EGU2010 SM1.3 Seismic Centers Data Acquisition session

ObsPy – What can it do for data centers and observatories?

Tobias Megies[1,*], Moritz Beyreuther[1], Robert Barsch[1], Lion Krischer[1], Joachim Wassermann[1]

[1] *Ludwig-Maximilians-University, Department of Earth and Environmental Sciences, Geophysical Observatory, Munich, Germany*

Article history
Received May 5, 2010; accepted November 25, 2010.
Subject classification:
Waves and wave analysis, Instruments and techniques, Data processing, Algorithms and implementation, Seismological data.

ABSTRACT

Data acquisition by seismic centers relies on real-time systems, like SeisComP3, Antelope and Earthworm. However, these are complex systems that are designed for fast and precisely defined standard real-time analyses. Therefore, it is not a simple task to access or modify internal routines, and to integrate them into custom-processing workflows or to perform in-depth data analyses. Often a library is necessary that provides convenient access to data and allows easy control over all of the operations that are to be performed on the data. ObsPy is such a library, which is designed to access and process seismological waveform data and metadata. We use short and simple examples here to demonstrate how effective it is to use Python for seismological data analysis. Then, we illustrate the general capabilities of ObsPy, and highlight some of its specific aspects that are relevant for seismological data centers and observatories, through presentation of real-world examples. Finally, we demonstrate how the ObsPy library can be used to develop custom graphical user interface applications.

Why Python?

In the scientific community in general, Python has been emerging as one of the most popular programming languages. Its syntax is easy to learn, and it enforces a unified coding style (e.g. by using indentations instead of brackets) that has very little command-flow overhead. Furthermore, it comes with a well-documented, easy-to-use, and powerful standard library (http://docs.python.org/library/). The readability is enhanced drastically, which makes Python a very good choice, and not just for students with little or no basic programming skills. In a computer laboratory practical held in the Munich Geophysics Department with undergraduate students with little programming background, under supervision, the students were able to write their own signal processing scripts after just two afternoon lessons. Just recently, the Massachusetts Institute of Technology also chose Python as the language to be taught to undergraduate Computer Science and Engineering students.

Using the interactive shell IPython (http://ipython.scipy.org) enables scientists to work with the data and to quickly develop complex processing routines. After establishing the processing workflow on small example datasets using the interactive shell, its command history can quickly be condensed to a script for batch processing of large datasets. For more complex programs, Python also provides the power for full-blown object-oriented programming.

An important point for scientists is the visualization of the data. Matplotlib (http://matplotlib.sourceforge.net/) provides an excellent module for generating publication-quality figures. Following the general Python paradigm, it provides convenience functions that are easy to use for beginners, while at the same time, any detail needed in the plots generated is fully customizable by more experienced users. The matplotlib gallery (http://matplotlib.sourceforge.net/gallery.html) illustrates the wide range of plotting capabilities, with source code provided along with all of the example plots.

What also makes Python attractive for scientists is the vast variety of third-party packages that are available at the Python Package Index (http://pypi.python.org/). Currently, there are 701 freely available packages that are tagged as "Scientific/Engineering", with many of those from neighboring disciplines also concerned with signal processing. How seismologists can profit from this central package hub is illustrated by the following short examples.

Consider the conversion of coordinate data from one coordinate system to another, for instance. When working with data from seismic surveys or exchanging data with geological services, these data often use regional Cartesian coordinate systems rather than the WGS84-based spherical system that is widely used in seismology. After looking up the EPSG codes (http://www.epsg-registry.org/) that provide a unique identification of the source and target coordinate system, the conversion can be done in just a few lines of code, using pyproj (http://pypi.python.org/pypi/pyproj). The following example converts the coordinates of two German stations to the regionally used Gauß-Krüger system:

```
>>> lat = [49.6919, 48.1629]
>>> lon = [11.2217, 11.2752]
>>> import pyproj
>>> 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, y
[4443947.179, 4446185.667] [5506428.401, 5336354.055]
```

Figure 1. Example code: Conversion of coordinate information using the Python package pyproj.

Implementation of hierarchical clustering is provided in the package hcluster (http://pypi.python.org/pypi/hcluster). Among other things, this allows clusters to be built from dissimilarity matrices (e.g. computed using the cross-correlation routines in obspy.signal), and dendrogram plots to be made. The following example shows how to do this for an already existing dissimilarity matrix. The dissimilarity data are deduced from events in an area with induced seismicity in southern Germany, as obtained from the ObsPy examples page (http://examples.obspy.org/). Note also how the Python standard library is used to read the data (previously saved in an interactive session) from the ObsPy example web server in only three lines of code:

```
>>> import pickle, urllib
>>> url = "http://examples.obspy.org/dissimilarities.pkl"
>>> dissimilarity = pickle.load(urllib.urlopen(url))
>>>
>>> import matplotlib.pyplot as plt
>>> plt.subplot(121)
>>> plt.imshow(dissimilarity, interpolation="nearest")
>>>
>>> import hcluster
>>> dissimilarity = hcluster.squareform(dissimilarity)
>>> threshold = 0.3
>>> linkage = hcluster.linkage(dissimilarity, method="single")
>>> clusters = hcluster.fcluster(linkage, threshold, criterion="distance")
>>>
>>> plt.subplot(122)
>>> hcluster.dendrogram(linkage, color_threshold=threshold)
```

Figure 2. Example code: Hierarchical clustering and dendrogram visualization of event dissimilarity data using the Python package hcluster.

