this definition the arguments are in round brackets `(self, datapath, ..., reward_zones=None)`. The first argument (`self`) is a special one, you don't need to worry about it for now. After that there are two types of arguments:

- Arguments **without** default value – no = sign.
- Arguments **with** default value – with = sign.

Here we list all of them and exmplain their meaning:

`datapath` : text, the absolute path where the `data` folder is.

`date_time, name, task` : texts, the date and time in `YYYY-MM-DD_HH-mm-ss` format, name of the animal and the task.

`suite2p_folder` : text, nam eof the suite2p folder

`imaging_logfile_name, TRIGGER_VOLTAGE_FILENAME` text, name of the corresponding files

`sessionID` : optional integer, the ID of the session. Used only for plotting

`selected_laps` : `np.nan` (default) or set of integers containing the list

`speed_threshold` : float, the minimum speed above which the activity is analysed (default: 5 cm/s)

`randseed` : integer, the random seed used for shuffling analysis (default: 123)

`elfiz` : boolean, electrophysiology analysis routines are used when True. default: False

`reward_zones` : optional, a numpy array with rows containing the corridor ID, the start and end of the reward zones [corridor, start, end]. Default: `None`, when zones are read from file. Example: `reward_zones = np.array([[12, 0.904544, 0.981042], [17, 0.904544, 0.981042]])`

2.4 Looking at the data

The loaded `D1` object now contains all your data. As you can imagine, it is quite complex. The main reason to organize it this way is that all data that belongs to this session is stored within the same object. Now you can load multiple sessions and store their data under different names (`D2`, `D3` ...) and the data that belongs to different sessions don't get mixed up.

I will not list all the different variables that are stored in this object. Here I only mention a few useful variables. The easiest way to access them is via printing. The following code prints the number of laps recorded and the ID of the laps with imaging data:

```
print(D1.n_laps)
print(D1.i_Laps_ImData)
```

3 Plotting the behavioral data

3.1 Behavioral data of the whole session

To view the behavioral data for the entire session, you can use the function `D1.plot_session()`. Here we have two fundamentally different options, illustrated in Figure 1. By default, all laps are included, but you can subselect e.g. all laps with imaging data using the option `selected_laps=D1.i_Laps_ImData`:

```
D1.plot_session(selected_laps=D1.i_Laps_ImData)
```

Other possible options include `filename='yourfile.pdf'` which is the name of a file to save the image in pdf format.

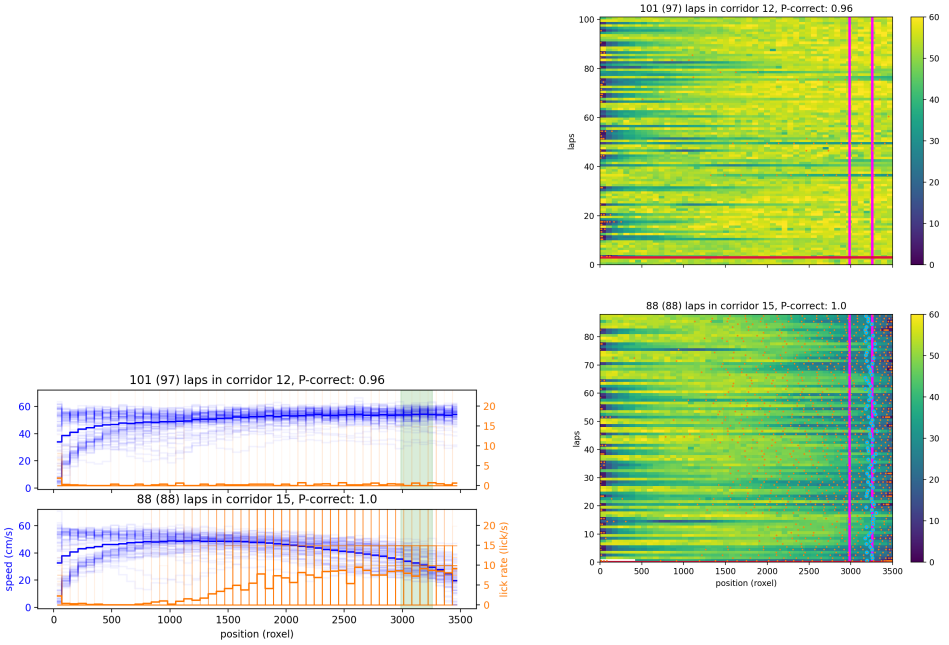

Figure 1: **left**: `D1.plot_session(selected_laps=D1.i_Laps_ImData)` plots the average running speed and lick rate as a function of position separately in the different corridors. On the top are shown the total number of laps, the number of correct laps (in parenthesis), the ID of the corridor and the proportion of correct laps. **right**: `D1.plot_session(selected_laps=D1.i_Laps_ImData, average=False)` plots the running speed in individual laps as an image plot and adds the lick events and the reward locations. Different corridors are plotted separately.

3.2 Behavioral data of a single lap

The `D1` object contains all laps in a **list** called `D1.ImLaps`. Each element of this list is a special object called `Lap_ImData`. There are several plotting functions defined on `Lap_ImData` objects, including a plot of the position, lick and reward as a function of time and speed, lick and reward as a function of position. This is illustrated in Figure 2:

```
D1.ImLaps[197].plot_txv()
```

There are no further options for this function.

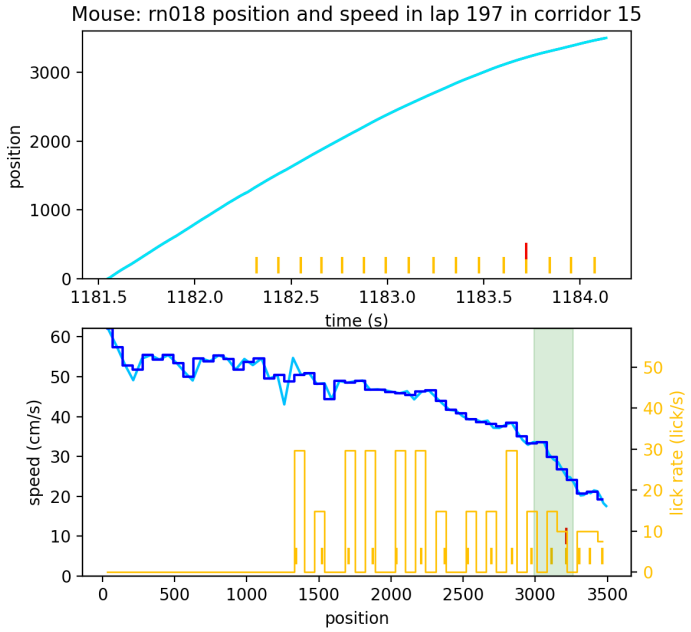

Figure 2: `D1.ImLaps[197].plot_txv()` plots the position, lick and reward as a function of time (top) and the speed, lick and reward as a function of position (bottom).

