Kilosort4

Release 0.0.1

Jacob Pennington (documentation)

May 17, 2024

GETTING STARTED

Kilosort4 is a Python package for spike sorting on GPUs with template matching. The software uses new graph-based approaches to clustering that improve performance compared to previous versions. For detailed comparisons to past versions of Kilosort and to other spike-sorting methods, please see our pre-print on BioArxiv.

CHAPTER

ONE

KILOSORT4

You can run Kilosort4 without installing it locally using google colab. An example colab notebook is available here. It will download some test data, run kilosort4 on it, and show some plots. Talk describing Kilosort4 is here.

Example notebooks are provided in the `docs/source/tutorials` folder and in the docs. The notebooks include:

1. `basic_example`: sets up run on example data and shows how to modify parameters
2. `load_data`: example data format conversion through SpikeInterface
3. `make_probe`: making a custom probe configuration.

If you use Kilosort1-4, please cite the paper:
Pachitariu, M., Sridhar, S., Pennington, J., & Stringer, C. (2024). Spike sorting with Kilosort4.

Note on multi-shank probes : We are aware of some issues with sorting data from probes with multiple shanks or 2D arrays. See documentation here for recommended workarounds until the code is updated to handle these probes.

Note on reusing parameters from previous versions: This probably will not work well. Kilosort4 is a new algorithm, and the main parameters (the thresholds) can affect the results in a different way for your data. Please start with the default parameters and adjust from there based on what you see in Phy.

Warning :bangbang:: There were two bugs in Kilosort 2, 2.5 and 3 (but not 4) which caused fewer spikes to be detected in ~7ms periods at batch boundaries (every 2.1866s, issue #594). The patch1 releases fix these bugs, please use the new default NT and ntbuff parameters.

1.1 Installation

1.1.1 System requirements

Linux and Windows 64-bit are supported for running the code. At least 8GB of GPU RAM is required to run the software (see docs for more recommendations). The software has been tested on Windows 10 and Ubuntu 20.04.

1.1.2 Instructions

If you have an older `kilosort` environment you can remove it with `conda env remove -n kilosort` before creating a new one.

1. Install an Anaconda distribution of Python. Note you might need to use an anaconda prompt if you did not add anaconda to the path.
2. Open an anaconda prompt / command prompt which has `conda` for **python 3** in the path
3. Create a new environment with `conda create --name kilosort python=3.9`. Python 3.10 should work as well.
4. To activate this new environment, run `conda activate kilosort`
5. To install kilosort and the GUI, run `python -m pip install kilosort[gui]`. If you're on a zsh server, you may need to use ' ' around the kilosort[gui] call: `python -m pip install 'kilosort[gui]'.
6. Instead of 5, you can install the minimal version of kilosort with `python -m pip install kilosort`.
7. Next, if the CPU version of pytorch was installed (will happen on Windows), remove it with `pip uninstall torch`
8. Then install the GPU version of pytorch `conda install pytorch pytorch-cuda=11.8 -c pytorch -c nvidia`

Note you will always have to run `conda activate kilosort` before you run kilosort. If you want to run jupyter notebooks in this environment, then also `conda install jupyter` or `pip install notebook`, and `python -m pip install matplotlib`.

1.1.3 Debugging pytorch installation

If step 8 does not work, you need to make sure the NVIDIA driver for your GPU is installed (available here). You may also need to install the CUDA libraries for it, we recommend CUDA 11.8.

If pytorch installation still fails, follow the instructions here to determine what version of pytorch to install. The Anaconda install is strongly recommended on Windows, and then choose the CUDA version that is supported by your GPU (newer GPUs may need newer CUDA versions > 10.2). For instance this command will install the 11.8 version on Linux and Windows (note the `torchvision` and `torchaudio` commands are removed because kilosort doesn't require them):

```
conda install pytorch pytorch-cuda=11.8 -c pytorch -c nvidia
```

This video has step-by-step installation instructions for NVIDIA drivers and pytorch in Windows (ignore the environment creation step with the .yml file, we have an environment already, to activate it use `conda activate kilosort`).

1.2 Running kilosort

1. Open the GUI with `python -m kilosort`
2. Select the path to the binary file and optionally the results directory. We recommend putting the binary file on an SSD for faster processing.
3. Select the probe configuration (mat files recommended, they actually exclude off channels unlike prb files)
4. Hit `LOAD`. The data should now be visible.
5. Hit `Run`. This will run the pipeline and output the results in a format compatible with Phy, the most popular spike sorting curating software.

1.2.1 Debugging qt.qpa.plugin error

Some users have encountered the following error (or similar ones with slight variations) when attempting to launch the Kilosort4 GUI:

```
QObject::moveToThread: Current thread (0x2a7734988a0) is not the object's thread␣
↪(0x2a77349d4e0).
Cannot move to target thread (0x2a7734988a0)

qt.qpa.plugin: Could not load the Qt platform plugin "windows" in "<FILEPATH>" even␣
↪though it was found.
This application failed to start because no Qt platform plugin could be initialized.␣
↪Reinstalling the application may fix this problem.

Available platform plugins are: minimal, offscreen, webgl, windows.
```

This is not specific to Kilosort4, it is a general problem with PyQt (the GUI library we chose to develop with) that doesn't appear to have a single cause or fix. If you encounter this error, please check issue 597 and issue 613 for some suggested solutions.

1.3 Integration with Phy GUI

Phy provides a manual clustering interface for refining the results of the algorithm. Kilosort4 automatically sets the "good" units in Phy based on a <10% estimated contamination rate with spikes from other neurons (computed from the refractory period violations relative to expected).

Check out the Phy repo for more install instructions. We recommend installing Phy in its own environment to avoid package conflicts.

After installation, activate your Phy environment and navigate to the results directory from Kilosort4 (by default, a folder named `kilosort4` in the same directory as the binary data file) and run:

```
phy template-gui params.py
```

1.4 Developer instructions

To get the most up-to-date changes to the code, clone the repository and install in editable mode in your Kilosort environment, along with the other installation steps mentioned above.

```
git clone git@github.com:MouseLand/Kilosort.git
conda activate kilosort
pip install -e Kilosort[gui]
pip uninstall torch
conda install pytorch pytorch-cuda=11.8 -c pytorch -c nvidia
```

For unit testing, you will need to install pytest

```
pip install pytest
```

Then run all tests with:

```
pytest --runslow
```

To run on GPU:

```
pytest --gpu --runslow
```

Omitting the `--runslow` flag will only run the faster unit tests, not the slower regression tests.

CHAPTER

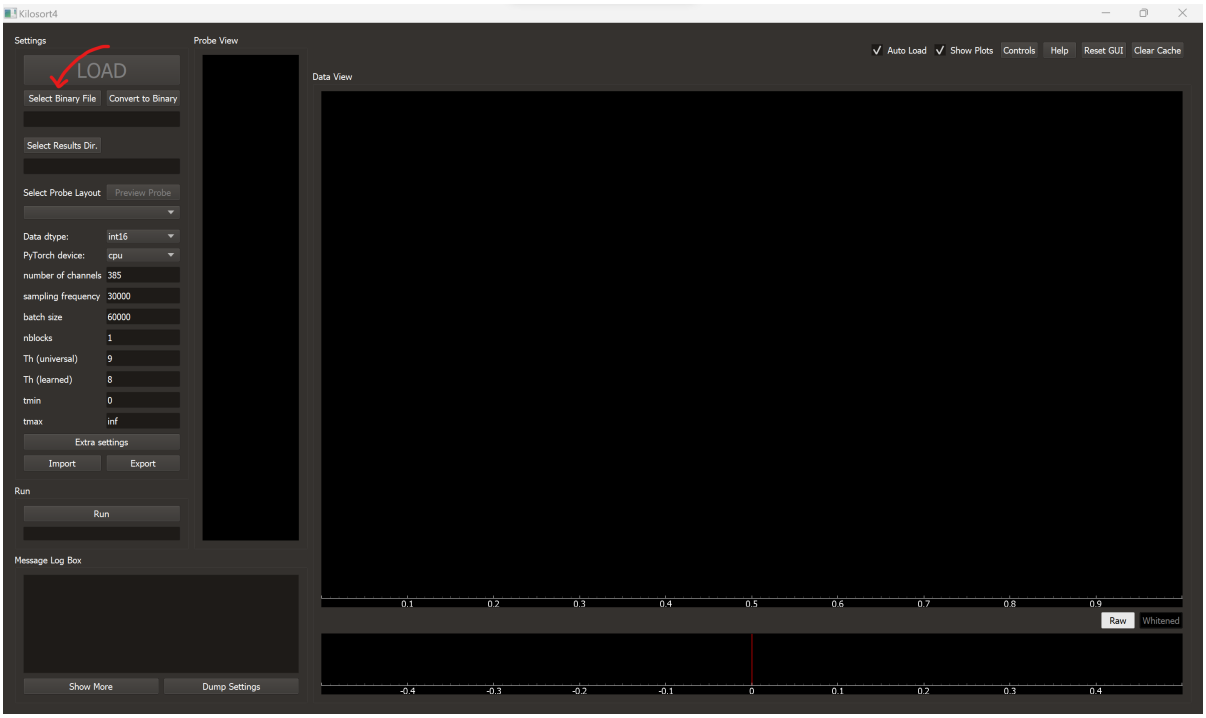
TWO

HOW TO USE THE GUI

This page explains how to use the basic functions of Kilosort4 GUI, in order of the steps you would take to start sorting.

