SplitWavePy Documentation

Release 0.2.6

Jack Walpole

Apr 18, 2018

Contents

Splitting made easy in Python.

Installation: `pip install splitwavepy`

Source: https://github.com/JackWalpole/splitwavepy

CHAPTER 1

Get started

First off, get yourself a functioning Python environment, I use Anaconda to manage environments and mainly work with Python 3.

Second, install **SplitWavePy** with the following terminal command.

```
pip install splitwavepy
```

Tip: Ensure you have the latest release.

```
pip install -U --no-deps splitwavepy
```

Now try the code.

```
import splitwavepy as sw
m = sw.EigenM( split=(50, 1.9), delta=0.05, noise=0.04)
m.plot()
```

Save your measurement to disk.

```
>>> m.save('temp.eigm')
```

Check your working directory for a file called `temp.eigm`. Is it there? How big is it? It should be less than 200K. It's a backup of your measurement together with the input data. The idea is to help make reproducibility as easy as possible.

Hint: To check your file size in a Unix style operating system: `du -sh temp.eigm`

Without closing your python session, load the data from the disk and check it's the same as that on memory.

```
>>> n = sw.load('temp.eigm')
>>> n == m
... True
```

(continues on next page)

(continued from previous page)

```
>>> n is m
... False
```

Try plotting `n`. Does it look the same as `m`?

If you've made it to here, great, you seem to have a working version of SplitWavePy.

Now check out the *Tutorial*.

CHAPTER 2

Tutorial

Assuming you have the code setup it's time to see what it can do. Fire up an interactive python session such as `ipython` and `import splitwavepy as sw`.

```
import splitwavepy as sw
```

2.1 Synthetic data

By default, if no data are provided, SplitWavePy will make some synthetic data. Data is stored in a *Pair* object. I will use synthetic data to demonstrate the basic features of the code.

```
data = sw.Pair( split=( 30, 1.4), noise=0.03, pol=-15.8, delta=0.05)
data.plot()
```

You can simulate splitting by specifying the fast direction and lag time using the `split = (fast, lag)` argument, similarly you can specify the *noise* level, and source *pol* -arisation as shown in the example above. Order is not important.

Note: Don't forget to set the sample interval *delta* appropriately (defaults to 1), it determines how your *lag* time is interpreted.

2.2 Adding and removing splitting

The data are already split, but that doesn't mean they can't be split again! You can add splitting using the *split()* method.

```
data.split( -12, 1.3) # fast, lag
data.plot()
```

Note: This method will also split the noise, which is probable undesirable. A better way to make synthetic with multiple splitting events is to provide the `split` keyword with a list of parameters, e.g. `split = [( 30, 1.4), ( -12, 1.3)]`, these will be applied in the order provided.

Sometimes we might want to do a *correction* and apply the *inverse* splitting operator. This is supported using the *unsplit()* method. If we correct the above data for the latter layer of anisotropy we should return the waveforms to their former state.

```
data.unsplit( -12, 1.3)
data.plot()
```

Note: Every time a lag operation is applied the traces are shortened. This is fine so long as your traces extend far enough beyond your window.

2.3 Setting the window

The window should be designed in such a way as to maximise the energy of a single pulse of shear energy relative to noise and other arrivals.

Set the window using the `set_window(start,end)` method.

```
data.set_window( 15, 32) # start, end
data.plot()
```

Note: By default the window will be centred on the middle of the trace with width 1/3 of the trace length, which is likely to be inappropriate, so make sure to set your window sensibly.

Tip: Interactive window picking is supported by `plot(pick=True)`. Left click to pick the window start, right click to pick the window end, hit the space bar to save and close. If you don't want to save your window, simply quit the figure using one of the built in matplotlib methods (e.g. click on the cross in the top left corner or hit the `q` key).pwd

2.4 Silver and Chan (1991) eigenvalue method

A powerful and popular method for measuring splitting is the eigenvalue method of Silver and Chan (1991). It uses a grid search to find the inverse splitting parameters that best linearise the particle motion. Linearisation is assessed by principal component analysis at each search node, taking the eigenvalues of the covariance matrix, where linearity maximises λ_1 and minimises λ_2. The code uses the ratio λ_1/λ_2 to find the best node (which is more stable than using only λ_1 or λ_2 as it accounts for the possibility that energy might be lost by sliding out of the window).

