Offline Analysis Tools

Working with TDT data in Python

Updated 2025-05-12

Tucker-Davis Technologies
11930 Research Circle
Alachua, FL 32615 USA
Phone: +1.386.462.9622
Fax: +1.386.462.5365

Notices

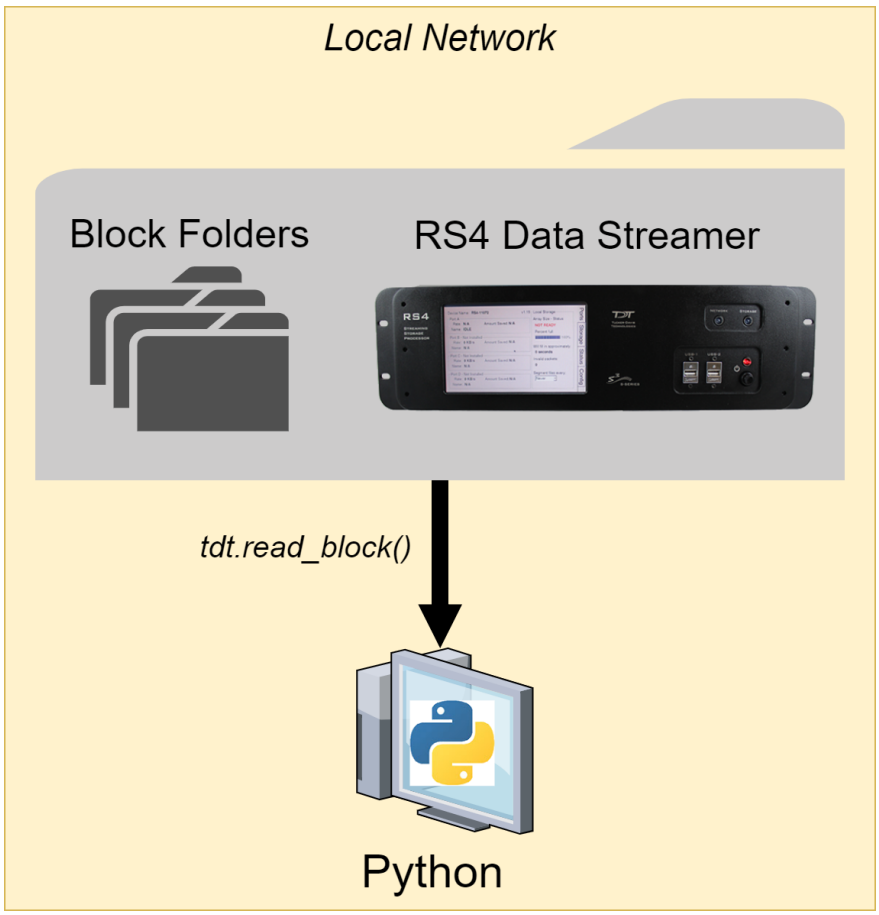
The information contained in this document is provided "as is," and is subject to being changed, without notice. TDT shall not be liable for errors or damages in connection with the furnishing, use, or performance of this document or of any information contained herein.

The latest versions of TDT documents are always online at https://www.tdt.com/docs/

Table of Contents

Overview of Python Offline Analysis Tools

Read block files directly from disk or SEV files directly from RS4.



See TDT Data Storage for a description of the folder structure.

read_block

read_block is an all-in-one function for reading TDT data into Python. It needs only one input: the block path.

```
import tdt
data = tdt.read_block('C:\TDT\TDTExampleData\Algernon-180308-130351')
```

read_block will return a structure containing all recorded data from that block, organized by type. See TDT Data Types for a description of the data types.

The returned structure also contains an **info** field with block start/stop times, duration, and information about the Subject, User, and Experiment that it came from (if the block was created in Synapse).

`read_block` uses input parameters to refine the imported data. To extract specific event types only, use the `evtype` parameter. For example, to import epocs and snippets only, use this:

```
data = tdt.read_block('C:\TDT\TDTExampleData\Algernon-180308-130351',
    evtype=['epocs','snips'])
```

Use the `store` parameter to extract a particular data store by name, in this example a streaming event called 'Wav1'. Combine this with the `channel` parameter to extract a single channel, or list of channels, in this case channels 2 and 4:

```
data = tdt.read_block('C:\TDT\TDTExampleData\Algernon-180308-130351',
    store='Wav1', channel=[2,4])
```

You can also filter by time, if you are only interested in portions of the recording, or if the entire recording won't fit into available memory (RAM) at one time. Use the `t1` and `t2` parameters to specify the start and stop time, in seconds, to retrieve from the block. This example reads only from time t1=10s to time t2=20s of the block into Python:

```
data = tdt.read_block('C:\TDT\TDTExampleData\Algernon-180308-130351',
    t1=10, t2=20)
```

`read_block` offers many more useful options that are described in its help documentation.

```
print(tdt.read_block.__doc__)
```

epoc_filter

`epoc_filter` applies advanced epoc filtering to extracted data. For example, if you only want to look at data around a certain epoc event, you will use `epoc_filter` to do this. See the Raster/PSTH example for a complete demonstration.

To only look at data around the epoc event timestamps, use a `t` parameter to set the filter time range. In this example, data from 20 ms before the *Levl* epoc to 50 ms after the onset is kept.

```
data = tdt.epoc_filter(data, 'Levl', t=[-0.02, 0.07])
```

To only look at data when an epoc was a certain value or values, use a `values` parameter. In this example, only data when the *Freq* epoc was equal to 9000 or 10000 is retained.

```
data = tdt.epoc_filter(data, 'Freq', values=[9000, 10000])
```

If you want to look for a particular behavioral response that occurs sometime during the allowed time range, use the `modifiers` filter. In this example, only data when the *Freq* epoc was 10000 **AND** the *Resp* epoc had a value of 1 sometime during the *Freq* epoc is retained.

```
data = tdt.epoc_filter(data, 'Freq', values=[10000])
data = tdt.epoc_filter(data, 'Resp', modifiers=[1])
```

As you can see, for complex filtering the output from one call to `epoc_filter` can become the input to the next call to `epoc_filter`. If your data sets are large, or if you are iterating through many combinations of epoc variables, it is preferred to extract only the epocs and do all of the epoc filtering first to find the valid time ranges that match the filter, and then use this as the `ranges` input to `read_block` to extract all events (including snips, streams) on only those valid time ranges.

```
# read just the epoc events
data = tdt.read_block(block_path, evtype=['epocs'])

# use the epocs to find time ranges we want
data = tdt.epoc_filter(data, 'Freq', values=[9000, 10000])
data = tdt.epoc_filter(data, 'Levl', values=[70, 80, 90])
data = tdt.epoc_filter(data, 'Resp', modifiers=[1])

# read just the value time ranges from the whole data set
data = tdt.read_block(block_path, ranges=data.time_ranges)
```

read_sev

`read_sev` reads SEV files into a Python structure. SEV files are created by the RS4 Data Streamer or by enabling the Discrete Files option when streaming continuous signals in Synapse. SEV files consist of a single channel of data per file, with a short header, so it is very fast to read them. `read_block` will automatically call `read_sev` if it finds SEV files in the block directory. Like `read_block`, `read_sev` needs only one input: the block path.

```
data = tdt.read_sev('C:\TDT\Synapse\Tanks\Exp1-160921-120606\Sub1-1')
```

`read_sev` will return a structure containing the streams that it found. Each stream field includes the data array and sampling rate.

TDT Data Storage

Data collected or used by TDT software is stored in tanks - special directories on your hard drive. Each time you press 'Record' in Synapse, a new block is created within the tank. Within a block different stores can record different types of events at different rates. The blocks are special folders within the tank directories.

TDT Data Types

1. **epocs** are values stored with onset and offset timestamps that can be used to create time-based filters on your data. They can be created by the Epoch Data Storage gizmo, Logic gizmos, Stimulation gizmos, and many more.
    a. If Runtime Notes were enabled in Synapse, they will appear in `data.epocs.Note`. The notes themselves will be in `data.epocs.Note.notes`.

    See the Synapse Manual for more information on Runtime Notes.
2. **streams** are continuous single channel or multichannel recordings, like those stored by the Stream Data Storage gizmo, the Fiber Photometry gizmo, and many others. The structure includes the data array and sampling rate.
3. **snips** are short snippets of data collected on a trigger. For example, action potentials recorded around threshold crossings in the Spike Sorting gizmos, or fixed duration snippets recorded by the Strobe Store gizmo. This structure includes the waveforms, channel numbers, sort codes, trigger timestamps, and sampling rate.
4. **scalars** are similar to epocs but can be single or multi-channel values and only store an onset timestamp when triggered. These can be created by the Strobe Store gizmo.

The returned structure also contains an **info** field with block start/stop times, duration, and information about the Subject, User, and Experiment that it came from (if the block was created in Synapse).

Merging Blocks

There is a tool called TankManager that installs with Synapse that allows you to concatenate blocks directly on disk. This can create a 'super block' by combining the snippets / epocs events from multiple blocks into a single block before you export the data to another format or import to OpenSorter. It shifts the timestamps of the second block (it adds ~10 second gap in between the recordings so there is no overlap).

TankManager is a command line executable. The format for merging blocks is:

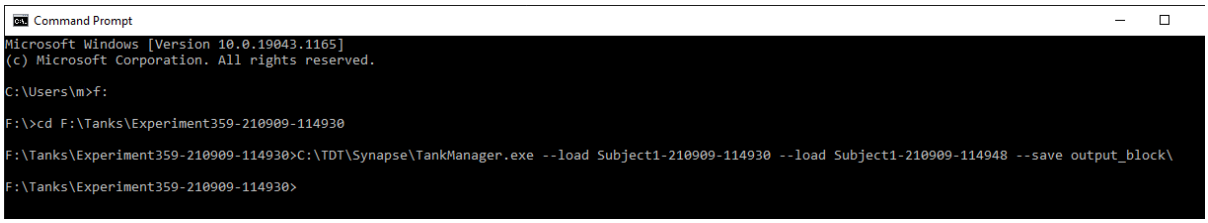
```
C:\TDT\Synapse\TankManager.exe --load block1 --load block2 --save outputblock
```

Replace the text with the two block names that you want to merge.

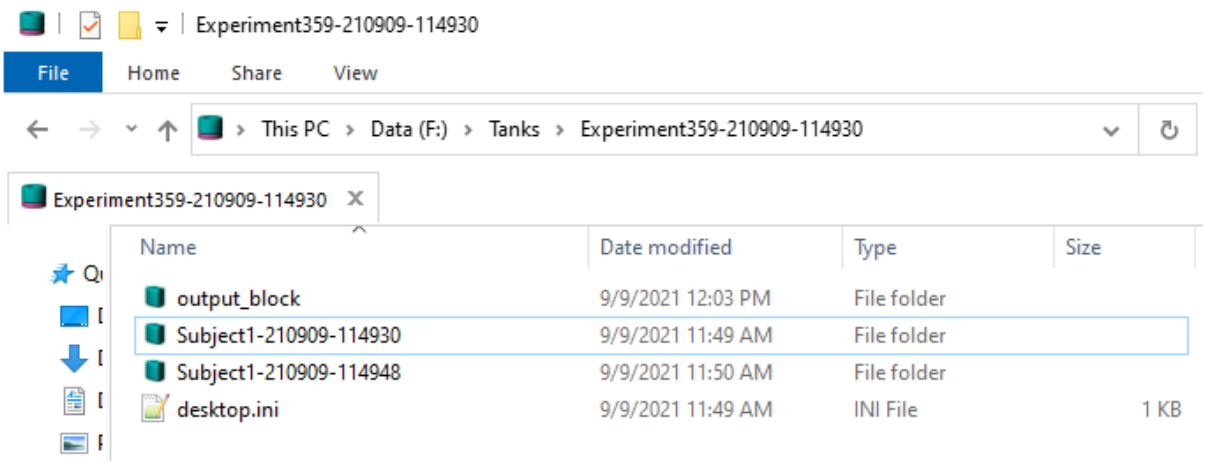
Here is an example that merges the two blocks called `Subject1-210909-114930` and `Subject1-210909-114948` in the tank `F:\Tanks\Experiment359-210909-114930` into a new block called `output_block` inside that same tank:

1. Press the Windows key (or Windows + R) and type `cmd` to enter the command line.
2. Change directory into your tank folder.
3. Run the TankManager command. In this example, it is

```
C:\TDT\Synapse\TankManager.exe --load "Subject1-210909-114930" --load
"Subject1-210909-114948" --save "output_block"
```



This makes a new block called `output_block` inside the same tank folder, which you can then use like any other block.



Splitting Blocks

You can extract time ranges of blocks and export them to smaller blocks using the same TankManager utility described above. This works best with snippet, epoc, and scalar data types.

Here's an example that will keep the data from time t=5s to t=10s from the block `C:\BLOCKPATH` and write it to a new block called `C:\BLOCKPATH_5_10`:

```
C:\TDT\Synapse\TankManager -l "C:\BLOCKPATH" -k "5-10" -s "C:\BLOCKPATH_5_10"
```

For example, you could use this to split up your blocks into hour long chunks like this:

```
C:\TDT\Synapse\TankManager -l "C:\BLOCKPATH" -k "0-3600" -s "C:\BLOCKPATH_0hr"
C:\TDT\Synapse\TankManager -l "C:\BLOCKPATH" -k "3600-7200" -s "C:
\BLOCKPATH_1hr"
```

And so on. The data in these new blocks won't start until the first timestamp, so for later blocks there will be a large gap in the beginning. If you have an issue reading it with OpenSorter or another of our data utilities you might try following the steps in Tech Note TN0909.

> **Note**
>
> You might get a message during the block creation that says `Failed to set the system attribute on the block`. It is okay to ignore this.

Importing TDT Data into Python for Offline Analysis

TDT provides a set of cross-platform tools for reading TDT data files directly into Python offline. It is compatible with Python 3.

Requirements

1. Python 3. We recommend installing Anaconda.

Installation

Just follow these three steps to get started:

1. Install the TDT Python package from pypi. From the Anaconda Prompt, type:

   `pip install tdt`

2. In Python, you can download the example data to the current directory with this:

   ```
   import tdt
   tdt.download_demo_data()
   ```

3. Importing data is easy

   ```
   data = tdt.read_block('path/to/block')
   ```

Example Notebooks

Introduction to Python

Using the tdt Package

This primer walks through installing the tdt Python package, shows how to convert existing MATLAB code to Python, and highlights some of the differences when working in Python.

Installation

1. Make sure that you have **Anaconda** installed.
2. **Open an Anaconda Prompt** and type:

```
pip install tdt
```

3. **Done**!

Converting Existing MATLAB Code to Python

The tdt Python library for reading TDT data is one-to-one compatible with the MATLAB library, however the function names and parameter names are different.

Extracting Block Data

TDTbin2mat extracts block data into a MATLAB structure.

```
data = TDTbin2mat(BLOCK_PATH);
```

The Python equivalent is **read_block**.

```
from tdt import read_block
data = read_block(BLOCK_PATH)
```

TDTbin2mat and **read_block** share parameters, but the parameters have different names.

Parameter Name Translation Table

MATLAB	Python	Description
T1	t1	scalar, retrieve data starting at t1 (default = 0 for beginning of recording)
T2	t2	scalar, retrieve data ending at t2 (default = 0 for end of recording)
TYPE	evtype	array of strings, specifies what type of data stores to retrieve from the tank
SORTNAME	sortname	string, specify sort ID to use when extracting snippets (default = 'TankSort')
RANGES	ranges	array of valid time range column vectors
NODATA	nodata	bool, only return timestamps, channels, and sort codes for snippets (default = false)
STORE	store	string or list of strings, specify specific store(s) to extract
CHANNEL	channel	integer, choose a single channel to extract from stream or snippet events
BITWISE	bitwise	string, specify an epoc store or scalar store that contains a 32-bit integer. Onsets/offsets from individual bits will be extracted
HEADERS	headers	var, set to 1 to return only the headers for this block, if you need to make fast consecutive calls to read_block
COMBINE	combine	list, specify store(s) that were saved by the Strobed Data Storage gizmo in Synapse. It will intelligently combine data into snippets.

