span Documentation

Release 0.2

Phillip Cloud

August 30, 2013

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

1.1 Name

span stands for **sp**ike **an**alysis.

1.2 Motivation

span arose out of the need to do the following operations in a reasonably efficient manner:

- Read in TDT files into NumPy arrays.
- Group data arbitrarily for firing rate analyses, e.g., average firing rate over shanks, collapsing across channels (thanks to Pandas).
- Perform cross-correlation analysis on the binned and thresholded spikes, again using arbitrary grouping

1.3 Disclaimer

Naturally, because this is software, if you use it you'll likely find a bug. If you're so inclined please create an issue using the Github tracker and I will attempt to fix it.

1.4 TODO

See the Github issues page.

CHAPTER

TWO

GETTING STARTED

2.1 Caveat Emptor

I developed this code with the bleeding edge versions of everything listed below, so don't be surprised if you try this out on an older version of NumPy (for example) and it doesn't work. When in doubt set up a virtualenv and install the git versions of all the dependencies. This code has only been used and tested on Arch Linux and Red Hat Enterprise Linux 5. If you set up a virtualenv and follow the install instructions there should be no problems.

2.2 Dependencies

1. Python >= 2.6
2. NumPy
3. SciPy
4. Cython
5. pandas
6. six

2.3 Optional Dependencies

1. nose if you want to run the tests
2. sphinx if you want to build the documentation
3. numpydoc if you want to build the documentation with NumPy formatting support (recommended)

2.4 Installation

You should set up a virtualenv when using this code, so that you don't break anything. Personally, I prefer the excellent virtualenvwrapper tool to set up Python environments.

First things first:

```
# change to span's directory
cd wherever/you/downloaded/span

# install dependencies
pip install -r requirements.txt

# install span
python setup.py install
```

or if you don't want to install it and you want to use `span` from its directory then you can do

```
python setup.py build_ext --inplace
python setup.py build
```

For the last one you must set the `PYTHONPATH` environment variable or mess around with `sys.path` (not recommended).

CHAPTER

THREE

READING IN TDT FILES

3.1 TDT File Structure

There are two types of TDT files necessary to create an instance of `span.tdt.spikedataframe.SpikeDataFrame`: one file ending in "tev" and one ending in "tsq". Note that this differs slightly from TDT's definition of a tank.

3.2 TSQ Event Headers

The TSQ file is a C `struct` making it trivial to work with in NumPy using a compound dtype.

According to Jaewon Hwang, the C struct is

```
struct TsqEventHeader {
    long size;
    long type;
    long name;
    unsigned short chan;
    unsigned short sortcode;
    double timestamp;
    union {
        __int64 fp_loc;
        double strobe;
    };

    long format;
    float frequency;
};
```

but this code **will not work** on most modern systems because `long` is implementation defined–the compiler writer defines it. I have not run across a compiler on a 64 bit system that defines `sizeof(long)` to be 32. Thus the most accurate version (and the one used in **span**) is

```
#include <stdint.h>

struct TsqEventHeader {
    int32_t size;
    int32_t type;
    int32_t name;

    uint16_t chan;
    uint16_t sortcode;

    double timestamp;

    union {
        int64_t fp_loc;
        double strobe;
    };

    int32_t format;
    float frequency;
};
```

Warning: If you're using this code on data that were created on a Windows 7 machine then may have to change `int32_t` to `int64_t`. I have not tested this code on data created on a Windows 7 machine so **use at your own risk**.

Reading the TSQ file into NumPy is, fortunately, very easy now that we have this `struct`.

```
import numpy as np
from pandas import DataFrame
from numpy import int32, uint32, uint16, float64, int64, int32, float32

names = ('size', 'type', 'name', 'channel', 'sort_code', 'timestamp',
         'fp_loc', 'strobe', 'format', 'fs')
formats = (int32, int32, uint32, uint16, uint16, float64, int64,
           float64, int32, float32)
offsets = 0, 4, 8, 12, 14, 16, 24, 24, 32, 36
tsq_dtype = np.dtype({'names': names, 'formats': formats,
                      'offsets': offsets}, align=True)
tsq_name = 'name/of/file.tsq'
tsq = np.fromfile(tsq_name, dtype=tsq_dtype)
df = DataFrame(tsq)
```

The variable `tsq` in the above code snippet is a NumPy record array. I personally find these very annoying. Luckily, Wes McKinney created the wonderful pandas library which automatically converts NumPy record arrays into a pandas DataFrame where each field from the record array is now a column in the DataFrame `df`.

3.3 TEV Raw Data

The raw data are contained in the file with the extension ".tev". There is a single function that does the heavy lifting in Cython and the rest is done in pure Python. The basic idea that the `fp_loc` field of the header DataFrame (from the tsq files) contains the location in the tev file of the samples for a particular channel. What's nice about `span` is that it hides this complexity from the user. If you like complexity, then read on.

3.3.1 TL;DR (too long; don't read)

Reading in the Raw Data

Now that we've got the header data we can get what we're really interested in: raw voltage traces. There are some indexing acrobatics here that require a little bit of detail about the tsq file and little bit of knowledge of "group by" style operations.

