Algorithmic Framework for Sound Localization in Polyphonic Noisy Environments

Tibor Haller, Jonas Länzlinger
School of Computer Science SCS, University of St. Gallen HSG
Rosenbergstrasse 30, 9000, St. Gallen, Switzerland
E-mail: tibor.haller@student.unisg.ch, jonas.laenzlinger@student.unisg.ch

***Abstract*—In the context of sound source localization, many approaches have been proposed and implemented to solve the inherent problems and achieve high localization accuracy. However, there is currently a lack of comprehensive solutions that integrate the individual components into a single, user-friendly library that offers all necessary capabilities for sound source localization. We present *pysoundlocalization*, a Python library aimed at the localization of stationary and moving sound sources in polyphonic, noisy environments. *Pysoundlocalization* offers ready-for-use implementations to support simulation of real-world experiment setups, preprocessing of audio signals, and localization of sound sources based on time difference of arrival and multilateration, which has been applied in research due to its simplicity and effectiveness. The library is validated through four experiments that measured localization accuracy in a range of randomized setups. The results show that the library can achieve accurate localization of single and multiple stationary and moving sound sources in polyphonic noisy environments.**

***Index Terms*—Sound Source Localization, Python Implementation, GCC-PHAT, TDoA, Multilateration**

I. Introduction

The localization of sound sources in different environments has been a topic of research for the past decades [1]. With a large number of useful applications across different industries—such as robotics, healthcare, Internet-of-Things (IoT), and many more [1]—a wide range of approaches has been proposed to achieve high localization accuracy and to solve the inherent problems like reduced localization accuracy in highly noisy, polyphonic environments [2], [3]. Industry examples are manifold and include the use of passive acoustic systems to track low flying aircraft [4], particularly relevant in the detection of cross-border smuggling; the use of SONAR to localize targets underwater [5]; construction site monitoring for accident prevention [6]; noise localization for industrial plants [7]; or predictive maintenance of machines.

Although the current state of research provides a variety of methods and tools for sound source localization for many use cases, fewer contributions have focused on practical implementations that combine the necessary tools and can be used out-of-the-box. Currently, the diversity of algorithms, each with distinct strengths and weaknesses, often requires combining these components manually, a process that is both time-consuming and complex.

Some attempts have been made at providing more support for sound source localization, like the software package *pyroomacoustics* [8], which is 'aimed at the rapid development and testing of audio array processing algorithms', but not specifically for localization. Other examples include selected implementations of relevant algorithms, such as audio signal filtering, noise reduction, and multilateration-based methods for localization [9], [10], [11], [12], [13]. However, to the best of our knowledge, there is currently no open source library that combines relevant functionalities into a single, ready-for-use toolbox that supports sound source localization using regular, omnidirectional microphones placed around the sound source(s).

We present *pysoundlocalization*, a Python library aimed at the localization of sound sources in polyphonic, noisy environments. Given a real-world experiment with deployed microphones, the library creates a digital twin [14] by simulating the localization setup for effective audio signal preprocessing and sound source localization. Contents of the library can be divided into three main components:

1) The simulation of the real-world environment by digitally recreating the experiment setup and microphone placements
2) The preprocessing of the recorded audio signals to prepare for sound source localization
3) The localization of the sound sources using implementations of different state-of-the-art algorithms

The library enables users to apply the optimal preprocessing and localization methods for the specific use case. Notably, the library supports relatively simple localization setups as it only requires that:

- The microphones are placed around the sound sources
- The microphone coordinates (or the distances between microphones) are known
- The audio recordings are taken for the same time interval

Through its range of built-in preprocessing and localization algorithms, the objective of the library is to localize single or multiple stationary and moving sound sources in both non-polyphonic and polyphonic noisy environments. As part of the defined objective, the following functionalities are derived to localize sound sources:

1) Single, stationary sound source
2) Multiple, stationary sound sources
3) Multiple, moving sound sources

The report is organized as follows: Section II presents the fundamentals relevant for the introduced library, which includes the major concepts and algorithms for implementing sound source localization. Section III introduces the *pysoundlocalization* library from a design and implementation perspective. Section IV evaluates the robustness of the library through four experiments that measure localization accuracy in randomized setups. Section V presents the final considerations, including summary, conclusions, and future work.

II. Fundamentals

This section will first highlight the specific localization approach implemented in the introduced library (*pysoundlocalization*) and then focus on the related work relevant for its implementation.

A. Localization

Pysoundlocalization focuses on multilateration [1], [15] as the underlying mathematical method for localizing the sound source, which is well-established in research and widely used due to its simplicity, reliability, and effectiveness [1]. Multilateration is in principle the same as trilateration, but uses four or more sensors rather than exactly three [1]. Given the distance of each sensor from the sound source, the way multilateration localizes the sound source can be understood visually, as seen in Figure 1: For each sensor, the method creates a circle with the radius corresponding to the distance of the sensor to the sound source. The circle represents all possible sound source locations for the given sensor and the intersection of all circles is the localized sound source [1], [15].

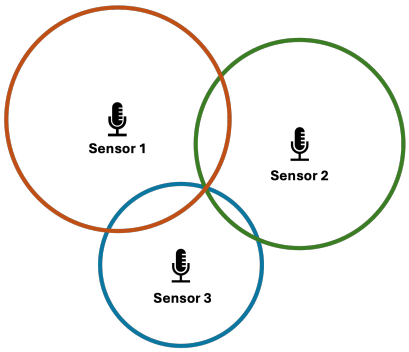


Fig. 1: Localizing the sound source at the intersection [1]

Multilateration implies that the distance of each sensor to the sound source is known. *Pysoundlocalization* uses the time difference of arrival (TDoA) [15] of acoustic signals at the sensors to determine the distances. This method measures the difference in time it takes for the acoustic signal to be captured by the sensors placed in different locations and knowing the speed of sound in the given environment, the distance to the sound source can be determined. Different approaches exist to compute the TDoA, such as the Generalized Cross-Correlation Function (GCC) [16], of which one variant was also implemented in this library. The use of TDoA implies that the sensor localizations, and by extension the distances between sensors, are known. Further, the method requires at least three sensors to capture audio signals.

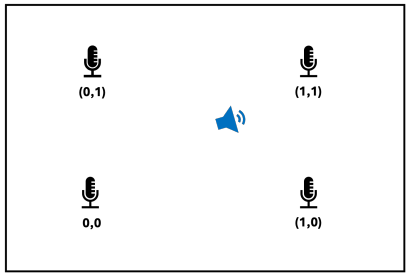


Fig. 2: A typical experiment setup for sound source localization

In practice, an experiment with deployed sensors might look like in Figure 2. Four omnidirectional microphones are placed around a sound source that should be localized. The microphone localizations are known, and one audio signal is recorded for each microphone during the same time span. Using the recorded audio signals, the library localizes the sound source via the described localization approach.

B. Selected Related Work

A robust library requires three components for effective implementation of sound source localization: a way to digitally simulate the real-world experiment to run localization algorithms, preprocessing capabilities to preprocess the possibly noisy and polyphonic audio signals, and sound source localization through implementations of state-of-the-art algorithms. Table I provides an overview of the selected related work relevant for the preprocessing and localization implementations.

Reference	Preprocessing			Localization	
	Filters	Normalizer	Separation	Multilat.	GCC-PHAT
Butterworth et al., 1930 [17]	✓				
Hussin et al., 2016 [18]	✓				
Hirano et al., 1974 [19]	✓				
Steinmetz and Reiss, 2021 [20]		✓			
Series, 2011 [21]		✓			
Lee and Seung, 2000 [22]			✓		
Lee and Seung, 1999 [23]			✓		
Widdison and Long, 2024 [15]				✓	
Popovici, 2020 [24]					✓
Knapp and Carter, 1976 [16]					✓
Chen et al., 2005 [25]					✓

TABLE I: Selected Related Work

1) Preprocessing: *Pysoundlocalization* offers a selected range of preprocessing capabilities applied to the audio signals to improve later localization accuracy.

Butterworth filter: The library includes an implementation of the Butterworth filter first introduced in 1930 by Stephen Butterworth [17], which allows frequencies

within a specified range to pass through while attenuating frequencies outside this range [18]. For example, a low-pass filter removes frequencies above a specified cutoff frequency, while the order controls the steepness of the slope of the filter. These filters work inversely: Low-pass only allows frequencies below the cutoff to pass, while high-cut achieves the same result. The filter's frequency response is maximally flat, *i.e.* the filter mask does not contain any ripples and goes towards zero. Formula 1 depicts the frequency response of a Butterworth filter [26]. Note that this is the form of a low-pass filter:

$|B(j\omega)|^2$: Squared magnitude of the filter's freq. response
$j\omega$: The angular frequency ($2\pi f$)
$j\omega_c$: The cutoff angular frequency ($2\pi f_c$)
N : The order of the filter

$$|B(j\omega)|^2 = \frac{1}{1 + \left(\frac{j\omega}{j\omega_c}\right)^{2N}} \tag{1}$$

The implementation leverages the Python *SciPy* library [10]. The filter is useful when certain frequencies, such as monotone background noise, can be excluded to improve localization accuracy.

Notch filter: The Notch filter was implemented to attenuate a specific narrow frequency band while allowing frequencies outside that band to pass through relatively unaffected [19]. The implementation leverages the Python *SciPy* library [11]. The filter is especially useful when a certain frequency decreases the audio signal quality and negatively affects localization accuracy.