Epoc Filtering

TDTfilter filters events around epoc events in MATLAB:

```
data = TDTbin2mat(BLOCK_PATH);
data = TDTfilter(data, 'Tick', 'TIME', [-0.3, 0.8], 'VALUES', [5, 10, 15]);
```

The Python equivalent is **epoc_filter**:

```
from tdt import read_block, epoc_filter
data = read_block(BLOCK_PATH)
data = epoc_filter(data, 'Tick', t=[-0.3, 0.8], values=[5, 10, 15])
```

TDTbin2mat and **read_block** share parameters, but the parameters have different names.

Parameter Name Translation Table

MATLAB	Python	Description
VALUES	values	array of allowed epoc values
MODIFIERS	modifiers	array of allowed modifier values. For example, only allow time ranges when allowed modifier occurred sometime during that event, e.g. a correct animal response.
TIME	t	onset/offset pair, extracts events around epoc onsets only
TIMEREF	tref	boolean, set to True to use the epoc event onset as a time reference
KEEPDATA	keepdata	boolean, keep the original stream data array and add a field called'filtered' that holds the data from each valid time range

Extracting SEV Data

SEV2mat extracts SEV files from a given directory into a MATLAB structure. These files are created on the RS4 Data Streamer or by enabling the Discrete Files option in the Synapse Stream Data Storage gizmo. Each SEV file contains a header and the raw binary data from a single channel.

```
data = SEV2mat(BLOCK_PATH);
```

The Python equivalent is **read_sev**.

```
from tdt import read_sev
data = read_sev(BLOCK_PATH)
```

SEV2mat and **read_sev** share parameters, but the parameters have different names.

Parameter Name Translation Table

MATLAB	Python	Description
T1	t1	scalar, retrieve data starting at t1 (default = 0 for beginning of recording)
T2	t2	scalar, retrieve data ending at t2 (default = 0 for end of recording)
CHANNEL	channel	integer, returns the SEV data from specified channel only (default = 0 for all channels)
RANGES	ranges	array of valid time range column vectors
JUSTNAMES	just_names	boolean, retrieve only the valid event names
EVENTNAME	event_name	string, specific event name to retrieve data from
VERBOSE	verbose	boolean, set to false to disable console output
FS	fs	float, sampling rate override. Useful for lower sampling rate recordings that aren't correctly written into the SEV header.

Walkthrough

Let's look at some basic concepts for working with Python and the tdt library.

***Python Tips!**

Use `print` in Python in place of `disp` in MATLAB.
`%` is a special command used mainly in Python notebooks
`#` creates a single-line comment in Python
Use `'''` to make a multi-line comment

First we'll import the critical libraries.

```
# this is a single line comment

''' this is a comment
spanning multiple lines'''

# special call that tells notebook to show matlplotlib figures inline
%matplotlib inline

import matplotlib.pyplot as plt  # standard Python plotting library
import numpy as np  # fundamental package for scientific computing, handles
arrays and maths

# import the tdt library
import tdt
```

***Python Tip!**

Use the **__doc__** function to get help on a function.

```
print(tdt.read_block.__doc__)
```

```
TDT tank data extraction.

    data = read_block(block_path), where block_path is a string, retrieves
    all data from specified block directory in struct format. This reads
    the binary tank data and requires no Windows-based software.

    data.epocs      contains all epoc store data (onsets, offsets, values)
    data.snips      contains all snippet store data (timestamps, channels,
                    and raw data)
    data.streams    contains all continuous data (sampling rate and raw
                    data)
    data.scalars    contains all scalar data (samples and timestamps)
    data.info       contains additional information about the block

    optional keyword arguments:
        t1          scalar, retrieve data starting at t1 (default = 0 for
                        beginning of recording)
        t2          scalar, retrieve data ending at t2 (default = 0 for end
                        of recording)
        sortname    string, specify sort ID to use when extracting snippets
                        (default = 'TankSort')
        evtype      array of strings, specifies what type of data stores to
                        retrieve from the tank. Can contain 'all' (default),
                        'epocs', 'snips', 'streams', or 'scalars'.
                      example:
                          data = read_block(block_path,
evtype=['epocs','snips'])
                              > returns only epocs and snips
        ranges      array of valid time range column vectors.
                      example:
                          tr = np.array([[1,3],[2,4]])
                          data = read_block(block_path, ranges=tr)
                              > returns only data on t=[1,2) and [3,4)
        nodata      boolean, only return timestamps, channels, and sort
                        codes for snippets, no waveform data (default =
false).
                        Useful speed-up if not looking for waveforms
        store       string, specify a single store to extract
                    list of strings, specify multiple stores to extract
        channel     integer, choose a single channel to extract from
                        stream or snippet events. Default is 0, to extract
                        all channels.
        bitwise     string, specify an epoc store or scalar store that
                        contains individual bits packed into a 32-bit
                        integer. Onsets/offsets from individual bits will
                        be extracted.
        headers     var, set to 1 to return only the headers for this
                        block, so that you can make repeated calls to read
                        data without having to parse the TSQ file every
                        time, for faster consecutive reads. Once created,
```

```
                        pass in the headers using this parameter.
                      example:
                        heads = read_block(block_path, headers=1)
                        data = read_block(block_path, headers=heads,
evtype=['snips'])
                        data = read_block(block_path, headers=heads,
evtype=['streams'])
        combine     list, specify one or more data stores that were saved
                        by the Strobed Data Storage gizmo in Synapse (or an
                        Async_Stream_store macro in OpenEx). By default,
                        the data is stored in small chunks while the strobe
                        is high. This setting allows you to combine these
                        small chunks back into the full waveforms that were
                        recorded while the strobe was enabled.
                      example:
                        data = read_block(block_path, combine=['StS1'])
        export      string, choose a data exporting format.
                        csv:        data export to comma-separated value files
                                    streams: one file per store, one channel
per column
                                    epocs: one column onsets, one column
offsets
                        binary:     streaming data is exported as raw binary
files
                                    one file per channel per store
                        interlaced: streaming data exported as raw binary
files
                                    one file per store, data is interlaced
        scale       float, scale factor for exported streaming data. Default
= 1.
        dtype       string, data type for exported binary data files
                        None: Uses the format the data was stored in (default)
                        'i16': Converts all data to 16-bit integer format
                        'f32': Converts all data to 32-bit integer format
        outdir      string, output directory for exported files. Defaults to
current
                        block folder if not specified
        prefix    string, prefix for output file name. Defaults to None
```

```
print(tdt.epoc_filter.__doc__)
```

```
TDT tank data filter. Extract data around epoc events.
    data = epoc_filter(data, epoc) where data is the output of read_block,
    epoc is the name of the epoc to filter on, and parameter value pairs
    define the filtering conditions.

    If no parameters are specified, then the time range of the epoc event
    is used as a time filter.

    Also creates data.filter, a string that describes the filter applied.
    Optional keyword arguments:
        values      specify array of allowed values
                      ex: tempdata = epoc_filter(data, 'Freq', values=[9000,
10000])
                        > retrieves data when Freq = 9000 or Freq = 10000
        modifiers   specify array of allowed modifier values.  For example,
                      only allow time ranges when allowed modifier occurred
                      sometime during that event, e.g. a correct animal
response.
                      ex: tempdata = epoc_filter(data, 'Resp', modifiers=[1])
                        > retrieves data when Resp = 1 sometime during the
allowed
                        time range.
        t           specify onset/offset pairs relative to epoc onsets. If the
                      offset is not provided, the epoc offset is used.
                      ex: tempdata = epoc_filter(data, 'Freq', t=[-0.1, 0.5])
                        > retrieves data from 0.1 seconds before Freq onset
to 0.4
                          seconds after Freq onset. Negative time ranges are
discarded.
        tref        use the epoc event onset as a time reference. All
timestamps for
                      epoc, snippet, and scalar events are then relative to
epoc onsets.
                      ex: tempdata = epoc_filter(data, 'Freq', tref=True)
                        > sets snippet timestamps relative to Freq onset
        keepdata    keep the original stream data array and add a field called
                      'filtered' that holds the data from each valid time
range.
                      Defaults to True.

    IMPORTANT! Use a time filter (t argument) only after all value filters
have been set.
```

```
print(tdt.read_sev.__doc__)
```

```
TDT sev file data extraction.

    data = read_sev(sev_dir), where sev_dir is a string, retrieves
    all sev data from specified directory in struct format. sev_dir can
    also be a single file. SEV files are generated by an RS4 Data Streamer,
    or by enabling the Discrete Files option in the Synapse Stream Data
    Storage gizmo, or by setting the Unique Channel Files option in
    Stream_Store_MC or Stream_Store_MC2 macro to Yes in OpenEx.

    If exporting is enabled, this function returns None.

    data    contains all continuous data (sampling rate and raw data)

    optional keyword arguments:
        t1          scalar, retrieve data starting at t1 (default = 0 for
                        beginning of recording)
        t2          scalar, retrieve data ending at t2 (default = 0 for end
                        of recording)
        channel     integer, returns the sev data from specified channel
                        only (default = 0 for all channels)
        ranges      array of valid time range column vectors
        just_names  boolean, retrieve only the valid event names
        event_name  string, specific event name to retrieve data from
        verbose     boolean, set to false to disable console output
        fs          float, sampling rate override. Useful for lower
                        sampling rates that aren't correctly written into
                        the SEV header.
        export      string, choose a data exporting format.
                        csv:        data export to comma-separated value files
                                    streams: one file per store, one channel
per column
                                    epocs: one column onsets, one column
offsets
                        binary:     streaming data is exported as raw binary
files
                                    one file per channel per store
                        interlaced: streaming data exported as raw binary
files
                                    one file per store, data is interlaced
        scale       float, scale factor for exported streaming data. Default
= 1.
        dtype       string, data type for exported binary data files
                        None: Uses the format the data was stored in (default)
                        'i16': Converts all data to 16-bit integer format
                        'f32': Converts all data to 32-bit integer format
        outdir      string, output directory for exported files. Defaults to
current
                        block folder if not specified
        prefix    string, prefix for output file name. Defaults to None
```

Download demo data from the TDT website

```
tdt.download_demo_data()
```

```
demo data ready
```

This example uses our example data sets. To import your own data, replace BLOCK_PATH with the full path to your own data block.

In Synapse, you can find the block path in the database. Go to Menu > History. Find your block, then Right-Click > Copy path to clipboard.

```
BLOCK_PATH = 'data/Algernon-180308-130351'
data = tdt.read_block(BLOCK_PATH)
```

```
read from t=0s to t=61.23s
```

```
print(data)
```

```
epocs   [struct]
snips   [struct]
streams [struct]
scalars [struct]
info    [struct]
time_ranges:    array([[ 0.],
       [inf]])
```

read_block returns a structured object. It is a Python dictionary but also allows you to use the dot syntax like in MATLAB, so you can access fields within the structure with either method. These two ways of looking at the block info field are equivalent:

```
data.info
```

```
tankpath:   'data'
blockname:  'Algernon-180308-130351'
start_date: datetime.datetime(2018, 3, 8, 13, 3, 53, 999999)
utc_start_time: '13:03:53'
stop_date:  datetime.datetime(2018, 3, 8, 13, 4, 55, 233578)
utc_stop_time:  '13:04:55'
duration:   datetime.timedelta(seconds=61, microseconds=233579)
stream_channel: 0
snip_channel:   0
```

```
data['info']
```

```
tankpath:   'data'
blockname:  'Algernon-180308-130351'
start_date: datetime.datetime(2018, 3, 8, 13, 3, 53, 999999)
utc_start_time: '13:03:53'
stop_date:  datetime.datetime(2018, 3, 8, 13, 4, 55, 233578)
utc_stop_time:  '13:04:55'
duration:   datetime.timedelta(seconds=61, microseconds=233579)
stream_channel: 0
snip_channel:   0
```

These three methods to access the 'Wav1' store sampling rate are equivalent:

```
data.streams.Wav1.fs # dot syntax
```

```
24414.0625
```

```
data['streams']['Wav1']['fs'] # dict keys only
```

```
24414.0625
```

```
data['streams'].Wav1['fs'] # mix of dot syntax and dict keys
```

```
24414.0625
```

***Python Tip!**

Spaces are important in python. Commands like `for`, `if`, `elif`, `while`, and others require indents to track their nests

```
for foo in foo_list:
    something
    if foo == check:
        conditional_something
    elif:
        still_in_for_loop
    else:
        still_in_for_loop

out_of_loop
```

Accessing a field with the string dictionary key method is useful when using a variable name, such as this example which loops through all the stream store names and prints their sampling rates.

```
print('Sampling rates in', data.info.blockname)
for store in data.streams.keys():
    print(store, '{:.4f} Hz'.format(data.streams[store].fs))
```

```
Sampling rates in Algernon-180308-130351
LFP1 3051.7578 Hz
pNe1 498.2462 Hz
Wav1 24414.0625 Hz
```

Explore Stream events

Let's look at the contents of the stream event structures.

```
print('all stream stores')
print(data.streams)
```

```
all stream stores
LFP1    [struct]
pNe1    [struct]
Wav1    [struct]
```

```
print(data.streams.Wav1)
```

```
name:   'Wav1'
code:   829841751
size:   2058
type:   33025
type_str:   'streams'
ucf:    False
fs: 24414.0625
dform:  0
start_time: 0.0
data:   array([[ 1.0028159e-03,  1.0012799e-03,  9.9590386e-04, ...,
        -1.5983999e-03, -1.5984639e-03, -1.5852799e-03],
       [ 5.5667193e-04,  5.6723197e-04,  5.6083198e-04, ...,
        -1.4531199e-03, -1.4584319e-03, -1.4480639e-03],
       [-4.6534397e-04, -4.5804796e-04, -4.6521597e-04, ...,
        -1.2184319e-03, -1.2098559e-03, -1.2177919e-03],
       ...,
       [ 2.9247998e-05,  2.3295999e-05,  3.2191998e-05, ...,
        -2.2208637e-03, -2.2241918e-03, -2.2300798e-03],
       [ 7.2191993e-04,  7.1571197e-04,  7.2358397e-04, ...,
        -2.1401597e-03, -2.1399679e-03, -2.1494399e-03],
       [ 2.3078399e-04,  2.3590398e-04,  2.4435198e-04, ...,
        -1.3180159e-03, -1.3103359e-03, -1.3012479e-03]], dtype=float32)
```

The actual data is store in numpy arrays. For a multi-channel stream store, each row is a channel. Python uses 0-based indexing, so we have to subtract 1 from our channel number when accessing the array

```
print('channel 1:', data.streams.Wav1.data[0,:])
```

```
channel 1: [ 0.00100282  0.00100128  0.0009959  ... -0.0015984  -0.00159846
 -0.00158528]
```

```
num_samples = len(data.streams.Wav1.data[0])
print('number of samples:', num_samples)
```

