ObsPy Tutorial

Release 1.1.1rc1.post0+436.g3ffa3209c6.obspy.master

The ObsPy Development Team (devs@obspy.org)

November 19 18 o'clock, 2018

Note: A one-hour introduction to ObsPy is available at YouTube.

This tutorial does not attempt to be comprehensive and cover every single feature. Instead, it introduces many of ObsPy's most noteworthy features, and will give you a good idea of the library's flavor and style.

A pdf version of the Tutorial is available `here`.

There are also IPython notebooks available online with an introduction to Python (with solutions/output), an introduction to ObsPy split up in multiple chapters (again, versions with/without solutions available) and a brief primer on data center access and visualization with ObsPy. There are also nice Jupyter notebooks with an introduction to matplotlib.

CHAPTER

ONE

INTRODUCTION TO OBSPY

1.1 Python Introduction for Seismologists

Here we want to give a small, incomplete introduction to the Python programming language, with links to useful packages and further resources. The key features are explained via the following Python script:

```
1  #!/usr/bin/env python
2  import glob
3  from obspy.core import read
4
5  for file in glob.glob('*.z'):
6      st = read(file)
7      tr = st[0]
8      msg = "%s %s %f %f" % (tr.stats.station, str(tr.stats.starttime),
9                             tr.data.mean(), tr.data.std())
10     print(msg)
```

Description of each line of the example above:

Line 1 Shebang, specifying the location of the Python interpreter for Unix-like operating systems.

Lines 2-3 Import modules/libraries/packages in the current namespace. The `glob` module, which allows wildcard matching on filenames, is imported here. All functions inside this module can be accessed via `glob.function()` afterwards, such as `glob.glob()`. Furthermore, a single function `read()` from the `obspy.core` module is imported, which is used to read various different seismogram file formats.

Line 5 Starts a `for`-loop using the `glob()` function of the module `glob` on all files ending with `'.z'`.

Note: The length of all loops in Python is determined by the indentation level. Do not mix spaces and tabs in your program code for indentation, this produces bugs that are not easy to identify.

Line 6 Uses the `read()` function from the `obspy.core` module to read in the seismogram to a `Stream` object named `st`.

Line 7 Assigns the first `Trace` object of the list-like `Stream` object to the variable `tr`.

Line 8-9 A Python counterpart for the well-known C function `sprintf` is the `%` operator acting on a format string. Here we print the header attributes `station` and `starttime` as well as the return value of the methods `mean()` and `std()` acting on the data sub-object of the `Trace` (which are of type `numpy.ndarray`).

Line 10 Prints content of variable `msg` to the screen.

As Python is an interpreter language, we recommend to use the IPython shell for rapid development and trying things out. It supports tab completion, history expansion and various other features. E.g. type `help(glob.glob)` or `glob.glob?` to see the help of the `glob()` function (the module must be imported beforehand).

Further Resources

- **https://docs.python.org/3/tutorial/** Official Python tutorial.
- **https://docs.python.org/3/library/index.html** Python library reference
- **http://software-carpentry.org/** Very instructive video lectures on various computer related topics. A good starting point for learning Python and Version Control with Subversion.
- **https://ipython.org/** An enhanced interactive Python shell.
- **https://docs.scipy.org/doc/** NumPy and SciPy are the matrix based computation modules of Python. The allow fast array manipulation (functions in C). NumPy and SciPy provide access to FFTW, LAPACK, ATLAS or BLAS. That is svd, eigenvalues... ObsPy uses the numpy.ndarrays for storing the data (e.g. tr.data).
- **http://matplotlib.org/gallery.html** matplotlib is the 2-D plotting package for Python. The gallery is the market place which allows you to go shopping for all kind of figures. The source code for each figure is linked. Note matplotlib has even its own latex renderer.
- **http://matplotlib.org/basemap/** Package plotting 2D data on maps in Python. Similar to GMT.
- **https://trac.osgeo.org/gdal/wiki/GdalOgrInPython** Package which allows to directly read a GeoTiff which then can be plotted with the basemap toolkit.
- **https://svn.geophysik.uni-muenchen.de/trac/mtspecpy** Multitaper spectrum bindings for Python

1.2 UTCDateTime

All absolute time values within ObsPy are consistently handled with the `UTCDateTime` class. It is based on a high precision POSIX timestamp and not the Python datetime class because precision was an issue.

1.2.1 Initialization

```
>>> from obspy.core import UTCDateTime
>>> UTCDateTime("2012-09-07T12:15:00")
UTCDateTime(2012, 9, 7, 12, 15)
>>> UTCDateTime(2012, 9, 7, 12, 15, 0)
UTCDateTime(2012, 9, 7, 12, 15)
>>> UTCDateTime(1347020100.0)
UTCDateTime(2012, 9, 7, 12, 15)
```

In most cases there is no need to worry about timezones, but they are supported:

```
>>> UTCDateTime("2012-09-07T12:15:00+02:00")
UTCDateTime(2012, 9, 7, 10, 15)
```

1.2.2 Attribute Access

```
>>> time = UTCDateTime("2012-09-07T12:15:00")
>>> time.year
2012
>>> time.julday
251
>>> time.timestamp
1347020100.0
```

```
>>> time.weekday
4
```

1.2.3 Handling time differences

```
>>> time = UTCDateTime("2012-09-07T12:15:00")
>>> print(time + 3600)
2012-09-07T13:15:00.000000Z
>>> time2 = UTCDateTime(2012, 1, 1)
>>> print(time - time2)
21644100.0
```

1.2.4 Exercises

- Calculate the number of hours passed since your birth. Optional: Include the correct time zone. The current date and time can be obtained with

```
>>> UTCDateTime()
```

- Get a list of 10 UTCDateTime objects, starting yesterday at 10:00 with a spacing of 90 minutes.
- The first session starts at 09:00 and lasts for 3 hours and 15 minutes. Assuming we want to have the coffee break 1234 seconds and 5 microseconds before it ends. At what time is the coffee break?
- Assume you had your last cup of coffee yesterday at breakfast. How many minutes do you have to survive with that cup of coffee?

1.3 Reading Seismograms

Seismograms of various formats (e.g. SAC, MiniSEED, GSE2, SEISAN, Q, etc.) can be imported into a `Stream` object using the `read()` function.

`Streams` are list-like objects which contain multiple `Trace` objects, i.e. gap-less continuous time series and related header/meta information.

Each `Trace` object has a attribute called `data` pointing to a NumPy `ndarray` of the actual time series and the attribute `stats` which contains all meta information in a dictionary-like `Stats` object. Both attributes `starttime` and `endtime` of the `Stats` object are `UTCDateTime` objects.

The following example demonstrates how a single GSE2-formatted seismogram file is read into a ObsPy `Stream` object. There exists only one `Trace` in the given seismogram:

```
>>> from obspy import read
>>> st = read('http://examples.obspy.org/RJOB_061005_072159.ehz.new')
>>> print(st)
1 Trace(s) in Stream:
.RJOB..Z | 2005-10-06T07:21:59.849998Z - 2005-10-06T07:24:59.844998Z | 200.0 Hz, 36000 samples
>>> len(st)
1
>>> tr = st[0]  # assign first and only trace to new variable
>>> print(tr)
.RJOB..Z | 2005-10-06T07:21:59.849998Z - 2005-10-06T07:24:59.844998Z | 200.0 Hz, 36000 samples
```

1.3.1 Accessing Meta Data

Seismogram meta data, data describing the actual waveform data, are accessed via the `stats` keyword on each `Trace`:

```
>>> print(tr.stats)
         network:
         station: RJOB
        location:
         channel: Z
       starttime: 2005-10-06T07:21:59.849998Z
         endtime: 2005-10-06T07:24:59.844998Z
   sampling_rate: 200.0
           delta: 0.005
            npts: 36000
           calib: 0.0948999971151
         _format: GSE2
            gse2: AttribDict({'instype': '      ', 'datatype': 'CM6', 'hang': -1.0, 'auxid': 'RJOB',
>>> tr.stats.station
'RJOB'
>>> tr.stats.gse2.datatype
'CM6'
```

1.3.2 Accessing Waveform Data

The actual waveform data may be retrieved via the `data` keyword on each `Trace`:

```
>>> tr.data
array([-38,  12,  -4, ..., -14,  -3,  -9])
>>> tr.data[0:3]
array([-38,  12,  -4])
>>> len(tr)
36000
```

1.3.3 Data Preview

`Stream` objects offer a `plot()` method for fast preview of the waveform (requires the `obspy.imaging` module):

```
>>> st.plot()
```

1.4 Waveform Plotting Tutorial

Read the files as shown at the *Reading Seismograms* page. We will use two different ObsPy `Stream` objects throughout this tutorial. The first one, `singlechannel`, just contains one continuous `Trace` and the other one, `threechannel`, contains three channels of a seismograph.

```
>>> from obspy.core import read
>>> singlechannel = read('https://examples.obspy.org/COP.BHZ.DK.2009.050')
>>> print(singlechannel)
1 Trace(s) in Stream:
DK.COP..BHZ | 2009-02-19T00:00:00.025100Z - 2009-02-19T23:59:59.975100Z | 20.0 Hz, 1728000 samples
```

```
>>> threechannels = read('https://examples.obspy.org/COP.BHE.DK.2009.050')
>>> threechannels += read('https://examples.obspy.org/COP.BHN.DK.2009.050')
>>> threechannels += read('https://examples.obspy.org/COP.BHZ.DK.2009.050')
>>> print(threechannels)
3 Trace(s) in Stream:
DK.COP..BHE | 2009-02-19T00:00:00.035100Z - 2009-02-19T23:59:59.985100Z | 20.0 Hz, 1728000 samples
DK.COP..BHN | 2009-02-19T00:00:00.025100Z - 2009-02-19T23:59:59.975100Z | 20.0 Hz, 1728000 samples
DK.COP..BHZ | 2009-02-19T00:00:00.025100Z - 2009-02-19T23:59:59.975100Z | 20.0 Hz, 1728000 samples
```

1.4.1 Basic Plotting

Using the `plot()` method of the `Stream` objects will show the plot. The default size of the plots is 800x250 pixel. Use the `size` attribute to adjust it to your needs.

```
>>> singlechannel.plot()
```

1.4.2 Customized Plots

This example shows the options to adjust the color of the graph, the number of ticks shown, their format and rotation and how to set the start and end time of the plot. Please see the documentation of method `plot()` for more details on all parameters.

```
>>> dt = singlechannel[0].stats.starttime
>>> singlechannel.plot(color='red', number_of_ticks=7,
...                    tick_rotation=5, tick_format='%I:%M %p',
...                    starttime=dt + 60*60, endtime=dt + 60*60 + 120)
```

1.4.3 Saving Plot to File

Plots may be saved into the file system by the `outfile` parameter. The format is determined automatically from the filename. Supported file formats depend on your matplotlib backend. Most backends support png, pdf, ps, eps and svg.

```
>>> singlechannel.plot(outfile='singlechannel.png')
```

1.4.4 Plotting multiple Channels

If the `Stream` object contains more than one `Trace`, each Trace will be plotted in a subplot. The start- and endtime of each trace will be the same and the range on the y-axis will also be identical on each trace. Each additional subplot will add 250 pixel to the height of the resulting plot. The `size` attribute is used in the following example to change the overall size of the plot.

```
>>> threechannels.plot(size=(800, 600))
```

1.4.5 Creating a One-Day Plot

A day plot of a `Trace` object may be plotted by setting the `type` parameter to `'dayplot'`:

```
>>> singlechannel.plot(type='dayplot')
```

Event information can be included in the plot as well (experimental feature, syntax might change):

```
>>> from obspy import read
>>> st = read("https://examples.obspy.org/GR.BFO..LHZ.2012.108")
>>> st.filter("lowpass", freq=0.1, corners=2)
>>> st.plot(type="dayplot", interval=60, right_vertical_labels=False,
...         vertical_scaling_range=5e3, one_tick_per_line=True,
...         color=['k', 'r', 'b', 'g'], show_y_UTC_label=False,
...         events={'min_magnitude': 6.5})
```

1.4.6 Plotting a Record Section

A record section can be plotted from a `Stream` object by setting parameter `type` to `'section'`:

```
>>> stream.plot(type='section')
```

To plot a record section the ObsPy header `trace.stats.distance` (Offset) must be defined in meters. Or a geographical location `trace.stats.coordinates.latitude` & `trace.stats.coordinates.longitude` must be defined if the section is plotted in great circle distances (`dist_degree=True`) along with parameter `ev_coord`. For further information please see `plot()`

1.4.7 Plot & Color Options

Various options are available to change the appearance of the waveform plot. Please see `plot()` method for all possible options.

1.4.8 Custom Plotting using Matplotlib

Custom plots can be done using matplotlib, like shown in this minimalistic example (see http://matplotlib.org/gallery.html for more advanced plotting examples):

```
import matplotlib.pyplot as plt
from obspy import read

st = read()
tr = st[0]

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.plot(tr.times("matplotlib"), tr.data, "b-")
ax.xaxis_date()
fig.autofmt_xdate()
plt.show()
```

1.5 Retrieving Data from Data Centers

This section is intended as a small guide to help you choose the best way to download data for a given purpose using ObsPy. Keep in mind that data centers and web services are constantly changing so this recommendation might not be valid anymore at the time you read this. For actual code examples, please see the documentation of the various modules.

Note: The most common use case is likely to download waveforms and event/station meta information. In almost

all cases you will want to use the `obspy.clients.fdsn` module for this. It supports the largest number of data centers and uses the most modern data formats. There are still a number of reasons to choose a different module but please make sure you have one.

1.5.1 The FDSN Web Services

Basic FDSN Web Services

Available Data Types	Format
Waveforms	MiniSEED and optionally others
Station Information	StationXML and Text
Event Information	QuakeML and Text

Note: Not all data providers offer all three data types. Many offer only one or two.

If you want to requests waveforms, or station/event meta information **you will most likely want to use the `obspy.clients.fdsn` module**. It is able to request data from any data center implementing the FDSN web services. Example data centers include IRIS/ORFEUS/INGV/ETH/GFZ/RESIF/... - a curated list can be found here. As a further advantage it returns data in the most modern and future proof formats.

FDSN Routing Web Services

If you don't know which data center has what data, use one of the routing services. ObsPy has support for two of them:

1. The IRIS Federator.
2. The EIDAWS Routing Service.

See the bottom part of the `obspy.clients.fdsn` module page for usage details.

FDSN Mass Downloader

If you want to download a lot of data across a number of data centers, ObsPy's mass (or batch) downloader is for you. You can formulate your queries for example in terms of geographical domains and ObsPy will download waveforms and corresponding station meta information to produce complete data sets, ready for research, including some basic quality control.

See the `obspy.clients.fdsn.mass_downloader` page for more details.

1.5.2 ArcLink

Available Data Types	Format
Waveforms	MiniSEED, SEED
Station Information	dataless SEED, SEED

ArcLink is a distributed data request protocol usable to access archived waveform data in the MiniSEED or SEED format and associated meta information as Dataless SEED files. You can use the `obspy.clients.arclink` module to request data from the EIDA initiative but most (or all) of that data can also be requested using the `obspy.clients.fdsn` module.

1.5.3 IRIS Web Services

Available Data Types and Formats: Various

IRIS (in addition to FDSN web services) offers a variety of special-purpose web services, for some of which ObsPy has interfaces in the `obspy.clients.iris` module. Use this if you require response information in the SAC poles & zeros or in the RESP format. If you just care about the instrument response, please use the `obspy.clients.fdsn` module to request StationXML data which contains the same information.

The interfaces for the calculation tools are kept around for legacy reasons; internal ObsPy functionality should be considered as an alternative when working within ObsPy:

IRIS Web Service	Equivalent ObsPy Function/Module
`obspy.clients.iris.client.Client.traveltime()`	`obspy.taup`
`obspy.clients.iris.client.Client.distaz()`	`obspy.geodetics`
`obspy.clients.iris.client.Client.flinnengdahl()`	`obspy.geodetics.flinnengdahl.FlinnEngdahl`

1.5.4 Earthworm Wave Server

Available Data Types	Format
Waveforms	Custom Format

Use the `obspy.clients.earthworm` module to request data from the Earthworm data acquisition system.

1.5.5 NERIES Web Services

This service is largely deprecated as the data can just as well be requested via the `obspy.clients.fdsn` module.

1.5.6 NEIC

Available Data Types	Format
Waveforms	MiniSEED

The Continuous Waveform Buffer (CWB) is a repository for seismic waveform data that passes through the NEIC "Edge" processing system. Use the `obspy.clients.neic` module to request data from it.

1.5.7 SeedLink

Available Data Types	Format
Waveforms	MiniSEED

To connect to a real time SeedLink server, use the `obspy.clients.seedlink` module. Also see the *ObsPy Tutorial* for a more detailed introduction.

1.5.8 Syngine Service

Available Data Types	Format
Waveforms	MiniSEED and zipped SAC files

Use the `obspy.clients.syngine` module to download high-frequency global synthetic seismograms for any source receiver combination from the IRIS syngine service.

1.6 Filtering Seismograms

The following script shows how to filter a seismogram. The example uses a zero-phase-shift low-pass filter with a corner frequency of 1 Hz using 2 corners. This is done in two runs forward and backward, so we end up with 4 corners de facto.