Noise reducer: The library includes an implementation of a noise reducer based on a non-stationary spectral gating approach, which is helpful in improving audio signal quality in the presence of noises. Spectral gating is a technique to transform the input audio signal into its spectral representation using Short-Time Fourier Transform (STFT) and selectively attenuates specific frequency features based on the ratio between their relative loudness and a given noise threshold. The implementation leverages the open-source project *noisereduce* [9] whose work also serves as an introduction to how *noisereduce* works [27]. Noise reduction is useful when audio samples present a highly distinct background noise that standard frequency filters do not account for well.

Audio normalizer: To normalize audio signals, a reference implementation was included based on prior work by [20]. It follows the ITU-R BS.1770-4 recommendation by [21], which has seen widespread adoption [20].

Audio source separation: When audio is recorded in a polyphonic environment, different overlapping sound sources are captured within the same audio signal—also called a mixture. For example, two different sounds playing at the same time are captured with a single sensor and the resulting audio signal is a mixture of both sounds, where individual components can become hard to distinguish. For sound source localization, it can become difficult to focus on the correct sound source for localization—especially when the desired sound source is not clearly distinguishable from the other sounds. The library implements a blind sound source separation (BSS) method based on Non-Negative Matrix Factorization (NMF) [22], initially introduced by [23] and later adapted to different problems of separation of sound sources [22], [28], [29], [30]. Given the number of sound sources as input, the algorithm splits the audio signal into its most distinct components, which can be used for more accurate localization.

The general approach consists of reconstructing and factorizing a presented data matrix V into two non-negative unknown matrices W and H until the product of them produces the original matrix, see Formula 2 [31].

$$V \approx WH = \hat{V} \tag{2}$$

This approximation is done while the following conditions are satisfied (non-negativity), Formula 3 [31].

$$\begin{aligned} W &\geq 0 \\ H &\geq 0 \end{aligned} \tag{3}$$

The estimation of the matrices W and H is obtained by optimizing a given cost function—*pysoundlocalization* uses the Euclidean cost function—by iteratively updating a gradient descent. The estimation function is expressed in Formula 4 [31].

$$\arg\min_{W,H\geq 0} D(V\|WH) = D(V\|\hat{V}) = \sum_{i=1}^{K}\sum_{j=1}^{N} d(V_{i,j}\|[WH]_{i,j}) \tag{4}$$

Moving sound sources: The problem of localizing a moving sound source has been a topic of research for some time. Though localization methods such as TDoA are well-suited for localization of static sound sources, the frequency shift caused by the Doppler effect creates issues in localization accuracy for moving sound sources [32]. The library does not implement a localization algorithm that accounts for the Doppler effect. However, it allows for accurate localization of moving sound sources that are stationary while the sound is emitted and only move their position in-between. The library's built-in chunking method allows for splitting the audio into chunks of a defined size, which allows for continuous sound source localization. By selecting chunks of a meaningful size, the audio signal can be repeatedly localized when it stops moving and emits a stationary sound.

2) Localization: Pysoundlocalization offers a selected range of localization algorithms to perform multilateration of one or multiple sound sources given that a) the sensors—such as regular, omnidirectional microphones—are placed around the sound source(s); b) that sensor coordinates—and implicitly the distances between the sensors—are known; and c) that all audio signal recordings are taken for the same time interval.

Multilateration: The library implements a multilateration algorithm to localize a sound source given sensor coordinates and the TDoA values computed using the recorded audio signals. Different approaches have been proposed to solve the multilateration problem [15]. The library implements an iterative, non-linear least squares algorithm based on an implementation by [24]. The constructed linear equations, see representative Formula 5 are collected to form a system of linear equations, see Formula 6 that are solved using the least squares algorithm, where M corresponds to the microphones, and the coefficients A_m, B_m, and D_m express the differences (microphone positions and reference position) in the x- and y-coordinates, scaled by the speed of sound v and the time delay τ. This procedure yields the estimated coordinates (x, y) [24].

$$0 = D_m + A_m x + B_m y, \tag{5}$$

$$\begin{bmatrix} A_2 & B_2 \\ A_3 & B_3 \\ \vdots & \vdots \\ A_M & B_M \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -D_2 \\ -D_3 \\ \vdots \\ -D_M \end{bmatrix} \tag{6}$$

TDoA using GCC-PHAT: The implemented multilateration algorithm requires TDoA values as input. Different approaches have been proposed to find the time differences, focusing on common challenges like distortions through noise or reverberations [25]. The library includes an implementation of the Generalized Cross-Correlation with Phase Transform (GCC-PHAT) algorithm [16], [33] as it has been shown to be robust against these challenges [34], [35].

GCC-PHAT consists of two main components. The first part is the Generalized Cross-Correlation (GCC) that estimates the TDoA values of signals from two microphones. The GCC equation is defined below, see Formula 7 [36]:

$$\begin{aligned} X_i(\omega) &: \text{Signal of microphone } i \\ X_j(\omega) &: \text{Signal of microphone } j \\ \omega &: \text{Angular frequency} \\ [.]^* &: \text{Conjugate transpose operation} \\ \Psi_{ij}(\omega) &: \text{Weight func. to optimize perf. criteria} \end{aligned}$$

$$R(\tau) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} \Psi_{ij}(\omega) X_i(\omega) X_j^*(\omega) e^{j\omega\tau} \, d\omega, \tag{7}$$

The second part is the phase transform (PHAT) defined below, see Formula 8 [36]. It can be interpreted as a filter that discards the amplitude, but preserves the phase of a given signal [36].

$$\Psi_{ij}(\omega) = \frac{1}{|X_i(\omega) X_j^*(\omega)|} = \frac{1}{|X_i(\omega)||X_j(\omega)|}, \tag{8}$$

GCC-PHAT is useful for noisy and heavily distorted audio signals, as no assumptions about the signal or room conditions have to be made, which are typically unknown [36].

TDoA using Threshold peak matching: For simple experimental setups, a straightforward algorithm has been implemented that selects the highest peaks of the audio signals to compute the TDoA values. The algorithm is efficient when each recorded audio signal has a single distinct peak that correlates with the sound source to localize. However, if the audio signals have multiple peaks, are convoluted or include multiple sound sources, the algorithm quickly experiences issues in accuracy as finding clear—and correct—audio peaks becomes difficult. This approach is particularly effective in experiment setups when there is a single, clearly identifiable peak in the audio signal that matches with the sound source that should be localized, such as when the sound source emits a single signal that is distinguishably louder than all other sounds.

DoA: The computation of the direction of arrival (DoA) of a sound source is not strictly necessary for the sound source localization. However, the library implemented a method for DoA computation as it is closely related to TDoA and often useful.

C. Summary

The related concepts and methods discussed form the theoretical foundation for the newly introduced library: *pysoundlocalization*, supporting an implementation of well-established preprocessing and localization capabilities for practical applications. An array of preprocessing methods, such as filtering, noise reduction, and audio source separation, improve audio signal quality for accurate sound source localization. For localization, the library leverages multilateration and TDoA estimation, particularly through the Threshold and GCC-PHAT algorithms, to achieve accurate sound source localization even in noisy or complex environments. By implementing these methodologies within a cohesive framework, *pysoundlocalization* bridges the gap between theoretical principles and practical applications.

III. PYSOUNDLOCALIZATION

In the applied part of this project, we worked on the creation of an exploratory Python library for sound localization. In the following sections, the proposed system design and implementation of the *pysoundlocalization* library is presented. Additional periphery work is described, such as recording setup, both streaming and standalone, test data collection, and extensive hands-on example scripts of how to utilize the library. Figure 3 illustrates the fundamental functions provided by the library.

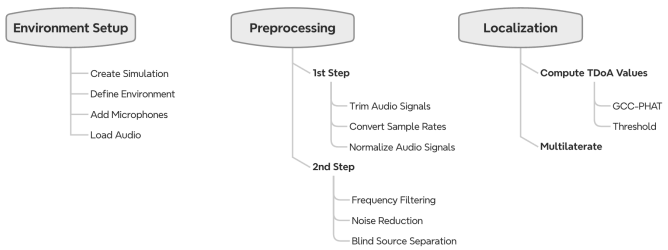


Fig. 3: *Pysoundlocalization* - functionalities

A. Introduction

Pysoundlocalization is a Python library for sound source localization in noisy polyphonic environments and is a specific contribution that can be used for many different applications. This highlights the difficulty of not knowing the optimal preprocessing approach and the localization algorithm choices a priori. The library equips users with a toolbox of functionalities that support the process of finding the domain-specific, optimal preprocessing and localization techniques.

In the following sections, a deep dive into the design and implementation of the proposed *pysoundlocalization* library is presented.

B. Requirements and Assumptions

Localization systems require the consideration of different factors that influence their implementation, accuracy, and performance. This section summarizes the requirements and assumptions that were taken for the proposed system.

1) Requirements: The following requirements are defined for the design of the library.

Audio signal timing: It is required that the audio signals are synchronized before localization is performed, particularly so that TDoA values can be computed accurately. If necessary, the library provides utility functions to synchronize the audio signals based on their recording start timestamps. Generally, the library assumes synchronized audio signals. If audio signals are neither synchronized nor possible to be synchronized using known timestamps, accurate localization becomes impossible. Further, even with synchronized audio signals, it is required that the recordings are latency-free as limitations, such as from the recording hardware, can introduce inaccuracies in localization.