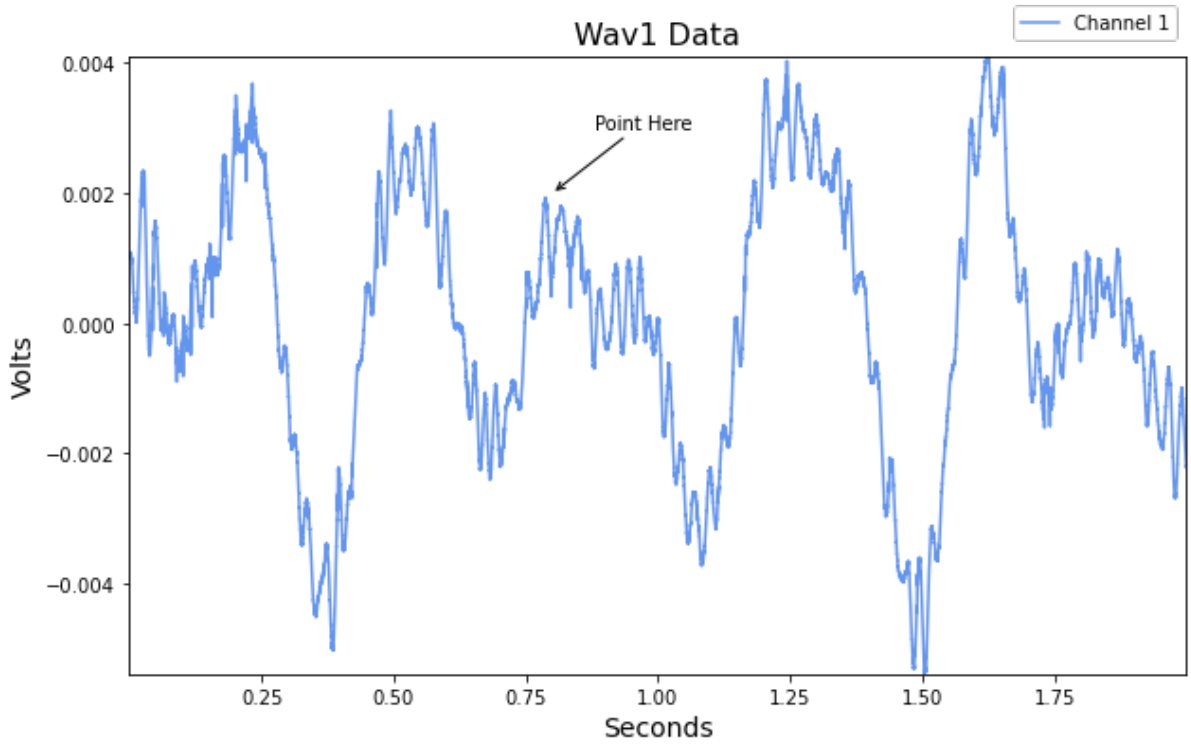
```
number of samples: 1490944
```

Create time vector for plotting by dividing the number of samples in the array by the sampling rate

```
Wav1_time = np.linspace(1, num_samples, num_samples) / data.streams.Wav1.fs
```

Plot the first 2 seconds from a single channel

```
t = int(2 * data.streams.Wav1.fs) # int rounds it to the nearest integer

# declare the figure size
fig1 = plt.subplots(figsize=(10, 6))

channel = 1

# plot the line using slices
plt.plot(Wav1_time[0:t], data.streams.Wav1.data[channel-1,0:t],
color='cornflowerblue')

# Some matplotlib stuff
# add an annotation mark to the figure
plt.annotate('Point Here',
             xy=(0.8,0.002),
             xytext=(.88,.003),
             arrowprops=dict(arrowstyle='->', color='k')
            )

# create title, axis labels, and legend
plt.title('Wav1 Data', fontsize=16)
plt.xlabel('Seconds', fontsize=14)
plt.ylabel('Volts', fontsize=14)
plt.legend(('Channel {}'.format(channel),),
           loc='lower right',
           bbox_to_anchor=(1.0,1.01)
          )
plt.autoscale(tight=True)
plt.show()
```



***Python Tip!**

Array slices in Python have some notable differences. 1. Zero-based indexing 2. Omit 'end' and just use a negative index to index starting from the end of the array 3. Omit the starting index if you want to include the first element

MATLAB array slices

```
arr = 1:10;          % arr = [1 2 3 4 5 6 7 8 9 10]
b = arr(3:5);        % b = [3 4 5]
c = arr(1:end-2);    % c = [1 2 3 4 5 6 7 8]
d = arr(end-1:end);  % d = [9 10]
```

Python equivalent

```
arr = np.arange(1,11)  # arr = [1 2 3 4 5 6 7 8 9 10]
b = arr[2:5]           # b = [3 4 5]
c = arr[:-2]           # c = [1 2 3 4 5 6 7 8]
d = arr[-2:]           # d = [9 10]
```

For reference, here are some matplotlib colors originally from this stackoverflow answer

Explore Epoc Events

Let's look at the contents of the epoc event structures.

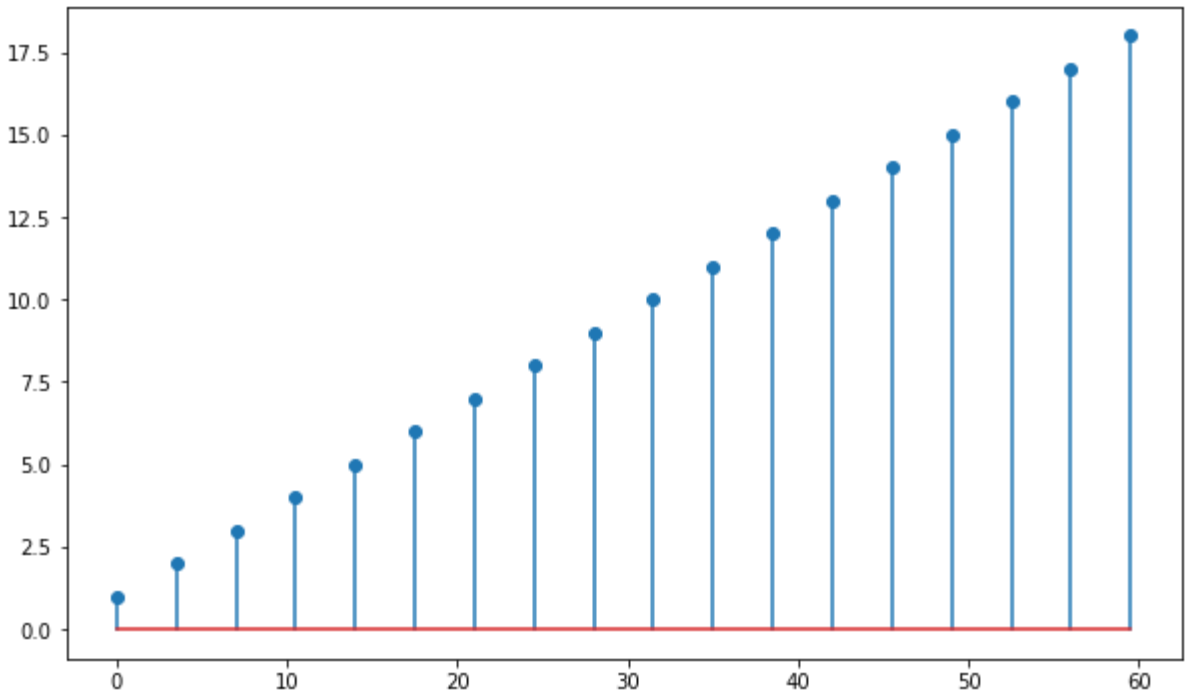
```
print('all epoc events')
print(data.epocs)
```

```
all epoc events
PC0_    [struct]
Pu1e    [struct]
```

```
print(data.epocs.Pu1e)
```

```
name:   'Pu1e'
onset:  array([1.22880000e-04, 3.50011392e+00, 7.00010496e+00, 1.05000960e+01,
       1.40000870e+01, 1.75000781e+01, 2.10000691e+01, 2.45000602e+01,
       2.80000512e+01, 3.15000422e+01, 3.50000333e+01, 3.85000243e+01,
       4.20000154e+01, 4.55000064e+01, 4.89999974e+01, 5.24999885e+01,
       5.59999795e+01, 5.94999706e+01])
offset: array([ 0.30011392,  3.80010496,  7.300096  , 10.80008704,
14.30007808,
       17.80006912, 21.30006016, 24.8000512 , 28.30004224, 31.80003328,
       35.30002432, 38.80001536, 42.3000064 , 45.79999744, 49.29998848,
       52.79997952, 56.29997056, 59.7999616 ])
type:   'onset'
type_str:   'epocs'
data:   array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.,
13.,
       14., 15., 16., 17., 18.])
dform:  4
size:   10
```

```
# a simple plot
fig1 = plt.subplots(figsize=(10, 6))
plt.stem(data.epocs.Pu1e.onset, data.epocs.Pu1e.data)
plt.show()
```



Stream Plot Example

Import Continuous Data into Python
Plot a single channel of data with various filtering schemes
Good for first-pass visualization of streamed data
Combine streaming data and epocs in one plot

Housekeeping

Import the tdt package and other python packages we care about.

```
# special call that tells notebook to show matlplotlib figures inline
%matplotlib inline

import matplotlib.pyplot as plt  # standard Python plotting library
import numpy as np  # fundamental package for scientific computing, handles
arrays and math

# import the tdt library
import tdt
```

Importing the Data

This example uses our example data sets. To import your own data, replace `BLOCK_PATH` with the full path to your own data block.

In Synapse, you can find the block path in the database. Go to Menu → History. Find your block, then Right-Click → Copy path to clipboard.

```
tdt.download_demo_data()
BLOCK_PATH = 'data/Algernon-180308-130351'
```

```
demo data ready
```

Now read channel 1 from all stream data into a Python structure called 'data'

```
data = tdt.read_block(BLOCK_PATH, evtype=['streams', 'epocs'], channel=1)
```

```
read from t=0s to t=61.23s
```

And that's it! Your data is now in Python. The rest of the code is a simple plotting example.

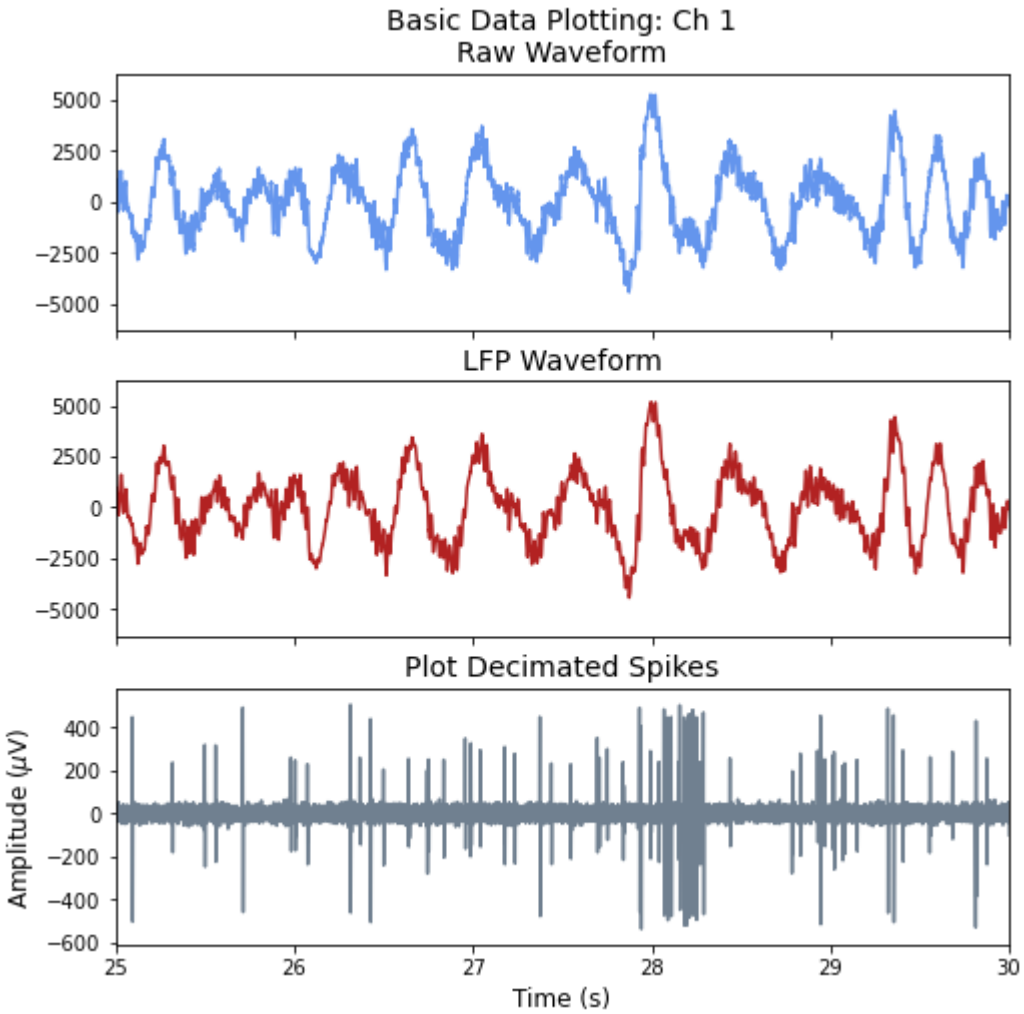
Stream Store Plotting

Let's create time vectors for each stream store for plotting in time.

```
time_Wav1 = np.linspace(1, len(data.streams.Wav1.data),
len(data.streams.Wav1.data)) / data.streams.Wav1.fs
time_LFP1 = np.linspace(1, len(data.streams.LFP1.data),
len(data.streams.LFP1.data)) / data.streams.LFP1.fs
time_pNe1 = np.linspace(1, len(data.streams.pNe1.data),
len(data.streams.pNe1.data)) / data.streams.pNe1.fs
```

Plot five seconds of data from each store

```
fig, (ax1, ax2, ax3) = plt.subplots(nrows=3, ncols=1, figsize=(8, 8),
sharex=True)

ax1.plot(time_Wav1, data.streams.Wav1.data*1e6, color='cornflowerblue')
ax1.set_title('Basic Data Plotting: Ch 1\nRaw Waveform', fontsize=14)

ax2.plot(time_LFP1, data.streams.LFP1.data*1e6, color='firebrick')
ax2.set_title('LFP Waveform', fontsize=14)

ax3.plot(time_pNe1, data.streams.pNe1.data, color='slategray')
ax3.set_title('Plot Decimated Spikes', fontsize=14)

ax3.set_xlabel('Time (s)', fontsize=12)
ax3.set_ylabel('Amplitude ($\mu$V)', fontsize=12)

ax1.set_xlim(25, 30)

plt.show()
```