4 Plotting neuronal data

4.1 plot_properties

The command `D1.plot_properties()` plots the different spatial and non-spatial properties of the recorded cells, separately for the different corridors (fig. 3). The size of the circles (in all spatial subplots) is proportional to the spatial information content of the activity of the cell in the given corridor. There are two options here:

- `D1.plot_properties(cellids=[1,2,3,324])` - You can select a list of cells to be plotted with filled symbols. Default: all cells are filled.
- `D1.plot_properties(interactive=True)` Interactive mode. You can read the cellid by hovering the mouse over the dot.

To select the cells to be plotted, you don't need to manually add the cellid of all cells. Use numpy set routines to create a set of neuronal IDs that you are interested. For example in fig. 3 I used the following code to plot the properties of the cells that are place cells in either of the two corridors (Note, that this requires running the shuffling first - see section 5):

```
place_cells = np.union1d(D1.accepted_PCs[0], D1.accepted_PCs[1]) # Hainmuller PCs in
    the two corridors
D1.plot_properties(cellids=place_cells)
```

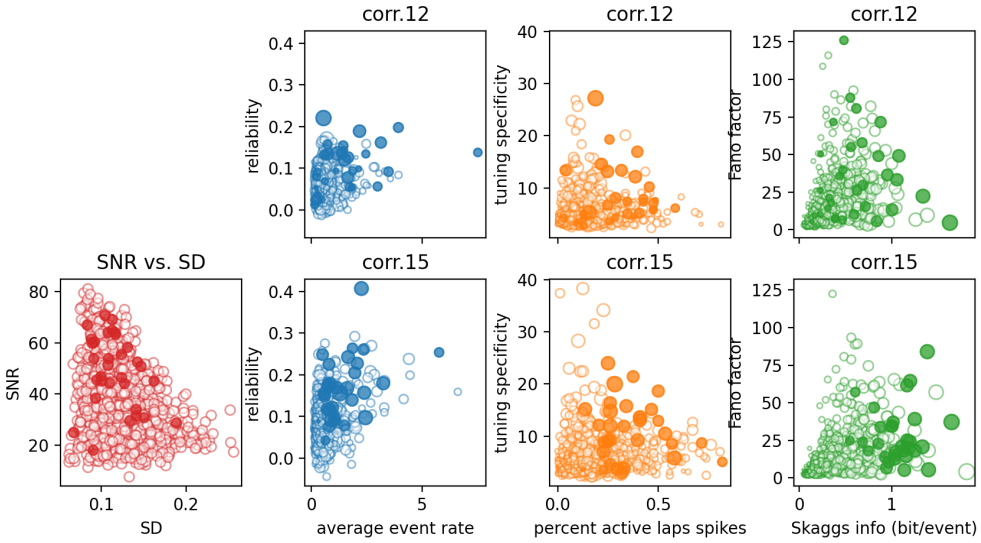

Figure 3: The command `D1.plot_properties()` plots the different spatial and non-spatial properties of the recorded cells, separately for the different corridors. The size of the circles (in all spatial subplots) is proportional to the spatial information content of the activity of the cell in the given corridor.

4.2 speed_vs_activity

The command `D1.speed_vs_activity()` plots the number of cells active in a given lap (left) and the total number of significant events in the DF/F signal (right) as a function of the average speed in that lap for all laps in the session (fig. 4). This command has no further options.

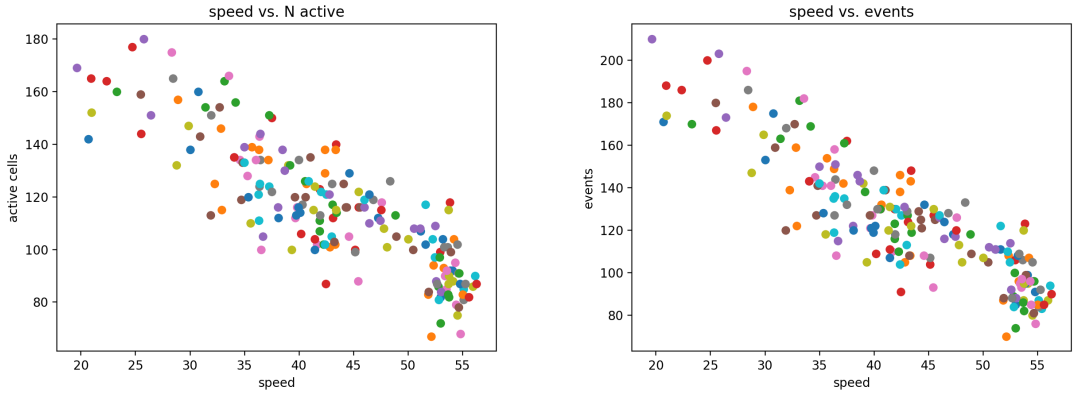

Figure 4: The command `D1.speed_vs_activity()` plots the number of cells active in a given lap (left) and the total number of significant events in the DF/F signal (right) as a function of the average speed in that lap for all laps in the session. Both of these show a strong negativ correlation. This is essentially because with a longer time the probability of a large event increases approximately linearly when the event probability is low.

4.3 plot_cell_laps

The function `plot_cell_laps` plots the activity of a given cell in each lap, separately for the different corridors. It has two basic type of plotting, illustrated in figs. 5 and 6:

- Plotting the raw dF/F trace as a function of time (fig. 5). Each lap has a different duration, so the length of the traces is variable.
- Plotting the raw inferred spike rate as a function of position (fig. 6). We use a 3-bin spatial moving average everywhere on the spatial rates - so these are not the raw counts.

This function has several arguments and optional arguments. You can use the same syntax with either of the two main plotting methods.

- `cellid` The ID of the cell to be plotted.
- `multipdf_object=-1` You can use this argument when you want to create a pdf file with multiple pages, each containing a single cells's data. For further information about multipdf plots, click here.
- `signal='dF'` The variable to be plotted. Should be `'dF'` or `'rate'`
- `corridor=-1` Integer, the ID of the corridor to be plotted. Default is -1 when all corridors are shown.
- `reward=True` Boolean, whether (`True`) or not (`False`) to add the reward information on the plots. Reward is shown with colored square if signal=='dF' or with orange dash when signal=='rate'.
- `write_pdf=False` Boolean, whether or not to save the plot into a pdf object.
- `plot_laps='all'` text, specifying the type of laps to be plotted. Can be either 'all', 'correct' or 'error'.

