Bachelor Thesis

Automated retrieval and quality control of seismic waveform data in Python

Development of a seismic data download tool using the ObsPy Python seismological data processing framework

Ludwig-Maximilians-University, Munich
Department of Earth and Environmental Sciences
Geophysics
Munich, August 2011
List of Figures

1.1 IRIS DMC Archive Growth .................................................. 6
1.2 IRIS DMC Shipments ....................................................... 7
1.3 No Data Left Behind - the grand scheme ............................ 8

2.1 Seismic wave refraction .................................................. 10
2.2 Phase names .................................................................. 11
2.3 Arrival times for the iasp91 model .................................. 12

3.1 ObsPyLoad source code diagram ...................................... 17

4.1 obspyload.py command line parameters helper diagram .......... 33
4.2 File structure inside the data folder .................................. 35
4.3 File structure inside the metadata folder ............................. 36
4.4 Waveform data plot ........................................................ 37
4.5 Filled-up plot ............................................................... 38
Abstract

To seismology, data is the most important resource and the only means to scientific progress. Although its retrieval can be one of the most tedious necessities for geophysicists, this activity does not advance science in the first place.
In recent years, the amount of available data grew exponentially. To take full advantage of this development, the data download workflow should therefore be performed as automated and convenient as possible.
To facilitate this common task, an automated and cross-platform command line data download tool has been developed in the course of this Bachelor Thesis. Additional to event based data and metadata selection, retrieval and management for various data providers, it features basic quality control and waveform plotting.
Using this tool provides the outlook of saving an significant amount of time, which may then be allocated to advanced scientific problems.
Chapter 1

Introduction

1.1 The exponential growth in seismic data volume

In seismology, like in all quantitative experiment- or observation-driven sciences, data is the key to new insight. To deduce correct theories and models, it is necessary to perceive nature undistorted. Various forms of data are our only means to achieve this.

As can be seen in Figure 1.1, the past decade brought an exponential rise in data volume. The times when data has been a bottleneck in seismology are gone. It is clear that this vast amount of data, in itself, is solely advantageous. However, the individual scientist’s expenditure of time needed for tasks like data acquisition and processing may now quickly exceed a tolerable limit. Seismology is already suffering from that confinement, a large amount of analysis needs to artificially restrict itself to a particular geographical area and technique. It is therefore critical to quickly adopt data processing techniques to the present situation (Crotwell, 2007).

The necessity arises that tasks like data acquisition, quality control and processing, but ultimately also tasks like basic interpretation and modelling, need to be performed as automated as possible to free up the seismologist’s time, which can then be used to perform other tasks.

Figure 1.1: Exponential Growth of the IRIS seismic waveform data archive. From IRIS annual report 2010.
1.2 Motivation

Data acquisition is the commencing task of all further analysis. Naturally, this tedious task should be as invisible and least time-consuming as possible. Centralized data providers like IRIS DMC (IRIS Data Management Center) or ORFEUS (see Section 2.3) deliver large amounts of data to scientists and students all over the planet. As can be seen in Figure 1.2, over 90 TB of data have been shipped in 2010 solely by the IRIS DMC, for example.

![DMC Shipments by Request Type](image)

**Figure 1.2**: Shipments of IRIS DMC by request type. From IRIS annual report 2010.

SOD (Standing Order for Data)\(^1\) (Owens et al., 2004) is a versatile tool to automatically select, download and process seismological data from data centers that support the IRIS/FISSURES Data Handling Interface (DHI) protocols (Ahern, 2001).

To select data, the user configures an XML. After downloading, the user may process the data with routines which may be either custom written or included within SOD. Additionally to downloading historical data, the user can also define a “standing order” to automatically download data for future events. Typically, a considerable amount of time is spend on figuring out how to best use SOD (see e.g. Jusri (2010, 14)).

The goal of this Bachelor Thesis is to provide a solid and simple command line tool as an alternative to SOD, uniting various datacenters. Objectives include automating the processes of event-based data selection, (meta)data acquisition and management as well as basic quality control. This is a very rewarding task, since writing software for a routine problem like data acquisition brings the prospect of helping a considerable number of seismologists. ObsPy (http://obspy.org, see Section 3.3) provides an excellent framework to achieve this.

The author chose the name ObsPyLoad for the tool that has been developed in the course of this thesis. Figure 1.3 shows how it fits into the grand scheme of the *No Data Left Behind (NDLB)* project (Seyed Kasra Hosseini zad, Karin Sigloch, Simon Stähler, and Tarje Nissen-Meyer, 2011).

Of course, this thesis can only try to cover a small part of the NDLB algorithm. It aims at the tasks in the top-left bounding box of Figure 1.3.

\(^1\)See [http://www.seis.sc.edu/SOD/](http://www.seis.sc.edu/SOD/)
Figure 1.3: How this thesis fits into a broader perspective. *No Data Left Behind* algorithm (Seyed Kasra Hosseini zad, Karin Sigloch, Simon Stähler, and Tarje Nissen-Meyer, 2011). *ObsPyLoad* aims to cover the top-left bounding box.
Chapter 2

Theoretical Background

This chapter will briefly present the theoretical background of seismic sources, rays, theoretical Earth models and data centers. It will end with concepts of seismic quality control.

2.1 Source Theory

2.1.1 Momentum Equation

The (inhomogeneous) moment equation is an important basis for seismological wave theory. It describes the wave motion in a continuum:

\[ \rho \frac{\partial^2 u_i}{\partial t^2} = \partial_j \tau_{ij} + f_i \] (2.1)

Here, the second derivative of the displacement \( u \) with respect to time \( t \) is the acceleration, while \( \tau \) is the stress tensor. \( f_i \) are the components of the force \( f \), which can be split up into a gravity term \( f_g \) and a source term \( f_s \). \( \rho \) denotes the density of the material (Shearer, 1999, 26).

2.1.2 Green’s Function and Moment Tensor

In a volume \( V \) with a surface \( S \), the internal displacement field is dependent on the original conditions, tractions on \( S \) as well as forces in \( V \). If a unit force vector \( f(x_0, t_0) \) is applied, and \( x_0 \) denotes the point while \( t_0 \) denotes the time of application, the displacement \( u(x, t) \) can be measured at another point \( x \) and time \( t \).

By defining the Green’s function \( G(x, t) \), it is possible to isolate the source terms from other aspects of wave propagation:

\[ u_i(x, t) = G_{ij}(x, t; x_0, t_0) \cdot f_j(x_0, t_0) \] (2.2)

Here, \( u \) denotes the displacement, \( f \) represents the force, while \( G_{ij} \) are the components of the Green’s function. The Green’s function is the impulse response of the system.

After calculating \( G \), which depends on all elastic properties as well as suitable boundary conditions, this equation becomes extremely powerful. It now is possible to calculate the displacement of arbitrary body force distributions by merely applying the superposition principle - that is, summing over all solutions of the contributing point sources. This works as long as the sources are small compared to the wavelength of the radiated energy.

It is clear that a single force can not just occur out of nowhere in a medium. If the system is closed, that is no external forces are applied, in order to preserve momentum, forces must always occur as couples that cancel each other out.

These couples are called force couple. In the more general case, force vectors are not acting on the same point of application, but are parted in the direction perpendicular to their orientation. Then, the angular momentum is not preserved by a single force couple, therefore a double couple consisting of four individual force vectors is necessary to preserve the momentum.

The individual entries \( M_{ij} \) of the moment tensor \( \mathbf{M} \) are respectively defined as the force couples pointing along the \( i \) direction and separated in the \( j \) direction. The moment tensor needs to be symmetric, that is \( M_{ij} = M_{ji} \).
order for the angular momentum to be conserved.

\[
M = \begin{bmatrix}
M_{11} & M_{12} & M_{13} \\
M_{21} & M_{22} & M_{23} \\
M_{31} & M_{32} & M_{33}
\end{bmatrix}
\]  \hspace{1cm} (2.3)

Using equation 2.2, the displacement of each force couple can be represented as

\[
u_i(x, t) = \frac{\partial G_{ij}(x, t; x_0, t_0)}{\partial x_k} M_{jk}(x_0, t_0)
\]  \hspace{1cm} (2.4)

Equation 2.4 shows the linear dependence of the displacement and the individual entries of the moment tensor (Shearer, 1999, 165-168).

### 2.2 Ray Theory

Although ray theory is a simplifying model for wave propagation inside the Earth, it is still sufficient for a wide range of applications, including earthquake locating algorithms. Having the advantage of simplicity, it is still sufficient to explain many problems.

#### 2.2.1 Law of Refraction

Seismic ray theory is largely equivalent to optics. The Law of Refraction, also known as Snell’s Law, describes the angles occurring while rays are refracted at a boundary:

\[
\frac{\sin i_1}{\sin i_2} = \frac{v_1}{v_2}
\]  \hspace{1cm} (2.5)

Here, \(v_1\) and \(v_2\) denote the velocities in the upper and lower layer, while \(i_1\) and \(i_2\) are the angle of incidence and the angle of refraction. If the lower medium has a higher velocity than the upper medium, as is often times the case for Earth, the ray is refracted away from the vertical.

In case the ray is reflected, the angle of incidence equals to angle of reflection.

#### 2.2.2 Phase Names

Since a variety of different seismic phases exist, it is necessary to assign names to those phases. The phase name of a ray consists of letters denoting the different phases it was converted to along its path through different layers:

- P: P-wave in the mantle
• S: S-wave in the mantle
• K: P-wave in the outer core
• I: P-wave in the inner core
• J: S-wave in the inner core
• c: reflection off the core-mantle-boundary
• i: reflection off the inner-core-boundary

For example, a ray that started in the mantle as a P-wave, then traveled through the outer core as a longitudinal (P) wave and then through the mantle as a transversal (S) wave would be termed PKS.

For multiple surface reflections, the phase is for instance called PP for a P-wave reflected once from the surface, PPP if reflected twice, and so on. A conversion from P to S due to one single surface reflection would be called PS. In the case of a deep earthquake, the waves traveling directly up are called p and s, a surface reflection of those may be called pP, for example. Figure 2.2 shows a simplified overview of some selected phase names for the whole Earth.

The visibility of seismic phases in the seismogram depends on various characteristics like frequency content, amplitude and polarization. Modern broadband seismometers measure the three components of ground motion distinctly as well as recording a broad frequency range.

A major task in the daily routine of seismological observatories is the so called picking of arrival times. Here, a seismologist first tries to spot events in an overview of continuous seismic data, and then tries to pin down the arrival times of different phases in the seismograms of individual stations. While, for example, a P wave can be best seen in the vertical component of the seismogram, it radiates only very small amounts of energy in the two horizontal components (Shearer, 1999, 36-53).
2.2.3 Velocity Models

This thesis will limit itself to models of sole radial dependency. These one-dimensional models can be represented and reviewed as a table listing various properties at different depth ranges. Using velocity models, it is possible to calculate the theoretical arrival time for a given event location and origin time. PREM (Dziewonski & Anderson, 1981) includes, besides several other parameters, a velocity structure for the Earth.

The model that, by default, is used by ObsPyLoad (see Section 3.5), is the *iasp91* model, which has been “a major international effort made by the Sub-Commission on Earthquake Algorithms of the International Association of Seismology and the Physics of the Earth’s Interior (IASPEI) to generate new global traveltime tables for seismic phases to update the tables for Jeffreys and Bullen (1940)” (Kennett & Engdahl, 1991). See Figure 2.3 for a plot of the arrival times for numerous phases for this model.

The second velocity model available in the ObsPyLoad script (see Section 3.5) is the *ak135* model (Kennett et al., 1995), which is an improvement on the *iasp91* model, especially for the S phase.

For both the *iasp91* and the *ak135* velocity models, ObsPyLoad uses the *obspy.taup* module. This module relies on *iaspei-tau* traveltime table package (Snoke, 2009), which is written in FORTRAN and first became available in 1991. Since then, it has been updated numerous times¹.

![Figure 2.3: A plot of the arrival times of numerous phases for the *iasp91* model using the *obspy.taup.travelTimePlot* function](image)

2.3 Seismological Data Providers

This section will quickly mention the seismological data providers and data centers which are of importance to ObsPyLoad (see Section 3.5).

NERIES² (Network of Excellence of Research and Infrastructures for European Seismology) aims to improve European seismic network access, such as data access and improving access to specific seismic infrastructures.

¹A Java alternative to *iaspei-tau*, including several additional velocity models and features such as a GUI, is the TauP toolkit (Crotwell *et al.*, 1999).

²See [http://www.neries-eu.org](http://www.neries-eu.org)
2.4. QUALITY CONTROL

The WebDC initiative\(^3\) of the German GEOFON (Geoforschungsnetz) and BGR (Bundesanstalt für Geowissenschaften und Rohstoffe) founded the ArcLink distributed data request protocol (Hanka & Kind (1994), WebDC (2011)). It is suitable to download MiniSEED, Dataless SEED and Full SEED files.

One goal of NERIES is to include all large European seismic data centers into the ArcLink network. This way, a European Integrated Data Center (EIDAC) is created.

ORFEUS\(^4\) (Observatories and Research Facilities for EUropean Seismology, van Eck & Dost (1999)), the non-profit foundation that, among other tasks, also coordinates NERIES, operates a major data center for the European-Mediterranean region that uses the ArcLink protocol.

The IRIS\(^5\) consortium currently consists of more than 100 US-American universities “dedicated to the operation of science facilities for the acquisition, management, and distribution of seismological data” (IRIS website, 2011). Working to gain knowledge of the Earth based on seismic and other geophysical methods, IRIS currently hosts arguably the most significant seismological network. Important policies of IRIS are to provide free, unrestricted data access as well as the use of data format and exchange protocol standards.

2.4 Quality Control

As mentioned in Section 1.1, data is the main source of insight to the seismologist, but no good scientist blindly trusts his data. Quality Control is therefore a critical part of the workflow. Some important aspects of Quality Control are described in the following.

Gaps are missing parts in a waveform data file. In real-time systems, data gaps can be caused by dropped packets due to the network connection. The unpleasant effect of data gaps in real-time systems can also impede offline processing significantly, because some processing routines require complete and gapless data (Morozov \& Pavlis, 2011). In ObsPy (see Section 3.3), gaps are represented by masked values\(^6\) in the Trace.

Overlaps are parts of the data where included individual time intervals overlap each other, causing ambiguously defined values.

In ObsPyLoad (see Section 3.5), the number of gaps and overlaps is counted for each station (that is, for each Trace) and saved in the respective column of the file quake.txt (see Section 3.5).

MiniSEED files have a fixed section in the data header which can hold the Data Quality information. This section includes these data quality flag bits (from the SEED manual, see Ahern et al. (2007)):

- Bit 0: Amplifier saturation detected (station dependent)
- Bit 1: Digitizer clipping detected
- Bit 2: Spikes detected
- Bit 3: Glitches detected
- Bit 4: Missing/padded data present
- Bit 5: Telemetry synchronization error
- Bit 6: A digital filter may be charging
- Bit 7: Time tag is questionable

The total count of these bits can be acquired with ObsPy (see Section 3.3). ObsPyLoad (Section 3.5) uses this feature, summing over all data quality bits of all stations for one event and adding this information to the event catalog (events.txt) file.

Another aspect of quality control is visual control. An overview of all waveforms in a dataset for a particular event can quickly reveal problems, for example with defect individual waveform data, which often stands out

---

\(^3\)See http://www.webdc.eu
\(^4\)See http://www.orfeus-eu.org
\(^5\)See http://www.iris.edu
\(^6\)Masked values, a functionality of masked arrays (http://docs.scipy.org/doc/numpy/reference/maskedarray.generic.html), are values in a given array (matrix) that have been marked as invalid or missing. Any operation acting on such an array will simply ignore those values, making a masked array easier to handle than an array that contains NaNs (http://en.wikipedia.org/wiki/NaN).
in color, or very noisy data. It is also possible to assess the quality of the dataset by comparing the actual arrival times to the theoretical ones.

*ObsPyLoad* (see Section 3.5) offers the possibility to save such a plot for each individual event as well as for all events stacked.
Chapter 3
Implementation

3.1 The Python programming language

Python (http://www.python.org) is a free and open source\(^1\), interpreted, interactive, object-orientated programming language that gained popularity at a very fast pace recently. Its module-extensible structure, combined with dynamic typing and classes, provides a very natural and elegant syntax and high capabilities. Good Python code, with well chosen variable and object names, almost reads like the English language. The syntax includes almost no unnecessary or distracting elements which often complicate other programming languages. As stated correctly in the official Python FAQ\(^2\), Python is an excellent programming language for beginners. At the same time, it does not limit advanced programmers. A core philosophy is to not enforce a certain programming paradigm, leaving a lot of freedom to the programmer.

Being an interpreted, high-level language, Python code runs considerably slower than equivalent code written in low-level languages like C or FORTRAN. However, for many applications, the disadvantage in execution speed is heavily outweighed by the immense benefit in both development speed and code simplicity/readability. If a subroutine runs too slow in native Python, it can be written as shared C library and wrapped with ctypes\(^3\).

3.2 NumPy, SciPy and Matplotlib

Python’s popularity definitely owes to a lot of sophisticated extensions created by the community, providing fast routines and vastly expanding Python’s capabilities. Three of them, which are required for in ObsPyLoad, will be briefly described here.

NumPy (http://www.numpy.org/) is an open-source Python module for performing numerical calculations with large, multi-dimensional arrays and matrices. Since it is mostly written in C, it is very fast. In a Python program using NumPy, the more operations can be expressed as array or matrix manipulation, the faster the code will run. In reality, just like with the re module\(^4\) used in ObsPyLoad, it seems best to find a sensible balance between performance using external modules and code readability using native Python code.

SciPy (http://www.scipy.org/) is a open-source Python library relying on NumPy. It is used for tasks like advanced math, signal processing or statistics. According to the official FAQ\(^5\), “SciPy is targeted at engineers, scientists, financial analysts, and others who consider programming a necessary evil” (Jones et al., 2001–2011).

Matplotlib (http://matplotlib.sourceforge.net/) is a popular package for two-dimensional plotting. For example, it can be used within GUI applications and Python scripts and is capable of producing publication-quality images (Hunter, 2007). In ObsPyLoad, the waveform data and theoretical arrival times are plotted using Matplotlib.

---

\(^1\)See http://docs.python.org/license.html for license details.

\(^2\)http://docs.python.org/faq/

\(^3\)http://python.net/crew/theller/ctypes/

\(^4\)http://docs.python.org/library/re.html

\(^5\)http://scipy.org/FAQ
3.3 ObsPy

ObsPy (http://obspy.org), a Python framework for processing seismological data, is a free\(^6\) and open-source project initiated by Moritz Beyreuther, Lion Krischer and Robert Barsch in 2008 at the Department of Earth and Environmental Sciences, Geophysics, LMU Munich. Its modular structure and platform independency is combined with a variety of elaborated tools. Time critical tasks are implemented via shared C libraries (Barsch, 2009, 58).

