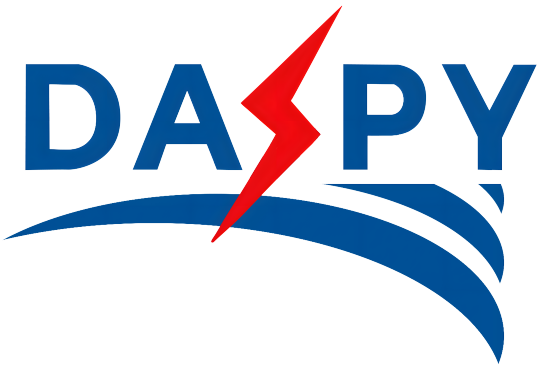
DASPy-tutorial

Release 1.0.0

Minzhe Hu

Jul 26, 2025

CONTENTS



DASPy is an open-source project that provides a python package for DAS (Distributed Acoustic Sensing) data processing.

The goal of the DASPy project is to lower the bar for DAS data processing and promote the development of DAS seismology.

CHAPTER

ONE

CONTENTS

1.1 Installation

DASPy can be installed through common methods such as Pip and Conda.

1.1.1 Install via Pip

Install from PyPI:

```
pip install daspy-toolbox
```

Install the latest version from GitHub:

```
pip install git+https://github.com/HMZ-03/DASPy.git
```

If you installed DASPy this way, you can upgrade DASPy with the following command:

```
pip install --upgrade git+https://github.com/HMZ-03/DASPy.git
```

1.1.2 Install via Conda

```
conda install daspy-toolbox
```

or

```
conda install conda-forge::daspy-toolbox
```

1.1.3 Manual Installation

1. Clone or download the DASPy repository from Github;
2. Install the following dependency packages: numpy, scipy >=1.13, matplotlib, geographiclib, pyproj, h5py, segyio, nptdms;
3. Add the DASPy directory to your Python package path.

1.2 Reading DAS Data

1.2.1 Reading Data from Files

The `read function` is used to read multiple types of DAS data files. The supported data formats are shown in the following table:

format	read	write	source
HDF5	√	√	OptaSense, Silixa, Febus, AP Sensing, ASN, Sintela, Aragón Photonics□T8 Sensors, Smart Earth Sensing, AI4EPS, INGV, JAMSTEC and FORESEE database
TDMS	√	√	Silixa and the Institute of Semiconductors, CAS
SEG-Y	√	√	OptaSens and Silixa
PICKLE	√	√	daspy.Section or numpy.ndarray class instances saved in binary
NPY	√	√	numpy.ndarray class instances saved in binary

By default, the function will output a `Section` class instance containing data and metadata:

```
>>> from daspy import read
>>> sec = read() # Read waveform example
```

You can set `output_type='array'` to output data in `numpy.array` format, in which case metadata will be output in `dict` format. Setting `ch1` or/and `ch2` limits the range of channels to read.

```
>>> data, metadata = read(output_type='array', ch1=2700, ch2=2800) # set channel range
```

When only metadata is needed, you can set `headonly=True` to save reading time. In this case, the returned data will be an all-zero array of the same size as the original data.

1.2.2 Customize a Read Function

If DASPy does not support reading your data, you can create a custom read function. Although the function can complete the reading with only the parameter `fname` , we recommend that users allow the function to input parameters `headonly`, `ch1` and `ch2` to ensure full compatibility:

```
>>> import h5py
>>> from daspy import DASDateTime
>>>
>>> def read_new_h5(fname, headonly=False, **kwargs):
>>>     with h5py.File(fname, 'r') as h5_file:
>>>         start_channel = h5_file.attrs['channel_start']
>>>         end_channel = h5_file.attrs['channel_end']
>>>         ch1 = kwargs.pop('ch1', start_channel)
>>>         ch2 = kwargs.pop('ch2', end_channel)
>>>         dch = kwargs.pop('dch', 1)
>>>         if headonly:
>>>             data = np.zeros_like(h5_file['raw_data'])
>>>         else:
>>>             data = h5_file['raw_data'][ch1-start_channel:ch2-start_channel:dch]
>>>         dx = h5_file.attrs['channel spacing m']
>>>         metadata = {'dx': dx, 'fs': h5_file.attrs['sampling rate Hz'],
```

(continues on next page)

(continued from previous page)

```
>>>                     'gauge_length': h5_file.attrs['GL m'],
>>>                     'start_channel': start_channel + ch1,
>>>                     'start_distance': (start_channel + ch1) * dx,
>>>                     'start_time': DASDateTime.strptime(h5_file.attrs['starttime'],
↪ '%Y-%m-%dT%H:%M:%S.%f'),
>>>                     'scale': h5_file.attrs['scale factor to strain']}
>>>     return data, metadata
```

The required keywords in `metadata` are `dx` and `fs` (which can be temporarily set to `None` if they are not included in the file metadata), and the remaining parameters can be read on demand. This read function can then be input as the `ftype` parameter to the `read` function or used to construct a `Collection` class (See *Processing continuous data* for details).

```
>>> sec = read(fname, ftype=read_new_h5, headonly=True)
>>> coll = Collection('../data/*.h5', ftype=read_new_h5)
```

1.2.3 Converting from Instances of Other Packages

It's supported to streaming data from ObsPy , DASCore and lightguide to DASPy:

```
>>> from daspy import Section
>>> sec_obspy = Section.from_obspy_stream(st) # convert obspy.core.stream.Stream␣
↪instance to daspy.section instance
>>> sec_dascore = Section.from_dascore_patch(patch) # convert dascore.core.patch.
↪Patch instance to daspy.section instance
>>> sec_lightguide = Section.from_lightguide_blast(blast) # convert lightguide.blast.
↪Blast instance to daspy.section instance
```

1.2.4 Section

The functions of DASPy can be implemented by calling functions (process-oriented) or through class methods (object-oriented). We recommend an object-oriented approach to operate on the `Section` class of the DAS data body. All user-facing functions and class methods in DASPy have the same name and are nouns (such as `normalization` and `Section.normalization`).

The attributes of the `Section` class save the DAS data and header information, among which the waveform data `data`, channel spacing `dx`, and sampling rate `fs` must be set (but can be temporarily Set to `None`); the starting channel number `start_channel`, the starting distance `start_distance`, and the starting time `start_time` default to 0 if not set; the event starting time `origin_time`, gauge length `gauge_length`, data type `data_type`, data scale `scale`, array geometry `geometry`, turning point channel number `turning_channels`, other header information `headers` are not necessary.

Accessing meta data

```
>>> print(sec)
              data: shape(500, 5000)
                dx: 1 m
                fs: 100.0 Hz
     start_channel: 2500
    start_distance: 2520 m
        start_time: 2016-03-21 07:37:30.532309+00:00
       origin_time: 2016-03-21 07:37:10.535000+00:00
         data_type: strain rate
```

In addition, the data size `shape`, the number of channels `nch`, the number of sampling points `nt`, the end channel number `end_channel`, the end distance `end_distance` and the end time `end_time` will Automatically calculated and can be called as a property.

1.2.5 DASDateTime

DASPy create the `DASDateTime` class to represent the time information of the data, including the start time `start_time`, the end time `end_time` and the event start time `origin_time`.

`DASDateTime` is a subclass of the `datetime.DateTime` class and inherits all methods of `datetime.DateTime`:

```
>>> from daspy.core import DASDateTime
>>> DASDateTime.strptime('2021-03-19T1:52:23', '%Y-%m-%dT%H:%M:%S')
DASDateTime(2021, 03, 19, 1, 52, 23)
```

DASPy has built-in local time zone `local_tz` and utc time zone `utc` for specifying the time zone:

```
>>> from daspy.core.dasdatetime import utc, local_tz
>>> DASDateTime.fromtimestamp(1616089943, tz=utc)
DASDateTime(2021, 3, 18, 17, 52, 23, tzinfo=datetime.timezone.utc)
```

Use `datetime.timezone(datetime.timedelta(hours=h))` to create other time zones if needed.

You may use the `local`, `utc`, and `remove_tz` methods to convert or remove timezone information:

```
>>> time = DASDateTime.strptime('2021-03-19T1:52:23Z', '%Y-%m-%dT%H:%M:%S%z')
>>> time.local()
DASDateTime(2021, 3, 19, 9, 52, 23, tzinfo=datetime.timezone(datetime.
↪timedelta(seconds=28800)))
>>> time.remove_tz()
DASDateTime(2021, 3, 19, 1, 52, 23)
```

In addition to the addition and subtraction operations between `datetime.datetime` and `datetime.timedelta` supported by the parent class itself, `DASDateTime` also supports input numbers and iterable objects `Iterable` to calculate additions and subtraction. All time differences are expressed in seconds (s), and problems with unspecified time zones are automatically handled:

```
>>> DASDateTime(2021, 3, 24, 14, 28, 0, 0) + 100
DASDateTime(2021, 3, 24, 14, 29, 40)
>>> DASDateTime(2021, 3, 24, 14, 28, 0, 0) + [10, 20, 30]
[DASDateTime(2021, 3, 24, 14, 28, 10), DASDateTime(2021, 3, 24, 14, 28, 20),␣
↪DASDateTime(2021, 3, 24, 14, 28, 30)]
>>> DASDateTime(2021, 3, 24, 14, 28, 0, 0) - 100
DASDateTime(2021, 3, 24, 14, 26, 20)
>>> DASDateTime(2021, 3, 24, 14, 28, 0, 0) - DASDateTime(2021, 3, 19, 1, 52, 23)
477337.0
```

`DASDateTime` instances can be converted to parent class `datetime.datetime` instances when necessary:

```
>>> DASDateTime(2021, 3, 19, 1, 52, 23).convert_to_datetime()
datetime.datetime(2021, 3, 19, 1, 52, 23)
```

1.3 Processing continuous data

When processing continuous-time signals such as various types of filtering, downsampling, cosine taper, time differentiation or integration, processing each single file and then splicing then will cause data discontinuity. Using the `collection` class to process continuous data can solve the continuity problem of these operations.