First off, there is a Cython function that does all of the heavy lifting in terms of reading raw bytes into a NumPy array. What is passed in to that function is important.

The first argument is of course the filename, no surprise there. The second argument is important. This is the numpy array of file locations grouped by channel number. This is an array that contains the file pointer location of each consective chunk of data in the TEV file. That means that if, for example, I want to read all of the data from channel 1 then I would loop over the first column of this array. Since each element is a file pointer location I would seek to that location and read `blocksize` bytes. The Cython function does this automatically for every channel. The third argument is `blocksize` and the fourth argument is the output array that contains the raw voltage data.

Here is the inner loop that does the work of reading in the raw data from the tev file.

```
        for i in prange(n, schedule='static'):
            pos = fp_locs[i]

            if fseek(f, pos, SEEK_SET) == -1:
                free(chunk)
                fclose(f)

                with gil:
                    raise IOError('Unable to seek to file position %d' % pos)

            if not fread(chunk, num_bytes, 1, f):
                free(chunk)
                fclose(f)

                with gil:
                    raise IOError('Unable to read any more bytes from '
```

```
                                '%s' % filename)

        for j in range(block_size):
            spikes[i, j] = chunk[j]
```

You can see here that this part of the `span.tdt._read_tev._read_tev_raw()` function skips to the point in the file where the next chunk lies and placing it in the array `spikes`. This codes works on any kind of floating point spike data (by used fused types and it also runs in parallel for a slight speedup in I/O.

As usual, the best way to understand what's going on is to read the source code.

Organizing the Data

Whew! Reading in these data are tricky.

Now we have a dataset. However it's not properly arranged, meaning the dimensions are not those that make sense from the point of analysis.

I'm not exactly sure how this works, but TDT stores their data in chunks and that chunk size is usually a power of 2.

The number of chunks depends on the length of the recording and is the number of rows in the TSQ array. So, `tsq.shape[0]` equals the number of chunks in the recording.

Now, each chunk has a few properties, which you can explore on your own if you're interested. For now, we'll only concern ourselves with the `channel` (`chan` in the C `struct`) column.

The `channel` column gives each chunk a ... you guessed it ... channel, and thus provides a way to map sample chunks to channels.

3.4 Electrode Array Configuration

See the `span.tdt.recording` module documentation.

3.5 span.tdt.tank

3.6 span.tdt.spikedataframe

CHAPTER

FOUR

SPAN.TDT.RECORDING

`span.tdt.recording` is a module for encapsulating information about the electrode array used in an experiment.

The single class `ElectrodeMap` and the module level function `distance_map()` are exported to allow the user to easily deal with the computation of pairwise distance between electrodes given an electrode map. `distance_map()` is more of a low-level function used by the `ElectrodeMap` class, but I chose to export it anyway for exploration purposes. A few string constants are also exported but can be ignored.

The main features of this module are:

- Encapsulation and visualization of electrode map structure
- Ability to easily compute pairwise distance between electrodes

One thing that I think might be a useful generalization is to add a database of electrode map structures from various vendors (NeuroNexus comes to mind).

An issue that might be considered a bug at worst or an inconvenience that needs to be addressed is that if your electrodes are not numbered from 1 to n, the labels when computing the pairwise distance will be those of an array that *is* numbered that way.

class `span.tdt.recording.`**`ElectrodeMap`**(*map_*, *within_shank=None*, *between_shank=None*)

Bases: `object`

Encapsulate the geometry of an electrode map.

Methods

`distance_map`(*metric='wminkowski'*, *p=2.0*)

Create a distance map from the current electrode configuration.

Parameters **metric** : str or callable, optional

Metric to use to calculate the distance between electrodes/shanks. Defaults to a weighted Minkowski distance

p : numbers.Real, optional

The p of the norm to use. Defaults to 2.0 for weighted Euclidean distance.

Returns **df** : DataFrame

A dataframe with pairwise distances between electrodes, indexed by channel, shank.

Raises **AssertionError** :

- If *p* is not an instance of `numbers.Real` or `numpy.floating`
- If metric is not an instance of `basestring` or a callable

Notes

This method performs some type checking on its arguments.

The default *metric* of `'wminkowski'` and the default *p* of `2.0` combine to give a weighted Euclidean distance metric. The weighted Minkowski distance between two points $\mathbf{x}, \mathbf{y} \in R^n$, and a weight vector $\mathbf{w} \in R^n$ is given by

$$\left(\sum_{i=1}^{n} w_i \left| x_i - y_i \right|^p \right)^{1/p}$$

`span.tdt.recording.`**`distance_map`**(*nshanks*, *electrodes_per_shank*, *within_shank*, *between_shank*, *metric='wminkowski'*, *p=2.0*)

Create an electrode distance map.

Parameters **nshanks, electrodes_per_shank** : int

within_shank, between_shank : float

metric : str or callable, optional

The distance measure to use to compute the distance between electrodes.

p : number, optional

See `scipy.spatial.distance` for more details.

