FISSA Documentation

Release 1.0.dev0

Sander Keemink and Scott Lowe

Mar 26, 2022

CONTENTS:

FISSA (Fast Image Signal Separation Analysis) is a Python package for decontaminating somatic signals from two-photon calcium imaging data. It can read images in tiff format and ROIs from zip files exported by ImageJ; or operate on numpy arrays, generated by importing files stored in other or as the output of other packages.

For details of the algorithm, please see our companion paper published in Scientific Reports.

CHAPTER

ONE

FISSA USER GUIDE

FISSA (Fast Image Signal Separation Analysis) is a Python package for decontaminating somatic signals from two-photon calcium imaging data. It can read images in tiff format and ROIs from zip files exported by ImageJ; or operate on numpy arrays, generated by importing files stored in other or as the output of other packages.

For details of the algorithm, please see our companion paper published in Scientific Reports. For the code used to generate the simulated data in the companion paper, see the SimCalc repository.

FISSA is compatible with both Python 2.7 and Python >=3.5, however it is strongly encouraged that you use Python 3, since as Python 2 has reached its end of life. FISSA is continually tested on Ubuntu, Windows, and Mac OSX during its development cycle.

Documentation, including the full API, is available online at readthedocs.

If you encounter a specific problem please open a new issue. For general discussion and help with installation or setup, please see the Gitter chat.

1.1 Usage

A concise example of how to use FISSA is as follows.

```
import fissa

result = fissa.run_fissa("path/to/tiffs", "path/to/rois.zip")

# The decontaminated time series is now available as
# result[roi_index, tiff_index][0, :]
```

We also have several example notebooks for a basic workflow and more complicated workflows where FISSA needs to interact with the outputs of other two-photon calcium imaging toolboxes which can be used to automatically detect cells.

You can try out each of the example notebooks interactively in your browser on Binder (note that it may take 10 minutes for Binder to boot up). Note that the Suite2p notebook is housed in its own repository, and runs on a separate Binder instance from the other notebooks.

Workflow	Jupyter Notebook			Script	
Function-based (ImageJ)	Docs	Launch Binder	Download	Linux/Mac	Windows
Object-oriented (ImageJ)	Docs	Launch Binder	Download	Linux/Mac	Windows
With suite2p	Docs	Launch Binder	Download		
With SIMA	Docs	Launch Binder	Download		
With CNMF (MATLAB)	Docs	Launch Binder	Download		

These notebooks can also be run on your own machine. To do so, you will need to:

0. If you want to run the Suite2p notebook, you'll have to install everything into a conda environment, as per their installation instructions.
1. Install fissa with its plotting dependencies `pip install fissa[plotting]`.
2. If you want to run the sima notebook, you will also have to install sima with `pip install sima`. Note that sima only supports python<=3.6.
3. Download a copy of the repository, unzip it and browse to the examples directory.
4. Start up a Jupyter notebook server to run our notebooks `jupyter notebook`.

If you're new to Jupyter notebooks, here is an approachable tutorial.

1.2 Installation

1.2.1 Quick Guide

FISSA is available on PyPI and the latest version can be installed into your current environment using pip.

```
pip install fissa
```

If you need more details or you're stuck with something in the dependency chain, more detailed instructions for both Windows and Ubuntu users are below.

1.2.2 Installation on Windows

We detail two different ways to install Python on your Windows. One is to download the *official Python installer*, and the other is to use *Anaconda*.

Official Python distribution

1. Go to the Python website and download the latest version of Python for Windows.
2. Run the executable file downloaded, which has a name formatted like **python-3.y.z.exe**.
3. In the installation window, tick the checkbox "Add Python 3.y to PATH".
4. Click "Install Now", and go though the installation process to install Python.
5. Open the **Command Prompt** application. We can run Python from the general purpose command prompt because we added its binaries to the global `PATH` variable in Step 3.
6. At the **Command Prompt** command prompt, install fissa and its dependencies by running the command:

   ```
   pip install fissa
   ```

7. You can check to see if fissa is installed with:

   ```
   python -c "import fissa; print(fissa.__version__)"
   ```

   You should see your FISSA version number printed in the terminal.

8. You can now use FISSA from the Python command prompt. To open a python command prompt, either execute the command `python` within the **Command Prompt**, or open Python executable which was installed in Step 4. At the python command prompt, you can run FISSA as described in *Usage* above.

Anaconda distribution

1. Download and install the latest version of either Anaconda or Miniconda. Miniconda is a lightweight version of Anaconda, the same thing but without any packages pre-installed.
2. Open the **Anaconda Prompt**, which was installed by either Anaconda or Miniconda in Step 1.
3. In the Anaconda Prompt, run the following command to install some of FISSA's dependencies with conda.

```
conda install -c conda-forge numpy scipy shapely tifffile
```

4. Run the following command to install FISSA, along with the rest of its dependencies.

```
pip install fissa
```

5. You can check to see if fissa is installed with:

```
python -c "import fissa; print(fissa.__version__)"
```

   You should see your FISSA version number printed in the terminal.

6. You can now use FISSA from the Python command prompt. To open a python command prompt, either execute the command `python` within the **Anaconda Prompt**. At the python command prompt, you can run FISSA as described in *Usage* above.
7. Optionally, if you want use suite2p, it and its dependencies can be installed as follows.

```
conda install -c conda-forge mkl mkl_fft numba pyqt
pip install suite2p rastermap
```

1.2.3 Installation on Linux

Before installing FISSA, you will need to make sure you have all of its dependencies (and the dependencies of its dependencies) installed.

Here we will outline how to do all of these steps, assuming you already have both Python and pip installed. It is highly likely that your Linux distribution ships with these. You can upgrade to a newer version of Python by downloading Python from the official website.

Alternatively, you can use an *Anaconda* environment (not detailed here).

1. Dependencies of dependencies
   - scipy requires a Fortran compiler and BLAS/LAPACK/ATLAS
   - shapely requires GEOS.
   - Pillow>=3.0.0 effectively requires a JPEG library.

   These packages can be installed on Debian/Ubuntu with the following shell commands.

```
sudo apt-get update
sudo apt-get install gfortran libopenblas-dev liblapack-dev libatlas-dev libatlas-
↪base-dev
sudo apt-get install libgeos-dev
sudo apt-get install libjpeg-dev
```

2. Install the latest release version of FISSA from PyPI using pip:

```
pip install fissa
```

3. You can check to see if FISSA is installed by running the command:

```
python -c "import fissa; print(fissa.__version__)"
```

   You will see your FISSA version number printed in the terminal.

4. You can now use FISSA from the Python command prompt, as described in *Usage* above.

1.3 Citing FISSA

If you use FISSA for your research, we would be grateful if you could cite our paper on FISSA in any resulting publications:

> S. W. Keemink, S. C. Lowe, J. M. P. Pakan, E. Dylda, M. C. W. van Rossum, and N. L. Rochefort. FISSA: A neuropil decontamination toolbox for calcium imaging signals, *Scientific Reports*, **8**(1):3493, 2018. doi: 10.1038/s41598-018-21640-2.

For your convenience, we provide a copy of this citation in bibtex and RIS format.

You can browse papers which utilise FISSA here.

1.4 License

CHAPTER

TWO

USAGE EXAMPLES

The following examples demonstrate how to use FISSA by applying it to a toy dataset.

The examples are each generated with a Jupyter notebooks, which you can run locally by downloading them from the corresponding directory in our repository. Alternatively, they can be run in the cloud using Binder.

The Suite2p notebook, which has more complicated dependencies, is not located with the other notebooks and can be found in its own example repository.

2.1 Basic FISSA usage

This notebook contains a step-by-step example of how to use the function-based high-level interface to the FISSA toolbox, fissa.run_fissa.

For more details about the methodology behind FISSA, please see our paper:

Keemink, S. W., Lowe, S. C., Pakan, J. M. P., Dylda, E., van Rossum, M. C. W., and Rochefort, N. L. FISSA: A neuropil decontamination toolbox for calcium imaging signals, *Scientific Reports*, **8**(1):3493, 2018. doi: 10.1038/s41598-018-21640-2.

See basic_usage_func.py (or basic_usage_func_windows.py for Windows users) for a short example script outside of a notebook interface.

2.1.1 Import packages

First, we need to import fissa.

[1]:
```
# Import the FISSA toolbox
import fissa
```

We also need to import some plotting dependencies which we'll make use in this notebook to display the results.

[2]:
```
# For plotting our results, import numpy and matplotlib
import matplotlib.pyplot as plt
import numpy as np
```

[3]:
```
# Fetch the colormap object for Cynthia Brewer's Paired color scheme
colors = plt.get_cmap("Paired")
```

2.1.2 Running FISSA

With the function-based interface to FISSA, everything is handled in a single function call to fissa.run_fissa. The function takes as its input is the raw signals, and returns the decontaminated signals.

The mandatory inputs to fissa.run_fissa are:

- the experiment images
- the regions of interest (ROIs) to extract

Images can be given as a path to a folder containing tiff stacks:

```
images = "folder"
```

Each of these tiff-stacks in the folder (e.g. `"folder/trial_001.tif"`) is a trial with many frames. Although we refer to one trial as an `image`, it is actually a video recording.

Alternatively, the image data can be given as a list of paths to tiffs:

```
images = ["folder/trial_001.tif", "folder/trial_002.tif", "folder/trial_003.tif"]
```

or as a list of arrays which you have already loaded into memory:

```
images = [array1, array2, array3, ...]
```

For the regions of interest (ROIs) input, you can either provide a single set of ROIs, or a set of ROIs for every image.

If the ROIs were defined using ImageJ, use ImageJ's export function to save them in a zip. Then, provide the ROI filename.

```
rois = "rois.zip"  # for a single set of ROIs used across all images
```

The same set of ROIs will be used for every image in `images`.

Sometimes there is motion between trials causing the alignment of the ROIs to drift. In such a situation, you may need to use a slightly different location of the ROIs for each trial. This can be handled by providing FISSA with a list of ROI sets — one ROI set (i.e. one ImageJ zip file) per trial.

```
rois = ["rois1.zip", "rois2.zip", ...]  # for a unique roiset for each image
```

Please note that the ROIs defined in each ROI set must correspond to the same physical reigons across all trials, and that the order must be consistent. That is to say, the 1st ROI listed in each ROI set must correspond to the same item appearing in each trial, etc.

In this notebook, we will demonstrate how to use FISSA with ImageJ ROI sets, saved as zip files. However, you are not restricted to providing your ROIs to FISSA in this format. FISSA will also accept ROIs which are arbitrarily defined by providing them as arrays (`numpy.ndarray` objects). ROIs provided in this way can be defined either as boolean-valued masks indicating the presence of a ROI per-pixel in the image, or defined as a list of coordinates defining the boundary of the ROI. For examples of such usage, see our Suite2p, CNMF, and SIMA example notebooks.

As an example, we will run FISSA on a small test dataset.

The test dataset can be found and downloaded from the examples folder of the fissa repository, along with the source for this example notebook.

[4]:
```
# Define path to imagery and to the ROI set
images_location = "exampleData/20150529"
rois_location = "exampleData/20150429.zip"
```

(continues on next page)

(continued from previous page)

```
# Call FISSA using the function-based interface
result, raw = fissa.run_fissa(images_location, rois_location)
```

```
Extracting traces:   0%|          | 0/3 [00:00<?, ?it/s]
```

```
Finished extracting raw signals from 4 ROIs across 3 trials in 0.879 seconds.
```

```
Separating data:   0%|          | 0/4 [00:00<?, ?it/s]
```

```
Finished separating signals from 4 ROIs across 3 trials in 1.705 seconds
```

The function-based interface is very straight forward, but note that you can only access the result which is returned by the function.

If you need to access the raw traces, ROI masks, or demixing matrix, you need to use the more flexible object-oriented (class based) interface using fissa.Experiment instead. An example of this is given in our object-oriented example usage notebook.

2.1.3 Working with results

The output of `fissa.run_fissa` is structured as a 2-d array of 2-d arrays (it can't be a 4-d array because of trials generally don't have the same number of frames).

The results from the cell (ROI) numbered `c` and the trial (TIFF) numbered `t` are located at `result[c, t][0, :]`.

The fourth and final dimension works through frames within the TIFF file (time).

The third dimension iterates over output signals. The 0-th entry of this is the signal which most closely corresponds to the raw signal within the ROI, and is FISSA's best guess for the decontaminated cell source signal. The other signals are the isolated signals from contaminants such as neuropil and neighbouring cells.

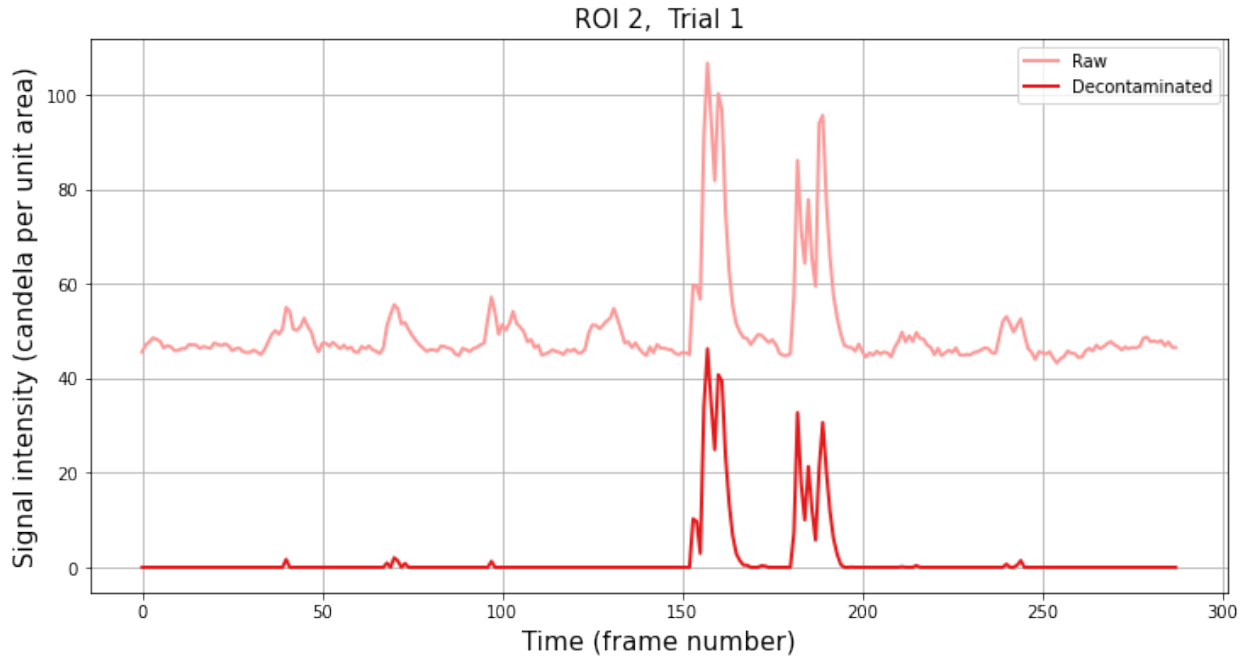
Let's compare the raw signal to the separated signal for a single trial from an example ROI.

[5]:
```
# Plot sample trace

# Select the ROI and trial to plot
roi = 2
trial = 1

# Create the figure
plt.figure(figsize=(12, 6))

plt.plot(
    raw[roi, trial][0, :],
    lw=2,
    label="Raw",
    color=colors((roi * 2) % colors.N),
)
plt.plot(
    result[roi, trial][0, :],
    lw=2,
    label="Decontaminated",
    color=colors((roi * 2 + 1) % colors.N),
)
```

(continues on next page)

(continued from previous page)

```
plt.title("ROI {},  Trial {}".format(roi, trial), fontsize=15)
plt.xlabel("Time (frame number)", fontsize=15)
plt.ylabel("Signal intensity (candela per unit area)", fontsize=15)
plt.grid()
plt.legend()
plt.show()
```



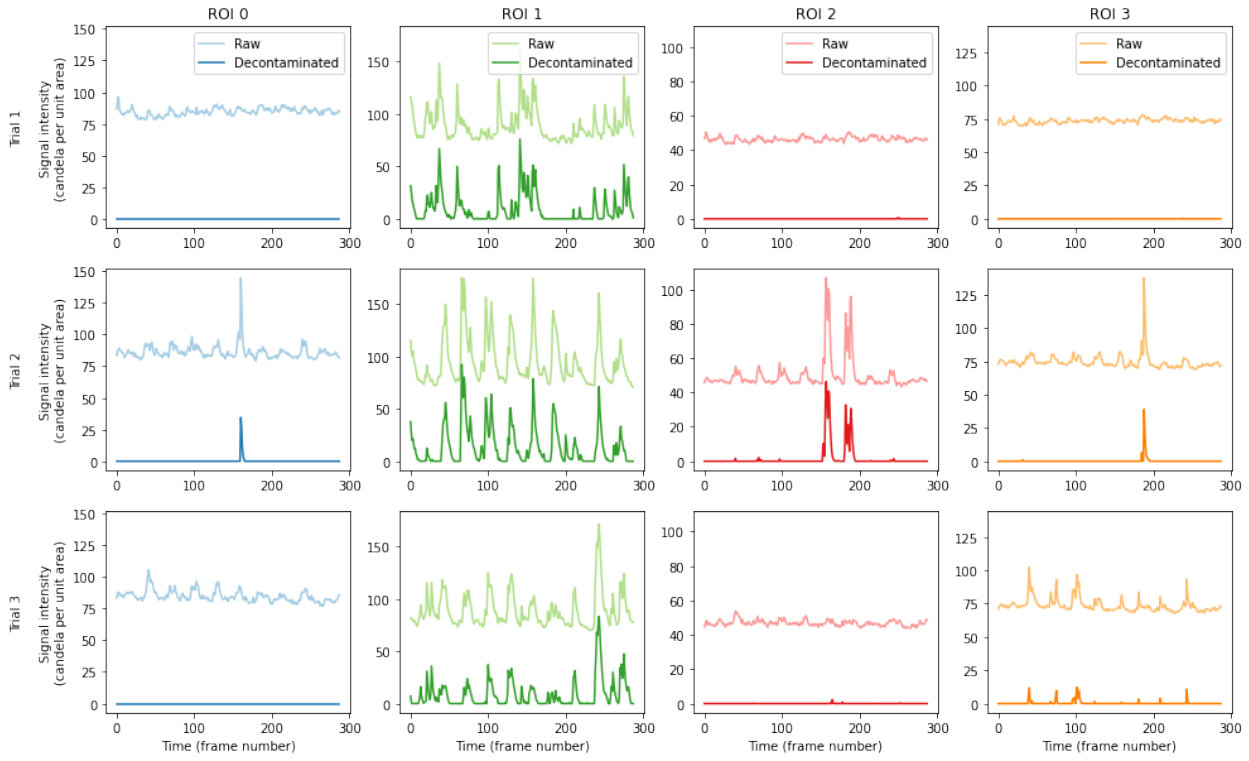
Let's next plot the traces across all ROIs and trials.

[6]:
```
# Plot all ROIs and trials

# Get the number of ROIs and trials
n_roi = result.shape[0]
n_trial = result.shape[1]

# Find the maximum signal intensities for each ROI
roi_max_raw = [
    np.max([np.max(raw[i_roi, i_trial][0]) for i_trial in range(n_trial)])
    for i_roi in range(n_roi)
]
roi_max_result = [
    np.max([np.max(result[i_roi, i_trial][0]) for i_trial in range(n_trial)])
    for i_roi in range(n_roi)
]
roi_max = np.maximum(roi_max_raw, roi_max_result)

# Plot our figure using subplot panels
plt.figure(figsize=(16, 10))
for i_roi in range(n_roi):
    for i_trial in range(n_trial):
        # Make subplot axes
```

(continues on next page)

(continued from previous page)

```
        i_subplot = 1 + i_trial * n_roi + i_roi
        plt.subplot(n_trial, n_roi, i_subplot)
        # Plot the data
        plt.plot(
            raw[i_roi][i_trial][0, :],
            label="Raw",
            color=colors((i_roi * 2) % colors.N),
        )
        plt.plot(
            result[i_roi][i_trial][0, :],
            label="Decontaminated",
            color=colors((i_roi * 2 + 1) % colors.N),
        )
        # Labels and boiler plate
        plt.ylim([-0.05 * roi_max[i_roi], roi_max[i_roi] * 1.05])
        if i_roi == 0:
            plt.ylabel(
                "Trial {}\n\nSignal intensity\n(candela per unit area)".format(
                    i_trial + 1
                )
            )
        if i_trial == 0:
            plt.title("ROI {}".format(i_roi))
            plt.legend()
        if i_trial == n_trial - 1:
            plt.xlabel("Time (frame number)")

plt.show()
```



Comparing ROI signal to neuropil region signals

It can be very instructive to compare the signal in the central ROI with the surrounding neuropil regions. These can be found for cell `c` and trial `t` in `raw[c, t][i, :]`, with `i=0` being the cell, and `i=1,2,3,...` indicating the surrounding regions.

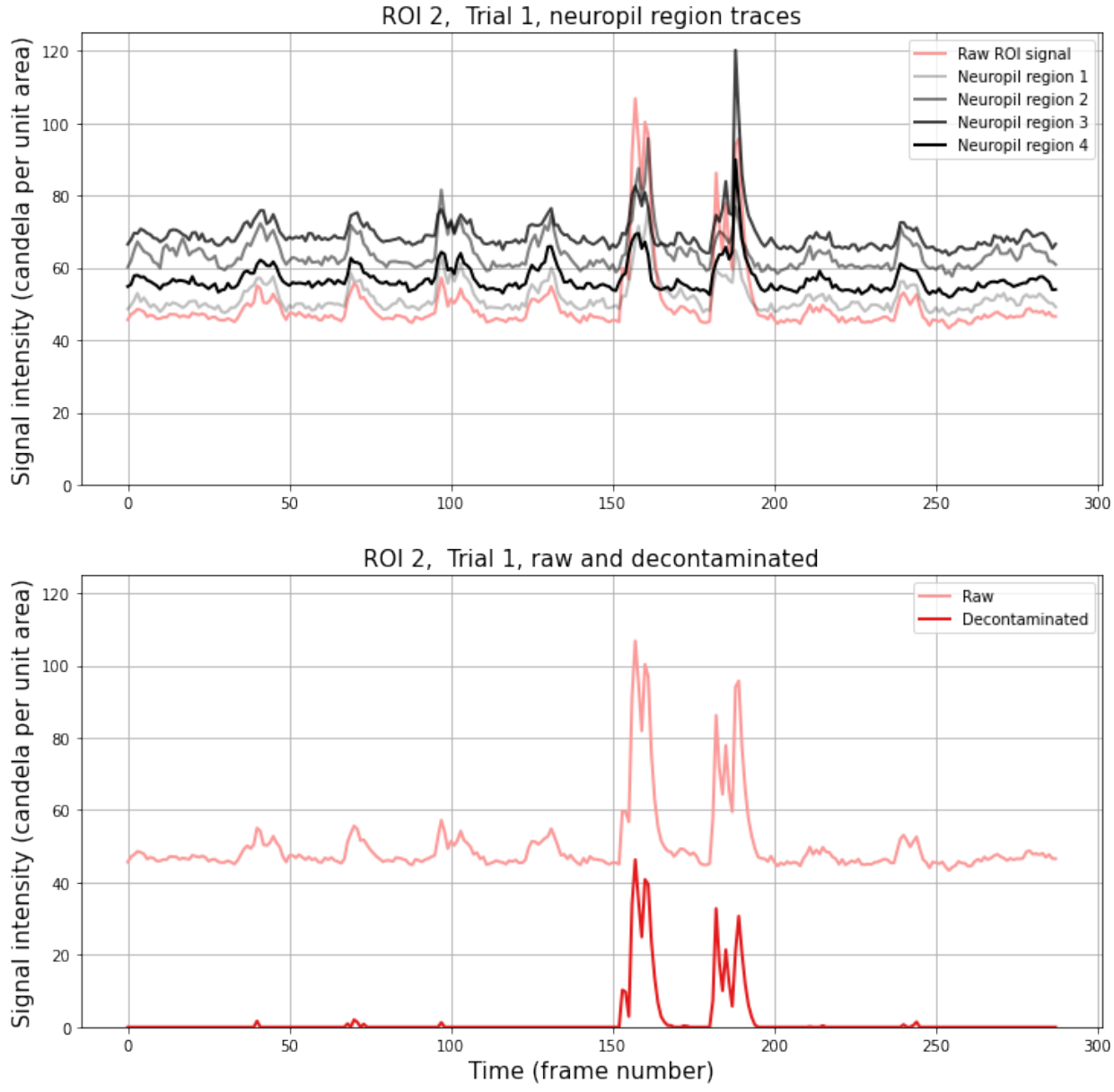
Below we compare directly the raw ROI trace, the decontaminated trace, and the surrounding neuropil region traces.

[7]:
```
# Get the number of neuropil/surrounding regions.
# The raw data has the raw ROI signal in raw[:, :][0] and raw surround
# signals in the rest of the 3rd dimension.
nRegions = raw[0, 0].shape[0] - 1

# Select the ROI and trial to plot
roi = 2
trial = 1

# Create the figure
plt.figure(figsize=(12, 12))

# Plot extracted traces for each neuropil subregion
plt.subplot(2, 1, 1)
# Plot trace of raw ROI signal
plt.plot(
    raw[roi, trial][0, :],
    lw=2,
    label="Raw ROI signal",
    color=colors((roi * 2) % colors.N),
)
# Plot traces from each neuropil region
for i_neuropil in range(1, nRegions + 1):
    alpha = i_neuropil / nRegions
    plt.plot(
        raw[roi, trial][i_neuropil, :],
        lw=2,
        label="Neuropil region {}".format(i_neuropil),
        color="k",
        alpha=alpha,
    )
plt.ylim([0, 125])
plt.grid()
plt.legend()
plt.ylabel("Signal intensity (candela per unit area)", fontsize=15)
plt.title("ROI {},  Trial {}, neuropil region traces".format(roi, trial), fontsize=15)

# Plot the ROI signal
plt.subplot(2, 1, 2)
# Plot trace of raw ROI signal
plt.plot(raw[roi, trial][0, :], lw=2, label="Raw", color=colors((roi * 2) % colors.N))
# Plot decontaminated signal matched to the ROI
plt.plot(
    result[roi, trial][0, :],
    lw=2,
    label="Decontaminated",
```

(continues on next page)

(continued from previous page)

```
    color=colors((roi * 2 + 1) % colors.N),
)

plt.ylim([0, 125])
plt.grid()
plt.legend()
plt.xlabel("Time (frame number)", fontsize=15)
plt.ylabel("Signal intensity (candela per unit area)", fontsize=15)
plt.title("ROI {},  Trial {}, raw and decontaminated".format(roi, trial), fontsize=15)

plt.show()
```



df/f0

The default output from `fissa.run_fissa` is in the same units as the raw input (candelas per unit area).

It is often desirable to calculate the intensity of a signal relative to the baseline value, df/f0, for the traces. `fissa.run_fissa` will do this for you provide the argument `return_deltaf=True`, and the sampling frequency of your TIFF files with `freq=sample_frequency`. The sampling frequency must be provided because the data is smoothed in order to determine the baseline value f0.

When `return_deltaf=True`, `run_fissa` will return the df/f0 output *instead* of the source signal traces scaled at the recording intensity. If you need to access both the standard FISSA output *and* the df/f0 output at the same time, you need to use the more flexible fissa.Experiment FISSA interface instead, as described in this example.

