Python Proxy

(Revised 05.01.2023)

RADEXPRO EUROPE OÜ
Järvevana tee 9-40
11314 Tallinn, ESTONIA

RADEXPRO SEISMIC SOFTWARE LLC
29, Tornike Eristavi str.
Tbilisi, Georgia

Visiting address:
26, S. Tsintsadze str.
Saburtalo, Tbilisi, Georgia

t: +995 557 659 289
www.radexpro.com

E-mails:
support@radexpro.com
sales@radexpro.com

Content

Introduction

This tutorial is for the users who aim to expand the functionality of **RadExPro** by introducing their own algorithms in Python via the Python Proxy module. The tutorial contains several example implementations of seismic processing algorithms in Python Proxy, including automatic gain control, deghosting, and semblance attribute computation.

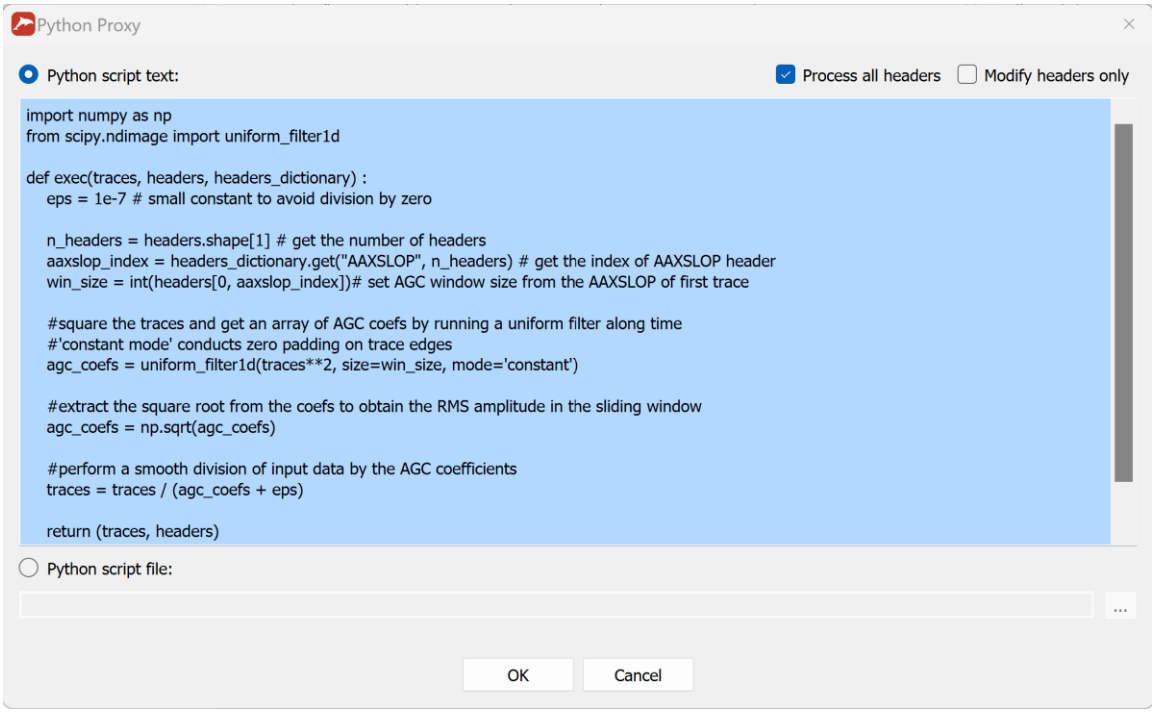
The tutorial assumes that the user is familiar with the fundamentals of Python processing language. Before starting the tutorial, it is recommended to read the section related to the Python Proxy in the **RadExPro** manual. Note that for Python Proxy to function correctly it is preferred that the user installs Python on their system before installing **RadExPro**.

The tutorial is accompanied by a project where each of the presented algorithms can be tested on seismic data.

Automatic Gain Control (AGC)

For simplicity, we start with an implementation of AGC in Python Proxy. In the tutorial project, the data input for this section of the tutorial and the following semblance computation occurs in the **010 data input** flow.

Here is the example Python Proxy window. We write the processing function exactly in this window. Note the **Process all headers** option is checked, as here we want to pass all headers from the input seismic dataset to the output.



Code

The annotated code for the AGC algorithm is presented below. The comments in the code (text following the comment symbol #) explain what happens in each line of code.