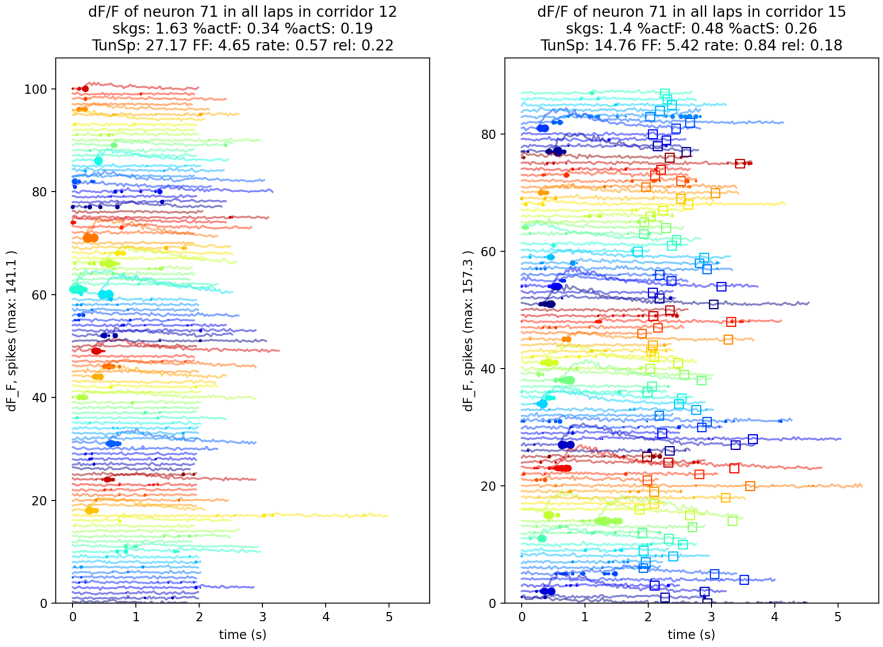

Figure 5: `D1.plot_cell_laps(cellid=71)` Plotting the raw dF/F trace as a function of time. Each lap has a different duration, so the length of the traces is variable.

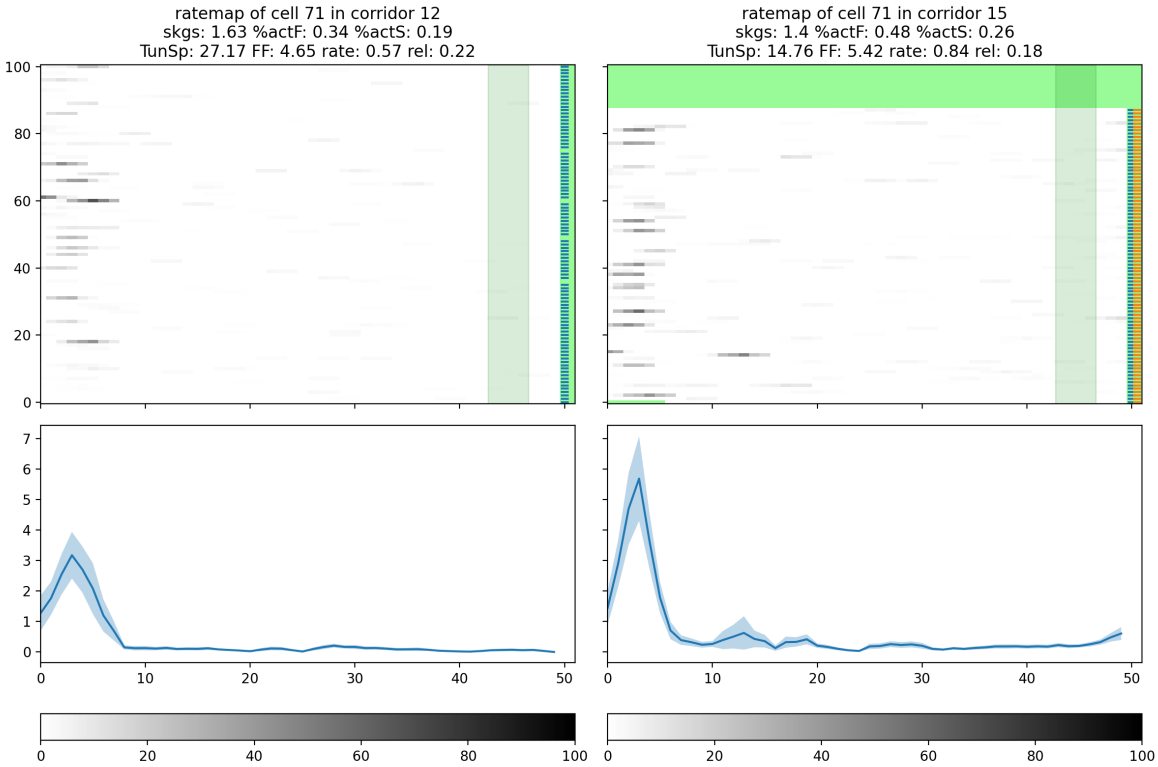

Figure 6: `D1.plot_cell_laps(cellid=71, signal='rate')` Plotting the raw inferred spike rate as a function of position. We use a 3-bin spatial moving average everywhere on the spatial rates - so these are not the raw counts.

4.4 plot_ratemaps

This function plots the ratemaps (firing rate as a function of position) of the recorded cells, separately for the different corridors. By default, it plots all neurons unnormalized and unsorted activity in both corridors averaged across all laps in a given corridor. There are many options that enable to select the plotted cells, sort them or normalize the activity of the cells by their maximum. Finally, it is also possible to use this function to plot ratemaps constructed by other functions - eg., `calc_even_odd_rates` (Section 6.2) or `calc_start_end_rates` (Section 6.3).

The function also returns the index used for sorting the cells.

Here we provide 3 examples: The default behaviour, plotting unsorted, unnormalized ratemaps of all cells tuned in corridor 15 (Figure 7, left), sorted and normalized ratemaps of the same cells (Figure 7, right) and sorted and normalized ratemaps calculated from even and odd laps separately for the same cells (Figure 8). The code for calculating and plotting ratemaps separately for even and odd laps is given in the following box:

```
D1.calc_even_odd_rates()
rmaps = [D1.ratemaps_odd[0], D1.ratemaps_even[0], D1.ratemaps_odd[1], D1.ratemaps_even
    [1]]
tcs = D1.plot_ratemaps(normalized=True, sorted=True, cellids=D1.tuned_cells[1],
    ratemaps_array=rmaps, ratemaps_title=['odd 12', 'even 12', 'odd 15', 'even 15'],
    corridor=[12, 12, 15, 15])
```

The function has the following options:

- `corridor=-1`: integer or array of corridor ids. By default, `corridor=-1` when activity in all corridors are shown using all laps in that corridor. You can specify a single corridor (eg. `corridor=12`) and then only that corridor is plotted. **Importantly**: when you want to plot ratemaps are provided the corresponding corridors must be specified as a list (e.g., `corridor=[12, 12, 15, 15]`) so that the reward zone can be added.
- `normalized=False`: Boolean. If True then each cell ratemap is normalized to have a max = 1.
- `sorted=False` Boolean: when True, sorts the ratemaps by their peaks
- `corridor_sort=-1` Integer. which corridor to use for sorting the ratemaps. It is the ID of the if you plot default ratemaps or the index of ratemap if you plot custom-defined ratemaps (when multiple maps can belong to the same corridor). By default, it is -1, when the first plotted corridor is used for sorting.
- `cellids=np.array([-1])`: np array with the indexes of the cells to be plotted. when -1: all cells are plotted.
- `vmax=0`: float. If ratemaps are not normalised then the max range of the colors will be at least vmax. If one of the ratemaps has a higher peak, then vmax is replaced by that peak
- `ratemaps_array=[]`: an array containing the ratemaps to visualize, if custom ratemaps are to be plotted. By default it is empy and default ratemaps are plotted.
- `ratemaps_title=[]` an array containing string which is used for annotating the corresponding ratemaps. Must have same length as `ratemaps_array`. Only needed if `ratemaps_array` is given. By default it is empy and default ratemap titles are used.
- `filename = None`: optional string. The name of the pdf file to save the figure