Providing a software standard sufficient for a complete seismological preprocessing work-flow, ObsPy relieves seismologists from the necessity to use a multitude of different software for subsequent processing steps. Since ObsPy is written in Python, a powerful and complete programming language with many scientific libraries and possibilities is right at hand. Being free software, it also liberates the user from restrictive license policies of proprietary alternatives (Beyreuther et al., 2010).

In fairly short time, useful and important applications have been developed on the basis of ObsPy, like \(H/V\) Toolbox\(^7\), a toolbox to calculate horizontal to vertical spectral ratios to use ambient seismic vibrations (Krischer, 2010), or ObsPyck\(^8\), a GUI application for daily seismological analysis like phase picking. Though not being a real-time data acquisition system, ObsPy should be highly useful to seismological data centers. Detailed tutorials and documentation is available on the project home page (Megies et al., 2011, 53-55).

For the project of this thesis, most of the necessary underlying functionality had already been implemented into ObsPy. Some small additions and modifications have been committed by the author, who previously had no experience with ObsPy. This shows how quickly beginning programmers can get started with this well-documented development framework. Parallel to this thesis, the module obspy.taup has been written by the ObsPy development team, providing theoretical arrival time calculation, which is frequently used in ObsPyLoad.

3.4 QuakeML

QuakeML (Wyss et al., 2004) is an XML representation of seismological metadata, such as descriptive event data like moment tensors (see Section 2.1.2). Due to its flexible, extensible and modular design, it is suitable for numerous fields in seismology.

Having the advantage of being an “an open standard and [being] developed by a distributed team in a transparent collaborative manner”\(^9\), it seems like a good long-term choice which will be useful to both present and future seismologists.

3.5 Description of the ObsPyLoad source code

The objective of this section is to clarify the proceedings of ObsPyLoad’s source code to the reader and potential code contributor.

It has been a design goal to keep the code structure simple and intelligible and provide many comments, so other programmers can get into it and improve or supplement the code quickly. This has in part been achieved by strictly sticking to the PEP8 style guide (Guido van Rossum, Barry Warsaw, 2001), as well as by providing exhaustive source-code commentation and usage help.

As a convenience, the full source code of ObsPyLoad has been added to this thesis as an attachment (see Attachment C). If in doubt how the code works, it sometimes may help to run ObsPyLoad in debug mode, by starting the script with

\[
\$ \text{obspyload.py} \ -d
\]

The rough code structure is defined inside a main function which then calls several evacuated functions. The reasons for this are to reduce code repetition and overloading of the main function. It may be useful to carefully follow along with Figure 3.1 while reading this section.

---

\(^6\)GNU Lesser General Public License, Version 3 (http://www.gnu.org/copyleft/lesser.html)

\(^7\)http://obspy.org/browser/trunk/apps/HtoVToolbox

\(^8\)http://obspy.org/wiki/AppsObsPyck

\(^9\)See https://quake.ethz.ch/quakeml
3.5 DESCRIPTION OF THE OBSPYLOAD SOURCE CODE

CHAPTER 3. IMPLEMENTATION

Figure 3.1: ObsPyLoad source code diagram.

3.5. DESCRIPTION OF THE OBSPYLOAD SOURCE CODE

CHAPTER 3. IMPLEMENTATION

Figure 3.1: ObsPyLoad source code diagram.
3.5.1 Module Imports

When the user imports or runs obspyload.py, after skipping the license comments, lines 18-62 will import the necessary modules. If a module it not crucial for the script, it is surrounded by a try-except statement. That way, the script will continue without those modules and tell the user about the missing modules and resulting missing functionality.

Computer code is often full of abbreviations - for such obvious reasons as to type less and being able to fit more logic while limiting your line length to 79 characters, as the PEP8 style guide (Guido van Rossum, Barry Warsaw, 2001) suggests. Though most abbreviations should be intelligible, starting from line 62 a large comment box explains abbreviations used throughout the code.

3.5.2 Keypress-Thread

One important necessity of a data downloader is to be able to pause or stop an unfinished downloading task and pick it up later. The natural behaviour of a user wanting to interrupt the download is probably to send the interrupt signal (SIGINT) to the program, for example by pressing Ctrl-C. The pitfall about using this method is, that unless in the unlikely case that the program is just in between two downloads (and is also not writing to one of the catalog or exception files), this would result in a corrupt file. Since most time is spend on downloading and saving waveform data, it is most likely that there would be a defect MiniSEED file.

It would be possible to double-check the unscathed condition of all files when resuming the download, but the preferred solution should be to not let the file get corrupted in the first place. The most obvious choice would be to catch when the user presses Ctrl-C, prevent the program from quitting immediately, finish the last task and then quit.

However, due to technical difficulties, the method that the author decided to implement is to define the class `keypress_thread` as well as the functions `getkey()` and `check_quit()`. The pertinent code can be found in lines 103-159 of obspyload.py.

`keypress_thread` is a child of `threading.Thread` and runs as a second thread capturing all keypress events. As soon as “q” is pressed, the `quit` flag\(^{10}\) is set to `True`. The function `getkey()` is used by `keypress_thread` and itself uses the `termios` module to stand by and wait for a keypress event. `getkey()` then returns the character to `keypress_thread` where it is called inside the `while not done` loop in line 118.

The final ingredient is the function `check_quit()`. It is called periodically between downloads from the main thread and checks if the global `quit` variable flag has been set to `True` by the `keypress_thread`. If so, the main thread will exit. The `keypress_thread` will exit automatically after it has set the `quit` flag.

Although this method works fine and does the job for now, there are several disadvantages attached to it. First, it uses the `termios` module which only works with UNIX versions that provide Posix termios style tty input/output control (Python termios module documentation, 2011). That means that e.g. Windows users are left out. In order not to raise an exception when trying to import the `termios` module, the program checks for the operating system and handles Windows differently. On Windows, `termios` is not imported, the `keypress_thread()` is just an empty thread and `check_quit()` is an empty function (see lines 85-103).

Another inconvenience with this method is that if the user presses Ctrl-C, he will not return to the command line prompt until he additionally presses “q”.

3.5.3 Main function

The `main()` function (lines 159-1177) is the heart of the program. It will run from top to bottom if the data download routine is used. As can be seen in Figure 3.1, in case the user wants to download metadata or enter exception mode, the respective functions are called and will end the program after finishing their task.

Config- and OptionParser section

`ObsPyLoad` uses the `ConfigParser` and `optparse.OptionParser` modules to handle command line options and accompanying variables, read from a config file and generate the short help message and options list. Lines 187-201 create a `ConfigParser` object. In this statement, all of ObsPyLoad's unconditional default values need to be defined. The program has been designed with the intention to not force the user to provide any option, so there are lots of default values. If the user wants to override some of the default values, he can set up a config

\(^{10}\)The `quit` flag is shared among both threads. To avoid possible problems when both threads would try to read or write the variable at the same time, it is accessed using the `threading.Lock()` instance `lock`, see line 121.
file at "/obspyloadrc".

In this implementation, ConfigParser is used together with OptionParser. It is therefore necessary that the variable names of the default values in the ConfigParser object match the ones given to OptionParser in the following lines.

In line 208, an OptionParser instance is created, whereas lines 216-329 add options to the parser instance. First, a short option that may only be one character long is given, then a long option that should be easier to understand. The action is either "store_true", "store_false" or "store". The first two of these store the respective boolean values. Options defined like this are switch-like options that do not take more information. If more information should be taken, "store" is the desired action. This will store the string that comes after the respective option in the command line call.

The boolean values and strings are stored inside options.dest, where dest is the string given in the parser.addoption statement. The help messages given here will be shown if the user calls

```
$ obspyload.py -h
```

The default values from ConfigParser will then be stored in the dictionary config.options (see line 333). Since this is a dictionary of strings, if one wants to use these variables directly it is necessary to override the default values with their respective correct types with the config.get* methods (see lines following 338), see the ConfigParser documentation\(^\text{11}\) for possible types.

After the ConfigParser defaults have been fed to the parser instance, in line 350 the command line options will be parsed to the tuple (options, args). Inside args, all unrecognized arguments will be stored, whereas inside options all options defined above will be stored and can be accessed via options.varname.

Variable splitting and sanity check section

Lines 374-523 take care of variable splitting. The first statement in this section just checks if the user typed -H to show the long help. This shows nicely how elegant Python lets you check those boolean options, there is no comparison operator needed, just a single if check. This pattern can also check if a variable exists in the namespace altogether and will appear numerous times throughout the code.

Since a lot of options, like the plot option

```
-I 800x600x3/60
```

or the station id code option

```
-i net.sta.loc.cha
```

include several distinct pieces of information in a single variable, it is necessary to split those variables to extract the values of interest. In the authors opinion, it also makes sense to perform sanity checks\(^\text{12}\) at this time since we can carry out some straightforward checks with a quick try-except statement around the split operation.

After checking if the options.model variable contains the name of a supported velocity model, the indented lines 870-451 try to split the options if the options.plot variable exists, that is if the user has given the -l option. The two main cases handled by the engulfing try-except statement are whether the user gave a trailing timespan (delimited by a slash) or not. Inside these, the rest of the information gets split up and converted to the correct types. If this fails, an error message will be printed.

Another task that gets performed inside this section is to check for special options like -a all, which selects all phases for plotting of theoretical arrival times, and then store the corresponding information inside the right variable.

The whole section is (unavoidably) quite long, but not really complicated. Like in the example above, what it always comes down to is, try to split the variable by a single character like a slash or a comma and store the individual values inside their distinct variables.

\(^{11}\)http://docs.python.org/library/configparser.html

\(^{12}\)Checking if the given as well as split up options are reasonable, e.g. have the correct type or are in a desired range, in order to make sure that the program will not raise an exception later, possibly somewhere halfway through the download after several hours of runtime.
Check for special tasks

Some special tasks like branching off to another routine when the user wants to download metadata or enter exception mode are handled inside this section, which begins in line 523. As ObsPyLoad uses the same command line parameter for changing the working directory for both the metadata as well as the data download mode, but different default directories, the if-statement following line 527 detects if the path has been left to its default value and changes the name to obspyload-metadata in metadata mode. If the -R option has been entered at the command line, the script tries to delete the datapath. In case the -u switch has been given, only the saved inventory, availability and event list files are deleted. In lines 544-568, if the user desires to download metadata or enter exception mode, one of the two functions queryMeta() and exceptionMode() will be called. They are discussed in full detail in Section 3.5.4. In either case, after these functions return, the main function will end here. Lines 568-603 will print a message containing various information, such as what will happen when the program continues and how to obtain help, is printed. The reason for this is that ObsPyLoad does not have any mandatory options and if a first-time user would just call obspyload.py without any further options, he would probably expect a help message.

Data download routine section

If the program did not branch off to the metadata mode or the exception mode, it progresses through the data download routine starting in line 603. Like in the branched off modes, the first thing that happens is starting the keypress thread. The done flag tells the keypress thread when the main thread is done so it will also quit. See Section 3.5.2 for a detailed description of the keypress thread. Inside the data download routine, for clarification, the general steps have been pointed out and numbered in the source code comments. As stated above, several tasks have been evacuated into functions and will only be discussed briefly here. For a detailed description of these functions, see their respective section inside this thesis.

Step (1) (lines 610-620) passes the necessary options to find matching events to the function get_events (see Section 3.5.4). This function returns a list of dictionaries describing individual events, which is stored inside the variable events.

Step (2) (lines 620-631) starts with a statement that shows up often throughout the code between individual downloading steps:

    check_quit()

This calls the function check_quit() which will end the program at this point if the user has pressed “q”. Then, the function get_inventory(), which will be described in full detail in section 3.5.4, is called. It returns a list of tuples of stations from ArcLink in the form

    [('net1.sta1.cha1.loc1',lat1,lon1), ('net2.sta2.cha2.loc2',lat2,lon2), ...]

which will be stored in arclink_stations.

Step (3) (lines 631-638) calls the function getnparse_availability, which will be described in section 3.5.4. The returned list of the form

    [(net1,sta1,cha1,loc1,lat1,lon1), (net2,sta2,cha2,loc2,lat2,lon2), ...]

is stored inside avail.

Step (4) (lines 638-660) will create and write to the event catalog file. The program first tries to open the catalog file in read and write mode, that way if ObsPyLoad runs on an existing directory in order to continue downloading, the event catalog file will not be overwritten, but newly downloaded events will just be added to the end of the file.
The basic code idea works like this, in order to avoid the necessity to handle the case of an existing catalog file separately:

```python
headline = '...'  
try:
    catalogfout = open(catalogfp, 'r+t')
except:
    catalogfout = open(catalogfp, 'wt')
catalogfout.write(headline)
catalogfout.seek(0, 2)
```

That way, no matter if `catalogfout` has been successfully opened in read and write mode or been created as a new file, the headline will be written (again). After that, the file handler jumps (seeks) to the last character of the file and thereby adds new entries to the end.

Before **Step (5)** (line 660) enters the event loop, the path to the *exception file* is created, in which all exceptions encountered during the data download are logged. If *ObsPyLoad* runs on an existing path in plain data download mode, the desired behaviour is to skip those event/station combinations where an exception has been encountered last time. Like for the event catalog file, it is also necessary to add new exceptions to the end of the file without overwriting former exceptions. This is achieved by a similar code structure as seen above. First, if possible, the exception file is opened in read and write mode. If successful, the whole file is read into the string `exceptionstr`. After that, the exception file handler (`exceptionfout`) jumps back to the beginning of the file. Again, it is now possible to use the same code for the cases of an existing exception file and a new one. Inside the event loop, the necessary information for theoretical travel time calculation using `obspy.taup` is extracted from the `eventdict` (lines 693-703). This is done here, since it's not necessary to do it inside every station loop iteration. The string `infoline`, containing all available information about the event, is put together. It will be added to the event catalog and quake catalog (station catalog) when the quality control information has been obtained.

After creating the event directory, if not present already, the *QuakeML* (see Section 3.4) is downloaded using `obspy.neries.Client.getEventDetail()` (line 724). This method returns the XML as a single string, which is saved using the file handler method `.write`

Data quality is not only calculated for each individual station, but also summed up over all stations for each event. To achieve this, the timing quality list `tqlist` (containing a list of all minimum entries of the `obspy.mseed.libmseed.LibMSEED.getTimingQuality` method) and the data quality sum `dqsum` (containing a sum of all MiniSEED data quality flags) are initialized following line 734. Both will be completed throughout the station loops. If the `-I` option has been passed to *ObsPyLoad*, the *NumPy* array `stmatrix` is initialized in line 759. This array holds the plotting matrix to which the individual station waveforms will be added to as columns inside the station loop. The height is the given plotting height +1, since the [0] entry of each column is used to count the number of traces that have been added to that column. This is needed to normalize the matrix later.

**Step (5.1)** (lines 761-946) begins the ArcLink station loop, inside which waveform data from all selected stations from the ArcLink webservice\(^{13}\) will be downloaded. Inside each loop, after checking if it should quit, the program creates the data file pointer by bringing together the event directory, the station name and the file ending `.mseed` (line 779). The following statement

```python
if os.path.isfile(datafout):
    print 'Data file for event exists (...)'  
    continue
```

checks if there is an already existing file at the file pointers location. If that is the case, the download is skipped and the station loop continues to the next iteration. Following this, lines 788-793 construct the string `skipstring`, which is unique for each `event/data provider/station`\(^{13}\) can be downloaded.

\(^{13}\)Since ArcLink supports routing, all stations that are taking part in the project to create an European Integrated Data Center (EIDAC) can be downloaded.
combination. If it appears in the exceptionstr mentioned above, an exception for this combination has previously been encountered and the download will be skipped.

Before downloading the actual waveform data, the theoretical arrival time is calculated. First, the great circle distance between the event and the station is calculated using *taup.locations2degrees*, which is an implementation of a special case of the Vincenty formula\(^\text{14}\). After using *taup.getTravelTimes* to obtain a list of dictionaries of all possible phases, the earliest arrival time is stored inside *arrivaltime* (line 810). The statement

\[
\begin{align*}
\text{starttime} &= \text{eventtime} + \text{arrivaltime} - \text{options.preset} \\
\text{endtime} &= \text{eventtime} + \text{arrivaltime} + \text{options.offset}
\end{align*}
\]

obviously calculates the time frame for the data download based on the preset and offset chosen by the user.

After reinitializing the *ObsPy* ArcLink client, the waveform data is saved using the client’s method *saveWaveform*. The data download is embedded in a try-except statement. In case of an error, the corresponding information is written to the exception file.

Lines 838-866 take care of the quality control information. First, the file handler *datafout* is used to add to *dqsum* and *tqlist*.

Then, the data file downloaded just before is read into a *ObsPy* stream object and its internal method *getGaps()* is used to obtain gaps and overlaps which are subsequently written to the *station catalog* (line 866).

The final task that is handled inside the station loop, the statement starting with line 870

\[
\begin{align*}
\text{if options.plt:} \\
\quad (\ldots) \\
\text{if options.fill:} \\
\quad (\ldots) \\
\text{else:}
\end{align*}
\]

is adding the data to *stmatrix* if the user gave the -I option and the variable *options.plt* exists. Depending on the user’s choice, the program will either download new data to fill the whole plot or use the existing file. In case the user gave the -F option to fill the whole plot, lines 873-887 are executed. The *ArcLink* client’s method *getWaveform* downloads the complete stream from *eventtime* until *eventtime+timespan*, where *timespan* is the user-configurable timespan of the plot.

If the user did not choose to give the -F option, the *Trace* is trimmed using it’s internal *.trim* method\(^\text{15}\). In the following (lines 900-944), the *Trace* is normalized and scaled to the correct length to be added as one column to *stmatrix*. This is done beforehand with the aid of *SciPy*, since this is a lot faster than letting the plotting function handle the scaling. The method *scipy.ndimage.interpolation.zoom* can take the array and scale it according to a given ratio. After rounding with *numpy.around*, the result is stored in the array *pixelcol*. The next problem is to find the column of *stmatrix* that matches the distance between the event and the location. Since the whole plot features 180 degrees on the horizontal axis, the *x_coord* can be calculated by

\[
x_{\text{coord}} = \text{int}((\text{distance} / 180.0) \times \text{pltWidth})
\]

where *pltWidth* is the width of the whole plotting matrix. Since *ObsPyLoad* provides an option to plot individual stations columns broader than one single pixel, the *x_coord* is floored down to the next multiple of the station column width using the *modulus operator* to obtain and subtract the amount of *x_coord* exceeding an integral multiple of *colWidth*, see line 919.

