

Documentation/Manual

Interactive processing of MBES bathymetry and backscatter data using Jupyter Notebook and Python

Field of Hydrography

HafenCity University Hamburg

Dated: February 7, 2021

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 1 |
| 1.1 | What is included (in the ZIP?) | 1 |
| 1.2 | What are Spyder and Jupyter Notebook used for? | 3 |
| 2 | Installation of Python using Anaconda | 3 |
| 2.1 | Creating the working environment | 3 |
| 2.2 | Spyder | 4 |
| 2.3 | Jupyter Notebook | 5 |
| 2.3.1 | Configuration of nbextensions | 5 |
| 3 | Kongsberg ALL preprocessing module for bathymetry and backscatter | 7 |
| 3.1 | Applying the preprocessing module to a directory | 8 |
| 4 | Jupyter Notebook for processing (filters and corrections) based on Entwine, PDAL and Potree | 9 |
| 4.1 | EPT build and Potree visualization | 10 |
| 4.2 | PDAL bathymetry workflow | 12 |
| 4.3 | PDAL backscatter workflow | 13 |
| 4.4 | Data export | 13 |
| 5 | “Developers guide“ ☺ | 14 |
| 5.1 | Preprocessing module | 14 |
| 5.2 | Processing notebook | 16 |
| 5.3 | Specific improvement possibilities | 18 |
| 5.3.1 | Tide correction | 18 |

1 Introduction

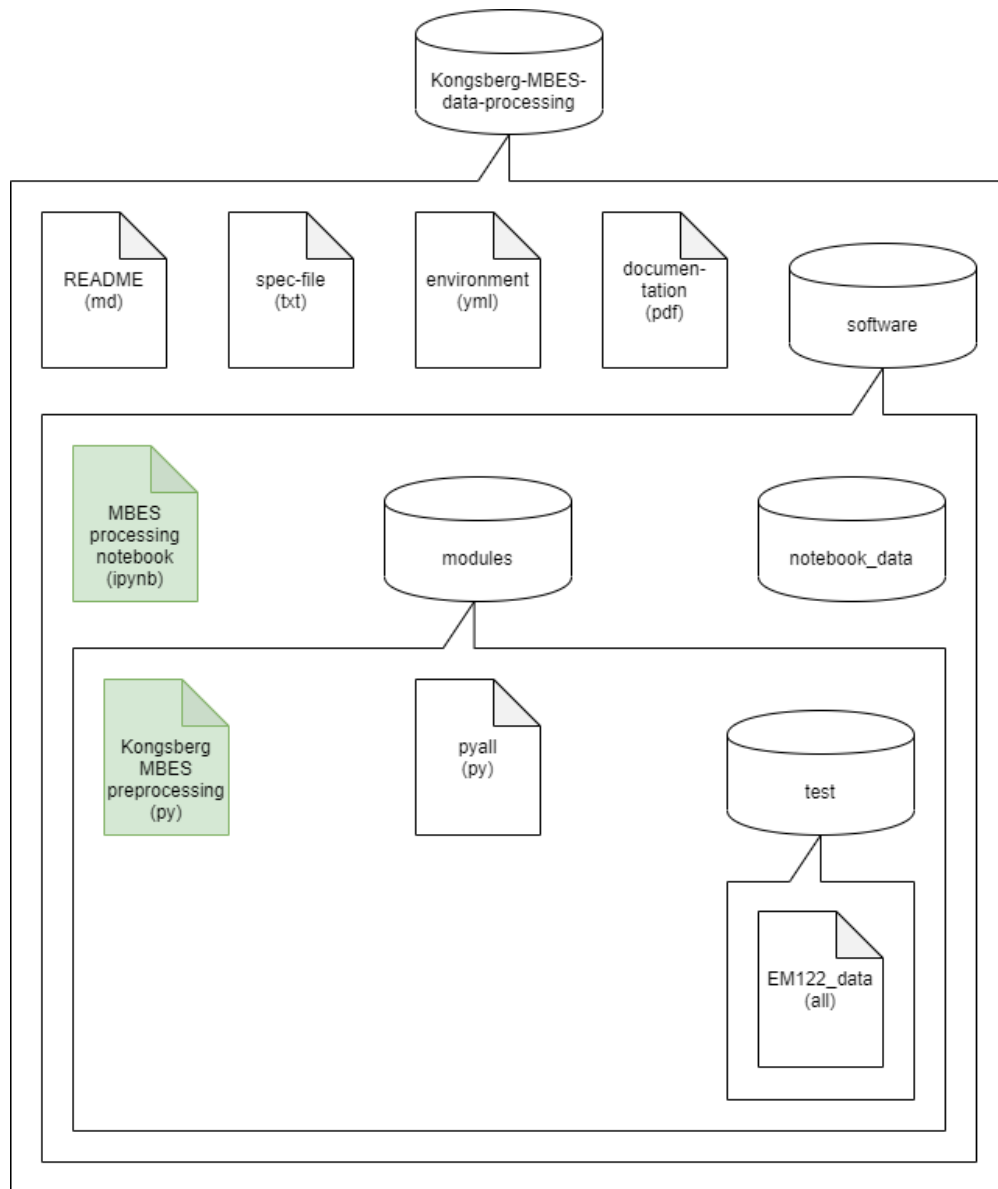
This little program was written as part of a thesis to assess the interactive processing possibility of MBES bathymetry and backscatter data using Jupyter Notebook and Python. It mainly consists of two components:

1. A Python module which includes the functions required for the bathymetry and backscatter preprocessing (from raw Kongsberg ALL files to attributed point cloud ASCII files) and

2. a Jupyter Notebook which acts as user interface and constructor/initiator for the data processing (bathymetry filters and backscatter corrections).
3. Both components are connected by an ASCII interface.