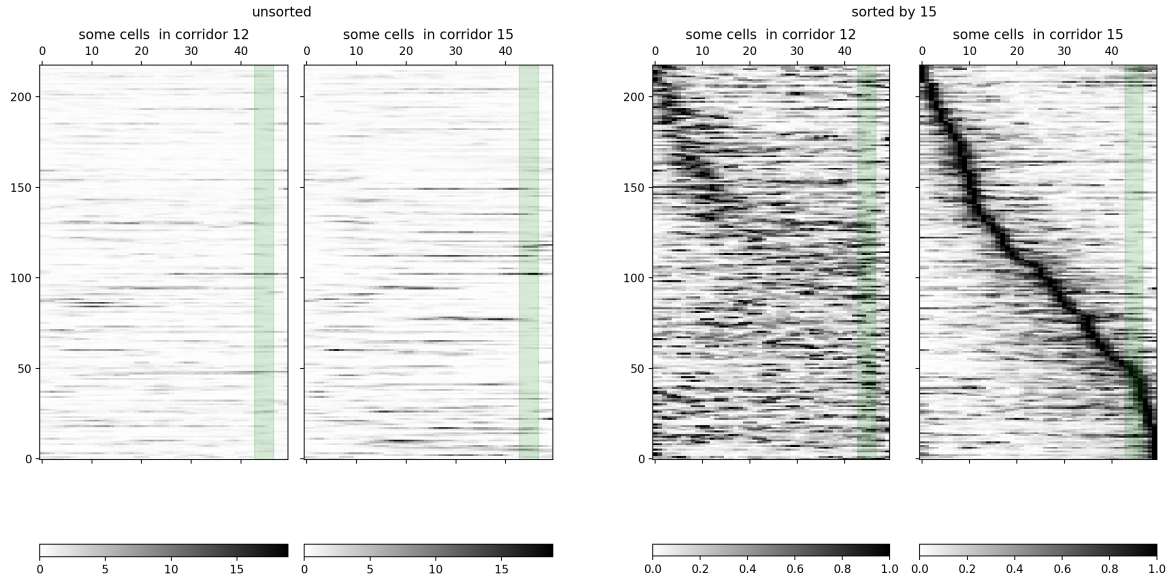

Figure 7: Left: `D1.plot_ratemaps(cellids=D1.tuned_cells[1])`.
Right: `D1.plot_ratemaps(normalized=True, sorted=True, cellids=D1.tuned_cells[1], corridor_sort=15)` Note, that the numbers on the right are not the cellIDs, just cell counts.

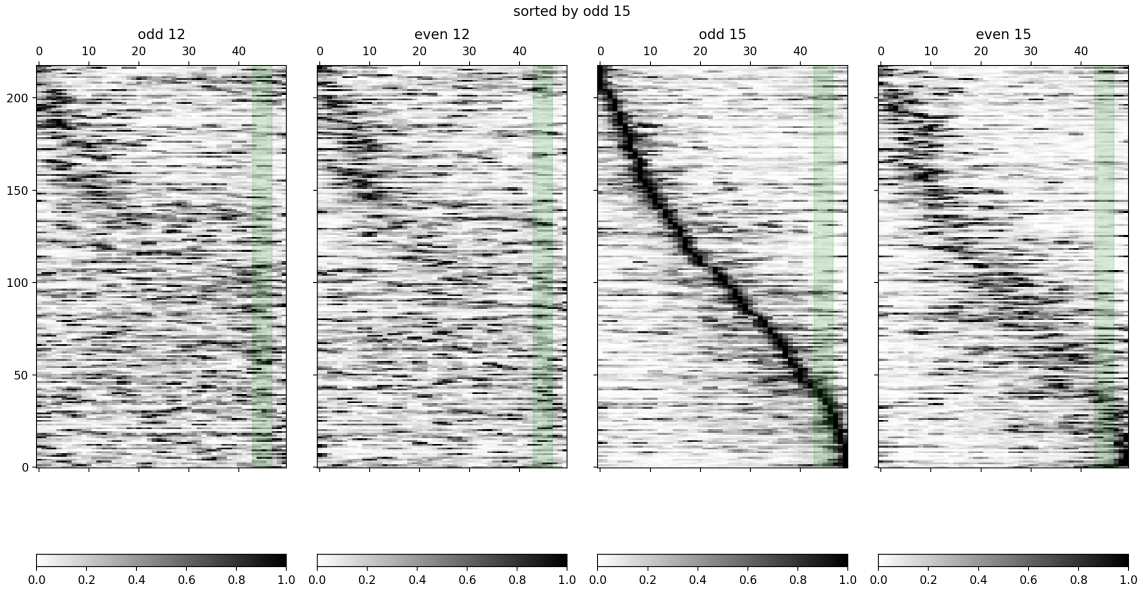

Figure 8: `D1.plot_ratemaps(normalized=True, sorted=True, cellids=D1.tuned_cells[1], ratemaps_array=rmaps, ratemaps_title=['odd 12', 'even 12', 'odd 15', 'even 15'], corridor=[12, 12, 15, 15], corridor_sort=2)` Cells are sorted according to odd laps in corridor 15. The same row is the same cell in each subplot. The strong diagonal line in the odd 15 subplot is an artefact of sorting. The real strength of the diagonal is better represented by the 4th subplot, even 15.

4.5 plot_popact

Plots the total population activity (sum of the neurons estimated firing rate) as a function of position within the track. There are two different ways of plotting: either by averaging across laps (Figure 9, top) or by plotting the population activity lap by lap (Figure 9, bottom). Options:

- `cellids`: numpy vector. The index of the cells included in the population activity.
- `corridor = -1`: integer, the ID of a valid corridor. When corridor = -1 then all corridors are used.
- `bylaps`: Boolean. When True, population activity is plotted lap by lap.
- `name_string='selected_cells'`: not used
- `set_ymax=None`: the maximum of the y-axis range, when plotting the total population activity

