Filtering and Displaying Spectrogram Data

Jakub Rybář
ČVUT–FIT
rybarja6@fit.cvut.cz

February 28, 2025

1 Goals

The goal of this project is to create a toolbox that can do the following:

1. Easily view signal data in the form of a spectrogram from the command line.
2. Process the spectrogram image to make it more readable and to find unusual areas of interest (usually strong or irregular signals). Make separate processing steps clear and viewable.
3. Save these found signals in some usable form for later use.

2 Input and Output

2.1 Input

The script expects signal data in the form of an `.hdf5` file. This signal is processed and previewed based on numerous user parameters in the command line.

2.2 Save the data

The script locates strong / irregular signals and is able to save them in the form of a `JSON` file.

The file mainly consists of a `metadata` key and a `segments` key. Individual segments are saved as **rectangles** (with `x`, `y`, `width` and `height`).

2.3 Preview the image

The processed / unprocessed spectrogram overlayed with found segments can be previewed based on user parameter. More information is included in the README.

3 Processing the signal

3.1 Processing steps

The original signal is processed with multiple sequential operations [1]:

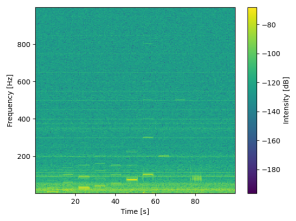

Figure 1: Original spectrogram

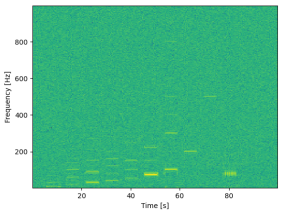

Figure 2: Pre-Processing

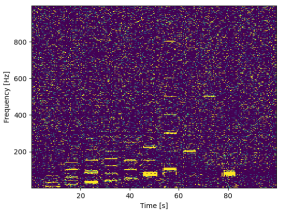

Figure 3: Normalization

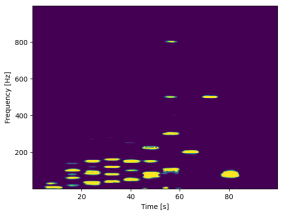

Figure 4: Filtering

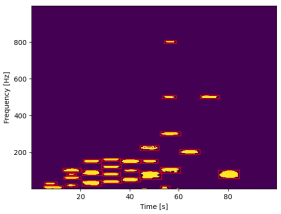

Figure 5: Segmentation

1. **Signal to spectrogram** – Convert the incoming signal into a spectrogram image.
2. **Pre-processing** – Get rid of unwanted regular patterns.
3. **Normalization** – Convert the pixel value range from arbitrary values to the `0-1` range to prepare for image operations.
4. **Image filtering** – Apply local operations like blurring or morphology to get rid of noise.
5. **Segmentation** – Locate and save locations with strong enough signal.

See an example of the processing chain on small test data in figures 1 to 5. All of these images are made only by adjusting the CLI arguments.

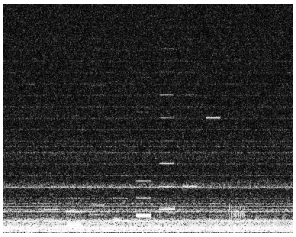

Figure 6: Original spectrogram mapped for preview

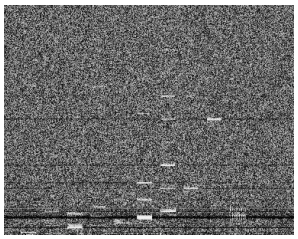

Figure 7: Row average

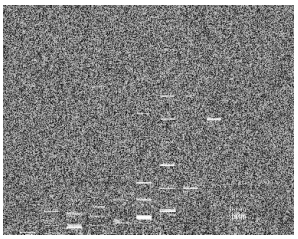

Figure 8: Row median

3.2 Spectrogram

Use a FFT algorithm to convert the incoming signal into a spectrogram.

3.3 Pre-Processing

The testing data often contained some strong and consistent frequencies. Even though these signals are strong, they are consistent and often cover up actually interesting signals, so it turns out to be very effective to "average them out". Another benefit of "averaging out" these values turned out to be detecting other irregularities like suddenly missing signal. With some data, these consistent signals can also be found in the vertical axis.

The following text explains how to average out the rows – the same process applies to the columns. To average out these signals, we want to divide all the row's pixels by the row's **median**. Using the arithmetic average or some other deviations turned out to skew the results if the row is intersected by some other strong, irregular signal (see figures 7 and 8). The horizontal pre-processing is also done in parts (vertical image slices) so the median is less biased for wider images with smooth transitions.

