LUND UNIVERSITY

Automated pipeline for processing fluorescence calcium imaging microscopy data and electrophysiological measurements from 3D brain organoids

Mattis Knulst

Master's Degree Project in Bioinformatics, 30 credits

2021

Department of Biology
Lund University

Bioinformatics Program, 2021
Advance Access Publication Date: 20 August 2021
Bioinformatics software development

Software engineering

Automated pipeline for processing fluorescence calcium imaging microscopy data and electrophysiological measurements from 3D brain organoids

Mattis Knulst1, *

[1]Department of Biology, Biology building, Sölvegatan 35, Lund, Sweden

*To whom correspondence should be addressed.

Supervisor: Dr. Bruno Fontinha, a:head bio ag

Received on 2021-08-23; revised on 2021-12-01; accepted on 2021-12-01

Abstract

Motivation: 3D brain organoids are biological models of human brian development from which a rich body of data can be extracted that allows for insights about neuronal dynamics. Several tools, such as fluorescence calcium imaging microscopy and electrophysiological recordings are being used to asses how 3D brain organoid dynamics recapitulate what is known from the human brain. In-silico automated workflows that process such datasets on a large scale in a robust and reproducible manner need to be developed.

Results: An automated, modular and scalable workflow was written using Python 3.9. ImageJ, Inscopix API and Caiman/CNMFe that outputs neuronal cell traces and additional scripts were produced to enable large scale analysis of Axion multi-electrode array output.

Contact: mattisknulst@gmail.com

Supplementary information: Supplementary information is found in the appendix. Documentation (README) sent separately.

Background

Introduction

Recent advances in cell cultures have enabled the creation of three-dimensional pluripotent stem cell aggregates that organize themselves into structures that closely resemble that of the human brain. The usage of stem cells means brain models can be created with the same genetic content that can be found in humans (Kim et al., 2020). These methods could be important both for prospective pharmaceutical research, where organs from animal models normally act surrogates for their human counterparts and in personalized medicine. While the methods for growing these cultures and techniques for observing them are getting more common (Marton & Pașca, 2020), there is a great need to scale up and automate the in-silico processing of the raw data.

Data types and general workflow

This paper will treat the automatization of the processing of two types of raw data; calcium fluorescence microscopy imaging (FM) data collected with an Olympus spinning disk confocal inverted microscope (IX83), equipped with a dual-camera Yokogawa W1 spinning disk (SD) for fast confocal acquisition, controlled by cellSens (https://www.olympus-lifescience.com/en/software/cellsens/) software, videos recorded using a 20x/0.75 UPLSApo WD with a 0.6 mm objective and 488 nm laser on brain organoids with the calcium reporter GCAMP6s (TW et al., 2013). and multi-electrode array data (MEA) using Axion's Maestro Pro multi-well microelectrode array system with a 48-well plate (providing a simultaneous recording of 768 channels). The computational processing of the data was performed on a workstation with 4 logical processor units and 128 GB of RAM, which during development was upgraded to 36 logical CPUs and 64 GB of RAM. The operating system was Windows 10 Pro.

The typical raw FM data takes up approximately 10 GB of disk space (for a 5-minute recording, 5000 frames at 1024x1024 pixel (px) resolution with one frame acquired every 65 msec, ~15.4 Hz) and is contained in a binary library which can be read with the Bio-Formats library (Linkert et al., 2010) and exported to a tagged image file (TIF). This, as well as spatial downsampling by 2x bilinear interpolation and removal of the leading 100 frames, is performed with ImageJ (Rueden et al., 2017). The output consists of an intermediate video file which is 25% the size of the original raw data and typically has 4900 frames at 512x512 px resolution.

The downsampled file is processed further using Inscopix (https://www.inscopix.com/software-analysis) proprietary Python API. Inscopix is a provider of hardware and software for FM brain imaging and their API contains many functions that can be incorporated in this workflow. First there is a preprocessing API function, which prepares the image stack (attempting to correct missing pixels) before it is sent through bandpass filtering to remove unwanted high or low-frequency fluorescence oscillations per pixel. Next, regions of interest (ROIs) that are showing activity are identified using the "constrained non-negative matrix factorization for microendoscopic data" (CNMFe) algorithm which allows for neuronal ROI segmentation and extraction of each ROI's calcium temporal dynamics (Zhou et al., 2018) contained within a Docker (by default Inscopix provides their own default version of CNMFe during setup), but accessed via the Inscopix API. This method (CNMFe) is advantageous in that it allows for a better resolution of spatially overlapping ROIs, provides good modelling of background activity and works well even in situations with low signal-to-noise ratio compared to other methods (Pnevmatikakis et al., 2014, 2016).

The ROI information is used to extract a trace, which is the raw fluorescence intensity value by frame and this information is used to detect events corresponding to neuronal spike activity (event detection). Up to this point, the intermediary information is held within binary files, and at the end of the Inscopix API module, it is exported to comma-separated values (CSV) tables. One intermediary file and a directory containing the settings and temporary files for Caiman is produced which take up roughly little over the in-going downsampled file size on the hard drive, these can be safely removed to use less storage. The CSV table output contains information about the quality of the data and ROI traces are listed in columns together with the corresponding time information.

CNMFe is the most computationally expensive step both regarding memory consumption and CPU load. The in-going video file shows populations of identified neural components (somata and dendrites) with variable shapes inside the previously imaged area of the organoid, the algorithm must separate fluorescing components and infer the underlying action potentials, while accounting for the fact that the analysis is done on a two-dimensional representation of a three-dimensional system, the calcium reporter will also have a slower decay than the time it takes for a neuron to return to resting potential (Pnevmatikakis et al., 2016). The most important parameters in inferring traces are the average component diameters, specified in pixels, minimum peak to noise ratio and with regards to the memory consumption, how to break images up and process them in patches (new in Inscopix 1.6).

Because traces are sometimes anomalous, a third module was written and introduced after this step that identified and flagged anomalous ROIs and a fourth module is planned to do different statistical analyses on the traces but is still being developed at the time of writing.

For the MEA analysis, the manufacturers' software can directly output statistics and metrics based on the raw input data (changes in voltage over time), this output is contained in a single CSV table containing merged tables of different shapes which need to be parsed and summarized in a single table so that individuals and experimental groups of organoids can be plotted against each other. The experimental setup of MEA is still being worked out, the focus here has been to enable parsing a large number of output files from this technology.

Aim

This paper aims to develop and evaluate methods to automate the above-specified steps into a workflow that can be reliably used with minimal user interaction.

- This workflow should be *scalable*, able to utilize hardware upgrades as well as accommodate large input batch sizes.
- It should be *reproducible*, meaning that the workflow application itself is portable and can be set up on other systems, where the installation can produce the same results.
- Further, the workflow application should be *robust*, if there are disturbances during processing, there should be checkpoints that minimize losses in processing time by picking up where the failure occurred automatically once restarted. There should also be logging for the scope of each layer of software so that if there are errors they can be quickly identified and remedied.
- All relevant software must be provided with *documentation* that explains the scripts and how they work well

enough that a user with minimal experience of working on the command line can use the software and validate results, as well as extend the pipeline with new modules.