The available filters are:

- `bandpass`
- `bandstop`
- `lowpass`
- `highpass`

```
import numpy as np
import matplotlib.pyplot as plt

import obspy


# Read the seismogram
st = obspy.read("https://examples.obspy.org/RJOB_061005_072159.ehz.new")

# There is only one trace in the Stream object, let's work on that trace...
tr = st[0]

# Filtering with a lowpass on a copy of the original Trace
tr_filt = tr.copy()
tr_filt.filter('lowpass', freq=1.0, corners=2, zerophase=True)

# Now let's plot the raw and filtered data...
t = np.arange(0, tr.stats.npts / tr.stats.sampling_rate, tr.stats.delta)
plt.subplot(211)
plt.plot(t, tr.data, 'k')
plt.ylabel('Raw Data')
plt.subplot(212)
plt.plot(t, tr_filt.data, 'k')
plt.ylabel('Lowpassed Data')
plt.xlabel('Time [s]')
plt.suptitle(tr.stats.starttime)
plt.show()
```

1.7 Downsampling Seismograms

The following script shows how to downsample a seismogram. Currently, a simple integer decimation is supported. If not explicitly disabled, a low-pass filter is applied prior to decimation in order to prevent aliasing. For comparison, the non-decimated but filtered data is plotted as well. Applied processing steps are documented in `trace.stats.processing` of every single Trace. Note the shift that is introduced because by default the applied filters are not of zero-phase type. This can be avoided by manually applying a zero-phase filter and deactivating automatic filtering during downsampling (`no_filter=True`).

```
import numpy as np
import matplotlib.pyplot as plt

import obspy
```

```
# Read the seismogram
st = obspy.read("https://examples.obspy.org/RJOB_061005_072159.ehz.new")

# There is only one trace in the Stream object, let's work on that trace...
tr = st[0]

# Decimate the 200 Hz data by a factor of 4 to 50 Hz. Note that this
# automatically includes a lowpass filtering with corner frequency 20 Hz.
# We work on a copy of the original data just to demonstrate the effects of
# downsampling.
tr_new = tr.copy()
tr_new.decimate(factor=4, strict_length=False)

# For comparison also only filter the original data (same filter options as in
# automatically applied filtering during downsampling, corner frequency
# 0.4 * new sampling rate)
tr_filt = tr.copy()
tr_filt.filter('lowpass', freq=0.4 * tr.stats.sampling_rate / 4.0)

# Now let's plot the raw and filtered data...
t = np.arange(0, tr.stats.npts / tr.stats.sampling_rate, tr.stats.delta)
t_new = np.arange(0, tr_new.stats.npts / tr_new.stats.sampling_rate,
                  tr_new.stats.delta)

plt.plot(t, tr.data, 'k', label='Raw', alpha=0.3)
plt.plot(t, tr_filt.data, 'b', label='Lowpassed', alpha=0.7)
plt.plot(t_new, tr_new.data, 'r', label='Lowpassed/Downsampled', alpha=0.7)
plt.xlabel('Time [s]')
plt.xlim(82, 83.5)
plt.suptitle(tr.stats.starttime)
plt.legend()
plt.show()
```

1.8 Merging Seismograms

The following example shows how to merge and plot three seismograms with overlaps, the longest one is taken to be the right one. Please also refer to the documentation of the `merge()` method.

```
import numpy as np
import matplotlib.pyplot as plt

import obspy


# Read in all files starting with dis.
st = obspy.read("https://examples.obspy.org/dis.G.SCZ.__.BHE")
st += obspy.read("https://examples.obspy.org/dis.G.SCZ.__.BHE.1")
st += obspy.read("https://examples.obspy.org/dis.G.SCZ.__.BHE.2")

# sort
st.sort(['starttime'])
# start time in plot equals 0
dt = st[0].stats.starttime.timestamp

# Go through the stream object, determine time range in julian seconds
```

```
# and plot the data with a shared x axis
ax = plt.subplot(4, 1, 1)  # dummy for tying axis
for i in range(3):
    plt.subplot(4, 1, i + 1, sharex=ax)
    t = np.linspace(st[i].stats.starttime.timestamp - dt,
                    st[i].stats.endtime.timestamp - dt,
                    st[i].stats.npts)
    plt.plot(t, st[i].data)

# Merge the data together and show plot in a similar way
st.merge(method=1)
plt.subplot(4, 1, 4, sharex=ax)
t = np.linspace(st[0].stats.starttime.timestamp - dt,
                st[0].stats.endtime.timestamp - dt,
                st[0].stats.npts)
plt.plot(t, st[0].data, 'r')
plt.show()
```

1.9 Beamforming - FK Analysis

The following code shows how to do an FK Analysis with ObsPy. The data are from the blasting of the AGFA skyscraper in Munich. We execute `array_processing()` using the following settings:

- The slowness grid is set to corner values of -3.0 to 3.0 s/km with a step fraction of `sl_s = 0.03`.
- The window length is 1.0 s, using a step fraction of 0.05 s.
- The data is bandpass filtered, using corners at 1.0 and 8.0 Hz, prewhitening is disabled.
- `semb_thres` and `vel_thres` are set to infinitesimally small numbers and must not be changed.
- The `timestamp` will be written in `'mlabday'`, which can be read directly by our plotting routine.
- `stime` and `etime` have to be given in the UTCDateTime format.

The output will be stored in `out`.

The second half shows how to plot the output. We use the output `out` produced by `array_processing()`, which are `numpy ndarrays` containing timestamp, relative power, absolute power, backazimuth, slowness. The colorbar corresponds to relative power.

```
import matplotlib.pyplot as plt
import matplotlib.dates as mdates

import obspy
from obspy.core.util import AttribDict
from obspy.imaging.cm import obspy_sequential
from obspy.signal.invsim import corn_freq_2_paz
from obspy.signal.array_analysis import array_processing


# Load data
st = obspy.read("https://examples.obspy.org/agfa.mseed")

# Set PAZ and coordinates for all 5 channels
st[0].stats.paz = AttribDict({
    'poles': [(-0.03736 - 0.03617j), (-0.03736 + 0.03617j)],
    'zeros': [0j, 0j],
    'sensitivity': 205479446.68601453,
```

```
    'gain': 1.0})
st[0].stats.coordinates = AttribDict({
    'latitude': 48.108589,
    'elevation': 0.450000,
    'longitude': 11.582967})

st[1].stats.paz = AttribDict({
    'poles': [(-0.03736 - 0.03617j), (-0.03736 + 0.03617j)],
    'zeros': [0j, 0j],
    'sensitivity': 205479446.68601453,
    'gain': 1.0})
st[1].stats.coordinates = AttribDict({
    'latitude': 48.108192,
    'elevation': 0.450000,
    'longitude': 11.583120})

st[2].stats.paz = AttribDict({
    'poles': [(-0.03736 - 0.03617j), (-0.03736 + 0.03617j)],
    'zeros': [0j, 0j],
    'sensitivity': 250000000.0,
    'gain': 1.0})
st[2].stats.coordinates = AttribDict({
    'latitude': 48.108692,
    'elevation': 0.450000,
    'longitude': 11.583414})

st[3].stats.paz = AttribDict({
    'poles': [(-4.39823 + 4.48709j), (-4.39823 - 4.48709j)],
    'zeros': [0j, 0j],
    'sensitivity': 222222228.10910088,
    'gain': 1.0})
st[3].stats.coordinates = AttribDict({
    'latitude': 48.108456,
    'elevation': 0.450000,
    'longitude': 11.583049})

st[4].stats.paz = AttribDict({
    'poles': [(-4.39823 + 4.48709j), (-4.39823 - 4.48709j), (-2.105 + 0j)],
    'zeros': [0j, 0j, 0j],
    'sensitivity': 222222228.10910088,
    'gain': 1.0})
st[4].stats.coordinates = AttribDict({
    'latitude': 48.108730,
    'elevation': 0.450000,
    'longitude': 11.583157})


# Instrument correction to 1Hz corner frequency
paz1hz = corn_freq_2_paz(1.0, damp=0.707)
st.simulate(paz_remove='self', paz_simulate=paz1hz)

# Execute array_processing
stime = obspy.UTCDateTime("20080217110515")
etime = obspy.UTCDateTime("20080217110545")
kwargs = dict(
    # slowness grid: X min, X max, Y min, Y max, Slow Step
    sll_x=-3.0, slm_x=3.0, sll_y=-3.0, slm_y=3.0, sl_s=0.03,
    # sliding window properties
```

```
    win_len=1.0, win_frac=0.05,
    # frequency properties
    frqlow=1.0, frqhigh=8.0, prewhiten=0,
    # restrict output
    semb_thres=-1e9, vel_thres=-1e9, timestamp='mlabday',
    stime=stime, etime=etime
)
out = array_processing(st, **kwargs)

# Plot
labels = ['rel.power', 'abs.power', 'baz', 'slow']

xlocator = mdates.AutoDateLocator()
fig = plt.figure()
for i, lab in enumerate(labels):
    ax = fig.add_subplot(4, 1, i + 1)
    ax.scatter(out[:, 0], out[:, i + 1], c=out[:, 1], alpha=0.6,
               edgecolors='none', cmap=obspy_sequential)
    ax.set_ylabel(lab)
    ax.set_xlim(out[0, 0], out[-1, 0])
    ax.set_ylim(out[:, i + 1].min(), out[:, i + 1].max())
    ax.xaxis.set_major_locator(xlocator)
    ax.xaxis.set_major_formatter(mdates.AutoDateFormatter(xlocator))

fig.suptitle('AGFA skyscraper blasting in Munich %s' % (
    stime.strftime('%Y-%m-%d'), ))
fig.autofmt_xdate()
fig.subplots_adjust(left=0.15, top=0.95, right=0.95, bottom=0.2, hspace=0)
plt.show()
```

Another representation would be a polar plot, which sums the relative power in gridded bins, each defined by backazimuth and slowness of the analyzed signal part. The backazimuth is counted clockwise from north, the slowness limits can be set by hand.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colorbar import ColorbarBase
from matplotlib.colors import Normalize

import obspy
from obspy.core.util import AttribDict
from obspy.imaging.cm import obspy_sequential
from obspy.signal.invsim import corn_freq_2_paz
from obspy.signal.array_analysis import array_processing


# Load data
st = obspy.read("https://examples.obspy.org/agfa.mseed")

# Set PAZ and coordinates for all 5 channels
st[0].stats.paz = AttribDict({
    'poles': [(-0.03736 - 0.03617j), (-0.03736 + 0.03617j)],
    'zeros': [0j, 0j],
    'sensitivity': 205479446.68601453,
    'gain': 1.0})
st[0].stats.coordinates = AttribDict({
    'latitude': 48.108589,
    'elevation': 0.450000,
```

```
    'longitude': 11.582967})

st[1].stats.paz = AttribDict({
    'poles': [(-0.03736 - 0.03617j), (-0.03736 + 0.03617j)],
    'zeros': [0j, 0j],
    'sensitivity': 205479446.68601453,
    'gain': 1.0})
st[1].stats.coordinates = AttribDict({
    'latitude': 48.108192,
    'elevation': 0.450000,
    'longitude': 11.583120})

st[2].stats.paz = AttribDict({
    'poles': [(-0.03736 - 0.03617j), (-0.03736 + 0.03617j)],
    'zeros': [0j, 0j],
    'sensitivity': 250000000.0,
    'gain': 1.0})
st[2].stats.coordinates = AttribDict({
    'latitude': 48.108692,
    'elevation': 0.450000,
    'longitude': 11.583414})

st[3].stats.paz = AttribDict({
    'poles': [(-4.39823 + 4.48709j), (-4.39823 - 4.48709j)],
    'zeros': [0j, 0j],
    'sensitivity': 222222228.10910088,
    'gain': 1.0})
st[3].stats.coordinates = AttribDict({
    'latitude': 48.108456,
    'elevation': 0.450000,
    'longitude': 11.583049})

st[4].stats.paz = AttribDict({
    'poles': [(-4.39823 + 4.48709j), (-4.39823 - 4.48709j), (-2.105 + 0j)],
    'zeros': [0j, 0j, 0j],
    'sensitivity': 222222228.10910088,
    'gain': 1.0})
st[4].stats.coordinates = AttribDict({
    'latitude': 48.108730,
    'elevation': 0.450000,
    'longitude': 11.583157})


# Instrument correction to 1Hz corner frequency
paz1hz = corn_freq_2_paz(1.0, damp=0.707)
st.simulate(paz_remove='self', paz_simulate=paz1hz)

# Execute array_processing
kwargs = dict(
    # slowness grid: X min, X max, Y min, Y max, Slow Step
    sll_x=-3.0, slm_x=3.0, sll_y=-3.0, slm_y=3.0, sl_s=0.03,
    # sliding window properties
    win_len=1.0, win_frac=0.05,
    # frequency properties
    frqlow=1.0, frqhigh=8.0, prewhiten=0,
    # restrict output
    semb_thres=-1e9, vel_thres=-1e9,
    stime=obspy.UTCDateTime("20080217110515"),
```

```
    etime=obspy.UTCDateTime("20080217110545")
)
out = array_processing(st, **kwargs)

# Plot

cmap = obspy_sequential

# make output human readable, adjust backazimuth to values between 0 and 360
t, rel_power, abs_power, baz, slow = out.T
baz[baz < 0.0] += 360

# choose number of fractions in plot (desirably 360 degree/N is an integer!)
N = 36
N2 = 30
abins = np.arange(N + 1) * 360. / N
sbins = np.linspace(0, 3, N2 + 1)

# sum rel power in bins given by abins and sbins
hist, baz_edges, sl_edges = \
    np.histogram2d(baz, slow, bins=[abins, sbins], weights=rel_power)

# transform to radian
baz_edges = np.radians(baz_edges)

# add polar and colorbar axes
fig = plt.figure(figsize=(8, 8))
cax = fig.add_axes([0.85, 0.2, 0.05, 0.5])
ax = fig.add_axes([0.10, 0.1, 0.70, 0.7], polar=True)
ax.set_theta_direction(-1)
ax.set_theta_zero_location("N")

dh = abs(sl_edges[1] - sl_edges[0])
dw = abs(baz_edges[1] - baz_edges[0])

# circle through backazimuth
for i, row in enumerate(hist):
    bars = ax.bar(left=(i * dw) * np.ones(N2),
                  height=dh * np.ones(N2),
                  width=dw, bottom=dh * np.arange(N2),
                  color=cmap(row / hist.max()))

ax.set_xticks(np.linspace(0, 2 * np.pi, 4, endpoint=False))
ax.set_xticklabels(['N', 'E', 'S', 'W'])

# set slowness limits
ax.set_ylim(0, 3)
[i.set_color('grey') for i in ax.get_yticklabels()]
ColorbarBase(cax, cmap=cmap,
             norm=Normalize(vmin=hist.min(), vmax=hist.max()))

plt.show()
```

1.10 Seismogram Envelopes

The following script shows how to filter a seismogram and plot it together with its envelope.

This example uses a zero-phase-shift bandpass to filter the data with corner frequencies 1 and 3 Hz, using 2 corners (two runs due to zero-phase option, thus 4 corners overall). Then we calculate the envelope and plot it together with the Trace. Data can be found here.

```
import numpy as np
import matplotlib.pyplot as plt

import obspy
import obspy.signal


st = obspy.read("https://examples.obspy.org/RJOB_061005_072159.ehz.new")
data = st[0].data
npts = st[0].stats.npts
samprate = st[0].stats.sampling_rate

# Filtering the Stream object
st_filt = st.copy()
st_filt.filter('bandpass', freqmin=1, freqmax=3, corners=2, zerophase=True)

# Envelope of filtered data
data_envelope = obspy.signal.filter.envelope(st_filt[0].data)

# The plotting, plain matplotlib
t = np.arange(0, npts / samprate, 1 / samprate)
plt.plot(t, st_filt[0].data, 'k')
plt.plot(t, data_envelope, 'k:')
plt.title(st[0].stats.starttime)
plt.ylabel('Filtered Data w/ Envelope')
plt.xlabel('Time [s]')
plt.xlim(80, 90)
plt.show()
```

1.11 Plotting Spectrograms

The following lines of code demonstrate how to make a spectrogram plot of an ObsPy `Stream` object.

Lots of options can be customized, see `spectrogram()` for more details. For example, the colormap of the plot can easily be adjusted by importing a predefined colormap from `matplotlib.cm`, nice overviews of available matplotlib colormaps are given at:

- http://www.physics.ox.ac.uk/users/msshin/science/code/matplotlib_cm/
- http://matplotlib.org/examples/color/colormaps_reference.html
- https://wiki.scipy.org/Cookbook/Matplotlib/Show_colormaps

```
import obspy

st = obspy.read("https://examples.obspy.org/RJOB_061005_072159.ehz.new")
st.spectrogram(log=True, title='BW.RJOB ' + str(st[0].stats.starttime))
```

1.12 Trigger/Picker Tutorial

This is a small tutorial that started as a practical for the UNESCO short course on triggering. Test data used in this tutorial can be downloaded here: trigger_data.zip.

The triggers are implemented as described in [Withers1998]. Information on finding the right trigger parameters for STA/LTA type triggers can be found in [Trnkoczy2012].

See also:

Please note the convenience method of ObsPy's `Stream.trigger` and `Trace.trigger` objects for triggering.

1.12.1 Reading Waveform Data

The data files are read into an ObsPy `Trace` object using the `read()` function.

```
>>> from obspy.core import read
>>> st = read("https://examples.obspy.org/ev0_6.a01.gse2")
>>> st = st.select(component="Z")
>>> tr = st[0]
```

The data format is automatically detected. Important in this tutorial are the `Trace` attributes:

`tr.data` contains the data as `numpy.ndarray`

`tr.stats` contains a dict-like class of header entries

`tr.stats.sampling_rate` the sampling rate

`tr.stats.npts` sample count of data

As an example, the header of the data file is printed and the data are plotted like this:

```
>>> print(tr.stats)
         network:
         station: EV0_6
        location:
         channel: EHZ
       starttime: 1970-01-01T01:00:00.000000Z
         endtime: 1970-01-01T01:00:59.995000Z
   sampling_rate: 200.0
           delta: 0.005
            npts: 12000
           calib: 1.0
         _format: GSE2
            gse2: AttribDict({'instype': '      ', 'datatype': 'CM6', 'hang': 0.0, 'auxid': '     ', '
```

Using the `plot()` method of the `Trace` objects will show the plot.