Following this, *pixelcol* is added to *stmatrix*. The code following line 936,

\[
\begin{align*}
\text{stmatrix}[:, x_{\text{coord}}:x_{\text{coord}} + \text{colWidth}] &= \text{\_\_} \\
\text{np.vstack([}\text{pixelcol}] * \text{colWidth}).\text{transpose()} \\
\end{align*}
\]


\(^\text{15}\)If used with the options *pad=True, fill_value=0*, this method can crop a trace outside the original timespan and will fill the missing values with zeros.
adds `pixelcol` to one or more columns in `stmatrix`, depending on `colWidth`. After ending this iteration of the station loop, the program continues to the next station.

The proceeding of the IRIS station loop (Step (5.2), see line 946) is by and large analogous to Step (5.1), therefore only the differences will be discussed here\(^\text{16}\).

One difference obviously lies in addressing the IRIS webservice instead of ArcLink, but this merely means using the `obspy.iris` client instead, which has the same options and behaves identical for the intended usage. Another difference is the info line for the station catalog file, which naturally features 'IRIS' instead of 'ArcLink' in the data provider column.

### 3.5.4 Data service functions

This section (lines 1177-1433) contains three functions that try to open existing files, access the different web-services and if necessary parse or filter the result.

**NERIES event webservice - Function `get_events()`**

The NERIES event service can be used with the `obspy.neries.Client`. Before the function `get_events()` does this, it tries to open an existing result that might have been saved during previous runs on the same datapath as is described below.

If no previous result is found, the `obspy.neries.Client` is initialized and the options are simply passed to its `.getEvents` method (following line 1229). The results are then dumped to a file using the `pickle` module\(^\text{17}\), which allows to convert a Python object hierarchy to a byte stream (Python pickle module documentation, 2011). If `ObsPyLoad` runs on the same datapath again, this file can be opened and the download can be skipped.

**ArcLink inventory webservice - Function `get_inventory()`**

Like the function `get_events`, this function first tries to open an existing file in case `ObsPyLoad` is run to resume a previously interrupted download.

Since the ArcLink webservice does not support wildcard searches for the network other than “*”, this function adds this capability. It first checks whether an advanced wildcard-type search is present and sets the `nwcheck` flag accordingly (line 1291). If an advanced wildcard-type search is given, the program sets `nw2="*"` to first download all possible networks from ArcLink. After the inventory has been requested from ArcLink using the clients method `.getInventory`, they are saved inside `inventory`. This method returns a dictionary in which individual channels (specifying everything: `net.sta.loc.cha`) as well as whole networks (`net`) and stations (`net.sta`) are given on the same level. It is therefore necessary to filter this dictionary, which is done with

\[
\text{stations} = \text{sorted}([i \text{ for } i \text{ in } \text{inventory}.\text{keys()} \text{ if } i.\text{count}('\.' \text{ == } 3)])
\]

Inside `stations`, there now are all individual stations saved as a list of strings. Since it might be necessary to filter those according to a possible wildcard search, the code following line 1318 uses the `fnmatch.translate` method to translate the “A*B?C”-type wildcard search to an equivalent regular expression. Using the `re` module, this regular expression (`regex`) is now being compared to every network entry inside `stations`. If the `regex` matches a network, the corresponding station passes through this filter. Before returning and saving the final result, so this operation can be skipped when possibly resuming the download, the latitude and longitude for each station is obtained from the `inventory` variable - this is needed for the `obspy.taup` theoretical travel time calculation.

**IRIS availability webservice - Function `getnparse_availability()`**

Since the IRIS availability webservice supports wildcard searches for every field, including network, it is not necessary to handle this inside the function `getnparse_availability`. After trying to open an existing file from previous runs (line 1374), the program simply passes all necessary parameters to the IRIS client provided by

\(^{16}\text{There is some potential to visually compact the code at this point by merging the ArcLink and IRIS station lists (and adding a data provider information to each entry) beforehand and handling both in a single loop. This might be a little bit slower than the current implementation due to the merging of the station lists needed, though this effect would probably not be significant.}\)

\(^{17}\text{http://docs.python.org/library/pickle.html}\)
3.5. DESCRIPTION OF THE OBSPYLOAD SOURCE CODE

**ObsPy.** At this time, the IRIS client can not return the result as a list, only as a string in two different output formats. The output parameter is set to XML, since this is more easily parsed than the output option `bulk`. After the XML has been fed into an `lxml.etree` object, its method `findall` is used to find all “Station” entries in the XML. The loops (lines 1401-1422)

```python
avail_list = []
stations = availxml.findall('Station')
for station in stations:
    net = station.values()[0]
    sta = station.values()[1]
    lat = float(station.find('Lat').text)
    lon = float(station.find('Lon').text)
    channels = station.findall('Channel')
    for channel in channels:
        loc = channel.values()[1]
        cha = channel.values()[0]
        avail_list.append((net, sta, loc, cha, lat, lon))  # (simplified)
```

append each individual channel together with it’s latitude and longitude position to `avail_list`. This is the final result which will be returned and saved with the `pickle` module.

3.5.5 Alternative modes functions

This section occupies lines 1433-1609.

**Metadata download mode - function queryMeta()**

This function downloads response and dataless seed files if the user gave the `-q` option. It uses the functions `getnparse_availability()` and `get_inventory()`. Response files are downloaded from IRIS, dataless seed files are downloaded from ArcLink.

After the keypress-thread has been started and both clients have been initialized, the availability information from IRIS is requested (with `getnparse_availability()`, see section 3.5.4). The loop following line 1460 iterates through tuples of the form `(net, sta, loc, cha, lat, lon)`, where each tuple defines an individual channel together with its latitude and longitude. At the beginning of this loop, the function `check_quit()` is invoked to check if the program should quit at this point. The file pointer `respfull` is assembled from the full datapath and the entries of the tuple, which are brought together using the string method `join`. The resp file is then downloaded and saved to this file pointer with the IRIS clients method `saveResponse` (see line 1479).

The subsequent loop (lines 1489-1522) works analogous to the IRIS loop, with the difference that the ArcLink client provided by ObsPy is used to save Dataless SEED files. Its method `saveResponse` takes a file location as well as a network, station, location, channel, starttime, endtime and an output format (which currently only supports “SEED”).

After both loops finished downloading the metadata, the `done` flag is set to `True` to tell the `keypress_thread` that it should quit. Then the function returns and ObsPyLoad quits.

**Exception file mode - function exceptionMode()**

This function, which will run if the user gave the `-E` option, first initializes both clients. It then reads the file `exceptions.txt` in the datapath into the list `exceptions` with the string method `readlines()`, which returns a list of strings, where each entry corresponds to one line in the file. See this sample exception file:

```
event_id;data provider;station;starttime;endtime;exception
##########################################################
20110311;ArcLink;IU.ANTO.00.BHE;2011-03-11;2011-03-11;No data available
```

---

18See Section 3.5.4. The `pickle` module allows to convert a Python object hierarchy to a byte stream (Python pickle module documentation, 2011). See [http://docs.python.org/library/pickle.html](http://docs.python.org/library/pickle.html).

19The entries in the event_id, starttime, endtime and exception columns have been truncated in order to fit on the width of this paper.
To avoid losing the original exceptions if the user interrupts the exception mode, the original exceptions.txt is not overwritten until the exception mode is done. Meanwhile, new exceptions will be added to the string further_exceptions, which will be used to overwrite the exceptions file in line 1605.

The loop iterating the exceptions list starts at the third entry, since the first three lines of exceptions.txt are only headlines, as can be seen above. It then splits each entry at “;”, since exceptions.txt is a CSV-file\(^\text{20}\). The split-up values are stored inside exsplit.

Then, from line 1552 until line 1603, the script checks whether the exception was only caused by no available data, or for instance by something like a timeout, as can be seen in the example file above. If so, the list entries inside exsplit are simply assigned to more readily comprehensible variable names and the file pointer for the download is put together. Then ObsPyLoad will use the exsplit[1] entry to download from the correct data provider. If an exception occurs again, it will be added to further_exceptions, which will finally be written to the file handler exceptionfout.

### 3.5.6 Additional functions

This section starts with line 1609.

**Function travelTimePlot()**

This function is a slightly modified version of taup.travelTimePlot from the obspy.taup module. It is used by the main function (see section 3.5.3) to plot the theoretical arrival times over the real data. It takes the number of points to plot (npoints), the phases to plot, the depth of the event, the timespan as well as the plotting width and height as parameters.

First, the dictionary data is created with all phase names as keys. The program then creates creates degrees, a NumPy array with npoints evenly spaced points from 0 to 180 which represent all the degrees where the arrival times will be calculated (see line 1634). This number is set by the main function according to the width of the plot chosen by the user in order to create a point plot with sufficiently densely spaced points.

While looping over all degree entries in degrees and all phases in phases, the corresponding distance and arrival time points are added to data. In the following loop, those points are added to the ObsPyLoad data plot. To achieve this, the corresponding coordinates in stmatrix have to be calculated. This is done by using Python’s built-in function map to divide all entries of the value list by 180 and multiply them by pltWidth (lines 1655-1661):

\[
\begin{align*}
\text{x_coord} &= \text{map}(\text{operator.div}, \text{value}[0], [180.0 / \text{pltWidth}] * \text{len(value}[0])] \\
\text{y_coord} &= \text{map}(\text{operator.div}, \text{value}[1], [\text{timespan} / \text{pltHeight}] * \text{len(value}[1])] 
\end{align*}
\]

The first argument of map is the operation needed (division in this case), the second and third are the numerator and the denominator. The denominator is a list multiplied by the length of the value list, because the map function needs two equally long sequences.

Then, the values are plotted with Matplotlib (matplotlib.sourceforge.net) as points over the data.

**Functions getFolderSize(), printWrap() and help()**

Since it seems unnecessary to discuss these merely auxiliary functions at great lengths, they will only be mentioned briefly here.

- **getFolderSize()** is used to calculate the size of the obspyload-data folder after downloading data to show some information to the user.

- **help()** prints the long help if the user gave the -H option.

- **printWrap()**, which takes two strings as mandatory options, is used to format the output of help() into two columns.

\(^{20}\)It is a Comma-separated values file (CSV) file with semicolons as delimiters. It may therefore also be called SSV-file.
Chapter 4

ObsPyLoad handbook

This chapter aims to provide a comprehensive manual for ObsPyLoad. The content of this chapter partially relies on the output of one of the two exhaustive help function written for ObsPyLoad. Additionally, more information and examples will be given here.

4.1 Usage

ObsPyLoad is meant to be used from a shell and can be called without the explicit utilization of the Python interpreter. The program tries to be intuitively usable, but as seems unavoidable with shell programs that reach a certain complexity, the list of options may be a little overwhelming at first glance.

Getting started with a command-line only tool may take a little more time for less shell-inclined folks, but there are definitely strong advantages over a GUI-only tool\(^1\). For example, tasks like automatically checking for new events, batch processing or remote server-side usage should be feasible with standard shell tools. The author assumes that most seismologists are into shell usage, anyway.

One design inclination has been to not force the user to provide mandatory options. That means that just typing \texttt{obspyload.py} is sufficient to download data. Most users probably will not do this more than once, but it seems unavoidable to tell first-time users what the program will do before entering the download procedure, so when no options are given, the program prints this clarifying message:

```
chris@gauss:~$ obspyload.py

Welcome, you provided no options, using all default values will
download every event that occurred in the last 3 months
with magnitude > 3 from every available station.

ObsPyLoad will now create the folder /home/chris/obspyload-data
and possibly download vast amounts of data. Continue?
Note: you can suppress this message with -f or --force
Brief help: obspyload.py -h
Long help: obspyload.py -H
[y/N]>
```

The prompt asks the user to enter “y” or “n”. As is common practice, the capital N indicates that answering “No” is the default. As is stated in the message, this text can be suppressed with \texttt{-f}, which is what most people probably want. As also written in the message, answering \texttt{yes} would result in using all default values for the data download.

\(^1\)This does not mean that ObsPyLoad would not benefit from a core and GUI-client architecture, especially if the capabilities keep growing. See section 5.2 for some ideas regarding this.
It will be interesting to the reader to know the default behaviour and how to change some or all aspects of it. Here is a list of all default values:

- the data download mode is default, alternative modes like the metadata download mode or the exception file mode must be entered explicitly
- the default datapath in which the data will be saved is `obspyload-data` in the current working directory. In metadata download mode, the default path is `obspyload-metadata`.
- the default starttime is three months ago, the default end time is now.
- the default velocity model for theoretical travel time calculation is `iasp91`.
- the default preset is 5 minutes, which means that data from each event/station combination will be downloaded starting 5 minutes before estimated arrival time.
- the default offset is 80 minutes, which means that data from each event/station combination will be downloaded until 80 minutes after estimated arrival time.
- the default minimum magnitude is 3.
- by default, there is no geographical restriction, events from the whole globe will be downloaded.
- by default, there is no network/station restriction. All available stations will be downloaded, including temporary ones.
- by default, no plot will be created. If the plot will be created, the default internal resolution will be 1200x800x1 and the default timespan for the plot will be 100 minutes. By default, the plot will not be filled with more data than is downloaded anyway.
- by default, if the plot is created, the ‘P’ and ‘S’ phases will be plotted on top.

The `ObsPyLoad` command line tool uses a syntax which has been, to some degree, influenced by the Generic Mapping Tools (GMT) (http://gmt.soest.hawaii.edu). Most options can be given in two different flavours. There usually is one option combining related values as one option divided by some delimiter like a slash or a dot, and another set of options achieving the same thing separately. It makes sense to have a look at the custom written long help function of `ObsPyLoad` at this time, since this will clarify the concept:

4.1.1 Long help function

Special effort has been put into the two help functions. This one will be printed if the user types

```
$ obspyload.py -H
```

```
ObsPyLoad: ObsPy Seismic Data Download tool.
============================================
The CLI allows for different flavors of usage, in short:
-----------------------------------------------
   e.g.: obspyload.py -r <west>/<east>/<south>/<north> -t
          <start>/<end> -m <min_mag> -M <max_mag> -i <nw>.<st>.<l>.<ch>
   e.g.: obspyload.py -y <min_lon> -Y <max_lon> -x <min_lat> -X
          <max_lat> -s <start> -e <end> -P <datapath> -o <offset>
          --reset --f
```
You may (no mandatory options):
-------------------------------------

* specify a geographical rectangle:
  Default: no constraints.
  Format: +/- 90 decimal degrees for latitudinal limits,
  +/- 180 decimal degrees for longitudinal limits.
  -r[--rect] <min.longitude>/<max.longitude>/<min.latitude>/<max.latitude>
  e.g.: -r -15.5/40/30.8/50
  -x[--lonmin] <min.latitude>
  -X[--lonmax] <max.longitude>
  -y[--latmin] <min.latitude>
  -Y[--latmax] <max.latitude>
  e.g.: -x -15.5 -X 40 -y 30.8 -Y 50

* specify a timeframe:
  Default: the last 3 months
  Format: Any obspy.core.UTCDateTime recognizable string.
  -t[--time] <start>/<end>
  e.g.: -t 2007-12-31/2011-01-31
  -s[--start] <starttime>
  -e[--end] <endtime>
  e.g.: -s 2007-12-31 -e 2011-01-31

* specify a minimum and maximum magnitude:
  Default: minimum magnitude 3, no maximum magnitude.
  Format: Integer or decimal.
  -m[--magmin] <min.magnitude>
  -M[--magmax] <max.magnitude>
  e.g.: -m 4.2 -M 9

* specify a station restriction:
  Default: no constraints.
  Format: Any station code, may include wildcards.
  -i[--identity] <nw>.<st>.<l>.<ch>
  e.g. -i IU.ANMO.00.BH* or -i *.*.?0.BHZ
  -N[--network] <network>
  -S[--station] <station>
  -L[--location] <location>
  -C[--channel] <channel>
  e.g. -N IU -S ANMO -L 00 -C BH*

* specify plotting options:
4.1. USAGE

CHAPTER 4. OBSPYLOAD HANDBOOK

Default: no plot. If the plot will be created with -I d (or -I default), the defaults are 1200x800x1/100 and the default phases to plot are ‘P’ and ‘S’.

-I[--plot] <pxHeight>x<pxWidth>x<colWidth>[/<timespan>]
For each event, create one plot with the data from all stations together with theoretical arrival times. You may provide the internal plotting resolution: e.g. -I 900x600x5. This gives you a resolution of 900x600, and 5 units broad station columns. If -I d, or -I default, the default of 1200x800x1 will be used. If this command line parameter is not passed to ObsPyLoad at all, no plots will be created. You may additionally specify the timespan of the plot after event origin time in minutes: e.g. for timespan lasting 30 minutes: -I 1200x800x1/30 (or -I d/30). The default timespan is 100 minutes. The final output file will be in pdf format.

-F[--fill-plot]
When creating the plot, download all the data needed to fill the rectangular area of the plot. Note: depending on your options, this will approximately double the data download volume (but you’ll end up with nicer plots ;-)).

-a[--phases] <phase1>,<phase2>,...
Specify phases for which the theoretical arrival times should be plotted on top if creating the data plot (see above, -I option). Default: -a P,S. To plot all available phases, use -a all. If you just want to plot the data and no phases, use -a none.
Available phases:
P, P’P’ab, P’P’bc, P’P’df, PKKPab, PKKPbc, PKKPdK, PKKSab, PKKSbc, PKKSdf, PKPab, PKPbc, PKPdf, PKPdf, PKSab, PKSbc, PKSdf, PKIKP, PP, PS, Pcs, Pcs, Pdiff, Pn, PnPs, PnS, S, S’S’ac, S’S’df, SKKPab, SKKPbc, SKKPdf, SKKSac, SKKSdf, SKPab, SKPbc, SKPdf, SKSac, SKSdf, SKiKP, SP, SPg, SPn, SPn, SS, Sc, Scs, Scdf, Sd, Sn, SnS, P, PKPab, PKPbc, PKPdf, PKPdf, PKPdiff, PKIPKP, Pn, Pn, Ps, PKSac, PKSdf, PDiff, SP, SPg, SPn, SPn, SS, SS, SSKSac, SSKSac, SSKSac, SSSdf, SSDiff, SSDiff, S
Note: if you select phases with ticks(‘) in the phase name, don’t forget to use quotes (-a "phase1’,phase2") to avoid unintended behaviour.

* specify additional options:

-n[--no-temporary]
Instead of downloading both temporary and permanent networks (default), download only permanent ones.

-p[--preset] <preset>
Time parameter given in seconds which determines how close the data will be cropped before estimated arrival time at each individual station. Default: 5 minutes.
4.1. USAGE

CHAPTER 4. OBSPYLOAD HANDBOOK

-o[--offset] <offset>
Time parameter given in seconds which determines how close
the data will be cropped after estimated arrival time at each
individual station. Default: 80 minutes.

-q[--query-resp]
Instead of downloading seismic data, download instrument
response files.