- The documentation also needs to explain the process of developing new scripts that are intended to be part of the pipeline to streamline the future inclusion of these additional steps into the workflow and/or the re-ordering of the same. To facilitate this, the scripts need to be *readable* and *modular*.
- The directory structure for batch intermediary and result output needs to be planned that is *easily navigated* by users.

Methods and Results

Pipeline application

The first consideration was whether to use a workflow management system such as Snakemake (Mölder et al., 2021) to automate the pipeline. Since the workstation was running Windows, this may have been possible through Windows Subsystem for Linux (WSL), but the specific setup meant that the Inscopix software environment was installed outside WSL, moving it inside WSL would have prevented the usage of the Inscopix GUI (the software license allows for only a single installation). A decision was therefore made to write a custom workflow application in Python. Since the Inscopix API depended on Conda, the workflow application could use other Conda environments for modules with conflicting requirements, adding minimal overhead in the installation process and software layers.

To make system calls from Python, the subprocess module (https://docs.python.org/3/library/subprocess.html) was used, this module has improved in collecting and returning child process terminal output on Windows systems since Python version 3.6 which is the required version of the Inscopix API, leading to the main body of the pipeline being written in Python 3.9 and having an isolated environment.

ImageJ has a CLI interface, which allows for running the tool in headless mode or batch mode, unfortunately, Bio-Formats is incompatible with the headless mode, so the batch mode is used instead with a macro script generated from a template by the pipeline. The major difference between this solution and the headless mode is that the batch mode results in a splash screen appearing on the desktop when a file is being processed, which may be distracting if the pipeline is running in the background. ImageJ additionally has a portable installation that is packaged inside the main directory.

The Inscopix API was implemented as a module inside a python package directory that sits in the main directory, because of requirement conflicts this script cannot be imported by the main script and is instead run in the Inscopix custom environment using Conda.

The main script is kept condensed by having a main function that imports modules in the pipeline, runs steps in the pipeline, logs output and errors as well as checks for final outputs at each module and moves to the next step if this output is present. Additional modules/steps, no matter the software used, can be added to the main function using a custom function that makes system calls. Further, the main function is parallelized using the python multiprocessing module (https://docs.python.org/3/library/multiprocessing.html).

The setup process only requires extraction of the pipeline application main directory on a computer where the Inscopix software has been installed, as well as the creation of the environment for the pipeline application itself and the accompanying anomaly detection module. In the hour of writing the pipeline solution has processed 1393 FM files, each file taking on average 25 minutes to process.

Anomaly detection

ROI traces should ideally have a fast rise and slow decay dependent on the action of GCAMP6s and neuron function (Chen et al., 2013), see fig 1a (appendix A). To explore options for automatically identifying anomalous ROI traces 7029 manually annotated (accepted or rejected) were collected. An estimated 5.7 % of the ROIs (in the manually annotated data) produced anomalous traces, see appendix A. An exploratory data analysis was conducted, and a random forest classifier model was trained using Scikit-Learn (https://scikit-learn.org/stable/user_guide.html) and then implemented as a python module in the pipeline. The data was split (using stratified sampling) into a training set with 3953 traces for training and 1318 for testing, after tuning (optimizing for high recall score) the classifier was evaluated on a validation set containing 1758 traces. The model has an F1 score of 0.77 (harmonic mean of precision and recall) for traces that should be rejected (see appendix A). The module produces a copy of the original ROI traces CSV including only the accepted traces and plots all traces output along with model predictions so that they can be reviewed.

Multi-electrode array data

The MEA data analysis software from Axion lacked a CLI interface and consisted of compiled Matlab scripts. The software can process one 48-well plate at a time in what will remain a manual step, but the results of this per-plate analysis must be directly exported to a CSV file. This CSV is the result of combining many tables and metrics into one table and is not suitable for analysis without parsing the data first. A parser script was written in Python which can iterate over many CSV outputs, extracting metrics and collecting them together with information about individual organoids, well number and original file in a Pandas (https://pandas.pydata.org/docs/) data frame, as well as exporting the result to a new portable CSV. The parser is written as a Python class that can be instantiated for each input file, allowing it to be used by other scripts working with batch analysis of the Axion output.

Discussion

The data processing was automated successfully on the target system and is modularized in such a way that it is readable and extendable with minimal Python knowledge. The pipeline is robust and has checkpoints for dealing with interruptions during processing. It is also portable and can be quickly set up on a new system. Inscopix version 1.6 which was released during development has been implemented in a future patch. The pipeline uses Git for version control and a separate Git branch that can be merged to the main branch to apply the Inscopix 1.6 update. During development, the main challenge was the proper implementation of Caiman/CNMFe. Most of the processing time is consumed by this step, the package is included with a default Inscopix installation, but had very long (>1 hour) processing times on the target systems. A Docker image was written interactively before the development of this pipeline and based on the GitHub installation of Caiman (https://github.com/flatironinstitute/CaImAn). The Docker container was implemented using the Docker Python software development kit (SDK) (https://docker-py.readthedocs.io/en/stable/). There were several challenges with the implementation, mainly the lack of a make file (which would contain the instructions to generate a new Docker image, and which could be altered to avoid limitations in the original build). Additionally, the Docker SDK changes affecting the Inscopix API were undocumented, and the container was named in such a way that the Docker could only have one instance at any time. This was corrected by giving container instances unique names, but parallel Docker containers would always set all available CPUs to 100%. While the memory of the target system would allow for at least 4 parallel processes, the highest safe setting was determined to be 2. This is probably due to the parallel processing of patches, before Inscopix 1.6 the CNMFe function only took a keyword argument for the number of processes per patch, in the new version the processing mode can be set to parallel or sequential. Sequential should be preferred here if Inscopix API functions are used at a lower level where parallel processes are managed from the top to avoid unpredictable results when running the entire pipeline in parallel.

During testing of the new version of the Inscopix software, Caiman's recommended installation (https://caiman.readthedocs.io/en/master/Installation.html#installing-caiman) was also tested, retrieved from conda-forge, and this version of the pipeline will be able to run completely without the use of Docker. The update will also result in slight changes to the output. ROI traces will now be normalized in the Inscopix CNMFe function using ΔF/F (change in fluorescence normalized to the background fluorescence). The same normalization needs to be applied to the random forest classification model in anomaly detection.

The anomaly detection module has faced a few challenges, the data used for training the random forest model was exclusively 4900 frames long and since dimensionality reduction/feature selection was not applied, the model can only accurately label vectors with dimensions (,4900). Since this is an unbalanced data set, several strategies were applied to increase the proportion of traces that should be rejected. The database was expanded two times, first with additional manually curated traces and the second time with data produced while configuring CNMFe's pixel seeding operation mode to extract the components spatial footprints, which contained a high proportion of rejected traces. Splitting the data was also reworked to make sure the proportion of rejected traces in the randomly selected sets remained the same. The F1 score of the model (0.77) likely reflects the fact that while a large proportion of rejected traces can be almost immediately identified by a human observer (see appendix A) there is a significant proportion of edge cases. A recent paper exploring hyperparameter optimization for machine learning models that detect anomalous ROI traces determined that human inter-rater agreement is 87% (Tran et al., 2020). Further, while some traces that should be rejected are flagged as false positives, there is not a considerable cost in traces that should be accepted (1% accepted traces mislabeled as rejected in the validation set, against 74% correctly labelled rejected traces). Since neuronal traces are likely to be heterogeneous during varying experimental conditions, this module only adds a clean table file, without destroying the in-going data and produces labelled plots for every trace, providing inherent quality control for this step.