1.3.1 Collection

The `Collection` class is used to store continuous data belonging to the same acquisition. It saves the list of continuous data file paths `flist` and the start time of each file `ftime`, as well as the metadata information of the acquisition, including the number of channels `nch`, the track spacing `dx`, the gauge length `gauge_length`, the sampling rate `fs`, the number of sampling points of a single file `nt` and the duration of a single file `flength`. When initializing a class instance, by default, the metadata of the second data file is read to obtain all metadata information, and the start time of each file `ftime` is automatically calculated based on the start time `start_time` of the second data:

```
>>> from daspy import Collection
>>> coll = Collection('data/*.h5')
>>> coll
       flist: 2608 files
              [data/TEST_2000m_4m_1_5000Hz_200Hz_UTC8_202312132304.h5,
               data/TEST_2000m_4m_1_5000Hz_200Hz_UTC8_202312132305.h5,
               ...,
               data/TEST_2000m_4m_1_5000Hz_200Hz_UTC8_202312151831.h5]
      ftime: 2023-12-13 23:04:00.558582+00:00 to 2023-12-15 18:32:00.558582+00:00
    flength: 60.0
        nch: 1956
         nt: 12000
         dx: 1.022494912147522
         fs: 200.0
gauge_length: 4.0
```

- Set `meta_from_file='all'` to read metadata for each file. DASPy will check if all metadata are consistent, data is continuous, and save and calculate required properties. Metadata of various types can also be specified directly. This method is very time-consuming for large amounts of continuous data.
- Set `timeinfo_format` to get the start time from the file name, check for continuity and calculate `flength`. `timeinfo_format='DAS_%Y-%m-%dT%H:%M:%S%z.h5'` indicates the time corresponding to the file name, `timeinfo_format=(slice(33:45), '%Y%m%d%H%M%S')` indicates the time corresponding to the slice of the file name.

1.3.2 Data Selection

Select data for a certain time period from the `Collection` class:

```
>>> from daspy import DASDateTime
>>> coll.select(stime = DASDateTime(2023, 12, 14, 12), etime=DASDateTime(2023, 12, 14,
↪ 13))
       flist: 61 files
              [data/TEST_2000m_4m_1_5000Hz_200Hz_UTC8_202312141159.h5,
               data/TEST_2000m_4m_1_5000Hz_200Hz_UTC8_202312141200.h5,
               ...,
               data/TEST_2000m_4m_1_5000Hz_200Hz_UTC8_202312141259.h5]
       ftime: 2023-12-13 23:04:00.558582+00:00 to 2023-12-15 18:32:00.558582+00:00
```

(continues on next page)

(continued from previous page)

```
     flength: 60.0
         nch: 1956
          nt: 12000
          dx: 1.022494912147522
          fs: 200.0
gauge_length: 4.0
```

Setting `readsec=True` will directly read the data of the required period as an instance of the `Section` class and return it.

1.3.3 Continuous Data Processing

The `Collection.process` method can read all files in turn as `Section` class instances, perform a series of required processing, add a suffix and save to the target directory. This method will solve data discontinuity by caching filter states and other methods. An example of low-pass filtering all data and downsampling by 5 times in both time and space domains, and then integrating:

```
>>> operations = [['downsampling', dict(tint=5, xint=5, lowpass_filter=True)],
                  ['time_integration', dict()]]
>>> coll.process(operations, savepath='../processed', suffix='_pro')
```

1.4 Basic Processing

DASPy integrates three modules: *Preprocessing*, *Filtering* and *Frequency Attribute* to meet basic data processing needs.

1.4.1 Preprocessing

DASPy integrates a large number of commonly used DAS preprocessing tools, including `phase2strain` , `normalization` , `demeaning` , `detrending` , `stacking` , `cosine_taper` , `downsampling` , `trimming` , `padding` , `time_integration` and `time_differential`

```
>>> from daspy import read
>>> sec = read()
>>> sec.detrending()
```

When processing the data, the corresponding attributes will be automatically changed if necessary:

```
>>> sec.data_type
'strain rate'
>>> sec.time_integration()
>>> sec.data_type
'strain'
```

1.4.2 Filtering

DASPy contains commonly used filtering methods, including `bandpass` , `bandstop` , `lowpass` , `highpass` , `envelope` and *lowpass_cheby_2`* (mainly used for low-pass filtering before downsampling):

```
>>> sec.bandpass(1, 15)
```

1.4.3 Frequency Attribute

The frequency domain attribute analysis supported by DASPy includes spectrum (x-t), spectrogram (f-t) and frequency-wavenumber spectrum (f-k), which are calculated using `spectrum`, `spectrogram` and `fk-transform` respectively. method calculation. `spectrogram` calculates the average spectrogram of all channels by default. You can use the `xmin` and `xmax` parameters to limit the starting and ending channels of the average spectrum:

```
>>> spec, f = sec.spectrum()
>>> Zxx, f, t = sec.spectrogram(xmin=2600, xmax=2620)
>>> fk, f, k = sec.fk_transform()
```

If you only need to plot these three frequency domain spectra without outputting the calculation results, you can directly use the `plot` method. For details, see *Visualization* .

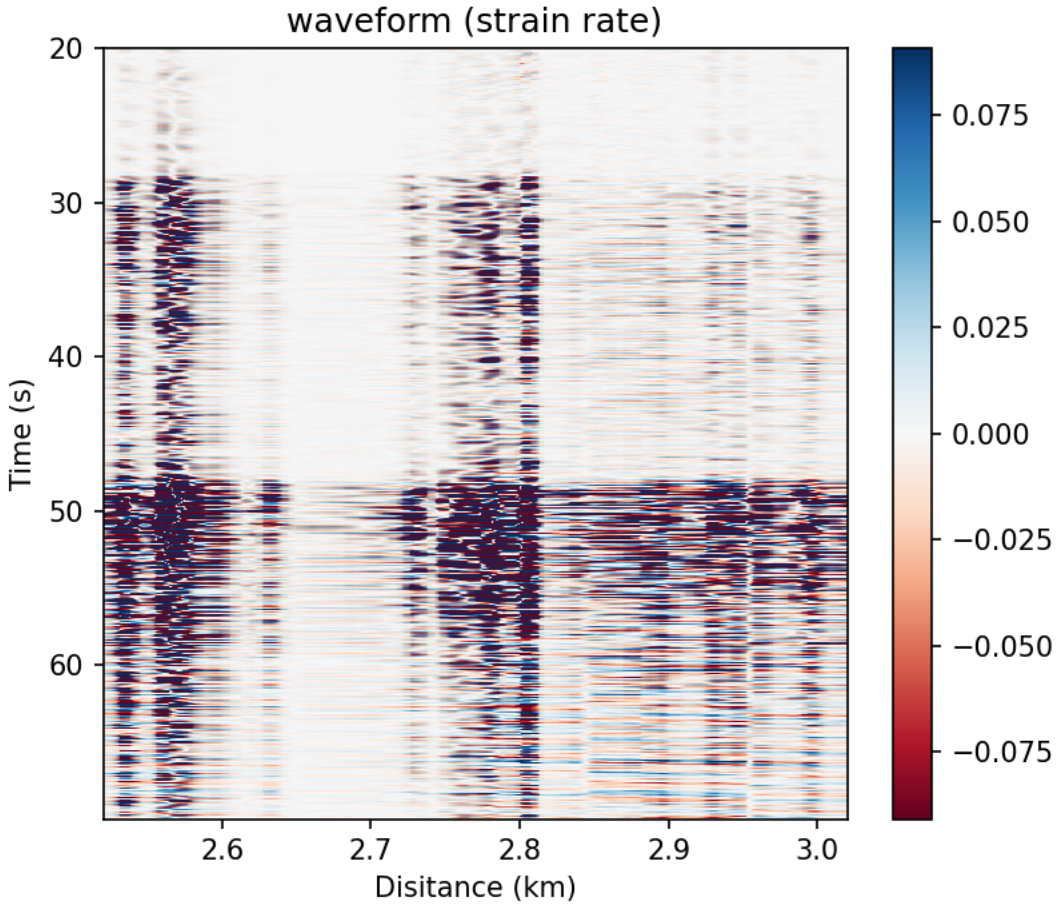
1.5 Visualization

DASPy uses the `Section` class method/function `plot` to plot images.

1.5.1 Call Class Method

Plot and show image

```
>>> from daspy import read
>>> sec = read()
>>> sec.plot()
```

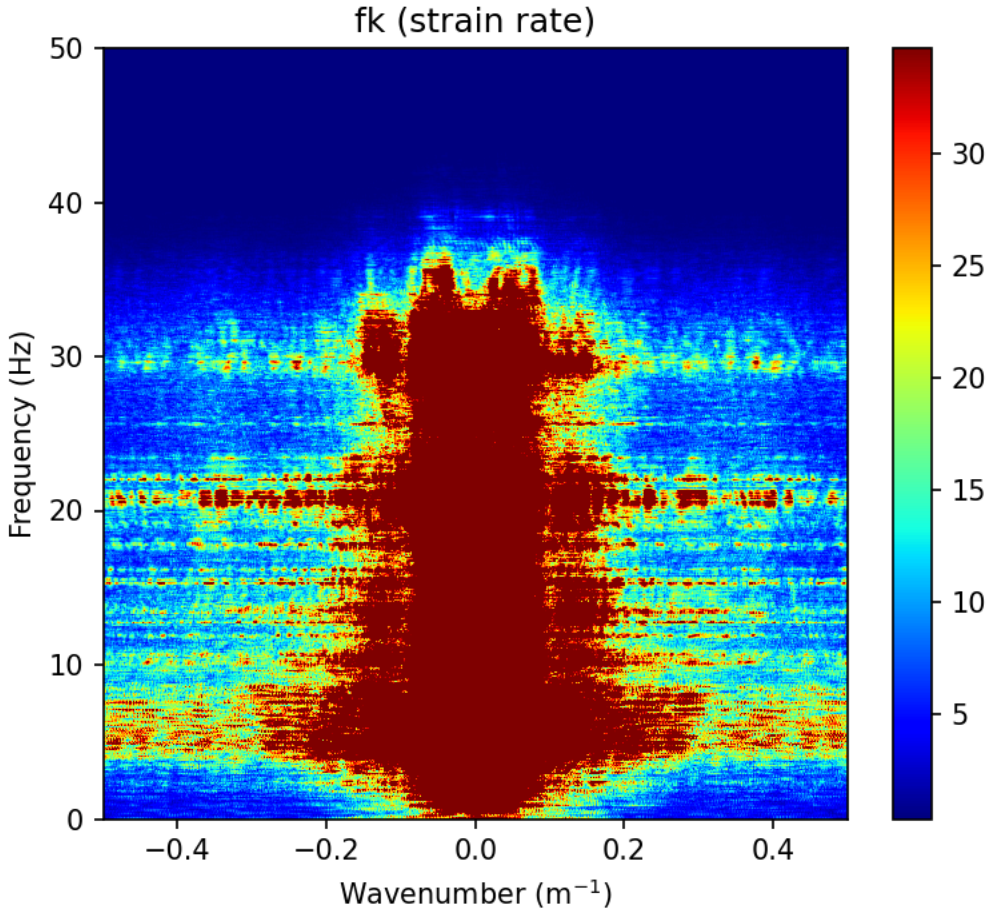


Adding `savefig` to save figure as specified filename, and `dpi` to set the resolution of the figure in dots-per-inch:

```
>>> sec.plot(savefig='waveform.png', dpi=400)
```