```
>>> tr.plot(type="relative")
```

1.12.2 Available Methods

After loading the data, we are able to pass the waveform data to the following trigger routines defined in `obspy.signal.trigger`:

recursive_sta_lta(a, nsta, nlta)	Recursive STA/LTA.
carl_sta_trig(a, nsta, nlta, ratio, quiet)	Computes the carlSTAtrig characteristic function.
classic_sta_lta(a, nsta, nlta)	Computes the standard STA/LTA from a given input array a. The length of
delayed_sta_lta(a, nsta, nlta)	Delayed STA/LTA.
z_detect(a, nsta)	Z-detector.
pk_baer(reltrc, samp_int, tdownmax, ...)	Wrapper for P-picker routine by M. Baer, Schweizer Erdbebendienst.
ar_pick(a, b, c, samp_rate, f1, f2, lta_p, ...)	Pick P and S arrivals with an AR-AIC + STA/LTA algorithm.

obspy.signal.trigger.recursive_sta_lta

recursive_sta_lta(*a*, *nsta*, *nlta*)

Recursive STA/LTA.

Fast version written in C.

Note This version directly uses a C version via CTypes

Parameters

- **a** (numpy.ndarray, dtype=float64) – Seismic Trace, numpy.ndarray dtype float64
- **nsta** (*int*) – Length of short time average window in samples
- **nlta** (*int*) – Length of long time average window in samples

Return type numpy.ndarray, dtype=float64

Returns Characteristic function of recursive STA/LTA

See also:

[Withers1998] (p. 98) and [Trnkoczy2012]

obspy.signal.trigger.carl_sta_trig

carl_sta_trig(*a*, *nsta*, *nlta*, *ratio*, *quiet*)

Computes the carlSTAtrig characteristic function.

eta = star - (ratio * ltar) - abs(sta - lta) - quiet

Parameters

- **a** (NumPy ndarray) – Seismic Trace
- **nsta** (*int*) – Length of short time average window in samples
- **nlta** (*int*) – Length of long time average window in samples
- **ratio** – as ratio gets smaller, carl_sta_trig gets more sensitive
- **quiet** (*float*) – as quiet gets smaller, carl_sta_trig gets more sensitive

Return type NumPy ndarray

Returns Characteristic function of CarlStaTrig

obspy.signal.trigger.classic_sta_lta

classic_sta_lta(*a*, *nsta*, *nlta*)
Computes the standard STA/LTA from a given input array a. The length of the STA is given by nsta in samples, respectively is the length of the LTA given by nlta in samples.

Fast version written in C.

Parameters

- **a** (NumPy ndarray) – Seismic Trace
- **nsta** (*int*) – Length of short time average window in samples
- **nlta** (*int*) – Length of long time average window in samples

Return type NumPy ndarray

Returns Characteristic function of classic STA/LTA

obspy.signal.trigger.delayed_sta_lta

delayed_sta_lta(*a*, *nsta*, *nlta*)
Delayed STA/LTA.

Parameters

- **a** (NumPy ndarray) – Seismic Trace
- **nsta** (*int*) – Length of short time average window in samples
- **nlta** (*int*) – Length of long time average window in samples

Return type NumPy ndarray

Returns Characteristic function of delayed STA/LTA

See also:

[Withers1998] (p. 98) and [Trnkoczy2012]

obspy.signal.trigger.z_detect

z_detect(*a*, *nsta*)
Z-detector.

Parameters **nsta** – Window length in Samples.

See also:

[Withers1998], p. 99

obspy.signal.trigger.pk_baer

pk_baer(*reltrc*, *samp_int*, *tdownmax*, *tupevent*, *thr1*, *thr2*, *preset_len*, *p_dur*)
Wrapper for P-picker routine by M. Baer, Schweizer Erdbebendienst.

Parameters

- **reltrc** – time series as numpy.ndarray float32 data, possibly filtered
- **samp_int** – number of samples per second

- **tdownmax** – if dtime exceeds tdownmax, the trigger is examined for validity
- **tupevent** – min nr of samples for itrm to be accepted as a pick
- **thr1** – threshold to trigger for pick (c.f. paper)
- **thr2** – threshold for updating sigma (c.f. paper)
- **preset_len** – no of points taken for the estimation of variance of SF(t) on preset()
- **p_dur** – p_dur defines the time interval for which the maximum amplitude is evaluated Originally set to 6 secs

Returns (pptime, pfm) pptime sample number of parrival; pfm direction of first motion (U or D)

Note: currently the first sample is not taken into account

See also:

[Baer1987]

obspy.signal.trigger.ar_pick

ar_pick(*a*, *b*, *c*, *samp_rate*, *f1*, *f2*, *lta_p*, *sta_p*, *lta_s*, *sta_s*, *m_p*, *m_s*, *l_p*, *l_s*, *s_pick=True*)
Pick P and S arrivals with an AR-AIC + STA/LTA algorithm.

The algorithm picks onset times using an Auto Regression - Akaike Information Criterion (AR-AIC) method. The detection intervals are successively narrowed down with the help of STA/LTA ratios as well as STA-LTA difference calculations. For details, please see [Akazawa2004].

An important feature of this algorithm is that it requires comparatively little tweaking and site-specific settings and is thus applicable to large, diverse data sets.

Parameters

- **a** (`numpy.ndarray`) – Z signal the data.
- **b** (`numpy.ndarray`) – N signal of the data.
- **c** (`numpy.ndarray`) – E signal of the data.
- **samp_rate** (*float*) – Number of samples per second.
- **f1** (*float*) – Frequency of the lower bandpass window.
- **f2** (*float*) – Frequency of the upper .andpass window.
- **lta_p** (*float*) – Length of LTA for the P arrival in seconds.
- **sta_p** (*float*) – Length of STA for the P arrival in seconds.
- **lta_s** (*float*) – Length of LTA for the S arrival in seconds.
- **sta_s** (*float*) – Length of STA for the S arrival in seconds.
- **m_p** (*int*) – Number of AR coefficients for the P arrival.
- **m_s** (*int*) – Number of AR coefficients for the S arrival.
- **l_p** (*float*) – Length of variance window for the P arrival in seconds.
- **l_s** (*float*) – Length of variance window for the S arrival in seconds.
- **s_pick** (*bool*) – If `True`, also pick the S phase, otherwise only the P phase.

Return type tuple

Returns A tuple with the P and the S arrival.

Help for each function is available HTML formatted or in the usual Python manner:

```
>>> from obspy.signal.trigger import classic_sta_lta
>>> help(classic_sta_lta)
Help on function classic_sta_lta in module obspy.signal.trigger...
```

The triggering itself mainly consists of the following two steps:

- Calculating the characteristic function
- Setting picks based on values of the characteristic function

1.12.3 Trigger Examples

For all the examples, the commands to read in the data and to load the modules are the following:

```
>>> from obspy.core import read
>>> from obspy.signal.trigger import plot_trigger
>>> trace = read("https://examples.obspy.org/ev0_6.a01.gse2")[0]
>>> df = trace.stats.sampling_rate
```

Classic Sta Lta

```
>>> from obspy.signal.trigger import classic_sta_lta
>>> cft = classic_sta_lta(trace.data, int(5 * df), int(10 * df))
>>> plot_trigger(trace, cft, 1.5, 0.5)
```

Z-Detect

```
>>> from obspy.signal.trigger import z_detect
>>> cft = z_detect(trace.data, int(10 * df))
>>> plot_trigger(trace, cft, -0.4, -0.3)
```

Recursive Sta Lta

```
>>> from obspy.signal.trigger import recursive_sta_lta
>>> cft = recursive_sta_lta(trace.data, int(5 * df), int(10 * df))
>>> plot_trigger(trace, cft, 1.2, 0.5)
```

Carl-Sta-Trig

```
>>> from obspy.signal.trigger import carl_sta_trig
>>> cft = carl_sta_trig(trace.data, int(5 * df), int(10 * df), 0.8, 0.8)
>>> plot_trigger(trace, cft, 20.0, -20.0)
```

Delayed Sta Lta

```
>>> from obspy.signal.trigger import delayed_sta_lta
>>> cft = delayed_sta_lta(trace.data, int(5 * df), int(10 * df))
>>> plot_trigger(trace, cft, 5, 10)
```

1.12.4 Network Coincidence Trigger Example

In this example we perform a coincidence trigger on a local scale network of 4 stations. For the single station triggers a recursive STA/LTA is used. The waveform data span about four minutes and include four local events. Two are easily recognizable (Ml 1-2), the other two can only be detected with well adjusted trigger settings (Ml <= 0).

First we assemble a Stream object with all waveform data, the data used in the example is available from our web server:

```
>>> from obspy.core import Stream, read
>>> st = Stream()
>>> files = ["BW.UH1..SHZ.D.2010.147.cut.slist.gz",
...          "BW.UH2..SHZ.D.2010.147.cut.slist.gz",
...          "BW.UH3..SHZ.D.2010.147.cut.slist.gz",
...          "BW.UH4..SHZ.D.2010.147.cut.slist.gz"]
>>> for filename in files:
...     st += read("https://examples.obspy.org/" + filename)
```

After applying a bandpass filter we run the coincidence triggering on all data. In the example a recursive STA/LTA is used. The trigger parameters are set to 0.5 and 10 second time windows, respectively. The on-threshold is set to 3.5, the off-threshold to 1. In this example every station gets a weight of 1 and the coincidence sum threshold is set to 3. For more complex network setups the weighting for every station/channel can be customized. We want to keep our original data so we work with a copy of the original stream:

```
>>> st.filter('bandpass', freqmin=10, freqmax=20)  # optional prefiltering
>>> from obspy.signal.trigger import coincidence_trigger
>>> st2 = st.copy()
>>> trig = coincidence_trigger("recstalta", 3.5, 1, st2, 3, sta=0.5, lta=10)
```

Using pretty print the results display like this:

```
>>> from pprint import pprint
>>> pprint(trig)
[{'coincidence_sum': 4.0,
  'duration': 4.5299999713897705,
  'stations': ['UH3', 'UH2', 'UH1', 'UH4'],
  'time': UTCDateTime(2010, 5, 27, 16, 24, 33, 190000),
  'trace_ids': ['BW.UH3..SHZ', 'BW.UH2..SHZ', 'BW.UH1..SHZ',
                'BW.UH4..SHZ']},
 {'coincidence_sum': 3.0,
  'duration': 3.440000057220459,
  'stations': ['UH2', 'UH3', 'UH1'],
  'time': UTCDateTime(2010, 5, 27, 16, 27, 1, 260000),
  'trace_ids': ['BW.UH2..SHZ', 'BW.UH3..SHZ', 'BW.UH1..SHZ']},
 {'coincidence_sum': 4.0,
  'duration': 4.7899999618530273,
  'stations': ['UH3', 'UH2', 'UH1', 'UH4'],
  'time': UTCDateTime(2010, 5, 27, 16, 27, 30, 490000),
  'trace_ids': ['BW.UH3..SHZ', 'BW.UH2..SHZ', 'BW.UH1..SHZ',
                'BW.UH4..SHZ']}]
```

With these settings the coincidence trigger reports three events. For each (possible) event the start time and duration is provided. Furthermore, a list of station names and trace IDs is provided, ordered by the time the stations have triggered, which can give a first rough idea of the possible event location. We can request additional information by specifying `details=True`:

```
>>> st2 = st.copy()
>>> trig = coincidence_trigger("recstalta", 3.5, 1, st2, 3, sta=0.5, lta=10,
...                            details=True)
```

For clarity, we only display information on the first item in the results here:

```
>>> pprint(trig[0])
{'cft_peak_wmean': 19.561900329259956,
 'cft_peaks': [19.535644192544272,
               19.872432918501264,
               19.622171410201297,
               19.217352795792998],
 'cft_std_wmean': 5.4565629691954713,
 'cft_stds': [5.292458320417178,
              5.6565387957966404,
              5.7582248973698507,
              5.1190298631982163],
 'coincidence_sum': 4.0,
 'duration': 4.5299999713897705,
 'stations': ['UH3', 'UH2', 'UH1', 'UH4'],
 'time': UTCDateTime(2010, 5, 27, 16, 24, 33, 190000),
 'trace_ids': ['BW.UH3..SHZ', 'BW.UH2..SHZ', 'BW.UH1..SHZ', 'BW.UH4..SHZ']}
```

Here, some additional information on the peak values and standard deviations of the characteristic functions of the single station triggers is provided. Also, for both a weighted mean is calculated. These values can help to distinguish certain from questionable network triggers.

For more information on all possible options see the documentation page for `coincidence_trigger()`.

1.12.5 Advanced Network Coincidence Trigger Example with Similarity Detection

This example is an extension of the common network coincidence trigger. Waveforms with already known event(s) can be provided to check waveform similarity of single-station triggers. If the corresponding similarity threshold is exceeded the event trigger is included in the result list even if the coincidence sum does not exceed the specified minimum coincidence sum. Using this approach, events can be detected that have good recordings on one station with very similar waveforms but for some reason are not detected on enough other stations (e.g. temporary station outages or local high noise levels etc.). An arbitrary number of template waveforms can be provided for any station. Computation time might get significantly higher due to the necessary cross correlations. In the example we use two three-component event templates on top of a common network trigger on vertical components only.

```
>>> from obspy.core import Stream, read, UTCDateTime
>>> st = Stream()
>>> files = ["BW.UH1..SHZ.D.2010.147.cut.slist.gz",
...          "BW.UH2..SHZ.D.2010.147.cut.slist.gz",
...          "BW.UH3..SHZ.D.2010.147.cut.slist.gz",
...          "BW.UH3..SHN.D.2010.147.cut.slist.gz",
...          "BW.UH3..SHE.D.2010.147.cut.slist.gz",
...          "BW.UH4..SHZ.D.2010.147.cut.slist.gz"]
>>> for filename in files:
...     st += read("https://examples.obspy.org/" + filename)
>>> st.filter('bandpass', freqmin=10, freqmax=20)  # optional prefiltering
```

Here we set up a dictionary with template events for one single station. The specified times are exact P wave onsets, the event duration (including S wave) is about 2.5 seconds. On station UH3 we use two template events with three-component data, on station UH1 we use one template event with only vertical component data.

```
>>> times = ["2010-05-27T16:24:33.095000", "2010-05-27T16:27:30.370000"]
>>> event_templates = {"UH3": []}
>>> for t in times:
...     t = UTCDateTime(t)
...     st_ = st.select(station="UH3").slice(t, t + 2.5)
...     event_templates["UH3"].append(st_)
```

```
>>> t = UTCDateTime("2010-05-27T16:27:30.574999")
>>> st_ = st.select(station="UH1").slice(t, t + 2.5)
>>> event_templates["UH1"] = [st_]
```

The triggering step, including providing of similarity threshold and event template waveforms. Note that the coincidence sum is set to 4 and we manually specify to only use vertical components with equal station coincidence values of 1.

```
>>> from obspy.signal.trigger import coincidence_trigger
>>> st2 = st.copy()
>>> trace_ids = {"BW.UH1..SHZ": 1,
...              "BW.UH2..SHZ": 1,
...              "BW.UH3..SHZ": 1,
...              "BW.UH4..SHZ": 1}
>>> similarity_thresholds = {"UH1": 0.8, "UH3": 0.7}
>>> trig = coincidence_trigger("classicstalta", 5, 1, st2, 4, sta=0.5,
...                            lta=10, trace_ids=trace_ids,
...                            event_templates=event_templates,
...                            similarity_threshold=similarity_thresholds)
```

The results now include two event triggers, that do not reach the specified minimum coincidence threshold but that have a similarity value that exceeds the specified similarity threshold when compared to at least one of the provided event template waveforms. Note the values of 1.0 when checking the event triggers where we extracted the event templates for this example.

```
>>> from pprint import pprint
>>> pprint(trig)
[{'coincidence_sum': 4.0,
  'duration': 4.1100001335144043,
  'similarity': {'UH1': 0.9414944738498271, 'UH3': 1.0},
  'stations': ['UH3', 'UH2', 'UH1', 'UH4'],
  'time': UTCDateTime(2010, 5, 27, 16, 24, 33, 210000),
  'trace_ids': ['BW.UH3..SHZ', 'BW.UH2..SHZ', 'BW.UH1..SHZ', 'BW.UH4..SHZ']},
 {'coincidence_sum': 3.0,
  'duration': 1.9900000095367432,
  'similarity': {'UH1': 0.65228204570577764, 'UH3': 0.72679293429214198},
  'stations': ['UH3', 'UH1', 'UH2'],
  'time': UTCDateTime(2010, 5, 27, 16, 25, 26, 710000),
  'trace_ids': ['BW.UH3..SHZ', 'BW.UH1..SHZ', 'BW.UH2..SHZ']},
 {'coincidence_sum': 3.0,
  'duration': 1.9200000762939453,
  'similarity': {'UH1': 0.89404458774338103, 'UH3': 0.74581409371425222},
  'stations': ['UH2', 'UH1', 'UH3'],
  'time': UTCDateTime(2010, 5, 27, 16, 27, 2, 260000),
  'trace_ids': ['BW.UH2..SHZ', 'BW.UH1..SHZ', 'BW.UH3..SHZ']},
 {'coincidence_sum': 4.0,
  'duration': 4.0299999713897705,
  'similarity': {'UH1': 1.0, 'UH3': 1.0},
  'stations': ['UH3', 'UH2', 'UH1', 'UH4'],
  'time': UTCDateTime(2010, 5, 27, 16, 27, 30, 510000),
  'trace_ids': ['BW.UH3..SHZ', 'BW.UH2..SHZ', 'BW.UH1..SHZ', 'BW.UH4..SHZ']}]
```

For more information on all possible options see the documentation page for `coincidence_trigger()`.

1.12.6 Picker Examples

Baer Picker

For `pk_baer()`, input is in seconds, output is in samples.

```
>>> from obspy.core import read
>>> from obspy.signal.trigger import pk_baer
>>> trace = read("https://examples.obspy.org/ev0_6.a01.gse2")[0]
>>> df = trace.stats.sampling_rate
>>> p_pick, phase_info = pk_baer(trace.data, df,
...                              20, 60, 7.0, 12.0, 100, 100)
>>> print(p_pick)
6894
>>> print(phase_info)
EPU3
>>> print(p_pick / df)
34.47
```

This yields the output 34.47 EPU3, which means that a P pick was set at 34.47s with Phase information EPU3.

AR Picker

For `ar_pick()`, input and output are in seconds.

```
>>> from obspy.core import read
>>> from obspy.signal.trigger import ar_pick
>>> tr1 = read('https://examples.obspy.org/loc_RJOB20050801145719850.z.gse2')[0]
>>> tr2 = read('https://examples.obspy.org/loc_RJOB20050801145719850.n.gse2')[0]
>>> tr3 = read('https://examples.obspy.org/loc_RJOB20050801145719850.e.gse2')[0]
>>> df = tr1.stats.sampling_rate
>>> p_pick, s_pick = ar_pick(tr1.data, tr2.data, tr3.data, df,
...                          1.0, 20.0, 1.0, 0.1, 4.0, 1.0, 2, 8, 0.1, 0.2)
>>> print(p_pick)
30.6350002289
>>> print(s_pick)
31.2800006866
```

This gives the output 30.6350002289 and 31.2800006866, meaning that a P pick at 30.64s and an S pick at 31.28s were identified.

1.12.7 Advanced Example

A more complicated example, where the data are retrieved via ArcLink and results are plotted step by step, is shown here:

```
import matplotlib.pyplot as plt

import obspy
from obspy.clients.arclink import Client
from obspy.signal.trigger import recursive_sta_lta, trigger_onset


# Retrieve waveforms via ArcLink
client = Client(host="erde.geophysik.uni-muenchen.de", port=18001,
                user="test@obspy.de")
```

```
t = obspy.UTCDateTime("2009-08-24 00:19:45")
st = client.get_waveforms('BW', 'RTSH', '', 'EHZ', t, t + 50)

# For convenience
tr = st[0]  # only one trace in mseed volume
df = tr.stats.sampling_rate

# Characteristic function and trigger onsets
cft = recursive_sta_lta(tr.data, int(2.5 * df), int(10. * df))
on_of = trigger_onset(cft, 3.5, 0.5)

# Plotting the results
ax = plt.subplot(211)
plt.plot(tr.data, 'k')
ymin, ymax = ax.get_ylim()
plt.vlines(on_of[:, 0], ymin, ymax, color='r', linewidth=2)
plt.vlines(on_of[:, 1], ymin, ymax, color='b', linewidth=2)
plt.subplot(212, sharex=ax)
plt.plot(cft, 'k')
plt.hlines([3.5, 0.5], 0, len(cft), color=['r', 'b'], linestyle='--')
plt.axis('tight')
plt.show()
```

1.13 Poles and Zeros, Frequency Response

Note: For metadata read using `read_inventory()` into `Inventory` objects (and the corresponding sub-objects `Network`, `Station`, `Channel`, `Response`), there is a convenience method to show Bode plots, see e.g. `Inventory.plot_response()` or `Response.plot()`).

The following lines show how to calculate and visualize the frequency response of a LE-3D/1s seismometer with sampling interval 0.005s and 16384 points of fft. We want the phase to go from 0 to 2*pi, instead of the output from angle that goes from -pi to pi .