Conclusion

The MEA data processing script efficiently parses output files and prepares a collected table output file for large scale analyses.

In the FM data processing, the pipeline has efficiently reduced a workflow that required manual script editing and file operations with each step and without the advantage of parallel processing throughout the workflow to a parallelized single step that can be started with minimal user interaction from the command line. The pipeline application produces logs for each layer, is well documented and has checkpoints to skip completed steps in case execution is interrupted. Intermediary and result files are kept in a numbered tree structure with a parent directory name based on the in-going raw files so that large batch outputs can be quickly navigated. The setup process on a new system is minimal.

After a major update to the Inscopix API, that module has been reworked with a future patch that can be applied, which increases the potential for taking advantage of parallel processing.

The software ecosystem involved in processing brain organoid data is still actively developing to meet the requirements of large-scale operations, demanding automated data processing pipelines to use version control, be flexible, modular and continuously updated to follow. The solution presented here takes advantage of existing resources in a target system to devise such a workflow, while at the same time being portable, reproducible on other systems and remaining approachable for the user. While stringing a set of algorithms together for massive analysis enables high throughput, it appears that there is a risk for a loss in the quality of output data. Filtering the output of the pipeline such as was done in the present pipeline is an intuitive first step, but in the future it should be explored whether especially the CNMF-e algorithm can be rewritten to allow for a greater flexibility in input data while producing robust and high-quality output.

Acknowledgements: The author would like to thank a:head bio ag for the opportunity to write and analyze the pipeline. In particular Dr. Bruno Fontinha and Joanna Kaczanowska have provided great feedback on the implementation and practical application of the pipeline. Finally, none of the pipeline would have been written at all if it weren't for the unwavering support of Sara Albertz. Thank you all!

References

Chen, T. W., Wardill, T. J., Sun, Y., Pulver, S. R., Renninger, S. L., Baohan, A., Schreiter, E. R., Kerr, R. A., Orger, M. B., Jayaraman, V., Looger, L. L., Svoboda, K., & Kim, D. S. (2013). Ultrasensitive fluorescent proteins for imaging neuronal activity. *Nature*, *499*(7458), 295–300. https://doi.org/10.1038/nature12354

Kim, J., Koo, B. K., & Knoblich, J. A. (2020). Human organoids: model systems for human biology and medicine. *Nature Reviews Molecular Cell Biology*, *21*(10), 571–584. https://doi.org/10.1038/s41580-020-0259-3

Linkert, M., Rueden, C. T., Allan, C., Burel, J.-M., Moore, W., Patterson, A., Loranger, B., Moore, J., Neves, C., MacDonald, D., Tarkowska, A., Sticco, C., Hill, E., Rossner, M., Eliceiri, K. W., & Swedlow, J. R. (2010). Metadata matters: access to image data in the real world. *Journal of Cell Biology*, *189*(5), 777–782. https://doi.org/10.1083/JCB.201004104

Marton, R. M., & Pașca, S. P. (2020). Organoid and Assembloid Technologies for Investigating Cellular Crosstalk in Human Brain Development and Disease. *Trends in Cell Biology*, *30*(2), 133–143. https://doi.org/10.1016/J.TCB.2019.11.004

Mölder, F., Jablonski, K. P., Letcher, B., Hall, M. B., Tomkins-Tinch, C. H., Sochat, V., Forster, J., Lee, S., Twardziok, S. O., Kanitz, A., Wilm, A., Holtgrewe, M., Rahmann, S., Nahnsen, S., & Köster, J. (2021). Sustainable data analysis with Snakemake. *F1000Research 2021 10:33*, *10*, 33. https://doi.org/10.12688/f1000research.29032.1

Pnevmatikakis, E. A., Gao, Y., Soudry, D., Pfau, D., Lacefield, C., Poskanzer, K., Bruno, R., Yuste, R., & Paninski, L. (2014). *A structured matrix factorization framework for large scale calcium imaging data analysis*. 1–16. http://arxiv.org/abs/1409.2903

Pnevmatikakis, E. A., Soudry, D., Gao, Y., Machado, T. A., Merel, J., Pfau, D., Reardon, T., Mu, Y., Lacefield, C., Yang, W., Ahrens, M., Bruno, R., Jessell, T. M., Peterka, D. S., Yuste, R., & Paninski, L. (2016). Simultaneous Denoising, Deconvolution, and Demixing of Calcium Imaging Data. *Neuron*, *89*(2), 285. https://doi.org/10.1016/j.neuron.2015.11.037

Rueden, C. T., Schindelin, J., Hiner, M. C., DeZonia, B. E., Walter, A. E., Arena, E. T., & Eliceiri, K. W. (2017). ImageJ2: ImageJ for the next generation of scientific image data. *BMC Bioinformatics*, *18*(1), 1–26. https://doi.org/10.1186/s12859-017-1934-z

Tran, L. M., Mocle, A. J., Ramsaran, A. I., Jacob, A. D., Frankland, P. W., & Josselyn, S. A. (2020). Automated Curation of CNMF-E-Extracted ROI Spatial Footprints and Calcium Traces Using Open-Source AutoML Tools. *Frontiers in Neural Circuits*, *14*(July), 1–9. https://doi.org/10.3389/fncir.2020.00042

TW, C., TJ, W., Y, S., SR, P., SL, R., A, B., ER, S., RA, K., MB, O., V, J., LL, L., K, S., & DS, K. (2013). Ultrasensitive fluorescent proteins for imaging neuronal activity. *Nature*, *499*(7458), 295–300. https://doi.org/10.1038/NATURE12354

Zhou, P., Resendez, S. L., Rodriguez-Romaguera, J., Jimenez, J. C., Neufeld, S. Q., Giovannucci, A., Friedrich, J., Pnevmatikakis, E. A., Stuber, G. D., Hen, R., Kheirbek, M. A., Sabatini, B. L., Kass, R. E., & Paninski, L. (2018). Efficient and accurate extraction of in vivo calcium signals from microendoscopic video data. *ELife*, *7*, 1–37. https://doi.org/10.7554/eLife.28728

Appendix A:

Anomaly detection

Table 1: *Showing validation scores for 0=Rejected and 1=Accepted traces in the validation data set*

	Precision	Recall	F1-score	Support
0	0.80	0.74	0.77	91
1	0.99	0.99	0.99	1667
Accuracy			0.98	1758
Macro avg	0.89	0.89	0.98	1758
Weighted avg	0.89	0.98	0.98	1758

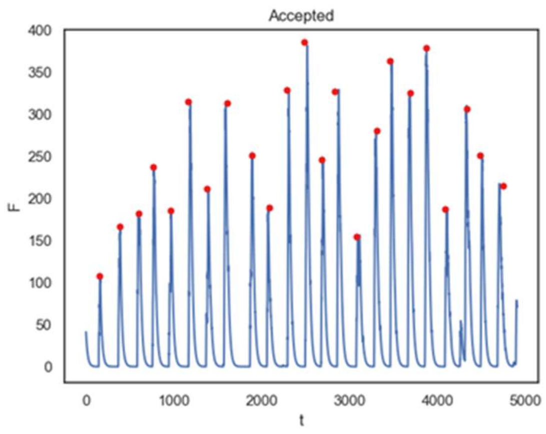


Figure 1: Showing the expected fluorescence peaks, peak events are marked with red dots and show a fast increase and calcium reporter decay time dependent decrease.