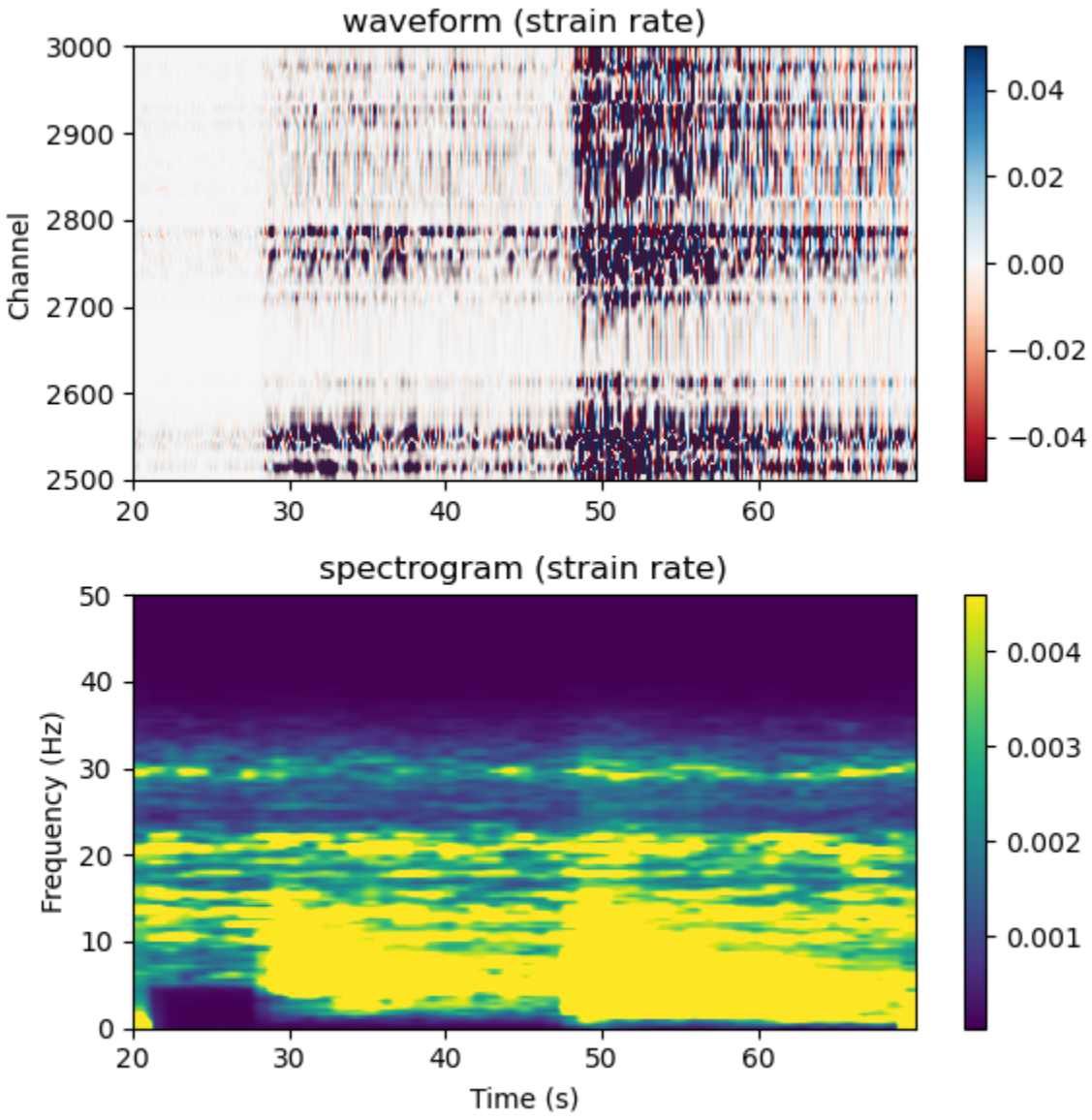
When drawing frequency domain graphs, you can set the `kwargs_pro` parameter to specify how the spectrum is plotted:

```
>>> sec.plot(obj='fk', kwargs_pro=dict(taper=(0.02, 0.05), nfft=(1024, 8192))) # set␣
↪the coefficient of 2D cosine taper to (0.02,0.05), output points of 2DFFT to (1024,␣
↪8192)
```



Plot in `Matplotlib.axes.Axes`:

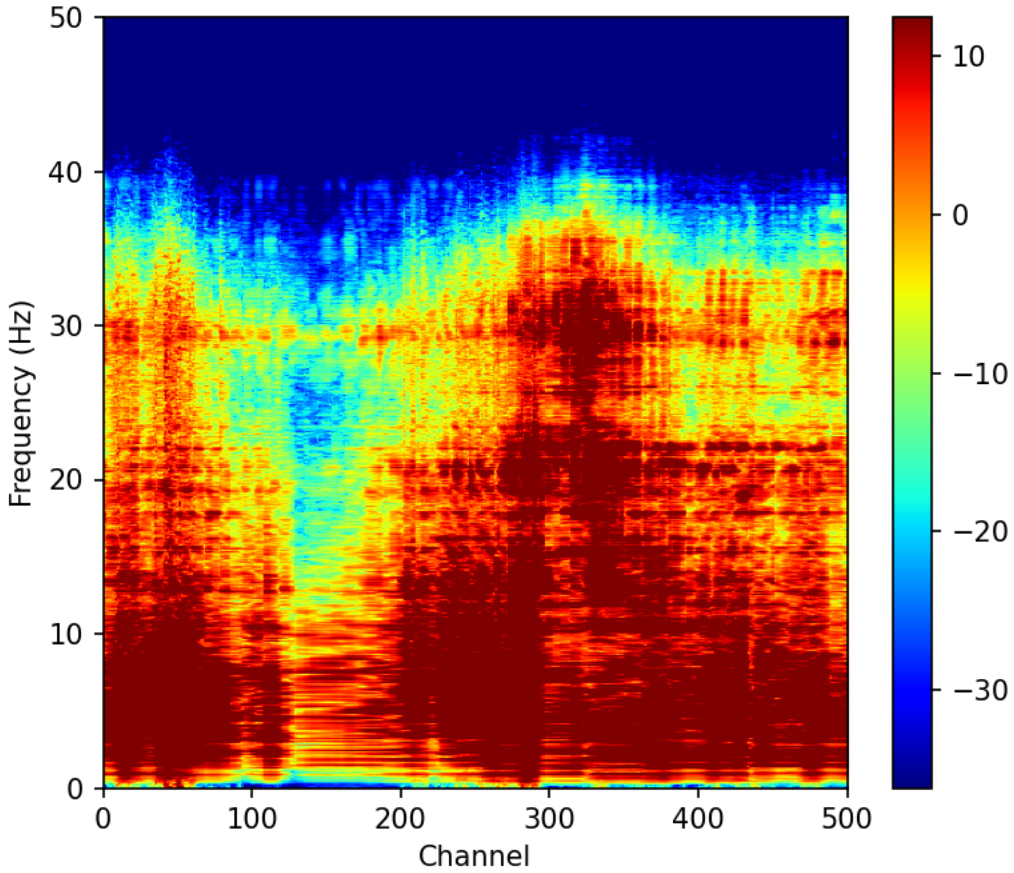
```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(2, 1, figsize=(6,6))
>>> sec.plot(ax=ax[0], obj='waveform', xmode='channel', tmode='origin', xlabel=False,␣
↪transpose=True, vmax=0.05) # set the spatial axis to the channel number, the time␣
↪axis to the time after the event occurred, do not draw the x-axis label, invert the␣
↪default x/y axis, and set the data range to -0.05~0.05
>>> sec.plot(ax=ax[1], obj='spectrogram', tmode='origin', kwargs_
↪pro=dict(noverlap=156)) # overlap between two windows is 156 points
>>> plt.tight_layout()
>>> plt.show()
```



1.5.2 Call the function

First calculate the spectrum, perform other calculations on the output, and then use the `daspy.basic_tools.visualization.plot` function to plot:

```
>>> import numpy as np
>>> from daspy.basic_tools.visualization import plot
>>> spec, f = sec.spectrum()
>>> spec = 10 * np.log10(abs(spec) ** 2) # convert the spectrum to units of decibels␣
↪(dB), using 1 as the reference value
>>> plot(spec, obj='spectrum', f=f, xmode='channel') # set the spatial axis to the␣
↪channel number, the time axis to the time after the event occurred, and invert the␣
↪default x/y axis
```



1.6 Channel Analysis

DASPy's channel attribute analysis module: *Location Interpolation*, *Turning Point Detection*, *Channel Quality Checking*.