-P[--datapath] <datapath>
Specify a different datapath, do not use do default one.

-R[--reset]
If the datapath is found, do not resume previous downloads as
is the default behaviour, but redownload everything. Same as
deleting the datapath before running ObsPyLoad.

-u[--update]
Update the event database if ObsPyLoad runs on the same
directory for a second time.

-f[--force]
Skip working directory warning (auto-confirm folder
creation).

Type obspyload.py -h for a list of all long and short options.

Examples:
---------
Alps region, obspyload.py -r 5/16.5/45.75/48 -t
minimum magnitude of 4.2:
2007-01-13T08:24:00/2011-02-25T22:41:00 -m 4.2

Sumatra region, obspyload.py -r 90/108/-7/7 -t "2004-12-24
Christmas 2004,
different timestring,
mind the quotation marks:
01:23:45/2004-12-26 12:34:56" -m 9

Mount Hochstaufen area (Ger/Aus),
default minimum magnitude:
obspyload.py -r 12.8/12.9/47.72/47.77 -t
2001-01-01/2011-02-28

Only one station, to quickly try out the plot:
obspyload.py -s 2011-03-01 -m 9 -I 400x300x3 -f -i IU.YSS.*.*
4.1. USAGE

CHAPTER 4. OBSPYLOAD HANDBOOK

ArcLink obspyload.py -N B? -S FURT -f

Network wildcard search:

Downloading obspyload.py -q -f -P metacatalog metadata from all available stations to folder "metacatalog":

Download obspyload.py -E -P thisOrderHadExceptions -f stations that failed last time (not necessary to re-enter the event/station restrictions):

4.1.2 Specifying a geographical rectangle

As can be seen above, to clearly lay out different ways of usage, this help function is structured thematically. For example, if the user wants to have a look at all events that happened (roughly) in the Japan region, these two commands are equivalent:

$ obspyload.py -x 125 -X 150 -y 30 -Y 48
$ obspyload.py -r 125/150/30/48

4.1.3 Specifying a time frame

If the user wants to download all events that occurred in the time from February 6th to May 20th 2011, he can achieve it in these two ways:

$ obspyload.py -s 2011-02-06 -e 2011-05-20
$ obspyload.py -t 2011-02-06/2011-05-20

There are many more ways to provide a time string. As stated in the help function printed above, any obspy.core.UTCDateTime recognizable string can be given. The interested user may have a look at the documentation of this module, since it is not appropriate to echo all of its examples in this thesis. Here is just an incomplete list of selected ways of usage:

- ISO8601 string, calendar date: -s 2009-12-31T12:23:34.5 -e 2009-12-31T12:23:34+01:15
- ISO8601, ordinal date, two ways: -s 2009-365T12:23:34.5 -e 2009-365T12:23:34.5
- ISO8601, week date: -s 2009-W53-7T12:23:34.5

Since the default end time is always now, to download as many events as possible it is sufficient to change the start time from the default value of 3 months ago further into the past, e.g. by specifying -s 1970-001.
4.1.4 Further options

As can be seen in section 4.1.1, there are numerous other options available, adding capabilities to restrict event magnitudes \((-m\) (minimum magnitude) and \(-M\) (maximum magnitude)), to restrict stations and networks \((-i, -N, -S, -L, -C\)) and to add a plot of all station data and theoretical arrival time for each event\((-I, -F, -a\)).

4.1.5 Combining options

Options and restrictions can be combined in any arbitrary way. For instance, if the user wants to download all events that occurred in the Japan region in the time from February 6th to May 20th 2011, he can use one of these commands:

```bash
$ obspyload.py -s 2011-02-06 -e 2011-05-20 -x 125 -X 150 -y 30 -Y 48
$ obspyload.py -r 125/150/30/48 -t 2011-02-06/2011-05-20
```

Figure 4.1 has been created to provide an optical guideline to the process of quickly finding a complete ObsPyLoad command without forgetting any options for a particular task. It should be useful to both regular and casual users of this tool.

4.1.6 A listing of all possible options: OptionParser help function

A list of all possible options, including customized help texts, is generated by the `OptionParser` module and can be accessed via

```bash
$ obspyload.py -h
```

Since the output of this command overlaps with the custom written long help function (see Section 4.1.1), it is not necessary to include it in this chapter. It can be found in the appendix of this thesis (see Appendix B).

4.2 Output

4.2.1 Shell output

Data download mode

When entering the data download procedure, the output of the script to the shell looks similar to the one seen below. The structure varies a bit depending on whether the plot option is used and whether extra data is downloaded to fill the plot area.

```
chriss@gauss:/data$ obspyload.py -f -m 9 -i BW.WETR.*.BH* -I d
Keypress capture thread initialized...
Press ‘q’ at any time to finish the file in progress and quit.
Downloading NERIES eventlist... done.
Received 1 event(s) from NERIES.
Downloading ArcLink inventory data... done.
Received 3 channel(s) from ArcLink.
Downloading IRIS availability data...
IRIS returned to matching stations.
Downloading quakeml xml file for event 20110311_0000010... done.
Download event 20110311_0000010 from ArcLink BW.WETR..BHE... done.
Scaling data for station plot... done.
Download event 20110311_0000010 from ArcLink BW.WETR..BHN... done.
Scaling data for station plot... done.
Download event 20110311_0000010 from ArcLink BW.WETR..BHZ... done.
Scaling data for station plot... done.
```
Figure 4.1: obspyload.py command line parameters helper diagram
In this example, due to restricting to only one station and setting the minimum magnitude to 9, the number of events was (fortunately) limited to one and the runtime was very short.

**Metadata download mode**

The metadata download mode, which can be used with the `-q` command line option, will first use the availability webservice from IRIS, then the inventory webservice from ArcLink. After printing how many stations have been returned from each data provider, `resp` files are downloaded from IRIS and Dataless SEED files are downloaded from ArcLink.

```
chris@gauss:~/data$ obspyload.py -q -s 1970-001
ObsPyLoad will download resp and dataless seed instrument files and quit.

Keypress capture thread initialized...
Press 'q' at any time to finish the file in progress and quit.
```

```
Downloading IRIS availability data... done.
```

```
Parsing IRIS availability xml to obtain nw.st.lo.ch... done.
```

```
Received 10678 station(s) from IRIS.
```

```
Received 100622 channel(s) from IRIS.
```

```
Downloading ArcLink inventory data... done.
```

```
Received 10866 channel(s) from ArcLink.
```

```
Downloading Resp file for 3A.L002..HHE.resp from IRIS... done.
```

```
Downloading Resp file for 3A.L002..HHN.resp from IRIS... done.
```

```
Downloading Resp file for 3A.L002..HHZ.resp from IRIS... done.
```

```
(...)
```

```
Downloading dataless seed file for AI.ESPZ..BHN.seed from ArcLink... done.
```

```
Downloading dataless seed file for AI.ESPZ..BHZ.seed from ArcLink... done.
```

```
Downloading dataless seed file for AI.JUBA..BHE.seed from ArcLink... done.
```

```
(...)
```

**4.2.2 Folder and data structure**

**Data download mode**

When downloading waveform data, ObsPyLoad creates the data folder specified with the option `-P` (or creates the default folder `obspyload-data`). Inside this folder, event directories, the event catalog file `catalog.txt`, the exception file `exceptions.txt` and some temporary files can be found (see Figure 4.2).

The event catalog file `catalog.txt` contains a list of all events in plain text formatted as following:

```
event_id;author;flynn_region;latitude;longitude;depth;magnitude;magnitude_type;DataQuality;TimingQualityMin
```

```
20110409_0000021;CSEM;KYUSHU, JAPAN;30.023;131.764;-30.0;6.0;mw;0 (OK);100.00
20110411_0000023;CSEM;EASTERN HONSHU, JAPAN;36.998;140.452;-20.5;6.7;mw;0 (OK);100.00
20110411_0000078;CSEM;NEAR EAST COAST OF HONSHU, JAPAN;35.388;140.719;-5.0;6.4;mw;0 (OK);100.00
```

The last two columns of this file concern quality control (see Section 2.4). The column `Data Quality` uses the `obspy.mseed.libmseed.getDataQualityFlagsCount` method which counts all data quality flags of a MiniSEED file set by the digitizer (see the `libmseed` module documentation, `http://docs.obspy.org/packages/auto/obspy.mseed.libmseed.html`). The individual flags and MiniSEED files are not distinguished in this catalog, this number is merely a sum of all data quality flags of all MiniSEED data files for this event. Ideally it is

---

2The `catalog.txt` file as included here has been shortened by the columns `datetime` and `origin_id` to fit on this paper format at reasonable font size.
4.2. OUTPUT

CHAPTER 4. OBSPYLOAD HANDBOOK

Figure 4.2: File structure inside the data folder

zero, and (OK) will be written next to it. A large number in this field indicates quality problems in the data for this event, and FAIL will be written next to it.

The column **TimingQualityMin** is the lowest of all minima entries of the timing quality information in Blockette 1001 in all files for this event. It ranges from 0 to 100, with large values being better. If this number is unsatisfactory, since this is the minimum of all files, it might be worth the time to have a look at the quake.txt file inside the concerning event folder. In this file, all stations and their minimum timing quality are listed separately, so it is possible to track the bad data and act as necessary.

The file exceptions.txt, which can also be found in the top-level of the data structure, provides a log of errors that occurred while downloading the data. Those may be errors like “no data available”, which is fine, or connection problems like timeout errors, which is unfavorable.

If this file reveals an unpleasant amount of connection problems, it may be worth to try again to download the missing data with the exception file mode (-E). When continuing an interrupted download, this file is used to skip former event/station combinations that resulted in an exception.

As described in Section 3.5.4, the temporary files (availability.pickle, inventory.pickle, events.pickle) are files saved as byte stream and therefore of no immediate use outside the program.

Inside each event-folder, which is named according to the NERIES event id, all the data MiniSEED files for this event are saved. The file quake.txt first contains a header depicting once more the event info line as seen in the event catalog (see above). Following this, a catalog of all station channels from which data has been downloaded can be found (see the provided sample below). It also features the data provider and three columns of quality control information.

```
Station;Data Provider;TQ min;Gaps;Overlaps
IU.SFJD.10.BHZ;ArcLink;None;0;0
IU.AFI.00.BHZ;IRIS;100.0;0;0
IU.ANMO.00.BHZ;IRIS;80.0;0;0
IU.CASY.10.BHZ;IRIS;85.0;0;0
IU.CHTO.00.BH1;IRIS;0.0;1;0
IU.HKT.00.BHZ;IRIS;45.0;0;0
```
The column TQ min is the minimum timing quality as described above for the catalog.txt file, but for each station separately. If no timing quality is available for a particular station, None will be printed in that field. The column Gaps is the number of gaps found in the data, whereas Overlaps counts the number of overlaps occurring in the MiniSEED file.
For each event, the relevant QuakeML xml file (see Section 3.4) is saved as quakeml.xml.
If the program has been instructed to create a plot for each event, those will be contained inside the event folders as waveforms.pdf. Additionally, a stack of all events will be included in the top level data directory.

Metadata download mode

Unless specified otherwise, the metadata download mode creates the folder obspyload-metadata. Inside, besides some internal saved files, all downloaded dataless MiniSEED files and instrument response files are contained at top level (see Figure 4.3). Their corresponding file extensions are .mseed and .resp.

4.3 Examples

4.3.1 Strong Events around the 2011 Tohoku earthquake

Just before the development of ObsPyLoad, the 2011 Tohoku earthquake in Japan with a magnitude of 9.0 $M_w$ caused tremendous tragedy. Before and after this event, some strong earthquakes ($M_w > 7$) occurred in the same area. For these events, this example will download all broadband vertical-component data (BHZ) solely from stations for which the device is set to 00, both from the IRIS and ArcLink webservises.

```bash
```

The resulting folder tohoku2011 can be found on the accompanying CD inside the folder examples. The command as seen above produced plots with a time frame of 60 minutes, a station column width of 3 and plotted theoretical arrival times of the $P$, $S$, PP wave phases on top. The example of Figure 4.4 shows the devastating earthquake that occurred on March 11th, 2011.

4.3.2 Building up a metadata database

It might be an interest of many seismologists to build up a central metadata database. This can be achieved by

```bash
$ obspyload.py -q -P metadata_db -s 1970-001
```

If this download (currently over 100000 files) is interrupted by pressing "q", it can be resumed by running the same command again.
Running the same command again, supplemented by the -u (update) option, will add newly registered stations.
Figure 4.4: Stacked waveform data of 202 stations and theoretical arrival times for the $M_w 9.0$ earthquake off the coast of Japan on March 11th, 2011.

to the database. Unfortunately, it is not possible to update existing files inside a database with solely internal methods of ObsPyLoad. A possible stopgap measure might be to always add new database folders for recent time-periods, or to download the whole database again at regular intervals.

To provide an example, the author downloaded instrument response and dataless seed files from only the KBS station of the IU network (additional command line options -N IU -S KBS). The result can be found inside the examples folder on the accompanying CD.
Figure 4.5: Another example plot filling the whole plotting area (-F option) without theoretical arrival times (-a none option) for $18 M_w \geq 7.7$ events stacked (the stacked events plot is saved as file `allevents_waveforms.pdf` in the top level data directory). This is broadband vertical component data from solely the stations for which the device was set to 00. The plot can be found in PDF format in the examples folder of the accompanying CD. The corresponding data folder is about 750 megabytes large and therefore not included on the CD.
Chapter 5

Conclusion and Discussion

5.1 Advantages of using ObsPyLoad

Sophisticated automated data retrieval, quality control and processing is the way seismology needs to confront the data avalanche (Crotwell, 2007) in future. For the working seismologist, this approach enormous potential time-savings.

Using ObsPyLoad in particular has strong advantages over currently available alternatives. Once used to the syntax, it will be significantly more efficient to use than e-mail based data request tools like BREQ_FAST\(^1\) or NetDC\(^2\), as well as webinterface-based tools like WILBER\(^3\). Certainly, there are numerous applications for which SOD\(^4\) (Owens et al. (2004), see Section 1.2) is better suited, but ObsPyLoad on the other hand has the advantage of not limiting itself to IRIS FISSURES/DHI servers as well as having some additional features. It might therefore be seen as an attractive alternative to SOD. Being able to specify every necessary option directly from the shell without the need to create an XML file might also be more convenient to some users.

5.2 Problems, possible future improvements and additions

Although ObsPyLoad surpassed the initial goals set for this thesis, many possibilities of further improvements remain. For one thing, it will probably like all programs contain bugs. It might prove superior to stick to the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. [...]

For ObsPyLoad, this might mean holding back more plotting capabilities in the core script, whereas adding a second tool which works on a directory structure as created by obspyload.py and adds various plotting functionality like station plots, raypath and ray coverage maps, data and theoretical arrival time plots, and so forth. Additionally, further tools to complete the NDLB algorithm (see Figure 1.3) may be included.

The key task of ObsPyLoad is downloading seismic waveform data. In the lifetime of the script, it will therefore always be of high interest to support as many datacenters as possible.

A Graphical User Interface (GUI) may also increase the appeal to some users. Without much restructuring of the code, the command line scripts could then be controlled from the GUI. With a graphical client, a lot of new possibilities would arise. Ultimately, the GUI could lead the user through the necessary steps of the NDLB Algorithm (Figure 1.3). ObsPyLoad expressly would benefit from a GUI in a variety of ways. For example, the user could select event and station latitude restrictions interactively using a map showing events and stations. If a special geographical area would be of interest, he could use a ray-coverage plot and change his options until the coverage is sufficient.

\(^1\)http://www.iris.edu/manuals/breq_fast.htm
\(^2\)http://www.iris.edu/manuals/netdc/intro.htm
\(^3\)http://www.iris.edu/wilber
\(^4\)http://www.seis.sc.edu/SOD/

39
First, I would like to thank Dr. Karin Sigloch for assigning me this thesis and therefore making it possible in the first place, as well as always improving the project with important suggestions and ideas. A very large part of my gratitude goes to Dr. Robert Barsch, whose sincere and extensive support has enthused me throughout the whole project and from whom I have learned a lot. Lion Krischer together with Robert did an impressive job creating the obspy.taup (FORTRAN wrappers) package and is always a great inspiration. But primarily, I want to thank him for being a great friend. Thanks to Seyed Kasra Hosseini zad for contributing valuable input and ideas. I also want to thank my fellow students with whom I have had a great time since commencing my studies in 2007. I also want to thank those lecturers who have shown an honest interest in teaching the students and therefore improving this Bachelors Program.

Of course, I particularly would like to thank my parents and my whole family for always supporting me in many ways. I especially want to express my gratitude and love to Ela for being the best conceivable girlfriend.

My current and past friends are of high importance to me, thanks for being around.
Bibliography

IRIS Education and Outreach Program. Iris education and outreach program. [http://www.iris.edu/hq/programs/education_and_outreach], July 14th, 2011.


P.M. Shearer. *Introduction to seismology*. Cambridge Univ Pr, 1999.


Appendix A

Installation of ObsPyLoad

A.1 Dependencies

- Python (http://www.python.org)
- NumPy (http://numpy.scipy.org)
- SciPy (http://scipy.org)
- Matplotlib (http://matplotlib.sourceforge.net)
- lxml (http://lxml.de)
- ObsPy (http://www.obspy.org)

On most modern operating GNU/Linux operating systems, all of these dependencies except ObsPy may probably be installed with the package manager. If that is not favored, for most operating systems it should be simple to follow the following procedure.