```
import numpy as np
import matplotlib.pyplot as plt

from obspy.signal.invsim import paz_to_freq_resp


poles = [-4.440 + 4.440j, -4.440 - 4.440j, -1.083 + 0.0j]
zeros = [0.0 + 0.0j, 0.0 + 0.0j, 0.0 + 0.0j]
scale_fac = 0.4

h, f = paz_to_freq_resp(poles, zeros, scale_fac, 0.005, 16384, freq=True)

plt.figure()
plt.subplot(121)
plt.loglog(f, abs(h))
plt.xlabel('Frequency [Hz]')
plt.ylabel('Amplitude')

plt.subplot(122)
phase = 2 * np.pi + np.unwrap(np.angle(h))
plt.semilogx(f, phase)
plt.xlabel('Frequency [Hz]')
```

```
plt.ylabel('Phase [radian]')
# ticks and tick labels at multiples of pi
plt.yticks(
    [0, np.pi / 2, np.pi, 3 * np.pi / 2, 2 * np.pi],
    ['$0$', r'$\frac{\pi}{2}$', r'$\pi$', r'$\frac{3\pi}{2}$', r'$2\pi$'])
plt.ylim(-0.2, 2 * np.pi + 0.2)
# title, centered above both subplots
plt.suptitle('Frequency Response of LE-3D/1s Seismometer')
# make more room in between subplots for the ylabel of right plot
plt.subplots_adjust(wspace=0.3)
plt.show()
```

1.14 Seismometer Correction/Simulation

1.14.1 Calculating response from filter stages using evalresp..

..using a StationXML file or in general an `Inventory` object

When using the `FDSN client` the response can directly be attached to the waveforms and then subsequently removed using `Stream.remove_response()`:

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

t1 = UTCDateTime("2010-09-3T16:30:00.000")
t2 = UTCDateTime("2010-09-3T17:00:00.000")
fdsn_client = Client('IRIS')
# Fetch waveform from IRIS FDSN web service into a ObsPy stream object
# and automatically attach correct response
st = fdsn_client.get_waveforms(network='NZ', station='BFZ', location='10',
                               channel='HHZ', starttime=t1, endtime=t2,
                               attach_response=True)
# define a filter band to prevent amplifying noise during the deconvolution
pre_filt = (0.005, 0.006, 30.0, 35.0)
st.remove_response(output='DISP', pre_filt=pre_filt)
```

Alternatively an `Inventory` object can be directly passed to the `Stream.remove_response()`: method:

```
from obspy import read, read_inventory

# simply use the included example waveform
st = read()
# the corresponding response is included in ObsPy as a StationXML file
inv = read_inventory()
# the routine automatically picks the correct response for each trace
# define a filter band to prevent amplifying noise during the deconvolution
pre_filt = (0.005, 0.006, 30.0, 35.0)
st.remove_response(inventory=inv, output='DISP', pre_filt=pre_filt)
```

Using the *plot* option it is possible to visualize the individual steps during response removal in the frequency domain to check the chosen *pre_filt* and *water_level* options to stabilize the deconvolution of the inverted instrument response spectrum:

```
from obspy import read, read_inventory
```

```
st = read("/path/to/IU_ULN_00_LH1_2015-07-18T02.mseed")
tr = st[0]
inv = read_inventory("/path/to/IU_ULN_00_LH1.xml")
pre_filt = [0.001, 0.005, 10, 20]
tr.remove_response(inventory=inv, pre_filt=pre_filt, output="DISP",
                   water_level=60, plot=True)
```

..using a RESP file

It is further possible to use evalresp to evaluate the instrument response information from a RESP file.

```
import matplotlib.pyplot as plt

import obspy
from obspy.core.util import NamedTemporaryFile
from obspy.clients.fdsn import Client as FDSN_Client
from obspy.clients.iris import Client as OldIris_Client


# MW 7.1 Darfield earthquake, New Zealand
t1 = obspy.UTCDateTime("2010-09-3T16:30:00.000")
t2 = obspy.UTCDateTime("2010-09-3T17:00:00.000")

# Fetch waveform from IRIS FDSN web service into a ObsPy stream object
fdsn_client = FDSN_Client("IRIS")
st = fdsn_client.get_waveforms('NZ', 'BFZ', '10', 'HHZ', t1, t2)

# Download and save instrument response file into a temporary file
with NamedTemporaryFile() as tf:
    respf = tf.name
    old_iris_client = OldIris_Client()
    # fetch RESP information from "old" IRIS web service, see obspy.fdsn
    # for accessing the new IRIS FDSN web services
    old_iris_client.resp('NZ', 'BFZ', '10', 'HHZ', t1, t2, filename=respf)

    # make a copy to keep our original data
    st_orig = st.copy()

    # define a filter band to prevent amplifying noise during the deconvolution
    pre_filt = (0.005, 0.006, 30.0, 35.0)

    # this can be the date of your raw data or any date for which the
    # SEED RESP-file is valid
    date = t1

    seedresp = {'filename': respf,  # RESP filename
                # when using Trace/Stream.simulate() the "date" parameter can
                # also be omitted, and the starttime of the trace is then used.
                'date': date,
                # Units to return response in ('DIS', 'VEL' or ACC)
                'units': 'DIS'
                }

    # Remove instrument response using the information from the given RESP file
    st.simulate(paz_remove=None, pre_filt=pre_filt, seedresp=seedresp)

# plot original and simulated data
```

```
tr = st[0]
tr_orig = st_orig[0]
time = tr.times()

plt.subplot(211)
plt.plot(time, tr_orig.data, 'k')
plt.ylabel('STS-2 [counts]')
plt.subplot(212)
plt.plot(time, tr.data, 'k')
plt.ylabel('Displacement [m]')
plt.xlabel('Time [s]')
plt.show()
```

..using a Dataless/Full SEED file (or XMLSEED file)

A `Parser` object created using a Dataless SEED file can also be used. For each trace the respective RESP response data is extracted internally then. When using `Stream`/`Trace`'s `simulate()` convenience methods the "date" parameter can be omitted (each trace's start time is used internally).

```
import obspy
from obspy.io.xseed import Parser


st = obspy.read("https://examples.obspy.org/BW.BGLD..EH.D.2010.037")
parser = Parser("https://examples.obspy.org/dataless.seed.BW_BGLD")
st.simulate(seedresp={'filename': parser, 'units': "DIS"})
```

1.14.2 Using a PAZ dictionary

The following script shows how to simulate a 1Hz seismometer from a STS-2 seismometer with the given poles and zeros. Poles, zeros, gain (*A0 normalization factor*) and sensitivity (*overall sensitivity*) are specified as keys of a dictionary.

```
import obspy
from obspy.signal.invsim import corn_freq_2_paz


paz_sts2 = {
    'poles': [-0.037004 + 0.037016j, -0.037004 - 0.037016j, -251.33 + 0j,
              - 131.04 - 467.29j, -131.04 + 467.29j],
    'zeros': [0j, 0j],
    'gain': 60077000.0,
    'sensitivity': 2516778400.0}
paz_1hz = corn_freq_2_paz(1.0, damp=0.707)  # 1Hz instrument
paz_1hz['sensitivity'] = 1.0

st = obspy.read()
# make a copy to keep our original data
st_orig = st.copy()

# Simulate instrument given poles, zeros and gain of
# the original and desired instrument
st.simulate(paz_remove=paz_sts2, paz_simulate=paz_1hz)

# plot original and simulated data
```

```
st_orig.plot()
st.plot()
```

For more customized plotting we could also work with matplotlib manually from here:

```
import numpy as np
import matplotlib.pyplot as plt

tr = st[0]
tr_orig = st_orig[0]

t = np.arange(tr.stats.npts) / tr.stats.sampling_rate

plt.subplot(211)
plt.plot(t, tr_orig.data, 'k')
plt.ylabel('STS-2 [counts]')
plt.subplot(212)
plt.plot(t, tr.data, 'k')
plt.ylabel('1Hz Instrument [m/s]')
plt.xlabel('Time [s]')
plt.show()
```

1.15 Clone an Existing Dataless SEED File

The following code example shows how to clone an existing DatalessSEED file (`dataless.seed.BW_RNON`) and use it as a template to build up a DatalessSEED file for a new station.

First of all, we have to make the necessary imports and read the existing DatalessSEED volume (stored on our examples webserver):

```
>>> from obspy import UTCDateTime
>>> from obspy.io.xseed import Parser
>>>
>>> p = Parser("https://examples.obspy.org/dataless.seed.BW_RNON")
>>> blk = p.blockettes
```

Now we can adapt the information only appearing once in the DatalessSEED at the start of the file, in this case Blockette 50 and the abbreviations in Blockette 33:

```
>>> blk[50][0].network_code = 'BW'
>>> blk[50][0].station_call_letters = 'RMOA'
>>> blk[50][0].site_name = "Moar Alm, Bavaria, BW-Net"
>>> blk[50][0].latitude = 47.761658
>>> blk[50][0].longitude = 12.864466
>>> blk[50][0].elevation = 815.0
>>> blk[50][0].start_effective_date = UTCDateTime("2006-07-18T00:00:00.000000Z")
>>> blk[50][0].end_effective_date = ""
>>> blk[33][1].abbreviation_description = "Lennartz LE-3D/1 seismometer"
```

After that we have to change the information for all of the three channels involved:

```
>>> mult = len(blk[58])/3
>>> for i, cha in enumerate(['Z', 'N', 'E']):
...     blk[52][i].channel_identifier = 'EH%s' % cha
...     blk[52][i].location_identifier = ''
...     blk[52][i].latitude = blk[50][0].latitude
...     blk[52][i].longitude = blk[50][0].longitude
```

```
...     blk[52][i].elevation = blk[50][0].elevation
...     blk[52][i].start_date = blk[50][0].start_effective_date
...     blk[52][i].end_date = blk[50][0].end_effective_date
...     blk[53][i].number_of_complex_poles = 3
...     blk[53][i].real_pole = [-4.444, -4.444, -1.083]
...     blk[53][i].imaginary_pole = [+4.444, -4.444, +0.0]
...     blk[53][i].real_pole_error = [0, 0, 0]
...     blk[53][i].imaginary_pole_error = [0, 0, 0]
...     blk[53][i].number_of_complex_zeros = 3
...     blk[53][i].real_zero = [0.0, 0.0, 0.0]
...     blk[53][i].imaginary_zero = [0.0, 0.0, 0.0]
...     blk[53][i].real_zero_error = [0, 0, 0]
...     blk[53][i].imaginary_zero_error = [0, 0, 0]
...     blk[53][i].A0_normalization_factor = 1.0
...     blk[53][i].normalization_frequency = 3.0
...     # stage sequence number 1, seismometer gain
...     blk[58][i*mult].sensitivity_gain = 400.0
...     # stage sequence number 2, digitizer gain
...     blk[58][i*mult+1].sensitivity_gain = 1677850.0
...     # stage sequence number 0, overall sensitivity
...     blk[58][(i+1)*mult-1].sensitivity_gain = 671140000.0
```

Note: FIR coefficients are not set in this example. In case you require correct FIR coefficients, either clone from an existing dataless file with the same seismometer type or set the corresponding blockettes with the correct values.

At the end we can write the adapted DatalessSEED volume to a new file:

```
>>> p.write_seed("dataless.seed.BW_RMOA")
```

1.16 Export Seismograms to MATLAB

The following example shows how to read in a waveform file with Python and save each `Trace` in the resulting `Stream` object to one MATLAB .MAT file. The data can the be loaded from within MATLAB with the `load` function.

```
from scipy.io import savemat

import obspy


st = obspy.read("https://examples.obspy.org/BW.BGLD..EH.D.2010.037")
for i, tr in enumerate(st):
    mdict = {k: str(v) for k, v in tr.stats.iteritems()}
    mdict['data'] = tr.data
    savemat("data-%d.mat" % i, mdict)
```

1.17 Export Seismograms to ASCII

1.17.1 Built-in Formats

You may directly export waveform data to any ASCII format available by ObsPy using the `write()` method on the generated `Stream` object.

```
>>> from obspy.core import read
>>> stream = read('https://examples.obspy.org/RJOB20090824.ehz')
>>> stream.write('outfile.ascii', format='SLIST')
```

The following ASCII formats are currently supported:

- `SLIST`, a ASCII time series format represented with a header line followed by a sample lists (see also `SLIST format description`):

  ```
  TIMESERIES BW_RJOB__EHZ_D, 6001 samples, 200 sps, 2009-08-24T00:20:03.000000, SLIST, INTEGER,
  288 300 292 285 265 287
  279 250 278 278 268 258
  ...
  ```

- `TSPAIR`, a ASCII format where data is written in time-sample pairs (see also `TSPAIR format description`):

  ```
  TIMESERIES BW_RJOB__EHZ_D, 6001 samples, 200 sps, 2009-08-24T00:20:03.000000, TSPAIR, INTEGER,
  2009-08-24T00:20:03.000000  288
  2009-08-24T00:20:03.005000  300
  2009-08-24T00:20:03.010000  292
  2009-08-24T00:20:03.015000  285
  2009-08-24T00:20:03.020000  265
  2009-08-24T00:20:03.025000  287
  ...
  ```

- `SH_ASC`, ASCII format supported by Seismic Handler:

  ```
  DELTA: 5.000000e-03
  LENGTH: 6001
  START: 24-AUG-2009_00:20:03.000
  COMP: Z
  CHAN1: E
  CHAN2: H
  STATION: RJOB
  CALIB: 1.000000e+00
  2.880000e+02 3.000000e+02 2.920000e+02 2.850000e+02
  2.650000e+02 2.870000e+02 2.790000e+02 2.500000e+02
  ...
  ```

1.17.2 Custom Format

In the following, a small Python script is shown which converts each `Trace` of a seismogram file to an ASCII file with a custom header. Waveform data will be multiplied by a given `calibration` factor and written using NumPy's `savetxt()` function.

```
"""
USAGE: export_seismograms_to_ascii.py in_file out_file calibration
"""
from __future__ import print_function

import sys

import numpy as np
import obspy


try:
```

```
    in_file = sys.argv[1]
    out_file = sys.argv[2]
    calibration = float(sys.argv[3])
except:
    print(__doc__)
    raise

st = obspy.read(in_file)
for i, tr in enumerate(st):
    f = open("%s_%d" % (out_file, i), "w")
    f.write("# STATION %s\n" % (tr.stats.station))
    f.write("# CHANNEL %s\n" % (tr.stats.channel))
    f.write("# START_TIME %s\n" % (str(tr.stats.starttime)))
    f.write("# SAMP_FREQ %f\n" % (tr.stats.sampling_rate))
    f.write("# NDAT %d\n" % (tr.stats.npts))
    np.savetxt(f, tr.data * calibration, fmt="%f")
    f.close()
```

1.18 Anything to MiniSEED

The following lines show how you can convert anything to MiniSEED format. In the example, a few lines of a weather station output are written to a MiniSEED file. The correct meta information `starttime`, the `sampling_rate`, `station` name and so forth are also encoded (Note: Only the ones given are allowed by the MiniSEED standard). Converting arbitrary ASCII to MiniSEED is extremely helpful if you want to send log messages, output of meteorologic stations or anything else via the SeedLink protocol.

```
from __future__ import print_function

import numpy as np
from obspy import UTCDateTime, read, Trace, Stream


weather = """
00.0000 0.0 ??? 4.7 97.7 1015.0 0.0 010308 000000
00.0002 0.0 ??? 4.7 97.7 1015.0 0.0 010308 000001
00.0005 0.0 ??? 4.7 97.7 1015.0 0.0 010308 000002
00.0008 0.0 ??? 4.7 97.7 1015.4 0.0 010308 000003
00.0011 0.0 ??? 4.7 97.7 1015.0 0.0 010308 000004
00.0013 0.0 ??? 4.7 97.7 1015.0 0.0 010308 000005
00.0016 0.0 ??? 4.7 97.7 1015.0 0.0 010308 000006
00.0019 0.0 ??? 4.7 97.7 1015.0 0.0 010308 000007
"""

# Convert to NumPy character array
data = np.fromstring(weather, dtype='|S1')

# Fill header attributes
stats = {'network': 'BW', 'station': 'RJOB', 'location': '',
         'channel': 'WLZ', 'npts': len(data), 'sampling_rate': 0.1,
         'mseed': {'dataquality': 'D'}}
# set current time
stats['starttime'] = UTCDateTime()
st = Stream([Trace(data=data, header=stats)])
# write as ASCII file (encoding=0)
st.write("weather.mseed", format='MSEED', encoding=0, reclen=256)
```

```
# Show that it worked, convert NumPy character array back to string
st1 = read("weather.mseed")
print(st1[0].data.tostring())
```

1.19 Beachball Plot

The following lines show how to create a graphical representation of a focal mechanism.

```
from obspy.imaging.beachball import beachball


mt = [0.91, -0.89, -0.02, 1.78, -1.55, 0.47]
beachball(mt, size=200, linewidth=2, facecolor='b')

mt2 = [150, 87, 1]
beachball(mt2, size=200, linewidth=2, facecolor='r')

mt3 = [-2.39, 1.04, 1.35, 0.57, -2.94, -0.94]
beachball(mt3, size=200, linewidth=2, facecolor='g')
```

1.20 Basemap Plots

1.20.1 Basemap Plot with Custom Projection Setup

Simple Basemap plots of e.g. `Inventory` or `Catalog` objects can be performed with builtin methods, see e.g. `Inventory.plot()` or `Catalog.plot()`.

