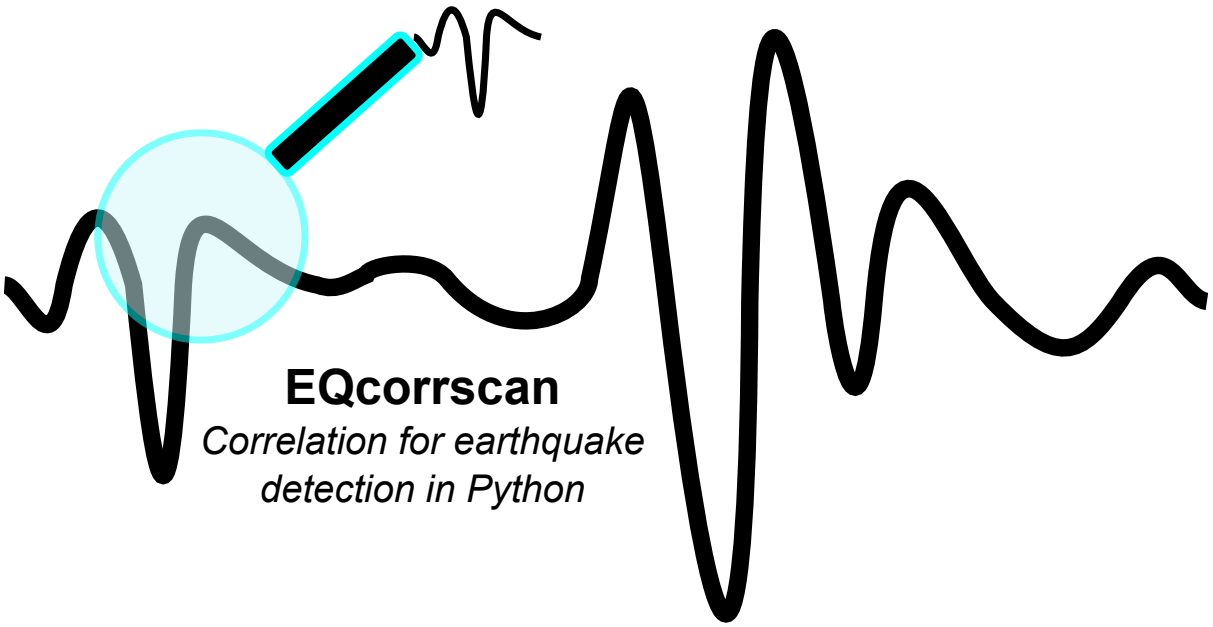


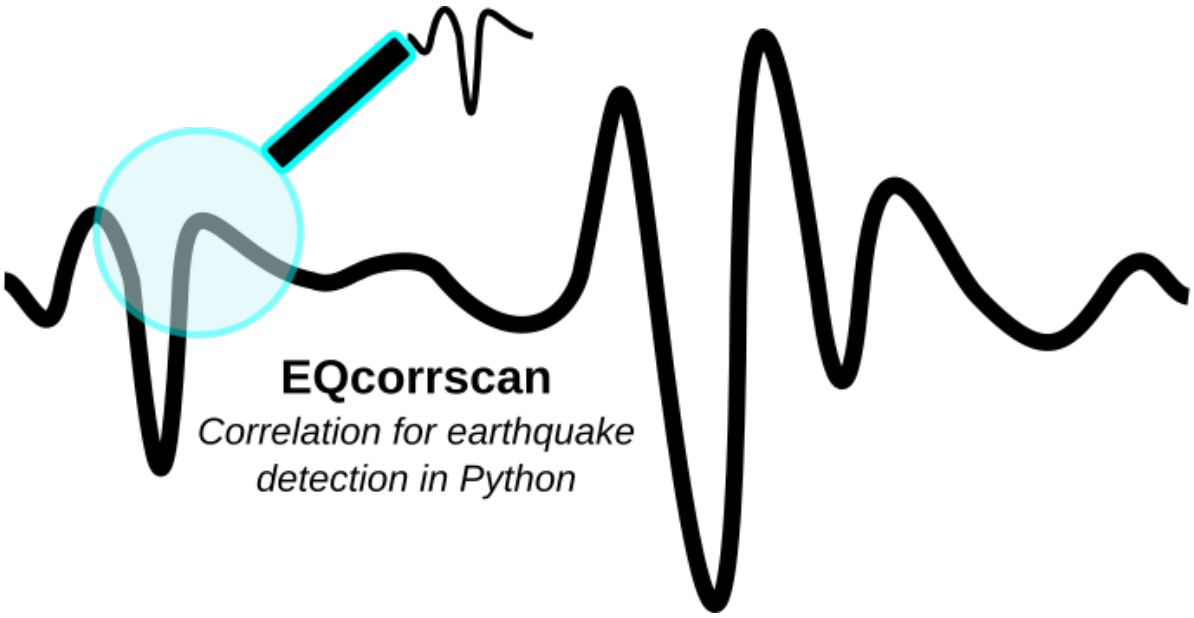
EQcorrscan Documentation

Release 0.4.3

Calum John Chamberlain

Dec 10, 2021

Contents



A Python package for the detection and analysis of repeating and near-repeating seismicity. EQcorrscan contains an efficient, multi-parallel, *matched-filter* detection routine (template-matching), as well as routines to implement *subspace* detection.

Code is stored on , the development branches are , or the latest stable release can be found .

EQcorrscan uses bindings when reading and writing seismic data, and for handling most of the event metadata, which ensures that detections can be easily migrated between software.

Also within this package are:

- *Correlation re-picking*;
- *Clustering routines for seismic data*;
- *Peak finding algorithm (basic)*;
- *Stacking routines* including phase-weighted stacking based on Thurber at al. (2014);

This package is written by the EQcorrscan developers, and is distributed under the LGPL GNU Licence, Copyright EQcorrscan developers 2017.

CHAPTER 1

Citation

If you use this package in your work, please cite the following paper: Chamberlain, C. J., Hopp, C. J., Boese, C. M., Warren-Smith, E., Chambers, D., Chu, S. X., Michailos, K., Townend, J., EQcorrscan: Repeating and near-repeating earthquake detection and analysis in Python. Seismological Research Letters, *2017*

CHAPTER 2

Contents:

2.1 Introduction to the EQcorrscan package

2.1.1 Supported environments

We support Linux, OSX and Windows environments running Python 3.x.

2.1.2 Functionality

Within EQcorrscan you will find the core routines to generate templates (*template_gen*), compute cross-channel correlations from these templates (*match_filter*), generate cross-correlation corrected pick-times (*lag_calc*), and run subspace detection (*subspace*).

2.1.3 Running tests

You can run tests yourself locally to ensure that everything runs as you would expect in your environment. Although we try to ensure that these tests run smoothly on all supported environments (using the ci bots), if you do find any issues, please let us know on the page.

To run the tests you will need to have pytest installed along with a couple of extras (pytest-pep8 and pytest-cov). These can be installed by pip:

```
pip install pytest pytest-pep8 pytest-cov
```

To test your installed version of EQcorrscan we provide a . For version<=0.3.2 you should download the script and run it. In later versions this script is included in the package.

This test-script will download the test data and run the tests (you no longer have to clone the git repository). Just run (from anywhere):

```
test_eqcorrscan.py
```

Tests will take about half an hour to run (as of version 0.3.2) and will provide a coverage report at the end and notify you of any failures.

2.2 EQcorrscan installation

EQcorrscan is a Python package with C extensions. The C extensions in EQcorrscan have their own dependencies on compiled libraries. We heavily recommend installing EQcorrscan using conda because this will:

- make your life easier;
- separate your EQcorrscan install from your system Python, meaning you can experiment to your hearts-content without breaking your operating system (yay);
- ensure that compiled modules are compiled using the correct C-compiler against the correct libraries

If you do not have either a miniconda or anaconda installation you can follow the instructions.

If you do not already have a conda environment we recommend creating one with the following:

```
conda create -n eqcorrscan -c conda-forge eqcorrscan
source activate eqcorrscan
```

This will create an environment called eqcorrscan and install eqcorrscan and its dependancies in that environment.

If you already have a conda environment that you want to use then to install EQcorrscan you can simply run:

```
conda install -c conda-forge eqcorrscan
```

2.2.1 Installation without conda

Installing EQcorrscan without conda involves two steps:

1. Installing fftw3 libraries;
2. Installing python dependancies and EQcorrscan.

How you undertake the first step depends on your operating system and system package manager.

Non-Python dependencies–Ubuntu:

Prior to installing the python routines you will need to install the fftw library. On linux use apt (or your default package manager - note you may need sudo access):

```
apt-get install libfftw3-dev
```

Note that you will need to ensure you have the single-precision libraries of fftw3 (files named fftw3f...). On CentOS you can install the *fftw-libs* package.

Non-Python dependencies–OSX:

For MacOS/OS-X systems we have tested using homebrew and macports (fink options are available, but we haven't tested them).

Homebrew

You will need a recent version of gcc (the homebrew gcc-4.9 port has issues with openMP). We have tested the following and found it to work (note that you may need to prepend sudo depending on your configuration):

```
brew install gcc6
brew install fftw
```

Then run the following to install EQcorrscan (note the need to select CC=gcc, you can install using clang, but you will need additional libraries for openmp support):

```
CC=gcc pip install eqcorrscan
```

MacPorts

The following has been tested and found to work (note that you may need to prepend sudo depending on your configuration):

1. Install an up-to-date gcc (gcc is needed for openmp compatibility) - any gcc should work (>4), here we use gcc6 for example:

```
port install gcc6
```

2. Install python from macports (tested for python35, but its up to you)

```
port install python35`
# optional: select python35 as default python for terminal:
port select --set python python35
```

3. Install numpy and pip from macports:

```
port install py35-numpy py35-pip
# optional, select pip35 as default pip
port select --set pip pip35
```

4. Install fftw3 from source:
    a. - link to fftw 3.3.7, most recent as of 10/01/2018
    b. unzip/untar
    c. Run the following from within the expanded directory:

```
./configure --enable-threads --enable-float && make
make install
./configure --enable-threads && make # Need both double and float␣
↪precision files
make install
```

5. Run: (if you didn't run the *port select –set pip pip35* command you will need to replace *pip* with *pip35*)

```
CC=gcc pip install eqcorrscan
```

Non-Python dependencies–Windows:

For Windows systems you should follow the instructions on the page and use the pre-compiled dynamic libraries. These should be installed somewhere on your system path, or the install location added to your path. The correlation routines use openMP for parallel workflows, however, some aspects of this run into issues with version of MSVC < 10.0 (due to old C standards being used), as such, by default, the correlation routines are compiled as serial workflows on windows. If you have a need for this threading in windows please get in touch with the developers.

EQcorrscan install via pip:

Once you have installed fftw the EQcorrscan install should be as simple as:

```
pip install eqcorrscan
```

Installation from source

pip pulls the package from the package repository and runs the *setup.py* file. If instead you wish to install from source, download the package (either by cloning the git repository, or by downloading the source code) from , change directory to the *EQcorrscan* directory and run:

```
python setup.py install
```

If this fails because the default compiler is *clang* you can run:

```
CC=gcc python setup.py install
```

Note though that this will compile EQcorrscan using a different compiler than used to build your Python, which may have unwanted effects, if you do this you MUST test you install using the instructions here: *Running tests*.

Using Intel's MKL

For versions >= 0.3.0 EQcorrscan supports compilation against the Intel Math Kernel Libraries (MKL). This has shown compared to the standard FFTW library. To enable this you must install MKL before compiling EQcorrscan. MKL is available from most package managers (including conda). Once you have MKL installed you can follow the *Installation from source* section. Check that near the top of the install that the MKL libraries are found.

2.2.2 Notes

You may have issues with these installs if you don't have numpy installed: but if you don't have numpy installed then you have bigger issues...

If you plan to generate a grid of synthetic templates you will need to have grid csv files, which the authors have previously used NonLinLoc to generate. This is not provided here and should be sourced from . This will provide the Grid2Time routine which is required to set-up a lag-time grid for your velocity model. You should read the NonLinLoc documentation for more information regarding how this process works and the input files you are required to give.

2.3 EQcorrscan FAQs

This is a developing list of frequently asked questions. If you find yourself experiencing a problem similar to one of these, then try the solution here first!

If your problem/question isn't answered here then check the issues on the github page and, if there isn't an issue related to your problem/question then open a new one and we will try to help.

2.3.1 Usage Questions

No output to terminal

EQcorrscan uses Logging to handle output. If you are seeing no output to screen you probably haven't set up your logging settings. A simple way to do this is to run code like:

You can set different levels to get more or less output (DEBUG is the most output, then INFO, then WARNNG, then ERROR, then CRITICAL). You will need to run this before any calls to EQcorrscan's functions to get logging output.

—

No correlations computed

Frequently the cause of no correlations being computed is that the SEED ID (network.station.location.channel) for your template do not match your continuous data. Check that they match, and try increasing the logging output (above) to help you find the issue.

—

Everything is done multiple times!

EQcorrscan uses multiprocessing under the hood, which will spawn multiple processes when called. To ensure that programs do not run multiple times you should always write your scripts with a form:

```
def main():
    # Do the mahi

if __name__ == "__main__":
    main()
```

See the multiprocessing note on "Safe importing of main module" for more info.

—

Making templates from SAC files

While there is support for making templates from SAC, it generally isn't a good idea, unless you have SAC waveforms of the length that you want to correlate with. This is for two reasons: 1. Because templates and continuous data *must* be processed the same, including using the same length FFT, for correlations to be accurate. If you make your template from short data you will be forced to correlate with short chunks of data, which is not very efficient 2. The programming team are not SAC users, and do not understand the nuances of where picks can be saved in SAC headers.

Instead: you might find it better to convert your SAC picks into another obspy readable pick/event format. You can try EQcorrscan's basic sac_utils to do this, but not everything will be retained.

2.3.2 Design Questions

Can I use different lengths for different channels in a template?

Not yet in EQcorrscan - we want this as well, but haven't had time to implement it. If you want this then we would really appreciate the contribution! There are two main issues with this that require some thought: 1) How correlations are computed, and 2) how correlations are weighted in the correlation sum.

Why doesn't EQcorrscan have a GUI?

This would be cool, and it would be great if someone wants to contribute this, however, the developers thus far have been focused on other things and don't have unlimited time.

If you want this, and know how to program GUIs then please do contribute, it would open EQcorrscan up to more users, which would be great!

Why do you have a functional and object-oriented API?

Simply legacy, when Calum started writing EQcorrscan he didn't know anything about classes. The functional API is retained so that old codes can still be run, but if you are starting from scratch please use the OO API where possible.

2.4 What's new

2.4.1 Version 0.4.2

- Add seed-ids to the _spike_test's message.
- utils.correlation - Cross-correlation normalisation errors no-longer raise an error - When "out-of-range" correlations occur a warning is given by the C-function

  with details of what channel, what template and where in the data vector the issue occurred for the user to check their data.

  - Out-of-range correlations are set to 0.0
  - After extensive testing these errors have always been related to data issues within regions where correlations should not be computed (spikes, step artifacts due to incorrectly padding data gaps).
  - USERS SHOULD BE CAREFUL TO CHECK THEIR DATA IF THEY SEE THESE WARNINGS
- utils.mag_calc.amp_pick_event - Added option to output IASPEI standard amplitudes, with static amplification

  of 1 (rather than 2080 as per Wood Anderson specs).

  - Added *filter_id* and *method_id* to amplitudes to make these methods more traceable.
- core.match_filter - Bug-fix - cope with data that are too short with *ignore_bad_data=True*.

  This flag is generally not advised, but when used, may attempt to trim all data to zero length. The expected behaviour is to remove bad data and run with the remaining data.

  - Party: - decluster now accepts a hypocentral_separation argument. This allows

    the inclusion of detections that occur close in time, but not in space. This is underwritten by a new findpeaks.decluster_dist_time function based on a new C-function.
  - Tribe: - Add monkey-patching for clients that do not have a *get_waveforms_bulk*

    method for use in *.client_detect*. See issue #394.
- utils.pre_processing - Only templates that need to be reshaped are reshaped now - this can be a lot

  faster.

2.4.2 Version 0.4.1

- core.match_filter - BUG-FIX: Empty families are no longer run through lag-calc when using Party.lag_calc(). Previously this resulted in a "No matching data" error, see #341.
- core.template_gen - BUG-FIX: Fix bug where events were incorrectly associated with templates in *Tribe().construct()* if the given catalog contained events outside of the time-range of the stream. See issue #381 and PR #382.
- utils.catalog_to_dd - Added ability to turn off parallel processing (this is turned off by default now) for *write_correlations* - parallel processing for moderate to large datasets was copying far too much data and using lots of memory. This is a short-term fix - ideally we will move filtering and resampling to C functions with shared-memory parallelism and GIL releasing. See PR #374. - Moved parallelism for *_compute_dt_correlations* to the C functions to reduce memory overhead. Using a generator to construct subcatalogs rather than making a list of lists in memory. See issue #361.
- utils.mag_calc: - *amp_pick_event* now works on a copy of the data by default - *amp_pick_event* uses the appropriate digital filter gain to correct the applied filter. See issue #376. - *amp_pick_event* rewritten for simplicity. - *amp_pick_event* now has simple synthetic tests for accuracy. - *_sim_wa* uses the full response information to correct to velocity this includes FIR filters (previously not used), and ensures that the wood-anderson poles (with a single zero) are correctly applied to velocity waveforms. - *calc_max_curv* is now computed using the non-cumulative distribution.

- Some problem solved in _match_filter_plot. Now it shows all new detections.
- Add plotdir to eqcorrscan.core.lag_calc.lag_calc function to save the images.

2.4.3 Version 0.4.0

- Change resampling to use pyFFTW backend for FFT's. This is an attempt to alleviate issue related to large-prime length transforms. This requires an additional dependency, but EQcorrscan already depends on FFTW itself (#316).
- Refactor of catalog_to_dd functions (#322): - Speed-ups, using new correlation functions and better resource management - Removed enforcement of seisan, arguments are now standard obspy objects.
- Add plotdir to lag-calc, template construction and matched-filter detection methods and functions (#330, #325).
- Wholesale re-write of lag-calc function and methods. External interface is similar, but some arguments have been depreciated as they were unnecessary (#321). - This was done to make use of the new internal correlation functions which

  are faster and more memory efficient.

  - Party.lag_calc and Family.lag_calc now work in-place on the events in the grouping.
  - Added relative_mags method to Party and Family; this can be called from lag-calc to avoid reprocessing data.
  - Added lag_calc.xcorr_pick_family as a public facing API to implement correlation re-picking of a group of events.
- Renamed utils.clustering.cross_chan_coherence to utils.clustering.cross_chan_correlation to better reflect what it actually does.
- Add –no-mkl flag for setup.py to force the FFTW correlation routines not to compile against intels mkl. On NeSI systems mkl is currently causing issues.
- BUG-FIX: *eqcorrscan.utils.mag_calc.dist_calc* calculated the long-way round the Earth when changing hemispheres. We now use the Haversine formula, which should give better results at short distances, and does not use a flat-Earth approximation, so is better suited to larger distances as well.
- Add C-openmp parallel distance-clustering (speed-ups of ~100 times).
- Allow option to not stack correlations in correlation functions.
- Use compiled correlation functions for correlation clustering (speed-up).
- Add time-clustering for catalogs and change how space-time cluster works so that it uses the time-clustering, rather than just throwing out events outside the time-range.
- Changed all prints to calls to logging, as a result, debug is no longer an argument for function calls.
- *find-peaks* replaced by compiled peak finding routine - more efficient both in memory and time #249 - approx 50x faster * Note that the results of the C-func and the Python functions are slightly

  different. The C function (now the default) is more stable when peaks are small and close together (e.g. in noisy data).
- multi-find peaks makes use of openMP parallelism for more efficient memory usage #249
- enforce normalization of continuous data before correlation to avoid float32 overflow errors that result in correlation errors (see pr #292).
- Add SEC-C style chunked cross-correlations. This is both faster and more memory efficient. This is now used by default with an fft length of 2 ** 13. This was found to be consistently the fastest length in testing. This can be changed by the user by passing the *fft_len* keyword argument. See PR #285.
- Outer-loop parallelism has been disabled for all systems now. This was not useful in most situations and is hard to maintain.

- Improved support for compilation on RedHat systems
- Refactored match-filter into smaller files. Namespace remains the same. This was done to ease maintenance - the match_filter.py file had become massive and was slow to load and process in IDEs.
- Refactored *_prep_data_for_correlation* to reduce looping for speed, now approximately six times faster than previously (minor speed-up) * Now explicitly doesn't allow templates with different length traces -

  previously this was ignored and templates with different length channels to other templates had their channels padded with zeros or trimmed.
- Add *skip_short_channels* option to template generation. This allows users to provide data of unknown length and short channels will not be used, rather than generating an error. This is useful for downloading data from datacentres via the *from_client* method.
- Remove pytest_namespace in conftest.py to support pytest 4.x
- Add *ignore_bad_data* kwarg for all processing functions, if set to True (defaults to False for continuity) then any errors related to bad data at process-time will be supressed and empty traces returned. This is useful for downloading data from datacentres via the *from_client* method when data quality is not known.
- Added relative amplitude measurements as *utils.mag_calc.relative_amplitude* (#306).
- Added relative magnitude calculation using relative amplitudes weighted by correlations to *utils.mag_calc.relative_magnitude*.
- Added *relative_magnitudes* argument to *eqcorrscan.core.match_filter.party.Party.lag_calc* to provide an in-flow way to compute relative magnitudes for detected events.
- Events constructed from detections now include estimated origins alongside the picks. These origins are time-shifted versions of the template origin and should be used with caution. They are corrected for prepick (#308).
- Picks in detection.event are now corrected for prepick *if* the template is given. This is now standard in all Tribe, Party and Family methods. Picks will not be corrected for prepick in match_filter (#308).
- Fix #298 where the header was repeated in detection csv files. Also added a *write_detections* function to *eqcorrscan.core.match_filter.detection* to streamline writing detections.
- Remove support for Python 2.7.
- Add warning about unused data when using *Tribe.detect* methods with data that do not fit into chunks. Fixes #291.
- Fix #179 when decimating for cccsum_hist in *_match_filter_plot*
- *utils.pre_processing* now uses the *.interpolate* method rather than *.resample* to change the sampling rate of data. This is generally more stable and faster than resampling in the frequency domain, but will likely change the quality of correlations.
- Removed depreciated *template_gen* functions and *bright_lights* and *seismo_logs*. See #315

—

2.4.4 Older Versions

Version 0.3.3

- Make test-script more stable - use the installed script for testing.
- Fix bug where *set_xcorr* as context manager did not correctly reset stream_xcorr methods.
- Correct test-script (*test_eqcorrscan.py*) to find paths properly.
- BUG-FIX in *Party.decluster* when detections made at exactly the same time the first, rather than the highest of these was taken.
- Catch one-sample difference in day properly in pre-processing.dayproc

- Shortproc now clips and pads to the correct length asserted by starttime and endtime.
- Bug-fix: Match-filter collection objects (Tribe, Party, Family) implemented addition (*__add__*) to alter the main object. Now the main object is left unchanged.
- *Family.catalog* is now an immutable property.

Version 0.3.2

- Implement reading Party objects from multiple files, including wildcard expansion. This will only read template information if it was not previously read in (which is a little more efficient).
- Allow reading of Party objects without reading the catalog files.
- Check quality of downloaded data in *Tribe.client_detect()* and remove it if it would otherwise result in errors.
- Add *process_cores* argument to *Tribe.client_detect()* and *Tribe.detect()* to provide a separate number of cores for processing and peak-finding - both functions are less memory efficient that fftw correlation and can result in memory errors if using lots of cores.
- Allow passing of *cores_outer* kwarg through to fftw correlate functions to control inner/outer thread numbers. If given, *cores* will define the number of inner-cores (used for parallel fft calculation) and *cores_outer* sets the number of channels to process in parallel (which results in increased memory usage).
- Allow Tribe and Party IO to use QUAKEML or SC3ML format for catalogs (NORDIC to come once obspy updates).
- Allow Party IO to not write detection catalogs if so desired, because writing and reading large catalogs can be slow.
- If detection-catalogs are not read in, then the detection events will be generated on the fly using *Detection._calculate_event*.
- BUG-FIX: When one template in a set of templates had a channel repeated, all detections had an extra, spurious pick in their event object. This should no-longer happen.
- Add *select* method to *Party* and *Tribe* to allow selection of a specific family/template.
- Add ability to "retry" downloading in *Tribe.client_detect*.
- Change behaviour of template_gen for data that are daylong, but do not start within 1 minute of a day-break - previous versions enforced padding to start and end at day-breaks, which led to zeros in the data and undesirable behaviour.
- BUG-FIX: Normalisation errors not properly passed back from internal fftw correlation functions, gaps not always properly handled during long-period trends - variance threshold is now raised, and Python checks for low-variance and applies gain to stabilise correlations if needed.
- Plotting functions are now tested and have a more consistent interface:
  - All plotting functions accept the keyword arguments *save*, *savefile*, *show*, *return_figure* and *title*.
  - All plotting functions return a figure.
  - *SVD_plot* renamed to *svd_plot*
- Enforce pre-processing even when no filters or resampling is to be done to ensure gaps are properly processed (when called from *Tribe.detect*, *Template.detect* or *Tribe.client_detect*)
- BUG-FIX in *Tribe.client_detect* where data were processed from data one sample too long resulting in minor differences in data processing (due to difference in FFT length) and therefore minor differences in resulting correlations (~0.07 per channel).
  - Includes extra stability check in fftw_normxcorr which affects the last sample before a gap when that sample is near-zero.

- BUG-FIX: fftw correlation dot product was not thread-safe on some systems. The dot-product did not have the inner index protected as a private variable. This did not appear to cause issues for Linux with Python 3.x or Windows, but did cause issues for on Linux for Python 2.7 and Mac OS builds.
- KeyboardInterrupt (e.g. ctrl-c) should now be caught during python parallel processes.
- Stopped allowing outer-threading on OSX, clang openMP is not thread-safe for how we have this set-up. Inner threading is faster and more memory efficient anyway.
- Added testing script (*test_eqcorrscan.py*, which will be installed to your path on installation of EQcorrscan) that will download all the relevant data and run the tests on the installed package - no need to clone EQcorrscan to run tests!

Version 0.3.1

- Cleaned imports in utils modules
- Removed parallel checking loop in archive_read.
- Add better checks for timing in lag-calc functions (#207)
- Removed gap-threshold of twice the template length in *Tribe.client_detect*, see issue #224.
- Bug-fix: give multi_find_peaks a cores kwarg to limit thread usage.
- Check for the same value in a row in continuous data when computing correlations and zero resulting correlations where the whole window is the same value repeated (#224, #230).
- BUG-FIX: template generation *from_client* methods for swin=P_all or S_all now download all channels and return them (as they should). See #235 and #206
- Change from raising an error if data from a station are not long enough, to logging a critical warning and not using the station.
- Add ability to give multiple *swin* options as a list. Remains backwards compatible with single *swin* arguments.
- Add option to *save_progress* for long running *Tribe* methods. Files are written to temporary files local to the caller.
- Fix bug where if gaps overlapped the endtime set in pre_processing an error was raised - happened when downloading data with a deliberate pad at either end.

Version 0.3.0

- Compiled peak-finding routine written to speed-up peak-finding.
- Change default match-filter plotting to not decimate unless it has to.
- BUG-FIX: changed minimum variance for fftw correlation backend.
- Do not try to process when no processing needs to be done in core.match_filter._group_process.
- Length checking in core.match_filter._group_process done in samples rather than time.
- BUG-FIX: Fix bug where data lengths were not correct in match_filter.Tribe.detect when sampling timestamps were inconsistent between channels, which previously resulted in error.
- BUG-FIX: Fix memory-leak in tribe.construct
- Add plotting options for plotting rate to Party.plot
- Add filtering detections by date as Party.filter
- BUG-FIX: Change method for Party.rethreshold: list.remove was not reliable.
- Add option *full_peaks* to detect methods to map to find_peaks.

- pre-processing (and match-filter object methods) are now gap-aware and will accept gappy traces and can return gappy traces. By default gaps are filled to maintain backwards compatibility. Note that the fftw correlation backend requires gaps to be padded with zeros.
- **Removed sfile_utils** This support for Nordic IO has been upgraded and moved to obspy for obspy version 1.1.0. All functions are there and many bugs have been fixed. This also means the removal of nordic-specific functions in EQcorrscan - the following functions have been removed: * template_gen.from_sfile * template_gen.from_contbase * mag_calc.amp_pick_sfile * mag_calc.pick_db All removed functions will error and tell you to use obspy.io.nordic.core. This now means that you can use obspy's *read_events* to read in sfiles.
- Added *P_all* and *S_all* options to template generation functions to allow creation of multi-channel templates starting at the P and S times respectively.
- Refactored *template_gen*, all options are available via *template_gen(method=...)*, and depreciation warnings are in place.
- Added some docs for converting older templates and detections into Template and Party objects.

Version 0.2.7

- Patch multi_corr.c to work with more versions of MSVC;
- Revert to using single-precision floats for correlations (as in previous, < 0.2.x versions) for memory efficiency.

Version 0.2.6

- Added the ability to change the correlation functions used in detection methods through the parameter xcorr_func of match_filter, Template.detect and Tribe.detect, or using the set_xcorr context manager in the utils.correlate module. Supported options are: * numpy * fftw * time-domain * or passing a function that implements the xcorr interface.
- Added the ability to change the concurrency strategy of xcorr functions using the paramter concurrency of match_filter, Template.detect and Tribe.detect. Supported options are: * None - for single-threaded execution in a single process * multithread - for multi-threaded execution * multiprocess- for multiprocess execution * concurrent - allows functions to describe their own preferred currency methods, defaults to multithread
- Change debug printing output, it should be a little quieter;
- Speed-up time-domain using a threaded C-routine - separate from frequency domain C-routines;
- Expose useful parallel options for all correlation routines;
- Expose cores argument for match-filter objects to allow limits to be placed on how much of your machine is used;
- Limit number of workers created during pre-processing to never be more than the number of traces in the stream being processed;
- Implement openMP parallelisation of cross-correlation sum routines - memory consumption reduced by using shared memory, and by computing the cross-correlation sums rather than individual channel cross-correlations. This also leads to a speed-up. This routine is the default concurrent correlation routine;
- Test examples in rst doc files to ensure they are up-to-date;
- Tests that were prone to timeout issues have been migrated to run on circleci to allow quick re-starting of fails not due to code errors

Version 0.2.5

- Fix bug with _group_process that resulted in stalled processes.
- Force NumPy version
- Support indexing of Tribe and Party objects by template-name.
- Add tests for lag-calc issue with preparing data
- Change internals of *eqcorrscan.core.lag_calc._prepare_data* to use a dictionary for delays, and to work correctly! Issues arose from not checking for masked data properly and not checking length properly.
- Fix bug in match_filter.match_filter when checking for equal length traces, length count was one sample too short.

Version 0.2.4

- Increase test coverage (edge-cases) in template_gen;
- Fix bug in template_gen.extract_from_stack for duplicate channels in template;
- Increase coverage somewhat in bright_lights, remove non-parallel option (previously only used for debugging in development);
- Increase test coverage in lag_calc;
- Speed-up tests for brightness;
- Increase test coverage for match_filter including testing io of detections;
- Increase subspace test coverage for edge cases;
- Speed-up catalog_to_dd_tests;
- Lag-calc will pick S-picks on channels ending E, N, 1 and 2, change from only picking on E and N before; warning added to docs;
- Add full tests for pre-processing;
- Run tests in parallel on ci, speed-up tests dramatically;
- Rename singular-value decomposition functions (with depreciation warnings);
- Rename SVD_moments to lower-case and add depreciation warning;
- Increase test coverage in utils.mag_calc;
- Add Template, Tribe, Family, Party objects and rename DETECTION to Detection * Template objects maintain meta-data associated with their creation to stream-line processing of data (e.g. reduce chance of using the wrong filters). * Template events have a detect method which takes unprocessed data and does the correct processing using the Template meta-data, and computes the matched-filter detections. * Tribe objects are containers for multiple Templates. * Tribe objects have a detect method which groups Templates with similar meta-data (processing information) and runs these templates in parallel through the matched-filter routine. Tribe.detect outputs a Party of Family objects. * The Party object is a container for many Family objects. * Family objects are containers for detections from the same Template. * Family and Party objects have a lag_calc method which computes the cross-correlation pick-refinements. * The upshot of this is that it is possible to, in one line, generate a Tribe of templates, compute their matched-filter detections, and generate cross-correlation pick refinements, which output Event objects, which can be written to a catalog: Tribe.construct(method, **kwargs).detect(st, **kwargs).lag_calc(**kwargs).write() * Added 25 tests for these methods. * Add parameters *threshold_type* and *threshold_input* to Detection class. Add support for legacy Detection objects via NaN and unset values.
- Removed support for obspy < 1.0.0
- Update / correct doc-strings in template-gen functions when describing processing parameters.
- Add warning message when removing channels from continuous data in match_filter;

- Add min_snr option for template generation routines, if the signal-to-noise ratio is below a user-defined threshold, the channel will not be used.
- Stop enforcing two-channel template channel names.
- Fix bug in detection_multiplot which didn't allow streams with fewer traces than template;
- Update internals to custom C fftw-based correlation rather than openCV (Major change); * OpenCV has been removed as a dependancy; * eqcorrscan.core.match_filter.normxcorr2 now calls a compiled C routine; * Parallel workflows handled by openMP rather than Python Multiprocessing for matched-filter operations to allow better memory handling. * It is worth noting that we tried re-writing using SciPy internals which led to a significant speed-up, but with high memory costs, we ended up going with this option, which was the more difficult option, because it allows effective use on SLURM managed systems where python multiprocessing results in un-real memory spikes (issue #88).

Version 0.2.0-0.2.3

- See 0.2.4: these versions were not fully released while trying to get anaconda packages to build properly.

Version 0.1.6

- Fix bug introduced in version 0.1.5 for match_filter where looping through multiple templates did not correctly match image and template data: 0.1.5 fix did not work;
- Bug-fix in catalog_to_dd for events without magnitudes;
- Amend match-filter to not edit the list of template names in place. Previously, if a template was not used (due to no matching continuous data) then the name of the template was removed: this now copies the list of template_names internally and does not change the external list.

Version 0.1.5

- Migrate coverage to codecov;
- Fix bug introduced in version 0.1.5 for match_filter where looping through multiple templates did not correctly match image and template data.

Version 0.1.4

- Bug-fix in plot_repicked removed where data were not normalized properly;
- Bug-fix in lag_calc where data were missing in the continuous data fixed (this led to incorrect picks, **major bug!**);
- Output cross-channel correlation sum in lag-calc output;
- Add id to DETECTION objects, which is consistent with the events within DETECTION objects and catalog output, and used in lag_calc to allow linking of detections to catalog events;
- Add lots of logging and error messages to lag-calc to ensure user understands limits;
- Add error to day-proc to ensure user is aware of risks of padding;
- Change utils.pre_processing.process to accept different length of data enforcement, not just full day (allow for overlap in processing, which might be useful for reducing day start and end effects);
- Bug-fix in mag_calc.amp_pick_event, broke loop if data were missing;
- Lots of docs adjustment to sort order of doc-strings and hyper-links;
- Allow multiple uses of the same channel in templates (e.g. you can now use a template with two windows from the same channel, such as a P and an S);

- Add evaluation mode filter to utils.catalog_utils.filter_picks;
- Update subspace plot to work when detector is not partitioned;
- Make tests run a little faster;
- Add pep8 testing for all code.

Version 0.1.3

- Now testing on OSX (python 2.7 and 3.5) - also added linux python 3.4;
- Add lag-calculation and tests for it;
- Change how lag-calc does the trace splitting to reduce memory usage;
- Added pick-filtering utility to clean up tutorials;
- Change template generation function names for clarity (wrappers for depreciated names);
- Add more useful error messages when picks are not associated with waveforms;
- Add example plots for more plotting functions;
- Add subspace detector including docs and tutorial.
- Add *delayed* option to all template_gen functions, set to True by default which retains old behaviour.

Version 0.1.2

- Add handling for empty location information in sfiles;
- Added project setup script which creates a useful directory structure and copies a default match-filter script to the directory;
- Add archive reader helper for default script, and parameter classes and definitions for default script;
- Re-write history to make repository smaller, removed trash files that had been added carelessly;
- Now tested on appveyor, so able to be run on Windows;
- Added ability to read hypoDD/tomoDD phase files to obspy events;
- Added simple despiking algorithm - not ideal for correlation as spikes are interpolated around when found: eqcorrscan.utils.despike;
- Option to output catalog object from match_filter - this will become the default once we introduce meta-data to templates - currently the picks for events are the template trace start-times, which will be before the phase-pick by the lag defined in the template creation - also added event into detection class, so you can access the event info from the detections, or create a catalog from a list of detections;
- Add option to extract detections at run-time in match_filter.match_filter;
- Edited multi_event_singlechan to take a catalog with multiple picks, but requires you to specify the station and channel to plot;
- Add normalize option to stacking routines;
- Add tests for stacking - PWS test needs more checks;
- Add many examples to doc-strings, not complete though;
- Change docs to have one page per function.
- Python 3.5 testing underway, all tests pass, but only testing about 65% of codebase.
- Add io functions to match_filter to simplify detection handling including writing detections to catalog and to text file.
- Stricter match_filter testing to enforce exactly the same result with a variety of systems.

- Add hack to template_gen tutorial to fix differences in sorting between python 3.x and python 2.
- Added advanced network triggering routine from Konstantinos, allows different parameters for individual stations - note only uses recursive sta-lta triggering at the moment. Useful for template generations alongside pickers.
- Added magnitude of completeness and b-value calculators to utils.mag_calc

Version 0.1.1

- Cope with events not always having time_errors in them in eventtoSfile;
- Convert Quakeml depths from m to km;
- Multiple little fixes to make Sfile conversion play well with GeoNet QuakeML files;
- Add function to convert from obspy.core.inventory.station.Station to string format for Seisan STATION0.HYP file;
- Merged feature branch - hypoDD into develop, this provides mappings for the hypoDD location program, including generation of dt.cc files;
- Added tests for functions in catalog_to_dd;
- Implemented unittest tests;
- Changed name of EQcorrscan_plotting to plotting;
- Added depreciation warnings;
- Changed internal structure of pre-processing to aid long-term upkeep;
- Added warnings in docs for template_gen relating to template generation from set length files;
- Updated template_creation tutorial to use day-long data;
- Renamed Sfile_util to sfile_util, and functions there-in: will warn about name changes;
- Updated template plotting to include pick labels;
- Updated template_creation tutorial to download S-picks as well as P-picks;
- Update sfile_util to cope with many possible unfilled objects;
- Added sac_util to convert from sac headers to useful event information - note, does not convert all things, just origin and pick times;
- Added from_sac function to template_gen.