1.1 What is included (in the ZIP?)

Currently, the components are delivered in a ZIP file called Kongsberg-MBES-data-processing. In the drawing below, the container symbol represent a folder and the paper symbol a file:



⇒ The **Kongsberg-MBES-data-processing.zip** file includes:

README.md file which includes some generic information about the program.

spec-file.txt is an explicit specification file which includes information to build an identical conda

environment (more information in [2.1](#)).

environment.yml is an alternative way to the spec-file which allows to recreate a conda environment cross platform.

documentation.pdf This document ☺.

⇒ The **software** folder contains the actual code and ancillary data:

MBES processing notebook.ipynb is the processing notebook for the above mentioned filters and corrections.

⇒ The **notebook_data** folder contains the graphics etc. which are used in the processing notebook.

⇒ The **modules** folder includes the Python modules for the bathymetry and backscatter pre-processing:

Kongsberg_MBES_preprocessing.py is the actual preprocessing module.

pyall.py is an external module which is used to decode the Kongsberg ALL files within the preprocessing module.

⇒ The **test** folder includes an ALL file (**EM122.data.all**) for testing.

1.2 What are Spyder and Jupyter Notebook used for?

Spyder as well as Jupyter Notebook might be summarized under the term development environment. While Spyder is used to run the Python modules of the preprocessing, the Jupyter Notebook provides the graphical user interface for the interactive processing of the bathymetry and backscatter data. The two components are linked by an ASCII interface which allows to either substitute one or the other component with another program/software. The installation is further explained in section [2](#).

2 Installation of Python using Anaconda

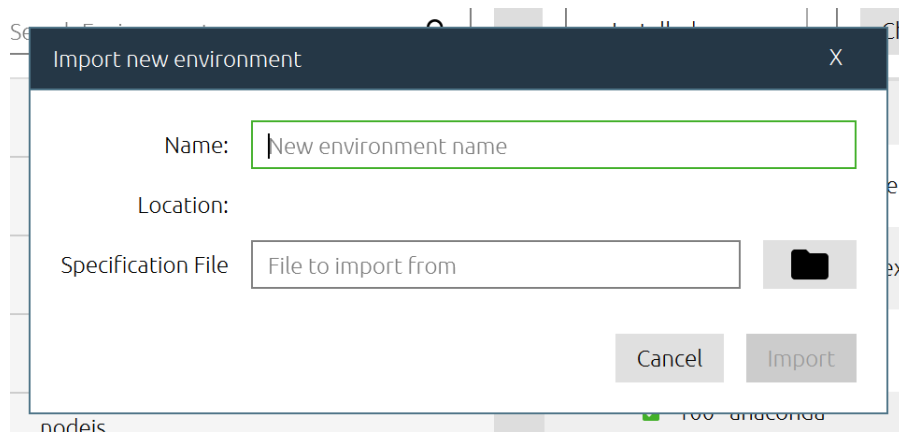
Anaconda (Individual Edition) is a Python distribution for scientific computing. It can directly launch Spyder and Jupyter Notebook. After installation, the **Anaconda Navigator** can be used to manage packages and environments (the site package dependencies and programming tools). By default the base (root) environment is active. To run the program, a working environment should be created which includes the required dependencies.

2.1 Creating the working environment

There are two options to create the working environment:

1. Either by using the environment.yml file which will recreate the depicted environment (name is already defined).
2. Or by using the spec-file.txt which will in this case only work on Windows 64-bit (name is passed during creation).

To build the environment using Anaconda Navigator, navigate to the 'Environments'-tab on the left hand panel. Subsequently, the 'Import'-button can be used to open following window:



The 'Name' of the new environment can freely be chosen (when using the spec-file.txt and will be ignored when using the environment.yml). Both, the provided spec-file.txt as well the environment.yml¹ file can be passed as 'Specification file' for the build. Once the new environment was created, it will typically be activated immediately. The active environment is shown on the 'Home'-tab behind 'Applications on'. If not, the drop-down menu can be used to activate it:

Applications on Kongsberg-MBES-data-processing Channels

This is also the way how to activate the environment when reopening the Anaconda Navigator.

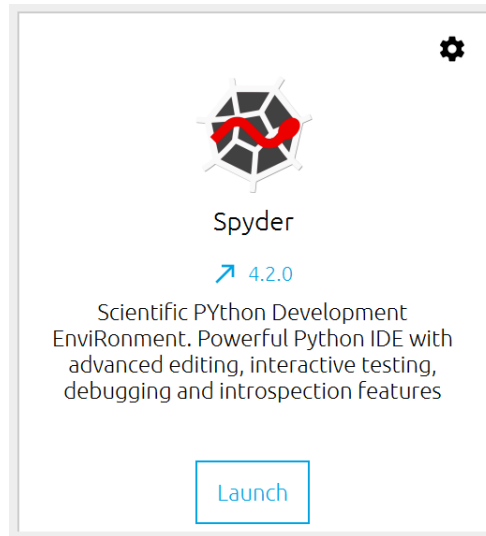
2.2 Spyder

[Spyder](#) is an open source cross-platform IDE² for scientific programming in the Python language. During the implementation of this program, Spyder was used and found to be very handy. However, you are free to use whatever other development environment you are used to and/or prefer.