For full control over the projection and map extent, a custom basemap can be set up (e.g. following the examples in the basemap documentation), and then be reused for plots of e.g. `Inventory` or `Catalog` objects:

```
from mpl_toolkits.basemap import Basemap
import numpy as np
import matplotlib.pyplot as plt

from obspy import read_inventory, read_events

# Set up a custom basemap, example is taken from basemap users' manual
fig, ax = plt.subplots()

# setup albers equal area conic basemap
# lat_1 is first standard parallel.
# lat_2 is second standard parallel.
# lon_0, lat_0 is central point.
m = Basemap(width=8000000, height=7000000,
            resolution='c', projection='aea',
            lat_1=40., lat_2=60, lon_0=35, lat_0=50, ax=ax)
m.drawcoastlines()
m.drawcountries()
m.fillcontinents(color='wheat', lake_color='skyblue')
# draw parallels and meridians.
m.drawparallels(np.arange(-80., 81., 20.))
m.drawmeridians(np.arange(-180., 181., 20.))
m.drawmapboundary(fill_color='skyblue')
```

```
ax.set_title("Albers Equal Area Projection")

# we need to attach the basemap object to the figure, so that obspy knows about
# it and reuses it
fig.bmap = m

# now let's plot some data on the custom basemap:
inv = read_inventory()
inv.plot(fig=fig, show=False)
cat = read_events()
cat.plot(fig=fig, show=False, title="", colorbar=False)

plt.show()
```

1.20.2 Basemap Plot of a Local Area with Beachballs

The following example shows how to plot beachballs into a basemap plot together with some stations. The example requires the basemap package (download site) to be installed. The SRTM file used can be downloaded here. The first lines of our SRTM data file (from CGIAR) look like this:

```
ncols         400
nrows         200
xllcorner     12°40'E
yllcorner     47°40'N
xurcorner     13°00'E
yurcorner     47°50'N
cellsize      0.00083333333333333
NODATA_value  -9999
682 681 685 690 691 689 678 670 675 680 681 679 675 671 674 680 679 679 675 671 668 664 659 660 656 6
```

```
import gzip

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap

from obspy.imaging.beachball import beach


# read in topo data (on a regular lat/lon grid)
# (SRTM data from: http://srtm.csi.cgiar.org/)
with gzip.open("srtm_1240-1300E_4740-4750N.asc.gz") as fp:
    srtm = np.loadtxt(fp, skiprows=8)

# origin of data grid as stated in SRTM data file header
# create arrays with all lon/lat values from min to max and
lats = np.linspace(47.8333, 47.6666, srtm.shape[0])
lons = np.linspace(12.6666, 13.0000, srtm.shape[1])

# create Basemap instance with Mercator projection
# we want a slightly smaller region than covered by our SRTM data
m = Basemap(projection='merc', lon_0=13, lat_0=48, resolution="h",
            llcrnrlon=12.75, llcrnrlat=47.69, urcrnrlon=12.95, urcrnrlat=47.81)

# create grids and compute map projection coordinates for lon/lat grid
x, y = m(*np.meshgrid(lons, lats))
```

```
# Make contour plot
cs = m.contour(x, y, srtm, 40, colors="k", lw=0.5, alpha=0.3)
m.drawcountries(color="red", linewidth=1)

# Draw a lon/lat grid (20 lines for an interval of one degree)
m.drawparallels(np.linspace(47, 48, 21), labels=[1, 1, 0, 0], fmt="%.2f",
                dashes=[2, 2])
m.drawmeridians(np.linspace(12, 13, 21), labels=[0, 0, 1, 1], fmt="%.2f",
                dashes=[2, 2])

# Plot station positions and names into the map
# again we have to compute the projection of our lon/lat values
lats = [47.761659, 47.7405, 47.755100, 47.737167]
lons = [12.864466, 12.8671, 12.849660, 12.795714]
names = [" RMOA", " RNON", " RTSH", " RJOB"]
x, y = m(lons, lats)
m.scatter(x, y, 200, color="r", marker="v", edgecolor="k", zorder=3)
for i in range(len(names)):
    plt.text(x[i], y[i], names[i], va="top", family="monospace", weight="bold")

# Add beachballs for two events
lats = [47.751602, 47.75577]
lons = [12.866492, 12.893850]
x, y = m(lons, lats)
# Two focal mechanisms for beachball routine, specified as [strike, dip, rake]
focmecs = [[80, 50, 80], [85, 30, 90]]
ax = plt.gca()
for i in range(len(focmecs)):
    b = beach(focmecs[i], xy=(x[i], y[i]), width=1000, linewidth=1)
    b.set_zorder(10)
    ax.add_collection(b)

plt.show()
```

Some notes:

- The Python package GDAL allows you to directly read a GeoTiff into NumPy `ndarray`

  ```
  >>> geo = gdal.Open("file.geotiff")
  >>> x = geo.ReadAsArray()
  ```

- GeoTiff elevation data is available e.g. from ASTER
- Shading/Illumination can be added. See the basemap example plotmap_shaded.py for more info.

1.20.3 Basemap Plot of the Globe with Beachballs

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap

from obspy.imaging.beachball import beach


m = Basemap(projection='cyl', lon_0=142.36929, lat_0=38.3215,
            resolution='c')

m.drawcoastlines()
```

```
m.fillcontinents()
m.drawparallels(np.arange(-90., 120., 30.))
m.drawmeridians(np.arange(0., 420., 60.))
m.drawmapboundary()

x, y = m(142.36929, 38.3215)
focmecs = [0.136, -0.591, 0.455, -0.396, 0.046, -0.615]

ax = plt.gca()
b = beach(focmecs, xy=(x, y), width=10, linewidth=1, alpha=0.85)
b.set_zorder(10)
ax.add_collection(b)
plt.show()
```

1.21 Interfacing R from Python

The rpy2 package allows to interface R from Python. The following example shows how to convert data (`numpy.ndarray`) to an R matrix and execute the R command `summary` on it.

```
>>> from obspy.core import read
>>> import rpy2.robjects as RO
>>> import rpy2.robjects.numpy2ri
>>> r = RO.r
>>> st = read("test/BW.BGLD..EHE.D.2008.001")
>>> M = RO.RMatrix(st[0].data)
>>> print(r.summary(M))
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-1056.0  -409.0  -393.0  -393.7  -378.0   233.0
```

1.22 Coordinate Conversions

Coordinate conversions can be done conveniently using pyproj. After looking up the EPSG codes of source and target coordinate system, the conversion can be done in just a few lines of code. The following example converts the station coordinates of two German stations to the regionally used Gauß-Krüger system:

```
>>> import pyproj
>>> lat = [49.6919, 48.1629]
>>> lon = [11.2217, 11.2752]
>>> proj_wgs84 = pyproj.Proj(init="epsg:4326")
>>> proj_gk4 = pyproj.Proj(init="epsg:31468")
>>> x, y = pyproj.transform(proj_wgs84, proj_gk4, lon, lat)
>>> print(x)
[4443947.179347951, 4446185.667319892]
>>> print(y)
[5506428.401023342, 5336354.054996853]
```

Another common usage is to convert location information in latitude and longitude to Universal Transverse Mercator coordinate system (UTM). This is especially useful for large dense arrays in a small area. Such conversion can be easily done using utm package. Below is its typical usages:

```
>>> import utm
>>> utm.from_latlon(51.2, 7.5)
(395201.3103811303, 5673135.241182375, 32, 'U')
```

```
>>> utm.to_latlon(340000, 5710000, 32, 'U')
(51.51852098408468, 6.693872395145327)
```

1.23 Hierarchical Clustering

An implementation of hierarchical clustering is provided in the SciPy package. Among other things, it allows to build clusters from similarity matrices and make dendrogram plots. The following example shows how to do this for an already computed similarity matrix. The similarity data are computed from events in an area with induced seismicity (using the cross-correlation routines in `obspy.signal`) and can be fetched from our examples webserver:

First, we import the necessary modules and load the data stored on our webserver:

```
>>> import io, urllib
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from scipy.cluster import hierarchy
>>> from scipy.spatial import distance
>>>
>>> url = "https://examples.obspy.org/dissimilarities.npz"
>>> with io.BytesIO(urllib.urlopen(url).read()) as fh, np.load(fh) as data:
...     dissimilarity = data['dissimilarity']
```

Now, we can start building up the plots. First, we plot the dissimilarity matrix:

```
>>> plt.subplot(121)
>>> plt.imshow(1 - dissimilarity, interpolation="nearest")
```

After that, we use SciPy to build up and plot the dendrogram into the right-hand subplot:

```
>>> dissimilarity = distance.squareform(dissimilarity)
>>> threshold = 0.3
>>> linkage = hierarchy.linkage(dissimilarity, method="single")
>>> clusters = hierarchy.fcluster(linkage, threshold, criterion="distance")
>>>
>>> plt.subplot(122)
>>> hierarchy.dendrogram(linkage, color_threshold=0.3)
>>> plt.xlabel("Event number")
>>> plt.ylabel("Dissimilarity")
>>> plt.show()
```

1.24 Visualizing Probabilistic Power Spectral Densities

The following code example shows how to use the `PPSD` class defined in `obspy.signal`. The routine is useful for interpretation of e.g. noise measurements for site quality control checks. For more information on the topic see [McNamara2004].

```
>>> from obspy import read
>>> from obspy.io.xseed import Parser
>>> from obspy.signal import PPSD
```

Read data and select a trace with the desired station/channel combination:

```
>>> st = read("https://examples.obspy.org/BW.KW1..EHZ.D.2011.037")
>>> tr = st.select(id="BW.KW1..EHZ")[0]
```

Metadata can be provided as an `Inventory` (e.g. from a StationXML file or from a request to a FDSN web service – be sure to use *level='response'*), a `Parser` (mostly legacy, dataless SEED files can be read into *Inventory* objects using `read_inventory()`), a filename of a local RESP file (also legacy, RESP files can be parsed into *Inventory* objects; or a legacy poles and zeros dictionary). Then we initialize a new `PPSD` instance. The ppsd object will then make sure that only appropriate data go into the probabilistic psd statistics.

```
>>> inv = read_inventory("https://examples.obspy.org/BW_KW1.xml")
>>> ppsd = PPSD(tr.stats, metadata=inv)
```

Now we can add data (either trace or stream objects) to the ppsd estimate. This step may take a while. The return value `True` indicates that the data was successfully added to the ppsd estimate.

```
>>> ppsd.add(st)
True
```

We can check what time ranges are represented in the ppsd estimate. `ppsd.times` contains a sorted list of start times of the one hour long slices that the psds are computed from (here only the first two are printed).

```
>>> print(ppsd.times[:2])
[UTCDateTime(2011, 2, 6, 0, 0, 0, 935000), UTCDateTime(2011, 2, 6, 0, 30, 0, 935000)]
>>> print("number of psd segments:", len(ppsd.times))
number of psd segments: 47
```

Adding the same stream again will do nothing (return value `False`), the ppsd object makes sure that no overlapping data segments go into the ppsd estimate.

```
>>> ppsd.add(st)
False
>>> print("number of psd segments:", len(ppsd.times))
number of psd segments: 47
```

Additional information from other files/sources can be added step by step.

```
>>> st = read("https://examples.obspy.org/BW.KW1..EHZ.D.2011.038")
>>> ppsd.add(st)
True
```

The graphical representation of the ppsd can be displayed in a matplotlib window..

```
>>> ppsd.plot()
```

..or saved to an image file:

```
>>> ppsd.plot("/tmp/ppsd.png")
>>> ppsd.plot("/tmp/ppsd.pdf")
```

A (for each frequency bin) cumulative version of the histogram can also be visualized:

```
>>> ppsd.plot(cumulative=True)
```

To use the colormap used by PQLX / [McNamara2004] you can import and use that colormap from `obspy.imaging.cm`:

```
>>> from obspy.imaging.cm import pqlx
>>> ppsd.plot(cmap=pqlx)
```

Below the actual PPSD (for a detailed discussion see [McNamara2004]) is a visualization of the data basis for the PPSD (can also be switched off during plotting). The top row shows data fed into the PPSD, green patches represent available data, red patches represent gaps in streams that were added to the PPSD. The bottom row in blue shows the single psd measurements that go into the histogram. The default processing method fills gaps with zeros, these data segments then show up as single outlying psd lines.

Note: Providing metadata from e.g. a Dataless SEED or StationXML volume is safer than specifying static poles and zeros information (see `PPSD`).

Time series of psd values can also be extracted from the PPSD by accessing the property `psd_values` and plotted using the `plot_temporal()` method (temporal restrictions can be used in the plot, see documentation):

```
>>> ppsd.plot_temporal([0.1, 1, 10])
```

Spectrogram-like plots can be done using the `plot_spectrogram()` method:

```
>>> ppsd.plot_spectrogram()
```

1.25 Array Response Function

The following code block shows how to plot the array transfer function for beam forming as a function of wavenumber using the ObsPy function `obspy.signal.array_analysis.array_transff_wavenumber()`.

```
import numpy as np
import matplotlib.pyplot as plt

from obspy.imaging.cm import obspy_sequential
from obspy.signal.array_analysis import array_transff_wavenumber


# generate array coordinates
coords = np.array([[10., 60., 0.], [200., 50., 0.], [-120., 170., 0.],
                   [-100., -150., 0.], [30., -220., 0.]])

# coordinates in km
coords /= 1000.

# set limits for wavenumber differences to analyze
klim = 40.
kxmin = -klim
kxmax = klim
kymin = -klim
kymax = klim
kstep = klim / 100.

# compute transfer function as a function of wavenumber difference
transff = array_transff_wavenumber(coords, klim, kstep, coordsys='xy')

# plot
plt.pcolor(np.arange(kxmin, kxmax + kstep * 1.1, kstep) - kstep / 2.,
           np.arange(kymin, kymax + kstep * 1.1, kstep) - kstep / 2.,
           transff.T, cmap=obspy_sequential)

plt.colorbar()
plt.clim(vmin=0., vmax=1.)
plt.xlim(kxmin, kxmax)
plt.ylim(kymin, kymax)
plt.show()
```

1.26 Continuous Wavelet Transform

1.26.1 Using ObsPy

The following is a short example for a continuous wavelet transform using ObsPy's internal routine based on [Kristekova2006].

```
import numpy as np
import matplotlib.pyplot as plt

import obspy
from obspy.imaging.cm import obspy_sequential
from obspy.signal.tf_misfit import cwt


st = obspy.read()
tr = st[0]
npts = tr.stats.npts
dt = tr.stats.delta
t = np.linspace(0, dt * npts, npts)
f_min = 1
f_max = 50

scalogram = cwt(tr.data, dt, 8, f_min, f_max)

fig = plt.figure()
ax = fig.add_subplot(111)

x, y = np.meshgrid(
    t,
    np.logspace(np.log10(f_min), np.log10(f_max), scalogram.shape[0]))

ax.pcolormesh(x, y, np.abs(scalogram), cmap=obspy_sequential)
ax.set_xlabel("Time after %s [s]" % tr.stats.starttime)
ax.set_ylabel("Frequency [Hz]")
ax.set_yscale('log')
ax.set_ylim(f_min, f_max)
plt.show()
```

1.26.2 Using MLPY

Small script doing the continuous wavelet transform using the mlpy package (version 3.5.0) for infrasound data recorded at Yasur in 2008. Further details on wavelets can be found at Wikipedia - in the article the omega0 factor is denoted as sigma. *(really sloppy and possibly incorrect: the omega0 factor tells you how often the wavelet fits into the time window, dj defines the spacing in the scale domain)*

```
import numpy as np
import matplotlib.pyplot as plt

import mlpy

import obspy
from obspy.imaging.cm import obspy_sequential


tr = obspy.read("https://examples.obspy.org/a02i.2008.240.mseed")[0]
```

```
omega0 = 8
wavelet_fct = "morlet"
scales = mlpy.wavelet.autoscales(N=len(tr.data), dt=tr.stats.delta, dj=0.05,
                                 wf=wavelet_fct, p=omega0)
spec = mlpy.wavelet.cwt(tr.data, dt=tr.stats.delta, scales=scales,
                        wf=wavelet_fct, p=omega0)
# approximate scales through frequencies
freq = (omega0 + np.sqrt(2.0 + omega0 ** 2)) / (4 * np.pi * scales[1:])

fig = plt.figure()
ax1 = fig.add_axes([0.1, 0.75, 0.7, 0.2])
ax2 = fig.add_axes([0.1, 0.1, 0.7, 0.60], sharex=ax1)
ax3 = fig.add_axes([0.83, 0.1, 0.03, 0.6])

t = np.arange(tr.stats.npts) / tr.stats.sampling_rate
ax1.plot(t, tr.data, 'k')

img = ax2.imshow(np.abs(spec), extent=[t[0], t[-1], freq[-1], freq[0]],
                 aspect='auto', interpolation='nearest', cmap=obspy_sequential)
# Hackish way to overlay a logarithmic scale over a linearly scaled image.
twin_ax = ax2.twinx()
twin_ax.set_yscale('log')
twin_ax.set_xlim(t[0], t[-1])
twin_ax.set_ylim(freq[-1], freq[0])
ax2.tick_params(which='both', labelleft=False, left=False)
twin_ax.tick_params(which='both', labelleft=True, left=True, labelright=False)

fig.colorbar(img, cax=ax3)

plt.show()
```

1.27 Time Frequency Misfit

The `tf_misfit` module offers various Time Frequency Misfit Functions based on [Kristekova2006] and [Kristekova2009].

Here are some examples how to use the included plotting tools:

1.27.1 Plot the Time Frequency Representation

```
import numpy as np

from obspy.signal.tf_misfit import plot_tfr


# general constants
tmax = 6.
dt = 0.01
npts = int(tmax / dt + 1)
t = np.linspace(0., tmax, npts)

fmin = .5
fmax = 10
```

```
# constants for the signal
A1 = 4.
t1 = 2.
f1 = 2.
phi1 = 0.