2.5 EQcorrscan tutorials

Welcome to EQcorrscan - this package is designed to compute earthquake detections using a paralleled matched-filter network cross-correlation routine, and analyse the results.

EQcorrscan is divided into two main sub-modules, the **core** and **utils** sub-modules. The core sub-module contains the main, high-level functions:

template_gen A series of routines to generate templates for match-filter detection from continuous or cut data, with pick-times either defined manually, or defined in event files;

match_filter The main matched-filter routines, this is split into several smaller functions to allow python-based parallel-processing;

subspace Subspace detection routine based on .

lag_calc Routines for calculating optimal lag-times for events detected by the match-filter routine, these lags can then be used to define new picks for high accuracy re-locations.

Some other high-level functions are included in the *utils* sub-module and are documented here with tutorials:

mag_calc Simple local magnitude calculation and high-precision relative moment calculation using singular-value decomposition.

clustering Routines for clustering earthquakes based on a range of metircs using agglomorative clustering methods.

The **utils** sub-module contains useful, but small functions. These functions are rarely cpu intensive, but perform vital operations, such as finding peaks in noisy data (*findpeaks*), converting a database to hypoDD formatted files and computing cross-correlations between detections for (a double difference relocation software) (*catalog_to_dd*), calculating magnitudes (*mag_calc*), clustering detections (*clustering*), stacking detections (*stacking*), making plots (*plotting*), and processing seismic data in the same way repeatedly using 's functionality (*pre_processing*).

As of EQcorrscan v.0.4.0, output is controlled by the module. This allows the user to decide now much output is needed (via the *level* option), and where output should be sent (either to a file, stdout, or both). By default, if no logging parameters are set before calling EQcorrscan functions only messages of WARNING or above are printed to stdout. To make things a little prettier you can begin your work using something like:

```
import logging

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s\t%(name)s\t%(levelname)s\t%(message)s")
```

This will output messages at level INFO and above (quite verbose), in messages like:

```
2018-06-25 22:48:17,113        eqcorrscan.utils.pre_processing INFO    Working on:␣
↪NZ.CPWZ.10.EHZ
```

For more options, see the guide.

The following is an expanding set of tutorials that should take you through some of the key functionality of the EQcorrscan package. The tutorials are a (slow) work in progress to convert to jupyter notebook. If you have anything you want to see in the tutorials please let us know on github.

2.5.1 Quick Start

The goal of this notebook is to provide you with a quick overview of some of the key features of EQcorrscan. From this you should be able to dig deeper into the API to learn some of the more advanced features.

Matched-filter earthquake detection

One of the main applications of EQcorrscan is to detect earthquakes or other seimic signals using matched-filters (aka template-matching). To undertake matched-filtering templates are required. Templates are simply processed waveforms. In EQcorrscan templates are realised as `Template` objects, which contain the waveform data and meta-data associated with their creation. Groups of `Template` objects are stored in `Tribe` objects. To demonstrate this, we will create a `Tribe` for the Parkfield 2004 earthquake.

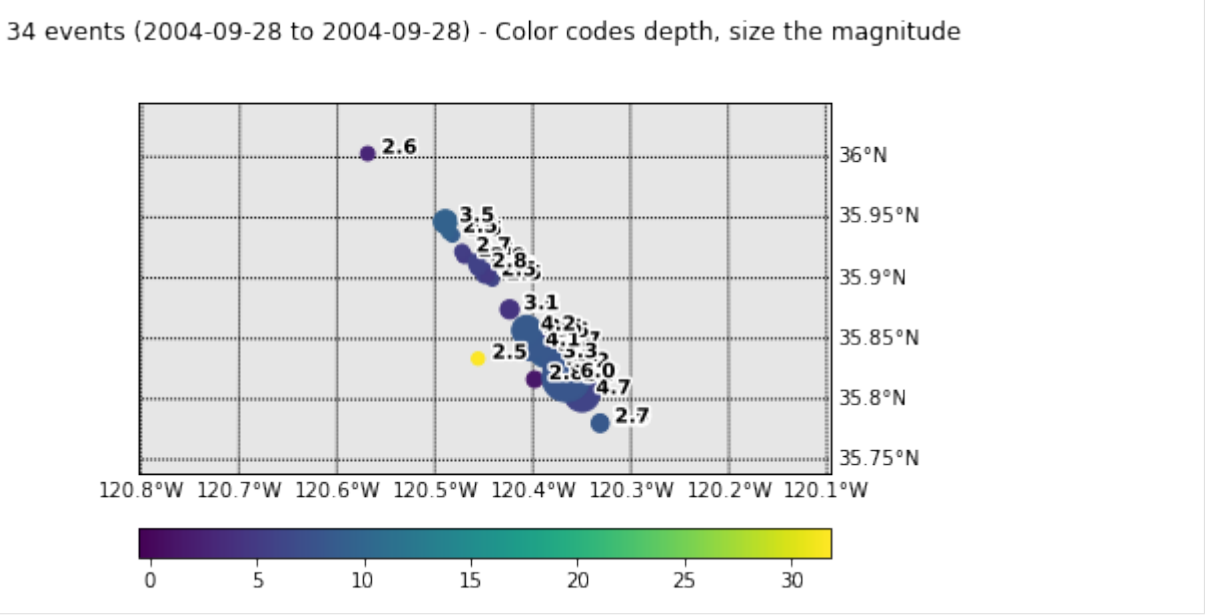
Before we begin we will set up logging - EQcorrscan provides textual output through the native Python logging library. The standard `DEBUG`, `INFO`, `WARNING`, `ERROR` and `CRITICAL` levels are supported. For this we will set to `ERROR` to reduce the output. When starting off you might want to set to `INFO`, or if something is happening that you don't expect, `DEBUG` will give you much more output.

[1]:
```
import logging

logging.basicConfig(
    level=logging.ERROR,
    format="%(asctime)s\t%(name)s\t%(levelname)s\t%(message)s")
```

First we need a catalog of events. We will just use a few earthquakes from the NCEDC. We are also only going to used picks from the 20 most picked stations.

[2]:
```
%matplotlib inline
from obspy import UTCDateTime
from obspy.clients.fdsn import Client
from eqcorrscan.utils.catalog_utils import filter_picks

client = Client("NCEDC")
t1 = UTCDateTime(2004, 9, 28)
t2 = t1 + 86400
catalog = client.get_events(
    starttime=t1, endtime=t2, minmagnitude=2.5, minlatitude=35.7, maxlatitude=36.1,
    minlongitude=-120.6, maxlongitude=-120.2, includearrivals=True)
fig = catalog.plot(projection="local", resolution="h")
catalog = filter_picks(catalog=catalog, evaluation_mode="manual", top_n_picks=20)
```

```
/home/calumch/miniconda3/envs/conda_37/lib/python3.7/site-packages/obspy/imaging/
↪maps.py:387: MatplotlibDeprecationWarning:
The dedent function was deprecated in Matplotlib 3.1 and will be removed in 3.3.␣
↪Use inspect.cleandoc instead.
  width=width, height=height, ax=ax)
/home/calumch/miniconda3/envs/conda_37/lib/python3.7/site-packages/obspy/imaging/
↪maps.py:435: MatplotlibDeprecationWarning:
The dedent function was deprecated in Matplotlib 3.1 and will be removed in 3.3.␣
↪Use inspect.cleandoc instead.
  bmap.drawcountries(color="0.75")
```



When data are available via a `Client`, you can generate templates directly from the client without having to download data yourself. However, if you have data locally you can pass that data. Have a look at the notebook on template creation for more details on other methods.

[3]:
```
from eqcorrscan import Tribe

tribe = Tribe().construct(
    method="from_client", lowcut=4.0, highcut=15.0, samp_rate=50.0, length=6.0,
    filt_order=4, prepick=0.5, client_id=client, catalog=catalog, data_pad=20.,
    process_len=21600, min_snr=5.0, parallel=True)
print(tribe)
```

```
Tribe of 34 templates
```

This makes a tribe by:

1. Downloading data for each event (if multiple events occur close in time they will share data;
2. Detrending, filtering and resampling the data;
3. Trimming the data around picks within each event in the catalog.

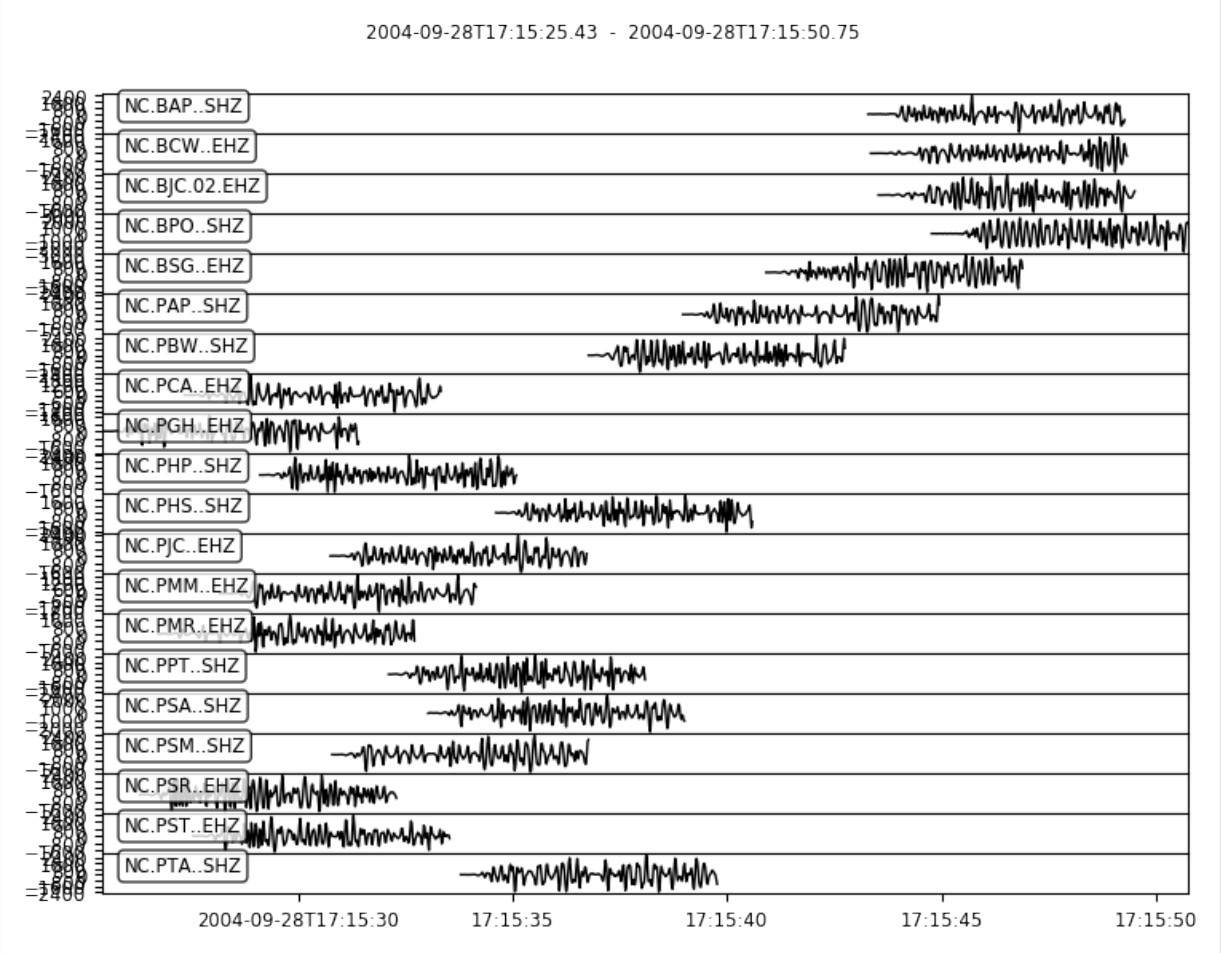
To make this `Tribe` we used a few different arguments:

- `lowcut`: This controls the lower corner frequency of a filter in Hz. EQcorrscan natively uses Obspy's butterworth filter functions. For more details, see the Obspy docs. This can optionally be set to `None` to not apply filtering at low frequencies.
- `highcut`: This controls the upper corner frequency of a filter in Hz. It is applied in tandem with lowcut, or can be set to `None` to not apply filtering at high frequencies.
- `samp_rate`: The desired sampling-rate of the templates in Hz. EQcorrscan requires that templates and continuous data be sampled at the same rate. In this case we are downsampling data from 100.0 Hz to 50.0 Hz. We do this to reduce memory-load. EQcorrscan uses Obspy's resampling method
- `length`: This is the length of the template waveform on each channel in seconds.
- `filt_order`: This sets the number of corners/order of the filter applied using `highcut` and `lowcut`.
- `prepick`: The number of seconds prior to a phase-pick to start a template waveform.
- `client_id`: This is the `Client` or client ID (e.g. our client id would be `"NCEDC"`) from which to download the data from.
- `catalog`: The catalog of events to generate templates for.
- `data_pad`: A fudge-factor to cope with data centres not providing quite the data you asked for. This is the number of seconds extra to download to cope with this.
- `process_len`: The length of data to download in seconds for each template. This should be the same as the length of your continuous data, e.g. if you data are in day-long files, this should be set to `86400`.
- `min_snr`: The minimum signal-to-noise ratio required for a channel to be included in a template. This provides a simple way of getting rid of poor-quality data from your templates.
- `parallel`: Whether to process the data in parallel or not. This will increase memory load, if you find yourself running out of memory, set `parallel=False` for these template construction methods.

Lets have a quick look at the attributes of a `Template`:

[4]:
```
print(tribe[0])
fig = tribe[0].st.plot(equal_scale=False, size=(800, 600))
```

```
Template 2004_09_28t17_15_25:
         20 channels;
         lowcut: 4.0 Hz;
         highcut: 15.0 Hz;
         sampling rate 50.0 Hz;
         filter order: 4;
         process length: 21600.0 s
```



We can see that this template has data from multiple stations, and that our processing parameters are stored alongside the template waveform. This means that we can ensure that when we work on continuous data to run matched-filtering the data are processed in the same way.

Lets remove templates with fewer than five stations then use this tribe to detect earthquakes within the first few days of the Parkfield aftershock sequence:

[5]:
```
tribe.templates = [t for t in tribe if len({tr.stats.station for tr in t.st}) >= 5]
print(tribe)
party, st = tribe.client_detect(
    client=client, starttime=t1, endtime=t1 + (86400 * 2), threshold=9.,
    threshold_type="MAD", trig_int=2.0, plot=False, return_stream=True)
```

```
Tribe of 28 templates
```

This will:

1. Download the data required;
2. Process the data in the same way as the templates in the `Tribe` (note that your tribe can contain templates processed in different ways: these will be grouped into similarly processed templates and the matched-filter process will be run multiple times; once for each group of similar templates);
3. Cross-correlate the templates with the processed data and stack the correlations;
4. Detect within each templates stacked cross-correlation array based on a given threshold;
5. Generate `Detection` objects with Obspy events and meta-data

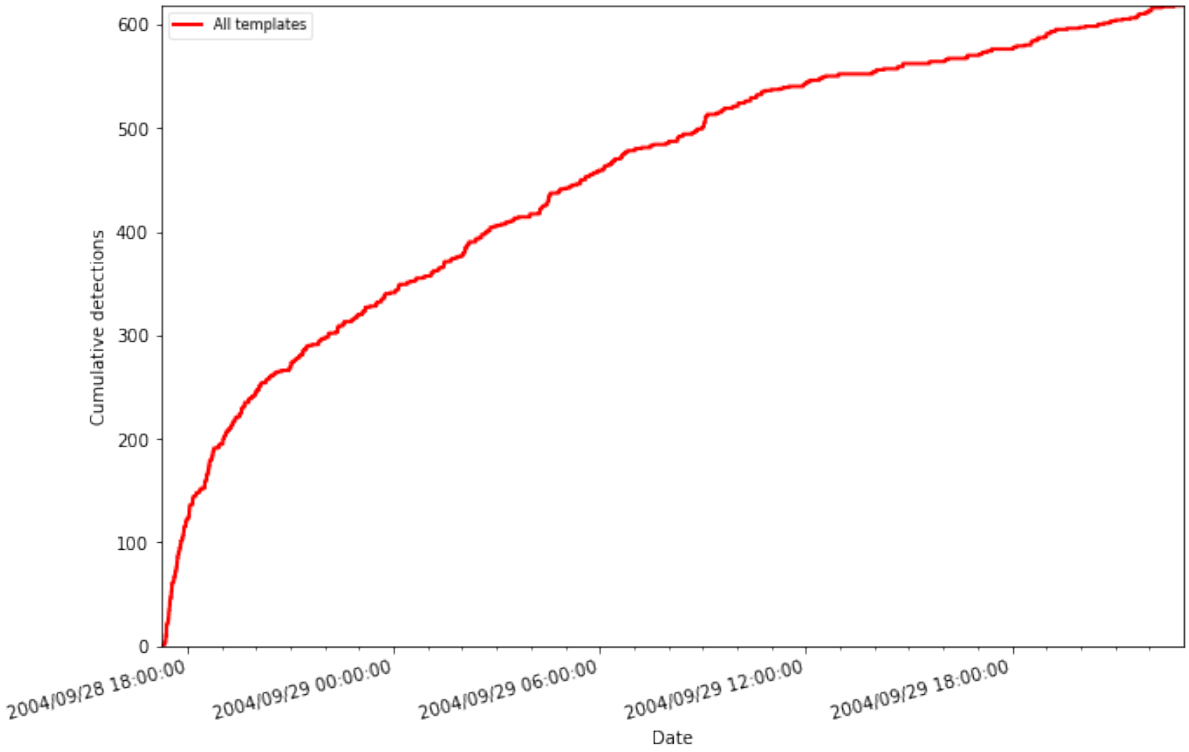
Again, we used a few arguments here:

- `client`: This again is the client from which to download data;

- `starttime`: The start of the data we want to detect within;
- `endtime`: The end of the data we want to detect within;
- `threshold`: The threshold for detection: note that the correlation sum is both positive and negative valued. Detections are made based on the absolute of the correlation vector, so both postively and negatively correlated detections are made;
- `threshold_type`: The type of threshold to use, in this case we used `"MAD"`, the median absolute deviation, our `threshold` is the MAD multiplier;
- `trig_int`: The minimum time (in seconds) between triggers. The highest absolute correlation value will be used within this window if multiple possible triggers occur. Note that this only applies within the detections from one template.
- `plotvar`: We turned plotting off in this case;
- `return_stream`: Setting this to `True` will return the pre-processed stream downloaded from the client, allowing us to reuse this stream for later processing.

Lets have a look at the cumulative detections we made.

[6]:
```
fig = party.plot(plot_grouped=True)
```



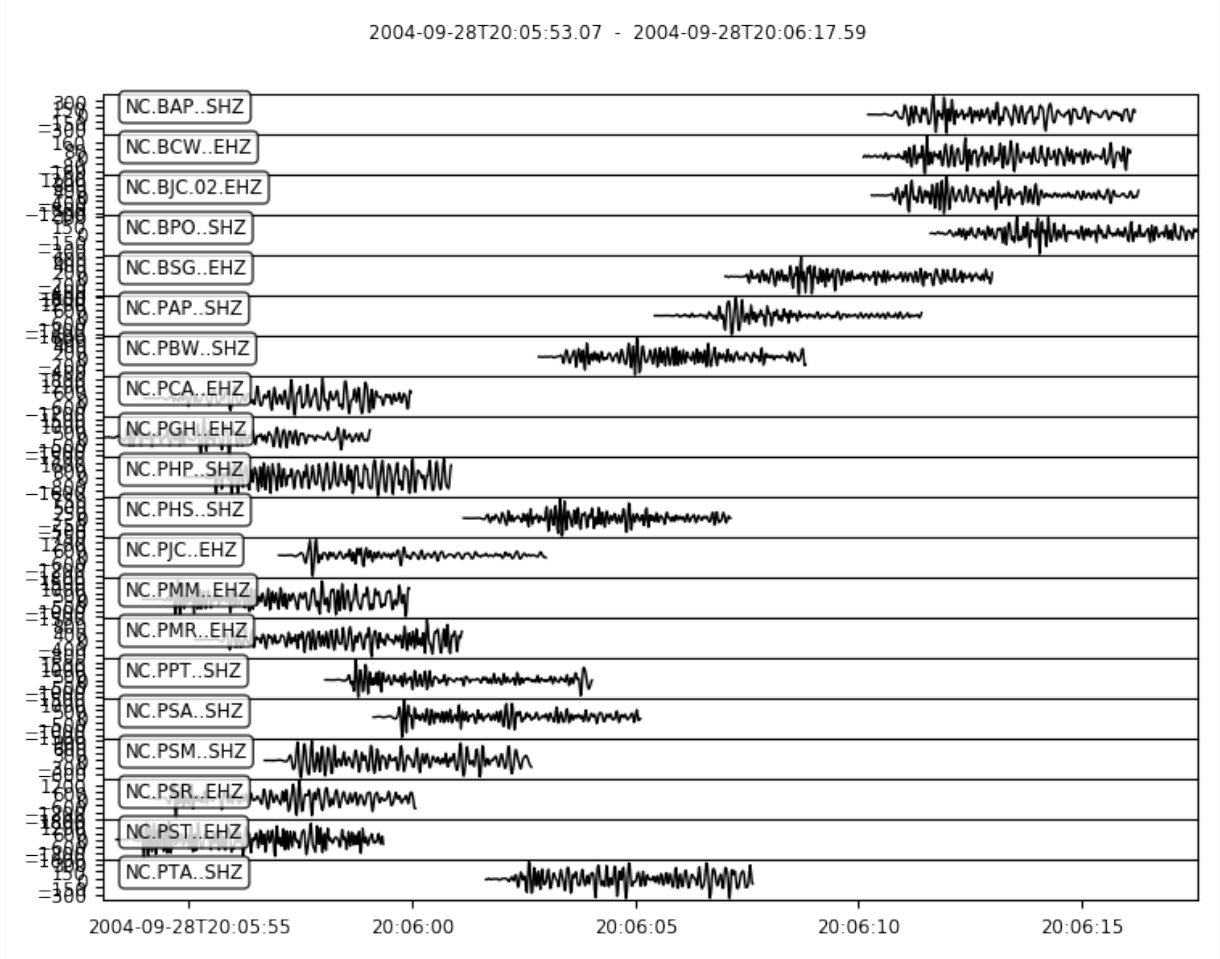
We were returned a `Party` and a `Stream`. The `Party` is a container for `Family` objects. Each `Family` contains a `Template` and all the detections associated with that `Template`. Detections are stored as `Detection` objects.

Lets have a look at the most productive family:

[7]:
```
family = sorted(party.families, key=lambda f: len(f))[-1]
print(family)
```

```
Family of 106 detections from template 2004_09_28t20_05_53
```

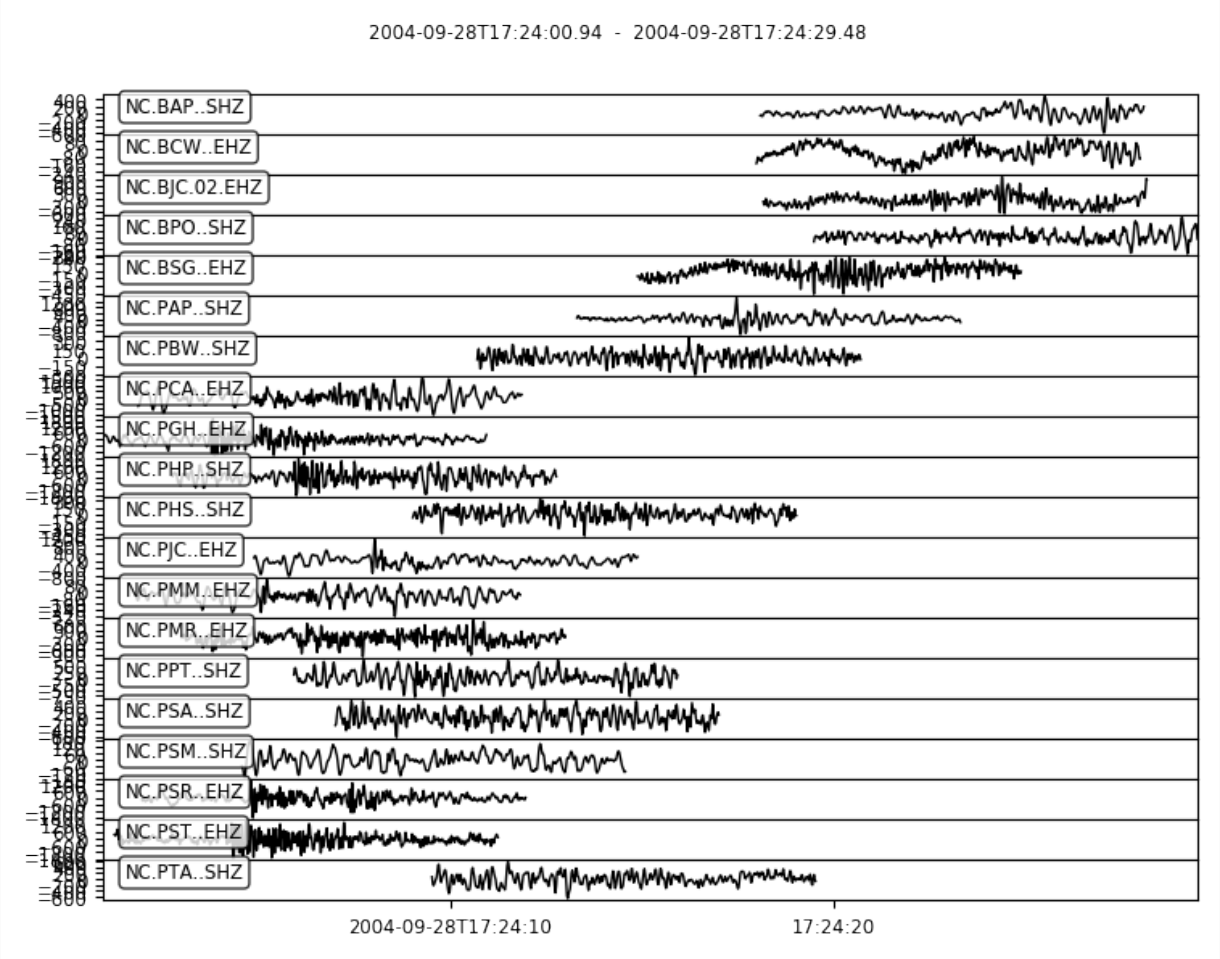
[8]:
```
fig = family.template.st.plot(equal_scale=False, size=(800, 600))
```



We can get a dictionary of streams for each detection in a `Family` and look at some of those:

[9]:
```
streams = family.extract_streams(stream=st, length=10, prepick=2.5)
print(family.detections[0])
fig = streams[family.detections[0].id].plot(equal_scale=False, size=(800, 600))
```

```
Detection on template: 2004_09_28t20_05_53 at: 2004-09-28T17:24:03.440000Z with 20␣
↪channels: [('BAP', 'SHZ'), ('BCW', 'EHZ'), ('BJC', 'EHZ'), ('BPO', 'SHZ'), ('BSG
↪', 'EHZ'), ('PAP', 'SHZ'), ('PBW', 'SHZ'), ('PCA', 'EHZ'), ('PGH', 'EHZ'), ('PHP
↪', 'SHZ'), ('PHS', 'SHZ'), ('PJC', 'EHZ'), ('PMM', 'EHZ'), ('PMR', 'EHZ'), ('PPT
↪', 'SHZ'), ('PSA', 'SHZ'), ('PSM', 'SHZ'), ('PSR', 'EHZ'), ('PST', 'EHZ'), ('PTA
↪', 'SHZ')]
```



On some stations we can clearly see the earthquake that was detected. Remember that these data have not been filtered.

We can also undertake cross-correlation phase-picking using our templates. This is achieved using the `lag_calc` method on `Party` or `Family` objects. You can also use the `eqcorrscan.core.lag_calc` module directly on other catalogs.

We need to provide a merged stream to the lag-calc function. The stream we have obtained from `tribe.detect` has overlaps in it. Using `stream.merge()` would result in gaps at those overlaps, which we do not want. We use `stream.merge(method=1)` to take include the real data in that gap.

[10]:
```
st = st.merge(method=1)
repicked_catalog = party.lag_calc(st, pre_processed=False, shift_len=0.5, min_cc=0.
↪4)
```

```
/home/calumch/miniconda3/envs/conda_37/lib/python3.7/site-packages/obspy/signal/
↪detrend.py:35: RuntimeWarning: invalid value encountered in true_divide
  data -= x1 + np.arange(ndat) * (x2 - x1) / float(ndat - 1)
```

This returns a catalog with picks for each detected event. The arguments we have used are:

- The stream that we downloaded previously;
- `pre_processed`: Our data have not been processed, so we rely on `lag_calc` to process our data - this means that our data will be processed in the same way as our templates were;
- `shift_len`: This is the maximum allowed shift (positive and negative) of the pick-time relative to the template moveout in seconds.
- `min_cc`: The minimum normalised cross-correlation value required to generate a pick. Picks are made on the maximum correlation within a window.

Lets look at one of the events that we have picked. The correlation value for the pick is stored in a comment:

[20]:
```
print(repicked_catalog[100].picks[0])
```

```
Pick
             resource_id: ResourceIdentifier(id="smi:local/08139539-1fec-4707-b60e-
↪b87f68ecfb9e")
                    time: UTCDateTime(2004, 9, 28, 18, 45, 22, 820000)
             waveform_id: WaveformStreamID(network_code='NC', station_code='BPO',␣
↪channel_code='SHZ', location_code='')
               method_id: ResourceIdentifier(id="EQcorrscan")
              phase_hint: 'P'
         evaluation_mode: 'automatic'
           creation_info: CreationInfo(agency_id='eqcorrscan.core.lag_calc')
                    ---------
                comments: 1 Elements
```

[21]:
```
print(repicked_catalog[100].picks[0].comments[0])
```

```
Comment(text='cc_max=0.439941', resource_id=ResourceIdentifier(id="smi:local/
↪383b3803-a499-407b-bf1c-8d78ab4b73d1"))
```

Note:

Using long templates like this that include both P and S energy does not usually produce good correlation derived phase arrivals. This is simply a demonstration. You should carefully select your template length, and ideally include both P and S picks in the catalog that you use to make templates.

Explore the rest of the tutorials and API docs for further hints.

2.5.2 Matched-filter detection

This tutorial will cover using both the match-filter objects, and using the internal functions within match-filter. The match-filter objects are designed to simplify meta-data handling allowing for shorter code with fewer mistakes and therefore more consistent results.

Match-filter objects

The match-filter module contains five objects:

- Tribe
- Template
- Party
- Family
- Detection

The Tribe object is a container for multiple Template objects. Templates contain the waveforms of the template alongside the metadata used to generate the template. Both Templates and Tribes can be written to disk as tar archives containing the waveform data in miniseed format, event catalogues associated with the Templates (if provided) in quakeml format and meta-data in a csv file. This archives can be read back in or transferred between machines.

The Detection, Family and Party objects are heirachical, a single Detection object describes a single event detection, and contains information regarding how the detection was made, what time it was made at alongside other useful information, it does not store the Template object used for the detection, but does store a reference to the name of the Template. Family objects are containers for multiple Detections made using a single Template (name chosen to match the literature). These objects do contain the Template used for the detections, and as such can be used to re-create the list of detections is necessary. Party objects are containers for multiple Family objects. All

objects in the detection heirachy have read and write methods - we recommend writing to tar archives (default) for Party and Family objects, as this will store all metadata used in detection, which should allow for straightforward reproduction of results.

Template creation

Templates have a construct method which accesses the functions in *template_gen*. Template.construct only has access to methods that work on individual events, and not catalogs; for that use the Tribe.construct method. For example, we can use the *from_sac* method to make a Template from a series of SAC files associated with a single event:

```
>>> import glob
>>> from eqcorrscan.core.match_filter import Template
>>> import os
>>> from eqcorrscan import tests
>>> # Get the path for the test-data so we can test this
>>> TEST_PATH = os.path.dirname(tests.__file__)
>>> sac_files = glob.glob(TEST_PATH + '/test_data/SAC/2014p611252/*')
>>> # sac_files is now a list of all the SAC files for event id:2014p611252
>>> template = Template().construct(
...       method='from_sac', name='test', lowcut=2.0, highcut=8.0,
...       samp_rate=20.0, filt_order=4, prepick=0.1, swin='all',
...       length=2.0, sac_files=sac_files)
```

Tribe creation

As eluded to above, Template.construct only works for individual events, to make a lot of templates we have to use the Tribe.construct method. The syntax is similar, but we don't specify names - templates are named according to their start-time, but you can rename them later if you wish:

```
>>> from eqcorrscan.core.match_filter import Tribe
>>> from obspy.clients.fdsn import Client

>>> client = Client('NCEDC')
>>> catalog = client.get_events(eventid='72572665', includearrivals=True)
>>> # To speed the example we have a catalog of one event, but you can have
>>> # more, we are also only using the first five picks, again to speed the
>>> # example.
>>> catalog[0].picks = catalog[0].picks[0:5]
>>> tribe = Tribe().construct(
...       method='from_client', catalog=catalog, client_id='NCEDC', lowcut=2.0,
...       highcut=8.0,  samp_rate=20.0, filt_order=4, length=6.0, prepick=0.1,
...       swin='all', process_len=3600, all_horiz=True)
```

Matched-filter detection using a Tribe

Both Tribe and Template objects have *detect* methods. These methods call the main match_filter function. They can be given an un-processed stream and will complete the appropriate processing using the same processing values stored in the Template objects. Because Tribe objects can contain Templates with a range of processing values, this work is completed in groups for groups of Templates with the same processing values. The Tribe object also has a client_detect method which will download the appropriate data. Both *detect* and *client_detect* methods return Party objects.

For example, we can use the Tribe we created above to detect through a day of data by running the following:

```
>>> from obspy import UTCDateTime

>>> party, stream = tribe.client_detect(
```

(continues on next page)

(continued from previous page)

```
...       client=client, starttime=UTCDateTime(2016, 1, 2),
...       endtime=UTCDateTime(2016, 1, 3), threshold=8, threshold_type='MAD',
...       trig_int=6, plotvar=False, return_stream=True)
```

Generating a Party from a Detection csv

If you are moving from detections written out as a csv file from an older version of EQcorrscan, but want to use Party objects now, then this section is for you!

First, you need to generate a Tribe from the templates you used to make the detections. Instructions for this are in the *Template creation tutorial* section.

Once you have a Tribe, you can generate a Party using the following:

```
>>> detections = read_detections(detection_file)
>>> party = Party()
>>> for template in tribe:
...     template_detections = [d for d in detections
...                            if d.template_name == template.name]
...     family = Family(template=template, detections=template_detections)
...     party += family
```

Lag-calc using a Party

Because parties contain Detection and Template information they can be used to generate re-picked catalogues using lag-calc:

```
>>> stream = stream.merge().sort(['station'])
>>> repicked_catalog = party.lag_calc(stream, pre_processed=False,
...                                   shift_len=0.2, min_cc=0.4)
```

By using the above examples you can go from a standard catalog available from data centers, to a matched-filter detected and cross-correlation repicked catalog in a handful of lines.

Simple example - match-filter.match-filter

This example does not work out of the box, you will have to have your own templates and data, and set things up for this. However, in principle matched-filtering can be as simple as:

```
from eqcorrscan.core.match_filter import match_filter
from eqcorrscan.utils import pre_processing
from obspy import read

# Read in and process the daylong data
st = read('continuous_data')
# Use the same filtering and sampling parameters as your template!
st = pre_processing.dayproc(
    st, lowcut=2, highcut=10, filt_order=4, samp_rate=50,
    starttime=st[0].stats.starttime.date)
# Read in the templates
templates = []
template_names = ['template_1', 'template_2']
for template_file in template_names:
     templates.append(read(template_file))
detections = match_filter(
     template_names=template_names, template_list=templates, st=st,
     threshold=8, threshold_type='MAD', trig_int=6, plotvar=False, cores=4)
```

This will create a list of detections, which are of class detection. You can write out the detections to a csv (colon separated) using the detection.write method, set *append=True* to write all the detections to one file. Beware though, if this is set and the file already exists, it will just add on to the old file.

```
for detection in detections:
    detection.write('my_first_detections.csv', append=True)
```

Data gaps and how to handle them

Data containing gaps can prove problematic for normalized cross-correlation. Because the correlations are normalized by the standard deviation of the data, if the standard deviation is low, floating-point rounding errors can occur. EQcorrscan tries to avoid this in two ways:

1. In the 'eqcorrscan.utils.correlate' (fftw) functions, correlations are not computed when the variance of the data window is less than 1e-10, or when there are fewer than *template_len - 1* non-flat data values (e.g. at-least one sample that is not in a gap), or when the mean of the data multiplied by the standard deviation of the data is less than 1e-10.
2. The *pre_processing* functions fill gaps prior to processing, process the data, then edit the data within the gaps to be zeros. During processing aliased signal will appear in the gaps, so it is important to remove those artifacts to ensure that gaps contain zeros (which will be correctly identified by the *correlate* functions.

As a caveat of point 1: if your data have very low variance, but real data, your data will be artificially gained by *pre_processing* to ensure stable correlations.

If you provide data with filled gaps (e.g. you used *st = st.merge(fill_value=0)* to either:

- The *detect* method of Tribe,
- The *detect* method of Template,
- *shortproc*,
- *dayproc*,

Then you will end up with the *wrong* result from the correlation or match_filter functions. You should provide data with gaps maintained, but merged (e.g. run *st = st.merge()* before passing the data to those functions).

If you have data that you know contains gaps that have been padded you must remove the pads and reinstate the gaps.