To launch Spyder, simply scroll in the applications window of the 'Home'-tab of the Anaconda Navigator and press the Spyder 'Launch'-button:

¹Uses the environment name given in the file which is (Kongsberg.MBES.data.processing)

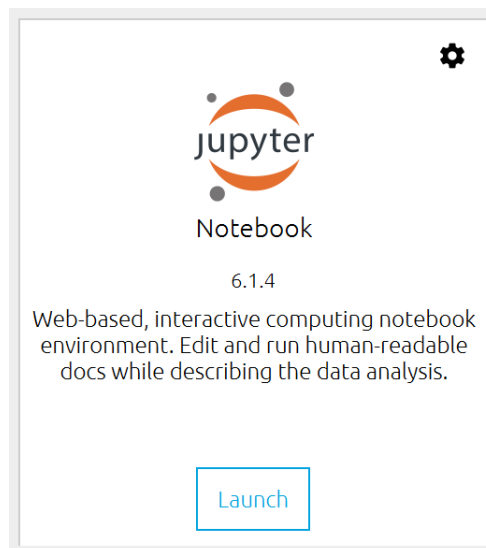
²Integrated Development Environment



How to run the preprocessing module will be explained in section 3.

2.3 Jupyter Notebook

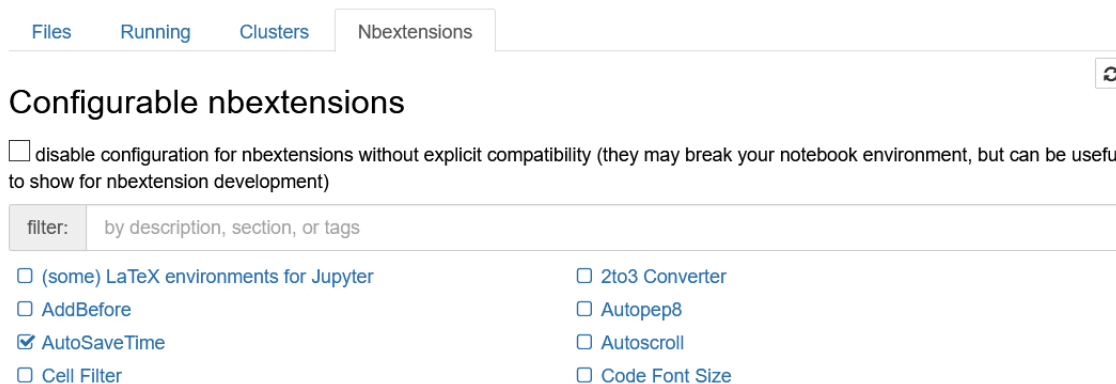
To launch Jupyter Notebook, scroll in the application window until you find Jupyter Notebook and use the provided 'Launch'-button. When firstly using Jupyter Notebook, you will be asked what browser to use. Good choices are Firefox, Chrome and Safari as they support the Javascript used in Jupyter.



How to run the processing notebook will be explained in section 4.

2.3.1 Configuration of nbextensions

The [nbextensions package](#) contains a collection of community-contributed unofficial extensions that add functionality to the Jupyter Notebook. For the processing notebook, some of them are helpful. To configure them, the 'Nbextensions'-tab on the Jupyter Notebook start page can be used:



When firstly configuring the nbextensions, it might happen that the individual extensions are disabled (grayed out):



In this case it helps to check the 'disable configuration [...]'-checkbox below the heading and then uncheck it again. Afterwards, the extensions should be check-able as in the figure above with the blue font.

Generally, all extensions which seem helpful can be used in the processing notebook. Some of them are however not yet bug-free and might cause unwanted behavior. Those should be unchecked again if causing issues. The extensions which are definitely recommended include:

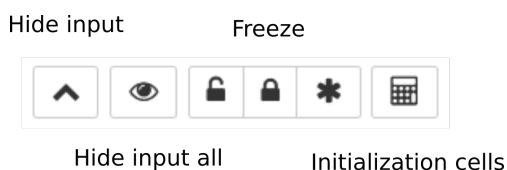
Hide input allows hiding of an individual code cell in a notebook.

Hide input all allows hiding all code cells of a notebook.

Freeze freezes cells (forbids editing and executing) or makes them read-only.

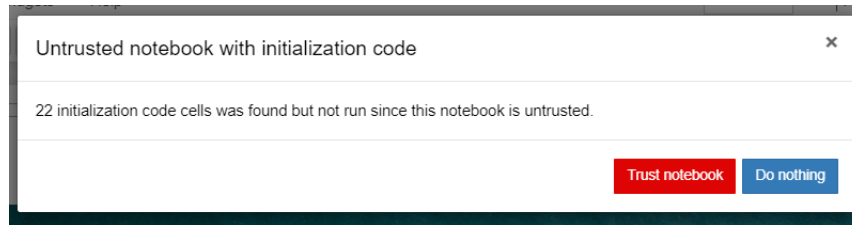
Initialization cells adds a cell toolbar selector to mark cells as 'initialization' cells. Such initialization cells can be run by clicking the provided button in the main toolbar, or configurably, run automatically on notebook load.

A short description of them can be seen when selecting the extension in the 'Nbextensions'-tab. In a Jupyter Notebook, they can be found as icons in the toolbar on the top:



All of the code cells in the processing notebook are already set to read-only and marked as initialization cell. To hide the code, simply press the 'Hide input all'-button with the eye symbol. Any notebooks, which were open while the configuration, have to be closed and opened again to show the extension changes.

The initialization cells might trigger a warning as shown below:



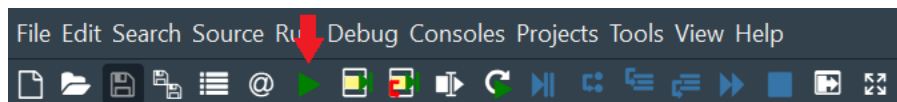
If you know the source of the notebook you are running, you can trust the notebook.