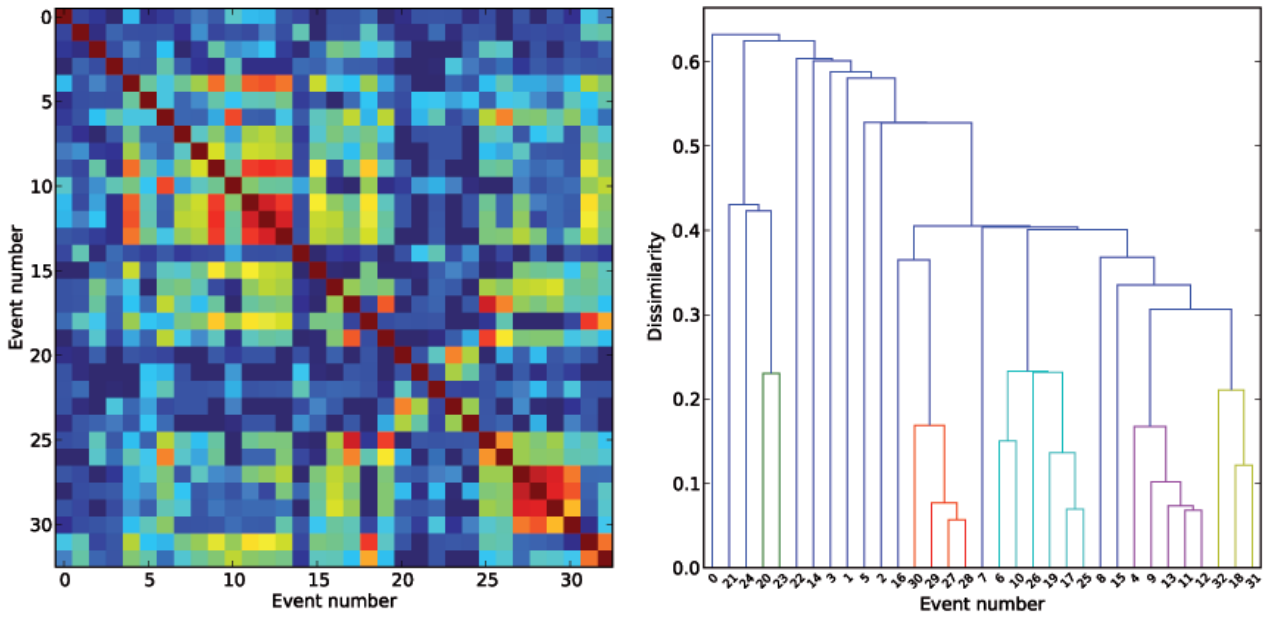

Figure 3. Left: Dissimilarity data for a set of 33 earthquakes. Bluish colors represent high, reddish colors represent low dissimilarities. Right: Dendrogram representation of the hierarchical clustering. Groups of events below the specified dissimilarity threshold are plotted in the same color.

In the last example of the exploitation of existing scientific third-party packages, we use mlpy (Machine Learning Python, http://pypi.python.org/pypi/MLPY) to perform a continuous wavelet transform with a Morlet wavelet, using infrasound data recorded at the active Mount Yasur volcano (Tanna Island, Vanuatu). The continuous wavelet transform allows good time resolution for high frequencies, and good frequency resolution for low frequencies. Thus, the spectral components are not equally resolved, as in the case of the short-term Fourier transform.

```
>>> import matplotlib.pyplot as plt
>>> from obspy.core import read
>>> import numpy as np
>>> import mlpy
>>>
>>> stream = read("http://examples.obspy.org/a02i.2008.240.mseed")
>>> trace = stream[0]
>>>
>>> omega0 = 8
>>> spec, scale = mlpy.cwt(trace.data, dt=trace.stats.delta, dj=0.05,
...                        wf='morlet', p=omega0, extmethod='none',
...                        extlength='powerof2')
>>> freq = (omega0 + np.sqrt(2.0 + omega0**2)) / (4*np.pi * scale[1:])
>>>
>>> t = np.arange(trace.stats.npts) / trace.stats.sampling_rate
>>> plt.imshow(np.abs(spec), extent=(t[0], t[-1], freq[-1], freq[0]))
>>> plt.xlabel('Time [s]')
>>> plt.ylabel('Frequency [Hz]')
>>> plt.show()
```

Figure 4. Computing a continuous wavelet transform using the Python package mlpy.

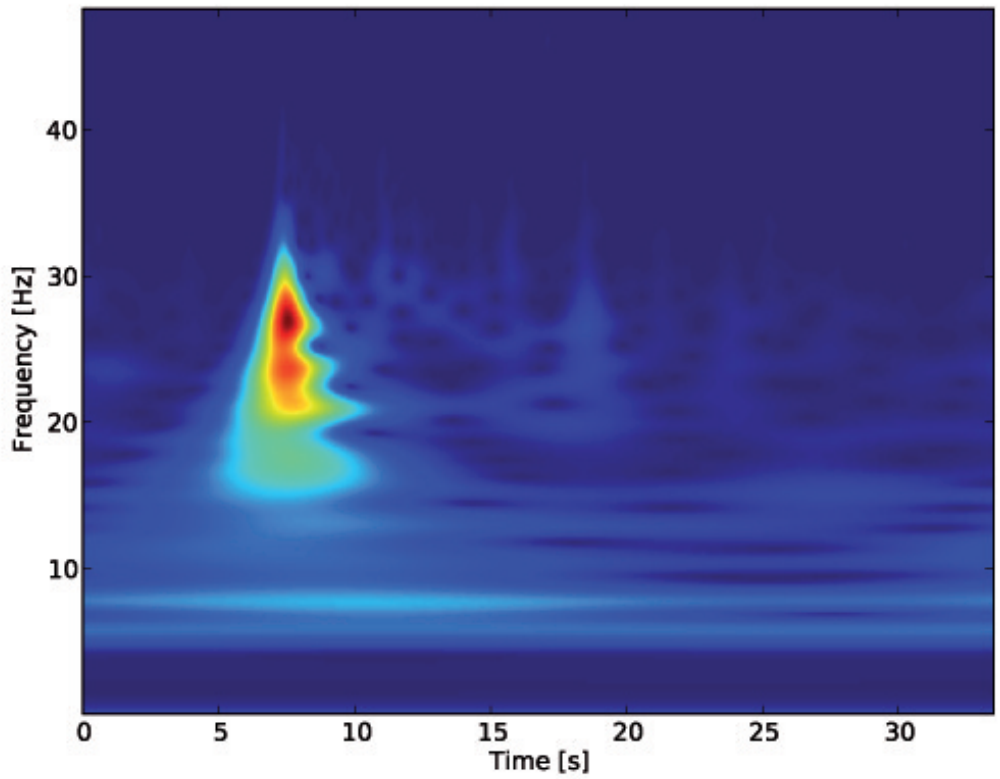

Figure 5. The result of the continuous wavelet transform. Bluish colors represent low energy, reddish colors represent high energy.