1.6.1 Location Interpolation

DASPy's channel position interpolation supports the joint use of two types of data: known points containing channel numbers (i.e. longitude & latitude & channel number) and track points without channel numbers (ie longitude & latitude), where the known points are necessary, and track points are optional to constrain the cable geometry. DASPy using Universal Transverse Mercator projection (UTM), interpolate in the plane coordinate system and back-project back to the WGS84 Coordinate System □

Note: The example data is coordinates of the North cable of the RAPID dataset, which can be read online via:

```
>>> import numpy as np
>>> txt_url = 'http://piweb.ooirsn.uw.edu/das/processed/metadata/Geometry/OOI_RCA_DAS_
↪channel_location/north_cable_latlon.txt'
```

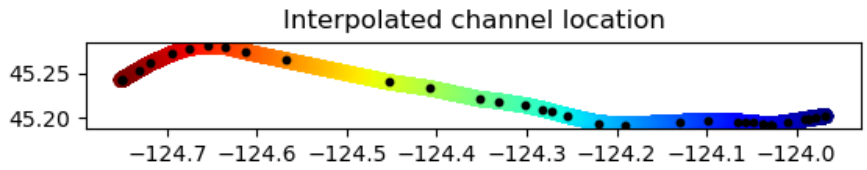
(continues on next page)

(continued from previous page)

```
>>> track_pt = np.loadtxt(txt_url)[:, ::-1] # read in the track points and swap the␣
↪two columns (let longitude precedes latitude)
>>> known_pt = np.array([[*track_pt[0], 942], [*track_pt[-1], 32459]]) # the 0th␣
↪track point corresponds to channel 942, the last track point corresponds to channel␣
↪32459, and the channel numbers of the remaining track points are unknown
```

Or read after downloading from http://piweb.ooirsn.uw.edu/das/processed/metadata/Geometry/OOI_RCA_DAS_channel_location/north_cable_latlon.txt.

```
>>> from daspy.advanced_tools.channel import location_interpolation
>>> interp_ch = location_interpolation(known_pt, track_pt=track_pt, dx=2) # input the␣
↪track points and the known number points, the channel spacing is 2m, and get the␣
↪interpolation result
>>> print(interp_ch) # longitude, latitude, and channel number
[[ -123.96715       45.2023       942.        ]
 [ -123.96717541    45.20229612   943.        ]
 [ -123.96720083    45.20229224   944.        ]
 ...
 [ -124.75097192    45.24242661 32457.        ]
 [ -124.75099513    45.2424183  32458.        ]
 [ -124.75101833    45.24241    32459.        ]]
>>>
>>> import matplotlib.pyplot as plt # plotting
>>> plt.scatter(interp_ch[:,0], interp_ch[:,1], c=interp_ch[:,2], cmap='jet')
>>> plt.scatter(track_pt[:,0], track_pt[:,1], c='k', s=10)
>>> plt.gca().set_aspect('equal')
>>> plt.title('Interpolated channel location')
>>> plt.show()
```



1.6.2 Turning Point Detection

DASPy detect turning points detection through two methods: calculating the corner of the fiber through the channel coordinates, or finding channels with low adjacent channel cross-correlation values in the waveform record.

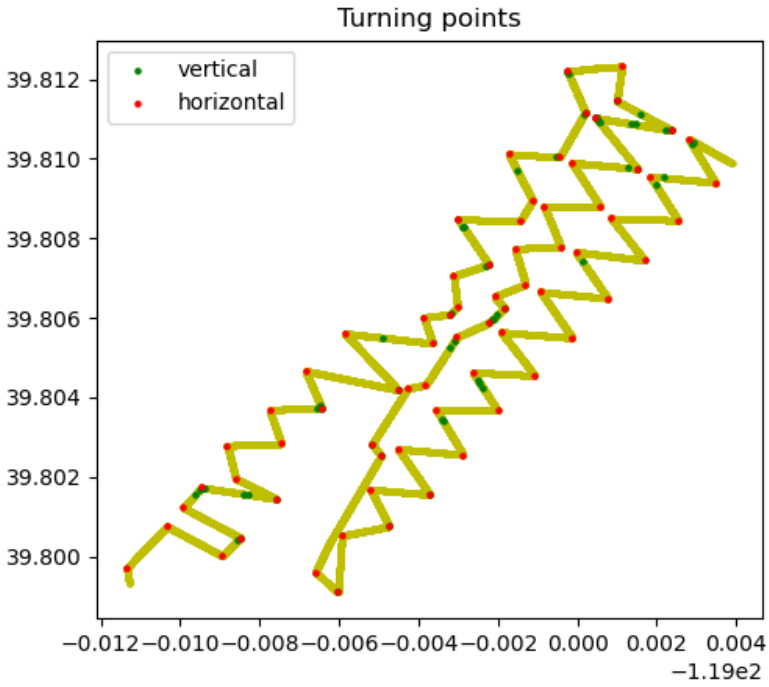
Calculated by channel latitude and longitude:

Note: The example data is the DAS channel coordinates of the Brady geothermal field, which can be read online via:

```
>>> import numpy as np
>>> csv_url = 'https://raw.githubusercontent.com/HMZ-03/DASPy-data/main/Brady_DAS_
↪Coordinates.csv'
>>> geometry = np.loadtxt(csv_url, delimiter=',', skiprows=2, usecols=[5,4,3])
>>> geometry = geometry[geometry[:,0] != 0] # exclude empty records
```

or read after downloading from https://raw.githubusercontent.com/HMZ-03/DASPy-data/main/Brady_DAS_Coordinates.csv.

```
>>> from daspy.advanced_tools.channel import turning_points
>>> turning_h, turning_v = turning_points(geometry, depth_info=True) # the data␣
↪contains depth information, detect turning points both horizontally and vertically
>>>
>>> import matplotlib.pyplot as plt # plotting
>>> plt.scatter(geometry[:, 0], geometry[:, 1], c='y', s=5)
>>> plt.scatter(geometry[turning_v, 0], geometry[turning_v, 1], c='g', s=5, label=
↪'vertical')
>>> plt.scatter(geometry[turning_h, 0], geometry[turning_h, 1], c='r', s=5, label=
↪'horizontal')
>>> plt.gca().set_aspect('equal')
>>> plt.title('Turning points')
>>> plt.legend()
>>> plt.show()
```



1.6.3 Channel Quality Checking

Sometimes there are areas with poor coupling conditions along optical fiber, such as reserved loops in communication optical cables, resulting in "bad channels". These "bad channels" usually correspond to areas on the waveform with abnormally low or abnormally high amplitudes. When the coupling conditions are unknown, DASPy can use a DAS record to check the channel quality and determine the so-called "bad channels":

Note: The sample data is a 15-second traffic signal recorded by Ridgecrest DAS, which can be downloaded from https://data.caltech.edu/records/31emd-wmv98/files/Traffic_noise_figure_4.mat?download=1 and read via the following method:

```
>>> import scipy.io as scio
>>> data = scio.loadmat('Traffic_noise_figure_4.mat')['Traffic_noise_figure_4'].T
>>> sec = Section(data, 8, 250)
```

Call a function to check good and bad channels:

```
>>> from daspy.advanced_tools.channel import channel_checking
>>> good_chn, bad_chn = channel_checking(data)
>>> print(bad_chn)
[  11   12   13   14   18   19   20   21   22   23   81   82   83   84
85   86   87   88   89  142  143  144  145  146  255  256  257  258
259  260  261  262  263  264  265  266  267  268  269  270  454  455
456  457  458  459  460  461  462  463  464  465  466  467  468  469
470  471  472  664  665  666  667  668  669  842  843  844  845  846
847  848  849  850  851  852  853  854  855  856  857  858  859  860
861  862  863  864  865  866  867  868  869  870  871 1059 1060 1061
1062 1063 1064 1065 1066]
```

Directly remove bad channels in the `daspy.Section` instance:

```
>>> sec.channel_checking(use=True)
```

1.7 Denoising

The noise in DAS signals mainly includes spike noise, common mode noise, stochastic noise and coherent noise. DASPy's data denoising module provides three functions: *Spike Removal*, *Common Mode Noise Removal* and *Stochastic Noise Removal*. Coherent noise can be removed by methods provided in *Wavefield Decomposition*.