# generate the signal
H1 = (np.sign(t - t1) + 1) / 2
st1 = A1 * (t - t1) * np.exp(-2 * (t - t1))
st1 *= np.cos(2. * np.pi * f1 * (t - t1) + phi1 * np.pi) * H1

plot_tfr(st1, dt=dt, fmin=fmin, fmax=fmax)
```

1.27.2 Plot the Time Frequency Misfits

Time Frequency Misfits are appropriate for smaller differences of the signals. Continuing the example from above:

```
from scipy.signal import hilbert
from obspy.signal.tf_misfit import plot_tf_misfits

# amplitude and phase error
phase_shift = 0.1
amp_fac = 1.1

# reference signal
st2 = st1.copy()

# generate analytical signal (hilbert transform) and add phase shift
st1p = hilbert(st1)
st1p = np.real(np.abs(st1p) * \
        np.exp((np.angle(st1p) + phase_shift * np.pi) * 1j))

# signal with amplitude error
st1a = st1 * amp_fac

plot_tf_misfits(st1a, st2, dt=dt, fmin=fmin, fmax=fmax, show=False)
plot_tf_misfits(st1p, st2, dt=dt, fmin=fmin, fmax=fmax, show=False)

plt.show()
```

1.27.3 Plot the Time Frequency Goodness-Of-Fit

Time Frequency GOFs are appropriate for large differences of the signals. Continuing the example from above:

```
from obspy.signal.tf_misfit import plot_tf_gofs

# amplitude and phase error
phase_shift = 0.8
amp_fac = 3.

# generate analytical signal (hilbert transform) and add phase shift
st1p = hilbert(st1)
st1p = np.real(np.abs(st1p) * \
        np.exp((np.angle(st1p) + phase_shift * np.pi) * 1j))
```

```
# signal with amplitude error
st1a = st1 * amp_fac

plot_tf_gofs(st1a, st2, dt=dt, fmin=fmin, fmax=fmax, show=False)
plot_tf_gofs(st1p, st2, dt=dt, fmin=fmin, fmax=fmax, show=False)

plt.show()
```

1.27.4 Multi Component Data

For multi component data and global normalization of the misfits, the axes are scaled accordingly. Continuing the example from above:

```
# amplitude error
amp_fac = 1.1

# reference signals
st2_1 = st1.copy()
st2_2 = st1.copy() * 5.
st2 = np.c_[st2_1, st2_2].T

# signals with amplitude error
st1a = st2 * amp_fac

plot_tf_misfits(st1a, st2, dt=dt, fmin=fmin, fmax=fmax)
```

1.27.5 Local normalization

Local normalization allows to resolve frequency and time ranges away from the largest amplitude waves, but tend to produce artifacts in regions where there is no energy at all. In this analytical example e.g. for the high frequencies before the onset of the signal. Manual setting of the limits is thus necessary:

```
# amplitude and phase error
amp_fac = 1.1

ste = 0.001 * A1 * np.exp(- (10 * (t - 2. * t1)) ** 2) \

# reference signal
st2 = st1.copy()

# signal with amplitude error + small additional pulse aftert 4 seconds
st1a = st1 * amp_fac + ste

plot_tf_misfits(st1a, st2, dt=dt, fmin=fmin, fmax=fmax, show=False)
plot_tf_misfits(st1a, st2, dt=dt, fmin=fmin, fmax=fmax, norm='local',
                clim=0.15, show=False)

plt.show()
```

1.28 Visualize Data Availability of Local Waveform Archive

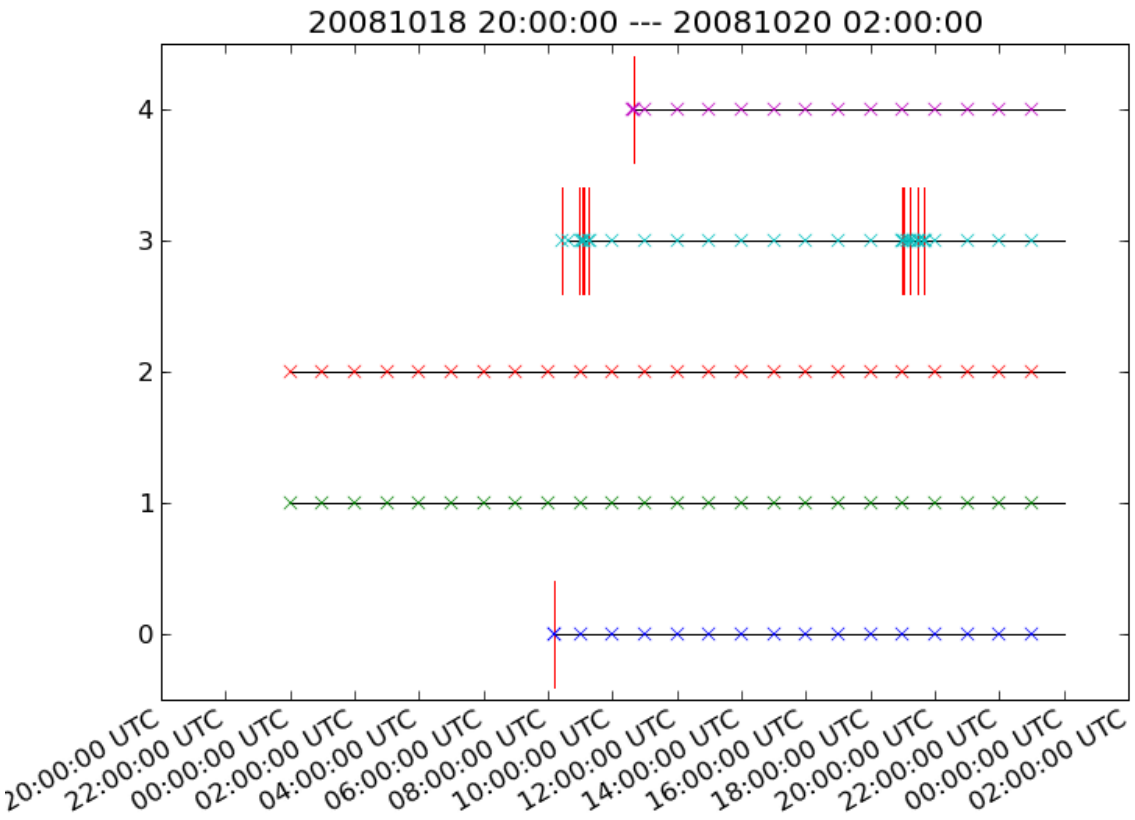
Often, you have a bunch of data and want to know which station is available at what time. For this purpose, ObsPy ships the `obspy-scan` script (automatically available after installation), which detects the file format (MiniSEED,

SAC, SACXY, GSE2, SH-ASC, SH-Q, SEISAN, etc.) from the header of the data files. Gaps are plotted as vertical red lines, start times of available data are plotted as crosses - the data itself are plotted as horizontal lines.

The script can be used to scan through 1000s of files (already used with 30000 files, execution time ca. 45min), month/year ranges are plotted automatically. It opens an interactive plot in which you can zoom in ...

Execute something like following line from the command prompt, use e.g. wildcards to match the files:

```
$ obspy-scan /bay_mobil/mobil/20090622/1081019/*_1.*
```



1.29 Travel Time and Ray Path Plotting

1.29.1 Travel Time Plot

The following lines show how to use the convenience wrapper function `plot_travel_times()` to plot the travel times for a given distance range and selected phases, calculated with the `iasp91` velocity model.

```
from obspy.taup import plot_travel_times
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax = plot_travel_times(source_depth=10, ax=ax, fig=fig,
                       phase_list=['P', 'PP', 'S'], npoints=200)
```

1.29.2 Cartesian Ray Paths

The following lines show how to plot the ray paths for a given distance, and phase(s). The ray paths are calculated with the `iasp91` velocity model, and plotted on a Cartesian map, using the `plot_rays()` method of the class `obspy.taup.tau.Arrivals`.

```
from obspy.taup import TauPyModel

model = TauPyModel(model='iasp91')
arrivals = model.get_ray_paths(500, 140, phase_list=['PP', 'SSS'])
arrivals.plot_rays(plot_type='cartesian', phase_list=['PP', 'SSS'],
                   plot_all=False, legend=True)
```

1.29.3 Spherical Ray Paths

The following lines show how to plot the ray paths for a given distance, and phase(s). The ray paths are calculated with the `iasp91` velocity model, and plotted on a spherical map, using the `plot_rays()` method of the class `obspy.taup.tau.Arrivals`.

```
from obspy.taup import TauPyModel

model = TauPyModel(model='iasp91')
arrivals = model.get_ray_paths(500, 140, phase_list=['Pdiff', 'SS'])
arrivals.plot_rays(plot_type='spherical', phase_list=['Pdiff', 'SS'],
                   legend=True)
```

1.29.4 Ray Paths for Multiple Distances

The following lines plot the ray paths for several epicentral distances, and phases. The rays are calculated with the `iasp91` velocity model, and the plot is made using the convenience wrapper function `plot_ray_paths()`.

```
from obspy.taup.tau import plot_ray_paths
import matplotlib.pyplot as plt

fig, ax = plt.subplots(subplot_kw=dict(polar=True))
ax = plot_ray_paths(source_depth=100, ax=ax, fig=fig, phase_list=['P', 'PKP'],
                    npoints=25)
```

For examples with rays for a single epicentral distance, try the `plot_rays()` method in the previous section. The following is a more advanced example with a custom list of phases and distances:

```
import numpy as np
import matplotlib.pyplot as plt

from obspy.taup import TauPyModel


PHASES = [
    # Phase, distance
    ('P', 26),
    ('PP', 60),
    ('PPP', 94),
    ('PPS', 155),
    ('p', 3),
    ('pPcP', 100),
    ('PKIKP', 170),
```

```
    ('PKJKP', 194),
    ('S', 65),
    ('SP', 85),
    ('SS', 134.5),
    ('SSS', 204),
    ('p', -10),
    ('pP', -37.5),
    ('s', -3),
    ('sP', -49),
    ('ScS', -44),
    ('SKS', -82),
    ('SKKS', -120),
]

model = TauPyModel(model='iasp91')

fig, ax = plt.subplots(subplot_kw=dict(polar=True))

# Plot all pre-determined phases
for phase, distance in PHASES:
    arrivals = model.get_ray_paths(700, distance, phase_list=[phase])
    ax = arrivals.plot_rays(plot_type='spherical',
                            legend=False, label_arrivals=True,
                            plot_all=True,
                            show=False, ax=ax)

# Annotate regions
ax.text(0, 0, 'Solid\ninner\ncore',
        horizontalalignment='center', verticalalignment='center',
        bbox=dict(facecolor='white', edgecolor='none', alpha=0.7))
ocr = (model.model.radius_of_planet -
       (model.model.s_mod.v_mod.iocb_depth +
        model.model.s_mod.v_mod.cmb_depth) / 2)
ax.text(np.deg2rad(180), ocr, 'Fluid outer core',
        horizontalalignment='center',
        bbox=dict(facecolor='white', edgecolor='none', alpha=0.7))
mr = model.model.radius_of_planet - model.model.s_mod.v_mod.cmb_depth / 2
ax.text(np.deg2rad(180), mr, 'Solid mantle',
        horizontalalignment='center',
        bbox=dict(facecolor='white', edgecolor='none', alpha=0.7))

plt.show()
```

1.30 Cross Correlation Pick Correction

This example shows how to align the waveforms of phase onsets of two earthquakes in order to correct the original pick times that can never be set perfectly consistent in routine analysis. A parabola is fit to the concave part of the cross correlation function around its maximum, following the approach by [Deichmann1992].

To adjust the parameters (i.e. the used time window around the pick and the filter settings) and to validate and check the results the options *plot* and *filename* can be used to open plot windows or save the figure to a file.

See the documentation of `xcorr_pick_correction()` for more details.

The example will print the time correction for pick 2 and the respective correlation coefficient and open a plot window for correlations on both the original and preprocessed data:

```
No preprocessing:
  Time correction for pick 2: -0.014459
  Correlation coefficient: 0.92
Bandpass prefiltering:
  Time correction for pick 2: -0.013025
  Correlation coefficient: 0.98
```

```
from __future__ import print_function

import obspy
from obspy.signal.cross_correlation import xcorr_pick_correction


# read example data of two small earthquakes
path = "https://examples.obspy.org/BW.UH1..EHZ.D.2010.147.%s.slist.gz"
st1 = obspy.read(path % ("a", ))
st2 = obspy.read(path % ("b", ))
# select the single traces to use in correlation.
# to avoid artifacts from preprocessing there should be some data left and
# right of the short time window actually used in the correlation.
tr1 = st1.select(component="Z")[0]
tr2 = st2.select(component="Z")[0]
# these are the original pick times set during routine analysis
t1 = obspy.UTCDateTime("2010-05-27T16:24:33.315000Z")
t2 = obspy.UTCDateTime("2010-05-27T16:27:30.585000Z")

# estimate the time correction for pick 2 without any preprocessing and open
# a plot window to visually validate the results
dt, coeff = xcorr_pick_correction(t1, tr1, t2, tr2, 0.05, 0.2, 0.1, plot=True)
print("No preprocessing:")
print("  Time correction for pick 2: %.6f" % dt)
print("  Correlation coefficient: %.2f" % coeff)
# estimate the time correction with bandpass prefiltering
dt, coeff = xcorr_pick_correction(t1, tr1, t2, tr2, 0.05, 0.2, 0.1, plot=True,
                                  filter="bandpass",
                                  filter_options={'freqmin': 1, 'freqmax': 10})
print("Bandpass prefiltering:")
print("  Time correction for pick 2: %.6f" % dt)
print("  Correlation coefficient: %.2f" % coeff)
```

1.31 Handling custom defined tags in QuakeML and the ObsPy Catalog/Event framework

QuakeML allows use of custom elements in addition to the 'usual' information defined by the QuakeML standard. It allows *a)* custom namespace attributes to QuakeML namespace tags and *b)* custom namespace subtags to QuakeML namespace elements. ObsPy can handle both basic custom tags in event type objects (*a*) and custom attributes (*b*) during input/output to/from QuakeML. The following basic example illustrates how to output a valid QuakeML file with custom xml tags/attributes:

```
from obspy import Catalog, UTCDateTime

extra = {'my_tag': {'value': True,
                    'namespace': 'http://some-page.de/xmlns/1.0',
                    'attrib': {'{http://some-page.de/xmlns/1.0}my_attrib1': '123.4',
                               '{http://some-page.de/xmlns/1.0}my_attrib2': '567'}},
```

```
         'my_tag_2': {'value': u'True',
                      'namespace': 'http://some-page.de/xmlns/1.0'},
         'my_tag_3': {'value': 1,
                      'namespace': 'http://some-page.de/xmlns/1.0'},
         'my_tag_4': {'value': UTCDateTime('2013-01-02T13:12:14.600000Z'),
                      'namespace': 'http://test.org/xmlns/0.1'},
         'my_attribute': {'value': 'my_attribute_value',
                          'type': 'attribute',
                          'namespace': 'http://test.org/xmlns/0.1'}}

cat = Catalog()
cat.extra = extra
cat.write('my_catalog.xml', format='QUAKEML',
          nsmap={'my_ns': 'http://test.org/xmlns/0.1'})
```

All custom information to be stored in the customized QuakeML has to be stored in form of a `dict` or `AttribDict` object as the `extra` attribute of the object that should carry the additional custom information (e.g. `Catalog`, `Event`, `Pick`). The keys are used as the name of the xml tag, the content of the xml tag is defined in a simple dictionary: `'value'` defines the content of the tag (the string representation of the object gets stored in the textual xml output). `'namespace'` has to specify a custom namespace for the tag. `'type'` can be used to specify whether the extra information should be stored as a subelement (`'element'`, default) or as an attribute (`'attribute'`). Attributes to custom subelements can be provided in form of a dictionary as `'attrib'`. If desired for better (human-)readability, namespace abbreviations in the output xml can be specified during output as QuakeML by providing a dictionary of namespace abbreviation mappings as *nsmap* parameter to `Catalog.write()`. The xml output of the above example looks like:

```
<?xml version='1.0' encoding='utf-8'?>
<q:quakeml xmlns:q='http://quakeml.org/xmlns/quakeml/1.2'
           xmlns:ns0='http://some-page.de/xmlns/1.0'
           xmlns:my_ns='http://test.org/xmlns/0.1'
           xmlns='http://quakeml.org/xmlns/bed/1.2'>
  <eventParameters publicID='smi:local/b425518c-9445-40c7-8284-d1f299ed2eac'
                   my_ns:my_attribute='my_attribute_value'>
    <ns0:my_tag ns0:my_attrib1='123.4' ns0:my_attrib2='567'>true</ns0:my_tag>
    <my_ns:my_tag_4>2013-01-02T13:12:14.600000Z</my_ns:my_tag_4>
    <ns0:my_tag_2>True</ns0:my_tag_2>
    <ns0:my_tag_3>1</ns0:my_tag_3>
  </eventParameters>
</q:quakeml>
```

When reading the above xml again, using `read_events()`, the custom tags get parsed and attached to the respective Event type objects (in this example to the Catalog object) as `.extra`. Note that all values are read as text strings:

```
from obspy import read_events

cat = read_events('my_catalog.xml')
print(cat.extra)
```

```
AttribDict({u'my_tag': {u'attrib': {'{http://some-page.de/xmlns/1.0}my_attrib2': '567',
                                    '{http://some-page.de/xmlns/1.0}my_attrib1': '123.4'},
                        u'namespace': u'http://some-page.de/xmlns/1.0',
                        u'value': 'true'},
            u'my_tag_4': {u'namespace': u'http://test.org/xmlns/0.1',
                          u'value': '2013-01-02T13:12:14.600000Z'},
            u'my_attribute': {u'type': u'attribute',
                              u'namespace': u'http://test.org/xmlns/0.1',
                              u'value': 'my_attribute_value'},
            u'my_tag_2': {u'namespace': u'http://some-page.de/xmlns/1.0',
```

```
                    u'value': 'True'},
          u'my_tag_3': {u'namespace': u'http://some-page.de/xmlns/1.0',
                        u'value': '1'}})
```

Custom tags can be nested:

```
from obspy import Catalog
from obspy.core import AttribDict

ns = 'http://some-page.de/xmlns/1.0'

my_tag = AttribDict()
my_tag.namespace = ns
my_tag.value = AttribDict()

my_tag.value.my_nested_tag1 = AttribDict()
my_tag.value.my_nested_tag1.namespace = ns
my_tag.value.my_nested_tag1.value = 1.23E+10

my_tag.value.my_nested_tag2 = AttribDict()
my_tag.value.my_nested_tag2.namespace = ns
my_tag.value.my_nested_tag2.value = True

cat = Catalog()
cat.extra = AttribDict()
cat.extra.my_tag = my_tag
cat.write('my_catalog.xml', 'QUAKEML')
```

This will produce an xml output similar to the following:

```
<?xml version='1.0' encoding='utf-8'?>
<q:quakeml xmlns:q='http://quakeml.org/xmlns/quakeml/1.2'
           xmlns:ns0='http://some-page.de/xmlns/1.0'
           xmlns='http://quakeml.org/xmlns/bed/1.2'>
  <eventParameters publicID='smi:local/97d2b338-0701-41a4-9b6b-5903048bc341'>
    <ns0:my_tag>
      <ns0:my_nested_tag1>12300000000.0</ns0:my_nested_tag1>
      <ns0:my_nested_tag2>true</ns0:my_nested_tag2>
    </ns0:my_tag>
  </eventParameters>
</q:quakeml>
```

The output xml can be read again using `read_events()` and the nested tags can be retrieved in the following way:

```
from obspy import read_events

cat = read_events('my_catalog.xml')
print(cat.extra.my_tag.value.my_nested_tag1.value)
print(cat.extra.my_tag.value.my_nested_tag2.value)
```

```
12300000000.0
true
```

The order of extra tags can be controlled by using an `OrderedDict` for the extra attribute (using a plain *dict* or `AttribDict` can result in arbitrary order of tags):