It is important to perform this averaging operation early to so that no inconsistencies are caused by any non-linear operations or by local filters.

3.4 Normalization

This step might reduce overall image information, but it is necessary to be able to easily apply image transformations later. After attempting multiple, approaches I decided to divide the pixel values by a `normalization_value`, that is ideally above the noise ceiling and below the signal floor. My best attempt to find this value was to take the `80th percentile` of all the image pixels. This percentile value is modifiable by the user because it might differ between different spectrograms. We can also subtract this value from the original since it is below the signal floor [1].

$$\texttt{normalized} = \frac{\texttt{pixel value} - \texttt{percentile}}{\texttt{percentile}} \quad (1)$$

Finding the `normalization_value`:

- **Using an arithmetic average** – This wouldn't work as well due to same reasons as mentioned in pre-processing.
- **Using a median** – Median is better because it doesn't get affected by small area strong intensity signals.
- **Using a percentile** (used) – Using a high percentile is better than a median; most pixels in the image are usually just noise. A median value is usually way too low.
- **Using Otsu's threshold** – Sounded promising but the results seemed to perform worse than using a percentile. This might be useful to investigate further.

Other attempted approaches for normalizing:

- **A logarithmic scale** – A logarithmic scale is often used to compress spectrogram images so they are easier to view. However, this non-linear operation compresses the brighter values, making it harder to threshold them. The results also differ heavily between spectrograms with different "exposure levels".
- **Global tone mapping** – Some global tone mapping equations are a good way to compress the dynamic range of any image data to any desired range. However, as they are also non-linear, they struggle from the same issues as the logarithmic scale.
- **Dividing by a normalization value** (equation 1, used) – Compared to the previous methods we lose a lot of data below 0 and above 1. However, this isn't an issue at all because we're only losing data that we already know to be noise / signal. What's important is that the operation is linear and that the whole value range between the noise ceiling and the signal floor is in the `0-1` range - ready for filtering and thresholding with minimal adjustments.

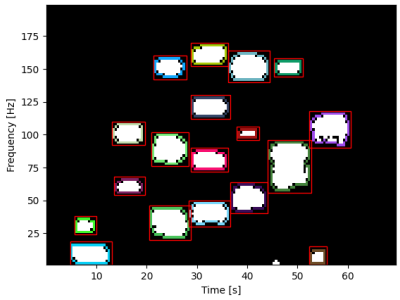

Figure 9: Contours and bounding boxes

3.5 Filtering

In this section we need to remove as much noise as feasible and prepare the image for thresholding and for segmentation.

The first logical operation for removing noise would be to apply a blur. However, applying a simple Gaussian blur on the image would destroy a lot of data. A small way to remedy that is to use some kind of an edge-preserving blur or to use a morphological filter.

For edge-preserving blurring I used a guided filter as from my experiments the results are similar to a bilateral filter but with better performance.

Filtering steps, my approach:

1. **Guided filter** – Eliminate most of the noise. A small radius of 1 to 2 pixels is enough for most images.
2. **Morphology open (erode, then dilate)** – Get rid of small, bright, isolated areas of the signal. Great for getting rid of salt-and-pepper noise.
3. **Guided filter, 2nd pass** – Used to clean up the remaining salt-and-pepper noise.
4. **Morphology close (dilate, then erode)** – To close up holes and to connect near signals without altering their overall size.
5. **Dilate (optional)** – To merge near signals into one larger future segment.

3.6 Segmentation

To prepare the image for segmentation we need to convert the single channel image into a binary image (an image, where the channel's value is only 0 or 1). The image threshold is calculated using the Otsu's method.

Segmentation was implemented using an `OpenCV` function to find the image contours. The output rectangle segments are calculated as bounding boxes from individual contours. Segments with small side lengths (based on user's input) are discarded and the rest of them are saved. For an example of contours and segments being calculated from a binary image see Figure 9

The total area of the segments is also saved but it should probably only ever be used for comparison between the same source spectrogram as with different signal processing settings the spectrogram's pixel aspect ratio can dramatically differ.

4 Viewing the data

As set by one of the goals, any part of the filtering process can be previewed by choosing the type of the `--preview` option in the command line. Most example images in this report were generated only using these user options.