Memory limitations and what to do about it

You may (if you are running large numbers of templates, long data durations, or using a machine with small memory) run in to errors to do with memory consumption. The most obvious symptom of this is your computer freezing because it has allocated all of its RAM, or declaring that it cannot allocate memory. Because EQcorrscan computes correlations in parallel for multiple templates for the same data period, it will generate a large number of correlation vectors. At start-up, EQcorrscan will try to assign the memory it needs (although it then requires a little more later to do the summation across channels), so you might find that it fills your memory very early - this is just to increase efficiency and ensure that the memory is available when needed.

To get around memory limitations you can:

- Reduce the number of templates you run in parallel at once - for example you can make groups of a number of templates and run that group in parallel, before running the next group in parallel. This is not much less efficient, unless you have a machine with more CPU cores than your group-size.
- Reduce the length of data you are correlating at any one time. The default is to use day-long files, but there is nothing stopping you using shorter waveform durations.
- Reduce the number of channels in templates to only those that you need. Note, EQcorrscan will generate vectors of zeros for templates that are missing a channel that is present in other templates, again for processing efficiency, if not memory efficiency.

- Reduce your sampling rate. Obviously this needs to be at-least twice as large as your upper frequency filter, but much above this is wasted data.

The three threshold parameters

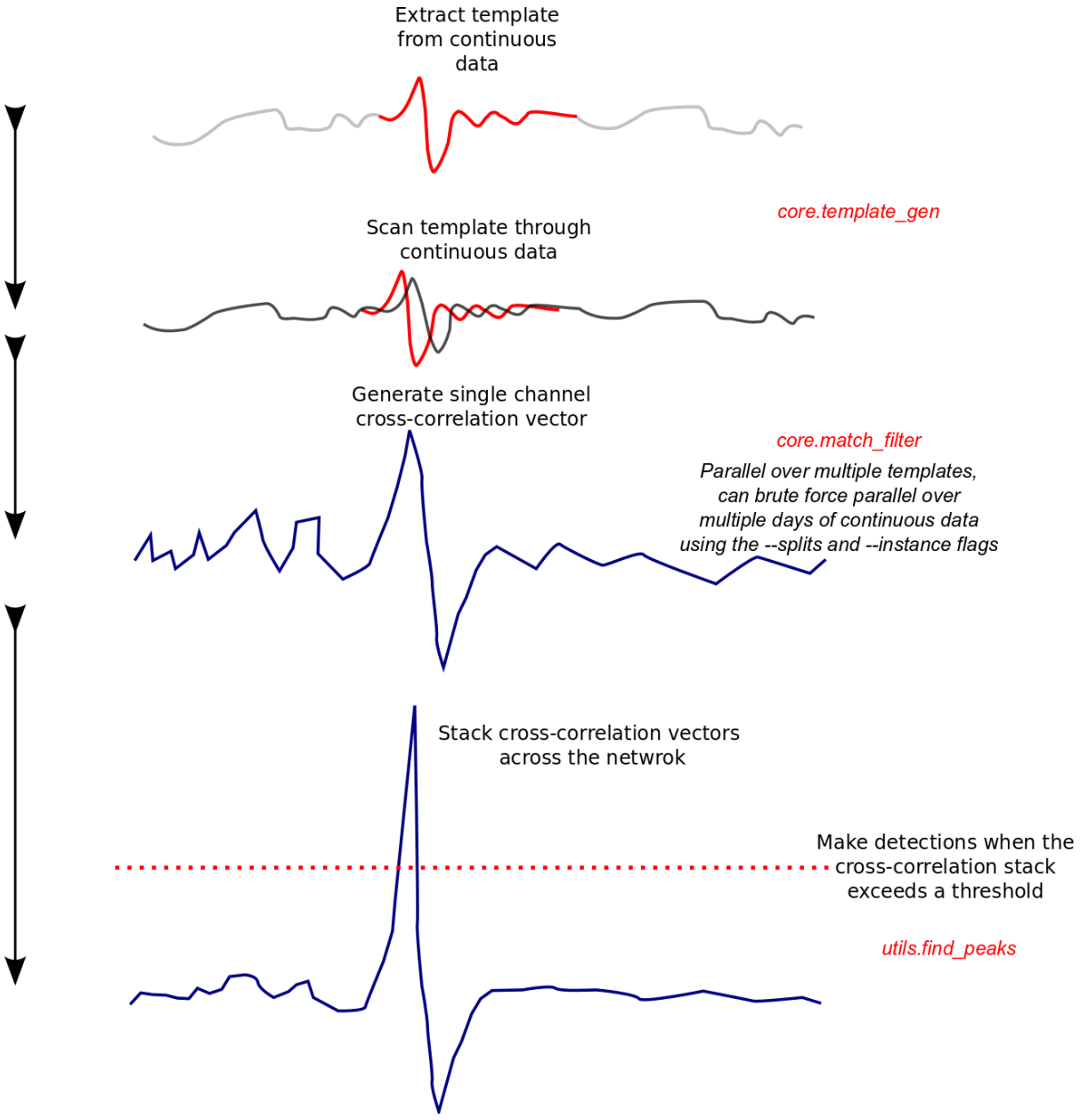
EQcorrscan detects both positively and negatively correlated waveforms. The match-filter routine has three key threshold parameters:

- **threshold_type** can either be MAD, abs or av_chan_corr. MAD stands for Median Absolute Deviation and is the most commonly used detection statistic in matched-filter studies. abs is the absolute cross-channel correlation sum, note that if you have different numbers of channels in your templates then this threshold metric probably isn't for you. av_chan_corr sets a threshold in the cross-channel correlation sum based on av_chan_corr x number of channels.
- **threshold** is the value used for the above metric.
- **trig_int** is the minimum interval in seconds for a detection using the same template. If there are multiple detections within this window for a single template then EQcorrscan will only give the best one (that exceeds the threshold the most).

Advanced example - match-filter-match-filter

In this section we will outline using the templates generated in the first tutorial to scan for similar earthquakes within a day of data. This small example does not truly exploit the parallel operations within this package however, so you would be encouraged to think about where parallel operations occur (*hint, the code can run one template per CPU*), and why there are –instance and–splits flags in the other scripts in the github repository (*hint, if you have heaps of memory and CPUs you can do some brute force day parallelisation!*).

The main processing flow is outlined in the figure below, note the main speedups in this process are achieved by running multiple templates at once, however this increases memory usage. If memory is a problem there are flags (mem_issue) in the match_filter.py source that can be turned on - the codes will then write temporary files, which is slower, but can allow for more data crunching at once, your trade-off, your call.




```
"""
Simple tutorial to demonstrate some of the basic capabilities of the EQcorrscan
matched-filter detection routine.  This builds on the template generation
tutorial and uses those templates.  If you haven't run that tutorial script
then you will need to before you can run this script.
"""

import glob
import logging

from http.client import IncompleteRead
from multiprocessing import cpu_count
from obspy.clients.fdsn import Client
from obspy import UTCDateTime, Stream, read

from eqcorrscan.utils import pre_processing
from eqcorrscan.utils import plotting
from eqcorrscan.core import match_filter
```

(continues on next page)

(continued from previous page)

```
# Set up logging
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s\t%(name)s\t%(levelname)s\t%(message)s")


def run_tutorial(plot=False, process_len=3600, num_cores=cpu_count(),
                 **kwargs):
    """Main function to run the tutorial dataset."""
    # First we want to load our templates
    template_names = glob.glob('tutorial_template_*.ms')

    if len(template_names) == 0:
        raise IOError('Template files not found, have you run the template ' +
                      'creation tutorial?')

    templates = [read(template_name) for template_name in template_names]

    # Work out what stations we have and get the data for them
    stations = []
    for template in templates:
        for tr in template:
            stations.append((tr.stats.station, tr.stats.channel))
    # Get a unique list of stations
    stations = list(set(stations))

    # We will loop through the data chunks at a time, these chunks can be any
    # size, in general we have used 1 day as our standard, but this can be
    # as short as five minutes (for MAD thresholds) or shorter for other
    # threshold metrics. However the chunk size should be the same as your
    # template process_len.

    # You should test different parameters!!!
    start_time = UTCDateTime(2016, 1, 4)
    end_time = UTCDateTime(2016, 1, 5)
    chunks = []
    chunk_start = start_time
    while chunk_start < end_time:
        chunk_end = chunk_start + process_len
        if chunk_end > end_time:
            chunk_end = end_time
        chunks.append((chunk_start, chunk_end))
        chunk_start += process_len

    unique_detections = []

    # Set up a client to access the GeoNet database
    client = Client("GEONET")

    # Note that these chunks do not rely on each other, and could be paralleled
    # on multiple nodes of a distributed cluster, see the SLURM tutorial for
    # an example of this.
    for t1, t2 in chunks:
        # Generate the bulk information to query the GeoNet database
        bulk_info = []
        for station in stations:
            bulk_info.append(('NZ', station[0], '*',
                              station[1][0] + 'H' + station[1][-1], t1, t2))

        # Note this will take a little while.
        print('Downloading seismic data, this may take a while')
```

(continues on next page)

(continued from previous page)

```
        st = Stream()
        for _bulk in bulk_info:
            try:
                st += client.get_waveforms(*_bulk)
            except IncompleteRead:
                print(f"Could not download {_bulk}")
        # Merge the stream, it will be downloaded in chunks
        st.merge()

        # Pre-process the data to set frequency band and sampling rate
        # Note that this is, and MUST BE the same as the parameters used for
        # the template creation.
        print('Processing the seismic data')
        st = pre_processing.shortproc(
            st, lowcut=2.0, highcut=9.0, filt_order=4, samp_rate=20.0,
            num_cores=num_cores, starttime=t1, endtime=t2)
        # Convert from list to stream
        st = Stream(st)

        # Now we can conduct the matched-filter detection
        detections = match_filter.match_filter(
            template_names=template_names, template_list=templates,
            st=st, threshold=8.0, threshold_type='MAD', trig_int=6.0,
            plotvar=plot, plotdir='.', cores=num_cores,
            plot_format='png', **kwargs)

        # Now lets try and work out how many unique events we have just to
        # compare with the GeoNet catalog of 20 events on this day in this
        # sequence
        for master in detections:
            keep = True
            for slave in detections:
                if not master == slave and abs(master.detect_time -
                                               slave.detect_time) <= 1.0:
                    # If the events are within 1s of each other then test which
                    # was the 'best' match, strongest detection
                    if not master.detect_val > slave.detect_val:
                        keep = False
                        print('Removed detection at %s with cccsum %s'
                              % (master.detect_time, master.detect_val))
                        print('Keeping detection at %s with cccsum %s'
                              % (slave.detect_time, slave.detect_val))
                        break
            if keep:
                unique_detections.append(master)
                print('Detection at :' + str(master.detect_time) +
                      ' for template ' + master.template_name +
                      ' with a cross-correlation sum of: ' +
                      str(master.detect_val))
                # We can plot these too
                if plot:
                    stplot = st.copy()
                    template = templates[template_names.index(
                        master.template_name)]
                    lags = sorted([tr.stats.starttime for tr in template])
                    maxlag = lags[-1] - lags[0]
                    stplot.trim(starttime=master.detect_time - 10,
                                endtime=master.detect_time + maxlag + 10)
                    plotting.detection_multiplot(
                        stplot, template, [master.detect_time.datetime])
    print('We made a total of ' + str(len(unique_detections)) + ' detections')
```

(continues on next page)

(continued from previous page)

```
    return unique_detections


if __name__ == '__main__':
    run_tutorial()
```

SLURM example

When the authors of EQcorrscan work on large projects, we use grid computers with the SLURM (Simple Linux Utility for Resource Management) job scheduler installed. To facilitate ease of setup, what follows is an example of how we run this.

```
#!/bin/bash
#SBATCH -J MatchTest
#SBATCH -A ##########
#SBATCH --time=12:00:00
#SBATCH --mem=7G
#SBATCH --nodes=1
#SBATCH --output=matchout_%a.txt
#SBATCH --error=matcherr_%a.txt
#SBATCH --cpus-per-task=16
#SBATCH --array=0-49

# Load the required modules here.
module load OpenCV/2.4.9-intel-2015a
module load ObsPy/0.10.3rc1-intel-2015a-Python-2.7.9
module load joblib/0.8.4-intel-2015a-Python-2.7.9

# Run your python script using srun
srun python2.7 LFEsearch.py --splits 50 --instance $SLURM_ARRAY_TASK_ID
```

Where we use a script (LFEsearch.py) that accepts splits and instance flags, this section of the script is as follows:

```
Split=False
instance=False
if len(sys.argv) == 2:
    flag=str(sys.argv[1])
    if flag == '--debug':
        Test=True
        Prep=False
    elif flag == '--debug-prep':
        Test=False
        Prep=True
    else:
        raise ValueError("I don't recognise the argument, I only know --debug and --debug-prep")
elif len(sys.argv) == 5:
    # Arguments to allow the code to be run in multiple instances
    Split=True
    Test=False
    Prep=False
    args=sys.argv[1:len(sys.argv)]
    for i in xrange(len(args)):
        if args[i] == '--instance':
            instance=int(args[i+1])
            print 'I will run this for instance '+str(instance)
        elif args[i] == '--splits':
            splits=int(args[i+1])
            print 'I will divide the days into '+str(splits)+' chunks'
```

(continues on next page)

(continued from previous page)

```
elif not len(sys.argv) == 1:
    raise ValueError("I only take one argument, no arguments, or two flags with␣
↪arguments")
else:
    Test=False
    Prep=False
    Split=False
```

The full script is not included in EQcorrscan, but is available on request.

2.5.3 Template creation (old API)

Note: These tutorials are for the functional workflow - for details on creating Template and Tribe objects see the *matched filter* docs page.

Simple example

Within *template_gen* there are lots of methods for generating templates depending on the type of pick data and waveform data you are using. Have a look at those to check their simple examples.

Example of how to generate a useful template from data available via FDSN (see for a list of possible clients):

```
>>> from obspy.clients.fdsn import Client
>>> from obspy.core.event import Catalog
>>> from eqcorrscan.core.template_gen import template_gen
>>> client = Client('NCEDC')
>>> catalog = client.get_events(eventid='72572665', includearrivals=True)
>>> templates = template_gen(method="from_client", catalog=catalog,
...                          client_id='NCEDC', lowcut=2.0, highcut=9.0,
...                          samp_rate=20.0, filt_order=4, length=3.0,
...                          prepick=0.15, swin='all', process_len=200)
```

This will download data for a single event (given by eventid) from the NCEDC database, then use that information to download relevant waveform data. These data will then be filtered and cut according the parameters set.

It is often wise to set these parameters once as variables at the start of your scripts to ensure that the same parameters are used for both template creation and matched-filtering.

The method *from_client* can cope with multiple events (hence the use of a Catalog object), so you can download a lot of events from your chosen client and generate templates for them all.

Useful considerations

With these data-mining techniques, memory consumption is often an issue, as well as speed. To reduce memory consumption and increase efficiency it is often worth using a subset of picks. The advanced example below shows one way to do this. In practice, five picks (and therefore traces in a template) is often sufficient for matched-filter detections. However, you should test this on your own data.

Some other things that you might want to consider when generating templates include:

- Template-length, you probably only want to include the real earthquake signal in your template, so really long templates are probably not the best idea.
- On the same note, don't include much (if any) data before the P-phase, unless you have good reason to - assuming your noise is random, including noise will reduce the correlations.
- Consider your frequency band - look for peak power in the chosen waveform **relative to the noise**.

- Coda waves often describe scatterers - scattered waves are very interesting, but may reduce the generality of your templates. If this is what you want, include coda, if you want a more general template, I would suggest not including coda. For examples of this you could try generating a lot of templates from a sequence and computing the SVD of the templates to see where the most coherent energy is (in the first basis vector), or just computing the stack of the waveforms.

Storing templates

Templates are returned as obspy Stream objects. You will likely want to store these templates on disk. This is usually best done by saving them as miniseed files. Miniseed uses an efficient compression algorithm and allows multiplexing, which is useful for template storage. However we do not constrain you to this.

```
>>> templates[0].write('template.ms', format="MSEED")
```

Advanced example

In this example we will download some data and picks from the New Zealand GeoNet database and make use of the functions in EQcorrscan to quickly and simply generate templates for use in matched-filter detection. In the example we are looking at an earthquake sequence on the east coast of New Zealand's North Island that occurred on the 4th of January 2016. We will take a set of four template events from the sequence that have been picked by GeoNet, of a range of magnitudes. You can decide if these were *good* templates or not. You could easily extend this by choosing more template events (the mainshock in the sequence is a M 5 and you can get the information by clicking).

You do not need to use data from an online server, many pick formats can be parsed, either by obspy, or (for seisan pick files) through the Sfile_utils module in EQcorrscan.

This template script is written to be general, so you should be able to give command line arguments to the script to generate templates from other FDSN databases. Note that some data-centers do not support full FDSN quakeml files, and working out which do is quite painful.

```
"""
Simple tutorial detailing how to generate a series of templates from catalog\
data available online.
"""

import logging

from obspy.clients.fdsn import Client
from obspy.core.event import Catalog

from eqcorrscan.utils.catalog_utils import filter_picks
from eqcorrscan.core import template_gen

# Set up logging
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s\t%(name)s\t%(levelname)s\t%(message)s")


def mktemplates(
        network_code='GEONET', plot=True, public_ids=None):
    """Functional wrapper to make templates"""
    public_ids = public_ids or [
        '2016p008122', '2016p008353', '2016p008155', '2016p008194']
    client = Client(network_code)
    # We want to download a few events from an earthquake sequence, these are
    # identified by publiID numbers, given as arguments
```

(continues on next page)

(continued from previous page)

```
    catalog = Catalog()
    for public_id in public_ids:
        try:
            catalog += client.get_events(
                eventid=public_id, includearrivals=True)
        except TypeError:
            # Cope with some FDSN services not implementing includearrivals
            catalog += client.get_events(eventid=public_id)

    # Lets plot the catalog to see what we have
    if plot:
        catalog.plot(projection='local', resolution='h')

    # We don't need all the picks, lets take the information from the
    # five most used stations - note that this is done to reduce computational
    # costs.
    catalog = filter_picks(catalog, top_n_picks=5)
    # We only want the P picks in this example, but you can use others or all
    #  picks if you want.
    for event in catalog:
        for pick in event.picks:
            if pick.phase_hint == 'S':
                event.picks.remove(pick)

    # Now we can generate the templates
    templates = template_gen.template_gen(
        method='from_client', catalog=catalog, client_id=network_code,
        lowcut=2.0, highcut=9.0, samp_rate=20.0, filt_order=4, length=3.0,
        prepick=0.15, swin='all', process_len=3600, plot=plot)

    # We now have a series of templates! Using Obspy's Stream.write() method we
    # can save these to disk for later use.  We will do that now for use in the
    # following tutorials.
    for i, template in enumerate(templates):
        template.write('tutorial_template_' + str(i) + '.ms', format='MSEED')
        # Note that this will warn you about data types.  As we don't care
        # at the moment, whatever obspy chooses is fine.
    return


if __name__ == '__main__':
    """Wrapper for template creation"""
    import sys
    import warnings
    if not len(sys.argv) > 1:
        warnings.warn('Needs a network ID followed by a list of event IDs, ' +
                      'will run the test case instead')
        mktemplates()
    else:
        net_code = sys.argv[1]
        idlist = list(sys.argv)[2:]
        print(idlist)
        mktemplates(network_code=net_code, public_ids=idlist)
```

Try this example for another, Northern California Data Center earthquake:

```
python template_creation.py NCEDC 72572665
```

This will (unfortunately for you) generate a warning about un-labelled picks, as many data-centers do not provide phase-hints with their picks. We care about which phase is which when we have to run our cross-correlation re-picker (yet to be completed), which means that we, by convention, assign P-picks to the vertical channel and S-picks to both horizontal channels.

This will also show template waveforms for both the automatic and the manual picks - you can change this by removing either automatic or manual picks y setting the *evaluation_mode* flag in *eqcorrscan.utils.catalog_utils.filter_picks()*.

Note: To run this script and for all map plotting you will need to install matplotlib-toolkits basemap package. Install instructions and a link to the download are . We recommend that you install the package using your system package manager if it is available.

Important considerations

In this tutorial we enforce downloading of day-long data for the template generation. This is to ensure that the data we make the template from, and the data we use for detection are processed in exactly the same way. If we were to only download a short segment of data around the event and process this we would find that the resampling process would result in minor differences between the templates and the continuous data. This has the effect that, for self-detections, the cross-correlation values are less than 1.

This is an important effect and something that you should consider when generating your own templates. You **MUST** process your templates in the exact same way (using the same routines, same filters, same resampling, and same data length) as your continuous data. It can have a very significant impact to your results.

The functions provided in *template_gen* are there to aid you, but if you look at the source code, all they are doing is:

- Detrending;
- Resampling;
- Filtering;
- and cutting.

If you want to do these things another way you are more then welcome to!

Converting from templates to Template and Tribe objects

This section should guide you through generating `eqcorrscan.core.match_filter.Template` and `eqcorrscan.core.match_filter.Tribe` objects from the simple templates created above. This should provide you with a means to migrate from older scripts to the more modern, object-oriented workflows in *matched filter*.

To generate a `eqcorrscan.core.match_filter.Template` from a cut, processed waveform generated by the *template_gen* functions you must fill all the attributes of the `eqcorrscan.core.match_filter.Template` object. You must use the parameters you used when generating the template waveform. Note that you do not need to include an *event* to make a complete `eqcorrscan.core.match_filter.Template` object, but it will be benficial for further location of detected events.

In the following example we take a cut-waveform and convert it to a `eqcorrscan.core.match_filter.Template` object using the parameters we used to generate the waveform:

```
>>> from eqcorrscan.core.match_filter import Template
>>> template = Template(
...     name='test_template', st=templates[0], lowcut=2.0, highcut=9.0,
...     samp_rate=20.0, filt_order=4, prepick=0.15, process_length=200)
```

You can then make a `eqcorrscan.core.match_filter.Tribe` from a list of `eqcorrscan.core.match_filter.Template` objects as follows:

```
>>> from eqcorrscan.core.match_filter import Tribe
>>> tribe = Tribe(templates=[template])
```

2.5.4 Subspace Detection

EQcorrscan's subspace detection methods are closely modelled on the method described by , Subspace Detectors: Theory. We offer options to multiplex data or leave as single-channels (multiplexing is significantly faster).

Subspace detection is implemented in an object-oriented style, whereby individual detectors are constructed from data, then used to detect within continuous data. At the core of the subspace routine is a Cython-based, static-typed routine to calculate the detection statistics. We do this to make use of numpy's vectorized calculations, while taking advantage of the speed-ups afforded by compiling the sliding window loop.

WARNING

Subspace in EQcorrscan is in **beta**, you must check your results match what you expect - if you find errors please report them. Although our test-cases run correctly, changes in data quality may affect the routines in ways we have not accounted for.

Important

How you generate you detector is likely to be the most important thing, careful selection and alignment is key, and because of this we haven't provided a total *cookie-cutter* system for doing this. You have freedom to chose your parameters, how to process, how you align, what traces to keep, whether you multiplex or not, etc. This also means you have a lot of freedom to **get it wrong**. You will have to do significant testing with your own dataset to work out what works and what doesn't. Anything that you find that doesn't work well in EQcorrscans system, it would be great to hear about so that we can make it better.

The following examples demonstrate some of the options, but not all of them. The advanced example is the example used to test and develop subspace and took a fair amount of effort over a number of weeks.

Simple example

To begin with you will need to create a **Detector**:

```
>>> from eqcorrscan.core import subspace
>>> detector = subspace.Detector()
```

This will create an empty *detector* object. These objects have various attributes, including the data to be used as a detector (*detector.data*), alongside the full input and output basis vector matrices (*detector.u* and *detector.v* respectively) and the vector of singular-values (*detector.sigma*). Meta-data are also included, including whether the detector is multiplexed or not (*detector.multiplex*), the filters applied (*detector.lowcut*, *detector.highcut*, *detection.filt_order*, *detector.sampling_rate*), the dimension of the subspace (*detector.dimension*), and the name of the detector, which you can use for book-keeping (*detector.name*).

To populate the empty detector you need a design set of streams that have been aligned (see clustering submodule for alignment methods).

```
>>> from obspy import read
>>> import glob
>>> import os
>>> from eqcorrscan import tests
>>> # Get the path for the test-data so we can test this
>>> TEST_PATH = os.path.dirname(tests.__file__)
>>> wavefiles = glob.glob(
...     TEST_PATH + '/test_data/similar_events_processed/*')
>>> wavefiles.sort()  # Sort the wavefiles to ensure reproducibility
>>> streams = [read(w) for w in wavefiles[0:3]]
>>> # Channels must all be the same length
>>> detector.construct(streams=streams, lowcut=2, highcut=9, filt_order=4,
...                    sampling_rate=20, multiplex=True, name='Test_1',
```

(continues on next page)

(continued from previous page)

```
...                     align=True, shift_len=0.5, reject=0.2)
Detector: Test_1
```

This will populate all the attributes of your *detector* object, and fill the *detector.data* with the full input basis vector matrix.

You will want to reduce the dimensions of your subspace detector, such that you are just describing the signal, preferably with a lot of generality. Details for selecting dimensionality should be found in . To do this in EQcorrscan simply use the *partition* method:

```
>>> detector.partition(2)
Detector: Test_1
```

This will populate *detector.data* with the first four, left-most input basis vectors. You can test to see how much of your original design set is described by this detector by using the *energy_capture* method:

```
>>> percent_capture = detector.energy_capture()
```

This will return a percentage capture, you can run this for multiple dimensions to test what dimension best suits your application. Again, details for this selection can be found in .

Finally, to use your detector to detect within continuous data you should use the *detect* method. This requires a stream with the same stations and channels used in the detector, and a threshold from 0-1, where 0 is no signal, and 1 is totally described by your detector. You can extract streams for the detections at the same time as the detections by setting the extract_detections flag to True.

```
>>> stream = read(wavefiles[0])
>>> detections = detector.detect(st=stream, threshold=0.5, trig_int=3)
```

Advanced Example

This example computes detections for a short data-period during an earthquake sequence in the Wairarapa region of New Zealand's North Island. This example only shows one subspace detector, but could be extended, using the various *clustering* routines in EQcorrscan, to create many subspace detectors. These could be run using the subspace_detect function, which runs similar detectors in parallel through the given data.

```
"""
Advanced subspace tutorial to show some of the capabilities of the method.

This example uses waveforms from a known earthquake sequence (in the Wairarapa
region north of Wellington, New Zealand). The catalogue locations etc can
be downloaded from this link:

http://quakesearch.geonet.org.nz/services/1.0.0/csv?bbox=175.37956,-40.97912,175.
↪53097,-40.84628&startdate=2015-7-18T2:00:00&enddate=2016-7-18T3:00:00

"""

import logging

from http.client import IncompleteRead
from obspy.clients.fdsn import Client
from obspy import UTCDateTime, Stream

from eqcorrscan.utils.catalog_utils import filter_picks
from eqcorrscan.utils.clustering import catalog_cluster
from eqcorrscan.core import subspace

# Set up logging
```

(continues on next page)

(continued from previous page)

```
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s\t%(name)s\t%(levelname)s\t%(message)s")


def run_tutorial(plot=False, multiplex=True, return_streams=False, cores=4,
                 verbose=False):
    """
    Run the tutorial.

    :return: detections
    """
    client = Client("GEONET", debug=verbose)
    cat = client.get_events(
        minlatitude=-40.98, maxlatitude=-40.85, minlongitude=175.4,
        maxlongitude=175.5, starttime=UTCDateTime(2016, 5, 1),
        endtime=UTCDateTime(2016, 5, 20))
    print(f"Downloaded a catalog of {len(cat)} events")
    # This gives us a catalog of events - it takes a while to download all
    # the information, so give it a bit!
    # We will generate a five station, multi-channel detector.
    cat = filter_picks(catalog=cat, top_n_picks=5)
    stachans = list(set(
        [(pick.waveform_id.station_code, pick.waveform_id.channel_code)
         for event in cat for pick in event.picks]))
    # In this tutorial we will only work on one cluster, defined spatially.
    # You can work on multiple clusters, or try to whole set.
    clusters = catalog_cluster(
        catalog=cat, metric="distance", thresh=2, show=False)
    # We will work on the largest cluster
    cluster = sorted(clusters, key=lambda c: len(c))[-1]
    # This cluster contains 32 events, we will now download and trim the
    # waveforms.  Note that each chanel must start at the same time and be the
    # same length for multiplexing.  If not multiplexing EQcorrscan will
    # maintain the individual differences in time between channels and delay
    # the detection statistics by that amount before stacking and detection.
    client = Client('GEONET')
    design_set = []
    st = Stream()
    for event in cluster:
        print(f"Downloading for event {event.resource_id.id}")
        bulk_info = []
        t1 = event.origins[0].time
        t2 = t1 + 25.1  # Have to download extra data, otherwise GeoNet will
        # trim wherever suits.
        t1 -= 0.1
        for station, channel in stachans:
            try:
                st += client.get_waveforms(
                    'NZ', station, '*', channel[0:2] + '?', t1, t2)
            except IncompleteRead:
                print(f"Could not download for {station} {channel}")
    print(f"Downloaded {len(st)} channels")
    for event in cluster:
        t1 = event.origins[0].time
        t2 = t1 + 25
        design_set.append(st.copy().trim(t1, t2))
    # Construction of the detector will process the traces, then align them,
    # before multiplexing.
    print("Making detector")
    detector = subspace.Detector()
```

(continues on next page)

(continued from previous page)

```
    detector.construct(
        streams=design_set, lowcut=2.0, highcut=9.0, filt_order=4,
        sampling_rate=20, multiplex=multiplex, name='Wairarapa1', align=True,
        reject=0.2, shift_len=6, plot=plot).partition(9)
    print("Constructed Detector")
    if plot:
        detector.plot()
    # We also want the continuous stream to detect in.
    t1 = UTCDateTime(2016, 5, 11, 19)
    t2 = UTCDateTime(2016, 5, 11, 20)
    # We are going to look in a single hour just to minimize cost, but you can
    # run for much longer.
    bulk_info = [('NZ', stachan[0], '*',
                  stachan[1][0] + '?' + stachan[1][-1],
                  t1, t2) for stachan in detector.stachans]
    print("Downloading continuous data")
    st = client.get_waveforms_bulk(bulk_info)
    st.merge().detrend('simple').trim(starttime=t1, endtime=t2)
    # We set a very low threshold because the detector is not that great, we
    # haven't aligned it particularly well - however, at this threshold we make
    # two real detections.
    print("Computing detections")
    detections, det_streams = detector.detect(
        st=st, threshold=0.4, trig_int=2, extract_detections=True,
        cores=cores)
    if return_streams:
        return detections, det_streams
    else:
        return detections


if __name__ == '__main__':
    run_tutorial()
```

2.5.5 Lag-time and pick correction

The following is a work-in-progress tutorial for lag-calc functionality.

An important note

Picks generated by lag-calc are relative to the start of the template waveform, for example, if you generated your templates with a pre_pick of 0.2, you should expect picks to occur 0.2 seconds before the actual phase arrival. The result of this is that origin-times will be shifted by the same amount.

If you have applied different pre_picks to different channels when generating template (currently not supported by any EQcorrscan functions), then picks generated here will not give the correct location.

Advanced Example: Parkfield 2004

```
"""Tutorial to illustrate the lag_calc usage."""
import logging
from multiprocessing import cpu_count

from obspy.clients.fdsn import Client
from obspy.clients.fdsn.header import FDSNException
from obspy.core.event import Catalog
from obspy import UTCDateTime, Stream
```

(continues on next page)

(continued from previous page)

```
from eqcorrscan.core import template_gen, match_filter, lag_calc
from eqcorrscan.utils import pre_processing, catalog_utils

# Set up logging
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s\t%(name)s\t%(levelname)s\t%(message)s")


def run_tutorial(min_magnitude=2, shift_len=0.2, num_cores=4, min_cc=0.5):
    """Functional, tested example script for running the lag-calc tutorial."""
    if num_cores > cpu_count():
        num_cores = cpu_count()
    client = Client('NCEDC')
    t1 = UTCDateTime(2004, 9, 28)
    t2 = t1 + 86400
    print('Downloading catalog')
    catalog = client.get_events(
        starttime=t1, endtime=t2, minmagnitude=min_magnitude,
        minlatitude=35.7, maxlatitude=36.1, minlongitude=-120.6,
        maxlongitude=-120.2, includearrivals=True)
    # We don't need all the picks, lets take the information from the
    # five most used stations - note that this is done to reduce computational
    # costs.
    catalog = catalog_utils.filter_picks(
        catalog, channels=['EHZ'], top_n_picks=5)
    # There is a duplicate pick in event 3 in the catalog - this has the effect
    # of reducing our detections - check it yourself.
    for pick in catalog[3].picks:
        if pick.waveform_id.station_code == 'PHOB' and \
                        pick.onset == 'emergent':
            catalog[3].picks.remove(pick)
    print('Generating templates')
    templates = template_gen.template_gen(
        method="from_client", catalog=catalog, client_id='NCEDC',
        lowcut=2.0, highcut=9.0, samp_rate=50.0, filt_order=4, length=3.0,
        prepick=0.15, swin='all', process_len=3600)
    # In this section we generate a series of chunks of data.
    start_time = UTCDateTime(2004, 9, 28, 17)
    end_time = UTCDateTime(2004, 9, 28, 20)
    process_len = 3600
    chunks = []
    chunk_start = start_time
    while chunk_start < end_time:
        chunk_end = chunk_start + process_len
        if chunk_end > end_time:
            chunk_end = end_time
        chunks.append((chunk_start, chunk_end))
        chunk_start += process_len

    all_detections = []
    picked_catalog = Catalog()
    template_names = [template[0].stats.starttime.strftime("%Y%m%d_%H%M%S")
                      for template in templates]
    for t1, t2 in chunks:
        print('Downloading and processing for start-time: %s' % t1)
        # Download and process the data
        bulk_info = [(tr.stats.network, tr.stats.station, '*',
                      tr.stats.channel, t1, t2) for tr in templates[0]]
        # Just downloading a chunk of data
```

(continues on next page)

(continued from previous page)

```
        try:
            st = client.get_waveforms_bulk(bulk_info)
        except FDSNException:
            st = Stream()
            for _bulk in bulk_info:
                st += client.get_waveforms(*_bulk)
        st.merge(fill_value='interpolate')
        st = pre_processing.shortproc(
            st, lowcut=2.0, highcut=9.0, filt_order=4, samp_rate=50.0,
            num_cores=num_cores)
        detections = match_filter.match_filter(
            template_names=template_names, template_list=templates, st=st,
            threshold=8.0, threshold_type='MAD', trig_int=6.0, plotvar=False,
            plotdir='.', cores=num_cores)
        # Extract unique detections from set.
        unique_detections = []
        for master in detections:
            keep = True
            for slave in detections:
                if not master == slave and\
                   abs(master.detect_time - slave.detect_time) <= 1.0:
                    # If the events are within 1s of each other then test which
                    # was the 'best' match, strongest detection
                    if not master.detect_val > slave.detect_val:
                        keep = False
                        break
            if keep:
                unique_detections.append(master)
        all_detections += unique_detections

        picked_catalog += lag_calc.lag_calc(
            detections=unique_detections, detect_data=st,
            template_names=template_names, templates=templates,
            shift_len=shift_len, min_cc=min_cc, interpolate=False, plot=False)
    # Return all of this so that we can use this function for testing.
    return all_detections, picked_catalog, templates, template_names


if __name__ == '__main__':
    run_tutorial(min_magnitude=4, num_cores=cpu_count())
```

2.5.6 Magnitude calculation

EQcorrscan contains both an automatic amplitude picker and a singular-value decomposition derived magnitude calculation, which is very accurate but requires high levels of event similarity.

Relative moment by singular-value decomposition

This method closely follows the method outlined by .

This example requires data downloaded from the eqcorrscan github repository.

```
>>> from eqcorrscan.utils.mag_calc import svd_moments
>>> from obspy import read
>>> import glob
>>> from eqcorrscan.utils.clustering import svd
>>> import numpy as np
>>> from eqcorrscan import tests
>>> import os
```

(continues on next page)

(continued from previous page)

```
>>> # Get the path for the test-data so we can test this
>>> testing_path = os.path.dirname(
...     tests.__file__) + '/test_data/similar_events_processed'
>>> stream_files = glob.glob(os.path.join(testing_path, '*'))
>>> stream_list = [read(stream_file) for stream_file in stream_files]
>>> event_list = []
>>> for i, stream in enumerate(stream_list):
...     st_list = []
...     for tr in stream:
...         # Only use the vertical channels of sites with known high similarity.
...         # You do not need to use this step for your data.
...         if (tr.stats.station, tr.stats.channel) not in\
...                 [('WHAT2', 'SH1'), ('WV04', 'SHZ'), ('GCSZ', 'EHZ')]:
...             stream.remove(tr)
...             continue
...         st_list.append(i)
...     event_list.append(st_list)
<obspy.core.stream.Stream object at ...>
>>> event_list = np.asarray(event_list).T.tolist()
>>> SVectors, SValues, Uvectors, stachans = svd(stream_list=stream_list)
>>> M, events_out = svd_moments(u=Uvectors, s=SValues, v=SVectors,
...                             stachans=stachans, event_list=event_list)
```

2.5.7 Clustering and stacking

Prior to template generation, it may be beneficial to cluster earthquake waveforms. Clusters of earthquakes with similar properties can then be stacked to create higher signal-to-noise templates that describe the dataset well (and require fewer templates to reduce computational cost). Clusters could also be reduced to their singular-vectors and employed in a subspace detection routine (coming soon to eqcorrscan).

The following outlines a few examples of clustering and stacking.

Cluster in space

Download a catalog of global earthquakes and cluster in space, set the distance threshold to 1,000km

```
>>> from eqcorrscan.utils.clustering import catalog_cluster
>>> from obspy.clients.fdsn import Client
>>> from obspy import UTCDateTime

>>> client = Client("IRIS")
>>> starttime = UTCDateTime("2002-01-01")
>>> endtime = UTCDateTime("2002-02-01")
>>> cat = client.get_events(starttime=starttime, endtime=endtime,
...                         minmagnitude=6, catalog="ISC")
>>> groups = catalog_cluster(
...     catalog=cat, metric="distance", thresh=1000, show=False)
```

Download a local catalog of earthquakes and cluster much finer (distance threshold of 2km).

```
>>> client = Client("NCEDC")
>>> cat = client.get_events(starttime=starttime, endtime=endtime,
...                         minmagnitude=2)
>>> groups = catalog_cluster(
...     catalog=cat, metric="distance", thresh=2, show=False)
```

Setting show to true will plot the dendrogram for grouping with individual groups plotted in different colours. The clustering is performed using scipy's . Specifically clustering is performed using the linkage method, which is an agglomorative clustering routine. EQcorrscan uses the average method with the euclidean distance metric.