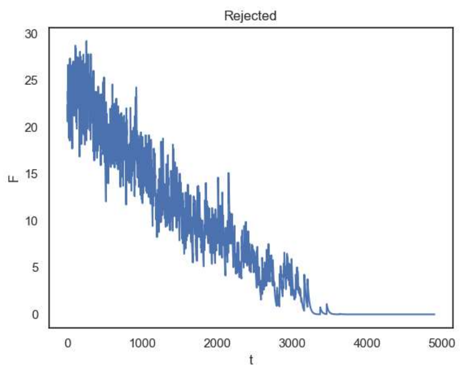


Figure 2: A typical rejected trace, with irregular peaks that do not decay to a baseline for many frames beyond the decay time.

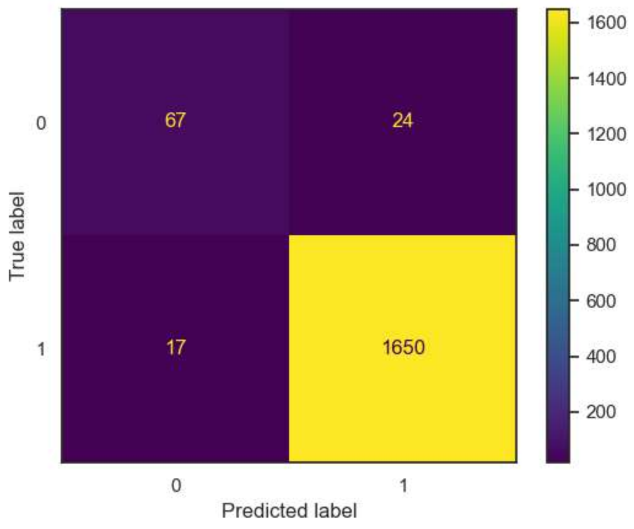
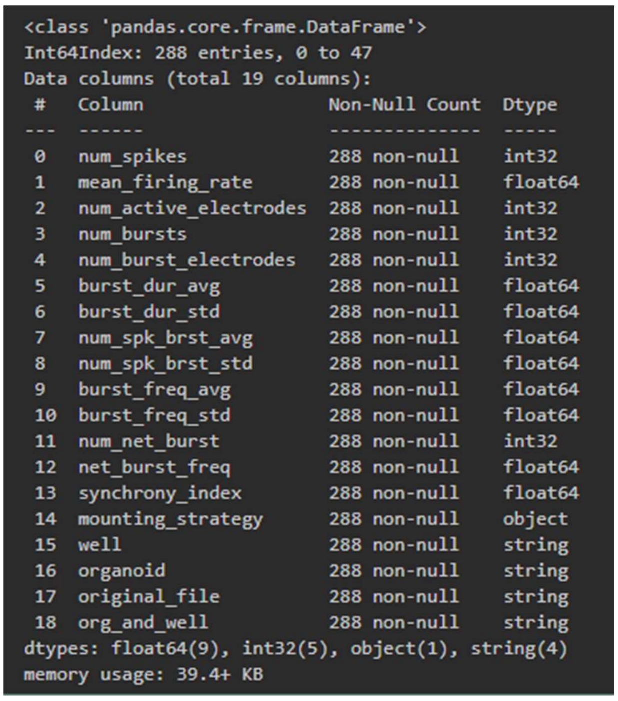


Figure 3: Confusion matrix showing the results of the random forest classifier on the validation set

Appendix B:

Axion multi-electrode array



```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 288 entries, 0 to 47
Data columns (total 19 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   num_spikes             288 non-null    int32
 1   mean_firing_rate       288 non-null    float64
 2   num_active_electrodes  288 non-null    int32
 3   num_bursts             288 non-null    int32
 4   num_burst_electrodes   288 non-null    int32
 5   burst_dur_avg          288 non-null    float64
 6   burst_dur_std          288 non-null    float64
 7   num_spk_brst_avg       288 non-null    float64
 8   num_spk_brst_std       288 non-null    float64
 9   burst_freq_avg         288 non-null    float64
 10  burst_freq_std         288 non-null    float64
 11  num_net_burst          288 non-null    int32
 12  net_burst_freq         288 non-null    float64
 13  synchrony_index        288 non-null    float64
 14  mounting_strategy      288 non-null    object
 15  well                   288 non-null    string
 16  organoid               288 non-null    string
 17  original_file          288 non-null    string
 18  org_and_well           288 non-null    string
dtypes: float64(9), int32(5), object(1), string(4)
memory usage: 39.4+ KB
```

Figure 1: Information extracted from multiple Axion MEA output files and exported to a single CSV table.

User guide a:head pipeline application v 1.3

Configuration

This is the current version of the conf.ini with instructions on what to change (if required):

```
[INFO]
author = Mattis Knulst
email = mattisknulst@gmail.com

[IMAGEJ]
script = pipeline_modules/file_convertion_frames_removal_SD.ijm
imagej_exe = fiji-win64/Fiji.app/ImageJ-win64.exe

[GENERAL]
watched_folder = C:/dir/in_folder,D:/other/dir #comma-separated, single line
results_folder = C:/dir/results_folder
version = 1.3

[ISX_OPTIONS]
num_processes=2
min_pnr = 20
event_threshold = 0.2

[SCRIPTS]
# not in use
```

Things to avoid:

- back-slashes (seems to work fine, but still always a good praxis)
- spaces in paths
- be aware that if the input folder is the parent directory of the output (results), this directory will be listed when the pipeline is run with the -l flag, obviously, don't pick this directory as input.
- keep everything on one line after a keyword in the configuration file, new lines will be read as new keywords or may cause errors

Running the scripts QUICK START (look below for setup instructions)

Make sure you have updated conf.ini! Also, be sure that docker is running. Open Anaconda Prompt (activate pipeline), enter:

```
cd path\to\pipeline\folder
conda activate pipeline
python pipeline.py -h
usage: pipeline.py [-h] [-i INPUT] [-v] [-l] [-w] [-o OUTPUT] [-p POOL] [-j JOBS]

Bringing pipeline execution to Windows!

optional arguments:
  -h, --help            show this help message and exit
  -i INPUT, --input INPUT
                        Specify single input directory
  -v, --version         show version and exit
  -l, --list            Lets you pick input from watched folder in conf.ini
  -w, --wildcard        iterate over folders matching a pattern
  -o OUTPUT, --output OUTPUT
                        override the output directory specified in conf.ini
  -p POOL, --pool POOL  Place copy of all final output csv files in this directory
  -j JOBS, --jobs JOBS  Specify number of parallel processes, default=2
```