5 About the code

5.1 Pylint

According to the assignment, the code is written in accordance with the PEP8 code style, excluding:

- `C0301` (long lines), `C0103` (short variable names)
- `R0913` (too many arguments) and by extension `R0914` (too many locals) – this choice was made due to a large number of inputable CLI arguments. Most of the arguments in these methods are optional, and grouping them together might reduce code clarity.
- There may be other exceptions caused by third-party libraries.

5.2 Pytest

The repository contains tests for `.hdf5` loading, spectrogram filtering and segmenting, as these were important to achieve the project's goals.

6 Underwater processing example

A more interesting example of larger and noisier underwater data can be seen on Figures 10 to 14. To deal with the noise, the percentile value was lowered, and the filtering strength was increased.

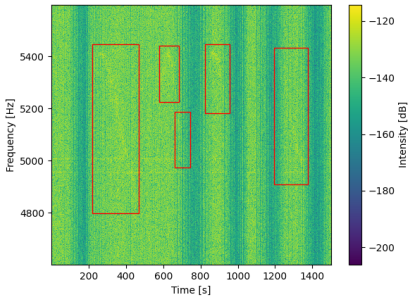

Figure 10: Original with the result

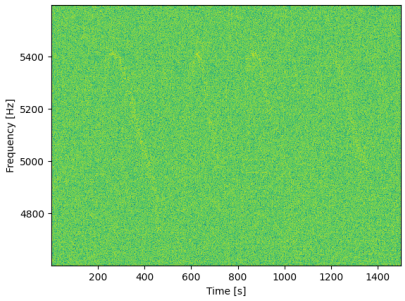

Figure 11: Pre-Processing

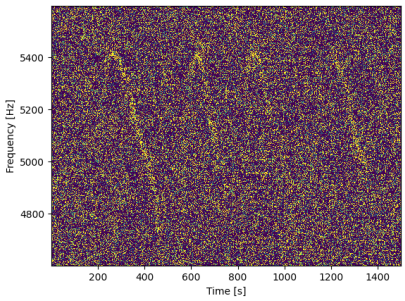

Figure 12: Normalization

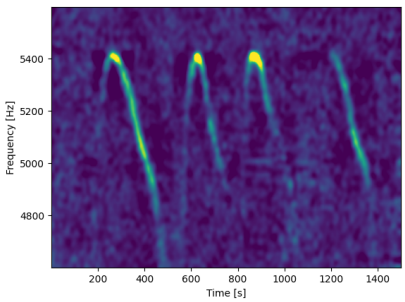

Figure 13: Filtering

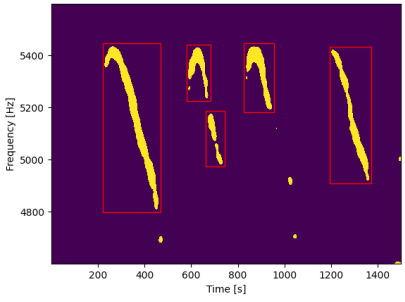

Figure 14: Segmentation

7 Conclusion

7.1 Achieved goals

Many areas of this project can still be very much expanded but all the project's goals were to some capacity fulfilled. The script makes it possible to (hopefully easily) process a spectrogram and view and adjust every step of the way.

7.2 Possible improvements

- **Being able to preview the JSON file** – Implementing a way to automatically view a location in a large spectrogram where segments were found might be useful. This wasn't done due to time constraints.
- **Normalizing an image only made out of noise is problematic** – Dividing by a percentile will heavily brighten the image and the segmentation process will find many invalid segments made out of bright noise values. I don't know how to deal with this. It might be useful to somehow check if the image is "consistently random" or just "weak" and just disregard it. Resolving this issue might not be practically necessary though.
- **Grouping the segments** – When processing large data it might be useful to only see large groups of segments instead of each individual one. Multiple attempts were made to group near segments together, but it turned out to be more complicated than I expected. For now, after adding the `--group-segments` option, the `JSON` file gets a `groups` key, which contains segments found on a heavily dilated version of the image. Even though there is no hierarchy involved, this might work well as a starting point when searching in a large spectrogram.

References

[1] Antonio Sánchez-García, Patricio Muñoz-Esparza, and José Luis Sancho-Gómez. A novel image-processing based method for the automatic detection, extraction and characterization of marine mammal tonal calls, 12 2010. Available at: `https://doi.org/10.1017/S0025315409000927` or online for free.

[2] Opencv documentation. online. OpenCV - Open Computer Vision Library. 2025. `https://docs.opencv.org/4.x/index.html`. [cit. 2025–01–12].