[8]:
```
sample_frequency = 10  # Hz

deltaf_result, deltaf_raw = fissa.run_fissa(
    images_location, rois_location, freq=sample_frequency, return_deltaf=True
)
```

```
Extracting traces:   0%|          | 0/3 [00:00<?, ?it/s]
Finished extracting raw signals from 4 ROIs across 3 trials in 0.846 seconds.
Separating data:   0%|          | 0/4 [00:00<?, ?it/s]
Finished separating signals from 4 ROIs across 3 trials in 1.670 seconds
Calculating Δf/f0:   0%|          | 0/4 [00:00<?, ?it/s]
Finished calculating Δf/f0 for raw and result signals in 0.046 seconds
```

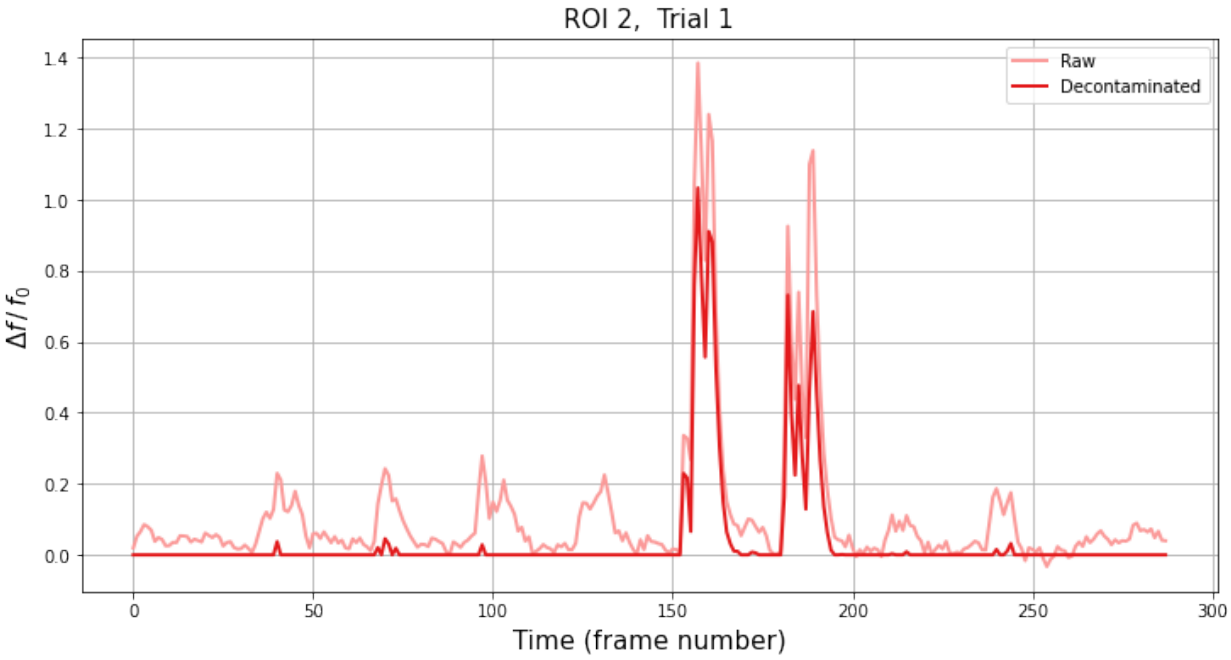
Note that by default, f0 is determined as the minimum across all trials (all TIFFs) to ensure that results are directly comparable between trials, but you can normalise each trial individually instead if you prefer by providing the parameter `deltaf_across_trials=False`.

[9]:
```
# Plot sample trace

# Select the ROI and trial to plot
roi = 2
trial = 1

# Create the figure
plt.figure(figsize=(12, 6))

plt.plot(
    deltaf_raw[roi, trial][0, :],
    lw=2,
    label="Raw",
    color=colors((roi * 2) % colors.N),
)
plt.plot(
    deltaf_result[roi, trial][0, :],
    lw=2,
    label="Decontaminated",
    color=colors((roi * 2 + 1) % colors.N),
)
```

(continues on next page)

(continued from previous page)

```
plt.title("ROI {},  Trial {}".format(roi, trial), fontsize=15)
plt.xlabel("Time (frame number)", fontsize=15)
plt.ylabel(r"$\Delta f\,/\,f_0$", fontsize=15)
plt.grid()
plt.legend()
plt.show()
```



Since FISSA is very good at removing contamination from the ROI signals, the minimum value on the decontaminated trace will typically be `0.`. Consequently, we use the minimum value of the (smoothed) raw signal to provide the f0 from the raw trace for both the raw and decontaminated df/f0.

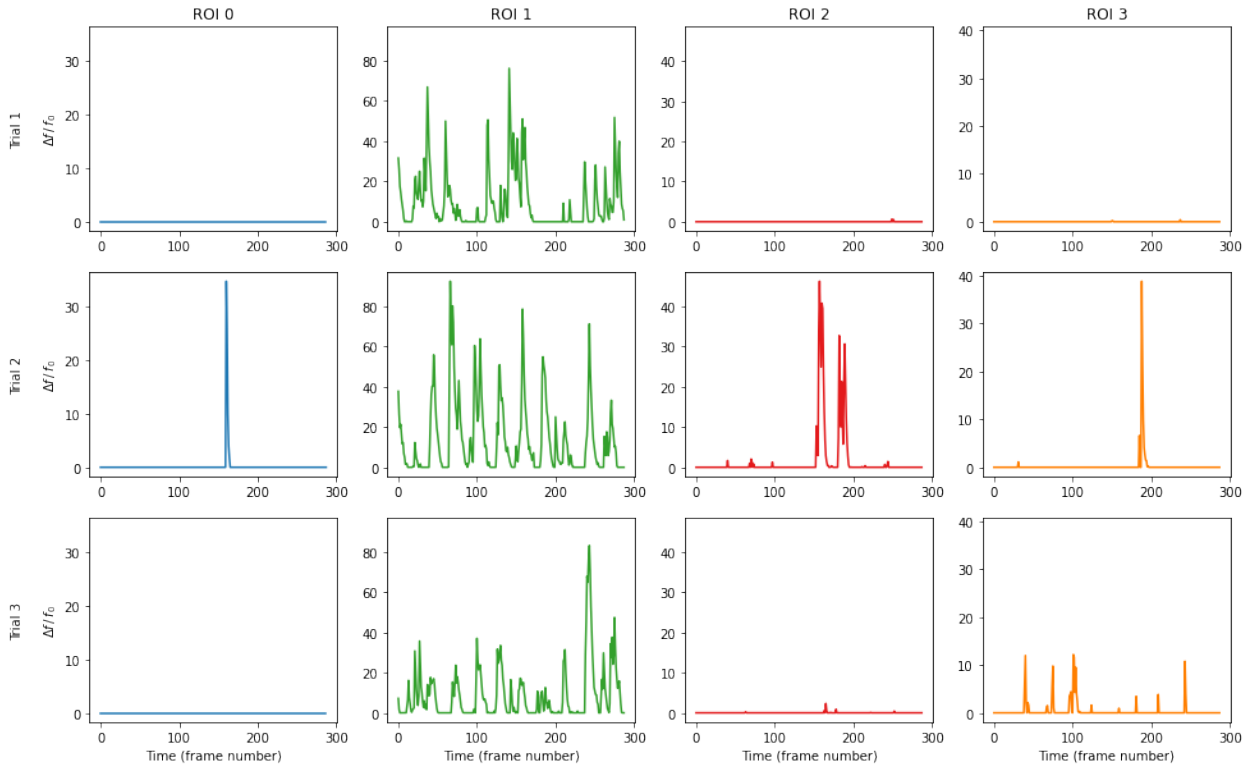
We can plot the df/f0 for every cell during every trial as follows.

[10]:
```
# Get the number of ROIs and trials
n_roi = result.shape[0]
n_trial = result.shape[1]

# Find the maximum signal intensities for each ROI,
# so we can scale ylim the same across subplots
roi_max = [
    np.max([np.max(result[i_roi, i_trial][0]) for i_trial in range(n_trial)])
    for i_roi in range(n_roi)
]

# Plot our figure using subplot panels
plt.figure(figsize=(16, 10))
for i_roi in range(n_roi):
    for i_trial in range(n_trial):
        # Make subplot axes
        i_subplot = 1 + i_trial * n_roi + i_roi
        plt.subplot(n_trial, n_roi, i_subplot)
```

(continues on next page)

(continued from previous page)

```
        # Plot the data
        plt.plot(
            result[i_roi][i_trial][0, :],
            color=colors((i_roi * 2 + 1) % colors.N),
        )
        # Labels and boiler plate
        plt.ylim([-0.05 * roi_max[i_roi], roi_max[i_roi] * 1.05])
        if i_roi == 0:
            plt.ylabel("Trial {}\n\n".format(i_trial + 1) + r"$\Delta f\,/\,f_0$")
        if i_trial == 0:
            plt.title("ROI {}".format(i_roi))
        if i_trial == n_trial - 1:
            plt.xlabel("Time (frame number)")

plt.show()
```



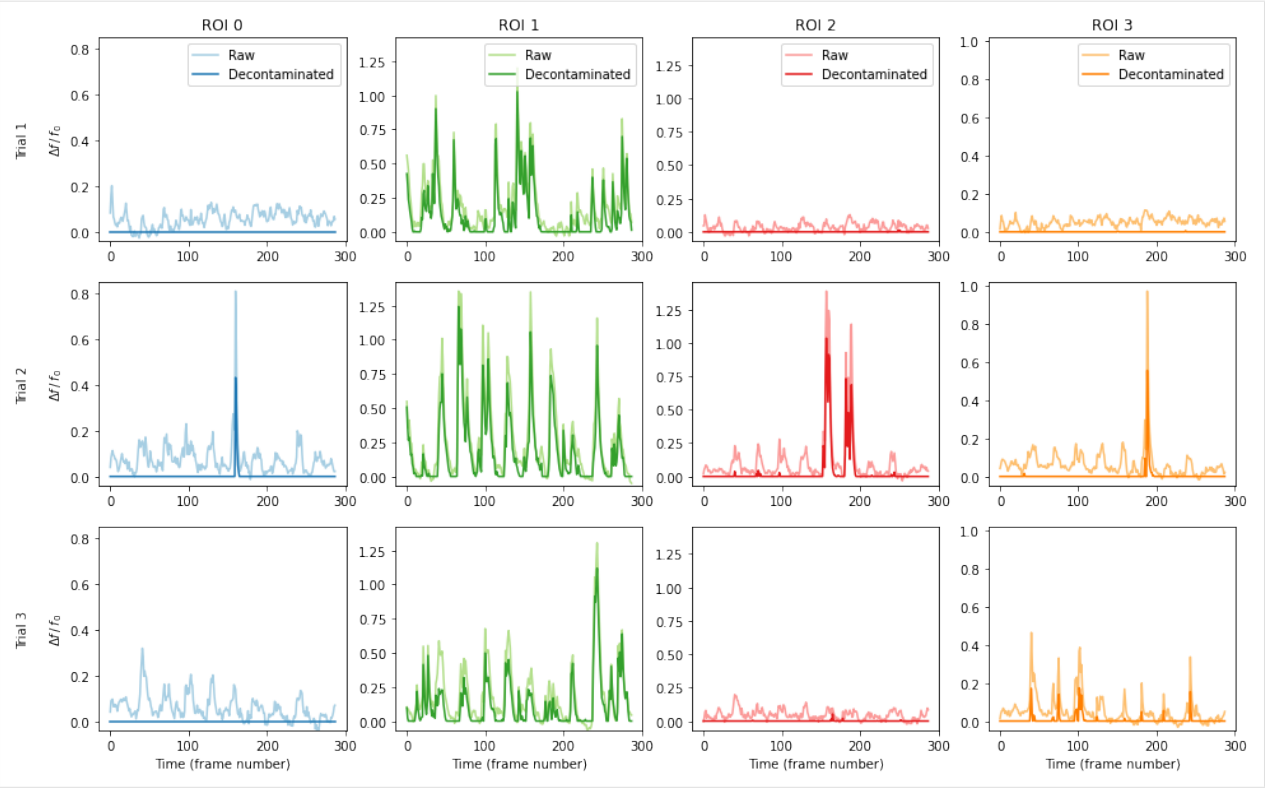
For comparison purposes, we can also plot the df/f0 for the raw data against the decontaminated signal.

[11]:
```
# Plot all ROIs and trials

# Get the number of ROIs and trials
n_roi = deltaf_result.shape[0]
n_trial = deltaf_result.shape[1]

# Find the maximum signal intensities for each ROI
roi_max_raw = [
    np.max([np.max(deltaf_raw[i_roi, i_trial][0]) for i_trial in range(n_trial)])
```

(continues on next page)

(continued from previous page)

```
    for i_roi in range(n_roi)
]
roi_max_result = [
    np.max([np.max(deltaf_result[i_roi, i_trial][0]) for i_trial in range(n_trial)])
    for i_roi in range(n_roi)
]
roi_max = np.maximum(roi_max_raw, roi_max_result)

# Plot our figure using subplot panels
plt.figure(figsize=(16, 10))
for i_roi in range(n_roi):
    for i_trial in range(n_trial):
        # Make subplot axes
        i_subplot = 1 + i_trial * n_roi + i_roi
        plt.subplot(n_trial, n_roi, i_subplot)
        # Plot the data
        plt.plot(
            deltaf_raw[i_roi][i_trial][0, :],
            label="Raw",
            color=colors((i_roi * 2) % colors.N),
        )
        plt.plot(
            deltaf_result[i_roi][i_trial][0, :],
            label="Decontaminated",
            color=colors((i_roi * 2 + 1) % colors.N),
        )
        # Labels and boiler plate
        plt.ylim([-0.05 * roi_max[i_roi], roi_max[i_roi] * 1.05])
        if i_roi == 0:
            plt.ylabel("Trial {}\n\n".format(i_trial + 1) + r"$\Delta f\,/\,f_0$")
        if i_trial == 0:
            plt.title("ROI {}".format(i_roi))
            plt.legend()
        if i_trial == n_trial - 1:
            plt.xlabel("Time (frame number)")

plt.show()
```



2.1.4 Caching

After using FISSA to decontaminate the data collected in an experiment, you will probably want to save the output for later use, so you don't have to keep re-running FISSA on the data.

To facilitate this, an option to cache the outputs is built into FISSA. If you provide `fissa.run_fissa` with an identifier to the experiment being analysed in the `folder` argument, it will cache results into the corresponding directory. Later, if you call `fissa.run_fissa` again with the same `folder` argument, it will load the saved results from that cache folder instead of recomputing them.

[12]:
```
# Define the folder where FISSA's outputs will be cached, so they can be
# quickly reloaded in the future without having to recompute them.
#
# This argument is optional; if it is not provided, FISSA will not save its
# results for later use.
#
# If the output directory already exists, FISSA will load the contents of
# the cache instead of recomputing it.
#
# Note: you *must* use a different folder for each experiment, otherwise
# FISSA will load the existing data instead of computing results for the
# new experiment.
#
# In this example, we will use the current datetime as the  name of the
# experiment, but you can name your experiments however you want to.
# If you want to take advantage of the caching of results, you should use
# a more descriptive name than this so you can identify the actual
```

(continues on next page)

(continued from previous page)

```
# dataset that the FISSA results correspond to, and load them appropriately.

import datetime

output_folder = "fissa-example_{}".format(
    datetime.datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
)

print(output_folder)
```

```
fissa-example_2022-03-25_23-59-33
```

Let's run FISSA on this experiment again, but this time save the results to the experiment's output directory.

[13]:
```
# Run FISSA, saving to results to output_folder
result, raw = fissa.run_fissa(images_location, rois_location, folder=output_folder)
```

```
Extracting traces:   0%|          | 0/3 [00:00<?, ?it/s]
```

```
Finished extracting raw signals from 4 ROIs across 3 trials in 0.828 seconds.
Saving extracted traces to fissa-example_2022-03-25_23-59-33/prepared.npz
```

```
Separating data:   0%|          | 0/4 [00:00<?, ?it/s]
```

```
Finished separating signals from 4 ROIs across 3 trials in 1.708 seconds
Saving results to fissa-example_2022-03-25_23-59-33/separated.npz
```

A subsequent call to `fissa.run_fissa` which uses the same experiment folder argument will load the cached data instead of re-running the FISSA signal separation routine from scratch.

[14]:
```
# Run FISSA, loading results from cache in output_folder
result, raw = fissa.run_fissa(images_location, rois_location, folder=output_folder)
```

```
Loading data from cache fissa-example_2022-03-25_23-59-33/prepared.npz
Loading data from cache fissa-example_2022-03-25_23-59-33/separated.npz
Loading data from cache fissa-example_2022-03-25_23-59-33/separated.npz
```

Exporting to MATLAB

The results can easily be exported to a MATLAB-compatible MAT-file as follows.

If we provide `export_to_matfile=True` to `fissa.run_fissa`, it will export the data a matfile named `"separated.mat"` within the cache directory (the cache directory as provided with the `folder` argument).

[15]:
```
result, raw = fissa.run_fissa(
    images_location, rois_location, folder=output_folder, export_to_matfile=True
)
```

```
Loading data from cache fissa-example_2022-03-25_23-59-33/prepared.npz
Loading data from cache fissa-example_2022-03-25_23-59-33/separated.npz
Loading data from cache fissa-example_2022-03-25_23-59-33/separated.npz
```

Alternatively, we can export to a matfile with a custom file name by setting the `export_to_matfile` argument to the target path.

[16]:
```
result, raw = fissa.run_fissa(
    images_location, rois_location, export_to_matfile="experiment_results.mat"
)
```

```
Extracting traces:   0%|          | 0/3 [00:00<?, ?it/s]
```

```
Finished extracting raw signals from 4 ROIs across 3 trials in 0.828 seconds.
```

```
Separating data:   0%|          | 0/4 [00:00<?, ?it/s]
```

```
Finished separating signals from 4 ROIs across 3 trials in 1.651 seconds
```

Loading the generated file (e.g. `"output_folder/separated.mat"`) in MATLAB will provide you with all of FISSA's outputs.

These are structured in the same way as the `raw` and `result` variables returned by `fissa.run_fissa`. With the python interface, the outputs are 2d numpy.ndarrays each element of which is itself a 2d numpy.ndarrays. Meanwhile, when the output is loaded into MATLAB the data is structured as a 2d cell-array each element of which is a 2d matrix.

Additionally, note that whilst Python indexes from 0, MATLAB indexes from 1 instead. As a consequence of this, the results seen on Python for a given roi and trial `experiment.result[roi, trial]` correspond to the index `S.result{roi + 1, trial + 1}` on MATLAB.

Our first plot in this notebook can be replicated in MATLAB as follows:

```
%% Plot example traces in MATLAB
% Load FISSA output data in MATLAB
% ... either from the automatic file name within a cache
% S = load('fissa-example/separated.mat')
% ... or from a custom output path
S = load('experiment_results.mat')
% Select the second trial
% (On Python, this would be trial = 1)
trial = 2;
% Plot the result traces for each ROI
figure;
hold on;
for i_roi = 1:size(S.result, 1);
    plot(S.result{i_roi, trial}(1, :));
end
xlabel('Time (frame number)');
ylabel('Signal intensity (candela per unit area)');
grid on;
box on;
set(gca,'TickDir','out');
```

2.1.5 Customisation

Controlling verbosity

The level of verbosity of FISSA can be controlled with the `verbosity` parameter.

The default is `verbosity=1`.

If the verbosity parameter is higher, FISSA will print out more information while it is processing. This can be helpful for debugging puproses. The verbosity reaches its maximum at `verbosity=6`.

If `verbosity=0`, FISSA will run silently.

[17]:
```
# Call FISSA with elevated verbosity
result = fissa.run_fissa(images_location, rois_location, verbosity=2)
```

```
Doing region growing and data extraction for 3 trials...
  Images:
    exampleData/20150529/AVG_A01.tif
    exampleData/20150529/AVG_A02.tif
    exampleData/20150529/AVG_A03.tif
  ROI sets:
    exampleData/20150429.zip
    exampleData/20150429.zip
    exampleData/20150429.zip
  nRegions: 4
  expansion: 1
```

```
Extracting traces:   0%|          | 0/3 [00:00<?, ?it/s]
```

```
Finished extracting raw signals from 4 ROIs across 3 trials in 0.829 seconds.
Doing signal separation for 4 ROIs over 3 trials...
  method: 'nmf'
  alpha: 0.1
  max_iter: 20000
  max_tries: 1
  tol: 0.0001
```

```
Separating data:   0%|          | 0/4 [00:00<?, ?it/s]
```

```
Finished separating signals from 4 ROIs across 3 trials in 1.656 seconds
```

Analysis parameters

FISSA has several user-definable settings, which can be set as optional arguments to `fissa.run_fissa`.

[18]:
```
# FISSA uses multiprocessing to speed up its processing.
# By default, it will spawn one worker per CPU core on your machine.
# However, if you have a lot of cores and not much memory, you many not
# be able to suport so many workers simultaneously.
# In particular, this can be problematic during the data preparation step
# in which TIFFs are loaded into memory.
# The default number of cores for the data preparation and separation steps
# can be changed as follows.
ncores_preparation = 4  # If None, uses all available cores
ncores_separation = None  # if None, uses all available cores

# By default, FISSA uses 4 subregions for the neuropil region.
# If you have very dense data with a lot of different signals per unit area,
# you may wish to increase the number of regions.
n_regions = 8

# By default, each surrounding region has the same area as the central ROI.
# i.e. expansion = 1
# However, you may wish to increase or decrease this value.
expansion = 0.75
```

(continues on next page)

(continued from previous page)

```
# The degree of signal sparsity can be controlled with the alpha parameter.
alpha = 0.02

# If you change the experiment parameters, you need to change the cache directory too.
# Otherwise FISSA will try to reload the results from the previous run instead of
# computing the new results. FISSA will throw an error if you try to load data which
# was generated with different analysis parameters to the current parameters.
output_folder2 = output_folder + "_alt"

# Run FISSA with these parameters
result, raw = fissa.run_fissa(
    images_location,
    rois_location,
    output_folder2,
    nRegions=n_regions,
    expansion=expansion,
    alpha=alpha,
    ncores_preparation=ncores_preparation,
    ncores_separation=ncores_separation,
)
```

```
Extracting traces:   0%|          | 0/3 [00:00<?, ?it/s]
```

```
Finished extracting raw signals from 4 ROIs across 3 trials in 1.030 seconds.
Saving extracted traces to fissa-example_2022-03-25_23-59-33_alt/prepared.npz
```

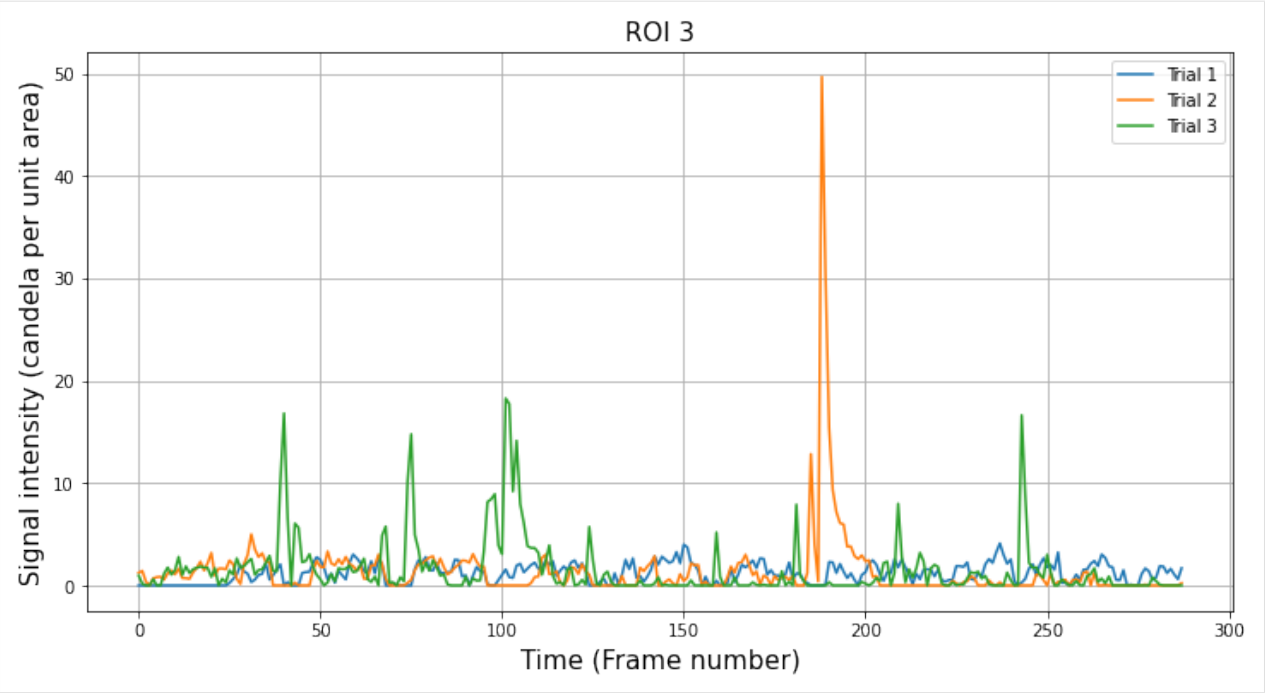
```
Separating data:   0%|          | 0/4 [00:00<?, ?it/s]
```

```
Finished separating signals from 4 ROIs across 3 trials in 4.92 seconds
Saving results to fissa-example_2022-03-25_23-59-33_alt/separated.npz
```

We can plot the new results for our example trace from before. Although we doubled the number of neuropil regions around the cell, very little has changed for this example because there were not many sources of contamination.

However, there will be more of a difference if your data has more neuropil sources per unit area within the image.

[19]:
```
n_roi = result.shape[0]
n_trial = result.shape[1]

i_roi = 3

plt.figure(figsize=(12, 6))

for i_trial in range(n_trial):
    plt.plot(result[i_roi, i_trial][0, :], label="Trial {}".format(i_trial + 1))

plt.title("ROI {}".format(i_roi), fontsize=15)
plt.xlabel("Time (Frame number)", fontsize=15)
plt.ylabel("Signal intensity (candela per unit area)", fontsize=15)
plt.grid()
plt.legend()
plt.show()
```



2.1.6 Working with very large tiff files

By default, FISSA loads entire TIFF files into memory at once and then manipulates all ROIs within the TIFF. This is more efficient, but can be problematic when working with very large TIFF files which are too big to be loaded into memory all at once.

If you run out of memory when running FISSA, you can try reducing the number of workers during the memory-intensive preparation step.

[20]:
```
result = fissa.run_fissa(images_location, rois_location, ncores_preparation=1)
```
```
Extracting traces:   0%|          | 0/3 [00:00<?, ?it/s]
Finished extracting raw signals from 4 ROIs across 3 trials in 0.493 seconds.
Separating data:   0%|          | 0/4 [00:00<?, ?it/s]
Finished separating signals from 4 ROIs across 3 trials in 1.319 seconds
```

Alternatively, you can activate FISSA's low memory mode. In this mode, it will load and process frames one at a time. This will run a fair bit slower than the default mode, but has a much lower memory requirement.

[21]:
```
result, raw = fissa.run_fissa(images_location, rois_location, lowmemory_mode=True)
```
```
Extracting traces:   0%|          | 0/3 [00:00<?, ?it/s]
Finished extracting raw signals from 4 ROIs across 3 trials in 3.03 seconds.
Separating data:   0%|          | 0/4 [00:00<?, ?it/s]
Finished separating signals from 4 ROIs across 3 trials in 3.85 seconds
```

2.2 Object-oriented FISSA interface

This notebook contains a step-by-step example of how to use the object-oriented (class-based) interface to the FISSA toolbox.