Common use cases:

```
python pipeline.py --input H:/raw/20210520_experiment_1
```

List all raw files in all directories specified (as watched_folder) in the configuration file and place the output under the directory specified in conf.ini (default)

```
python pipeline.py --list
```

Choose single input directory from the watched folder in conf.ini (type all to process all)

```
python pipeline.py --wildcard
```

Start wildcard batch processing by specifying a pattern using * e.g. *plate3* will list all the directories containing the string ...plate3... or *202105* will list all directories that start (and contains the string, use with care) with 202105 and are found in the path specified in conf.ini watched_folder. This option will always display the results of your search and prompt you before starting the batch processing. It will do this for each of the specified input directories in the configuration file.

Finally:

```
python pipeline.py --jobs 6 --wildcard --output H:/my_batch_results
```

Will start a wildcard batch with 6 parallel processes and override the results folder in conf.ini, giving the processed directories a common root on drive H called my_batch_results.

EXTRA: For troubleshooting, it is recommended to redirect errors from Powershell to a log file (this is extremely helpful when trying to figure out why 1/300 files didn't finish correctly):

```
conda activate pipeline
python pipeline.py --list -j 4 2> error_log.txt
```

Setup Instructions

1. Make sure Inscopix is set up according to its own documentation first (this should involve installing Anaconda, but there may be an exception to the instructions for installing CNMFe/Caiman, see Inscopix new version notes at end of this document)
2. Open the anaconda powershell prompt
3. Enter:

```
# notice that the environment names are given here and in the pipeline.py script
# these are hardcoded so make sure to spell them correctly!
conda create --name pipeline -c bioconda -c conda-forge python=3.9 alive-progress
# hit enter, let it work
# then:
conda create --name anomaly_detection -c bioconda -c conda-forge numpy=1.20.2 \
python=3.9.2 scikit-learn=0.24.1 scipy=1.6.2 matplotlib==3.4.1 seaborn==0.11.1 pip
# it is possible you will have to remove the \ and put it all on a single line (Windows)
```

4. Download and extract the latest version of the pipeline scripts, it makes sense to put them somewhere near Inscopix, but this is not necessary for the scripts to work.
   IMPORTANT: Be sure to have read the previous section about the configuration file before running the pipeline!

USEFUL BASH COMMANDS: Validating the pipeline and monitor Docker

Bash is the right tool for batch file operations, and since you have WSL, you have Bash.
After starting WSL, note that paths will change from C:\dir to /mnt/c/dir, do NOT forget the leading /.
Navigate to the parent directory of the results.
Count all folders:

```
#there is always at least one folder called temp which should not be in the output so subtract 1
from result
find . -mindepth 1 -maxdepth 1 -type d | wc -l
```

List all folders that still have the temp_cnmfe directory:

```
 for incomplete in $(find . -type d -path "*/*/temp_cnmfe");do echo ${incomplete%/*};done
```

optionally, count them:

```
 for incomplete in $(find . -type d -path "*/*/temp_cnmfe");do echo ${incomplete%/*};done | wc -
l
```

Remove the entire 2_preprocessing directory, only in those folders where cnmfe failed:

```
 for incomplete in $(find . -type d -path "*/*/temp_cnmfe");do rm -rf ${incomplete%/*};done
```

This lets the pipeline automatically use the processed ImageJ file to redo the ISX part.
Now, maybe you are interested in all the files that do not have a temp_cnmfe:

```
find . -maxdepth 1 -type d -not -path "*/*/temp_cnmfe"
# and to count:
find . -maxdepth 1 -type d -not -path "*/*/temp_cnmfe" | wc -l
```

Wonder how many directories have cellset traces files?

```
for trace in ls */*/*cellset-traces*;do echo ${trace%*/*/*};done | sort | uniq
# and count:
for trace in ls */*/*cellset-traces*;do echo ${trace%*/*/*};done | sort | uniq |wc -l
```

And finally, any of these can instead of printing the results to screen, dump the output into a file:

```
find . -maxdepth 1 -type d -not -path "*/*/temp_cnmfe" > no_failed_cnmfe.txt
for trace in ls */*/*cellset-traces*;do echo ${trace%*/*/*};done | sort | uniq >
successfully_completed.txt
```

Now for Docker, which can be controlled from WSL

```
# list all active containers
docker container ls
# shut down a container gracefully
docker stop ID
# force stop container
docker kill ID
# view active container stats
docker stats
# loop to check up time, sleep value in seconds to wait before calling again
while true;do clear;docker container ls;sleep 10;done
# this loop may look similar to docker stats, but it is easier to
# copy the IDs in the first row
```

WSL is a very useful tool on Windows, following is an example Bash/Python script sitting in the home directory of WSL on the workstation.
If the Bash script is called validate.sh and set as executable (chmod +x ~/validate.sh) it can be called from Windows:
These scripts are not optimized code, but were written quickly to produce an updatable report in the WSL home dir to track pipeline progress.
They also serve well to demonstrate how to work with the directory structure produced by the pipeline.

```
bash -e "~/validate.sh" #from PowerShell
################
validate.sh
#!/bin/bash
raw_super_dir="/mnt/f/Raw_data_M40/"
find /mnt/d/pipeline_output -mindepth 1 -maxdepth 1 -type d > ~/out_dirs_list.txt
count=0
res_count=0
declare -i res_count #these should contain integers, not strings
declare -i count
while read dir
# count how many raw dirs have produced outputs
        do count+=$(find ${raw_super_dir}${dir##*/} -mindepth 1 -maxdepth 1 -type d | egrep -c
^/mnt/)
        res_count+=$(find $dir -mindepth 1 -maxdepth 1 -type d | awk 'BEGIN { FS = "/" } ; {
print $5 $6 }' | egrep -c "^[0-9]")
done < ~/out_dirs_list.txt
# counting the raw data based on the result folders
echo "This many should be processed"
echo $count

# counting how many of the listed raw data directories have produced a corresponding output
echo "This many raw data directories have produced output"
echo $res_count
```

```
# listing relevant files
echo "examining these files"
egrep "[0-9]+_[A-Z]+$" out_dirs_list.txt

echo -e "OUT_DIR_NAME\tFILE_NAME\tSTATUS\tPROCESSING_TIME\tNUM_COMPONENTS" > report.tsv
# this loop only works on the output that has output in the results folder
egrep "[0-9]+_[A-Z]+$" out_dirs_list.txt | while read dir
do
        for log_file in ${dir}/*timed_log.tsv
        do
                python3 time_diff.py $log_file
        done
done >> report.tsv
# now we need to look for results that should have been processed but are missing
# generate search list
for p in $(egrep "[0-9]+_[A-Z]+$" out_dirs_list.txt);
do find "/mnt/f/Raw_data_M40/"${p##*/} -maxdepth 1 -type d;
done > to_be_processed.txt
# check if expected directories have been created
while read line;
do
[[ -d "/mnt/d/pipeline_output/"${line#*/*/*/*/} ]] || echo -e
"NA\t/mnt/d/pipeline_output/"${line#*/*/*/*/}"\t-1\tNA\tNA"
# this checks for the glob expanded directories and if the directory does not exist prints the
directory and a -1 in the table
done < to_be_processed.txt >> report.tsv
```

```
########################################
#time_diff.py
#########################################
#!/usr/bin/python3
import sys
from datetime import datetime
from pathlib import Path
log_file = sys.argv[1]
times = []
try:
        with open(log_file, 'r') as log:
                for line in log:
                        line = line.strip().split()
                        time = datetime.strptime(line[0], '%X')
                        times.append(time)
except:
        quit()
# "OUT_DIR_NAME      FILE_NAME       STATUS  PROCESSING_TIME NUM_COMPONENTS"
processing_time = times[-1]-times[0]
out_dir_name = Path(log_file).parent
file_name = log_file.replace('timed_log.tsv', '')
traces_file = Path(file_name).joinpath('3_event_detection_export', Path(file_name).name[1:] + '-
cellset-traces.csv')
status=0
num_components=0
if not Path(file_name).exists():
    status = -1
elif Path(file_name).joinpath('3_event_detection_export').exists():
    status = 1
    num_components = 0
elif traces_file.exists():
    status = 1
    with open(traces_file) as tf:
        header_ls = tf.readline().strip().split(',')
        num_components = len(header_ls)-1

print(f'{out_dir_name}\t{file_name}\t{status}\t{processing_time}\t{num_components}')
```

