Originally, MSNoise was a “Python Package for Monitoring Seismic Velocity Changes using Ambient Seismic Noise”. With the release of MSNoise 1.4, and because of the Plugin Support, we could call MSNoise: “Measuring with Seismic Noise”. The current release version of MSNoise is MSNoise 1.6.

The standard MSNoise workflow is designed to go from seismic data archives to dv/v curves. The monitoring is achieved by computing the cross-correlation of continuous seismic records for each pair of a network and by studying the changes in the cross-correlation function relative to a reference.

The goal of the “suite” is to provide researchers with an efficient processing tool, while keeping the need for coding to a minimum and avoiding being a black box. Moreover, as long as the in- and outputs of each step are respected, they can easily be replaced with one’s own codes! (See Workflow (page 13)).

Plugins can be added and extend the standard workflow from any steps, e.g., using MSNoise as a cross-correlation toolbox until the stack step, and then branching to the workflow provided by one’s plugin.


This documentation is also available in PDF format on the MSNoise Website (PDF).
1.1 Installation

MSNoise is a python package that uses a database (sqlite or MySQL) for storing station and files metadata together with jobs. When installed, it provides a top level command `msnoise` in the console.

This version will be the last to be tested on Python 2.7. The EOL (end of life) of 2.7 is 2020, which means it is high time for users to migrate. For users having a complete set of tools in Python 2.7 and not keen to move to 3.x soon, the incredible easiness of creating a Python 3.x environment in conda, for example, will allow them to run MSNoise in the future.

Note that MSNoise is always tested against the latest release versions of the main packages, so older installations that are not maintained/updated regularly (years) could encounter issues. Please make sure you have the latest version of Numpy and Scipy (and MKL), as performance gets better and better (especially since Anaconda Inc. released its fast MKL implementations for all users, in the conda-forge channel).

To run MSNoise, you need:

- A recent version of Python (3.x recommended). We suggest using Anaconda with a few extra modules. MSNoise is tested “continuously” by automatic build systems (TravisCI and Appveyor) for Python 2.7 and Python 3.7, on Windows, Linux and MacOSX 64 bits systems! Support for Python 2.7 will be dropped as soon as the TravisCI test don’t pass and the corrections would take too much dev time.
  - Those modules are already distributed with Anaconda:
    * setuptools
    * numpy
    * scipy
    * pandas
    * matplotlib
    * statsmodels
    * sqlalchemy
    * click
    * flask
    * pymysql
* wtforms
  – Not shipped with Anaconda:
    * obspy
    * flask-admin
    * markdown
    * folium
    * flask-wtf

- MySQL: if you want to use MySQL, you need to install and configure a MySQL Server beforehand. This is not needed for sqlite. Read About Databases and Performances (page 106) for more information. We recommend using MySQL.

### 1.1.1 Full Installation

1. Download and install Anaconda for your machine, make sure Anaconda’s Python is the default python for your user

2. Execute the following command to install the missing packages:

   ```
   conda install -c conda-forge flask-admin flask-wtf markdown folium pymysql
   conda install -c conda-forge obspy
   ```

3. Install a MySQL server and MySQL Workbench:

   Download and install MySQL Community Server (MySQLs ) and MySQL Workbench (MySQLw ) ; On Windows one can also use the MySQL installer (MySQLi ).

   On Linux, the MySQL server can also be installed using the following command:

   ```
   sudo apt-get install mysql-server
   ```

4. Create a privileged user and a database:

   - Start MySQL Workbench and connect to the local database
   - Click on “Privileges” and create a new user, with all privileges (Select all). Ideally, create user “msnoise” with password “msnoise”.

5. Install the latest release version of MSNoise:

   ```
   pip install msnoise
   ```

   Power user could install the development version too, but it is not recommended.

6. Check which required packages you are still missing by executing the msnoise bugreport command. (See Testing the Dependencies (page 60))

7. To be sure all is running OK, one could start the msnoise test command. This will start the standard MSNoise test suite, which should end with a “Ran xx tests in yy seconds : OK”.

8. Proceed to the Workflow (page 13) description to start MSNoise!

Done!
1.1.2 MySQL Server and Workbench

Using the MySQL Server and Workbench is fairly easy and lots of tutorials are available online as text or videos.

Once both are installed, start Workbench and you should see the local MySQL server automatically identified:

And by clicking on “Local Instance . . .” another tab should open, connected to the local database.

Create a msnoise user

Select “Users and Privileges” in the left sidebar, then “Add Account”. Define the username and the password (msnoise:msnoise could do, although “weak”):
Then, under “Administrative Roles”, grant this user the **DBA** mode (user can perform all tasks on the database server) and click “Apply”. 
Create an empty database

Ideally, each “project” needs a database. For example, if one has two different volcanoes and wants to run MSNoise using the these distinct datasets, one needs to create two empty databases. For users who have access to only 1 database, the `msnoise db init` allows to provide a prefix, which works like the Wordpress prefixes: for example if a prefix is “vA”, the `config` table that will be created is `vA_config` in the database.

Click on the “Create new schema” button in the taskbar:

![Create new schema in MySQL Workbench](image)

and provide a name for the database (for example msnoise; or msnoise_project1, or project1, or anything else) ; and click “Apply”: 

![Create new database in MySQL Workbench](image)
and click “Apply” again and it should state all is OK:
When done, the database we can be seen in the left sidebar:

And you're ready to start your first project: *Workflow* (page 13).
When moving your project to a larger server, HPC or else, just add the connection to this server in Workbench and you’re good to go with the very same interface/tool!

1.1.3 MySQL/MariaDB configuration

You can also set up a database server using MariaDB, there are plenty tutorials of how to set it up as well. The new default character set for MySQL or MariaDB is not simple utf8, so make sure that the configuration file (/etc/mysql/my.cnf under Linux) contains the following lines. There are issues with the latest MySQL versions which prevent a “traditional group by” statement.

```
[mysqld]
character-set-server=utf8
collation-server=utf8_unicode_ci
sql_mode="TRADITIONAL,NO_AUTO_CREATE_USER"
```

For Mac, this seemed to work for users (see Issue72):

```
[mysqld]
sql_mode=STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,
        NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION
```

1.1.4 Database Structure - Tables

MSNoise will create the tables automatically upon running the installer script (see Workflow (page 13)).

1.1.5 Building this documentation

To build this documentation, some modules are required:

```
pip install sphinx
pip install sphinx_bootstrap_theme
```

Then, this should simply work:

```
make html
```

it will create a .build folder containing the documentation.

You can also build the doc to Latex and then use your favorite Latex-to-PDF tool.

1.1.6 Using the development version

This is not recommended, but users willing to test the latest development (hopefully stable) version of MSNoise can:

```
pip uninstall msnoise
pip install http://msnoise.org/master.zip
```
Please note this version most probably uses the very latest version of every package: Release versions of *numpy*, *scipy*, etc obtained from conda-forge and “master” version of *obspy*. The development version (master) of obspy can be installed from github:

```
pip uninstall obspy
pip install https://github.com/obspy/obspy/archive/master.zip
```

If you are using the master version, please use the issue tracker of github to communicate about bugs and not the mailing list, preferably used for Releases.
2.1 Workflow

This section only presents the “init” and configuration of MSNoise (read “the first startup of MSNoise”), not the installation of the required software, which is described in Installation (page 3).

2.1.1 Initialize Project

This console script is responsible for asking questions about the database connection, to create the db.ini file and to create the tables in the database.

Questions are:

- What database technology do you want to use?
  - sqlite: this will create a file in the current folder and use it as DB
  - mysql: this will connect to a local or remote mysql server, additional information is then required:
    - hostname: of the mysql server, defaults to 127.0.0.1
database: must already exist on hostname
username: as registered in the privileged users of the mysql server
password: his password
prefix: useful when users have only access to a single database. Similar to the way wordpress handles prefixes. The tables will be named %prefix%config (etc) instead of config, for example.

The SQLite choice will create a xxx.sqlite file in the current (project) folder, while, for MySQL, one has to create an empty database first on the mysql server, see how to do this (page 7). To run this script:

```
msnoise db init --help
```

Usage: [OPTIONS]

This command initializes the current folder to be a MSNoise Project by creating a database and a db.ini file.

Options:

--tech TEXT Database technology: 1=SQLite 2=MySQL
--help Show this message and exit.

Warning: The credentials will be saved in a flat text file in the current directory. It’s not very safe, but until now we haven’t thought of another solution.

### 2.1.2 MSNoise Admin (Web Interface)

MSNoise Admin is a web interface that helps the user define the configuration for all the processing steps. It allows configuring the stations and filters to be used in the different steps of the workflow and provides a view on the database tables.

To start the admin:

```
$ msnoise admin
```

Which, by default, starts a web server listening on all interfaces on port 5000. This can be overridden by passing parameters to the command, e.g. for port 5099:

```
$ msnoise admin -p 5099
```

The next step consists of opening a web browser and open the ip address of the machine, by default on the current machine, it’ll be http://localhost:5000/ or http://127.0.0.1:5000/.
The top level menu shows four items:

**Home**

The index page shows

- The project location and its database
- Stats of the Data Availability, the CC, STACK, MWCS and DTT jobs.
- Quick action buttons for resetting or deleting jobs.

The name and the logo of the page can be overridden by setting an environment variable with a name and the HTML tag of the logo image:

```
set msnoise_brand="ROB|<img src='http://www.seismologie.be/img/oma/ROB-logo.svg' width=200 height=200>
```

and then starting msnoise admin:

2.1. **Workflow**
ROB Dashboard

Project Folder: C:\tmp
Project Database: SQLite: msnoise.sqlite

Configuration

Station

Stations appear as a table and are editable.

Stations are defined as:

class msnoise.msnoise_admin.Station(*args)
    Station Object

Parameters

- **ref** (*int*) – The Station ID in the database
- **net** (*str*) – The network code of the Station
- **sta** (*str*) – The station code
- **X** (*float*) – The X coordinate of the station
- **Y** (*float*) – The Y coordinate of the station
- **altitude** (*float*) – The altitude of the station
- **coordinates** (*str*) – The coordinates system. “DEG” is WGS84 latitude/longitude in degrees. “UTM” is expressed in meters.
- **instrument** (*str*) – The instrument code, useful with PAZ correction
- **used** (*bool*) – Whether this station must be used in the computations.

Attributes

- **X**
- **Y**
- **altitude**
- **coordinates**
- **instrument**
- **net**
Filters appear as a table and are editable. The filter parameters are validated before submission, so no errors should happen. Note: by default, the `used` parameter is set to `False`, **don’t forget to change it**!

Filters are defined as:

```python
class msnoise.msnoise.admin.Filter(**kwargs)
    Filter base class.
```

**Parameters**

- `ref (int)` – The id of the Filter in the database
- `low (float)` – The lower frequency bound of the Whiten function (in Hz)
- `high (float)` – The upper frequency bound of the Whiten function (in Hz)
- `mwcs_low (float)` – The lower frequency bound of the linear regression done in MWCS (in Hz)
- `mwcs_high (float)` – The upper frequency bound of the linear regression done in MWCS (in Hz)
- `rms_threshold (float)` – Not used anymore
- `mwcs_wlen (float)` – Window length (in seconds) to perform MWCS
- `mwcs_step (float)` – Step (in seconds) of the windowing procedure in MWCS
- `used (bool)` – Is the filter activated for the processing

**Attributes**

- `high`
- `low`
- `mwcs_high`
- `mwcs_low`
- `mwcs_step`
- `mwcs_wlen`
- `ref`
- `rms_threshold`
- `used`

---

2.1. Workflow
**Config**

All configuration bits appear as a table and are editable. When editing one configuration item, the Edit tab shows extra information about the parameter, where it is used and its default value. Most of the configuration bits are case-sensitive!

Example view:

```
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_folder</td>
<td>CC Output Folder in case keep_all=Y, to store the individual windows. The daily CCF will always be stored in the STACKS/001_DAYS folder.</td>
<td>CROSS_CORRELATIONS</td>
</tr>
<tr>
<td>output_folder</td>
<td>CC Output Folder</td>
<td>CROSS_CORRELATIONS</td>
</tr>
<tr>
<td>data_structure</td>
<td>Either a predefined acronym [SDS]/BUD/IDDS, or /-separated path (e.g. NET/STA/YEAR/NET.STA.YEAR.DAY.MSEED).</td>
<td>SDS</td>
</tr>
<tr>
<td>archive_format</td>
<td>Force format of archive files to read? Leave empty for slightly slower auto-detection by Obspy, or specify any format supported by obspy.core.stream.read.</td>
<td></td>
</tr>
<tr>
<td>network</td>
<td>Network to analyse [*]</td>
<td>*</td>
</tr>
</tbody>
</table>
```

The table below lists the different fields:

Continued on next page
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>channels</td>
<td>Channels need to match the value (ex: [*], <em>Z, BH</em>, HHZ...)</td>
<td>*</td>
</tr>
<tr>
<td>startdate</td>
<td>Start Date to process: [1970-01-01]=’since beginning of the archive’</td>
<td>1970-01-01</td>
</tr>
<tr>
<td>enddate</td>
<td>End Date to process: [2100-01-01]=’No end’</td>
<td>2021-01-01</td>
</tr>
<tr>
<td>analysis_duration</td>
<td>Duration of the Analysis (total in seconds : 3600, [86400])</td>
<td>86400</td>
</tr>
<tr>
<td>cc_sampling_rate</td>
<td>Sampling Rate for the Cross-Correlation [20.0]</td>
<td>20.0</td>
</tr>
<tr>
<td>resampling_method</td>
<td>Resampling method Decimate/[Lanczos]</td>
<td>Lanczos</td>
</tr>
<tr>
<td>preprocess_lowpass</td>
<td>Preprocessing Low-pass value in Hz [8.0]</td>
<td>8.0</td>
</tr>
<tr>
<td>preprocess_highpass</td>
<td>Preprocessing High-pass value in Hz [0.01]</td>
<td>0.01</td>
</tr>
<tr>
<td>preprocess_max_gap</td>
<td>Preprocessing maximum gap length that will be filled by interpolation [10.0] seconds</td>
<td>10.0</td>
</tr>
<tr>
<td>preprocess_taper_length</td>
<td>Duration of the taper applied at the beginning and end of trace during the preprocessing, to allow highpassfiltering</td>
<td>20.0</td>
</tr>
<tr>
<td>remove_response</td>
<td>Remove instrument response Y/[N]</td>
<td>N</td>
</tr>
<tr>
<td>response_format</td>
<td>Remove instrument file format [dataless]/inventory/paz/resp</td>
<td>dataless</td>
</tr>
<tr>
<td>response_path</td>
<td>Instrument correction file(s) location (path relative to db.ini), defaults to ‘./inventory’, i.e. a subfolder in the current project folder.&lt;br&gt;All files in that folder will be parsed.</td>
<td>inventory</td>
</tr>
<tr>
<td>response_prefilt</td>
<td>Remove instrument correction pre-filter (0.005, 0.006, 30.0, 35.0)</td>
<td>(0.005, 0.006, 30.0, 35.0)</td>
</tr>
<tr>
<td>maxlag</td>
<td>Maximum lag (in seconds) [120.0]</td>
<td>120.</td>
</tr>
<tr>
<td>corr_duration</td>
<td>Data windows to correlate (in seconds) [1800.]</td>
<td>1800.</td>
</tr>
<tr>
<td>overlap</td>
<td>Amount of overlap between data windows [0:1] [0.]</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>windsorizing</td>
<td>Windsorizing at N time RMS, 0 disables windsorizing, -1 enables 1-bit normalization[3]</td>
<td>3</td>
</tr>
<tr>
<td>whitening</td>
<td>Whiten Traces before cross-correlation: [A]ll (except for autocorr), [N]one, or only if [C]omponents are different: [A]/N/C</td>
<td>A</td>
</tr>
<tr>
<td>whitening_type</td>
<td>Type of spectral whitening function to use: [B]rutal (amplitude to 1.0), divide spectrum by its [PSD]: [B]/PSD. WARNING: only works for compute_cc, not compute_cc_rot, where it will always be [B]</td>
<td>B</td>
</tr>
<tr>
<td>stack_method</td>
<td>Stack Method: Linear Mean or Phase Weighted Stack: [linear]/pws</td>
<td>linear</td>
</tr>
<tr>
<td>pws_timegate</td>
<td>If stack_method='pws', width of the smoothing in seconds: 10.0</td>
<td>10.0</td>
</tr>
<tr>
<td>pws_power</td>
<td>If stack_method='pws', Power of the Weighting: 2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>crondays</td>
<td>Number of days to monitor with scan_archive, typically used in cron (should be a float representing a number of days, or a string designating weeks, days, and/or hours using the format ‘XwXdXh’) [1]</td>
<td>1</td>
</tr>
<tr>
<td>components_to_compute</td>
<td>List (comma separated) of components to compute between two different stations [ZZ]</td>
<td>ZZ</td>
</tr>
<tr>
<td>cc_type</td>
<td>Cross-Correlation type [CC]</td>
<td>CC</td>
</tr>
<tr>
<td>components_to_compute_single</td>
<td>List (comma separated) of components within a single station. ZZ would be the autocorrelation of Z component, while ZE or ZN are the cross-components. Defaults to [], no single-station computations are done.</td>
<td></td>
</tr>
<tr>
<td>cc_type_single_station_AC</td>
<td>Auto-Correlation type [CC]</td>
<td>CC</td>
</tr>
</tbody>
</table>