Returns **dists** : DataFrame

DataFrame of pairwise distances between electrodes.

Raises **AssertionError** :

- If *nshanks* < 1 or *nshanks* is not an integer
- If neither of those same conditions holds for *electrodes_per_shank*
- If *metric* is not a string or callable or
- If *p* is not an instance of `numbers.Real` and `<= 0`

CHAPTER

FIVE

CROSS-CORRELATION

Cross-correlation is useful for determining the lag at which the maximum correlation between two processes lies.

Formally, it is

$$\rho(\ell) = \frac{E[U_0 V_\ell] - E[U_0] E[V_0]}{\mathrm{std}(U_0)\,\mathrm{std}(V_0)}$$

where U_k and V_k are binary time series and $k, \ell \in Z$ [1]. This module provides functions to compute $\rho(\ell)$ in an efficient manner by using the identity

$$\mathrm{xcorr}(\mathbf{x}, \mathbf{y}) = \mathrm{ifft}(\mathrm{fft}(\mathbf{x}) \cdot \mathrm{fft}(\mathbf{y})^*)$$

where $\mathrm{ifft}(\mathbf{x})$ is the inverse Fourier transform of a vector, $\mathrm{fft}(\mathbf{x})$ is the Fourier transform of a vector, $\mathbf{x}$ and $\mathbf{y}$ are vectors, and $*$ is the complex conjugate.

See any signal processing text for a presentation of cross-correlation in a more general setting.

5.1 span.xcorr

span.xcorr.xcorr.**xcorr**(*x*, *y=None*, *maxlags=None*, *detrend=None*, *scale_type=None*)

Compute the cross correlation of *x* and *y*.

[1] Amarasingham et. al (2012), in press

This function computes the cross correlation of *x* and *y*. It uses the equivalence of the cross correlation with the negative convolution computed using a FFT to achieve much faster cross correlation than is possible with the signal processing definition.

By default it computes the unnormalized cross correlation.

Parameters **x** : array_like

The array to correlate.

y : array_like, optional

If y is None or equal to *x* or x and y reference the same object, the autocorrelation is computed.

maxlags : int, optional

The maximum lag at which to compute the cross correlation. Must be less than or equal to the max(x.size, y.size) if not None.

detrend : callable, optional

A callable to detrend the data. It must take a single parameter and return an array

scale_type : {None, 'none', 'unbiased', 'biased', 'normalize'}, optional

- The type of scaling to perform on the data
- The default of 'normalize' returns the cross correlation scaled by the lag 0 cross correlation i.e., the cross correlation scaled by the product of the standard deviations of the arrays at lag 0.

Returns **c** : Series or DataFrame or array_like

Autocorrelation of *x* if *y* is `None`, cross-correlation of *x* if *x* is a matrix and *y* is `None`, or the cross-correlation of *x* and *y* if both *x* and *y* are vectors.

Raises **AssertionError** :

- If *y* is not None and *x* is a matrix
- If *x* is not a vector when *y* is None or *y* is *x* or `all(x == y)`.
- If *detrend* is not callable
- If *scale_type* is not a string or `None`
- If *scale_type* is not in `(None, 'none', 'unbiased', 'biased', 'normalize')`
- If *maxlags* > *lsize*, see source for details.

CHAPTER

SIX

UTILITY FUNCTIONS

The `span.utils.utils` module is a collection of various helper functions that I found myself using repeatedly. If you're interested in understanding them, I suggest reading the source code.

6.1 span.utils.utils

6.2 span.utils.math

CHAPTER

SEVEN

FULL EXAMPLE

Here is a worked example to get to a dataset that is ready for analysis.

All of the following steps assume that you've executed `import span` in a Python interpreter.

7.1 1. Read a TDT file

```
import span
tankname = 'some/path/to/a/tdt/tank/file'
tank = span.tdt.PandasTank(tankname)
sp = tank.spik # spikes is a computed property based on the names of events
```

7.2 2. Threshold the data

```
# create an array of bools indicating which spikes have voltage values
# greater than 4 standard deviations from the mean
thr = sp.threshold(4 * sp.std())
```

7.3 3. Clear the refractory period

```
# clear the refractory period of any spikes; in place to save memory
thr.clear_refrac(inplace=True)
```

7.4 4. Bin the data

```
# bin the data in 1 second bins
binned = clr.resample('S', how='sum')
```

7.5 5. Compute the cross correlation

```
# compute the cross-correlation of all channels
# note that there are a lot more options to this method
# you should explore the docs
xcorr = sp.xcorr(binned)
```

7.6 Full Code Block

```
import span
tankname = 'some/path/to/a/tdt/tank/file'
tank = span.tdt.TdtTank(tankname)
sp = tank.spik

# create an array of bools indicating which spikes have voltage values
# greater than 4 standard deviations
thr = sp.threshold(4 * sp.std())

# clear the refractory period of any spikes
thr.clear_refrac(inplace=True)

# binned the data in 1 second bins
binned = clr.resample('S', how='sum')

# compute the cross-correlation of all channels
xcorr = sp.xcorr(binned)
```

CHAPTER

EIGHT

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

S