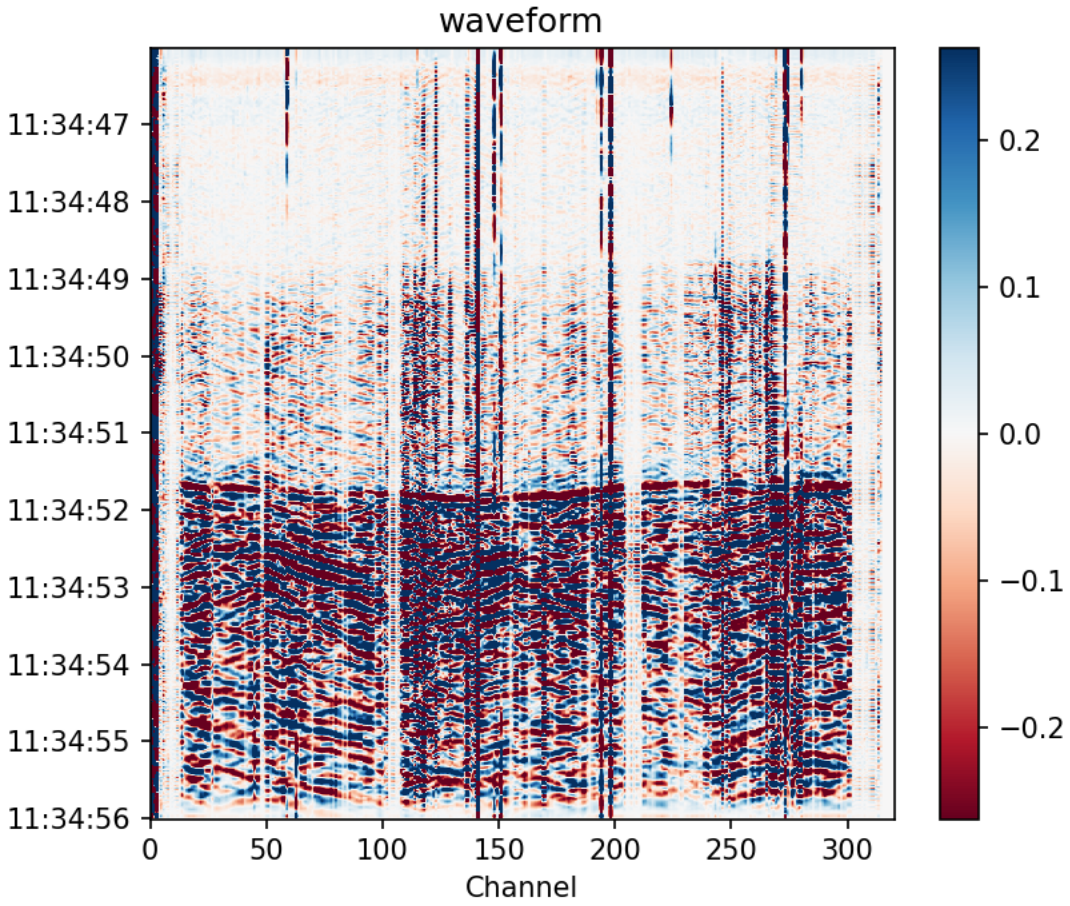
1.7.1 Spike Removal

Spikes are unusually large amplitudes and could be caused by laser frequency drift or laser noise. The spike removal function first applies the across-channel median filter and then the across-time median filter to generate a median map from the absolute amplitudes. Points with amplitudes exceeding a predefined threshold of the median map are identified as spikes. All spikes are subsequently substituted with interpolated values from adjacent channels.

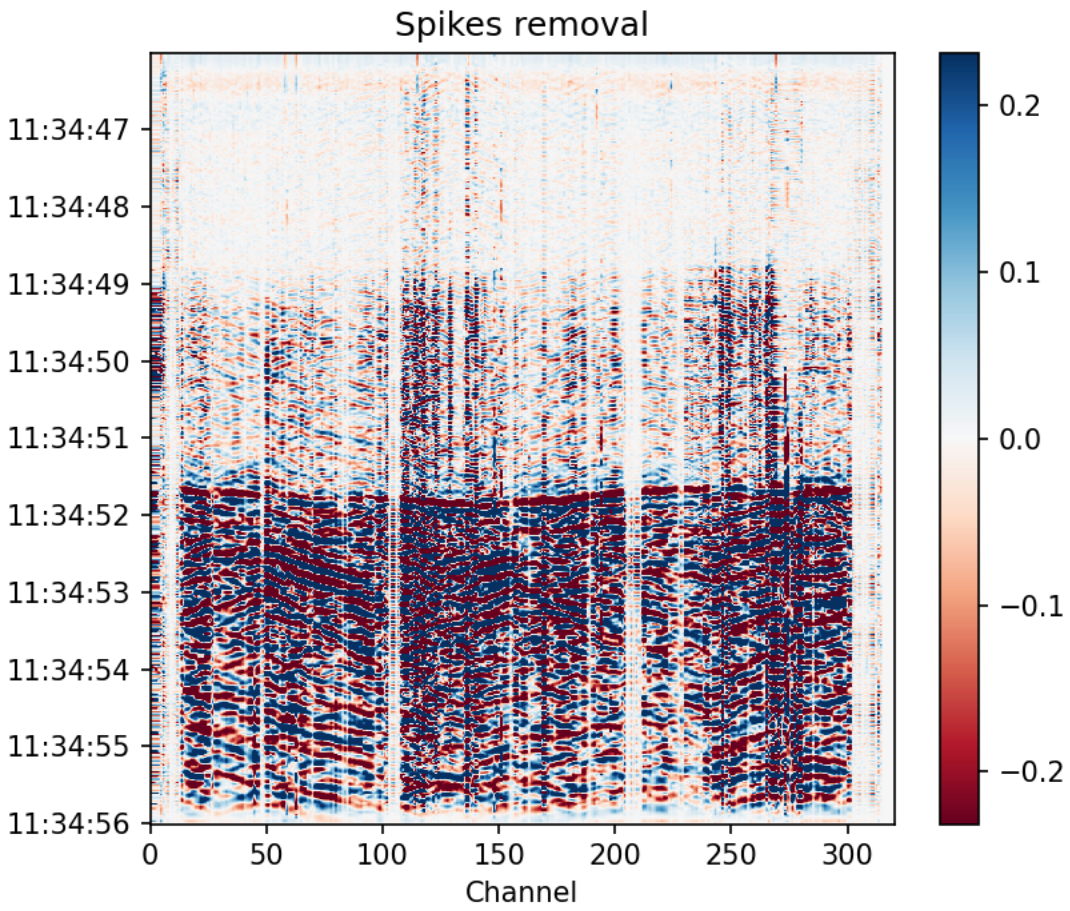
Note: The example data is the seismic signal recorded by Stanford DAS-1, which can be downloaded from https://raw.githubusercontent.com/HMZ-03/DASPy-data/main/Stanford1_20180104_113058.566+0000.sgy , read and preprocess via:

```
>>> from daspy import read, DASDateTime
>>> sec = read('Stanford1_20180104_113058.566+0000.sgy', ch2=320)
UserWarning: This data format segy doesn't include channel interval.
Please set Section.dx manually.
>>> sec.dx = 8
>>> sec.start_time = DASDateTime(2018, 1, 4, 11, 30, 58, 566000)
>>> origin_time = DASDateTime(2018, 1, 4, 11, 34, 44)
>>> sec.bandpass(1, 20)
>>> sec.trimming(tmin=origin_time+2, tmax=origin_time+12)
```

```
>>> sec.plot(xmode='channel')
```




```
>>> sec.spike_removal()
>>> sec.plot(xmode='channel')
```



1.7.2 Common Mode Noise Removal

Common-mode noise, also known as in-phase noise is generated by vibrations of the optoelectronic system and arises on all channels simultaneously. DASPy employs spatial median or mean of waveforms to obtain common mode noise. Subsequently, we compute the correlation coefficient with the channel record and the common-mode noise, multiply the common-mode noise by the coefficient, and subtract it from the channel record.

Note: The example data is the waveforms recorded by channels away from the coast of the RAPID dataset, which can be downloaded from http://piweb.ooirsn.uw.edu/das/data/Optasense/NorthCable/TransmitFiber/North-C1-LR-P1kHz-GL50m-Sp2m-FS200Hz_2021-11-03T15_06_51-0700/North-C1-LR-P1kHz-GL50m-Sp2m-FS200Hz_2021-11-04T015902Z.h5 , read and preprocess via:

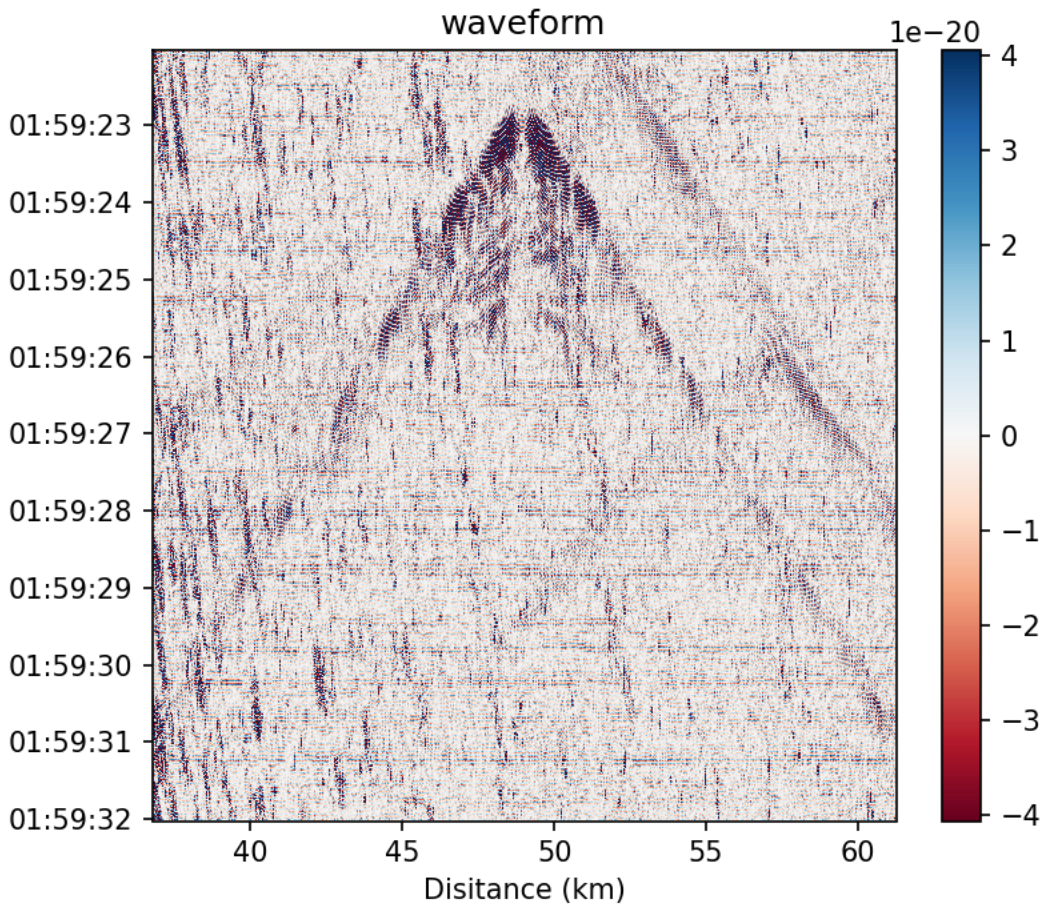
```
>>> import numpy as np
>>> from daspy import read
>>> sec = read('North-C1-LR-P1kHz-GL50m-Sp2m-FS200Hz_2021-11-04T015902Z.h5')
>>> sec.trimming(mode=0, xmin=18000, xmax=30000)
>>> sec.scale = 2 * np.pi / 2 ** 16 # data scaling factor, see sec.headers[
↪'Acquisition']['Raw[0]']['attrs']['RawDataUnit']
>>> sec.phase2strain(1550.12 * 1e-9, 0.78, sec.headers['Acquisition']['Custom']['attrs
↪']['Fibre Refractive Index']) # convert optical phase shift to strain
```