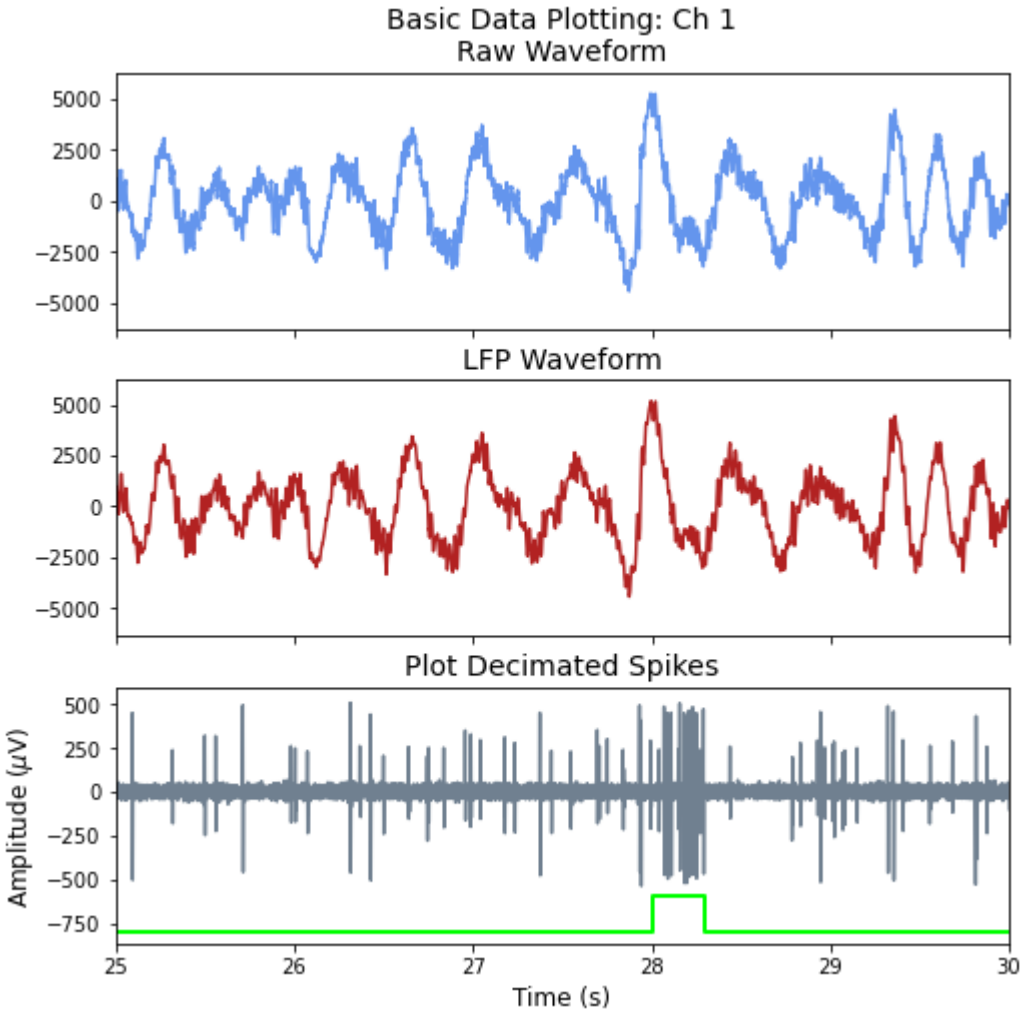
Epoc Events

Generate continuous time series for epoc data using epoc timestamps

```
# StimSync epoc event
STIM_SYNC = 'PC0_'
pc0_on = data.epocs[STIM_SYNC].onset
pc0_off = data.epocs[STIM_SYNC].offset
pc0_x = np.reshape(np.kron([pc0_on, pc0_off], np.array([[1], [1]])).T, [1,
-1])[0]
```

Make a time series waveform of epoc values and plot them.

```
sz = len(pc0_on)
d = data.epocs[STIM_SYNC].data
pc0_y = np.reshape(np.vstack([np.zeros(sz), d, d, np.zeros(sz)]).T, [1,-1])[0]

ax3.plot(pc0_x, 200*(pc0_y) - 800, color='lime', linewidth=2)
fig
```



Averaging Example

Import stream and epoc data into Python using **read_block**
Plot the average waveform around the epoc event using **epoc_filter**
Good for Evoked Potential detection

Housekeeping

Import the tdt package and other python packages we care about.

```
# special call that tells notebook to keep matlplotlib figures open
%config InlineBackend.close_figures = False

# special call that tells notebook to show matlplotlib figures inline
%matplotlib inline

import matplotlib.pyplot as plt  # standard Python plotting library
import numpy as np  # fundamental package for scientific computing, handles
arrays and math

# import the tdt library
import tdt
```

Importing the Data

This example uses our example data sets. To import your own data, replace `BLOCK_PATH` with the full path to your own data block.

In Synapse, you can find the block path in the database. Go to Menu → History. Find your block, then Right-Click → Copy path to clipboard.

```
tdt.download_demo_data()
BLOCK_PATH = 'data/Algernon-180308-130351'
```

```
demo data ready
```

Set up the varibles for the data you want to extract. We will extract channel 3 from the LFP1 stream data store, created by the Neural Stream Processor gizmo, and use our PulseGen epoc event ('PC0/') as our stimulus onset.

```
REF_EPOC     = 'PC0/'
STREAM_STORE = 'LFP1'
ARTIFACT     = np.inf       # optionally set an artifact rejection level
CHANNEL      = 3
TRANGE       = [-0.3, 0.8]  # window size [start time relative to epoc onset,
window duration]
```

Now read the specified data from our block into a Python structure.

```
data = tdt.read_block(BLOCK_PATH, evtype=['epocs','scalars','streams'],
channel=CHANNEL)
```

```
read from t=0s to t=61.23s
```

Use epoc_filter to extract data around our epoc event

Using the 't' parameter extracts data only from the time range around our epoc event. For stream events, the chunks of data are stored in a list.

```
data = tdt.epoc_filter(data, 'PC0/', t=TRANGE)
```

Optionally remove artifacts.

```
art1 = np.array([np.any(x>ARTIFACT) for x in
data.streams[STREAM_STORE].filtered], dtype=bool)
art2 = np.array([np.any(x<-ARTIFACT) for x in
data.streams[STREAM_STORE].filtered], dtype=bool)
good = np.logical_not(art1) & np.logical_not(art2)
data.streams[STREAM_STORE].filtered = [data.streams[STREAM_STORE].filtered[i]
for i in range(len(good)) if good[i]]
num_artifacts = np.sum(np.logical_not(good))
if num_artifacts == len(art1):
    raise Exception('all waveforms rejected as artifacts')
```

Applying a time filter to a uniformly sampled signal means that the length of each segment could vary by one sample. Let's find the minimum length so we can trim the excess off before calculating the mean.

```
min_length = np.min([len(x) for x in data.streams[STREAM_STORE].filtered])
data.streams[STREAM_STORE].filtered = [x[:min_length] for x in
data.streams[STREAM_STORE].filtered]
```

Find the average signal.

```
all_signals = np.vstack(data.streams[STREAM_STORE].filtered)
mean_signal = np.mean(all_signals, axis=0)
std_signal = np.std(all_signals, axis=0)
```

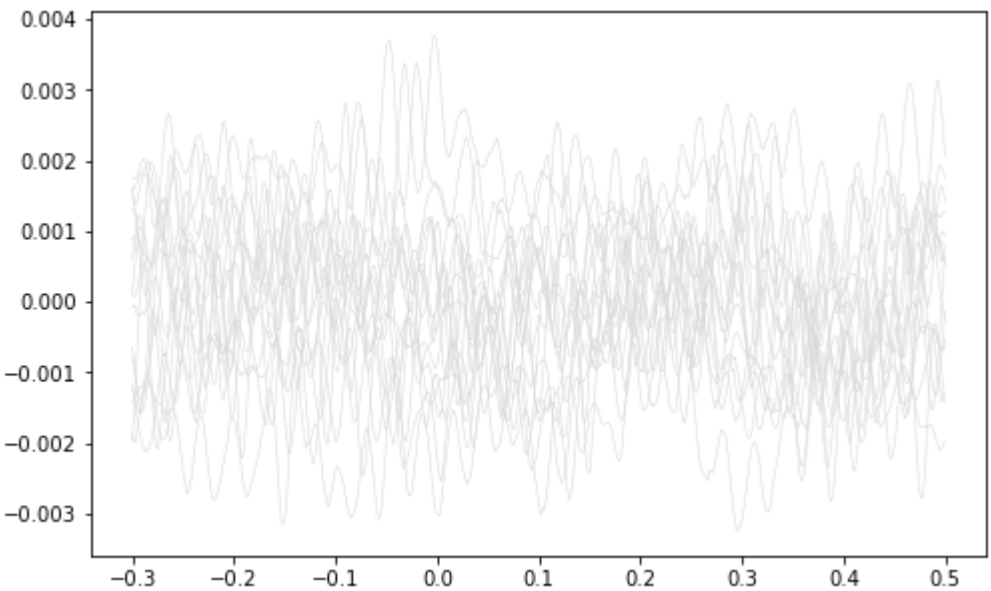
Ready to plot

Create the time vector.

```
ts = TRANGE[0] + np.arange(0, min_length) / data.streams[STREAM_STORE].fs
```

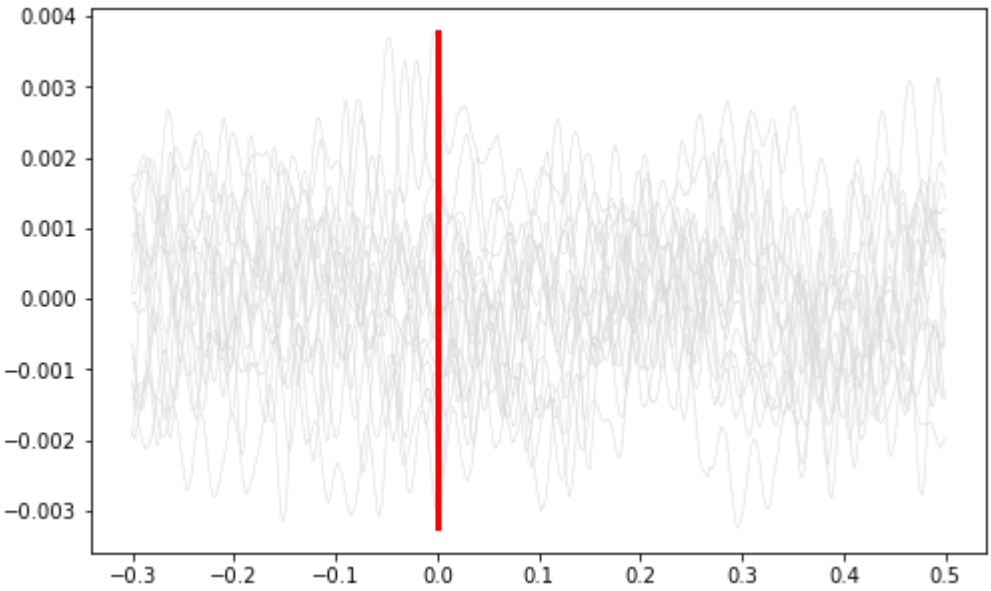
Plot all the signals as gray.

```
fig, ax1 = plt.subplots(1, 1, figsize=(8,5))
ax1.plot(ts, all_signals.T, color=(.85,.85,.85), linewidth=0.5)
plt.show()
```



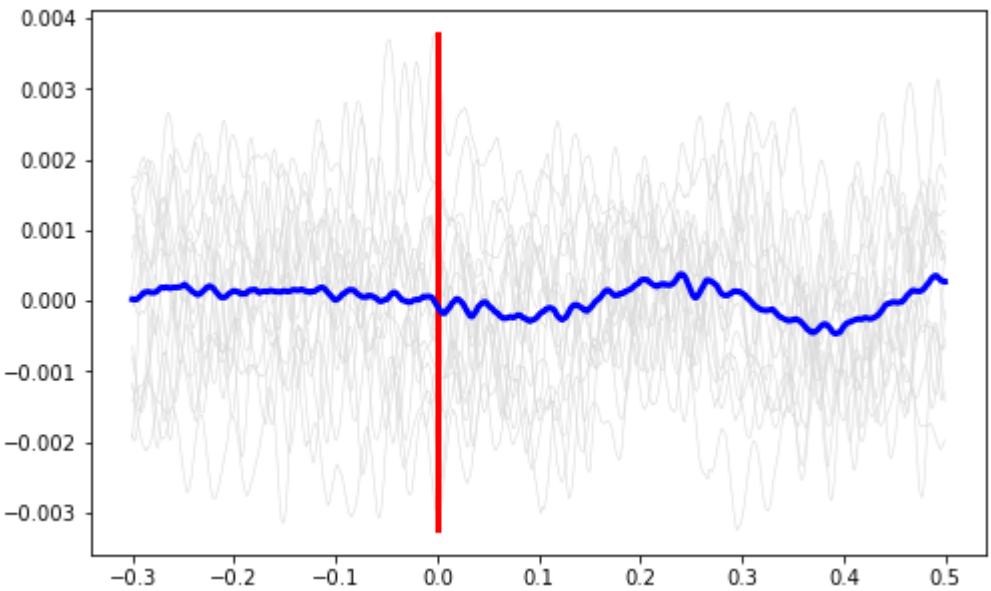
Plot vertical line at time=0.

```
ax1.plot([0, 0], [np.min(all_signals), np.max(all_signals)], color='r',
linewidth=3)
plt.show()
```



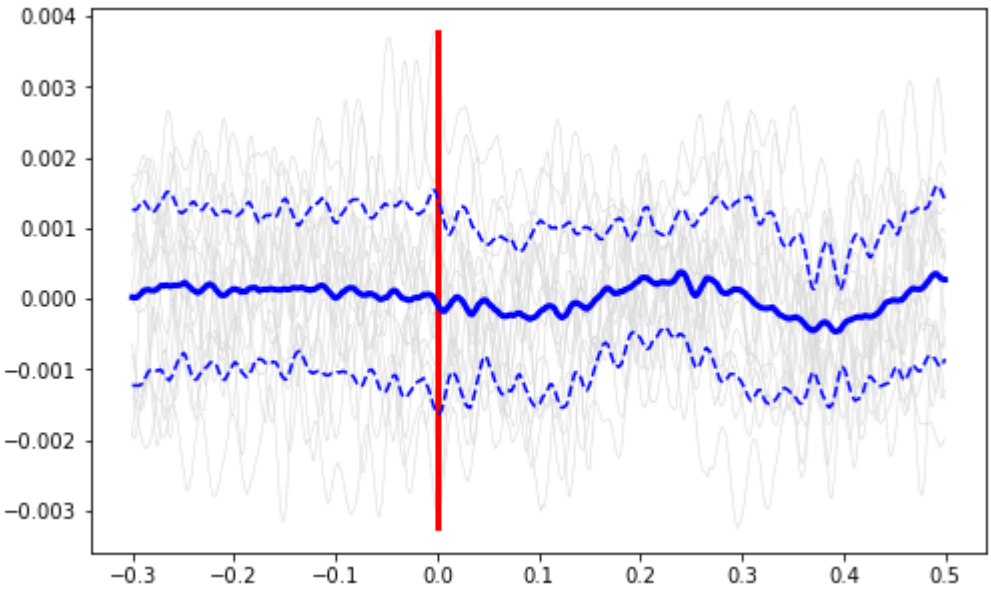
Plot the average signal.

```
ax1.plot(ts, mean_signal, color='b', linewidth=3)
plt.show()
```



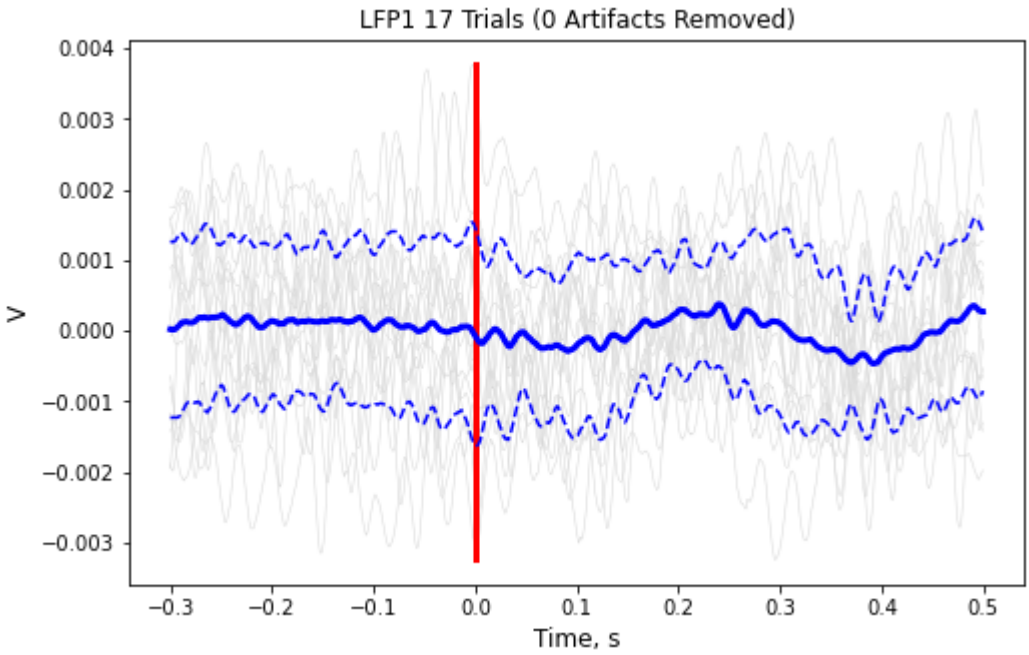
Plot the standard deviation bands.

```
ax1.plot(ts, mean_signal + std_signal, 'b--', ts, mean_signal - std_signal,
'b--')
plt.show()
```



Finish up the plot

```
ax1.axis('tight')
ax1.set_xlabel('Time, s',fontsize=12)
ax1.set_ylabel('V',fontsize=12)
ax1.set_title('{0} {1} Trials ({2} Artifacts Removed)'.format(
    STREAM_STORE,
    len(data.streams[STREAM_STORE].filtered),
    num_artifacts))
plt.show()
```


LFP1 17 Trials (0 Artifacts Removed)
0.004
0.003
0.002
0.001
0.000
−0.001
−0.002
−0.003
V
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
0.4
0.5
Time, s

Raster Peristimulus Time Histogram (PSTH) Example

Import snippet and epoc data into Python using **read_block**
Generate peristimulus raster and histogram plots over all trials using **epoc_filter**
Good for stim-response experiments, such as optogenetic or electrical stimulation

Housekeeping

Import the tdt package and other python packages we care about.

```
%config InlineBackend.close_figures = False
%matplotlib inline

import numpy as np # fundamental package for scientific computing, handles
arrays and math
import matplotlib.pyplot as plt # standard Python plotting library
from matplotlib.ticker import MaxNLocator # so we can force integer tick
labels later

# import the tdt library
import tdt
```

Importing the Data

This example uses our example data sets. To import your own data, replace `BLOCK_PATH` with the full path to your own data block.