Audio signal format: The audio signals that are provided to the system are expected to have an identical length (sample count) and sample rate; otherwise, the source positions are wrongly approximated. However, the library provides utility functions to adjust the sample rates.

Audio signal characteristics: The provided audio signals are expected to exhibit clearly identifiable amplitude peaks. This is a prerequisite for one of the proposed localization algorithms, *i.e.,* the Threshold algorithm, as it is based on those amplitude peaks. Furthermore, the to-be-localized sounds cannot be both monotone and persistent at the same time, *e.g.,* a single frequency sine wave spanning over multiple seconds; otherwise, GCC-PHAT fails.

2) Assumptions: The assumptions define the scope and the implemented capabilities of the library.

Dimensionality: While this library is a starting point for a fully-fledged analysis tool, the scope is defined for planar 2D localization. This reduces complexity in terms of processing logic, recording setup, and computational overhead while making it easy to extend the system towards 3D localization in the future.

Speed of sound: As the speed of sound strongly depends on temperature as well as the medium through which a sound wave is propagating, the default library configuration is $343.2 m/s$. This corresponds to the speed of sound in dry air conditions at a temperature of 32°F. Users can easily adjust this to a desired value in the library via the configuration file, *i.e., config.py*.

Detection area: When microphones are positioned within the environment, they define the detection area. The sound sources are expected to be found inside this area; otherwise, the accuracy of the localization algorithm declines rapidly.

Result interpretation: If the sound source duration is greater than the localization update interval, the visualization result might not be as expected. In the visualization plots, only the interval in which the peak of a given sound source appears displays the source position.

C. Structure

The library is organized into different packages that are logically separated; see Figure 4. Together with the library, a collection of test audio data, various demo examples, the conducted experiments, and the documentation form the content of the repository.

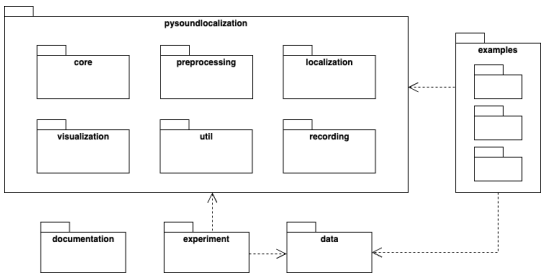


Fig. 4: *Pysoundlocalization* - package diagram

The following section gives an overview of the modules in *pysoundlocalization*:

Core: Fundamental data entities that orchestrate the library's core functionalities, see section III-D1.

Preprocessing: A collection of different preprocessing tools that can be applied to the audio data at runtime such as various filters, converters, noise reducer, amplitude normalizer, and a source splitting algorithm, see section III-D2.

Localization: Algorithms used for localization, such as the TDoA computation, either by GCC-PHAT or Threshold, and the multilateration implementation, see section III-D3.

Visualization: Various visualization plots tailored for the usage with the domain-specific entities such as simulation, environment, microphone, and audio.

Util: Contains utility functions for setting up synthetic experiment data to be used with the library.

Recording: Script for both—recording audio data with the microphone connected to the device, and audio streaming to a central processing unit—, as well as sample scripts for editing a given audio recording file towards the required shape.

D. Workflow Pipeline

The concept of the localization workflow can be divided into three distinct phases:

1) Environment setup
2) Preprocessing
3) Localization

The implementation follows a forward-only progression through the individual phases. While this behavior is not enforced, it is recommended.

1) Environment setup: In the first phase of the pipeline, the experiment needs to be defined. The experiment is represented internally as a digital twin that lets the user interact with it in an intuitive manner. For this, the user is presented with the following entities, see Figure 5.

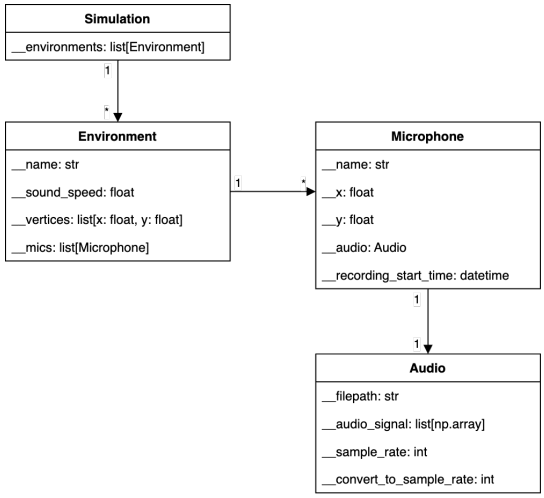


Fig. 5: Module "core" - UML diagram

Simulation: This is the top-level entity of the system. Through the maintained list of environments, it is possible to switch back and forth between different environments, enabling their parallel comparison at runtime.

Environment: The environment represents one individual recording setup. The environment can be defined with its own custom speed of sound setting and is constructed by a list of vertices passed by the user. The environment allows for the addition of microphones and verifies whether each microphone is positioned within the defined boundaries of the environment. By passing the vertices and adding the microphones, a digital twin of the real world can be modeled.

Microphone: Microphone objects are placed within the environment, and together, they define the detection area that yields the best results for the localization. Once a microphone is placed, an audio recording can be associated with it. By this, the recorded audio gets linked to a position within the environment. As defined by the III-B1, the audio signal needs to have a recording start time, which is done by the microphone object.

Audio: The library offers two ways how to create an audio object: 1) Provide a file path to an audio recording, which then independently loads the signal from the file. This is the correct choice for localization setups where the audio recordings are already at hand. 2) The second option is *Audio.create_from_signal()*: An audio object is created by providing an audio signal, *i.e.,* a *np.array* consisting of amplitude values ranging from -1.0 to 1.0. It is important to also provide the sample rate for the newly created audio object, as this determines the resolution of the audio signal and, therefore, the total duration of the audio; see the provided Formula 9:

$$\begin{aligned} t_{\text{signal}} &: \text{Duration of the signal (in seconds)} \\ N_{\text{samples}} &: \text{Total number of samples in the signal} \\ f_s &: \text{Sample rate (in Hz, samples per second)} \end{aligned}$$

$$t_{signal} = \frac{N_{samples}}{f_s} \tag{9}$$

Furthermore, when stereo audio data is provided to the object, it is formatted to mono automatically. Let's examine how the audio signal is stored: The audio signal is a list of *np.array* elements. The reason is that when the user wants to continuously locate the sound source position by an update interval, the audio signal needs to be split into those intervals, *i.e.,* chunks, see III-D3. Note that when the list of audio signals contains only one *np.array*, the signal is implicitly unchunked. The audio entity provides several utility functions, *e.g.,* resampling the audio signal to a target sample rate, splitting the audio signal into chunks—either by duration in milliseconds or by number of samples—, exporting the audio signal as a *.wav*-file to the current location in the file system, or playing (audible) the audio signal.

2) Preprocessing: In the preprocessing phase, two different activities can be performed. The recommended approach is to first bring the loaded audio signals in the correct format. Regarding the requirements III-B1, we established that the loaded audio signals need to follow various criteria before being suitable for the localization, such as 1) identical duration, *i.e.,* number of samples, 2) identical sample rate, and therefore, 3) have the same *recording_start_time*. The library supports the user in meeting these criteria for the audio signals in a seamless way.

SampleRateConverter: Multiple utility functions to convert the sample rate of single audio signals or entire environments and its loaded audio signals. The library offers functions for converting both environment and audio objects. Every function implements specialties of how to handle the different entities, but they ultimately call the external function *librosa.resample()* [37].

SampleTrimmer: Helper functions to trim the audio signal from the beginning or the end, as well as slicing certain portions from the signal, by either providing a duration, or a number of samples. The method *SampleTrimmer.sync_environment()* trims automatically every loaded audio signal in a given environment according to the following Formulas 10, 11, 12, 13:

ts_{start} : Audio start timestamps

TS_{start} : List of all audio start timestamps

$ts_{\text{sync\_start}}$: Start timestamp of synced audios

$ts_{\text{sync\_end}}$: End timestamp of synced audios

T_{audio} : List of all audio durations

t_{sync} : Duration of synced audios

a_{sync} : Synced audio

a : Audio

$$ts_{\text{sync\_start}} = \max(TS_{\text{start}}) \tag{10}$$

$$ts_{\text{sync\_end}} = \min(TS_{\text{start}} + T_{\text{audio}}) \tag{11}$$

$$t_{\text{sync}} = TS_{\text{sync\_end}} - TS_{\text{sync\_start}} \tag{12}$$

Each audio object is then trimmed to:

$$\text{a}_{\text{sync}} = \text{a}[ts_{\text{sync\_start}} - ts_{\text{start}} : ts_{\text{sync\_start}} - ts_{\text{start}} + t_{\text{sync}}] \tag{13}$$

AudioNormalizer: The localization of sound sources is based on the timestamps of peak amplitude levels above a certain threshold. When various filters are applied to a given audio signal, the amplitude of the signal vanishes increasingly. Therefore, the AudioNormalizer can be used to keep the amplitude at desired levels, see Figure 6.