Cluster in time and space

EQcorrscan's space-time clustering routine first computes groups in space, using the space_cluster method, then splits the returned groups based on their inter-event time.

The following example extends the example of the global catalog with a 1,000km distance threshold and a one-day temporal limit.

```
>>> from eqcorrscan.utils.clustering import space_time_cluster
>>> client = Client("IRIS")
>>> cat = client.get_events(starttime=starttime, endtime=endtime,
...                         minmagnitude=6, catalog="ISC")
>>> groups = space_time_cluster(catalog=cat, t_thresh=86400, d_thresh=1000)
```

Cluster according to cross-correlation values

Waveforms from events are often best grouped based on their similarity. EQcorrscan has a method to compute clustering based on average cross-correlations. This again uses scipy's , however in this case clusters are computed using the *single* method. Distances are computed from the average of the multi-chanel cross-correlation values.

The following example uses data stored in the EQcorrscan github repository, in the tests directory.

```
>>> from obspy import read
>>> import glob
>>> import os
>>> from eqcorrscan.utils.clustering import cluster
>>> from eqcorrscan import tests
>>> # You will need to edit this line to the location of your eqcorrscan repo.
>>> TEST_PATH = os.path.dirname(tests.__file__)
>>> testing_path = TEST_PATH + '/test_data/similar_events_processed'
>>> stream_files = glob.glob(os.path.join(testing_path, '*'))
>>> stream_list = [(read(stream_file), i)
...                for i, stream_file in enumerate(stream_files)]
>>> for stream in stream_list:
...     for tr in stream[0]:
...         if tr.stats.station not in ['WHAT2', 'WV04', 'GCSZ']:
...             stream[0].remove(tr)
...             continue
>>> groups = cluster(template_list=stream_list, show=False,
...                  corr_thresh=0.3, cores=2)
```

Stack waveforms (linear)

Following from clustering, similar waveforms can be stacked. EQcorrscan includes two stacking algorithms, a simple linear stacking method, and a phase-weighted stacking method.

The following examples use the test data in the eqcorrscan github repository.

```
>>> from eqcorrscan.utils.stacking import linstack

>>> # groups[0] should contain 3 streams, which we can now stack
>>> # Groups are returned as lists of tuples, of the stream and event index
>>> group_streams = [st_tuple[0] for st_tuple in groups[0]]
>>> stack = linstack(streams=group_streams)
```

Stack waveforms (phase-weighted)

The phase-weighted stack method closely follows the method outlined by . In this method the linear stack is weighted by the stack of the instantaneous phase. In this manor coherent signals are amplified.

```
>>> from eqcorrscan.utils.stacking import PWS_stack

>>> # groups[0] should contain 3 streams, which we can now stack
>>> # Groups are returned as lists of tuples, of the stream and event index
>>> stack = PWS_stack(streams=group_streams)
```

2.6 EQcorrscan API

EQcorrscan contains two main modules, core and utils. Core contains most of the most useful functions (including matched-filtering), and utils contains a range of possibly helpful functions that may be useful to you and/or are used by the core routines.

2.6.1 Core

Core routines of the EQcorrscan project.

lag_calc

Functions to generate pick-corrections for events detected by correlation.

copyright EQcorrscan developers.

license GNU Lesser General Public License, Version 3 (https://www.gnu.org/copyleft/lesser.html)

Functions for generating pick-corrections from cross-correlations with a template. Originally this was designed for events detected by matched-filtering, however you can use any well correlated events. Based on the method of Shelly and Hardebeck (2010).

Functions

eqcorrscan.core.lag_calc.**lag_calc**(*detections*, *detect_data*, *template_names*, *templates*, *shift_len=0.2*, *min_cc=0.4*, *min_cc_from_mean_cc_factor=None*, *horizontal_chans=['E', 'N', '1', '2']*, *vertical_chans=['Z']*, *cores=1*, *interpolate=False*, *plot=False*, *plotdir=None*, *export_cc=False*, *cc_dir=None*)

Cross-correlation derived picking of seismic events.

Overseer function to take a list of detection objects, cut the data for them to lengths of the same length of the template + shift_len on either side. This will output a `obspy.core.event.Catalog` of picked events. Pick times are based on the lag-times found at the maximum correlation, providing that correlation is above the min_cc.

Parameters

- **detections** (*list*) – List of `eqcorrscan.core.match_filter.Detection` objects.
- **detect_data** (*obspy.core.stream.Stream*) – All the data needed to cut from - can be a gappy Stream.
- **template_names** (*list*) – List of the template names, used to help identify families of events. Must be in the same order as templates.
- **templates** (*list*) –

  List of the templates, templates must be a list of `obspy.core.stream.Stream` objects.

- **shift_len** (*float*) – Shift length allowed for the pick in seconds, will be plus/minus this amount - default=0.2
- **min_cc** (*float*) – Minimum cross-correlation value to be considered a pick, default=0.4.
- **min_cc_from_mean_cc_factor** (*float*) – If set to a value other than None, then the minimum cross-correlation value for a trace is set individually for each detection based on: min(detect_val / n_chans * min_cc_from_mean_cc_factor, min_cc).
- **horizontal_chans** (*list*) – List of channel endings for horizontal-channels, on which S-picks will be made.
- **vertical_chans** (*list*) – List of channel endings for vertical-channels, on which P-picks will be made.
- **cores** (*int*) – Number of cores to use in parallel processing, defaults to one.
- **interpolate** (*bool*) – Interpolate the correlation function to achieve sub-sample precision.
- **plot** (*bool*) – To generate a plot for every detection or not, defaults to False
- **plotdir** – Path to plotting folder, plots will be output here.
- **export_cc** (*bool*) – To generate a binary file in NumPy for every detection or not, defaults to False
- **cc_dir** (*str*) – Path to saving folder, NumPy files will be output here.

Returns Catalog of events with picks. No origin information is included. These events can then be written out via `obspy.core.event.Catalog.write()`, or to Nordic Sfiles using `eqcorrscan.utils.sfile_util.eventtosfile()` and located externally.

Return type obspy.core.event.Catalog

Note: Picks output in catalog are generated relative to the template start-time. For example, if you generated your template with a pre_pick time of 0.2 seconds, you should expect picks generated by lag_calc to occur 0.2 seconds before the true phase-pick. This is because we do not currently store template meta-data alongside the templates.

Warning: Because of the above note, origin times will be consistently shifted by the static pre_pick applied to the templates.

Warning: This routine requires only one template per channel (e.g. you should not use templates with a P and S template on a single channel). If this does occur an error will be raised.

Note: S-picks will be made on horizontal channels, and P picks made on vertical channels - the default is that horizontal channels end in one of: 'E', 'N', '1' or '2', and that vertical channels end in 'Z'. The options vertical_chans and horizontal_chans can be changed to suit your dataset.

Note: Individual channel cross-correlations are stored as a `obspy.core.event.Comment` for each pick, and the summed cross-correlation value resulting from these is stored as a `obspy.core.event.Comment` in the main `obspy.core.event.Event` object.

Note: The order of events is preserved (e.g. detections[n] == output[n]), providing picks have been made for that event. If no picks have been made for an event, it will not be included in the output. However, as each detection has an ID associated with it, these can be mapped to the output resource_id for each Event in the output Catalog. e.g.

detections[n].id == output[m].resource_id

if the output[m] is for the same event as detections[n].

Note: The correlation data that are saved to the binary files can be useful to select an appropriate threshold for your data.

eqcorrscan.core.lag_calc.**xcorr_pick_family**(*family, stream, shift_len=0.2, min_cc=0.4, min_cc_from_mean_cc_factor=None, horizontal_chans=['E', 'N', '1', '2'], vertical_chans=['Z'], cores=1, interpolate=False, plot=False, plotdir=None, export_cc=False, cc_dir=None*)

Compute cross-correlation picks for detections in a family.

Parameters

- **family** (*eqcorrscan.core.match_filter.family.Family*) – Family to calculate correlation picks for.
- **stream** (*obspy.core.stream.Stream*) – Data stream containing data for all (or a subset of) detections in the Family
- **shift_len** (*float*) – Shift length allowed for the pick in seconds, will be plus/minus this amount - default=0.2
- **min_cc** (*float*) – Minimum cross-correlation value to be considered a pick, default=0.4.
- **min_cc_from_mean_cc_factor** (*float*) – If set to a value other than None, then the minimum cross-correlation value for a trace is set individually for each detection based on: min(detect_val / n_chans * min_cc_from_mean_cc_factor, min_cc).
- **horizontal_chans** (*list*) – List of channel endings for horizontal-channels, on which S-picks will be made.
- **vertical_chans** (*list*) – List of channel endings for vertical-channels, on which P-picks will be made.
- **cores** (*int*) – Number of cores to use in parallel processing, defaults to one.
- **interpolate** (*bool*) – Interpolate the correlation function to achieve sub-sample precision.
- **plot** (*bool*) – To generate a plot for every detection or not, defaults to False
- **plotdir** (*str*) – Path to plotting folder, plots will be output here.
- **export_cc** (*bool*) – To generate a binary file in NumPy for every detection or not, defaults to False
- **cc_dir** (*str*) – Path to saving folder, NumPy files will be output here.

Returns Dictionary of picked events keyed by detection id.

Private Functions

Note that these functions are not designed for public use and may change at any point.

eqcorrscan.core.lag_calc.**_concatenate_and_correlate**(*streams*, *template*, *cores*)
Concatenate a list of streams into one stream and correlate that with a template.

All traces in a stream must have the same length.

eqcorrscan.core.lag_calc.**_prepare_data**(*family*, *detect_data*, *shift_len*)
Prepare data for lag_calc - reduce memory here.

Parameters

- **family** (*eqcorrscan.core.match_filter.family.Family*) – The Family containing the template and detections.
- **detect_data** (*obspy.core.stream.Stream*) – Stream to extract detection streams from.
- **shift_len** (*float*) – Shift length in seconds allowed for picking.

Returns Dictionary of detect_streams keyed by detection id to be worked on

Return type dict

eqcorrscan.core.lag_calc.**_xcorr_interp**(*ccc*, *dt*)
Interpolate around the maximum correlation value for sub-sample precision.

Parameters

- **ccc** (*numpy.ndarray*) – Cross-correlation array
- **dt** (*float*) – sample interval

Returns Position of interpolated maximum in seconds from start of ccc

Return type float

match_filter

Classes and functions for matched-filtering.

This is designed for large-scale, multi-paralleled detection, with moderate to large numbers (hundreds to a few thousands) of templates.

Object-oriented API

EQcorrscan's matched-filter object oriented API is split into two halves, input (Template and Tribe objects) and output (Detection, Family and Party objects).

- A Tribe is a collection of Templates.
- A Family is a collection of Detections for a single Template.
- A Party is a collection of Families for a collection of Templates (a Tribe).

These objects retain useful meta-data including the obspy Event associated with the Template, and how the Template was processed. A core aim of the object-oriented API is reproducibility, and we encourage first-time users to adopt this from the off.

The methods of Template and Tribe objects mirror each other and in general you are best-off working with Tribes . Both the Template and Tribe objects have `detect` methods, which allow you to conduct matched-filtering.

The Detection, Family and Party objects contain all the metadata needed to re-create your detections. Furthermore, the Family and Party objects have a `lag_calc` method for conducting cross-correlation phase-picking based on correlation with the Template.

Object	Purpose
Template	To contain the template waveform and meta-data used to create the template.
Tribe	A collection of multiple Templates. Use the *detect* method to run matched-filter detections!
Detection	Root of the detection object tree - contains information relevant to a single detection from a single template.
Family	Collection of detections for a single Template.
Party	Collection of Family objects.

Function-based API

core.match_filter.matched_filter

Functions for network matched-filter detection of seismic data.

Designed to cross-correlate templates generated by template_gen function with data and output the detections.

copyright EQcorrscan developers.

license GNU Lesser General Public License, Version 3 (https://www.gnu.org/copyleft/lesser.html)

Functions

match_filter	Main matched-filter detection function.

eqcorrscan.core.match_filter.matched_filter.match_filter

eqcorrscan.core.match_filter.matched_filter.**match_filter**(*template_names*, *template_list*, *st*, *threshold*, *threshold_type*, *trig_int*, *plot=False*, *plotdir=None*, *xcorr_func=None*, *concurrency=None*, *cores=None*, *plot_format='png'*, *output_cat=False*, *output_event=True*, *extract_detections=False*, *arg_check=True*, *full_peaks=False*, *peak_cores=None*, *spike_test=True*, *copy_data=True*, *export_cccsums=False*, ***kwargs*)

Main matched-filter detection function.

Over-arching code to run the correlations of given templates with a day of seismic data and output the detections based on a given threshold. For a functional example see the tutorials.

Parameters

- **template_names** (*list*) – List of template names in the same order as template_list

- **template_list** (*list*) – A list of templates of which each template is a obspy.core.stream.Stream of obspy traces containing seismic data and header information.
- **st** (obspy.core.stream.Stream) – A Stream object containing all the data available and required for the correlations with templates given. For efficiency this should contain no excess traces which are not in one or more of the templates. This will now remove excess traces internally, but will copy the stream and work on the copy, leaving your input stream untouched.
- **threshold** (*float*) – A threshold value set based on the threshold_type
- **threshold_type** (*str*) – The type of threshold to be used, can be MAD, absolute or av_chan_corr. See Note on thresholding below.
- **trig_int** (*float*) – Minimum gap between detections from one template in seconds. If multiple detections occur within trig_int of one-another, the one with the highest cross-correlation sum will be selected.
- **plot** (*bool*) – Turn plotting on or off
- **plotdir** (*str*) – Path to plotting folder, plots will be output here, defaults to None, and plots are shown on screen.
- **xcorr_func** (*str or callable*) – A str of a registered xcorr function or a callable for implementing a custom xcorr function. For more information see: *eqcorrscan.utils.correlate.register_array_xcorr()*
- **concurrency** (*str*) – The type of concurrency to apply to the xcorr function. Options are 'multithread', 'multiprocess', 'concurrent'. For more details see *eqcorrscan.utils.correlate.get_stream_xcorr()*
- **cores** (*int*) – Number of cores to use
- **plot_format** (*str*) – Specify format of output plots if saved
- **output_cat** (*bool*) – Specifies if matched_filter will output an obspy.Catalog class containing events for each detection. Default is False, in which case matched_filter will output a list of detection classes, as normal.
- **output_event** (*bool*) – Whether to include events in the Detection objects, defaults to True, but for large cases you may want to turn this off as Event objects can be quite memory intensive.
- **extract_detections** (*bool*) – Specifies whether or not to return a list of streams, one stream per detection.
- **arg_check** (*bool*) – Check arguments, defaults to True, but if running in bulk, and you are certain of your arguments, then set to False.
- **full_peaks** (*bool*) – See :func: *eqcorrscan.utils.findpeaks.find_peaks_compiled*
- **peak_cores** (*int*) – Number of processes to use for parallel peak-finding (if different to *cores*).
- **spike_test** (*bool*) – If set True, raise error when there is a spike in data. defaults to True.
- **copy_data** (*bool*) – Whether to copy data to keep it safe, otherwise will edit your templates and stream in place.
- **export_cccsums** (*bool*) – Whether to save the cross-correlation statistic.

Note: When using the "fftw" correlation backend the length of the fft can be set. See *eqcorrscan.utils.correlate* for more info.

Note: Returns:

If neither *output_cat* or *extract_detections* are set to *True*, then only the list of `eqcorrscan.core.match_filter.Detection`'s will be output:

return `eqcorrscan.core.match_filter.Detection` detections for each detection made.

rtype list

If *output_cat* is set to *True*, then the `obspy.core.event.Catalog` will also be output:

return Catalog containing events for each detection, see above.

rtype `obspy.core.event.Catalog`

If *extract_detections* is set to *True* then the list of `obspy.core.stream.Stream`'s will also be output.

return list of `obspy.core.stream.Stream`'s for each detection, see above.

rtype list

Note: If your data contain gaps these must be padded with zeros before using this function. The *eqcorrscan.utils.pre_processing* functions will provide gap-filled data in the appropriate format. Note that if you pad your data with zeros before filtering or resampling the gaps will not be all zeros after filtering. This will result in the calculation of spurious correlations in the gaps.

Note: Detections are not corrected for *pre-pick*, the detection.detect_time corresponds to the beginning of the earliest template channel at detection.

Note: Data overlap:

Internally this routine shifts and trims the data according to the offsets in the template (e.g. if trace 2 starts 2 seconds after trace 1 in the template then the continuous data will be shifted by 2 seconds to align peak correlations prior to summing). Because of this, detections at the start and end of continuous data streams **may be missed**. The maximum time-period that might be missing detections is the maximum offset in the template.

To work around this, if you are conducting matched-filter detections through long-duration continuous data, we suggest using some overlap (a few seconds, on the order of the maximum offset in the templates) in the continous data. You will then need to post-process the detections (which should be done anyway to remove duplicates).

Note: Thresholding:

MAD threshold is calculated as the:

$$threshold \times (median(abs(cccsum)))$$

where $cccsum$ is the cross-correlation sum for a given template.

absolute threshold is a true absolute threshold based on the cccsum value.

av_chan_corr is based on the mean values of single-channel cross-correlations assuming all data are present as required for the template, e.g:

$$av\_chan\_corr\_thresh = threshold \times (cccsum \ / \ len(template))$$

where *template* is a single template from the input and the length is the number of channels within this template.

Note: The output_cat flag will create an `obspy.core.event.Catalog` containing one event for each `eqcorrscan.core.match_filter.Detection`'s generated by match_filter. Each event will contain a number of comments dealing with correlation values and channels used for the detection. Each channel used for the detection will have a corresponding `obspy.core.event.Pick` which will contain time and waveform information. **HOWEVER**, the user should note that the pick times do not account for the prepick times inherent in each template. For example, if a template trace starts 0.1 seconds before the actual arrival of that phase, then the pick time generated by match_filter for that phase will be 0.1 seconds early.

Note: xcorr_func can be used as follows:

```
>>> import obspy
>>> import numpy as np
>>> from eqcorrscan.core.match_filter.matched_filter import (
...    match_filter)
>>> from eqcorrscan.utils.correlate import time_multi_normxcorr
>>> # define a custom xcorr function
>>> def custom_normxcorr(templates, stream, pads, *args, **kwargs):
...     # Just to keep example short call other xcorr function
...     # in practice you would define your own function here
...     print('calling custom xcorr function')
...     return time_multi_normxcorr(templates, stream, pads)
>>> # generate some toy templates and stream
>>> random = np.random.RandomState(42)
>>> template = obspy.read()
>>> stream = obspy.read()
>>> for num, tr in enumerate(stream):  # iter st and embed templates
...     data = tr.data
...     tr.data = random.randn(6000) * 5
...     tr.data[100: 100 + len(data)] = data
>>> # call match_filter ane ensure the custom function is used
>>> detections = match_filter(
...     template_names=['1'], template_list=[template], st=stream,
...     threshold=.5, threshold_type='absolute', trig_int=1,
...     plotvar=False,
...     xcorr_func=custom_normxcorr)  # doctest:+ELLIPSIS
calling custom xcorr function...
```

core.match_filter.helpers

Helper functions for network matched-filter detection of seismic data.

copyright EQcorrscan developers.

license GNU Lesser General Public License, Version 3 (https://www.gnu.org/copyleft/lesser.html)

Functions

extract_from_stream	Extract waveforms for a list of detections from a stream.

Continued on next page

Table 2 – continued from previous page

normxcorr2	Thin wrapper to eqcorrscan.utils.correlate functions.
temporary_directory	make a temporary directory, yeild its name, cleanup on exit

eqcorrscan.core.match_filter.helpers.extract_from_stream

eqcorrscan.core.match_filter.helpers.**extract_from_stream**(*stream*, *detections*, *pad=5.0*, *length=30.0*)

Extract waveforms for a list of detections from a stream.

Parameters

- **stream** (*obspy.core.stream.Stream*) – Stream containing the detections.
- **detections** (*list*) – list of eqcorrscan.core.match_filter.detection
- **pad** (*float*) – Pre-detection extract time in seconds.
- **length** (*float*) – Total extracted length in seconds.

Returns list of obspy.core.stream.Stream, one for each detection.

Type list

eqcorrscan.core.match_filter.helpers.normxcorr2

eqcorrscan.core.match_filter.helpers.**normxcorr2**(*template*, *image*)

Thin wrapper to eqcorrscan.utils.correlate functions.

Parameters

- **template** (*numpy.ndarray*) – Template array
- **image** (*numpy.ndarray*) – Image to scan the template through. The order of these matters, if you put the template after the image you will get a reversed correlation matrix

Returns New numpy.ndarray of the correlation values for the correlation of the image with the template.

Return type numpy.ndarray

Note: If your data contain gaps these must be padded with zeros before using this function. The *eqcorrscan.utils.pre_processing* functions will provide gap-filled data in the appropriate format. Note that if you pad your data with zeros before filtering or resampling the gaps will not be all zeros after filtering. This will result in the calculation of spurious correlations in the gaps.

---

eqcorrscan.core.match_filter.helpers.temporary_directory

eqcorrscan.core.match_filter.helpers.**temporary_directory**()

make a temporary directory, yeild its name, cleanup on exit

subspace

This module contains functions relevant to executing subspace detection for earthquake catalogs.

We recommend that you read Harris' detailed report on subspace detection theory which can be found here: https://e-reports-ext.llnl.gov/pdf/335299.pdf

copyright EQcorrscan developers.

license GNU Lesser General Public License, Version 3 (https://www.gnu.org/copyleft/lesser.html)

Subspace detection for either single-channel cases, or network cases. This is modelled on the methods described by Harris. This method allows for slightly more variation in detected waveforms than the traditional matched-filter method. In this method, templates are constructed either by using the empirical subspace method, or by computing the basis vectors by singular-value decomposition. Both methods are provided as part of EQcorrscan in the clustering module.

Classes

class eqcorrscan.core.subspace.**Detector**(*name=None*, *sampling_rate=None*, *multiplex=None*, *stachans=None*, *lowcut=None*, *highcut=None*, *filt_order=None*, *data=None*, *u=None*, *sigma=None*, *v=None*, *dimension=None*)

Bases: object

Class to serve as the base for subspace detections.

Parameters

- **name** (*str*) – Name of subspace detector, used for book-keeping
- **sampling_rate** (*float*) – Sampling rate in Hz of original waveforms
- **multiplex** (*bool*) – Is this detector multiplexed.
- **stachans** (*list*) – List of tuples of (station, channel) used in detector. If multiplexed, these must be in the order that multiplexing was done.
- **lowcut** (*float*) – Lowcut filter in Hz
- **highcut** (*float*) – Highcut filter in Hz
- **filt_order** (*int*) – Number of corners for filtering
- **data** (*numpy.ndarray*) – The actual detector
- **u** (*numpy.ndarray*) – Full rank U matrix of left (input) singular vectors.
- **sigma** (*numpy.ndarray*) – Full rank vector of singular values.
- **v** (*numpy.ndarray*) – Full rank right (output) singular vectors.
- **dimension** (*int*) – Dimension of data.

Warning: Changing between scipy.linalg.svd solvers (obvious changes between scipy version 0.18.x and 0.19.0) result in sign changes in svd results. You should only run a detector created using the same scipy version as you currently run.

Methods

construct(streams, lowcut, highcut, ... [, ...])	Construct a subspace detector from a list of streams, full rank.
detect(st, threshold, trig_int[, moveout, ...])	Detect within continuous data using the subspace method.
energy_capture([stachans, size, show])	Calculate the average percentage energy capture for this subspace.
partition(dimension)	Partition subspace into desired dimension.
plot([stachans, size, show])	Plot the output basis vectors for the detector at the given dimension.

Continued on next page

Table 3 – continued from previous page

read(filename)	Read detector from a file, must be HDF5 format.
write(filename)	Write detector to a file - uses HDF5 file format.

__init__(*name=None*, *sampling_rate=None*, *multiplex=None*, *stachans=None*, *lowcut=None*, *highcut=None*, *filt_order=None*, *data=None*, *u=None*, *sigma=None*, *v=None*, *dimension=None*)

Initialize self. See help(type(self)) for accurate signature.

construct(*streams*, *lowcut*, *highcut*, *filt_order*, *sampling_rate*, *multiplex*, *name*, *align*, *shift_len=0*, *reject=0.3*, *no_missed=True*, *plot=False*)

Construct a subspace detector from a list of streams, full rank.

Subspace detector will be full-rank, further functions can be used to select the desired dimensions.

Parameters

- **streams** (*list*) – List of obspy.core.stream.Stream to be used to generate the subspace detector. These should be pre-clustered and aligned.
- **lowcut** (*float*) – Lowcut in Hz, can be None to not apply filter
- **highcut** (*float*) – Highcut in Hz, can be None to not apply filter
- **filt_order** (*int*) – Number of corners for filter.
- **sampling_rate** (*float*) – Desired sampling rate in Hz
- **multiplex** (*bool*) – Whether to multiplex the data or not. Data are multiplexed according to the method of Harris, see the multi function for details.
- **name** (*str*) – Name of the detector, used for book-keeping.
- **align** (*bool*) – Whether to align the data or not - needs to be done at some point
- **shift_len** (*float*) – Maximum shift allowed for alignment in seconds.
- **reject** (*float*) – Minimum correlation to include traces - only used if align=True.
- **no_missed** (*bool*) – Reject streams with missed traces, defaults to True. A missing trace from lots of events will reduce the quality of the subspace detector if multiplexed. Only used when multi is set to True.
- **plot** (*bool*) – Whether to plot the alignment stage or not.

Note: The detector will be normalized such that the data, before computing the singular-value decomposition, will have unit energy. e.g. We divide the amplitudes of the data by the L1 norm of the data.

Warning: EQcorrscan's alignment will attempt to align over the whole data window given. For long (more than 2s) chunks of data this can give poor results and you might be better off using the *eqcorrscan.utils.stacking.align_traces()* function externally, focusing on a smaller window of data. To do this you would align the data prior to running construct.

detect(*st*, *threshold*, *trig_int*, *moveout=0*, *min_trig=0*, *process=True*, *extract_detections=False*, *cores=1*)

Detect within continuous data using the subspace method.

Parameters

- **st** (*obspy.core.stream.Stream*) – Un-processed stream to detect within using the subspace detector.
- **threshold** (*float*) – Threshold value for detections between 0-1

- **trig_int** (*float*) – Minimum trigger interval in seconds.
- **moveout** (*float*) – Maximum allowable moveout window for non-multiplexed, network detection. See note.
- **min_trig** (*int*) – Minimum number of stations exceeding threshold for non-multiplexed, network detection. See note.
- **process** (*bool*) – Whether or not to process the stream according to the parameters defined by the detector. Default is True, which will process the data.
- **extract_detections** (*bool*) – Whether to extract waveforms for each detection or not, if True will return detections and streams.
- **cores** (*int*) – Number of threads to process data with.

Returns list of `eqcorrscan.core.match_filter.Detection`

Return type list

Warning: Subspace is currently in beta, see note in the subspace tutorial for information.

Note: If running in bulk with detectors that all have the same parameters then you can pre-process the data and set process to False. This will speed up this detect function dramatically.

Warning: If the detector and stream are multiplexed then they must contain the same channels and multiplexed in the same order. This is handled internally when process=True, but if running in bulk you must take care.

Note: Non-multiplexed, network detection. When the detector is not multiplexed, but there are multiple channels within the detector, we do not stack the single-channel detection statistics because we do not have a one-size-fits-all solution for computing delays for a subspace detector (if you want to implement one, then please contribute it!). Therefore, these parameters provide a means for declaring a network coincidence trigger using single-channel detection statistics, in a similar fashion to the commonly used network-coincidence trigger with energy detection statistics.

energy_capture(*stachans='all'*, *size=(10*, *7)*, *show=False*)
Calculate the average percentage energy capture for this subspace.

Returns Percentage energy capture

Return type float

partition(*dimension*)
Partition subspace into desired dimension.

Parameters **dimension** (*int*) – Maximum dimension to use.

plot(*stachans='all'*, *size=(10*, *7)*, *show=True*)
Plot the output basis vectors for the detector at the given dimension.

Corresponds to the first n horizontal vectors of the V matrix.

Parameters

- **stachans** (*list*) – list of tuples of station, channel pairs to plot.
- **stachans** – List of tuples of (station, channel) to use. Can set to 'all' to use all the station-channel pairs available. If detector is multiplexed, will just plot that.

- **size** (*tuple*) – Figure size.
- **show** (*bool*) – Whether or not to show the figure.

Returns Figure

Return type matplotlib.pyplot.Figure

Note: See *eqcorrscan.utils.plotting.subspace_detector_plot()* for example.

read(*filename*)
Read detector from a file, must be HDF5 format.

Reads a Detector object from an HDF5 file, usually created by eqcorrscan.

Parameters **filename** (*str*) – Filename to save the detector to.

write(*filename*)
Write detector to a file - uses HDF5 file format.

Meta-data are stored alongside numpy data arrays. See h5py.org for details of the methods.

Parameters **filename** (*str*) – Filename to save the detector to.

Functions

eqcorrscan.core.subspace.**read_detector**(*filename*)
Read detector from a filename.

Parameters **filename** (*str*) – Filename to save the detector to.

Returns Detector object

Return type *eqcorrscan.core.subspace.Detector*

eqcorrscan.core.subspace.**multi**(*stream*)
Internal multiplexer for multiplex_detect.

Parameters **stream** (*obspy.core.stream.Stream*) – Stream to multiplex

Returns trace of multiplexed data

Return type obspy.core.trace.Trace

Maps a standard multiplexed stream of seismic data to a single traces of multiplexed data as follows:

Input: x = [x1, x2, x3, . . .] y = [y1, y2, y3, . . .] z = [z1, z2, z3, . . .]

Output: xyz = [x1, y1, z1, x2, y2, z2, x3, y3, z3, . . .]

eqcorrscan.core.subspace.**subspace_detect**(*detectors*, *stream*, *threshold*, *trig_int*, *moveout=0*, *min_trig=1*, *parallel=True*, *num_cores=None*)
Conduct subspace detection with chosen detectors.

Parameters

- **detectors** (*list*) – list of *eqcorrscan.core.subspace.Detector* to be used for detection.
- **stream** (*obspy.core.stream.Stream*) – Stream to detect within.
- **threshold** (*float*) – Threshold between 0 and 1 for detection, see *Detector.detect()*
- **trig_int** (*float*) – Minimum trigger interval in seconds.
- **moveout** (*float*) – Maximum allowable moveout window for non-multiplexed, network detection. See note.

- **min_trig** (*int*) – Minimum number of stations exceeding threshold for non-multiplexed, network detection. See note in *Detector.detect()*.
- **parallel** (*bool*) – Whether to run detectors in parallel in groups.
- **num_cores** (*int*) – How many cpu cores to use if parallel==True. If set to None (default), will use all available cores.

Return type list

Returns List of `eqcorrscan.core.match_filter.Detection` detections.

Note: This will loop through your detectors using their detect method. If the detectors are multiplexed it will run groups of detectors with the same channels at the same time.

---

template_gen

Functions to generate template waveforms and information to go with them for the application of cross-correlation of seismic data for the detection of repeating events.

Note: All functions use obspy filters, which are implemented such that if both highcut and lowcut are set a bandpass filter will be used, but of highcut is not set (None) then a highpass filter will be used and if only the highcut is set then a lowpass filter will be used.

---

copyright EQcorrscan developers.

license GNU Lesser General Public License, Version 3 (https://www.gnu.org/copyleft/lesser.html)

Routines for cutting waveforms around picks for use as templates for matched filtering.

Functions

`eqcorrscan.core.template_gen.`**`template_gen`**(*method*, *lowcut*, *highcut*, *samp_rate*, *filt_order*, *length*, *prepick*, *swin='all'*, *process_len=86400*, *all_horiz=False*, *delayed=True*, *plot=False*, *plotdir=None*, *return_event=False*, *min_snr=None*, *parallel=False*, *num_cores=False*, *save_progress=False*, *skip_short_chans=False*, ***kwargs*)

Generate processed and cut waveforms for use as templates.

Parameters

- **method** (*str*) – Template generation method, must be one of ('from_client', 'from_seishub', 'from_sac', 'from_meta_file'). - Each method requires associated arguments, see note below.
- **lowcut** (*float*) – Low cut (Hz), if set to None will not apply a lowcut.
- **highcut** (*float*) – High cut (Hz), if set to None will not apply a highcut.
- **samp_rate** (*float*) – New sampling rate in Hz.
- **filt_order** (*int*) – Filter level (number of corners).
- **length** (*float*) – Length of template waveform in seconds.
- **prepick** (*float*) – Pre-pick time in seconds
- **swin** (*str*) – P, S, P_all, S_all or all, defaults to all: see note in *eqcorrscan.core.template_gen.template_gen()*

- **process_len** (*int*) – Length of data in seconds to download and process.
- **all_horiz** (*bool*) – To use both horizontal channels even if there is only a pick on one of them. Defaults to False.
- **delayed** (*bool*) – If True, each channel will begin relative to it's own pick-time, if set to False, each channel will begin at the same time.
- **plot** (*bool*) – Plot templates or not.
- **plotdir** (*str*) – The path to save plots to. If *plotdir=None* (default) then the figure will be shown on screen.
- **return_event** (*bool*) – Whether to return the event and process length or not.
- **min_snr** (*float*) – Minimum signal-to-noise ratio for a channel to be included in the template, where signal-to-noise ratio is calculated as the ratio of the maximum amplitude in the template window to the rms amplitude in the whole window given.
- **parallel** (*bool*) – Whether to process data in parallel or not.
- **num_cores** (*int*) – Number of cores to try and use, if False and parallel=True, will use either all your cores, or as many traces as in the data (whichever is smaller).
- **save_progress** (*bool*) – Whether to save the resulting templates at every data step or not. Useful for long-running processes.
- **skip_short_chans** (*bool*) – Whether to ignore channels that have insufficient length data or not. Useful when the quality of data is not known, e.g. when downloading old, possibly triggered data from a datacentre

Returns List of `obspy.core.stream.Stream` Templates

Return type list

Note: By convention templates are generated with P-phases on the vertical channel and S-phases on the horizontal channels, normal seismograph naming conventions are assumed, where Z denotes vertical and N, E, R, T, 1 and 2 denote horizontal channels, either oriented or not. To this end we will **only** use Z channels if they have a P-pick, and will use one or other horizontal channels **only** if there is an S-pick on it.

Warning: If there is no phase_hint included in picks, and swin=all, all channels with picks will be used.

Note: If swin=all, then all picks will be used, not just phase-picks (e.g. it will use amplitude picks). If you do not want this then we suggest that you remove any picks you do not want to use in your templates before using the event.

Note: *Method specific arguments:*

- ***from_client* requires:**

  param str client_id string passable by obspy to generate Client, or any object with a *get_waveforms* method, including a Client instance.

  param *obspy.core.event.Catalog* catalog Catalog of events to generate template for

  param float data_pad Pad length for data-downloads in seconds

- ***from_seishub* requires:**

  param str url url to seishub database

param *obspy.core.event.Catalog* catalog Catalog of events to generate template for

param float data_pad Pad length for data-downloads in seconds

- ***from_sac* requires:**

  param list sac_files osbpy.core.stream.Stream of sac waveforms, or list of paths to sac waveforms.

Note: See *eqcorrscan.utils.sac_util.sactoevent* for details on how pick information is collected.

- ***from_meta_file* requires:**

  param str meta_file Path to obspy-readable event file, or an obspy Catalog

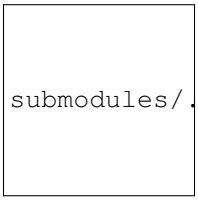
  param *obspy.core.stream.Stream* st Stream containing waveform data for template. Note that this should be the same length of stream as you will use for the continuous detection, e.g. if you detect in day-long files, give this a day-long file!

  param bool process Whether to process the data or not, defaults to True.

Note: process_len should be set to the same length as used when computing detections using match_filter.match_filter, e.g. if you read in day-long data for match_filter, process_len should be 86400.

Example

```
>>> from obspy.clients.fdsn import Client
>>> from eqcorrscan.core.template_gen import template_gen
>>> client = Client('NCEDC')
>>> catalog = client.get_events(eventid='72572665', includearrivals=True)
>>> # We are only taking two picks for this example to speed up the
>>> # example, note that you don't have to!
>>> catalog[0].picks = catalog[0].picks[0:2]
>>> templates = template_gen(
...     method='from_client', catalog=catalog, client_id='NCEDC',
...     lowcut=2.0, highcut=9.0, samp_rate=20.0, filt_order=4, length=3.0,
...     prepick=0.15, swin='all', process_len=300, all_horiz=True)
>>> templates[0].plot(equal_scale=False, size=(800,600)) # doctest: +SKIP
```



Example

```
>>> from obspy import read
>>> from eqcorrscan.core.template_gen import template_gen
>>> # Get the path to the test data
>>> import eqcorrscan
>>> import os
>>> TEST_PATH = os.path.dirname(eqcorrscan.__file__) + '/tests/test_data'
>>> st = read(TEST_PATH + '/WAV/TEST_/' +
```

(continues on next page)

(continued from previous page)

```
...           '2013-09-01-0410-35.DFDPC_024_00')
>>> quakeml = TEST_PATH + '/20130901T041115.xml'
>>> templates = template_gen(
...     method='from_meta_file', meta_file=quakeml, st=st, lowcut=2.0,
...     highcut=9.0, samp_rate=20.0, filt_order=3, length=2, prepick=0.1,
...     swin='S', all_horiz=True)
>>> print(len(templates[0]))
10
>>> templates = template_gen(
...     method='from_meta_file', meta_file=quakeml, st=st, lowcut=2.0,
...     highcut=9.0, samp_rate=20.0, filt_order=3, length=2, prepick=0.1,
...     swin='S_all', all_horiz=True)
>>> print(len(templates[0]))
15
```

Example

```
>>> from eqcorrscan.core.template_gen import template_gen
>>> import glob
>>> # Get all the SAC-files associated with one event.
>>> sac_files = glob.glob(TEST_PATH + '/SAC/2014p611252/*')
>>> templates = template_gen(
...     method='from_sac', sac_files=sac_files, lowcut=2.0, highcut=10.0,
...     samp_rate=25.0, filt_order=4, length=2.0, swin='all', prepick=0.1,
...     all_horiz=True)
>>> print(templates[0][0].stats.sampling_rate)
25.0
>>> print(len(templates[0]))
15
```

eqcorrscan.core.template_gen.**extract_from_stack**(*stack*, *template*, *length*, *pre_pick*, *pre_pad*, *Z_include=False*, *pre_processed=True*, *samp_rate=None*, *lowcut=None*, *highcut=None*, *filt_order=3*)

Extract a multiplexed template from a stack of detections.