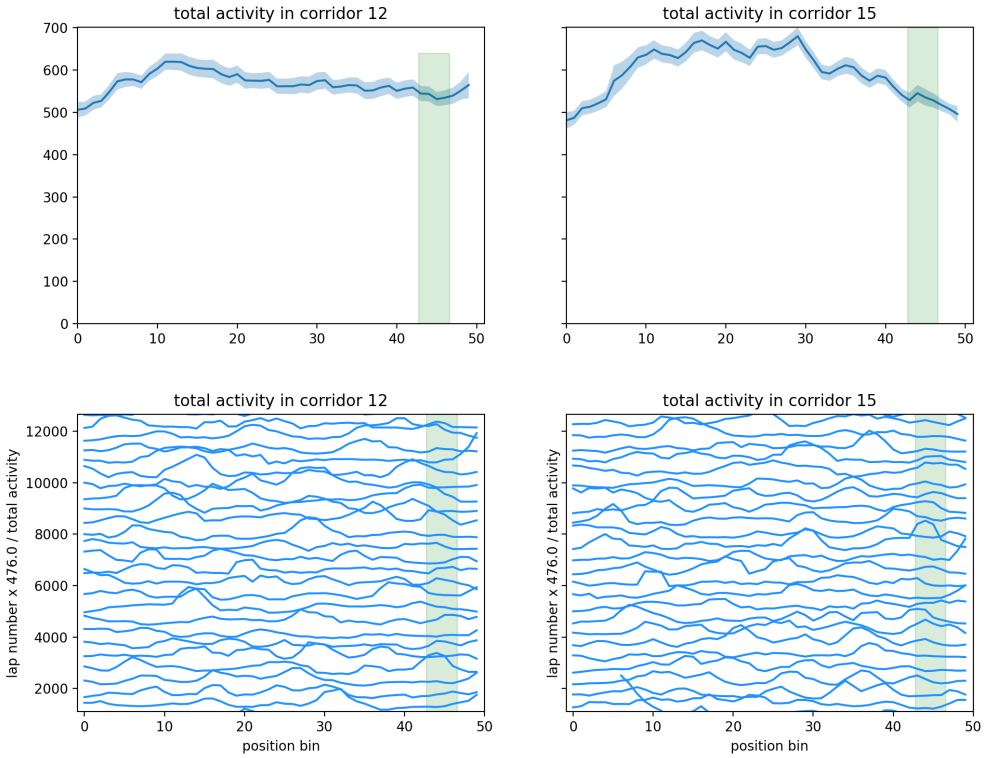

Figure 9: Top: `D1.plot_popact(cellids=np.arange(D1.N_cells))` Average of the population activity in the two corridors. Shaded area shows the standard error of mean across laps.
Bottom: `D1.plot_popact(cellids=D1.active_cells, bylaps=True)` Population activity in individual laps. Activity in each lap is shown on the original scale, and is shifted by an amount proportional to the lap number. The proportionality constant is shown in the y-axis label – here it is 476. Only a few laps are shown here.

4.6 plot_masks

Plots the location of the ROIs belonging to each cell. It requires the `ops.npy` saved by Suite2P. It plots both the full ROIs on the left and the nonoverlapping parts on the right. Cells are colored according to their cellID (default) or according to any property given as an argument.

This is how I would try this function to plot the ROI of the active cells colored by their reliability in the first corridor (I don't have the `ops.npy` file, so it is not actually working for me):

```
D1.plot_masks(cellids=D1.active_cells, cell_property=D1.cell_reliability[0],
    title_string='colored by reliability')
```

- `cellids`: numpy vector. The index of the cells included in the population activity.

- `cell_property=np.array([np.nan])`: an optional numpy array with the property of the cells used for coloring. Could be used to test whether some property has a systematic bias - more spatially tuned cells are in the centrer or perifery. By default the cellID is used for coloring.
- `title_string=''`: optional text in the title of the plot.

4.7 plot_dF_lapstarts

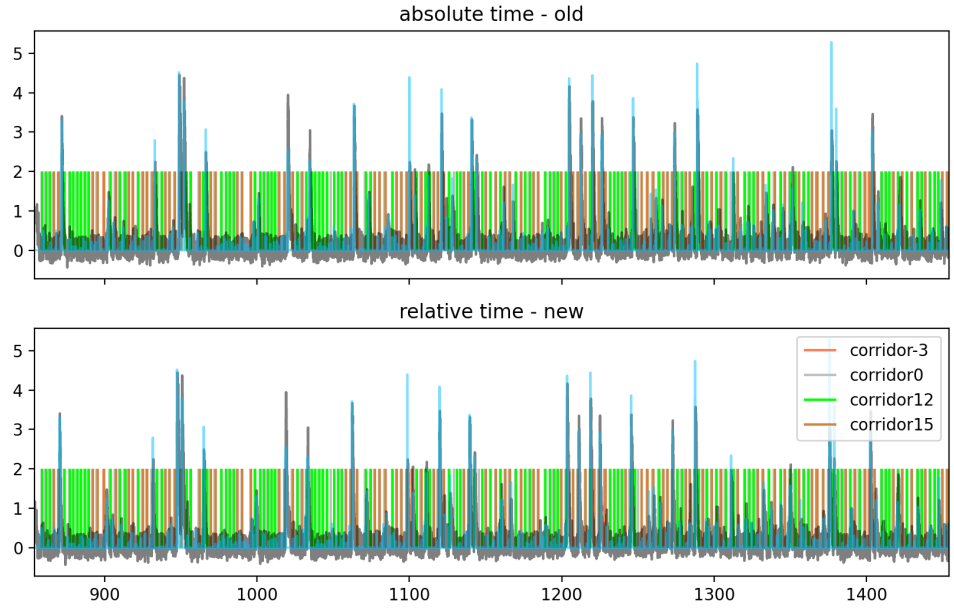

Figure 10: `D1.plot_dF_lapstarts(cellid=71)` Plotting the raw fluorescence trace of a single ROI together with the start of the start of the laps in the different corridors according to the trigger signal. This was used for debugging.

4.8 show_autocorr and show_crosscorr

The functions `show_autocorr` and `show_crosscorr` are deisigned to show the similaritiy between the neuronal activity at different locations. Here the similarity is measured by Pearson's correlation and the neuronal activity is the average (across laps) firing rate of the neurons. The autocorrelation is calculated between different positions within the same ratemap or corridor. Cross-correlation is calculated between two ratemaps. The two ratemaps can come from the same (even versus odd or early versus late) corridor or from different corridors.

Figure 11 shows three examples. The top panel shows the autocorrelation of the ratemap in corridor 12 calculated from the even laps. To plot this, you need to create the corresponding ratemap using the `D1.calc_even_odd_rates()` command (Section 6.2).

The `show_autocorr` function has the following arguments:

- `ratemap`: N_bins × N_cells numpy array, the ratemap used to calculate the autocorrelation.
- `cellids`: numpy vector. The index of the cells included in the population activity.
- `title_string=''`: optional text in the title of the plot.

Figure 11 bottom two panels shows cross-correlation between ratemaps calculated from even and odd laps from the same corridor (left) and even laps of two different corridors (right). What you need to notice is that values in the diagonal of the even versus odd correlation are much lower than the same entries in the even autocorrelation.

The `show_crosscorr` function has the following arguments:

- `ratemap1`: N_bins × N_cells numpy array, the first ratemap used to calculate the autocorrelation.
- `ratemap2`: N_bins × N_cells numpy array, the second ratemap used to calculate the autocorrelation.
- `cellids`: numpy vector. The index of the cells included in the population activity.
- `ratemap1_annot='map 1'`: text for annotating the x axis
- `ratemap2_annot='map 2'`: text for annotating the y axis
- `main_title='Cross correlation'`: optional text in the title of the plot.
- `return_matrix=False`: Boolean, when True the matrix is returned.
- `plot_ccm=True`: Boolean, when True, the matrix is plotted.