Continued on next page
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc_type_single_station_SC</td>
<td>Cross-Correlation type for Cross-Components [CC]</td>
<td>CC</td>
</tr>
<tr>
<td>autocorr</td>
<td>DEPRECATED, add the components to compute on single stations in the components_to_compute_single_station config parameter.</td>
<td>N</td>
</tr>
<tr>
<td>keep_all</td>
<td>Keep all cross-corr (length: corr_duration) [Y]/N</td>
<td>N</td>
</tr>
<tr>
<td>keep_days</td>
<td>Keep all daily cross-corr [Y]/N</td>
<td>Y</td>
</tr>
<tr>
<td>ref_begin</td>
<td>Beginning or REF stacks. Can be absolute (2012-01-01) or relative (-100) days</td>
<td>1970-01-01</td>
</tr>
<tr>
<td>ref_end</td>
<td>End or REF stacks. Same as ref_begin</td>
<td>2021-01-01</td>
</tr>
<tr>
<td>mov_stack</td>
<td>Number of days to stack for the Moving-window stacks ([5]= [day-4:day]), can be a comma-separated list 1,2,5,10</td>
<td>5</td>
</tr>
<tr>
<td>export_format</td>
<td>Export stacks in which format(s) ? SAC/MSEED/[BOTH]</td>
<td>MSEED</td>
</tr>
<tr>
<td>sac_format</td>
<td>Format for SAC stacks ? [doublets]/clarke</td>
<td>doublets</td>
</tr>
<tr>
<td>dtt_lag</td>
<td>How is the lag window defined [dynamic]/static</td>
<td>static</td>
</tr>
<tr>
<td>dtt_v</td>
<td>If dtt_lag=dynamic: what velocity to use to avoid ballistic waves [1.0]km/s</td>
<td>1.0</td>
</tr>
<tr>
<td>dtt_minlag</td>
<td>If dtt_lag=static: min lag time</td>
<td>5.0</td>
</tr>
<tr>
<td>dtt_width</td>
<td>Width of the time lag window [30]s</td>
<td>30.0</td>
</tr>
<tr>
<td>dtt_sides</td>
<td>Which sides to use [both]/left/right</td>
<td>both</td>
</tr>
<tr>
<td>dtt_mincoh</td>
<td>Minimum coherence on dt measurement, MWCS points with values lower than that will not be used in the WLS</td>
<td>0.65</td>
</tr>
<tr>
<td>dtt_maxerr</td>
<td>Maximum error on dt measurement, MWCS points with values larger than that will not be used in the WLS</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Continued on next page
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtt_maxdt</td>
<td>Maximum dt values, MWCS points with values larger than that will not be used in the WLS</td>
<td>0.1</td>
</tr>
<tr>
<td>plugins</td>
<td>Comma separated list of plugin names. Plugins names should be importable Python modules.</td>
<td></td>
</tr>
<tr>
<td>hpc</td>
<td>Is MSNoise going to run on an HPC? Y/N</td>
<td>N</td>
</tr>
<tr>
<td>stretching_max</td>
<td>Maximum stretching coefficient, e.g. 0.5 = 50%, 0.01 = 1%</td>
<td>0.01</td>
</tr>
<tr>
<td>stretching_nsteps</td>
<td>Number of stretching steps between 1-stretching_max and 1+stretching_max</td>
<td>1000</td>
</tr>
</tbody>
</table>

Database

Data Availability

Gives a view of the data_availability table. Allows to bulk edit/select rows. Its main goal is to check that the scan_archive procedure has successfully managed to list all files from one’s archive.

Jobs

Gives a view of the jobs table. Allows to bulk edit/select rows. Its main goal is to check the new_jobs or any other workflow step (or Plugins) successfully inserted/updated jobs.

Help

About

Shows some links and information about the package. Mostly the information present on the github readme file.

Bug Report

Web view of the msnoise bugreport -m, allows viewing if all required python modules are properly installed and available for MSNoise.
2.1.3 Populate Station Table

This script is responsible for rapidly scanning the data archive, identifying the Networks/Stations and inserting them in the stations table in the database.

The data_folder (as defined in the config) is scanned following the data_structure. Possible values for the data_structure are defined in data_structures.py:

```python
data_structure['SDS'] = "YEAR/NET/STA/CHAN.TYPE/YEAR.STA.LOC.CHAN.TYPE.YEAR.DAY"
data_structure['BUD'] = "NET/STA/STA.NET.LOC.CHAN.YEAR.DAY"
data_structure['IDDS'] = "YEAR/STA/CHAN.TYPE/YEAR.STA.LOC.CHAN.TYPE.YEAR.DAY.
˓→HOUR"
data_structure['PDF'] = "YEAR/STA/CHAN.TYPE/YEAR.STA.LOC.CHAN.TYPE.YEAR.DAY"
```

If your data structure corresponds to one of these 4 structures, you need to select the corresponding acronym (SDS, BUD, IDDS or PDF) for the data_structure field.

More info on the recommended SDS (“SeisComP Data Structure”) can be found here: https://www.seiscomp3.org/wiki/doc/applications/slarchive/SDS For other simple structures, one has to edit the data_structure configuration (see below).

By default, station coordinates are initialized at 0.

To run this script:

```
$ msnoise populate
```

Custom data structure & station table population

If one’s data structure does not belong to the pre-defined ones, it can be defined directly in the data_structure configuration field using forward slashes, e.g.:

```python
data_structure = "NET/STA/YEAR/NET.STA.YEAR.DAY.MSEED"
```

MSNoise expects to find a file named custom.py in the current folder. This python file will contain a function called populate which will accept one argument and return a station dictionary with keys of the format NET_STA, and fields for the stations table in the database: Net,Sta,X,Y,Altitude, Coordinates(UTM/DEG),Instrument.

```python
import os, glob

def populate(data_folder):
    datalist = sorted(glob.glob(os.path.join(data_folder, "*", "*")))
    stationdict = {}
    for di in datalist:
        tmp = os.path.split(di)
        sta = tmp[1]
        net = os.path.split(tmp[0])[1]
        stationdict[net+"_"+sta]=[net,sta,0.0,0.0,0.0,"UTM","N/A"]
    return stationdict
```

Expert (lazy) mode:

If the DataAvailability has already been filled in by another process, for example using the “scan from path” (page 24) procedure, the network/station names can be “populated” from the DataAvailability table automatically. To do this, simply run:

2.1. Workflow
msnoise populate --fromDA

and MSNoise will insert the unique NET.STA in the Stations table.

### 2.1.4 Scan Archive

One advantage of MSNoise is its ability to be used as an automated monitoring tool. In order to run every night on the data acquired during the previous day, MSNoise needs to check the data archive for new or modified files.

Those files could have been acquired during the last day, but be data of a previously offline station and contain useful information for, say, a month ago. The time to search for is defined in the config from the `crondays` value. For convenience, this parameter can be temporarily redefined on the command line using the `--crondays` option of the `scan_archive` sub-command. In both cases, it can be a float designating a number of days in the past, or a string designating a number of weeks, days, and/or hours in the format ‘Xw Xd Xh’ (each group being optional, as well as the separating blanks).

The `scan_archive` script inspects the modified time attribute (‘mtime’) of files in the archives to locate new or modified files. Once located, they are inserted (if new) or updated (if modified) in the data availability table.

To run the code on two Process, execute the following in console:

```
$ msnoise -t 2 scan_archive
```

**Compulsory Special case: first run**

This script is the same as for the routine, but one has to pass the `--init` option. The `scan_archive` will scan all files in the data folders, regardless of their modification time.

```
$ msnoise -t 2 scan_archive --init
```

This will scan the data_archive folder the configured stations and will insert all files found in the data_availability table in the database. As usual, calling the script with a `--help` argument will show its usage.

**Expert (lazy) mode:**

Sometimes, you only want to scan a few files and run MSNoise on them. To do this simply run:

```
$ msnoise scan_archive --path /path/to/where/files/are --init
```

and MSNoise will read anything ObsPy can (provided the files have a proper header (network code, station code and channel code)). Then, once done, simply run the **“populate from DataAvailability”** (page 23) procedure.

This command can also scan folders recursively:

```
$ msnoise scan_archive --path /path/to/archive --recursively --init
```
2.1.5 New Jobs

This script searches the database for files flagged “N”ew or “M”odified. For each date in the configured range, it checks if other stations are available and defines the new jobs to be processed. Those are inserted in the jobs table of the database.

To run it from the console:

```
$ msnoise new_jobs
```

Upon first run, if you expect the number of jobs to be large (many days, many stations), pass the `--init` parameter to optimize the insert. Only use this flag once, otherwise problems will arise from duplicate entries in the jobs table.

```
$ msnoise new_jobs --init
```

Performance / running on HPC

By setting the `hpc` configuration parameter to `Y`, you will disable the automatic creation of jobs during the workflow, to avoid numerous interactions with the database (select & update or insert). The jobs have then to be inserted manually:

```
$ msnoise new_jobs --hpc CC:STACK
```

should be run after the `msnoise compute_cc` step in order to create the `STACK` jobs.

2.1.6 Compute Cross-Correlations

This code is responsible for the computation of the cross-correlation functions.

This script will group jobs marked “T”odo in the database by day and process them using the following scheme. As soon as one day is selected, the corresponding jobs are marked “I”n Progress in the database. This allows running several instances of this script in parallel.

As of MSNoise 1.6, the compute step has been completely rewritten:

- The `compute_cc` step has been completely rewritten to make use of 2D arrays holding the data, processing them “in place” for the different steps (FFT, whitening, etc). This results in much more efficient computation. The process slides on time windows and computes the correlations using indexes in a 2D array, therefore avoiding an exponential number of identical operations on data windows.

- This new code is the default `compute_cc`, and it doesn’t allow computing rotated components. For users needing R or T components, there are two options: either use the old code, now named `compute_cc_rot`, or compute the full (6 components actually are enough) tensor using the new code, and rotate the components afterwards. From initial tests, this latter solution is a lot faster than the first, thanks to the new processing in 2D.

- It is now possible to do the Cross-Correlation (classic “CC”), the Auto-Correlation (“AC”) or the Cross-Components within the same station (“SC”). To achieve this, we removed the ZZ, ZT, etc parameters from the configuration and replaced it with `components_to_compute` which takes a list: e.g. `ZZ, ZE, ZN, EZ, EE, EN, NZ, NE, NN` for the full non-rotated tensor between stations. Adding components to the new
components_to_compute_single_station will allow computing the cross-components (SC) or auto-correlation (AC) of each station.

- The cross-correlation is done on sliding windows on the available data. For each window, if one trace contains a gap, it is eliminated from the computation. This corrects previous errors linked with gaps synchronised in time that lead to perfect sinc autocorrelation functions. The windows should have a duration of at least “2 times the ‘maxlag’+1” to be computable.

Configuration Parameters

The following parameters (modifiable via `msnoise admin`) are used for this step:

- components_to_compute: List (comma separated) of components to compute between two different stations [ZZ] (default=ZZ)
- components_to_compute_single_station: List (comma separated) of components within a single station. ZZ would be the autocorrelation of Z component, while ZE or ZN are the cross-components. Defaults to [], no single-station computations are done. (default=)
  | new in 1.6
- cc_sampling_rate: Sampling Rate for the CrossCorrelation [20.0] (default=20.0)
- analysis_duration: Duration of the Analysis (total in seconds : 3600, [86400]) (default=86400)
- overlap: Amount of overlap between data windows [0:1] [0.] (default=0.0)
- maxlag: Maximum lag (in seconds) [120.0] (default=120.)
- corr_duration: Data windows to correlate (in seconds) [1800.] (default=1800.)
- windsorizing: Windsorizing at N time RMS , 0 disables windsorizing, -1 enables 1-bit normalization [3] (default=3)
- resampling_method: Resampling method Decimate/[Lanczos] (default=Lanczos)
- remove_response: Remove instrument response Y/[N] (default=N)
- response_format: Remove instrument file format [dataless]/inventory/paz/resp (default=dataless)
- response_path: Instrument correction file(s) location (path relative to db.ini), defaults to ‘./inventory’, i.e. a subfolder in the current project folder. | All files in that folder will be parsed. (default=inventory)
- response_prefilt: Remove instrument correction pre-filter (0.005, 0.006, 30.0, 35.0) (default=(0.005, 0.006, 30.0, 35.0))
- preprocess_lowpass: Preprocessing Low-pass value in Hz [8.0] (default=8.0)
- preprocess_highpass: Preprocessing High-pass value in Hz [0.01] (default=0.01)
- preprocess_max_gap: Preprocessing maximum gap length that will be filled by interpolation [10.0] seconds (default=10.0) | new in 1.6
- preprocess_taper_length: Duration of the taper applied at the beginning and end of trace during the preprocessing, to allow highpassfiltering (default=20.0) | new in 1.6
- keep_all: Keep all cross-corr (length: corr_duration) [Y]/N (default=N)
- **keep_days**: Keep all daily cross-corr [Y]/N (default=Y)
- **stack_method**: Stack Method: Linear Mean or Phase Weighted Stack: [linear]/pws (default=linear)
- **pws_timegate**: If stack_method='pws', width of the smoothing in seconds : 10.0 (default=10.0)
- **pws_power**: If stack_method='pws', Power of the Weighting: 2.0 (default=2.0)
- **whitening**: Whiten Traces before cross-correlation: [A]ll (except for autocorr), [N]one, or only if [C]omponents are different: [A]/N/C (default=A) | new in 1.5
- **whitening_type**: Type of spectral whitening function to use: [B]rutal (amplitude to 1.0), divide spectrum by its [PSD]: [B]/PSD. WARNING: only works for compute_cc, not compute_cc_rot, where it will always be [B] (default=B) | new in 1.6
- **hpc**: Is MSNoise going to run on an HPC? Y/[N] (default=N) | new in 1.6

### Waveform Pre-processing

Pairs are first split and a station list is created. The database is then queried to get file paths. For each station, all files potentially containing data for the day are opened. The traces are then merged and split, to obtain the most continuous chunks possible. The different chunks are then demeaned, tapered and merged again to a 1-day long trace. If a chunk is not aligned on the sampling grid (that is, start at an integer times the sample spacing in s), the chunk is phase-shifted in the frequency domain. This requires tapering and fft/ifft. If the gap between two chunks is small, compared to `preprocess_max_gap`, the gap is filled with interpolated values. Larger gaps will not be filled with interpolated values.

Each 1-day long trace is then high-passed (at `preprocess_highpass` Hz), then if needed, low-passed (at `preprocess_lowpass` Hz) and decimated/downsampled. Decimation/Downsampling are configurable (`resampling_method`) and users are advised testing Decimate. One advantage of Downsampling over Decimation is that it is able to downsample the data by any factor, not only integer factors. Downsampling is achieved with the ObsPy Lanczos resampler which we tested against the old scikits.samplerate.

If configured, each 1-day long trace is corrected for its instrument response. Currently, only dataless seed and inventory XML are supported.
Preprocessing

Begin

Next Station in station list?

Get File Names from DB and read files

Select net + sta + component and merge

Sample Alignment?

bad

FFT → Align → ifft

good

Are there gaps?

yes

Merge small gaps
Keep large gaps

no

Remove too short individual traces

Demean, detrend & taper each trace

Highpass filter

Decimation or Resampling?

yes

Lowpass filter Decimate & Resample

no

Remove response?

yes

Remove instrument response

no

Next Station in station list?

yes

end
As from MSNoise 1.5, the preprocessing routine is separated from the compute_cc and can be used by plugins with their own parameters. The routine returns a Stream object containing all the traces for all the stations/components.

**Computing the Cross-Correlations**

**Processing using** msnoise compute_cc

**Todo:** We still need to describe the workflow in plain text, but the following graph should help you understand how the code is structured
Compute CC

begin

New CC job ?

Slide over time
Next Slide ?

Remove traces still containing gaps

No traces?
All traces too short ?

Detrend/ demean

Windowsing?

Taper 4%

Whitening?

Determine indexes in 2D data

Next filter?

Whitening

Band-Pass Filter

AC ?

Band-Pass Filter

CC ?

2D FFT[data]

SC ?

2D Whiten [whiten2]

Precompute energy

Correlator [myCorr2]

2D Whiten [whiten2]

Precompute energy

Next filter?

Keep all CCF?

Keep daily CCF?