Function to extract a new template from a stack of previous detections. Requires the stack, the template used to make the detections for the stack, and we need to know if the stack has been pre-processed.

Parameters

- **stack** (*obspy.core.stream.Stream*) – Waveform stack from detections. Can be of any length and can have delays already included, or not.
- **template** (*obspy.core.stream.Stream*) – Template used to make the detections in the stack. Will use the delays of this for the new template.
- **length** (*float*) – Length of new template in seconds
- **pre_pick** (*float*) – Extract additional data before the detection, seconds
- **pre_pad** (*float*) – Pad used in seconds when extracting the data, e.g. the time before the detection extracted. If using clustering.extract_detections this half the length of the extracted waveform.
- **Z_include** (*bool*) – If True will include any Z-channels even if there is no template for this channel, as long as there is a template for this station at a different channel. If this is False and Z channels are included in the template Z channels will be included in the new_template anyway.

- **pre_processed** (*bool*) – Have the data been pre-processed, if True (default) then we will only cut the data here.
- **samp_rate** (*float*) – If pre_processed=False then this is required, desired sampling rate in Hz, defaults to False.
- **lowcut** (*float*) – If pre_processed=False then this is required, lowcut in Hz, defaults to False.
- **highcut** (*float*) – If pre_processed=False then this is required, highcut in Hz, defaults to False
- **filt_order** (*int*) – If pre_processed=False then this is required, filter order, defaults to False

Returns Newly cut template.

Return type obspy.core.stream.Stream

2.6.2 Utils

Various utility functions to help the core routines, and/or for use to analyse the results of the core routines.

archive_read

Helper functions for reading data from archives for the EQcorrscan package.

Note: These functions are tools to aid simplification of general scripts, they do not cover all use cases, however if you have a use case you want to see here, then let the authors know, or implement it yourself and contribute it back to the project.

copyright EQcorrscan developers.

license GNU Lesser General Public License, Version 3 (https://www.gnu.org/copyleft/lesser.html)

Classes & Functions

read_data	Function to read the appropriate data from an archive for a day.

eqcorrscan.utils.archive_read.read_data

eqcorrscan.utils.archive_read.**read_data**(*archive*, *arc_type*, *day*, *stachans*, *length=86400*)

Function to read the appropriate data from an archive for a day.

Parameters

- **archive** (*str*) – The archive source - if arc_type is seishub, this should be a url, if the arc_type is FDSN then this can be either a url or a known obspy client. If arc_type is day_vols, then this is the path to the top directory.
- **arc_type** (*str*) – The type of archive, can be: seishub, FDSN, day_volumes
- **day** (*datetime.date*) – Date to retrieve data for
- **stachans** (*list*) – List of tuples of Stations and channels to try and get, will not fail if stations are not available, but will warn.
- **length** (*float*) – Data length to extract in seconds, defaults to 1 day.

Returns Stream of data

Return type obspy.core.stream.Stream

Note: A note on arc_types, if arc_type is day_vols, then this will look for directories labelled in the IRIS DMC conventions of Yyyyy/Rjjj.01/... where yyyy is the year and jjj is the julian day. Data within these files directories should be stored as day-long, single-channel files. This is not implemented in the fasted way possible to allow for a more general situation. If you require more speed you will need to re-write this.

Example

```
>>> from obspy import UTCDateTime
>>> t1 = UTCDateTime(2012, 3, 26)
>>> stachans = [('JCNB', 'SP1')]
>>> st = read_data('NCEDC', 'FDSN', t1, stachans)
>>> print(st)
1 Trace(s) in Stream:
BP.JCNB.40.SP1 | 2012-03-26T00:00:00.000000Z - 2012-03-26T23:59:59.950000Z |␣
↪20.0 Hz, 1728000 samples
```

Example, missing data

```
>>> t1 = UTCDateTime(2012, 3, 26)
>>> stachans = [('JCNB', 'SP1'), ('GCSZ', 'HHZ')]
>>> st = read_data('NCEDC', 'FDSN', t1, stachans)
>>> print(st)
1 Trace(s) in Stream:
BP.JCNB.40.SP1 | 2012-03-26T00:00:00.000000Z - 2012-03-26T23:59:59.950000Z |␣
↪20.0 Hz, 1728000 samples
```

Example, local day-volumes

```
>>> # Get the path to the test data
>>> import eqcorrscan
>>> TEST_PATH = os.path.dirname(eqcorrscan.__file__) + '/tests/test_data'
>>> t1 = UTCDateTime(2012, 3, 26)
>>> stachans = [('WHYM', 'SHZ'), ('EORO', 'SHZ')]
>>> st = read_data(TEST_PATH + '/day_vols', 'day_vols',
...                t1, stachans)
>>> print(st)
2 Trace(s) in Stream:
AF.WHYM..SHZ | 2012-03-26T00:00:00.000000Z - 2012-03-26T23:59:59.000000Z | 1.0␣
↪Hz, 86400 samples
AF.EORO..SHZ | 2012-03-26T00:00:00.000000Z - 2012-03-26T23:59:59.000000Z | 1.0␣
↪Hz, 86400 samples
```

Private Functions

_check_available_data	Function to check what stations are available in the archive for a given day.
_get_station_file	Helper function to find the correct file.

eqcorrscan.utils.archive_read._check_available_data

eqcorrscan.utils.archive_read.**_check_available_data**(*archive*, *arc_type*, *day*)
Function to check what stations are available in the archive for a given day.

Parameters

- **archive** (*str*) – The archive source
- **arc_type** (*str*) – The type of archive, can be:
- **day** (*datetime.date*) – Date to retrieve data for

Returns list of tuples of (station, channel) as available.

Note: Currently the seishub options are untested.

eqcorrscan.utils.archive_read._get_station_file

eqcorrscan.utils.archive_read.**_get_station_file**(*path_name*, *station*, *channel*)
Helper function to find the correct file.

Parameters **path_name** (*str*) – Path to files to check.

Returns list of filenames, str

catalog_to_dd

Functions to generate hypoDD input files from catalogs.

copyright EQcorrscan developers.

license GNU Lesser General Public License, Version 3 (https://www.gnu.org/copyleft/lesser.html)

Classes & Functions

compute_differential_times	Generate groups of differential times for a catalog.
read_phase	Read hypoDD phase files into Obspy catalog class.
write_catalog	Generate a dt.ct file for hypoDD for a series of events.
write_correlations	Write a dt.cc file for hypoDD input for a given list of events.
write_event	Write obspy.core.event.Catalog to a hypoDD format event.dat file.
write_phase	Write a phase.dat formatted file from an obspy catalog.
write_station	Write a hypoDD formatted station file.

eqcorrscan.utils.catalog_to_dd.compute_differential_times

eqcorrscan.utils.catalog_to_dd.**compute_differential_times**(*catalog*, *correlation*, *stream_dict=None*, *event_id_mapper=None*, *max_sep=8.0*, *min_link=8*, *min_cc=None*, *extract_len=None*, *pre_pick=None*, *shift_len=None*, *interpolate=False*, *max_workers=None*, **args*, ***kwargs*)

Generate groups of differential times for a catalog.

Parameters

- **catalog** (*obspy.core.event.Catalog*) – Catalog of events to get differential times for
- **correlation** (*bool*) – If True will generate cross-correlation derived differential-times for a dt.cc file. If false, will generate catalog times for a dt.ct file.
- **stream_dict** (*dict*) – Dictionary of streams keyed by event-id (the event.resource_id.id, NOT the hypoDD event-id)
- **event_id_mapper** (*dict*) – Dictionary mapping event resource id to an integer event id for hypoDD. If this is None, or missing events then the dictionary will be updated to include appropriate event-ids. This should be of the form {event.resource_id.id: integer_id}
- **max_sep** (*float*) – Maximum hypocentral separation in km to link events
- **min_link** (*int*) – Minimum shared phase observations to link events
- **min_cc** (*float*) – Threshold to include cross-correlation results.
- **extract_len** (*float*) – Length in seconds to extract around the pick
- **pre_pick** (*float*) – Time before the pick to start the correlation window
- **shift_len** (*float*) – Time (+/-) to allow pick to vary in seconds. e.g. if shift_len is set to 1s, the pick will be allowed to shift between pick_time - 1 and pick_time + 1.
- **interpolate** (*bool*) – Whether to interpolate correlations or not. Allows subsample accuracy
- **max_workers** (*int*) – Maximum number of workers for parallel processing. If None then all threads will be used - only used if correlation = True

Return type dict

Returns Dictionary of differential times keyed by event id.

Return type dict

Returns Dictionary of event_id_mapper

Note: The arguments min_cc, stream_dict, extract_len, pre_pick, shift_len and interpolate are only required if correlation=True.

---

eqcorrscan.utils.catalog_to_dd.read_phase

eqcorrscan.utils.catalog_to_dd.**read_phase**(*ph_file*)

Read hypoDD phase files into Obspy catalog class.

Parameters **ph_file** (*str*) – Phase file to read event info from.

Returns Catalog of events from file.

Return type obspy.core.event.Catalog

```
>>> from obspy.core.event.catalog import Catalog
>>> # Get the path to the test data
>>> import eqcorrscan
>>> import os
>>> TEST_PATH = os.path.dirname(eqcorrscan.__file__) + '/tests/test_data'
>>> catalog = read_phase(TEST_PATH + '/tunnel.phase')
>>> isinstance(catalog, Catalog)
True
```

eqcorrscan.utils.catalog_to_dd.write_catalog

eqcorrscan.utils.catalog_to_dd.**write_catalog**(*catalog*, *event_id_mapper=None*, *max_sep=8*, *min_link=8*)

Generate a dt.ct file for hypoDD for a series of events.

Parameters

- **catalog** (*obspy.core.event.Catalog*) – Catalog of events
- **event_id_mapper** (*dict*) – Dictionary mapping event resource id to an integer event id for hypoDD. If this is None, or missing events then the dictionary will be updated to include appropriate event-ids. This should be of the form {event.resource_id.id: integer_id}
- **max_sep** (*float*) – Maximum separation between event pairs in km
- **min_link** (*int*) – Minimum links for an event to be paired, e.g. minimum number of picks from the same station and channel (and phase) that are shared between two events for them to be paired.

Returns event_id_mapper

eqcorrscan.utils.catalog_to_dd.write_correlations

eqcorrscan.utils.catalog_to_dd.**write_correlations**(*catalog*, *stream_dict*, *extract_len*, *pre_pick*, *shift_len*, *event_id_mapper=None*, *lowcut=1.0*, *highcut=10.0*, *max_sep=8*, *min_link=8*, *min_cc=0.0*, *interpolate=False*, *max_workers=None*, *parallel_process=False*, **args*, ***kwargs*)

Write a dt.cc file for hypoDD input for a given list of events.

Takes an input list of events and computes pick refinements by correlation. Outputs a single files, dt.cc file with weights as the square of the cross-correlation.

Parameters

- **catalog** (*obspy.core.event.Catalog*) – Catalog of events to get differential times for

- **event_id_mapper** (*dict*) – Dictionary mapping event resource id to an integer event id for hypoDD. If this is None, or missing events then the dictionary will be updated to include appropriate event-ids. This should be of the form {event.resource_id.id: integer_id}
- **stream_dict** (*dict*) – Dictionary of streams keyed by event-id (the event.resource_id.id, NOT the hypoDD event-id)
- **extract_len** (*float*) – Length in seconds to extract around the pick
- **pre_pick** (*float*) – Time before the pick to start the correlation window
- **shift_len** (*float*) – Time to allow pick to vary in seconds
- **lowcut** (*float*) – Lowcut in Hz - set to None to apply no lowcut
- **highcut** (*float*) – Highcut in Hz - set to None to apply no highcut
- **max_sep** (*float*) – Maximum separation between event pairs in km
- **min_link** (*int*) – Minimum links for an event to be paired
- **min_cc** (*float*) – Threshold to include cross-correlation results.
- **interpolate** (*bool*) – Whether to interpolate correlations or not. Allows subsample accuracy
- **max_workers** (*int*) – Maximum number of workers for parallel processing. If None then all threads will be used.
- **parallel_process** (*bool*) – Whether to process streams in parallel or not. Experimental, may use too much memory.

Return type dict

Returns event_id_mapper

Note: You can provide processed waveforms, or let this function filter your data for you. Filtering is undertaken by detrending and bandpassing with a 8th order zerophase butterworth filter.

---

eqcorrscan.utils.catalog_to_dd.write_event

eqcorrscan.utils.catalog_to_dd.**write_event**(*catalog*, *event_id_mapper=None*)

Write obspy.core.event.Catalog to a hypoDD format event.dat file.

Parameters

- **catalog** (*obspy.core.event.Catalog*) – A catalog of obspy events.
- **event_id_mapper** (*dict*) – Dictionary mapping event resource id to an integer event id for hypoDD. If this is None, or missing events then the dictionary will be updated to include appropriate event-ids. This should be of the form {event.resource_id.id: integer_id}

Returns dictionary of event-id mappings.

eqcorrscan.utils.catalog_to_dd.write_phase

eqcorrscan.utils.catalog_to_dd.**write_phase**(*catalog*, *event_id_mapper=None*)

Write a phase.dat formatted file from an obspy catalog.

Parameters

- **catalog** (*obspy.core.event.Catalog*) – Catalog of events

- **event_id_mapper** (*dict*) – Dictionary mapping event resource id to an integer event id for hypoDD. If this is None, or missing events then the dictionary will be updated to include appropriate event-ids. This should be of the form {event.resource_id.id: integer_id}

Returns event_id_mapper

eqcorrscan.utils.catalog_to_dd.write_station

eqcorrscan.utils.catalog_to_dd.**write_station**(*inventory*, *use_elevation=False*, *filename='station.dat'*)

Write a hypoDD formatted station file.

Parameters

- **inventory** (*obspy.core.Inventory*) – Inventory of stations to write - should include channels if use_elevation=True to incorporate channel depths.
- **use_elevation** (*bool*) – Whether to write elevations (requires hypoDD >= 2)
- **filename** (*str*) – File to write stations to.

catalog_utils

Helper functions for common handling tasks for catalog objects.

Note: These functions are tools to aid simplification of general scripts, they do not cover all use cases, however if you have a use case you want to see here, then let the authors know, or implement it yourself and contribute it back to the project, or, if its really good, give it to the obspy guys!

---

copyright EQcorrscan developers.

license GNU Lesser General Public License, Version 3 (https://www.gnu.org/copyleft/lesser.html)

Classes & Functions

filter_picks	Filter events in the catalog based on a number of parameters.

eqcorrscan.utils.catalog_utils.filter_picks

eqcorrscan.utils.catalog_utils.**filter_picks**(*catalog*, *stations=None*, *channels=None*, *networks=None*, *locations=None*, *top_n_picks=None*, *evaluation_mode='all'*)

Filter events in the catalog based on a number of parameters.

Parameters

- **catalog** (*obspy.core.event.Catalog*) – Catalog to filter.
- **stations** (*list*) – List for stations to keep picks from.
- **channels** (*list*) – List of channels to keep picks from.
- **networks** (*list*) – List of networks to keep picks from.
- **locations** (*list*) – List of location codes to use
- **top_n_picks** (*int*) – Filter only the top N most used station-channel pairs.

- **evaluation_mode** (*str*) – To select only manual or automatic picks, or use all (default).

Returns Filtered Catalog - if events are left with no picks, they are removed from the catalog.

Return type obspy.core.event.Catalog

Note: Will filter first by station, then by channel, then by network, if using top_n_picks, this will be done last, after the other filters have been applied.

Note: Doesn't work in place on the catalog, your input catalog will be safe unless you overwrite it.

Note: Doesn't expand wildcard characters.

Example

```
>>> from obspy.clients.fdsn import Client
>>> from eqcorrscan.utils.catalog_utils import filter_picks
>>> from obspy import UTCDateTime
>>> client = Client('NCEDC')
>>> t1 = UTCDateTime(2004, 9, 28)
>>> t2 = t1 + 86400
>>> catalog = client.get_events(starttime=t1, endtime=t2, minmagnitude=3,
...                             minlatitude=35.7, maxlatitude=36.1,
...                             minlongitude=-120.6, maxlongitude=-120.2,
...                             includearrivals=True)
>>> print(len(catalog))
12
>>> filtered_catalog = filter_picks(catalog, stations=['BMS', 'BAP',
...                                                    'PAG', 'PAN',
...                                                    'PBI', 'PKY',
...                                                    'YEG', 'WOF'])
>>> print(len(filtered_catalog))
12
>>> stations = []
>>> for event in filtered_catalog:
...     for pick in event.picks:
...         stations.append(pick.waveform_id.station_code)
>>> print(sorted(list(set(stations))))
['BAP', 'BMS', 'PAG', 'PAN', 'PBI', 'PKY', 'WOF', 'YEG']
```

clustering

Functions to cluster seismograms by a range of constraints.

copyright EQcorrscan developers.

license GNU Lesser General Public License, Version 3 (https://www.gnu.org/copyleft/lesser.html)

Classes & Functions

svd	Compute the SVD of a number of templates.

Continued on next page

Table 8 – continued from previous page

svd_to_stream	Convert the singular vectors output by SVD to streams.
catalog_cluster	Cluster a catalog by distance only.
cluster	Cluster template waveforms based on average correlations.
corr_cluster	Group traces based on correlations above threshold with the stack.
cross_chan_correlation	Calculate cross-channel correlation.
dist_mat_km	Compute the distance matrix for a catalog using hypocentral separation.
distance_matrix	Compute distance matrix for waveforms based on cross-correlations.
empirical_svd	Empirical subspace detector generation function.
extract_detections	Extract waveforms associated with detections
group_delays	Group template waveforms according to their arrival times (delays).
re_thresh_csv	Remove detections by changing the threshold.
space_time_cluster	Cluster detections in space and time.

eqcorrscan.utils.clustering.svd

eqcorrscan.utils.clustering.**svd**(*stream_list*, *full=False*)

Compute the SVD of a number of templates.

Returns the singular vectors and singular values of the templates.

Parameters

- **stream_list** (*List of :class: obspy.Stream*) – List of the templates to be analysed
- **full** (*bool*) – Whether to compute the full input vector matrix or not.

Returns SValues(list) for each channel, SVectors(list of ndarray), UVectors(list of ndarray) for each channel, stachans, List of String (station.channel)

Note: We recommend that you align the data before computing the SVD, e.g., the P-arrival on all templates for the same channel should appear at the same time in the trace. See the stacking.align_traces function for a way to do this.

Note: Uses the numpy.linalg.svd function, their U, s and V are mapped to UVectors, SValues and SVectors respectively. Their V (and ours) corresponds to V.H.

---

eqcorrscan.utils.clustering.svd_to_stream

eqcorrscan.utils.clustering.**svd_to_stream**(*uvectors*, *stachans*, *k*, *sampling_rate*)

Convert the singular vectors output by SVD to streams.

One stream will be generated for each singular vector level, for all channels. Useful for plotting, and aiding seismologists thinking of waveforms!

Parameters

- **svectors** (*list*) – List of numpy.ndarray Singular vectors
- **stachans** (*list*) – List of station.channel Strings

- **k** (*int*) – Number of streams to return = number of SV's to include
- **sampling_rate** (*float*) – Sampling rate in Hz

Returns svstreams, List of obspy.core.stream.Stream, with svStreams[0] being composed of the highest rank singular vectors.

eqcorrscan.utils.clustering.catalog_cluster

eqcorrscan.utils.clustering.**catalog_cluster**(*catalog*, *thresh*, *metric='distance'*, *show=True*)

Cluster a catalog by distance only.

Will compute the matrix of physical distances between events and utilize the scipy.clustering.hierarchy module to perform the clustering.

Parameters

- **catalog** (*obspy.core.event.Catalog*) – Catalog of events to clustered
- **thresh** (*float*) – Maximum separation, either in km (*metric="distance"*) or in seconds (*metric="time"*)
- **metric** (*str*) – Either "distance" or "time"

Returns list of obspy.core.event.Catalog objects

Return type list

```
>>> from eqcorrscan.utils.clustering import catalog_cluster
>>> from obspy.clients.fdsn import Client
>>> from obspy import UTCDateTime
>>> client = Client("NCEDC")
>>> starttime = UTCDateTime("2002-01-01")
>>> endtime = UTCDateTime("2002-02-01")
>>> cat = client.get_events(starttime=starttime, endtime=endtime,
...                         minmagnitude=2)
>>> groups = catalog_cluster(catalog=cat, thresh=2, show=False)
```

```
>>> from eqcorrscan.utils.clustering import catalog_cluster
>>> from obspy.clients.fdsn import Client
>>> from obspy import UTCDateTime
>>> client = Client("https://earthquake.usgs.gov")
>>> starttime = UTCDateTime("2002-01-01")
>>> endtime = UTCDateTime("2002-02-01")
>>> cat = client.get_events(starttime=starttime, endtime=endtime,
...                         minmagnitude=6)
>>> groups = catalog_cluster(catalog=cat, thresh=1000, metric="time",
...     show=False)
```

eqcorrscan.utils.clustering.cluster

eqcorrscan.utils.clustering.**cluster**(*template_list*, *show=True*, *corr_thresh=0.3*, *shift_len=0*, *allow_individual_trace_shifts=True*, *save_corrmat=False*, *cores='all'*)

Cluster template waveforms based on average correlations.

Function to take a set of templates and cluster them, will return groups as lists of streams. Clustering is done by computing the cross-channel correlation sum of each stream in stream_list with every other stream in the list. scipy.cluster.hierarchy functions are then used to compute the complete distance matrix, where distance is 1 minus the normalised cross-correlation sum such that larger distances are less similar events. Groups are then created by clustering the distance matrix at distances less than 1 - corr_thresh.

Will compute the distance matrix in parallel, using all available cores

Parameters

- **template_list** (*list*) – List of tuples of the template (obspy.core.stream.Stream) and the template id to compute clustering for
- **show** (*bool*) – plot linkage on screen if True, defaults to True
- **corr_thresh** (*float*) – Cross-channel correlation threshold for grouping
- **shift_len** (*float*) – How many seconds to allow the templates to shift
- **allow_individual_trace_shifts** (*bool*) – Controls whether templates are shifted by shift_len in relation to the picks as a whole, or whether each trace can be shifted individually. Defaults to True.
- **save_corrmat** (*bool*) – If True will save the distance matrix to dist_mat.npy in the local directory.
- **cores** (*int*) – number of cores to use when computing the distance matrix, defaults to 'all' which will work out how many cpus are available and hog them.

Returns List of groups. Each group is a list of obspy.core.stream.Stream making up that group.

eqcorrscan.utils.clustering.corr_cluster

eqcorrscan.utils.clustering.**corr_cluster**(*trace_list*, *thresh=0.9*)

Group traces based on correlations above threshold with the stack.

Will run twice, once with 80% of threshold threshold to remove large outliers that would negatively affect the stack, then again with your threshold.

Parameters

- **trace_list** (*list*) – List of obspy.core.stream.Trace to compute similarity between
- **thresh** (*float*) – Correlation threshold between -1-1

Returns numpy.ndarray of bool of whether that trace correlates well enough (above your given threshold) with the stack.

Note: We recommend that you align the data before computing the clustering, e.g., the P-arrival on all templates for the same channel should appear at the same time in the trace. See the *eqcorrscan.utils.stacking.align_traces()* function for a way to do this.

---

eqcorrscan.utils.clustering.cross_chan_correlation

eqcorrscan.utils.clustering.**cross_chan_correlation**(*st1*, *streams*, *shift_len=0.0*, *allow_individual_trace_shifts=True*, *xcorr_func='fftw'*, *concurrency='concurrent'*, *cores=1*, ***kwargs*)

Calculate cross-channel correlation.

Determine the cross-channel correlation between two streams of multichannel seismic data.

Parameters

- **st1** (*obspy.core.stream.Stream*) – Stream one
- **streams** (*list*) – Streams to compare to.

- **shift_len** (*float*) – How many seconds for templates to shift
- **allow_individual_trace_shifts** (*bool*) – Controls whether templates are shifted by shift_len in relation to the picks as a whole, or whether each trace can be shifted individually. Defaults to True.
- **xcorr_func** (*str, callable*) – The method for performing correlations. Accepts either a string or callable. See *eqcorrscan.utils.correlate.register_array_xcorr()* for more details
- **concurrency** (*str*) – Concurrency for xcorr-func.
- **cores** (*int*) – Number of threads to parallel over

Returns cross channel correlation, float - normalized by number of channels. locations of maximums

Return type numpy.ndarray, numpy.ndarray

Note: If no matching channels were found then the coherance and index for that stream will be nan.

eqcorrscan.utils.clustering.dist_mat_km

eqcorrscan.utils.clustering.**dist_mat_km**(*catalog*, *num_threads=None*)

Compute the distance matrix for a catalog using hypocentral separation.

Will give physical distance in kilometers.

Parameters **catalog** (*obspy.core.event.Catalog*) – Catalog for which to compute the distance matrix

Returns distance matrix

Return type numpy.ndarray

eqcorrscan.utils.clustering.distance_matrix

eqcorrscan.utils.clustering.**distance_matrix**(*stream_list*, *shift_len=0.0*, *allow_individual_trace_shifts=True*, *cores=1*)

Compute distance matrix for waveforms based on cross-correlations.

Function to compute the distance matrix for all templates - will give distance as 1-abs(cccoh), e.g. a well correlated pair of templates will have small distances, and an equally well correlated reverse image will have the same distance as a positively correlated image - this is an issue.

Parameters

- **stream_list** (*list*) – List of the obspy.core.stream.Stream to compute the distance matrix for
- **shift_len** (*float*) – How many seconds for templates to shift
- **allow_individual_trace_shifts** (*bool*) – Controls whether templates are shifted by shift_len in relation to the picks as a whole, or whether each trace can be shifted individually. Defaults to True.
- **cores** (*int*) – Number of cores to parallel process using, defaults to 1.

Returns

- distance matrix (numpy.ndarray) of size len(stream_list)**2

- shift matrix (numpy.ndarray) containing shifts between traces of the sorted streams. Size is len(stream_list)**2 * x, where x is 1 for shift_len=0 and/or allow_individual_trace_shifts=False. Missing correlations are indicated by nans.
- shift dict (dict): dictionary of (template_id: trace_dict) where trace_dict contains (trace.id: shift matrix (size *len(stream_list)**2*) for trace.id)

Warning: Because distance is given as $1 - abs(coherence)$, negatively correlated and positively correlated objects are given the same distance.

Note: Requires all traces to have the same sampling rate and same length.

eqcorrscan.utils.clustering.empirical_svd

eqcorrscan.utils.clustering.**empirical_svd**(*stream_list*, *linear=True*)

Empirical subspace detector generation function.

Takes a list of templates and computes the stack as the first order subspace detector, and the differential of this as the second order subspace detector following the empirical subspace method of Barrett & Beroza, 2014 - SRL.

Parameters

- **stream_list** (*list*) – list of streams to compute the subspace detectors from, where streams are obspy.core.stream.Stream objects.
- **linear** (*bool*) – Set to true by default to compute the linear stack as the first subspace vector, False will use the phase-weighted stack as the first subspace vector.

Returns list of two obspy.core.stream.Stream s

eqcorrscan.utils.clustering.extract_detections

eqcorrscan.utils.clustering.**extract_detections**(*detections*, *templates*, *archive*, *arc_type*, *extract_len=90.0*, *outdir=None*, *extract_Z=True*, *additional_stations=[]*)

Extract waveforms associated with detections

Takes a list of detections for the template, template. Waveforms will be returned as a list of obspy.core.stream.Stream containing segments of extract_len. They will also be saved if outdir is set. The default is unset. The default extract_len is 90 seconds per channel.

Parameters

- **detections** (*list*) – List of eqcorrscan.core.match_filter.Detection.
- **templates** (*list*) – A list of tuples of the template name and the template Stream used to detect detections.
- **archive** (*str*) – Either name of archive or path to continuous data, see *eqcorrscan.utils.archive_read()* for details
- **arc_type** (*str*) – Type of archive, either seishub, FDSN, day_vols
- **extract_len** (*float*) – Length to extract around the detection (will be equally cut around the detection time) in seconds. Default is 90.0.

- **outdir** (*str*) – Default is None, with None set, no files will be saved, if set each detection will be saved into this directory with files named according to the detection time, NOT than the waveform start time. Detections will be saved into template subdirectories. Files written will be multiplexed miniseed files, the encoding will be chosen automatically and will likely be float.
- **extract_Z** (*bool*) – Set to True to also extract Z channels for detections delays will be the same as horizontal channels, only applies if only horizontal channels were used in the template.
- **additional_stations** (*list*) – List of tuples of (station, channel) to also extract data for using an average delay.

Returns list of obspy.core.streams.Stream

Return type list

```
>>> from eqcorrscan.utils.clustering import extract_detections
>>> from eqcorrscan.core.match_filter import Detection
>>> from obspy import read, UTCDateTime
>>> # Get the path to the test data
>>> import eqcorrscan
>>> import os
>>> TEST_PATH = os.path.dirname(eqcorrscan.__file__) + '/tests/test_data'
>>> # Use some dummy detections, you would use real one
>>> detections = [Detection(
...     template_name='temp1', detect_time=UTCDateTime(2012, 3, 26, 9, 15),
...     no_chans=2, chans=['WHYM', 'EORO'], detect_val=2, threshold=1.2,
...     typeofdet='corr', threshold_type='MAD', threshold_input=8.0),
...                Detection(
...     template_name='temp2', detect_time=UTCDateTime(2012, 3, 26, 18, 5),
...     no_chans=2, chans=['WHYM', 'EORO'], detect_val=2, threshold=1.2,
...     typeofdet='corr', threshold_type='MAD', threshold_input=8.0)]
>>> archive = os.path.join(TEST_PATH, 'day_vols')
>>> template_files = [os.path.join(TEST_PATH, 'temp1.ms'),
...                   os.path.join(TEST_PATH, 'temp2.ms')]
>>> templates = [('temp' + str(i), read(filename))
...              for i, filename in enumerate(template_files)]
>>> extracted = extract_detections(detections, templates,
...                                archive=archive, arc_type='day_vols')
>>> print(extracted[0].sort())
2 Trace(s) in Stream:
AF.EORO..SHZ | 2012-03-26T09:14:15.000000Z - 2012-03-26T09:15:45.000000Z | 1.0␣
↪Hz, 91 samples
AF.WHYM..SHZ | 2012-03-26T09:14:15.000000Z - 2012-03-26T09:15:45.000000Z | 1.0␣
↪Hz, 91 samples
>>> print(extracted[1].sort())
2 Trace(s) in Stream:
AF.EORO..SHZ | 2012-03-26T18:04:15.000000Z - 2012-03-26T18:05:45.000000Z | 1.0␣
↪Hz, 91 samples
AF.WHYM..SHZ | 2012-03-26T18:04:15.000000Z - 2012-03-26T18:05:45.000000Z | 1.0␣
↪Hz, 91 samples
>>> # Extract from stations not included in the detections
>>> extracted = extract_detections(
...     detections, templates, archive=archive, arc_type='day_vols',
...     additional_stations=[('GOVA', 'SHZ')])
>>> print(extracted[0].sort())
3 Trace(s) in Stream:
AF.EORO..SHZ | 2012-03-26T09:14:15.000000Z - 2012-03-26T09:15:45.000000Z | 1.0␣
↪Hz, 91 samples
AF.GOVA..SHZ | 2012-03-26T09:14:15.000000Z - 2012-03-26T09:15:45.000000Z | 1.0␣
↪Hz, 91 samples
AF.WHYM..SHZ | 2012-03-26T09:14:15.000000Z - 2012-03-26T09:15:45.000000Z | 1.0␣
↪Hz, 91 samples
```

(continues on next page)

(continued from previous page)

```
>>> # The detections can be saved to a file:
>>> extract_detections(detections, templates, archive=archive,
...                    arc_type='day_vols',
...                    additional_stations=[('GOVA', 'SHZ')], outdir='.')
```

eqcorrscan.utils.clustering.group_delays

eqcorrscan.utils.clustering.**group_delays**(*stream_list*)

Group template waveforms according to their arrival times (delays).

Parameters **stream_list** (*list*) – List of obspy.core.stream.Stream waveforms you want to group.

Returns list of List of obspy.core.stream.Stream where each initial list is a group with the same delays.

Return type list

eqcorrscan.utils.clustering.re_thresh_csv

eqcorrscan.utils.clustering.**re_thresh_csv**(*path*, *old_thresh*, *new_thresh*, *chan_thresh*)

Remove detections by changing the threshold.

Can only be done to remove detection by increasing threshold, threshold lowering will have no effect.

Parameters

- **path** (*str*) – Path to the .csv detection file
- **old_thresh** (*float*) – Old threshold MAD multiplier
- **new_thresh** (*float*) – New threshold MAD multiplier
- **chan_thresh** (*int*) – Minimum number of channels for a detection

Returns List of detections

Return type list

Example

```
>>> from eqcorrscan.utils.clustering import re_thresh_csv
>>> # Get the path to the test data
>>> import eqcorrscan
>>> import os
>>> TEST_PATH = os.path.dirname(eqcorrscan.__file__) + '/tests/test_data'
>>> det_file = os.path.join(TEST_PATH, 'expected_tutorial_detections.txt')
>>> detections = re_thresh_csv(path=det_file, old_thresh=8, new_thresh=10,
...                            chan_thresh=3)
>>> print(len(detections))
17
```

Note: This is a legacy function, and will read detections from all versions.

Warning: Only works if thresholding was done by MAD.

eqcorrscan.utils.clustering.space_time_cluster

eqcorrscan.utils.clustering.**space_time_cluster**(*catalog*, *t_thresh*, *d_thresh*)

Cluster detections in space and time.

Use to separate repeaters from other events. Clusters by distance first, then removes events in those groups that are at different times.

Parameters

- **catalog** (*obspy.core.event.Catalog*) – Catalog of events to clustered
- **t_thresh** (*float*) – Maximum inter-event time threshold in seconds
- **d_thresh** (*float*) – Maximum inter-event distance in km

Returns list of obspy.core.event.Catalog objects

Return type list

```
>>> from eqcorrscan.utils.clustering import space_time_cluster
>>> from obspy.clients.fdsn import Client
>>> from obspy import UTCDateTime
>>> client = Client("https://earthquake.usgs.gov")
>>> starttime = UTCDateTime("2002-01-01")
>>> endtime = UTCDateTime("2002-02-01")
>>> cat = client.get_events(starttime=starttime, endtime=endtime,
...                         minmagnitude=6)
>>> groups = space_time_cluster(catalog=cat, t_thresh=86400, d_thresh=1000)
```

correlate

Correlation functions for multi-channel cross-correlation of seismic data.

Various routines used mostly for testing, including links to a compiled routine using FFTW, a Numpy fft routine which uses bottleneck for normalisation and a compiled time-domain routine. These have varying levels of efficiency, both in terms of overall speed, and in memory usage. The time-domain is the most memory efficient but slowest routine (although fast for small cases of less than a few hundred correlations), the Numpy routine is fast, but memory inefficient due to a need to store large double-precision arrays for normalisation. The fftw compiled routine is faster and more memory efficient than the Numpy routine.

copyright EQcorrscan developers.

license GNU Lesser General Public License, Version 3 (https://www.gnu.org/copyleft/lesser.html)

Classes & Functions

fftw_multi_normxcorr	Use a C loop rather than a Python loop - in some cases this will be fast.
fftw_normxcorr	Normalised cross-correlation using the fftw library.
numpy_normxcorr	Compute the normalized cross-correlation using numpy and bottleneck.
time_multi_normxcorr	Compute cross-correlations in the time-domain using C routine.
get_array_xcorr	Get an normalized cross correlation function that takes arrays as inputs.
get_stream_xcorr	Return a function for performing normalized cross correlation on lists of streams.
register_array_xcorr	Decorator for registering correlation functions.

eqcorrscan.utils.correlate.fftw_multi_normxcorr

eqcorrscan.utils.correlate.**fftw_multi_normxcorr**(*template_array*, *stream_array*, *pad_array*, *seed_ids*, *cores_inner*, *stack=True*, **args*, ***kwargs*)

Use a C loop rather than a Python loop - in some cases this will be fast.

Parameters

- **template_array** (*dict*) –
- **stream_array** (*dict*) –
- **pad_array** (*dict*) –
- **seed_ids** (*list*) –

rtype: np.ndarray, list :return: 3D Array of cross-correlations and list of used channels.

eqcorrscan.utils.correlate.fftw_normxcorr

eqcorrscan.utils.correlate.**fftw_normxcorr**(*templates*, *stream*, *pads*, *threaded=False*, **args*, ***kwargs*)

Normalised cross-correlation using the fftw library.

Internally this function used double precision numbers, which is definitely required for seismic data. Cross-correlations are computed as the inverse fft of the dot product of the ffts of the stream and the reversed, normalised, templates. The cross-correlation is then normalised using the running mean and standard deviation (not using the N-1 correction) of the stream and the sums of the normalised templates.

This python function wraps the C-library written by C. Chamberlain for this purpose.

Parameters

- **templates** (*np.ndarray*) – 2D Array of templates
- **stream** (*np.ndarray*) – 1D array of continuous data
- **pads** (*list*) – List of ints of pad lengths in the same order as templates
- **threaded** (*bool*) – Whether to use the threaded routine or not - note openMP and python multiprocessing don't seem to play nice for this.