To use this method on your data.

```
measure = sw.EigenM(data)
measure.plot()
```

2.4.1 Setting the lag time grid search

The code automatically sets the maximum lag time to be half the window length. To set the max search time manually you use the `lags` keyword. This accepts a tuple of length 1, 2, or 3, and will be interpreted differently depending on this length. The rules are as follows: for a 1-tuple `lags = (maxlag,)`, a 2-tuple `lags = (maxlag, nlags)`, and finally a 3-tuple `(minlag, maxlag, nlags)`. Alternatively will accept a numpy array containing all nodes to search.

2.4.2 Setting the fast direction grid search

The code automatically grid searches every 2 degrees along the fast direction axis. That's `degs = 90` nodes in total (180/2). You can change this number using the `degs` keyword and providing an integer. Alternatively will accept a numpy array containing all nodes to search.

2.5 Saving and loading your measurements

To save your measurement to disk simply use the `save(filename)` method. This will backup the input data complete with the λ_1 and λ_2 surfaces.

This can be recovered at a later time using `splitwavepy.load(filename)`.

This feature is demonstrated here *Get started*.

2.6 Splitting corrections

In the case where you have a good estimation of the splitting parameters beneath the receiver or the source it is possible to correct the waveforms and to measure the residual splitting. The residual splitting can then be attributed to anisotropy elsewhere along the path.

Let's consider a simple 2-layer case.

```
# srcside and rceiver splitting parameters
srcsplit = (  30, 1.3)
rcvsplit = ( -44, 1.7)

# Create synthetic
a = sw.Pair( split=([ srcsplit, rcvsplit]), noise=0.03, delta=0.02)

# standard measurement
m = sw.EigenM(a, lags=(3,))
m.plot()
```

The *apparent* splitting measured above is some non-linear combination of the 2-layers (non-linear because the order of splitting is important).

2.6.1 Receiver correction

If we know the layer 2 contribution we can back this off and resolve the splitting in layer 1 using the `rcvcorr=(fast, lag)` keyword.

```
m = sw.EigenM(a, lags=(3,), rcvcorr=(-44,1.7))
m.plot()
```

If it's worked we should have measured splitting parameters of $\phi = 30$ and $\delta t = 1.3$.

2.6.2 Source correction

Alternatively, if we know the layer 1 contribution we can use `srccorr=(fast, lag)` to correct for the source side anisotropy.

```
m = sw.EigenM(a, lags=(3,), srccorr=(30,1.3))
m.plot()
```

If this has worked we should have measured splitting parameters of $\phi = -44$ and $\delta t = 1.7$.

If we apply both the source and receiver correction to the above synthetic example we should yield a *null* result (no splitting).

```
m = sw.EigenM(a, lags=(3,), rcvcorr=(-44,1.7), srccorr=(30,1.3))
m.plot()
```

We do as can be seen by the concentration of energy at delay time 0.

CHAPTER 3

Real data example

If you've got real data you need to get it into a numpy array. For the purposes of this tutorial, let's use Obspy to download some data from the IRIS servers.