New CC job ?

end
Processing using `msnoise compute_cc_rot`

Once all traces are preprocessed, station pairs are processed sequentially. If a component different from ZZ is to be computed, the traces are first rotated. This supposes the user has provided the station coordinates in the `station` table. The rotation is computed for Radial and Transverse components.

Then, for each `corr_duration` window in the signal, and for each filter configured in the database, the traces are clipped to `windsorizing` times the RMS (or 1-bit converted) and then whitened in the frequency domain (see whitening) between the frequency bounds. The whitening procedure can be skipped by setting the whitening configuration to `None`. The two other whitening modes are “[A]ll except for auto-correlation” or “Only if [C]omponents are differ-
ent”. This allows skipping the whitening when, for example, computing ZZ components for very close by stations (much closer than the wavelength sampled), leading to spatial autocorrelation issues.

When both traces are ready, the cross-correlation function is computed (see mycorr). The function returned contains data for time lags corresponding to maxlag in the acausal (negative lags) and causal (positive lags) parts.

### Saving Results (stacking the daily correlations)

If configured (setting keep all to ‘Y’), each corr duration CCF is saved to the hard disk in the output folder. By default, the keep days setting is set to True and so “N = 1 day / corr duration” CCF are stacked to produce a daily cross-correlation function, which is saved to the hard disk in the STACKS/001_DAYS folder.

**Note:** Currently, the keep-all data (every CCF) are not used by next steps.

If stack method is ‘linear’, then a simple mean CCF of all windows is saved as the daily CCF. On the other hand, if stack method is ‘pws’, then all the Phase Weighted Stack (PWS) is computed and saved as the daily CCF. The PWS is done in two steps: first the mean coherence between the instantaneous phases of all windows is calculated, and eventually serves a weighting factor on the mean. The smoothness of this weighting array is defined using the pws timegate parameter in the configuration. The weighting array is the power of the mean coherence array. If pws power is equal to 0, a linear stack is done (then it’s faster to do set stack method = ‘linear’). Usual value is 2.

**Warning:** PWS is largely untested, not cross-validated. It looks good, but that doesn’t mean a lot, does it? Use with Caution! And if you cross-validate it, please let us know!!


Once done, each job is marked “D”one in the database.

### Usage

To run this script:

```bash
$ msnoise compute_cc
```

This step also supports parallel processing/threading:

```bash
$ msnoise -t 4 compute_cc
```

will start 4 instances of the code (after 1 second delay to avoid database conflicts). This works both with SQLite and MySQL but be aware problems could occur with SQLite.

New in version 1.4: The Instrument Response removal & The Phase Weighted Stack & Parallel Processing
New in version 1.5: The Obspy Lanczos resampling method, gives similar results as the sci-kits.samplerate package, thus removing the requirement for it. This method is defined by default.

New in version 1.5: The preprocessing routine is separated from the compute_cc and can be called by external plugins.

New in version 1.6: The compute_cc has been completely rewritten to be much faster, taking advantage from 2D FFT computation and in-place array modifications. The standard compute_cc does process CC, AC and SC in the same code. Only if users need to compute R and/or T components, they will have to use the slower previous code, now called compute_cc_rot.

2.1.7 Stack

MSNoise is capable of using a reference function defined by absolute or relative dates span. For example, an absolute range could be “from 1 January 2010 to 31 December 2011” and a relative range could be “the last 200 days”. In the latter case, the REF will need to be exported at every run, meaning the following steps (MWCS and DTT) will be executed on the whole configured period. If the REF is defined between absolute dates, excluding “today”, the MWCS and DTT will only be calculated for new data (e.g. “yesterday” and “today”). The corresponding configuration bits are ref_begin and ref_end. In the future, we plan on allowing multiple references to be defined.

Only data for new/modified dates need to be exported. If any CC-job has been marked “Done” within the last day and triggered the creation of STACK jobs, the stacks will be calculated and a new MWCS job will be inserted in the database. For dates in the period of interest, the moving-window stack will only be exported if new/modified CCF is available. The export directory are “REF/” and “DAY%03i/” where %03i will be replaced by the number of days stacked together (DAYS_005 for a 5-days stack, e.g.).

Please note that within MSNoise, stacks are always inclusive of the time/day mentioned. For example, a 5-days stack on January 10, will contain cross-correlation functions computed for January 6, 7, 8, 9 AND 10! The graphical representation centered on a “January 10” tick might then display changes in the CCF that occurred on the 10th.

Moving-window stacks are configured using the mov_stack parameter in msnoise admin. If stack_method is ‘linear’, then a simple mean CFF of all daily is saved as the mov or ref CCF. On the other hand, if stack_method is ‘pws’, then all the Phase Weighted Stack (PWS) is computed and saved as the mov or ref CCF. The PWS is done in two steps: first the mean coherence between the instantaneous phases of all windows is calculated, and eventually serves a weighting factor on the mean. The smoothness of this weighting array is defined using the pws_timegate parameter in the configuration. The weighting array is the power of the mean coherence array. If pws_power is equal to 0, a linear stack is done (then it’s faster to do set stack_method = ‘linear’). Usual value is 2.

Warning: PWS is largely untested, not cross-validated. It looks good, but that doesn’t mean a lot, does it? Use with Caution! And if you cross-validate it, please let us know!!

Configuration Parameters

- **ref_begin**: Beginning or REF stacks. Can be absolute (2012-01-01) or relative (-100) days (default=1970-01-01)
- **ref_end**: End or REF stacks. Same as ref_begin (default=2021-01-01)
- **mov_stack**: Number of days to stack for the Moving-window stacks ([5]=[day-4:day]), can be a comma-separated list 1,2,5,10 (default=5)
- **stack_method**: Stack Method: Linear Mean or Phase Weighted Stack: [linear]/pws (default=linear) | new in 1.4
- **pws_timegate**: If stack_method='pws', width of the smoothing in seconds : 10.0 (default=10.0) | new in 1.4
- **pws_power**: If stack_method='pws', Power of the Weighting: 2.0 (default=2.0) | new in 1.4
- **hpc**: Is MSNoise going to run on an HPC? Y/[N] (default=N) | new in 1.6

Once done, each job is marked “D”one in the database and, unless hpc is Y, MWCS jobs are inserted/updated in the database.

Usage:

```bash
msnoise stack --help
```

Usage: [OPTIONS]

Stacks the [REF] or [MOV] windows. Computes the STACK jobs.

Options:

- `-r`, `--ref`  Compute the REF Stack
- `-m`, `--mov`  Compute the MOV Stacks
- `-s`, `--step` Compute the STEP Stacks
- `--help`  Show this message and exit.

For most users, the REF stack will need to be computed only once for specific dates and then, on routine basis, only compute the MOV stacks:

```bash
$ msnoise stack -r
$ msnoise reset STACK
$ msnoise stack -m
```

as for all other steps, this procedure can be run in parallel:

```bash
$ msnoise -t 4 stack -r
$ msnoise reset STACK
$ msnoise -t 4 stack -m
```

New in version 1.4: The Phase Weighted Stack.

New in version 1.6: The hpc parameter that can prevent the automatic creation of MWCS jobs. The REF and MOV stacks have been separated and need to be run independently.
2.1.8 Compute MWCS

**Warning:** if using only `mov_stack = 1`, no MWCS jobs is inserted in the database and consequently, no MWCS calculation will be done! FIX!

Following Clarke et al (2011), we apply the mwcs to study the relative dephasing between Moving-Window stacks (“Current”) and a Reference using Moving-Window Cross-Spectral analysis. The jobs “T”o do have been inserted in the database during the stack procedure.

**Filter Configuration Parameters**

- `mwcs_low`: The lower frequency bound of the linear regression done in MWCS (in Hz)
- `mwcs_high`: The upper frequency bound of the linear regression done in MWCS (in Hz)
- `mwcs_wlen`: Window length (in seconds) to perform MWCS
- `mwcs_step`: Step (in seconds) of the windowing procedure in MWCS
- `hpc`: Is MSNoise going to run on an HPC? Y/[N] (default=N) *new in 1.6*

In short, both time series are sliced in several overlapping windows and preprocessed. The similarity of the two time-series is assessed using the cross-coherence between energy densities in the frequency domain. The time delay between the two cross correlations is found in the unwrapped phase of the cross spectrum and is linearly proportional to frequency. This “Delay” for each window between two signals is the slope of a weighted linear regression (WLS) of the samples within the frequency band of interest.

For each filter, the frequency band can be configured using `mwcs_low` and `mwcs_high`, and the window and overlap lengths using `mwcs_wlen` and `mwcs_step`.

The output of this process is a table of delays measured at each window in the functions. The following is an example for lag times between -115 and -90. In this case, the window length was 10 seconds with an overlap of 5 seconds.

<table>
<thead>
<tr>
<th>LAG_TIME</th>
<th>DELAY</th>
<th>ERROR</th>
<th>MEAN</th>
<th>COHERENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.1500000000e+02</td>
<td>-1.4781146383e-01</td>
<td>5.3727119135e-02</td>
<td>2.7585243911e-01</td>
<td></td>
</tr>
<tr>
<td>-1.1000000000e+02</td>
<td>-6.8207526992e-02</td>
<td>2.0546644311e-02</td>
<td>3.1620999352e-01</td>
<td></td>
</tr>
<tr>
<td>-1.0500000000e+02</td>
<td>-1.0337029577e-01</td>
<td>8.6645155402e-03</td>
<td>4.2439269880e-01</td>
<td></td>
</tr>
<tr>
<td>-1.0000000000e+02</td>
<td>-2.8668775696e-02</td>
<td>6.2522215988e-03</td>
<td>5.7159849528e-01</td>
<td></td>
</tr>
<tr>
<td>-9.5000000000e+01</td>
<td>4.1803941008e-02</td>
<td>1.5102285789e-02</td>
<td>4.1238557789e-01</td>
<td></td>
</tr>
<tr>
<td>-9.0000000000e+01</td>
<td>4.8139400233e-02</td>
<td>3.2700657018e-02</td>
<td>3.0586187792e-01</td>
<td></td>
</tr>
</tbody>
</table>

This process is job-based, so it is possible to run several instances in parallel.

Once done, each job is marked “D”one in the database and, unless `hpc` is Y, DTT jobs are inserted/updated in the database.

To run this step:

```
$ msnoise compute_mwcs
```

This step also supports parallel processing/threading:
$ \text{msnoise} -t 4 \text{ compute_mwcs}$

will start 4 instances of the code (after 1 second delay to avoid database conflicts). This works both with SQLite and MySQL but be aware problems could occur with SQLite.

New in version 1.4: Parallel Processing

2.1.9 Compute $\text{dt/t}$

This code is responsible for the calculation of $\text{dt/t}$ using the result of the MWCS calculations.

Configuration Parameters

- $\text{dtt\_lag}$: How is the lag window defined [dynamic]/static (default=static)
- $\text{dtt\_v}$: If $\text{dtt\_lag}$=dynamic: what velocity to use to avoid ballistic waves [1.0]km/s (default=1.0)
- $\text{dtt\_minlag}$: If $\text{dtt\_lag}$=static: min lag time (default=5.0)
- $\text{dtt\_width}$: Width of the time lag window [30]s (default=30.0)
- $\text{dtt\_sides}$: Which sides to use [both]/left/right (default=both)
- $\text{dtt\_mincoh}$: Minimum coherence on dt measurement, MWCS points with values lower than that will not be used in the WLS (default=0.65)
- $\text{dtt\_maxerr}$: Maximum error on dt measurement, MWCS points with values larger than that will not be used in the WLS (default=0.1)
- $\text{dtt\_maxdt}$: Maximum dt values, MWCS points with values larger than that will not be used in the WLS (default=0.1)

The $\text{dt/t}$ is determined as the slope of the delays vs time lags. The slope is calculated a weighted linear regression (WLS) through selected points.

1. The selection of points is first based on the time lag criteria. The minimum time lag can either be defined absolutely or dynamically. When $\text{dtt\_lag}$ is set to “dynamic” in the database, the inter-station distance is used to determine the minimum time lag. This lag is calculated from the distance and a velocity configured ($\text{dtt\_v}$). The velocity is determined by the user so that the minlag doesn’t include the ballistic waves. For example, if ballistic waves are visible with a velocity of 2 km/s, one could configure $\text{dtt\_v}$=1.0. This way, if stations are located 15 km apart, the minimum lag time will be set to 15 s. The $\text{dtt\_width}$ determines the width of the lag window used. A value of 30.0 means the process will use time lags between 15 and 45 s in the example above, on both sides if configured ($\text{dtt\_sides}$), or only causal or acausal parts of the CCF. The following figure shows the static time lags of $\text{dtt\_width} = 40s$ starting at $\text{dtt\_minlag} = 10s$ and the dynamic time lags for a $\text{dtt\_v} = 1.0$ km/s for the Piton de La Fournaise network (including stations not on the volcano).

\textbf{Note:} It seems obvious that these parameters are frequency-dependent, but they are currently common for all filters!
Warning: In order to use the dynamic time lags, one has to provide the station coordinates.
2. Using example values above, we chose to use only 15-45 s coda part of the signal, neglecting direct waves in the 0-15 seconds range. We then select data which match three other thresholds: dtt_mincoh, dtt_maxerr and dtt_maxdt.

Each of the 4 left subplot of this figure shows a colormapper matrix of which each row corresponds to the data of 1 station pair and each column corresponds to different time lags. The cells are then colored using, from left to right: Delays, Errors, Phase Coherence and Data Selection.

Once data (cells) have been selected, they are analyzed two times: first using a WLS that is forced to pass the origin (0,0) and second when a constant is added to allow for the WLS to be offset from the origin. For each value, the error is computed and stored. M0 and EM0 are the slope and its error for the first WLS, and M, EM together with A and EA are the slope, its error, the constant and its error for the second WLS. The output of this calculation is a table, with one row for each station pair.

<table>
<thead>
<tr>
<th>Date, PAIRS</th>
<th>A</th>
<th>EA</th>
<th>EM</th>
<th>EMO</th>
<th>M</th>
<th>M0</th>
<th>Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-06, BE_HOU→BE_MEM</td>
<td>-0.1683728</td>
<td>0.0526606</td>
<td>0.00208377</td>
<td>0.00096521</td>
<td>0.00692021</td>
<td>0.00037757</td>
<td>BE_GES</td>
</tr>
<tr>
<td>2013-01-06, BE_MEM→BE_RCHB</td>
<td>0.1007472</td>
<td>0.0144648</td>
<td>0.00179566</td>
<td>0.00454172</td>
<td>-0.00145738</td>
<td>0.00741478</td>
<td>BE_GES</td>
</tr>
<tr>
<td>2013-01-06, BE_RCHB→BE_SKQ</td>
<td>-0.0556811</td>
<td>0.0098926</td>
<td>0.00057839</td>
<td>-0.00108102</td>
<td>-0.00328965</td>
<td>-0.00136075</td>
<td>BE_GES</td>
</tr>
<tr>
<td>2013-01-06, BE_SKQ→BE_STI</td>
<td>0.0150866</td>
<td>0.0202243</td>
<td>0.00096543</td>
<td>0.00089832</td>
<td>0.00083714</td>
<td>0.00104507</td>
<td>BE_GES</td>
</tr>
<tr>
<td>2013-01-06, BE_STI→BE_UCC</td>
<td>0.0268309</td>
<td>0.0328997</td>
<td>0.00153137</td>
<td>0.00150261</td>
<td>0.00302331</td>
<td>0.00302451</td>
<td>BE_GES</td>
</tr>
<tr>
<td>2013-01-06, BE_UCC→BE_MEM</td>
<td>-0.0121293</td>
<td>0.0043351</td>
<td>0.00039019</td>
<td>0.00041347</td>
<td>0.00025836</td>
<td>-0.00042709</td>
<td>BE_HOU</td>
</tr>
<tr>
<td>2013-01-06, BE_MEM→BE_RCHB</td>
<td>0.1076247</td>
<td>0.0188662</td>
<td>0.00076824</td>
<td>0.00216383</td>
<td>-0.0030791</td>
<td>0.00112692</td>
<td>BE_HOU</td>
</tr>
<tr>
<td>2013-01-06, BE_RCHB→BE_SKQ</td>
<td>-0.0468485</td>
<td>0.0194492</td>
<td>0.00069968</td>
<td>0.00078207</td>
<td>-0.00066133</td>
<td>0.00027102</td>
<td>BE_HOU</td>
</tr>
<tr>
<td>2013-01-06, BE_SKQ→BE_STI</td>
<td>0.0203057</td>
<td>0.0161316</td>
<td>0.00131522</td>
<td>0.00131182</td>
<td>0.00051626</td>
<td>-3.10306611</td>
<td>BE_HOU</td>
</tr>
<tr>
<td>2013-01-06, BE_STI→BE_UCC</td>
<td>0.0022588</td>
<td>0.0037141</td>
<td>0.00010340</td>
<td>9.1996e-05</td>
<td>0.00073635</td>
<td>0.00076238</td>
<td>ALL</td>
</tr>
</tbody>
</table>

To run this script:
msnoise compute_dtt

### Grouping Station Pairs

Although not clearly visible on the figure above, the very last row of the matrix doesn’t contain information about one station pair, but contains a weighted mean of all delays (from all pairs) for each time lag. For each time lag, delays from each pair is taken into account if it satisfies the same criteria as for the individual data selection. Once the last row (the ALL line) has been calculated, it goes through the normal process of the double WLS and is saved to the output file, as visible above. In the future, MSNoise will be able to treat as many groups as the user want, allowing, e.g. a “crater” and a “slopes” groups.