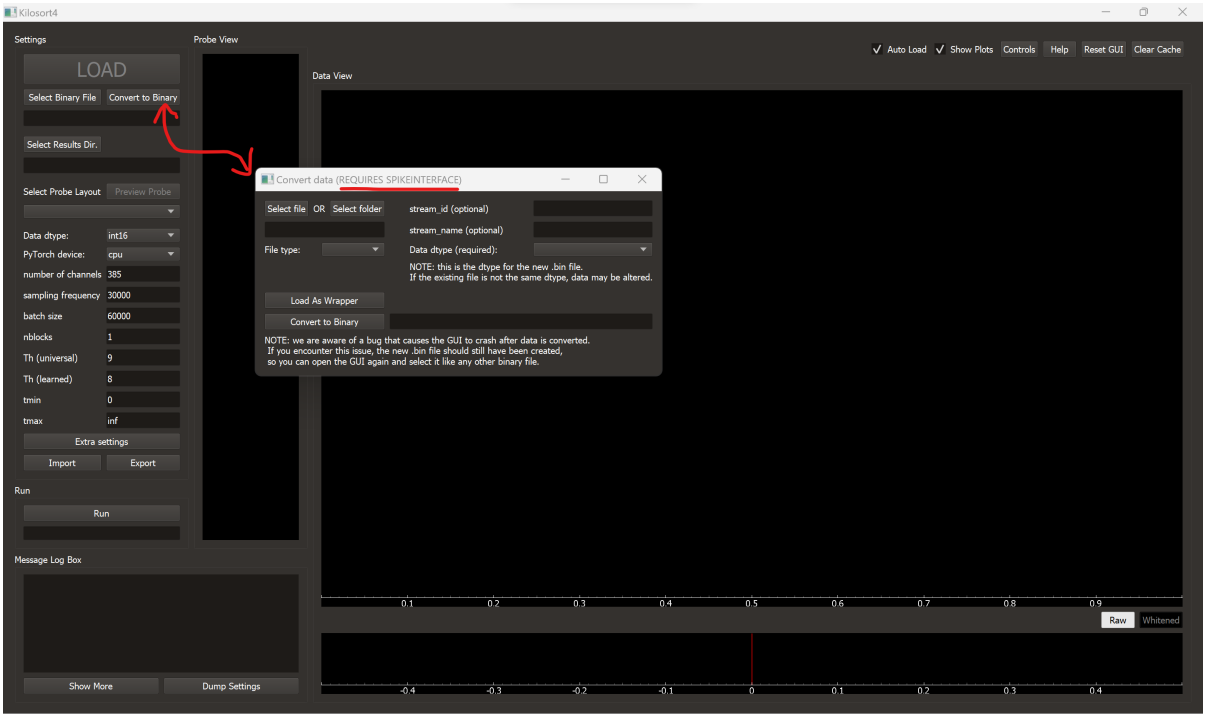
2.1 Select data

Start by selecting a binary data file (.bin, .bat, .dat, or .raw) to load, by clicking on the “Select Binary File” button near the top-left of the GUI. This will open a file dialog that will populate the neighboring text field after a file is selected. You can also paste a filepath directly into the text field.



2.2 Convert from other formats (optional)

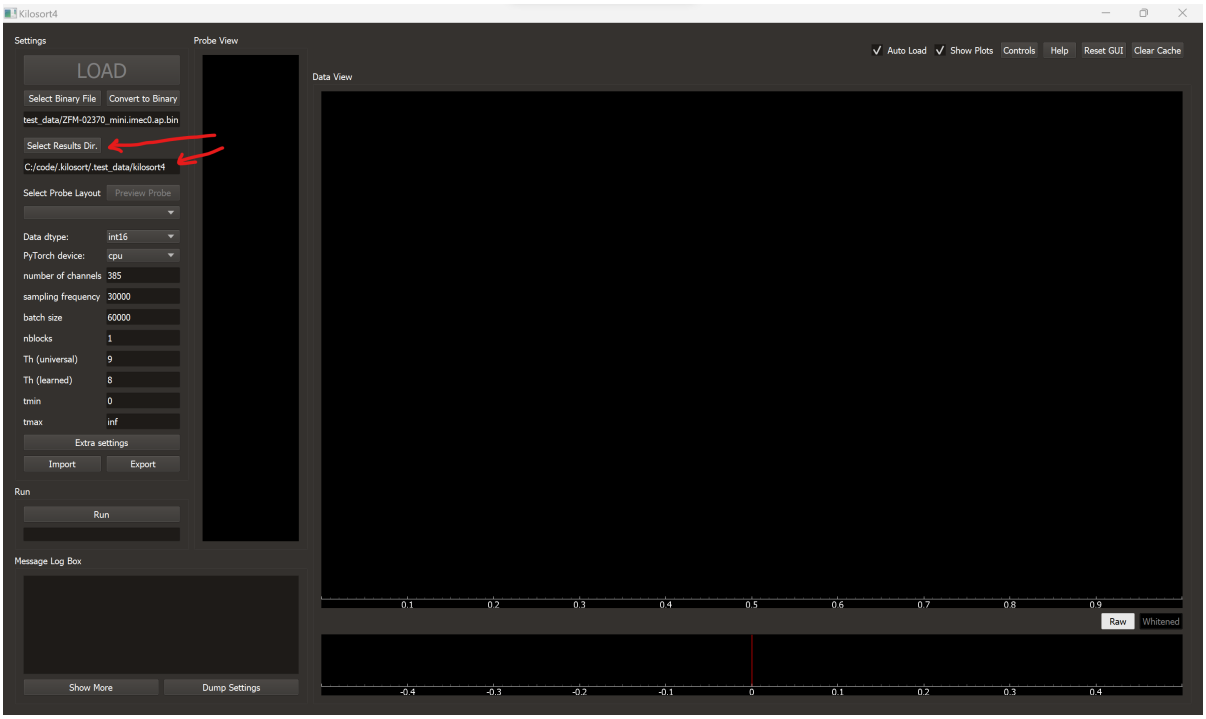
If your data is not in one of the supported formats listed in the previous step, you can click the "Convert to Binary" button to open the data conversion tool. Using this tool requires the SpikeInterface package. To convert your data, you will need to select either a file OR a folder (not both) using the, choose the filetype from the list of supported options, and select the dtype. Then, click "Convert to Binary" (recommended) to copy the data to a new .bin file. Alternatively, you can click "Load As Wrapper" to use the data without copying to a new file, but you will not be able to view results in Phy.



Note: this tool is only intended to cover the most common formats and options for getting your data into Kilosort. If you don't see your data format or if you run into errors related to extra options that aren't in our GUI, we recommend using SpikeInterface directly to convert your data. Their documentation here shows an example of how to save recording traces to .raw format using their own tools.

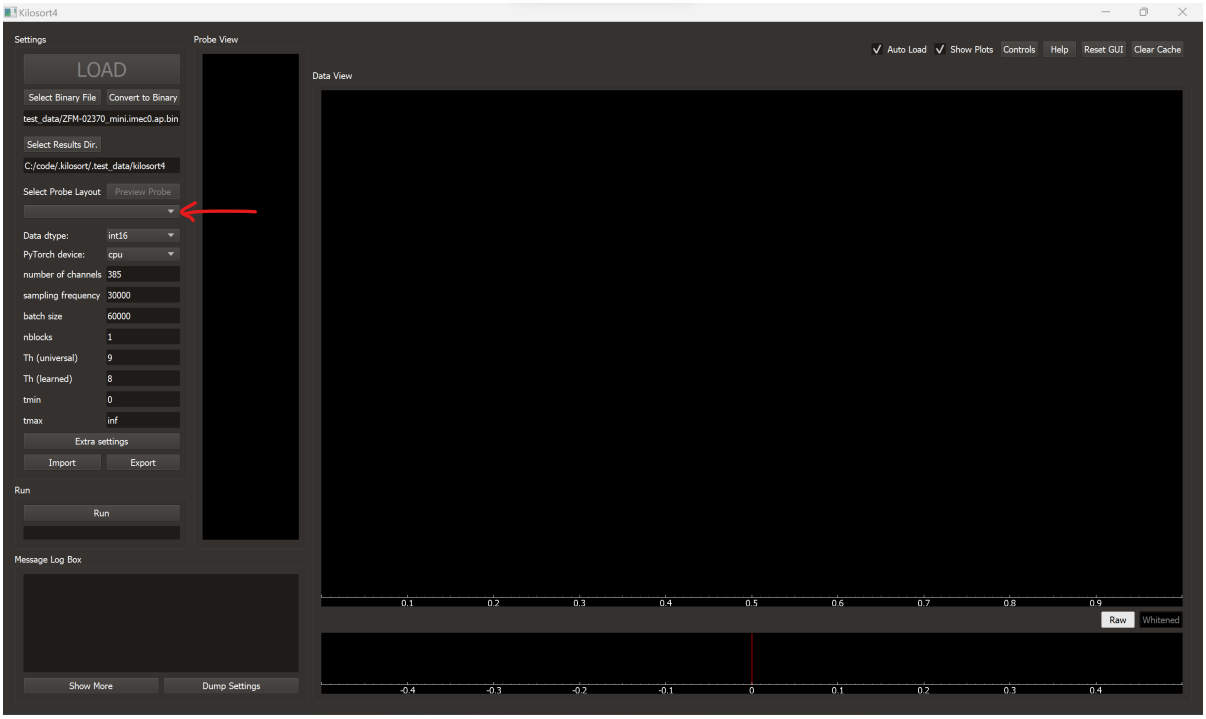
2.3 Choose where to save results

After a binary file is selected, the GUI will automatically populate the text field under "Select Results Dir." with the same path as your binary file, but ending in a "/kilosort4" directory instead of the binary file. If you wanted to change this, you can click the "Select Results Dir." button to open another file dialog, or edit the text field.

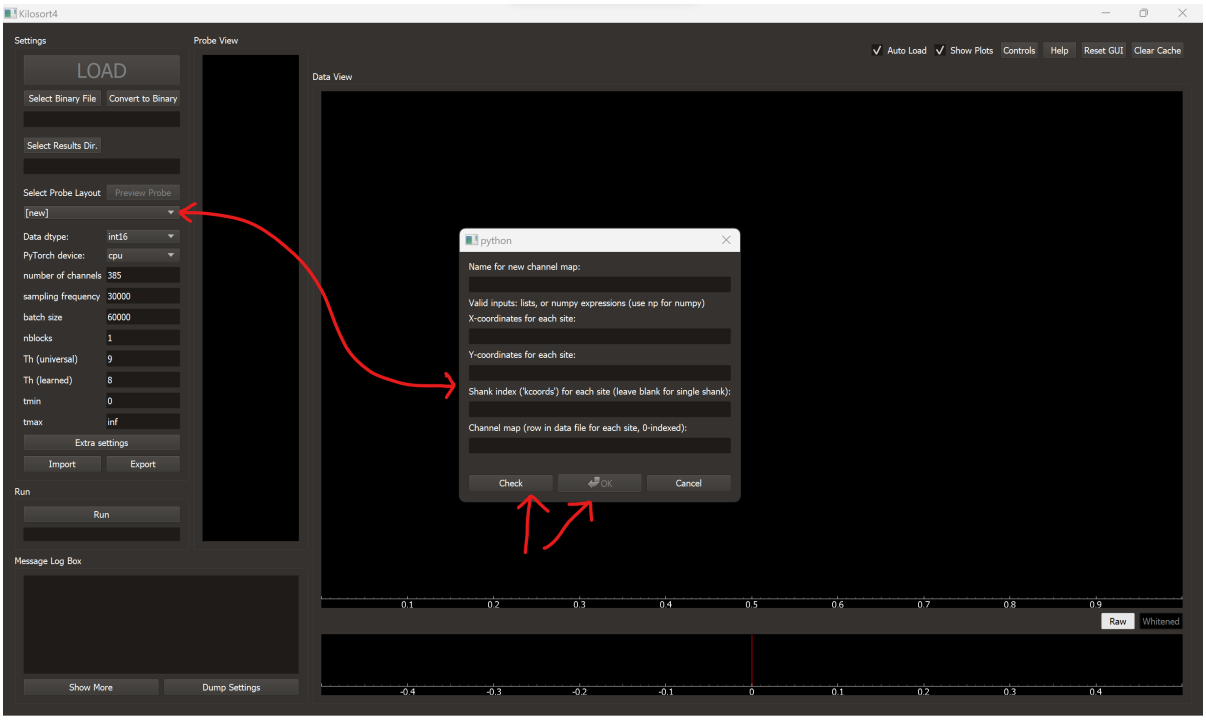


2.4 Select a probe

To select a probe, click the drop-down menu just below "Select Probe Layout." The list will include some default Neuropixels probe layouts. If you've already created your own probe file (.mat, .prb, or .json), you can select "other…" to open a file dialog and navigate to it.