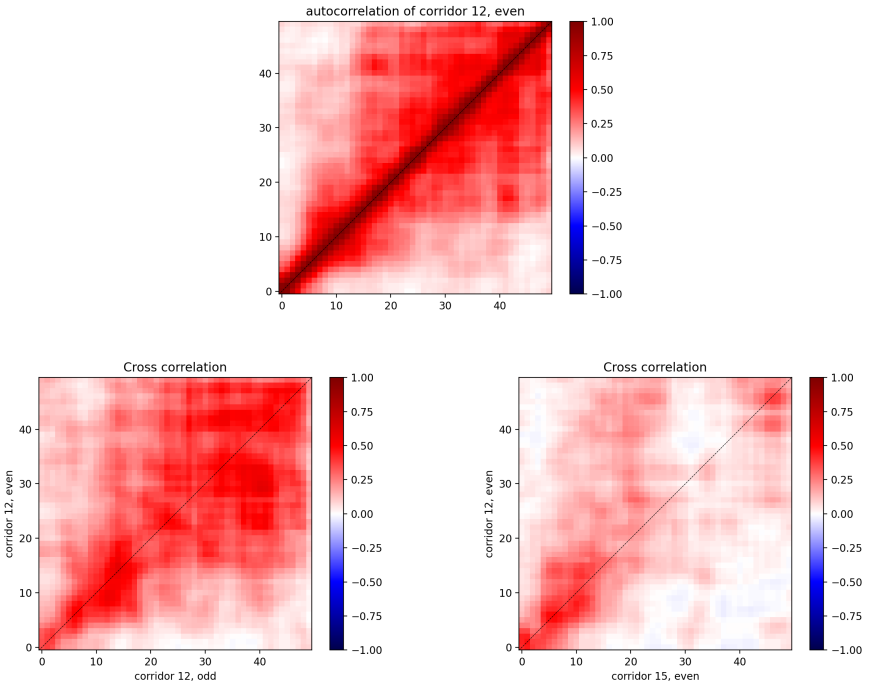

Figure 11: Left: `D1.show_autocorr(D1.ratemaps_even[0], title='autocorrelation of corridor 12, even')`
Middle: `D1.show_crosscorr(D1.ratemaps_even[0], D1.ratemaps_odd[0], ratemap1_annot='corridor 12, even', ratemap2_annot='corridor 12, odd')`
Right: `D1.show_crosscorr(D1.ratemaps_even[0], D1.ratemaps_even[1], ratemap1_annot='corridor 12, even', ratemap2_annot='corridor 15, even')`

4.9 lap_decode

This function plots the correlation between a single lap neuronal activity matrix (N_bins × N_cells) and the average neuronal activity (aka. ratemap) for for corridors and for each lap separately. This can be used as a simple workaround for decoding corridor identity from neuronal activity – the corridor with the highest correlation is the most likely candidate. Note, that to obtain reliable estimates for decoding performance you would need cross-validation, which is not easily implemented here.

Figure 12 shows two different usage of this function. The top panel is the default mode, when all laps are used to construct the template ratemaps and empty circles indicate the identity of the corridor in a given lap. Although correlations are not high, the empty circles are almost always above the filled ones, indicating that activity better correlates with the true template.

Figure 12 bottom panels use only the first half of the laps to construct the templates ratemaps. This bottom left shows the entire session and illustrates the necessity of cross/validation: the correlation drops sharply at the middle. Note, that in this case the template ratemaps are given to the plotting function as arguments, and the true corridor identities are not shown. This is because th program can not know what corridors the provided templates correspond to. For example, in some case the templates might correspond to the start and the end of the same corridor. The program below gives you an example to add the true corridor information as colored vertical lines to this plot (Figure 12 bottom right). The function also returns the correlations calculated as a numpy array (`lap_corrs` in line 3 of the code below) that you can use with your own plotting function.

```
1  # Code to calculate correlation between activity in individual laps and the average ('
       ratemap') calculated from only the first half of the session
2  D1.calc_start_end_rates() # First, we
3  lap_corrs = D1.lap_decode(cellids=D1.active_cells, ratemaps=D1.ratemaps_start, labels
       =['start - 12', 'start - 15'], title='lap by lap correlation')
4  i_corrs = D1.i_corridors[D1.i_Laps_ImData]
5  N_laps = len(i_corrs)
6  lap_cols = ['tab:blue'] * N_laps
7  for i in np.arange(N_laps):
8      if (i_corrs[i] == 15):
9          lap_cols[i] = 'tab:orange'
10 plt.vlines(np.arange(N_laps), 0, 50, linewidths=0.5, colors=lap_cols)
```

This function has the following arguments:

- `cellids`: numpy vector. The index of the cells included in the population activity.
- `ratemaps=None`: a list containing the template ratemaps. Each ratemap is an N_bins × N_cells numpy array. At least 2 ratemaps have to be provided.
- `labels=None`: a list containing the labels for the template ratemaps. Has to be the same length as the ratemaps list.
- `title=''`: optional text in the title of the plot.

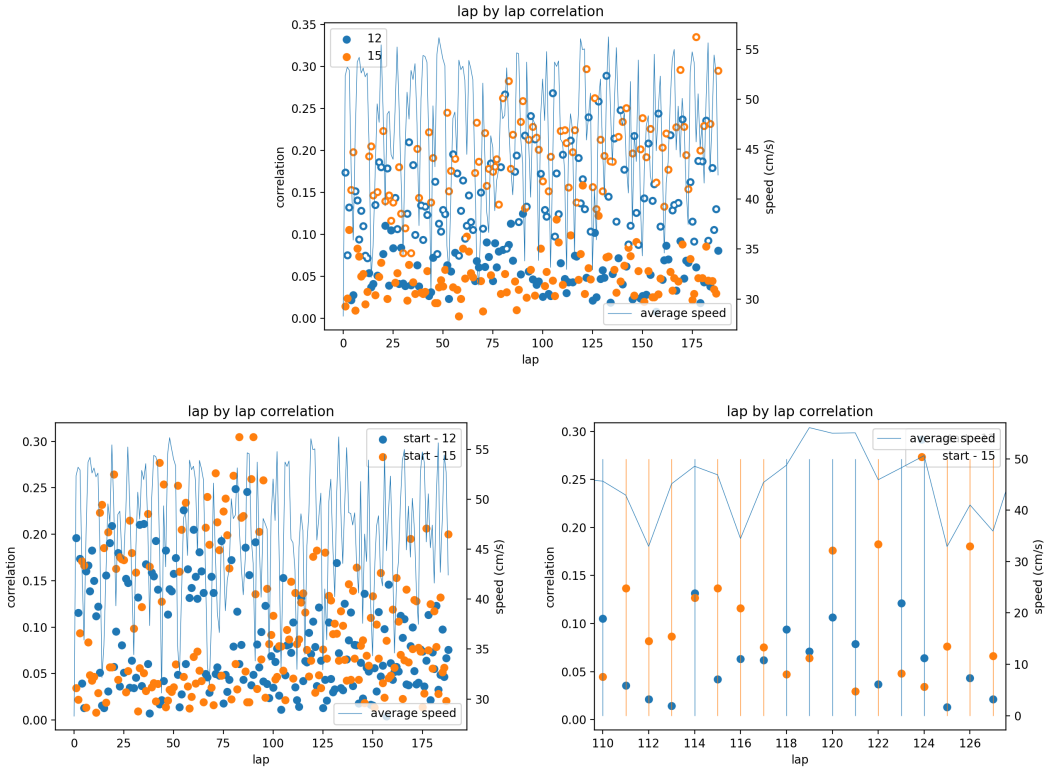

Figure 12: Top: `D1.lap_decode(cellids=D1.active_cells, title='lap by lap correlation')`
Left: `D1.lap_decode(cellids=D1.active_cells, ratemaps=D1.ratemaps_start, labels=['start - 12', 'start - 15'], title='lap by lap correlation')` Note thet sharp decrease of the maximum correlations after $\sim$ 90 laps – the templates are based on the first half of the session. Also note, that by default the true lap ID is not indicated.
Right: Magnified part of the plot on the left, with the true corridor IDs indicated by the color of the vertical lines. Note that lap 120 is incorrectly decoded as from corridor 12.