The object-oriented interface, which involves creating a fissa.Experiment instance, allows more flexiblity than the fissa.run_fissa function.

For more details about the methodology behind FISSA, please see our paper:

Keemink, S. W., Lowe, S. C., Pakan, J. M. P., Dylda, E., van Rossum, M. C. W., and Rochefort, N. L. FISSA: A neuropil decontamination toolbox for calcium imaging signals, *Scientific Reports*, **8**(1):3493, 2018. doi: 10.1038/s41598-018-21640-2.

See basic_usage.py (or basic_usage_windows.py for Windows users) for a short example script outside of a notebook interface.

2.2.1 Import packages

Before we can begin, we need to import fissa.

[1]:
```
# Import the FISSA toolbox
import fissa
```

We also need to import some plotting dependencies which we'll make use in this notebook to display the results.

[2]:
```
# For plotting our results, import numpy and matplotlib
import matplotlib.pyplot as plt
import numpy as np
```

[3]:
```
# Fetch the colormap object for Cynthia Brewer's Paired color scheme
colors = plt.get_cmap("Paired")
```

2.2.2 Defining an experiment

To run a separation step with fissa, you need create a fissa.Experiment object, which will hold your extraction parameters and results.

The mandatory inputs to `fissa.Experiment` are:

- the experiment images
- the regions of interest (ROIs) to extract

Images can be given as a path to a folder containing tiff stacks:

```
images = "folder"
```

Each of these tiff-stacks in the folder (e.g. `"folder/trial_001.tif"`) is a trial with many frames. Although we refer to one trial as an `image`, it is actually a video recording.

Alternatively, the image data can be given as a list of paths to tiffs:

```
images = ["folder/trial_001.tif", "folder/trial_002.tif", "folder/trial_003.tif"]
```

or as a list of arrays which you have already loaded into memory:

```
images = [array1, array2, array3, ...]
```

For the regions of interest (ROIs) input, you can either provide a single set of ROIs, or a set of ROIs for every image.

If the ROIs were defined using ImageJ, use ImageJ's export function to save them in a zip. Then, provide the ROI filename.

```
rois = "rois.zip"  # for a single set of ROIs used across all images
```

The same set of ROIs will be used for every image in `images`.

Sometimes there is motion between trials causing the alignment of the ROIs to drift. In such a situation, you may need to use a slightly different location of the ROIs for each trial. This can be handled by providing FISSA with a list of ROI sets — one ROI set (i.e. one ImageJ zip file) per trial.

```
rois = ["rois1.zip", "rois2.zip", ...]  # for a unique roiset for each image
```

Please note that the ROIs defined in each ROI set must correspond to the same physical reigons across all trials, and that the order must be consistent. That is to say, the 1st ROI listed in each ROI set must correspond to the same item appearing in each trial, etc.

In this notebook, we will demonstrate how to use FISSA with ImageJ ROI sets, saved as zip files. However, you are not restricted to providing your ROIs to FISSA in this format. FISSA will also accept ROIs which are arbitrarily defined by providing them as arrays (`numpy.ndarray` objects). ROIs provided in this way can be defined either as boolean-valued masks indicating the presence of a ROI per-pixel in the image, or defined as a list of coordinates defining the boundary of the ROI. For examples of such usage, see our Suite2p, CNMF, and SIMA example notebooks.

As an example, we will run FISSA on a small test dataset.

The test dataset can be found and downloaded from the examples folder of the fissa repository, along with the source for this example notebook.

[4]:
```
# Define path to imagery and to the ROI set
images_location = "exampleData/20150529"
rois_location = "exampleData/20150429.zip"

# Create the experiment object
experiment = fissa.Experiment(images_location, rois_location)
```

Extracting traces and separating them

Now we have our experiment object, we need to call the separate() method to run FISSA on the data. FISSA will extract the traces, and then separate them.

[5]:
```
experiment.separate()
```

```
Extracting traces:   0%|          | 0/3 [00:00<?, ?it/s]
Finished extracting raw signals from 4 ROIs across 3 trials in 0.868 seconds.
Separating data:   0%|          | 0/4 [00:00<?, ?it/s]
Finished separating signals from 4 ROIs across 3 trials in 1.694 seconds
```

2.2.3 Accessing results

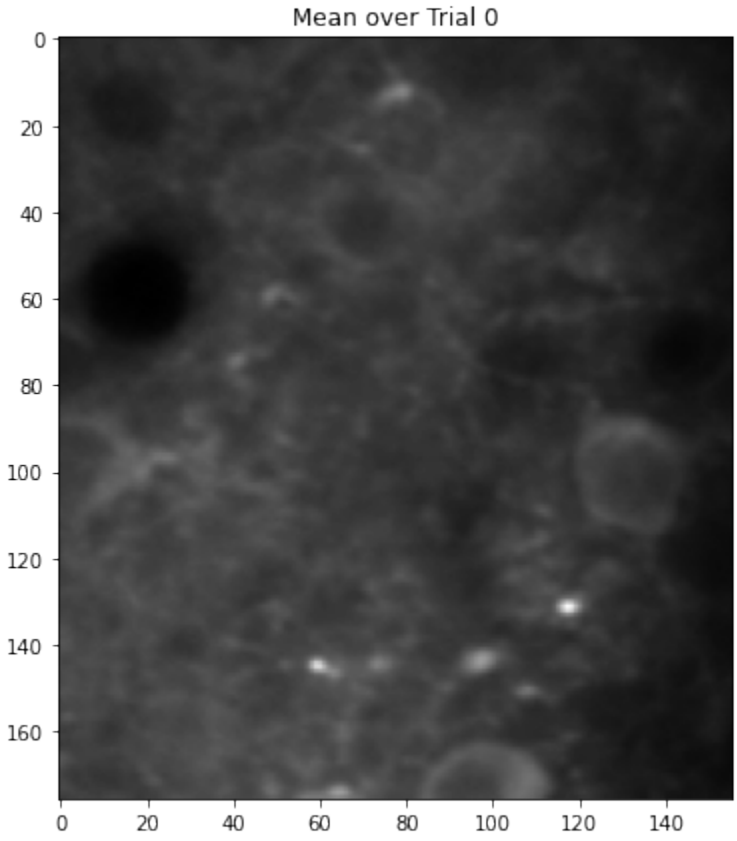
After running experiment.separate() the analysis parameters, raw traces, output signals, ROI definitions, and mean images are stored as attributes of the experiment object, and can be accessed as follows.

Mean image

The temporal-mean image for each trial is stored in `experiment.means`.

We can read out and plot the mean of one of the trials as follows.

[6]:
```
trial = 0
# Plot the mean image for one of the trials
plt.figure(figsize=(7, 7))
plt.imshow(experiment.means[trial], cmap="gray")
plt.title("Mean over Trial {}".format(trial))
plt.show()
```



Plotting the mean image for each trial can be useful to see if there is motion drift between trials.

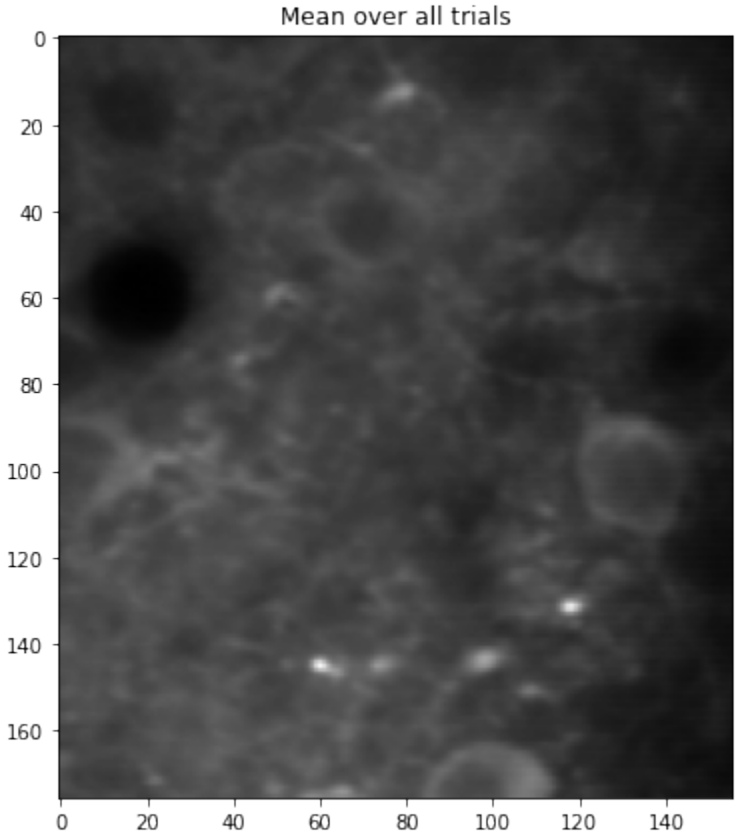
As a summary, you can also take the mean over all trials. Some cells don't appear in every trial, so the overall mean may indicate the location of more cells than the mean image from a single trial.

[7]:
```
# Plot the mean image over all the trials
plt.figure(figsize=(7, 7))
plt.imshow(np.mean(experiment.means, axis=0), cmap="gray")
```

(continues on next page)

(continued from previous page)

```
plt.title("Mean over all trials")
plt.show()
```



ROI outlines

The ROI outlines, and the definitions of the surrounding neuropil regions added by FISSA to determine the contaminating signals, are stored in the `experiment.roi_polys` attribute. For cell number `c` and TIFF number `t`, the set of ROIs for that cell and TIFF is located at

```
experiment.roi_polys[c, t][0][0]  # user-provided ROI, converted to polygon format
experiment.roi_polys[c, t][n][0]  # n = 1, 2, 3, ... the neuropil regions
```

Sometimes ROIs cannot be expressed as a single polygon (e.g. a ring-ROI, which needs a line for the outside and a line for the inside); in those cases several polygons are used to describe it as:

```
experiment.roi_polys[c, t][n][i]  # i iterates over the series of polygons defining the␣
↪ROI
```

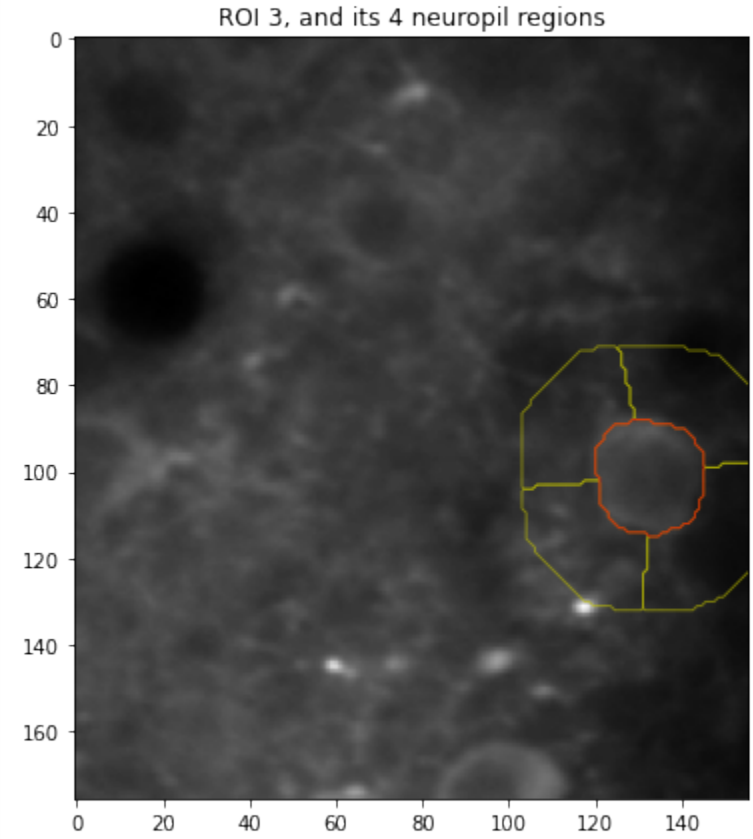
As an example, we will plot the first ROI along with its surrounding neuropil subregions, overlaid on top of the mean image for one trial.

[8]:
```
# Plot one ROI along with its neuropil regions

# Select which ROI and trial to plot
```

(continues on next page)

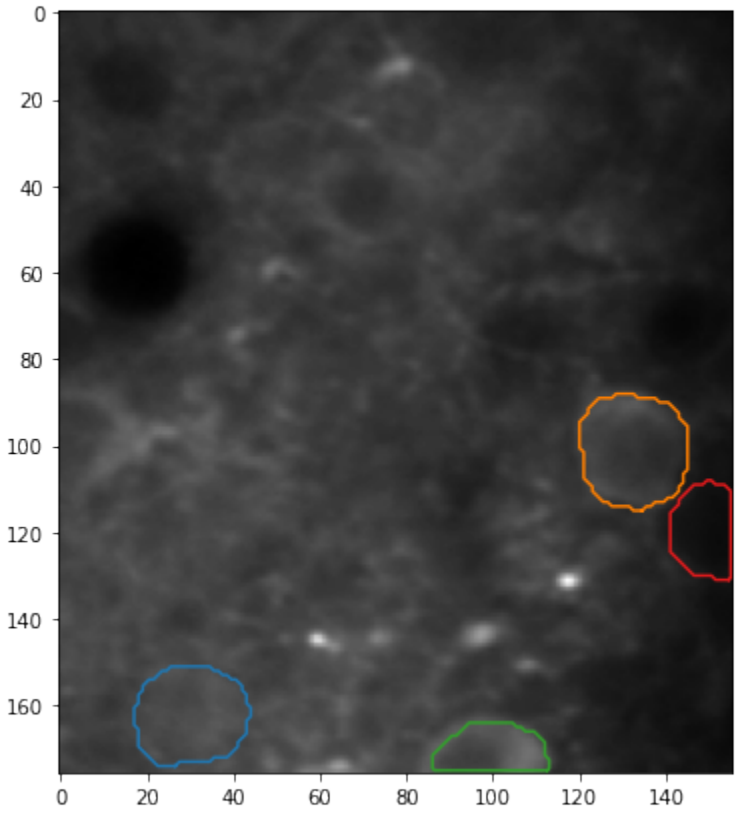
(continued from previous page)

```
trial = 0
roi = 3

# Plot the mean image for the trial
plt.figure(figsize=(7, 7))
plt.imshow(experiment.means[trial], cmap="gray")
# Get current axes limits
XLIM = plt.xlim()
YLIM = plt.ylim()

# Check the number of neuropil regions
n_npil = len(experiment.roi_polys[roi, trial]) - 1

# Plot all the neuropil regions in yellow
for i_npil in range(1, n_npil + 1):
    for contour in experiment.roi_polys[roi, trial][i_npil]:
        plt.fill(
            contour[:, 1],
            contour[:, 0],
            facecolor="none",
            edgecolor="y",
            alpha=0.6,
        )

# Plot the ROI outline in red
for contour in experiment.roi_polys[roi, trial][0]:
    plt.fill(
        contour[:, 1],
        contour[:, 0],
        facecolor="none",
        edgecolor="r",
        alpha=0.6,
    )

# Reset axes limits to be correct for the image
plt.xlim(XLIM)
plt.ylim(YLIM)

plt.title("ROI {}, and its {} neuropil regions".format(roi, experiment.nRegions))
plt.show()
```



Similarly, we can plot the location of all 4 ROIs used in this experiment.

[9]:
```
# Plot all cell ROI locations

# Select which trial (TIFF index) to plot
trial = 0

# Plot the mean image for the trial
plt.figure(figsize=(7, 7))
plt.imshow(experiment.means[trial], cmap="gray")

# Plot each of the cell ROIs
for i_roi in range(len(experiment.roi_polys)):
    # Plot border around ROI
    for contour in experiment.roi_polys[i_roi, trial][0]:
        plt.plot(
            contour[:, 1],
            contour[:, 0],
            color=colors((i_roi * 2 + 1) % colors.N),
        )

plt.show()
```



FISSA extracted traces

The final signals after separation can be found in `experiment.result` as follows. For cell number `c` and TIFF number `t`, the extracted trace is given by:

```
experiment.result[c, t][0, :]
```

In `experiment.result` one can find the signals present in the cell ROI, ordered by how strongly they are present (relative to the surrounding regions). `experiment.result[c, t][0, :]` gives the most strongly present signal, and is considered the cell's "true" signal. `[i, :]` for `i=1,2,3,...` gives the other signals which are present in the ROI, but driven by other cells or neuropil.

Before decontamination

The raw extracted signals can be found in `experiment.raw` in the same way. Now in `experiment.raw[c, t][i, :]`, `i` indicates the region number, with `i=0` being the cell, and `i=1,2,3,...` indicating the surrounding regions.

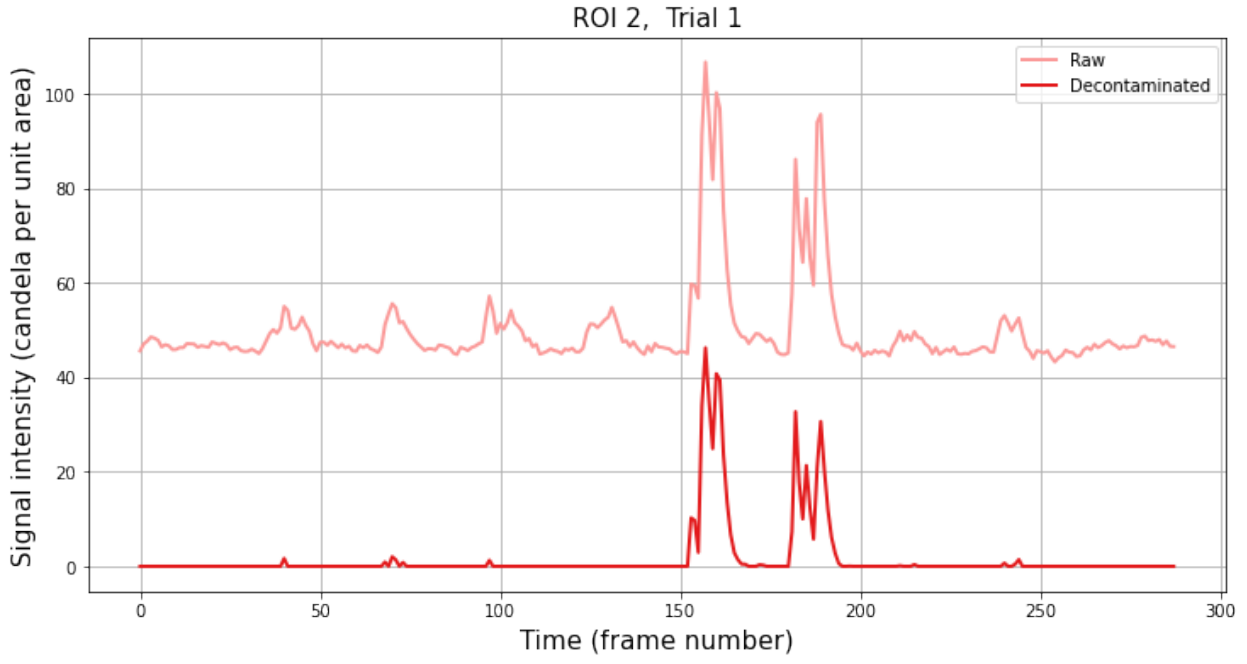
As an example, plotting the raw and extracted signals for the second trial for the third cell:

[10]:
```
# Plot sample trace

# Select the ROI and trial to plot
roi = 2
trial = 1
```

(continues on next page)

(continued from previous page)

```
# Create the figure
plt.figure(figsize=(12, 6))

plt.plot(
    experiment.raw[roi, trial][0, :],
    lw=2,
    label="Raw",
    color=colors((roi * 2) % colors.N),
)
plt.plot(
    experiment.result[roi, trial][0, :],
    lw=2,
    label="Decontaminated",
    color=colors((roi * 2 + 1) % colors.N),
)

plt.title("ROI {},  Trial {}".format(roi, trial), fontsize=15)
plt.xlabel("Time (frame number)", fontsize=15)
plt.ylabel("Signal intensity (candela per unit area)", fontsize=15)
plt.grid()
plt.legend()
plt.show()
```



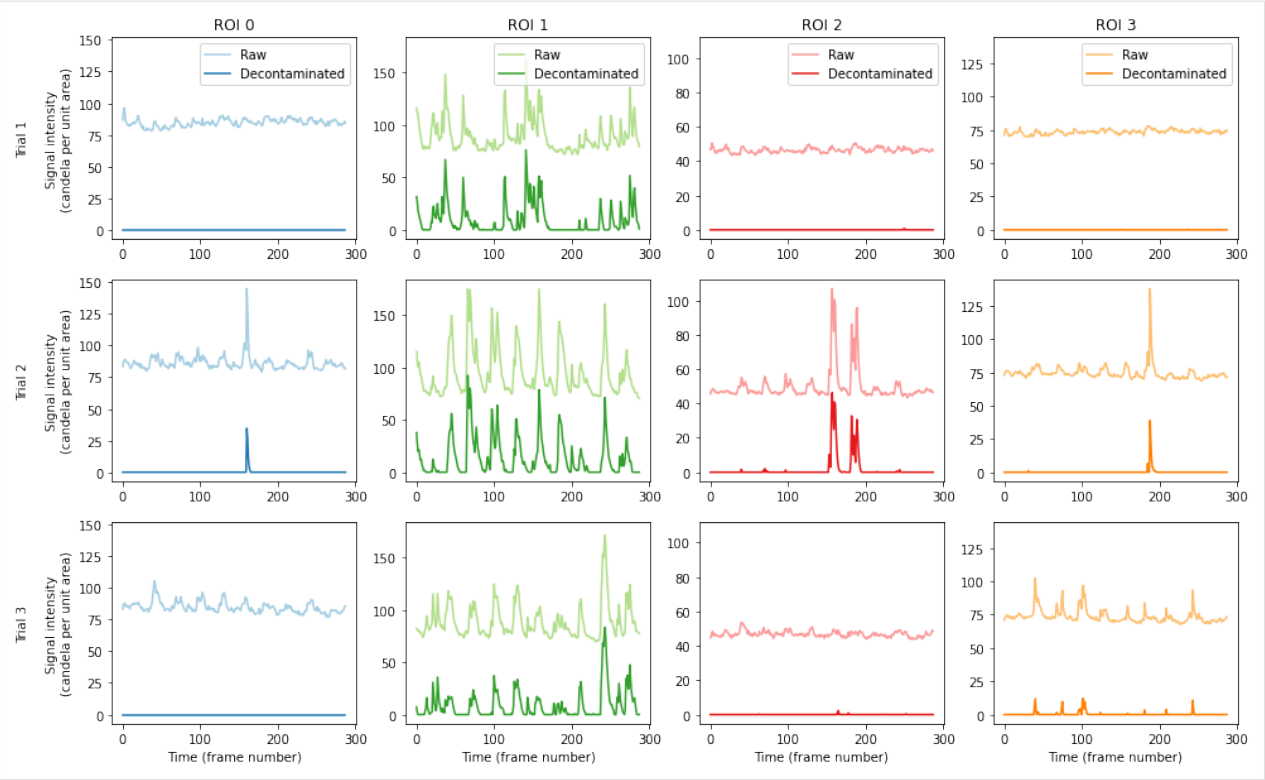
We can similarly plot raw and decontaminated traces for every ROI and every trial.

[11]:
```
# Plot all ROIs and trials

# Get the number of ROIs and trials
n_roi = experiment.result.shape[0]
n_trial = experiment.result.shape[1]
```

(continues on next page)

(continued from previous page)

```
# Find the maximum signal intensities for each ROI
roi_max_raw = [
    np.max([np.max(experiment.raw[i_roi, i_trial][0]) for i_trial in range(n_trial)])
    for i_roi in range(n_roi)
]
roi_max_result = [
    np.max([np.max(experiment.result[i_roi, i_trial][0]) for i_trial in range(n_trial)])
    for i_roi in range(n_roi)
]
roi_max = np.maximum(roi_max_raw, roi_max_result)

# Plot our figure using subplot panels
plt.figure(figsize=(16, 10))
for i_roi in range(n_roi):
    for i_trial in range(n_trial):
        # Make subplot axes
        i_subplot = 1 + i_trial * n_roi + i_roi
        plt.subplot(n_trial, n_roi, i_subplot)
        # Plot the data
        plt.plot(
            experiment.raw[i_roi][i_trial][0, :],
            label="Raw",
            color=colors((i_roi * 2) % colors.N),
        )
        plt.plot(
            experiment.result[i_roi][i_trial][0, :],
            label="Decontaminated",
            color=colors((i_roi * 2 + 1) % colors.N),
        )
        # Labels and boiler plate
        plt.ylim([-0.05 * roi_max[i_roi], roi_max[i_roi] * 1.05])
        if i_roi == 0:
            plt.ylabel(
                "Trial {}\n\nSignal intensity\n(candela per unit area)".format(
                    i_trial + 1
                )
            )
        if i_trial == 0:
            plt.title("ROI {}".format(i_roi))
            plt.legend()
        if i_trial == n_trial - 1:
            plt.xlabel("Time (frame number)")

plt.show()
```



The figure above shows the raw signal from the annotated ROI location (pale), and the result after decontaminating the signal with FISSA (dark). The hues match the ROI locations drawn above. Each column shows the results from one of the ROI, and each row shows the results from one of the three trials.

Comparing ROI signal to neuropil region signals

It can be very instructive to compare the signal in the central ROI with the surrounding neuropil regions. These can be found for cell `c` and trial `t` in `experiment.raw[c, t][i, :]`, with `i=0` being the cell, and `i=1,2,3,...` indicating the surrounding regions.

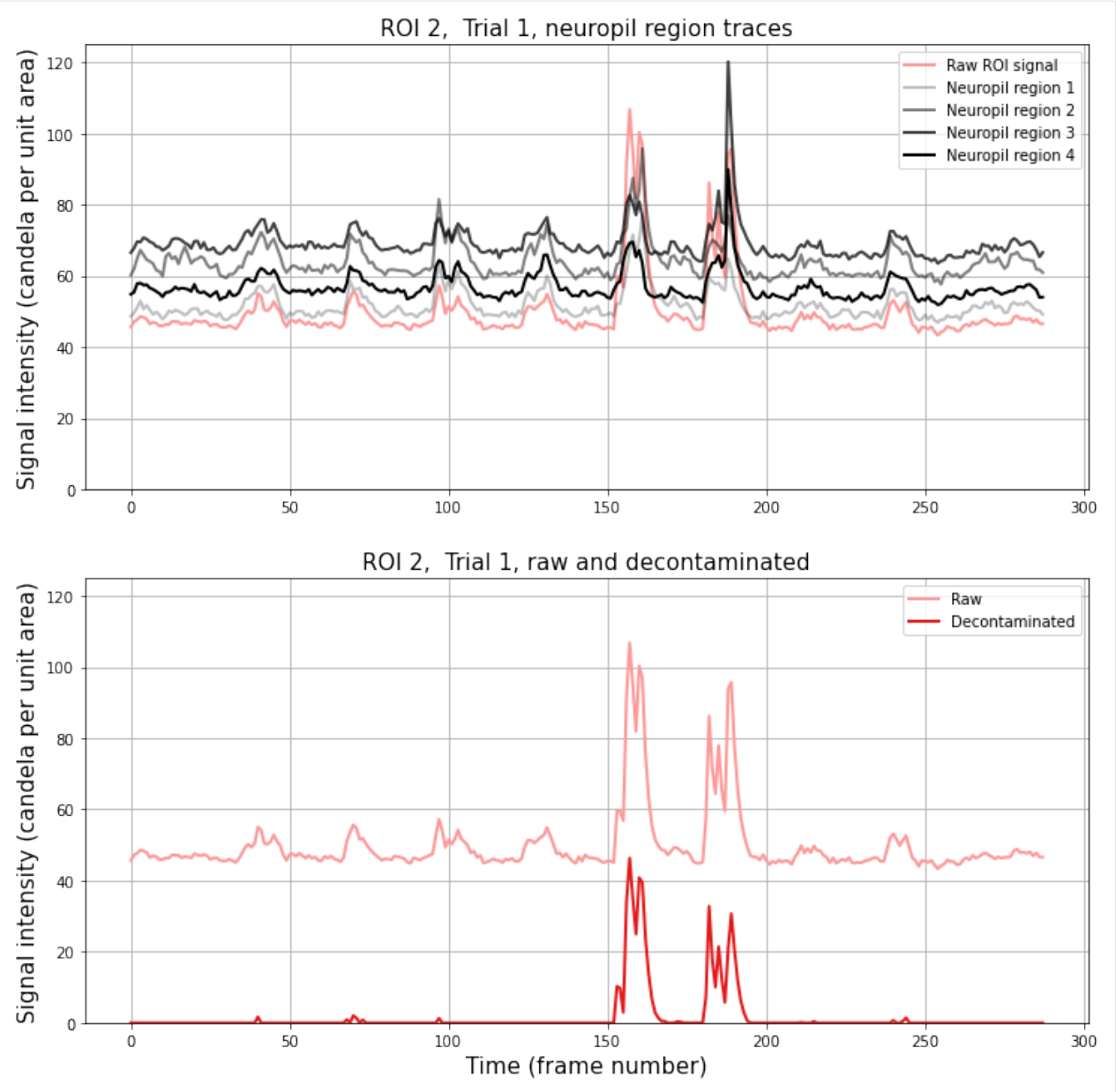
Below we compare directly the raw ROI trace, the decontaminated trace, and the surrounding neuropil region traces.