If you need to create a new probe layout, select "[new]" to open the probe creation tool. Values for 'x-coordinates' and 'y-coordinates' need to be in microns, and can be specified with numpy expressions. For example, a 1-shank linear probe with 4 channels could have *np.ones(4)* in the 'x-coordinates' field instead of *1, 1, 1, 1*. Each field (except name) must have the same number of elements, corresponding to the number of ephys channels in the data. WHen you are finished setting the values, click "Check" to verify that your inputs are valid. If they are not, an error message will be displayed. Otherwise, the "OK" button will become clickable, which will save the probe to the Kilosort4 probes directory.

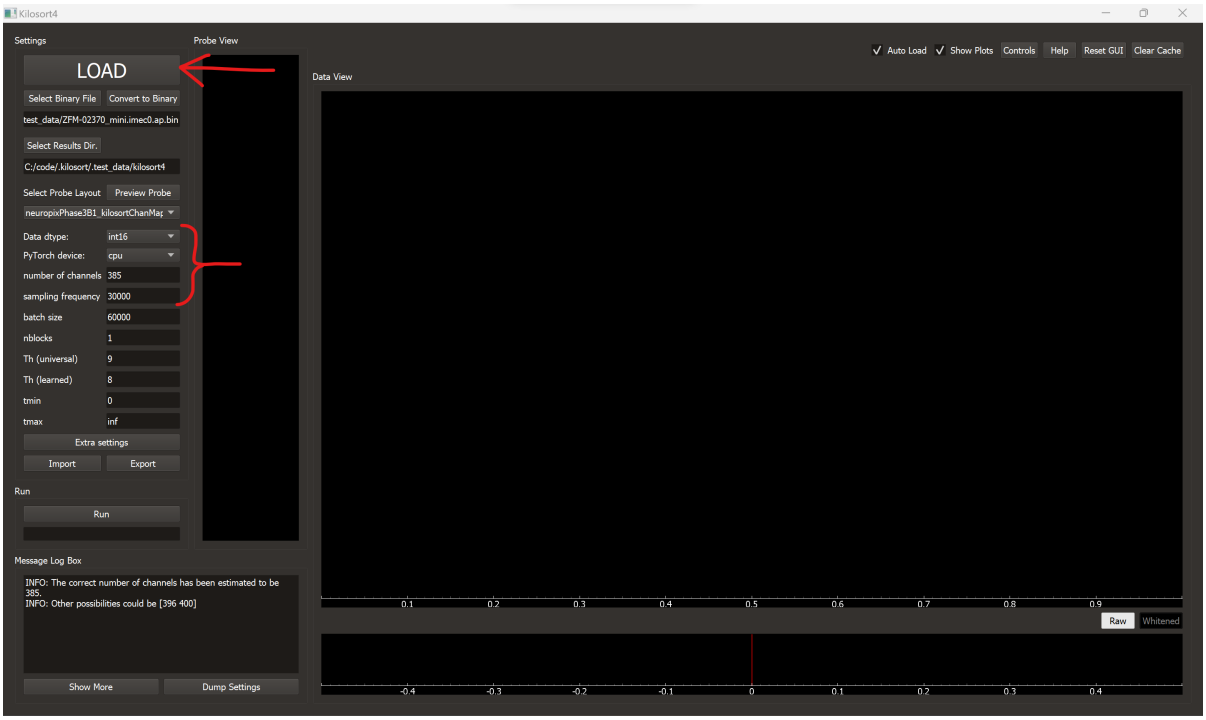


After a probe is selected, you can click "Preview Probe" to see a visualization and verify that the probe geometry looks correct. Checking "True Aspect Ratio" will show a physically proportional representation. Moving the slider will adjust the displayed scale of the contacts.

2.5 Load the data

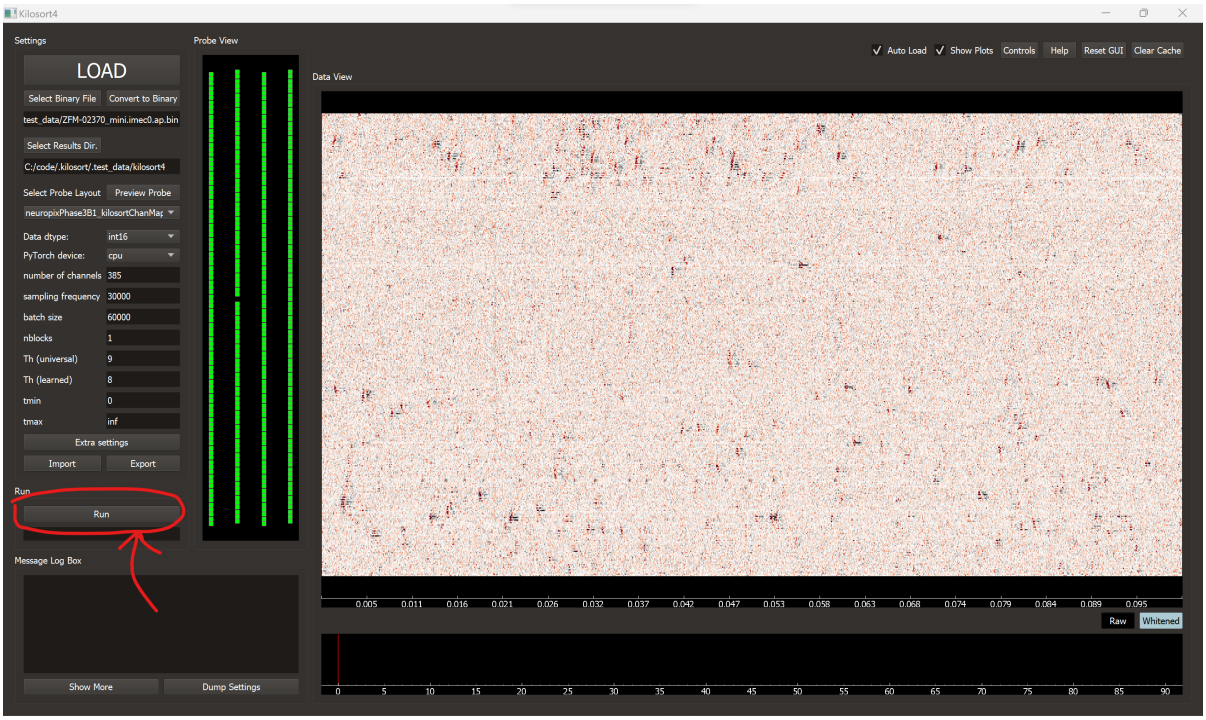
After you select a probe, the GUI will attempt to automatically determine the correct value for 'number of channels.' Make sure this correctly reflects the number of channels in your datafile, including non-ephys channels. For example, Neuropixels 1 probes output data with 385 channels. Only 384 of those are the ephys data used for sorting, but 'number of channels' should still be set to 385. You may also need to change the dtype of the data (int16 by default) or the sampling rate (30000hz by default). Additionally, you can choose which computing device. By default, the GUI will select the first CUDA GPU detected by PyTorch, or CPU if no GPU is detected.

When you are satisfied with these settings, click "LOAD" at the top left of the GUI to load the data.

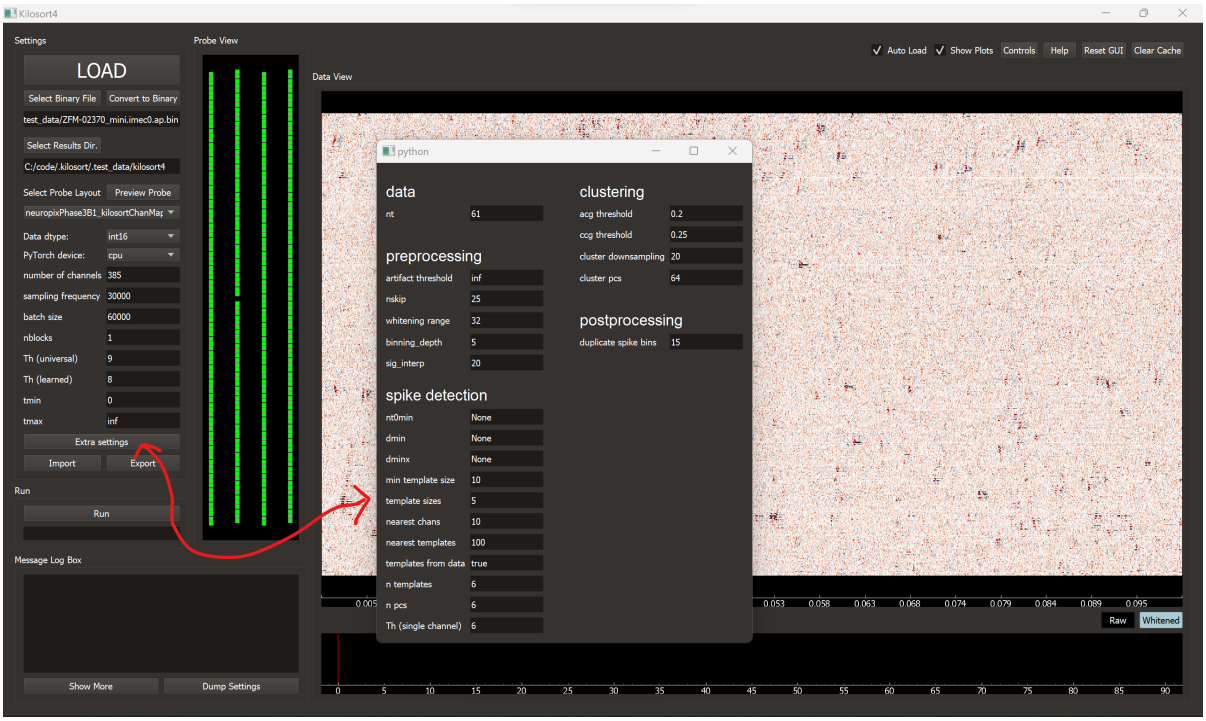


2.6 Run spike sorting

After loading the data, a heatmap will appear on the right half of the GUI showing a preprocessed version of the data. You can click "raw" at the bottom right to view the data without preprocessing applied. Make sure the data looks like what you expect, including the correct number of seconds along the bottom of the GUI. A common error to look for is diagonal lines in the heatmap, which usually indicates that 'number of channels' does not match the data. When everything looks good, click "Run" near the bottom left to begin spike sorting. When sorting is finished, the results will be saved to the directory indicated under "Select Results Dir."