Here, we use the ***uniform_filter1d*** function from the ***scipy*** library, so ***scipy*** needs to be installed on your computer (which can be done using pip, similar to the installation of ***numpy*** in the Python Proxy manual). This function implements a running mean filter along a specified axis. We run this uniform filter on squared seismic data along the time axis (which is set by the ***axis*=1** parameter). Note that the ***uniform_filter1d*** takes care of the padding and returns an array of the same shape as input.

After this, we extract a square root from the result of the running mean to obtain the RMS amplitudes in a sliding window which serve as our AGC coefficients. Finally, we divide the 2D seismic data array ***traces*** by the array of AGC coefficients ***agc_coefs***. One can return ***agc_coefs*** instead of ***traces*** to conduct quality control of the resulting coefficients.

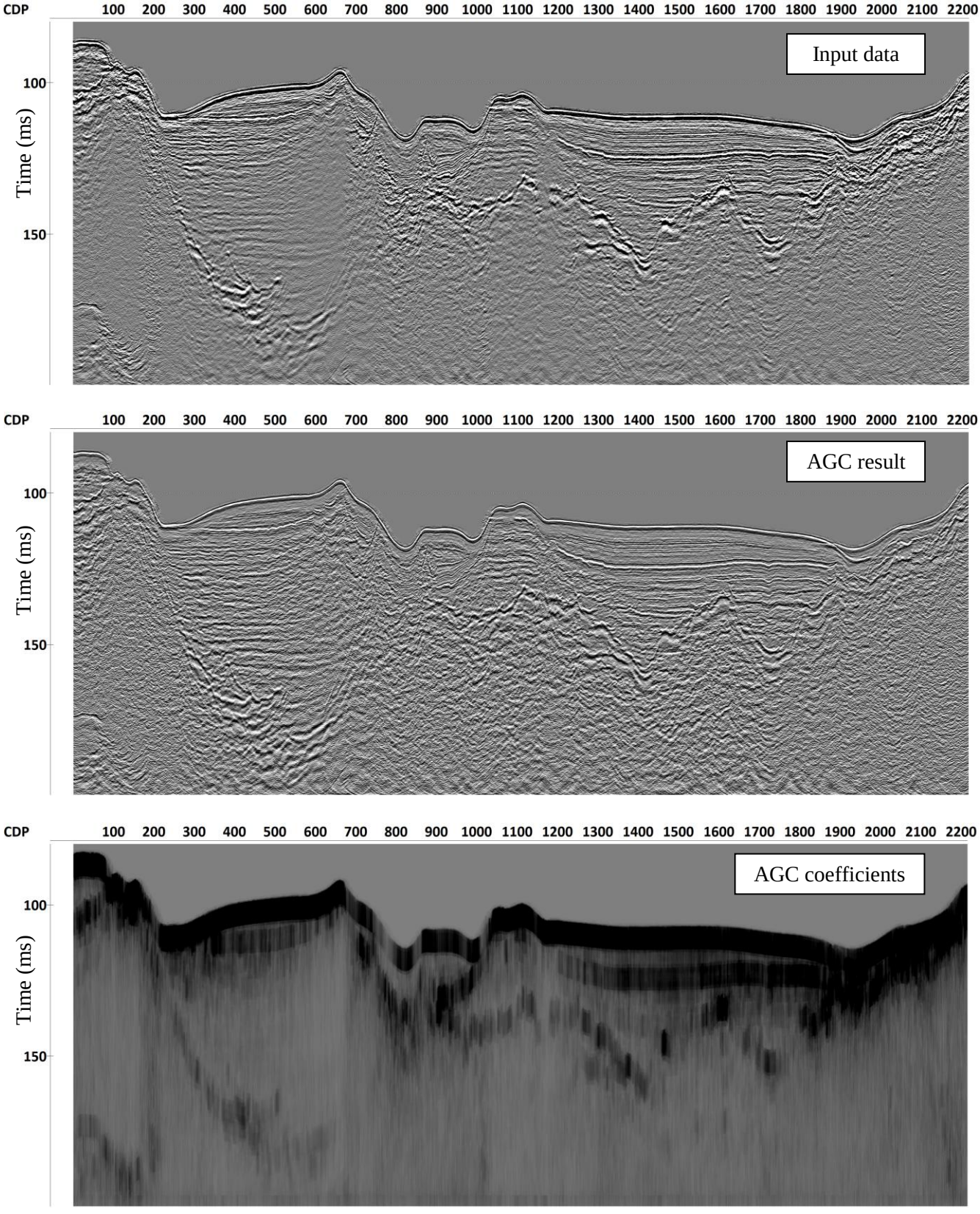
Two parameters are used in this module. ***win_size*** is the AGC window size (in number of