### Forcing vs No Forcing through Origin

The reason for allowing the WLS to cross the axis elsewhere than on (0,0) is, for example, to study the potential clock drifts or noise source position variations. By default, the `msnoise plot dvv` plot shows the results of the `Not Forced WLS`.

### Mean of All Pairs vs Mean Pair

**Warning:** the ALL pair is still calculated and output in the DTT files, but its output is no longer displayed on the graphs. *new in 1.6.*

The dt/t calculated using the mean pair (ALL, in red on subplots 4 and 5) and by calculating the weighted mean of the dt/t of all pairs (in green) don’t show a significant difference. The standard deviation around the latter is more spread than on the former, but this has to be investigated.
3.1 Plotting

MSNoise comes with some default plotting tools.

All plotting commands accept the `-- outfile` argument. If provided, the figure will be saved to the disk. Names can be explicit, or tell the code to generate the filename automatically (using the `?` question mark), for example:

```bash
# automatic naming, save to PNG
msnoise plot dvv -o ?.png

# automatic naming, save to PDF
msnoise plot dvv -o ?.pdf

# explicit naming, save to JPG
msnoise plot dvv -o mydvv.jpg
```

- Customizing Plots (page 41)
- Data Availability Plot (page 42)
- Station Map (page 43)
- Interferogram Plot (page 43)
- CCF vs Time (page 44)
- CCF’s spectrum vs Time (page 47)
- MWCS Plot (page 50)
- Distance Plot (page 51)
- dv/v Plot (page 52)
- dt/t Plot (page 53)

3.1.1 Customizing Plots

All plots commands can be overridden using a `-c` argument in front of the plot command!!

Examples:
To make this work, one has to copy the plot script from the msnoise install directory to the project directory (where your db.ini file is located, then edit it to one's desires. The first thing to edit in the code is the import of the MSNoise API (page 66):

```
from ..api import *
```
to
```
from msnoise.api import *
```
and it should work.

New in version 1.4.

### 3.1.2 Data Availability Plot

Plots the data availability, as contained in the database. Every day which has a least some data will be coloured in red. Days with no data remain blank.

```
msnoise plot data_availability --help
```

Usage: `[OPTIONS]`

Plots the Data Availability vs time

Options:
- `-s, --show BOOLEAN` Show interactively?
- `-o, --outfile TEXT` Output filename (default auto)
- `--help` Show this message and exit.

Example:
```
msnoise plot data_availability:
```
3.1.3 Station Map

3.1.4 Interferogram Plot

This plot shows the cross-correlation functions (CCF) vs time in a very similar manner as on the `ccftime` plot above, but shows an image instead of wiggles. The parameters allow to plot the daily or the mov-stacked CCF. Filters and components are selectable too. Passing `--refilter` allows to bandpass filter CCFs before plotting (new in 1.5).

```
msnoise plot interferogram --help
```

Usage: `[OPTIONS] STA1 STA2 [EXTRA_ARGS]...`

```
Plots the interferogram between sta1 and sta2 (parses the CCFs)
```

```
STA1 and STA2 must be provided with this format: NET.STA !
```

Options:

```
-f, --filterid INTEGER  Filter ID
-c, --comp TEXT         Components (ZZ, ZR,...)
-m, --mov_stack INTEGER  Mov Stack to read from disk
-s, --show BOOLEAN      Show interactively?
-o, --outfile TEXT      Output filename (?=auto)
-r, --refilter TEXT     Refilter CCFs before plotting (e.g. 4:8 for filtering CCFs between 4.0 and 8.0 Hz. This will
```

(continues on next page)
Example:

```
msnoise plot interferogram YA.UV06 YA.UV11 -m5 will plot the ZZ component (default),
filter 1 (default) and mov_stack 5:
```

![Graph showing cross-correlation functions (CCF) vs time]

### 3.1.5 CCF vs Time

This plot shows the cross-correlation functions (CCF) vs time. The parameters allow to plot
the daily or the mov-stacked CCF. Filters and components are selectable too. The `--ampli`
argument allows to increase the vertical scale of the CCFs. The `--seismic` shows the up-going
wiggles with a black-filled background (very heavy!). Passing `--refilter` allows to bandpass
filter CCFs before plotting (new in 1.5).

```
msnoise plot ccftime --help
```

Usage: `[OPTIONS] STA1 STA2 [EXTRA_ARGS]...`

Plots the ccf vs time between sta1 and sta2

STA1 and STA2 must be provided with this format: NET.STA !
Example:

```
msnoise plot ccftime YA.UV06 YA.UV11 will plot all defaults:
```

For zooming in the CCFs:

```
msnoise plot ccftime YA.UV05 YA.UV11 --xlim=-10,10 --ampli=15:
```
It is sometimes useful to refilter the CCFs on the fly:

```bash
msnoise plot ccftime YA.UV05 YA.UV11 -r 0.5:1.0:
```
This plot shows the cross-correlation functions’ spectrum vs time. The parameters allow to plot the daily or the mov-stacked CCF. Filters and components are selectable too. The --ampli argument allows to increase the vertical scale of the CCFs. Passing --refilter allows to band-pass filter CCFs before computing the FFT and plotting. Passing --startdate and --enddate parameters allows to specify which period of data should be plotted. By default the plot uses dates determined in database.

```
msnoise plot spectime --help
```

Usage: [OPTIONS] STA1 STA2 [EXTRA_ARGS]...

Plots the ccf's spectrum vs time between sta1 and sta2

STA1 and STA2 must be provided with this format: NET.STA!

Options:

- `-f, --filterid INTEGER` Filter ID
- `-c, --comp TEXT` Components (ZZ, ZR,...)
- `-m, --mov_stack INTEGER` Mov Stack to read from disk
- `-a, --ampli FLOAT` Amplification
- `-s, --show BOOLEAN` Show interactively?
- `-o, --outfile TEXT` Output filename (?=auto)
- `-r, --refilter TEXT` Refilter CCFs before plotting (e.g. 4:8 for filtering CCFs between 4.0 and 8.0 Hz. This will

(continues on next page)
Example:

\texttt{msnoise plot spectime YA.UV05 YA.UV11} will plot all defaults:

![Plot of YA.UV05 and YA.UV11](image)

Zooming in the X-axis and playing with the amplitude:

\texttt{msnoise plot spectime YA.UV05 YA.UV11 --xlim=0.08,1.1 --ampli=10}

And refiltering to enhance high frequency content:

```bash
msnoise plot spectime YA.UV05 YA.UV11 --xlim=0.5,1.1 --ampli=10 -r0.7:1.0
```
3.1.7 MWCS Plot

This plot shows the result of the MWCS calculations in two superposed images. One is the dt calculated vs time lag and the other one is the coherence. The image is constructed by horizontally stacking the MWCS of different days. The two right panels show the mean and standard deviation per time lag of the whole image. The selected time lags for the dt/t calculation are presented with green horizontal lines, and the minimum coherence or the maximum dt are in red.

The `filterid`, `comp` and `mov_stack` allow filtering the data used.

```
msnoise plot mwcs --help
```

```
Usage: [OPTIONS] STA1 STA2

Plots the mwcs results between sta1 and sta2 (parses the CCFs)

STA1 and STA2 must be provided with this format: NET.STA !

Options:
-f, --filterid INTEGER  Filter ID
-c, --comp TEXT  Components (ZZ, ZR,...)
-m, --mov_stack INTEGER  Mov Stack to read from disk
-s, --show BOOLEAN  Show interactively?
-o, --outfile TEXT  Output filename (?=auto)
--help  Show this message and exit.
```
Example:

msnoise plot mwcs ID.KWUI ID.POSI -m 3 will plot all defaults with the mov_stack = 3:

![Distance Plot Example]

### 3.1.8 Distance Plot

Plots the REF stacks vs interstation distance. This could help deciding which parameters to use in the dt/t calculation step. Passing --refilter allows to bandpass filter CCFs before plotting (new in 1.5). It is also possible to only draw CCFs for pairs including one station by passing --virtual-pair followed by the desired NET.STA (new in 1.5).

```
msnoise plot distance --help
```

**Usage:** `[OPTIONS] [EXTRA_ARGS]...`

Plots the REFs of all pairs vs distance

**Options:**

- `-f, --filterid INTEGER` Filter ID
- `-c, --comp TEXT` Components (ZZ, ZR, ...)
- `-a, --ampli FLOAT` Amplification
- `-s, --show BOOLEAN` Show interactively?
- `-o, --outfile TEXT` Output filename (?=auto)
- `-r, --refilter TEXT` Refilter CCFs before plotting (e.g. 4:8 for filtering CCFs between 4.0 and 8.0 Hz. This will update the plot title.
- `--virtual-source TEXT` Use only pairs including this station. Format must

(continues on next page)
Example:

msnoise plot distance will plot all defaults:

3.1.9 dv/v Plot

This plot shows the final output of MSNoise.

msnoise plot dvv --help

Usage: [OPTIONS]

Plots the dv/v (parses the dt/t results)

Individual pairs can be plotted extra using the -p flag one or more times.

Example: msnoise plot dvv -p ID_KWUI_ID_POSI

Example: msnoise plot dvv -p ID_KWUI_ID_POSI -p ID_KWUI_ID_TRWI

Remember to order stations alphabetically!

Options:
-f, --filterid INTEGER Filter ID
-c, --comp TEXT Components (ZZ, ZR,...)
Example:

```
msnoise plot dvv will plot all defaults:
```

```
msnoise plot dtt --help

Usage: [OPTIONS] STA1 STA2 DAY

Plots a graph of dt against t

STA1 and STA2 must be provided with this format: NET.STA
```

3.1.10 dt/t Plot

This plots dt (delay time) against t (time lag). It shows the results from the MWCS step, plus the calculated regression lines M0 and M. The errors in the regression lines are also plotted as fainter lines. The time lags used to calculate the regression are shown in blue.
DAY must be provided in the ISO format: YYYY-MM-DD

Options:
- `f, --filterid INTEGER`  Filter ID
- `c, --comp TEXT`  Components (ZZ, ZR,...)
- `m, --mov_stack INTEGER`  Mov Stack to read from disk
- `s, --show BOOLEAN`  Show interactively?
- `o, --outfile TEXT`  Output filename (=?auto)
- `--help`  Show this message and exit.

Example

```
msnoise plot dtt Z7.HRIM Z7.LIND 2014-08-10 -f 14 -m 20 will plot:
```

```
Z7.HRIM-Z7.LIND f14 m20 2014-08-10
```

New in version 1.4: (Thanks to C.G. Donaldson)
4.1 How To’s

4.1.1 Run the simplest MSNoise run ever

This recipe is a kind of “let’s check this data rapidly”:

```bash
msnoise db init --tech 1
msnoise config set startdate=2019-01-01
msnoise config set enddate=2019-02-01
msnoise config set overlap=0.5
msnoise config set mov_stack=1,5,10
msnoise scan_archive --path /path/to/archive --recursively
msnoise populate --fromDA
msnoise new_jobs --init
msnoise admin
# add 1 filter in the Filter table
# or
msnoise db execute "insert into filters (ref, low, mwcs_low, high, mwcs_high, rms_threshold, mwcs_wlen, mwcs_step, used) values (1, 0.1, 0.1, 1.0, 1.0, 0.0, 12.0, 4.0, 1)"
msnoise compute_cc
msnoise stack -r
msnoise reset STACK
msnoise stack -m
msnoise compute_mwcs
msnoise compute_dtt
msnoise plot dvv
```

4.1.2 Run MSNoise using lots of cores on a HPC

**Avoid Database I/O by using the hpc flag**

With MSNoise 1.6, most of the API calls have been cleaned from calling the database, for example the `def stack()` called a SELECT on the database for each call, which is useless as configuration parameters are not supposed to change during the execution of the code. This modification allows running MSNoise on an HPC infrastructure with a remote central MySQL database.
The new configuration parameter `hpc` is used for flagging if MSNoise is running High Performance. If True, the jobs processed at each step are marked Done when finished, but the next jobtype according to the workflow is not created. This removes a lot of select/update/insert actions on the database and makes the whole much faster (one INSERT instead of tons of SELECT/UPDATE/INSERT).

Commands and actions with `hpc = N`:
- `msnoise new jobs`: creates the CC jobs
- `msnoise compute cc`: processes the CC jobs and creates the STACK jobs
- `msnoise stack -m`: processes the STACK jobs and creates the MWCS jobs
- etc...

Commands and actions with `hpc = Y`:
- `msnoise new jobs`: creates the CC jobs
- `msnoise compute cc`: processes the CC jobs
- `msnoise new jobs --hpc CC:STACK`: creates the STACK jobs based on the CC jobs marked “D”one
- `msnoise stack -m`: processes the STACK jobs
- `msnoise new jobs --hpc STACK:MWCS`: creates the MWCS jobs based on the STACK jobs marked “D”one
- etc...

**Set up the HPC**

To avoid having to rewrite MSNoise for using techniques relying on MPI or other parallel computing tools, I decided to go “simple”, and this actually works. The only limitation of the following is that you need to have a strong MySQL server machine that accepts hundreds or thousands of connections. In my case, the MySQL server is running on a computing blade, and its my.cnf is configured to allow 1000 users/connections, and to listen on all its IPs.

The easiest set up (maybe not your sysadmin’s preferred, please check), is to
- install miniconda on your home directory and make miniconda’s python executable your default python (I add the paths to .profile).
- Then install the requirements and finally MSNoise.
- As usual, create a project folder and `msnoise db init` there, choose MySQL and provide the hostname of the machine running the MySQL server.

At that point, your project is ready. I usually request an interactive node on the HPC for doing the `msnoise populate` and `msnoise scan archive`. Our jobs scheduler is PBS, so this command

```
qsub -I -l walltime=02:00:00 -l select=1:ncpus=16:mem=1g
```

requests an Interactive node with 16 cpus, 1GB ram, for 2 hours. Once connected, check that the python version is correct (or source .profile again). Because we requested 16 cores, we can `msnoise -t 16 scan_archive --init`. 

Depending on the server configuration, you can maybe run the `msnoise admin` on the login node, and access it via its hostname:5000 in your browser. If not, the easiest way to set up the config is running `msnoise config set <parameter>=<value>` from the console. To add filters, do it either:

- in the Admin
- using MySQL workbench connected to your MySQL server
- using such commands `msnoise db execute "insert into filters (ref, low, mwcs_low, high, mwcs_high, rms_threshold, mwcs_wlen, mwcs_step, used) values (1, 0.1, 0.1, 1.0, 1.0, 0.0, 12.0, 4.0, 1)"
- using `msnoise db dump`, edit the filter table in CSV format, then `msnoise db import filters --force`

Once done, the project is set up and should run. Again, test if all goes OK in an interactive node.

To run on N cores in parallel, we have the advantage that, e.g. for CC jobs, the day-jobs are independent. We can thus request an “Array” of single cores, which is usually quite easy to get on HPCs (most users run heavily parallel codes and request large number of “connected” cores, while we can run “shared”).

The job file in my PBS case looks like this for computing the CC:

```
#!/bin/bash
#PBS -N MSNoise_PDF_CC
#PBS -l walltime=01:00:00
#PBS -l select=1:ncpus=1:mem=1g
#PBS -l place=shared
#PBS -J 1-400
cd /scratch-a/thomas/2019_PDF
source /space/hpc-home/thomas/.profile
msnoise compute_cc2
```

This requests 400 cores with 1GB of RAM. The content of my .profile file contains:

```
# added by Miniconda3 installer
export PATH="/home/thomas/miniconda3/bin:$PATH"
export MPLBACKEND="Agg"
```

The last line is important as nodes are usually “head-less” and matplotlib and packages relating to it would fail if they expect a gui-capable system.

For submitting this job, run `qsub qc.job`. The process usually routes stdout and stderr to files in the current directory, make sure to check them if jobs seem to have failed. If all goes well, calling `msnoise info -j` repeatedly from the login or interactive node’s console should show the evolution of Todo, In Progress and Done jobs.