An important aspect for data centers and observatories is also their representation on the internet. Apache web servers provide a module to facilitate the execution of Python code embedded in web pages (http://www.modpython.org/). Using this module makes it easy to embed Python code, creating diagrams for routine analysis steps from off-line or on-line data using Python and ObsPy.

Using Python and ObsPy at data centers and observatories

Why use ObsPy?

ObsPy (http://www.obspy.org) [Beyreuther et al. 2010] extends Python by providing routines for the handling of seismological data. It provides read/write support for the most relevant waveform data formats in use at data centers and observatories, it supports the standard metadata exchange format of Dataless SEED, and it comes with clients to interact with the most important data centers at IRIS and ORFEUS/GEOFON. As well as providing support for metadata stored in Dataless SEED or Full SEED volumes, ObsPy can also work with XML-SEED (see section *Handling metadata at data centers using XML-SEED*). Furthermore, it provides numerous routines for signal processing, data analysis and visualization (e.g. plotting waveforms, spectrograms and source mechanism beach balls).

When combined with ObsPy, Python represents a powerful tool for seismologists. Using ObsPy, the general strong points of Python are extended to fit the needs of data centers, observatories, and seismologists in general. It can be seen as an interface that couples various different data formats and data servers to one common, simple object inside Python. The interaction with the actual file formats is handled internally by the various ObsPy submodules, so the user only has to work with the ObsPy simple native object. In this way, users are able to process data from all kinds of different data sources without making any adjustments to file format specifics in their code (see section *Unified handling of data from different sources*).

The advantages of this approach can be seen by looking at the development of the database solution SeisHub (http://www.seishub.org) [Barsch2009]. Initially, it was designed to serve as the new database for the MiniSEED archive at the Geophysical Observatory, Fürstenfeldbruck. In the final stage already, it was decided to use ObsPy for all of the file read/write operations instead of using libmseed directly. In this way, SeisHub can now be used not only to store and serve MiniSEED, but it is also possible to incorporate any other data ObsPy can read into the same database, e.g. the GSE2 data regularly acquired during temporary field experiments.

In addition to handling different data sources, ObsPy provides a library of signal processing routines that are often used in seismology. Among others, these include tapering, filtering and instrument simulation, applying different triggering routines, or using complex trace analyses. Routines for beam-forming and frequency-wave-number analyses were added recently. All of these are ready to use and can be easily integrated in custom processing routines. Through contributions from various graduate and undergraduate students, the signal-processing toolbox is continuously growing.

Less specific signal-processing tools, like the fast Fourier transform, are provided by popular and widely used Python packages, such as NumPy (http://numpy.scipy.org) and SciPy (http://scipy.org). The already existing shared C or Fortran libraries can easily be accessed using the Python foreign function library ctypes (http://docs.python.org/library/ctypes.html). Examples of how to re-use existing code can be found in the source code of the ObsPy `signal` submodule (e.g. the cross-correlation routine `xcorr()`).

To make the start of ObsPy use as easy as possible, the most frequently used operations (e.g. filtering, instrument simulation) are implemented as convenience methods on the `Stream` and `Trace` objects. Reading the ObsPy Tutorial pages (http://www.obspy.org/wiki/ObspyTutorial) is the best way to get an impression of the capabilities of ObsPy, and to get simple example code covering a wide variety of problems in seismological data handling.

Unified handling of data from different sources

One of the major points of ObsPy is its support for all important file formats for seismological waveforms, combined with the ability to connect to the different kinds of servers used by data centers like IRIS and ORFEUS/GEOFON. In this way, data can either be read from local files or imported from data centers by specifying the desired combination of network code, station code, and time range of interest. All of the data read or acquired by ObsPy end up in a simple Python object: `Stream`. This `Stream` object is basically a Python list that contains blocks of contiguous waveform data in one or more `Trace` objects. The actual data is accessible at any time as NumPy arrays (the Python *de-facto* standard module for working on large data arrays) via `trace.data`, which allows the use of fast numerical array-programming routines, as found in NumPy or SciPy, for example. Metadata are stored in a dictionary object (a simple key-value map) as `trace.stats` and can be easily accessed and modified (for details on data structures see Beyreuther [2010] and http://docs.obspy.org/packages/obspy.core.html). New items can be added for use inside processing routines, and they are simply ignored if the data is later written to an output format that does not use these custom fields.

At observatories that run local networks regularly, the need arises to include data from external sources into standard analysis routines. For instance, when analyzing seismic events at border regions of one network, it is essential to include data from the neighboring networks. The following example demonstrates how easy it is to work on data from all kinds of different sources using ObsPy, without the need to adapt their specifics in the code. This is shown using a magnitude re-estimation for a magnitude 4.9 earthquake in Poland. The complete example code can be found in the Appendix. Here, only the major points are illustrated, with short code snippets.

The first thing is to acquire the data: we start off by reading a local three-component GSE2 file, and attaching the metadata by reading a Full/Dataless SEED using the metadata Parser in the ObsPy XML-SEED submodule:

```
>>> from obspy.core import read
>>> from obspy.xseed import Parser
>>> stream = read('20100206_045515_012.BGLD')
>>> parser = Parser('dataless.seed.BW_BGLD')
>>> for trace in stream:
...     trace.stats.paz = parser.getPAZ(trace.stats.channel)
...     trace.stats.coordinates = parser.getCoordinates(trace.stats.channel)
```

Figure 6. Example code: Reading waveform data and attaching metadata from local files.