- Download Python 2.6.x from http://www.python.org/download/. Uncompress the archive. For Windows users, an executable installer is provided.
- Run

        ./configure --prefix=$HOME
        make
        make install
        export PATH="$HOME/bin:$PATH"

- Download Easy Install from http://peak.telecommunity.com/dist/ez_setup.py
- Run

        python ez_setup.py

- Now use easy_install to install the required dependencies:

        easy_install numpy
        easy_install scipy
        easy_install matplotlib
        easy_install lxml

- Installing ObsPy can either also be done using easy_install:
A.2. OBSPYLOAD

APPENDIX A. INSTALLATION OF OBSPYLOAD

- Or using SVN (subversion), which retrieves the latest version:

```
svn checkout https://svn.obspy.org/trunk obspy
cd obspy/misc/scripts/
./develop.sh
```

A.2 ObsPyLoad

Finally, ObsPyLoad can either be retrieved from the supplemented CD, or using SVN:

```
svn checkout https://svn.obspy.org/branches/scheingraber obspyload
```

It may be convenient to use the tool without the full path, e.g. by creating a symlink

```
ln -s /path/to/obspyload.py /usr/local/bin/obspyload.py
```

or an alias, e.g. for bash:

```
echo "alias obspyload.py="/path/to/obspyload.py"" >> ~/.bashrc
```
Appendix B

OptionParser help message

Usage: obspyload.py [options]

Options:
- `h, --help` show this help message and exit
- `H, --more-help` Show explanatory help and exit.
- `q, --query-metadata` Instead of downloading seismic data, download metadata: resp instrument and dataless seed files.
- `P DATAPATH, --datapath=DATAPATH` The path where ObsPyLoad will store the data (default is ./ObsPyLoad-data for the data download mode and ./ObsPyLoad-metadata for metadata download mode).
- `u, --update` Update the event database when ObsPyLoad runs on the same directory a second time in order to continue data downloading.
- `R, --reset` If the datapath is found, do not resume previous downloads as is the default behaviour, but redownload everything. Same as deleting the datapath before running ObsPyLoad.
- `s START, --starttime=START` Start time. Default: 3 months ago.
- `e END, --endtime=END` End time. Default: now.
- `t TIME, --time=TIME` Start and End Time delimited by a slash.
- `v MODEL, --velocity-model=MODEL` Velocity model for arrival time calculation used to crop the data, either 'iasp91' or 'ak135'. Default: 'iasp91'.
- `p PRESET, --preset=PRESET` Time parameter in seconds which determines how close the event data will be cropped before the calculated arrival time. Default: 5 minutes.
- `o OFFSET, --offset=OFFSET` Time parameter in seconds which determines how close the event data will be cropped after the calculated arrival time. Default: 80 minutes.
- `m MAGMIN, --magmin=MAGMIN` Minimum magnitude. Default: 3
- `M MAGMAX, --magmax=MAGMAX` Maximum magnitude.
- `r RECT, --rect=RECT` Provide rectangle with GMT syntax: `<west>/<east>/<south>/<north>` (alternative to `-x -X -y -Y`).
- `x SOUTH, --latmin=SOUTH`
APPENDIX B. OPTIONPARSER HELP MESSAGE

Minimum latitude.
-X NORTH, --latmax=NORTH

Maximum latitude.
-y WEST, --lonmin=WEST

Minimum longitude.
-Y EAST, --lonmax=EAST

Maximum longitude.

-i IDENTITY, --identity=IDENTITY
Identity code restriction, syntax: nw.st.l.ch
(alternative to -N -S -L -C).

-N NW, --network=NW Network restriction.
-S ST, --station=ST Station restriction.
-L LO, --location=LO Location restriction.
-C CH, --channel=CH Channel restriction.

-n, --no-temporary Do not request all networks (default), but only permanent ones.

-f, --force Skip working directory warning.

-E, --exceptions Instead entering the normal download procedure, read the file exceptions.txt in the datapath, in which all errors ObsPyLoad encountered while downloading are saved. This mode will try to download the data from every station that returned an error other than 'no data available' last time.

-I PLT, --plot=PLT For each event, create one plot with the data from all stations together with theoretical arrival times. You may provide the internal plotting resolution: e.g. -I 900x600x5. This gives you a resolution of 900x600, and 5 units broad station columns. If -I d, or -I default, the default of 1200x800x1 will be used. If this parameter is not passed to ObsPyLoad at all, no plots will be created. You may additionally specify the timespan of the plot after event origin time in minutes: e.g. for timespan lasting 30 minutes: -I 1200x800x1/30 (or -I d/30). The default timespan is 100 minutes. The final output file will be in pdf format.

-F, --fill-plot When creating the plot, download all the data needed to fill the rectangular area of the plot. Note: depending on your options, this will approximately double the data download volume (but you’ll end up with nicer plots ;-)).

-a PHASES, --phases=PHASES Specify phases for which the theoretical arrival times should be plotted on top if creating the data plot (see above, -I option). Usage: -a phase1,phase2,(...). Default: -a P,S. See the long help for available phases. To plot all available phases, use -a all. If you just want to plot the data and no phases, use -a none.

-d, --debug Show debugging information.
Appendix C

Source code: obspyload.py

For the readers convenience, the source code is supplied in this Appendix. It is the same version as on the accompanying CD. It may become outdated with time, the newest version is found online at:

http://obspy.org/browser/obspy/branches/scheingraber/obspyload.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

# ObsPyLoad: ObsPy Seismic Data Downloader tool. Meant to be used from the shell.
This has been part of a Bachelor's Thesis at the University of Munich.

:copyright:
The ObsPy Development Team (devs@obspy.org).
Developed by Chris Scheingraber.

:license:
GNU General Public License, Version 3
(http://www.gnu.org/licenses/gpl-3.0-standalone.html)

# IMPORT SECTION as described in the thesis #

import sys
import os
import operator
import re
import fnmatch
import time
import pickle
# do not need signal, no `c handling - quit d/l with q now.
# left the remainders in the code since it would be nicer to have 1 thread
# and real `c handling - perhaps someone will pick up on this, had to give up
# import signal
# using threads to be able to capture keypress event without a GUI like
tkinter or pyqt and run the main loop at the same time.
# this only works on posix style unix
windows = sys.platform.startswith('win')
if not windows:
    import threading
    import termios
    TERMIOS = termios
    # need a lock for the global quit variable which is used in two threads
    lock = threading.RLock()
from ConfigParser import ConfigParser
from optparse import OptionParser
```
from obspy.core import UTCDateTime, read
import obspy.neries
import obspy.arclink
import obspy.iris
from obspy.mseed.libmseed import LibMSEED
from lxml import etree
from obspy.taup import taup
# using these modules to wrap the custom(long) help function
from textwrap import wrap
from itertools import izip_longest
try:
    from numpy import np
    from matplotlib import mpl
    from matplotlib.pyplot import plt
    from scipy.ndimage import
except Exception, error:
    print error
    print "Missing dependencies, no plotting available."
    pass

## may use these abbreviations ##
# comments: #
# d/l: download #
# wf: waveform #
# #
# variable/object names: #
# net: network #
# sta: station #
# loc: location #
# cha: channel #
# *fp: file pointer #
# *fh: file handler #
# *fout: file out #
# *fin: file in #
# il: info line #
# hl: headline #
# plt*: plot #

# KEYPRESS-THREAD SECTION as described in the thesis #
# this is to support windows without changing the rest of the code
if windows:
    class keypress_thread():
        
        Empty class, for windows support.
        
        def __init__(self):
            print "Detected windows, no keypress-thread started."
        
        def start(self):
            print "No 'q' key support on windows."

        def check_quit():
            
            Does nothing, for windows support.
            
            return
else:
```python
class keypress_thread (threading.Thread):
    
    """
    This class will run as a second thread to capture keypress events
    """
    global quit, done

    def run(self):
        global quit, done

        msg = 'Keypress capture thread initialized...
        msg += "Press 'q' at any time to finish " \
        + "the file in progress and quit."
        print msg
        while not done:
            c = getkey()
            if c == 'q' and not done:
                with lock:
                    quit = True
                print "You pressed q."
                msg = "ObsPyLoad will finish downloading and saving the " \
                + "last file and quit gracefully."
                print msg
                # exit this thread
                sys.exit(0)

    def getkey():
        """
        Uses termios to wait for a keypress event and return the char.
        """
        fd = sys.stdin.fileno()
        old = termios.tcgetattr(fd)
        new = termios.tcgetattr(fd)
        new[6][TERMIOS.VMIN] = 1
        new[6][TERMIOS.VTIME] = 0
        termios.tcsetattr(fd, TERMIOS.TCSANOW, new)
        try:
            c = os.read(fd, 1)
        finally:
            termios.tcsetattr(fd, TERMIOS.TCSANOW, old)
        return c

    def check_quit():
        """
        Checks if the user pressed q to quit downloading meanwhile.
        """
        global quit
        with lock:
            if quit:
                msg = "Quitting. To resume the download, just run " + \
                "ObsPyLoad again, using the same arguments."
                print msg
                sys.exit(0)

        # MAIN FUNCTION SECTION as described in the thesis #

    def main():
        """
        Main function to run as a dedicated program.
```

APPENDIX C. SOURCE CODE: OBSPYLOAD.PY