3 Kongsberg ALL preprocessing module for bathymetry and backscatter

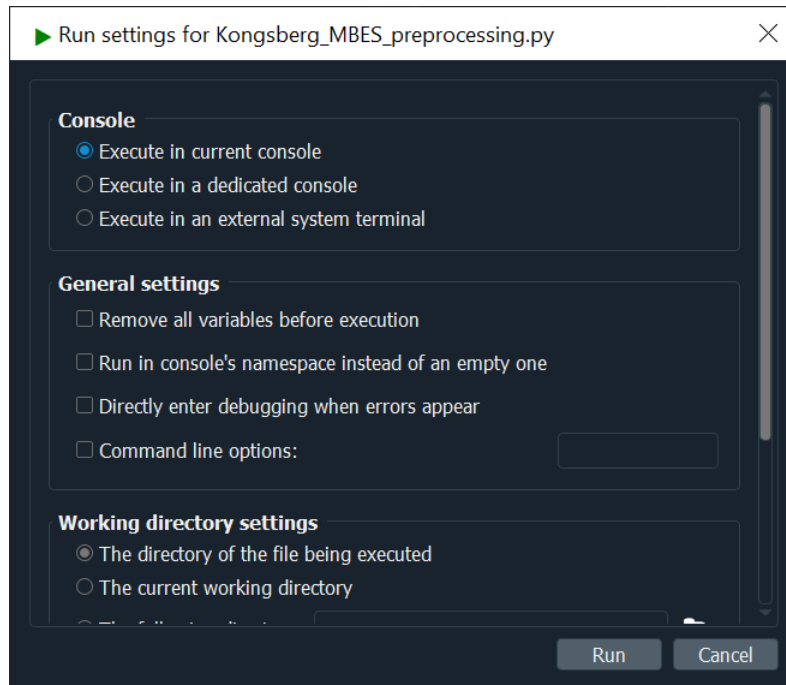
The preprocessing module (Kongsberg-MBES-data-processing/software/modules/Kongsberg_MBES_preprocessing.py) is meant to cover the required stages from raw Kongsberg ALL files to (attributed) point clouds. Kongsberg has different formats for their various MBES systems. Some of them are tested or assumed to be readable by the implemented program while the ones not mentioned below are expected to be unreadable:

- EM 122 (tested)
- EM 302
- EM 710 (tested)
- EM 2040 and 2040C

To run the preprocessing, Spyder needs to be launched (as explained in section 2.2). Alternatively any other Python development environment can be used. When using Spyder, the module can be opened by File → Open... and then navigating to the Kongsberg_MBES_preprocessing.py file. The most basic way to run the code is by using the play-button which is indicated by the red arrow:



On the first run you might be asked to define the 'Run settings'. The default settings are sufficient, so you can simply press the 'Run'-button:



The `main()` function is implying how the functions have to be combined in order to preprocess raw Kongsberg ALL files. The statement at the end of the module (`if __name__ == "__main__"`) and following expressions will only be executed when the module is directly run as script instead of being called by e.g. another module. Meaning that this code will be ignored when imported to another module or script. Therefore, this section can be used to call the desired function (as can already be seen). In its original state, the `main()` function is called to produce the bathymetry and backscatter (`backscatter=True` by default and hence will be computed if not explicitly turned to `backscatter=False` during the function call) point clouds of the test ALL file. If the code is run without modification, it will produce two ASCII files, one for the bathymetry and one for the backscatter. Thereby the ALL file path can simply be changed to whatever path. The same applies to the paths of the ASCII files which are written.

3.1 Applying the preprocessing module to a directory

When planning to apply this function to several files such as the ALL files in a directory, the code should be modified to loop through the files:

```
if __name__ == "__main__":

    from pathlib import Path

    # List all files in directory which end with .all using pathlib
    basepath = (Path(__file__).parent).joinpath('test')
    files_in_basepath = (entry for entry in basepath.iterdir() if entry.suffix == '.all')
    # For each file in the list compute the bathymetry and backscatter
    for filename in files_in_basepath:
        bathymetry, backscatter = main(filename)
        bathymetry.to_csv(
            basepath.joinpath('bathy_' + filename.stem + '.csv'),
            sep=',',
            columns=[
                'longitude',
                'latitude',
                'depth',
```



```

        'classification'
    ],
    header=[
        'X',
        'Y',
        'Z',
        'Classification'
    ],
    index=False,
    mode='w'
)

np.savetxt(
    basepath.joinpath('back_' + filename.stem + '.csv'),
    backscatter,
    delimiter=',',
    header='X,Y,Z,Amplitude',
    comments=''
)

```

The `basepath = (Path(__file__).parent).joinpath('test')` takes the path of the current script (module), joins it with the `'test'`-folder and assigns it to the variable `basepath`. However, `basepath` could also be assigned an absolute path such as `basepath = 'C:/whatever/comes/in/here'` to any other directory. The statement `basepath.joinpath('bathy_' or 'back_' + filename.stem + '.csv')` produces an altered name of the ALL file to save the ASCII files such as (`bathy_EM122_data.csv` or `back_EM122_data.csv`). Thereby `filename.stem` is the name of the ALL file without the extension (`EM122_data`). Path handling can be quite confusing but very handy to automate the preprocessing. A [tutorial](#) might help to manage the path handling.

As can be seen, the two function calls which write out the bathymetry and backscatter data differ. This is because the backscatter georeferencing uses a different approach. That is why, the bathymetry data comes in a pandas dataframe and the backscatter data in a numpy array. To understand the individual calls and alter them, it can be helpful to check the function API reference ([pandas.DataFrame.to_csv](#) and [numpy.savetxt](#)). The calls above are already configured to work as import for the processing notebook.