In the same way, we could read waveform data in all of the other formats that are supported by ObsPy (e.g. SEED, MiniSEED, GSE2, SAC, SEISAN, SH-Q). The format is autodetected. Now we also want to include data from external data centers, for example via ArcLink:

```
>>> from obspy.arclink import Client
>>> client = Client('webdc.eu')
>>> stream = client.getWaveform('CZ', 'PVCC', '', 'BH*', start, end,
...                             getPAZ=True, getCoordinates=True)
```

Figure 7. Example code: Retrieving data from an ArcLink server.

or via a Fissures Client:

```
>>> from obspy.fissures import Client
>>> client = Client(('/edu/iris/dmc', 'IRIS_NetworkDC'),
...                 ('/edu/iris/dmc', 'IRIS_DataCenter'),
...                 'dmc.iris.washington.edu:6371/NameService')
>>> stream = client.getWaveform('OE', 'ARSA', '', 'BH*', start, end,
...                             getPAZ=True, getCoordinates=True)
```

Figure 8. Example code: Retrieving data from a Fissures server.

We are then free to apply arbitrary processing routines on any acquired data-stream, regardless of the data source. Assume we have saved all of the previously acquired streams in a list of streams. We can then loop over this list of streams and process them one by one (or apply more complicated processing routines involving multiple streams together). In the example here, we de-mean the data and apply a cosine taper to avoid artifacts at the start and end of the trace during the band-pass filtering that is applied afterwards:

```
>>> for stream in streams:
...     for trace in stream:
...         trace.data = trace.data - trace.data.mean()
...         trace.data = trace.data * obspy.signal.cosTaper(len(trace), 0.05)
...         trace.filter('bandpass', dict(freqmin=0.5, freqmax=10))
```

Figure 9. Example code: Looping over a list of streams and de-meaning, tapering and filtering all of the traces in every stream.

In the complete example that can be found in the Appendix (including the output), we finally make a rough local magnitude estimation by simply using the maximum and minimum amplitudes in each stream. This includes instrument correction using the instrument response information that was attached to the waveform data earlier. This extremely simplified estimation is shown just to demonstrate the possibilities for automated processing of data from arbitrary sources. The output of the complete program shows that at least the mean of all of the results, of 4.63, fits the catalog entry of 4.9 fairly well (see Appendix).

The example given above shows how simple it is to automate processing routines, even when working on heterogeneous datasets (in terms of the data source).

Handling metadata at data centers using XML-SEED

The ObsPy XML-SEED module is of especial interest for observatories or other data hosts. XML-SEED was introduced by Tsuboi, Tromp and Komatitsch [Tsuboi et al. 2004]. It is a proposal for an XML mark-up standard of Dataless SEED. The XML-SEED format is verbose, human-readable, and easy to extend for data center internal purposes. For instance, additional information on station localities or comments on problems during station operation can be included directly into the XML-SEED (e.g. simply add custom tags using a text editor). Thus, they do not have to be stored in a different place than the rest of the station metadata, which facilitates the reasonable handling of metadata at data centers. For public distribution, the locally stored extended XML-SEED can at any time be converted back to the standard exchange format Dataless SEED. During this process, the additional custom fields (which were intended for internal use anyway) are simply ignored.

The verbosity of XML-SEED is ideal to, for example, store and access Dataless SEED files in databases [Barsch 2009]. To date, the ObsPy XML-SEED module represents the only publicly available XML-SEED implementation (to the authors' best knowledge). ObsPy ships command line programs to convert from Dataless SEED to XML-SEED and back, as well as from Dataless SEED to RESP files (using the ObsPy programs `obspy-dataless2resp`, `obspy-dataless2xseed` and `obspy-xseed2dataless`). These converters are tested against the complete ORFEUS Dataless SEED archive, the whole IRIS Dataless SEED archive, and also the ArcLink response requests.

In addition to the simple conversion, for example, from SEED to XML-SEED (just run `obspy-dataless2xseed my.dataless.seed.file`), the ObsPy XML-SEED module can be used to adapt the values in Dataless SEED volumes for other stations. The IRIS Portable Data Collection Center is certainly the best tool for interactive modification of the Dataless SEED volumes. However, if we just want to adapt a few values in a Dataless SEED volume in an automated way, the ObsPy XML-SEED module might well come in handy.

The following example shows how to clone a Dataless SEED volume and fill in values for a new station. For the sake of clarity, only a few fields are processed (for how to change all of the important fields, refer to the Appendix):

```
>>> from urllib import urlopen
>>> from obspy.xseed import Parser
>>>
>>> url = 'http://examples.obspy.org/dataless.seed.BW_RNON'
>>> p = Parser(urlopen(url).read())
>>> blk = p.blockettes
>>>
>>> blk[50][0].network_code = 'BW'
>>> blk[50][0].station_call_letters = 'RMOA'
>>> blk[50][0].site_name = 'Moar Alm, Bavaria, BW-Net'
>>> blk[33][1].abbreviation_description = 'Lennartz LE-3D/1 seismometer'
>>>
>>> p.writeSEED('dataless.seed.BW_RMOA')
```

Figure 10. Example code: Reading a Dataless SEED file, changing several fields to adapt it to another station, and writing the new Dataless SEED file.

Rapid development of useful tools for observatory practice

In this section, we want to demonstrate how the combination of Python and ObsPy helps to speed up the development of applications for data centers and observatories. The scope of these applications can range from little helper scripts with just a few lines of code, to platform-independent, graphical user interface (GUI) applications that accomplish complex workflows in daily routine work.

ObsPy-Scan

ObsPy-Scan is an example of how the ability of ObsPy to handle different data formats can be exploited in little helper applications that are useful in daily work at data centers and observatories.

In many cases, for example after recovering instruments which were deployed in temporary field experiments without a real-time data link, it is important to obtain an overview of the data availability, and also its quality, in terms of sometimes shorter, sometimes longer gaps in recorded data. ObsPy-Scan is a lightweight script that was written to provide such an overview of data stored in directory trees of local file systems. Using the ObsPy read functionalities, it is possible to parse all kinds of different file formats at the same time.