In Synapse, you can find the block path in the database. Go to Menu → History. Find your block, then Right-Click → Copy path to clipboard.

```
tdt.download_demo_data()
BLOCK_PATH = 'data/Algernon-180308-130351'
```

```
demo data ready
```

Set up the varibles for the data you want to extract. We will extract channel 1 from the eNe1 snippet data store, created by the PCA Sorting gizmo, and use our PulseGen epoc event `PC0/` as our stimulus onset.

```
REF_EPOC = 'PC0/'
SNIP_STORE = 'eNe1'
SORTID = 'TankSort'
CHANNEL = 3
SORTCODE = 0           # set to 0 to use all sorts
TRANGE = [-0.3, 0.8]
```

Now read the specified data from our block into a Python structure. The `nodata` flag means that we are only intereseted in the snippet timestamps, not the actual snippet waveforms in this example.

```
data = tdt.read_block(BLOCK_PATH, evtype=['epocs', 'snips', 'scalars'],
sortname=SORTID, channel=CHANNEL, nodata=1)
```

```
read from t=0s to t=61.23s
```

Use epoc_filter to extract data around our epoc event

Using the `t` parameter extracts data only from the time range around our epoc event.

```
raster_data = tdt.epoc_filter(data, REF_EPOC, t=TRANGE)
```

Adding the `tref` flag makes all of the timestamps relative to the epoc event, which is ideal for generating histograms.

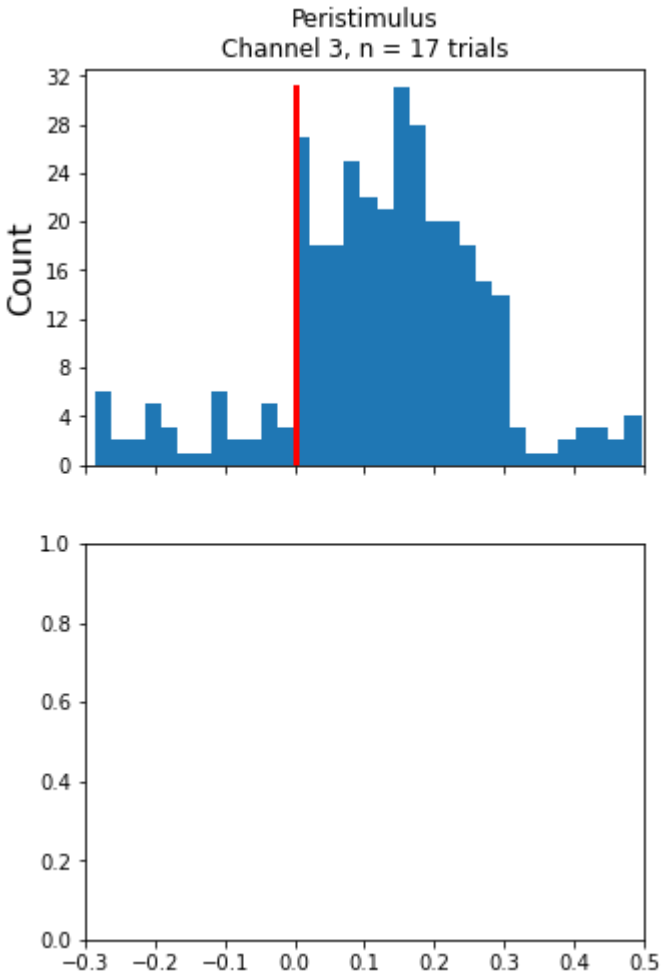
```
hist_data = tdt.epoc_filter(data, REF_EPOC, t=TRANGE, tref=1)
```

And that's it! Your data is now in Python. The rest of the code is a simple plotting example. First, we'll find matching timestamps for our selected sort code (unit).

```
ts = raster_data.snips[SNIP_STORE].ts
if SORTCODE != 0:
    i = np.where(raster_data.snips[SNIP_STORE].sortcode == SORTCODE)[0]
    ts = ts[i]
if len(ts) == 0:
    raise Exception('no matching timestamps found')

num_trials = raster_data.time_ranges.shape[1]
```

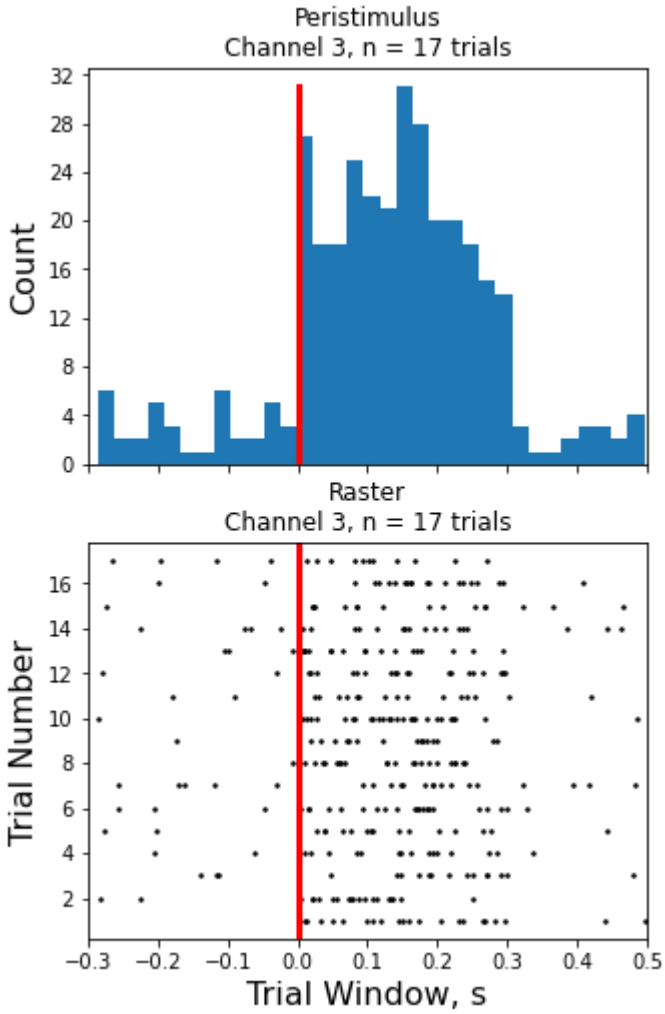
Make the histogram plot

```
fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True, figsize=(5, 8))

hist_ts = hist_data.snips[SNIP_STORE].ts
nbins = np.int64(np.floor(len(hist_ts)/10.))
hist_n = ax1.hist(hist_ts, nbins)[0]

ax1.axis('tight')
ax1.set_xlim(left=TRANGE[0], right=TRANGE[0]+TRANGE[1])
ax1.set_ylabel('Count',fontsize=16)
ax1.set_title('Peristimulus\nChannel {0}, n = {1} trials'.format(CHANNEL,
num_trials))
ax1.yaxis.set_major_locator(MaxNLocator(integer=True))

# Draw a vertical line at t=0.
ax1.plot([0, 0], [0, np.max(hist_n)], 'r-', linewidth=3)
plt.show()
```



Creating the Raster Plot

```
# For the raster plot, make an array of lists containing timestamps for each
trial.
all_ts = [[] for x in range(num_trials)]
all_y = [[] for x in range(num_trials)]
for trial in range(num_trials):
    trial_on = raster_data.time_ranges[0, trial]
    trial_off = raster_data.time_ranges[1, trial]
    ind1 = ts >= trial_on
    ind2 = ts < trial_off
    trial_ts = ts[ind1 & ind2]
    all_ts[trial] = trial_ts - trial_on + TRANGE[0]
    all_y[trial] = (trial+1) * np.ones(len(trial_ts))

all_x = np.concatenate(all_ts)
all_y = np.concatenate(all_y)

# Make the raster plot.
ax2.plot(all_x, all_y, 'k.', markersize=3)
ax2.axis('tight')
ax2.set_xlim(left=TRANGE[0], right=TRANGE[0]+TRANGE[1])
ax2.set_xlabel('Trial Window, s',fontsize=16)
ax2.set_ylabel('Trial Number',fontsize=16)
ax2.set_title('Raster\nChannel {0}, n = {1} trials'.format(CHANNEL,
num_trials))

# Draw a vertical line at t=0.
ax2.plot([0, 0], [0, trial+2], 'r-', linewidth=3)

ax2.yaxis.set_major_locator(MaxNLocator(integer=True))

plt.show()
```


Peristimulus
Channel 3, n = 17 trials
Count
Raster
Channel 3, n = 17 trials
Trial Number
Trial Window, s

Note Filter Example

Import streaming EEG data into Python using **read_block**
Filter around behavioral events that were timestamped by the user using the Run-time Notes feature in Synapse, using **epoc_filter**
Plot each occurrence in a subplot organized by Note type
Good for sleep scoring and behavioral discrimination

Housekeeping

Import the tdt package and other python packages we care about

```
# magic for Jupyter
%matplotlib inline

import matplotlib.pyplot as plt  # standard Python plotting library
import numpy as np

import tdt
```

Importing the Data

This example uses our example data sets. To import your own data, replace `BLOCK_PATH` with the full path to your own data block.

In Synapse, you can find the block path in the database. Go to Menu → History. Find your block, then Right-Click → Copy path to clipboard.

```
tdt.download_demo_data()
BLOCK_PATH = 'data/Subject1-180426-120951'
```

```
demo data ready
```

Set up the variables for the data you want to extract.

We will extract channel 1 from the EEG1 stream data store.

```
STORE = 'EEG1'
CHANNEL = 1
ONSET = [-3] # relative onset, in seconds, from the note timestamp
```

Now read the specified data from our block into a Python structure

```
data = tdt.read_block(BLOCK_PATH, channel=CHANNEL)
```

```
Found Synapse note file: data/Subject1-180426-120951\Notes.txt
read from t=0s to t=31.81s
```

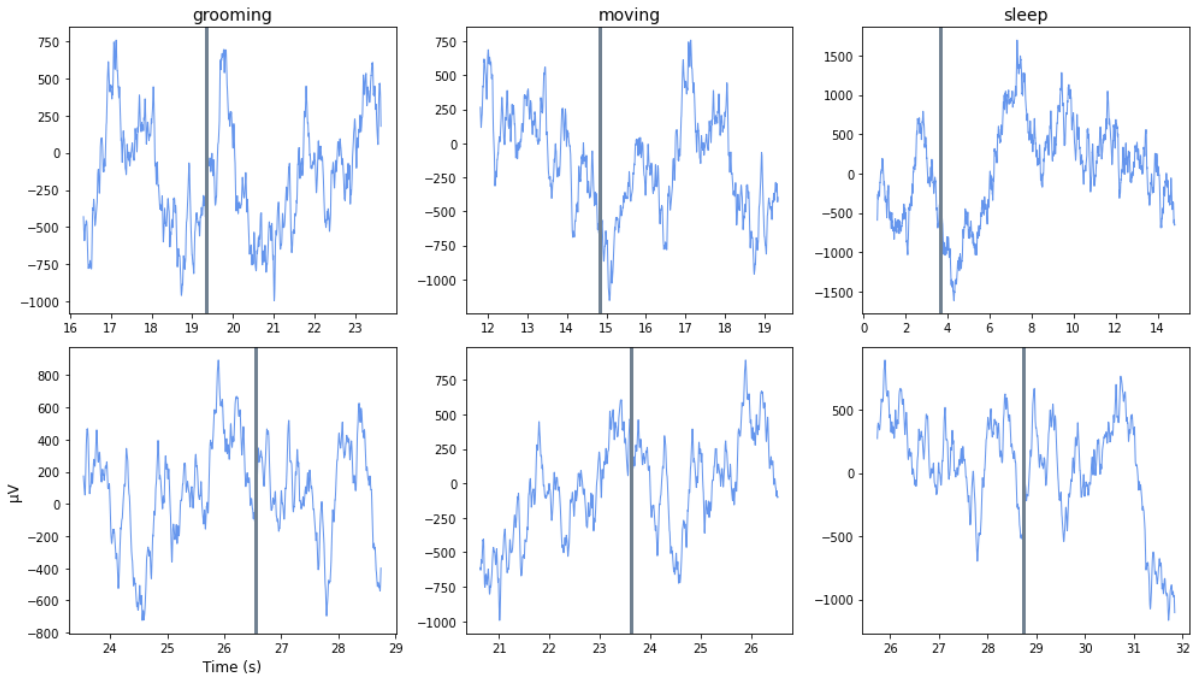
All user notes are stored in a special epoc event called 'Note'

```
# find all the unique note values
notes, counts = np.unique(data.epocs.Note.notes, return_counts=True)

# find the highest number of occurrences (to inform our plot)
maxOccur = np.max(counts)
```

Loop through the notes for plotting

```
# some useful variables
num_notes = len(notes)
fs = data.streams[STORE].fs

fig = plt.figure(figsize=(14, 8))
for ind, note in enumerate(notes,1):

    print('Reading note:', note)

    # look at only the data around this note type
    filtered = tdt.epoc_filter(data, 'Note', values=note, t=ONSET)

    # for each note occurrence, plot the data from
    # the note onset to the next note onset
    n = len(filtered.streams[STORE].filtered)
    for j in range(n):
        plotInd = j * num_notes + ind
        ax = fig.add_subplot(maxOccur, num_notes, plotInd)

        # x-axis is the valid time ranges, in seconds
        len_wav = len(filtered.streams[STORE].filtered[j])
        ts = filtered.time_ranges[0,j] + np.linspace(1, len_wav, len_wav) / fs

        # plot the snippet, in microvolts
        y = np.transpose(1e6 * filtered.streams[STORE].filtered[j])
        trace1 = ax.plot(ts, y, lw=1, color='cornflowerblue')

        # if we specified an ONSET, draw the vertical line at the note onset
        if ONSET != 0:
            trace2 = ax.axvline(x = (ts[0]-ONSET),
                                color='slategray',
                                linewidth=3)

        # plot labels
        if j == 0:
            ax.set_title(note,fontsize=14)
        elif j == (n-1):
            if ind == 1:
                ax.set_ylabel('\u03BCV',fontsize=12)
                ax.set_xlabel('Time (s)',fontsize=12)

fig.tight_layout()
```

```
Reading note: grooming
Reading note: moving
Reading note: sleep
```


grooming
moving
sleep
µV
Time (s)

Fiber Photometry Epoch Averaging Example

This example goes through fiber photometry analysis using techniques such as data smoothing, bleach detrending, and z-score analysis.
The epoch averaging was done using `epoc_filter`.