3.1 Preparing data

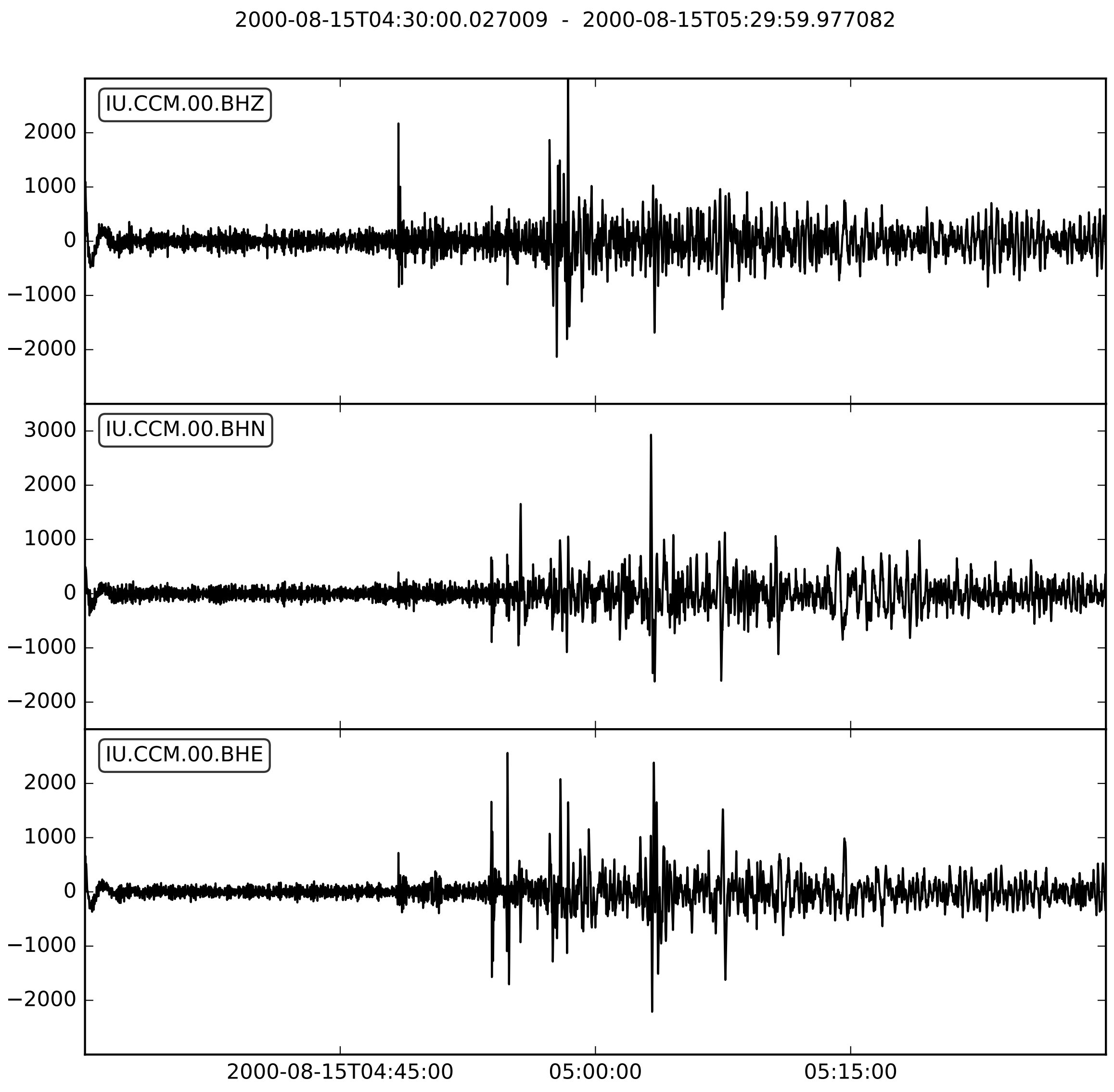
With real data it's worth doing a bit of pre-processing which at minimum will involve removing the mean from data, and might also involve bandpass filtering, interpolation, and/or rotating the components. It is also necessary to pick the shear arrival of interest. All of this is achievable in Obspy.

```python
from obspy import read
from obspy.clients.fdsn import Client
from obspy import UTCDateTime

client = Client("IRIS")
t = UTCDateTime("2000-08-15T04:30:0.000")
st = client.get_waveforms("IU", "CCM", "00", "BH?", t, t + 60 * 60,attach_
↪response=True)

# filter the data (this process removes the mean)
st.filter("bandpass",freqmin=0.01,freqmax=0.5)

st.plot()
```



When measuring splitting we need to have a specific shear wave arrival to target. Let's use the obspy taup module to home in on the SKS arrival.

```
from obspy.taup import TauPyModel

# from location and time, get event information
lat=-31.56
lon=179.74

# server does not accept longitude greater than 180.
cat = client.get_events(starttime=t-60,endtime=t+60,minlatitude=lat-1,
                  maxlatitude=lat+1,minlongitude=lon-1,maxlongitude=180)
evtime = cat.events[0].origins[0].time
evdepth = cat.events[0].origins[0].depth/1000
evlat = cat.events[0].origins[0].latitude
evlon = cat.events[0].origins[0].longitude
```