After recursively scanning the specified file system locations, ObsPy-Scan provides a diagram that groups the data by station code, while indicating the start of contiguous data blocks by crosses, the time periods covered by the data by horizontal lines, and the gaps in the recordings by vertical

red lines. This plot is interactively zoomable, so that it is possible to more closely inspect especially interesting periods (e.g. with many short gaps) in more detail.

Figure 11 shows such an example, with data from a temporary network with twelve stations that was run for about five months. Parsing the vertical component data (30,000 files) takes about one hour. The script had already been used to parse 180,000 files in a single run.

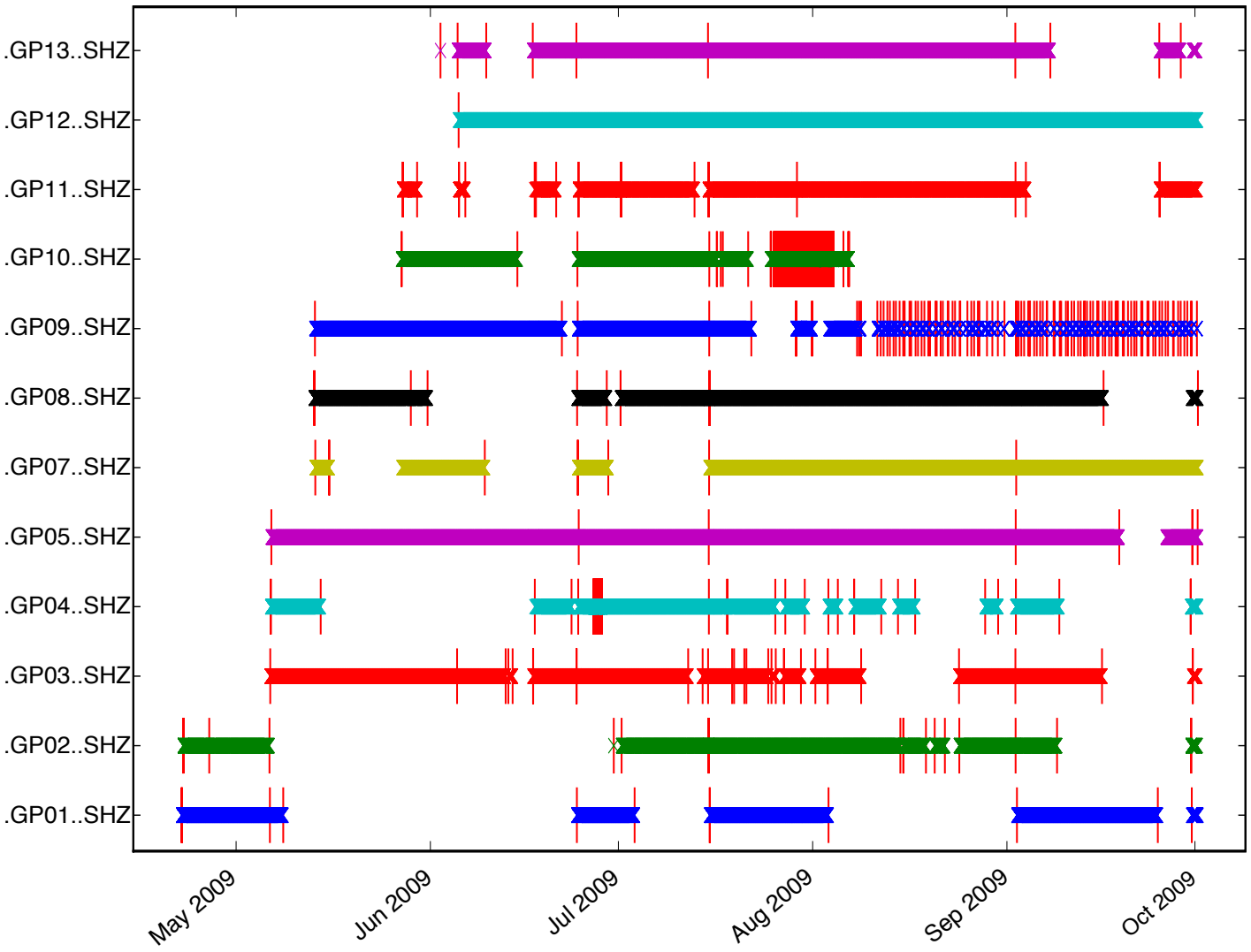

Figure 11. An overview plot for the data of a temporary network that consists of 12 stations and runs for about 5 months, created with obspy-scan. The data coverage for each station is presented in a separate line. Gaps in data availability are indicated by vertical red lines.

H/V Toolbox

The H/V Toolbox was developed in the course of a Bachelor thesis at the Geophysical Observatory, Fürstenfeldbruck. It demonstrates what young scientists on courses at seismological observatories can achieve using ObsPy. The toolbox is used to calculate horizontal to vertical spectral ratios (HVSR), and it can interactively adjust several parameters. The program handles the whole workflow: from the reading and preprocessing of the data, to the automated selection of the appropriate time windows with little seismic activity, to the final calculation of the HVSR in an easy-to-use interface, while giving visual feedback, step by step. It demonstrates how ObsPy can be used in combination with other Python libraries to quickly develop a fully working GUI application for use in applied seismology.

Most of the internal calculations are immensely simplified by using ObsPy. For example, selecting quiet-time windows is done using the built-in ObsPy triggering algorithms in an inverse fashion to create an anti-trigger. The final spectral calculations use Python bindings for a multitaper library, which is written in Fortran to gain better spectral estimates (http://svn.geophysik.uni-muenchen.de/trac/mtspecpy).