4 Jupyter Notebook for processing (filters and corrections) based on Entwine, PDAL and Potree

The processing notebook (`Kongsberg-MBES-data-processing/software/MBES processing notebook.ipynb`) is meant to cover the required stages from measured point clouds to firstly cleaned (for outliers) bathymetry and (partly) corrected backscatter. Both data types are processed separately. This decision is mainly caused by the fact that the backscatter point cloud is interpolated between the measured bathymetry soundings and therefore has a huge data amount. Both, the bathymetry as well as the backscatter processing start with the respective ASCII file(s). These file(s) are converted into the [Entwine EPT format](#) that is used as basis for both, the data visualization in [Potree](#) and data processing in [PDAL](#). Potree is a WebGL based point cloud renderer which can be used to visualize point clouds in a browser. The Potree viewer is embedded in the processing notebook and will appear once the EPTs are available. PDAL³ is a C++ library for fast point cloud processing which is executed by the underlying Python code in the Jupyter Notebook. The processing notebook can therefore be understood as a processing handler and initiator. The whole functionality is supported by ipywidgets, which are supposed to predefine and facilitate the processing. An ipywidget can for example be a button, text field, a slider, etc.

³Point Data Abstraction Library

4.1 EPT build and Potree visualization

The bathymetry as well as the backscatter processing starts with a point cloud in ASCII format as it is exploratory produced by the preprocessing module. To convert them in the EPT format, the ASCII files need a certain formatting:

X,Y,Z(,Classification/Amplitude)

-168.06797219085004,29.735780872667572,-5376.169933795929(,0/-2.380)

...

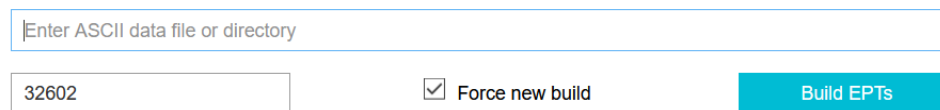
Thereby it is mainly important that the longitude, latitude and depth values are written down in that order and with the 'X,Y,Z' heading in order for Entwine to recognize it. The backscatter has to be stored in the 'Amplitude'-dimension. Other PDAL [dimensions](#) can be appended and will be interpreted as a known dimension if the header implies one of them.

To visualize the data in Potree, the geographic coordinates of the point clouds have to be transformed to projected (UTM) coordinates. To find the suitable UTM EPSG code of the data set, the provided widgets can be used by adding a set of geographic coordinates (extracted e.g. from an ASCII file) and pressing the 'Get UTM EPSG code'-button:



Longitude: -168.067 Latitude: 29.735 Get UTM EPSG code EPSG: 32602

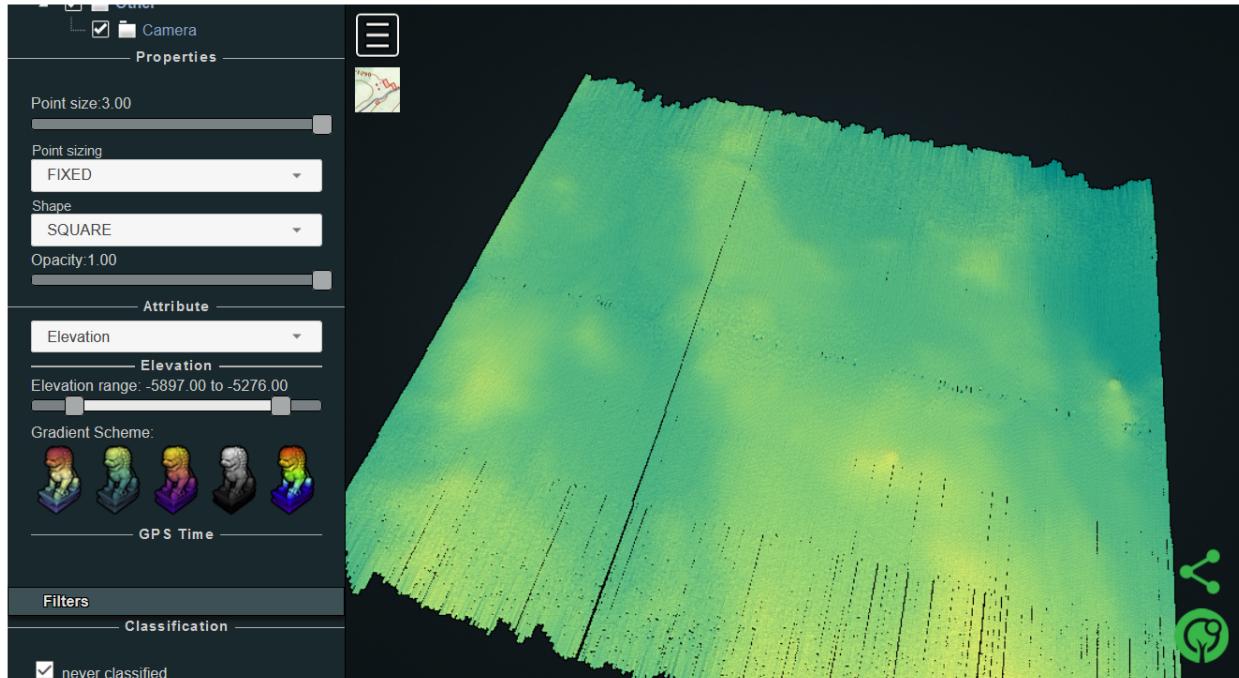
The printed EPSG code can then be copied in the interface provided for the EPT build. It also requires the path to either an individual ASCII file or a directory including the ASCII files to be read:



Enter ASCII data file or directory

32602 ☒ Force new build Build EPTs

The 'Force new build'-checkbox will force a new EPT build rather than appending to a possibly existing one. Thereby the EPT data set will be automatically stored in the given directory or the same directory as the single file and named 'EPT.data'. Once the data set is created, the Potree viewer is opened:



*Hint! Sometimes the output of the code cells is cropped such as in the image below. To fold out the output you can simply press the grayish rectangle on the left side:



On double press the output is hidden:

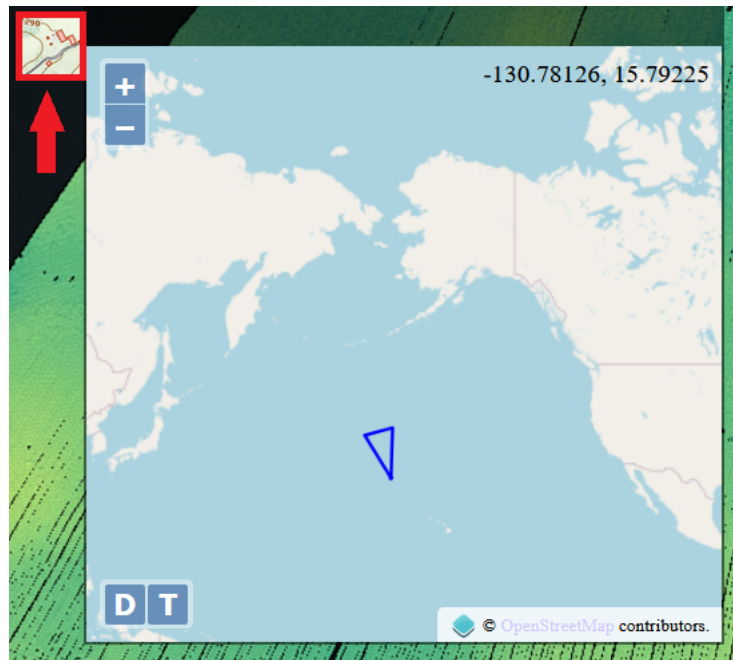


The Potree viewer can be configured in the panel on the left hand side. In the beginning the data will be shown in black (visible in the image above) as there is no RGB information available. To visualize e.g. the depth, simply select the 'Elevation'-attribute in the 'Properties'-tab. There are more options beyond this to adjust the point view, just test whatever you like. If you want to reset the viewer you can just use the 'Update Potree viewer'-button top right of the window:

Update Potree viewer

Very unfortunately, the viewer cannot (or hopefully rather not yet) visualize the Amplitude dimension. It is definitely desired for the future, however, so far exported products of the backscatter processing (section 4.3) have to be reviewed externally. A good choice for this purpose is [QGIS](#), an open source GIS software

with a lot of community support. QGIS can be used to inspect the output grids of the backscatter corrections. An additional feature of the Potree viewer that might come in handy, is the overview map which can be opened with the little icon indicated by the red square and arrow:



4.2 PDAL bathymetry workflow

The bathymetry processing is based on different [PDAL filters](#) which are designated for outlier classification. The general approach is to use ipywidgets to build a PDAL [pipeline](#) of those filters. The pipeline is automatically configured to start with an EPT reader of the above entered data set and end with a writer of an intermediate file that is used to automatically update the EPT. The individual filters have to manually turned on with the provided checkboxes such as the 'Use depth window filter' checkbox below:

Min depth:

Max depth:

☐ Use depth window filter

Then the filter is configured in real time. For example in this case the min and max depth are adjusted to the desired values. Once all filters are configured, the pipeline is constructed on 'Build bathymetry pipeline'-button pressed:

Build bathymetry pipeline

After that the pipeline you have just built is displayed:

```
[  
  {  
    "type": "readers.ept",
```

```

    "filename": "C:/Users/.../Kongsberg-MBES-data-processing/software/modules/test/
                  EPT_data/ept.json"
  },
  {
    "type": "filters.assign",
    "value": "Classification = 18 WHERE Z > -200.0"
  },
  {
    "type": "filters.assign",
    "value": "Classification = 18 WHERE Z < -5000.0"
  },
  {
    "type": "writers.las",
    "filename": "C:/Users/.../Kongsberg-MBES-data-processing/software/modules/test/temp.
                las"
  }
]

```

The pipeline is written in JSON syntax. To keep track of good processing settings, the JSON string can be copied and stored in a text file. If the pipeline is set correctly, you can press the 'Run bathymetry pipeline'-button:

Run bathymetry pipeline

This will apply the filter pipeline to the data and update the EPT data set. To check the result in the Potree viewer, you can use the 'Update Potree viewer' on the upper right side.

Attention!!! Depending on your browser settings, Potree might be updated from the cache. In that case Potree will not reflect any recent changes. Check your browser setting to find out how to change this behavior.

4.3 PDAL backscatter workflow

The PDAL backscatter workflow follows the same structure as the bathymetry workflow described in section 4.2. The pipeline is composed with the provided widgets and have to be manually turned on and off using the checkboxes. Afterwards, the 'Build backscatter pipeline'- and the 'Run backscatter pipeline'-button can again be used to apply the pipeline to the EPT data set:

Build backscatter pipeline

Run backscatter pipeline

Attention!!! The major difference to the backscatter pipeline is that points are discarded based on the backscatter value instead of being flagged. Therefore it is the better approach use some basic settings for the backscatter processing, export the result as grid (section 4.4) and check the it in QGIS or any alternative software choice before increasing the filter settings. At this point, it is a very dirty work around, sorry for that!!!

4.4 Data export

Once the bathymetry or the backscatter data is finished processing, a grid can be exported. When using the export widgets, it is always the currently deposited data set that will be exported:

Filename:

Grid resolu...