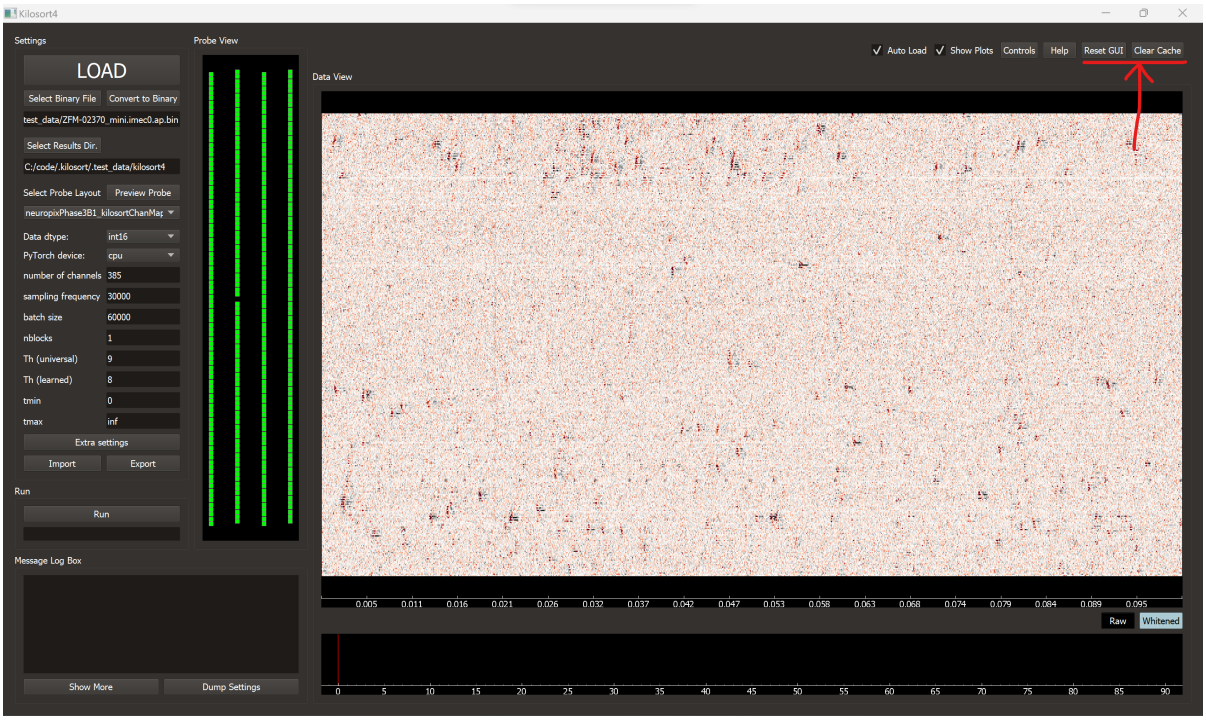
If you run into errors or the results look strange, you may need to tweak some of the other settings. A handful are shown below 'number of channels' and 'sampling frequency,' or you can click "Extra settings" to open a new window with more options. Mousing over the name of a setting for about half a second will show a description of what the setting does, along with information about which values are allowed. For more detailed suggestions, see *When to adjust default settings*



If you're still not sure how to proceed, check issues page on our github for more help.

2.7 Resetting the GUI

If the GUI gets stuck on a loading animation or some other odd behavior, try clicking on "Reset GUI" near the top right, which should reset it to the state shown in the first step on this page. If you want to make sure all previous settings are deleted, you can also click "Clear Cache" and then close and re-open the GUI.



CHAPTER

THREE

WHEN TO ADJUST DEFAULT SETTINGS

This page will give suggestions for when to change specific settings and why. Not every configurable setting will be covered here, but all settings include a short description in kilosort/parameters.py. The same description can also be seen in the GUI by mousing over the name of a setting for about half a second.

3.1 `n_chan_bin` (number of channels)

This should reflect the total number of channels in the binary file, *including non-ephys channels not used for sorting*. If you load your data in the GUI and see repeating diagonal patterns in the data, you probably need to change this value.

3.2 `batch_size`

This sets the number of samples included in each batch of data to be sorted, with a default of 60000 corresponding to 2 seconds for a sampling rate of 30000. For probes with fewer channels (say, 64 or less), increasing `batch_size` to include more data may improve results because it allows for better drift estimation (more spikes to estimate drift from).

3.3 `nblocks`

This is the number of sections the probe is divided into when performing drift correction. The default of `nblocks = 1` indicates rigid registration (the same amount of drift is applied to the entire probe). If you see different amounts of drift in your data depending on depth along the probe, increasing `nblocks` will help get a better drift estimate. `nblocks=5` can be a good choice for single-shank Neuropixels probes. For probes with fewer channels (around 64 or less) or with sparser spacing (around 50um or more between contacts), drift estimates are not likely to be accurate, so drift correction should be skipped by setting `nblocks = 0`.

3.4 `Th_universal` and `Th_learned`

These control the threshold for spike detection when applying the universal and learned templates, respectively (loosely similar to Th(1) and Th(2) in previous versions). If few spikes are detected, or if you see neurons disappearing and reappearing over time when viewing results in Phy, it may help to decrease `Th_learned`. To detect more units overall, it may help to reduce `Th_universal`. Try reducing each threshold by 1 or 2 at a time.

3.5 tmin and tmax

This sets the start and end of data used for sorting (in seconds). By default, all data is included. If your data contains recording artifacts near the beginning or end of the session, you can adjust these to omit that data. "inf" and "np.inf" can be used for tmax to indicate end of session in the GUI and API respectively.

3.6 nt

This is the number of time samples used to represent spike waveforms, as well as the amount of symmetric padding for filtering. The default represents 2ms + 1 bin for a sampling rate of 30kHz. For a different sampling rate, you may want to adjust accordingly. For example, `nt = 81` would be the 2ms equivalent for 40kHz.

3.7 dmin and dminx

These adjust the vertical and lateral spacing, respectively, of the universal templates used during spike detection, as well as the vertical and lateral sizes of channel neighborhoods used for clustering. By default, Kilosort will attempt to determine a good value for `dmin` based on the median distance between contacts, which tends to work well for Neuropixels-like probes. However, if contacts are irregularly spaced, you may need to specify this manually. The default for `dminx` is 32um, which is also well suited to Neuropixels probes. For other probes, try setting `dminx` to the median lateral distance between contacts as a starting point.

3.8 min_template_size

This sets the standard deviation of the smallest Gaussian spatial envelope used to generate universal templates, with a default of 10 microns. You may need to increase this for probes with wider spaces between contacts.

3.9 nearest_chans and nearest_templates

This is the number of nearest channels and template locations, respectively, used when assigning templates to spikes during spike detection. `nearest_chans` cannot be larger than the total number of channels on the probe, so it will need to be reduced for probes with less than 10 channels. `nearest_templates` does not have this restriction. However, for probes with around 64 channels or less and sparsely spaced contacts, decreasing `nearest_templates` to be less than or equal to the number of channels helps avoid numerical instability.

3.10 x_centers

The number of x-positions to use when determining centers for template groupings. Specifically, this is the number of centroids to look for when using k-means to cluster the x-positions for the probe. In most cases you should not need to specify this. However, **for probes with contacts arranged in a 2D grid**, we recommend setting `x_centers` such that centers are placed every 200-300um so that there are not too many templates in each group. For example, for an array that is 2000um in width, try `x_centers = 10`. If contacts are very densely spaced, you may need to use a higher value for better performance.

3.11 duplicate_spike_bins

After sorting has finished, spikes that occur within this number of bins of each other, from the same unit, are assumed to be artifacts and removed. The default of 7 bins corresponds to approximately 0.25ms for a sampling rate of 30000hz. If your sampling rate is different, you may need to increase or decrease this accordingly. If you see otherwise good neurons with large peaks around 0ms when viewing correlograms in Phy, increasing this value can help remove those artifacts.

Warning!!! Do not increase this value beyond 0.5ms as it will interfere with the ACG and CCG refractory period estimations (which normally ignores the central 1ms of the correlogram).

CHAPTER

FOUR

USING PROBES WITH MULTIPLE SHANKS

Update: the latest version of the code addresses this issue. After getting feedback from users, this page will be removed if there are no further issues. For 2D arrays, you may want to set the new *x_centers* parameter (under "Extra Settings" in the GUI) for best results.

Currently, Kilosort4 does not process shanks separately. Until this is added, we recommend changing your probe layout to artificially stack the shanks in a single column with a bit of vertical space (~100um) between each shank.

If that option isn't feasible for some reason, you can also try adjusting the *min_template_size* and/or *dminx* parameters in the GUI, or in the settings argument for *run_kilosort* if you're using the API. Setting *dminx* to around half the total width of the probe seems to be a good starting point, and you can adjust from there.

See issue 606 and issue 617 for additional context.

CHAPTER

FIVE

HARDWARE RECOMMENDATIONS

We start with a list of recommendations for a standard Neuropixels recording (384 channels, <3 hours), and then we continue with a list of modifications depending on how your recordings differ from this standard one. Note it is possible to run Kilosort4 on the CPU, but this should only be done for testing purposes as it is much slower than even a cheap GPU (i.e. RTX 3060 12GB).

5.1 Recommendations for Neuropixels/large recordings

1. GPU: only Nvidia GPUs are supported via pytorch. We recommend at least 8GB of RAM, which can also be had in relatively cheap GPU. Speed increases with a newer generation and larger GPU. The Nvidia numbering scheme is N0X0, where N is the generation and X is the tier level which increases with GPU size. For example, GTX 1080 Ti is generation "1" and tier "8", and the Ti suffix indicates a slightly bigger/better card. Current cards are generation "4" and RTX 4070 is probably a sweet spot. Note that the "professional" GPUs are not much faster for Kilosort processing despite being many times more expensive.
2. SSD: check for read/write speed. Any good SATA SSD is around 500MB/s, and PCI-based SSD are usually a lot better. You shouldn't need anything faster than 500MB/s, but you might want to have generous capacity for the purpose of batch processing of many datasets. A typical workflow is to copy datasets you want processed to the SSD, then run Kilosort on them, then run Phy on the local copy of the data with the Kilosort results. Once you are happy with the sorting, you would typically free up the local copy of the data to make space for the next round of spike sorting.
3. rest of the system: 32GB of RAM are recommended and an 8-core CPU. There is little performance to be gained by increasing these.