[12]:
```
# Get the total number of regions
nRegions = experiment.nRegions

# Select the ROI and trial to plot
roi = 2
trial = 1

# Create the figure
plt.figure(figsize=(12, 12))

# Plot extracted traces for each neuropil subregion
plt.subplot(2, 1, 1)
# Plot trace of raw ROI signal
plt.plot(
    experiment.raw[roi, trial][0, :],
    lw=2,
```

(continues on next page)

(continued from previous page)

```
    label="Raw ROI signal",
    color=colors((roi * 2) % colors.N),
)
# Plot traces from each neuropil region
for i_neuropil in range(1, nRegions + 1):
    alpha = i_neuropil / nRegions
    plt.plot(
        experiment.raw[roi, trial][i_neuropil, :],
        lw=2,
        label="Neuropil region {}".format(i_neuropil),
        color="k",
        alpha=alpha,
    )
plt.ylim([0, 125])
plt.grid()
plt.legend()
plt.ylabel("Signal intensity (candela per unit area)", fontsize=15)
plt.title("ROI {},  Trial {}, neuropil region traces".format(roi, trial), fontsize=15)

# Plot the ROI signal
plt.subplot(2, 1, 2)
# Plot trace of raw ROI signal
plt.plot(
    experiment.raw[roi, trial][0, :],
    lw=2,
    label="Raw",
    color=colors((roi * 2) % colors.N),
)
# Plot decontaminated signal matched to the ROI
plt.plot(
    experiment.result[roi, trial][0, :],
    lw=2,
    label="Decontaminated",
    color=colors((roi * 2 + 1) % colors.N),
)

plt.ylim([0, 125])
plt.grid()
plt.legend()
plt.xlabel("Time (frame number)", fontsize=15)
plt.ylabel("Signal intensity (candela per unit area)", fontsize=15)
plt.title("ROI {},  Trial {}, raw and decontaminated".format(roi, trial), fontsize=15)

plt.show()
```



df/f0

It is often useful to calculate the intensity of a signal relative to the baseline value, df/f0, for the traces. This can be done with FISSA by calling the experiment.calc_deltaf method as follows.

[13]:
```
sampling_frequency = 10  # Hz

experiment.calc_deltaf(freq=sampling_frequency)
```

```
Calculating Δf/f0:   0%|          | 0/4 [00:00<?, ?it/s]
```

```
Finished calculating Δf/f0 for raw and result signals in 0.050 seconds
```

The sampling frequency is required because we our process for determining f0 involves applying a lowpass filter to the

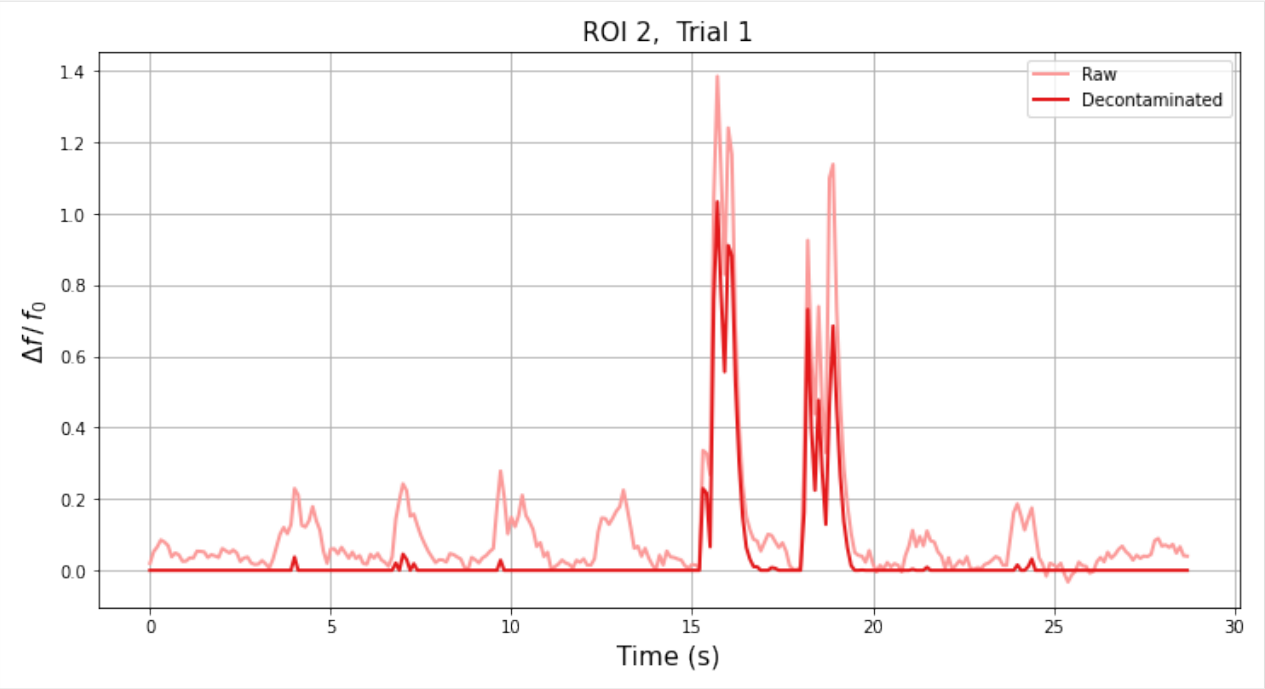
data.

Note that by default, f0 is determined as the minimum across all trials (all TIFFs) to ensure that results are directly comparable between trials, but you can normalise each trial individually instead if you prefer by providing the parameter `across_trials=False`.

Since FISSA is very good at removing contamination from the ROI signals, the minimum value on the decontaminated trace will typically be `0.`. Consequently, we use the minimum value of the (smoothed) raw signal to provide the f0 from the raw trace for both the raw and decontaminated df/f0.

As we performed above, we can plot the raw and decontaminated df/f0 for each ROI in each trial.

[14]:
```
# Plot sample df/f0 trace

# Select the ROI and trial to plot
roi = 2
trial = 1

# Create the figure
plt.figure(figsize=(12, 6))

n_frames = experiment.deltaf_result[roi, trial].shape[1]
tt = np.arange(0, n_frames, dtype=np.float64) / sampling_frequency

plt.plot(
    tt,
    experiment.deltaf_raw[roi, trial][0, :],
    lw=2,
    label="Raw",
    color=colors((roi * 2) % colors.N),
)
plt.plot(
    tt,
    experiment.deltaf_result[roi, trial][0, :],
    lw=2,
    label="Decontaminated",
    color=colors((roi * 2 + 1) % colors.N),
)

plt.title("ROI {},  Trial {}".format(roi, trial), fontsize=15)
plt.xlabel("Time (s)", fontsize=15)
plt.ylabel(r"$\Delta f\,/\,f_0$", fontsize=15)
plt.grid()
plt.legend()
plt.show()
```



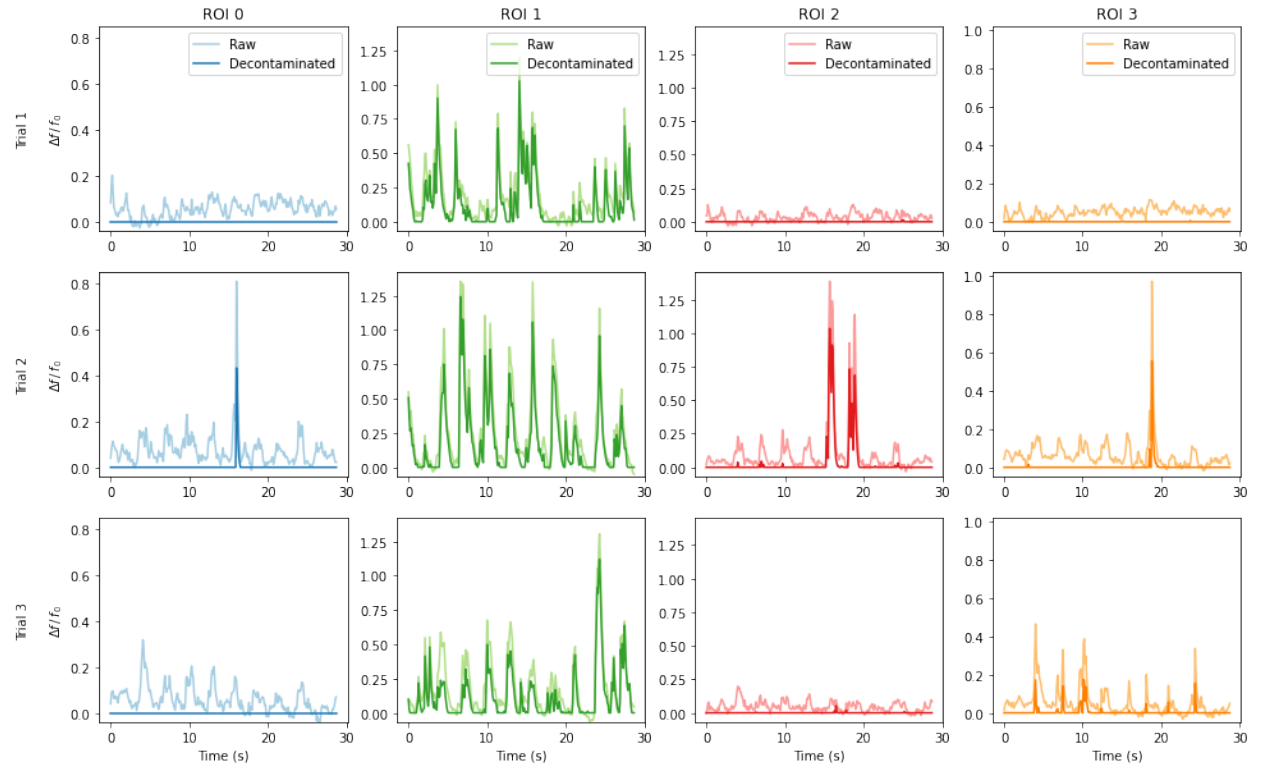
We can also plot df/f0 for the raw data to compare against the decontaminated signal for each ROI and each trial.

[15]:
```
# Plot df/f0 for all ROIs and trials

# Find the maximum df/f0 values for each ROI
roi_max_raw = [
    np.max(
        [np.max(experiment.deltaf_raw[i_roi, i_trial][0]) for i_trial in range(n_trial)]
    )
    for i_roi in range(n_roi)
]
roi_max_result = [
    np.max(
        [
            np.max(experiment.deltaf_result[i_roi, i_trial][0])
            for i_trial in range(n_trial)
        ]
    )
    for i_roi in range(n_roi)
]
roi_max = np.maximum(roi_max_raw, roi_max_result)

# Plot our figure using subplot panels
plt.figure(figsize=(16, 10))
for i_roi in range(n_roi):
    for i_trial in range(n_trial):
        # Make subplot axes
        i_subplot = 1 + i_trial * n_roi + i_roi
        plt.subplot(n_trial, n_roi, i_subplot)
        # Plot the data
        n_frames = experiment.deltaf_result[i_roi, i_trial].shape[1]
```

(continues on next page)

(continued from previous page)

```
        tt = np.arange(0, n_frames, dtype=np.float64) / sampling_frequency
        plt.plot(
            tt,
            experiment.deltaf_raw[i_roi][i_trial][0, :],
            label="Raw",
            color=colors((i_roi * 2) % colors.N),
        )
        plt.plot(
            tt,
            experiment.deltaf_result[i_roi][i_trial][0, :],
            label="Decontaminated",
            color=colors((i_roi * 2 + 1) % colors.N),
        )
        # Labels and boiler plate
        plt.ylim([-0.05 * roi_max[i_roi], roi_max[i_roi] * 1.05])
        if i_roi == 0:
            plt.ylabel("Trial {}\n\n".format(i_trial + 1) + r"$\Delta f\,/\,f_0$")
        if i_trial == 0:
            plt.title("ROI {}".format(i_roi))
            plt.legend()
        if i_trial == n_trial - 1:
            plt.xlabel("Time (s)")

plt.show()
```



The figure above shows the df/f0 for the raw signal from the annotated ROI location (pale), and the result after decontaminating the signal with FISSA (dark). For each figure, the baseline value f0 is the same (taken from the raw signal). The hues match the ROI locations and fluorescence intensity traces from above. Each column shows the results from

one of the ROI, and each row shows the results from one of the three trials.

2.2.4 Caching

After using FISSA to clean the data from an experiment, you will probably want to save the output for later use, so you don't have to keep re-running FISSA on the data all the time.

An option to cache the results is built into FISSA. If you provide `fissa.run_fissa` with the name of the experiment using the `folder` argument, it will cache results into that directory. Later, if you call `fissa.run_fissa` again with the same experiment name (`folder` argument), it will load the saved results from the cache instead of recomputing them.

[16]:
```
# Define the folder where FISSA's outputs will be cached, so they can be
# quickly reloaded in the future without having to recompute them.
#
# This argument is optional; if it is not provided, FISSA will not save its
# results for later use.
#
# Note: you *must* use a different folder for each experiment,
# otherwise FISSA will load in the folder provided instead
# of computing results for the new experiment.

output_folder = "fissa-example"
```

[17]:
```
# Create a new experiment object set up to save results to the specified output folder
experiment = fissa.Experiment(images_location, rois_location, folder=output_folder)
```

```
Loading data from cache fissa-example/prepared.npz
Loading data from cache fissa-example/separated.npz
```

Because we have created a new experiment object, it is yet not populated with our results.

We need to run the separate routine again to generate the outputs. But this time, our results will be saved to the directory named `fissa-example` for future reference.

[18]:
```
experiment.separate()
```

```
Loading data from cache fissa-example/separated.npz
```

Calling the separate method again, or making a new Experiment with the same experiment `folder` name will not have to re-run FISSA because it can use load the pre-computed results from the cache instead.

[19]:
```
experiment.separate()
```

```
Loading data from cache fissa-example/separated.npz
```

If you need to *force* FISSA to ignore the cache and rerun the preparation and/or separation step, you can call it with `redo_prep=True` and/or `redo_sep=True` as appropriate.

[20]:
```
experiment.separate(redo_prep=True, redo_sep=True)
```

```
Extracting traces:   0%|          | 0/3 [00:00<?, ?it/s]
```

```
Finished extracting raw signals from 4 ROIs across 3 trials in 0.828 seconds.
Saving extracted traces to fissa-example/prepared.npz
```

```
Separating data:   0%|          | 0/4 [00:00<?, ?it/s]
```

```
Finished separating signals from 4 ROIs across 3 trials in 1.708 seconds
Saving results to fissa-example/separated.npz
```

Exporting to MATLAB

The results can easily be exported to a MATLAB-compatible MAT-file by calling the experiment.to_matfile() method.

The results can easily be exported to a MATLAB-compatible matfile as follows.

The output file, `"separated.mat"`, will appear in the `output_folder` we supplied to `experiment` when we created it.

[21]:
```
experiment.to_matfile()
```

Loading the generated file (e.g. `"output_folder/separated.mat"`) in MATLAB will provide you with all of FISSA's outputs.

These are structured similarly to `experiment.raw` and `experiment.result` described above, with a few small differences.

With the python interface, the outputs are 2d numpy.ndarrays each element of which is itself a 2d numpy.ndarrays. In comparison, when the output is loaded into MATLAB this becomes a 2d cell-array each element of which is a 2d matrix.

Additionally, whilst Python indexes from 0, MATLAB indexes from 1 instead. As a consequence of this, the results seen on Python for a given roi and trial `experiment.result[roi, trial]` correspond to the index `S.result{roi + 1, trial + 1}` on MATLAB.

Our first plot in this notebook can be replicated in MATLAB as follows:

```
%% Plot example traces
% Load the FISSA output data
S = load('fissa-example/separated.mat')
% Select the third ROI, second trial
% (On Python, this would be roi = 2; trial = 1;)
roi = 3; trial = 2;
% Plot the raw and result traces for the ROI signal
figure;
hold on;
plot(S.raw{roi, trial}(1, :));
plot(S.result{roi, trial}(1, :));
title(sprintf('ROI %d, Trial %d', roi, trial));
xlabel('Time (frame number)');
ylabel('Signal intensity (candela per unit area)');
legend({'Raw', 'Result'});
grid on;
box on;
set(gca,'TickDir','out');
```

Assuming all ROIs are contiguous and described by a single contour, the mean image and ROI locations can be plotted in MATLAB as follows:

```
%% Plot ROI locations overlaid on mean image
% Load the FISSA output data
S = load('fissa-example/separated.mat')
trial = 1;
figure;
hold on;
% Plot the mean image
imagesc(squeeze(S.means(trial, :, :)));
colormap('gray');
% Plot ROI locations
for i_roi = 1:size(S.result, 1);
    contour = S.roi_polys{i_roi, trial}{1};
    plot(contour(:, 2), contour(:, 1));
end
set(gca, 'YDir', 'reverse');
```

2.2.5 Addendum

Finding the TIFF files

If you find something noteworthy in one of the traces and need to backreference to the corresponding TIFF file, you can look up the path to the TIFF file with `experiment.images`.

[22]:
```
trial_of_interest = 1

print(experiment.images[trial_of_interest])
```

```
exampleData/20150529/AVG_A02.tif
```

2.2.6 FISSA customisation settings

FISSA has several user-definable settings, which can be set when defining the `fissa.Experiment` instance.

Controlling verbosity

The level of verbosity of FISSA can be controlled with the `verbosity` parameter.

The default is `verbosity=1`.

If the verbosity parameter is higher, FISSA will print out more information while it is processing. This can be helpful for debugging puproses. The verbosity reaches its maximum at `verbosity=6`.

If `verbosity=0`, FISSA will run silently.

[23]:
```
# Call FISSA with elevated verbosity
experiment = fissa.Experiment(images_location, rois_location, verbosity=2)
experiment.separate()
```

```
Doing region growing and data extraction for 3 trials...
  Images:
    exampleData/20150529/AVG_A01.tif
    exampleData/20150529/AVG_A02.tif
```

(continues on next page)

(continued from previous page)

```
    exampleData/20150529/AVG_A03.tif
  ROI sets:
    exampleData/20150429.zip
    exampleData/20150429.zip
    exampleData/20150429.zip
  nRegions: 4
  expansion: 1
```

```
Extracting traces:   0%|          | 0/3 [00:00<?, ?it/s]
```

```
Finished extracting raw signals from 4 ROIs across 3 trials in 0.737 seconds.
Doing signal separation for 4 ROIs over 3 trials...
  method: 'nmf'
  alpha: 0.1
  max_iter: 20000
  max_tries: 1
  tol: 0.0001
```

```
Separating data:   0%|          | 0/4 [00:00<?, ?it/s]
```

```
Finished separating signals from 4 ROIs across 3 trials in 1.563 seconds
```

Analysis parameters

The analysis performed by FISSA can be controlled with several parameters.

[24]:
```
# FISSA uses multiprocessing to speed up its processing.
# By default, it will spawn one worker per CPU core on your machine.
# However, if you have a lot of cores and not much memory, you many not
# be able to suport so many workers simultaneously.
# In particular, this can be problematic during the data preparation step
# in which tiffs are loaded into memory.
# The default number of cores for the data preparation and separation steps
# can be changed as follows.
ncores_preparation = 4  # If None, uses all available cores
ncores_separation = None  # if None, uses all available cores

# By default, FISSA uses 4 subregions for the neuropil region.
# If you have very dense data with a lot of different signals per unit area,
# you may wish to increase the number of regions.
nRegions = 8

# By default, each surrounding region has the same area as the central ROI.
# i.e. expansion = 1
# However, you may wish to increase or decrease this value.
expansion = 0.75

# The degree of signal sparsity can be controlled with the alpha parameter.
alpha = 0.02

# If you change the experiment parameters, you need to change the cache directory too.
# Otherwise FISSA will try to reload the results from the previous run instead of
# computing the new results. FISSA will throw an error if you try to load data which
```

(continues on next page)

(continued from previous page)

```
# was generated with different analysis parameters to its parameters.
output_folder2 = output_folder + "_alt"

# Set up a FISSA experiment with these parameters
experiment = fissa.Experiment(
    images_location,
    rois_location,
    output_folder2,
    nRegions=nRegions,
    expansion=expansion,
    alpha=alpha,
    ncores_preparation=ncores_preparation,
    ncores_separation=ncores_separation,
)

# Extract the data with these new parameters.
experiment.separate()
```

```
Loading data from cache fissa-example_alt/prepared.npz
Loading data from cache fissa-example_alt/separated.npz
Loading data from cache fissa-example_alt/separated.npz
```

We can plot the new results for our example trace from before. Although we doubled the number of neuropil regions around the cell, very little has changed for this example because there were not many sources of contamination.

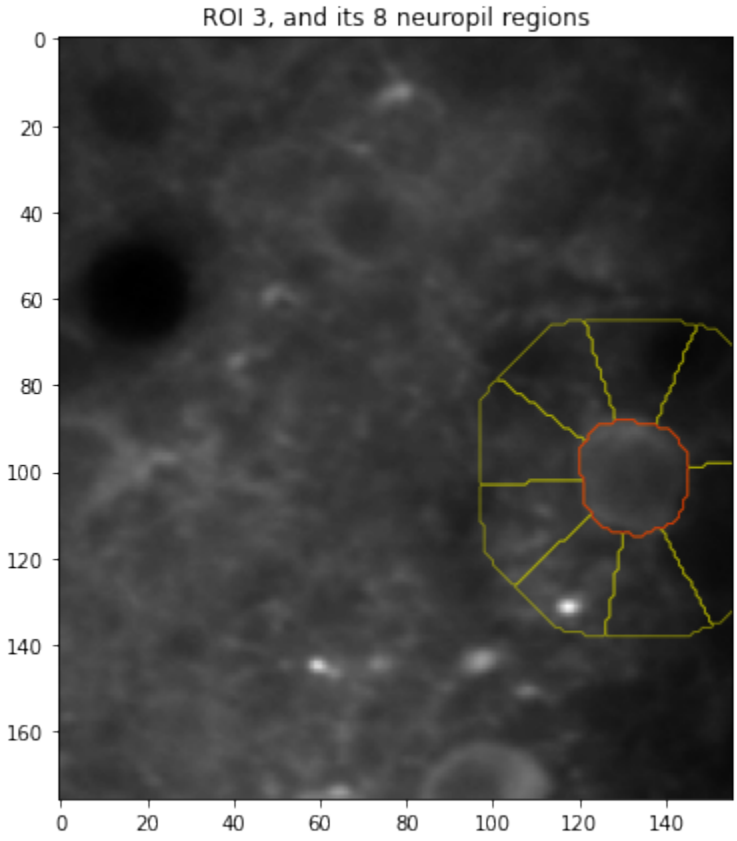
However, there will be more of a difference if your data has more neuropil sources per unit area within the image.

[25]:
```
# Plot one ROI along with its neuropil regions

# Select which ROI and trial to plot
trial = 0
roi = 3

# Plot the mean image for the trial
plt.figure(figsize=(7, 7))
plt.imshow(experiment.means[trial], cmap="gray")
# Get axes limits
XLIM = plt.xlim()
YLIM = plt.ylim()

# Check the number of neuropil
n_npil = len(experiment.roi_polys[roi, trial]) - 1

# Plot all the neuropil regions in yellow
for i_npil in range(1, n_npil + 1):
    for contour in experiment.roi_polys[roi, trial][i_npil]:
        plt.fill(
            contour[:, 1],
            contour[:, 0],
            facecolor="none",
            edgecolor="y",
            alpha=0.6,
        )
```

(continues on next page)

(continued from previous page)

```
# Plot the ROI outline in red
for contour in experiment.roi_polys[roi, trial][0]:
    plt.fill(
        contour[:, 1],
        contour[:, 0],
        facecolor="none",
        edgecolor="r",
        alpha=0.6,
    )

# Reset axes limits
plt.xlim(XLIM)
plt.ylim(YLIM)

plt.title("ROI {}, and its {} neuropil regions".format(roi, experiment.nRegions))
plt.show()
```

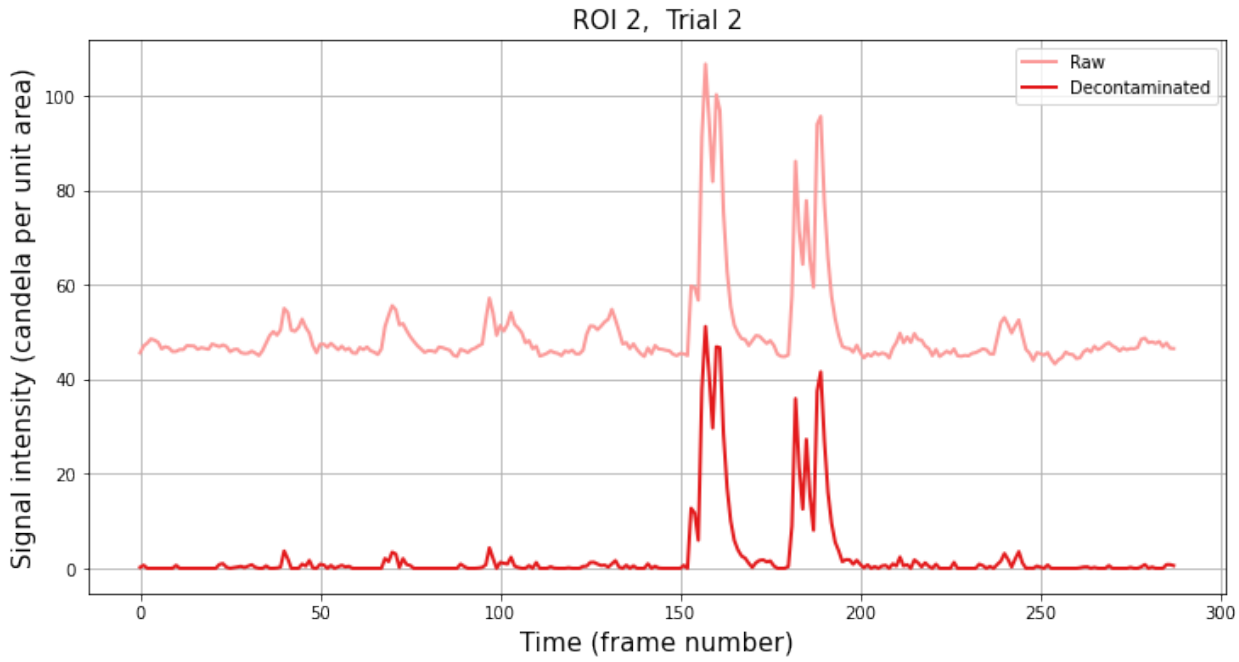


[26]:
```
# Plot the new results
roi = 2
trial = 1

plt.figure(figsize=(12, 6))
```

(continues on next page)

(continued from previous page)

```
plt.plot(
    experiment.raw[roi, trial][0, :],
    lw=2,
    label="Raw",
    color=colors((roi * 2) % colors.N),
)
plt.plot(
    experiment.result[roi, trial][0, :],
    lw=2,
    label="Decontaminated",
    color=colors((roi * 2 + 1) % colors.N),
)

plt.title("ROI {},  Trial {}".format(roi, i_trial), fontsize=15)
plt.xlabel("Time (frame number)", fontsize=15)
plt.ylabel("Signal intensity (candela per unit area)", fontsize=15)
plt.grid()
plt.legend()
plt.show()
```



Alternatively, these settings can be refined after creating the `experiment` object, as follows.