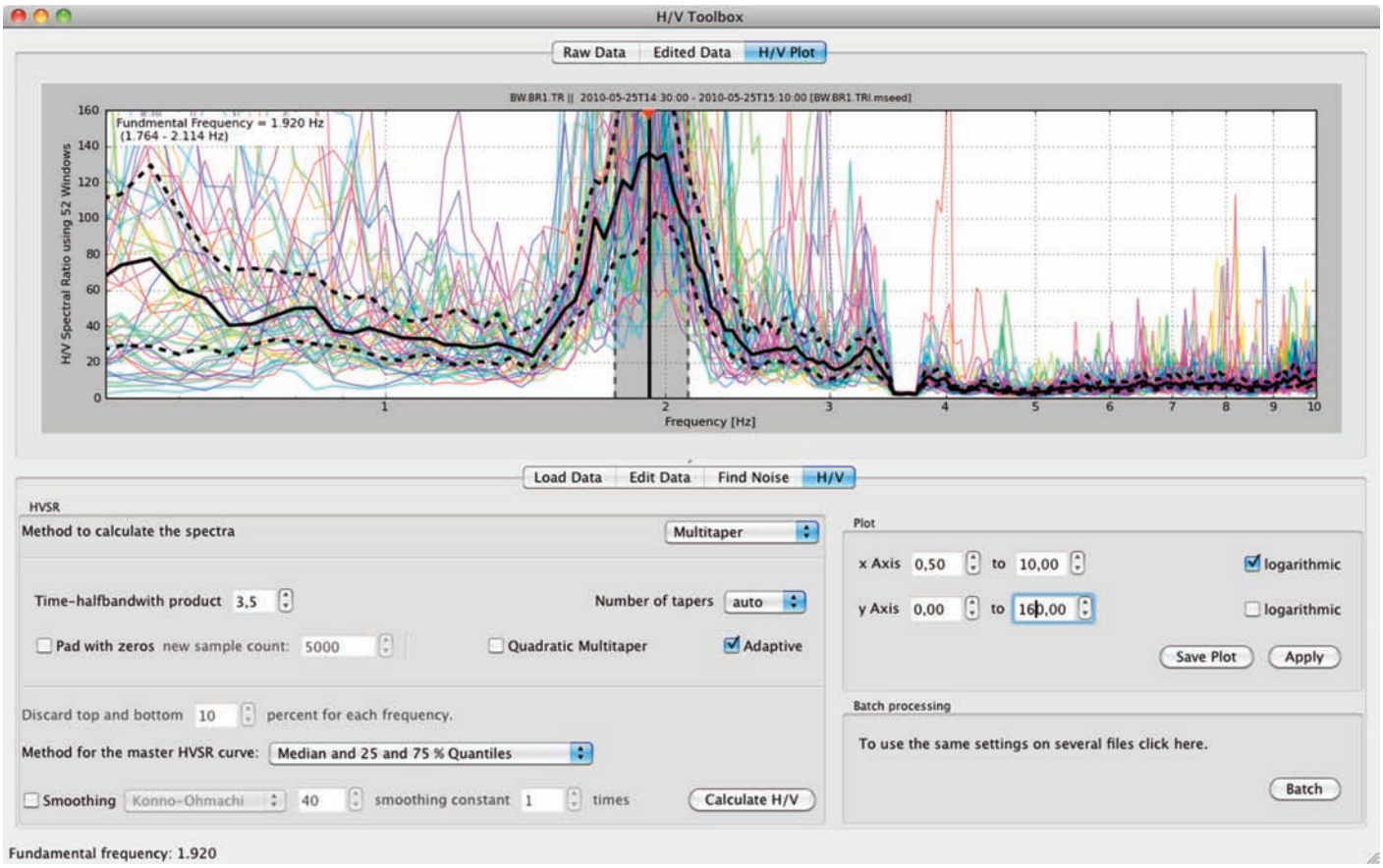

Figure 12. Screenshot: The H/V Toolbox displaying the results of an analysis run.