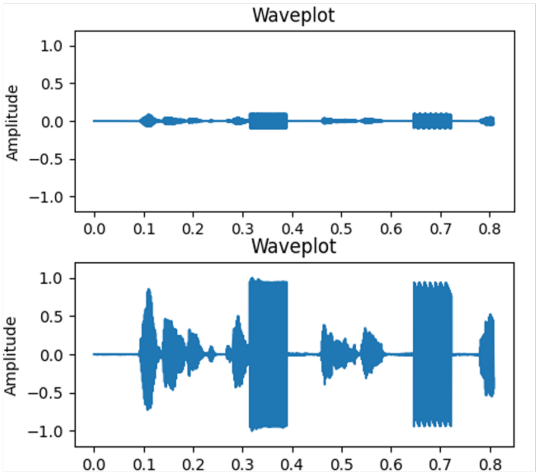


Fig. 6: Audio normalizing

The normalization procedure is implemented as depicted in Formulas 14, 15, 16:

chunk_i : i-th chunk of the audio signal

A_{target} : Max. amplitude after normalization

$A^{(i)}_{\text{max}}$: Maximum amplitude of i-th chunk

scale_factor_i : Scaling factor of i-th chunk

$\text{chunk}^{(i)}_{\text{norm.}}$: Normalized i-th chunk

$$A^{(i)}_{\text{max}} = \max(|\text{chunk}_i|) \tag{14}$$

$$\text{scale\_factor}_i = \frac{A_{\text{target}}}{A^{(i)}_{\text{max}}} \tag{15}$$

$$\text{chunk}^{(i)}_{\text{norm.}} = \text{chunk}_i \cdot \text{scale\_factor}_i \tag{16}$$

The second activity includes the preprocessing of the audio data in terms of spectral features. The initial goal of this project is to localize sound source positions in noisy environments where the localization itself is performed based on amplitude peaks. Depending on the data and use case, identifying peaks might get tricky due to potential noise. There are cases where the noise is too loud to even hear the sound source at all. Therefore, the task is to apply filtering to isolate certain sounds. The library supports finding the optimal filters for a given use case.

In the library, three different types of processing tool are implemented: Frequency filtering, noise reduction, and blind source separation.

Frequency filtering: As described in detail in section II, frequency filtering consists of digital filters that are applied to the input signal, producing an output signal of which certain frequency regions have been attenuated. Consequently, an output signal can be redirected and used as the input signal for another frequency filter. Thanks to this property, chaining different frequency filters becomes possible. In the library, this attribute is leveraged by the FrequencyFilterChain. The design of this functionality is closely related to the Chain of Responsibility design pattern. Each concrete frequency filter implementation is a subtype of the interface IFrequencyFilter to ensure each filter has the same capabilities. In this case, the interface only defines the method *IFrequencyFilter.apply()*, which takes an audio object as a parameter and returns the modified audio object itself. The FrequencyFilterChain object holds a list of IFrequencyFilter objects that can be added or removed from the chain at runtime. Once the modeling of the desired filter chain is done, by invoking the *FrequencyFilterChain.apply()* method, the audio objects get passed through all filters in this chain in a queue-like manner. If the user wants to apply the identical filter chain to all audio signals in a given environment, he may just use the same chain for all audio signals. Examples of the filter chain and the application of filters on audio signals can be found in the respective example implementation from the library [38].

The proposed library uses the Butterworth filter design, which has been discussed in detail in section I.

Here is a list of all frequency filters included in the library. Note that a High[Cut/Pass]Filter is the complement of a Low[Cut/Pass]Filter and vice versa:

- *LowCutFilter:* Useful for removing low frequencies, for example, when machine rumbling noises need to be removed.
- *LowPassFilter:* Useful for letting low frequencies pass, for example, when the humming sound of a power line needs to be isolated.
- *HighCutFilter:* Useful for removing high frequencies, for example, when sharp sibilance sounds in speech audio need to be removed.
- *HighPassFilter:* Useful for letting high frequencies pass, for example, when bird chirping needs to be isolated.

- *NotchFilter:* Useful for attenuating one specific frequency band, for example, when an alarm sound needs to be canceled out of the audio data.

Frequency filtering is the most suitable approach when the frequency domain of the sound source is clearly separable from the most prevalent background noise. In this case, inspecting the frequency domain by using the *spectrogram_plot* visualization, can help in identifying the specific background noise frequencies that need to be attenuated.

Noise reducing: When the frequency domain of the sound source is not clearly separable from the background noise, *i.e.,* the sound source shares frequency bands with the noise, frequency filtering is not sufficient. In these cases, a noise reduction technique is needed. In the *pysoundlocalization* library, the open-source tool *noisereduce* [9] is used *Pysoundlocalization* relies on the non-stationary noise reduction approach, where the assumption is made that the noise is not monotone, *i.e.,* varies in terms of the intensity or the frequency spectrum. Moreover, *noisereduce* estimates a noise profile either from a silent segment or using an adaptive model during low-energy frames. It then applies a frequency mask, that suppresses the frequencies of the signal where the noise is dominant. The facade class NoiseReducer in *pysoundlocalization* removes much of the complexity and makes noise reduction very easy.

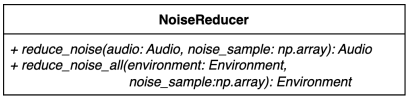


Fig. 7: NoiseReducer - UML diagram

As depicted in Figure 7, the NoiseReducer provides both a method for reducing the noise in a single audio signal and a method for reducing the noise for each audio in the environment. Optionally, a noise sample can be passed, which may lead to even better results.

Blind source separation: As defined in functionality 2, the library achieves the localization of multiple sound sources at the same time. In this setting, the microphones in the recording setup capture the sound signals of all sound sources in the environment in parallel and produce a mixed recording signal. Since the localization of a sound source is based on the TDoA values that are obtained from the comparison between the amplitude peaks, this mixed signal becomes an issue, as it is not possible to decide which sound source a certain peak belongs or sometimes even to identify a peak.

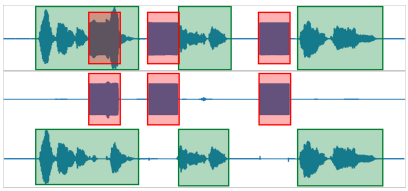


Fig. 8: Mixed audio signal

In Figure 8, the mixed audio signal of two distinct sound sources is depicted. Like this, it is impossible to tell which peak belongs to what sound. The solution is to utilize the frequency representation.

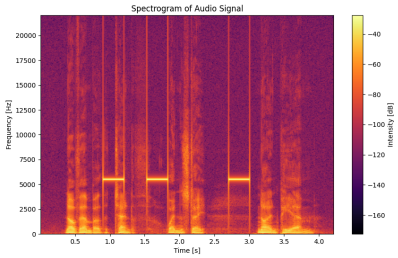


Fig. 9: Mixed audio signal spectrogram

Figure 9 shows the spectrogram of the exact same audio signal and reveals that, indeed, two different sound emitters are present:

- *Source 1:* Monotonous sound source, emitting three sounds with frequency $5'500hz$ at an interval of about half a seconds.
- *Source 2:* Complex sound source, emitting sounds around the entire frequency band.

These differences in the frequency representation can be leveraged for the localization of multiple sound sources. The approach in *pysoundlocalization* is to separate the mixed audio signal into N individual audio signals that are individually localized. There are multiple BSS algorithms such as Independent Component Analysis (FastICA), see implementation by Sklearn [39], or NMF II-B1, see reference implementation [31]. Testing both approaches in *pysoundlocalization*, see *ex_fast_ica.py*[38] and *NonNegativeMatrixFactorization.py*[38], the approach with NMF yields substantially better results. As the output array of audio signals produced by the NMF algorithm is non-deterministic, running NMF for each microphone individually raises two questions: At which position of the NMF output array do we find which sound source audio signal and—see Figure 10—whether a given audio signal, so the separated sound source, is at position zero for both microphones.

```
mic_1_splits = nmf.run(mic_1)
mic_2_splits = nmf.run(mic_2)

mic_1_audio_0 = mic_1_splits[0]
mic_2_audio_0 = mic_2_splits[0]
```

Fig. 10: NMF - non-determinism

To solve this problem, the generic approach is to first create the environment, preprocess the audio data, and then export the audio splits for each microphone to the file system. In a second script, recreate the environment and load the audio signals into the microphones correctly ordered, after which the localization can be done.

In scenarios where real time localization or a fully automated approach is needed, the library offers the following workaround:

```
dictionary = nmf.run_for_environment(environment=env_1)
```

Fig. 11: NMF - workaround

The implemented procedure is described in this example as follows. The corresponding Python code of this example can be found in *ex_nmf.py* [38].

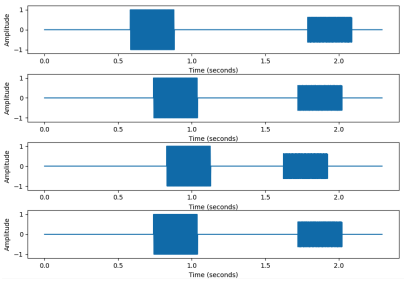


Fig. 12: NMF - original audio

Figure 12 depicts the original loaded audio signal for the four microphones in the environment:

Step 1: All audio signals of the four microphones in the passed environment are concatenated into one single audio signal; see Figure 13.

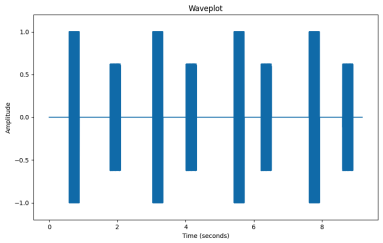


Fig. 13: NMF - concatenated

Step 2: The concatenated audio signal is passed to the NMF algorithm to separate the signal into the sound sources. The result is an audio signal array representing the separated sound sources, *i.e.,* two elements in this example, see Figure 14.