Returns np.ndarray of cross-correlations

Returns np.ndarray channels used

eqcorrscan.utils.correlate.numpy_normxcorr

eqcorrscan.utils.correlate.**numpy_normxcorr**(*templates*, *stream*, *pads*, **args*, ***kwargs*)

Compute the normalized cross-correlation using numpy and bottleneck.

Parameters

- **templates** (*np.ndarray*) – 2D Array of templates
- **stream** (*np.ndarray*) – 1D array of continuous data
- **pads** (*list*) – List of ints of pad lengths in the same order as templates

Returns np.ndarray of cross-correlations

Returns np.ndarray channels used

eqcorrscan.utils.correlate.time_multi_normxcorr

eqcorrscan.utils.correlate.**time_multi_normxcorr**(*templates*, *stream*, *pads*, *threaded=False*, **args*, ***kwargs*)

Compute cross-correlations in the time-domain using C routine.

Parameters

- **templates** (*np.ndarray*) – 2D Array of templates
- **stream** (*np.ndarray*) – 1D array of continuous data
- **pads** (*list*) – List of ints of pad lengths in the same order as templates
- **threaded** (*bool*) – Whether to use the threaded routine or not

Returns np.ndarray of cross-correlations

Returns np.ndarray channels used

eqcorrscan.utils.correlate.get_array_xcorr

eqcorrscan.utils.correlate.**get_array_xcorr**(*name_or_func=None*)

Get an normalized cross correlation function that takes arrays as inputs.

See `eqcorrscan.utils.correlate.array_normxcorr()` for expected function signature.

Parameters **name_or_func** (*str or callable*) – Either a name of a registered xcorr function or a callable that implements the standard array_normxcorr signature.

Returns callable wth array_normxcorr interface

see also *eqcorrscan.utils.correlate.get_stream_xcorr()*

eqcorrscan.utils.correlate.get_stream_xcorr

eqcorrscan.utils.correlate.**get_stream_xcorr**(*name_or_func=None*, *concurrency=None*)

Return a function for performing normalized cross correlation on lists of streams.

Parameters

- **name_or_func** – Either a name of a registered function or a callable that implements the standard array_normxcorr signature.
- **concurrency** – Optional concurrency strategy, options are below.

Returns A callable with the interface of stream_normxcorr

Concurrency options

- multithread - use a threadpool for concurrency;
- multiprocess - use a process pool for concurrency;
- concurrent - use a customized concurrency strategy for the function, if not defined threading will be used.

eqcorrscan.utils.correlate.register_array_xcorr

eqcorrscan.utils.correlate.**register_array_xcorr**(*name*, *func=None*, *is_default=False*)

Decorator for registering correlation functions.

Each function must have the same interface as numpy_normxcorr, which is *f(templates, stream, pads, *args, **kwargs)* any number of specific kwargs can be used.

Register_normxcorr can be used as a decorator (with or without arguments) or as a callable.

Parameters

- **name** (*str, callable*) – The name of the function for quick access, or the callable that will be wrapped when used as a decorator.
- **func** (*callable, optional*) – The function to register
- **is_default** (*bool*) – True if this function should be marked as default normxcorr

Returns callable

All functions within this module (and therefore all correlation functions used in EQcorrscan) are normalised cross-correlations, which follow the definition that Matlab uses for .

In the time-domain this is as follows

$$c(t) = \frac{\sum_y (a_y - \overline{a})(b_{t+y} - \overline{b_t})}{\sqrt{\sum_y (a_y - \overline{a})(a_y - \overline{a}) \sum_y (b_{t+y} - \overline{b_t})(b_{t+y} - \overline{b_t})}}$$

where $c(t)$ is the normalised cross-correlation at at time t, a is the template window which has length y, b is the continuous data, and $\overline{a}$ is the mean of the template and $\overline{b_t}$ is the local mean of the continuous data.

In practice the correlations only remove the mean of the template once, but we do remove the mean from the continuous data at every time-step. Without doing this (just removing the global mean), correlations are affected by jumps in average amplitude, which is most noticeable during aftershock sequences, or when using short templates.

In the frequency domain (functions *eqcorrscan.utils.correlate.numpy_normxcorr()*, *eqcorrscan.utils.correlate.fftw_normxcorr()*, *fftw_multi_normxcorr()*), correlations are computed using an equivalent method. All methods are tested against one-another to check for internal consistency, and checked against results computed using Matlab's function. These routines also give the same (within 0.00001) of openCV cross-correlations.

Selecting a correlation function

EQcorrscan strives to use sensible default algorithms for calculating correlation values, however, you may want to change how correlations are caclulated to be more advantageous to your specific needs.

There are currently 3 different correlations functions currently included in EQcorrscan:

1. *eqcorrscan.utils.correlate.numpy_normxcorr()* known as "numpy"
2. *eqcorrscan.utils.correlate.time_multi_normxcorr()* known as "time_domain"
3. *eqcorrscan.utils.correlate.fftw_normxcorr()* known as "fftw"

Number 3 is the default.

Setting FFT length

For version 0.4.0 onwards, the "fftw" backend allows the user to pass an *fft_len* keyword to it. This will set the length of the FFT internally. Smaller FFT's use less memory, but can be faster. Larger FFTs use more memory and can be slower. By default this is set to the minimum of 2 ** 13, or the next fast length of the sum of the template length and the stream length. showed that 2 ** 13 was consistently fastest over a range of data shapes on an intel i7 with 8-threads. Powers of two are generally fastest.

Using Fast Matched Filter within EQcorrscan

provides fast time-domain correlations for both CPU and GPU architectures. For massively multi-threaded environment this runs faster than the frequency-domain routines native to EQcorrscan (when more than 40 CPU cores, or an NVIDIA GPU card is available) with less memory consumption. Documentation on how to call Fast Matched Filter from within EQcorrscan is provided here: fast_matched_filter

Switching which correlation function is used

You can switch between correlation functions using the *xcorr_func* parameter included in: - `eqcorrscan.core.match_filter.match_filter()`, - `eqcorrscan.core.match_filter.Tribe.detect()`,- `eqcorrscan.core.match_filter.Template.detect()`

by:

1. passing a string (eg "numpy", "time_domain", or "fftw") or;
2. passing a function

for example:

```
>>> import obspy
>>> import numpy as np
>>> from eqcorrscan.utils.correlate import numpy_normxcorr, set_xcorr
>>> from eqcorrscan.core.match_filter import match_filter


>>> # generate some toy templates and stream
>>> random = np.random.RandomState(42)
>>> template = obspy.read()
>>> stream = obspy.read()
>>> for num, tr in enumerate(stream):  # iter stream and embed templates
...     data = tr.data
...     tr.data = random.randn(6000) * 5
...     tr.data[100: 100 + len(data)] = data

>>> # do correlation using numpy rather than fftw
>>> detections = match_filter(['1'], [template], stream, .5, 'absolute',
...                           1, False, xcorr_func='numpy')

>>> # do correlation using a custom function
>>> def custom_normxcorr(templates, stream, pads, *args, **kwargs):
...     # Just to keep example short call other xcorr function
...     print('calling custom xcorr function')
...     return numpy_normxcorr(templates, stream, pads, *args, **kwargs)

>>> detections = match_filter(
...     ['1'], [template], stream, .5, 'absolute', 1, False,
...     xcorr_func=custom_normxcorr)
calling custom xcorr function...
```

You can also use the set_xcorr object (eqcorrscan.utils.correlate.set_xcorr) to change which correlation function is used. This can be done permanently or within the scope of a context manager:

```
>>> # change the default xcorr function for all code in the with block
>>> with set_xcorr(custom_normxcorr):
...     detections = match_filter(['1'], [template], stream, .5,
...                               'absolute', 1, False)
calling custom xcorr function...

>>> # permanently set the xcorr function (until the python kernel restarts)
```

(continues on next page)

(continued from previous page)

```
>>> set_xcorr(custom_normxcorr)
default changed to custom_normxcorr
>>> detections = match_filter(['1'], [template], stream, .5, 'absolute',
...                           1, False)
calling custom xcorr function...
>>> set_xcorr.revert()  # change it back to the previous state
```

Notes on accuracy

To cope with floating-point rounding errors, correlations may not be calculated for data with low variance. If you see a warning saying: *"Some correlations not computed, are there zeros in the data? If not consider increasing gain"*, check whether your data have zeros, and if not, but have low, but real amplitudes, multiply your data by some value.

Warning

If data are padded with zeros prior to filtering then correlations may be incorrectly calculated where there are zeros. You should always pad after filtering. If you see warnings saying *"Low variance, possible zeros, check result"*, you should check that you have padded correctly and check that correlations haven't been calculated when you don't expect them.

despike

Functions for despiking seismic data.

Warning: Despike functions are in beta, they do not work that well.

Todo: Deconvolve spike to remove it, find peaks in the f-domain.

copyright EQcorrscan developers.

license GNU Lesser General Public License, Version 3 (https://www.gnu.org/copyleft/lesser.html)

Classes & Functions

median_filter	Filter out spikes in data above a multiple of MAD of the data.
template_remove	Looks for instances of template in the trace and removes the matches.

eqcorrscan.utils.despike.median_filter

eqcorrscan.utils.despike.**median_filter**(*tr*, *multiplier=10*, *windowlength=0.5*, *interp_len=0.05*)

Filter out spikes in data above a multiple of MAD of the data.

Currently only has the ability to replaces spikes with linear interpolation. In the future we would aim to fill the gap with something more appropriate. Works in-place on data.

Parameters

- **tr** (*obspy.core.trace.Trace*) – trace to despike
- **multiplier** (*float*) – median absolute deviation multiplier to find spikes above.
- **windowlength** (*float*) – Length of window to look for spikes in in seconds.
- **interp_len** (*float*) – Length in seconds to interpolate around spikes.

Returns obspy.core.trace.Trace

Warning: Not particularly effective, and may remove earthquake signals, use with caution.

eqcorrscan.utils.despike.template_remove

eqcorrscan.utils.despike.**template_remove**(*tr*, *template*, *cc_thresh*, *windowlength*, *interp_len*)

Looks for instances of template in the trace and removes the matches.

Parameters

- **tr** (*obspy.core.trace.Trace*) – Trace to remove spikes from.
- **template** (*osbpy.core.trace.Trace*) – Spike template to look for in data.
- **cc_thresh** (*float*) – Cross-correlation threshold (-1 - 1).
- **windowlength** (*float*) – Length of window to look for spikes in in seconds.
- **interp_len** (*float*) – Window length to remove and fill in seconds.

Returns tr, works in place.

Return type obspy.core.trace.Trace

findpeaks

Functions to find peaks in data above a certain threshold.

copyright EQcorrscan developers.

license GNU Lesser General Public License, Version 3 (https://www.gnu.org/copyleft/lesser.html)

Classes & Functions

coin_trig	Find network coincidence triggers within peaks of detection statistics.
decluster	Decluster peaks based on an enforced minimum separation.
find_peaks_compiled	Determine peaks in an array of data above a certain threshold.
find_peaks2_short	Determine peaks in an array of data above a certain threshold.
multi_find_peaks	Wrapper for find-peaks for multiple arrays.

eqcorrscan.utils.findpeaks.coin_trig

eqcorrscan.utils.findpeaks.**coin_trig**(*peaks*, *stachans*, *samp_rate*, *moveout*, *min_trig*, *trig_int*)

Find network coincidence triggers within peaks of detection statistics.

Useful for finding network detections from sets of detections on individual stations.

Parameters

- **peaks** (*list*) – List of lists of tuples of (peak, index) for each station-channel. Index should be in samples.
- **stachans** (*list*) – List of tuples of (station, channel) in the order of peaks.
- **samp_rate** (*float*) – Sampling rate in Hz
- **moveout** (*float*) – Allowable network moveout in seconds.
- **min_trig** (*int*) – Minimum station-channels required to declare a trigger.
- **trig_int** (*float*) – Minimum allowable time between network triggers in seconds.

Returns List of tuples of (peak, index), for the earliest detected station.

Return type list

```
>>> peaks = [[(0.5, 100), (0.3, 800)], [(0.4, 120), (0.7, 850)]]
>>> triggers = coin_trig(peaks, [('a', 'Z'), ('b', 'Z')], 10, 3, 2, 1)
>>> print(triggers)
[(0.45, 100)]
```

eqcorrscan.utils.findpeaks.decluster

eqcorrscan.utils.findpeaks.**decluster**(*peaks*, *index*, *trig_int*, *threshold=0*)

Decluster peaks based on an enforced minimum separation.

Parameters

- **peaks** (*np.array*) – array of peak values
- **index** (*np.ndarray*) – locations of peaks
- **trig_int** (*int*) – Minimum trigger interval in samples
- **threshold** (*float*) – Minimum absolute peak value to retain it.

Returns list of tuples of (value, sample)

eqcorrscan.utils.findpeaks.find_peaks_compiled

eqcorrscan.utils.findpeaks.**find_peaks_compiled**(*arr*, *thresh*, *trig_int*, *full_peaks=False*)

Determine peaks in an array of data above a certain threshold.

Parameters

- **arr** (*numpy.ndarray*) – 1-D numpy array is required
- **thresh** (*float*) – The threshold below which will be considered noise and peaks will not be found in.
- **trig_int** (*int*) – The minimum difference in samples between triggers, if multiple peaks within this window this code will find the highest.
- **full_peaks** (*bool*) – If True, will decluster within data-sections above the threshold, rather than just taking the peak within that section. This will take more time. This defaults to False for match_filter.

Returns peaks: Lists of tuples of peak values and locations.

Return type list

eqcorrscan.utils.findpeaks.find_peaks2_short

eqcorrscan.utils.findpeaks.**find_peaks2_short**(*arr*, *thresh*, *trig_int*, *full_peaks=False*)

Determine peaks in an array of data above a certain threshold.

Uses a mask to remove data below threshold and finds peaks in what is left.

Parameters

- **arr** (*numpy.ndarray*) – 1-D numpy array is required
- **thresh** (*float*) – The threshold below which will be considered noise and peaks will not be found in.
- **trig_int** (*int*) – The minimum difference in samples between triggers, if multiple peaks within this window this code will find the highest.
- **full_peaks** (*bool*) – If True will by decluster within data-sections above the threshold, rather than just taking the peak within that section. This will take more time. This defaults to False for match_filter.

Returns peaks: Lists of tuples of peak values and locations.

Return type list

```
>>> import numpy as np
>>> arr = np.random.randn(100)
>>> threshold = 10
>>> arr[40] = 20
>>> arr[60] = 100
>>> find_peaks2_short(arr, threshold, 3)
[(20.0, 40), (100.0, 60)]
```

eqcorrscan.utils.findpeaks.multi_find_peaks

eqcorrscan.utils.findpeaks.**multi_find_peaks**(*arr*, *thresh*, *trig_int*, *parallel=True*, *full_peaks=False*, *cores=None*, *internal_func=<function find_peaks_compiled>*)

Wrapper for find-peaks for multiple arrays.

Parameters

- **arr** (*numpy.ndarray*) – 2-D numpy array is required
- **thresh** (*list*) – The threshold below which will be considered noise and peaks will not be found in. One threshold per array.
- **trig_int** (*int*) – The minimum difference in samples between triggers, if multiple peaks within this window this code will find the highest.
- **parallel** (*bool*) – Whether to compute in parallel or not - will use multiprocessing if not using the compiled internal_func
- **full_peaks** (*bool*) – See *eqcorrscan.utils.findpeaks.find_peaks2_short*
- **cores** (*int*) – Maximum number of processes to spin up for parallel peak-finding
- **internal_func** (*callable*) – Function to use for peak finding - defaults to the compiled version.

Returns List of list of tuples of (peak, index) in same order as input arrays

mag_calc

Functions to aid magnitude estimation.

copyright EQcorrscan developers.

license GNU Lesser General Public License, Version 3 (https://www.gnu.org/copyleft/lesser.html)

Classes & Functions

amp_pick_event	Pick amplitudes for local magnitude for a single event.
calc_b_value	Calculate the b-value for a range of completeness magnitudes.
calc_max_curv	Calculate the magnitude of completeness using the maximum curvature method.
dist_calc	Function to calculate the distance in km between two points.
svd_moments	Calculate relative moments/amplitudes using singular-value decomposition.
relative_amplitude	Compute the relative amplitudes between two streams.
relative_magnitude	Compute the relative magnitudes between two events.

eqcorrscan.utils.mag_calc.amp_pick_event

eqcorrscan.utils.mag_calc.**amp_pick_event**(*event*, *st*, *inventory*, *chans=('Z'*, *)*, *var_wintype=True*, *winlen=0.9*, *pre_pick=0.2*, *pre_filt=True*, *lowcut=1.0*, *highcut=20.0*, *corners=4*, *min_snr=1.0*, *plot=False*, *remove_old=False*, *ps_multiplier=0.34*, *velocity=False*, *water_level=0*, *iaspei_standard=False*)

Pick amplitudes for local magnitude for a single event.

Looks for maximum peak-to-trough amplitude for a channel in a stream, and picks this amplitude and period. There are a few things it does internally to stabilise the result:

1. Applies a given filter to the data using obspy's bandpass filter. The filter applied is a time-domain digital SOS filter. This is often necessary for small magnitude earthquakes. To correct for this filter later the gain of the filter at the period of the maximum amplitude is retrieved using scipy's sosfreqz, and used to divide the resulting picked amplitude.

2. Picks the peak-to-trough amplitude, but records half of this to cope with possible DC offsets.

3. The maximum amplitude within the given window is picked. Care must be taken to avoid including surface waves in the window;

4. A variable window-length is used by default that takes into account P-S times if available, this is in an effort to include only the body waves. When P-S times are not available the ps_multiplier variable is used, which defaults to 0.34 x hypocentral distance.

Parameters

- **event** (*obspy.core.event.event.Event*) – Event to pick
- **st** (*obspy.core.stream.Stream*) – Stream associated with event
- **inventory** (*obspy.core.inventory.Inventory*) – Inventory containing response information for the stations in st.

- **chans** (*tuple*) – Tuple of the components to pick on, e.g. (Z, 1, 2, N, E)
- **var_wintype** (*bool*) – If True, the winlen will be multiplied by the P-S time if both P and S picks are available, otherwise it will be multiplied by the hypocentral distance*ps_multiplier, defaults to True
- **winlen** (*float*) – Length of window, see above parameter, if var_wintype is False then this will be in seconds, otherwise it is the multiplier to the p-s time, defaults to 0.9.
- **pre_pick** (*float*) – Time before the s-pick to start the cut window, defaults to 0.2.
- **pre_filt** (*bool*) – To apply a pre-filter or not, defaults to True
- **lowcut** (*float*) – Lowcut in Hz for the pre-filter, defaults to 1.0
- **highcut** (*float*) – Highcut in Hz for the pre-filter, defaults to 20.0
- **corners** (*int*) – Number of corners to use in the pre-filter
- **min_snr** (*float*) – Minimum signal-to-noise ratio to allow a pick - see note below on signal-to-noise ratio calculation.
- **plot** (*bool*) – Turn plotting on or off.
- **remove_old** (*bool*) – If True, will remove old amplitudes and associated picks from event and overwrite with new picks. Defaults to False.
- **ps_multiplier** (*float*) – A p-s time multiplier of hypocentral distance - defaults to 0.34, based on p-s ratio of 1.68 and an S-velocity 0f 1.5km/s, deliberately chosen to be quite slow.
- **velocity** (*bool*) – Whether to make the pick in velocity space or not. Original definition of local magnitude used displacement of Wood-Anderson, MLv in seiscomp and Antelope uses a velocity measurement. *velocity and iaspei_standard are mutually exclusive*.
- **water_level** (*float*) – Water-level for seismometer simulation, see https://docs.obspy.org/packages/autogen/obspy.core.trace.Trace.remove_response.html
- **iaspei_standard** (*bool*) – Whether to output amplitude in IASPEI standard IAML (wood-anderson static amplification of 1), or AML with wood-anderson static amplification of 2080. Note: Units are SI (and specified in the amplitude)

Returns Picked event

Return type `obspy.core.event.Event`

Note: Signal-to-noise ratio is calculated using the filtered data by dividing the maximum amplitude in the signal window (pick window) by the normalized noise amplitude (taken from the whole window supplied).

Note: With *iaspei_standard=False*, picks will be returned in SI units (m or m/s), with the standard Wood-Anderson sensitivity of 2080 applied such that the measurements reflect the amplitude measured on a Wood Anderson instrument, as per the original local magnitude definitions of Richter and others.

---

eqcorrscan.utils.mag_calc.calc_b_value

`eqcorrscan.utils.mag_calc.`**`calc_b_value`**(*magnitudes*, *completeness*, *max_mag=None*, *plotvar=True*)

Calculate the b-value for a range of completeness magnitudes.

Calculates a power-law fit to given magnitudes for each completeness magnitude. Plots the b-values and residuals for the fitted catalogue against the completeness values. Computes fits using numpy.polyfit, which uses a least-squares technique.

Parameters

- **magnitudes** (*list*) – Magnitudes to compute the b-value for.
- **completeness** (*list*) – list of completeness values to compute b-values for.
- **max_mag** (*float*) – Maximum magnitude to attempt to fit in magnitudes.
- **plotvar** (*bool*) – Turn plotting on or off.

Return type list

Returns List of tuples of (completeness, b-value, residual, number of magnitudes used)

Note: High "residuals" indicate better fit. Residuals are calculated according to the Wiemer & Wyss 2000, Minimum Magnitude of Completeness in Earthquake Catalogs: Examples from Alaska, the Western United States, and Japan, BSSA.

Example

```
>>> from obspy.clients.fdsn import Client
>>> from obspy import UTCDateTime
>>> from eqcorrscan.utils.mag_calc import calc_b_value
>>> client = Client('IRIS')
>>> t1 = UTCDateTime('2012-03-26T00:00:00')
>>> t2 = t1 + (3 * 86400)
>>> catalog = client.get_events(starttime=t1, endtime=t2, minmagnitude=3)
>>> magnitudes = [event.magnitudes[0].mag for event in catalog]
>>> b_values = calc_b_value(magnitudes, completeness=np.arange(3, 7, 0.2),
...                         plotvar=False)
>>> round(b_values[4][1], 1)
1.0
>>> # We can set a maximum magnitude:
>>> b_values = calc_b_value(magnitudes, completeness=np.arange(3, 7, 0.2),
...                         plotvar=False, max_mag=5)
>>> round(b_values[4][1], 1)
1.0
```

eqcorrscan.utils.mag_calc.calc_max_curv

eqcorrscan.utils.mag_calc.**calc_max_curv**(*magnitudes*, *bin_size=0.5*, *plotvar=False*)

Calculate the magnitude of completeness using the maximum curvature method.

Parameters

- **magnitudes** (*list or numpy array*) – List of magnitudes from which to compute the maximum curvature which will give an estimate of the magnitude of completeness given the assumption of a power-law scaling.
- **bin_size** (*float*) – Width of magnitude bins used to compute the non-cumulative distribution
- **plotvar** (*bool*) – Turn plotting on and off

Return type float

Returns Magnitude at maximum curvature

Note: Should be used as a guide, often under-estimates Mc.

Example

```
>>> import numpy as np
>>> mags = np.arange(3, 6, .1)
>>> N = 10 ** (5 - 1 * mags)
>>> magnitudes = [0, 2, 3, 2.5, 2.2, 1.0]  # Some below completeness
>>> for mag, n in zip(mags, N):
...     magnitudes.extend([mag for _ in range(int(n))])
>>> calc_max_curv(magnitudes, plotvar=False)
3.0
```

eqcorrscan.utils.mag_calc.dist_calc

eqcorrscan.utils.mag_calc.**dist_calc**(*loc1*, *loc2*)

Function to calculate the distance in km between two points.

Uses the haversine formula to calculate great circle distance at the Earth's surface, then uses trig to include depth.

Parameters

- **loc1** (*tuple*) – Tuple of lat, lon, depth (in decimal degrees and km)
- **loc2** (*tuple*) – Tuple of lat, lon, depth (in decimal degrees and km)

Returns Distance between points in km.

Return type float

eqcorrscan.utils.mag_calc.svd_moments

eqcorrscan.utils.mag_calc.**svd_moments**(*u*, *s*, *v*, *stachans*, *event_list*, *n_svs=2*)

Calculate relative moments/amplitudes using singular-value decomposition.

Convert basis vectors calculated by singular value decomposition (see the SVD functions in clustering) into relative moments.

For more information see the paper by Rubinstein & Ellsworth (2010).

Parameters

- **u** (*list*) – List of the numpy.ndarray input basis vectors from the SVD, one array for each channel used.
- **s** (*list*) – List of the numpy.ndarray of singular values, one array for each channel.
- **v** (*list*) – List of numpy.ndarray of output basis vectors from SVD, one array per channel.
- **stachans** (*list*) – List of station.channel input
- **event_list** (*list*) – List of events for which you have data, such that event_list[i] corresponds to stachans[i], U[i] etc. and event_list[i][j] corresponds to event j in U[i]. These are a series of indexes that map the basis vectors to their relative events and channels - if you have every channel for every event generating these is trivial (see example).
- **n_svs** (*int*) – Number of singular values to use, defaults to 4.

Returns M, array of relative moments

Return type numpy.ndarray

Returns events_out, list of events that relate to M (in order), does not include the magnitude information in the events, see note.

Return type obspy.core.event.event.Event

Note: M is an array of relative moments (or amplitudes), these cannot be directly compared to true moments without calibration.

Note: When comparing this method with the method used for creation of subspace detectors (Harris 2006) it is important to note that the input *design set* matrix in Harris contains waveforms as columns, whereas in Rubinstein & Ellsworth it contains waveforms as rows (i.e. the transpose of the Harris data matrix). The U and V matrices are therefore swapped between the two approaches. This is accounted for in EQcorrscan but may lead to confusion when reviewing the code. Here we use the Harris approach.

Example

```
>>> from eqcorrscan.utils.mag_calc import svd_moments
>>> from obspy import read
>>> import glob
>>> import os
>>> from eqcorrscan.utils.clustering import svd
>>> import numpy as np
>>> # Do the set-up
>>> testing_path = 'eqcorrscan/tests/test_data/similar_events_processed'
>>> stream_files = glob.glob(os.path.join(testing_path, '*'))
>>> stream_list = [read(stream_file) for stream_file in stream_files]
>>> event_list = []
>>> remove_list = [('WHAT2', 'SH1'), ('WV04', 'SHZ'), ('GCSZ', 'EHZ')]
>>> for i, stream in enumerate(stream_list):
...     st_list = []
...     for tr in stream:
...         if (tr.stats.station, tr.stats.channel) not in remove_list:
...             stream.remove(tr)
...             continue
...         st_list.append(i)
...     event_list.append(st_list) # doctest: +SKIP
>>> event_list = np.asarray(event_list).T.tolist()
>>> SVec, SVal, U, stachans = svd(stream_list=stream_list) # doctest: +SKIP
['GCSZ.EHZ', 'WV04.SHZ', 'WHAT2.SH1']
>>> M, events_out = svd_moments(u=U, s=SVal, v=SVec, stachans=stachans,
...                             event_list=event_list) # doctest: +SKIP
```

eqcorrscan.utils.mag_calc.relative_amplitude

eqcorrscan.utils.mag_calc.**relative_amplitude**(*st1*, *st2*, *event1*, *event2*, *noise_window=(-20*, *-1)*, *signal_window=(-0.5*, *20)*, *min_snr=1.5*, *use_s_picks=False*)

Compute the relative amplitudes between two streams.

Uses standard deviation of amplitudes within trace. Relative amplitudes are computed as:

$$\frac{std(tr2)}{std(tr1)}$$

where tr1 is a trace from st1 and tr2 is a matching (seed ids match) trace from st2. The standard deviation of the amplitudes is computed in the signal window given. If the ratio of amplitudes between the signal window and the noise window is below *min_snr* then no result is returned for that trace. The SNR here is defined as the ratio of RMS-amplitudes of signal and noise (equal to ratio of L2-norms of signal and noise, but normalized for signal length). The Windows are computed relative to the first pick for that station.

If one stream has insufficient data to estimate noise amplitude, the noise amplitude of the other will be used.

Parameters

- **st1** (*obspy.core.stream.Stream*) – Stream for event1
- **st2** (*obspy.core.stream.Stream*) – Stream for event2
- **event1** (*obspy.core.event.Event*) – Event with picks (nothing else is needed)
- **event2** (*obspy.core.event.Event*) – Event with picks (nothing else is needed)
- **noise_window** (*tuple of float*) – Start and end of noise window in seconds relative to pick
- **signal_window** (*tuple of float*) – Start and end of signal window in seconds relative to pick
- **min_snr** (*float*) – Minimum signal-to-noise ratio allowed to make a measurement
- **use_s_picks** (*bool*) – Whether to allow relative amplitude estimates to be made from S-picks. Note that noise and signal windows are relative to pick-times, so using an S-pick might result in a noise window including P-energy.

Return type dict, dict, dict

Returns Dictionary of relative amplitudes keyed by seed-id Dictionary of signal-to-noise ratios for st1 Dictionary of signal-to-noise ratios for st2

eqcorrscan.utils.mag_calc.relative_magnitude

eqcorrscan.utils.mag_calc.**relative_magnitude**(*st1*, *st2*, *event1*, *event2*, *noise_window=(-20*, *-1)*, *signal_window=(-0.5*, *20)*, *min_snr=5.0*, *min_cc=0.7*, *use_s_picks=False*, *correlations=None*, *shift=0.2*, *return_correlations=False*, *correct_mag_bias=True*)

Compute the relative magnitudes between two events.

See *eqcorrscan.utils.mag_calc.relative_amplitude()* for information on how relative amplitudes are calculated. To compute relative magnitudes from relative amplitudes this function can weight the amplitude ratios by the cross-correlation of the two events. The relation used is similar to Schaff and Richards (2014), equation 4 and is:

$$\Delta m = \log \frac{std(tr2)}{std(tr1)} + \log \frac{(1 + \frac{1}{snr_x^2})}{1 + \frac{1}{snr_y^2}} \times CC$$

If you decide to use this function you should definitely read the paper to understand what you can use this for and cite the paper!

Parameters

- **st1** (*obspy.core.stream.Stream*) – Stream for event1
- **st2** (*obspy.core.stream.Stream*) – Stream for event2
- **event1** (*obspy.core.event.Event*) – Event with picks (nothing else is needed)

- **event2** (*obspy.core.event.Event*) – Event with picks (nothing else is needed)
- **noise_window** (*tuple of float*) – Start and end of noise window in seconds relative to pick
- **signal_window** (*tuple of float*) – Start and end of signal window in seconds relative to pick
- **min_snr** (*float*) – Minimum signal-to-noise ratio allowed to make a measurement
- **min_cc** (*float*) – Minimum inter-event correlation (between -1 and 1) allowed to make a measurement.
- **use_s_picks** (*bool*) – Whether to allow relative amplitude estimates to be made from S-picks. Note that noise and signal windows are relative to pick-times, so using an S-pick might result in a noise window including P-energy.
- **correlations** (*dict*) – Pre-computed dictionary of correlations keyed by seed-id. If None (default) then correlations will be computed for the provided data in the *signal_window*.
- **shift** (*float*) – Shift length for correlations in seconds - maximum correlation within a window between +/- shift of the P-pick will be used to weight the magnitude.
- **return_correlations** (*bool*) – If true will also return maximum correlations as a dictionary.
- **correct_mag_bias** (*bool*) – Whether to correct for the magnitude-bias introduced by cc<1 and the presence of noise (i.e., SNR << ∞). Without bias-correction, the relative magnitudes are simple L2-norm-ratio relative magnitudes.

Return type dict

Returns Dictionary of relative magnitudes keyed by seed-id

Private Functions

Note that these functions are not designed for public use and may change at any point.

_sim_WA	Remove the instrument response from a trace and simulate a Wood-Anderson.
_pairwise	Wrapper on itertools for SVD_magnitude.
_max_p2t	Finds the maximum peak-to-trough amplitude and period.

eqcorrscan.utils.mag_calc._sim_WA

eqcorrscan.utils.mag_calc.**_sim_WA**(*trace*, *inventory*, *water_level*, *velocity=False*)

Remove the instrument response from a trace and simulate a Wood-Anderson.

Returns a de-meaned, de-trended, Wood Anderson simulated trace in its place.

Works in-place on data and will destroy your original data, copy the trace before giving it to this function!

Parameters

- **trace** (*obspy.core.trace.Trace*) – A standard obspy trace, generally should be given without pre-filtering, if given with pre-filtering for use with amplitude determination for magnitudes you will need to worry about how you cope with the response of this filter yourself.
- **inventory** (*obspy.core.inventory.Inventory*) – Inventory containing response information for the stations in st.

- **water_level** (*float*) – Water level for the simulation.
- **velocity** (*bool*) – Whether to return a velocity trace or not - velocity is non-standard for Wood-Anderson instruments, but institutes that use seiscomp3 or Antelope require picks in velocity.

Returns Trace of Wood-Anderson simulated data

Return type obspy.core.trace.Trace

eqcorrscan.utils.mag_calc._pairwise

eqcorrscan.utils.mag_calc.**_pairwise**(*iterable*)
Wrapper on itertools for SVD_magnitude.

eqcorrscan.utils.mag_calc._max_p2t

eqcorrscan.utils.mag_calc.**_max_p2t**(*data*, *delta*, *return_peak_trough=False*)
Finds the maximum peak-to-trough amplitude and period.

Originally designed to be used to calculate magnitudes (by taking half of the peak-to-trough amplitude as the peak amplitude).

Parameters

- **data** (*numpy.ndarray*) – waveform trace to find the peak-to-trough in.
- **delta** (*float*) – Sampling interval in seconds
- **return_peak_trough** (*bool*) – Optionally return the peak and trough

Returns tuple of (amplitude, period, time) with amplitude in the same scale as given in the input data, and period in seconds, and time in seconds from the start of the data window.

Return type tuple

picker

Functions to pick earthquakes detected by EQcorrscan.

Designed primarily locate stacks of detections to give family locations. Extensions may later be written, not tested for accuracy, just simple wrappers.

copyright EQcorrscan developers.

license GNU Lesser General Public License, Version 3 (https://www.gnu.org/copyleft/lesser.html)

Classes & Functions

cross_net	Generate picks using a simple envelope cross-correlation.
stalta_pick	Basic sta/lta picker, suggest using alternative in obspy.

eqcorrscan.utils.picker.cross_net

eqcorrscan.utils.picker.**cross_net**(*stream*, *env=False*, *master=False*)
Generate picks using a simple envelope cross-correlation.

Picks are made for each channel based on optimal moveout defined by maximum cross-correlation with master trace. Master trace will be the first trace in the stream if not set. Requires good inter-station coher-

ance.

Parameters

- **stream** (*obspy.core.stream.Stream*) – Stream to pick
- **env** (*bool*) – To compute cross-correlations on the envelope or not.
- **master** (*obspy.core.trace.Trace*) – Trace to use as master, if False, will use the first trace in stream.

Returns obspy.core.event.event.Event

Example

```
>>> from obspy import read
>>> from eqcorrscan.utils.picker import cross_net
>>> st = read()
>>> event = cross_net(st, env=True)
>>> print(event.creation_info.author)
EQcorrscan
```

Warning: This routine is not designed for accurate picking, rather it can be used for a first-pass at picks to obtain simple locations. Based on the waveform-envelope cross-correlation method.

eqcorrscan.utils.picker.stalta_pick

eqcorrscan.utils.picker.**stalta_pick**(*stream*, *stalen*, *ltalen*, *trig_on*, *trig_off*, *freqmin=False*, *freqmax=False*, *show=False*)

Basic sta/lta picker, suggest using alternative in obspy.

Simple sta/lta (short-term average/long-term average) picker, using obspy's obspy.signal.trigger.classic_sta_lta() routine to generate the characteristic function.

Currently very basic quick wrapper, there are many other (better) options in obspy in the obspy.signal.trigger module.

Parameters

- **stream** (*obspy.core.stream.Stream*) – The stream to pick on, can be any number of channels.
- **stalen** (*float*) – Length of the short-term average window in seconds.
- **ltalen** (*float*) – Length of the long-term average window in seconds.
- **trig_on** (*float*) – sta/lta ratio to trigger a detection/pick
- **trig_off** (*float*) – sta/lta ratio to turn the trigger off - no further picks will be made between exceeding trig_on until trig_off is reached.
- **freqmin** (*float*) – Low-cut frequency in Hz for bandpass filter
- **freqmax** (*float*) – High-cut frequency in Hz for bandpass filter
- **show** (*bool*) – Show picks on waveform.

Returns obspy.core.event.event.Event

Example

```
>>> from obspy import read
>>> from eqcorrscan.utils.picker import stalta_pick
>>> st = read()
>>> event = stalta_pick(st, stalen=0.2, ltalen=4, trig_on=10,
...                     trig_off=1, freqmin=3.0, freqmax=20.0)
>>> print(event.creation_info.author)
EQcorrscan
```

Warning: This function is not designed for accurate picking, rather it can give a first idea of whether picks may be possible. Proceed with caution.

plotting

Utility code for most of the plots used as part of the EQcorrscan package.

copyright EQcorrscan developers.

license GNU Lesser General Public License, Version 3 (https://www.gnu.org/copyleft/lesser.html)

All plotting functions accept the *save*, *savefile*, *title* and *show* keyword arguments.

chunk_data	Downsample data for plotting.
cumulative_detections	Plot cumulative detections or detection rate in time.
detection_multiplot	Plot a stream of data with a template on top of it at detection times.
freq_mag	Plot a frequency-magnitude histogram and cumulative density plot.
interev_mag	Plot inter-event times against magnitude.
multi_event_singlechan	Plot data from a single channel for multiple events.
obspy_3d_plot	Plot obspy Inventory and obspy Catalog classes in three dimensions.
peaks_plot	Plot peaks to check that the peak finding routine is running correctly.
plot_repicked	Plot a template over a detected stream, with picks corrected by lag-calc.
plot_synth_real	Plot multiple channels of data for real data and synthetic.
pretty_template_plot	Plot of a single template, possibly within background data.
spec_trace	Plots seismic data with spectrogram behind.
subspace_detector_plot	Plotting for the subspace detector class.
svd_plot	Plot singular vectors from the *eqcorrscan.utils.clustering* routines.
threeD_gridplot	Plot in a series of grid points in 3D.
threeD_seismplot	Plot seismicity and stations in a 3D, movable, zoomable space.
triple_plot	Plot a seismogram, correlogram and histogram.
xcorr_plot	Plot a template overlying an image aligned by correlation.
noise_plot	Plot signal and noise fourier transforms and the difference.

eqcorrscan.utils.plotting.chunk_data

eqcorrscan.utils.plotting.**chunk_data**(*tr*, *samp_rate*, *state='mean'*)

Downsample data for plotting.