[27]:
```
experiment.ncores_preparation = 8
experiment.alpha = 0.02
experiment.expansion = 0.75
```

Loading data from large tiff files

By default, FISSA loads entire tiff files into memory at once and then manipulates all ROIs within the tiff. This can sometimes be problematic when working with very large tiff files which can not be loaded into memory all at once. If you have out-of-memory problems, you can activate FISSA's low memory mode, which will cause it to manipulate each tiff file frame-by-frame.

[28]:
```
experiment = fissa.Experiment(
    images_location, rois_location, output_folder, lowmemory_mode=True
)
experiment.separate(redo_prep=True)
```

```
Loading data from cache fissa-example/prepared.npz
Loading data from cache fissa-example/separated.npz
```

```
Extracting traces:   0%|          | 0/3 [00:00<?, ?it/s]
```

```
Finished extracting raw signals from 4 ROIs across 3 trials in 3.13 seconds.
Saving extracted traces to fissa-example/prepared.npz
```

```
Separating data:   0%|          | 0/4 [00:00<?, ?it/s]
```

```
Finished separating signals from 4 ROIs across 3 trials in 3.91 seconds
Saving results to fissa-example/separated.npz
```

Handling custom formats

By default, FISSA can use tiff files or numpy arrays as its input image data, and numpy arrays or ImageJ zip files for the ROI definitions. However, it is also possible to extend this functionality and integrate other data formats into FISSA in order to work with other custom and/or proprietary formats that might be used in your lab.

This is done by defining your own DataHandler class. Your custom data handler should be a subclass of fissa.extraction.DataHandlerAbstract, and implement the following methods:

- `image2array(image)` takes an image of whatever format and turns it into *data* (typically a numpy.ndarray).
- `getmean(data)` calculates the 2D mean for a video defined by *data*.
- `rois2masks(rois, data)` creates masks from the rois inputs, of appropriate size *data*.
- `extracttraces(data, masks)` applies the *masks* to *data* in order to extract traces.

See fissa.extraction.DataHandlerAbstract for further description for each of the methods.

If you only need to handle a new *image* input format, which is converted to a numpy.ndarray, you may find it is easier to create a subclass of the default datahandler, fissa.extraction.DataHandlerTifffile. In this case, only the `image2array` method needs to be overwritten and the other methods can be left as they are.

[29]:
```
from fissa.extraction import DataHandlerTifffile

# Define a custom datahandler class.
#
# By inheriting from DataHandlerTifffile, most methods are defined
# appropriately. In this case, we only need to overwrite the
# `image2array` method to work with our custom data format.


class DataHandlerCustom(DataHandlerTifffile):
```

(continues on next page)

(continued from previous page)

```
    @staticmethod
    def image2array(image):
        """Open a given image file as a custom instance.

        Parameters
        ----------
        image : custom
            Your image format (avi, hdf5, etc.)

        Returns
        -------
        numpy.ndarray
            A 3D array containing the data, shaped
            ``(frames, y_coordinate, x_coordinate)``.
        """
        # Some custom code
        pass


# Then pass an instance of this class to fissa.Experiment when creating
# a new experiment.
datahandler = DataHandlerCustom()

experiment = fissa.Experiment(
    images_location,
    rois_location,
    datahandler=datahandler,
)
```

For advanced users that want to entirely replace the DataHandler with their own methods, you can also inherit a class from the abstract class, fissa.extraction.DataHandlerAbstract. This can be useful if you want to integrate FISSA into your workflow without changing everything into the numpy array formats that FISSA usually uses internally.

2.3 Using FISSA with Suite2p

suite2p is a blind source separation toolbox for cell detection and signal extraction.

Here we illustrate how to use suite2p to detect cell locations, and then use FISSA to remove neuropil signals from the ROI signals.

The suite2p parts of this tutorial are based on their Jupyter notebook example.

Note that the below results are not representative of either suite2p or FISSA performance, as we are using a very small example dataset.

Reference: Pachitariu, M., Stringer, C., Dipoppa, M., Schröder, S., Rossi, L. F., Dalgleish, H., Carandini, M. & Harris, K. D. (2017). Suite2p: beyond 10,000 neurons with standard two-photon microscopy. bioRxiv: 061507; doi: 10.1101/061507.

2.3.1 Import packages

[1]:
```
# FISSA toolbox
import fissa

# suite2p toolbox
import suite2p

# For plotting our results, use numpy and matplotlib
import matplotlib.pyplot as plt
import numpy as np
```

2.3.2 Run suite2p

[2]:
```
# Set your options for running
ops = suite2p.default_ops()  # populates ops with the default options

# Provide an h5 path in 'h5py' or a tiff path in 'data_path'
# db overwrites any ops (allows for experiment specific settings)
db = {
    "h5py": [],  # a single h5 file path
    "h5py_key": "data",
    "look_one_level_down": False,  # whether to look in ALL subfolders when searching␣
↪for tiffs
    "data_path": ["exampleData/20150529"],  # a list of folders with tiffs
    # (or folder of folders with tiffs if look_one_level_down is True,
    # or subfolders is not empty)
    "save_path0": "./",  # save path
    "subfolders": [],  # choose subfolders of 'data_path' to look in (optional)
    "fast_disk": "./",  # specify where the binary file will be stored (should be an SSD)
    "reg_tif": True,  # save the motion corrected tiffs
    "tau": 0.7,  # timescale of gcamp6f
    "fs": 1,  # sampling rate
    "spatial_scale": 10,  # rough guess of spatial scale cells
    "batch_size": 288,  # length in frames of each trial
}

# Run one experiment
opsEnd = suite2p.run_s2p(ops=ops, db=db)
```

```
{'h5py': [], 'h5py_key': 'data', 'look_one_level_down': False, 'data_path': [
↪'exampleData/20150529'], 'save_path0': './', 'subfolders': [], 'fast_disk': './', 'reg_
↪tif': True, 'tau': 0.7, 'fs': 1, 'spatial_scale': 10, 'batch_size': 288}
tif
** Found 3 tifs - converting to binary **
time 0.53 sec. Wrote 864 frames per binary for 1 planes
>>>>>>>>>>>>>>>>>>>> PLANE 0 <<<<<<<<<<<<<<<<<<<<
NOTE: not registered / registration forced with ops['do_registration']>1
      (no previous offsets to delete)
----------- REGISTRATION
registering 864 frames
Reference frame, 4.05 sec.
```

(continues on next page)

(continued from previous page)

```
added enhanced mean image
----------- Total 11.24 sec
NOTE: Applying builtin classifier at ~/miniconda/envs/suite2p-fissa/lib/python3.7/site-
↪packages/suite2p/classifiers/classifier.npy
----------- ROI DETECTION
Binning movie in chunks of length 01
Binned movie [500,172,152] in 0.07 sec.
NOTE: FORCED spatial scale ~48 pixels, time epochs 1.00, threshold 20.00
0 ROIs, score=237.58
Detected 23 ROIs, 0.57 sec
After removing overlaps, 23 ROIs remain
----------- Total 0.71 sec.
----------- EXTRACTION
Masks created, 0.06 sec.
Extracted fluorescence from 23 ROIs in 864 frames, 2.42 sec.
----------- Total 2.48 sec.
----------- CLASSIFICATION
['skew', 'compact', 'npix_norm']
----------- Total 0.02 sec.
----------- SPIKE DECONVOLUTION
----------- Total 0.00 sec.
Plane 0 processed in 14.45 sec (can open in GUI).
total = 14.98 sec.
TOTAL RUNTIME 14.98 sec
```

2.3.3 Load the relevant data from the analysis

[3]:
```
# Extract the motion corrected tiffs (make sure that the reg_tif option is set to true,␣
↪see above)
images = "./suite2p/plane0/reg_tif"

# Load the detected regions of interest
stat = np.load("./suite2p/plane0/stat.npy", allow_pickle=True)  # cell stats
ops = np.load("./suite2p/plane0/ops.npy", allow_pickle=True).item()
iscell = np.load("./suite2p/plane0/iscell.npy", allow_pickle=True)[:, 0]

# Get image size
Lx = ops["Lx"]
Ly = ops["Ly"]

# Get the cell ids
ncells = len(stat)
cell_ids = np.arange(ncells)  # assign each cell an ID, starting from 0.
cell_ids = cell_ids[iscell == 1]  # only take the ROIs that are actually cells.
num_rois = len(cell_ids)

# Generate ROI masks in a format usable by FISSA (in this case, a list of masks)
rois = [np.zeros((Ly, Lx), dtype=bool) for n in range(num_rois)]

for i, n in enumerate(cell_ids):
```

(continues on next page)

(continued from previous page)

```
    # i is the position in cell_ids, and n is the actual cell number
    ypix = stat[n]["ypix"][~stat[n]["overlap"]]
    xpix = stat[n]["xpix"][~stat[n]["overlap"]]
    rois[i][ypix, xpix] = 1
```

2.3.4 Run FISSA with the defined ROIs and data

[4]:
```
output_folder = "fissa_suite2p_example"
experiment = fissa.Experiment(images, [rois[:ncells]], output_folder)
experiment.separate()
```

```
Extracting traces:   0%|          | 0/3 [00:00<?, ?it/s]
```

```
Finished extracting raw signals from 21 ROIs across 3 trials in 0.317 seconds.
Saving extracted traces to fissa_suite2p_example/prepared.npz
```

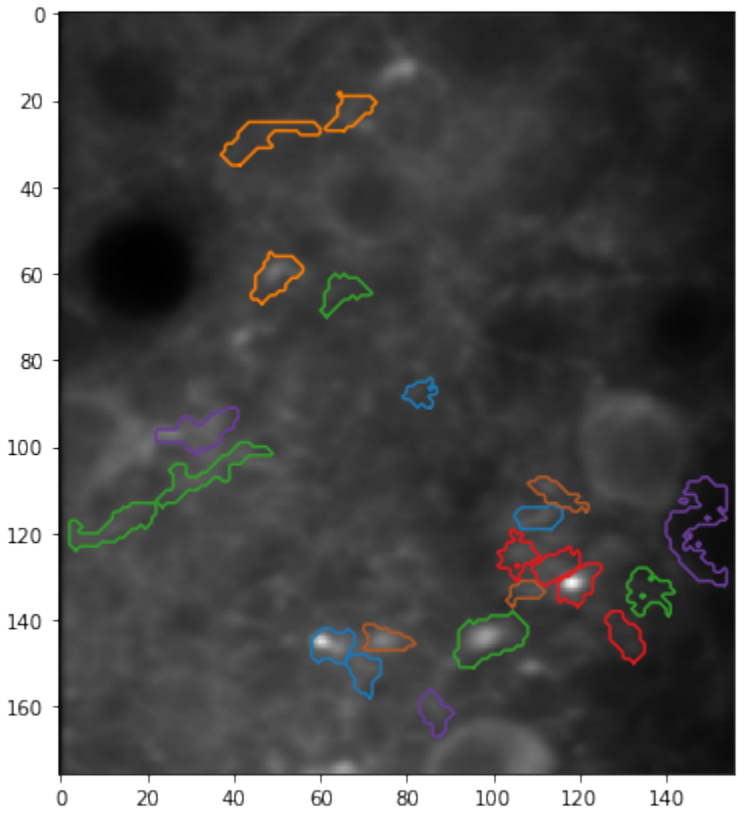
```
Separating data:   0%|          | 0/21 [00:00<?, ?it/s]
```

```
Finished separating signals from 21 ROIs across 3 trials in 5.47 seconds
Saving results to fissa_suite2p_example/separated.npz
```

2.3.5 Plot the resulting ROI signals

[5]:
```
# Fetch the colormap object for Cynthia Brewer's Paired color scheme
cmap = plt.get_cmap("Paired")
```

[6]:
```
# Select which trial (TIFF index) to plot
trial = 0

# Plot the mean image and ROIs from the FISSA experiment
plt.figure(figsize=(7, 7))
plt.imshow(experiment.means[trial], cmap="gray")

XLIM = plt.xlim()
YLIM = plt.ylim()

for i_roi in range(len(experiment.roi_polys)):
    # Plot border around ROI
    for contour in experiment.roi_polys[i_roi, trial][0]:
        plt.plot(
            contour[:, 1],
            contour[:, 0],
            color=cmap((i_roi * 2 + 1) % cmap.N),
        )

# ROI co-ordinates are half a pixel outside the image,
# so we reset the axis limits
plt.xlim(XLIM)
plt.ylim(YLIM)
plt.show()
```



[7]:
```
# Plot all ROIs and trials

# Get the number of ROIs and trials
n_roi = experiment.result.shape[0]
n_trial = experiment.result.shape[1]

# Find the maximum signal intensities for each ROI
roi_max_raw = [
    np.max([np.max(experiment.raw[i_roi, i_trial][0]) for i_trial in range(n_trial)])
    for i_roi in range(n_roi)
]
roi_max_result = [
    np.max([np.max(experiment.result[i_roi, i_trial][0]) for i_trial in range(n_trial)])
    for i_roi in range(n_roi)
]
roi_max = np.maximum(roi_max_raw, roi_max_result)

# Plot our figure using subplot panels
plt.figure(figsize=(16, 2.5 * n_roi))
for i_roi in range(n_roi):
    for i_trial in range(n_trial):
        # Make subplot axes
        i_subplot = 1 + i_roi * n_trial + i_trial
        plt.subplot(n_roi, n_trial, i_subplot)
        # Plot the data
```

(continues on next page)

(continued from previous page)

```
        plt.plot(
            experiment.raw[i_roi][i_trial][0, :],
            label="Raw (suite2p)",
            color=cmap(i_roi * 2 % cmap.N),
        )
        plt.plot(
            experiment.result[i_roi][i_trial][0, :],
            label="FISSA",
            color=cmap((i_roi * 2 + 1) % cmap.N),
        )
        # Labels and boiler plate
        plt.ylim([-0.05 * roi_max[i_roi], roi_max[i_roi] * 1.05])
        if i_trial == 0:
            plt.ylabel(
                "ROI {}\n\nSignal intensity\n(candela per unit area)".format(i_roi)
            )
        if i_roi == 0:
            plt.title("Trial {}".format(i_trial + 1))
            plt.legend()

        if i_roi == n_roi - 1:
            plt.xlabel("Time (frame number)")
plt.show()
```


Trial 1
Trial 2
Trial 3
Raw (suite2p)
FISSA
ROI 0
ROI 1
ROI 2
ROI 3
ROI 4
ROI 5
ROI 6
ROI 7
ROI 8
ROI 9
ROI 10
ROI 11
ROI 12
ROI 13
ROI 14
ROI 15
ROI 16
ROI 17
ROI 18
ROI 19
ROI 20
Time (frame number)

The figure shows the raw signal from the ROI identified by suite2p (pale), and after decontaminating with FISSA (dark). The hues match the ROI locations drawn above. Each row shows the results from one of the ROIs detected by suite2p. Each column shows the results from one of the three trials.

Note that with the above settings for suite2p it seems to have detected more small local axon signals, instead of cells. This can possibly be improved with manual curation and suite2p setting changes, but as noted above these results should not be seen as indicative for either suite2p or FISSA due to the small dataset size.

Also note that the above Suite2P traces are done without suite2p's own neuropil removal algorithm.

2.4 Using FISSA with SIMA

SIMA is a toolbox for motion correction and cell detection. Here we illustrate how to create a workflow which uses SIMA to detect cells and FISSA to extract decontaminated signals from those cells.

Reference: Kaifosh, P., Zaremba, J. D., Danielson, N. B., Losonczy, A. SIMA: Python software for analysis of dynamic fluorescence imaging data. *Frontiers in neuroinformatics*, **8**(80), 2014. doi: 10.3389/fninf.2014.00080.

Please note that SIMA only supports Python 3.6 and below.

2.4.1 Import packages

[1]:
```
# FISSA toolbox
import fissa

# SIMA toolbox
import sima
import sima.segment

# File operations
import glob

# For plotting our results, use numpy and matplotlib
import matplotlib.pyplot as plt
import numpy as np
```

2.4.2 Detecting cells with SIMA

Setup data

[2]:
```
# Define folder where tiffs are present
tiff_folder = "exampleData/20150529/"

# Find tiffs in folder
tiffs = sorted(glob.glob(tiff_folder + "/*.tif*"))

# define motion correction method
mc_approach = sima.motion.DiscreteFourier2D()

# Define SIMA dataset
```

(continues on next page)

(continued from previous page)

```
sequences = [sima.Sequence.create("TIFF", tiff) for tiff in tiffs[:1]]
try:
    dataset = sima.ImagingDataset(sequences, "example.sima")
except BaseException:
    dataset = sima.ImagingDataset.load("example.sima")
```

Run SIMA segmentation algorithm

[3]:
```
stica_approach = sima.segment.STICA(components=2)
stica_approach.append(sima.segment.SparseROIsFromMasks())
stica_approach.append(sima.segment.SmoothROIBoundaries())
stica_approach.append(sima.segment.MergeOverlapping(threshold=0.5))
rois = dataset.segment(stica_approach, "auto_ROIs")
```

```
/opt/hostedtoolcache/Python/3.6.15/x64/lib/python3.6/site-packages/sima/ROI.py:242:␣
↪ShapelyDeprecationWarning: Iteration over multi-part geometries is deprecated and will␣
↪be removed in Shapely 2.0. Use the `geoms` property to access the constituent parts of␣
↪a multi-part geometry.
  for polygon in self.polygons:
/opt/hostedtoolcache/Python/3.6.15/x64/lib/python3.6/site-packages/sima/ROI.py:655:␣
↪ShapelyDeprecationWarning: __len__ for multi-part geometries is deprecated and will be␣
↪removed in Shapely 2.0. Check the length of the `geoms` property instead to get the ␣
↪number of parts of a multi-part geometry.
  if len(polygons) == 0:
/opt/hostedtoolcache/Python/3.6.15/x64/lib/python3.6/site-packages/sima/ROI.py:660:␣
↪ShapelyDeprecationWarning: __getitem__ for multi-part geometries is deprecated and␣
↪will be removed in Shapely 2.0. Use the `geoms` property to access the constituent␣
↪parts of a multi-part geometry.
  elif isinstance(polygons[0], Polygon):
/opt/hostedtoolcache/Python/3.6.15/x64/lib/python3.6/site-packages/sima/ROI.py:679:␣
↪ShapelyDeprecationWarning: Iteration over multi-part geometries is deprecated and will␣
↪be removed in Shapely 2.0. Use the `geoms` property to access the constituent parts of␣
↪a multi-part geometry.
  for poly in polygons:
/opt/hostedtoolcache/Python/3.6.15/x64/lib/python3.6/site-packages/sima/ROI.py:557:␣
↪ShapelyDeprecationWarning: Iteration over multi-part geometries is deprecated and will␣
↪be removed in Shapely 2.0. Use the `geoms` property to access the constituent parts of␣
↪a multi-part geometry.
  for poly in polygons:
```

Plot detected cells

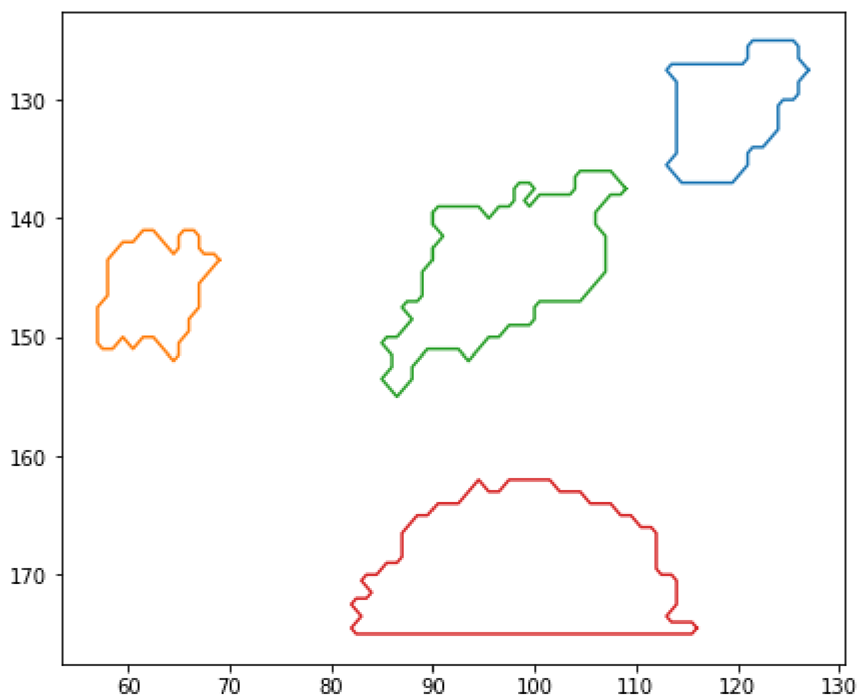
[4]:
```
# Plotting lines surrounding each of the ROIs
plt.figure(figsize=(7, 6))

for roi in rois:
    # Plot border around cell
    plt.plot(roi.coords[0][:, 0], roi.coords[0][:, 1])
```

(continues on next page)

(continued from previous page)

```
# Invert the y-axis because image co-ordinates are labelled from top-left
plt.gca().invert_yaxis()
plt.show()
```

```
/opt/hostedtoolcache/Python/3.6.15/x64/lib/python3.6/site-packages/sima/ROI.py:242:␣
↪ShapelyDeprecationWarning: Iteration over multi-part geometries is deprecated and will␣
↪be removed in Shapely 2.0. Use the `geoms` property to access the constituent parts of␣
↪a multi-part geometry.
  for polygon in self.polygons:
/opt/hostedtoolcache/Python/3.6.15/x64/lib/python3.6/site-packages/sima/ROI.py:242:␣
↪ShapelyDeprecationWarning: Iteration over multi-part geometries is deprecated and will␣
↪be removed in Shapely 2.0. Use the `geoms` property to access the constituent parts of␣
↪a multi-part geometry.
  for polygon in self.polygons:
/opt/hostedtoolcache/Python/3.6.15/x64/lib/python3.6/site-packages/sima/ROI.py:242:␣
↪ShapelyDeprecationWarning: Iteration over multi-part geometries is deprecated and will␣
↪be removed in Shapely 2.0. Use the `geoms` property to access the constituent parts of␣
↪a multi-part geometry.
  for polygon in self.polygons:
```



2.4.3 Extract decontaminated signals with FISSA

FISSA needs either ImageJ ROIs or numpy arrays as inputs for the ROIs.

SIMA outputs ROIs as numpy arrays, and can be directly read into FISSA.

A given roi is given as

```
rois[i].coords[0][:, :2]
```

FISSA expects rois to be provided as a list of lists

```
[[roiA1, roiA2, roiA3, ...]]
```

So some formatting will need to be done first.

[5]:
```
rois_fissa = [roi.coords[0][:, :2] for roi in rois]
```

```
/opt/hostedtoolcache/Python/3.6.15/x64/lib/python3.6/site-packages/sima/ROI.py:242:␣
↪ShapelyDeprecationWarning: Iteration over multi-part geometries is deprecated and will␣
↪be removed in Shapely 2.0. Use the `geoms` property to access the constituent parts of␣
↪a multi-part geometry.
  for polygon in self.polygons:
```

[6]:
```
rois[0].coords[0][:, :2].shape
```

[6]:
```
(26, 2)
```

We can then run FISSA on the data using the ROIs supplied by SIMA having converted them to a FISSA-compatible format, `rois_fissa`.