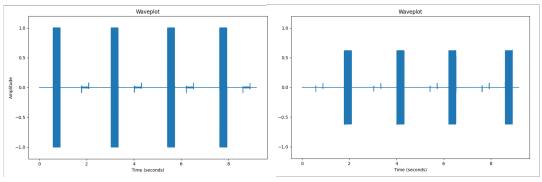


Fig. 14: NMF - splits

Step 3: The next step is to split the two sound source audio signals into the respective parts for the four microphones; see Figure 15.

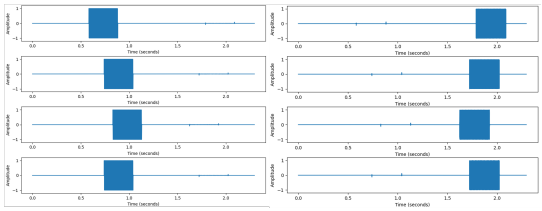


Fig. 15: NMF - split microphones

NMF - sample loss: Because NMF is a BSS technique with losses, the output signal consists of fewer samples than the input signal, and unfortunately, it is not possible to find out which exact samples were lost. According to the tests that we have performed, see *nmf_sample_loss_test.py*, the loss of samples is expected to be around 0.018% of the total duration, which can cause considerable time shifts in the audio signal that influence the outcome of the localization. The implementation of *pysoundlocalization* takes this loss into account by splitting the separated sound source signals according to this Formula 17:

$$
\begin{aligned}
l_{\text{orig}}^{(i)} &: \text{Original } i\text{-th microphone signal length} \\
l_{\text{adj.}}^{(i)} &: \text{Adjusted } i\text{-th microphone signal length} \\
L_{\text{total}} &: \text{Total original signal length} \\
S_{\text{loss}} &: \text{Total sample loss}
\end{aligned}
$$

$$
l_{\text{adj.}}^{(i)} = l_{\text{orig}}^{(i)} - \frac{l_{\text{orig}}^{(i)}}{L_{\text{total}}} \times S_{\text{loss}} \tag{17}
$$

Step 4: To localize the two sound sources, the microphone splits of each sound source are loaded iteratively into the respective microphones. The localized source position of each iteration is appended to a dictionary that keeps track of all sound source locations.

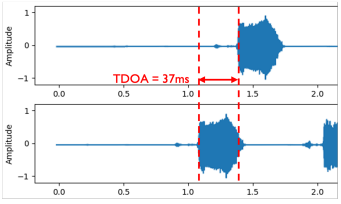


Fig. 16: TDoA illustration

3) Localization: Finally, the prepared audio data is ready for localization. To localize the sound sources, the TDoA, see Figure 16, between the microphones, needs to be calculated. To obtain these TDoA values, the *pysoundlocalization* library offers two different approaches. 1) The first approach is the GCC-PHAT algorithm, where the cross-correlation of the signals is calculated as described in the section I.

2) The second approach is the Threshold algorithm proposed by the authors, where the timestamp of the peak amplitude where the predefined threshold level is exceeded is used, see Formula 17, to compute the TDoA values: Build all unique microphone pairs, combinations (n choose 2),

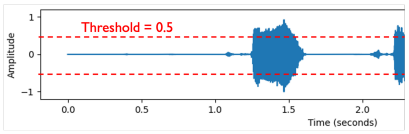


Fig. 17: Threshold illustration

see Figure 18, and compute the delta of their timestamps where the threshold level has been exceeded.

$$\binom{n}{2} = \frac{n(n-1)}{2} \tag{18}$$

This approach is an exact algorithm that achieves substantially better performance. Note that for this approach, the normalization of the audio signal to ensure that peak amplitudes are above the desired threshold level, as well as a thorough preprocessing of the raw audio data, is needed to achieve good results. The implementations for both the GCC-PHAT and Threshold algorithms used in the library can be viewed on the project's GitHub repository [38].

The preceding multilateration step is identical for both approaches. The approximation of the sound source location is implemented as described in Section II-B2. Given the initial position, this algorithm approximates the location of the sound source by performing a least-squares minimization. Note that the algorithm only implements a 2D approximation, but it can easily be expanded into 3D space. *Pysoundlocalization* initializes the sound source position with the coordinates of the center within the detection area. The output of the multilateration function is the tuple consisting of the x- and y-coordinates of the estimated sound source location.

When the system needs to locate multiple sound sources, special consideration must be given to the multilateration process. In such cases, the NMF algorithm produces an array of audio signals, where each array element represents one sound source. Therefore, the multilateration method needs to be called for each array element: First, iterate over the indices of the audio signal array. Then, at each iteration, the respective array element splits are loaded into the microphones, after which the *Environment.localize()* method gets invoked to obtain the source position of the currently loaded sound source. In the following iterations, the remaining signals are loaded and individually localized. At each iteration, an array of positions is maintained to collect the sound source positions. This array can be passed to the visualization plot; see Figure 18.

Chunking - Moving sound sources: Even though functionality 3 defined the capability of localizing moving sound sources, the focus has been only on stationary sound sources.

The proposed solution for localizing moving sound sources is called chunking. Section III-D1 explains how the audio entity stores a list of audio signals. Initially, the list holds one element at index 0, *i.e.,* the loaded audio signal. When multilaterating the sound source positions,

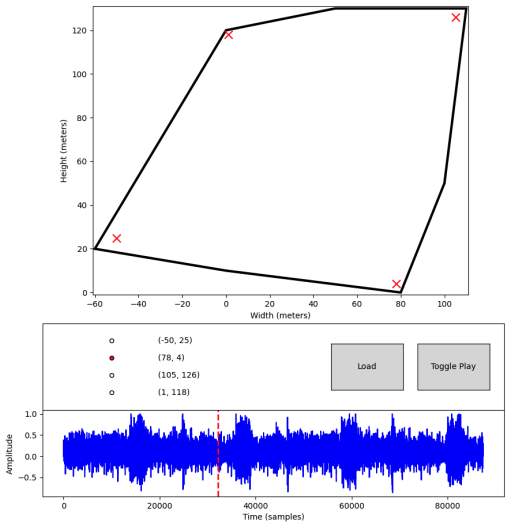


Fig. 18: Result - localization of two sound sources

only one position will be computed, namely the position of the sound source where it first appeared in the audio wave. If the sound source is in motion, the position would not be updated. For this, chunking needs to be applied, and *pysoundlocalization* offers two ways of doing this:

- *Split by chunk time:* Splits the audio signal into chunks of the given duration and stores them in the list of audio signals in the audio entity.
- *Split by chunk samples:* Splits the audio signal into chunks of the given number of samples and stores them in the list of audio signals in the audio entity.

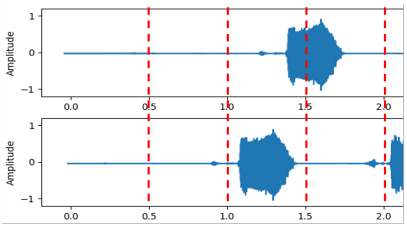


Fig. 19: Chunking

Figure 19 illustrates the concept of the mechanism. When the audio is chunked, the localization loops through all chunks. For each iteration, the TDoA values are computed and used individually for the multilateration. Therefore, the output of the multilateration is a dictionary containing the start sample index of each chunk, mapped to either *None*, which equals to no sound present in the chunk, or (x, y), which equals to the location of the sound source event during the chunk. The dictionary is then appended to the *source_positions* array, which stores the localization dictionaries for each separated sound source.

The implemented chunking procedure for one audio signal is described in Formulas 19, 20, 21:

a : Original audio signal

C_i : Chunk at i-th position

t_c : Chunk duration (in seconds)

f_s : Sample rate (in Hz, samples per second)

S_c : Chunk duration (in samples)

$C_{pad}^{(i)}$: Last chunk, *zero*-padded

$$S_c = f_s \times t_c \tag{19}$$

$$C_i = a[i \times S_c : (i+1) \times S_c] \tag{20}$$

For the last chunk:

$$C_{pad}^{(i)} = \begin{cases} C_i, & \text{if } \text{len}(c_i) = S_c \\ \text{pad}(C_i, S_c - \text{len}(C_i)), & \text{if } \text{len}(c_i) < S_c \end{cases} \tag{21}$$

E. Visualizations

As *pysoundlocalization* is an exploratory library, visualizations are an essential component. The *visualization* module holds a collection of various plots that are summarized in Table II.

F. Audio generation

A valuable tool in the *pysoundlocalization* library is the generation of raw audio data in a constructed experiment setup, which brings benefits to the user in terms of exploration of library functionalities, quick algorithm assessments, and comparison of different preprocessing techniques. The *simulate_noise_util.py* module implements the *generate_audios* method, see Figure 20, which is a helper function for generating audio signals.

```
def generate_audios(
    environment: Environment,
    sample_rate: int,
    sound_sources: list[dict[int, tuple[float, float]]],
    background_noise: Audio | None = None,
    loudness_mix: list[float] | None = None,
    default_sound_duration: float = 0.3
) -> Environment:
```

Fig. 20: Generate audios - signature

By providing the environment along with the following parameter values, the method returns the environment containing the generated audio signals loaded into the microphones:

- *environment:* The environment contains the room dimensions (vertices) and the placed microphones.
- *sample_rate:* Defines the target sample rate for the generated audio signals.
- *sound_sources:* The list defines for each entry a unique sound source that occurs in the environment. Depicted in Figure 21 is an example of such a list.