**Note:** HPC experts are welcome to suggest, comment, etc... It’s a quick’n’dirty solution, but it works for me!
4.1.3 Reprocess data

When starting to use MSNoise, one will most probably need to re-run different parts of the Workflow more than one time. By default, MSNoise is designed to only process “what’s new”, which is antagonistic to what is wanted. Hereafter, we present cases that will cover most of the re-run techniques:

When adding a new filter

If new filter are added to the filters list in the Configurator, one has to reprocess all CC jobs, but not for filters already existing. The recipe is:

- Add a new filter, be sure to mark ‘used’=1
- Set all other filters ‘used’ value to 0
- Redefine the flag of the CC jobs, from ‘D’one to ‘T’odo with the following:
  - Run `msnoise reset CC --all`
  - Run `msnoise compute cc`
  - Run next commands if needed (stack, mwcs, dtt)
- Set back the other filters ‘used’ value to 1

The compute cc will only compute the CC’s for the new filter(s) and output the results in the STACKS/ folder, in a sub-folder named by a formatted integer from the filter ID. For example: STACKS/01 for ‘filter id’=1, STACKS/02 for ‘filter id’=2, etc.

When changing the REF

When changing the REF (`ref.begin` and `ref.end`), the REF stack has to be re-computed:

```
msnoise reset STACK --all
msnoise stack -r
```

The REF will then be re-output, and you probably should reset the MWCS jobs to recompute daily correlations against this new ref:

```
msnoise reset MWCS --all
msnoise compute_mwcs
```

When changing the MWCS parameters

If the MWCS parameters are changed in the database, all MWCS jobs need to be reprocessed:

```
msnoise reset MWCS --all
msnoise compute_mwcs
```

should do the trick.
When changing the dt/t parameters

```bash
msnoise reset DTT --all
msnoise compute_dtt
```

Recompute only the specific days

You want to recompute CC jobs after a certain date only, for whatever reason:

```bash
msnoise reset CC --rule="day>='2019-01-01'"
```

SQL experts can also use the `msnoise db execute` command (with caution!):

```bash
msnoise db execute "update jobs set flag='T' where jobtype='CC' and day>='2019-01-01"
```

If you want to only reprocess one day:

```bash
msnoise reset CC --rule="day='2019-01-15'"
```

4.1.4 Define one’s own data structure of the waveform archive

The `data_structure.py` file contains the known data archive formats. If another data format needs to be defined, it will be done in the `custom.py` file in the current project folder:

See also:

Check the “Populate Station Table” step in the Populate Station Table (page 23).

4.1.5 How to have MSNoise work with 2+ data structures at the same time

In this case, the easiest solution is to scan the archive(s) with the “Lazy Mode”:

```bash
msnoise scan_archive --path /path/to/archive1/ --recursively
msnoise scan_archive --path /path/to/archive2/ --recursively
```

etc.

Remember to either manually fill in the station table, or

```bash
msnoise populate --fromDA
```

4.1.6 How to duplicate/dump the MSNoise configuration

To export all tables of the current database, run

```bash
msnoise db dump
```

This will create as many CSV files as there are tables in the database.

Then, on a new location, init a new msnoise project and import the tables one by one:
4.1.7 Testing the Dependencies

Once installed, you should be able to import the python packages in a python console. MSNoise comes with a little script called `bugreport.py` that can be useful to check if you have all the required packages (+ some extras).

The usage is such:

```
$ msnoise bugreport -h
```

```
usage: msnoise bugreport [-h] [-s] [-m] [-e] [-a]

Helps determining what didn't work

optional arguments:
  -h, --help    show this help message and exit
  -s, --sys     Outputs System info
  -m, --modules Outputs Python Modules Presence/Version
  -e, --env     Outputs System Environment Variables
  -a, --all     Outputs all of the above
```

On my Windows machine, the execution of

```
$ msnoise bugreport -s -m
```

results in:

```
************* Computer Report *************
----------------+SYSTEM+-------------------
Windows          PC1577-as
10                10.0.17134
AMD64            Intel64 Family 6 Model 158 Stepping 9, GenuineIntel
----------------+PYTHON+-------------------
Python:3.7.3      packaged by conda-forge (default, Jul  1 2019, 22:01:29) [MSC v.1900_64 bit (AMD64)]
This script is at d:\pythonforsource\msnoise_stack\msnoise\msnoise\bugreport.py
----------------+MODULES+-------------------
Required:
[X] setuptools: 41.2.0
[X] numpy: 1.15.4
```

(continues on next page)
Only necessary if you plan to build the doc locally:
[X] sphinx: 2.2.0
[X] sphinx_bootstrap_theme: 0.7.1

Graphical Backends: (at least one is required)
[ ] wx: not found
[ ] PyQt: not found
[ ] PyQt4: not found
[X] PyQt5: present (no version)
[ ] PySide: not found

Not required, just checking:
[X] json: 2.0.9
[X] psutil: 5.6.3
[ ] reportlab: not found
[ ] configobj: not found
[X] pkg_resources: present (no version)
[ ] paramiko: not found
[X] ctypes: 1.1.0
[X] pyparsing: 2.4.2
[X] distutils: 3.7.3
[X] IPython: 7.7.0
[ ] vtk: not found
[ ] enable: not found
[ ] traitsui: not found
[ ] traits: not found
[ ] scikits.samplerate: not found

The [X] marks the presence of the module. In the case above, PyQt4 is missing, but that’s not a problem because PyQt5 is present. The “not-required” packages are checked for information, those packages can be useful for reporting / hacking / rendering the data.

4.2 Interaction Examples & Gallery

The following examples are meant to show you how to interact with MSNoise using its API, thus avoiding having to dive into the folder structure.

Users should try examples while checking the MSNoise API (page 66). (application programming interface) for understanding the calls to different functions.
In a nutshell, all examples start with the following Python code:

```python
from msnoise.api import db
db = connect()
```

This, if run in an MSNoise project folder (= a folder where you have already run `msnoise db init`), will provide a `Session` object, connected to the database.

**Note:** Click [here](page 63) to download the full example code

### 4.2.1 Plot a Reference CCF

![Reference Function](image)

```
# The following two lines are only needed for building this documentation
# Delete them and run the code in your project folder.

import os
if "SPHINX_DOC_BUILD" in os.environ:
    os.chdir(r"C:\tmp\msnoise_doc_project")

import matplotlib
matplotlib.use("agg")

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

(continues on next page)
import pandas as pd
from pandas.plotting import register_matplotlib_converters

register_matplotlib_converters()

plt.style.use("ggplot")

from msnoise.api import connect, get_results, build_movstack_datelist, get_params,
→ get_t_axis

# connect to the database
db = connect()

# Obtain a list of dates between `start_date` and `enddate`
start, end, datelist = build_movstack_datelist(db)

# Get the list of parameters from the DB:
params = get_params(db)

# Get the time axis for plotting the CCF:
taxis = get_t_axis(db)

# Get the results for two station, filter id=1, ZZ component, mov_stack=1 and stack
→ the results:
n, ccf = get_results(db, "YA_UV05", "YA_UV12", 1, "ZZ", datelist, 1, format="stack",
→ params=params)

plt.figure()
plt.plot(taxis, ccf)
plt.title("Reference Function")
plt.xlabel("Lag Time (s)")
plt.ylabel("Amplitude")

#EOF

Total running time of the script: ( 0 minutes 0.926 seconds)

Note: Click here (page 66) to download the full example code

4.2.2 Plot an interferogram

import os
if "SPHINX_DOC_BUILD" in os.environ:
    os.chdir(r"C:\tmp\msnoise_doc_project")

import matplotlib
matplotlib.use("agg")

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

4.2. Interaction Examples & Gallery 63
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
plt.style.use("ggplot")

from msnoise.api import connect, get_results, build_movstack_datelist, get_params,
get_t_axis

# connect to the database
db = connect()

# Obtain a list of dates between `start_date` and `enddate`
start, end, datelist = build_movstack_datelist(db)

# Get the list of parameters from the DB:
params = get_params(db)

# Get the time axis for plotting the CCF:
taxis = get_t_axis(db)

# Get the results for two station, filter id=1, ZZ component, mov_stack=1 and the
results as a 2D array:
n, ccfs = get_results(db, "YA_UV05", "YA_UV12", 1, "ZZ", datelist, 1, format="matrix",
params=params)

# Convert to a pandas DataFrame object for convenience, and drop empty rows:
df = pd.DataFrame(ccfs, index=pd.DatetimeIndex(datelist), columns=taxis)
df = df.dropna()

# Define the 99% percentile of the data, for visualisation purposes:
clim = df.mean(axis="index").quantile(0.99)

fig, ax = plt.subplots()
plt.pcolormesh(df.columns, df.index.to_pydatetime(), df.values,
vmin=-clim, vmax=clim, rasterized=True)
plt.colorbar()
plt.title("Interferogram")
plt.xlabel("Lag Time (s)")
plt.ylim(df.index[0], df.index[-1])
plt.xlim(df.columns[0], df.columns[-1])
plt.subplots_adjust(left=0.15)
Running a simple moving window average can be done with pandas’s functions:

```python
smooth = df.rolling(5).mean()

fig, ax = plt.subplots()
plt.pcolormesh(smooth.columns, smooth.index.to_pydatetime(), smooth.values, 
              vmin=-clim, vmax=clim, rasterized=True)
plt.colorbar()
plt.title("Interferogram (smoothed over 5 days)")
plt.xlabel("Lag Time (s)")
plt.ylim(smooth.index[0], smooth.index[-1])
plt.xlim(smooth.columns[0], smooth.columns[-1])
plt.subplots_adjust(left=0.15)
plt.show()
```

#EOF
Out:

D:\PythonForSource\MSNoise_Stack\MSNoise\examples\plot_interferogram.py:75:
UserWarning: Matplotlib is currently using agg, which is a non-GUI backend, so
cannot show the figure.
plt.show()

Total running time of the script: ( 0 minutes 1.870 seconds)

4.3 MSNoise API

msnoise.api.get_logger(name, loglevel=None, with_pid=False)
    Returns the current configured logger or configure a new one.

msnoise.api.get_engine(inifile=None)
    Returns the a SQLAlchemy Engine

    Parameters
    inifile (str) – The path to the db.ini file to use. Defaults to
    os.getcwd() + db.ini

    Return type
    sqlalchemy.engine.Engine

    Returns
    An Engine Object

msnoise.api.connect(inifile=None)
    Establishes a connection to the database and returns a Session object.
Parameters  
inifile (string) – The path to the db.ini file to use. Defaults to os.cwd() + db.ini

Return type  
sqlalchemy.orm.session.Session

Returns  
A Session object, needed for many of the other API methods.

msnoise.api.create_database_inifile(tech, hostname, database, username, password, prefix=“”)

Creates the db.ini file based on supplied parameters.

Parameters

• tech (int) – The database technology used: 1=sqlite 2=mysql
• hostname (string) – The hostname of the server (if tech=2) or the name of the sqlite file if tech=1
• database (string) – The database name
• username (string) – The user name
• prefix (string) – The prefix to use for all tables
• password (string) – The password of user

Returns  
None

msnoise.api.read_db_inifile(inifile=None)

Reads the parameters from the db.ini file.

Parameters  
inifile (string) – The path to the db.ini file to use. Defaults to os.cwd() + db.ini

Return type  
tuple

Returns  
technology, hostname, database, username, password

msnoise.api.get_config(session, name=None, isbool=False, plugin=None)

Get the value of one or all config bits from the database.

Parameters

• session (sqlalchemy.orm.session.Session) – A Session object, as obtained by connect() (page 66)
• name (str) – The name of the config bit to get. If omitted, a dictionary with all config items will be returned
• isbool (bool) – if True, returns True/False for config name. Defaults to False
• plugin (str) – if provided, gives the name of the Plugin config to use. E.g. if “Amazing” is provided, MSNoise will try to load the “AmazingConfig” entry point. See Extending MSNoise with Plugins (page 82) for details.

Return type  
str, bool or dict

Returns  
the value for name or a dict of all config values

msnoise.api.update_config(session, name, value, plugin=None)

Update one config bit in the database.
Parameters

- `session`: A `Session` object, as obtained by `connect()` (page 66)
- `name`: The name of the config bit to set.
- `value`: The value of parameter `name`. Can also be NULL if you don’t want to use this particular parameter.
- `plugin`: if provided, gives the name of the Plugin config to use. E.g. if “Amazing” is provided, MSNoise will try to load the “AmazingConfig” entry point. See `Extending MSNoise with Plugins` (page 82) for details.

`msnoise.api.get_params(session)`
Get config parameters from the database.

**Parameters**

- `session`: A `Session` object, as obtained by `connect()` (page 66)

**Returns**
a `Param` class containing the parameters

`msnoise.api.get_filters(session, all=False)`
Get Filters from the database.

**Parameters**

- `session`: A `Session` object, as obtained by `connect()` (page 66)
- `all`: Returns all filters from the database if True, or only filters where `used = 1` if False (default)

**Return type**
list of `Filter`

**Returns**
a list of `Filter`

`msnoise.api.update_filter(session, ref, low, mwcs_low, high, mwcs_high, rms_threshold, mwcs_wlen, mwcs_step, used)`
Updates or Insert a new Filter in the database.

**See also:**
`msnoise.msnoise_table_def.declare_tables.Filter`

**Parameters**

- `session`: A `Session` object, as obtained by `connect()` (page 66)
- `ref`: The id of the Filter in the database
- `low`: The lower frequency bound of the Whiten function (in Hz)
- `high`: The upper frequency bound of the Whiten function (in Hz)
- `rms_threshold`: Not used anymore
- `mwcs_wlen`: Window length (in seconds) to perform MWCS
• `mwcs_step (float)` – Step (in seconds) of the windowing procedure in MWCS
• `used (bool)` – Is the filter activated for the processing

`msnoise.api.get_networks(session, all=False)`
Get Networks from the database.

**Parameters**

- `session (sqlalchemy.orm.session.Session)` – A Session object, as obtained by `connect()` (page 66)
- `all (bool)` – Returns all networks from the database if True, or only networks at least one station has `used` = 1 if False (default)

**Return type** list of str

**Returns** a list of network codes

`msnoise.api.get_stations(session, all=False, net=None)`
Get Stations from the database.

**Parameters**

- `session (sqlalchemy.orm.session.Session)` – A Session object, as obtained by `connect()` (page 66)
- `all (bool)` – Returns all stations from the database if True, or only stations where `used` = 1 if False (default)
- `net (str)` – if set, limits the stations returned to this network

**Return type** list of `msnoise.msnoise_table_def.declare_tables.Station`

**Returns** list of Station

`msnoise.api.get_station(session, net, sta)`
Get one Station from the database.

**Parameters**

- `session (sqlalchemy.orm.session.Session)` – A Session object, as obtained by `connect()` (page 66)
- `net (str)` – the network code
- `sta (str)` – the station code

**Return type** `msnoise.msnoise_table_def.declare_tables.Station`

**Returns** a Station Object

`msnoise.api.update_station(session, net, sta, X, Y, altitude, coordinates='UTM', instrument='N/A', used=1)`
Updates or Insert a new Station in the database.

**See also:**

`msnoise.msnoise_table_def.declare_tables.Station`

**Parameters**
- `session` *(sqlalchemy.orm.session.Session)* – A Session object, as obtained by `connect()` (page 66)
- `net` *(str)* – The network code of the Station
- `sta` *(str)* – The station code
- `X` *(float)* – The X coordinate of the station
- `Y` *(float)* – The Y coordinate of the station
- `altitude` *(float)* – The altitude of the station
- `coordinates` *(str)* – The coordinates system. “DEG” is WGS84 latitude/longitude in degrees. “UTM” is expressed in meters.
- `instrument` *(str)* – The instrument code, useful with PAZ correction
- `used` *(bool)* – Whether this station must be used in the computations.

**msnoise.api.get_station_pairs** *(session, used=None, net=None)*

Returns an iterator over all possible station pairs. If auto-correlation is configured in the database, returns N*N pairs, otherwise returns N*(N-1)/2 pairs.

**Parameters**
- `session` *(sqlalchemy.orm.session.Session)* – A Session object, as obtained by `connect()` (page 66)
- `used` *(bool, int)* – Select only stations marked used if False (default) or all stations present in the database if True
- `net` *(str)* – Network code to filter for the pairs.

**Return type** iterable

**Returns** An iterable of Station object pairs

**msnoise.api.get_interstation_distance** *(station1, station2, coordinates='DEG')*

Returns the distance in km between `station1` and `station2`.

**Warning:** Currently the stations coordinates system have to be the same!

**Parameters**
- `station1` *(Station)* – A Station object
- `station2` *(Station)* – A Station object
- `coordinates` *(str)* – The coordinates system. “DEG” is WGS84 latitude/longitude in degrees. “UTM” is expressed in meters.

**Return type** float

**Returns** The interstation distance in km

**msnoise.api.update_data_availability** *(session, net, sta, comp, path, file, starttime, endtime, data_duration, gaps_duration, samplerate)*

Updates a DataAvailability object in the database
Parameters

- **session** (*sqlalchemy.orm.session.Session*) – A Session object, as obtained by *connect()* (page 66)
- **net** (*str*) – The network code of the Station
- **sta** (*str*) – The station code
- **comp** (*str*) – The component (channel)
- **path** (*str*) – The full path to the folder containing the file
- **file** (*str*) – The name of the file
- **starttime** (*datetime.datetime*) – Start time of the file
- **endtime** (*datetime.datetime*) – End time of the file
- **data_duration** (*float*) – Cumulative duration of available data in the file
- **gaps_duration** (*float*) – Cumulative duration of gaps in the file
- **samplerate** (*float*) – Sample rate of the data in the file (in Hz)