5.2 Additional recommendations

1. Same number of channels, but longer recordings, like 6h or more. Right now, this situation typically requires more RAM, like 32 or 64 GB. Kilosort splits the data into small batches (default = 2s) and sorts each separately. However, a few steps (spike extraction, clustering) require keeping some information from each batch in RAM, before it is used or saved at the end of the run through the entire data. This situation is also likely to arise with fewer channels, and correspondingly longer recordings.
2. More channels. This has not been tested but is likely to require more GPU RAM. Reduce the batch size if running into memory limitations.
3. Very few channels. Kilosort4 might fail for 1 channel.

CHAPTER

SIX

DRIFT CORRECTION

Here we get into the details of drift: why it happens, what it looks like, how you can characterize it for your own recordings, and how you can tell if Kilosort is fixing it. The examples in this section are from acute Neuropixels 1.0 recordings in head-fixed mice.

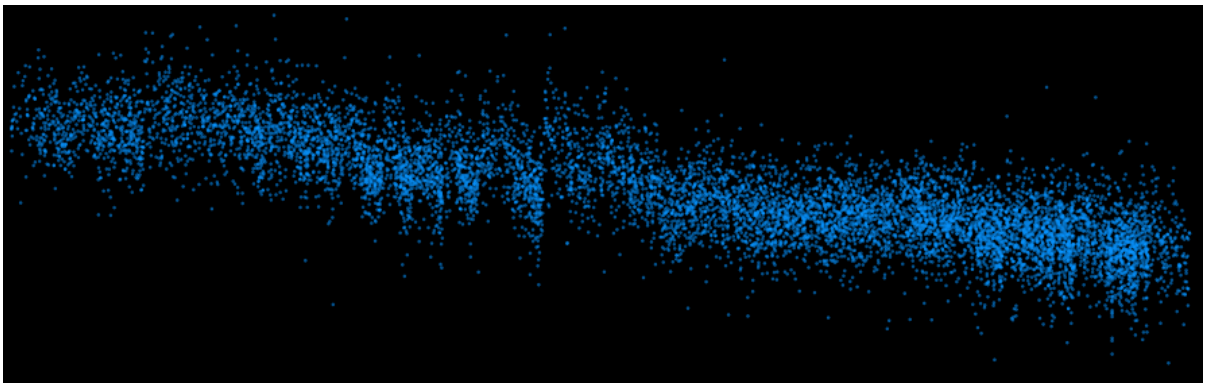
6.1 Why drift happens

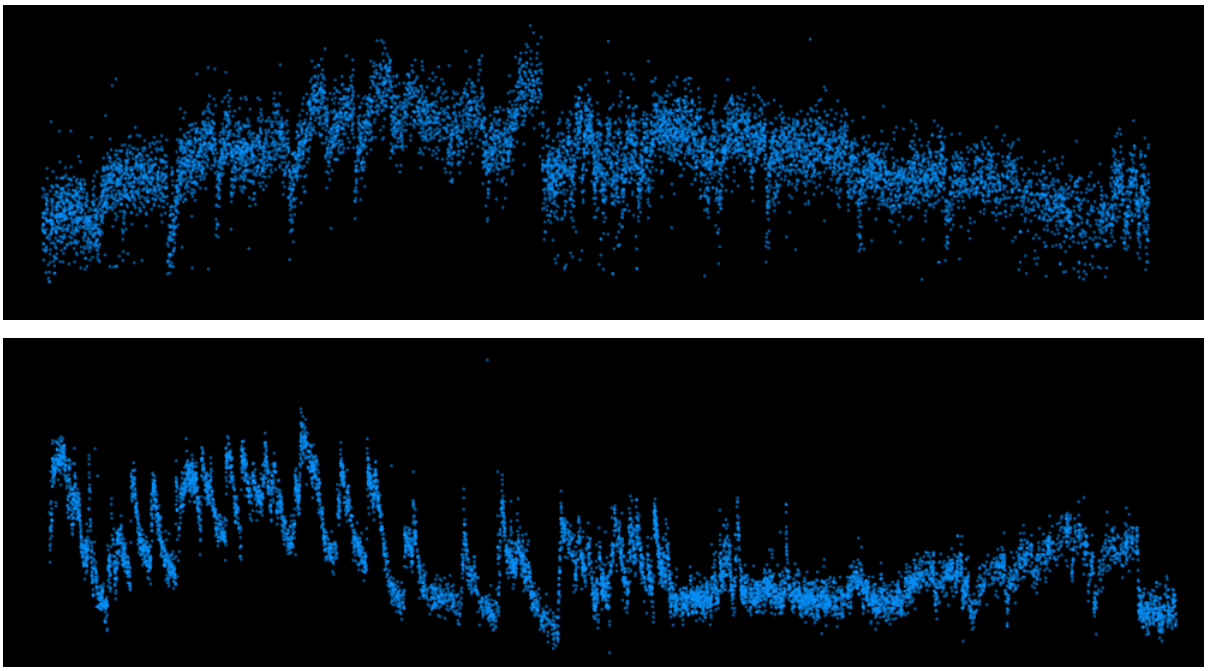
Some amount of drift is unavoidable. The brain floats inside the skull, and moves when the animal moves, which can be very fast. At slower timescales, physiological changes happen that change the tissue or move it relative to the probe. Some of these physiological changes may be induced by the presence of the probe itself, for example due to an inflammatory response. Thus, there are two main types of drift, which we treat differently: slow (10s of minutes) and fast (10s of seconds).

Slow drift is not a huge problem if the recording is short (<10 minutes). Fast drift is not a huge problem if the animal is not behaving. However, neither is true in a typical neuroscience experiment: the animal is performing a task which involves some motor actions, and it takes at least 30 minutes to characterize the neural activity. In a typical recording of 1-2 hours, the amount of slow and fast drift we'd expect is comparable, and can be anywhere from 5 to 20um and more if your preparation is unstable.

6.2 What drift looks like

To recognize drift, look for changes in spike amplitude over time, or changes in feature amplitudes, where the feature can be the projection on the principal components of a channel. More dramatically, drift may be recognized by a cluster that "drops out" because all its spikes are lost when the drift is too large. Here is are three examples tracked by Kilosort2, all from the same recording. These clusters appear to have different amounts of drift, which is primarily because they have different sizes: the unit that drifts the most can only be seen on one channel. However, the moments at which drift happens for these units are similar.



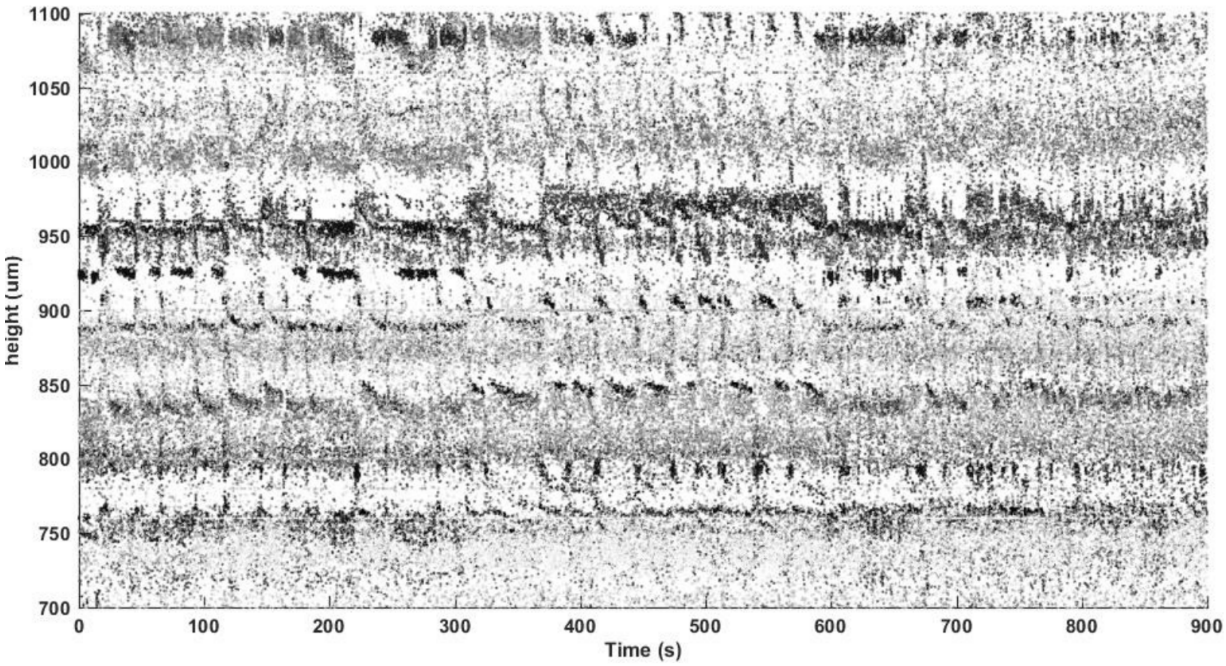


Slow drift is generally easier to recognize and fix. A well-known version of slow drift happens after probe insertion in an acute experiment, which is why it is typical to wait 20-30 minutes for the tissue to relax before recording. However, even after that initial phase, there is a smaller amplitude, slow timescale drift that continues for at least a few hours, and has been reported to us even in chronic implants. Cumulative over time, slow drift can have a significant impact on spike sorting.

Fast drift is harder to diagnose but potentially more dangerous, because it may introduce behavior-dependent biases into the data. The bias may be produced if every time an animal performs a certain motor action, the probe moves a little, in which case the spikes from a small neuron may be completely lost. It will then appear as if this neuron was inhibited by movement. Conversely, a neuron may only come into the range of the electrodes during the movement, in which case it will appear as if that neuron is activated by movement. This behavior also makes fast drift harder to diagnose, because most neurons in the brain genuinely have movement-related spiking activity.

6.3 How to diagnose drift

The main way to diagnose drift is, in fact, to run the first step of Kilosort4, which produces a picture of spike times by depth, with the size of the points representing their amplitudes.

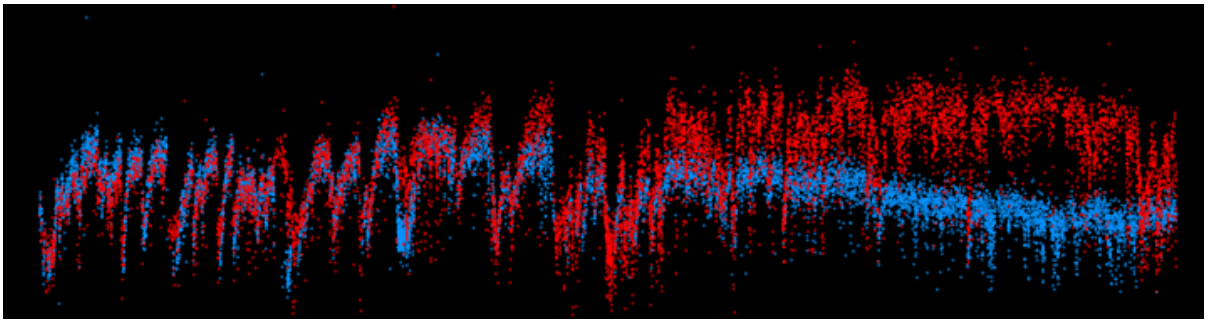


The spike sorting routine for each batch is a very fast, lite clustering algorithm, where the spikes are first detected as threshold crossings in PCA space, their PCA features are then extracted and the spikes are clustered with scaled k-means. To get enough information from a single batch, there should be many spikes inside that time interval. The default time interval is 2s, which may be too short for <32 channels, in which case we recommend increasing the batch size.