Author Contributions:
TDT, David Root, and the Morales Lab contributed to the writing and/or conceptualization of the code.
The signal processing pipeline was inspired by the workflow developed by David Barker et al. (2017) for the Morales Lab.
The data used in the example were provided by David Root.

Author Information:
David H. Root
Assistant Professor
Department of Psychology & Neuroscience
University of Colorado, Boulder
Lab Website: https://www.root-lab.org
david.root@colorado.edu

About the authors:
The Root lab and Morales lab investigate the neurobiology of reward, aversion, addiction, and depression.

TDT edits all user submissions in coordination with the contributing author(s) prior to publishing.

Housekeeping

Import the tdt package and other python packages we care about.

```
# magic for Jupyter
%matplotlib inline

#import the read_block function from the tdt package
#also import other python packages we care about
import numpy as np
from sklearn.metrics import auc
import matplotlib.pyplot as plt  # standard Python plotting library
import scipy.stats as stats

import tdt
```

Importing the Data

This example uses our example data sets. To import your own data, replace `BLOCK_PATH` with the full path to your own data block.

In Synapse, you can find the block path in the database. Go to Menu → History. Find your block, then Right-Click → Copy path to clipboard.

```
tdt.download_demo_data()
BLOCKPATH = 'data/FiPho-180416'
```

```
demo data ready
```

```
# Jupyter has a bug that requires import of matplotlib outside of cell with
# matplotlib inline magic to properly apply rcParams
import matplotlib
matplotlib.rcParams['font.size'] = 16 #set font size for all plots
```

Setup the variables for the data you want to extract

We will extract two different stream stores surrounding the 'PtAB' epoch event. We are interested in a specific event code for the shock onset.

```
REF_EPOC = 'PtAB' #event store name. This holds behavioral codes that are
# read through ports A & B on the front of the RZ
SHOCK_CODE = [64959] #shock onset event code we are interested in

# make some variables up here to so if they change in new recordings you won't
# have to change everything downstream
ISOS = '_4054' # 405nm channel. Formally STREAM_STORE1 in maltab example
GCaMP = '_4654' # 465nm channel. Formally STREAM_STORE2 in maltab example
TRANGE = [-10, 20] # window size [start time relative to epoc onset, window
duration]
BASELINE_PER = [-10, -6] # baseline period within our window
ARTIFACT = float("inf") # optionally set an artifact rejection level

#call read block - new variable 'data' is the full data structure
data = tdt.read_block(BLOCKPATH)
```

```
read from t=0s to t=583.86s
```

Use epoc_filter to extract data around our epoc event

Using the 't' parameter extracts data only from the time range around our epoc event.
Use the 'values' parameter to specify allowed values of the REF_EPOC to extract.
For stream events, the chunks of data are stored in cell arrays structured as `data.streams[GCaMP].filtered`

```
data = tdt.epoc_filter(data, REF_EPOC, t=TRANGE, values=SHOCK_CODE)
```

```
# Optionally remove artifacts. If any waveform is above ARTIFACT level, or
# below -ARTIFACT level, remove it from the data set.
total1 = np.size(data.streams[GCaMP].filtered)
total2 = np.size(data.streams[ISOS].filtered)

# List comprehension checking if any single array in 2D filtered array is >
Artifact or < -Artifact
data.streams[GCaMP].filtered = [x for x in data.streams[GCaMP].filtered
                                if not np.any(x > ARTIFACT) or np.any(x < -
ARTIFACT)]
data.streams[ISOS].filtered = [x for x in data.streams[ISOS].filtered
                               if not np.any(x > ARTIFACT) or np.any(x < -
ARTIFACT)]

# Get the total number of rejected arrays
bad1 = total1 - np.size(data.streams[GCaMP].filtered)
bad2 = total2 - np.size(data.streams[ISOS].filtered)
total_artifacts = bad1 + bad2
```

Applying a time filter to a uniformly sampled signal means that the length of each segment could vary by one sample. Let's find the minimum length so we can trim the excess off before calculating the mean.

```
# More examples of list comprehensions
min1 = np.min([np.size(x) for x in data.streams[GCaMP].filtered])
min2 = np.min([np.size(x) for x in data.streams[ISOS].filtered])
data.streams[GCaMP].filtered = [x[1:min1] for x in
data.streams[GCaMP].filtered]
data.streams[ISOS].filtered = [x[1:min2] for x in data.streams[ISOS].filtered]

# Downsample and average 10x via a moving window mean
N = 10 # Average every 10 samples into 1 value
F405 = []
F465 = []
for lst in data.streams[ISOS].filtered:
    small_lst = []
    for i in range(0, min2, N):
        small_lst.append(np.mean(lst[i:i+N-1])) # This is the moving window
mean
    F405.append(small_lst)

for lst in data.streams[GCaMP].filtered:
    small_lst = []
    for i in range(0, min1, N):
        small_lst.append(np.mean(lst[i:i+N-1]))
    F465.append(small_lst)

#Create a mean signal, standard error of signal, and DC offset
meanF405 = np.mean(F405, axis=0)
stdF405 = np.std(F405, axis=0)/np.sqrt(len(data.streams[ISOS].filtered))
dcF405 = np.mean(meanF405)
meanF465 = np.mean(F465, axis=0)
stdF465 = np.std(F465, axis=0)/np.sqrt(len(data.streams[GCaMP].filtered))
dcF465 = np.mean(meanF465)
```

Plot epoc averaged response

```
# Create the time vector for each stream store
ts1 = TRANGE[0] + np.linspace(1, len(meanF465), len(meanF465))/
data.streams[GCaMP].fs*N
ts2 = TRANGE[0] + np.linspace(1, len(meanF405), len(meanF405))/
data.streams[ISOS].fs*N

# Subtract DC offset to get signals on top of one another
meanF405 = meanF405 - dcF405
meanF465 = meanF465 - dcF465

# Start making a figure with 4 subplots
# First plot is the 405 and 465 averaged signals
fig = plt.figure(figsize=(9, 14))
ax0 = fig.add_subplot(411) # work with axes and not current plot (plt.)

# Plotting the traces
p1, = ax0.plot(ts1, meanF465, linewidth=2, color='green', label='GCaMP')
p2, = ax0.plot(ts2, meanF405, linewidth=2, color='blueviolet', label='ISOS')

# Plotting standard error bands
p3 = ax0.fill_between(ts1, meanF465+stdF465, meanF465-stdF465,
                      facecolor='green', alpha=0.2)
p4 = ax0.fill_between(ts2, meanF405+stdF405, meanF405-stdF405,
                      facecolor='blueviolet', alpha=0.2)

# Plotting a line at t = 0
p5 = ax0.axvline(x=0, linewidth=3, color='slategray', label='Shock Onset')

# Finish up the plot
ax0.set_xlabel('Seconds')
ax0.set_ylabel('mV')
ax0.set_title('Foot Shock Response, %i Trials (%i Artifacts Removed)'
              % (len(data.streams[GCaMP].filtered), total_artifacts))
ax0.legend(handles=[p1, p2, p5], loc='upper right')
ax0.set_ylim(min(np.min(meanF465-stdF465), np.min(meanF405-stdF405)),
             max(np.max(meanF465+stdF465), np.max(meanF405+stdF405)))
ax0.set_xlim(TRANGE[0], TRANGE[1]+TRANGE[0]);

plt.close() # Jupyter cells will output any figure calls made, so if you
don't want to see it just yet, close existing axis
            # https://stackoverflow.com/questions/18717877/prevent-plot-from-
showing-in-jupyter-notebook
            # Note that this is not good code practice - Jupyter lends it
self to these types of bad workarounds
```

Fitting 405 channel onto 465 channel to detrend signal bleaching

Scale and fit data. Algorithm sourced from Tom Davidson's Github: https://github.com/tjd2002/tjd-shared-code/blob/master/matlab/photometry/FP_normalize.m

```
Y_fit_all = []
Y_dF_all = []
for x, y in zip(F405, F465):
    x = np.array(x)
    y = np.array(y)
    bls = np.polyfit(x, y, 1)
    fit_line = np.multiply(bls[0], x) + bls[1]
    Y_fit_all.append(fit_line)
    Y_dF_all.append(y-fit_line)

# Getting the z-score and standard error
zall = []
for dF in Y_dF_all:
   ind = np.where((np.array(ts2)<BASELINE_PER[1]) &
(np.array(ts2)>BASELINE_PER[0]))
   zb = np.mean(dF[ind])
   zsd = np.std(dF[ind])
   zall.append((dF - zb)/zsd)

zerror = np.std(zall, axis=0)/np.sqrt(np.size(zall, axis=0))
```

Heat Map based on z score of 405 fit subtracted 465

```
ax1 = fig.add_subplot(412)
cs = ax1.imshow(zall, cmap=plt.cm.Greys, interpolation='none', aspect="auto",
                extent=[TRANGE[0], TRANGE[1]+TRANGE[0], 0,
len(data.streams[GCaMP].filtered)])
cbar = fig.colorbar(cs, pad=0.01, fraction=0.02)

ax1.set_title('Individual z-Score Traces')
ax1.set_ylabel('Trials')
ax1.set_xlabel('Seconds from Shock Onset')

plt.close() # Suppress figure output again
```

Plot the z-score trace for the 465 with std error bands

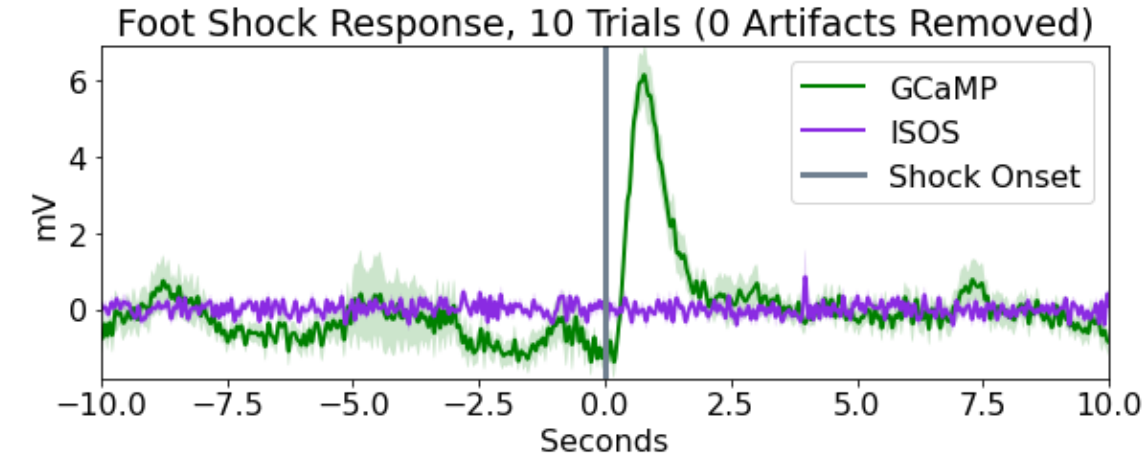
```
ax2 = fig.add_subplot(413)
p6 = ax2.plot(ts2, np.mean(zall, axis=0), linewidth=2, color='green',
label='GCaMP')
p7 = ax2.fill_between(ts1, np.mean(zall, axis=0)+zerror
                      ,np.mean(zall, axis=0)-zerror, facecolor='green',
alpha=0.2)
p8 = ax2.axvline(x=0, linewidth=3, color='slategray', label='Shock Onset')
ax2.set_ylabel('z-Score')
ax2.set_xlabel('Seconds')
ax2.set_xlim(TRANGE[0], TRANGE[1]+TRANGE[0])
ax2.set_title('Foot Shock Response')

plt.close()
```

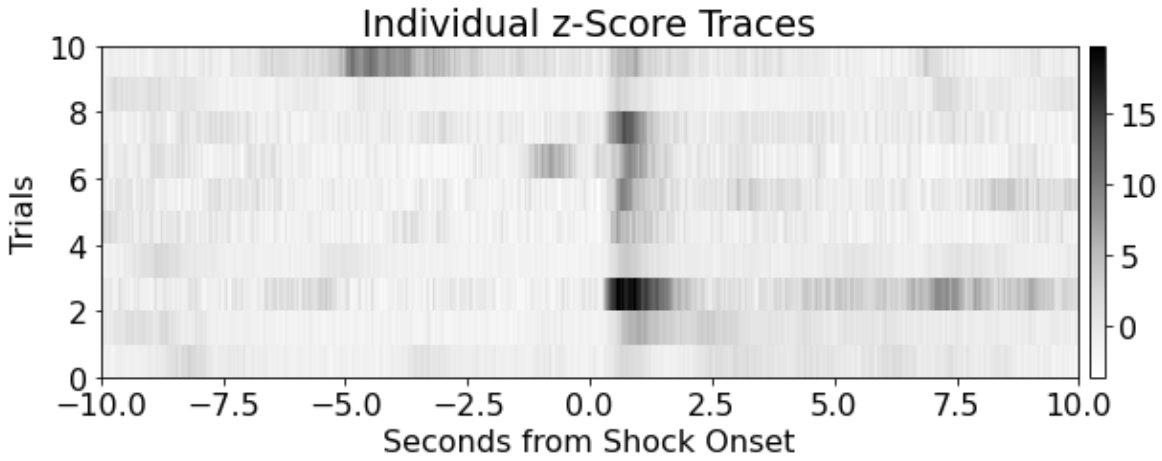
Quantify changes as an area under the curve for cue (-5 sec) vs shock (0 sec)

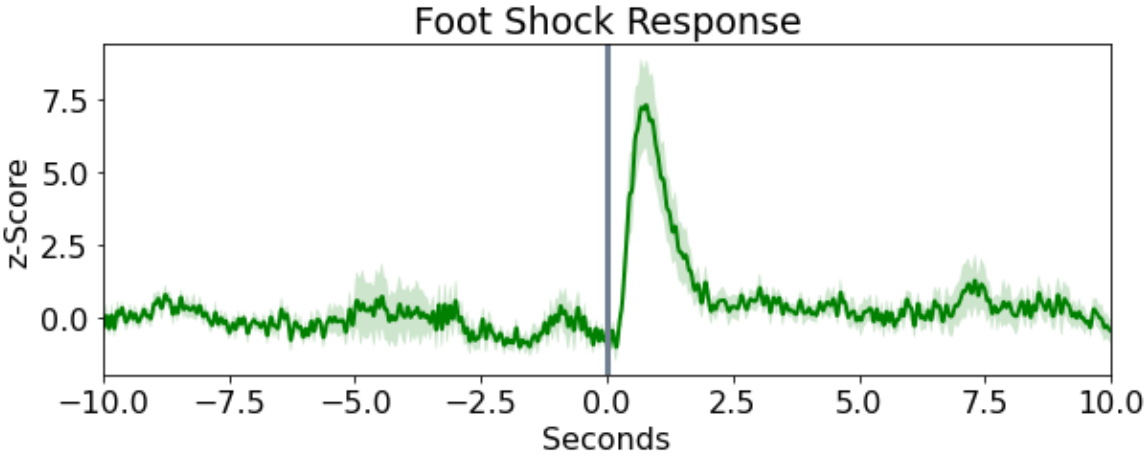
```
AUC = [] # cue, shock
ind1 = np.where((np.array(ts2)<-3) & (np.array(ts2)>-5))
AUC1= auc(ts2[ind1], np.mean(zall, axis=0)[ind1])
ind2 = np.where((np.array(ts2)>0) & (np.array(ts2)<2))
AUC2= auc(ts2[ind2], np.mean(zall, axis=0)[ind2])
AUC.append(AUC1)
AUC.append(AUC2)

# Run a two-sample T-test
t_stat,p_val = stats.ttest_ind(np.mean(zall, axis=0)[ind1],
                               np.mean(zall, axis=0)[ind2], equal_var=False)
```