```python
msnoise.api.get_new_files(session)
```

Returns the files marked “N”ew or “M”odified in the database

**Parameters**

- **session** (*sqlalchemy.orm.session.Session*) – A Session object, as obtained by *connect()* (page 66)

**Return type** list

**Returns** list of DataAvailability

```python
msnoise.api.get_data_availability(session, net=None, sta=None, comp=None, starttime=None, endtime=None)
```

Returns the DataAvailability objects for specific net, sta, starttime or endtime

**Parameters**

- **session** (*sqlalchemy.orm.session.Session*) – A Session object, as obtained by *connect()* (page 66)
- **net** (*str*) – Network code
- **sta** (*str*) – Station code
- **starttime** (*datetime.datetime, datetime.date*) – Start time of the search
- **endtime** (*datetime.datetime, datetime.date*) – End time of the search

**Return type** list

**Returns** list of DataAvailability

```python
msnoise.api.mark_data_availability(session, net, sta, flag)
```

Updates the flag of all DataAvailability objects matching net.sta in the database

**Parameters**
• session (sqlalchemy.orm.session.Session) – A Session object, as obtained by connect() (page 66)

• net (str) – Network code

• sta (str) – Station code

• flag (str) – Status of the DataAvailability object: New, Modified or Archive. Values accepted are {'N', 'M', 'A'}

msnoise.api.count_data_availability_flags(session)

Count the number of DataAvailability, grouped by flag

Parameters

• session (sqlalchemy.orm.session.Session) – A Session object, as obtained by connect() (page 66)

Return type

list

Returns

list of [count, flag] pairs

msnoise.api.update_job(session, day, pair, jobtype, flag, commit=True, returnjob=True, ref=None)

Updates or Inserts a new Job in the database.

Parameters

• day (str) – The day in YYYY-MM-DD format

• pair (str) – the name of the pair (EXAMPLE?)

• jobtype (str) – CrossCorrelation (CC) or dt/t (DTT) Job?


• commit (bool) – Whether to directly commit (True, default) or not (False)

• returnjob (bool) – Return the modified/inserted Job (True, default) or not (False)

Return type

Job or None

Returns

If returnjob is True, returns the modified/inserted Job.

msnoise.api.massive_insert_job(jobs)

Routine to use a low level function to insert much faster a list of Job. This method uses the Engine directly, no need to pass a Session object.

Parameters

• jobs (list) – a list of Job to insert.

msnoise.api.massive_update_job(session, jobs, flag='D')

Routine to use a low level function to update much faster a list of Job. This method uses the Job.ref which is unique.

Parameters

• jobs (list) – a list of Job to update.

• flag (str) – The destination flag.

msnoise.api.is_next_job(session, flag='T', jobtype='CC')

Are there any Job in the database, with flag='flag' and jobtype='type'

Parameters
- **session** *(sqlalchemy.orm.session.Session)* – A Session object, as obtained by `connect()` (page 66)
- **jobtype** *(str)* – CrossCorrelation (CC) or dt/t (DTT) Job?

**Return type** bool

**Returns** True if at least one Job matches, False otherwise.

**msnoise.api.get_next_job**(session, flag='T', jobtype='CC')

Get the next Job in the database, with flag='flag' and jobtype='jobtype'. Jobs of the same type are grouped per day. This function also sets the flag of all selected Jobs to “I”n progress.

**Parameters**
- **session** *(sqlalchemy.orm.session.Session)* – A Session object, as obtained by `connect()` (page 66)
- **jobtype** *(str)* – CrossCorrelation (CC) or dt/t (DTT) Job?

**Return type** list

**Returns** list of Job

**msnoise.api.is_dtt_next_job**(session, flag='T', jobtype='DTT', ref=False)

Are there any DTT Job in the database, with flag='flag' and jobtype='jobtype'. If ref is provided, checks if a DTT “REF” job is present.

**Parameters**
- **session** *(sqlalchemy.orm.session.Session)* – A Session object, as obtained by `connect()` (page 66)
- **jobtype** *(str)* – CrossCorrelation (CC) or dt/t (DTT) Job?
- **ref** *(bool)* – Whether to check for a REF job (True) or not (False, default)

**Return type** bool

**Returns** True if at least one Job matches, False otherwise.

**msnoise.api.get_dtt_next_job**(session, flag='T', jobtype='DTT')

Get the next DTT Job in the database, with flag='flag' and jobtype='jobtype'. Jobs are then grouped per station pair. This function also sets the flag of all selected Jobs to “I”n progress.

**Parameters**
- **session** *(sqlalchemy.orm.session.Session)* – A Session object, as obtained by `connect()` (page 66)
- **jobtype** *(str)* – CrossCorrelation (CC) or dt/t (DTT) Job?

**Return type** tuple

**Returns**
Returns (pairs, days, refs): List of station pair names - Days of the next DTT jobs - Job IDs (for later being able to update their flag).

```python
msnoise.api.reset_jobs(session, jobtype, alljobs=False, rule=None)
```
Sets the flag of all `jobtype` Jobs to “T”odo.

**Parameters**
- `session` (`sqlalchemy.orm.session.Session`) – A Session object, as obtained by `connect()` (page 66)
- `jobtype` (`str`) – CrossCorrelation (CC) or dt/t (DTT) Job?
- `alljobs` (`bool`) – If True, resets all jobs. If False (default), only resets jobs “I”n progress.

```python
msnoise.api.reset_dtt_jobs(session, pair)
```
Sets the flag of all DTT Jobs of one `pair` to “T”odo.

**Parameters**
- `session` (`sqlalchemy.orm.session.Session`) – A Session object, as obtained by `connect()` (page 66)
- `pair` (`str`) – The pair to update

```python
msnoise.api.get_job_types(session, jobtype='CC')
```
Count the number of Jobs of a specific `type`, grouped by `flag`.

**Parameters**
- `session` (`sqlalchemy.orm.session.Session`) – A Session object, as obtained by `connect()` (page 66)
- `jobtype` (`str`) – CrossCorrelation (CC) or dt/t (DTT) Job?

**Return type** `list`

**Returns** list of [count, flag] pairs

```python
msnoise.api.get_jobs_by_lastmod(session, jobtype='CC', lastmod=datetime.datetime(2019, 9, 3, 16, 18, 56, 303960))
```

**Parameters**
- `session` (`sqlalchemy.orm.session.Session`) – A Session object, as obtained by `connect()` (page 66)
- `jobtype` (`str`) – CrossCorrelation (CC) or dt/t (DTT) Job?
- `lastmod` (`datetime.datetime`) – Jobs’ modification time

**Return type** `list`

**Returns** list of Job objects.

```python
msnoise.api.export_allcorr(session, ccfid, data)
msnoise.api.export_allcorr2(session, ccfid, data)
msnoise.api.add_corr(session, station1, station2, filterid, date, time, duration, components, CF, sampling_rate, day=False, ncorr=0, params=None)
```
Adds a CCF to the data archive on disk.
Parameters

- **session** (*sqlalchemy.orm.session.Session*) – A Session object, as obtained by `connect()` (page 66)
- **station1** (*str*) – The name of station 1 (formatted NET.STA)
- **station2** (*str*) – The name of station 2 (formatted NET.STA)
- **filterid** (*int*) – The ID (ref) of the filter
- **date** (*datetime.date or str*) – The date of the CCF
- **time** (*datetime.time or str*) – The time of the CCF
- **duration** (*float*) – The total duration of the exported CCF
- **components** (*str*) – The name of the components used (ZZ, ZR, ...)
- **sampling_rate** (*float*) – The sampling rate of the exported CCF
- **day** (*bool*) – Whether this function is called to export a daily stack (True) or each CCF (when keep_all parameter is set to True in the configuration). Defaults to True.
- **ncorr** (*int*) – Number of CCF that have been stacked for this CCF.
- **params** (*dict*) – A dictionary of MSNoise config parameters as returned by `get_params()` (page 68).

```python
msnoise.api.export_sac(db, filename, pair, components, filterid, corr, ncorr=0, sac_format=None, maxlag=None, cc_sampling_rate=None, params=None)
msnoise.api.export_mseed(db, filename, pair, components, filterid, corr, ncorr=0, maxlag=None, cc_sampling_rate=None, params=None)
msnoise.api.stack(data, stack_method='linear', pws_timegate=10.0, pws_power=2, goal_sampling_rate=20.0)
```

Parameters

- **data** (*numpy.ndarray*) – the data to stack, each row being one CCF
- **stack_method** (*str*) – either linear: average of all CCF or pws to compute the phase weighted stack. If pws is selected, the function expects the pws_timegate and pws_power.
- **pws_timegate** (*float*) – PWS time gate in seconds. Width of the smoothing window to convolve with the PWS spectrum.
- **pws_power** (*float*) – Power of the PWS weights to be applied to the CCF stack.
- **goal_sampling_rate** (*float*) – Sampling rate of the CCF array submitted

**Return type** `numpy.ndarray`

**Returns** the stacked CCF.

```python
msnoise.api.get_results(session, station1, station2, filterid, components, dates, mov_stack=1, format='stack', params=None)
```

Parameters
• session (sqlalchemy.orm.session.Session) – A Session object, as obtained by `connect()` (page 66)

• station1 (str) – The name of station 1 (formatted NET_STA)

• station2 (str) – The name of station 2 (formatted NET_STA)

• filterid (int) – The ID (ref) of the filter

• components (str) – The name of the components used (ZZ, ZR, ...)

• dates (list) – List of TODO datetime.datetime

• mov_stack (int) – Moving window stack.

• format (str) – Either `stack`: the data will be stacked according to the parameters passed with `params` or `matrix`: to get a 2D array of CCF.

• params (dict) – A dictionary of MSNoise config parameters as returned by `get_params()` (page 68).

**Return type** numpy.ndarray

**Returns** Either a 1D CCF (if format is `stack` or a 2D array (if format= `matrix`).

```python
msnoise.api.get_results_all(session, station1, station2, filterid, components, dates)
```

**Parameters**

• session (sqlalchemy.orm.session.Session) – A Session object, as obtained by `connect()` (page 66)

• station1 (str) – The name of station 1 (formatted NET_STA)

• station2 (str) – The name of station 2 (formatted NET_STA)

• filterid (int) – The ID (ref) of the filter

• components (str) – The name of the components used (ZZ, ZR, ...)

• dates (list) – List of TODO datetime.datetime

**Return type** pandas.DataFrame

**Returns** All CCF results in a pandas.DataFrame, where the index is the time of the CCF and the columns are the times in the coda.

```python
msnoise.api.get_maxlag_samples(session)
```

**Returns** the length of the CCF in samples

**Parameters** session (sqlalchemy.orm.session.Session) – A Session object, as obtained by `connect()` (page 66)

**Return type** int

**Returns** the length of the CCF in samples

```python
msnoise.api.get_t_axis(session)
```

Returns the time axis (in seconds) of the CC functions. Gets the maxlag from the database and uses `get_maxlag_samples` function.
Parameters session (sqlalchemy.orm.session.Session) – A Session object, as obtained by `connect()` (page 66)

Return type numpy.array

Returns the time axis in seconds

msnoise.api.get_components_to_compute(session, plugin=None)

Returns the components configured in the database.

Parameters session (sqlalchemy.orm.session.Session) – A Session object, as obtained by `connect()` (page 66)

Return type list of str

Returns a list of components to compute

msnoise.api.get_components_to_compute_single_station(session, plugin=None)

Returns the components configured in the database.

Parameters session (sqlalchemy.orm.session.Session) – A Session object, as obtained by `connect()` (page 66)

Return type list of str

Returns a list of components to compute

msnoise.api.build_ref_datelist(session)

Creates a date array for the REF. The returned tuple contains a start and an end date, and a list of individual dates between the two.

Parameters session (sqlalchemy.orm.session.Session) – A Session object, as obtained by `connect()` (page 66)

Return type tuple

Returns (start, end, datelist)

msnoise.api.build_movstack_datelist(session)

Creates a date array for the analyse period. The returned tuple contains a start and an end date, and a list of individual dates between the two.

Parameters session (sqlalchemy.orm.session.Session) – A Session object, as obtained by `connect()` (page 66)

Return type tuple

Returns (start, end, datelist)

msnoise.api.updated_days_for_dates(session, date1, date2, pair, jobtype='CC', interval=datetime.timedelta(days=1), return_days=False)

Determines if any Job of jobtype='jobtype' and for pair='pair', concerning a date between `date1` and `date2` has been modified in the last interval='interval'.

Parameters

- session (sqlalchemy.orm.session.Session) – A Session object, as obtained by `connect()` (page 66)
- date1 (datetime.datetime) – Beginning of the period of interest
- date2 (datetime.datetime) – End of the period of interest
• **pair** (*str*) – Pair of interest
• **jobtype** (*str*) – CrossCorrelation (CC) or dt/t (DTT) Job?
• **interval** (*datetime.timedelta*) – Interval of time before now to search for updated days
• **return_days** (*bool*) – Whether to return a list of days (True) or not (False, default)

**Return type** list or bool

**Returns** List of days if return_days is True, only “True” if not. (not clear!)

```python
msnoise.api.azimuth(coordinates, x0, y0, x1, y1)
```

Returns the azimuth between two coordinate sets.

**Parameters**
- **coordinates** (*str*) – {'DEG', 'UTM', 'MIX'}
- **x0** (*float*) – X coordinate of station 1
- **y0** (*float*) – Y coordinate of station 1
- **x1** (*float*) – X coordinate of station 2
- **y1** (*float*) – Y coordinate of station 2

**Return type** float

**Returns** The azimuth in degrees

```python
msnoise.api.nextpow2(x)
```

Returns the next power of 2 of `x`.

**Parameters** `x` (*int*) – any value

**Return type** int

**Returns** the next power of 2 of `x`

```python
msnoise.api.check_and_phase_shift(trace, taper_length=20.0)
```

```python
msnoise.api.getGaps(stream, min_gap=None, max_gap=None)
```

```python
msnoise.api.make_same_length(st)
```

This function takes a stream of equal sampling rate and makes sure that all channels have the same length and the same gaps.

```python
msnoise.api.clean_scipy_cache()
```

This function wraps all destroy scipy cache at once. It is a workaround to the memory leak induced by the “caching” functions in scipy fft.