Radius:

IDW power: 1

Window size: 1

Dimension:

☒ Reject outliers (bathymetry only)

Export data

The requested name is the name without a file extension under which the grid will be saved. It will automatically be stored in the same directory as the EPT data set. For the bathymetry, either the 'Z' or the 'Classification' dimension can be exported. When exporting Z, classified outliers can be ignored by using the 'Reject outliers'-checkbox. However, that can only be used with the bathymetric data. For the export of a backscatter data set, simply select the 'Amplitude' dimension.

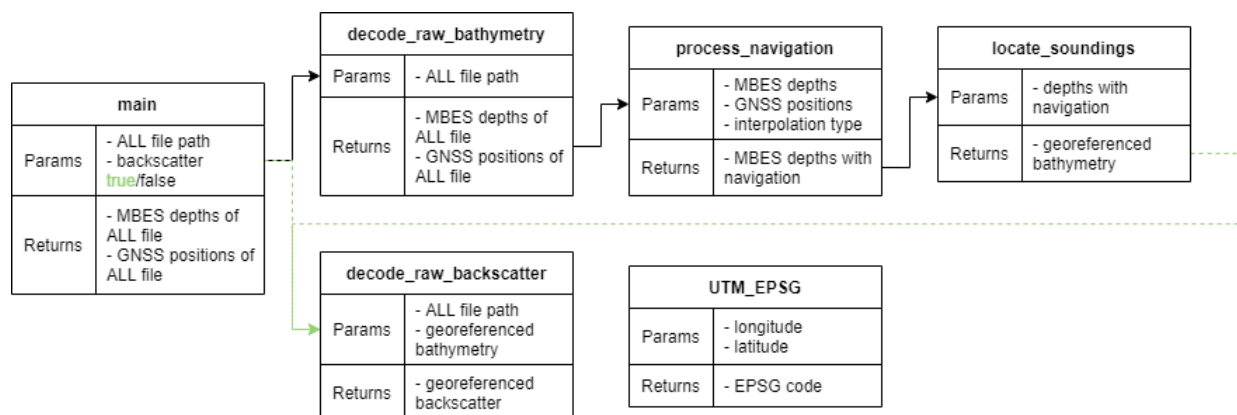
An error is typically caused when a dimension is attempted to be exported which does not exist in the data set. E.g. export of the amplitude dimension of a bathymetric data set.

5 “Developers guide“ 😊

Some of the rather conceptual improvements and an outlook for this little program are illustrated in the thesis itself. To alter and adjust the program as it is, the general structure is subsequently explained and the more specific improvement possibilities are listed.

5.1 Preprocessing module

The preprocessing module includes different Python functions which have to be called in a given order to write out ASCII files with longitude, latitude, depth for the bathymetry and for the backscatter data also the amplitude values:



The bathymetry processing chain is based on a [pandas dataframe](#). The dataframe is constructed in the `decode_raw_bathymetry(filename)` function. For that purpose empty lists for the position and XYZ datagrams are created. While the pyall ALLReader still includes data, each datagram header is checked. If the header corresponds to one of the two datagrams, the corresponding datagram is read and an object with the respective attributes initiated. The relevant attributes are then appended to the corresponding list as a Python [dictionary](#). Once there is not data in the raw file anymore, the lists of dictionaries are converted to pandas dataframes. These are passed and altered through the next processing functions. The function's docstrings will give some information on what the corresponding function is doing and what kind of data input and output it uses:

```

def decode_raw_bathymetry(filename):
    """Decode raw Kongsberg ALL bathymetry and return dataframes.

    Parameters
    -----
    filename : str
        The file location of the Kongsberg ALL file.

    Returns
    -----
    position : pandas DataFrame
        Position datagrams of the ALL file.
    XYZ : pandas DataFrame
        XYZ datagrams of the ALL file.
    """

```

After all the required stages/functions have been applied, the bathymetry dataframe can simply be read out by using the `dataframe.to_csv()` function.

For the backscatter data it is a little more complicated. It is based on a triple index (swath index - beam index - sample index), which couldn't really be handled appropriately in pandas, therefore, a [numpy array](#) is used instead. The backscatter processing starts with the decoding of the corresponding seabed image datagram. The individual datagram (swath) will only be decoded, if the corresponding swath exists in the bathymetry dataframe:

```

try:
    bathymetry_swath = bathymetry.loc[datagram.Counter]
    # Ignore the swath when no bathymetry is available
except KeyError:
    continue

```

Sometimes this is not the case in the beginning or ending of a raw data file. However, without the bathymetry swath, the backscatter samples cannot be georeferenced. If it is available, the datagram will be passed:

```

# Else georeference samples of seabed image datagram
backscatter = np.vstack(
    (backscatter,
     reduce(
         lambda acc, beam: acc.update(beam),
         datagram.beams,
         GeoReferencingAccumulator(bathymetry_swath)
     ).georeferencedSamples
    )
)

```

This part can be difficult to track. It could be helpful to understand the individual elements first:

- [numpy vstack](#)
- [functools reduce](#)
- [lambda functions](#)

Python's reduce function is applied to each seabed image datagram in order to accumulate the georeferenced samples beam by beam for that respective swath. The actual georeferencing of the individual beam time-series can be found in the `update(self, nextBeam)` function of the `GeoReferencingAccumulator` class. It forms new samples series between the two samples corresponding to the beam bottom detection samples. If a beam is the first beam of the swath, the swath `self.georeferencedSamples` is initialized and the beam is set to `self.lastBeam`:

```

if self.lastBeam is None:
    # Initialize numpy array
    self.georeferencedSamples = np.array(
        [
            [
                self.bathymetryOfSwath.longitude.loc[0],
                self.bathymetryOfSwath.latitude.loc[0],
                self.bathymetryOfSwath.depth.loc[0],
                nextBeam.samples[nextBeam.centreSampleNumber]*0.1
            ]
        ]
    )
    # Proceed to next beam
    self.lastBeam = nextBeam
    self.last_beam_index = 0

```

For all following beams behind the `else` clause, the samples between `self.lastBeam` and `self.nextBeam` are stacked. Afterwards, the corresponding soundings are searched in the bathymetry swath and the coordinates are interpolated for each sample. The samples with coordinates are then collected in respective numpy arrays. Those are stacked together for the entire raw data file.