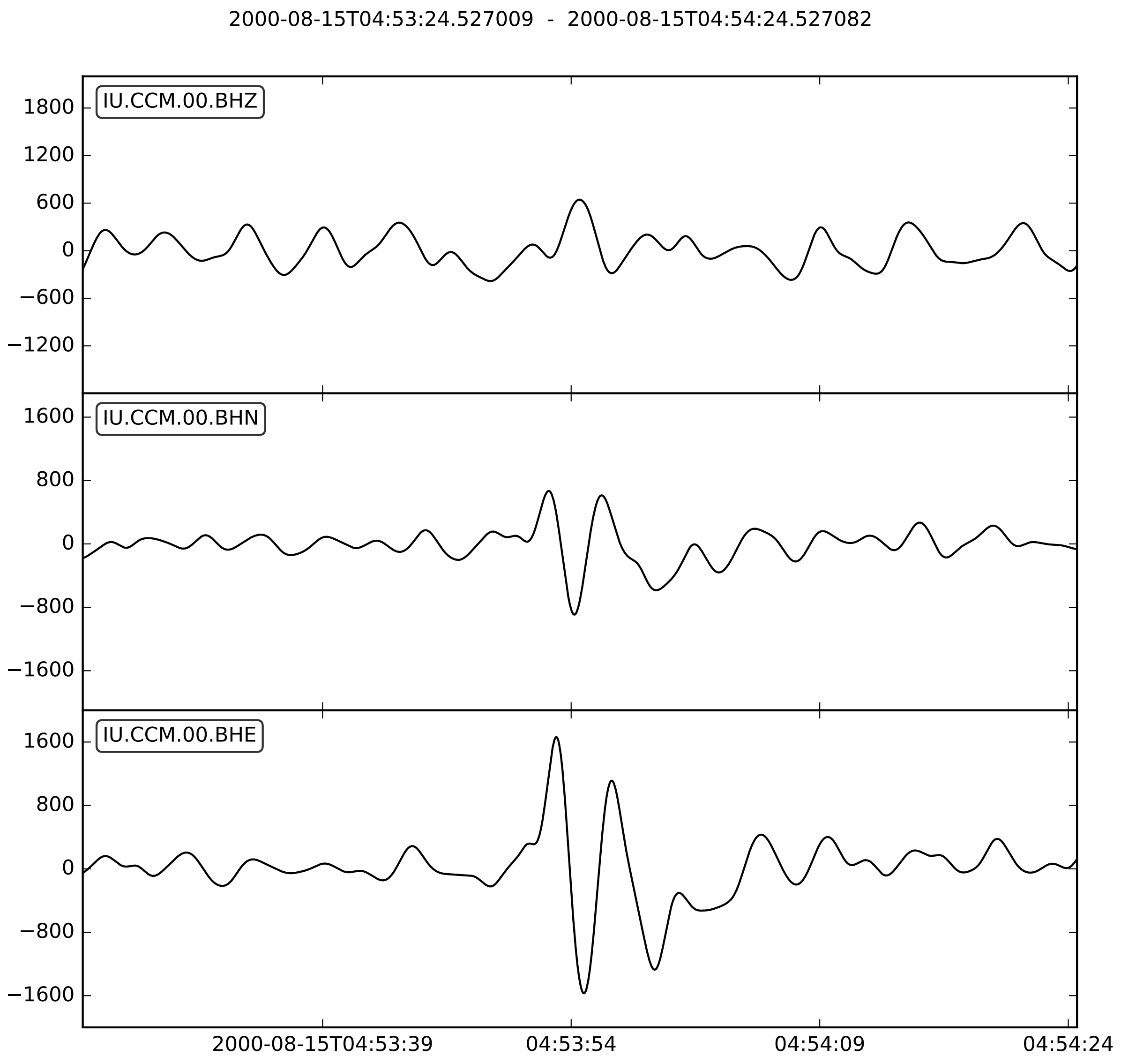
(continues on next page)

(continued from previous page)

```
# station information
inventory = client.get_stations(network="IU",station="CCM",starttime=t-60,
↪endtime=t+60)
stlat = inventory[0][0].latitude
stlon = inventory[0][0].longitude

# find arrival times
model = TauPyModel('iasp91')
arrivals = model.get_travel_times_geo(evdepth,evlat,evlon,stlat,stlon,phase_list=['SKS
↪'])
skstime = evtime + arrivals[0].time

# trim around SKS
st.trim( skstime-30, skstime+30)
st.plot()
```



3.2 Measuring splitting

Now we have prepared data we are ready to measure splitting.

First read the shear plane components (horizontals in this case) into a *Pair* object.

```
import splitwavepy as sw

# get data into Pair object and plot
north = st[1].data
east = st[0].data
sample_interval = st[0].stats.delta
realdata = sw.Pair(north, east, delta=sample_interval)
realdata.plot()
```

Note: Order is important. SplitWavePy expects the North component first.