samples) which is read from the AAXSLOP header from the first trace. AAXSLOP value is set in the preceding Trace Header Math module. Note that one can use space-variant parameters by appropriately setting varying headers and extracting their values for each trace into an array. ***eps*** is a small constant number which is used here to avoid division by zero and is directly hardcoded into the Python function.

```
import numpy as np
from scipy.ndimage import uniform_filter1d

def exec(traces, headers, headers_dictionary):
    eps = 1e-7 # small constant to avoid division by zero

    n_headers = headers.shape[1] # get the number of headers

    # get the index of AAXSLOP header and set AGC window size
    # from the AAXSLOP first trace
    aaxslop_index = headers_dictionary.get("AAXSLOP", n_headers)
    win_size = int(headers[0, aaxslop_index])

    #square the traces and get an array of squared AGC coefs by running a uniform
    # filter along time, 'constant mode' conducts zero padding on trace edges
    agc_coefs = uniform_filter1d(traces**2, size=win_size, mode='constant')

    #extract the square root from the coefs to obtain the RMS in the sliding window
    agc_coefs = np.sqrt(agc_coefs)

    #perform a smooth division of input data by the AGC coefficients
    traces = traces / (agc_coefs + eps)

    return (traces, headers)
```

Results

The module performs AGC as expected.



Semblance computation

The semblance attribute can be computed with two 2D filtering procedures and serves as an example of 2D operations in Python Proxy.

Code

This module reads its horizontal and vertical window size for semblance computation from AAXSLOP and AAXFILT headers, similarly to the previous example. We then apply the ***uniform_filter*** procedure from ***scipy*** on the 2D ***traces*** array to compute 'sum of squares' and 'square of sums' arrays, one of which can be divided by the other to obtain the semblance values (e.g., Kington, 2015):

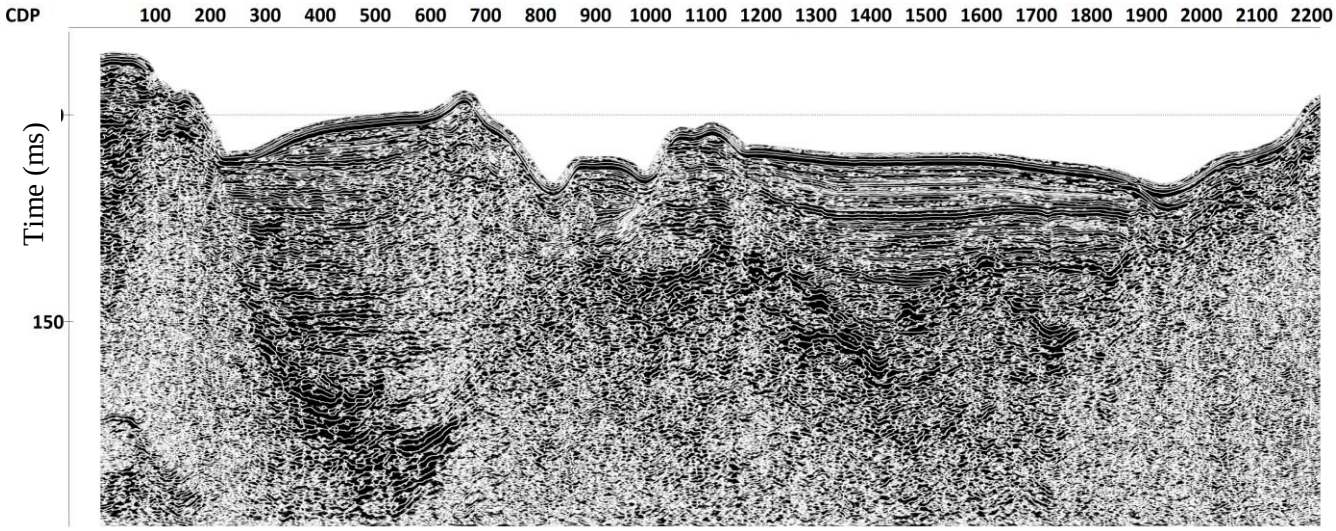
$$S = (\sum_{i=1}^{N}\sum_{t=1}^{M} d_{it})^2 / MN \sum_{i=1}^{N}\sum_{t=1}^{M}(d_{it})^2.$$