5.2 Processing notebook

In order to adjust and modify the processing notebook, it can be helpful to understand the built-in modules:

- [PDAL filters](#) and [pipeline construction](#)
- [Entwine and Potree](#)

Beyond that, an understanding of the [ipywidgets](#) can be helpful.

The main idea of the processing notebook is to use Python in order to glue together PDAL, Entwine and Potree. To inspect and change code cells, you can simply unhide one or all code cells and lift any restrictions on the code cell(s) from the 'Freeze'-extension. The notebook will be in idle mode until it is given either the

ASCII data for conversion to EPT or directly an EPT data set. This will cause the global 'directory'-variable to be assigned the respective path. All following function calls will be using this in order to access the EPT data set. A change of the variable either by user input or by the function called on 'Update Potree viewer'-button clicked will cause a time limited serving of the NodeJs http server which is used to show the EPT data in Potree and an update of the iFrame used to embed the viewer.

For the Entwine build as well some of the PDAL calls a Python `subprocess` is used. A Python subprocess can among others be used to execute commands from within Python which are usually invoked from command line. As for example Entwine does not provide an interface to Python, this is a convenient way to still be able to build the EPT data set from within the notebook. A helper function is available for this purpose that only requires a list of the command arguments, such as `['entwine', 'build', '-c', Path.joinpath(directory, "ASCII_to_EPT_config.json")].as_posix()`:

```
def run_subprocess(args):
    """ Runs a subprocess defined by the program args list and returns a CompletedProcess
        instance. """
    return subprocess.run(args, capture_output=True, text=True, check=True)
```

PDAL as well as Entwine work with `json` strings for most of the configuration tasks. With the `json.dump()` function they can be saved as temporary file until the PDAL or Entwine command is executed. Afterwards they can be deleted with the `os.remove()` function.

Each of the PDAL filters that are used throughout the notebook are equipped with ipywidgets for parametrization. The widgets are linked to a `dictionary` that holds the parameters along with the according values. An example is shown below (check comments for explanations):

```
# Widget for cell parameter
ELM_cell_floattext = widgets.BoundedFloatText(
    value=200.0,
    min=0.1,
    max=500.0,
    step=0.1,
    description='Cell size:')

# Function updating the cell value in the dictionaries
def on_ELM_cell_floattext_change(change):
    ELM_filter['cell'] = change['new']

# Observer that calls the updating function above when the widget value changes
ELM_cell_floattext.observe(on_ELM_cell_floattext_change, 'value')

# Same pattern as for the first parameter
ELM_threshold_floattext = widgets.BoundedFloatText(
    value=30.0,
    min=0.1,
    max=500.0,
    step=0.1,
    description='Threshold:')

def on_ELM_threshold_floattext_change(change):
    ELM_filter['threshold'] = change['new']

ELM_threshold_floattext.observe(on_ELM_threshold_floattext_change, 'value')

# Initialization of dictionary for filter parameters
ELM_filter = {
    "type": "filters.elm",
    "where": "(Classification == 0)",
    "cell": ELM_cell_floattext.value,
```

```

    "threshold": ELM_threshold_floattext.value,
    "class": 7
}

# Checkbox widget --> Status True/False is later used as condition in the pipeline build
ELM_filter_checkbox = widgets.Checkbox(
    value=False,
    description='Use extended local minimum filter')

# Display of widgets
display(ELM_cell_floattext, ELM_threshold_floattext, ELM_filter_checkbox)

```

Behind the build button there is a function that checks the value (True or False) of the respective use filter checkbox. If the value is True, the dictionary with the filter settings is included in the configuration JSON file. If not, the dictionary is simply ignored. At the beginning and end of the configuration, the EPT reader and a writer are automatically added. Since PDAL itself does not have an EPT writer, it is written to a temporary file, which is then read by an Entwine build command. For all pipelines configured in this way (bathymetry, backscatter and data export), the procedure is structured in this way. The only difference in the export is that a grid writer is used.

5.3 Specific improvement possibilities

5.3.1 Tide correction

The tide correction should be implemented in the preprocessing module. At the beginning it would be important to consider in which form the tide data should be read in or if different formats should be accepted. Maybe a good start would be to work with the ASCII format that [Pegel Online](#) provides. Once that is decided, a function could be written to read in the data and provide it in a pandas dataframe:

```

def correct_tide(filepath to tide file, bathymetry_df):
    """This is only meant as pseudo code :)."""
    # Read tide data
    tide_df = pd.read_csv(
        filepath to tide file,
        some format handling stuff,
        parse_dates=True,
        index_col='Column with time information'
    )

    # Combine bathymetry and tide dataframe (both indexed by time)
    bathymetry_with_tide = pd.concat([bathymetry_df, tide_df], join='outer')

    # Interpolate tide data (at bathymetry timestamps)
    bathymetry_with_tide['tide'].interpolate() # Maybe the limit option can be used to
                                                control the interpolation

    # Add tide to bathymetry depth
    bathymetry_with_tide['depth_corrected'] = bathymetry_with_tide['depth'] +
                                                bathymetry_with_tide['tide']

    return bathymetry_with_tide

```