Computes the maximum of data within chunks, useful for plotting waveforms or cccsums, large datasets that would otherwise exceed the complexity allowed, and overflow.

Parameters

- **tr** (*obspy.core.trace.Trace*) – Trace to be chunked
- **samp_rate** (*float*) – Desired sampling rate in Hz
- **state** (*str*) – Either 'Min', 'Max', 'Mean' or 'Maxabs' to return one of these for the chunks. Maxabs will return the largest (positive or negative) for that chunk.

Returns obspy.core.trace.Trace

eqcorrscan.utils.plotting.cumulative_detections

eqcorrscan.utils.plotting.**cumulative_detections**(*dates=None*, *template_names=None*, *detections=None*, *plot_grouped=False*, *group_name=None*, *rate=False*, *binsize=None*, *plot_legend=True*, *ax=None*, ***kwargs*)

Plot cumulative detections or detection rate in time.

Simple plotting function to take a list of either datetime objects or eqcorrscan.core.match_filter.Detection objects and plot a cumulative detections list. Can take dates as a list of lists and will plot each list separately, e.g. if you have dates from more than one template it will overlay them in different colours.

Parameters

- **dates** (*list*) – Must be a list of lists of datetime.datetime objects
- **template_names** (*list*) – List of the template names in order of the dates
- **detections** (*list*) – List of eqcorrscan.core.match_filter.Detection
- **plot_grouped** (*bool*) – Plot detections for each template individually, or group them all together - set to False (plot template detections individually) by default.
- **group_name** (*str*) – Name to put in legend for the group, only used if *plot_grouped=True*
- **rate** (*bool*) – Whether or not to plot the rate of detection per day. Only works for plot_grouped=True
- **binsize** (*int*) – Bin size for rate plotting in seconds.
- **plot_legend** (*bool*) – Specify whether to plot legend of template names. Defaults to True.
- **ax** (*matplotlib.pyplot.Axis*) – Axis to plot into, if you want to re-use a figure.
- **title** (*str*) – Title of figure

- **show** (*bool*) – Whether to show the figure or not (defaults to True)
- **save** (*bool*) – Whether to save the figure or not (defaults to False)
- **savefile** (*str*) – Filename to save figure to, if *save==True* (defaults to "EQcorrscan_figure.png")
- **return_figure** (*bool*) – Whether to return the figure or not (defaults to True), if False then the figure will be cleared and closed.
- **size** (*tuple of float*) – Figure size as (width, height) in inches. Defaults to (10.5, 7.5)

Returns matplotlib.figure.Figure

Note: Can either take lists of eqcorrscan.core.match_filter.Detection objects directly, or two lists of dates and template names - either/or, not both.

Example

```
>>> import datetime as dt
>>> import numpy as np
>>> from eqcorrscan.utils.plotting import cumulative_detections
>>> dates = []
>>> for i in range(3):
...     dates.append([dt.datetime(2012, 3, 26) + dt.timedelta(n)
...                   for n in np.random.randn(100)])
>>> cumulative_detections(dates, ['a', 'b', 'c'],
...                       show=True) # doctest: +SKIP
```

Example 2: Rate plotting

```
>>> import datetime as dt
>>> import numpy as np
>>> from eqcorrscan.utils.plotting import cumulative_detections
>>> dates = []
>>> for i in range(3):
...     dates.append([dt.datetime(2012, 3, 26) + dt.timedelta(n)
...                   for n in np.random.randn(100)])
>>> cumulative_detections(dates, ['a', 'b', 'c'], plot_grouped=True,
...                       rate=True, show=True) # doctest: +SKIP
```

eqcorrscan.utils.plotting.detection_multiplot

eqcorrscan.utils.plotting.**detection_multiplot**(*stream*, *template*, *times*, *streamcolour='k'*, *templatecolour='r'*, ***kwargs*)

Plot a stream of data with a template on top of it at detection times.

Parameters

- **stream** (*obspy.core.stream.Stream*) – Stream of data to be plotted as the background.
- **template** (*obspy.core.stream.Stream*) – Template to be plotted on top of the base stream.
- **times** (*list*) – list of detection times, one for each event

- **streamcolour** (*str*) – String of matplotlib colour types for the stream
- **templatecolour** (*str*) – Colour to plot the template in.
- **title** (*str*) – Title of figure
- **show** (*bool*) – Whether to show the figure or not (defaults to True)
- **save** (*bool*) – Whether to save the figure or not (defaults to False)
- **savefile** (*str*) – Filename to save figure to, if *save==True* (defaults to "EQcorrscan_figure.png")
- **return_figure** (*bool*) – Whether to return the figure or not (defaults to True), if False then the figure will be cleared and closed.
- **size** (*tuple of float*) – Figure size as (width, height) in inches. Defaults to (10.5, 7.5)

Returns matplotlib.figure.Figure

Example

```
>>> from obspy import read, read_events
>>> import os
>>> from eqcorrscan.core import template_gen
>>> from eqcorrscan.utils.plotting import detection_multiplot
>>> # Get the path to the test data
>>> import eqcorrscan
>>> import os
>>> TEST_PATH = os.path.dirname(eqcorrscan.__file__) + '/tests/test_
↪data'
>>>
>>> test_file = os.path.join(TEST_PATH, 'REA',
...                          'TEST_', '01-0411-15L.S201309')
>>> test_wavefile = os.path.join(
...     TEST_PATH, 'WAV', 'TEST_', '2013-09-01-0410-35.DFDPC_024_00')
>>> event = read_events(test_file)[0]
>>> st = read(test_wavefile)
>>> st = st.filter('bandpass', freqmin=2.0, freqmax=15.0)
>>> for tr in st:
...     tr = tr.trim(tr.stats.starttime + 30, tr.stats.endtime - 30)
...     # Hack around seisan 2-letter channel naming
...     tr.stats.channel = tr.stats.channel[0] + tr.stats.channel[-1]
>>> template = template_gen._template_gen(event.picks, st, 2)
>>> times = [min([pk.time -0.05 for pk in event.picks])]
>>> detection_multiplot(stream=st, template=template,
...                     times=times) # doctest: +SKIP
```

eqcorrscan.utils.plotting.freq_mag

eqcorrscan.utils.plotting.**freq_mag**(*magnitudes*, *completeness*, *max_mag*, *binsize=0.2*, ***kwargs*)

Plot a frequency-magnitude histogram and cumulative density plot.

Currently this will compute a b-value, for a given completeness. B-value is computed by linear fitting to section of curve between completeness and max_mag.

Parameters

- **magnitudes** (*list*) – list of float of magnitudes
- **completeness** (*float*) – Level to compute the b-value above

- **max_mag** (*float*) – Maximum magnitude to try and fit a b-value to
- **binsize** (*float*) – Width of histogram bins, defaults to 0.2
- **title** (*str*) – Title of figure
- **show** (*bool*) – Whether to show the figure or not (defaults to True)
- **save** (*bool*) – Whether to save the figure or not (defaults to False)
- **savefile** (*str*) – Filename to save figure to, if *save==True* (defaults to "EQcorrscan_figure.png")
- **return_figure** (*bool*) – Whether to return the figure or not (defaults to True), if False then the figure will be cleared and closed.
- **size** (*tuple of float*) – Figure size as (width, height) in inches. Defaults to (10.5, 7.5)

Returns matplotlib.figure.Figure

Note: See *eqcorrscan.utils.mag_calc.calc_b_value()* for a least-squares method of estimating completeness and b-value. For estimating maximum curvature see *eqcorrscan.utils.mag_calc.calc_max_curv()*.

Example

```
>>> from obspy.clients.fdsn import Client
>>> from obspy import UTCDateTime
>>> from eqcorrscan.utils.plotting import freq_mag
>>> client = Client('IRIS')
>>> t1 = UTCDateTime('2012-03-26T00:00:00')
>>> t2 = t1 + (3 * 86400)
>>> catalog = client.get_events(starttime=t1, endtime=t2,␣
↪minmagnitude=3)
>>> magnitudes = [event.preferred_magnitude().mag for event in␣
↪catalog]
>>> freq_mag(magnitudes, completeness=4, max_mag=7) # doctest: +SKIP
```

eqcorrscan.utils.plotting.interev_mag

eqcorrscan.utils.plotting.**interev_mag**(*times*, *mags*, ***kwargs*)

Plot inter-event times against magnitude.

Parameters

- **times** (*list*) – list of the detection times, must be sorted the same as mags
- **mags** (*list*) – list of magnitudes
- **title** (*str*) – Title of figure
- **show** (*bool*) – Whether to show the figure or not (defaults to True)
- **save** (*bool*) – Whether to save the figure or not (defaults to False)
- **savefile** (*str*) – Filename to save figure to, if *save==True* (defaults to "EQcorrscan_figure.png")
- **return_figure** (*bool*) – Whether to return the figure or not (defaults to True), if False then the figure will be cleared and closed.
- **size** (*tuple of float*) – Figure size as (width, height) in inches. Defaults to (10.5, 7.5)

Returns matplotlib.figure.Figure

Example

```
>>> from obspy.clients.fdsn import Client
>>> from obspy import UTCDateTime
>>> from eqcorrscan.utils.plotting import interev_mag
>>> client = Client('IRIS')
>>> t1 = UTCDateTime('2012-03-26T00:00:00')
>>> t2 = t1 + (3 * 86400)
>>> catalog = client.get_events(starttime=t1, endtime=t2,␣
↪minmagnitude=3)
>>> magnitudes = [event.preferred_magnitude().mag for event in␣
↪catalog]
>>> times = [event.preferred_origin().time for event in catalog]
>>> interev_mag(times, magnitudes) # doctest: +SKIP
```

eqcorrscan.utils.plotting.multi_event_singlechan

eqcorrscan.utils.plotting.**multi_event_singlechan**(*streams*, *catalog*, *station*, *channel*, *clip=10.0*, *pre_pick=2.0*, *freqmin=False*, *freqmax=False*, *realign=False*, *cut=(-3.0, 5.0)*, *PWS=False*, ***kwargs*)

Plot data from a single channel for multiple events.

Data will be aligned by their pick-time given in the appropriate picks. Requires an individual stream for each event you want to plot, events are stored in the obspy.core.event.Catalog object, and there must be picks present for the streams you wish to plot. Events will be aligned if *realign=True*, in this case the traces will be aligned using the window defined by *cut*.

Parameters

- **streams** (*list*) – List of the obspy.core.stream.Stream objects to use, can contain more traces than you plan on plotting (e.g. from more channels) - must be in the same order as events in catalog.
- **catalog** (*obspy.core.event.Catalog*) – Catalog of events, one for each stream.
- **station** (*str*) – Station to plot.
- **channel** (*str*) – Channel to plot.
- **clip** (*float*) – Length in seconds to plot, defaults to 10.0
- **pre_pick** (*float*) – Length in seconds to extract and plot before the pick, defaults to 2.0
- **freqmin** (*float*) – Low cut for bandpass in Hz
- **freqmax** (*float*) – High cut for bandpass in Hz
- **realign** (*bool*) – To compute best alignment based on correlation with the stack or not.

- **cut** (*tuple*) – tuple of start and end times for cut in seconds from the pick, used for alignment. Will only use this window to align the traces.
- **PWS** (*bool*) – compute Phase Weighted Stack, if False, will compute linear stack for alignment.
- **title** (*str*) – Title of figure
- **show** (*bool*) – Whether to show the figure or not (defaults to True)
- **save** (*bool*) – Whether to save the figure or not (defaults to False)
- **savefile** (*str*) – Filename to save figure to, if *save==True* (defaults to "EQcorrscan_figure.png")
- **return_figure** (*bool*) – Whether to return the figure or not (defaults to True), if False then the figure will be cleared and closed.
- **size** (*tuple of float*) – Figure size as (width, height) in inches. Defaults to (10.5, 7.5)

Returns Aligned and cut `obspy.core.trace.Trace`

Return type list

Returns New picks in based on alignment (if alignment is performed, if not will return the same as input)

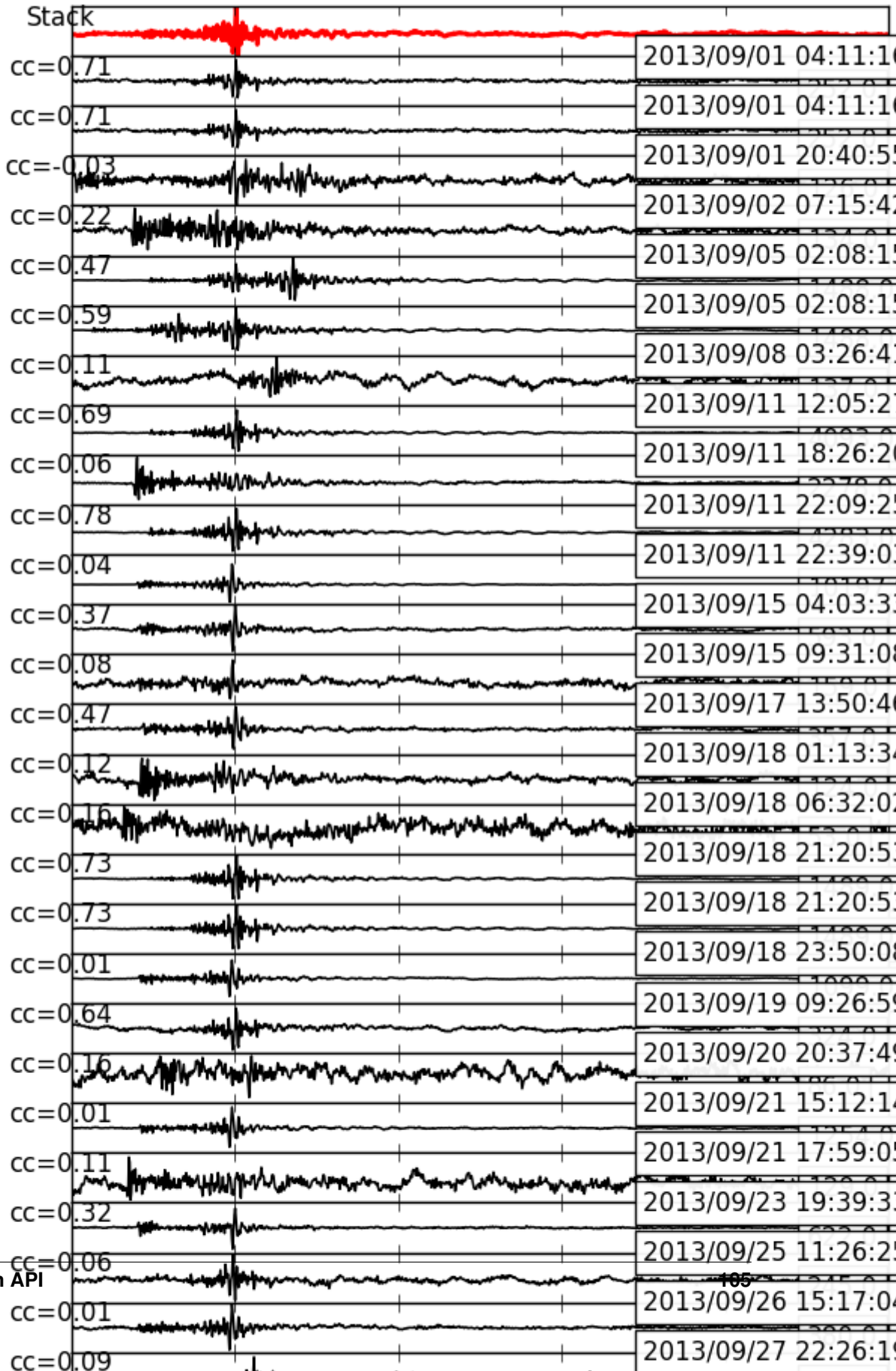
Return type `obspy.core.event.Catalog`

Returns Figure object for further editing

Return type `matplotlib.figure.Figure`

Example

```
>>> from obspy import read, Catalog, read_events
>>> from obspy.io.nordic.core import readwavename
>>> from eqcorrscan.utils.plotting import multi_event_singlechan
>>> import glob
>>> sfiles = glob.glob('eqcorrscan/tests/test_data/REA/TEST_/*.S??????
↪')
>>> catalog = Catalog()
>>> streams = []
>>> for sfile in sfiles:
...     catalog += read_events(sfile)
...     wavfile = readwavename(sfile)[0]
...     stream_path = 'eqcorrscan/tests/test_data/WAV/TEST_/' +␣
↪wavfile
...     stream = read(stream_path)
...     # Annoying coping with seisan 2 letter channels
...     for tr in stream:
...         tr.stats.channel = tr.stats.channel[0] + tr.stats.
↪channel[-1]
...     streams.append(stream)
>>> multi_event_singlechan(streams=streams, catalog=catalog,
...                        station='GCSZ', channel='EZ') # doctest:␣
↪+SKIP
```


Stack
cc=0.71
cc=0.71
cc=-0.03
cc=0.22
cc=0.47
cc=0.59
cc=0.11
cc=0.69
cc=0.06
cc=0.78
cc=0.04
cc=0.37
cc=0.08
cc=0.47
cc=0.12
cc=0.16
cc=0.73
cc=0.73
cc=0.01
cc=0.64
cc=0.16
cc=0.01
cc=0.11
cc=0.32
cc=0.06
cc=0.01
cc=0.09
2013/09/01 04:11:1
2013/09/01 04:11:1
2013/09/01 20:40:5
2013/09/02 07:15:4
2013/09/05 02:08:1
2013/09/05 02:08:1
2013/09/08 03:26:4
2013/09/11 12:05:2
2013/09/11 18:26:2
2013/09/11 22:09:2
2013/09/11 22:39:0
2013/09/15 04:03:3
2013/09/15 09:31:0
2013/09/17 13:50:4
2013/09/18 01:13:3
2013/09/18 06:32:0
2013/09/18 21:20:5
2013/09/18 21:20:5
2013/09/18 23:50:0
2013/09/19 09:26:5
2013/09/20 20:37:4
2013/09/21 15:12:1
2013/09/21 17:59:0
2013/09/23 19:39:3
2013/09/25 11:26:2
2013/09/26 15:17:0
2013/09/27 22:26:1

eqcorrscan.utils.plotting.obspy_3d_plot

eqcorrscan.utils.plotting.**obspy_3d_plot**(*inventory*, *catalog*, ***kwargs*)

Plot obspy Inventory and obspy Catalog classes in three dimensions.

Parameters

- **inventory** (*obspy.core.inventory.inventory.Inventory*) – Obspy inventory class containing station metadata
- **catalog** (*obspy.core.event.catalog.Catalog*) – Obspy catalog class containing event metadata
- **title** (*str*) – Title of figure
- **show** (*bool*) – Whether to show the figure or not (defaults to True)
- **save** (*bool*) – Whether to save the figure or not (defaults to False)
- **savefile** (*str*) – Filename to save figure to, if *save==True* (defaults to "EQcorrscan_figure.png")
- **return_figure** (*bool*) – Whether to return the figure or not (defaults to True), if False then the figure will be cleared and closed.
- **size** (*tuple of float*) – Figure size as (width, height) in inches. Defaults to (10.5, 7.5)

Returns matplotlib.figure.Figure

Example:

```
>>> from obspy.clients.fdsn import Client
>>> from obspy import UTCDateTime
>>> from eqcorrscan.utils.plotting import obspy_3d_plot
>>> client = Client('IRIS')
>>> t1 = UTCDateTime(2012, 3, 26)
>>> t2 = t1 + 86400
>>> catalog = client.get_events(starttime=t1, endtime=t2, latitude=-
↪43,
...                             longitude=170, maxradius=5)
>>> inventory = client.get_stations(starttime=t1, endtime=t2,␣
↪latitude=-43,
...                                 longitude=170, maxradius=10)
>>> obspy_3d_plot(inventory=inventory, catalog=catalog) # doctest:␣
↪+SKIP
```

eqcorrscan.utils.plotting.peaks_plot

eqcorrscan.utils.plotting.**peaks_plot**(*data*, *starttime*, *samp_rate*, *peaks=None*, ***kwargs*)

Plot peaks to check that the peak finding routine is running correctly.

Used in debugging for the EQcorrscan module.

Parameters

- **data** (*numpy.array*) – Numpy array of the data within which peaks have been found
- **starttime** (*obspy.core.utcdatetime.UTCDateTime*) – Start time for the data
- **samp_rate** (*float*) – Sampling rate of data in Hz

- **peaks** (*list*) – List of tuples of peak locations and amplitudes (loc, amp)
- **title** (*str*) – Title of figure
- **show** (*bool*) – Whether to show the figure or not (defaults to True)
- **save** (*bool*) – Whether to save the figure or not (defaults to False)
- **savefile** (*str*) – Filename to save figure to, if *save==True* (defaults to "EQcorrscan_figure.png")
- **return_figure** (*bool*) – Whether to return the figure or not (defaults to True), if False then the figure will be cleared and closed.
- **size** (*tuple of float*) – Figure size as (width, height) in inches. Defaults to (10.5, 7.5)

Returns matplotlib.figure.Figure

Example

```
>>> import numpy as np
>>> from eqcorrscan.utils import findpeaks
>>> from eqcorrscan.utils.plotting import peaks_plot
>>> from obspy import UTCDateTime
>>> data = np.random.randn(200)
>>> data[30] = 100
>>> data[60] = 40
>>> threshold = 10
>>> peaks = findpeaks.find_peaks2_short(data, threshold, 3)
>>> peaks_plot(data=data, starttime=UTCDateTime("2008001"),
...            samp_rate=10, peaks=peaks)  # doctest: +SKIP
```

eqcorrscan.utils.plotting.plot_repicked

eqcorrscan.utils.plotting.**plot_repicked**(*template*, *picks*, *det_stream*, ***kwargs*)

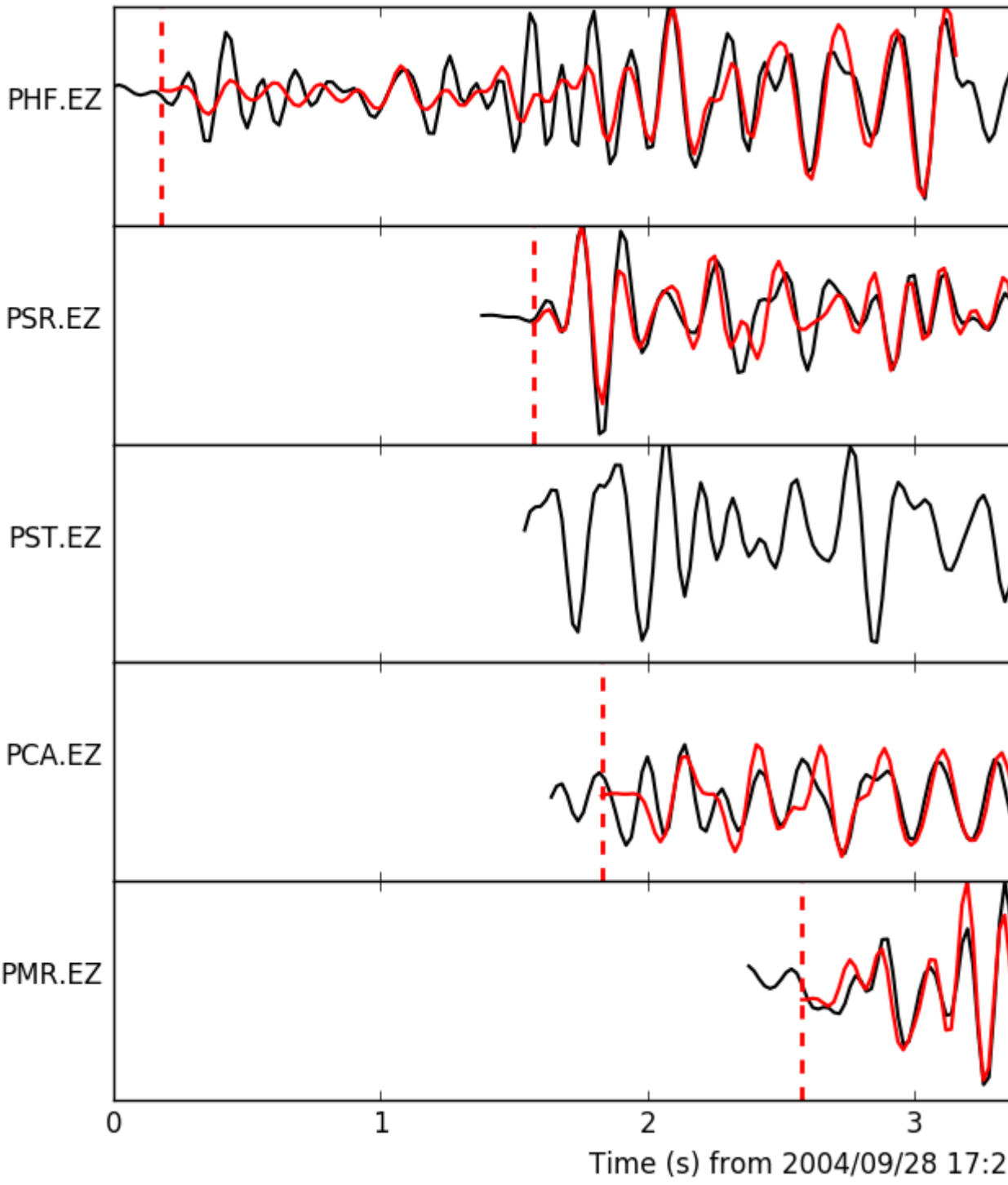
Plot a template over a detected stream, with picks corrected by lag-calc.

Parameters

- **template** (*obspy.core.stream.Stream*) – Template used to make the detection, will be aligned according to picks.
- **picks** (*list*) – list of corrected picks, each pick must be an obspy.core.event.origin.Pick object.
- **det_stream** (*obspy.core.stream.Stream*) – Stream to plot in the background, should be the detection, data should encompass the time the picks are made.
- **title** (*str*) – Title of figure
- **show** (*bool*) – Whether to show the figure or not (defaults to True)
- **save** (*bool*) – Whether to save the figure or not (defaults to False)
- **savefile** (*str*) – Filename to save figure to, if *save==True* (defaults to "EQcorrscan_figure.png")
- **return_figure** (*bool*) – Whether to return the figure or not (defaults to True), if False then the figure will be cleared and closed.
- **size** (*tuple of float*) – Figure size as (width, height) in inches. Defaults to (10.5, 7.5)

Returns Figure handle which can be edited.

Return type matplotlib.figure.Figure



eqcorrscan.utils.plotting.plot_synth_real

eqcorrscan.utils.plotting.**plot_synth_real**(*real_template*, *synthetic*, *channels=False*, ***kwargs*)

Plot multiple channels of data for real data and synthetic.

Parameters

- **real_template** (*obspy.core.stream.Stream*) – Stream of the real template
- **synthetic** (*obspy.core.stream.Stream*) – Stream of synthetic template
- **channels** (*list*) – List of tuples of (station, channel) to plot, default is False, which plots all.
- **title** (*str*) – Title of figure
- **show** (*bool*) – Whether to show the figure or not (defaults to True)
- **save** (*bool*) – Whether to save the figure or not (defaults to False)
- **savefile** (*str*) – Filename to save figure to, if *save==True* (defaults to "EQcorrscan_figure.png")
- **return_figure** (*bool*) – Whether to return the figure or not (defaults to True), if False then the figure will be cleared and closed.
- **size** (*tuple of float*) – Figure size as (width, height) in inches. Defaults to (10.5, 7.5)

Returns matplotlib.figure.Figure

```
>>> from obspy import read, Stream, Trace
>>> from eqcorrscan.utils.synth_seis import seis_sim
>>> from eqcorrscan.utils.plotting import plot_synth_real
>>> real = read()
>>> synth = Stream(Trace(seis_sim(sp=100, flength=200)))
>>> synth[0].stats.station = 'RJOB'
>>> synth[0].stats.channel = 'EHZ'
>>> synth[0].stats.sampling_rate = 100
>>> synth = synth.filter('bandpass', freqmin=2, freqmax=8)
>>> real = real.select(
...     station='RJOB', channel='EHZ').detrend('simple').filter(
...         'bandpass', freqmin=2, freqmax=8)
>>> real = real.trim(
...     starttime=real[0].stats.starttime + 4.9,
...     endtime=real[0].stats.starttime + 6.9).detrend('simple')
>>> plot_synth_real(real_template=real, synthetic=synth,
...                 size=(7, 4)) # doctest: +SKIP
```

eqcorrscan.utils.plotting.pretty_template_plot

eqcorrscan.utils.plotting.**pretty_template_plot**(*template*, *background=False*, *event=False*, *sort_by='distance'*, ***kwargs*)

Plot of a single template, possibly within background data.

Parameters

- **template** (*obspy.core.stream.Stream*) – Template stream to plot
- **background** (*obspy.core.stream.stream*) – Stream to plot the template within.
- **event** (*obspy.core.event.event.Event*) – Event object containing picks, and optionally information on the origin and arrivals. When supplied,

function tries to extract hypocentral distance from origin/arrivals, to sort the template traces by hypocentral distance.

- **sort_by** (*string*) – "distance" (default) or "pick_time" (not relevant if no event supplied)
- **title** (*str*) – Title of figure
- **show** (*bool*) – Whether to show the figure or not (defaults to True)
- **save** (*bool*) – Whether to save the figure or not (defaults to False)
- **savefile** (*str*) – Filename to save figure to, if *save==True* (defaults to "EQcorrscan_figure.png")
- **return_figure** (*bool*) – Whether to return the figure or not (defaults to True), if False then the figure will be cleared and closed.
- **size** (*tuple of float*) – Figure size as (width, height) in inches. Defaults to (10.5, 7.5)

Returns matplotlib.figure.Figure

Example

```
>>> from obspy import read, read_events
>>> import os
>>> from eqcorrscan.core import template_gen
>>> from eqcorrscan.utils.plotting import pretty_template_plot
>>> # Get the path to the test data
>>> import eqcorrscan
>>> import os
>>> TEST_PATH = os.path.dirname(eqcorrscan.__file__) + '/tests/test_
↪data'
>>>
>>> test_file = os.path.join(TEST_PATH, 'REA', 'TEST_',
...                          '01-0411-15L.S201309')
>>> test_wavefile = os.path.join(
...     TEST_PATH, 'WAV', 'TEST_', '2013-09-01-0410-35.DFDPC_024_00')
>>> event = read_events(test_file)[0]
>>> st = read(test_wavefile)
>>> st = st.filter('bandpass', freqmin=2.0, freqmax=15.0)
>>> for tr in st:
...     tr = tr.trim(tr.stats.starttime + 30, tr.stats.endtime - 30)
...     # Hack around seisan 2-letter channel naming
...     tr.stats.channel = tr.stats.channel[0] + tr.stats.channel[-1]
>>> template = template_gen._template_gen(event.picks, st, 2)
>>> pretty_template_plot(template, background=st, # doctest +SKIP
...                      event=event) # doctest: +SKIP
```

eqcorrscan.utils.plotting.spec_trace

eqcorrscan.utils.plotting.**spec_trace**(*traces*, *cmap=None*, *wlen=0.4*, *log=False*, *trc='k'*, *tralpha=0.9*, *fig=None*, ***kwargs*)

Plots seismic data with spectrogram behind.

Takes a stream or list of traces and plots the trace with the spectra beneath it.

Parameters

- **traces** (*list*) – Traces to be plotted, can be a single obspy.core.stream.Stream, or a list of obspy.core.trace.Trace.

- **cmap** (*str*) – Matplotlib colormap.
- **wlen** (*float*) – Window length for fft in seconds
- **log** (*bool*) – Use a log frequency scale
- **trc** (*str*) – Color for the trace.
- **tralpha** (*float*) – Opacity level for the seismogram, from transparent (0.0) to opaque (1.0).
- **fig** – Figure to plot onto, defaults to self generating.
- **title** (*str*) – Title of figure
- **show** (*bool*) – Whether to show the figure or not (defaults to True)
- **save** (*bool*) – Whether to save the figure or not (defaults to False)
- **savefile** (*str*) – Filename to save figure to, if *save==True* (defaults to "EQcorrscan_figure.png")
- **return_figure** (*bool*) – Whether to return the figure or not (defaults to True), if False then the figure will be cleared and closed.
- **size** (*tuple of float*) – Figure size as (width, height) in inches. Defaults to (10.5, 7.5)

Returns matplotlib.figure.Figure

Example

```
>>> from obspy import read
>>> from eqcorrscan.utils.plotting import spec_trace
>>> st = read()
>>> spec_trace(st, trc='white') # doctest: +SKIP
```

eqcorrscan.utils.plotting.subspace_detector_plot

eqcorrscan.utils.plotting.**subspace_detector_plot**(*detector*, *stachans*, ***kwargs*)

Plotting for the subspace detector class.

Plot the output basis vectors for the detector at the given dimension.

Corresponds to the first n horizontal vectors of the V matrix.

Parameters

- **stachans** (*list*) – list of tuples of station, channel pairs to plot.
- **stachans** – List of tuples of (station, channel) to use. Can set to 'all' to use all the station-channel pairs available. If detector is multiplexed, will just plot that.
- **title** (*str*) – Title of figure
- **show** (*bool*) – Whether to show the figure or not (defaults to True)
- **save** (*bool*) – Whether to save the figure or not (defaults to False)
- **savefile** (*str*) – Filename to save figure to, if *save==True* (defaults to "EQcorrscan_figure.png")
- **return_figure** (*bool*) – Whether to return the figure or not (defaults to True), if False then the figure will be cleared and closed.

- **size** (*tuple of float*) – Figure size as (width, height) in inches. Defaults to (10.5, 7.5)

Returns Figure

Return type matplotlib.pyplot.Figure

Example

```
>>> from eqcorrscan.core import subspace
>>> import os
>>> detector = subspace.Detector()
>>> detector.read(os.path.join(
...     os.path.abspath(os.path.dirname(__file__)),
...     '..', 'tests', 'test_data', 'subspace',
...     'stat_test_detector.h5'))
Detector: Tester
>>> subspace_detector_plot(detector=detector, stachans='all',␣
↪size=(10, 7),
...                         show=True) # doctest: +SKIP
```

eqcorrscan.utils.plotting.svd_plot

eqcorrscan.utils.plotting.**svd_plot**(*svstreams*, *svalues*, *stachans*, ***kwargs*)

Plot singular vectors from the *eqcorrscan.utils.clustering* routines.

One plot for each channel.

Parameters

- **svstreams** (*list*) – See *eqcorrscan.utils.clustering.svd_to_stream()* - these should be ordered by power, e.g. first singular vector in the first stream.
- **svalues** (*list*) – List of floats of the singular values corresponding to the SVStreams
- **stachans** (*list*) – List of station.channel
- **title** (*str*) – Title of figure
- **show** (*bool*) – Whether to show the figure or not (defaults to True)
- **save** (*bool*) – Whether to save the figure or not (defaults to False)
- **savefile** (*str*) – Filename to save figure to, if *save==True* (defaults to "EQcorrscan_figure.png")
- **return_figure** (*bool*) – Whether to return the figure or not (defaults to True), if False then the figure will be cleared and closed.
- **size** (*tuple of float*) – Figure size as (width, height) in inches. Defaults to (10.5, 7.5)

Returns matplotlib.figure.Figure

Example

```
>>> from obspy import read
>>> import glob
>>> from eqcorrscan.utils.plotting import svd_plot
>>> from eqcorrscan.utils.clustering import svd, svd_to_stream
```

(continues on next page)

(continued from previous page)

```
>>> wavefiles = glob.glob('eqcorrscan/tests/test_data/WAV/TEST_/2013-*
↪')
>>> streams = [read(w) for w in wavefiles[1:10]]
>>> stream_list = []
>>> for st in streams:
...     tr = st.select(station='GCSZ', channel='EHZ')
...     tr = tr.detrend('simple').resample(100).filter(
...         'bandpass', freqmin=2, freqmax=8)
...     stream_list.append(tr)
>>> uvec, sval, svec, stachans = svd(stream_list=stream_list)
>>> svstreams = svd_to_stream(uvectors=uvec, stachans=stachans, k=3,
...                           sampling_rate=100)
>>> svd_plot(svstreams=svstreams, svalues=sval,
...          stachans=stachans) # doctest: +SKIP
```

eqcorrscan.utils.plotting.threeD_gridplot

eqcorrscan.utils.plotting.**threeD_gridplot**(*nodes*, ***kwargs*)

Plot in a series of grid points in 3D.

Parameters

- **nodes** (*list*) – List of tuples of the form (lat, long, depth)
- **title** (*str*) – Title of figure
- **show** (*bool*) – Whether to show the figure or not (defaults to True)
- **save** (*bool*) – Whether to save the figure or not (defaults to False)
- **savefile** (*str*) – Filename to save figure to, if *save==True* (defaults to "EQcorrscan_figure.png")
- **return_figure** (*bool*) – Whether to return the figure or not (defaults to True), if False then the figure will be cleared and closed.
- **size** (*tuple of float*) – Figure size as (width, height) in inches. Defaults to (10.5, 7.5)

Returns matplotlib.figure.Figure

Example

```
>>> from eqcorrscan.utils.plotting import threeD_gridplot
>>> nodes = [(-43.5, 170.4, 4), (-43.3, 170.8, 12), (-43.4, 170.3, 8)]
>>> threeD_gridplot(nodes=nodes)  # doctest: +SKIP
```

eqcorrscan.utils.plotting.threeD_seismplot

eqcorrscan.utils.plotting.**threeD_seismplot**(*stations*, *nodes*, ***kwargs*)

Plot seismicity and stations in a 3D, movable, zoomable space.

Uses matplotlibs Axes3D package.

Parameters

- **stations** (*list*) – list of one tuple per station of (lat, long, elevation), with up positive.
- **nodes** (*list*) – list of one tuple per event of (lat, long, depth) with down positive.