The dictionaries for the sound sources define the sample index as the key and a tuple (x, y) for its value. Note that the sample index is valid for the reference location in the environment, which is the location of the passed sound at a given sample index.

```
sound_sources = [
    {
        "sound_1": sound_1,
        15000: (50, 85),
        37000: (100, 85),
    },
    {
        "sound_2": sound_2,
        12000: (20, 72),
        24000: (25, 74),
        39500: (24, 77)
    }
]
```

Fig. 21: Generate audio - sound_sources

The implementation determines the total length of the output audio signal as follows, see Formulas 22, 23, 24, 25:

X, Y : Coordinates for environment vertices

v_{sound} : Speed of sound (in m/s)

f_s : Sample rate (in Hz, samples per second)

d_{max} : Max. poss. delay for env. (in meters)

Δt_{max} : Max. poss. delay for env. (in seconds)

$S_{\Delta t}$: Max. poss. delay for env. (in samples)

I_{last} : Last input signal sample index

S_{max} : Max. input signal length (in samples)

S_{output} : Output signal length (in samples)

$$d_{max} = \sqrt{(\max(X) - \min(X))^2 + (\max(Y) - \min(Y))^2} \tag{22}$$

$$\Delta t_{max} = \frac{d_{max}}{v_{sound}} \tag{23}$$

$$S_{\Delta t} = f_s \cdot \Delta t_{max} \tag{24}$$

$$S_{output} = I_{last} + S_{max+S_{\Delta t}} \tag{25}$$

Now, these resulting output audio signals are constructed in such a way that they simulate and reflect the times of arrival of a certain sound source between the microphone locations and the sound source's own current location. The implemented behavior for constructing the audio signal of one microphone is described as follows, see Formulas 26, 27, 28, 29, 30, 31:

sse : Sound source event

d : Distance between sse and mic. (in meters)

v_{sound} : Speed of sound (in m/s)

t_{delay} : Delay between sse and mic. (in seconds)

f_s : Sample rate (in Hz, samples per second)

S_{delay} : Delay between sse and mic. (in samples)

I_{start} : Start sample index of sse at mic.

I_{sse} : Start sample index of sse

a_{sse} : Audio signal of sse

a_{scaled}^{sse} : Mixed audio signal of sse

$mix[i]$: Loudness mix i-th element

a_{output}^{sse} : Output audio signal of sse

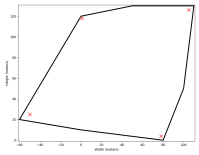
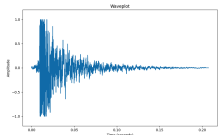
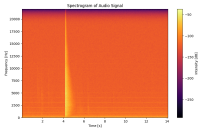
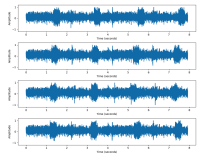
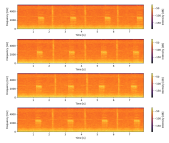
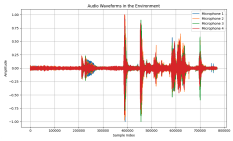
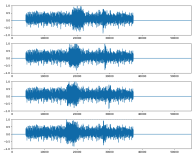
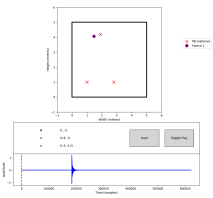
Plot Name	Description	Pictogram
Environment plot	Plots the environment with the defined dimensions and the contained microphones.	
Audio wave plot	Plots the audio signal waveform.	
Audio spectrogram plot	Plots the audio signal spectrogram.	
Environment wave plot	Plots all audio wave plots loaded in the environment.	
Environment spectrogram plot	Plots all audio spectrogram plots loaded in the environment.	
Environment overlap wave plot	Plots all audio wave plots loaded in the environment in the same plot.	
Environment play audio plot	Plots all audio wave plots loaded in the environment and plays the audio while updating the plot continuously.	
Multilaterate plot	Plots the environment with the defined dimensions and the contained microphones. It lets the user select an audio signal from a radio button list that can be played via the play and stop buttons. While playing, the multilaterated locations per sound source are displayed in real-time.	

TABLE II: Module “visualization” – plots

For each *sound_source* entry, perform the following for each sound source event:

$$d = \sqrt{(x_{sse} - x_{\text{mic}})^2 + (y_{sse} - y_{\text{mic}})^2} \tag{26}$$

$$t_{\text{delay}} = \frac{d}{v_{\text{sound}}} \tag{27}$$

$$S_{\text{delay}} = f_s \cdot t_{\text{delay}} \tag{28}$$

$$I_{\text{start}} = I_{sse} + S_{\text{delay}} \tag{29}$$

$$\mathrm{a}^{sse}_{\text{scaled}} = \mathrm{a}_{sse} \cdot \text{mix}[i] \tag{30}$$

$$\mathrm{a}^{sse}_{\text{output}}[I_{\text{start}} : I_{\text{start}} + \text{len}(\mathrm{a}_{sse})] += \mathrm{a}^{sse}_{\text{scaled}} \tag{31}$$

background_noise: It is possible to add a custom audio signal as the background noise, which will be mixed with the sound source audio signals. The background noise is continuously repeated for the total output duration.

loudness_mix: The loudness mix offers a way to adjust the volume levels of the individual sounds passed to the

method. The list needs to have the exact same number of elements as the background noise together with the not-*None* sound sources. All elements in the list need to have values between 0.0 and 1.0.

default_sound_duration: If sound sources of a dictionary in *sound_sources* are *None*, an artificial sound is used for the duration given by this parameter. For the generation of artificial sounds, the method *generate_maximally_different_sounds()* is used. It defines a set of generator functions for the basic waveforms such as sine wave, sawtooth wave, square wave, triangle wave, and white noise, along with a range of valid frequencies between which the method can independently interpolate to generate maximally different sounds.

G. Module - Recording

For this project, we investigated potential designs of an optimal cost-efficient recording setup for the proposed library. As established in the beginning, the localization approach of *pysoundlocalization* requires the exact positions of the microphones to be known, as well as the recordings to be exposed to as little latency as possible. Both requirements are essential for precise localization results.

The module contains the following components:

streaming.sh: This scenario assumes the setup of streaming the recorded audio signals from the microphones to a central processing unit. For this setup, we place a dedicated router in the room, to which four Raspberry Pi's (RP) that are hooked to the microphones and the processing unit are connected. When starting the script, the processing unit accesses the RP's and starts the recording with the tool called *GStreamer* [40]. At each RP, *GStreamer* establishes a pipeline that streams the recorded audio directly to the processing unit. The processing unit on the receiving end collects all of those audio streams and saves the data on its disk.

record.py: The approach with the minimal latency overhead involves recording over the local device itself. In this case, the microphone is connected to the device and can be selected at runtime via this script by prompting the user for the desired recording parameters. The recordings are saved on the disk and annotated with the recording start timestamp. Note that each device that is involved in this recording setup needs to be synchronized with the same time server.

verify_audio_files.py: A utility script for quickly assessing the dimensions of a given audio recording.

trim_from_to.py: A utility script for trimming a given audio recording.

trim_to_timestamp.py: A utility script for trimming a set of given audio recordings such that all recordings start in sync at the latest recording start timestamp among the given files. Inversely, it trims all recordings such that they end at the earliest recording end timestamp.

H. Data

The repository of this project provides a variety of experimental test data. Test data is either collected from the recording setup used in this project in real-world environments or sample generations from the *generate_audios()* utility function. Each directory encapsulates one test data collection, along with a detailed description of the experiment setup and its properties.

In *00_SOUND_BANK* a wide range of different noises (long audio signals for simulating background noise) and sounds (one-shot audio signals) samples are available to be used for the *generate_audios()* utility function. Note that this is just a small selection and that the user can easily add additional datasets.

I. Examples

We curated a selection of useful example scripts that showcase the recommended usage of the *pysoundlocalization* library. Most examples are specifically tailored toward one individual use case or a single implementation-specific topic.

It is recommended to start investigating the script *tutorial.py*, which includes all necessary steps to generate synthetic audio data for each microphone, as well as straightforward guidance through all three phases, *i.e.,* environment setup, preprocessing, localization. The examples can be used for reference and serve the purpose of demonstrating the library's capabilities.

IV. Experimental Evaluation

The library was evaluated in a simulated environment setup to verify the initially defined objectives: to localize single, multiple, and moving sound sources in both non-polyphonic and polyphonic noisy environments. The experiment was conducted to confirm whether the objectives were achieved. The relevant code to run the experiments and raw result are included in the library as part of the *experiments* folder.

A. Experiment Setup

Four experiments were performed to test the robustness of the library by measuring localization accuracy in different setups. The setups differed in a) the number of generated sound sources and b) whether the sound sources were stationary or moving. Both algorithms—Threshold and GCC-PHAT—were tested and compared. Figure 22 shows the experiment setup. The simulated environment is of size 600 x 600 meters with four microphones placed to form a 500 x 500 meter grid.