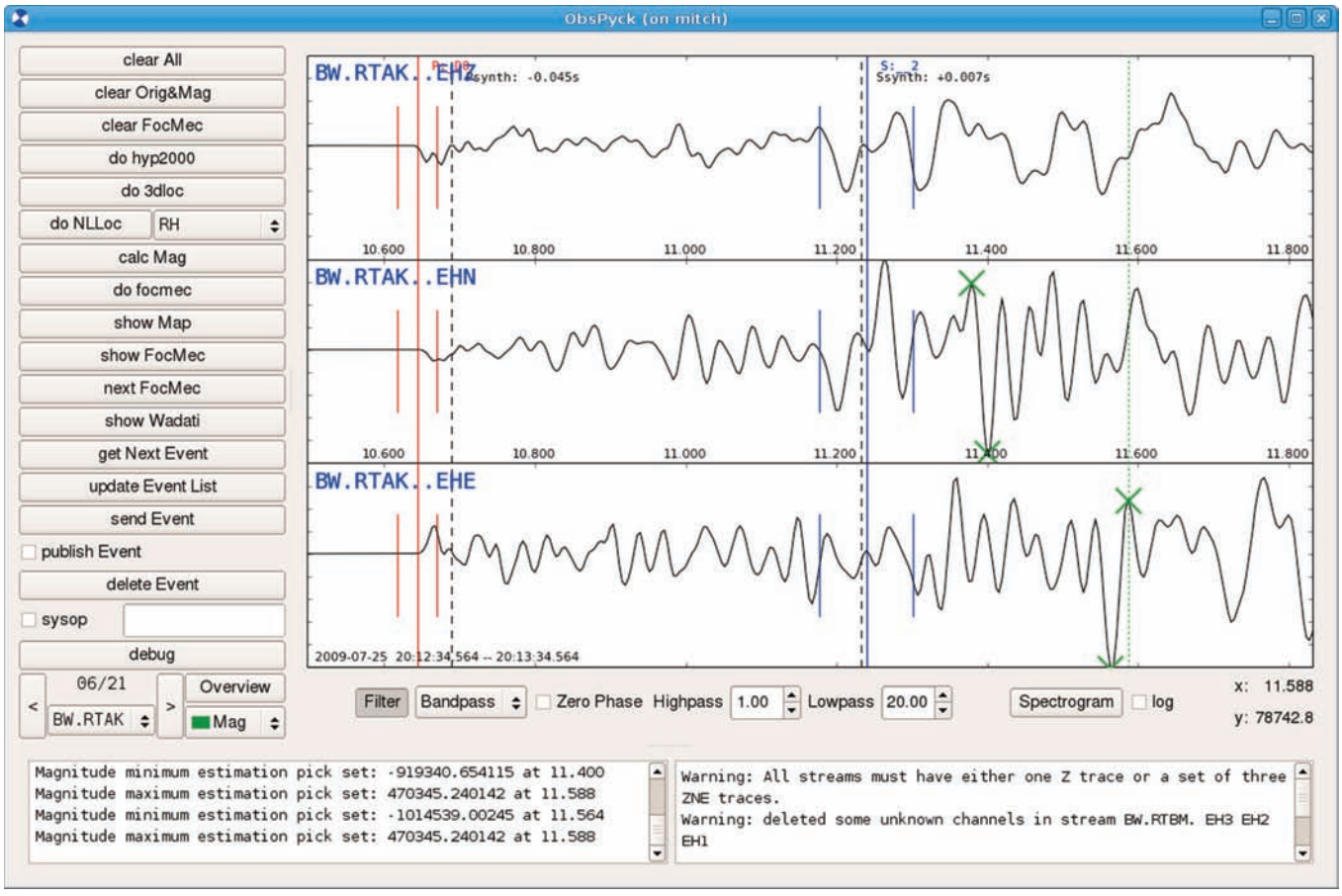

Figure 13. Screenshot: Picking seismic phases and amplitude maxima/minima for magnitude estimation with ObsPyck.

ObsPyck

ObsPyck is a Qt-based (http://qt.nokia.com/) GUI application for routine daily seismological analysis that was developed and is used at the observatory in Fürstenfeldbruck. It serves as an example of how ObsPy can facilitate and speed up the development of complex, important applications for daily use at data centers and observatories.

After switching the whole database systems of the observatory to the new in-house development SeisHub (http://www.seishub.org) [Barsch 2009], there was the need for a new graphical front-end so as to be able to perform daily routine analyses like phase picking and locating events. This was to replace the long-serving workhorse PITSA [Scherbaum and Johnson 1992], which unfortunately fails to compile on modern operating systems. ObsPyck has been in use for routine analysis by young and senior seismologists at the observatory in Fürstenfeldbruck on a daily basis for several months now.

Using the matplotlib plotting capabilities, ObsPyck allows the user to zoom into the custom-filtered waveform data and set phase picks using the mouse and/or the keyboard. Event location is achieved by passing the pick and station metadata via system calls to external programs (e.g. Hypo Inverse 2000, NonLinLoc) and parsing their output. Local magnitudes are estimated using a routine implemented in ObsPy. At the end of the analysis workflow, the results are visualized and converted to a QuakeML-like [Schorlemmer et al. 2004] XML document, and uploaded to the database system.

ObsPyck is based on several ObsPy modules for handling waveform data and metadata, for communicating with data servers, and also for filtering, instrument simulation, and magnitude estimation. By not having to worry about these details, it was possible to develop the first running version that was usable in daily routine in only two months of development. Another advantage lies in the delegation of core routines to ObsPy: further developments on data formats or protocols to communicate with data-center servers are decoupled from the GUI application. Just recently, ObsPyck was extended to be able to import data from ArcLink and Fissures servers, to integrate external data into the location process. Support for files stored locally in the file system has also been added.

For additional screenshots, and information and news of recent developments, please refer to the respective page in ObsPy wiki (https://www.obspy.org/wiki/AppsObsPyck).

Conclusions and Outlook

Although ObsPy is not a real-time data-acquisition system, it should nevertheless be a valuable add-on for data centers when performing custom processing workflows or performing in-depth data analyses. Especially considering the intuitive syntax of Python, together with the available seismological routines in the ObsPy library and the interactive shell, this makes it easy to develop custom code snippets and to apply them on the full data archive. Since the beginning of the ObsPy project about two and a half years ago, ObsPy has proven to be a valuable tool for the handling of seismological data, and it has spread internationally. The free and open ObsPy source package and the detailed documentation that includes an extensive tutorial are available at http://www.obspy.org. Recent additions are alphanumeric SAC support, SH-Q format and beam-forming. We are currently working on an object-based seismic event on QuakeML, which includes communication with common earthquake location algorithms, CM6 GSE1 support, and write capabilities for SEISAN.

To interact with the developers and to get help with any ObsPy-related problems, the ObsPy homepage (http://www.obspy.org) provides the possibility for support request tickets asking for help to be opened anonymously. Usually these are answered within the same day.

Acknowledgments. We would like to thank Heiner Igel for his ongoing commitment and magnanimous support. We would also like to thank Yannik Behr, Conny Hammer, Lars Krieger and Martin van Driel, for their contributions to ObsPy, and Chad Trabant, Stefan Stange and Charles J. Ammon, whose libraries form the basis of the MiniSEED, GSE2 and SAC modules. This study was partially funded by the Leibniz Institute for Applied Geophysics (LIAG) and by the German Ministry for Education and Research (BMBF), GEOTECHNOLOGIEN grant 03G0646H.

References

Barsch, R. (2009). Web-based technology for storage and processing of multi-component data in seismology. First steps towards a new design, PhD thesis, Ludwig-Maximilians-University, München, Faculty of Geosciences.

Beyreuther, M., R. Barsch, L. Krischer, T. Megies, Y. Behr and J. Wassermann (2010). ObsPy: A Python toolbox for seismology, Seismol. Res. Lett., 81, 530-533.

Scherbaum, F. and J. Johnson (1992). PITSA - Programmable Interactive Toolbox for Seismological Analysis, IASPEI Software Library, 5.