```python
168
169
170
171
global datapath, quit, done, skip_networks
172
# dead networks deactivated for now
173
skip_networks = []
174
# if hardcoded skip networks are ok, uncomment this line:
175
# skip_networks = ['AI', 'BA']
176
#############################################################
177
# CONFIG AND OPTIONPARSER SECTION as described in the thesis#
178
#############################################################
179
# create ConfigParser object.
180
# set variable names as dict keys, default values as _STRINGS_!
181
# you don't need to provide every possible option here, just the ones with
182
# default values
183
# need to provide default start and end time, otherwise
184
# obspy.arclink.getInventory will raise an error if the user does not
185
# provide start and end time
186
# default for start is three months ago, end is now
187
# default offset is 80 min, preset 5min, default velocity model is 'iasp91'
188
cfg = ConfigParser({'magmin': '3',
189
'dt': '10',
190
'start': str(UTCDateTime.utcnow() - 60 * 60 * 24 * 30 * 3),
191
'end': str(UTCDateTime.utcnow()),
192
'preset': '300',
193
'offset': '4800',
194
'datapath': 'obspyload-data',
195
'model': 'iasp91',
196
'phases': 'P,S',
197
'nw': '*',
198
'st': '*',
199
'lo': '*',
200
'ch': '*'}
201
202
# read config file, if it exists, possibly overriding defaults as set above
203
config.read('˜/.obspyloadrc')
204
205
# create command line option parser
206
# parser = OptionParser("%prog [options]" + __doc__.rstrip())
207
parser = OptionParser("%prog [options]")
208
209
# configure command line options
210
# action=".." tells OptionsParser what to save:
211
# store_true saves bool TRUE,
212
# store_false saves bool FALSE, store saves string; into the variable
213
# given with dest="var"
214
# * you need to provide every possible option here.
215
# reihenfolge wird eingehalten in help msg.
216
parser.add_option("-H", "--more-help", action="store_true",
217
dest="showhelp", help="Show explanatory help and exit.")
218
helpmsg = "Instead of downloading seismic data, download metadata: " + \
219
"resp instrument and dataless seed files."
220
parser.add_option("-q", "--query-metadata", action="store_true",
221
dest="metadata", help=helpmsg)
222
helpmsg = "The path where ObsPyLoad will store the data (default is " + \
223
"./ObspyLoad-data for the data download mode and " + \
224
"./ObspyLoad-metadata for metadata download mode)."
225
parser.add_option("-P", "--datapath", action="store", dest="datapath",
226
dest=helpmsg)
227
helpmsg = "Update the event database when ObsPyLoad runs on the same " + \
228
"directory a second time in order to continue data downloading." + \
229
parser.add_option("-u", "--update", help=helpmsg,
```

50
APPENDIX C. SOURCE CODE: OBSPYLOAD.PY

action="store_true", dest="update")
    helpmsg = "If the datapath is found, do not resume previous downloads " + "as is the default behaviour, but redownload everything. " + "Same as deleting the datapath before running ObsPyLoad."
parser.add_option(('-R', '--reset', action="store_true", dest="reset", help=helpmsg)
parser.add_option(('-s', '--starttime', action="store", dest="start", help="Start time. Default: 3 months ago.")
parser.add_option(('-e', '--endtime', action="store", dest="end", help="End time. Default: now.")
parser.add_option(('-t', '--time', action="store", dest="time", help="Start and End Time delimited by a slash.")
helpmsg = "Velocity model for arrival time calculation used to crop " + "the data, either 'iasp91' or 'ak135'. Default: 'iasp91'."
parser.add_option(('-v', '--velocity-model', action="store", dest="model", help=helpmsg)
helpmsg = "Time parameter in seconds which determines how close the " + "event data will be cropped before the calculated arrival " + "time. Default: 5 minutes."
parser.add_option(('-p', '--preset', action="store", dest="preset", help=helpmsg)
helpmsg = "Time parameter in seconds which determines how close the " + "event data will be cropped after the calculated arrival time." + " Default: 80 minutes."
parser.add_option(('-o', '--offset', action="store", dest="offset", help=helpmsg)
parser.add_option(('-m', '--magmin', action="store", dest="magmin", help="Minimum magnitude. Default: 3")
parser.add_option(('-M', '--magmax', action="store", dest="magmax", help="Maximum magnitude.")
helpmsg = "Provide rectangle with GMT syntax: <west>/<east>/<south>/<north> (alternative to -x -X -y -Y)."
parser.add_option(('-r', '--rect', action="store", dest="rect", help=helpmsg)
parser.add_option(('-x', '--latmin', action="store", dest="south", help="Minimum latitude.")
parser.add_option(('-X', '--latmax', action="store", dest="north", help="Maximum latitude.")
parser.add_option(('-y', '--lonmin', action="store", dest="west", help="Minimum longitude.")
parser.add_option(('-Y', '--lonmax', action="store", dest="east", help="Maximum longitude.")
helpmsg = "Identity code restriction, syntax: nw.st.l.ch (alternative " + "to -N -S -L -C)."
parser.add_option(('-i', '--identity', action="store", dest="identity", help=helpmsg)
parser.add_option(('-N', '--network', action="store", dest="nw", help="Network restriction.")
parser.add_option(('-S', '--station', action="store", dest="st", help="Station restriction.")
parser.add_option(('-L', '--location', action="store", dest="lo", help="Location restriction.")
parser.add_option(('-C', '--channel', action="store", dest="ch", help="Channel restriction.")
helpmsg = "Do not request all networks (default), but only permanent ones."
parser.add_option(('-n', '--no-temporary', action="store_true", dest="permanent", help=helpmsg)
parser.add_option(('-f', '--force', action="store_true", dest="force", help="Skip working directory warning.")
helpmsg = "Instead entering the normal download procedure, read the " + "file exceptions.txt in the datapath, in which all " + "errors ObsPyLoad encountered while downloading are saved. " + "This mode will try to download the data from every." + "

"station that returned an error other than 'no data " + "available' last time."
parser.add_option("-E", "--exceptions", action="store_true",
dest="exceptions", help=helpmsg)
helpmsg = "For each event, create one plot with the data from all " + "stations together with theoretical arrival times. You may " + "provide the internal plotting resolution: e.g. " + "-I 900x600x5. This gives you a resolution of 900x600, " + "and 5 units broad station columns. If -I d, " + "or -I default, the default of " + "1200x800x1 will be used. If this parameter is not " + "passed to ObsPyLoad at all, no plots will be created." + "You may additionally specify the timespan of the plot " + "after event origin time in minutes: e.g. for timespan " + "lasting 30 minutes: -I 1200x800x1/30 (or -I d/30). The " + "default timespan is 100 minutes. The final output file " + "will be in pdf format."
parser.add_option("-I", "--plot", action="store", dest="plt",
help=helpmsg)
helpmsg = "When creating the plot, download all the data needed to " + "fill the rectangular area of the plot. Note: depending on " + "your options, this will approximately double the data " + "download volume (but you'll end up with nicer plots ;-) )."
parser.add_option("-F", "--fill-plot", action="store_true", dest="fill",
help=helpmsg)
helpmsg = "Specify phases for which the theoretical arrival times " + "should be plotted on top if creating the data plot (see " + "above, -I option). Usage: -a phase1,phase2, (...)." + " Default: -a P,S. See the long help for available phases. " + "To plot all available phases, use -a all. If you just " + "want to plot the data and no phases, use -a none."
parser.add_option("-a", "--phases", action="store", dest="phases",
help=helpmsg)
parser.add_option("-d", "--debug", action="store_true", dest="debug",
help="Show debugging information.")

# read from ConfigParser object's defaults section into a dictionary.
# config.defaults() (ConfigParser method) returns a dict of the default
# options as specified above
config_options = config.defaults()

# config_options is dictionary of _strings_ (see above dict),
# override respective correct # default types here
# * you dont need to provide every possible option here, just the ones with
# default values overriden
config_options['magmin'] = config.getfloat('DEFAULT', 'magmin')
config_options['dt'] = config.getfloat('DEFAULT', 'dt')
config_options['preset'] = config.getfloat('DEFAULT', 'preset')
config_options['offset'] = config.getfloat('DEFAULT', 'offset')
# it's not possible to override the start and end time defaults here, since
# they are of obspy's own UTCDateTime type. will handle below.

# feed config_options dictionary of defaults into parser object
parser.set_defaults(**config_options)

# parse command line options
(options, args) = parser.parse_args()
if options.debug:
    print "(options, args) created"
    print "options: ", options
    print "args: ", args
# command line options can now be accessed via options.varname.
# check flags just like if options.flag:, so without == True, because even
# if they do not have a default False value, they are None/don't exist,
# which also leads to False in the if-statement

# * override respective correct default types for _every_ possible option
# that is not of type 'string' here. take care that it is only done if the
# var. really exists
if options.south:
    options.south = float(options.south)
if options.north:
    options.north = float(options.north)
if options.west:
    options.west = float(options.west)
if options.east:
    options.east = float(options.east)

##########################################################################
# VARIABLE SPLITTING AND SANITY CHECK SECTION as described in the thesis #
##########################################################################
# print long help if -H
if options.showhelp:
    help()
    sys.exit()

# Sanity check for velocity model
if options.model != 'iasp91' and options.model != 'ak135':
    print "Erroneous velocity model given."
    print "correct are '-v iasp91' or '-v ak135'."
    sys.exit(2)

# parse pixel sizes and timespan of the plot if -I
if options.plt:
    try:
        # this will do it's job if the user has given a timespan
        size, timespan = options.plt.split('/')
        if size == 'd' or size == 'default':
            pltWidth, pltHeight, colWidth = 1200, 800, 1
        else:
            try:
                pltWidth, pltHeight, colWidth = size.split('x')
                pltWidth = int(pltWidth)
                pltHeight = int(pltHeight)
                colWidth = int(colWidth)
            except:
                print "Erroneous plot size given."
                print "Format: e.g. -I 800x600x1/80"
                sys.exit(0)

        try:
            timespan = float(timespan)
            # we need the timespan in seconds later
            timespan *= 60
        except:
            print "Erroneous timespan given."
            print "Format: e.g. -I d/80"
            sys.exit(0)
    except:
        # we're in here if the user did not provide a timespan
        if options.plt == 'd' or options.plt == 'default':
            pltWidth, pltHeight, colWidth = 1200, 800, 1
        else:
            try:
                pltWidth, pltHeight, colWidth = options.plt.split('x')
                pltWidth = int(pltWidth)
                pltHeight = int(pltHeight)
                colWidth = int(colWidth)
            except:
                print "Erroneous plot size given."
print "Format: e.g. -I 800x600x3"
sys.exit(0)

# this is the default timespan if no timespan was provided
timespan = 100 * 60.0
if options.debug:
    print "pltWidth: ", pltWidth
    print "pltHeight: ", pltHeight
    print "colWidth: ", colWidth
    print "timespan: ", timespan

    # parse phases into a list of strings usable with travelTimePlot
    try:
        if options.phases == 'none':
            pltPhases = []
        elif options.phases == 'all':
            pltPhases = ['P', "P'P'ab", "P'P'bc", "P'P'df", 'PKKPab', 'PKKPbc', '
            'PKKPd', 'PKKSab', 'PKKSbc', 'PKKSdf', 'PKPab', 'PKPbc', '
            'PKPdf', 'PKPdiff', 'PKSab', 'PKSbc', 'PKSdf', 'PKiKP', '
            'PP', 'PS', 'PcP', 'PcS', 'Pdiff', 'Pn', 'PnPn', 'Pns', '
            'S', 'S'S'ac", "S'S'df", 'SKKPab', 'SKKPbc', 'SKKPdf', '
            'SKKSac', 'SKKSdf', 'SKPab', 'SKPbc', 'SKPdiff', 'SKSac', '
            'SKSdf', 'SKiKP', 'SP', 'SPg', 'SPn', 'SS', 'ScP', 'ScS', '
            'Sdiff', 'Sn', 'SnSn', 'pP', 'pPKPab', 'pPKPbc', 'pPKPdf', '
            'pPKPdiff', 'pPKiKP', 'pPdiff', 'pPn', 'pP', 'pPKSac', '
            'pPKSdf', 'pPdiff', 'SP', 'SPkPab', 'SPkPbc', 'SPkPdf', '
            'SPkPdiff', 'SPkIKP', 'SPb', 'SPdiff', 'SPg', 'SPn', 'SS', '
            'sPKSac', 'sSKSac', 'sSKSdf', 'sSdiff', 'sSn']
        else:
            pltPhases = options.phases.split(',')
    except:
        print "Erroneous phases given."
        print "Format: e.g. -a P,S,PKPdiff"
        sys.exit(0)

## if the user has given e.g. -r x/x/x/x or -t time1/time
## extract min. and max. longitude and latitude if the user has given the
## coordinates with -r (GMT syntax)
if options.rect:
    if options.west or options.east or options.south or options.north:
        msg = "Either provide the rectangle with GMT syntax, or with " + 
        "-x -X -y -Y, not both."
        msg = msg
        sys.exit(2)
    try:
        options.rect = options.rect.split('/')
    except:
        print "Erroneous rectangle given."
        sys.exit(2)
    options.west = float(options.rect[0])
    options.east = float(options.rect[1])
    options.south = float(options.rect[2])
    options.north = float(options.rect[3])
except:
    print "Erroneous rectangle given."
print optarg, rect
sys.exit(2)

## Extract start and end time if the user has given the timeframe with
## -t start/end (GMT syntax)
if options.time:
    msg = "It makes no sense to provide start and end time with -s -e " + 
    "and -t at the same time, but if you do so, -t will override -s -e."
print msg
try:
    options.start = options.time.split('/')[0]
    options.end = options.time.split('/')[1]
except:
    print 'Erroneous timeframe given.'
    sys.exit(2)
if options.debug:
    print "options.start", options.start
    print "options.end", options.end
# Extract network, station, location, channel if the user has given an
# identity code (-i xx.xx.xx.xx)
if options.identity:
    msg = "It makes no sense to provide station restrictions with -i and" \
    + " -N -S -L -C at the same time, but if you do so, -i will override."
    print msg
    try:
        options.nw, options.st, options.lo, options.ch = \
        options.identity.split('.')
    except:
        print 'Erroneous identity code given.'
        sys.exit(2)
    if options.debug:
        print "options.nw:	", options.nw
        print "options.st:	", options.st
        print "options.lo:	", options.lo
        print "options.ch:	", options.ch
# change time string to UTCDateTime object. This is done last, so it's
# only necessary once, no matter if -t or -s -e
try:
    options.start = UTCDateTime(options.start)
    options.end = UTCDateTime(options.end)
except:
    print 'Given time string not compatible with ObsPy UTCDateTime method.'
    sys.exit(2)
if options.debug:
    print "Now it's UTCDateTime:"
    print "options.start", options.start
    print "options.end", options.end
### SPECIAL TASK SECTION as described in the thesis ###
cwd = os.getcwd()
# change default datapath if in metadata mode
if options.metadata and options.datapath == 'obspyload-data':
    options.datapath = os.path.join(cwd, 'obspyload-metadata')
# parse datapath (check if given absolute or relative)
if os.path.isabs(options.datapath):
    datapath = options.datapath
else:
    datapath = os.path.join(cwd, options.datapath)
# delete data path if -R or --reset args are given at cmdline
if options.reset:
    # try-except so we don't get an exception if path doesn't exist
    try:
        from shutil import rmtree
        rmtree(datapath)
    except:
        pass
# if -q oder --query-metadata, do not enter normal data download operation,
# but download metdata and quit.
if options.metadata:
print "ObsPyLoad will download resp and dataless seed instrument /* files and quit." 
queryMeta(options.west, options.east, options.south, options.north, 
    options.start, options.end, options.nw, options.st, 
    options.lo, options.ch, options.permanent, options.debug)
return
# if -E oder --exceptions, do not enter normal data download operation, 
# operation, but read exceptions.txt and try to download again and quit. 
if options.exceptions:
    print "ObsPyLoad will now try to download the data that returned /* an error other than 'no data available' last time." 
    exceptionMode(debug=options.debug)
return
# if -u or --update, delete event and catalog pickled objects 
if options.update:
    try:
        os.remove(os.path.join(datapath, 'events.pickle'))
        os.remove(os.path.join(datapath, 'inventory.pickle'))
        os.remove(os.path.join(datapath, 'availability.pickle'))
    except:
        pass
# Warn that datapath will be created and give list of further options 
if not options.force:
    if not os.path.isdir(datapath):
        if len(sys.argv) == 1:
            print "Welcome," print "you provided no options, using all default values will print "download every event that occurred in the last 3 months" print "with magnitude > 3 from every available station." print "\nObsPyLoad will now create the folder %s" % datapath print "and possibly download vast amounts of data. Continue?" print "Brief help: obspyload.py -h" print "Long help: obspyload.py -H" answer = raw_input("[y/N]> ")
if answer != "y":
    print "Exiting ObsPyLoad." sys.exit(2)
else:
    print "Found existing data folder %s" % datapath 
    msg = "Resume download?\nNotes:" 
    msg += "-- suppress this message with -f or --force\n" 
    msg += "-- update the event database before resuming download " 
    msg += "-- with -u or --update\n" 
    msg += "-- reset and redownload everything, including all data, " 
    msg += "-- with -R or --reset\n" 
    msg += "Brief help: obspyload.py -h\n" 
    msg += "Long help: obspyload.py -H"
    print msg 
    answer = raw_input("[y/N]> ")
    if answer != "y":
        print "Exiting obspy.
        sys.exit(2)
############################################################
# DATA DOWNLOAD ROUTINE SECTION as described in the thesis #
############################################################
# create datapath
if not os.path.exists(datapath):
    os.mkdir(datapath)
# startkeypress thread, so we can quit by pressing 'q' anytime from now on 
# during the downloads
done = False
keypress_thread().start()

# (1) get events from NERIES-eventservice
if options.debug:
    print '#############'
    print "options: ", options
    print '#############'
events = get_events(options.west, options.east, options.south,
                      options.north, options.start, options.end,
                      options.magmin, options.magmax)
if options.debug:
    print 'events from NERIES:', events

# (2) get inventory data from ArcLink
# check if the user pressed 'q' while we did d/l eventlists.
check_quit()
arclink_stations = get_inventory(options.start, options.end, options.nw,
                                  options.st, options.lo, options.ch,
                                  permanent=options.permanent,
                                  debug=options.debug)

# arclink_stations is a list of tuples of all stations:
# [(station1, lat1, lon1), (station2, lat2, lon2), ...]
if options.debug:
    print 'arclink_stations returned from get_inventory:', arclink_stations

# (3) Get availability data from IRIS
# check if the user pressed 'q' while we did d/l the inventory from ArcLink
check_quit()
avail = getnparse_availability(start=options.start, end=options.end,
                                nw=options.nw, st=options.st, lo=options.lo,
                                ch=options.ch, debug=options.debug)
irisclient = obspy.iris.Client(debug=options.debug)

# (4) create and write to catalog file
headline = "event_id;datetime;origin_id;author;flynn_region;"
headline += "latitude;longitude;depth;magnitude;magnitude_type;"
headline += "DataQuality;TimingQualityMin\n" + "#" * 126 + "\n\n"
hl_eventf = "Station;Data Provider;TQ min;Gaps;Overlaps" + "\n"
hl_eventf += "#" * 42 + "\n\n"
catalogfp = os.path.join(datapath, 'catalog.txt')
# open catalog file in read and write mode in case we are continuing d/l,
# so we can append to the file
try:
    catalogfout = open(catalogfp, 'r+t')
except:
    # the file did not exist, we are not continuing d/l
    catalogfout = open(catalogfp, 'wt')
catalogfout.write(headline)
# move to end of catalog file. that way if we are continuing downloading,
# we overwrote the headline with the same headline again and now continue
# to write new entries to the end of the file.
catalogfout.seek(0, 2)
# initialize ArcLink webservice client
arcclient = obspy.arclink.Client(timeout=5, debug=options.debug)

# (5) Loop through events
# create exception file
# this file will contain any information about exceptions while trying to
# download data: the event we were trying to d/l, starttime, endtime,
# the station, the exception
exceptionfp = os.path.join(datapath, 'exceptions.txt')
# try open exceptionfile in read and write mode if we continue d/l
try:
    exceptionfout = open(exceptionfp, 'r+t')
# i'll just read the whole file into one string and check for each
# station whether it's in the string
exceptionstr = exceptionfout.read()
if options.debug:
    print "exceptionstr: ", exceptionstr
# go back to beginning of exceptionfout
exceptionfout.seek(0)
except:
    # the file did not exist, we are not continuing d/l
    exceptionfout = open(exceptionfp, 'wt')
    exceptionstr = ''
    exceptionhl = 'event_id;data provider;station;starttime;endtime;exception'
    exceptionhl += '\n' + '#' * 58 + '\n\n'
    exceptionfout.write(exceptionhl)
    # just like for the catalog file, move to end of exception file
    exceptionfout.seek(0, 2)
if options.plt:
    alleventsmatrix = np.zeros((pltHeight, pltWidth))
    alleventsmatrix_counter = 0
    for eventdict in events:
        check_quit()
        eventid = eventdict['event_id']
        eventtime = eventdict['datetime']
        # extract information for taup
        eventlat = float(eventdict['latitude'])
        eventlon = float(eventdict['longitude'])
        eventdepth = float(eventdict['depth'])
        if options.debug:
            print '#############
            print 'event:', eventid
            for key in eventdict:
                print key, eventdict[key]
            # create event info line for catalog file and quakefile
            infoline = eventdict['event_id'] + ';' + str(eventdict['datetime'])
            infoline += ';' + str(eventdict['origin_id']) + ';
            infoline += eventdict['author'] + ';' + eventdict['flynn_region']
            infoline += ';' + str(eventdict['latitude']) + ';
            infoline += str(eventdict['longitude']) + ';
            infoline += str(eventdict['depth']) + ';
            infoline += str(eventdict['magnitude']) + ';' + str(eventdict['magnitude_type'])
            # create event-folder
            eventdir = os.path.join(datapath, eventid)
            if not os.path.exists(eventdir):
                os.mkdir(eventdir)
            # re-init neriesclient here, seems to reduce problems
            neriesclient = obspy.neries.Client()
            # download quake ml xml
            quakemlfp = os.path.join(eventdir, 'quakeml.xml')
            if not os.path.isfile(quakemlfp):
                print "Downloading quakeml xml file for event %s..." % eventid,
                try:
                    quakeml = neriesclient.getEventDetail(eventid, 'xml')
                    quakemlfout = open(quakemlfp, 'wt')
                    quakemlfout.write(quakeml)
                    quakemlfout.close()
                except Exception, error:
                    print "error: ", error
                    else:
                        print "done."
                else:
                    print "Found existing quakeml xml file for event %s, skip..." % eventid
# init/reset dqsum
dqsum = 0
tqlist = []

# create event file in event dir
# DQ: all min entries in event folder txt file differently
# this is handled inside the station loop
quakefp = os.path.join(eventdir, 'quake.txt')
# open quake file in read and write mode in case we are continuing d/l,
# so we can append to the file
try:
    quakefout = open(quakefp, 'r+t')
except:
    # the file did not exist, we are not continuing d/l
    quakefout = open(quakefp, 'wt')
quakefout.write(headline[:97] + '
' + '#' * 97 + '

')
quakefout.write(infoline + '


')
quakefout.write(hl_eventf)
quakefout.flush()

# just like for catalog and exception file, move to end of quake file
# to write new stations to the end of it
quakefout.seek(0, 2)

# init matrix containing all station plots - will be used to plot
# all station waveforms later. +1 because the [0] entry of each col
# works as a counter
if options.plt:
    stmatrix = np.zeros((pltHeight + 1, pltWidth))

# (5.1) ArcLink wf data download loop (runs inside event loop)
# Loop trough arclink_stations
for station in arclink_stations:
    check_quit()
    try:
        stationlat = station[1]
        stationlon = station[2]
        station = station[0]
    except:
        continue
    if options.debug:
        print "station: ", station
    # skip dead networks
    net, sta, loc, cha = station.split('.

    if net in skip_networks:
        print 'Skipping dead network %s...' % net
        # continue the for-loop to the next iteration
        continue
    # create data file pointer
    datafout = os.path.join(eventdir, "%s.mseed" % station)
    if os.path.isfile(datafout):
        print 'Data file for event %s from %s exists, skip...' \
        % (eventid, station)
        continue
    # if this string has already been in the exception file when we
    # were starting the d/l, we had an exception for this event/data
    # provider/station combination last time and won't try again.
    skipstr = eventid + ';ArcLink;' + station
    if skipstr in exceptionstr:
        msg = 'Encountered exception for event %s from ArcLink %s last'
        msg += ' time, skip...
        print msg % (eventid, station)
        continue
    # use taup to calculate the correct starttime and endtime for
    # waveform retrieval at this station
    distance = taup.locations2degrees(eventlat, eventlon, stationlat,
                                        stationlon)
if options.debug:
    print "distance ", distance, type(distance)
print "eventdepth ", eventdepth, type(eventdepth)
print "options.model ", options.model
traveltimes = taup.getTravelTimes(distance, eventdepth, 
    model=options.model)
if options.debug:
    print "traveltimes ", traveltimes
# find the earliest arrival time
arrivaltime = 99999
for phase in traveltimes:
    if phase['time'] < arrivaltime:
        arrivaltime = phase['time']
if options.debug:
    print "earliest arrival time ", arrivaltime
starttime = eventtime + arrivaltime - options.preset
endtime = eventtime + arrivaltime + options.offset
print 'Downloading event %s from ArcLink %s...
    % (eventid, station),
    try:
    # I have been told that often initializing the client reduces
    # problems
    arcclient = obspy.arclink.Client(timeout=5,
        debug=options.debug)
    # catch exception so the d/l continues if only one doesn't work
    arcclient.saveWaveform(filename=datafout, network=net,
        station=sta, location=loc, channel=cha,
        starttime=starttime, endtime=endtime)
except Exception, error:
    print "download error ", print error
    # create exception file info line
    il_exception = str(eventid) + ';ArcLink;' + station + ';'
    il_exception += str(starttime) + ';' + str(endtime) + '
    il_exception += str(error) + '
    exceptionfout.write(il_exception)
    exceptionfout.flush()
    continue
else:
    # else code will run if try returned no exception!
    # write station name to event info line
    il_quake = station + ';ArcLink;' 
    # Quality Control with libmseed
    dqsum += sum(mseed.getDataQualityFlagsCount(datafout)) 
    # Timing Quality, trying to get all stations into one line in
    # eventfile, and handling the case that some station's mseeds
    # provide TQ data, and some do not
    tq = mseed.getTimingQuality(datafout)
    if tq != {}:
        tqlist.append(tq['min'])
        il_quake += str(tq['min'])
    else:
        il_quake += str('None')
    # finally, gaps&overlaps into quakefile
    # read mseed into stream, use .getGaps method
    st = read(datafout)
    # this code snippet is taken from stream.printGaps since I need
    # gaps and overlaps distinct.
    result = st.getGaps()
    gaps = 0
    overlaps = 0
for r in result:
    if r[6] > 0:
gaps += 1
else:
    overlaps += 1
il_quake += ';%d;%d
' % (gaps, overlaps)
quakefout.write(il_quake)
quakefout.flush()
# if there has been no Exception, assume d/l was ok
print "done."
if options.plt:
    # referencing st[0] with tr
    tr = st[0]
    if options.fill:
        # if the user gave -F option
        print "Getting and scaling data for station plot...",
        del st
        # get data for the whole timeframe needed for the
        # plot. We don't want to save this, it's just needed
        # for the (rectangular) plot
        try:
            st = arcclient.getWaveform(network=net,
                                       station=sta, location=loc,
                                       channel=cha, starttime=eventtime,
                                       endtime=eventtime + timespan)
        except Exception, error:
            print "error: ",
            print error
            continue
    else:
        # if the user did not provide -F, we wont d/l any more
        # data. we need trim existing data:
        print "Scaling data for station plot...",
        tr.trim(starttime=eventtime,
                endtime=eventtime + timespan, pad=True,
                fill_value=0)
    # x axis / abscissa - distance
    # y axis / ordinate - time
    # normalize the trace, needed for plotting
    tr.normalize()
    # obtain the time increment that passes between samples
    # delta = tr.stats['delta']
    # scale down the trace array so it matches the output size
    # using scipy since that's faster than letting the plotting
    # function handle it
    pixelcol = np.around(scipy.ndimage.interpolation.zoom(
        tr,
        float(pltHeight) / len(tr)),
        7)
    if options.debug:
        print "pixelcol: ", pixelcol
        # Find the pixel column that represents the distance of
        # this station. if the colWidth is >1, we need to plot the
        # station to the according width, reducing the internal
        # resolution of the plot by this factor
        x_coord = int((distance / 180.0) * pltWidth)
        # now we need to floor down to the next multiple of the
        # station column width:
        x_coord -= x_coord % colWidth
        # Add trace as one column to waveform matrix. the [0] entry
        # of the matrix counts how many waveforms have been added
        # to that column (this will be normalized later)
        # For no (to me) apparent reason, sometimes
# scipy.ndimage.interpolation.zoom returns a slightly
# different array size, so I use try-except.
# It seems to be worse with some output sizes and no
# problem at all with other ones.
if options.debug:
    print "len stack: ", len(np.hstack((1, abs(pixelcol))))
    print "len stmatrixslice: ", len(stmatrix[:, x_coord])
    # add counter entry to pixelcol and take absolute of all
    # values in pixelcol
    pixelcol = np.hstack((1, abs(pixelcol)))
try:
    # add pixelcol to 1 or more columns, depending on the
    # chosen width of the station columns
    stmatrix[:, x_coord:x_coord + colWidth] += \
        np.vstack([pixelcol] * colWidth).transpose()
except:
    print "failed."
    continue
if options.debug:
    print "stmatrix: ", stmatrix
    print "done."
del st

# (5.2) Iris wf data download loop
for net, sta, loc, cha, stationlat, stationlon in avail:
    check_quit()
    # construct filename:
    station = ' '.join((net, sta, loc, cha))
    irisfn = station + '.mseed'
    irisfnfull = os.path.join(datapath, eventid, irisfn)
    if options.debug:
        print 'irisfnfull: ', irisfnfull
    if os.path.isfile(irisfnfull):
        print 'Data file for event %s from %s exists, skip...' % \
            (eventid, station)
        continue
    print 'Downloading event %s from IRIS %s...' % (eventid, station),
    # use taup to calculate the correct starttime and endtime for
    # waveform retrieval at this station
    distance = taup.locations2degrees(eventlat, eventlon, stationlat,
        stationlon)
    traveltimes = taup.getTravelTimes(distance, eventdepth,
        model=options.model)
    # find the earliest arrival time
    arrivaltime = 99999
    for phase in traveltimes:
        if phase['time'] < arrivaltime:
            arrivaltime = phase['time']
    if options.debug:
        print "earliest arrival time: ", arrivaltime
    starttime = eventtime + arrivaltime - options.preset
    endtime = eventtime + arrivaltime + options.offset
    try:
        # I have been told that initializing the client often reduces
        # problems
irisclient = obspy.iris.Client(debug=options.debug)
irisclient.saveWaveform(filename=irisfnfull,
    network=net, station=sta,
    location=loc, channel=cha,
    starttime=starttime, endtime=endtime)

except Exception, error:
    print "download error: ", error
    # create exception file info line
    il_exception = str(eventid) + ';IRIS;' + station + ';
    il_exception += str(starttime) + ';' + str(endtime) + ';
    il_exception += str(error) + '\n'
    exceptionfout.write(il_exception)
    exceptionfout.flush()
    continue
else:
    # if there was no exception, the d/l should have worked
    # data quality handling for iris
    # write station name to event info line
    il_quake = station + ';IRIS;'  
    # Quality Control with libmseed
    dqsum += sum(mseed.getDataQualityFlagsCount(irisfnfull))
    # Timing Quality, trying to get all stations into one line in
    # eventfile, and handling the case that some station's mseeds
    # provide TQ data, and some do not
    try:
        tq = mseed.getTimingQuality(irisfnfull)
        if tq != {}:  
            tqlist.append(tq['min'])
        il_quake += str(tq['min'])
    except:  
        il_quake += str('None')
    except:  
        pass
    # finally, gaps&overlaps into quakefile
    # read mseed into stream, use .getGaps method
    st = read(irisfnfull)
    # this code snippet is taken from stream.printGaps since I need
    # gaps and overlaps distinct.
    result = st.getGaps()
    gaps = 0
    overlaps = 0
    for r in result:
        if r[6] > 0:
            gaps += 1
        else:
            overlaps += 1
    print "done."
if options.plt:
    # this is the same as for arclink, I did not want to
    # replicate the comments, see above for them
    tr = st[0]
    if options.fill:
        print "Getting and scaling data for station plot...",
        del st
    try:
        st = irisclient.getWaveform(network=net,
            station=sta, location=loc,
            channel=cha, starttime=eventtime,
            endtime=eventtime + timespan)
    except Exception, error:
        print "error: ", error,
        print error
    continue
else:
    # if the user did not provide -F, fill up existing data:
    print "Scaling data for station plot...",
    tr.trim(starttime=eventtime,
            endtime=eventtime + timespan, pad=True,
            fill_value=0)
    tr.normalize()
    pixelcol = np.around(scipy.ndimage.interpolation.zoom(
        tr,
        float(pltHeight) / len(tr)), 7)
    if options.debug:
        print "pixelcol: ", pixelcol
    x_coord = int((distance / 180.0) * pltWidth)
    x_coord = x_coord % colWidth
    if options.debug:
        print "len stack: ", len(np.hstack((1, abs(pixelcol))))
        print "len stmatrixslice: ", len(stmatrix[:, x_coord])
    try:
        stmatrix[:, x_coord:x_coord + colWidth] += \
            np.vstack([pixelcol] * colWidth).transpose()
    except:
        print "failed."
        continue
    if options.debug:
        print "stmatrix: ", stmatrix
print "done."
delete at
il_quake = ';%d;%d
' % (gaps, overlaps)
quakefout.write(il_quake)
quakefout.flush()
# write data quality info into catalog file event info line
if dqsum == 0:
    infoline += ';0 (OK);'
else:
    infoline += ';\d\d\n' % (gaps, overlaps)
quakefout.write(il_quake)
quakefout.flush()
    # write timing quality into event info line (minimum of all 'min'
    # entries
if tqlist != []:
    infoline += '%.2f' % min(tqlist) + '
else:
    infoline += 'None\n'
# write event info line to catalog file (including QC)
catalogfout.write(infoline)
catalogfout.flush()
### end of station loop ###
if options.plt:
    # normalize each distance column - the [0, i] entry has been
    # counting how many stations we did add at that distance
    for i in range(pltWidth - 1):
        if stmatrix[0, i] != 0:
            stmatrix[:, i] /= stmatrix[0, i]
    # [1:,1] because we do not want to display the counter
    plt.imshow(stmatrix[1:, :], vmin=0.001, vmax=1,
                origin='lower', cmap=pl.cm.hot_r,
                norm=mpl.colors.LogNorm(vmin=0.001, vmax=1))
    plt.xticks(range(0, pltWidth, pltWidth / 4),
                ('0', '45', '90', '135', '180'), rotation=45)
    y_incr = timespan / 60 / 4
    plt.yticks(range(0, pltHeight, pltHeight / 4),
APPENDIX C. SOURCE CODE: OBSPYLOAD.PY

1113 ('0', str(y_incr), str(2 * y_incr), str(3 * y_incr),
1114 str(3 * y_incr)))
1115 plt.xlabel('Distance from epicenter in degrees')
1116 plt.ylabel('Time after origin time in minutes')
1117 titlemsg = "Event %s:ndata and % eventid + 
1118 "theoretical arrival times
"n"
1119 plt.title(titlemsg)
1120 cbar = plt.colorbar()
1121 mpl.colorbar.ColorbarBase.set_label(cbar, 'Relative amplitude')
1122 # add taupe theoretical arrival times points to plot
1123 # invoicing travelTimePlot function, taken and fitted to my needs
1124 # from the obspy.taup package
1125 # choose npoints value depending on plot size, but not for every
1126 # pixel so pdf conversion won't convert the points to a line
1127 travelTimePlot(npoints=pltWidth / 10, phases=pltPhases,
1128 depth=eventdepth, model=options.model,
1129 pltWidth=pltWidth, pltHeight=pltHeight,
1130 timespan=timespan)
1131 # construct filename and save event plots
1132 print "Done with event %s, saving plots..." % eventid
1133 if options.debug:
1134 print "stmatrix: ", stmatrix
1135 plotfn = os.path.join(datapath, eventid, 'waveforms.pdf')
1136 plt.savefig(plotfn)
1137 # clear figure
1138 plt.clf()
1139 alleventsmatrix += stmatrix[1:, :]
1131 alleventsmatrix_counter += 1
1141 del stmatrix
1142 # save plot of all events, similar as above, for comments see above
1143 if options.plt:
1144 print "Saving plot of all events stacked..."
1145 plt.imshow(alleventsmatrix / alleventsmatrix_counter,
1146 origin='lower', cmap=plt.cm.hot_r,
1147 norm=mpl.colors.LogNorm(vmin=0.01, vmax=1))
1148 plt.xticks(
1149 range(0, pltWidth, pltWidth / 4),
1150 ('0', '45', '90', '135', '180'), rotation=45)
1151 plt.yticks(
1152 range(0, pltHeight, pltHeight / 4),
1153 ('0', str(y_incr), str(2 * y_incr), str(3 * y_incr),
1154 str(3 * y_incr)))
1155 plt.xlabel('Distance from epicenter in degrees')
1156 plt.ylabel('Time after origin time in minutes')
1157 titlemsg = "%s events data stacked
" % len(events)
1158 plt.title(titlemsg)
1159 cbar = plt.colorbar()
1160 mpl.colorbar.ColorbarBase.set_label(cbar, 'Relative amplitude')
1161 travelTimePlot(npoints=pltWidth / 10, phases=pltPhases,
1162 depth=10, model=options.model,
1163 pltWidth=pltWidth, pltHeight=pltHeight,
1164 timespan=timespan)
1165 plotfn = os.path.join(datapath, 'allevents_waveforms.pdf')
1166 plt.savefig(plotfn)
1167 # done with ArcLink, remove ArcLink client
1168 del arcclient
1169 # done with iris, remove client
1170 del irisclient
1171 ### end of event loop ###
1172 # close event catalog info file and exception file
1173 catalogfout.close()
1174 exceptionfout.close()
1175 done = True
1176 return
APPENDIX C. SOURCE CODE: OBSYPLOAD.PY

def get_events(west, east, south, north, start, end, magmin, magmax):
    
    Downloads and saves a list of events if not present in datapath.
    
    Parameters
    ----------
    west : int or float, optional
        Minimum ("left-side") longitude.
        Format: +/- 180 decimal degrees.
    east : int or float, optional
        Maximum ("right-side") longitude.
        Format: +/- 180 decimal degrees.
    south : int or float, optional
        Minimum latitude.
        Format: +/- 90 decimal degrees.
    north : int or float, optional
        Maximum latitude.
        Format: +/- 90 decimal degrees.
    start : str, optional
        Earliest date and time.
    end : str, optional
        Latest date and time.
    magmin : int or float, optional
        Minimum magnitude.
    magmax : int or float, optional
        Maximum magnitude.
    
    Returns
    -------
    List of event dictionaries.
    
    eventfp = os.path.join(datapath, 'events.pickle')
    try:
        fh = open(eventfp, 'rb')
        result = pickle.load(fh)
        fh.close()
        print "Found eventlist in datapath, skip download."
    except:
        print "Downloading NERIES eventlist...",
        client = obspy.neries.Client()
        # the maximum no of allowed results seems to be not allowed to be too
        # large, but 9999 seems to work, 99999 results in a timeout error in
        # urllib. implemented the while-loop to work around this restriction:
        # query is repeated until we receive less than 9999 results.
        events = range(9999)
        while len(events) == 9999:
            events = client.getEvents(min_latitude=south, max_latitude=north,
                                       min_longitude=west, max_longitude=east,
                                       min_datetime=start, max_datetime=end,
                                       min_magnitude=magmin, max_magnitude=magmax,
                                       max_results=9999)
            result.extend(events)
        try:
            start = events[-1]['datetime']
except:
    pass

del client

# dump events to file
fh = open(eventfp, 'wb')
pickle.dump(result, fh)
fh.close()

print "done."
print("Received %d event(s) from NERIES." % (len(result)))
return result

def get_inventory(start, end, nw, st, lo, ch, permanent, debug=False):

    
    Searches the ArcLink inventory for available networks and stations.
    Because the ArcLink webservice does not support wildcard searches for
    networks (but for everything else), this method uses the re module to
    find * and ? wildcards in ArcLink networks and returns only matching
    network/station combinations.

    Parameters
    ----------
    start : str, optional
        ISO 8601-formatted, in UTC: yyyy-MM-dd['T'HH:mm:ss].
        e.g.: "2002-05-17" or "2002-05-17T05:24:00"
    end : str, optional
        ISO 8601-formatted, in UTC: yyyy-MM-dd['T'HH:mm:ss].
        e.g.: "2002-05-17" or "2002-05-17T05:24:00"

    Returns
    -------
    A list of tuples of the form [(station1, lat1, lon1), ...]
    
    # create data path:
    if not os.path.isdir(datapath):
        os.mkdir(datapath)
    inventoryfp = os.path.join(datapath, 'inventory.pickle')
    try:
        
        # first check if inventory data has already been downloaded
        fh = open(inventoryfp, 'rb')
        stations3 = pickle.load(fh)
        fh.close()
        print "Found inventory data in datapath, skip download."
        return stations3
    except:

        # first take care of network wildcard searches as arclink does not
        # support anything but '*' here:
        nwcheck = False
        if '*' in nw and nw != '*' or '?' in nw:
            if debug:
                print "we're now setting nwcheck = True"
        nw2 = '*'
        nwcheck = True
    else:
        nw2 = nw
        arcclient = obspy.arclink.client.Client()
        print "Downloading ArcLink inventory data....",
        # restricted = false, we don't want restricted data
        # permanent is handled via command line flag
        if debug:
            print "permanent flag: ", permanent
        try:
            inventory = arcclient.getInventory(network=nw2, station=st,
except Exception, error:
    print "download error: ", error
    print "ArcLink returned no stations."
    return ([], [])
else:
    print "done."
stations = sorted([i for i in inventory.keys() if i.count('.') == 3])
if debug:
    print "inventory inside get_inventory(): ", inventory
    print "stations inside get_inventory(): ", stations
# stations is a list of 'nw.st.lo.ch' strings and is what we want
# check if we need to search for wildcards:
if nwcheck:
    stations2 = []
    # convert nw (which is 'b?a*' type string, using normal wildcards into
    # equivalent regular expression
    # using fnmatch.translate to translate ordinary wildcard into regex.
    nw = fnmatch.translate(nw)
    if debug:
        print "regex nw: ", nw
    p = re.compile(nw, re.IGNORECASE)
    for i in range(len(stations)):
        # split every station('nw.st.lo.ch') by the . and take the first
        # object which is 'nw', search for the regular expression inside
        # this network string. if it matches, the if condition will be met
        # (p.match returns None if nothing is found)
        if p.match(stations[i].split('.')[0]):
            # everything is fine, we can return this station
            stations2.append(stations[i])
else:
    # just return the whole stations list otherwise
    stations2 = stations
    # include latitude and longitude for taup in the dict stations3, which will
    # be a list of tuples (station, lat, lon)
stations3 = []
    for station in stations2:
        # obtain key for station Attrib dict
        net, sta, loc, cha = station.split('.')
        key = ".".join((net, sta))
        stations3.append((station, inventory[key]['latitude'],
                          inventory[key]['longitude']))
    print("Received %d channel(s) from ArcLink." % (len(stations3)))
if debug:
    print "stations2 inside get_inventory: ", stations2
    print "stations3 inside get_inventory: ", stations3
    # dump result to file so we can quickly resume d/l if obspyload
    # runs in the same dir more than once. we're only dumping stations (the
    # regex matched ones, since only those are needed. see the try statement
    # above, if this file is found later, we don't have to perform the regex
    # search again.
    fh = open(inventoryfp, 'wb')
    pickle.dump(stations3, fh)
    fh.close()
    return stations3

def getnparse_availability(start, end, nw, st, lo, ch, debug):
    """
Downloads and parses IRIS availability XML.