B. Experiment Specifics

As part of simulating randomly generated sound sources, the library's *simulate_noise_util.py* function was leveraged to construct the necessary audio signals. In addition, a constant background noise was added to every experiment, reflecting a factory machine sound, to make localization more

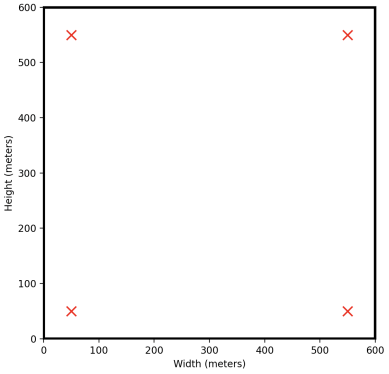


Fig. 22: The experiment setup (microphones as cross marks)

complex. For localization, the same preprocessing steps—to reduce background noises and isolate sounds—were included for each experiment. All experiments were repeated 100 times. The results of each experiment were stored in a text file—including approximated positions and localization error.

1) Experiment 1 (One stationary sound source):

- One audio signal of length 10s with a simulated sound of length 0.3s at timestamps 1s and 6s.
- The sound source coordinates are randomly generated within the detection area and the sound source remains stationary during the experiment.

2) Experiment 2 (One moving sound source):

- One audio signal of length 10 seconds with a simulated sound of length 0.3s at timestamps 1s and 6s.
- To localize the moving sound source, the audio signal was chunked into two chunks of length 5s each.
- Unlike experiment 1, the sound source moves to a second randomly generated coordinate for when the sound is emitted again.

3) Experiment 3 (Two stationary sound sources):

- Two audio signals of length 10s. Both audio signals include a distinct, simulated sound of length 0.3s at timestamps 1s and 6s.
- The sound source coordinates are randomly generated within the detection area, and both sound sources remain stationary during the experiment.

4) Experiment 4 (Two moving sound sources):

- Two audio signals of length 10s. Both audio signals include a distinct, simulated sound of length 0.3s at timestamps 1s and 6s.
- To localize the moving sound source, the audio signals were chunked into two chunks of length 5s each.
- Unlike experiment 3, one sound source remains stationary while the other moves to a second randomly generated coordinate for when the sound is emitted again.

C. Results

The localization accuracy of the Threshold and GCC-PHAT methods was measured across the four described experiments. Table III presents the error metrics for all experiments. The mean error, standard deviation (Std Dev), min and max error, and root mean square error (RMSE) were measured.

	Method	Mean Error	Std Dev	Min Error	Max Error	RMSE
Experiment 1	Threshold	0.089	0.053	0.002	0.194	0.104
	GCC-PHAT	0.009	0.004	0.001	0.019	0.010
Experiment 2	Threshold	0.089	0.055	0.001	0.209	0.105
	GCC-PHAT	0.009	0.004	0.0004	0.020	0.010
Experiment 3	Threshold	0.588	0.384	0.278	3.383	0.702
	GCC-PHAT	9.463	21.648	0.143	136.176	23.626
Experiment 4	Threshold	0.575	0.360	0.251	3.753	0.678
	GCC-PHAT	10.327	23.844	0.349	159.956	25.984

TABLE III: Localization accuracy across the four described experiments (in meters)

In experiments 1 and 2—where one stationary and moving sound source were localized respectively—low localization errors were measured; with mean errors being 0.089 meters for the Threshold algorithm and 0.009 meters for GCC-PHAT. The result is supported by low standard deviation (0.054 and 0.004 meters respectively) and low root mean square error (0.105 and 0.010 meters respectively) across both experiments.

In experiments 3 and 4—where two stationary and moving sound sources were localized respectively–localization accuracy remained relatively constant for the Threshold algorithm, with a mean error of 0.588 and 0.575 meters for experiments 3 and 4, respectively. The GCC-PHAT method performed significantly worse, with a mean error of 9.463 meters for experiment 3 and 10.327 meters for experiment 4. The result is reflected in the standard deviations of 21.648 and 23.844 meters. Although a low min error (0.143 and 0.349 meters) shows that good accuracy was achieved for some of the randomized setups, the max errors of 136.176 and 159.956 meters for experiments 3 and 4 also show high localization inaccuracy for some setups.

D. Discussion

The results show that both algorithms have good localization accuracy when one sound source is localized [41], [42]. When two sound sources are localized, the Threshold algorithm continues to show robust results. However, the localization accuracy of GCC-PHAT is negatively affected by the reconstruction of sound sources during the audio source separation. To localize multiple sound sources, audio source separation decomposes an audio signal into its most distinct components to reconstruct individual sound sources. While this process is effective for isolating sounds, it adversely impacts the performance of the GCC-PHAT algorithm, which relies on the integrity of the original signal structure for accurate localization. Consequently, the reconstruction process introduces significant localization errors when using GCC-PHAT [22], [33].

The results of the experiments show that the library can achieve the initial objectives to localize single, multiple, and moving sound sources in both non-polyphonic and

polyphonic, noisy environments. However, GCC-PHAT is not recommended for localization when sound sources are first separated. It further highlights that while the library's range of functionalities can be applied to many use cases, the implications of the underlying algorithms must be understood.

E. Limitations

The library currently has multiple limitations, three stemming from the current implementations of (existing) algorithms, one requiring precise audio recordings, and one being known but not accounted for yet.

1) Audio source separation: The lossy reconstruction property of the implemented audio source separation technique, particularly through NMF, leads to slightly wrong-separated output audio signals with individually lower resolution. While the localization with the Threshold algorithm remains unaffected, the GCC-PHAT yields significantly worse results due to its waveform-based cross-correlation approach.

2) Performance bottlenecks: For setups with only one sound source, the library performs very well. In the case of multiple sound sources, we discovered substantial increases in performance cost. The reason lies within the iterative optimization approach of the NMF algorithm, which is needed in the scenario of multiple sound sources. Furthermore, we found that the localization phase, namely the multilateration of the sound sources, does not significantly negatively affect the performance of the library.

3) Data recording: As established in the requirements III-B1, the recording data needs to be latency-free. The work on the examples in the library has revealed the abundance of potential places that may introduce latencies, such as hardware, software, or even network latencies. We have detected latencies in some of our test data sets that we use to demonstrate the rapid accuracy decline of the library in the GitHub repository; see *ex_errors_by_latencies.py*.

4) Moving sound sources: Section II-B2 describes how localization methods based on TDoA create issues in accuracy for moving sound sources. This is due to the frequency shift caused by the Doppler effect [32]. At least in theory, *pysoundlocalization* is not able to achieve perfect accuracy for constantly moving sound sources. However, the experiments showed good localization accuracy for moving sound sources that are stationary when emitting sound.

5) Audio chunking: An audio signal is split into many small chunks that are localized individually. For localization, the TDoA of sounds at the sensors is measured, implying that sound does not arrive at all sensors at the same time. In certain edge case experiment setups, particularly when chunks are very small and distances between sensors are very long, an audio signal may not reach all sensors within the same chunk. During consequent localization, this particular chunk may then not include the same audio signal for all sensors, which makes localization impossible. One approach to combat the issue—in the spirit of an explorative toolbox—is to play with the chunking size based on the specific localization use case at hand.

V. Final Considerations

A. Summary

Sound source localization has seen many proposed approaches to solve the different inherent problems at hand, with many solutions targeting individual problems, like audio signal filtering, noise reduction, TDoA computation, and multilateration. However, a gap was identified in the number of open-source implementations that provided a holistic toolbox of all relevant functionalities for successful and accurate sound source localization. With *pysoundlocalization*, we introduced a Python library for sound source localization of one or multiple stationary or moving sound sources in polyphonic noisy environments. The library was made open-source and is available at [38]. Our library offers an explorative toolbox of implementations of selected state-of-the-art algorithms for processing audio signals and localizing sound sources. The localization accuracy of the library was validated through four experiments, and results showed that both the Threshold algorithm and GCC-PHAT performed accurately with one sound source. While the Threshold algorithm continued to perform accurate localization for multiple sound sources, GCC-PHAT performed notably worse, being negatively affected by the sound source separation algorithm.

B. Conclusions

We introduced the Python library *pysoundlocalization* and achieved all three objectives to localize 1) a single, stationary sound source, 2) multiple, stationary sound sources, and 3) multiple moving sound sources. The experimental evaluation validated the robustness of the algorithms used in the library for accurate sound source localization. With this, the library fills the gap of practical open-source implementations as a general-purpose solution to many different application areas. The analysis of the discovered limitations revealed potential directions for future work and an understanding of the implications regarding the specific algorithm choices based on the given application areas. As we have learned from our extensive activities in audio data collection, the accuracy of distance measurements and latency-free recording data, as inputs for the library, are crucial components for a robust and reliable localization system. Particularly in research, the library may be interesting as it offers with the *generate_audios* module, an intuitive way of creating custom experiment data that can be used for testing new sound processing and localization techniques in scientific and practical projects.

C. Future Work

In the future, multiple enhancements can improve the *pysoundlocalization* library. First, as the implemented

sound source separation negatively affects localization accuracy when using GCC-PHAT, different sound source separation algorithms need to be evaluated to understand how GCC-PHAT can be applied to multiple sound sources. Second, the current sound source separation algorithm limits real-time localization as it is computationally intensive. Computationally less intense audio source separation algorithms can enable real-time localization. Third, the dimensionality of the localization capabilities can be extended from 2D to 3D space, which requires additional considerations regarding the recording and environment setup, the localization and multilateration techniques, and new types of visualization plots. Fourth, a challenge associated with a current limitation from the implemented chunking technique, see IV-E: Finding a solution for reliably localizing moving sound sources without distance-bound limitations would enable the use of *pysoundlocalization* in many more application areas. Lastly, by accounting for the Doppler effect when computing the TDoAs, localization of permanently moving sound sources becomes possible [32].