[7]:
```
output_folder = "fissa_sima_example"
experiment = fissa.Experiment(tiff_folder, [rois_fissa], output_folder)
experiment.separate()
```

```
Extracting traces:   0%|          | 0/3 [00:00<?, ?it/s]
```

```
/opt/hostedtoolcache/Python/3.6.15/x64/lib/python3.6/site-packages/fissa/polygons.py:73:␣
↪ShapelyDeprecationWarning: Iteration over multi-part geometries is deprecated and will␣
↪be removed in Shapely 2.0. Use the `geoms` property to access the constituent parts of␣
↪a multi-part geometry.
  for poly in polygons:
```

```
Finished extracting raw signals from 4 ROIs across 3 trials in 0.732 seconds.
Saving extracted traces to fissa_sima_example/prepared.npz
```

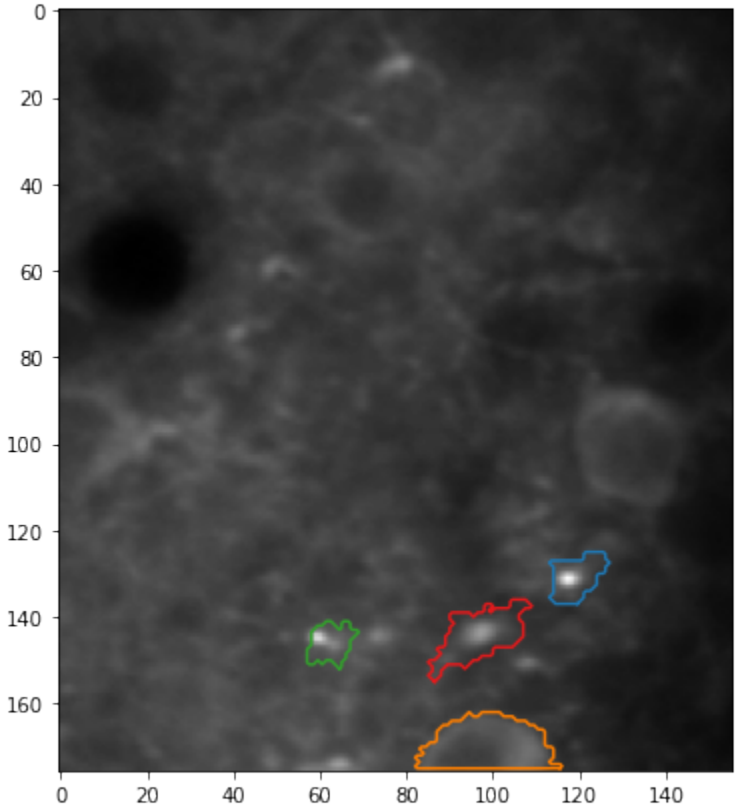
```
Separating data:   0%|          | 0/4 [00:00<?, ?it/s]
```

```
Finished separating signals from 4 ROIs across 3 trials in 1.735 seconds
Saving results to fissa_sima_example/separated.npz
```

Plotting the results

[8]:
```
# Fetch the colormap object for Cynthia Brewer's Paired color scheme
cmap = plt.get_cmap("Paired")
```

[9]:
```
# Select which trial (TIFF index) to plot
trial = 0

# Plot the mean image and ROIs from the FISSA experiment
plt.figure(figsize=(7, 7))
plt.imshow(experiment.means[trial], cmap="gray")

for i_roi in range(len(experiment.roi_polys)):
    # Plot border around ROI
    for contour in experiment.roi_polys[i_roi, trial][0]:
        plt.plot(
            contour[:, 1],
            contour[:, 0],
            color=cmap((i_roi * 2 + 1) % cmap.N),
        )

plt.show()
```

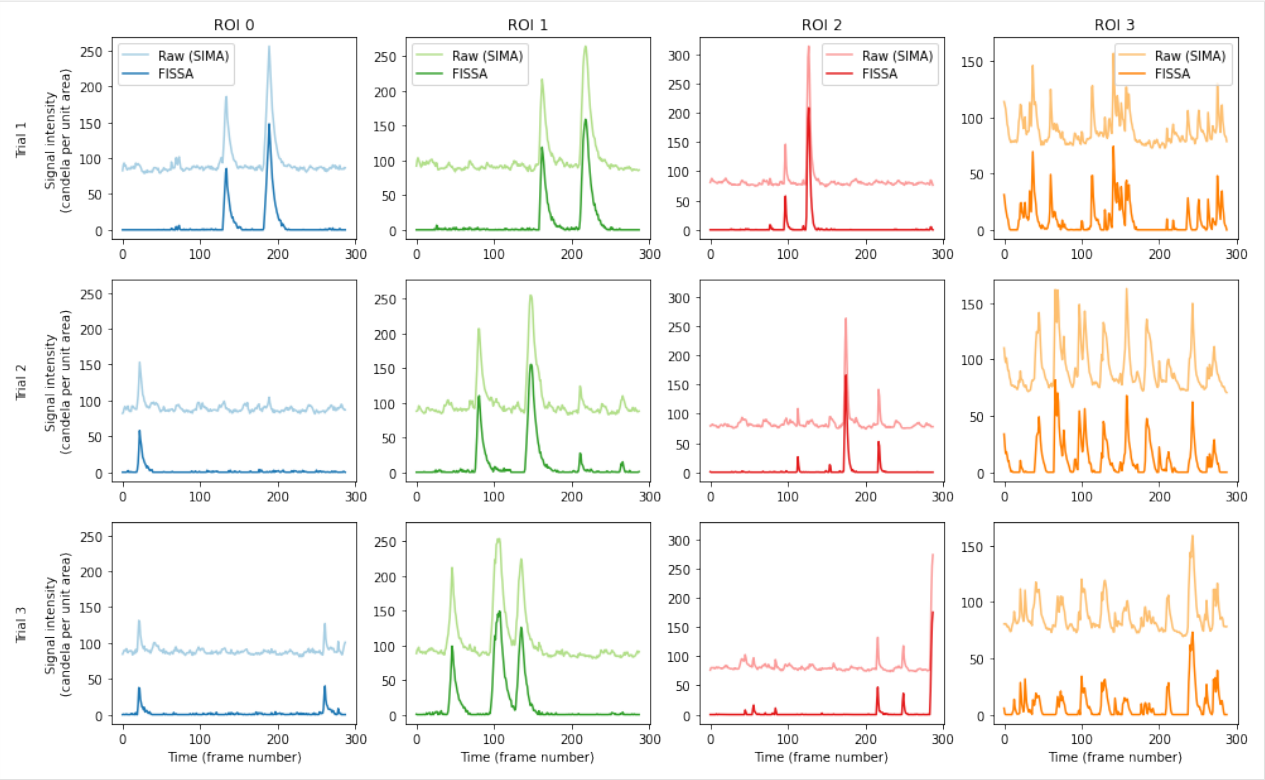


[10]:
```
# Plot all ROIs and trials

# Get the number of ROIs and trials
```

(continues on next page)

(continued from previous page)

```
n_roi = experiment.result.shape[0]
n_trial = experiment.result.shape[1]

# Find the maximum signal intensities for each ROI
roi_max_raw = [
    np.max([np.max(experiment.raw[i_roi, i_trial][0]) for i_trial in range(n_trial)])
    for i_roi in range(n_roi)
]
roi_max_result = [
    np.max([np.max(experiment.result[i_roi, i_trial][0]) for i_trial in range(n_trial)])
    for i_roi in range(n_roi)
]
roi_max = np.maximum(roi_max_raw, roi_max_result)

# Plot our figure using subplot panels
plt.figure(figsize=(16, 10))
for i_roi in range(n_roi):
    for i_trial in range(n_trial):
        # Make subplot axes
        i_subplot = 1 + i_trial * n_roi + i_roi
        plt.subplot(n_trial, n_roi, i_subplot)
        # Plot the data
        plt.plot(
            experiment.raw[i_roi][i_trial][0, :],
            label="Raw (SIMA)",
            color=cmap((i_roi * 2) % cmap.N),
        )
        plt.plot(
            experiment.result[i_roi][i_trial][0, :],
            label="FISSA",
            color=cmap((i_roi * 2 + 1) % cmap.N),
        )
        # Labels and boiler plate
        plt.ylim([-0.05 * roi_max[i_roi], roi_max[i_roi] * 1.05])
        if i_roi == 0:
            plt.ylabel(
                "Trial {}\n\nSignal intensity\n(candela per unit area)".format(
                    i_trial + 1
                )
            )
        if i_trial == 0:
            plt.legend()
            plt.title("ROI {}".format(i_roi))
        if i_trial == n_trial - 1:
            plt.xlabel("Time (frame number)")

plt.show()
```



The figure shows the raw signal from the ROI identified by SIMA (pale), and after decontaminating with FISSA (dark). The hues match the ROI locations drawn above. Each column shows the results from one of the ROIs detected by SIMA. Each row shows the results from one of the three trials.

2.5 Using FISSA with CNMF from MATLAB

CNMF is blind source separation toolbox for cell detection and signal extraction.

Here we illustrate how one can use the ROIs detected by CNMF, and use FISSA to extract and decontaminate the traces.

In this tutorial, we assume the user is using the MATLAB implementation of CNMF. As such, this also serves as a tutorial on how to import data from MATLAB into Python to use with FISSA.

However, note that there is also a Python implementation of CNMF, which you can use instead to keep your whole workflow in Python.

Reference: Pnevmatikakis, E.A., Soudry, D., Gao, Y., Machado, T., Merel, J., … and Paninski, L. Simultaneous denoising, deconvolution, and demixing of calcium imaging data. *Neuron*, **89**(2):285-299, 2016. doi: 10.1016/j.neuron.2015.11.037.

2.5.1 Import packages

[1]:
```
# FISSA package
import fissa

# For plotting our results, import numpy and matplotlib
import matplotlib.pyplot as plt
import numpy as np
```

[2]:
```
# Need this utility from scipy to load data from matfiles
from scipy.io import loadmat
```

Running CNMF in MATLAB, and importing into Python

We ran CNMF in MATLAB using the run_pipeline.m script available from the CNMF repository on our example data (found at ../exampleData/20150529/).

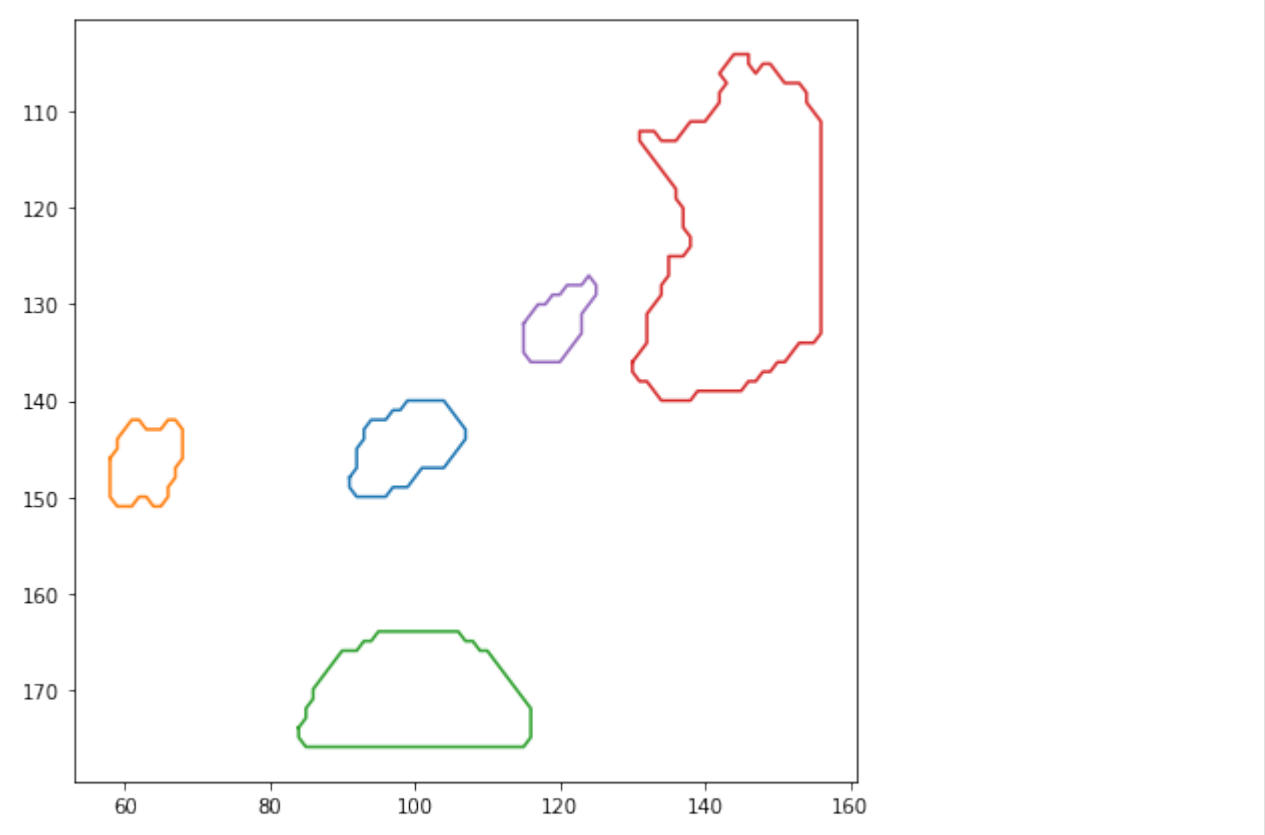
We saved the `Coor` and `F_df` variables generated by that script into a `.mat` file (cNMFdata.mat) which we now load here.

[3]:
```
# Load data from cNMFdata.mat file
cNMFdata = loadmat("cNMFdata")["dat"]

# Get the F_df recording traces out of the loaded object
F_df = cNMFdata["F_df"][0, 0]

# Get the ROI outlines out of the loaded object
Coor = cNMFdata["Coor"][0, 0]
```

2.5.2 Show detected cells

Let's render the ROIs using matplotlib.

[4]:
```
# Plotting lines surrounding each of the ROIs
plt.figure(figsize=(7, 7))

for i_cell in range(len(Coor)):
    x = Coor[i_cell, 0][0]
    y = Coor[i_cell, 0][1]
    # Plot border around cells
    plt.plot(x, y)

# Invert the y-axis because image co-ordinates are labelled from top-left
plt.gca().invert_yaxis()
plt.show()
```



Running FISSA on cells detected by CNMF

FISSA needs ROIs to be provided either as an ImageJ zip file, or a set of numpy arrays.

CNMF can output ROIs in coordinates (as we imported above), which can be directly read into FISSA. A given ROI after importing from MATLAB is given as

```
Coor[i, 0]
```

FISSA expects a set of rois to be given as a list of lists,

```
[[roiA1, roiA2, roiA3, ...]]
```

so we will need to change the format of the ROIs first.

[5]:
```
numROI = len(Coor)
rois_FISSA = [[Coor[i, 0][0], Coor[i, 0][1]] for i in range(numROI)]
```

Which can then be put into FISSA and run as follows.

[6]:
```
output_folder = "fissa_cnmf_example"
tiff_folder = "exampleData/20150529/"

experiment = fissa.Experiment(tiff_folder, [rois_FISSA], output_folder)
experiment.separate(redo_prep=True)
```

```
Loading data from cache fissa_cnmf_example/prepared.npz
Loading data from cache fissa_cnmf_example/separated.npz
```

```
Extracting traces:   0%|          | 0/3 [00:00<?, ?it/s]
```

```
Finished extracting raw signals from 5 ROIs across 3 trials in 0.726 seconds.
Saving extracted traces to fissa_cnmf_example/prepared.npz
```

```
Separating data:   0%|          | 0/5 [00:00<?, ?it/s]
```

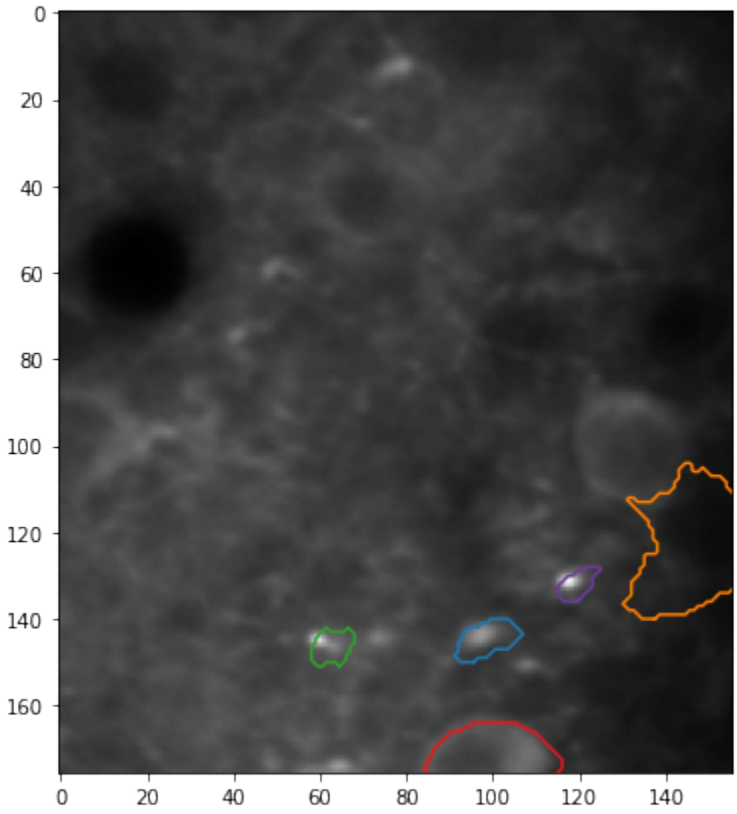
```
Finished separating signals from 5 ROIs across 3 trials in 1.706 seconds
Saving results to fissa_cnmf_example/separated.npz
```

Plotting the results

Let's plot the traces for ROIs as they were detected by CNMF, and after removing neuropile with FISSA.

[7]:
```
# Fetch the colormap object for Cynthia Brewer's Paired color scheme
cmap = plt.get_cmap("Paired")
```

[8]:
```
# Select which trial (TIFF index) to plot
trial = 0

# Plot the mean image and ROIs from the FISSA experiment
plt.figure(figsize=(7, 7))
plt.imshow(experiment.means[trial], cmap="gray")

XLIM = plt.xlim()
YLIM = plt.ylim()

for i_roi in range(len(experiment.roi_polys)):
    # Plot border around ROI
    for contour in experiment.roi_polys[i_roi, trial][0]:
        plt.plot(
            contour[:, 1],
            contour[:, 0],
            color=cmap((i_roi * 2 + 1) % cmap.N),
        )

# ROI co-ordinates are half a pixel outside the image,
# so we reset the axis limits
plt.xlim(XLIM)
plt.ylim(YLIM)
plt.show()
```

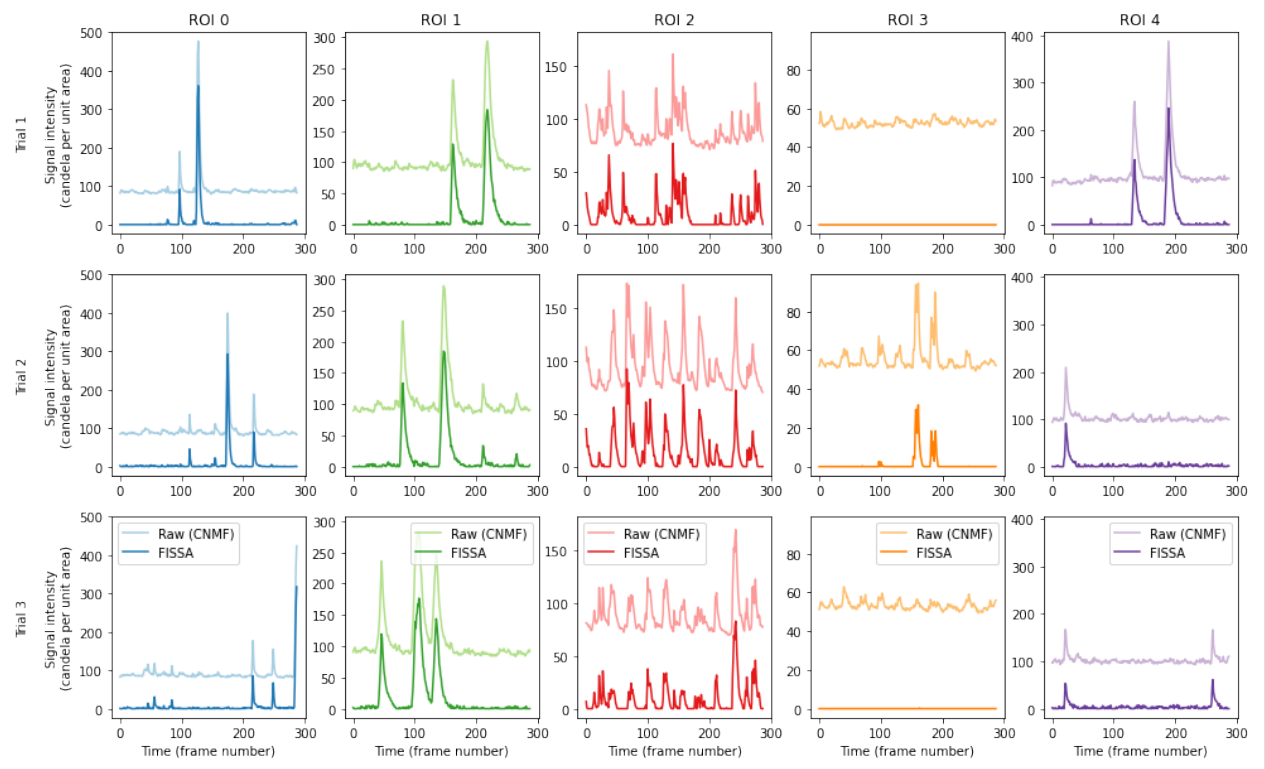


[9]:
```
# Plot all ROIs and trials

# Get the number of ROIs and trials
n_roi = experiment.result.shape[0]
n_trial = experiment.result.shape[1]

# Find the maximum signal intensities for each ROI
roi_max_raw = [
    np.max([np.max(experiment.raw[i_roi, i_trial][0]) for i_trial in range(n_trial)])
    for i_roi in range(n_roi)
]
roi_max_result = [
    np.max([np.max(experiment.result[i_roi, i_trial][0]) for i_trial in range(n_trial)])
    for i_roi in range(n_roi)
]
roi_max = np.maximum(roi_max_raw, roi_max_result)

# Plot our figure using subplot panels
plt.figure(figsize=(16, 10))
for i_roi in range(n_roi):
    for i_trial in range(n_trial):
        # Make subplot axes
        i_subplot = 1 + i_trial * n_roi + i_roi
        plt.subplot(n_trial, n_roi, i_subplot)
        # Plot the data
```

(continues on next page)

(continued from previous page)

```
        plt.plot(
            experiment.raw[i_roi][i_trial][0, :],
            label="Raw (CNMF)",
            color=cmap((i_roi * 2) % cmap.N),
        )
        plt.plot(
            experiment.result[i_roi][i_trial][0, :],
            label="FISSA",
            color=cmap((i_roi * 2 + 1) % cmap.N),
        )
        # Labels and boiler plate
        plt.ylim([-0.05 * roi_max[i_roi], roi_max[i_roi] * 1.05])
        if i_roi == 0:
            plt.ylabel(
                "Trial {}\n\nSignal intensity\n(candela per unit area)".format(
                    i_trial + 1
                )
            )
        if i_trial == 0:
            plt.title("ROI {}".format(i_roi))
        if i_trial == n_trial - 1:
            plt.xlabel("Time (frame number)")
            plt.legend()

plt.show()
```



The figure shows the raw signal from the ROI identified by CNMF (pale), and after decontaminating with FISSA (dark). The hues match the ROI locations drawn above. Each column shows the results from one of the ROIs detected

by CNMF. Each row shows the results from one of the three trials.

CHAPTER

THREE

API REFERENCE

3.1 fissa package

3.1.1 Submodules

fissa.core module

Main FISSA user interface.

Authors:

- Sander W Keemink <swkeemink@scimail.eu>
- Scott C Lowe <scott.code.lowe@gmail.com>

class fissa.core.**Experiment**(*images*, *rois*, *folder=None*, *nRegions=None*, *expansion=None*, *method=None*, *alpha=None*, *max_iter=None*, *tol=None*, *max_tries=None*, *ncores_preparation=- 1*, *ncores_separation=- 1*, *lowmemory_mode=False*, *datahandler=None*, *verbosity=1*)

Bases: `object`

Class-based interface for running FISSA on experiments.

Uses the methodology described in FISSA: A neuropil decontamination toolbox for calcium imaging signals.

Parameters

images [str or list] The raw imaging data. Should be one of:

- the path to a directory containing TIFF files (string),
- a list of paths to TIFF files (list of strings),
- a list of array_like data already loaded into memory, each shaped `(n_frames, height, width)`.

Note that each TIFF or array is considered a single trial.

rois [str or list] The region of interest (ROI) definitions. Should be one of:

- the path to a directory containing ImageJ ZIP files (string),
- the path of a single ImageJ ZIP file (string),
- a list of ImageJ ZIP files (list of strings),
- a list of arrays, each encoding a ROI polygons,
- a list of lists of binary arrays, each representing a ROI mask.

This can either be a single roiset for all trials, or a different roiset for each trial.

folder [str, optional] Path to a cache directory from which pre-extracted data will be loaded if present, and saved to otherwise. If *folder* is unset, the experiment data will not be saved.

nRegions [int, default=4] Number of neuropil regions and signals to use. Default is `4`. Use a higher number for densely labelled tissue.

expansion [float, default=1] Expansion factor for each neuropil region, relative to the ROI area. Default is `1`. The total neuropil area will be `nRegions * expansion * area(ROI)`.

method ["nmf" or "ica", default="nmf"] Which blind source-separation method to use. Either `"nmf"` for non-negative matrix factorization, or `"ica"` for independent component analysis. Default is `"nmf"` (recommended).

alpha [float, default=0.1] Sparsity regularizaton weight for NMF algorithm. Set to zero to remove regularization. Default is `0.1`.

max_iter [int, default=20000] Maximum number of iterations before timing out on an attempt.

New in version 1.0.0.

tol [float, default=1e-4] Tolerance of the stopping condition.

New in version 1.0.0.

max_tries [int, default=1] Maximum number of random initial states to try. Each random state will be optimized for *max_iter* iterations before timing out.

New in version 1.0.0.

ncores_preparation [int or None, default=-1] The number of parallel subprocesses to use during the data preparation steps of *separation_prep()*. These steps are ROI and neuropil subregion definitions, and extracting raw signals from TIFFs.

If set to `None` or `-1` (default), the number of processes used will equal the number of threads on the machine. If this is set to `-2`, the number of processes used will be one less than the number of threads on the machine; etc.

Note that the preparation process can be quite memory-intensive and it may be necessary to reduce the number of processes from the default.

ncores_separation [int or None, default=-1] The number of parallel subprocesses to use during the signal separation steps of *separate()*.

If set to `None` or `-1` (default), the number of processes used will equal the number of threads on the machine. If this is set to `-2`, the number of processes used will be one less than the number of threads on the machine; etc.

The separation routine requires less memory per process than the preparation routine, and so *ncores_separation* be often be set higher than *ncores_preparation*.

lowmemory_mode [bool, optional] If `True`, FISSA will load TIFF files into memory frame-by-frame instead of holding the entire TIFF in memory at once. This option reduces the memory load, and may be necessary for very large inputs. Default is `False`.

datahandler [*fissa.extraction.DataHandlerAbstract*, optional] A custom datahandler object for handling ROIs and calcium data can be given here. See *fissa.extraction* for example datahandler classes. The default datahandler is *DataHandlerTifffile*. If *datahandler* is set, the *lowmemory_mode* parameter is ignored.

verbosity [int, default=1] How verbose the processing will be. Increase for more output messages. Processing is silent if `verbosity=0`.

New in version 1.0.0.

Attributes

result [numpy.ndarray] A numpy.ndarray of shape `(n_rois, n_trials)`, each element of which is itself a numpy.ndarray shaped `(n_signals, n_timepoints)`.

The final output of FISSA, with separated signals ranked in order of their weighting toward the raw cell ROI signal relative to their weighting toward other mixed raw signals. The ordering is such that `experiment.result[roi, trial][0, :]` is the signal with highest score in its contribution to the raw neuronal signal. Subsequent signals are sorted in order of diminishing score. The units are same as *raw* (candelas per unit area).

This field is only populated after *separate()* has been run; until then, it is set to `None`.

roi_polys [numpy.ndarray] A numpy.ndarray of shape `(n_rois, n_trials)`, each element of which is itself a list of length `nRegions + 1`, each element of which is a list of length `n_contour` containing a numpy.ndarray of shape `(n_nodes, 2)`.

Polygon contours describing the outline of each region.

For contiguous ROIs, the outline of the `i_roi`-th ROI used in the `i_trial`-th trial is described by the array at `experiment.roi_polys[i_roi, i_trial][0][0]`. This array consists of `n_nodes` rows, each representing the coordinate of a node in `(y, x)` format. For non-contiguous ROIs, a contour is needed for each disconnected polygon making up the total aggregate ROI. These contours are found at `experiment.roi_polys[i_roi, i_trial][0][i_contour]`.

Similarly, the *nRegions* neuropil regions are each described by the polygons `experiment.roi_polys[i_roi, i_trial][i_neurpil + 1][i_contour]`, respectively.

means [list of *n_trials* numpy.ndarray, each shaped `(height, width)`] The temporal-mean image for each trial (i.e. for each TIFF file, the average image over all of its frames).

raw [numpy.ndarray] A numpy.ndarray of shape `(n_rois, n_trials)`, each element of which is itself a numpy.ndarray shaped `(n_signals, n_timepoints)`.

For each ROI and trial (`raw[i_roi, i_trial]`) we extract a temporal trace of the average value within the spatial area of each of the `nRegions + 1` regions. The 0-th region is the `i_roi`-th ROI (`raw[i_roi, i_trial][0]`). The subsequent `nRegions` vectors are the traces for each of the neuropil regions.

The units are the same as the supplied imagery (candelas per unit area).

sep [numpy.ndarray] A numpy.ndarray of shape `(n_rois, n_trials)`, each element of which is itself a numpy.ndarray shaped `(n_signals, n_timepoints)`.