- **title** (*str*) – Title of figure
- **show** (*bool*) – Whether to show the figure or not (defaults to True)
- **save** (*bool*) – Whether to save the figure or not (defaults to False)
- **savefile** (*str*) – Filename to save figure to, if *save==True* (defaults to "EQcorrscan_figure.png")
- **return_figure** (*bool*) – Whether to return the figure or not (defaults to True), if False then the figure will be cleared and closed.
- **size** (*tuple of float*) – Figure size as (width, height) in inches. Defaults to (10.5, 7.5)

Returns matplotlib.figure.Figure

Note: See *eqcorrscan.utils.plotting.obspy_3d_plot()* for example output.

---

eqcorrscan.utils.plotting.triple_plot

eqcorrscan.utils.plotting.**triple_plot**(*cccsum*, *cccsum_hist*, *trace*, *threshold*, ***kwargs*)

Plot a seismogram, correlogram and histogram.

Parameters

- **cccsum** (*numpy.ndarray*) – Array of the cross-channel cross-correlation sum
- **cccsum_hist** (*numpy.ndarray*) – cccsum for histogram plotting, can be the same as cccsum but included if cccsum is just an envelope.
- **trace** (*obspy.core.trace.Trace*) – A sample trace from the same time as cccsum
- **threshold** (*float*) – Detection threshold within cccsum
- **title** (*str*) – Title of figure
- **show** (*bool*) – Whether to show the figure or not (defaults to True)
- **save** (*bool*) – Whether to save the figure or not (defaults to False)
- **savefile** (*str*) – Filename to save figure to, if *save==True* (defaults to "EQcorrscan_figure.png")
- **return_figure** (*bool*) – Whether to return the figure or not (defaults to True), if False then the figure will be cleared and closed.
- **size** (*tuple of float*) – Figure size as (width, height) in inches. Defaults to (10.5, 7.5)

Returns matplotlib.figure.Figure

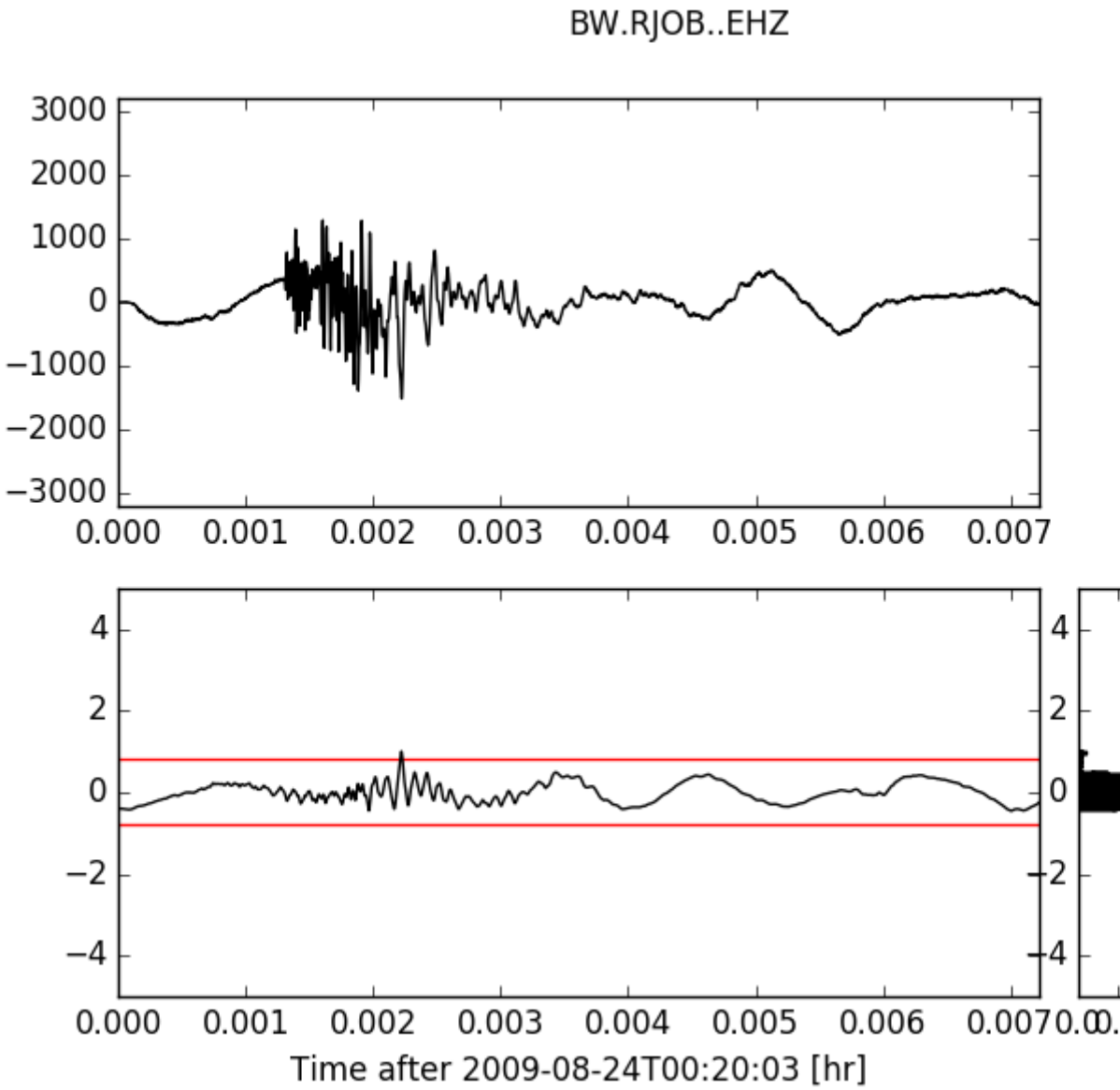
Example

```
>>> from obspy import read
>>> from eqcorrscan.core.match_filter import normxcorr2
>>> from eqcorrscan.utils.plotting import triple_plot
>>> st = read()
>>> template = st[0].copy().trim(st[0].stats.starttime + 8,
...                              st[0].stats.starttime + 12)
>>> tr = st[0]
```

(continues on next page)

(continued from previous page)

```
>>> ccc = normxcorr2(template=template.data, image=tr.data)
>>> tr.data = tr.data[0:len(ccc[0])]
>>> triple_plot(cccsum=ccc[0], cccsum_hist=ccc[0], trace=tr,
...             threshold=0.8) # doctest: +SKIP
```



eqcorrscan.utils.plotting.xcorr_plot

eqcorrscan.utils.plotting.**xcorr_plot**(*template*, *image*, *shift=None*, *cc=None*, *cc_vec=None*, ***kwargs*)

Plot a template overlying an image aligned by correlation.

Parameters

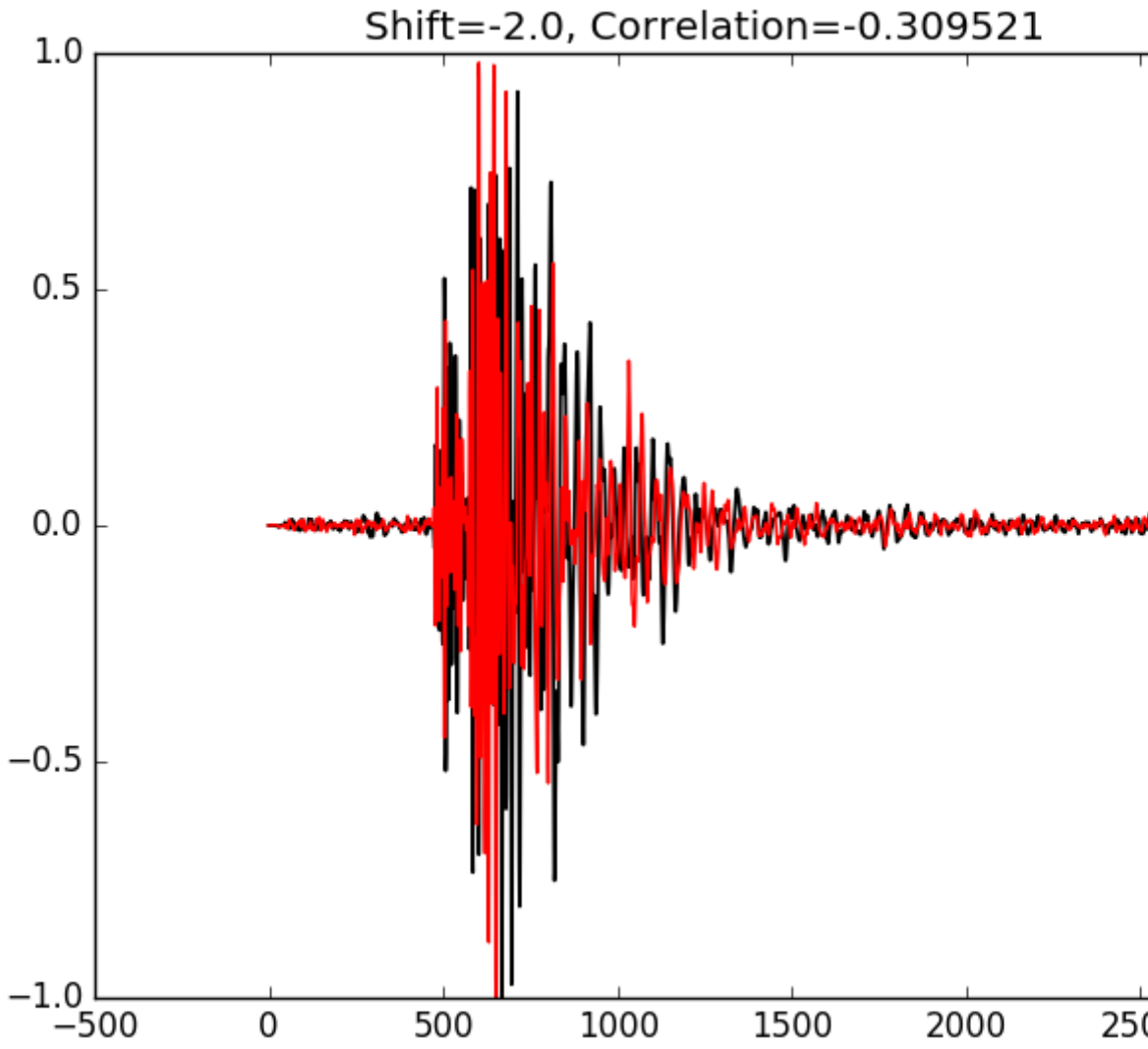
- **template** (*numpy.ndarray*) – Short template image
- **image** (*numpy.ndarray*) – Long master image
- **shift** (*int*) – Shift to apply to template relative to image, in samples
- **cc** (*float*) – Cross-correlation at shift

- **cc_vec** (*numpy.ndarray*) – Cross-correlation vector.
- **title** (*str*) – Title of figure
- **show** (*bool*) – Whether to show the figure or not (defaults to True)
- **save** (*bool*) – Whether to save the figure or not (defaults to False)
- **savefile** (*str*) – Filename to save figure to, if *save==True* (defaults to "EQcorrscan_figure.png")
- **return_figure** (*bool*) – Whether to return the figure or not (defaults to True), if False then the figure will be cleared and closed.
- **size** (*tuple of float*) – Figure size as (width, height) in inches. Defaults to (10.5, 7.5)

Returns matplotlib.figure.Figure

Example

```
>>> from obspy import read
>>> from eqcorrscan.utils.plotting import xcorr_plot
>>> from eqcorrscan.utils.stacking import align_traces
>>> st = read().detrend('simple').filter('bandpass', freqmin=2,␣
↪freqmax=15)
>>> shifts, ccs = align_traces([st[0], st[1]], 40)
>>> shift = shifts[1] * st[1].stats.sampling_rate
>>> cc = ccs[1]
>>> xcorr_plot(template=st[1].data, image=st[0].data, shift=shift,
...            cc=cc) # doctest: +SKIP
```



eqcorrscan.utils.plotting.noise_plot

eqcorrscan.utils.plotting.**noise_plot**(*signal*, *noise*, *normalise=False*, ***kwargs*)

Plot signal and noise fourier transforms and the difference.

Parameters

- **signal** (*obspy.core.stream.Stream*) – Stream of "signal" window
- **noise** (*obspy.core.stream.Stream*) – Stream of the "noise" window.
- **normalise** (*bool*) – Whether to normalise the data before plotting or not.
- **title** (*str*) – Title of figure
- **show** (*bool*) – Whether to show the figure or not (defaults to True)
- **save** (*bool*) – Whether to save the figure or not (defaults to False)
- **savefile** (*str*) – Filename to save figure to, if *save==True* (defaults to "EQcorrscan_figure.png")

- **return_figure** (*bool*) – Whether to return the figure or not (defaults to True), if False then the figure will be cleared and closed.
- **size** (*tuple of float*) – Figure size as (width, height) in inches. Defaults to (10.5, 7.5)

Returns *matplotlib.pyplot.Figure*

pre_processing

Pre-processing modules take care of ensuring all data are processed the same. Note that gaps should be padded after filtering if you decide not to use these routines (see notes in correlation_warnings). Utilities module whose functions are designed to do the basic processing of the data using obspy modules (which also rely on scipy and numpy).

copyright EQcorrscan developers.

license GNU Lesser General Public License, Version 3 (https://www.gnu.org/copyleft/lesser.html)

Classes & Functions

dayproc	Wrapper for dayproc to parallel multiple traces in a stream.
process	Basic function to process data, usually called by dayproc or shortproc.
shortproc	Basic function to bandpass and downsample.

eqcorrscan.utils.pre_processing.dayproc

eqcorrscan.utils.pre_processing.**dayproc**(*st*, *lowcut*, *highcut*, *filt_order*, *samp_rate*, *starttime*, *parallel=True*, *num_cores=False*, *ignore_length=False*, *seisan_chan_names=False*, *fill_gaps=True*, *ignore_bad_data=False*, *fft_threads=1*)

Wrapper for dayproc to parallel multiple traces in a stream.

Works in place on data. This is employed to ensure all parts of the data are processed in the same way.

Parameters

- **st** (*obspy.core.stream.Stream*) – Stream to process (can be trace).
- **lowcut** (*float*) – Low cut in Hz for bandpass.
- **highcut** (*float*) – High cut in Hz for bandpass.
- **filt_order** (*int*) – Corners for bandpass.
- **samp_rate** (*float*) – Desired sampling rate in Hz.
- **starttime** (*obspy.core.utcdatetime.UTCDateTime*) – Desired start-date of trace.
- **parallel** (*bool*) – Set to True to process traces in parallel, this is often faster than serial processing of traces: defaults to True.
- **num_cores** (*int*) – Control the number of cores for parallel processing, if set to False then this will use all the cores.
- **ignore_length** (*bool*) – See warning below.
- **seisan_chan_names** (*bool*) – Whether channels are named like seisan channels (which are two letters rather than SEED convention of three) - defaults to True.

- **fill_gaps** (*bool*) – Whether to pad any gaps found with zeros or not.
- **ignore_bad_data** (*bool*) – If False (default), errors will be raised if data are excessively gappy or are mostly zeros. If True then no error will be raised, but an empty trace will be returned.
- **fft_threads** (*int*) – Number of threads to use for pyFFTW FFT in resampling. Note that it is not recommended to use fft_threads > 1 and num_cores > 1.

Returns Processed stream.

Return type obspy.core.stream.Stream

Note: If your data contain gaps you should *NOT* fill those gaps before using the pre-process functions. The pre-process functions will fill the gaps internally prior to processing, process the data, then re-fill the gaps with zeros to ensure correlations are not incorrectly calculated within gaps. If your data have gaps you should pass a merged stream without the *fill_value* argument (e.g.: *st = st.merge()*).

Warning: Will fail if data are less than 19.2 hours long - this number is arbitrary and is chosen to alert the user to the dangers of padding to day-long, if you don't care you can ignore this error by setting *ignore_length=True*. Use this option at your own risk! It will also warn any-time it has to pad data - if you see strange artifacts in your detections, check whether the data have gaps.

Example

```
>>> import obspy
>>> if int(obspy.__version__.split('.')[0]) >= 1:
...     from obspy.clients.fdsn import Client
... else:
...     from obspy.fdsn import Client
>>> from obspy import UTCDateTime
>>> from eqcorrscan.utils.pre_processing import dayproc
>>> client = Client('NCEDC')
>>> t1 = UTCDateTime(2012, 3, 26)
>>> t2 = t1 + 86400
>>> bulk_info = [('BP', 'JCNB', '40', 'SP1', t1, t2)]
>>> st = client.get_waveforms_bulk(bulk_info)
>>> st_keep = st.copy()  # Copy the stream for later examples
>>> # Example of bandpass filtering
>>> st = dayproc(st=st, lowcut=2, highcut=9, filt_order=3, samp_rate=20,
...              starttime=t1, parallel=True, num_cores=2)
>>> print(st[0])
BP.JCNB.40.SP1 | 2012-03-26T00:00:00.000000Z - 2012-03-26T23:59:59.950000Z |␣
↪20.0 Hz, 1728000 samples
>>> # Example of lowpass filtering
>>> st = dayproc(st=st, lowcut=None, highcut=9, filt_order=3, samp_rate=20,
...              starttime=t1, parallel=True, num_cores=2)
>>> print(st[0])
BP.JCNB.40.SP1 | 2012-03-26T00:00:00.000000Z - 2012-03-26T23:59:59.950000Z |␣
↪20.0 Hz, 1728000 samples
>>> # Example of highpass filtering
>>> st = dayproc(st=st, lowcut=2, highcut=None, filt_order=3, samp_rate=20,
...              starttime=t1, parallel=True, num_cores=2)
>>> print(st[0])
BP.JCNB.40.SP1 | 2012-03-26T00:00:00.000000Z - 2012-03-26T23:59:59.950000Z |␣
↪20.0 Hz, 1728000 samples
```

eqcorrscan.utils.pre_processing.process

eqcorrscan.utils.pre_processing.**process**(*tr*, *lowcut*, *highcut*, *filt_order*, *samp_rate*, *starttime=False*, *clip=False*, *length=86400*, *seisan_chan_names=False*, *ignore_length=False*, *fill_gaps=True*, *ignore_bad_data=False*, *fft_threads=1*)

Basic function to process data, usually called by dayproc or shortproc.

Functionally, this will bandpass, downsample and check headers and length of trace to ensure files start when they should and are the correct length. This is a simple wrapper on obspy functions, we include it here to provide a system to ensure all parts of the dataset are processed in the same way.

Note: Usually this function is called via dayproc or shortproc.

Parameters

- **tr** (*obspy.core.trace.Trace*) – Trace to process
- **lowcut** (*float*) – Low cut in Hz, if set to None and highcut is set, will use a lowpass filter.
- **highcut** (*float*) – High cut in Hz, if set to None and lowcut is set, will use a highpass filter.
- **filt_order** (*int*) – Number of corners for filter.
- **samp_rate** (*float*) – Desired sampling rate in Hz.
- **starttime** (*obspy.core.utcdatetime.UTCDateTime*) – Desired start of trace
- **clip** (*bool*) – Whether to expect, and enforce a set length of data or not.
- **length** (*float*) – Use to set a fixed length for data from the given starttime.
- **seisan_chan_names** (*bool*) – Whether channels are named like seisan channels (which are two letters rather than SEED convention of three) - defaults to True.
- **ignore_length** (*bool*) – See warning in dayproc.
- **fill_gaps** (*bool*) – Whether to pad any gaps found with zeros or not.
- **ignore_bad_data** (*bool*) – If False (default), errors will be raised if data are excessively gappy or are mostly zeros. If True then no error will be raised, but an empty trace will be returned.
- **fft_threads** (*int*) – Number of threads to use for pyFFTW FFT in resampling

Returns Processed trace.

Type obspy.core.stream.Trace

Note: If your data contain gaps you should *NOT* fill those gaps before using the pre-process functions. The pre-process functions will fill the gaps internally prior to processing, process the data, then re-fill the gaps with zeros to ensure correlations are not incorrectly calculated within gaps. If your data have gaps you should pass a merged stream without the *fill_value* argument (e.g.: *tr = tr.merge()*).

eqcorrscan.utils.pre_processing.shortproc

eqcorrscan.utils.pre_processing.**shortproc**(*st*, *lowcut*, *highcut*, *filt_order*, *samp_rate*, *parallel=False*, *num_cores=False*, *starttime=None*, *endtime=None*, *seisan_chan_names=False*, *fill_gaps=True*, *ignore_length=False*, *ignore_bad_data=False*, *fft_threads=1*)

Basic function to bandpass and downsample.

Works in place on data. This is employed to ensure all parts of the data are processed in the same way.

Parameters

- **st** (*obspy.core.stream.Stream*) – Stream to process
- **lowcut** (*float*) – Low cut for bandpass in Hz
- **highcut** (*float*) – High cut for bandpass in Hz
- **filt_order** (*int*) – Number of corners for bandpass filter
- **samp_rate** (*float*) – Sampling rate desired in Hz
- **parallel** (*bool*) – Set to True to process traces in parallel, for small numbers of traces this is often slower than serial processing, defaults to False
- **num_cores** (*int*) – Control the number of cores for parallel processing, if set to False then this will use all the cores available.
- **starttime** (*obspy.core.utcdatetime.UTCDateTime*) – Desired data start time, will trim to this before processing
- **endtime** (*obspy.core.utcdatetime.UTCDateTime*) – Desired data end time, will trim to this before processing
- **seisan_chan_names** (*bool*) – Whether channels are named like seisan channels (which are two letters rather than SEED convention of three) - defaults to True.
- **fill_gaps** (*bool*) – Whether to pad any gaps found with zeros or not.
- **ignore_length** (*bool*) – Whether to allow data that are less than 80% of the requested length. Defaults to False which will error if short data are found.
- **ignore_bad_data** (*bool*) – If False (default), errors will be raised if data are excessively gappy or are mostly zeros. If True then no error will be raised, but an empty trace will be returned.
- **fft_threads** (*int*) – Number of threads to use for pyFFTW FFT in resampling. Note that it is not recommended to use fft_threads > 1 and num_cores > 1.

Returns Processed stream

Return type obspy.core.stream.Stream

Note: If your data contain gaps you should *NOT* fill those gaps before using the pre-process functions. The pre-process functions will fill the gaps internally prior to processing, process the data, then re-fill the gaps with zeros to ensure correlations are not incorrectly calculated within gaps. If your data have gaps you should pass a merged stream without the *fill_value* argument (e.g.: *st = st.merge()*).

Warning: If you intend to use this for processing templates you should consider how resampling will impact your cross-correlations. Minor differences in resampling between day-long files (which you are likely to use for continuous detection) and shorter files will reduce your cross-correlations!

Example, bandpass

```
>>> from obspy import read
>>> from eqcorrscan.utils.pre_processing import shortproc
>>> # Get the path to the test data
>>> import eqcorrscan
>>> import os
>>> TEST_PATH = os.path.dirname(eqcorrscan.__file__) + '/tests/test_data'
>>> st = read(TEST_PATH + '/WAV/TEST_/2013-09-01-0410-35.DFDPC_024_00')
>>> st = shortproc(st=st, lowcut=2, highcut=9, filt_order=3, samp_rate=20,
...                parallel=True, num_cores=2)
>>> print(st[0])
AF.LABE..SHZ | 2013-09-01T04:10:35.700000Z - 2013-09-01T04:12:05.650000Z | 20.
↪0 Hz, 1800 samples
```

Example, low-pass

```
>>> from obspy import read
>>> from eqcorrscan.utils.pre_processing import shortproc
>>> # Get the path to the test data
>>> import eqcorrscan
>>> import os
>>> TEST_PATH = os.path.dirname(eqcorrscan.__file__) + '/tests/test_data'
>>> st = read(TEST_PATH + '/WAV/TEST_/2013-09-01-0410-35.DFDPC_024_00')
>>> st = shortproc(st=st, lowcut=None, highcut=9, filt_order=3,
...                samp_rate=20)
>>> print(st[0])
AF.LABE..SHZ | 2013-09-01T04:10:35.700000Z - 2013-09-01T04:12:05.650000Z | 20.
↪0 Hz, 1800 samples
```

Example, high-pass

```
>>> from obspy import read
>>> from eqcorrscan.utils.pre_processing import shortproc
>>> # Get the path to the test data
>>> import eqcorrscan
>>> import os
>>> TEST_PATH = os.path.dirname(eqcorrscan.__file__) + '/tests/test_data'
>>> st = read(TEST_PATH + '/WAV/TEST_/2013-09-01-0410-35.DFDPC_024_00')
>>> st = shortproc(st=st, lowcut=2, highcut=None, filt_order=3,
...                samp_rate=20)
>>> print(st[0])
AF.LABE..SHZ | 2013-09-01T04:10:35.700000Z - 2013-09-01T04:12:05.650000Z | 20.
↪0 Hz, 1800 samples
```

sac_util

Part of the EQcorrscan package: tools to convert SAC headers to obspy event objects.

Note: This functionality is not supported for obspy versions below 1.0.0 as references times are not read in by SACIO, which are needed for defining pick times.

---

copyright EQcorrscan developers.

license GNU Lesser General Public License, Version 3 (https://www.gnu.org/copyleft/lesser.html)

Classes & Functions

sactoevent	Convert SAC headers (picks only) to obspy event class.

eqcorrscan.utils.sac_util.sactoevent

eqcorrscan.utils.sac_util.**sactoevent**(*st*)

Convert SAC headers (picks only) to obspy event class.

Picks are taken from header values a, t[0-9]. A phase-hint in the corresponding kt[0-9] slot is recommended.

Parameters **st** (*obspy.core.stream.Stream*) – Stream of waveforms including SAC headers.

Returns Event with picks taken from SAC headers.

Return type obspy.core.event.event.Event

Note: This functionality is not supported for obspy versions below 1.0.0 as reference times are not read in by SACIO, which are needed for defining pick times.

Note: Takes the event origin information from the first trace in the stream - to ensure this works as you expect, please populate the evla, evlo, evdp and nzyear, nzjday, nzhour, nzmin, nzsec, nzmsec for all traces with the same values.

```
>>> from obspy import read
>>> # Get the path to the test data
>>> import eqcorrscan
>>> import os
>>> TEST_PATH = os.path.dirname(eqcorrscan.__file__) + '/tests/test_data'
>>> st = read(TEST_PATH + '/SAC/2014p611252/*')
>>> event = sactoevent(st)
>>> print(event.origins[0].time)
2014-08-15T03:55:21.057000Z
>>> print(event.picks[0].phase_hint)
S
```

stacking

Utility module of the EQcorrscan package to allow for different methods of stacking of seismic signal in one place.

copyright EQcorrscan developers.

license GNU Lesser General Public License, Version 3 (https://www.gnu.org/copyleft/lesser.html)

Classes & Functions

linstack	Compute the linear stack of a series of seismic streams of multiplexed data.
PWS_stack	Compute the phase weighted stack of a series of streams.

Continued on next page

Table 18 – continued from previous page

align_traces	Align traces relative to each other based on their cross-correlation value.

eqcorrscan.utils.stacking.linstack

eqcorrscan.utils.stacking.**linstack**(*streams*, *normalize=True*)

Compute the linear stack of a series of seismic streams of multiplexed data.

Parameters

- **streams** (*list*) – List of streams to stack
- **normalize** (*bool*) – Normalize traces before stacking, normalizes by the RMS amplitude.

Returns stacked data

Return type obspy.core.stream.Stream

eqcorrscan.utils.stacking.PWS_stack

eqcorrscan.utils.stacking.**PWS_stack**(*streams*, *weight=2*, *normalize=True*)

Compute the phase weighted stack of a series of streams.

Note: It is recommended to align the traces before stacking.

Parameters

- **streams** (*list*) – List of obspy.core.stream.Stream to stack.
- **weight** (*float*) – Exponent to the phase stack used for weighting.
- **normalize** (*bool*) – Normalize traces before stacking.

Returns Stacked stream.

Return type obspy.core.stream.Stream

eqcorrscan.utils.stacking.align_traces

eqcorrscan.utils.stacking.**align_traces**(*trace_list*, *shift_len*, *master=False*, *positive=False*, *plot=False*)

Align traces relative to each other based on their cross-correlation value.

Uses the eqcorrscan.core.match_filter.normxcorr2() function to find the optimum shift to align traces relative to a master event. Either uses a given master to align traces, or uses the trace with the highest MAD amplitude.

Parameters

- **trace_list** (*list*) – List of traces to align
- **shift_len** (*int*) – Length to allow shifting within in samples
- **master** (*obspy.core.trace.Trace*) – Master trace to align to, if set to False will align to the largest amplitude trace (default)
- **positive** (*bool*) – Return the maximum positive cross-correlation, or the absolute maximum, defaults to False (absolute maximum).
- **plot** (*bool*) – If true, will plot each trace aligned with the master.

Returns list of shifts and correlations for best alignment in seconds.

Return type list

synth_seis

Early development functions to do **very** basic simulations of seismograms to be used as general matched-filter templates and see how well a simple model would fit with real data. Mostly used in EQcorrscan for testing.

copyright EQcorrscan developers.

license GNU Lesser General Public License, Version 3 (https://www.gnu.org/copyleft/lesser.html)

Classes & Functions

generate_synth_data	Generate a synthetic dataset to be used for testing.
seis_sim	Generate a simulated seismogram from a given S-P time.
SVD_sim	Generate basis vectors of a set of simulated seismograms.
template_grid	Generate a group of synthetic seismograms for a grid of sources.

eqcorrscan.utils.synth_seis.generate_synth_data

eqcorrscan.utils.synth_seis.**generate_synth_data**(*nsta*, *ntemplates*, *nseeds*, *samp_rate*, *t_length*, *max_amp*, *max_lag*, *phaseout='all'*, *jitter=0*, *noise=True*, *same_phase=False*)

Generate a synthetic dataset to be used for testing.

This will generate both templates and data to scan through. Templates will be generated using the utils.synth_seis functions. The day of data will be random noise, with random signal-to-noise ratio copies of the templates randomly seeded throughout the day. It also returns the seed times and signal-to-noise ratios used.

Parameters

- **nsta** (*int*) – Number of stations to generate data for < 15.
- **ntemplates** (*int*) – Number of templates to generate, will be generated with random arrival times.
- **nseeds** (*int*) – Number of copies of the template to seed within the day of noisy data for each template.
- **samp_rate** (*float*) – Sampling rate to use in Hz
- **t_length** (*float*) – Length of templates in seconds.
- **max_amp** (*float*) – Maximum signal-to-noise ratio of seeds.
- **max_lag** (*float*) – Maximum lag time in seconds (randomised).
- **jitter** (*int*) – Random range to allow arrival shifts for seeded phases (samples)
- **noise** (*bool*) – Set to False to give noise-free data.
- **same_phase** (*bool*) – Whether to enforce all positive repeats (True) or not.

Returns Templates: List of obspy.core.stream.Stream

Return type list

Returns Data: obspy.core.stream.Stream of seeded noisy data

Return type obspy.core.stream.Stream

Returns Seeds: dictionary of seed SNR and time with time in samples.

Return type dict

eqcorrscan.utils.synth_seis.seis_sim

eqcorrscan.utils.synth_seis.**seis_sim**(*sp*, *amp_ratio=1.5*, *flength=False*, *phaseout='all'*)

Generate a simulated seismogram from a given S-P time.

Will generate spikes separated by a given S-P time, which are then convolved with a decaying sine function. The P-phase is simulated by a positive spike of value 1, the S-arrival is simulated by a decaying boxcar of maximum amplitude 1.5. These amplitude ratios can be altered by changing the amp_ratio, which is the ratio S amplitude:P amplitude.

Note: In testing this can achieve 0.3 or greater cross-correlations with data.

Parameters

- **sp** (*int*) – S-P time in samples
- **amp_ratio** (*float*) – S:P amplitude ratio
- **flength** (*int*) – Fixed length in samples, defaults to False
- **phaseout** (*str*) – Either 'P', 'S' or 'all', controls which phases to cut around, defaults to 'all'. Can only be used with 'P' or 'S' options if flength is set.

Returns Simulated data.

Return type numpy.ndarray

eqcorrscan.utils.synth_seis.SVD_sim

eqcorrscan.utils.synth_seis.**SVD_sim**(*sp*, *lowcut*, *highcut*, *samp_rate*, *amp_range=array([-10., -9.99, -9.98, ..., 9.97, 9.98, 9.99])*)

Generate basis vectors of a set of simulated seismograms.

Inputs should have a range of S-P amplitude ratios, in theory to simulate a range of focal mechanisms.

Parameters

- **sp** (*int*) – S-P time in seconds - will be converted to samples according to samp_rate.
- **lowcut** (*float*) – Low-cut for bandpass filter in Hz
- **highcut** (*float*) – High-cut for bandpass filter in Hz
- **samp_rate** (*float*) – Sampling rate in Hz
- **amp_range** (*numpy.ndarray*) – Amplitude ratio range to generate synthetics for.

Returns set of output basis vectors

Return type numpy.ndarray

eqcorrscan.utils.synth_seis.template_grid

eqcorrscan.utils.synth_seis.**template_grid**(*stations*, *nodes*, *travel_times*, *phase*, *PS_ratio=1.68*, *samp_rate=100*, *flength=False*, *phaseout='all'*)

Generate a group of synthetic seismograms for a grid of sources.

Used to simulate phase arrivals from a grid of known sources in a three-dimensional model. Lags must be known and supplied, these can be generated from the bright_lights function: read_tt, and resampled to fit the desired grid dimensions and spacing using other functions therein. These synthetic seismograms are very simple models of seismograms using the seis_sim function herein. These approximate body-wave P and S first arrivals as spikes convolved with damped sine waves.

Parameters

- **stations** (*list*) – List of the station names
- **nodes** (*list*) – List of node locations in (lon,lat,depth)
- **travel_times** (*numpy.ndarray*) – Array of travel times where travel_times[i][:] refers to the travel times for station=stations[i], and travel_times[i][j] refers to stations[i] for nodes[j]
- **phase** (*str*) – Can be either 'P' or 'S'
- **PS_ratio** (*float*) – P/S velocity ratio, defaults to 1.68
- **samp_rate** (*float*) – Desired sample rate in Hz, defaults to 100.0
- **flength** (*int*) – Length of template in samples, defaults to False
- **phaseout** (*str*) – Either 'S', 'P', 'all' or 'both', determines which phases to clip around. 'all' Encompasses both phases in one channel, but will return nothing if the flength is not long enough, 'both' will return two channels for each stations, one SYN_Z with the synthetic P-phase, and one SYN_H with the synthetic S-phase.

Returns List of obspy.core.stream.Stream

trigger

Functions to enable simple energy-base triggering in a network setting where different stations have different noise parameters.

copyright EQcorrscan developers.

license GNU Lesser General Public License, Version 3 (https://www.gnu.org/copyleft/lesser.html)

Classes & Functions

network_trigger	Main function to compute triggers for a network of stations.
read_trigger_parameters	Read the trigger parameters into trigger_parameter classes.
TriggerParameters	Base class for trigger parameters.

eqcorrscan.utils.trigger.network_trigger

eqcorrscan.utils.trigger.**network_trigger**(*st*, *parameters*, *thr_coincidence_sum*, *moveout*, *max_trigger_length=60*, *despike=True*, *parallel=True*)

Main function to compute triggers for a network of stations. Computes single-channel characteristic func-

tions using given parameters, then combines these to find network triggers on a number of stations within a set moveout window.

Parameters

- **st** (*obspy.core.stream.Stream*) – Stream to compute detections within
- **parameters** (*list*) – List of parameter class
- **thr_coincidence_sum** (*int*) – Minimum number of stations required to raise a network trigger.
- **moveout** (*float*) – Window to find triggers within in the network detection stage.
- **max_trigger_length** (*float*) – Maximum trigger length in seconds, used to remove long triggers - can set to False to not use.
- **despike** (*bool*) – Whether to apply simple despiking routine or not
- **parallel** (*bool*) – Whether to run in parallel or not

Returns List of triggers

Return type list

Example

```
>>> from obspy import read
>>> from eqcorrscan.utils.trigger import TriggerParameters, network_trigger
>>> st = read("https://examples.obspy.org/" +
...           "example_local_earthquake_3stations.mseed")
>>> parameters = []
>>> for tr in st:
...     parameters.append(TriggerParameters({'station': tr.stats.station,
...                                          'channel': tr.stats.channel,
...                                          'sta_len': 0.5,
...                                          'lta_len': 10.0,
...                                          'thr_on': 10.0,
...                                          'thr_off': 3.0,
...                                          'lowcut': 2.0,
...                                          'highcut': 15.0}))
>>> triggers = network_trigger(st=st, parameters=parameters,
...                            thr_coincidence_sum=5, moveout=30,
...                            max_trigger_length=60, despike=False)
>>> print(len(triggers))
1
```

eqcorrscan.utils.trigger.read_trigger_parameters

eqcorrscan.utils.trigger.**read_trigger_parameters**(*filename*)

Read the trigger parameters into trigger_parameter classes.

Parameters **filename** (*str*) – Parameter file

Returns List of *eqcorrscan.utils.trigger.TriggerParameters*

Return type list

Example

```
>>> from eqcorrscan.utils.trigger import read_trigger_parameters
>>> parameters = read_trigger_parameters('parameters') # doctest: +SKIP
```

eqcorrscan.utils.trigger.TriggerParameters

class eqcorrscan.utils.trigger.**TriggerParameters**(*header={}*)

Bases: obspy.core.util.attribdict.AttribDict

Base class for trigger parameters.

```
>>> from eqcorrscan.utils.trigger import TriggerParameters
>>> defaults = TriggerParameters()
>>> defaults.station = 'TEST'
>>> # Or use dictionaries
>>> defaults['station'] = 'ALF'
>>> defaults = TriggerParameters({'station': 'TEST',
...                               'channel': 'SHZ',
...                               'sta_len': 0.3,
...                               'lta_len': 10.0,
...                               'thr_on': 10,
...                               'thr_off': 3,
...                               'lowcut': 2,
...                               'highcut': 20})
>>> print(defaults.station)
TEST
```

__init__(*header={}*)

Methods

__init__([header])	
clear()	
copy()	
get(k[,d])	
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem()	as a 2-tuple; but raise KeyError if D is empty.
setdefault(k[,d])	
update([E,]**F)	If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v
values()	
write(filename[, append])	Write the parameters to a file as a human-readable series of dicts.

Attributes

defaults
do_not_warn_on
readonly
warn_on_non_default_key

Python Module Index

e

Index

Symbols

A

C

D

E

F

G

I

L

M

N

O

P

R

S

T

W

X