Here, S is semblance and d is the input data. Explicit division by window sizes M and N is not needed in our code, as the ***uniform_filter*** computes the mean instead of plain sum in each window.

```
import numpy as np
from scipy.ndimage import uniform_filter

def exec(traces, headers, headers_dictionary) :
    eps = 1e-7 # small constant to avoid division by zero

    n_headers = headers.shape[1] # get the number of headers

    # get the index of AAXSLOP header and set horizontal semblance window size
    # from the AAXSLOP of first trace
    aaxslop_index = headers_dictionary.get("AAXSLOP", n_headers)
    win_size_hor = int(headers[0, aaxslop_index])#

    # get the index of AAXFILT header and set vertical semblance window size
    # from the AAXFILT of first trace
    aaxfilt_index = headers_dictionary.get("AAXFILT", n_headers)
    win_size_vert = int(headers[0, aaxfilt_index])

    # soft division of square of sum by sum of squares to obtain semblance
    sum_of_squares = uniform_filter(traces**2,
                        size=(win_size_hor,win_size_vert), mode='constant')
    square_of_sum = uniform_filter(traces,
                        size=(win_size_hor,win_size_vert), mode='constant')**2
    traces = square_of_sum/(sum_of_squares + eps)

    return (traces, headers)
```

Results

Here is the semblance computation result.



Deghosting

In the tutorial project, the data input for this section of the tutorial occurs in the **110 deghost data input** flow. The input dataset is a common-offset section of a high-resolution marine survey.

We implement a simplistic frequency-domain deghosting algorithm here. This serves as an example for the implementation of Fourier-domain procedures in Python Proxy.

Code

Here, we implement the following deghosting filter $F(\omega)$ (a rewritten equation 4 from the paper by Zhang et al. (2018)):

$$F(\omega) = \frac{(1-re^{-j\omega\tau})^*}{(1-re^{-j\omega\tau})^*(1-re^{-j\omega\tau})+\varepsilon}.$$