```
from collections import OrderedDict
from obspy.core.event import Catalog, Event
```

```
ns = 'http://some-page.de/xmlns/1.0'

my_tag1 = {'namespace': ns, 'value': 'some value 1'}
my_tag2 = {'namespace': ns, 'value': 'some value 2'}

event = Event()
cat = Catalog(events=[event])
event.extra = OrderedDict()
event.extra['myFirstExtraTag'] = my_tag2
event.extra['mySecondExtraTag'] = my_tag1
cat.write('my_catalog.xml', 'QUAKEML')
```

1.32 Handling custom defined tags in StationXML with the Obspy Inventory

StationXML allows use of custom elements in addition to the 'usual' information defined by the StationXML standard. It allows *a)* custom namespace attributes to StationXML namespace tags and *b)* custom namespace subtags to StationXML namespace elements. ObsPy can handle both basic custom tags in all main elements (Network, Station, Channel, etc.) (*a*) and custom attributes (*b*) during input/output to/from StationXML. The following basic example illustrates how to output a StationXML file that contains additional xml tags/attributes:

```
from obspy import Inventory, UTCDateTime
from obspy.core.inventory import Network
from obspy.core.util import AttribDict

extra = AttribDict({
            'my_tag': {
                 'value': True,
                 'namespace': 'http://some-page.de/xmlns/1.0',
                 'attrib': {
                   '{http://some-page.de/xmlns/1.0}my_attrib1': '123.4',
                   '{http://some-page.de/xmlns/1.0}my_attrib2': '567'
                 }
             },
            'my_tag_2': {
                 'value': u'True',
                 'namespace': 'http://some-page.de/xmlns/1.0'
             },
            'my_tag_3': {
                 'value': 1,
                 'namespace': 'http://some-page.de/xmlns/1.0'
             },
            'my_tag_4': {
                 'value': UTCDateTime('2013-01-02T13:12:14.600000Z'),
                 'namespace': 'http://test.org/xmlns/0.1'
             },
            'my_attribute': {
                 'value': 'my_attribute_value',
                 'type': 'attribute',
                 'namespace': 'http://test.org/xmlns/0.1'
             }
        })

inv = Inventory([Network('XX')], 'XX')
inv[0].extra = extra
```

```
inv.write('my_inventory.xml', format='STATIONXML',
          nsmap={'my_ns': 'http://test.org/xmlns/0.1',
                 'somepage_ns': 'http://some-page.de/xmlns/1.0'})
```

All custom information to be stored in the customized StationXML has to be stored in form of a `dict` or `AttribDict` object as the `extra` attribute of the object that should carry the additional custom information (e.g. `Network`, `Station`, `Channel`). The keys are used as the name of the xml tag, the content of the xml tag is defined in a simple dictionary: `'value'` defines the content of the tag (the string representation of the object gets stored in the textual xml output). `'namespace'` has to specify a custom namespace for the tag. `'type'` can be used to specify whether the extra information should be stored as a subelement (`'element'`, default) or as an attribute (`'attribute'`). Attributes to custom subelements can be provided in form of a dictionary as `'attrib'`. If desired for better (human-)readability, namespace abbreviations in the output xml can be specified during output as StationXML by providing a dictionary of namespace abbreviation mappings as *nsmap* parameter to `Inventory.write()`. The xml output of the above example looks like:

```
<?xml version='1.0' encoding='UTF-8'?>
<FDSNStationXML xmlns:my_ns="http://test.org/xmlns/0.1" xmlns:somepage_ns="http://some-page.de/xmlns/
  <Source>XX</Source>
  <Module>ObsPy 1.0.2</Module>
  <ModuleURI>https://www.obspy.org</ModuleURI>
  <Created>2016-10-17T18:32:28.696287+00:00</Created>
  <Network code="XX">
    <somepage_ns:my_tag somepage_ns:my_attrib1="123.4" somepage_ns:my_attrib2="567">True</somepage_ns
    <my_ns:my_tag_4>2013-01-02T13:12:14.600000Z</my_ns:my_tag_4>
    <my_ns:my_attribute>my_attribute_value</my_ns:my_attribute>
    <somepage_ns:my_tag_2>True</somepage_ns:my_tag_2>
    <somepage_ns:my_tag_3>1</somepage_ns:my_tag_3>
  </Network>
</FDSNStationXML>
```

When reading the above xml again, using `read_inventory()`, the custom tags get parsed and attached to the respective Network type objects (in this example to the Inventory object) as `.extra`. Note that all values are read as text strings:

```
from obspy import read_inventory

inv = read_inventory('my_inventory.xml')
print(inv[0].extra)

AttribDict({
    u'my_tag': AttribDict({
        'attrib': {
          '{http://some-page.de/xmlns/1.0}my_attrib2': '567',
          '{http://some-page.de/xmlns/1.0}my_attrib1': '123.4'
        },
        'namespace': 'http://some-page.de/xmlns/1.0',
        'value': 'True'
    }),
    u'my_tag_4': AttribDict({
        'namespace': 'http://test.org/xmlns/0.1',
        'value': '2013-01-02T13:12:14.600000Z'
    }),
    u'my_attribute': AttribDict({
        'namespace': 'http://test.org/xmlns/0.1',
        'value': 'my_attribute_value'
    }),
    u'my_tag_2': AttribDict({
        'namespace': 'http://some-page.de/xmlns/1.0',
```

```
        'value': 'True'
    }),
    u'my_tag_3': AttribDict({
        'namespace': 'http://some-page.de/xmlns/1.0',
        'value': '1'
    })
})
```

Custom tags can be nested:

```
from obspy import Inventory
from obspy.core.inventory import Network
from obspy.core.util import AttribDict

ns = 'http://some-page.de/xmlns/1.0'

my_tag = AttribDict()
my_tag.namespace = ns
my_tag.value = AttribDict()

my_tag.value.my_nested_tag1 = AttribDict()
my_tag.value.my_nested_tag1.namespace = ns
my_tag.value.my_nested_tag1.value = 1.23E+10

my_tag.value.my_nested_tag2 = AttribDict()
my_tag.value.my_nested_tag2.namespace = ns
my_tag.value.my_nested_tag2.value = True

inv = Inventory([Network('XX')], 'XX')
inv[0].extra = AttribDict()
inv[0].extra.my_tag = my_tag
inv.write('my_inventory.xml', format='STATIONXML',
          nsmap={'somepage_ns': 'http://some-page.de/xmlns/1.0'})
```

This will produce an xml output similar to the following:

```
<?xml version='1.0' encoding='UTF-8'?>
<FDSNStationXML xmlns:somepage_ns="http://some-page.de/xmlns/1.0" xmlns="http://www.fdsn.org/xml/stat
  <Source>XX</Source>
  <Module>ObsPy 1.0.2</Module>
  <ModuleURI>https://www.obspy.org</ModuleURI>
  <Created>2016-10-17T18:45:14.302265+00:00</Created>
  <Network code="XX">
    <somepage_ns:my_tag>
      <somepage_ns:my_nested_tag1>12300000000.0</somepage_ns:my_nested_tag1>
      <somepage_ns:my_nested_tag2>True</somepage_ns:my_nested_tag2>
    </somepage_ns:my_tag>
  </Network>
</FDSNStationXML>
```

The output xml can be read again using `read_inventory()` and the nested tags can be retrieved in the following way:

```
from obspy import read_inventory

inv = read_inventory('my_inventory.xml')
print(inv[0].extra.my_tag.value.my_nested_tag1.value)
print(inv[0].extra.my_tag.value.my_nested_tag2.value)
```

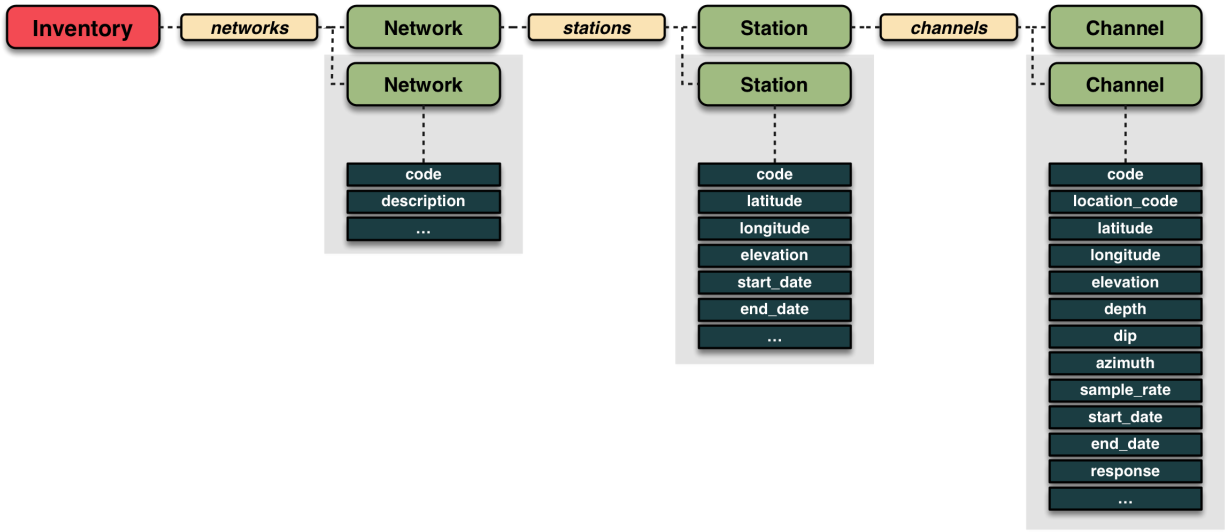
```
12300000000.0
True
```

1.33 Creating a StationXML file from Scratch

Creating a custom StationXML file is a task that sometimes comes up in seismology. This section demonstrates how to it with ObsPy. Please note that this is not necessarily easier or more obvious then directly editing an XML file but it does provider tighter integration with the rest of ObsPy and can guarantee a valid result at the end.

Note that this assumes a certain familiarity with the FDSN StationXML standard. We'll create a fairly simplistic StationXML file and many arguments are optional. ObsPy will validate the resulting StationXML file against its schema upon writing so the final file is assured to be valid against the StationXML schema.

The following illustration shows the basic structure of ObsPy's internal representation.



Each big box will be an object and all objects will have to be hierarchically linked to form a single `Inventory` object. An inventory can contain any number of `Network` objects, which in turn can contain any number of `Station` objects, which once again in turn can contain any number of `Channel` objects. For each channel, the instrument response can be stored as the `response` attribute.

Instrument Response can be looked up and attached to the channels from the IRIS DMC Library of Nominal Responses for Seismic Instruments (NRL) using ObsPy's `NRL client`.

```
import obspy
from obspy.core.inventory import Inventory, Network, Station, Channel, Site
from obspy.clients.nrl import NRL


# We'll first create all the various objects. These strongly follow the
# hierarchy of StationXML files.
inv = Inventory(
    # We'll add networks later.
    networks=[],
    # The source should be the id whoever create the file.
    source="ObsPy-Tutorial")
```

```
net = Network(
    # This is the network code according to the SEED standard.
    code="XX",
    # A list of stations. We'll add one later.
    stations=[],
    description="A test stations.",
    # Start-and end dates are optional.
    start_date=obspy.UTCDateTime(2016, 1, 2))

sta = Station(
    # This is the station code according to the SEED standard.
    code="ABC",
    latitude=1.0,
    longitude=2.0,
    elevation=345.0,
    creation_date=obspy.UTCDateTime(2016, 1, 2),
    site=Site(name="First station"))

cha = Channel(
    # This is the channel code according to the SEED standard.
    code="HHZ",
    # This is the location code according to the SEED standard.
    location_code="",
    # Note that these coordinates can differ from the station coordinates.
    latitude=1.0,
    longitude=2.0,
    elevation=345.0,
    depth=10.0,
    azimuth=0.0,
    dip=-90.0,
    sample_rate=200)

# By default this accesses the NRL online. Offline copies of the NRL can
# also be used instead
nrl = NRL()
# The contents of the NRL can be explored interactively in a Python prompt,
# see API documentation of NRL submodule:
# http://docs.obspy.org/packages/obspy.clients.nrl.html
# Here we assume that the end point of data logger and sensor are already
# known:
response = nrl.get_response( # doctest: +SKIP
    sensor_keys=['Streckeisen', 'STS-1', '360 seconds'],
    datalogger_keys=['REF TEK', 'RT 130 & 130-SMA', '1', '200'])


# Now tie it all together.
cha.response = response
sta.channels.append(cha)
net.stations.append(sta)
inv.networks.append(net)

# And finally write it to a StationXML file. We also force a validation against
# the StationXML schema to ensure it produces a valid StationXML file.
#
# Note that it is also possible to serialize to any of the other inventory
# output formats ObsPy supports.
inv.write("station.xml", format="stationxml", validate=True)
```

1.34 Connecting to a SeedLink Server

The `obspy.clients.seedlink` module provides a Python implementation of the SeedLink client protocol. The `obspy.clients.seedlink.easyseedlink` submodule contains a high-level interface to the SeedLink implementation that facilitates the creation of a SeedLink client.

1.34.1 The create_client function

The easiest way to connect to a SeedLink server is using the `create_client()` function to create a new instance of the `EasySeedLinkClient` class. It accepts as an argument a function that handles new data received from the SeedLink server, for example:

```
def handle_data(trace):
    print('Received the following trace:')
    print(trace)
    print()
```

This function can then be passed to `create_client()` together with a SeedLink server URL to create a client instance:

```
client = create_client('geofon.gfz-potsdam.de', on_data=handle_data)
```

The client immediately connects to the server when it is created.

Sending INFO requests to the server

The client instance can be used to send SeedLink `INFO` requests to the server:

```
# Send the INFO:ID request
client.get_info('ID')

# Returns:
# <?xml version="1.0"?>\n<seedlink software="SeedLink v3.2 (2014.071)" organization="GEOFON" started=
```

The responses to `INFO` requests are in XML format. The client provides a shortcut to retrieve and parse the server's capabilities (via an `INFO:CAPABILITIES` request):

```
>>> client.capabilities
['dialup', 'multistation', 'window-extraction', 'info:id', 'info:capabilities', 'info:stations', 'in
```

The capabilities are fetched and parsed when the attribute is first accessed and are cached after that.

Streaming data from the server

In order to start receiving waveform data, a *stream* needs to be selected. This is done by calling the `select_stream()` method:

```
client.select_stream('BW', 'MANZ', 'EHZ')
```

Multiple streams can be selected. SeedLink wildcards are also supported:

```
client.select_stream('BW', 'ROTZ', 'EH?')
```

After having selected the streams, the client is ready to enter streaming mode:

```
client.run()
```

This starts streaming data from the server. Upon every complete trace that is received from the server, the function defined above is called with the trace object:

```
Received new data:
BW.MANZ..EHZ | 2014-09-04T19:47:25.625000Z - 2014-09-04T19:47:26.770000Z | 200.0 Hz, 230 samples

Received new data:
BW.ROTZ..EHZ | 2014-09-04T19:47:22.685000Z - 2014-09-04T19:47:24.740000Z | 200.0 Hz, 412 samples

Received new data:
BW.ROTZ..EHZ | 2014-09-04T19:47:24.745000Z - 2014-09-04T19:47:26.800000Z | 200.0 Hz, 412 samples

Received new data:
BW.ROTZ..EHN | 2014-09-04T19:47:20.870000Z - 2014-09-04T19:47:22.925000Z | 200.0 Hz, 412 samples

Received new data:
BW.ROTZ..EHN | 2014-09-04T19:47:22.930000Z - 2014-09-04T19:47:24.985000Z | 200.0 Hz, 412 samples
```

The `create_client()` function also accepts functions to be called when the connection terminates or when a SeedLink error is received. See the `documentation` for details.

1.34.2 Advanced usage: subclassing the client

For advanced use cases, subclassing the `EasySeedLinkClient` class allows for finer control over the instance. Implementing the same client as above:

```
class DemoClient(EasySeedLinkClient):
    """
    A custom SeedLink client
    """
    def on_data(self, trace):
        """
        Override the on_data callback
        """
        print('Received trace:')
        print(trace)
        print()
```

The `documentation` has more details about the client.

CHAPTER

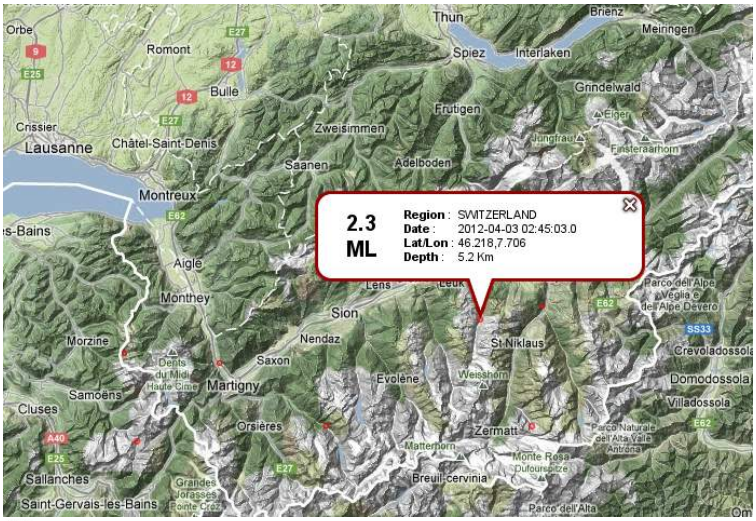
TWO

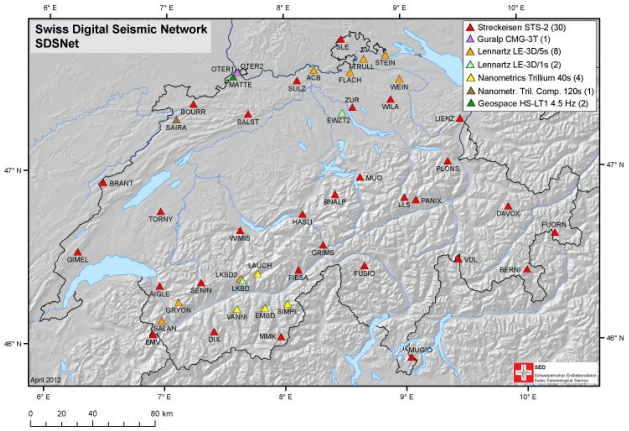
ADVANCED EXERCISE

In the advanced exercise we show how ObsPy can be used to develop an automated processing workflow. We start out with very simple tasks and then automate the routine step by step. For all exercises solutions are provided.

2.1 Advanced Exercise

This practical intends to demonstrate how ObsPy can be used to develop workflows for data processing and analysis that have a short, easy to read and extensible source code. The overall task is to automatically estimate local magnitudes of earthquakes using data of the SED network. We will start with simple programs with manually specified, hard-coded values and build on them step by step to make the program more flexible and dynamic. Some details in the magnitude estimation should be done a little bit different technically but we rather want to focus on the general workflow here.





2.1.1 1. Request Earthquake Information from EMSC/NERIES/NERA

Fetch a list of events from EMSC for the region of Valais/SW-Switzerland on 3rd April of 2012. Use the Client provided in `obspy.clients.fdsn`. Note down the catalog origin times, epicenters and magnitudes.

2.1.2 2. Estimate Local Magnitude

1. Use the file LKBD_WA_CUT.MSEED to read MiniSEED waveform data of the larger earthquake. These data have already been simulated to (demeaned) displacement on a Wood-Anderson seismometer (in meter) and trimmed to the right time span. Compute the absolute maximum for both North and East component and use the larger value as the zero-to-peak amplitude estimate. Estimate the local magnitude M_{lh} used at the Swiss Seismological Service (SED) using a epicentral distance of $d_{\mathrm{epi}} = 20$ (km), $a = 0.018$ and $b = 2.17$ with the following formula (mathematical functions are available in Python's `math` module):

$$M_{\mathrm{lh}} = \log_{10}\left(amp \cdot 1000 \frac{\mathrm{mm}}{\mathrm{m}}\right) + a \cdot d_{\mathrm{epi}} + b$$

2. Calculate the epicentral distance from the station coordinates (46.387°N, 7.627°E) and catalog epicenter fetched above (46.218°N, 7.706°E). Some useful routines for such tasks are included in `obspy.core.util.geodetics`.

2.1.3 3. Seismometer Correction/Simulation

1. Modify the existing code and use the file LKBD.MSEED to read the original MiniSEED waveform data in counts. Set up two dictionaries containing the response information of both the original instrument (a LE3D-5s) and the Wood-Anderson seismometer in poles-and-zeros formulation. Please note that for historic reasons the naming of keys differs from the usual naming. Each PAZ dictionary needs to contain *sensitivity* (overall sensitivity of seismometer/digitizer combination), *gain* (A0 / normalization factor), *poles* and *zeros*. Check that the value of *water_level* is not too high, to avoid overamplified low frequency noise at short-period stations. After the instrument simulation, trim the waveform to a shorter time window around the origin time (*2012-04-03T02:45:03*) and calculate M_{lh} like before. Use the following values for the PAZ dictionaries:

–	LE3D-5s	Wood-Anderson
poles	-0.885+0.887j -0.885-0.887j -0.427+0j	-6.2832-4.7124j -6.2832+4.7124j
zeros	0j, 0j, 0j	0j
gain	1.009	1
sensitivity	167364000.0	2800

2. Instead of the hard-coded values, read the response information from a locally stored *dataless SEED* LKBD.dataless. Use the Parser of module `obspy.xseed` to extract the poles-and-zeros information of the used channel.

3. We can also request the response information from WebDC using the ArcLink protocol. Use the Client provided in `obspy.arclink` module (specify e.g. *user="sed-workshop@obspy.org"*).

2.1.4 4. Fetch Waveform Data from WebDC

1. Modify the existing code and fetch waveform data around the origin time given above for station *LKBD* (network *CH*) via ArcLink from WebDC using *obspy.arclink*. Use a wildcarded *channel="EH*"* to fetch all three components. Use keyword argument *metadata=True* to fetch response information and station coordinates along with the waveform. The PAZ and coordinate information will get attached to the `Stats` object of all traces in the returned Stream object during the waveform request automatically. During instrument simulation use keyword argument *paz_remove='self'* to use every trace's attached PAZ information fetched from WebDC. Calculate M_{lh} like before.

2. Use a list of station names (e.g. *LKBD*, *SIMPL*, *DIX*) and perform the magnitude estimation in a loop for each station. Use a wildcarded *channel="[EH]H*"* to fetch the respective streams for both short-period and broadband stations. Compile a list of all station magnitudes and compute the network magnitude as its median (available in `numpy` module).

3. Extend the network magnitude estimate by using all available stations in network *CH*. Get a list of stations using the ArcLink client and loop over this list. Use a wildcarded *channel="[EH]H[ZNE]"*, check if there are three traces in the returned stream and skip to next station otherwise (some stations have inconsistent component codes). Put a *try/except* around the waveform request and skip to the next station and avoid interruption of the routine in case no data can be retrieved and an Exception gets raised. Also add an *if/else* and use $a = 0.0038$ and $b = 3.02$ in station magnitude calculation for epicentral distances of more than 60 kilometers.

2.1.5 5. Construct Fully Automatic Event Detection, Magnitude Estimation

In this additional advanced exercise we can enhance the routine to be independent of a-priori known origin times by using a coincidence network trigger for event detection.

- fetch a few hours of Z component data for 6 stations in Valais / SW-Switzerland
- run a coincidence trigger like shown in the *Trigger Tutorial*
- loop over detected network triggers, store the coordinates of the closest station as the epicenter

- loop over triggers, use the trigger time to select the time window and use the network magnitude estimation code like before

2.1.6 Solutions

Advanced Exercise Solution 1

```
from __future__ import print_function

import obspy
import obspy.clients.fdsn


client = obspy.clients.fdsn.Client("EMSC")

events = client.get_events(minlatitude=46.1, maxlatitude=46.3,
                           minlongitude=7.6, maxlongitude=7.8,
                           starttime=obspy.UTCDateTime("2012-04-03"),
                           endtime=obspy.UTCDateTime("2012-04-04"))

print("found %s event(s):" % len(events))
for event in events:
    print(event)
```

Advanced Exercise Solution 2a

```
from __future__ import print_function

from math import log10

from obspy import read


st = read("../data/LKBD_WA_CUT.MSEED")

tr_n = st.select(component="N")[0]
ampl_n = max(abs(tr_n.data))

tr_e = st.select(component="E")[0]
ampl_e = max(abs(tr_e.data))

ampl = max(ampl_n, ampl_e)

epi_dist = 20

a = 0.018
b = 2.17
ml = log10(ampl * 1000) + a * epi_dist + b
print(ml)
```

Advanced Exercise Solution 2b

```
from __future__ import print_function

from math import log10

from obspy import read
from obspy.geodetics import gps2dist_azimuth


st = read("../data/LKBD_WA_CUT.MSEED")

tr_n = st.select(component="N")[0]
ampl_n = max(abs(tr_n.data))
tr_e = st.select(component="E")[0]
ampl_e = max(abs(tr_e.data))
ampl = max(ampl_n, ampl_e)

sta_lat = 46.38703
sta_lon = 7.62714
event_lat = 46.218
event_lon = 7.706

epi_dist, az, baz = gps2dist_azimuth(event_lat, event_lon, sta_lat, sta_lon)
epi_dist = epi_dist / 1000

a = 0.018
b = 2.17
ml = log10(ampl * 1000) + a * epi_dist + b
print(ml)
```

Advanced Exercise Solution 3a

```
from __future__ import print_function

from math import log10

from obspy import UTCDateTime, read
from obspy.geodetics import gps2dist_azimuth


st = read("../data/LKBD.MSEED")

paz_le3d5s = {'gain': 1.009,
              'poles': [-0.885 + 0.887j,
                        -0.885 - 0.887j,
                        -0.427 + 0j],
              'sensitivity': 167364000.0,
              'zeros': [0j, 0j, 0j]}
paz_wa = {'sensitivity': 2800, 'zeros': [0j], 'gain': 1,
          'poles': [-6.2832 - 4.7124j, -6.2832 + 4.7124j]}

st.simulate(paz_remove=paz_le3d5s, paz_simulate=paz_wa, water_level=10)

t = UTCDateTime("2012-04-03T02:45:03")
st.trim(t, t + 50)

tr_n = st.select(component="N")[0]
ampl_n = max(abs(tr_n.data))
```

```
tr_e = st.select(component="E")[0]
ampl_e = max(abs(tr_e.data))
ampl = max(ampl_n, ampl_e)

sta_lat = 46.38703
sta_lon = 7.62714
event_lat = 46.218
event_lon = 7.706

epi_dist, az, baz = gps2dist_azimuth(event_lat, event_lon, sta_lat, sta_lon)
epi_dist = epi_dist / 1000

a = 0.018
b = 2.17
ml = log10(ampl * 1000) + a * epi_dist + b
print(ml)
```

Advanced Exercise Solution 3b

```
from __future__ import print_function

from math import log10

from obspy import UTCDateTime, read
from obspy.geodetics import gps2dist_azimuth
from obspy.io.xseed import Parser


st = read("../data/LKBD.MSEED")

paz_wa = {'sensitivity': 2800, 'zeros': [0j], 'gain': 1,
          'poles': [-6.2832 - 4.7124j, -6.2832 + 4.7124j]}

parser = Parser("../data/LKBD.dataless")
paz_le3d5s = parser.get_paz("CH.LKBD..EHZ")

st.simulate(paz_remove=paz_le3d5s, paz_simulate=paz_wa, water_level=10)

t = UTCDateTime("2012-04-03T02:45:03")
st.trim(t, t + 50)

tr_n = st.select(component="N")[0]
ampl_n = max(abs(tr_n.data))
tr_e = st.select(component="E")[0]
ampl_e = max(abs(tr_e.data))
ampl = max(ampl_n, ampl_e)

sta_lat = 46.38703
sta_lon = 7.62714
event_lat = 46.218
event_lon = 7.706

epi_dist, az, baz = gps2dist_azimuth(event_lat, event_lon, sta_lat, sta_lon)
epi_dist = epi_dist / 1000

a = 0.018
b = 2.17
```

```
ml = log10(ampl * 1000) + a * epi_dist + b
print(ml)
```

Advanced Exercise Solution 3c

```
from __future__ import print_function

from math import log10

from obspy.clients.arclink import Client
from obspy import UTCDateTime, read
from obspy.geodetics import gps2dist_azimuth


st = read("../data/LKBD.MSEED")

paz_wa = {'sensitivity': 2800, 'zeros': [0j], 'gain': 1,
          'poles': [-6.2832 - 4.7124j, -6.2832 + 4.7124j]}

client = Client(user="sed-workshop@obspy.org")
t = st[0].stats.starttime
paz_le3d5s = client.get_paz("CH", "LKBD", "", "EHZ", t)

st.simulate(paz_remove=paz_le3d5s, paz_simulate=paz_wa, water_level=10)

t = UTCDateTime("2012-04-03T02:45:03")
st.trim(t, t + 50)

tr_n = st.select(component="N")[0]
ampl_n = max(abs(tr_n.data))
tr_e = st.select(component="E")[0]
ampl_e = max(abs(tr_e.data))
ampl = max(ampl_n, ampl_e)

sta_lat = 46.38703
sta_lon = 7.62714
event_lat = 46.218
event_lon = 7.706

epi_dist, az, baz = gps2dist_azimuth(event_lat, event_lon, sta_lat, sta_lon)
epi_dist = epi_dist / 1000

a = 0.018
b = 2.17
ml = log10(ampl * 1000) + a * epi_dist + b
print(ml)
```

Advanced Exercise Solution 4a

```
from __future__ import print_function

from math import log10

from obspy.clients.arclink import Client
from obspy.core import UTCDateTime
```

```
from obspy.geodetics import gps2dist_azimuth


paz_wa = {'sensitivity': 2800, 'zeros': [0j], 'gain': 1,
          'poles': [-6.2832 - 4.7124j, -6.2832 + 4.7124j]}

client = Client(user="sed-workshop@obspy.org")
t = UTCDateTime("2012-04-03T02:45:03")
st = client.get_waveforms("CH", "LKBD", "", "EH*", t - 300, t + 300,
                          metadata=True)

st.simulate(paz_remove="self", paz_simulate=paz_wa, water_level=10)
st.trim(t, t + 50)

tr_n = st.select(component="N")[0]
ampl_n = max(abs(tr_n.data))
tr_e = st.select(component="E")[0]
ampl_e = max(abs(tr_e.data))
ampl = max(ampl_n, ampl_e)

sta_lat = 46.38703
sta_lon = 7.62714
event_lat = 46.218
event_lon = 7.706

epi_dist, az, baz = gps2dist_azimuth(event_lat, event_lon, sta_lat, sta_lon)
epi_dist = epi_dist / 1000

a = 0.018
b = 2.17
ml = log10(ampl * 1000) + a * epi_dist + b
print(ml)
```

Advanced Exercise Solution 4b

```
from __future__ import print_function

from math import log10

import numpy as np

from obspy.clients.arclink import Client
from obspy import UTCDateTime
from obspy.geodetics import gps2dist_azimuth


paz_wa = {'sensitivity': 2800, 'zeros': [0j], 'gain': 1,
          'poles': [-6.2832 - 4.7124j, -6.2832 + 4.7124j]}

client = Client(user="sed-workshop@obspy.org")
t = UTCDateTime("2012-04-03T02:45:03")

stations = ["LKBD", "SIMPL", "DIX"]
mags = []

for station in stations:
    st = client.get_waveforms("CH", station, "", "[EH]H*", t - 300, t + 300,
```

```
                                metadata=True)

    st.simulate(paz_remove="self", paz_simulate=paz_wa, water_level=10)
    st.trim(t, t + 50)

    tr_n = st.select(component="N")[0]
    ampl_n = max(abs(tr_n.data))
    tr_e = st.select(component="E")[0]
    ampl_e = max(abs(tr_e.data))
    ampl = max(ampl_n, ampl_e)

    sta_lat = st[0].stats.coordinates.latitude
    sta_lon = st[0].stats.coordinates.longitude
    event_lat = 46.218
    event_lon = 7.706

    epi_dist, az, baz = gps2dist_azimuth(event_lat, event_lon, sta_lat,
                                         sta_lon)
    epi_dist = epi_dist / 1000

    a = 0.018
    b = 2.17
    ml = log10(ampl * 1000) + a * epi_dist + b
    print(station, ml)
    mags.append(ml)

net_mag = np.median(mags)
print("Network magnitude:", net_mag)
```

Advanced Exercise Solution 4c

```
from __future__ import print_function

from math import log10

import numpy as np

from obspy.clients.arclink import Client
from obspy import UTCDateTime
from obspy.geodetics import gps2dist_azimuth


paz_wa = {'sensitivity': 2800, 'zeros': [0j], 'gain': 1,
          'poles': [-6.2832 - 4.7124j, -6.2832 + 4.7124j]}

client = Client(user="sed-workshop@obspy.org")
t = UTCDateTime("2012-04-03T02:45:03")

stations = client.get_stations(t, t + 300, "CH")
mags = []

for station in stations:
    station = station['code']
    try:
        st = client.get_waveforms("CH", station, "", "[EH]H[ZNE]", t - 300,
                                  t + 300, metadata=True)
        assert(len(st) == 3)
```

```
    except Exception:
        print(station, "---")
        continue

    st.simulate(paz_remove="self", paz_simulate=paz_wa, water_level=10)
    st.trim(t, t + 50)

    tr_n = st.select(component="N")[0]
    ampl_n = max(abs(tr_n.data))
    tr_e = st.select(component="E")[0]
    ampl_e = max(abs(tr_e.data))
    ampl = max(ampl_n, ampl_e)

    sta_lat = st[0].stats.coordinates.latitude
    sta_lon = st[0].stats.coordinates.longitude
    event_lat = 46.218
    event_lon = 7.706

    epi_dist, az, baz = gps2dist_azimuth(event_lat, event_lon, sta_lat,
                                         sta_lon)
    epi_dist = epi_dist / 1000

    if epi_dist < 60:
        a = 0.018
        b = 2.17
    else:
        a = 0.0038
        b = 3.02
    ml = log10(ampl * 1000) + a * epi_dist + b
    print(station, ml)
    mags.append(ml)

net_mag = np.median(mags)
print("Network magnitude:", net_mag)
```

Advanced Exercise Solution 5

```
from __future__ import print_function

from math import log10

import numpy as np

from obspy.clients.arclink import Client
from obspy import Stream, UTCDateTime
from obspy.geodetics import gps2dist_azimuth
from obspy.signal.trigger import coincidence_trigger


client = Client(user="sed-workshop@obspy.org")

t = UTCDateTime("2012-04-03T01:00:00")
t2 = t + 4 * 3600

stations = ["AIGLE", "SENIN", "DIX", "LAUCH", "MMK", "SIMPL"]
st = Stream()
```

```
for station in stations:
    try:
        tmp = client.get_waveforms("CH", station, "", "[EH]HZ", t, t2,
                                   metadata=True)
    except Exception:
        print(station, "---")
        continue
    st += tmp

st.taper()
st.filter("bandpass", freqmin=1, freqmax=20)
triglist = coincidence_trigger("recstalta", 10, 2, st, 4, sta=0.5, lta=10)
print(len(triglist), "events triggered.")

for trig in triglist:
    closest_sta = trig['stations'][0]
    tr = st.select(station=closest_sta)[0]
    trig['latitude'] = tr.stats.coordinates.latitude
    trig['longitude'] = tr.stats.coordinates.longitude

paz_wa = {'sensitivity': 2800, 'zeros': [0j], 'gain': 1,
          'poles': [-6.2832 - 4.7124j, -6.2832 + 4.7124j]}

for trig in triglist:
    t = trig['time']
    print("#" * 80)
    print("Trigger time:", t)
    mags = []

    stations = client.get_stations(t, t + 300, "CH")

    for station in stations:
        station = station['code']
        try:
            st = client.get_waveforms("CH", station, "", "[EH]H[ZNE]", t - 300,
                                      t + 300, metadata=True)
            assert(len(st) == 3)
        except Exception:
            print(station, "---")
            continue

        st.simulate(paz_remove="self", paz_simulate=paz_wa, water_level=10)
        st.trim(t, t + 50)

        tr_n = st.select(component="N")[0]
        ampl_n = max(abs(tr_n.data))
        tr_e = st.select(component="E")[0]
        ampl_e = max(abs(tr_e.data))
        ampl = max(ampl_n, ampl_e)

        sta_lat = st[0].stats.coordinates.latitude
        sta_lon = st[0].stats.coordinates.longitude
        event_lat = trig['latitude']
        event_lon = trig['longitude']

        epi_dist, az, baz = gps2dist_azimuth(event_lat, event_lon, sta_lat,
                                             sta_lon)
        epi_dist = epi_dist / 1000
```

```
        if epi_dist < 60:
            a = 0.018
            b = 2.17
        else:
            a = 0.0038
            b = 3.02
        ml = log10(ampl * 1000) + a * epi_dist + b
        print(station, ml)
        mags.append(ml)

    net_mag = np.median(mags)
    print("Network magnitude:", net_mag)
```

A

C

D

P

R

Z