Pipeline.py

The following chapters describe the different parts of the pipeline and how they work, they are here to assist with troubleshooting and or developing the pipeline further but may not be necessary for normal operation.

This script is written to take a single directory of inputs and execute steps in the pipeline consecutively on outputs until final outputs are produced. The pipeline is a main function called

```
run_pipeline()
```

that takes a specific directory containing raw data as an argument with the flag -i, or optionally looks through a folder specified in config.ini and prompts the user to select an input folder. The main function determines the entire workflow for a single file.

The pipeline itself is an intermediate layer, another script can import its main function and iterate over a set of input directories. Below it is a directory of modules containing native python scripts as functions that operate on a single set (derived from one directory of input data) of files.

Scripts and programs in other languages need to be executed with the subprocess module. A helper function in this script called

```
run_script()
```

attempts to check the commands that are passed to subprocess and capture the output and can be added into the main function similar to how ImageJ is implemented.

```
shell_command = "conda activate my_env && python my_script.py arg1 arg2 arg3"
run_script(shell_command)
```

The pipeline is designed to be run from its own Python 3.9 environment (requirements specified under setup instructions).

Inside the main function, there is a variable called "root_out" which will always expand to output_dir/name_of_experiment_dir/

If intermediary or additional results are created, the structure under root_out is a numbered set of directories, this helps see in which order everything happens and aids in troubleshooting, so it is recommended that future directories be added as root_out/n+1_future_step.

Pipeline modules

This is a python package folder which sits next to pipeline.py. Any .py script that is put here can then be imported into the main script using

```
from pipeline_modules import script
```

When imported like this, the entire script will run once, except the part that is below:

```
if __name__ == "__main__":
    main() # code block for testing a module script directly
```

This means anything defined in that script will be accessible from the main script which greatly increases the readability of the main script and troubleshooting, since a broken module script can be worked on separately from the rest of the application. Because of this , it is a good idea to give anything defined inside the script a docstring which declares how the input should look and what the expected output is. Additionally it is a good idea to add tests below the if name = main so functions can be evaluated without running the full application. If python scripts follow these general guidelines they can just be dropped in the pipeline_modules folder, imported in pipeline.py and run on an input in its main function.

pre_isx.py

Contains 4 functions, 3 of which summarize the main functionality of the previous Jupyter notebooks that call Inscopix functions to preprocess fluorescence microscopy image data. The fourth function strings these functions together so they can be run sequentially from the main script with one line.

```
def run(out_root):
    print('Writing to this directory: ', out_root)
    preprocessing(out_root=out_root)
    cnmfe_processing(out_root=out_root)
    event_detection(out_root=out_root)
```

The input comes from the output of ImageJ which downsamples and rescales the raw data and outputs large Tif files. At the end of these steps the Tif files will have gone through image analysis to extract components and their associated information (traces and so on) as csv files.

search.py

This is a housekeeping module which supplies functions for searching through directories via the CLI and either running the pipeline on a single directory or in batch mode. Wildcard search

anomaly_detection.py

This module loads a pre-generated random forest model that is stored within the main directory (pipeline/models), it then reads CSV files with traces into Numpy arrays, normalizes the traces (also using a model from pipeline/models) and predicts whether a trace should be accepted or rejected. All traces will be plotted and Accepted traces are collected in a new clean_traces.csv.

The models are generated with the random_forest.py script. This script is built using Scikit-learn. Models are stored as file-streams using pickle. Since this is an unbalanced dataset, the data is split using stratified sampling into training, testing and validation.

qc_log.py

Another housekeeping module, containing functions for logging and quality checks. Also contains a function for removing temporary files that stay open in the subprocess.

ImageJ

A full version of ImageJ Fiji (1.53c portable install) is packaged with the pipeline application. It is called from the main script and fed a custom script so the user doesn't have to interact with the software. The script template is a form of pseudo-java, ImageJ:s own scripting language. This script is in the same directory as the python modules and could be edited as long as the input and output definitions remain unaltered, because the python script just looks for the lines that begin with dir1, dir2 and my_name. Since the introduction of multiprocessing, the intermediary script that is sent as a batch file to ImageJ is named with the same name as the in-going file and deleted after ImageJ completes. This is to avoid concurrent read/write to the same file by parallel processes.

Standalone scripts

Multi-electrode analysis

This script is delivered separately from the pipeline.

Installation

Use either conda or pip to set up an environment with numpy, pandas, matplotlib, seaborn and jupyter or use one that has these packages installed.

The scripts were written under python 3.9, but should work for lower versions of python, though this has not been tested. I suggest using 3.8+ as your python version.

```
conda create -n mea python=3.9 numpy pandas matplotlib seaborn jupyter
conda activate mea
```

Program structure

There is one script file, one jupyter notebook file, a plot directory, and a data directory. The user should start the jupyter server from within the environment:

```
jupyter notebook
```

When mea_plotter is selected, just run all cells. The python script will parse all .csv files in data and generate plots that show the value specified in y_var, which sits in the jupyter notebook, under settings.

Above settings the column names are printed, y_var can be changed to any column name that is an int or float data type, changing what is shown in the plots.

Main function

The program will parse all files in the data directory. If the python script is called directly, plots will be generated in the plots directory. Otherwise the jupyter notebook is currently the recommended interface.