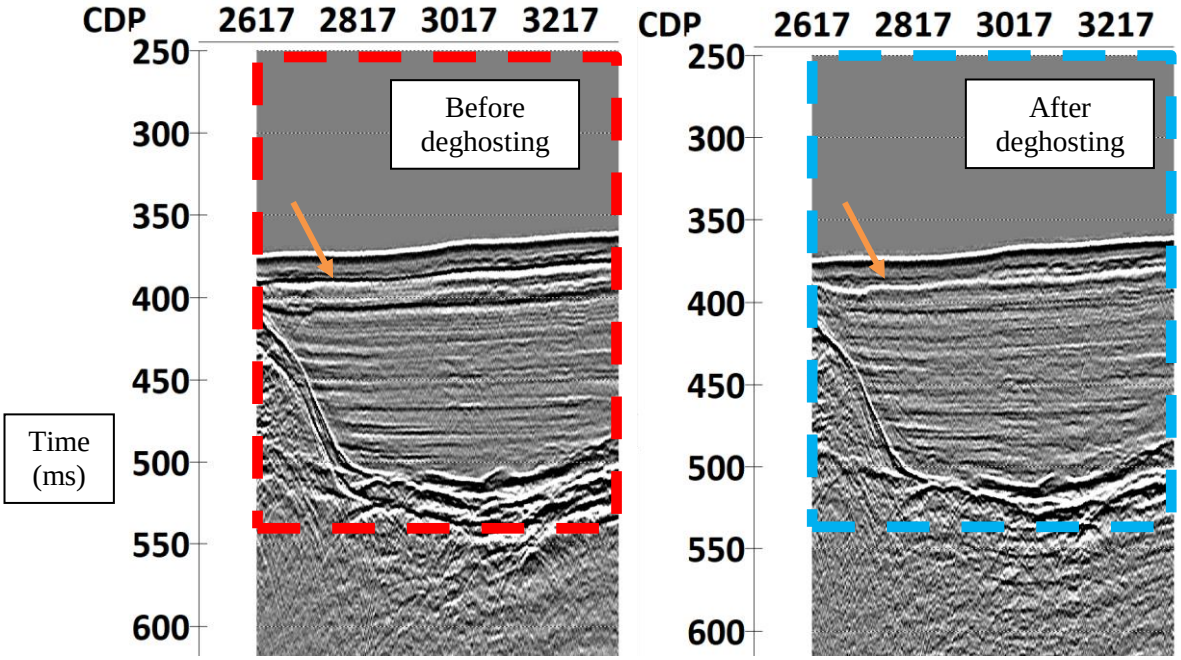
Here, j is the imaginary unit, ω is the angular frequency, r is the absolute value of the sea surface reflection coefficient (we assume pressure/hydrophone data being taken as an input, so the signed value of the reflection coefficient is close to -1), τ is the ghost delay measured in seconds (for vertical propagation, can be approximated by $2h/V$, where h is the sensor depth and V is acoustic wave velocity in water), * stands for complex conjugate, and ε is the regularization coefficient.

```
import numpy as np

def exec(traces, headers, headers_dictionary) :
    eps = 6e-2 # regularization parameter

    nt = traces.shape[1] # get the number of time samples in the data
    n_headers = headers.shape[1] # get the number of headers

    # get the index of PICK1 header and set the reflection coefficient
    # from the PICK1 of first trace
    pick1_index = headers_dictionary.get("PICK1", n_headers)
    ref_coef = headers[0, pick1_index]

    # get the index of PICK2 header and set the ghost delay
    # from the PICK2 of first trace
    pick2_index = headers_dictionary.get("PICK2", n_headers)
    ghost_delay = headers[0, pick2_index]

    dt_index = headers_dictionary.get("dt", n_headers) # get the index of DT header
    dt = headers[0, dt_index]/1e3 # get sample rate in seconds for fft

    traces_fft = np.fft.rfft(traces,axis=1) # run FFT of traces array along time axis
    freqs = np.fft.rfftfreq(nt,d=dt) # get Fourier frequencies for this array sampling
    w = 2*np.pi*freqs # compute angular frequencies as 2*pi*f

    # compute the Fourier ghost operator, the main building block of our filter
    oper_denom = (1 - ref_coef * np.exp(-1j * w * ghost_delay))

    # multiply the array of traces by the deghosting filter
    traces_fft = traces_fft*np.conj(oper_denom)/(oper_denom*np.conj(oper_denom)+eps)

    # run inverse fft specifying the number of samples in the output
    # and convert result back to float32
    traces = np.fft.irfft(traces_fft, n=nt, axis=1).astype('float32')

    return (traces, headers)
```

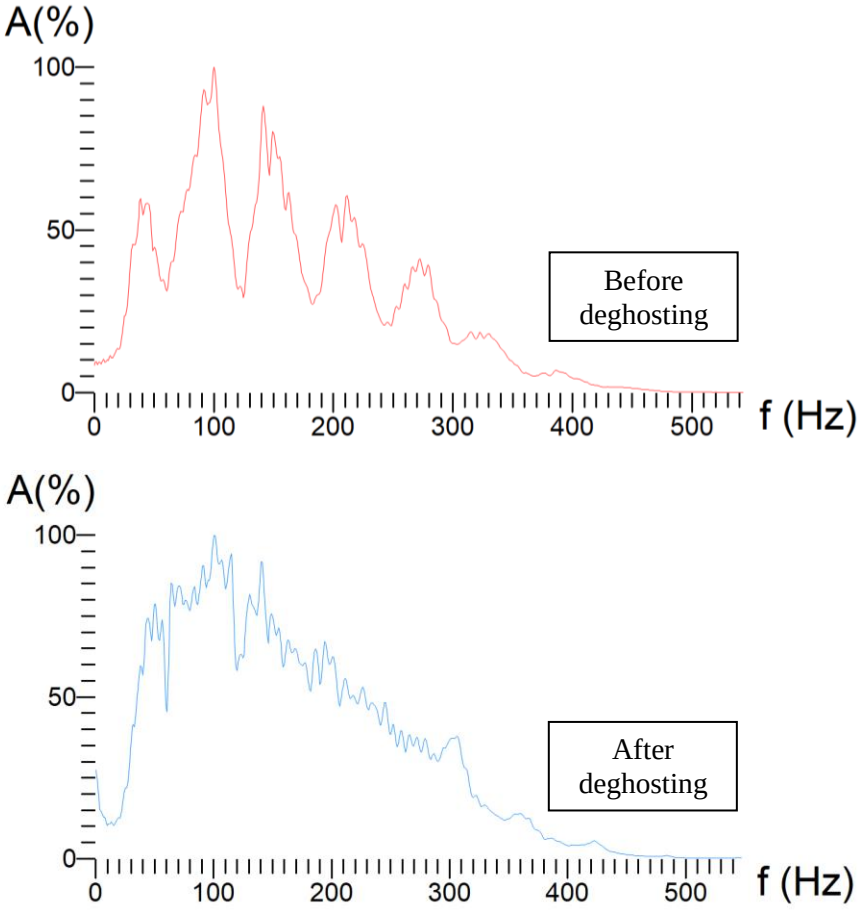
To compute and apply the above filter, we use the standard fast Fourier transform (FFT) functionality of ***numpy***. For FFT, we use the ***rfft*** function from ***numpy***. It is an implementation of FFT for a real-valued array. Using the forward and reverse Fourier transforms of a real-valued array (***rfft*** and ***irfft***) allows us not to worry about the required symmetry of Fourier transforms, as the real-valued-array FFT functions take this into account internally. This means that we can simply implement the algorithm for positive frequencies, and the negative-frequency part will be automatically inferred from the symmetry. Another useful feature of ***numpy*** FFT is the ***rfftfreq*** function, which computes an array of positive Fourier frequencies for the given information of the array sampling. Also, note the conversion back to ***float32*** is used as a safety measure here, as some ***numpy*** functions tend to internally convert the data to ***float64***, and **RadExPro** needs ***float32*** arrays.