The separated signals, before output signals are ranked according to their matching against the raw signal from within the ROI. Separated signal `i` for a specific ROI and trial can be found at `experiment.sep[roi, trial][i, :]`.

This field is only populated after *separate()* has been run; until then, it is set to `None`.

mixmat [numpy.ndarray] A numpy.ndarray of shape `(n_rois, n_trials)`, each element of which is itself a numpy.ndarray shaped `(n_rois, n_signals)`.

The mixing matrix, which maps from `experiment.raw` to `experiment.sep`. Because we use the collate the traces from all trials to determine separate the signals, the mixing matrices for a given ROI are the same across all trials. This means all `n_trials` elements in `mixmat[i_roi, :]` are identical.

This field is only populated after *separate()* has been run; until then, it is set to `None`.

info [numpy.ndarray shaped (n_rois, n_trials) of dicts] Information about the separation routine.

Each dictionary in the array has the following fields:

converged [bool] Whether the separation model converged, or if it ended due to reaching the maximum number of iterations.

iterations [int] The number of iterations which were needed for the separation model to converge.

max_iterations [int] Maximum number of iterations to use when fitting the separation model.

random_state [int or None] Random seed used to initialise the separation model.

This field is only populated after *separate()* has been run; until then, it is set to None.

deltaf_raw [numpy.ndarray] A numpy.ndarray of shape (n_rois, n_trials), each element of which is itself a numpy.ndarray shaped (1, n_timepoint).

The amount of change in fluorence relative to the baseline fluorence ($\Delta f/f_0$).

This field is only populated after *calc_deltaf()* has been run; until then, it is set to None.

Changed in version 1.0.0: The shape of the interior arrays changed from (n_timepoint,) to (1, n_timepoint).

deltaf_result [numpy.ndarray] A numpy.ndarray of shape (n_rois, n_trials), each element of which is itself a numpy.ndarray shaped (n_signals, n_timepoints).

The amount of change in fluorence relative to the baseline fluorence ($\Delta f/f_0$). By default, the baseline is taken from raw because the minimum values in result are typically zero. See *calc_deltaf()* for details.

This field is only populated after *calc_deltaf()* has been run; until then, it is set to None.

Methods

calc_deltaf(freq[, use_raw_f0, across_trials])	Calculate $\Delta f/f0$ for raw and result traces.
clear([verbosity])	Clear prepared data, and all data downstream of prepared data.
clear_separated([verbosity])	Clear separated data, and all data downstream of separated data.
load([path, force, skip_clear])	Load data from cache file in npz format.
save_prep([destination])	Save prepared raw signals, extracted from images, to an npz file.
save_separated([destination])	Save separated signals to an npz file.
save_to_matlab([fname])	Save the results to a MATLAB file.
separate([redo_prep, redo_sep])	Separate all the trials with FISSA algorithm.
separation_prep([redo])	Prepare and extract the data to be separated.
to_matfile([fname, legacy])	Save the results to a MATLAB file.

calc_deltaf(*freq*, *use_raw_f0=True*, *across_trials=True*)
Calculate $\Delta f/f0$ for raw and result traces.

The outputs are found in the deltaf_raw and deltaf_result attributes, which can be accessed at experiment.deltaf_raw and experiment.deltaf_result.

Parameters

freq [float] Imaging frequency, in Hz.

use_raw_f0 [bool, optional] If `True` (default), use an f0 estimate from the raw ROI trace for both raw and result traces. If `False`, use individual f0 estimates for each of the traces.

across_trials [bool, optional] If `True`, we estimate a single baseline f0 value across all trials. If `False`, each trial will have their own baseline f0, and $\Delta f/f_0$ value will be relative to the trial-specific f0. Default is `True`.

`clear`(*verbosity=None*)

Clear prepared data, and all data downstream of prepared data.

New in version 1.0.0.

Parameters

verbosity [int, optional] Whether to show the data fields which were cleared. By default, the object's `verbosity` attribute is used.

`clear_separated`(*verbosity=None*)

Clear separated data, and all data downstream of separated data.

New in version 1.0.0.

Parameters

verbosity [int, optional] Whether to show the data fields which were cleared. By default, the object's `verbosity` attribute is used.

`load`(*path=None*, *force=False*, *skip_clear=False*)

Load data from cache file in npz format.

New in version 1.0.0.

Parameters

path [str, optional] Path to cache file (.npz format) or a directory containing `"prepared.npz"` and/or `"separated.npz"` files. Default behaviour is to use the `folder` parameter which was provided when the object was initialised is used (`experiment.folder`).

force [bool, optional] Whether to load the cache even if its experiment parameters differ from the properties of this experiment. Default is `False`.

skip_clear [bool, optional] Whether to skip clearing values before loading. Default is `False`.

`property nCell`

`property nTrials`

`save_prep`(*destination=None*)

Save prepared raw signals, extracted from images, to an npz file.

New in version 1.0.0.

Parameters

destination [str, optional] Path to output file. The default destination is `"prepared.npz"` within the cache directory `experiment.folder`.

`save_separated`(*destination=None*)

Save separated signals to an npz file.

New in version 1.0.0.

Parameters

destination [str, optional] Path to output file. The default destination is `"separated.npz"` within the cache directory `experiment.folder`.

`save_to_matlab`(*fname=None*)

Save the results to a MATLAB file.

Deprecated since version 1.0.0: Use `experiment.to_matfile(legacy=True)` instead.

This will generate a .mat file which can be loaded into MATLAB to provide structs: ROIs, result, raw.

If $\Delta f/f_0$ was calculated, these will also be stored as `df_result` and `df_raw`, which will have the same format as `result` and `raw`.

These can be interfaced with as follows, for ROI 0, trial 0:

`ROIs.cell0.trial0{1}` Polygon outlining the ROI.

`ROIs.cell0.trial0{2}` Polygon outlining the first (of `nRegions`) neuropil region.

`result.cell0.trial0(1, :)` Final extracted neuronal signal.

`result.cell0.trial0(2, :)` Contaminating signal.

`raw.cell0.trial0(1, :)` Raw measured cell signal, average over the ROI.

`raw.cell0.trial0(2, :)` Raw signal from first (of `nRegions`) neuropil region.

Parameters

fname [str, optional] Destination for output file. Default is a file named `"matlab.mat"` within the cache save directory for the experiment (the *folder* argument when the `Experiment` instance was created).

See also:

`Experiment.to_matfile`

`separate`(*redo_prep=False*, *redo_sep=False*)

Separate all the trials with FISSA algorithm.

After running `separate`, data can be found as follows:

experiment.sep Raw separation output, without being matched. Signal `i` for a specific ROI and trial can be found as `experiment.sep[roi, trial][i, :]`.

experiment.result Final output, in order of presence in the ROI. Signal `i` for a specific ROI and trial can be found at `experiment.result[roi, trial][i, :]`. Note that the ordering is such that `i = 0` is the signal most strongly present in the ROI, and subsequent entries are in diminishing order.

experiment.mixmat The mixing matrix, which maps from `experiment.raw` to `experiment.sep`.

experiment.info Information about separation routine, iterations needed, etc.

Parameters

redo_prep [bool, optional] Whether to redo the preparation. Default is `False`. Note that if this is true, we set `redo_sep = True` as well.

redo_sep [bool, optional] Whether to redo the separation. Default is `False`. Note that this parameter is ignored if *redo_prep* is set to `True`.

separation_prep(*redo=False*)

Prepare and extract the data to be separated.

For each trial, performs the following steps:

- Load in data as arrays.
- Load in ROIs as masks.
- Grow and seaparate ROIs to define neuropil regions.
- Using neuropil and original ROI regions, extract traces from data.

After running this you can access the raw data (i.e. pre-separation) as `experiment.raw` and `experiment.rois`. `experiment.raw` is a list of arrays. `experiment.raw[roi, trial]` gives you the traces of a specific ROI and trial, across the ROI and neuropil regions. `experiment.roi_polys` is a list of lists of arrays. `experiment.roi_polys[roi, trial][region][0]` gives you the polygon for the region for a specific ROI, trial and region. `region=0` is the ROI itself (i.e. the outline of the neuron cell), and `region>0` gives the different neuropil regions. For separable masks, it is possible multiple outlines are found, which can be accessed as `experiment.roi_polys[roi, trial][region][i]`, where `i` is the outline index.

Parameters

redo [bool, optional] If `False`, we load previously prepared data when possible. If `True`, we re-run the preparation, even if it has previously been run. Default is `False`.

to_matfile(*fname=None*, *legacy=False*)

Save the results to a MATLAB file.

New in version 1.0.0.

This will generate a MAT-file (.mat) which can be loaded into MATLAB. The MAT-file contains structs for all the experiment output attributes (`roi_polys`, `result`, `raw`, etc.) and analysis parameters (`expansion`, `nRegions`, `alpha`, etc.). If $\Delta f/f_0$ was calculated with *calc_deltaf()*, `deltaf_result` and `deltaf_raw` are also included.

These can be interfaced with as illustrated below.

result{1, 1}(1, :) The separated signal for the first ROI and first trial. This is equivalent to `experiment.result[0, 0][0, :]` when interacting with the *Experiment* object in Python.

result{roi, trial}(1, :) The separated signal for the `roi`-th ROI and `trial`-th trial. This is equivalent to `experiment.result[roi - 1, trial - 1][0, :]` when interacting with the *Experiment* object in Python.

result{roi, trial}(2, :) A contaminating signal.

raw{roi, trial}(1, :) Raw measured neuronal signal, averaged over the ROI. This is equivalent to `experiment.raw[roi - 1, trial - 1][0, :]` when interacting with the *Experiment* object in Python.

raw{roi, trial}(2, :) Raw signal from first neuropil region (of `nRegions`).

roi_polys{roi, trial}{1} Polygon outlining the ROI, as an n-by-2 array of coordinates.

roi_polys{roi, trial}{2} Polygon outlining the first neuropil region (of `nRegions`), as an n-by-2 array of coordinates.

Parameters

fname [str, optional] Destination for output file. The default is a file named `"separated.mat"` within the cache save directory for the experiment (the `folder` argument when the *Experiment* instance was created).

legacy [bool, default=False] Whether to use the legacy format of *save_to_matlab()*. This also changes the default output name to "matlab.mat".

Examples

Here are some example MATLAB plots.

Plotting raw and decontaminated traces:

```
% Load the FISSA output data
S = load('separated.mat')
% Separated signal for the third ROI, second trial
roi = 3; trial = 2;
% Plot the raw and result traces for the ROI signal
figure; hold on;
plot(S.raw{roi, trial}(1, :));
plot(S.result{roi, trial}(1, :));
title(sprintf('ROI %d, Trial %d', roi, trial));
xlabel('Time (frame number)');
ylabel('Signal intensity (candela per unit area)');
legend({'Raw', 'Result'});
```

If all ROIs are contiguous and described by a single contour, the the mean image and ROI locations for one trial can be plotted as follows:

```
% Load the FISSA output data
S = load('separated.mat')
trial = 1;
figure; hold on;
% Plot the mean image
imagesc(squeeze(S.means(trial, :, :)));
colormap('gray');
% Plot ROI locations
for i_roi = 1:size(S.result, 1);
    contour = S.roi_polys{i_roi, trial}{1};
    plot(contour(:, 2), contour(:, 1));
end
set(gca, 'YDir', 'reverse');
```

fissa.core.**extract**(*image*, *rois*, *nRegions=4*, *expansion=1*, *datahandler=None*, *verbosity=1*, *label=None*, *total=None*)

Extract data for all ROIs in a single 3d array or TIFF file.

New in version 1.0.0.

Parameters

image [str or array_like shaped (time, height, width)] The imaging data. Either a path to a multipage TIFF file, or 3d array_like data.

rois [str or list of array_like] The regions-of-interest, specified by either a string containing a path to an ImageJ roi zip file, or a list of arrays encoding polygons, or list of binary arrays representing masks.

nRegions [int, default=4] Number of neuropil regions to draw. Use a higher number for densely labelled tissue. Default is 4.

expansion [float, default=1] Expansion factor for the neuropil region, relative to the ROI area. Default is 1. The total neuropil area will be `nRegions * expansion * area(ROI)`.

datahandler [fissa.extraction.DataHandlerAbstract, optional] A datahandler object for handling ROIs and calcium data. The default is *DataHandlerTifffile*.

verbosity [int, default=1] Level of verbosity. The options are:

- `0`: No outputs.
- `1`: Print extraction start.
- `2`: Print extraction end.
- `3`: Print start of each step within the extraction process.

label [str or int, optional] The label for the current trial. Only used for reporting progress.

total [int, optional] Total number of trials. Only used for reporting progress.

Returns

traces [numpy.ndarray shaped `(n_rois, nRegions + 1, n_frames)`] The raw signal, determined as the average fluorence trace extracted from each ROI and neuropil region.

Each vector `traces[i_roi, 0, :]` contains the traces for the `i_roi`-th ROI. The following *nRegions* arrays in `traces[i_roi, 1 : nRegions + 1, :]` contain the traces from the *nRegions* grown neuropil regions surrounding the `i_roi`-th ROI.

polys [list of list of list of numpy.ndarray shaped `(n_nodes, 2)`] Polygon contours describing the outline of each region.

For contiguous ROIs, the outline of the `i_roi`-th ROI is described by the array at `polys[i_roi][0][0]`. This array is `n_nodes` rows, each representing the coordinate of a node in `(y, x)` format. For non-contiguous ROIs, a contour is needed for each disconnected polygon making up the total aggregate ROI. These contours are found at `polys[i_roi][0][i_contour]`.

Similarly, the *nRegions* neuropil regions are each described by the polygons `polys[i_roi][i_neurpil + 1][i_contour]` respectively.

mean [numpy.ndarray shaped (height, width)] Mean image.

`fissa.core.`**`run_fissa`**(*images*, *rois*, *folder=None*, *freq=None*, *return_deltaf=False*, *deltaf_across_trials=True*, *export_to_matfile=False*, ***kwargs*)

Function-based interface to run FISSA on an experiment.

New in version 1.0.0.

Uses the methodology described in FISSA: A neuropil decontamination toolbox for calcium imaging signals.

Parameters

images [str or list] The raw recording data. Should be one of:

- the path to a directory containing TIFF files (string),
- a list of paths to TIFF files (list of strings),
- a list of array_like data already loaded into memory, each shaped `(n_frames, height, width)`.

Note that each TIFF/array is considered a single trial.

rois [str or list] The roi definitions. Should be one of:

- the path to a directory containing ImageJ ZIP files (string),

- the path of a single ImageJ ZIP file (string),
- a list of ImageJ ZIP files (list of strings),
- a list of arrays, each encoding a ROI polygons,
- a list of lists of binary arrays, each representing a ROI mask.

This can either be a single roiset for all trials, or a different roiset for each trial.

folder [str, optional] Path to a cache directory from which pre-extracted data will be loaded if present, and saved to otherwise. If *folder* is unset, the experiment data will not be saved.

freq [float, optional] Imaging frequency, in Hz. Required if `return_deltaf=True`.

return_deltaf [bool, optional] Whether to return $\Delta f/f_0$. Otherwise, the decontaminated signal is returned scaled against the raw recording. Default is `False`.

deltaf_across_trials [bool, default=True] If `True`, we estimate a single baseline f0 value across all trials when computing $\Delta f/f_0$. If `False`, each trial will have their own baseline f0, and $\Delta f/f_0$ value will be relative to the trial-specific f0. Default is `True`.

export_to_matfile [bool or str or None, default=False] Whether to export the data to a MATLAB-compatible .mat file. If *export_to_matfile* is a string, it is used as the path to the output file. If `export_to_matfile=True`, the matfile is saved to the default path of `"separated.mat"` within the *folder* directory, and *folder* must be set. If this is `None`, the matfile is exported to the default path if *folder* is set, and otherwise is not exported. Default is `False`.

****kwargs** Additional keyword arguments as per *Experiment*.

Returns

result [2d numpy.ndarray of 2d numpy.ndarrays of np.float64] The vector `result[roi, trial][0, :]` is the trace from ROI `roi` in trial `trial`. If `return_deltaf=True`, this is $\Delta f/f_0$; otherwise, it is the decontaminated signal scaled as per the raw signal. f_0 is the baseline as calculated from the raw signal.

raw [2d numpy.ndarray of 2d numpy.ndarrays of np.float64] The raw traces without separation. The vector `raw[c, t][0, :]` is the ROI trace from cell `c` in trial `t`. The vector `raw[c, t][i, :]` for i>=1 the trace from cell `c` in trial `t`, from neuropil region `i-1`. If `return_deltaf=True`, this is $\Delta f/f_0$; otherwise it's the raw extracted signal.

See also:

fissa.core.Experiment

fissa.core.**separate_trials**(*raw*, *alpha=0.1*, *max_iter=20000*, *tol=0.0001*, *max_tries=1*, *method='nmf'*, *verbosity=1*, *label=None*, *total=None*)

Separate signals within a set of 2d arrays.

New in version 1.0.0.

Parameters

raw [list of n_trials array_like, each shaped `(nRegions + 1, observations)`] Raw signals. A list of 2-d arrays, each of which contains observations of mixed signals, mixed in the same way across all trials. The *nRegions* signals must be the same for each trial, and the 0-th region, `raw[trial][0]`, should be from the region of interest for which a matching source signal should be identified.

alpha [float, default=0.1] Sparsity regularizaton weight for NMF algorithm. Set to zero to remove regularization. Default is `0.1`. (Only used for `method="nmf"`.)

max_iter [int, default=20000] Maximum number of iterations before timing out on an attempt.

tol [float, default=1e-4] Tolerance of the stopping condition.

max_tries [int, default=1] Maximum number of random initial states to try. Each random state will be optimized for *max_iter* iterations before timing out.

method [{"nmf", "ica"}, default="nmf"] Which blind source-separation method to use. Either `"nmf"` for non-negative matrix factorization, or `"ica"` for independent component analysis. Default is `"nmf"`.

verbosity [int, default=1] Level of verbosity. The options are:

- `0`: No outputs.
- `1`: Print separation start.
- `2`: Print separation end.
- `3`: Print progress details during separation.

label [str or int, optional] Label/name or index of the ROI currently being processed. Only used for progress messages.

total [int, optional] Total number of ROIs. Only used for reporting progress.

Returns

Xsep [list of n_trials `numpy.ndarray`, each shaped `(nRegions + 1, observations)`] The separated signals, unordered.

Xmatch [list of n_trials `numpy.ndarray`, each shaped `(nRegions + 1, observations)`] The separated traces, ordered by matching score against the raw ROI signal.

Xmixmat [`numpy.ndarray`, shaped `(nRegions + 1, nRegions + 1)`] Mixing matrix.

convergence [dict] Metadata for the convergence result, with the following keys and values:

converged [bool] Whether the separation model converged, or if it ended due to reaching the maximum number of iterations.

iterations [int] The number of iterations which were needed for the separation model to converge.

max_iterations [int] Maximum number of iterations to use when fitting the separation model.

random_state [int or None] Random seed used to initialise the separation model.

fissa.deltaf module

Compute changes in fluorescence signals relative to baseline activity.

Authors:

- Scott C Lowe <scott.code.lowe@gmail.com>

`fissa.deltaf.`**`findBaselineF0`**(*rawF*, *fs*, *axis=0*, *keepdims=False*)

Find the baseline for a fluorescence imaging trace line.

The baseline, F0, is the 5th-percentile of the 1Hz lowpass filtered signal.

Parameters

rawF [array_like] Raw fluorescence signal.

fs [float] Sampling frequency of *rawF*, in Hz.

axis [int, default=0] Dimension which contains the time series. Default is `0`.

keepdims [bool, optional] Whether to preserve the dimensionality of the input. Default is `False`.

Returns

baselineF0 [numpy.ndarray] The baseline fluorescence of each recording, as an array.

fissa.extraction module

DataHandler classes to handle and manipulate image and ROI objects.

Authors:

- Sander W Keemink <swkeemink@scimail.eu>
- Scott C Lowe <scott.code.lowe@gmail.com>

class fissa.extraction.**DataHandlerAbstract**

Bases: abc.ABC

Abstract class for a data handler.

See also:

DataHandlerTifffile, *DataHandlerPillow*

Methods

extracttraces(masks)	Extract from data the average signal within each mask, across time.
get_frame_size()	Determine the shape of each frame within the recording.
getmean()	Determine the mean image across all frames.
image2array()	Load data (from a path) as an array, or similar internal structure.
rois2masks(rois, data)	Convert ROIs into a collection of binary masks.

abstract extracttraces(*masks*)

Extract from data the average signal within each mask, across time.

Must return a 2D numpy.ndarray.

Parameters

data [data_type] The same object as returned by *image2array()*.

masks [mask_type] The same object as returned by *rois2masks()*.

Returns

traces [numpy.ndarray] Trace for each mask, shaped `(len(masks), n_frames)`.

abstract get_frame_size()

Determine the shape of each frame within the recording.

Parameters

data [data_type] The same object as returned by *image2array()*.

Returns

shape [tuple of ints] The 2D, y-by-x, shape of each frame in the movie.

abstract getmean()

Determine the mean image across all frames.

Must return a 2D numpy.ndarray.

Parameters

data [data_type] The same object as returned by *image2array()*.

Returns

mean [numpy.ndarray] Mean image as a 2D, y-by-x, array.

abstract image2array()

Load data (from a path) as an array, or similar internal structure.

Parameters

image [image_type] Some handle to, or representation of, the raw imagery data.

Returns

data [data_type] Internal representation of the images which will be used by all the other methods in this class.

classmethod rois2masks(*rois*, *data*)

Convert ROIs into a collection of binary masks.

Parameters

rois [str or list of array_like] Either a string containing a path to an ImageJ roi zip file, or a list of arrays encoding polygons, or list of binary arrays representing masks.

data [data_type] The same object as returned by *image2array()*.

Returns

masks [mask_type] Masks, in a format accepted by *extracttraces()*.

See also:

fissa.roitools.getmasks, *fissa.roitools.readrois*

class fissa.extraction.**DataHandlerPillow**

Bases: *fissa.extraction.DataHandlerAbstract*

Extract data from TIFF images frame-by-frame using Pillow (PIL.Image).

Slower, but less memory-intensive than *DataHandlerTifffile*.

Methods

extracttraces(data, masks)	Extract the average signal within each mask across the data.
get_frame_size(data)	Determine the shape of each frame within the recording.
getmean(data)	Determine the mean image across all frames.
image2array(image)	Open an image file as a `PIL.Image` instance.
`rois2masks`(rois, data)	Convert ROIs into a collection of binary masks.

static extracttraces(*data*, *masks*)
Extract the average signal within each mask across the data.

Parameters

data [PIL.Image] An open `PIL.Image` handle to a multi-frame TIFF image.

masks [list of array_like] List of binary arrays.

Returns

traces [numpy.ndarray] Trace for each mask, shaped `(len(masks), n_frames)`.

static get_frame_size(*data*)
Determine the shape of each frame within the recording.

Parameters

data [PIL.Image] An open `PIL.Image` handle to a multi-frame TIFF image.

Returns

shape [tuple of ints] The 2D, y-by-x, shape of each frame in the movie.

static getmean(*data*)
Determine the mean image across all frames.

Parameters

data [PIL.Image] An open `PIL.Image` handle to a multi-frame TIFF image.

Returns

mean [numpy.ndarray] y-by-x array for the mean values.

static image2array(*image*)
Open an image file as a `PIL.Image` instance.

Parameters

image [str or file] A filename (string) of a TIFF image file, a `pathlib.Path` object, or a file object.

Returns

data [PIL.Image] Handle from which frames can be loaded.

class fissa.extraction.**DataHandlerTifffile**
Bases: *fissa.extraction.DataHandlerAbstract*

Extract data from TIFF images using tifffile.

Methods

extracttraces(data, masks)	Extract a temporal trace for each spatial mask.
get_frame_size(data)	Determine the shape of each frame within the recording.
getmean(data)	Determine the mean image across all frames.
image2array(image)	Load a TIFF image from disk.
rois2masks(rois, data)	Convert ROIs into a collection of binary masks.

static extracttraces(*data*, *masks*)

Extract a temporal trace for each spatial mask.

Parameters

data [array_like] Data array as made by *image2array()*, shaped `(frames, y, x)`.

masks [`list` of array_like] List of binary arrays.

Returns

traces [numpy.ndarray] Trace for each mask, shaped `(len(masks), n_frames)`.

static get_frame_size(*data*)

Determine the shape of each frame within the recording.

Parameters

data [data_type] The same object as returned by *image2array()*.

Returns

shape [tuple of ints] The 2D, y-by-x, shape of each frame in the movie.

static getmean(*data*)

Determine the mean image across all frames.

Parameters

data [array_like] Data array as made by *image2array()*, shaped `(frames, y, x)`.

Returns

numpy.ndarray y by x array for the mean values

static image2array(*image*)

Load a TIFF image from disk.

Parameters

image [str or array_like shaped (time, height, width)] Either a path to a TIFF file, or array_like data.

Returns

numpy.ndarray A 3D array containing the data, with dimensions corresponding to `(frames, y_coordinate, x_coordinate)`.

class fissa.extraction.**DataHandlerTifffileLazy**

Bases: *fissa.extraction.DataHandlerAbstract*

Extract data from TIFF images using tifffile, with lazy loading.

Methods

extracttraces(data, masks)	Extract a temporal trace for each spatial mask.
get_frame_size(data)	Determine the shape of each frame within the recording.
getmean(data)	Determine the mean image across all frames.
image2array(image)	Load a TIFF image from disk.
rois2masks(rois, data)	Convert ROIs into a collection of binary masks.

static extracttraces(*data*, *masks*)
Extract a temporal trace for each spatial mask.

Parameters

data [tiffile.TiffFile] Open tiffile.TiffFile object.

masks [list of array_like] List of binary arrays.

Returns

traces [numpy.ndarray] Trace for each mask, shaped `(len(masks), n_frames)`.

static get_frame_size(*data*)
Determine the shape of each frame within the recording.

Parameters

data [data_type] The same object as returned by *image2array()*.

Returns

shape [tuple of ints] The 2D, y-by-x, shape of each frame in the movie.

static getmean(*data*)
Determine the mean image across all frames.

Parameters

data [tiffile.TiffFile] Open tiffile.TiffFile object.

Returns

numpy.ndarray y by x array for the mean values

static image2array(*image*)
Load a TIFF image from disk.

Parameters

image [str] A path to a TIFF file.

Returns

data [tiffile.TiffFile] Open tiffile.TiffFile object.

fissa.neuropil module

Functions for removal of neuropil from calcium signals.

Authors:

- Sander W Keemink <swkeemink@scimail.eu>
- Scott C Lowe <scott.code.lowe@gmail.com>

Created: 2015-05-15

`fissa.neuropil.`**`separate`**(*S*, *sep_method='nmf'*, *n=None*, *max_iter=10000*, *tol=0.0001*, *random_state=892*, *max_tries=10*, *W0=None*, *H0=None*, *alpha=0.1*, *verbosity=1*, *prefix=''*)

Find independent signals, sorted by matching score against the first input signal.

Parameters

S [array_like shaped (signals, observations)] 2-d array containing mixed input signals. Each column of *S* should be a different signal, and each row an observation of the signals. For `S[i, j]`, `j` is a signal, and `i` is an observation. The first column, `j = 0`, is considered the primary signal and the one for which we will try to extract a decontaminated equivalent.

sep_method [{"ica", "nmf"}] Which source separation method to use, either ICA or NMF.