Schorlemmer, D., A. Wyss, S. Maraini, S. Wiemer and M. Baer (2004). QuakeML—An XML schema for seismology, ORFEUS Newsletter, 6 (2), 9; www.orfeus-eu.org/Organization/Newsletter/vol6no2/quakeml.shtml.

Tsuboi, S., J. Tromp and D. Komatitsch (2004). An XML-SEED format for the exchange of synthetic seismograms, American Geophysical Union, Fall Meeting 2004, SF31B-03.

*Corresponding author: Tobias Megies,
Ludwig-Maximilians-University, Department of Earth and Environmental Sciences, Geophysics Observatory, Munich, Germany;
e-mail: tobias.megies@geophysik.uni-muenchen.de.

APPENDIX

Example code:
unified handling of data from different sources

```
from obspy.core import read, UTCDateTime
from obspy.xseed import Parser
from obspy.arclink import Client as AClient
from obspy.fissures import Client as FClient
from obspy.seishub import Client as SClient
from obspy.signal import estimateMagnitude, utlGeoKm, cosTaper
import numpy as np

a_client = AClient('webdc.eu')
f_client = FClient(('/edu/iris/dmc', 'IRIS_NetworkDC'),
                   ('/edu/iris/dmc', 'IRIS_DataCenter'),
                   'dmc.iris.washington.edu:6371/NameService')
# seishub server is only accessible from intranet
s_client = SClient('http://teide.geophysik.uni-muenchen.de:8080')

start = UTCDateTime('2010-02-06T04:55:15')
end = start + 240
origin = dict(lat=51.52, lon=16.10)

streams = []
# fetch data from data centers, metadata get attached automatically
options = dict(start_datetime=start, end_datetime=end,
               getPAZ=True, getCoordinates=True)
streams.append(s_client.waveform.getWaveform('BW', 'ROTZ', '', 'EH*', **options))
streams.append(f_client.getWaveform('OE', 'ARSA', '', 'BH*', **options))
streams.append(a_client.getWaveform('CZ', 'PVCC', '', 'BH*', **options))

# load waveform data from gse2 file and attach metadata from dataless seed
tmp_stream = read('http://examples.obspy.org/20100206_045515_012.BGLD')
parser = Parser('http://examples.obspy.org/dataless.seed.BW_BGLD')
for trace in tmp_stream:
    trace.stats.paz = parser.getPAZ(trace.stats.channel)
    trace.stats.coordinates = parser.getCoordinates(trace.stats.channel)
streams.append(tmp_stream)

# load waveform data from full seed file and attach metadata
tmp_stream = read('http://examples.obspy.org/fullseed_EE_VSU')
parser = Parser('http://examples.obspy.org/fullseed_EE_VSU')
for trace in tmp_stream:
    trace.stats.paz = parser.getPAZ(trace.stats.channel)
    trace.stats.coordinates = parser.getCoordinates(trace.stats.channel)
streams.append(tmp_stream)

# now we do some computations on the streams regardless where they came from
magnitudes = []
filter_options = dict(freqmin=0.5, freqmax=10, zerophase=False)

for stream in streams:
    for trace in (stream.select(component='N')[0], stream.select(component='E')[0]):
        # preprocess data: demean, taper and filter
        trace.data = trace.data - trace.data.mean()
        trace.data = trace.data * cosTaper(len(trace), 0.05)
        trace.filter('bandpass', filter_options)
        # simply use min/max amplitude for magnitude estimation
        delta_amp = trace.data.max() - trace.data.min()
        delta_t = trace.data.argmax() - trace.data.argmin()
        delta_t = delta_t / trace.stats.sampling_rate
        delta_x, delta_y = utlGeoKm(origin['lon'], origin['lat'],
                                    trace.stats.coordinates.longitude,
                                    trace.stats.coordinates.latitude)
        hypodist = np.sqrt(delta_x**2 + delta_y**2) # neglect depth
        mag = estimateMagnitude(trace.stats.paz, delta_amp, delta_t, hypodist)
        magnitudes.append(mag)
        print '%s: %.1f' % (trace.id, mag)

print '\nNetwork Magnitude: %.2f' % np.mean(magnitudes)
```

Figure 14. Example code: This program shows how to fetch data from different servers, read data from local files (in the example, read from a web server) and do batch processing on all of the streams acquired.

Program output:

```
BW.ROTZ..EHN: 4.4
BW.ROTZ..EHE: 3.8
OE.ARSA..BHN: 3.7
OE.ARSA..BHE: 3.9
CZ.PVCC..BHN: 3.7
CZ.PVCC..BHE: 3.2
BW.BGLD..EHN: 6.7
BW.BGLD..EHE: 5.9
EE.VSU..BHN: 5.9
EE.VSU..BHE: 5.1

Network Magnitude: 4.63
```

Figure 15. Example code: Output of code in Figure 14.

Example Code: XML-SEED

The following code shows how to customize all of the important fields when cloning a Dataless SEED file using the ObsPy XML-SEED module.

```
from urllib import urlopen
from obspy.core import UTCDateTime
from obspy.xseed import Parser

url = "http://examples.obspy.org/dataless.seed.BW_RNON"
p = Parser(urlopen(url).read())
blk = p.blockettes

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"

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
```

Figure 16 (continues on next page). Example code: Cloning a Dataless SEED file, filling in all of the important fields.

```
39      blk[53][i].normalization_frequency = 3.0
40      # stage sequence number 1, seismometer gain
41      blk[58][i*mult].sensitivity_gain = 400.0
42      # stage sequence number 2, digitizer gain
43      blk[58][i*mult+1].sensitivity_gain = 1677850.0
44      # stage sequence number 0, overall sensitivity
45      blk[58][(i+1)*mult-1].sensitivity_gain = 671140000.0
46
47  p.writeSEED("dataless.seed.BW_RMOA")
```

Figure 16 (continues from previous page). Example code: Cloning a Dataless SEED file, filling in all of the important fields.