4.10 lap_correlate

Plot the correlation between the neuronal activity matrices (N_bins × N_cells) in different laps.
This function has the following arguments:

- `cellids`: numpy vector. The index of the cells included in the population activity.
- `filename=None`: the plot is saved into a pdf file if not None
- `corridors=None`: a list of integers containing corridor IDs. If None, then all corridors are used. If not None, only the corridors included here are used on the right hand side plot, when laps are ordered by corridor id.
- `normalize_rates=False`: Boolean. When Ture, rates of individual cells are normalized between 0 and 1.
- `add_switch_ordered=False`: Boolean. When True, the switch lap is also indicated in the rigth plot, when laps are ordered by corridor id.

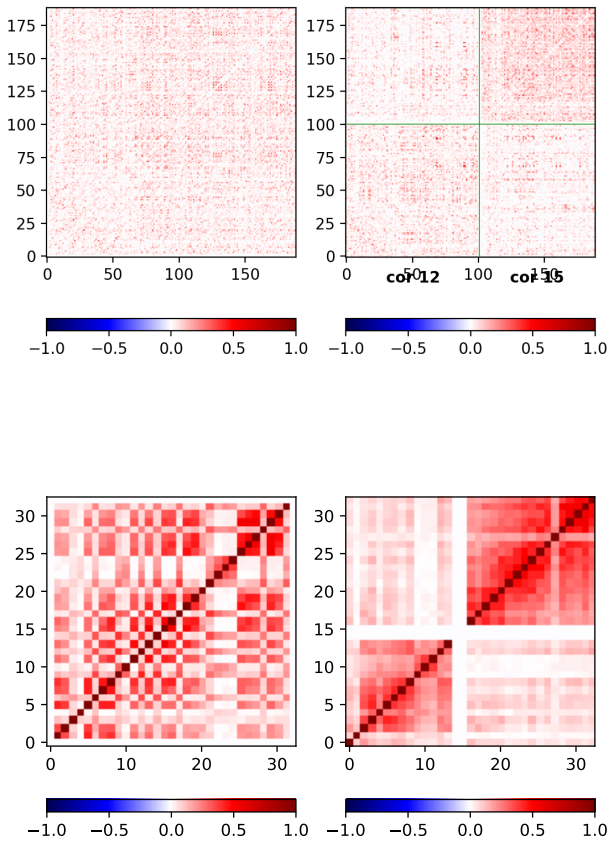

Figure 13: Top: `D1.lap_correlate(cellids=D1.tuned_cells[1])` The diagonal (self-similarity) is shown in white.
Bottom: Similar to Top, on a different dataset, where the neuronal activity is more distinctive - laps from the same corridor show high correlation, and the correlation matrix has a checkerboard structure. This plot was generated with an earlier version of the function and the diagonal (self-similarity) is shown in dark red.

4.11 plot_xv and plot_tx

There are two functions for plotting behavioral and neuronal activity together in a single lap. The first one plots these variables as a function of time (`plot_tx()`, Figure 14, Left) and the second one as a function of space (`plot_xv()`, Figure 14, Right). These functions belong to the `Lap_ImData` objects stored in the list `D1.ImLaps` as described in Section 3.2. You can call these functions on laps with no imaging data - then the second part of the plot will remain empty.

The function `plot_tx()` has two optional arguments:

- `fluo=False`: Boolean. When True, the raw fluorescence is also shown for some cells.
- `th=25`: Float, threshold for plotting the fluorescence data - only cells with spikes ¿ th are shown

The function `plot_xv()` has no arguments.

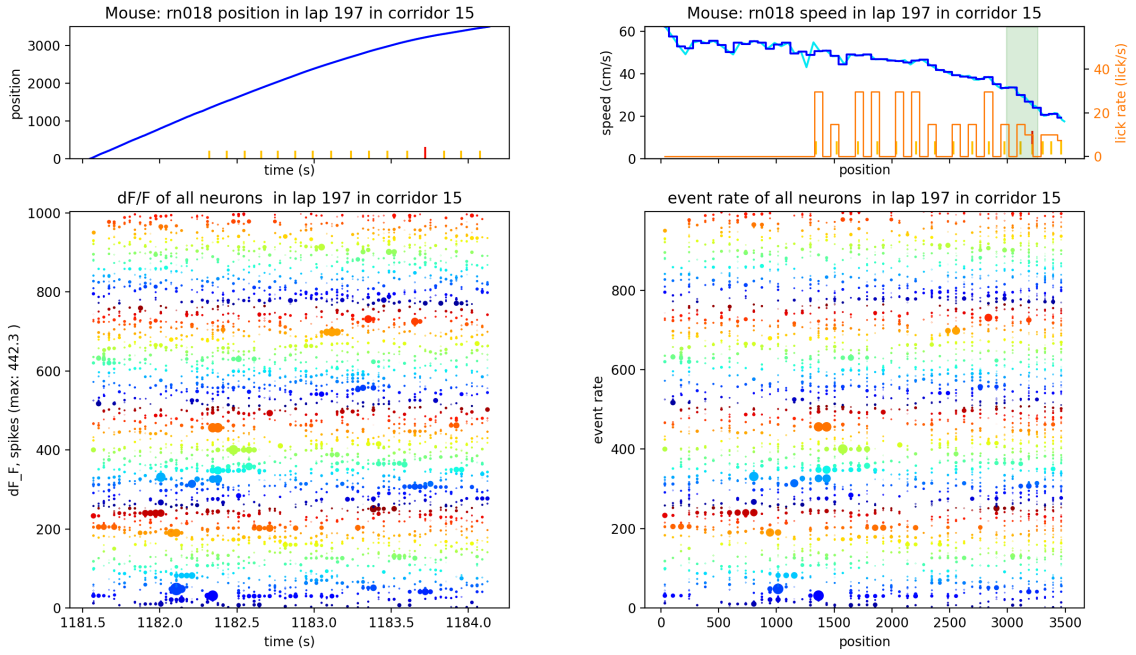

Figure 14: Left: `D1.ImLaps[197].plot_tx()` Top: The blue line is the position as a function of time. Orange ticks are licks and red tick is the reward. Bottom: Estimated spike time of the neurons - the size of each dot is proportional to spike count.
Right: `D1.ImLaps[197].plot_xv()` Top: The blue line is the speed as a function of position. Orange ticks are licks and red tick is the reward. Bottom: Estimated spikes of the neurons - the size of each dot is proportional to firing rate.