```python
irisclient = obspy.iris.Client(debug=debug)

try:
    # create data path:
    if not os.path.isdir(datapath):
        os.mkdir(datapath)
    # try to load availability file
    availfp = os.path.join(datapath, 'availability.pickle')
    fh = open(availfp, 'rb')
    avail_list = pickle.load(fh)
    fh.close()
    print("Found IRIS availability in datapath, skip download.")
    return avail_list
except:
    print("Downloading IRIS availability data...")
    try:
        result = irisclient.availability(
            network=nw, station=st, location=lo,
            channel=ch, starttime=UTCDateTime(start),
            endtime=UTCDateTime(end), output='xml')
    except Exception, error:
        print("IRIS returned no matching stations.")
        if debug:
            print("iris client error: ", error)
        # return an empty list (iterable empty result)
        return []
    else:
        print("done.")
        print("Parsing IRIS availability xml to obtain nw.st.lo.ch...")
        availxml = etree.fromstring(result)
        if debug:
            print('availxml:
                 ', availxml)
        stations = availxml.findall('Station')
        # I will construct a list of tuples of stations of the form:
        # [(net,sta,cha,loc,lat,lon), (net,sta,loc,cha,lat,lon), ...]
        avail_list = []
        for station in stations:
            net = station.values()[0]
            sta = station.values()[1]
            # find latitude and longitude of station
            lat = float(station.find('Lat').text)
            lon = float(station.find('Lon').text)
            channels = station.findall('Channel')
            for channel in channels:
                loc = channel.values()[1]
                cha = channel.values()[0]
                if debug:
                    print('#### station/channel: ####
                        net', net
                        sta', sta
                        loc', loc
                        cha', cha
                # strip it so we can use it to construct nicer filenames
                # as well as to construct a working IRIS ws query
                avail_list.append((net.strip(' '), sta.strip(' '),
                    loc.strip(' '), cha.strip(' '), lat,
                    lon))
        # dump availability to file
        fh = open(availfp, 'wb')
        pickle.dump(avail_list, fh)
        fh.close()
        print "done."
if debug:
    print "avail_list: ", avail_list
print("Received %d station(s) from IRIS." % (len(stations)))
print("Received %d channel(s) from IRIS." % (len(avail_list)))
return avail_list