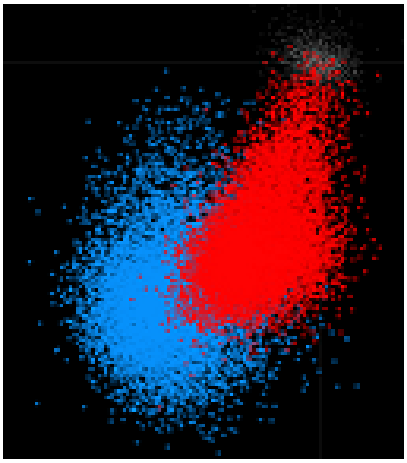
6.4 Drift tracking improves cluster separation

As a result of drift tracking, units are no longer split into multiple small pieces like in Kilosort1 and other algorithms. Merging back together these small pieces was the most time consuming part of the manual curation in Kilosort1, which is no longer required. On top of the automation benefit, an additional benefit of drift correction is the overall improvement in cluster separation. Why would that be the case? Imagine two neurons that have very similar waveforms but are slightly shifted along the probe. As the probe drifts up, the bottom neuron starts taking the place of the top neuron, and since they have similar waveforms, it is impossible to tell apart spikes of the bottom neuron at this time from spikes of the top neuron before the drift started. Unless, that is, the spike sorting algorithm is aware of the time in the recording when the different spikes happened. Algorithms like Kilosort2-4, which track units as they drift, maintain a constant separation between two such units, because the separation is only for spikes at a particular drift position. Here is an example from a real recording.

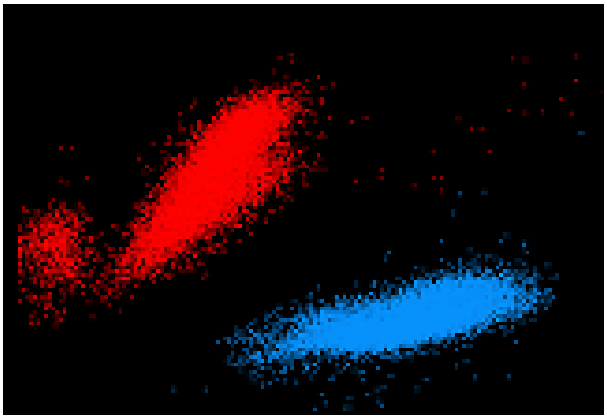
First, the amplitude timecourses of two very similar units:



Now the PCA feature projections across all times, for the two channels where these units are biggest:



Finally, here are the projections on the time-varying templates of Kilosort2, showing that the units maintain a constant high separation throughout the recording:



CHAPTER

SEVEN

EXAMPLE SPIKE-SORTING ANALYSIS WITH SAMPLE DATA

This tutorial is also available as a collab notebook if you would like to try Kilosort4 without installing the code locally.

7.1 1. Download example data

This is an example electrophysiological recording from the International Brain Laboratory, recorded using a Neuropixels 1.0 probe (all data here). The full recording is over 4000 seconds long, and the cropped recording is 90 seconds long.

Downloading the cropped recording will take around 3 minutes in google colab. If it fails, please try again (press play again), it sometimes hangs.

You can alternatively use any .bin file. See the "Loading other data formats" tutorial for loading other file extensions.

We also used data from Nick Steinmetz, available to download here. For that data use `probe_name='neuropixPhase3A_kilosortChanMap.mat'`.

[ ]:
```
import urllib.request
from pathlib import Path
from tqdm import tqdm

# NOTE: Be sure to update this filepath if you want the data downloaded to
#       a specific location.
SAVE_PATH = Path('ZFM-02370_mini.imec0.ap.bin')

class DownloadProgressBar(tqdm):
    """ from https://stackoverflow.com/a/53877507 """
    def update_to(self, b=1, bsize=1, tsize=None):
        if tsize is not None:
            self.total = tsize
        self.update(b * bsize - self.n)
def download_url(url, output_path):
    with DownloadProgressBar(unit='B', unit_scale=True,
                             miniters=1, desc=url.split('/')[-1]) as t:
        urllib.request.urlretrieve(url, filename=output_path, reporthook=t.update_to)

## FULL DATASET (download locally then decompress)
# compressed using mtscomp (https://github.com/int-brain-lab/mtscomp)
# URL = 'https://ibl.flatironinstitute.org/public/mainenlab/Subjects/ZFM-02370/2021-04-
↪28/001/raw_ephys_data/probe00/_spikeglx_ephysData_g0_t0.imec0.ap.e510da60-53e0-4e00-
↪b369-3ea16c45623a.cbin'
```

(continues on next page)

(continued from previous page)

```
# NOTE: There are some extra steps to get this uncompressed,
#       you cannot load the .cbin file directly into Kilosort.
#       Additionally, this dataset is not maintained by the Kilosort team.
#       We recommend using the smaller sample dataset linked below for testing.

## CROPPED DATASET
URL = 'http://www.kilosort.org/downloads/ZFM-02370_mini.imec0.ap.bin'
download_url(URL, SAVE_PATH)
```

[ ]:
```
# Download channel maps for default probes
from kilosort.utils import download_probes
download_probes()
```

7.1.1 Run kilosort

[ ]:
```
from kilosort import run_kilosort

# NOTE: 'n_chan_bin' is a required setting, and should reflect the total number
#       of channels in the binary file. For information on other available
#       settings, see `kilosort.run_kilosort.default_settings`.
settings = {'data_dir': SAVE_PATH.parent, 'n_chan_bin': 385}

ops, st, clu, tF, Wall, similar_templates, is_ref, est_contam_rate = \
    run_kilosort(settings=settings, probe_name='neuropixPhase3B1_kilosortChanMap.mat')
```

7.1.2 Plot the results

Note: at this point, you can also load the results in phy.

Load outputs

[ ]:
```
import numpy as np
import pandas as pd
from pathlib import Path

# outputs saved to results_dir
results_dir = Path(settings['data_dir']).joinpath('kilosort4')
ops = np.load(results_dir / 'ops.npy', allow_pickle=True).item()
camps = pd.read_csv(results_dir / 'cluster_Amplitude.tsv', sep='\t')['Amplitude'].values
contam_pct = pd.read_csv(results_dir / 'cluster_ContamPct.tsv', sep='\t')['ContamPct'].
↪values
chan_map =  np.load(results_dir / 'channel_map.npy')
templates =  np.load(results_dir / 'templates.npy')
chan_best = (templates**2).sum(axis=1).argmax(axis=-1)
chan_best = chan_map[chan_best]
amplitudes = np.load(results_dir / 'amplitudes.npy')
st = np.load(results_dir / 'spike_times.npy')
clu = np.load(results_dir / 'spike_clusters.npy')
firing_rates = np.unique(clu, return_counts=True)[1] * 30000 / st.max()
dshift = ops['dshift']
```

Plot outputs

[ ]:
```
%matplotlib inline
import matplotlib.pyplot as plt
from matplotlib import gridspec, rcParams
rcParams['axes.spines.top'] = False
rcParams['axes.spines.right'] = False
gray = .5 * np.ones(3)

fig = plt.figure(figsize=(10,10), dpi=100)
grid = gridspec.GridSpec(3, 3, figure=fig, hspace=0.5, wspace=0.5)

ax = fig.add_subplot(grid[0,0])
ax.plot(np.arange(0, ops['Nbatches'])*2, dshift);
ax.set_xlabel('time (sec.)')
ax.set_ylabel('drift (um)')

ax = fig.add_subplot(grid[0,1:])
t0 = 0
t1 = np.nonzero(st > ops['fs']*5)[0][0]
ax.scatter(st[t0:t1]/30000., chan_best[clu[t0:t1]], s=0.5, color='k', alpha=0.25)
ax.set_xlim([0, 5])
ax.set_ylim([chan_map.max(), 0])
ax.set_xlabel('time (sec.)')
ax.set_ylabel('channel')
ax.set_title('spikes from units')

ax = fig.add_subplot(grid[1,0])
nb=ax.hist(firing_rates, 20, color=gray)
ax.set_xlabel('firing rate (Hz)')
ax.set_ylabel('# of units')

ax = fig.add_subplot(grid[1,1])
nb=ax.hist(camps, 20, color=gray)
ax.set_xlabel('amplitude')
ax.set_ylabel('# of units')

ax = fig.add_subplot(grid[1,2])
nb=ax.hist(np.minimum(100, contam_pct), np.arange(0,105,5), color=gray)
ax.plot([10, 10], [0, nb[0].max()], 'k--')
ax.set_xlabel('% contamination')
ax.set_ylabel('# of units')
ax.set_title('< 10% = good units')

for k in range(2):
    ax = fig.add_subplot(grid[2,k])
    is_ref = contam_pct<10.
    ax.scatter(firing_rates[~is_ref], camps[~is_ref], s=3, color='r', label='mua',␣
↪alpha=0.25)
    ax.scatter(firing_rates[is_ref], camps[is_ref], s=3, color='b', label='good',␣
↪alpha=0.25)
    ax.set_ylabel('amplitude (a.u.)')
    ax.set_xlabel('firing rate (Hz)')
```

(continues on next page)

(continued from previous page)

```
    ax.legend()
    if k==1:
        ax.set_xscale('log')
        ax.set_yscale('log')
        ax.set_title('loglog')
```

[ ]:
```
probe = ops['probe']
# x and y position of probe sites
xc, yc = probe['xc'], probe['yc']
nc = 16 # number of channels to show
good_units = np.nonzero(contam_pct <= 0.1)[0]
mua_units = np.nonzero(contam_pct > 0.1)[0]


gstr = ['good', 'mua']
for j in range(2):
    print(f'~~~~~~~~~~~~~~ {gstr[j]} units ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~')
    print('title = number of spikes from each unit')
    units = good_units if j==0 else mua_units
    fig = plt.figure(figsize=(12,3), dpi=150)
    grid = gridspec.GridSpec(2,20, figure=fig, hspace=0.25, wspace=0.5)

    for k in range(40):
        wi = units[np.random.randint(len(units))]
        wv = templates[wi].copy()
        cb = chan_best[wi]
        nsp = (clu==wi).sum()

        ax = fig.add_subplot(grid[k//20, k%20])
        n_chan = wv.shape[-1]
        ic0 = max(0, cb-nc//2)
        ic1 = min(n_chan, cb+nc//2)
        wv = wv[:, ic0:ic1]
        x0, y0 = xc[ic0:ic1], yc[ic0:ic1]

        amp = 4
        for ii, (xi,yi) in enumerate(zip(x0,y0)):
            t = np.arange(-wv.shape[0]//2,wv.shape[0]//2,1,'float32')
            t /= wv.shape[0] / 20
            ax.plot(xi + t, yi + wv[:,ii]*amp, lw=0.5, color='k')

        ax.set_title(f'{nsp}', fontsize='small')
        ax.axis('off')
    plt.show()
```

CHAPTER

EIGHT

CREATING A KILOSORT4 PROBE DICTIONARY

Kilosort4 uses a dictionary to track probe interformation. The dictionary needs the following keys, all of which correspond to NumPy ndarrays.

```
'chanMap': the channel indices that are included in the data.
'xc':      the x-coordinates (in micrometers) of the probe contact centers.
'yc':      the y-coordinates (in micrometers) of the probe contact centers.
'kcoords': shank or channel group of each contact (not used yet, set all to 0).
'n_chan':  the number of channels.
```

To demonstrate, we'll create a probe dictionary corresponding to a real example, a 128-channel probe from Diagnostic Biochips.