Results

Here is the deghosting result. It is not perfect, but remarkable for less than 20 lines of code. Note the visually noticeable attenuation of the seabottom reflection receiver ghost (arrows).



The spectrum of the original data computed in the window highlighted by the dashed line has clear notches, which are filled in after the deghosting.



Geophone-to-DAS conversion

This section presents a method for converting seismic data acquired with geophones to an approximation of distributed acoustic sensor (DAS) data, which may be used for quality control or DAS survey design.

In the tutorial project, the data input for this section of the tutorial occurs in the **210 DAS conversion data input** flow. The input dataset is a VSP survey acquired with both DAS and geophones by Zulic et al. (2022) and was published as an open-source dataset at Research Data Australia (https://doi.org/10.25917/7h0e-d392). This dataset was provided under a CC BY 4.0 license. More details can be found at https://creativecommons.org/licenses/by/4.0/.

For the purposes of this tutorial, we extracted the vertical component of the geophone data for the source number 24 and depth interval of 90-870 m. We kept only one trace for each of the depth levels which were acquired several times. The same depth interval and shot point were taken for DAS data, only FFID 13876 was used. After this, both DAS and geophone datasets were correlated with the pilot sweep (which is also provided in the dataset) and output as SEG-Y files. These SEG-Y files are used as an input to this tutorial project.

Code

Here, we implement the conversion method used by Zulic et al. (2022) and, originally, by Bakku (2015):

$$\dot{\varepsilon}_{zz}^{DAS} = \frac{v_z\left(z+\frac{L}{2}\right)-v_z\left(z-\frac{L}{2}\right)}{L},$$

where $\dot{\varepsilon}_{zz}^{DAS}$ is the strain rate along the cable as measured by the DAS system, v_z is the vertical geophone recording, z is the depth, and L is the gauge length – a depth interval over which the strain rate is averaged in the DAS system.