Jupyter will run the main functions of the parser and a file called export_v1.csv will be generated, the same information is imported into the jupyter environment as df, which is a pandas dataframe. This means all the nasty parsing is kept off stage and the user can focus on exploring the data with jupyter.

General notes on writing modules

The general structure of a module that can be imported is a script put in the pipeline_modules folder that has a main function:

```python
# example_script.py
from pathlib import Path
def main(my_path_as_string):
    return Path(my_path_as_string)
```

Then in the pipeline.py script:

```python
from pipeline_modules import example_script
def run_pipeline():
    my_wdir = os.getcwd()
    my_wdir_as_path_obj = example_script.main(my_wdir)
```

Input/Output

There are two general cases for input and output of a module. In the first case the module produces output inside the input directory.

Here is an example of a module that generates output in the input folder (these can be pasted to files and run to get a general sense of the naming conventions, they only produce text output):

```
from pathlib import Path
def main(root_dir):
    root_dir = Path(root_dir)
    # root_dir = results/{experiment_dir}
    # experiment_dir is used in naming all output files
    # and is taken directly from the name of the input dir in step 1
    base_file_name = root_dir.name
    # extract {experiment_dir} as a string
    input_file_1 = root_dir.joinpath("step_1", base_file_name + "_1.isxd")
    output_file_1 = input_file_1.parent.joinpath(base_file_name + "_2.isxd")
    print(f"Doing something to {input_file_1}\n and outputting {output_file_1}")

main(Path.cwd())
```

To avoid clutter it can be nice to instead move output into a directory that is relative to other output directories from the same source:

(It is probably best to construct output paths inside the main script and write new scripts just take these as arguments)

```
from pathlib import Path
def main(root_dir):
    root_dir = Path(root_dir)
    # root_dir = results/{experiment_dir}
    # experiment_dir is used in naming all output files
    # and is taken directly from the name of the input dir in step 1
    base_file_name = root_dir.name
    # extract {experiment_dir} as a string
    input_file_1 = root_dir.joinpath("step_1",base_file_name + "_1.isxd")
    output_file_1 = root_dir.joinpath("step_2", base_file_name + "_2.isxd")
    #output_file_1.mkdir(exist_ok=True)
    print(f"Doing something to {input_file_1}\n and outputting {output_file_1}")

main(Path.cwd())
```

If only a few files are output from a module, create the output dir inside pipeline.py and take the full output path inside the main function.
Reasons: 1. pipeline.py should be readable and it should be understandable where output will end up, 2. root_out is already a path object inside pipeline.py, save yourself some work

Finally, if you wrote a module that is imported, always add tests e.g.:

```
def main(root_out):
    pass #do something
if __name__ == "__main__":
    print(main("test"))
```

Environment

The pipeline has its own environment. Changing the requirements of the pipeline environment is not recommended because changed requirements affects the Setup instructions. If this is done, also update Setup instructions in this document!
The pipeline integrates well with Conda. Refer to https://docs.conda.io/en/latest/ for how to setup and export environments for your scripts.

Adding external scripts

Python packages and modules can be put inside pipeline_modules or next to it and straightforwardly imported in the pipeline.py script, then their main function can be added to run_pipeline().

R-scripts should be inside the pipeline main directory, and can be added with the run_script() function (make sure the script is designed to be run from CLI and takes input and output as CLI arguments)

```
[SCRIPTS]
new_script = bin/my_fancy.rscript
```

Inside pipeline.py the SCRIPTS section is accessible as a dictionary after doing script_dict = config['SCRIPTS']. In this case we would specify the new script path as new_script_path = script_dict['new_script']. The external script should take parameters somehow, which is easiest to find by running the script in a CLI and looking at its command line options. Note how the script should be executed and prepare the proper variables containing inputs and outputs for the script, then do run_script(f"{new_script_path} arg1 arg2 argn") where args are flags or parameters that are sent to the script. The run_script() function then attempts to make sure that special characters are escaped properly and sends the command to the script directly.

If the program or script lives anywhere else on the system, specifying a full path to its executable works, but it will not follow the pipeline to a new system, so the conf.ini needs to be updated when installing. Otherwise the procedure is the same.

run_script() will return the text output, consisting of standard out and standard error, you can store each in variables like

```
out, err = run_script()
# print output
print(out, '\n', err)
```

Changes to Inscopix API relating to Docker implementation

```
#Original setup instructions for Docker
1. run a docker container with latest ubuntu
   `docker run -ti --name caiman ubuntu:latest /bin/bash`
2. Install conda, caiman and isx conda wrapper
   ```
   apt update && apt install -y time libgl1 htop psmisc wget
   wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
```
```

~~~
   wget https://raw.githubusercontent.com/inscopix/isx-cnmfe-
wrapper/master/isx_cnmfe_wrapper/runner.py
   bash Miniconda3-latest-Linux-x86_64.sh -b -p $HOME/miniconda
   $HOME/miniconda/bin/conda init bash
   source $HOME/.bashrc
   conda install caiman=1.8.5 python=3.7.8 numpy=1.19.4 pandas=1.1.4 scipy=1.5.3 libgcc-ng=9.3.0
libgfortran-ng=9.3.0 libgfortran5=9.3.0 hdf5=1.10.5 freetype=2.10.4 dbus=1.13.6 pyqt=5.9.2
qtpy=1.9.0 requests=2.25.0 matplotlib=3.3.3 libopenblas=0.3.12 libblas=3.9.0 libcblas=3.9.0
libxml2=2.9.9 zipp=3.4.0 bokeh=2.2.3 -c conda-forge -y
   exit
   ```
3. Save the container to an image and tag it
   (also set ipcluster to be started as the container's main process)
   `docker commit --change "ENTRYPOINT bash -c '/root/miniconda/bin/ipcluster start -n 4'"
caiman caiman`
4. Define the caiman container
   `docker run -v e:/:/mnt/e -v f:/:/mnt/f -v g:/:/mnt/g -v h:/:/mnt/h -v i:/:/mnt/i -d --name
caiman caiman`
5. Test by hand
    * `docker exec -ti caiman bash` to get a container shell
    * smallest file
        * with data in host
          (wall time 10:17.17, memory 4.2 GB)
          `/usr/bin/time -v python /root/runner.py --input_files
/mnt/g/IDPS_dir_test_JBA/CNMFe_test/cnmfe-SD2TD2/20201027_plate20_org-A3_area-E4_0_trimSD-
SD2TD2_001-BP.tiff --params_file /mnt/g/IDPS_dir_test_JBA/CNMFe_test/cnmfe-
SD2TD2/caiman_params.yaml --output_file  /mnt/g/IDPS_dir_test_JBA/CNMFe_test/cnmfe-
SD2TD2/output_docker_host.hd5`
        * with data copied into the container
          (wall time 9:23.07+0:03.00 for copying the input file, memory 4.2 GB)
          ```
          /usr/bin/time cp /mnt/g/IDPS_dir_test_JBA/CNMFe_test/cnmfe-
SD2TD2/20201027_plate20_org-A3_area-E4_0_trimSD-SD2TD2_001-BP.tiff /root/input.tiff
          /usr/bin/time -v python /root/runner.py --input_files /root/input.tiff --params_file
/mnt/g/IDPS_dir_test_JBA/CNMFe_test/cnmfe-SD2TD2/caiman_params.yaml --output_file
/root/output.hd5
          cp /root/output.hd5 /mnt/g/IDPS_dir_test_JBA/CNMFe_test/cnmfe-
SD2TD2/output_docker_guest.hd5
          ```
    * full file
        * with data in host
          (wall time 57:21.95, memory 26.8 GB)
          ```
          /usr/bin/time -v python /root/runner.py --input_files
/mnt/g/IDPS_dir_test_JBA/CNMFe_test/cnmfe-bp/20201027_plate20_org-A3_area-E4_0_trimSD_bp.tiff --
params_file /mnt/g/IDPS_dir_test_JBA/CNMFe_test/cnmfe-bp/caiman_params.yaml --output_file
/mnt/g/IDPS_dir_test_JBA/CNMFe_test/cnmfe-bp/output_docker_host.hd5
          ```
        * with data copied into the container
          (wall time 55:23.28+0:31:00 for copying the input file, memory 26.8 GB)
          ```
          /usr/bin/time cp /mnt/g/IDPS_dir_test_JBA/CNMFe_test/cnmfe-bp/20201027_plate20_org-
A3_area-E4_0_trimSD_bp.tiff /root/input.tiff
          /usr/bin/time -v python /root/runner.py --input_files /root/input.tiff --params_file
/mnt/g/IDPS_dir_test_JBA/CNMFe_test/cnmfe-bp/caiman_params.yaml --output_file /root/output.hd5
          cp /root/output.hd5 /mnt/g/IDPS_dir_test_JBA/CNMFe_test/cnmfe-
bp/output_docker_guest.hd5
          ```
6. Install docker python package in the host environment
    * Run Powershell Prompt from Anaconda Navigator
    * Run `conda activate isxenv` followed by `conda install docker-py`
~~~