- `"ica"`: Independent Component Analysis
- `"nmf"`: Non-negative Matrix Factorization

n [int, optional] How many components to estimate. If `None` (default), for the NMF method, `n` is the number of input signals; for the ICA method, we use PCA to estimate how many components would explain at least 99% of the variance and adopt this value for `n`.

max_iter [int, default=10000] Maximum number of iterations before timing out on an attempt.

Changed in version 1.0.0: Argument *maxiter* renamed to *max_iter*.

tol [float, default=1e-4] Tolerance of the stopping condition.

random_state [int or None, default=892] Initial state for the random number generator. Set to `None` to use the `numpy.random` default. Default seed is `892`.

max_tries [int, default=10] Maximum number of random initial states to try. Each random state will be optimized for *max_iter* iterations before timing out.

Changed in version 1.0.0: Argument *maxtries* renamed to *max_tries*.

W0 [array_like, optional] Optional starting condition for `W` in NMF algorithm. (Ignored when using the ICA method.)

H0 [array_like, optional] Optional starting condition for `H` in NMF algorithm. (Ignored when using the ICA method.)

alpha [float, default=0.1] Sparsity regularizaton weight for NMF algorithm. Set to zero to remove regularization. Default is `0.1`. (Ignored when using the ICA method.)

verbosity [int, default=1] Level of verbosity. The options are:

- `0`: No outputs.
- `1`: Print separation progress.

prefix [str, optional] String to include before any progress statements.

Returns

S_sep [`numpy.ndarray` shaped (signals, observations)] The raw separated traces.

S_matched [numpy.ndarray shaped (signals, observations)] The separated traces matched to the primary signal, in order of matching quality (see Notes below).

A_sep [numpy.ndarray shaped (signals, signals)] Mixing matrix.

convergence [dict] Metadata for the convergence result, with the following keys and values:

convergence["random_state"] Seed for estimator initiation.

convergence["iterations"] Number of iterations needed for convergence.

convergence["max_iterations"] Maximum number of iterations allowed.

convergence["converged"] Whether the algorithm converged or not (bool).

See also:

sklearn.decomposition.NMF, sklearn.decomposition.FastICA

Notes

To identify which independent signal matches the primary signal best, we first normalize the columns in the output mixing matrix *A* such that `sum(A[:, separated]) = 1`. This results in a relative score of how strongly each raw signal contributes to each separated signal. From this, we find the separated signal to which the ROI trace makes the largest (relative) contribution.

fissa.polygons module

Polygon tools.

Changed in version 1.0.0: Module renamed from `fissa.ROI` to `fissa.polygons`.

The functions below were adapted from the sima package http://www.losonczylab.org/sima, version 1.3.0. https://github.com/losonczylab/sima/blob/1.3.0/sima/ROI.py

License

This file is Copyright (C) 2014 The Trustees of Columbia University in the City of New York.

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

`fissa.polygons.`**poly2mask**(*polygons*, *im_size*)

Convert polygons to a sparse binary mask.

```
>>> from fissa.polygons import poly2mask
>>> poly1 = [[0, 0], [0, 1], [1, 1], [1, 0]]
>>> poly2 = [[0, 1], [0, 2], [2, 2], [2, 1]]
>>> mask = poly2mask([poly1, poly2], (3, 3))
>>> mask[0].todense()
matrix([[ True, False, False],
```

(continues on next page)

(continued from previous page)

```
        [ True,  True, False],
        [False, False, False]], dtype=bool)
```

Parameters

polygons [sequence of coordinates or sequence of Polygons] A sequence of polygons where each is either a sequence of `(x, y)` or `(x, y, z)` coordinate pairs, an Nx2 or Nx3 numpy array, or a Polygon object.

im_size [tuple] Final size of the resulting mask.

Returns

masks [list of sparse matrices] A list of sparse binary masks of the points contained within the polygons, one mask per plane. Each mask is in linked list sparse matrix format.

fissa.readimagejrois module

Tools for reading ImageJ files.

Based on code originally written by Luis Pedro Coelho <luis@luispedro.org>, 2012, available at https://gist.github.com/luispedro/3437255, distributed under the MIT License.

Modified

- 2014 by Jeffrey Zaremba (@jzaremba), https://github.com/losonczylab/sima
- 2015 by Scott Lowe (@scottclowe) and Sander Keemink (@swkeemink).

`fissa.readimagejrois.`**`parse_roi_file`**`(`*roi_source*`)`

Parse an individual ImageJ ROI.

Parameters

roi_source [str or file object] Path to file, or file object containing a single ImageJ ROI.

Returns

dict Returns a parsed ROI object, a dictionary with either a `"polygons"` or a `"mask"` field.

Raises

IOError If there is an error reading the roi file object.

ValueError If unable to parse ROI.

`fissa.readimagejrois.`**`read_imagej_roi_zip`**`(`*filename*`)`

Read an ImageJ ROI zip set and parse each ROI individually.

Parameters

filename [str] Path to the ImageJ ROIs zip file.

Returns

roi_list [list] List of the parsed ImageJ ROIs.

fissa.roitools module

Functions used for ROI manipulation.

Authors:

- Sander W Keemink <swkeemink@scimail.eu>

fissa.roitools.**find_roi_edge**(*mask*)
Find the outline of a mask.

Uses skimage.measure.find_contours().

Parameters

mask [array_like] The mask, as a binary array.

Returns

outline [list of (n,2)-ndarray] Array with coordinates of pixels in the outline of the mask.

See also:

skimage.measure.find_contours

fissa.roitools.**get_mask_com**(*mask*)
Get the center of mass for a boolean mask.

Parameters

mask [array_like] A two-dimensional boolean-mask.

Returns

x [float] Center of mass along first dimension.

y [float] Center of mass along second dimension.

fissa.roitools.**get_npil_mask**(*mask*, *totalexpansion=4*)
Given the masks for a ROI, find the surrounding neuropil.

Our implementation is as follows:

- On even iterations (where indexing begins at zero), expand the mask in each of the 4 cardinal directions.
- On odd numbered iterations, expand the mask in each of the 4 diagonal directions.

This procedure generates a neuropil whose shape is similar to the shape of the input ROI mask.

Parameters

mask [array_like] The reference ROI mask to expand the neuropil from. The array should contain only boolean values.

totalexpansion [float, default=4] How much larger to make the neuropil total area than mask area. Default is 4.

Returns

grown_mask [numpy.ndarray] A boolean numpy.ndarray mask, where the region surrounding the input is now `True` and the region of the input mask is `False`.

fissa.roitools.**getmasks**(*rois*, *shpe*)
Get the masks for the specified rois.

Parameters

rois [array_like] List of roi coordinates. Each roi coordinate should be a 2d-array or equivalent list. i.e.: `roi = [[0, 0], [0, 1], [1, 1], [1, 0]]` or `roi = np.array([[0, 0], [0, 1], [1, 1], [1, 0]])` i.e. a n by 2 array, where n is the number of coordinates. If a 2 by n array is given, this will be transposed.

shpe [array_like] Shape of underlying image `(width, height)`.

Returns

masks [list of `numpy.ndarray`] List of masks for each ROI in *rois*.

`fissa.roitools.`**`getmasks_npil`**(*cellMask*, *nNpil=4*, *expansion=1*)

Generate neuropil masks using *`get_npil_mask()`* function.

Parameters

cellMask [array_like] The cell mask (boolean 2d arrays).

nNpil [int, default=4] Number of neuropil subregions. Default is `4`.

expansion [float, default=1] Area of each neuropil region, relative to the area of *cellMask*. Default is `1`.

Returns

masks_split [list] Returns a list with soma and neuropil masks (boolean 2d arrays).

`fissa.roitools.`**`readrois`**(*roiset*)

Read ImageJ rois from a roiset zipfile.

We ensure that the third dimension (i.e. frame number) is always zero.

Parameters

roiset [str] Path to a roiset zipfile.

Returns

rois [list] The ROIs (regions of interest) from within roiset, as polygons describing the outline of each ROI.

`fissa.roitools.`**`rois2masks`**(*rois*, *shape*)

Convert ROIs into a list of binary masks.

Parameters

rois [str or list of array_like] Either a string containing a path to an ImageJ roi zip file, or a list of arrays encoding polygons, or list of binary arrays representing masks.

shape [array_like] Image shape as a length 2 vector.

Returns

masks [list of numpy.ndarray] List of binary arrays.

`fissa.roitools.`**`shift_2d_array`**(*a*, *shift=1*, *axis=0*)

Shift array values, without wrap around.

Parameters

a [array_like] Input array.

shift [int, default=1] How much to shift array by. Default is `1`.

axis [int, default=0] The axis along which elements are shifted. Default is `0`.

Returns

out [numpy.ndarray] Array with the same shape as *a*, but shifted appropriately.

fissa.roitools.**split_npil**(*mask*, *centre*, *num_slices*, *adaptive_num=False*)

Split a mask into approximately equal slices by area around its center.

Parameters

mask [array_like] Mask as a 2d boolean array.

centre [tuple] The center co-ordinates around which the mask will be split.

num_slices [int] The number of slices into which the mask will be divided.

adaptive_num [bool, optional] If `True`, the *num_slices* input is treated as the number of slices to use if the ROI is surrounded by valid pixels, and automatically reduces the number of slices if it is on the boundary of the sampled region.

Returns

masks [list] A list with *num_slices* many masks, each of which is a 2d boolean numpy array.

CHAPTER

FOUR

CONTRIBUTING CODE

4.1 Reporting Issues

If you encounter a problem when implementing or using FISSA, we want to hear about it!

4.1.1 Gitter

To get help resolving implementation difficulties, or similar one-off problems, please ask for help on our gitter channel.

4.1.2 Reporting Bugs and Issues

If you experience a bug, please report it by opening a new issue. When reporting issues, please include code that reproduces the issue and whenever possible, an image that demonstrates the issue. The best reproductions are self-contained scripts with minimal dependencies.

Make sure you mention the following things:

- What did you do?
- What did you expect to happen?
- What actually happened?
- What versions of FISSA and Python are you using, and on which operating system?

4.1.3 Feature requests

If you have a new feature or enhancement to an existing feature you would like to see implemented, please check the list of existing issues and if you can't find it make a new issue to request it. If you do find it in the list, you can post a comment saying +1 (or `:+1:` if you are a fan of emoticons) to indicate your support for this feature.

4.2 Documentation

We are glad to accept any sort of documentation: function docstrings, tutorials, Jupyter notebooks demonstrating implementation details, etc.

reStructuredText documents and notebooks live in the source code repository under the `docs` directory.

4.2.1 Docstrings

Documentation for classes and functions should follow the format prescribed for numpy.

A complete example of this is available here.

4.3 How to contribute

The preferred way to contribute to FISSA is to fork the main repository on GitHub.

1. Fork the project repository. Click on the 'Fork' button near the top of the page. This creates a copy of the code under your account on the GitHub server.
2. Clone this copy to your local disk:

   ```
   git clone git@github.com:YourUserName/fissa.git
   cd fissa
   ```

3. Create a branch to hold and track your changes

   ```
   git checkout -b my-feature
   ```

   and start making changes.

4. Work on this copy on your computer using Git to do the version control. When you're done editing, do:

   ```
   git add modified_files
   git commit
   ```

   to record your changes in Git, writing a commit message following the *specifications below*, then push them to GitHub with:

   ```
   git push -u origin my-feature
   ```

5. Finally, go to the web page of your fork of the FISSA repo, and click 'Pull request' to issue a pull request and send your changes to the maintainers for review. This will also send a notification email to the committers.

If any of the above seems like magic to you, then look up the Git documentation on the web.

It is recommended to check that your contribution complies with the following rules before submitting a pull request.

- All public functions and methods should have informative docstrings, with sample usage included in the doctest format when appropriate.
- All unit tests pass. Check with (from the top level source folder):

  ```
  pip install pytest
  pytest
  ```

- Code with good unit test coverage (at least 90%, ideally 100%). Check with

```
pip install pytest pytest-cov
pytest --cov=fissa --cov-config .coveragerc
```

  and look at the value in the 'Cover' column for any files you have added or amended.

  If the coverage value is too low, you can inspect which lines are or are not being tested by generating a html report. After opening this, you can navigate to the appropriate module and see lines which were not covered, or were only partially covered. If necessary, you can do this as follows:

```
pytest --cov=fissa --cov-config .coveragerc --cov-report html --cov-report term-
↪missing
sensible-browser ./htmlcov/index.html
```

- No pyflakes warnings. Check with:

```
pip install pyflakes
pyflakes path/to/module.py
```

- No PEP8 warnings. Check with:

```
pip install pep8
pep8 path/to/module.py
```

  AutoPEP8 can help you fix some of the easier PEP8 errors.

```
pip install autopep8
autopep8 -i -a -a path/to/module.py
```

  Note that using the `-i` flag will modify your existing file in-place, so be sure to save any changes made in your editor beforehand.

These tests can be collectively performed in one line with:

```
pip install -r requirements-test.txt
pytest --flake8 --cov=fissa --cov-config .coveragerc --cov-report html --cov-report term
```

4.3.1 Commit messages

Commit messages should be clear, precise and stand-alone. Lines should not exceed 72 characters.

It is useful to indicate the nature of your commits with a commit flag, as described in the numpy development guide.

You can use these flags at the start of your commit messages:

API: an (incompatible) API change
BLD: change related to building the package
BUG: bug fix
CI: change continuous integration build
DEP: deprecate something, or remove a deprecated object
DEV: development tool or utility
DOC: documentation; only change/add/remove docstrings, markdown or comments
ENH: enhancement; add a new feature without removing existing features
JNB: changing a jupyter notebook

MNT: maintenance commit (refactoring, typos, etc.); no functional change
REL: related to releases
REV: revert an earlier commit
RF: refactoring
STY: style fix (whitespace, PEP8)
TST: addition or modification of tests

4.4 Notes

This document was based on the contribution guidelines for sklearn, numpy and Pillow.

CHAPTER

FIVE

CHANGELOG

All notable changes to FISSA will be documented here.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

Categories for changes are: Added, Changed, Deprecated, Removed, Fixed, Security.

5.1 Unreleased

Full commit changelog.

5.2 Version 1.0.0

Release date: 2022-03-25. Full commit changelog. This is a major release which offers a serious update to the interface, documentation, and backend of FISSA.

Please note that some of the changes to the codebase are backward-incompatible changes. For the most part, the only breaking changes which users will need to be concerned with are listed below. We also recommend looking through our updated tutorials and examples.

- The format of the cache used by the new release is different to previous versions. The new version is not capable of restoring previous results saved in the old cache format. If it is run with `folder` set to a directory containing a cache in the old format, it will ignore the old cache and run from scratch, saving the new cache in the same directory with a different file name and format.
- The shape of `experiment.deltaf_raw` has been changed to be a `numpy.ndarray` of 2d numpy arrays shaped `(1, time)` to match the shape of the other outputs.

Although we have noted some other breaking changes, end users are very unlikely to be affected by these. These other changes will only affect users that have written their own custom datahandler, or who are directly calling backend tools such as the `ROI` module, or the functions `fissa.core.separate_func` and `fissa.core.extract_func`, all of which which were either removed or refactored and renamed in this release.

The most important addition is `fissa.run_fissa`, a new high-level function-based interface to FISSA. This function does the same operations as `fissa.Experiment`, but has a more streamlined interface.

5.2.1 Breaking changes

- The names of the cache files have changed from `"preparation.npy"` and `"separated.npy"` to `"prepared.npz"` and `"separated.npz"`, and the structure of the contents was changed. FISSA output caches generated with version 0.7.x will not no longer be loaded when using version 1.0.0. The new version stores analysis parameters and TIFF means along with the raw and decontaminated traces. (#177, #223, #245, #246)
- The shape of `experiment.deltaf_raw` was changed from a `numpy.ndarray` shaped `(cells, trials)`, each containing a `numpy.ndarray` shaped `(time, )`, to `numpy.ndarray` shaped `(cells, trials)`, each element of which is shaped `(1, time)`. The new output shape matches that of `experiment.raw` and `experiment.deltaf_result`. (#198)
- The way data handlers were defined was reworked to use classes instead. The `datahandler` and `datahandler_framebyframe` modules were dropped, and replaced with an `extraction` module containing both of these data handlers as classes instead. Custom data handlers will need to be rewritten to be a class inheriting from `fissa.extraction.DataHandlerAbstract` instead of a custom module.
- The `ROI` module was renamed to `polygons`. (#219)
- In `neuropil.separate`, the keyword arguments `maxiter` and `maxtries` were renamed to be `max_iter` and `max_tries`. This change was made so that the parameter name `max_iter` is the same as the parameter name used by `sklearn.decomposition.NMF`. (#230)
- The internal functions `fissa.core.extract_func` and `fissa.core.separate_func` were removed. New functions which are comparable but actually have user-friendly interfaces, `fissa.core.extract` and `fissa.core.separate_trials`, were added in their place.

5.2.2 Changed

- The outputs `experiment.raw`, `experiment.sep`, and `experiment.deltaf_raw` were changed from a list of lists of 2d numpy arrays, to a 2d numpy array of 2d numpy arrays. Other outputs, such as `experiment.result` were already a 2d numpy array of 2d numpy arrays. (#164)
- Output arrays (`experiment.result`, etc.) are now initialized as empty arrays instead of lists of lists of `None` objects. (#212)
- The multiprocessing back-end now uses joblib instead of the multiprocessing module. (#227)
- Unnecessary warnings about ragged numpy.ndarrays were suppressed. (#243, #247)
- Output properties are now automatically wiped when parameter attributes are changes. (#254)
- The set of extra requirements named `"dev"` which specified the requirements needed to run the test suite was renamed to `"test"`. This can be installed with `pip install fissa[test]`. There is still a `"dev"` set of extras, but these are now development tools and no longer include the requirements needed to run the unit tests. (#185)

5.2.3 Fixed

- The preparation and separation steps are no longer needlessly re-run (#171, #172)
- Mean images are saved with float64 precision regardless of the precision of the source TIFFs file. (#176)
- Various bugs in the Suite2p workflow. (#181, #257)
- Variables set to `None` are no longer saved as `numpy.ndarray`. (#199)
- An error is now raised when both lowmemory mode and a manual datahandler are provided. (#206)

- Mismatches between the number of rois/trials/etc. and the array shapes (which can occur when the data in `experiment.raw` is overwritten) are resolved by determining these attributes dynamically. (#244)
- Use `np.array` instead of the deprecated `np.matrix`. (#174)
- Use `np.float64` instead of the deprecated `np.float`. (#213)
- Iterate over elements in `shapely.geometry.MultiPolygon` by using the `geoms` attribute instead treating the whole object as an iterable (which is now deprecated). (#272)

5.2.4 Added

- Added `fissa.run_fissa`, a high-level function-based interface to FISSA. This does the same operations as `fissa.Experiment`, but in a more streamlined interface. (#169, #237)
- Added a `verbosity` argument to control how much feedback is given by FISSA when it is running. (#200, #225, #238, #240)
- A new `fissa.Experiment.to_matfile` method was added. The arrays saved in this matfile have a different format to the previous `fissa.Experiment.save_to_matlab` method, which is now deprecated. (#249)
- A new data handler `extract.DataHandlerTifffileLazy` was added, which is able to handle TIFF files of all data types while in low-memory mode. (#156, #179, #187).
- In `fissa.Experiment`, arguments `max_iter`, `tol`, and `max_tries` were added which pass through to `neuropil.separate` to control the stopping conditions for the signal separation routine. (#224, #230)
- In `fissa.Experiment`, add `__repr__` and `__str__` methods. These changes mean that `str(experiment)` describes the content of a `fissa.Experiment instance` in a human readable way, and `repr(experiment)` in a machine-readable way. (#209, #231)
- Support for arbitrary `sklearn.decomposition` classes (e.g. FactorAnalysis), not just ICA and NMF. (#188)

5.2.5 Deprecated

- The `fissa.Experiment.save_to_matlab` method was deprecated. Please use the new `fissa.Experiment.to_matfile` method instead. The new method has a different output structure by default (which better matches the structure in Python). If you need to continue using the old structure, you can use `fissa.Experiment.to_matfile(legacy=True)`. (#249)

5.2.6 Documentation

- Reworked all the tutorial notebooks to have better flow, and use matplotlib instead of holoviews which is more approachable for new users. (#205, #228, #239, #279)
- The Suite2p example notebook was moved to a separate repository. This change was made because we want to test our other notebooks with the latest versions of their dependencies, but this did not fit well with running Suite2p, which needs a precise combination of dependencies to run.
- Integrated the example notebooks into the documentation generated by Sphinx and shown on readthedocs. (#273)
- Other various notebook improvements. (#248)
- Various documentation improvements. (#153, #162, #166, #167, #175, #182, #183, #184, #193, #194, #204, #207, #210, #214, #218, #232, #233, #236, #253)

5.2.7 Dev changes

- Changed the code style to black. (#215, #258)
- Add pre-commit hooks to enforce code style and catch pyflake errors. (#161, #180, #217, #234, #261)
- Migrate CI test suite to GitHub Actions. (#154, #195)
- Various changes and updates to the test suite. (#170, #191, #197, #201 #202, #211, #221, #222, #226, #235, #255)
- Notebooks are now automatically deployed on github pages. (#178)

5.3 Version 0.7.2

Release date: 2020-05-24. Full commit changelog.

5.3.1 Fixed

- Loading ovals and ellipses which are partially offscreen (to the top or left of the image). (#140)

5.3.2 Changed

- Attempting to load any type of ROI which is fully offscreen to the top or left of the image now produces an error. (#140)

5.4 Version 0.7.1

Release date: 2020-05-22. Full commit changelog.

5.4.1 Fixed

- Loading oval, ellipse, brush/freehand, freeline, and polyline ImageJ ROIs on Python 3. (#135)

5.4.2 Added

- Support for rotated rectangle and multipoint ROIs on Python 3. (#135)

5.5 Version 0.7.0

Release date: 2020-05-04. Full commit changelog.

5.5.1 Security

- **Caution:** This release knowingly exposes a new security vulnerability. In numpy 1.16, the default behaviour of numpy.load changed to stop loading files saved with pickle compression by default, due to potential security problems. However, the default behaviour of numpy.save is still to save with pickling enabled. In order to preserve our user-experience and backward compatibility with existing fissa cache files, we have changed our behaviour to allow numpy to load from pickled files. (#111)

5.5.2 Changed

- Officially drop support for Python 3.3 and 3.4. Add `python_requires` to package metadata, specifying Python 2.7 or >=3.5 is required. (#114)
- Allow tuples and other sequences to be image and roi inputs to FISSA, not just lists. (#73)
- Multiprocessing is no longer used when the number of cores is specified as 1. (#74)
- Changed default `axis` argument to internal function `fissa.roitools.shift_2d_array` from `None` to `0`. (#54)
- Documentation updates. (#112, #115, #119, #120, #121)

5.5.3 Fixed

- Allow loading from pickled numpy saved files. (#111)
- Problems reading ints correctly from ImageJ rois on Windows; fixed for Python 3 but not Python 2. This problem does not affect Unix, which was already working correctly on both Python 2 and 3. (#90)
- Reject unsupported `axis` argument to internal function `fissa.roitools.shift_2d_array`. (#54)
- Don't round number of npil segments down to 0 in `fissa.roitools.split_npil` when using `adaptive_num=True`. (#54)
- Handling float `num_slices` in `fissa.roitools.split_npil`, for when `adaptive_num=True`, which was causing problems on Python 3. (#54)

5.5.4 Added

- Test suite additions. (#54, #99)

5.6 Version 0.6.4

Release date: 2020-04-07. Full commit changelog.

This version fully supports Python 3.8, but unfortunately this information was not noted correctly in the PyPI metadata for the release.

5.6.1 Fixed

- Fix multiprocessing pool closure on Python 3.8. (#105)

5.7 Version 0.6.3

Release date: 2020-04-03. Full commit changelog.

5.7.1 Fixed

- Specify a maximum version for the panel dependency of holoviews on Python <3.6, which allows us to continue supporting Python 3.5, otherwise dependencies fail to install. (#101)
- Save deltaf to MATLAB compatible output. (#70)
- Wipe downstream data stored in the experiment object if upstream data changes, so data that is present is always consistent with each other. (#93)
- Prevent slashes in paths from doubling up if the input path has a trailing slash. (#71)
- Documentation updates. (#91, #88, #97, #89)

5.8 Version 0.6.2

Release date: 2020-03-11. Full commit changelog.

5.8.1 Fixed

- Specify a maximum version for tifffile dependency on Python <3.6, which allows us to continue supporting Python 2.7 and 3.5, which otherwise fail to import dependencies correctly. (#87)
- Documentation fixes and updates. (#64, #65, #67, #76, #77, #78, #79, #92)

5.9 Version 0.6.1

Release date: 2019-03-11. Full commit changelog.

5.9.1 Fixed

- Allow `deltaf.findBaselineF0` to run with fewer than 90 samples, by reducing the pad-length if necessary. (#62)
- Basic usage notebook wasn't supplying the correct `datahandler_custom` argument for the custom datahandler (it was using `datahandler` instead, which is incorrect; this was silently ignored previously but will now trigger an error). (#62)
- Use `ncores_preparation` for perparation step, not `ncores_separation`. (#59)
- Only use `ncores_separation` for separation step, not all cores. (#59)

- Allow both byte strings and unicode strings to be arguments of functions which require strings. Previously, byte strings were required on Python 2.7 and unicode strings on Python 3. (#60)

5.10 Version 0.6.0

Release date: 2019-02-26. Full commit changelog.

5.10.1 Added

- Python 3 compatibility. (#33)
- Documentation generation, with Sphinx, Sphinx-autodoc, and Napoleon. (#38)

5.11 Version 0.5.3

Release date: 2019-02-18. Full commit changelog.

5.11.1 Fixed

- Fix f0 detection with low sampling rates. (#27)

5.12 Version 0.5.2

Release date: 2018-03-07. Full commit changelog.

5.12.1 Changed

- The default alpha value was changed from 0.2 to 0.1. (#20)

5.13 Version 0.5.1

Release date: 2018-01-10. Full commit changelog.

5.13.1 Added

- Possibility to define custom datahandler script for other formats
- Added low memory mode option to load larger tiffs frame-by-frame (#14)
- Added option to use ICA instead of NMF (not recommended, but is a lot faster).
- Added the option for users to define a custom data and ROI loading script. (#13)

5.13.2 Fixed

- Fixed custom datahandler usage. (#14)
- Documentation fixes. (#12)

5.14 Version 0.5.0

Release date: 2017-10-05

Initial release

CHAPTER

SIX

CREDITS

6.1 Development

- Sander Keemink : Conception, Overall development
- Scott Lowe : Overall development

6.2 Supervision

- Nathalie Rochefort
- Mark van Rossum

6.3 Testing

- Janelle Pakan
- Evelyn Dylda

6.4 External Code

- polygons.py was adapted from code contained in the SIMA package under the GNU GPL v2 License.
- NIH ImageJ ROI parsing was apapted from code originally written by Luis Pedro Coelho under the MIT license.

6.5 Funding

- SK: DTC & Eurospin
- Wellcome Trust (Sir Henry Dale Fellowship), EU funding (Career Integration Grant)
- JP: EU intra-European Fellowship
- Centre for Integrative Physiology, University of Edinburgh

6.6 Citation

If you use FISSA for your research, we would be grateful if you could cite our paper on FISSA in any resulting publications:

> S. W. Keemink, S. C. Lowe, J. M. P. Pakan, E. Dylda, M. C. W. van Rossum, and N. L. Rochefort. FISSA: A neuropil decontamination toolbox for calcium imaging signals, *Scientific Reports*, **8**(1):3493, 2018. doi: 10.1038/s41598-018-21640-2.

For your convenience, we provide a copy of this citation in bibtex and RIS format.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

f

INDEX

N

P

R

S

T