(continues on next page)

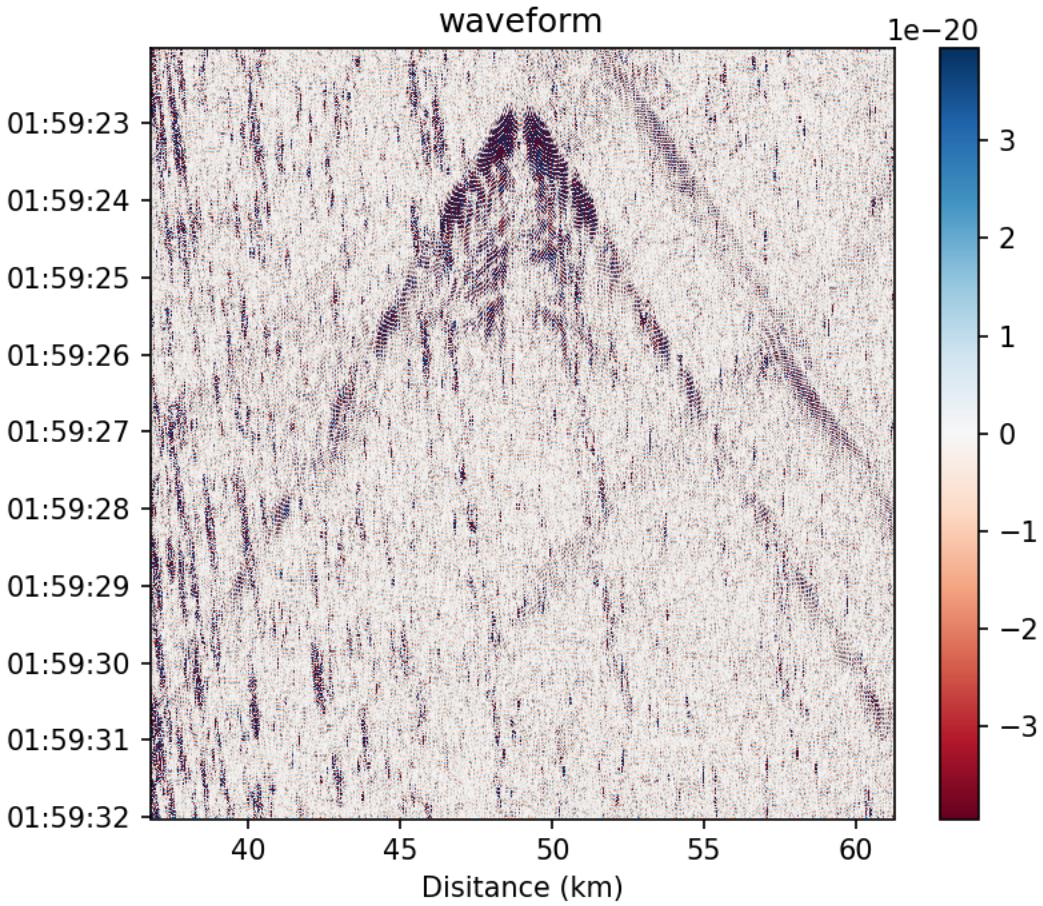
(continued from previous page)

```
>>> sec.bandpass(15, 27, detrend=True, taper=0.1)
>>> sec.trimming(tmin=sec.start_time+20, tmax=sec.start_time+30)
```

```
>>> sec.plot()
```




```
>>> sec.common_mode_noise_removal()
>>> sec.plot()
```



1.7.3 Stochastic Noise Removal

The inherent stochastic noise in DAS data is primarily caused by instrumental deficiencies such as sampling error and phase noise. DASPy remove stochastic noise by curvelet transform.

Note: Consistent with the example data used by *Spike Removal* . The waveform with spikes removed is used here. Read and preprocess via:

```
>>> from daspy import read, DASDateTime
>>> sec = read('Stanford1_20180104_113058.566+0000.sgy', ch2=320)
UserWarning: This data format segy doesn't include channel interval.
Please set Section.dx manually.
>>> sec.dx = 8
>>> sec.start_time = DASDateTime(2018, 1, 4, 11, 30, 58, 566000)
>>> sec.bandpass(1, 20)
>>> origin_time = DASDateTime(2018, 1, 4, 11, 34, 44)
>>> sec.trimming(tmin=origin_time-10, tmax=origin_time+12)
>>> sec.spike_removal()
```

Use a noise record as the noise baseline and remove the noise with a soft threshold (default) in the curvelet domain:

```
>>> sec_eq = sec.copy().trimming(tmin=origin_time+2, tmax=origin_time+12) #␣
↪earthquake records
>>> sec_ns = sec.copy().trimming(tmin=origin_time-10, tmax=origin_time) # noise␣
↪records
>>> sec_eq_soft = sec_eq.copy().curvelet_denoising(noise=sec_ns)
```

Also using the reference noise record, removing the noise with a hard threshold in the curvelet domain can keep the absolute amplitude of the waveform unchanged and cause little distortion:

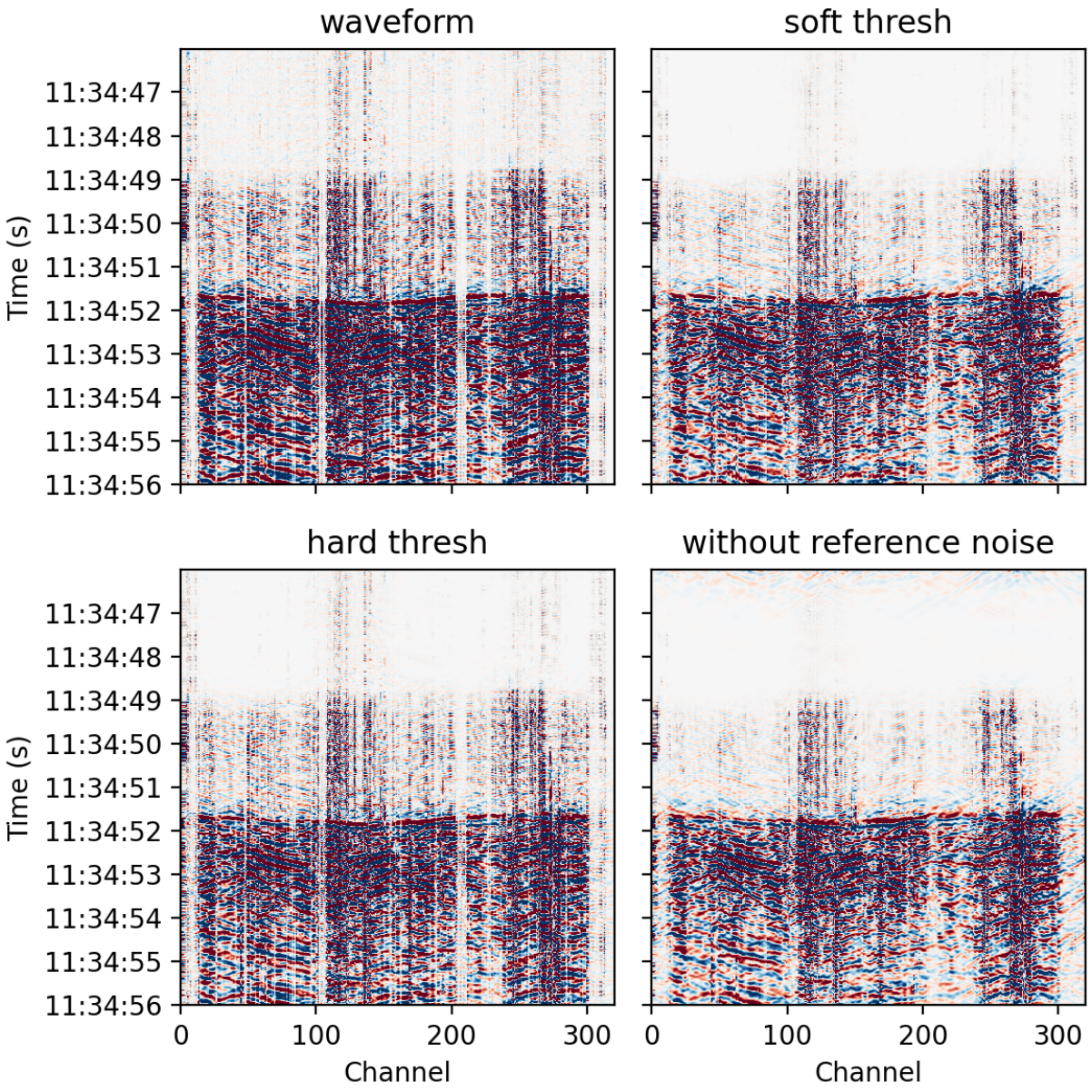
```
>>> sec_eq_hard = sec_eq.copy().curvelet_denoising(noise=sec_ns, soft_thresh=False)
```

When there is no reference noise record available, the function will calculate the inflection points of the curvelet coefficients to determine the noise threshold. It is recommended to set `pad=0` and adjust the `knee_fac` parameter to reduce artificial artifacts (this method is not recommended) :

```
>>> sec_eq_knee = sec_eq.copy().curvelet_denoising(pad=0, knee_fac=0.1)
```

Plot the original waveform and the above three denoising effects:

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(2, 2, figsize=(6,6), sharex=True, sharey=True, dpi=200)
>>> sec_eq.plot(ax=ax[0,0], xmode='channel', vmax=0.2, xlabel=False, colorbar=False)
>>> sec_eq_soft.plot(ax=ax[0,1], xmode='channel', vmax=0.2, xlabel=False,␣
↪ylabel=False, colorbar=False, title='soft thresh')
>>> sec_eq_hard.plot(ax=ax[1,0], xmode='channel', vmax=0.2, colorbar=False, title=
↪'hard thresh')
>>> sec_eq_knee.plot(ax=ax[1,1], xmode='channel', vmax=0.2, ylabel=False,␣
↪colorbar=False, title='without reference noise')
>>> plt.tight_layout()
>>> plt.show()
```



1.8 Wavefield Decomposition

The wavefield recorded by DAS usually has complex components. We often need to separate the wavefield according to apparent velocity to extract some of the required signals, such as separating seismic signals and traffic signals, separating direct wave and scattered wave, etc. Due to the good equidistant characteristics of DAS, the wavefield recorded by DAS can be quickly and easily decomposed using image processing technology. DASPy uses two-dimensional fast Fourier transform (called FK transform in seismic data processing, see *FK Filtering*) and fast discrete curvelet transform technology (see *Curvelet Windowing Technology*) to decompose the DAS wavefield.

Note: The example data for this Section is the 30-second M1.79 earthquake signal recorded by Ridgecrest DAS, which can be downloaded from https://data.caltech.edu/records/31emd-wmv98/files/EQ_raw_figure_1.mat?download=1,

read and preprocess via:

```
>>> import scipy.io as scio
>>> from daspy import Section
>>> data = scio.loadmat('EQ_raw_figure_1.mat')['EQ_raw'].T # read the data
>>> sec = Section(data, 8, 250) # build Section instance
>>> sec.spike_removal() # remove spikes
>>> sec.channel_checking(use=True) # remove bad channels
```

1.8.1 FK Filtering

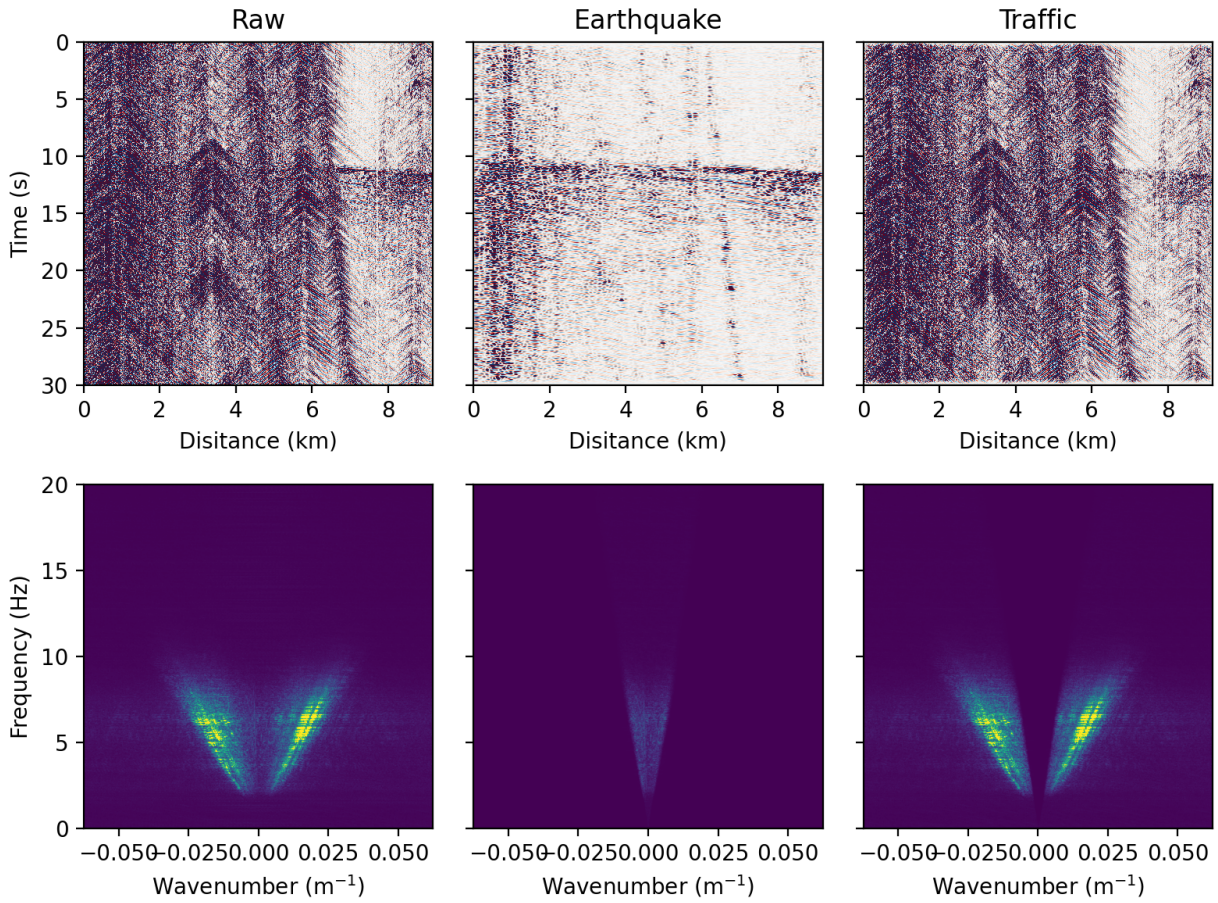
The FK filtering method uses two-dimensional fast Fourier transform to convert the data into the frequency-wavenumber (FK) domain, multiplies the FK domain by the filter window and then inversely transforms it back to the time-distance domain. If the apparent wave speed of traffic signals in the area is less than 1000m/s and the apparent wave speed of seismic waves is greater than 1000m/s, you can use `vmin=(800, 1200)` to filter out the traffic signals. The first output is seismic waves, and the second outputs are traffic signals. `800` and `1200` are the taper boundaries:

```
>>> sec_eq, sec_tf = sec.fk_filter(vmin=(800, 1200), mode='decompose')
```

FK filtering often causes various artifacts, particularly edge artifacts caused by discontinuities at the waveform's edge, and star-like artifacts originating from discontinuities in the FK domain. To minimize these artifacts, DASPy employs cosine tapers (e.g. Tukey window) on the waveforms, as well as the filtering window in the FK domain.

Plot the waveforms and FK spectra of the original waveform, the separated direct wave and the scattered wave:

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(3, 2, figsize=(6,6), sharex='col', sharey='col', dpi=200)
>>> sec.plot(ax=ax[0,0], xlabel=False)
>>> sec_dr.plot(ax=ax[1,0], xlabel=False)
>>> sec_sc.plot(ax=ax[2,0])
>>> from daspy.basic_tools.visualization import plot
>>> vmin, vmax = 0, 2e5
>>> plot(fk, obj='fk', f=f, k=k, ax=ax[0,1], vmin=vmin, vmax=vmax,xlabel=False)
>>> plot(fk*mask, obj='fk', f=f, k=k, ax=ax[1,1], vmin=vmin, vmax=vmax, xlabel=False)
>>> plot(fk*(1-mask), obj='fk', f=f, k=k, ax=ax[2,1], vmin=vmin, ylim=[0,30],␣
↪vmax=vmax)
>>> plt.tight_layout()
>>> plt.show()
```



1.8.2 Curvelet Windowing Technology

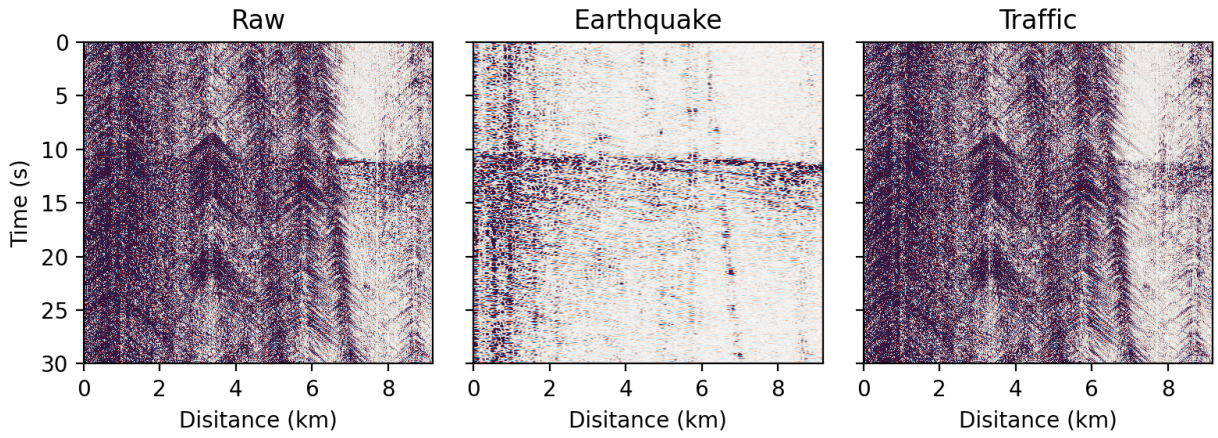
The curvelet transform can similarly achieve the effect of decomposing the wave field at the apparent wave velocity.

Separate seismic waves and traffic signals at a speed of 1000m/s:

```
>>> sec_eq, sec_tf = sec.curvelet_windowing(mode='decompose', vmin=1000)
```

Plot the original waveform, the separated direct wave, and the scattered wave:

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 3, figsize=(8,3), sharex='row', sharey='row', dpi=200)
>>> plot_kwargs = dict(vmax=1, colorbar=False)
>>> sec.plot(ax=ax[0], title='Raw', **plot_kwargs)
>>> sec_eq.plot(ax=ax[1], title='Earthquake', ylabel=False, **plot_kwargs)
>>> sec_tf.plot(ax=ax[2], title='Traffic', ylabel=False, **plot_kwargs)
>>> plt.tight_layout()
>>> plt.show()
```



1.9 Strain-velocity Conversion

DAS measures strain or strain rate, whereas traditional seismological studies typically use displacement, velocity or acceleration. Strain can be converted to particle velocity by multiplying the apparent phase velocity. Based on this principle, DASPy integrates three methods for converting strain/strain rate into velocity/acceleration (hereinafter collectively referred to as strain-velocity conversion): *FK Rescaling Method*, *Curvelet Transform Method* and *Time-Domain Slowness Detection Method*.

Note: The example data for this section is the M4.3 earthquake recorded by a section of DAS in the Brady geothermal experimental field, which can be downloaded from https://raw.githubusercontent.com/HMZ-03/DASPy-data/main/Brady_DAS_M4.3_2688_2826.pkl; The data recorded at the same time by No.165 node seismometer for comparison can be downloaded from https://raw.githubusercontent.com/HMZ-03/DASPy-data/main/Brady_node_M4.3_165.sac. The distance between DAS channel 2781 and the seismometer at this node is 2.6 meters. The preprocessing that has been done on the DAS data includes: intercepting a certain linear segment (channels 2688 to 2826), removing the timing error (1.048 seconds), integrating the strain rate into strain, downsampling from 1000Hz to 500Hz, and bandpass filtering to 1~5Hz, trimmed from 19.99 seconds to 70.01 seconds after the earthquake origin (allowing a time difference of ±0.01 seconds when calculating cross-correlation with node seismometer data). The preprocessing that has been done on the node seismometer includes: rotating to the DAS axial component, filtering to 1~5Hz, and trimmed 20 seconds to 70 seconds after the earthquake occurs. Read via (requires ObsPy installed):

```
>>> import obspy
>>> from daspy import read
>>> st = obspy.read('Brady_node_M4.3_165.sac')
>>> sec = read('Brady_DAS_M4.3_2688_2826.pkl')
```

1.9.1 FK Rescaling Method

The FK rescaling method achieves strain-to-velocity conversion by multiplying each point in the FK domain by its corresponding apparent velocity (slope in the FK domain).