```
7. Running from python isxenv
    * Open Powershell Prompt from Anaconda Navigator
    * Run `conda activate isxenv`
    * Run `python`
    * Run
      ```
import os
import isx.cnmfe
os.chdir('G:/IDPS_dir_test_JBA/CNMFe_test')
isx.cnmfe.run_cnmfe('20201027_plate20_org-A3_area-E4_0_trimSD-SD2TD2_001-BP.isxd', 'cnmfe-SD2TD2/outCell_docker.isxd', 'cnmfe-SD2TD2/outEvent_docker.isxd', output_dir='cnmfe-SD2TD2', overwrite_tiff=True, inDocker=True)
# wall time 0:09:38 (0:09:32 on CaImAn, 0:00:06 on pre/postprocessing)
isx.cnmfe.run_cnmfe('20201027_plate20_org-A3_area-E4_0_trimSD-SD2TD2_001-BP.isxd', 'cnmfe-SD2TD2/outCell.isxd',   'cnmfe-SD2TD2/outEvent.isxd',   output_dir='cnmfe-SD2TD2', overwrite_tiff=True, num_processes=4)
# wall time 0:16:11 (0:16:05 on CaImAn, 0:00:06 on pre/postprocessing)
isx.cnmfe.run_cnmfe('20201027_plate20_org-A3_area-E4_0_trimSD_bp.isxd', 'cnmfe-bp/outCell_docker.isxd', 'cnmfe-bp/outEvent_docker.isxd', output_dir='cnmfe-bp', overwrite_tiff=True, inDocker=True)
# wall time 0:58:34 (0:58:05 on CaImAn, 0:00:29 on pre/postprocessing)
isx.cnmfe.run_cnmfe('20201027_plate20_org-A3_area-E4_0_trimSD_bp.isxd', 'cnmfe-bp/outCell_4c.isxd',   'cnmfe-bp/outEvent_4c.isxd',   output_dir='cnmfe-bp', overwrite_tiff=True, num_processes=4)
# wall time 9:40:25 (9:39:57 on CaImAn, 0:00:28 on pre/postprocessing)
isx.cnmfe.run_cnmfe('20201027_plate20_org-A3_area-E4_0_trimSD_bp.isxd', 'cnmfe-bp/outCell_4c.isxd',   'cnmfe-bp/outEvent_4c.isxd',   output_dir='cnmfe-bp', overwrite_tiff=True, num_processes=2)
# wall time (on CaImAn, on pre/postprocessing)
      ```
```

Notes on directories and files that were altered, along with the copies of these files are inside the pipeline directory, under a directory named "isx_backup_scripts".

INSCOPIX 1.6 UPDATE:

See new branch for the changes that need to be made to upgrade to the new inscopix API version.

```
NOTES:
20210720

* Moved the latest version of the pipeline to E: so it can sit next to the ISX scripts.

* Setting up the conda environment for the pipeline and for anomaly_detection

* Created a TestResults folder

* Updated conf.ini

  - num_processes=1
  - added test folders

* running pipeline with -j 1
```

```
 * release notes do not mention any particular changes to the function kwargs

 * importing cnmfe yields error: ImportError: cannot import name 'incremental_pca'

   - following docs chapter 5.1.4 attempting to use pip to install caiman
     pip install "git+https://github.com/flatironinstitute/CaImAn.
     git@7dc5b42ab06c6a6b86ff1520dfc5b2334f335a78"
     pip install “git+https://github.com/inscopix/isx-cnmfe-wrapper@v1.2”

   # succesfully installed!

 * pre_isx.py ln 144 going through the isx.run_cnmfe() function and checking what keywords have
changed

   - commenting keywords out for now
   - commenting out the import isx.cnmfe

 * ISX.log: CNMFE analysis failed with error  run_cnmfe() got an unexpected keyword argument
'output_events_files'

   - commenting this out as well

 * isx.event_detection() now outputs the event files, updating line 190 in pre_isx.py

   - something is not working here, log reveals that CNMFE is working, but not event detection
   - 20210726 after updating keywords and setting them to what Bruno, Joanna and Xiaoliang
suggested, event detection now seems to be working

 * adding profile to the isx.run_cnmfe function to report runtimes

 * bg_spatial_subsampling=2

 * stopped processing after 1+ hour

 * used pip to remove the git clone of caiman

 * installed caiman package from conda-forge
   -CNMFe: Total runtime = 1.502e+06 ms = ~25 minutes
   20210727

 * after updating to isx.event_detection(), this needs to happen after looking for the final
output!

 * the traces file now contains negative values and anomaly detection is definitely not working

 * ln 217 adding isx.auto_accept_reject()

 * When setting params to suggested - error:
   - There are too many patches. Try increasing the patch size, decreasing the patch overlap, or
spatially downsampling the data to reduce the number of patches.
   - wrote loop to try +1 patch size until it works, stopped at 43 MINIMUM VALUE

   # Conclusion
   The output traces are now normalized using dF/F, which means anomaly detection will have to
be adjusted accordingly (perhaps if the data used to train the model can be normalized in the
same way)
   Using CNMFe without the custom Docker image works. It is likely using the conda-forge version
as recommended in Caiman documentation would work for older versions as well.
```