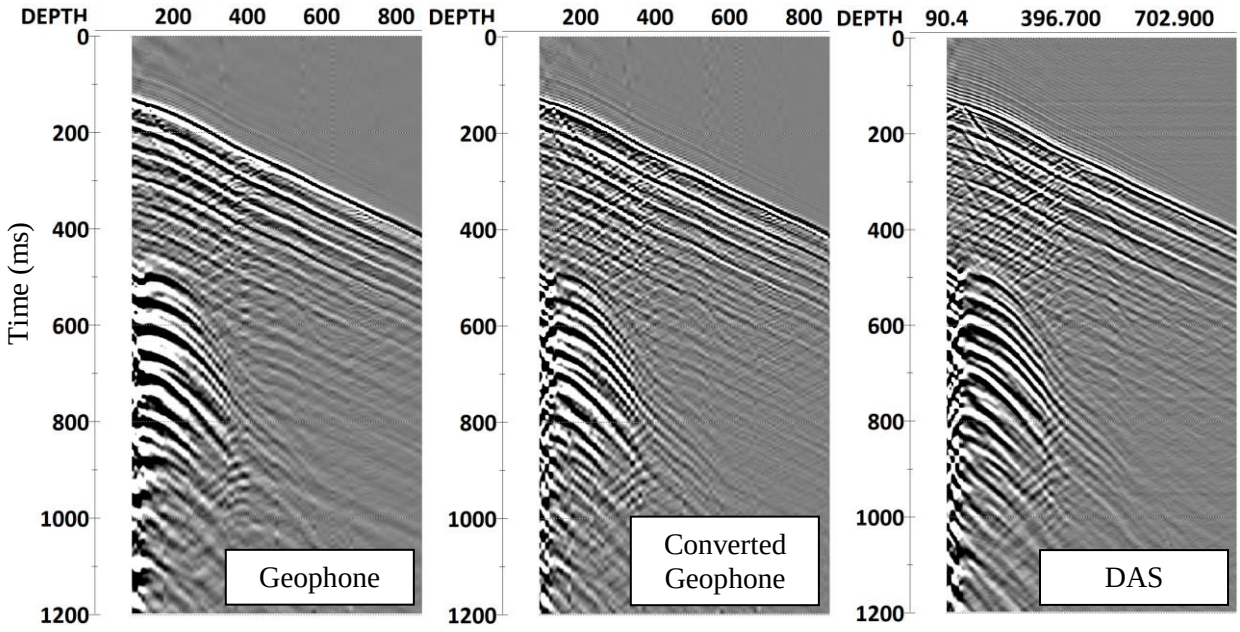
The annotated code is given below. We compute the numerator in the equation above via introducing a finite-difference operator of a given length (set by ***oper_len*** and expressed in the number of traces) and convolving it with the data by ***scipy***'s ***convolve1d***.

We do not input any parameters from headers, we only manually set ***oper_len*** in the Python Proxy window. We also use the ***oper_len*** with the DEPTH header to compute the value of the modeled gauge length L. This value of L is printed into the log of the **220 Geo-DAS conversion** flow, e.g., for the 2-point operator the modeled gauge length is equal to the receiver spacing, which is 10 m in this dataset. Note that in this simplified implementation we do not check whether ***oper_len*** is even or odd, so the operator is either central-difference or staggered and the operator origin shifts uncontrollably, but this is enough for the purposes of this demonstration. Also note that after differentiation we reverse the polarity manually by Trace Math to fit the DAS data polarity.

```
import numpy as np
from scipy.ndimage import convolve1d

def exec(traces, headers, headers_dictionary):

    oper_len = 2 # the differentiation operator length in number of traces

    # get the index of DEPTH header and compute the
    # median trace interval just to compute the gauge length
    n_headers = headers.shape[1]
    depth_index = headers_dictionary.get("DEPTH", n_headers)
    depth = headers[:, depth_index]
    dz = np.diff(depth)
    dz = np.median(dz)

    # compute the modeled gauge length and print it to Radex log
    L = (oper_len - 1) * dz
    print('Modeled gauge length is', L)

    # create a finite-difference operator of the given size
    diff_oper = np.zeros((oper_len)) # initialize by zeros
    diff_oper[0] = 1 # set first element to 1
    diff_oper[-1] = -1 # set last element (note -1 in index) to -1

    # convolve the operator with the data with the required normalization
    # axis=0 mean convolution along first axis, i.e., depth
    traces = convolve1d(traces, diff_oper, mode='constant',axis=0)/L

    return (traces.astype('float32'), headers)
```

Results

Here is the comparison of geophone, converted geophone (L = 10 m) and DAS datasets.



References

Bakku, S. K. (2015). Fracture characterization from seismic measurements in a borehole (Doctoral dissertation, Massachusetts Institute of Technology).

Kington, J. (2015). Semblance, coherence, and other discontinuity attributes. The Leading Edge, 34(12), 1510-1512.

Zhang, Z., Masoomzadeh, H., & Wang, B. (2018). Evolution of deghosting process for single-sensor streamer data from 2D to 3D. Geophysical Prospecting, 66(5), 975-986.

Zulic, S., Sidenko, E., Yurikov, A., Tertyshnikov, K., Bona, A., & Pevzner, R. (2022). Comparison of Amplitude Measurements on Borehole Geophone and DAS Data. Sensors, 22(23), 9510.