5 Shuffling

We use shuffling to determine the significance of the tuning of the cells. Shuffling is performed using a simple function that handles all underlying computations and saves the data. An example code is shown below:

```
cellids = D1.active_cells
D1.calc_shuffle(cellids, 1000, 'shift', batchsize=50)
place_cells = np.union1d(D1.accepted_PCs[0], D1.accepted_PCs[1])
```

The shuffling procedure creates a lots of mock data that needs a lots of memory and thus takes a long time to complete. You can reduce the memory need by testing only selected cells with selections based on properties unrelated to the tested quantities – e.g., activity.

The summary results of the shuffling analysis are saved into a file (`shuffle_stats....csv` in the folder named `analysed_data` in the `suite2p` folder) and can be read from file later when the same shuffling analysis is performed again on the same dataset. If any of the parameters is changed, then a new shuffling will be computed.

The function `calc_shuffle` has several arguments that control the exact behavior of the shuffling:

- `cellids`: numpy vector. The index of the cells tested for significance during shuffling.
- `n=1000`: The number of shuffles used to estimate significance. Both the speed of the execution and the accuracy are proportional to n. Low n (~ 100) can be used for initial testing, but high n (≥ 1000) is recommended for final analysis.
- `mode='shift'`: text, 'shift' or 'random'. When 'random' than spike times are totally randomized. When 'shift', then spike times are broken into 6 segments of at least 600 frames, permuted and circularly shifted to maintain firing patterns but break the relationship between neuronal activity and behavior.
- `batchsize=25`: Integer, defining the number of neurons involved in simultaneous calculations. It only influences the speed of execution but not the results. Should be the highest number below the memory limits of the computer.
- `verbous=1`: Integer controlling the amount of feedback - higher, more feedback.
- `name_string=''`: optional text saved in the file to specify the peculiarities of the analysis. This can be used when the same session is analysed multiple times.

The random seed used for the shuffling analysis is set by the `randseed` parameter of the `ImagingSessionData` object. By using the same random seed you can ensure that the same random spike patterns are generated when repeating the shuffling procedure.

After shuffling, we perform the following tests for each neurons in each corridor: Skaggs tuning, tuning specificity, spatial reliability and Hainmuller place cell criteria. We also test for the corridor selectivity and similarity. In Rita's contingency learning experiment we calculate corridor selectivity in 4 different zones separately. We use Holm–Bonferroni method to correct for multiple comparisons.

6 Some useful functions

6.1 get_lap_indexes

This function helps you to find the index of a particular lap in a corridor. I will show it in an example. Assume you see that cell 71 is doing something interesting in lap 60 in corridor 12 (Figure 6). You want to have a look at the behavior and other cell's activity in the same lap as plotted in Figure 14. So what is the index of lap 60 in corridor 12 amongst the total laps that also include laps before imaging and laps run in other corridors? This is what the function `get_lap_indexes` tells you.

```
D1.get_lap_indexes(corridor=12, i_lap=60)
# output:
# lap # in corridor 12 with imaging data;    lap # within session
60   204
```

The first argument is the corridor and the second is the lap. It prints the required index (204), that you can use for further analysis (eg., `D1.ImLaps[204].plot_tx(fluo=True)`).

6.2 calc_even_odd_rates

```
D1.calc_even_odd_rates()
```

Simply estimates the ratemaps separately from odd and even laps within each corridor separately. The results are saved in two lists within the `ImagingSessionData` object, `self.ratemaps_even` and `self.ratemaps_odd`.

6.3 calc_start_end_rates

```
D1.calc_start_end_rates(n_used=-1)
```

Simply estimates the ratemaps separately using laps from the first or last part of the session within each corridor separately. The results are saved in two lists within the `ImagingSessionData` object, `self.ratemaps_start` and `self.ratemaps_end`. The parameter `n_used` controls how many laps are used. By default (when `n_used=-1`) half of the laps are in the first, and the other half in the second. Otherwise the first `n_used` laps are included in the `self.ratemaps_start`, and the last `n_used` laps are included in the `self.ratemaps_end`.

7 Saving the data

The function `save_data` saves some of the processed data into `csv` files for further analysis. All data are saved in the folder named `analysed_data` in the `suite2p` folder. The following arguments can be used to control the kind of data saved:

- `save_properties=True`: Boolean. When True, the spatial tuning properties are saved into separate files per corridor (Figure 3).
- `save_ratemaps=True`: Boolean. When True, the ratemaps of all cells (Figure 7) are saved into csv files. Separate files per corridor.
- `save_laptime=True`: Boolean. When True, the time, position, speed, lick, reward and spikes of the 998 cells are saved into separate files for each lap.
- `save_lick_speed_stats=True`: Boolean. Prepares an array that contains all the behavioral measures of the session. Each row is a separate lap. The first 5 columns are: 0: lap number, 1: corridor ID, 2: correct, 3: reward, 4: imaging available. The remaining columns are speed and lick rate in the corresponding spatial bins.
- `save_place_code_stats=True`: Boolean. When True, percent of tuned cells per position bins in the different corridors and the correlation between the ratemaps of all tuned cells across the first two corridors is saved.
- `plot=False'`: Boolean, when True, the `plot_session` (Figure 1) command is invoked if `save_lick_speed_stats=True`. It also plots the place code stats as in Figure 15 if `save_place_code_stats=True`.

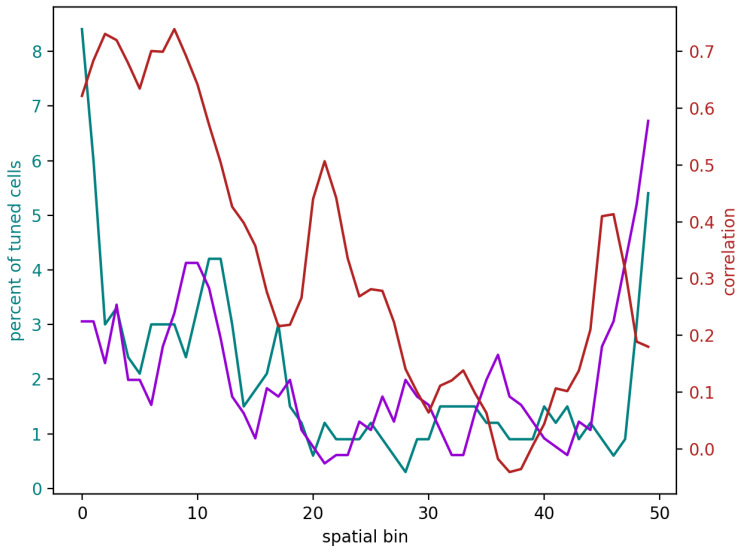

Figure 15: Left: `D1.save_data(plot=True)` Percent of tuned cells as a function of position (left axis) and the correlation between the ratemaps (right axis). The high fraction in the first and the last few bins is probably an artefact of spatial smoothing.