```
>>> sec_fk = sec.copy().fk_rescaling(fmax=(5, 6))
```

1.9.2 Curvelet Transform Method

The basis function of the curvelet transform appears as a wedge with a certain velocity range in the FK domain. The curvelet transform method multiplies each curvelet coefficient by the median velocity of its basis function to achieve strain-velocity conversion.

```
>>> sec_cv = sec.copy().curvelet_conversion()
```

1.9.3 Time-Domain Slowness Detection Method

The time domain slowness determination method obtains the apparent speed at each time step by searching for the maximum semblance.

```
>>> sec_ts = sec.copy().slant_stacking(L=10, frqlow=1, frqhigh=5, channel=2781) #␣
↪only convert channel 2781 and save into the Section instance for efficiency
```

1.9.4 Comparison of Three Methods

Define a function for calculating the cross-correlation coefficient and shift:

```
>>> import numpy as np
>>> def X_corr(a, b):
>>>     M, N = len(a), len(b)
>>>     b = (b - np.mean(b)) / np.std(b) / N
>>>     cc = np.zeros(M-N+1)
>>>     for i in range(M-N+1):
>>>         a_p = a[i:i+N]
>>>         a_p = (a_p - np.mean(a_p)) / np.std(a_p)
>>>         cc[i] = np.correlate(a_p, b, 'valid')[0]
>>>     return np.argmax(cc), max(cc)
```

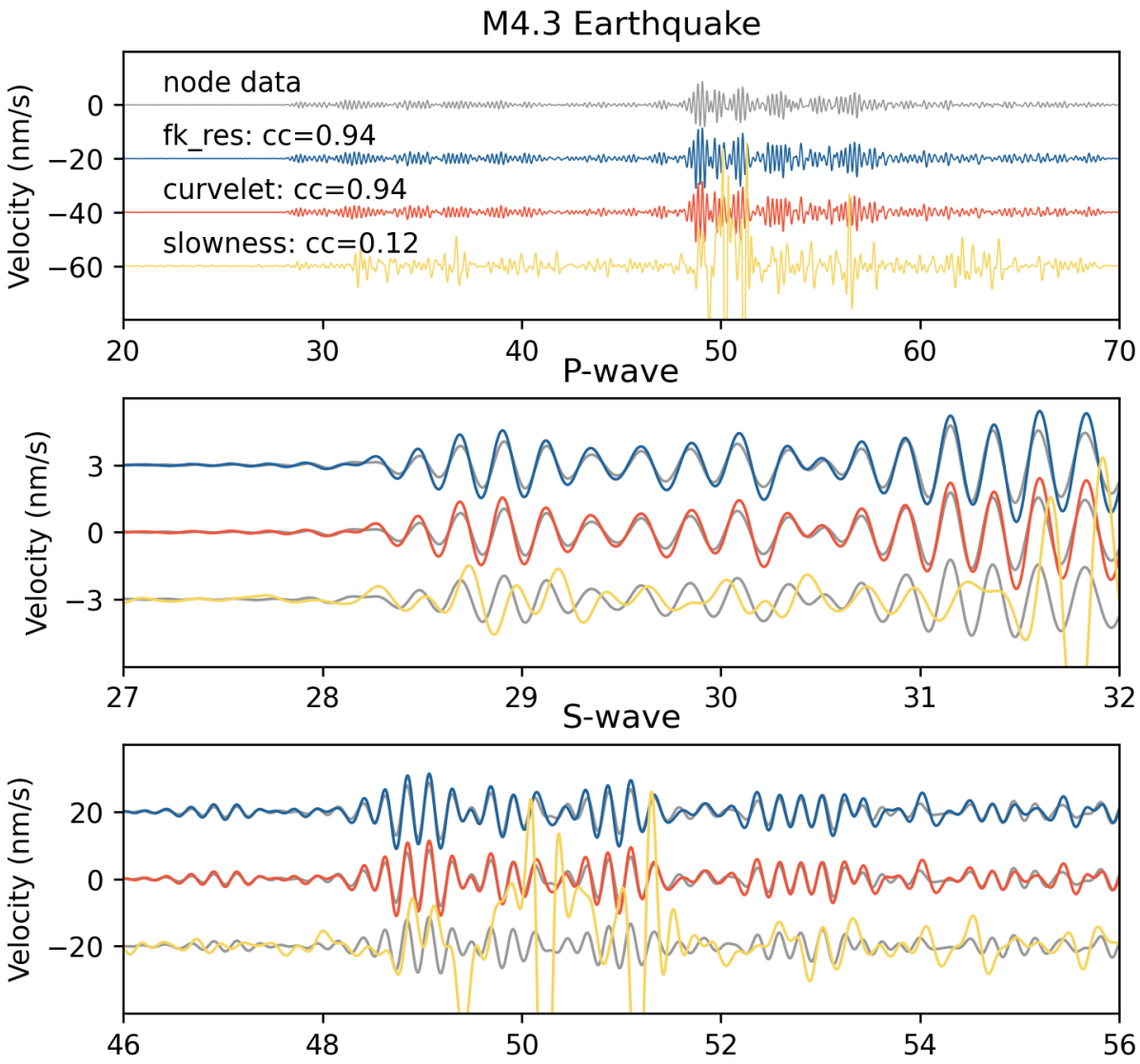
Plot the strain-velocity conversion results for the three methods:

```
>>> import matplotlib.pyplot as plt
>>> node_data = st[0].data
>>> nt = len(node_data)
>>> dt1, cc1 = X_corr(sec_fk.channel_data(2781), node_data)
>>> fk_data = sec_fk.channel_data(2781)[dt1:dt1+nt]
>>> dt2, cc2 = X_corr(sec_cv.channel_data(2781), node_data)
>>> cv_data = sec_cv.channel_data(2781)[dt2:dt2+nt]
>>> dt3, cc3 = X_corr(sec_ts.channel_data(2781), node_data)
>>> ts_data = sec_ts.channel_data(2781)[dt3:dt3+nt]
>>>
>>> fig, ax = plt.subplots(3, 1, figsize=(6, 8), dpi=250)
>>> t = np.arange(nt)/sec.fs + 20
```

(continues on next page)

(continued from previous page)

```
>>> c = ['#999999', '#20639B', '#ED553B', '#F6D55C']
>>> ax[0].set_title('M4.3 Earthquake')
>>> ax[0].plot(t, node_data, c=c[0], lw=0.5)
>>> ax[0].plot(t, fk_data-20, c=c[1], lw=0.5)
>>> ax[0].plot(t, cv_data-40, c=c[2], lw=0.5)
>>> ax[0].plot(t, ts_data-60, c=c[3], lw=0.5)
>>> ax[0].text(22, 5, f'node data')
>>> ax[0].text(22, -15, f'fk_res: cc={cc1:.2f}')
>>> ax[0].text(22, -35, f'curvelet: cc={cc2:.2f}')
>>> ax[0].text(22, -55, f'slowness: cc={cc3:.2f}')
>>> ax[0].set_xlim([20, 70])
>>> ax[0].set_ylim([-80, 20])
>>> ax[0].set_yticks(np.arange(-60, 20, 20))
>>> ax[0].set_ylabel('Velocity (nm/s)')
>>>
>>> ax[1].set_title('P-wave')
>>> ax[1].plot(t, node_data-3, c=c[0], lw=1)
>>> ax[1].plot(t, node_data, c=c[0], lw=1)
>>> ax[1].plot(t, node_data+3, c=c[0], lw=1)
>>> ax[1].plot(t, fk_data+3, c=c[1], lw=1)
>>> ax[1].plot(t, cv_data, c=c[2], lw=1)
>>> ax[1].plot(t, ts_data-3, c=c[3], lw=1)
>>> ax[1].set_xlim([27, 32])
>>> ax[1].set_ylim([-6, 6])
>>> ax[1].set_yticks(np.arange(-3, 6, 3))
>>> ax[1].set_ylabel('Velocity (nm/s)')
>>>
>>> ax[2].set_title('S-wave')
>>> ax[2].plot(t, node_data+20, c=c[0], lw=1)
>>> ax[2].plot(t, node_data, c=c[0], lw=1)
>>> ax[2].plot(t, node_data-20, c=c[0], lw=1)
>>> ax[2].plot(t, fk_data+20, c=c[1], lw=1)
>>> ax[2].plot(t, cv_data, c=c[2], lw=1)
>>> ax[2].plot(t, ts_data-20, c=c[3], lw=1)
>>> ax[2].set_xlim([46, 56])
>>> ax[2].set_ylim([-40, 40])
>>> ax[2].set_yticks(np.arange(-20, 40, 20))
>>> ax[2].set_ylabel('Velocity (nm/s)')
>>>
>>> plt.tight_layout()
>>> plt.show()
```



If you have any questions, please contact mailto:hmz2018@mail.ustc.edu.cn.