We'll assume all channels are used, so 'chanMap' will just be the range of integers from 0 to 127. 'kcoords' can be set to all zeroes as mentioned above.

[8]:
```
import numpy as np

chanMap = np.arange(128)
kcoords = np.zeros(128)
n_chan = 128
```

Since the contacts are 11 micrometers wide (x) and 15 micrometers high (y), our first contact center has coordinates (5.5, 7.5). There is a single column of contacts, so all x-coordinates are the same. Finally, the diagram indicates that contacts are spaced 20 micrometers apart.

[9]:
```
xc = np.ones(128)*5.5
yc = np.array([7.5 + 20*i for i in range(128)])
```

So, our probe dictionary looks like:

[10]:
```
probe = {
    'chanMap': chanMap,
    'xc': xc,
    'yc': yc,
    'kcoords': kcoords,
    'n_chan': n_chan
}

print(probe)
```

```
{'chanMap': array([  0,   1,   2,   3,   4,   5,   6,   7,   8,   9,  10,  11,  12,
        13,  14,  15,  16,  17,  18,  19,  20,  21,  22,  23,  24,  25,
```

(continues on next page)

(continued from previous page)

```
         26,  27,  28,  29,  30,  31,  32,  33,  34,  35,  36,  37,  38,
         39,  40,  41,  42,  43,  44,  45,  46,  47,  48,  49,  50,  51,
         52,  53,  54,  55,  56,  57,  58,  59,  60,  61,  62,  63,  64,
         65,  66,  67,  68,  69,  70,  71,  72,  73,  74,  75,  76,  77,
         78,  79,  80,  81,  82,  83,  84,  85,  86,  87,  88,  89,  90,
         91,  92,  93,  94,  95,  96,  97,  98,  99, 100, 101, 102, 103,
        104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116,
        117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127]), 'xc': array([5.5, 5.5, 5.
↪5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5,
        5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5,
        5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5,
        5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5,
        5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5,
        5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5,
        5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5,
        5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5,
        5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5,
        5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5]), 'yc': array([   7.5,  ␣
↪27.5,   47.5,   67.5,   87.5,  107.5,  127.5,  147.5,
         167.5,  187.5,  207.5,  227.5,  247.5,  267.5,  287.5,  307.5,
         327.5,  347.5,  367.5,  387.5,  407.5,  427.5,  447.5,  467.5,
         487.5,  507.5,  527.5,  547.5,  567.5,  587.5,  607.5,  627.5,
         647.5,  667.5,  687.5,  707.5,  727.5,  747.5,  767.5,  787.5,
         807.5,  827.5,  847.5,  867.5,  887.5,  907.5,  927.5,  947.5,
         967.5,  987.5, 1007.5, 1027.5, 1047.5, 1067.5, 1087.5, 1107.5,
        1127.5, 1147.5, 1167.5, 1187.5, 1207.5, 1227.5, 1247.5, 1267.5,
        1287.5, 1307.5, 1327.5, 1347.5, 1367.5, 1387.5, 1407.5, 1427.5,
        1447.5, 1467.5, 1487.5, 1507.5, 1527.5, 1547.5, 1567.5, 1587.5,
        1607.5, 1627.5, 1647.5, 1667.5, 1687.5, 1707.5, 1727.5, 1747.5,
        1767.5, 1787.5, 1807.5, 1827.5, 1847.5, 1867.5, 1887.5, 1907.5,
        1927.5, 1947.5, 1967.5, 1987.5, 2007.5, 2027.5, 2047.5, 2067.5,
        2087.5, 2107.5, 2127.5, 2147.5, 2167.5, 2187.5, 2207.5, 2227.5,
        2247.5, 2267.5, 2287.5, 2307.5, 2327.5, 2347.5, 2367.5, 2387.5,
        2407.5, 2427.5, 2447.5, 2467.5, 2487.5, 2507.5, 2527.5, 2547.5]), 'kcoords':␣
↪array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0.]), 'n_chan': 128}
```

This probe dictionary can now be used as an argument for `run_kilosort`, as demonstrated above. We can also save it to a `.json` file for later use:

[ ]:
```
from kilosort.io import save_probe

save_probe(probe, '.../probe.json')
```

Now Kilsort4 can load the probe file and extract the contact coordinates and other relevant information. There are two ways to provide this information when running Kilosort4:

1. Specify the path to the probe file in the GUI.

Or

2. Load the probe using `kilosort.io.load_probe` and provide the resulting dictionary to `kilosort.run_kilosort` using the `probe` keyword argument (demonstrated below).

[ ]:
```
from kilosort import run_kilosort
from kilosort.io import load_probe

# Abreviated arguments, for demonstration only.
p = load_probe('.../test_prb.prb')
results = run_kilosort(..., probe=p)
```

CHAPTER

NINE

LOADING OTHER DATA FORMATS WITH SPIKEINTERFACE

Kilosort 4 natively supports data in binary format, `.bin`. The simplest way to save your data in this format is to load it into memory one chunk at a time and save it to a `.bin` file using `NumPy's memmap` function. However, if you aren't comfortable with that process, the `SpikeInterface` package can load most common electrophysiology formats in a standardized way that makes it easy to extract the data.

To follow the steps in this notebook, you will first need to install `SpikeInterface`:

```
pip install spikeinterface[full]
```

For each data format, `SpikeInterface` has a `read_<format>` utility that loads the data as a `RecordingExtractor` object, which we can use to extract the data and relevant meta information like sampling frequency. The following example shows the steps for the `NWB` data format. At the bottom of the notebook, there are notes on how to load several other common formats. For all cells after the first, all steps should be the same regardless of format.

1. Load NWB data

[ ]:
```
from pathlib import Path
import numpy as np
from spikeinterface.extractors import read_nwb_recording

# Specify the path where the data will be copied to, and where Kilosort 4
# results will be saved.
DATA_DIRECTORY = Path('/home/example_path')  # NOTE: You should change this
# Create path if it doesn't exist
DATA_DIRECTORY.mkdir(parents=True, exist_ok=True)

# Specify path to your existing data
filepath = Path(".../my_data.nwb")       # NOTE: You must change this
# Load existing data with spikeinterface
# NOTE: You may need to specify additional keyword arguments for
#       `read_nwb_recording`, such as `electrical_series_name`. Any required
#       arguments should be clearly spelled out by an error message.
recording = read_nwb_recording(filepath)
```

2. Create a new binary file and copy the data to it 60,000 samples at a time. Depending on your system's memory, you could increase or decrease the number of samples loaded on each iteration. This will also export the associated probe information as a '.prb' file, if present.

[ ]:
```
from kilosort import io

# NOTE: Data will be saved as np.int16 by default since that is the standard
```

(continues on next page)

(continued from previous page)

```
#       for ephys data. If you need a different data type for whatever reason
#       such as `np.uint16`, be sure to update this.
dtype = np.int16
filename, N, c, s, fs, probe_path = io.spikeinterface_to_binary(
    recording, DATA_DIRECTORY, data_name='data.bin', dtype=dtype,
    chunksize=60000, export_probe=True, probe_name='probe.prb'
    )
```

If no probe information was loaded through spikeinterface, you will need to specify the probe yourself, either as a .prb file or as a .json with Kilosort4's expected format. Follow the steps at the bottom of this notebook, or see the tutorial notebook titled, 'Creating a Kilosort4 probe dictionary'

At this point, it's a good idea to open the Kilosort gui and check that the data and probe appear to have been loaded correctly and no settings need to be tweaked. You will need to input the path to the binary datafile, the folder where results should be saved, and select a probe file.

```
python -m kilosort
```

From there, you can either launch Kilosort using the GUI or run the next notebook cell to run it through the API.

3. Run Kilosort (API)

Note that in this case, we don't actually need to specify a probe since it's the same as the default Neuropixels 1 configuration. For handling different probe layouts, provide your own .prb file and/or see the tutorial on creating a new probe file from scratch.

[ ]:
```
from kilosort import run_kilosort

# NOTE: 'n_chan_bin' is a required setting, and should reflect the total number
#       of channels in the binary file, while probe['n_chans'] should reflect
#       the number of channels that contain ephys data. In many cases these will
#       be the same, but not always. For example, neuropixels data often contains
#       385 channels, where 384 channels are for ephys traces and 1 channel is
#       for some other variable. In that case, you would specify
#       'n_chan_bin': 385.
settings = {'fs': fs, 'n_chan_bin': c}

# Specify probe configuration.
assert probe_path is not None, 'No probe information exported by SpikeInterface'
probe = io.load_probe(probe_path)

# This command will both run the spike-sorting analysis and save the results to
# `DATA_DIRECTORY`.
ops, st, clu, tF, Wall, similar_templates, is_ref, est_contam_rate = run_kilosort(
    settings=settings, probe=probe, filename=filename, dtype=dtype
    )
```

Whether you used the gui or the API, the results can now be browsed in Phy from a terminal with:

```
phy template-gui <DATA_DIRECTORY>/kilosort4/params.py
```

(replacing DATA_DIRECTORY with the appropriate path)

9.1 Using the API to load data through SpikeInterface without copying

We also provide a wrapper for SpikeInterface recordings that will allow them to be read by Kilosort4 without first copying the data to binary. However, in most cases the copy-to-binary approach is recommended since the binary file can be read by the Kilosort4 gui and Phy, while other dataformats cannot. To use this option, you will still need to provide the probe configuration and the filename for the source file.