------------------------------------------------------------------------------------------------
# ALTERNATIVE MODES FUNCTIONS SECTION as described in the thesis #
------------------------------------------------------------------------------------------------

def queryMeta(west, east, south, north, start, end, nw, st, lo, ch, permanent, debug):
    ""
    Downloads Resp instrument data and dataless seed files.
    ""
    global quit, done, skip_networks
    # start keypress thread, so we can quit by pressing 'q' anytime from now on
    # during the downloads
    done = False
    keypress_thread().start()
    irisclient = obspy.iris.Client(debug=debug)
    arclinkclient = obspy.arclink.client.Client(debug=debug)
    # (0) get availability and inventory first
    # get and parse IRIS availability xml
    avail = getnparse_availability(start=start, end=end, nw=nw, st=st, lo=lo, ch=ch, debug=debug)
    # get ArcLink inventory
    stations = get_inventory(start, end, nw, st, lo, ch, permanent=permanent, debug=debug)
    # (1) IRIS: resp files
    # stations is a list of all stations (nw.st.l.ch, so it includes networks)
    # loop over all tuples of a station in avail list:
    for (net, sta, loc, cha) in avail:
        check_quit()
        respfn = '.'.join((net, sta, loc, cha)) + '.resp'
        respfnfull = os.path.join(datapath, respfn)
        if debug:
            print 'respfnfull:', respfnfull
            print 'type cha: ', type(cha)
            print 'length cha: ', len(cha)
            print 'net: %s sta: %s loc: %s cha: %s' % (net, sta, loc, cha)
        if os.path.isfile(respfnfull):
            print 'Resp file for %s exists, skip download...' % respfn
            continue
        print 'Downloading Resp file for %s from IRIS...' % respfn
        try:
            # initializing the client each time should reduce problems
            irisclient = obspy.iris.Client(debug=debug)
            irisclient.saveResponse(respfnfull, net, sta, loc, cha, start, end, format='RESP')
        except Exception, error:
            print "\ndownload error: ",
            print error
            continue
        else:
            # if there has been no exception, the d/l should have worked
            print 'done.'
# (2) ArcLink: dataless seed
# loop over stations to d/l every dataless seed file...
# skip dead ArcLink networks
for station in stations:
check_quit()
# we don't need lat and lon
station = station[0]
net, sta, loc, cha = station.split('.
# skip dead networks
if net in skip_networks:
    print 'Skipping dead network %s...' % net
    # continue the for-loop to the next iteration
    continue
# construct filename
dlseedfn = 'sl').join((net, sta, loc, cha)) + '.seed'
dlseedfnfull = os.path.join(datapath, dlseedfn)
# create data file handler
dlseedfnfull = os.path.join(datapath, "%s.mseed" % station)
if os.path.isfile(dlseedfnfull):
    print 'Dataless file for %s exists, skip download...' % dlseedfn
    continue
print 'Downloading dataless seed file for %s from ArcLink...
    % dlseedfn,
    try:
        # catch exception so the d/l continues if only one doesn't work
        # again, initializing the client should reduce problems
        arclinkclient = obspy.arclink.Client(debug=debug)
        arclinkclient.saveResponse(dlseedfnfull, net, sta, loc, cha,
                                   start, end, format='SEED')
    except Exception, error:
        print "download error: ",
        print error
        continue
    else:
        # if there has been no exception, the d/l should have worked
        print 'done.'
done = True
return

def exceptionMode(debug):
    ""
    This will read the file 'exceptions.txt' and try to download all the data
    that returned an exception other than 'no data available' last time.
    ""
    # initialize both clients, needed inside every loop.
arclinkclient = obspy.arclink.Client(timeout=5, debug=debug)
irisclient = obspy.iris.Client(debug=debug)
# read exception file
exceptionfp = os.path.join(datapath, 'exceptions.txt')
exceptionfin = open(exceptionfp, 'rt')
extceptions = exceptionfin.readlines()
extceptionfin.close()
# create further_exceptions string, this will be used to overwrite the
# exception file, but only after the process if done so we won't loose our
# original exceptions (exception file) if the user presses q while d/l
if debug:
    print "further_exceptions: ", further_exceptions
for exception in exceptions[3:]:
    check_quit()
    if debug:
        print "exception: ", exception
        exsplit = exception.split(';')
        if debug:
            print "exsplit: ", exsplit
        if not "data available" in exsplit[5]:
# we want to d/l this one again
if debug:
    print "passed no data available test."
eventid = exsplit[0]
station = exsplit[2]
net, sta, loc, cha = station.split('.')
starttime = UTCDateTime(exsplit[3])
endtime = UTCDateTime(exsplit[4])
datafout = os.path.join(datapath, eventid, station + '.mseed')
if debug:
    print "datafout: ", datafout
# check if ArcLink or IRIS
if exsplit[1] == "ArcLink":
    print "Trying to download event %s from ArcLink %s..." % 
        (eventid, station),
    try:
        arcclient = obspy.arclink.Client(timeout=5, debug=debug)
        arcclient.saveWaveform(filename=datafout, network=net,
            station=sta, location=loc, channel=cha,
            starttime=starttime, endtime=endtime)
    except Exception, error:
        print "download error: ", error
        # create exception file info line
        il_exception = str(eventid) + ';ArcLink;' + station + ';'
        il_exception += str(starttime) + ';\' + str(endtime) + ';'
        il_exception += str(error) + '\n'
        further_exceptions += il_exception
        continue
else:
    print "done."
elif exsplit[1] == "IRIS":
    print "Trying to download event %s from IRIS %s..." % 
        (eventid, station),
    try:
        irisclient = obspy.iris.Client(debug=debug)
        irisclient.saveWaveform(filename=datafout,
            network=net, station=sta,
            location=loc, channel=cha,
            starttime=starttime, endtime=endtime)
    except Exception, error:
        print "download error: ", error
        # create exception file info line
        il_exception = str(eventid) + ';IRIS;' + station + ';'
        il_exception += str(starttime) + ';\' + str(endtime) + ';'
        il_exception += str(error) + '\n'
        further_exceptions += il_exception
        continue
else:
    print "done."
exceptionfout = open(exceptionfp, 'wt')
exceptionfout.write(further_exceptions)
exceptionfout.close()
done = True
return

###########################################################
# ADDITIONAL FUNCTIONS SECTION as described in the thesis #
###########################################################
def travelTimePlot(npoints, phases, depth, model, pltWidth, pltHeight,}
timespan):
Plots taupe arrival times on top of event data. This is just a modified version of taupe.travelTimePlot().

:param npoints: int, optional
Number of points to plot.
:param phases: list of strings, optional
List of phase names which should be used within the plot. Defaults to all phases if not explicit set.
:param depth: float, optional
Depth in kilometer. Defaults to 100.
:param model: string

```
data = {}
for phase in phases:
    data[phase] = [[], []]
degrees = np.linspace(0, 180, npoints)
# Loop over all degrees.
for degree in degrees:
    tt = taup.getTravelTimes(degree, depth, model)
    # Mirror if necessary.
    if degree > 180:
        degree = 180 - (degree - 180)
    for item in tt:
        phase = item['phase_name']
        if phase in data:
            try:
                data[phase][1].append(item['time'])
                data[phase][0].append(degree)
            except:
                data[phase][1].append(np.NaN)
                data[phase][0].append(degree)
# Plot and some formatting.
for key, value in data.iteritems():  # value[0] stores all degrees, value[1] all times as lists
    # divide every entry of value[0] list by 180 and sort of "multiply with pltWidth" to get correct stmatrix indices
    # pltWidth* to get correct stmatrix index
    x_coord = map(operator.div, value[0], [180.0 / pltWidth] * len(value[0]))
    # for the y coord, divide every entry by the timespan and multiply with pltHeight
    y_coord = map(operator.div, value[1], [timespan / pltHeight] * len(value[1]))
    # plot arrival times on top of data
    plt.plot(x_coord, y_coord, ',', label=key)
plt.legend()
```

```
def getFolderSize(folder):
    """
    Returns the size of a folder in bytes.
    """
    total_size = os.path.getsize(folder)
    for item in os.listdir(folder):
        itempath = os.path.join(folder, item)
        if os.path.isfile(itempath):
            total_size += os.path.getsize(itempath)
        elif os.path.isdir(itempath):
            total_size += getFolderSize(itempath)
    return total_size
```
def printWrap(left, right, l_width=14, r_width=61, indent=2, separation=3):
    """
    Formats and prints a text output into 2 columns. Needed for the custom
    (long) help.
    """
    lefts = wrap(left, width=l_width)
    rights = wrap(right, width=r_width)
    results = []
    for l, r in izip_longest(lefts, rights, fillvalue=' '):
        results.append('{0:{1}}{2:{5}}{0:{3}}{4}'.format('', indent, l,
                  separation, r, l_width))
    print "\n".join(results)
    return

def help():
    """
    Print more help.
    """
    print "\nObsPyLoad: ObsPy Seismic Data Download tool."
    print """\nThe CLI allows for different flavors of usage, in short:
--------------------------------------------------------\n" + 
    printWrap("e.g.:", "obspyload.py -r <west>/<east>/<south>/<north> -t " + 
             "<start>/<end> -m <min_mag> -M <max_mag> -i <nw>.<st>.<l>.<ch>"
             )
    printWrap("e.g.:", "obspyload.py -y <min_lon> -Y <max_lon> " + 
             "-x <min_lat> -X <max_lat> -s <start> -e <end> -P <datapath> " + 
             "-o <offset> --reset -f")
    print "\n\nYou may (no mandatory options):
-------------------------------\n" + 
    printWrap("-r[--rect]", "<min.longitude>/<max.longitude>/<min.latitude>/<max.latitude>"
             )
    printWrap("", "e.g.: -r -15.5/40/30.8/50")
    print
    printWrap("x[-lonmin]", "<min.latitude>"
             )
    printWrap("X[-lonmax]", "<max.longitude>"
             )
    printWrap("y[-latmin]", "<min.latitude>"
             )
    printWrap("Y[-latmax]", "<max.latitude>"
             )
    printWrap("", "e.g.: +X +15.5 -Y 40.30.38 -Y 50")
    print
    printWrap("-t[--time]", "<start>/<end>"
             )
    printWrap("", "e.g.: -t 2007-12-31/2011-01-31"
             )
    print
    printWrap("s[--start]", "<starttime>"
             )
    printWrap("e[--end]", "<endtime>"
             )
    printWrap("", "e.g.: s 2007-12-31 -e 2011-01-31"
             )
    print
    printWrap("m[--magmin]", "<min.magnitude>"
             )
    printWrap("Default:", "minimum magnitude 3, no maximum magnitude.")
    printWrap("Format:", "Integer or decimal.")
    print
    printWrap("-n[--magmax]", "<min.magnitude>")
printWrap("-M[--magmax]", "<max.magnitude>")
printWrap("", "e.g. -m 4.2 -M 9")
printWrap("\n")
print("* specify a station restriction:
")
printWrap("Default:", "no constraints.")
printWrap("Format:", "Any station code, may include wildcards.")
printWrap("-i[--identity]", "<nw>.<st>.<l>.<ch>")
printWrap("", "e.g. -i IU.ANMO.00.BHZ")
printWrap("\n\n")
printWrap("* specify plotting options:
")
printWrap("Default:", "no plot. If the plot will be created with -I d " + \
"(or -I default), the defaults are 1200x800x1/100 and the " + \
"default phases to plot are 'P' and 'S'.")
printWrap("-I[--plot]", "<pxHeight>x<pxWidth>x<colWidth>[/<timespan>]")
printWrap("", "For each event, create one plot with the data from all " + \
"stations together with theoretical arrival times. You " + \
"may provide the internal plotting resolution: e.g. -I " + \
"900x600x5. This gives you a resolution of 900x600, and " + \
"5 units broad station columns. If -I d, or -I default, " + \
"the default of 1200x800x1 will be used. If this " + \
"command line parameter is not passed to ObsPyLoad at " + \
"all, no plots will be created. You may additionally " + \
"specify the timespan of the plot after event origin " + \
"time in minutes: e.g. for timespan lasting 30 minutes: " + \
"-I 1200x800x1/30 (or -I d/30). The default timespan is " + \
"100 minutes. The final output file will be in pdf " + \
"format.").")
printWrap("-F[--fill-plot]", "")
print("when creating the plot, download all the data needed " + \
"to fill the rectangular area of the plot. Note: " + \
"depending on your options, this will approximately " + \
"double the data download volume (but you'll end up " + \
"with nicer plots ;-)).")
printWrap("-a[--phases]", "<phase1>,<phase2>,...")
printWrap("", "Specify phases for which the theoretical arrival times " + \
"should be plotted on top if creating the data plot(see " + \
"above, -I option). " + \
"Default: -a P,S. To plot all available phases, use -a all. " + \
"If you just want to plot the data and no phases, use -a " + \
"none.")
printWrap("", "Available phases:")
printWrap("", "P, P'P'ab, P'P'bc, P'P'df, PKKPab, PKKPbc, " + \
"PKPdab, PKKSab, PKKSbc, PKKdab, PKPab, PKPbc, " + \
"PKPdaf, PKPddf, PKSab, PKSbc, PKSdab, PKKf, PKKP, " + \
"PP, PS, Pcp, Pcs, Pdiff, Pn, PnPn, PnS, " + \
"S, S'S'ac, S'S'df, SKKPab, SKKPbc, SKKpdf, " + \
"SKKsabc, SKKSdab, SKPab, SKPbc, SKPdab, Sksac, " + \
"SKSdab, SKKP, SP, Sp, SP, Sn, SS, SP, Scp, Sc, " + \
"Sdiff, Sn, SnSn, pP, pPKPab, pPKPbc, pPKPab, pPKPbc, " + \
"PKPdiff, pPKPdif, pPKKab, PKksab, PKKsdab, PKPd, PKKp, " + \
"pS, pS'S'ac, pS'S'df, pSKKPab, pSKKPbc, pSKKpdf, " + \
"pSKKSabc, pSKKSdab, pSKPab, pSKPbc, pSKPdab, pSKksac, " + \
"pSKSdab, pSKKP, pSP, pSP, pSN, pSS, pScp, pSc, " + \
"pSdiff, pSn, pSnSn, pP, pPKPab, pPKPbc, pPKPab, pPKPbc, " + \
"pPKPdiff, pPKPdif, pPKKab, pPKksab, pPKKsdab, pPKPd, " + \
"pS, pS'S'ac, pS'S'df, pSKKPab, pSKKPbc, pSKKpdf, " + \
"pSKKSabc, pSKKSdab, pSKPab, pSKPbc, pSKPdab, pSKksac, " + \
"pSKSdab, pSKKP, pSP, pSP, pSN, pSS, " + \
"pSdiff, pSn, pSnSn")
printWrap("", "Note: if you select phases with ticks(\') in the " + \
"
"phase name, don't forget to use quotes " + \
"(a \"phase1\",phase2\") to avoid unintended behaviour.")
print "\n\n* specify additional options:\n"printWrap("-n|--no-temporary", "")
printWrap("", "Instead of downloading both temporary and permanent " + \
"networks (default), download only permanent ones.")
print
printWrap("-a|--preset", "<preset>")
printWrap("", "Time parameter given in seconds which determines how " + \
"close the data will be cropped before estimated arrival time at " + \
"each individual station. Default: 5 minutes.")
print
printWrap("-p|--preset", "<preset>")
printWrap("", "Time parameter given in seconds which determines how " + \
"close the data will be cropped after estimated arrival time at " + \
"each individual station. Default: 80 minutes.")
print
printWrap("-o|--offset", "<offset>")
printWrap("", "Time parameter given in seconds which determines how " + \
"close the data will be cropped after estimated arrival time at " + \
"each individual station. Default: 80 minutes.")
print
printWrap("-q|--query-resp", "")
printWrap("", "Instead of downloading seismic data, download " + \
"instrument response files.")
print
printWrap("-P|--datapath", "<datapath>")
printWrap("", "Specify a different datapath, do not use do default one.")
print
printWrap("-R|--reset", "")
printWrap("", "If the datapath is found, do not resume previous " + \
"downloads as is the default behaviour, but redownload " + \
"everything. Same as deleting the datapath before running " + \
"ObsPyLoad.")
print
printWrap("-u|--update", "")
printWrap("", "Update the event database if ObsPyLoad runs on the " + \
"same directory for a second time.")
print
printWrap("-f|--force", "")
printWrap("", "Skip working directory warning (auto-confirm folder" + \
"creation).")
print "\nType obspyload.py -h for a list of all long and short options."
print "\n\nExamples:
"printWrap("---------
")
printWrap("Alps region, minimum magnitude of 4.2:","obspyload.py -r 5/16.5/45.75/48 -t 2007-01-13T08:24:00/+ \
"2011-02-25T22:41:00 -m 4.2")
print
printWrap("Sumatra region, Christmas 2004, different timestring, " + \
"mind the quotation marks:","obspyload.py -r 90/108/-7/7 -t "2004-12-24 01:23:45/+ \
"2004-12-26 12:34:56" -m 9")
print
printWrap("Mount Hochstaufen area(Ger/Aus), default minimum magnitude:","obspyload.py -r 12.8/12.9/47.72/47.77 -t 2001-01-01/2011-02-28")
print
printWrap("Only one station, to quickly try out the plot:","obspyload.py -s 2011-03-01 -m 9 -i IU.YSS.*.* -f " + \
"-i IU.YSS.*.*")
print
printWrap("ArcLink Network wildcard search:", "obspyload.py -N B? -S " + \
"FURT -f")
print
printWrap("Downloading metadata from all available stations " + \
"to folder \"metacatalog\":", "obspyload.py -q -f -P metacatalog")
print
printWrap("Download stations that failed last time " + \
"download: obspyload.py -f -P 2011-03-01")
APPENDIX C. SOURCE CODE: OBSPYLOAD.PY

1869 "(not necessary to re-enter the event/station restrictions):",
1870 "obspyload.py -E -P thisOrderHadExceptions -f")
1871 
1872 print
1873 return
1874
1875 if __name__ == "__main__":
1876     
1877     global quit
1878     # I could not get my interrupt handler to work. The plan was to capture
1879     # `c`, prevent the program from quitting immediately, finish the last
1879     # download and then quit. Perhaps someone could pick up on this.
1880     # It almost worked, but select.select couldn't restart after receiving
1881     # SIGINT. I have been told that's a bad design in the python bindings, but
1882     # that's above me. Had to give up.
1883     # Meanwhile, I think the method with 2 threads and pressing "q" instead
1884     # works fine.
1885     # The implementation uses class keypress_thread and function getkey(see
1886     # above).
1887     def interrupt_handler(signal, frame):
1888         global quit
1889         if not quit:
1890             print "You pressed `C (SIGINT)."
1891             msg = "ObsPyLoad will finish downloading and saving the last " + "\n1892             "file and quit gracefully."
1893             print msg
1894             print "Press `C again to interrupt immediately."
1895         else:
1896             msg = "Interrupting immediately. The last file will most likely"+ "\n1897             " be corrupt."
1898             sys.exit(2)
1899         quit = True
1900         signal.signal(signal.SIGINT, interrupt_handler)
1901         signal.siginterrupt(signal.SIGINT, False)
1902     
1903     global quit, done
1904     quit = False
1905     begin = time.time()
1906     status = main()
1907     size = getFolderSize(datapath)
1908     elapsed = time.time() - begin
1909     print "Downloaded %d bytes in %d seconds." % (size, elapsed)
1910     # sorry for the inconvenience, AFAIK there is no other way to quit the
1911     # second thread since getkey is waiting for input:
1912     print "Done, press any key to quit."
1913     # pass the return of main to the command line.
1914     sys.exit(status)
Appendix D

Supplementary CD

The attached CD contains this:

- **folder thesis**: all the \LaTeX{} and Figure files necessary to create this document and the Figures.
- **folder manual**: all the \LaTeX{} and Figure files necessary to create the manual.
- **folder examples**: example files mentioned in the Thesis.
- **folder source**: source code of ObsPyLoad.
Erklärung
Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebe-
enen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus an-
deren Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder
ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Declaration
I hereby declare that I wrote this thesis entirely on my own and have not used outside sources without decla-
ration in the text. Any concepts or quotations applicable to these sources are clearly attributed to them. This
thesis has not been submitted in the same or substantially similar version, not even in part, to any other au-
thority for grading.

Munich, August 1, 2011

Chris Scheingraber