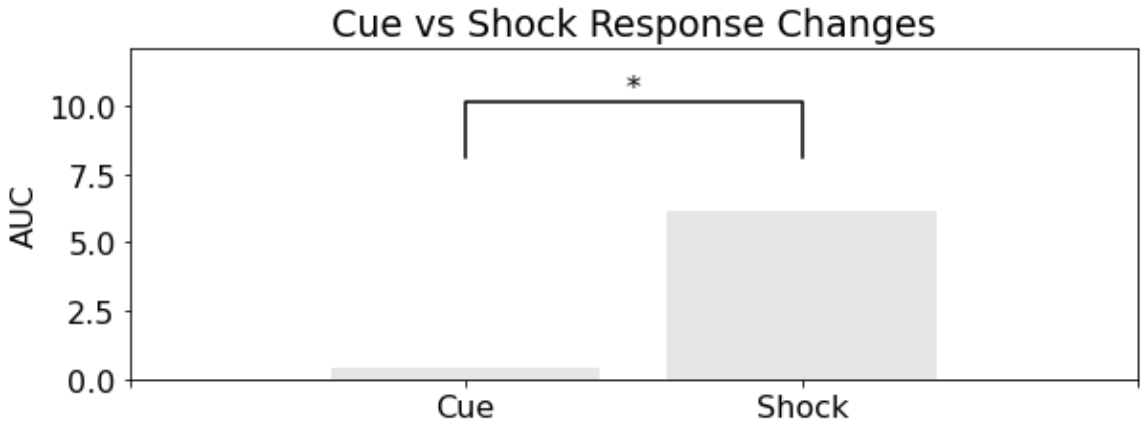
Make a bar plot

```
ax3 = fig.add_subplot(414)
p9 = ax3.bar(np.arange(len(AUC)), AUC, color=[.8, .8, .8], align='center',
alpha=0.5)

# statistical annotation
x1, x2 = 0, 1 # columns indices for labels
y, h, col = max(AUC) + 2, 2, 'k'
ax3.plot([x1, x1, x2, x2], [y, y+h, y+h, y], lw=1.5, c=col)
p10 = ax3.text((x1+x2)*.5, y+h, "*", ha='center', va='bottom', color=col)

# Finish up the plot
ax3.set_ylim(0,y+2*h)
ax3.set_ylabel('AUC')
ax3.set_title('Cue vs Shock Response Changes')
ax3.set_xticks(np.arange(-1, len(AUC)+1))
ax3.set_xticklabels(['','Cue','Shock',''])

fig.tight_layout()
fig
```


Foot Shock Response, 10 Trials (0 Artifacts Removed)
GCaMP
ISOS
Shock Onset
mV
Seconds


Individual z-Score Traces
Trials
Seconds from Shock Onset


Foot Shock Response
z-Score
Seconds


Cue vs Shock Response Changes
AUC
*
Cue
Shock

Licking Bout Epoc Filtering

This example looks at fiber photometry data in the VTA where subjects are provided sucrose water after a fasting period.
Lick events are captured as TTL pulses.
Objective is to combine many consecutive licking events into a single event based on time difference and lick count thresholds.
New lick bout events can then be used for clear peri-event filtering.

Housekeeping

Import the tdt package and other python packages we care about.

```
# Jupyter magic
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt  # standard Python plotting library

# import the tdt library
import tdt
```

Importing the Data

This example uses our example data sets. To import your own data, replace `BLOCK_PATH` with the full path to your own data block.

In Synapse, you can find the block path in the database. Go to Menu → History. Find your block, then Right-Click → Copy path to clipboard.

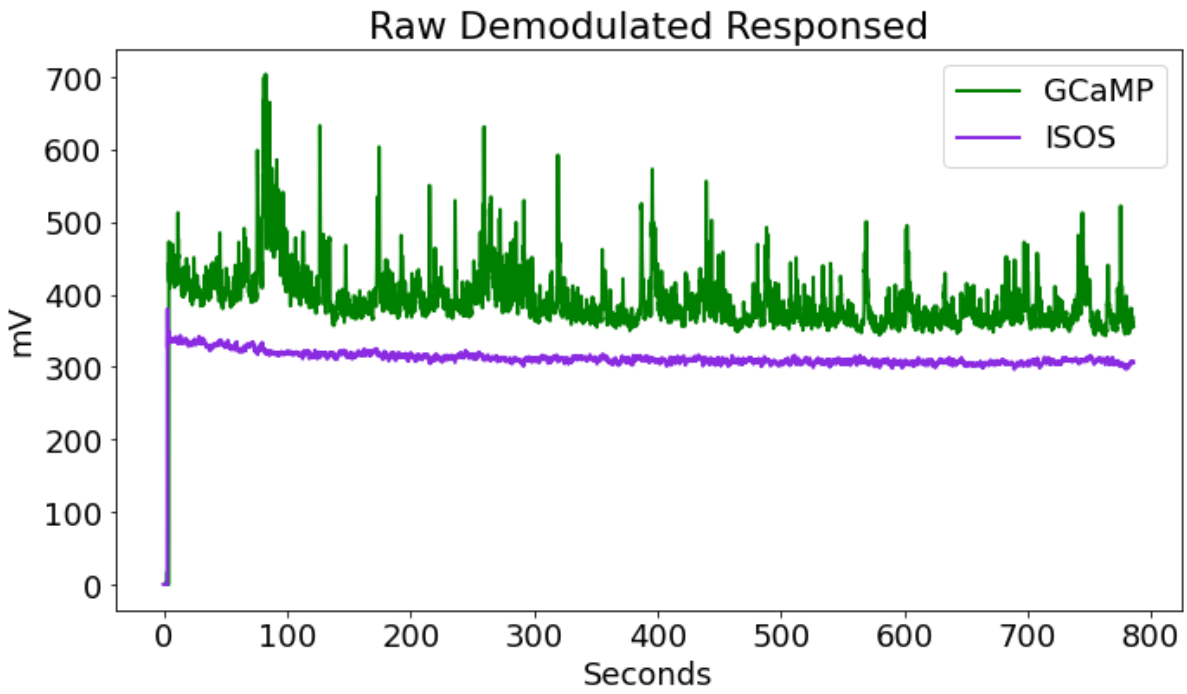
```
tdt.download_demo_data()
BLOCKPATH = 'data/VTA4-190125-100559'
data = tdt.read_block(BLOCKPATH)
```

```
demo data ready
Found Synapse note file: data/VTA4-190125-100559\Notes.txt
read from t=0s to t=785.44s
```

Basic plotting and artifact removal

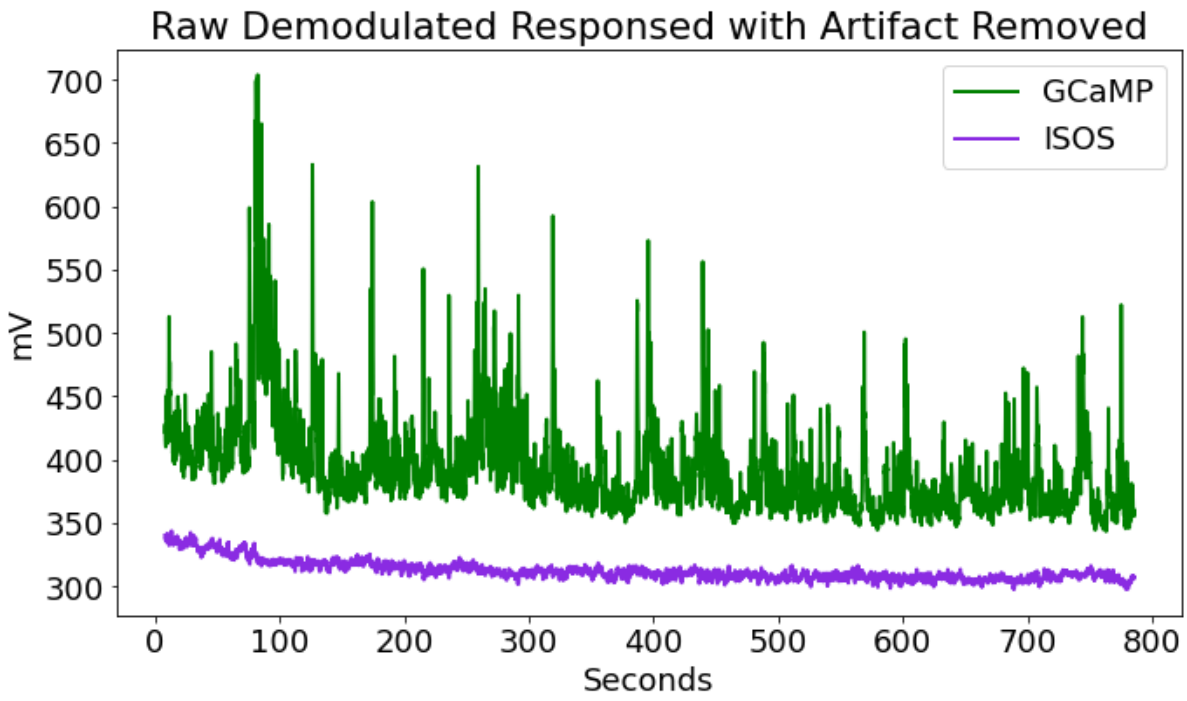
```
#Jupyter has a bug that requires import of matplotlib outside of cell with
matplotlib inline magic to properly apply rcParams

import matplotlib
matplotlib.rcParams['font.size'] = 18 # set font size for all figures

# Make some variables up here to so if they change in new recordings you
won't have to change everything downstream
GCAMP = '_480G' # GCaMP channel
ISOS = '_405G' # Isosbestic channel
LICK = 'Ler_'
```

```
# Make a time array based on the number of samples and sample freq of
# the demodulated streams
time = np.linspace(1,len(data.streams[GCAMP].data),
len(data.streams[GCAMP].data))/data.streams[GCAMP].fs

# Plot both unprocessed demodulated stream
fig1 = plt.figure(figsize=(10,6))
ax0 = fig1.add_subplot(111)

# Plotting the traces
p1, = ax0.plot(time, data.streams[GCAMP].data, linewidth=2, color='green',
label='GCaMP')
p2, = ax0.plot(time, data.streams[ISOS].data, linewidth=2, color='blueviolet',
label='ISOS')

ax0.set_ylabel('mV')
ax0.set_xlabel('Seconds')
ax0.set_title('Raw Demodulated Responsed')
ax0.legend(handles=[p1,p2], loc='upper right')
fig1.tight_layout()

# Jupyter for some reason (sometimes) shows the figure without be called,
# Likely when plt.figure() is called
# otherwise you would call fig in a line by itself like:
# fig
```



Artifact Removal

```
# There is often a large artifact on the onset of LEDs turning on
# Remove data below a set time t
t = 8
inds = np.where(time>t)
ind = inds[0][0]
time = time[ind:] # go from ind to final index
data.streams[GCAMP].data = data.streams[GCAMP].data[ind:]
data.streams[ISOS].data = data.streams[ISOS].data[ind:]

# Plot again at new time range
fig2 = plt.figure(figsize=(10, 6))
ax1 = fig2.add_subplot(111)

# Plotting the traces
p1, = ax1.plot(time,data.streams[GCAMP].data, linewidth=2, color='green',
label='GCaMP')
p2, = ax1.plot(time,data.streams[ISOS].data, linewidth=2, color='blueviolet',
label='ISOS')

ax1.set_ylabel('mV')
ax1.set_xlabel('Seconds')
ax1.set_title('Raw Demodulated Responsed with Artifact Removed')
ax1.legend(handles=[p1,p2],loc='upper right')
fig2.tight_layout()
# fig
```


Raw Demodulated Responsed with Artifact Removed
GCaMP
ISOS
mV
700
650
600
550
500
450
400
350
300
0
100
200
300
400
500
600
700
800
Seconds

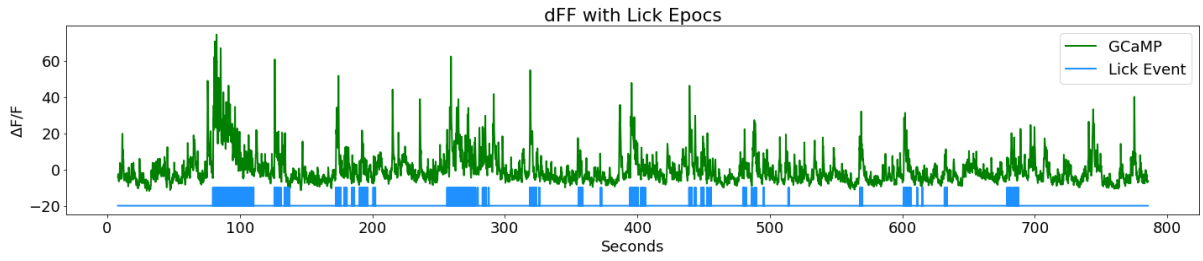
Downsample Data Doing Local Averaging

```
# Average around every Nth point and downsample Nx
N = 10 # Average every 10 samples into 1 value
F405 = []
F465 = []

for i in range(0, len(data.streams[GCAMP].data), N):
    F465.append(np.mean(data.streams[GCAMP].data[i:i+N-1])) # This is the
moving window mean
data.streams[GCAMP].data = F465

for i in range(0, len(data.streams[ISOS].data), N):
    F405.append(np.mean(data.streams[ISOS].data[i:i+N-1]))
data.streams[ISOS].data = F405

#decimate time array to match length of demodulated stream
time = time[::N] # go from beginning to end of array in steps on N
time = time[:len(data.streams[GCAMP].data)]

# Detrending and dFF
# Full trace dFF according to Lerner et al. 2015
# https://dx.doi.org/10.1016/j.cell.2015.07.014
# dFF using 405 fit as baseline

x = np.array(data.streams[ISOS].data)
y = np.array(data.streams[GCAMP].data)
bls = np.polyfit(x, y, 1)
Y_fit_all = np.multiply(bls[0], x) + bls[1]
Y_dF_all = y - Y_fit_all

dFF = np.multiply(100, np.divide(Y_dF_all, Y_fit_all))
std_dFF = np.std(dFF)
```