```python
msnoise.api.preload_instrument_responses(session)
```

This function preloads all instrument responses from response_format and stores the seed ids, start and end dates, and paz for every channel in a DataFrame.

**Warning:** This function only works for response_format being “inventory” or “data-less”.
Parameters

session (sqlalchemy.orm.session.Session) – A Session object, as obtained by `connect()` (page 66)

Return type pandas.DataFrame

Returns A table containing all channels with the time of operation and poles and zeros.

4.4 Core Functions

`msnoise.move2obspy.myCorr(data, maxlag, plot=False, nfft=None)`

This function takes ndimensional data array, computes the cross-correlation in the frequency domain and returns the cross-correlation function between [-maxlag:maxlag].

Parameters

- data (numpy.ndarray) – This array contains the fft of each timeseries to be cross-correlated.
- maxlag (int) – This number defines the number of samples (N=2*maxlag + 1) of the CCF that will be returned.

Return type numpy.ndarray

Returns The cross-correlation function between [-maxlag:maxlag]

`msnoise.move2obspy.myCorr2(data, maxlag, energy, index, plot=False, nfft=None, normalized=False)`

This function takes ndimensional data array, computes the cross-correlation in the frequency domain and returns the cross-correlation function between [-maxlag:maxlag].

Parameters

- data (numpy.ndarray) – This array contains the fft of each timeseries to be cross-correlated.
- maxlag (int) – This number defines the number of samples (N=2*maxlag + 1) of the CCF that will be returned.

Return type numpy.ndarray

Returns The cross-correlation function between [-maxlag:maxlag]

`msnoise.move2obspy.pcc_xcorr(data, maxlag, energy, index, plot=False, nfft=None, normalized=False)`

Parameters

- data
- maxlag
- energy
- index
- plot
- nfft
- normalized
Returns

`msnoise.move2obspy.whiten(data, Nfft, delta, freqmin, freqmax, plot=False)`

This function takes 1-dimensional `data` timeseries array, goes to frequency domain using fft, whitens the amplitude of the spectrum in frequency domain between `freqmin` and `freqmax` and returns the whitened fft.

Parameters

- `data` *(numpy.ndarray)* – Contains the 1D time series to whiten
- `Nfft` *(int)* – The number of points to compute the FFT
- `delta` *(float)* – The sampling frequency of the `data`
- `freqmin` *(float)* – The lower frequency bound
- `freqmax` *(float)* – The upper frequency bound
- `plot` *(bool)* – Whether to show a raw plot of the action (default: False)

Return type `numpy.ndarray`

Returns

The FFT of the input trace, whitened between the frequency bounds

`msnoise.move2obspy.smooth(x, window='boxcar', half_win=3)`

some window smoothing

`msnoise.move2obspy.getCoherence(dcs, ds1, ds2)`

`msnoise.move2obspy.mwcs(current, reference, freqmin, freqmax, df, tmin, window_length, step, smoothing_half_win=5)`

The current time series is compared to the reference. Both time series are sliced in several overlapping windows. Each slice is mean-adjusted and cosine-tapered (85% taper) before being Fourier-transformed to the frequency domain. \( F_{\text{car}}(\nu) \) and \( F_{\text{ref}}(\nu) \) are the first halves of the Hermitian symmetric Fourier-transformed segments. The cross-spectrum \( X(\nu) \) is defined as \( X(\nu) = F_{\text{ref}}(\nu)F_{\text{car}}^*(\nu) \)
in which \* denotes the complex conjugation. \(X(\nu)\) is then smoothed by convolution with a Hanning window. The similarity of the two time-series is assessed using the cross-coherence between energy densities in the frequency domain:

\[
C(\nu) = \frac{|X(\nu)|}{\sqrt{|F_{ref}(\nu)|^2|F_{cur}(\nu)|^2}}
\]

in which the over-line here represents the smoothing of the energy spectra for \(F_{ref}\) and \(F_{cur}\) and of the spectrum of \(X\). The mean coherence for the segment is defined as the mean of \(C(\nu)\) in the frequency range of interest. The time-delay between the two cross correlations is found in the unwrapped phase, \(\phi(u)\), of the cross spectrum and is linearly proportional to frequency:

\[
\phi_j = m.u_j, m = 2\pi\delta t
\]

The time shift for each window between two signals is the slope \(m\) of a weighted linear regression of the samples within the frequency band of interest. The weights are those introduced by [Clarke2011], which incorporate both the cross-spectral amplitude and cross-coherence, unlike [Poupinet1984]. The errors are estimated using the weights (thus the coherence) and the squared misfit to the modelled slope:

\[
e_m = \sqrt{\sum_j (\frac{\sum_i w_i \nu_i}{\sum_i w_i \nu_i^2})^2 \sigma^2_\phi}
\]

where \(w\) are weights, \(\nu\) are cross-coherences and \(\sigma^2_\phi\) is the squared misfit of the data to the modelled slope and is calculated as \(\sigma^2_\phi = \frac{\sum_j (\phi_j - m\nu_j)^2}{N-1}\)

The output of this process is a table containing, for each moving window: the central time lag, the measured delay, its error and the mean coherence of the segment.

**Warning:** The time series will not be filtered before computing the cross-spectrum! They should be band-pass filtered around the \(\text{freqmin}-\text{freqmax}\) band of interest beforehand.

**Parameters**

- **current** *(numpy.ndarray)* – The “Current” timeseries
- **reference** *(numpy.ndarray)* – The “Reference” timeseries
- **freqmin** *(float)* – The lower frequency bound to compute the dephasing (in Hz)
- **freqmax** *(float)* – The higher frequency bound to compute the dephasing (in Hz)
- **df** *(float)* – The sampling rate of the input timeseries (in Hz)
- **tmin** *(float)* – The leftmost time lag (used to compute the “time lags array”)
- **window_length** *(float)* – The moving window length (in seconds)
- **step** *(float)* – The step to jump for the moving window (in seconds)
- **smoothing_half_win** *(int)* – If different from 0, defines the half length of the smoothing hanning window.

**Return type** *numpy.ndarray*

---

### 4.4. Core Functions

81
Returns [time_axis, delta_t, delta_err, delta_mcoh]. time_axis contains the central times of the windows. The three other columns contain dt, error and mean coherence for each window.

4.5 Extending MSNoise with Plugins

New in version 1.4.

Starting with releasenotes/msnoise-1.4, MSNoise supports Plugins, this means the default workflow “from archive to dv/v” can be branched at any step!

- What is a Plugin and how to declare it in MSNoise (page 82)
- Plugin minimal structure (page 82)
- Declaring Job Types - Hooking (page 84)
- Plugin’s own config table (page 86)
- Adding Web Admin Pages (page 88)
- Uninstalling Plugins (page 89)
- Download Amazing Plugin (page 89)

4.5.1 What is a Plugin and how to declare it in MSNoise

A plugin is a python package, properly structured, that can be imported from msnoise, i.e. it has to be “installed” like any other python package.

After installing a plugin, its package name must be declared in the plugins parameter in the configuration. This must be done PER PROJECT. This configuration field supports a list of plugins, separated by a simple comma (!no space), e.g. msnoise_amazing,msnoise_plugin101.

Once configured in a project, the plugin should appear when calling the msnoise plugin command:

```
$ msnoise plugin

Usage: msnoise-script.py plugin [OPTIONS] COMMAND [ARGS]...

Runs a command in a named plugin

Options:
  --help    Show this message and exit.

Commands:
  amazing  Example Amazing Plugin for MSNoise
```

4.5.2 Plugin minimal structure

A plugin is a python package, so its minimal structure is:
The setu.py Declares where the plugin Actually hooks into MSNoise:

```python
from setuptools import setup, find_packages
setup(
    name='msnoise_amazing',
    version='0.1a',
    packages=find_packages(),
    include_package_data=True,
    install_requires=['msnoise',
                      'obspy'],
    entry_points = {
        'msnoise.plugins.commands': [
            'amazing = msnoise_amazing.plugin_definition:amazing',
        ],
    },
    author = "Thomas Lecocq & MSNoise dev team",
    author_email = "Thomas.Lecocq@seismology.be",
    description = "An example plugin",
    license = "EUPL-1.1",
    url = "http://www.msnoise.org",
    keywords="amazing seismology"
)
```

The Most important line of this file is the one declaring the amazing entry point in msnoise.plugins.commands and linking it to the plugin’s plugin_definition.py file.

The content of plugin_definition.py must then Provide at least one click.Command, or more commonly, one click.Group and many click.Command.

```python
import click
@click.group()
def amazing():
    """Example Amazing Plugin for MSNoise""
    pass
@click.command()
def sayhi():
    """A Very Polite Command""
    print("Hi")

amazing.add_command(sayhi)
```

This way, once properly installed and activated (declared in the plugins config), the plugin will be callable from msnoise:

```
$ msnoise plugin amazing
```

(continues on next page)
Usage: msnoise-script.py plugin amazing [OPTIONS] COMMAND [ARGS]...

Example Amazing Plugin for MSNoise

Options:
--help  Show this message and exit.

Commands:
sayhi  A Very Polite Command

and its command too:

$ msnoise plugin amazing sayhi
Hi

Amazing, isn’t it?

4.5.3 Declaring Job Types - Hooking

Plugin-based job types are defined by providing a register_job_types method in plugin_definition.py. A new job type is defined with two parameters:

- **name**: the actual job name (acronym style) used all over (example: CC2, TEST)
- **after**: when is this job added to the database.

Current supported “after” are:

- **new_files**: will be created when running the new_jobs command and will create a job with those parameters (nf is a new file identified in the scan_archive procedure). In this specific case, the pair field of the job will only be NET.STA, not a “pair”. A job will only be inserted if the station is “Used” in the configuration.

```python
all_jobs.append({'day': current_date, 'pair': '%s.%s'%(nf.net,nf.sta),
                 'jobtype': jobtype, 'flag': 'T',
                 'lastmod': datetime.datetime.utcnow()})
```

- **scan_archive**: will be created when running the new_jobs command, in parallel to CC jobs. This is, for example, useful when one wants to compute relative amplitude ratios between station pairs. In this case, the pair field of the job is set to the pair name.

- **refstack**: will be created when running the stack command and when a new REF stack needed to be calculated. This is, for example, useful when one wants to work on the REF stacks using a Ambient Seismic Noise Tomography code.

Plugin’s Job Types are first declared in setup.py (in Entry Points):

```python
'msnoise.plugins.jobtypes': [
    'register = msnoise_amazing.plugin_definition:register_job_types',
],
```

```python
def register_job_types():
    jobtypes = []
```

(continues on next page)
jobtypes.append( {"name": "AMAZ1", "after": "new_files"} )
return jobtypes

Then, adding a `compute` command to the `plugin_definition.py`:

```python
@click.command()
def compute():
    """Compute an Amazing Value""
    from .compute import main()
    main()

amazing.add_command(compute)
```

and creating a `compute.py` file in the plugin folder:

```python
import os
from obspy.core import UTCDateTime, read
from msnoise.api import connect, is_next_job, get_next_job, 
    get_data_availability, get_config, update_job

def main():
    db = connect()
    while is_next_job(db, jobtype='AMAZ1'):
        jobs = get_next_job(db, jobtype='AMAZ1')
        for job in jobs:
            net, sta = job.pair.split('.
            gd = UTCDateTime(job.day).datetime
            print("Processing %s.%s for day %s\n"%(net, sta, job.day))
            files = get_data_availability( 
                db, net=net, sta=sta, starttime=gd, endtime=gd, 
                comp="Z")
            for file in files:
                fn = os.path.join(file.path, file.file)
                st = read(fn, starttime=UTCDateTime(job.day), endtime=UTCDateTime(job.
                    day)+86400)
                print(st)
```

Aaand:

```
$ msnoise plugin amazing compute
Processing YA.UV05 for day 2010-09-01
1 Trace(s) in Stream:
YA.UV05.00.HHZ | 2010-09-01T00:00:00.000000Z - 2010-09-01T23:59:59.990000Z | 100.0 Hz, 
                ←8640000 samples
Processing YA.UV06 for day 2010-09-01
1 Trace(s) in Stream:
YA.UV06.00.HHZ | 2010-09-01T00:00:00.000000Z - 2010-09-01T23:59:59.990000Z | 100.0 Hz, 
                ←8640000 samples
Processing YA.UV10 for day 2010-09-01
1 Trace(s) in Stream:
YA.UV10.00.HHZ | 2010-09-01T00:00:00.000000Z - 2010-09-01T23:59:59.990000Z | 100.0 Hz, 
                ←8640000 samples
```

Provided you have reset the DataAvailability rows with a “M” or “N” flag so that when you
ran new_jobs it actually inserted the AMAZ1 jobs!

Because job-based stuff always requires a lot of trial-and-error, remember that the msnoise reset command is your best friend. In this example, we would need to msnoise reset AMAZ1 to reset “I”n Progress jobs, or msnoise reset AMAZ1 --all to reset all AMAZ1 jobs to “T”o Do.

---

**Note:**

- Currently, not all MSNoise workflow steps use the is_next_job - get_next_job logic, but it’ll be the case for MSNoise 1.5
- Only three hooks are currently present, of course, more will be added in in the future.

---

### 4.5.4 Plugin’s own config table

Plugins can create a new table in the database, e.g. in an install command. First, a amazing_table_def.py table definition file must be created:

```python
# Table definitions for Amazing
from sqlalchemy import Column, String
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class AmazingConfig(Base):
    """
    Config Object
    """
    :type name: str
    :param name: The name of the config bit to set.

    :type value: str
    :param value: The value of parameter `name`
    """
    __tablename__ = "amazing-config"
    name = Column(String(255), primary_key=True)
    value = Column(String(255))

    def __init__(self, name, value):
        """
        self.name = name
        self.value = value

    def __init__(self, name, value):
        """
        self.name = name
        self.value = value

    def __init__(self, name, value):
        """
        self.name = name
        self.value = value
```

and a default.py file containing the parameters names, explanation and default value:

```python
from collections import OrderedDict
default = OrderedDict()
default['parameter1'] = ['Some really useful text','1']
default['parameter2'] = ['Some really useful text','1']
default['parameter3'] = ['Some really useful text','1']
default['parameter4'] = ['Some really useful text','1']
```
default['question1'] = ['Is this a useful text [Y]/N', 'Y']

Then, the `install.py` file contains the method to add this table to the database:

```python
from msnoise.api import *
from .amazing_table_def import AmazingConfig
from .default import default

def main():
    engine = get_engine()
    Session = sessionmaker(bind=engine)
    session = Session()

    AmazingConfig.__table__.create(bind=engine, checkfirst=True)
    for name in default.keys():
        session.add(AmazingConfig(name=name, value=default[name][-1]))
    session.commit()

then add the command to the `plugin_definition.py`:

```python
@click.command()
def install():
    """Create the Config table""
    from .install import main
    main()

amazing.add_command(install)
```

When all this is prepared, running the `msnoise plugin amazing install` command will connect to the current database, create the `amazing-config` table and add the parameters names and their default value.

An entry point to the `setup.py` file has to be defined in order to access Plugin’s config tables via the `msnoise api` `msnoise.api.get_config()` (page 67) method:

```
'msnoise.plugins.table_def': [
    'AmazingConfig = msnoise_amazing.amazing_table_def:AmazingConfig',
],
```

Then, running a simple python command:

```python
from msnoise.api import connect, get_config

db = connect()
print(get_config(db, "parameter1", plugin="Amazing"))
print(get_config(db, "parameter2", plugin="Amazing"))
print(get_config(db, "parameter3", plugin="Amazing"))
print(get_config(db, "parameter4", plugin="Amazing"))
print(get_config(db, "question1", plugin="Amazing", isbool=True))
```

should print:
4.5.5 Adding Web Admin Pages

Plugins can also declare new pages to the Web Admin! This is simply done by, again, declaring some entry points in setup.py:

```python
'msnoise.plugins.admin_view': [ 'AmazingConfigView = msnoise_amazing.plugin_definition:AmazingConfigView', ],
```

and the corresponding object in plugin_definition.py:

```python
from flask.ext.admin.contrib.sqla import ModelView
from .amazing_table_def import AmazingConfig
class AmazingConfigView(ModelView):
    # Disable model creation
    view_title = "MSNoise Amazing Configuration"
    name = "Configuration"

    can_create = False
    can_delete = False
    page_size = 50
    # Override displayed fields
    column_list = ('name', 'value')

    def __init__(self, session, **kwargs):
        # You can pass name and other parameters if you want to
        super(AmazingConfigView, self).__init__(AmazingConfig, session,
            endpoint="amazingconfig",
            name="Config",
            category="Amazing", **kwargs)
```

Then (as always, after re-developing/installing the package), the magic occurs:
Or, changing the last 4 lines of the previous code to:

```python
super(AmazingConfigView, self).__init__(AmazingConfig, session,
    endpoint="amazingconfig",
    name="Amazing Config",
    category="Configuration", **kwargs)
```

4.5.6 Uninstalling Plugins

Plugins can be de-activated by removing their package name from the `plugins` configuration parameter. Ideally, plugins should provide an `uninstall` command similar to the `install` to take care of deleting/dropping the tables in the project database.

4.5.7 Download Amazing Plugin

That’s cheating, you know? :-)

Download the Amazing Plugin
4.6 Help on the msnoise commands

This page shows all the command line interface commands.

4.6.1 msnoise admin

```
msnoise admin --help
Usage: [OPTIONS]

Starts the Web Admin on http://localhost:5000 by default

Options:
   -p, --port INTEGER  Port to open
   --help              Show this message and exit.
```

4.6.2 msnoise bugreport

```
msnoise bugreport --help
Usage: [OPTIONS]

This command launches the Bug Report script.

Options:
   -s, --sys System Info
   -m, --modules Modules Info
   -e, --env Environment Info
   -a, --all All Info
   --help Show this message and exit.
```

4.6.3 msnoise compute_cc

```
msnoise compute_cc --help
Usage: [OPTIONS]

Computes the CC jobs (based on the "New Jobs" identified)

Options:
   --help Show this message and exit.
```

4.6.4 msnoise compute_cc_rot

```
msnoise compute_cc_rot --help
Usage: [OPTIONS]

Computes the CC jobs (based on the "New Jobs" identified)
```

(continues on next page)
Options:
--help  Show this message and exit.

4.6.5 msnoise compute_dtt

msnoise compute_dtt --help
Usage: [OPTIONS]

Computes the dt/t jobs based on the new MWCS data

Options:
-i, --interval FLOAT  Number of days before now to search for modified Jobs
--help  Show this message and exit.

4.6.6 msnoise compute_mwcs

msnoise compute_mwcs --help
Usage: [OPTIONS]

Computes the MWCS jobs

Options:
--help  Show this message and exit.

4.6.7 msnoise compute_stretching

msnoise compute_stretching --help
Usage: [OPTIONS]

[experimental] Computes the stretching based on the new stacked data

Options:
--help  Show this message and exit.

4.6.8 msnoise config

msnoise config get

msnoise config get --help
Usage: [OPTIONS] [NAMES]...
Display the value of the given configuration variable(s).

Options:
--help  Show this message and exit.

**msnoise config gui**

`msnoise config gui --help`

**Usage:** [OPTIONS]

Run the deprecated configuration GUI tool. Please use the configuration web interface using 'msnoise admin' instead.

Options:
--help  Show this message and exit.