[ ]:
```
# First get `recording` through SpikeInterface and specify probe & settings,
# as described above.
wrapper = io.RecordingExtractorAsArray(recording)
ops, st, clu, tF, Wall, similar_templates, is_ref, est_contam_rate = run_kilosort(
    settings=settings, probe=probe, filename=filepath, file_object=wrapper
)
```

9.2 Instructions for additional data formats

The following cells demonstrate how to load other dataformats using spikeinterface. Use these code snippets to modify the first cell of this notebook to work with different datasets.

See SpikeInterface's documentation for additional details.

SpikeGLX

[ ]:
```
# NOTE: You do not need to load SpikeGLX data this way. It is already saved in
#       binary format, so you should just point Kilosort 4 to the .bin file.
from spikeinterface.extractors import read_spikeglx
# Provide path to directory containing .bin file.
filepath = Path(".../TEST_20210920_0_g0/")
recording = read_spikeglx(filepath)
```

Blackrock

[ ]:
```
from spikeinterface.extractors import read_blackrock
# Provide path to nsX file, not nev file.
filepath = Path(".../file_spec_3_0.ns6")
recording = read_blackrock(filepath)
```

Neuralynx

[ ]:
```
from spikeinterface.extractors import read_neuralynx
# Provide path to directory containing .Ncs file(s).
filepath = Path("C:/code/ephy_testing_data/neuralynx/BML/original_data/")
recording = read_neuralynx(filepath)
```

Openephys

[ ]:
```
from spikeinterface.extractors import read_openephys
filepath = Path(".../ecephys_tutorial_v2.5.0.nwb")
# NOTE: Open Ephys data can have multiple streams, specify `stream_id` to
#       load different ones.
recording = read_openephys(filepath)
```

Intan

[ ]:
```
from spikeinterface.extractors import read_intan
# NOTE: You will need to select the appropriate data stream. If you run without
#       specifying `stream_id`, you will get an error message explaining what
#       each stream corresponds to.
filepath = Path(".../intan_rhs_test_1.rhs")
recording = read_intan(filepath, stream_id='0')
```

9.3 Exporting probes from SpikeInterface

To create a new probe file, we can use `ProbeInterface` (a subpackage of `SpikeInterface`). You will also need `matplotlib` if you want to visualize the probe geometry (recommended).

You can follow the steps in this ProbeInterface tutorial to create a new probe from scratch, or to plot a probe to check that it is configured correctly.

Then use the following steps to export to a .prb file that can be read by Kilosort4.

[ ]:
```
from probeinterface import ProbeGroup, write_prb

probe = ...  # From SpikeInterface tutorial, or recording.get_probe()

# Multiple probes can be added to a ProbeGroup. We only have one, but a
# ProbeGroup wrapper is still necessary for `write_prb` to work.
pg = ProbeGroup()
pg.add_probe(probe)
# CHANGE THIS PATH to wherever you want to save your probe file.
write_prb('.../test_prb.prb', pg)
```

Note that the probe object must have channel indices specified in order to save to a .prb file. If `write_prb` results in an error indicating these are not set, you can use the `probe.set_device_channel_indices` method to set them. For example, for a 24-channel probe with all contacts connected:

[ ]:
```
# Must set channel indices for .prb files.
# Indicate "not connected" with a value of -1.
probe.set_device_channel_indices(np.arange(24))
```

CHAPTER

TEN

KILOSORT4 API

10.1 run_kilosort

kilosort.run_kilosort.**compute_drift_correction**(*ops*, *device*, *tic0=nan*, *progress_bar=None*, *file_object=None*)

Compute drift correction parameters and save them to *ops*.

Parameters

- **ops** (*dict*) – Dictionary storing settings and results for all algorithmic steps.
- **device** (*torch.device*) – Indicates whether *pytorch* operations should be run on cpu or gpu.
- **tic0** (*float; default=np.nan.*) – Start time of *run_kilosort*.
- **progress_bar** (*TODO; optional.*) – Informs *tqdm* package how to report progress, type unclear.
- **file_object** (*array-like file object; optional.*) – Must have 'shape' and 'dtype' attributes and support array-like indexing (e.g. [:100,:], [5, 7:10], etc). For example, a numpy array or memmap.

Returns

- **ops** (*dict*)
- **bfile** (*kilosort.io.BinaryFiltered*) – Wrapped file object for handling data.

kilosort.run_kilosort.**compute_preprocessing**(*ops*, *device*, *tic0=nan*, *file_object=None*)

Compute preprocessing parameters and save them to *ops*.

Parameters

- **ops** (*dict*) – Dictionary storing settings and results for all algorithmic steps.
- **device** (*torch.device*) – Indicates whether *pytorch* operations should be run on cpu or gpu.
- **tic0** (*float; default=np.nan*) – Start time of *run_kilosort*.
- **file_object** (*array-like file object; optional.*) – Must have 'shape' and 'dtype' attributes and support array-like indexing (e.g. [:100,:], [5, 7:10], etc). For example, a numpy array or memmap.

Returns

ops

Return type
dict

kilosort.run_kilosort.**detect_spikes**(*ops*, *device*, *bfile*, *tic0=nan*, *progress_bar=None*)

Run spike sorting algorithm and save intermediate results to *ops*.

Parameters

- **ops** (*dict*) – Dictionary storing settings and results for all algorithmic steps.
- **device** (*torch.device*) – Indicates whether *pytorch* operations should be run on cpu or gpu.
- **bfile** (*kilosort.io.BinaryFiltered*) – Wrapped file object for handling data.
- **tic0** (*float; default=np.nan.*) – Start time of *run_kilosort*.
- **progress_bar** (*TODO; optional.*) – Informs *tqdm* package how to report progress, type unclear.

Returns

- **st** (*np.ndarray*) – 1D vector of spike times for all clusters.
- **clu** (*np.ndarray*) – 1D vector of cluster ids indicating which spike came from which cluster, same shape as *st*.
- **tF** (*np.ndarray*) – TODO
- **Wall** (*np.ndarray*) – TODO

kilosort.run_kilosort.**get_run_parameters**(*ops*) → list

Get *ops* dict values needed by *run_kilosort* subroutines.

kilosort.run_kilosort.**initialize_ops**(*settings*, *probe*, *data_dtype*, *do_CAR*, *invert_sign*, *device*) → dict

Package settings and probe information into a single *ops* dictionary.

kilosort.run_kilosort.**run_kilosort**(*settings*, *probe=None*, *probe_name=None*, *filename=None*, *data_dir=None*, *file_object=None*, *results_dir=None*, *data_dtype=None*, *do_CAR=True*, *invert_sign=False*, *device=None*, *progress_bar=None*, *save_extra_vars=False*)

Run full spike sorting pipeline on specified data.

Parameters

- **settings** (*dict*) – Specifies a number of configurable parameters used throughout the spike sorting pipeline. See *kilosort/parameters.py* for a full list of available parameters. NOTE: *n_chan_bin* must be specified here, but all other settings are

  optional.
- **probe** (*dict; optional.*) – A Kilosort4 probe dictionary, as returned by *kilosort.io.load_probe*.
- **probe_name** (*str; optional.*) – Filename of probe to use, within the default *PROBE_DIR*. Only include the filename without any preceeding directories. Will ony be used if *probe is None*. Alternatively, the full filepath to a probe stored in any directory can be specified with *settings = {'probe_path': ...}*. See *kilosort.utils* for default *PROBE_DIR* definition.
- **filename** (*str or Path; optional.*) – Full path to binary data file. If specified, will also set *data_dir = filename.parent*.

- **data_dir** (*str or Path; optional.*) – Specifies directory where binary data file is stored. Kilosort will attempt to find the binary file. This works best if there is exactly one file in the directory with a .bin, .bat, .dat, or .raw extension. Only used if *filename is None*. Also see *kilosort.io.find_binary*.
- **file_object** (*array-like file object; optional.*) – Must have 'shape' and 'dtype' attributes and support array-like indexing (e.g. [:100,:], [5, 7:10], etc). For example, a numpy array or memmap. Must specify a valid *filename* as well, even though data will not be directly loaded from that file.
- **results_dir** (*str or Path; optional.*) – Directory where results will be stored. By default, will be set to *data_dir / 'kilosort4'*.
- **data_dtype** (*str or type; optional.*) – dtype of data in binary file, like *'int32'* or *np.uint16*. By default, dtype is assumed to be *'int16'*.
- **do_CAR** (*bool; default=True.*) – If True, apply common average reference during preprocessing (recommended).
- **invert_sign** (*bool; default=False.*) – If True, flip positive/negative values in data to conform to standard expected by Kilosort4.
- **device** (*torch.device; optional.*) – CPU or GPU device to use for PyTorch calculations. By default, PyTorch will use the first detected GPU. If no GPUs are detected, CPU will be used. To set this manually, specify *device = torch.device(<device_name>)*. See PyTorch documentation for full description.
- **progress_bar** (*tqdm.std.tqdm or QtWidgets.QProgressBar; optional.*) – Used by sorting steps and GUI to track sorting progress. Users should not need to specify this.
- **save_extra_vars** (*bool; default=False.*) – If True, save tF and Wall to disk after sorting.

Raises

ValueError – If settings[*n_chan_bin*] is None (default). User must specify, for example: *run_kilosort(settings={'n_chan_bin': 385})*.

Returns

Description TODO

Return type

ops, st, clu, tF, Wall, similar_templates, is_ref, est_contam_rate

kilosort.run_kilosort.**save_sorting**(*ops*, *results_dir*, *st*, *clu*, *tF*, *Wall*, *imin*, *tic0=nan*, *save_extra_vars=False*)

Save sorting results, and format them for use with Phy

Parameters

- **ops** (*dict*) – Dictionary storing settings and results for all algorithmic steps.
- **results_dir** (*pathlib.Path*) – Directory where results should be saved.
- **st** (*np.ndarray*) – 1D vector of spike times for all clusters.
- **clu** (*np.ndarray*) – 1D vector of cluster ids indicating which spike came from which cluster, same shape as *st*.
- **tF** (*np.ndarray*) – TODO
- **Wall** (*np.ndarray*) – TODO
- **imin** (*int*) – Minimum sample index used by BinaryRWFile, exported spike times will be shifted forward by this number.

- **tic0** (*float; default=np.nan.*) – Start time of *run_kilosort*.

Returns

- **ops** (*dict*)
- **similar_templates** (*np.ndarray*)
- **is_ref** (*np.ndarray*)
- **est_contam_rate** (*np.ndarray*)

kilosort.run_kilosort.**set_files**(*settings*, *filename*, *probe*, *probe_name*, *data_dir*, *results_dir*)

Parse file and directory information for data, probe, and results.

PYTHON MODULE INDEX

k

INDEX