Turn Licking Events into Lick Bouts

```
# First make a continous time series of Licking TTL events (epocs) and plot
LICK_on = data.epocs[LICK].onset
LICK_off = data.epocs[LICK].offset
# Add the first and last time stamps to make tails on the TTL stream
LICK_x = np.append(np.append(time[0], np.reshape(np.kron([LICK_on, LICK_off],
                   np.array([[1], [1]])).T, [1,-1])[0]), time[-1])
sz = len(LICK_on)
d = data.epocs[LICK].data
# Add zeros to beginning and end of 0,1 value array to match len of LICK_x
LICK_y = np.append(np.append(0,np.reshape(np.vstack([np.zeros(sz),
    d, d, np.zeros(sz)]).T, [1, -1])[0]),0)

y_scale = 10 #adjust according to data needs
y_shift = -20 #scale and shift are just for aesthetics
```

```
# First subplot in a series: dFF with lick epocs
fig3 = plt.figure(figsize=(20,12))
ax2 = fig3.add_subplot(311)

p1, = ax2.plot(time, dFF, linewidth=2, color='green', label='GCaMP')
p2, = ax2.plot(LICK_x, y_scale*LICK_y+y_shift, linewidth=2,
color='dodgerblue', label='Lick Event')
ax2.set_ylabel(r'$\Delta$F/F')
ax2.set_xlabel('Seconds')
ax2.set_title('dFF with Lick Epocs')
ax2.legend(handles=[p1,p2], loc='upper right')
fig3.tight_layout()
```



Lick Bout Logic

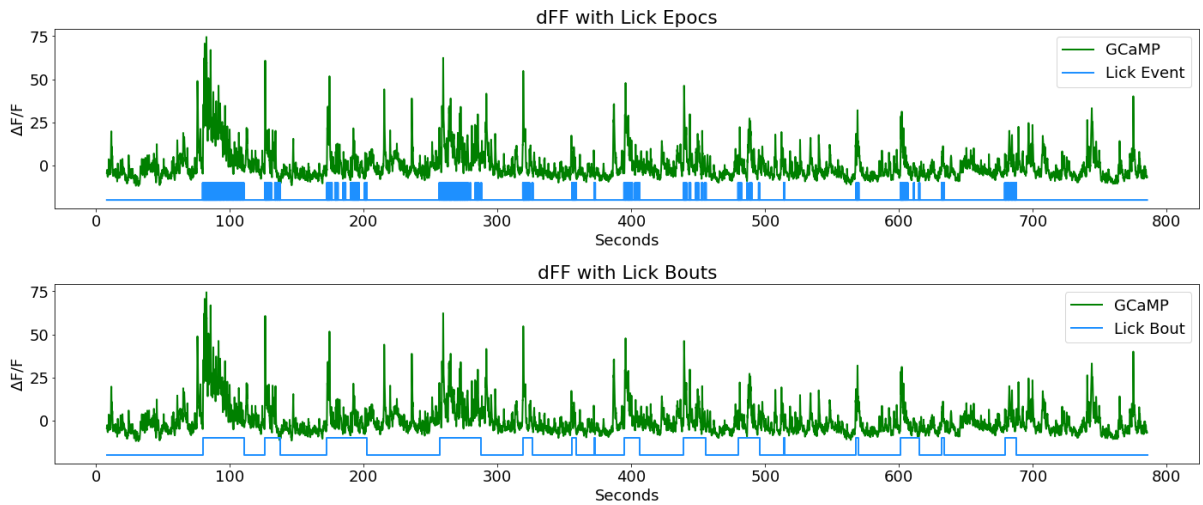
Now combine lick epocs that happen in close succession to make a single on/off event (a lick BOUT). Top view logic: if difference between consecutive lick onsets is below a certain time threshold and there was more than one lick in a row, then consider it as one bout, otherwise it is its own bout. Also, make sure a minimum number of licks was reached to call it a bout.

```
LICK_EVENT = 'LICK_EVENT'

LICK_DICT = {
        "name":LICK_EVENT,
        "onset":[],
        "offset":[],
        "type_str":data.epocs[LICK].type_str,
        "data":[]
        }
# pass StructType our new dictionary to make keys and values
data.epocs.LICK_EVENT = tdt.StructType(LICK_DICT)

lick_on_diff = np.diff(data.epocs[LICK].onset)
BOUT_TIME_THRESHOLD = 10
lick_diff_ind = np.where(lick_on_diff >= BOUT_TIME_THRESHOLD)[0]
#for some reason np.where returns a 2D array, hence the [0]

# Make an onset/ offset array based on threshold indicies
diff_ind = 0
for ind in lick_diff_ind:
    # BOUT onset is thresholded onset index of lick epoc event
    data.epocs[LICK_EVENT].onset.append(data.epocs[LICK].onset[diff_ind])
    # BOUT offset is thresholded offset of lick event before next onset
    data.epocs[LICK_EVENT].offset.append(data.epocs[LICK].offset[ind])
    # set the values for data, arbitrary 1
    data.epocs[LICK_EVENT].data.append(1)
    diff_ind = ind + 1

# special case for last event to handle lick event offset indexing
data.epocs[LICK_EVENT].onset.append(data.epocs[LICK].onset[lick_diff_ind[-1]
+1])
data.epocs[LICK_EVENT].offset.append(data.epocs[LICK].offset[-1])
data.epocs[LICK_EVENT].data.append(1)

# Now determine if it was a 'real' lick bout by thresholding by some
# user-set number of licks in a row
MIN_LICK_THRESH = 4 #four licks or more make a bout
licks_array = []

# Find number of licks in licks_array between onset and offset of
# our new lick BOUT LICK_EVENT
for on, off in
zip(data.epocs[LICK_EVENT].onset,data.epocs[LICK_EVENT].offset):
    licks_array.append(
        len(np.where((data.epocs[LICK].onset >= on) & (data.epocs[LICK].onset
<= off))[0]))

# Remove onsets, offsets, and data of thrown out events
licks_array = np.array(licks_array)
inds = np.where(licks_array<MIN_LICK_THRESH)[0]
```

```
for index in sorted(inds, reverse=True):
    del data.epocs[LICK_EVENT].onset[index]
    del data.epocs[LICK_EVENT].offset[index]
    del data.epocs[LICK_EVENT].data[index]

# Make a continuous time series for lick BOUTS for plotting
LICK_EVENT_on = data.epocs[LICK_EVENT].onset
LICK_EVENT_off = data.epocs[LICK_EVENT].offset
LICK_EVENT_x = np.append(time[0], np.append(
    np.reshape(np.kron([LICK_EVENT_on, LICK_EVENT_off],np.array([[1],
[1]])).T, [1,-1])[0], time[-1]))
sz = len(LICK_EVENT_on)
d = data.epocs[LICK_EVENT].data
LICK_EVENT_y = np.append(np.append(
    0, np.reshape(np.vstack([np.zeros(sz), d, d, np.zeros(sz)]).T, [1 ,-1])
[0]), 0)
```

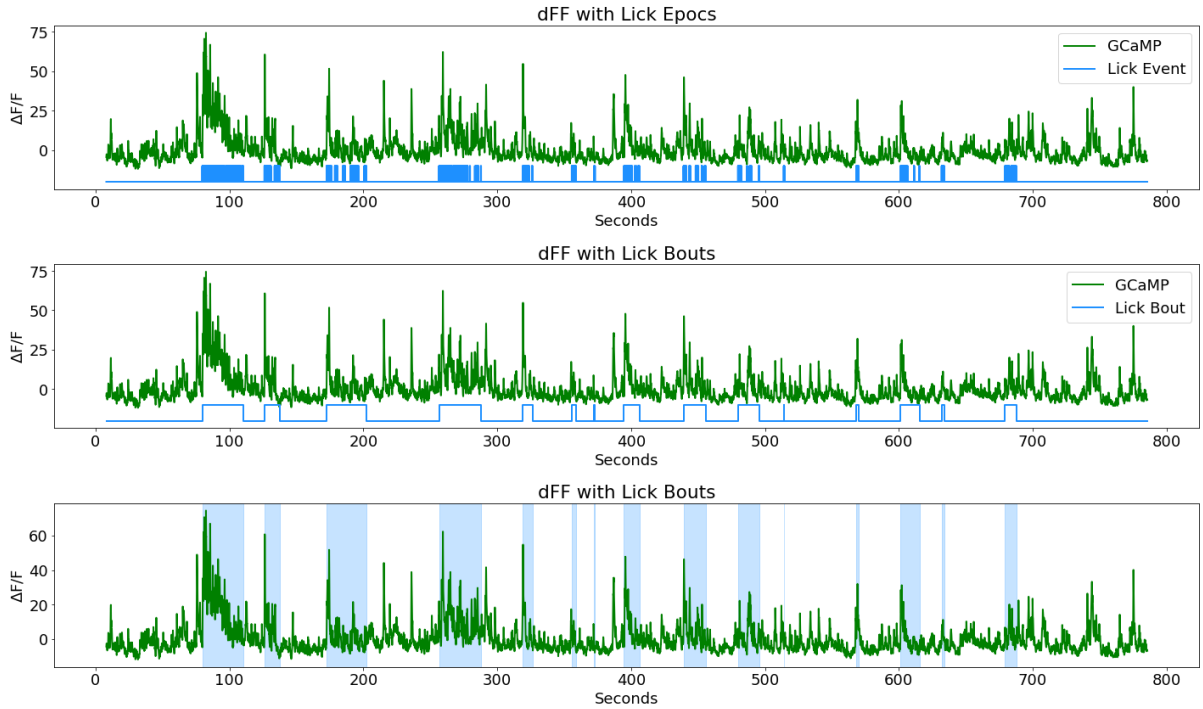
Plot dFF with newly defined lick bouts

```
ax3 = fig3.add_subplot(312)
p1, = ax3.plot(time, dFF, linewidth=2, color='green', label='GCaMP')
p2, = ax3.plot(LICK_EVENT_x, y_scale*LICK_EVENT_y+y_shift, linewidth=2,
color='dodgerblue', label='Lick Bout')
ax3.set_ylabel(r'$\Delta$F/F')
ax3.set_xlabel('Seconds')
ax3.set_title('dFF with Lick Bouts')
ax3.legend(handles=[p1, p2], loc='upper right')
fig3.tight_layout()
fig3
```



Make nice area fills instead of epocs for aesthetics

```
ax4 = fig3.add_subplot(313)
p1, = ax4.plot(time, dFF,linewidth=2, color='green', label='GCaMP')
for on, off in zip(data.epocs[LICK_EVENT].onset,
data.epocs[LICK_EVENT].offset):
    ax4.axvspan(on, off, alpha=0.25, color='dodgerblue')
ax4.set_ylabel(r'$\Delta$F/F')
ax4.set_xlabel('Seconds')
ax4.set_title('dFF with Lick Bouts')
fig3.tight_layout()
fig3
```



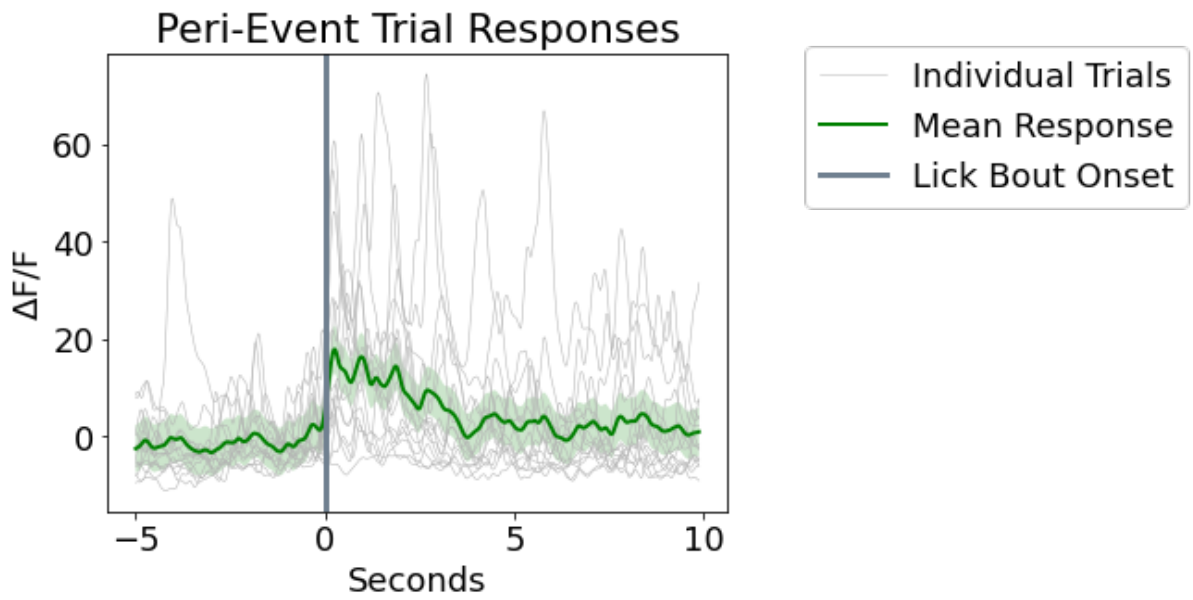
Time Filter Around Lick Bout Epocs

Note that we are using dFF of the full time series, not peri-event dFF where f0 is taken from a pre-event basaeline period.

```
PRE_TIME = 5 # five seconds before event onset
POST_TIME = 10 # ten seconds after
fs = data.streams[GCAMP].fs/N #recall we downsampled by N = 10 earlier

# time span for peri-event filtering, PRE and POST, in samples
TRANGE = [-PRE_TIME*np.floor(fs), POST_TIME*np.floor(fs)]

dFF_snips = []
array_ind = []
pre_stim = []
post_stim = []

for on in data.epocs[LICK_EVENT].onset:
    # If the bout cannot include pre-time seconds before event, make zero
    if on < PRE_TIME:
        dFF_snips.append(np.zeros(TRANGE[1]-TRANGE[0]))
    else:
        # find first time index after bout onset
        array_ind.append(np.where(time > on)[0][0])
        # find index corresponding to pre and post stim durations
        pre_stim.append(array_ind[-1] + TRANGE[0])
        post_stim.append(array_ind[-1] + TRANGE[1])
        dFF_snips.append(dFF[int(pre_stim[-1]):int(post_stim[-1])])

# Make all snippets the same size based on min snippet length
min1 = np.min([np.size(x) for x in dFF_snips])
dFF_snips = [x[1:min1] for x in dFF_snips]

mean_dFF_snips = np.mean(dFF_snips, axis=0)
std_dFF_snips = np.std(mean_dFF_snips, axis=0)

peri_time = np.linspace(1, len(mean_dFF_snips), len(mean_dFF_snips))/fs -
PRE_TIME
```

Make a Peri-Event Stimulus Plot and Heat Map

```
fig4 = plt.figure(figsize=(6,10))
ax5 = fig4.add_subplot(211)

for snip in dFF_snips:
    p1, = ax5.plot(peri_time, snip, linewidth=.5, color=[.7, .7, .7],
label='Individual Trials')
p2, = ax5.plot(peri_time, mean_dFF_snips, linewidth=2, color='green',
label='Mean Response')

# Plotting standard error bands
p3 = ax5.fill_between(peri_time, mean_dFF_snips+std_dFF_snips,
                      mean_dFF_snips-std_dFF_snips, facecolor='green',
alpha=0.2)
p4 = ax5.axvline(x=0, linewidth=3, color='slategray', label='Lick Bout Onset')

ax5.axis('tight')
ax5.set_xlabel('Seconds')
ax5.set_ylabel(r'$\Delta$F/F')
ax5.set_title('Peri-Event Trial Responses')
ax5.legend(handles=[p1, p2, p4], bbox_to_anchor=(1.1, 1.05));
```



```
ax6 = fig4.add_subplot(212)
cs = ax6.imshow(dFF_snips, cmap=plt.cm.Greys,
                interpolation='none', extent=[-
PRE_TIME,POST_TIME,len(dFF_snips),0],)
ax6.set_ylabel('Trial Number')
ax6.set_yticks(np.arange(.5, len(dFF_snips), 2))
ax6.set_yticklabels(np.arange(0, len(dFF_snips), 2))
fig4.colorbar(cs)
fig4
```