**msnoise config set**

`msnoise config set --help`

**Usage:** [OPTIONS] NAME_VALUE

Set a configuration value. The argument should be of the form 'variable=value'.

Options:
--help  Show this message and exit.

**msnoise config sync**

`msnoise config sync --help`

**Usage:** [OPTIONS]

Synchronise station metadata from inventory/dataless.

Options:
--help  Show this message and exit.

4.6.9 *msnoise db*

`msnoise db clean_duplicates`
msnoise db clean_duplicates --help

Usage: [OPTIONS]

Checks the Jobs table and deletes duplicate entries

Options:
--help  Show this message and exit.

msnoise db dump

msnoise db dump --help

Usage: [OPTIONS]

Dumps the complete database in a formatted structure.

Options:
--format TEXT
--help  Show this message and exit.

msnoise db execute

msnoise db execute --help

Usage: [OPTIONS] SQL_COMMAND

EXPERT MODE: Executes 'sql_command' on the database. Use this command at your own risk!!

Options:
--help  Show this message and exit.

msnoise db import

msnoise db import --help

Usage: [OPTIONS] TABLE

Imports msnoise tables from formatted files (csv).

Options:
--format TEXT
--force
--help  Show this message and exit.

msnoise db init
msnoise db init --help

Usage: [OPTIONS]

This command initializes the current folder to be a MSNoise Project by creating a database and a db.ini file.

Options:
--tech TEXT Database technology: 1=SQLite 2=MySQL
--help Show this message and exit.

msnoise db upgrade

msnoise db upgrade --help

Usage: [OPTIONS]

Upgrade the database from previous to a new version.

This procedure adds new parameters with their default value in the config database.

Options:
--help Show this message and exit.

4.6.10 msnoise info

msnoise info --help

Usage: [OPTIONS]

Outputs general information about the current install and config, plus information about jobs and their status.

Options:
-j, --jobs Jobs Info only
--help Show this message and exit.

4.6.11 msnoise install

msnoise install --help

Usage: [OPTIONS]

DEPRECATED: since MSNoise 1.6, please use "msnoise db init" instead

Options:
--help Show this message and exit.
4.6.12 msnoise jupyter

msnoise jupyter --help

Usage: [OPTIONS]

Launches an jupyter notebook in the current folder

Options:
--help  Show this message and exit.

4.6.13 msnoise new_jobs

msnoise new_jobs --help

Usage: [OPTIONS]

Determines if new CC jobs are to be defined

Options:
-i, --init First run ? This disables the check for existing jobs.
--nocc Disable the creation of CC jobs.
--hpc TEXT Format PREVIOUS:NEXT. When running on HPC, create the next jobs in the workflow based on the previous step mentioned here. Example: "msnoise new_jobs --hpc CC:STACK" will create STACK jobs based on CC jobs marked "D"one.
--help Show this message and exit.

4.6.14 msnoise p

Will be automatically populated with the commands declared by the plugins (p is an alias for plugin)

4.6.15 msnoise plot

msnoise plot ccftime

msnoise plot ccftime --help

Usage: [OPTIONS] STA1 STA2 [EXTRA_ARGS]...

Plots the ccf vs time between sta1 and sta2

STA1 and STA2 must be provided with this format: NET.STA!

Options:
-f, --filterid INTEGER  Filter ID

(continues on next page)
msnoise plot data_availability

msnoise plot data_availability --help

Usage: [OPTIONS]

Plots the Data Availability vs time

Options:
- -s, --show BOOLEAN  Show interactively?
- -o, --outfile TEXT  Output filename (default: auto)
- --help

msnoise plot distance

msnoise plot distance --help

Usage: [OPTIONS] [EXTRA_ARGS]...

Plots the REFs of all pairs vs distance

Options:
- -f, --filterid INTEGER  Filter ID
- -c, --comp TEXT  Components (ZZ, ZR,...)
- -a, --ampli FLOAT  Amplification
- -s, --show BOOLEAN  Show interactively?
- -o, --outfile TEXT  Output filename (default: auto)
- -r, --refilter TEXT  Refilter CCFs before plotting (e.g. 4:8 for filtering CCFs between 4.0 and 8.0 Hz. This will update the plot title.)
- --virtual-source TEXT  Use only pairs including this station. Format must be NET.STA
- --help

msnoise plot dtt

(continued from previous page)
msnoise plot dtt --help

Usage: [OPTIONS] STA1 STA2 DAY

Plots a graph of dt against t

STA1 and STA2 must be provided with this format: NET.STA

DAY must be provided in the ISO format: YYYY-MM-DD

Options:
-f, --filterid INTEGER  Filter ID
-c, --comp TEXT  Components (ZZ, ZR,...)
-m, --mov_stack INTEGER  Mov Stack to read from disk
-s, --show BOOLEAN  Show interactively?
-o, --outfile TEXT  Output filename (=?auto)
--help  Show this message and exit.

msnoise plot dvv

msnoise plot dvv --help

Usage: [OPTIONS]

Plots the dv/v (parses the dt/t results)

Individual pairs can be plotted extra using the -p flag one or more times.

Example: msnoise plot dvv -p ID_KWUI_ID_POSI

Example: msnoise plot dvv -p ID_KWUI_ID_POSI -p ID_KWUI_ID_TRWI

Remember to order stations alphabetically!

Options:
-f, --filterid INTEGER  Filter ID
-c, --comp TEXT  Components (ZZ, ZR,...)
-m, --mov_stack INTEGER  Plot specific mov stacks
-p, --pair TEXT  Plot a specific pair
-A, --all  Show the ALL line?
-M, --dttname TEXT  Plot M or M0?
-s, --show BOOLEAN  Show interactively?
-o, --outfile TEXT  Output filename (=?auto)
--help  Show this message and exit.

msnoise plot interferogram

msnoise plot interferogram --help

Usage: [OPTIONS] STA1 STA2 [EXTRA_ARGS]...

Plots the interferogram between sta1 and sta2 (parses the CCFs)

(continues on next page)
STA1 and STA2 must be provided with this format: NET.STA!

Options:
- `-f`, `--filterid` INTEGER Filter ID
- `-c`, `--comp` TEXT Components (ZZ, ZR,...)
- `-m`, `--mov_stack` INTEGER Mov Stack to read from disk
- `-s`, `--show` BOOLEAN Show interactively?
- `-o`, `--outfile` TEXT Output filename (=auto)
- `-r`, `--refilter` TEXT Refilter CCFs before plotting (e.g. 4:8 for filtering CCFs between 4.0 and 8.0 Hz. This will update the plot title.

--help
Show this message and exit.

msnoise plot mwcs

msnoise plot mwcs --help

Usage: [OPTIONS] STA1 STA2

Plots the mwcs results between sta1 and sta2 (parses the CCFs)

STA1 and STA2 must be provided with this format: NET.STA!

Options:
- `-f`, `--filterid` INTEGER Filter ID
- `-c`, `--comp` TEXT Components (ZZ, ZR,...)
- `-m`, `--mov_stack` INTEGER Mov Stack to read from disk
- `-s`, `--show` BOOLEAN Show interactively?
- `-a`, `--ampli` FLOAT Amplification
- `-o`, `--outfile` TEXT Output filename (=auto)
- `-r`, `--refilter` TEXT Refilter CCFs before plotting (e.g. 4:8 for filtering CCFs between 4.0 and 8.0 Hz. This will update the plot title.

--help
Show this message and exit.

msnoise plot spectime

msnoise plot spectime --help

Usage: [OPTIONS] STA1 STA2 [EXTRA_ARGS]...

Plots the ccf's spectrum vs time between sta1 and sta2

STA1 and STA2 must be provided with this format: NET.STA!

Options:
- `-f`, `--filterid` INTEGER Filter ID
- `-c`, `--comp` TEXT Components (ZZ, ZR,...)
- `-m`, `--mov_stack` INTEGER Mov Stack to read from disk
- `-a`, `--ampli` FLOAT Amplification
- `-s`, `--show` BOOLEAN Show interactively?
- `-o`, `--outfile` TEXT Output filename (=auto)
- `-r`, `--refilter` TEXT Refilter CCFs before plotting (e.g. 4:8 for filtering CCFs between 4.0 and 8.0 Hz. This will update the plot title.

--help
Show this message and exit.
msnoise plot station_map

Usage: [OPTIONS]

Plots the station map (very very basic)

Options:
-s, --show BOOLEAN  Show interactively?
-o, --outfile TEXT  Output filename (=?auto)
--help              Show this message and exit.

msnoise plot timing

Usage: [OPTIONS]

Plots the timing (parses the dt/t results)

Individual pairs can be plotted extra using the -p flag one or more times.

Example: msnoise plot timing -p ID_KWUI_ID_POSI
Example: msnoise plot timing -p ID_KWUI_ID_POSI -p ID_KWUI_ID_TRWI

Remember to order stations alphabetically!

Options:
-f, --filterid INTEGER  Filter ID
-c, --comp TEXT  Components (ZZ, ZR,...)
-m, --mov_stack INTEGER  Plot specific mov stacks
-p, --pair TEXT  Plot a specific pair
-A, --all  Show the ALL line?
-M, --dttname TEXT  Plot M or M0?
-s, --show BOOLEAN  Show interactively?
-o, --outfile TEXT  Output filename (=?auto)
--help              Show this message and exit.

4.6.16 msnoise plugin

Will be automatically populated with the commands declared by the plugins (p is an alias for plugin)

4.6.17 msnoise populate

Usage: [OPTIONS]

(continues on next page)
Rapidly scan the archive filenames and find Network/Stations

Options:
--fromDA Populates the station table using network and station codes found in the data_availability table, overrides the default workflow step.
--help Show this message and exit.

4.6.18 msnoise reset

msnoise reset --help

Usage: [OPTIONS] JOBTYPE

Resets the job to "T"odo. JOBTYPE is the acronym of the job type. By default only resets jobs "I"n progress. --all resets all jobs, whatever the flag value. Standard Job Types are CC, STACK, MWCS and DTT, but plugins can define their own.

Options:
-a, --all Reset all jobs
-r, --rule TEXT Reset job that match this SQL rule
--help Show this message and exit.

4.6.19 msnoise scan_archive

msnoise scan_archive --help

Usage: [OPTIONS]

Scan the archive and insert into the Data Availability table.

Options:
-i, --init First run ?
--path TEXT Scan all files in specific folder, overrides the default workflow step.
-r, --recursively When scanning a path, walk subfolders automatically ?
--crondays TEXT Number of past days to monitor, typically used in cron jobs (overrides the 'crondays' configuration value). Must be a float representing a number of days, or designate weeks, days, and/or hours using the format 'Xw Xd Xh'.
--help Show this message and exit.

4.6.20 msnoise stack

msnoise stack --help

Usage: [OPTIONS]

(continues on next page)
Stacks the [REF] or [MOV] windows. Computes the STACK jobs.

Options:
- r, --ref Compute the REF Stack
- m, --mov Compute the MOV Stacks
- s, --step Compute the STEP Stacks
--help Show this message and exit.

4.6.21 msnoise test

msnoise test --help

Usage: [OPTIONS]

Runs the test suite, should be executed in an empty folder!

Options:
- p, --prefix TEXT Prefix for tables
--help Show this message and exit.

4.6.22 msnoise upgrade-db

msnoise upgrade-db --help

Usage: [OPTIONS]

DEPRECATED: since MSNoise 1.6, please use "msnoise db upgrade" instead

Options:
--help Show this message and exit.
5.1 Table Definitions

class msnoise.msnoise_table_def.Filter(**kwargs)
    Filter base class.

Parameters
    • ref (int) – The id of the Filter in the database
    • low (float) – The lower frequency bound of the Whiten function (in Hz)
    • high (float) – The upper frequency bound of the Whiten function (in Hz)
    • mwcs_low (float) – The lower frequency bound of the linear regression done in MWCS (in Hz)
    • mwcs_high (float) – The upper frequency bound of the linear regression done in MWCS (in Hz)
    • rms_threshold (float) – Not used anymore
    • mwcs_wlen (float) – Window length (in seconds) to perform MWCS
    • mwcs_step (float) – Step (in seconds) of the windowing procedure in MWCS
    • used (bool) – Is the filter activated for the processing

Attributes
    high
    low
    mwcs_high
    mwcs_low
    mwcs_step
    mwcs_wlen
    ref
    rms_threshold
    used
class msnoise.msnoise_table_def.Job(day, pair, jobtype, flag, last-mod=datetime.datetime(2019, 9, 3, 14, 18, 56, 296951))

Job Object

Parameters

- ref (int) – The Job ID in the database
- day (str) – The day in YYYY-MM-DD format
- pair (str) – the name of the pair (EXAMPLE?)
- jobtype (str) – CrossCorrelation (CC) or dt/t (DTT) Job?

Attributes

day
flag
jobtype
lastmod
pair
ref

class msnoise.msnoise_table_def.Station(*args)

Station Object

Parameters

- ref (int) – The Station ID in the database
- net (str) – The network code of the Station
- sta (str) – The station code
- X (float) – The X coordinate of the station
- Y (float) – The Y coordinate of the station
- altitude (float) – The altitude of the station
- coordinates (str) – The coordinates system. “DEG” is WGS84 latitude/longitude in degrees. “UTM” is expressed in meters.
- instrument (str) – The instrument code, useful with PAZ correction
- used (bool) – Whether this station must be used in the computations.

Attributes

X
Y
altitude
coordinates
instrument

104 Chapter 5. Development & Miscellaneous
class msnoise.msnoise_table_def.Config(name, value)

Config Object

Parameters

• name (str) – The name of the config bit to set.
• value (str) – The value of parameter name

Attributes

name
value

class msnoise.msnoise_table_def.DataAvailability(net, sta, comp, path, file, start-
time, endtime, data_duration, gaps_duration, samplerate, flag)

DataAvailability Object

Parameters

• ref (int) – The Station ID in the database
• net (str) – The network code of the Station
• sta (str) – The station code
• comp (str) – The component (channel)
• path (str) – The full path to the folder containing the file
• file (str) – The name of the file
• starttime (datetime) – Start time of the file
• endtime (datetime) – End time of the file
• data_duration – Cumulative duration of available data in the file
• gaps_duration (float) – Cumulative duration of gaps in the file
• samplerate (float) – Sample rate of the data in the file (in Hz)
• flag (str) – The status of the entry: “N”ew, “M”odified or “A”rchive

Attributes

comp
data_duration
endtime
file
flag
5.2 About Databases and Performances

To quote the SQLite website:

Appropriate Uses For SQLite

SQLite is different from most other SQL database engines in that its primary design goal is to be simple:

- Simple to administer
- Simple to operate
- Simple to embed in a larger program
- Simple to maintain and customize

Many people like SQLite because it is small and fast. But those qualities are just happy accidents. Users also find that SQLite is very reliable. Reliability is a consequence of simplicity. With less complication, there is less to go wrong. So, yes, SQLite is small, fast, and reliable, but first and foremost, SQLite strives to be simple.

Simplicity in a database engine can be either a strength or a weakness, depending on what you are trying to do. In order to achieve simplicity, SQLite has had to sacrifice other characteristics that some people find useful, such as high concurrency, fine-grained access control, a rich set of built-in functions, stored procedures, esoteric SQL language features, XML and/or Java extensions, tera- or peta-byte scalability, and so forth. If you need some of these features and do not mind the added complexity that they bring, then SQLite is probably not the database for you. SQLite is not intended to be an enterprise database engine. It is not designed to compete with Oracle or PostgreSQL.

The basic rule of thumb for when it is appropriate to use SQLite is this: Use SQLite in situations where simplicity of administration, implementation, and maintenance are more important than the countless complex features that enterprise database engines provide. As it turns out, situations where simplicity is the better choice are more common than many people realize.

Another way to look at SQLite is this: SQLite is not designed to replace Oracle. It is designed to replace fopen().

To test MSNoise, one can work with a SQLite database. SQLite communication is supported by default in Python (part of the standard library). The major drawback of SQLite is that it doesn’t support high concurrency. In the case of MSNoise, this means that only one Thread (or
Process) can interact with the database “at a time”. For small batch tests or small runs, that is OK, but when processing larger archives (years of data of 5+ stations), then the implementation of a MySQL database will allow to process the jobs in parallel.

**Note:** I have been working on some sort of API server layer above a single SQLite database, working as a Queuing system. The API server is the only client of the database, and exchanges data with the code *via* json HTTP requests. Any help, idea, brainstorming on this is welcome!

### 5.3 References

### 5.4 Contributors

The following people have contributed to MSNoise (sorted alphabetically):

- Xavier Béguin
- Corentin Caudron
- Clare Donaldson
- Raphaël De Plaen
- Robert Green
- Damiam Kula
- Thomas Lecocq
- Aurélien Mordret
- Lukas E. Preiswerk
- Carmelo Sammarco
- Arnaud Watlet

### 5.5 Release Notes

The release notes are not converted to PDF, please read them online.
BIBLIOGRAPHY