The library provides a first toolbox for sound source localization and through its high potential for extensibility, it can serve as a basis for future enhancements. It is our hope that the open source nature of the contribution encourages future additions to improve and extend its functionalities.

REFERENCES

[1] J. Lanslots, F. Deblauwe, and K. Janssens, "Selecting sound source localization techniques for industrial applications," *Sound and Vibration*, Vol. 44, pp. 6–10, 06 2010.

[2] P. Odya, J. Kotus, A. Kurowski, and B. Kostek, "Acoustic sensing analytics applied to speech in reverberation conditions," *Sensors*, Vol. 21, No. 18, p. 6320, 2021.

[3] J. Moragues, L. Vergara, J. Gosalbez, T. Machmer, A. Swerdlow, and K. Kroschel, "Background noise suppression for acoustic localization by means of an adaptive energy detection approach," *2008 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2008, pp. 2421–2424.

[4] A. Sedunov, A. Sutin, N. Sedunov, H. Salloum, A. Yakubovskiy, and D. Masters, "Passive acoustic system for tracking low-flying aircraft," *IET Radar, Sonar & Navigation*, Vol. 10, No. 9, pp. 1561–1568, 2016.

[5] X.-y. SUN, N.-s. LI, and L. Xiao, "Three-dimensional passive localization method for underwater target using regular triangular array," *2019 13th Symposium on Piezoelectrcity, Acoustic Waves and Device Applications (SPAWDA)*. IEEE, 2019, pp. 1–7.

[6] I.-C. Kim, Y.-J. Kim, and S.-Y. Chin, "Sound localization framework for construction site monitoring," *Applied Sciences*, Vol. 12, No. 21, p. 10783, 2022.

[7] W. Fiebig and D. Dąbrowski, "Use of acoustic camera for noise sources localization and noise reduction in the industrial plant," *Archives of Acoustics*, Vol. 45, No. 1, pp. 111–117, 2020.

[8] R. Scheibler, E. Bezzam, and I. Dokmanić, "Pyroomacoustics: A python package for audio room simulation and array processing algorithms," *2018 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, 2018, pp. 351–355.

[9] T. Sainburg, "timsainb/noisereduce: v1.0," Jun. 2019. [Online]: https://doi.org/10.5281/zenodo.3243139

[10] SciPy. (2025) Butterworth digital and analog filter design. Accessed: 2025-01-10. [Online]: https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.butter.html

[11] ——. (2025) Design second-order iir notch digital filter. Accessed: 2025-01-10. [Online]: https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.iirnotch.html

[12] MathWorks. (2025) Generalized cross-correlation. Accessed: 2025-01-10. [Online]: https://www.mathworks.com/help/phased/ref/gccphat.html

[13] M. Jurasovic. (2025) Multilateration in 2d: Iot/lorawan mass surveillance. Accessed: 2024-11-20. [Online]: https://github.com/jurasofish/multilateration

[14] D. Jones, C. Snider, A. Nassehi, J. Yon, and B. Hicks, "Characterising the digital twin: A systematic literature review," *CIRP journal of manufacturing science and technology*, Vol. 29, pp. 36–52, 2020.

[15] E. Widdison and D. G. Long, "A review of linear multilateration techniques and applications," *IEEE Access*, 2024.

[16] C. Knapp and G. Carter, "The generalized correlation method for estimation of time delay," *IEEE transactions on acoustics, speech, and signal processing*, Vol. 24, No. 4, pp. 320–327, 1976.

[17] S. Butterworth *et al.*, "On the theory of filter amplifiers," *Wireless Engineer*, Vol. 7, No. 6, pp. 536–541, 1930.

[18] S. F. Hussin, G. Birasamy, and Z. Hamid, "Design of butterworth bandpass filter," *Politeknik & Kolej Komuniti Journal of Engineering and Technology*, Vol. 1, No. 1, pp. 32–46, 2016.

[19] K. Hirano, S. Nishimura, and S. Mitra, "Design of digital notch filters," *IEEE Transactions on Communications*, Vol. 22, No. 7, pp. 964–970, 1974.

[20] C. J. Steinmetz and J. Reiss, "pyloudnorm: A simple yet flexible loudness meter in python," *Audio Engineering Society Convention 150*. Audio Engineering Society, 2021.

[21] B. Series, "Algorithms to measure audio programme loudness and true-peak audio level," *International Telecommunication Union Radiocommunication Assembly*, 2011.

[22] D. Lee and H. S. Seung, "Algorithms for non-negative matrix factorization," *Advances in neural information processing systems*, Vol. 13, 2000.

[23] D. D. Lee and H. S. Seung, "Learning the parts of objects by non-negative matrix factorization," *nature*, Vol. 401, No. 6755, pp. 788–791, 1999.

[24] P.-C. POPOVICI, "An overview of signal processing methods for signal source localization," *Journal of Military Technology Vol*, Vol. 3, No. 2, 2020.

[25] J. Chen, Y. Huang, and J. Benesty, "A comparative study on time delay estimation in reverberant and noisy environments," *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics, 2005.* IEEE, 2005, pp. 21–24.

[26] A. V. Oppenheim. (1987) Signals and systems - course - lecture 24-2. Accessed: 2025-01-17. [Online]: https://ocw.mit.edu/courses/res-6-007-signals-and-systems-spring-2011/6ffe3f6c387555a8db26f1f3bbaddfb5_MITRES_6_007S11_lec24.pdf

[27] T. Sainburg and A. Zorea, "Noisereduce: Domain general noise reduction for time series signals," 12 2024.

[28] V. Leplat, N. Gillis, and A. M. Ang, "Blind audio source separation with minimum-volume beta-divergence nmf," *IEEE Transactions on Signal Processing*, Vol. 68, pp. 3400–3410, 2020.

[29] B. Wang and M. D. Plumbley, "Investigating single-channel audio source separation methods based on non-negative matrix factorization," *Proc. ICA Research Network International Workshop*. Citeseer, 2006, pp. 17–20.

[30] ——, "Musical audio stream separation by non-negative matrix factorization," *Proc. DMRN summer conf*, 2005, pp. 23–24.

[31] Z. Benslimane, "Audio source separation using non-negative matrix factorization (nmf)," Feb. 2023. [Online]: https://shorturl.at/HoQX1

[32] F. Miao, D. Yang, R. Wang, J. Wen, Z. Wang, and X. Lian, "A moving sound source localization method based on tdoa," *43rd International Congress on Noise Control Engineering*, 2014.

[33] M. S. Hosseini, A. Rezaie, and Y. Zanjireh, "Time difference of arrival estimation of sound source using cross correlation and modified maximum likelihood weighting function," *Scientia Iranica*, Vol. 24, No. 6, pp. 3268–3279, 2017.

[34] J. Nikunen and T. Virtanen, "Direction of arrival based spatial covariance model for blind sound source separation," *IEEE/ACM transactions on audio, speech, and language processing*, Vol. 22, No. 3, pp. 727–739, 2014.

[35] G. C. Carter, A. H. Nuttall, and P. G. Cable, "The smoothed coherence transform," *Proceedings of the IEEE*, Vol. 61, No. 10, pp. 1497–1498, 1973.

[36] J. Velasco, M. J. Taghizadeh, A. Asaei, H. Bourlard, C. J. Martín-Arguedas, J. Macias-Guarasa, and D. Pizarro, "Novel gcc-phat model in diffuse sound field for microphone array pairwise distance based calibration," *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2015,

pp. 2669–2673. [Online]: https://publications.idiap.ch/downloads/papers/2015/Velasco_ICASSP15_2015.pdf
[37] Librosa. (2023) Librosa - api documentation. Accessed: 2025-01-16. [Online]: https://librosa.org/doc/main/generated/librosa.resample.html
[38] L. J. Haller T., "Imp - pysoundlocalization, v1.0," Jan. 2025. [Online]: https://github.com/jonaslanzlinger/IMP
[39] S. learn. (2025) Blind source separation using fastica. Accessed: 2025-01-15. [Online]: https://scikit-learn.org/stable/auto_examples/decomposition/plot_ica_blind_source_separation.html
[40] (2025) Gstreamer. Accessed: 2025-01-15. [Online]: https://gstreamer.freedesktop.org/
[41] T. Letowski and S. Letowski, "Localization error: Accuracy and precision of auditory localization," *Advances in sound localization*, Vol. 55, pp. 55–78, 2011.
[42] R. Lee, M.-S. Kang, B.-H. Kim, K.-H. Park, S. Q. Lee, and H.-M. Park, "Sound source localization based on gcc-phat with diffuseness mask in noisy and reverberant environments," *IEEE Access*, Vol. 8, pp. 7373–7382, 2020.

APPENDIX

A. Statement of Contribution

The authors confirm that they have both equally contributed to this project.

B. Contents of the Library

The library is available as an open-source project at: https://github.com/jonaslanzlinger/IMP.

C. Aids

Aid	**Usage**	**Affected Part**
ChatGPT	Text reduction, Grammar correction	Complete Paper
Zotero	Compilation of the literature index	Literature Index