By chance the window looks not bad. If you want to change it see *Setting the window*. For now let's press on with measuring the splitting.

```
measure = sw.EigenM(realdata)
measure.plot()
```

It worked, kind of. The maximum delay time in the grid search is a bit high. We can change this using the lags keyword `lags=(2,)` as explained in *Setting the lag time grid search*.

```
measure = sw.EigenM(realdata, lags=(2,))
measure.plot()
```

Now see how to apply the *Transverse Minimisation Method*.

CHAPTER 4

Transverse Minimisation Method

If the polarisation of the shear energy is known then the energy on the transverse component is indicative of splitting (Silver and Chan, 1998; Vinnik et al., 1989). The method works by searching for the splitting parameters that when applied to the data removes the most energy from the transverse component. In principle the method is very similar to the eigenvalue method of Silver and Chan 1991, except the polarisation of the shear energy us defined by the user, rather than being searched for by principal component analysis. By including the polarisation as a priori information the measurement is more robust to noise, however, the method is vulnerable to error if the polarisation is specified incorrectly.

Continuing with the *Real data example*, for which we obtained the following plot using the eigenvalue method.

To do the transverse minimisation method we use the `TransM` class with the polarisation specified. In the case of the SKS wave the polarisation is equal to the backazimuth direction, this can be calculated using obspy.

```
# use obspy to get backazimuth
from obspy import geodetics
dist, az, baz = geodetics.base.gps2dist_azimuth(evlat,evlon,stlat,stlon)

# make the measurement
m = sw.TransM(realdata, pol=baz, lags=(2,))
m.plot()
```

Notice that the transverse minimisation method returns a more focussed result.

Now checkout the *Rotation Correlation Method*.

CHAPTER 5

Rotation Correlation Method

The rotation correlation method searches for the rotation and lag which maximises the similarity of pulse shapes and aligns them in time. The method was introduced by Fukao, 1984.

In my testing I have found it is very important to use the normalised correlation coefficient.

Again, continuing with the *Real data example*, for which we have used the eigenvalue method and the transverse minimisation method plots:

To do the rotation correlation method we use the `CrossM` class as follows.

```
m = sw.CrossM(realdata, lags=(2,))
m.plot()
```

CHAPTER 6

3–Component Data

SplitWavePy supports working with 3–component data and offers tools for rotating the data into a user-defined reference frame. This is implemented by providing a *ray* direction, which ideally should be normal to the shear plane.

3–component data is easily incorporated into the eigenvalue method, offering additional information in a third eigenvalue (e.g. Walsh et al., 2014). In principle, if a shear wave has been fully corrected for splitting, this should maximise the first eigenvalue and minimise the second and third eigenvalues.

3–component data is handled by the Trio class. This works in much the same way as the Pair class. If no data are provided as arguments, the class will produce synthetic data with parameters set by keywords.

```
import splitwavepy as sw

a = sw.Trio( split=( -50, 2.4), delta=0.025, noise=0.01, ray=(20,30))
a.plot()
```

The main complication in dealing with 3-components is choosing the axis about which to perform the rotations and lags that define the splitting operator. As a rule, the code will use the axis defined by the ray. A trio object must, therefore, contain a ray. The ray is set using the `ray=(azi,inc)` keyword which expects a tuple containing the azimuth and incidence angle, by default `ray=(0,0)`. Note, that the ray can be any angle you like, for example you might want to have it pointing in the wavefront normal direction, which could be different from the geometric ray direction. The ray is plotted as a red line on the particle motion plots.

You can rotate your data to the ray as follows.

```
a.rotate2ray()
a.plot()
```

To measure splitting on a Trio use the Eig3dM class. You don't need to rotate to the ray before doing this. The code will internally ensure the data is rotated into the ray frame before grid searching for the splitting parameters.

```
b = sw.Eig3dM(a,lags=(3,))
b.plot()
```

Note: In 3-D *all* fast directions are defined in the **ray frame**. This includes the directions you assign to synthetics, receiver- and source-corrections, and the fast direction you measure. Directions are measured relative to "up", which is actually the projection of the true up direction in the plane normal to the ray, i.e., the *SV* direction.

Note: When using the trio you must ensure your data x, y, z correspond to north, east, up directions.

CHAPTER 7

Acknowledge

Feel free to use this code, and if you find it helpful please acknowledge this in your work.

CHAPTER 8

Contribute

If you can help by improving the code in any way please submit an issue on github or fork on github and submit a pull request.