

Project: real-time sound localisation

Signal Processing

Nov-Dec 2022

Introduction

Real-time signal processing consists in building systems capable of performing a specific task in the given time. In many fields, these types of systems are very useful as they use *feedback*, meaning that they adapt themselves to control the output.

The goal of this project is to give you an intuition of the importance of online¹ signal processing systems. You are going to implement a system that processes audio signals to estimate the position of a sound. At the end of this project you should be able to deploy your work on a Raspberry Pi equipped with an array of microphones to locate a sound position in real-time.

Time difference of arrival

Many techniques exist to understand where a sound is coming from. Our ears for example are able to give us an idea of the position thanks in part to the timing difference between their pressure perception² [1]. On the same principle, systems such as sonars or seismic surveillance systems use a technique called TIME DIFFERENCE OF ARRIVAL (TDOA) to find the localisation of the sound. They measure the time difference between signals from multiple synchronised microphones and compute the position of their source.

Dataset

Online applications are always evaluated twice: once offline to measure the capability of the application to achieve acceptable performance (e.g. accuracy, precision); then online, at runtime, to check that it operates in real-time. For the first part, you will need data created for the sake of the application you are implementing. The dataset **LocateClaps** provided to you was recorded in the NUMEDIART room from ISIA LAB and contains the recording from three microphones of clap sounds located at 12 different positions.

¹i.e. real-time, in opposition to offline.

²when in low frequency range, below 10 kHz

The setup for the recording is as follows: three microphones simultaneously record sharp sounds. The position of each sound is referred to by its angle with the x-axis while maintaining a distance of roughly 3 meters from the centre of the microphone array. The information about the position of a source and the microphone used is embedded in the name of the corresponding file (e.g: *M1_30.wav* is the file related to microphone 1 and source is at 30°).

Procedure

In what follows, you will divide your project into several functions put in a single python file (referred to as module) and name this module `UTILS.PY`. Create a second module named `MAIN.PY` in which you will import your functions and use them, build your project and do your analysis.

You are asked to test each of the functions you build to verify that the output goes along with your predictions when injecting a specific input. During the last laboratory session, you will be able to push your code on a Raspberry Pi and evaluate the online performance of your system.

Your report will contain the required analysis and interpretations (which includes answering the questions asked explicitly as well as your own implementation choices and the performance of your system). Every result you obtain needs to be analysed and interpreted even when not explicitly asked to do so.

Useful Python libraries

Since Python has a lot of useful libraries, some of them have functions that you might want to use in order to implement this project. You can find the inputs and outputs as well as examples for each function in the documentation available online. Here is a list of some libraries you might find use:

- `SCIPY` (`.SIGNAL` and `.OPTIMIZE`): is a well-known library for many signal processing or mathematical operations respectively.
- `AUDIOFILE`: contains the function **read** that allows one to read a wav file in Python.
- `GLOB`: contains the function **glob** that allows one to save multiple folder and/or file paths in the same list.
- `COLLECTIONS`: manages different types of vectors/lists.
- `TIME`: contains the functions **time** or **time_ns** that allows one to record the time.

- **PYAUDIO**: is mainly used to access audio-related hardware features (microphones or speakers) through the **PyAudio class**³.

1 Offline system

This part of the project concerns the implementation of each function separately, their cascading to build the system and the evaluation of its offline performance. It includes data management, pre-processing and position computation based on the cross-correlation. Figure 1 summaries the steps of the project in blocks.

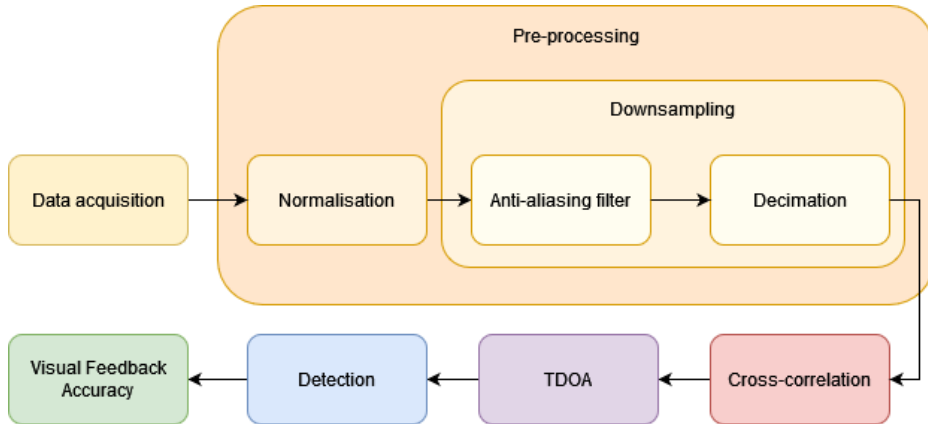


Figure 1: Block diagram of the project.

1.1 Data generation and dataset

In order to evaluate the system, you have to use signals that will give you predictable output. On the first hand, you will generate a known signal that will be used to demonstrate the efficiency of most of your functions. On the other hand you will read the signals from the dataset **LocateClaps** to evaluate the accuracy of your system to locate real sound.

1. **SINE WAVE SIGNAL: Create sinusoidal signal** built with 8000 SAMPLES, with a peak-to-peak amplitude of 8 and a frequency of 20 Hz. The sampling frequency is 44.1 kHz.
2. **LOCATECLAPS DATA: Complete the `read_wavefile` function so that it outputs data from a given file path.** One option is to use `read` from `SCIPY.SIGNAL.WAVEFILE` library.

Display the content of at least one .wav file in a figure, as well as your sine wave in an other one.

³You won't use this library yourself during this project.

1.2 Buffering

To manage the continuous flow of data (stream) recorded by a microphone, we store it in a buffer. A buffer is similar to a waiting room for data to be processed. When you don't need to keep already processed data, a ring buffer can be used. A ring buffer is a specific type of buffer of limited size where we update the oldest sample slots with new samples. When a certain amount of slots have been updated, we process the buffer again (cf. Figure 2).

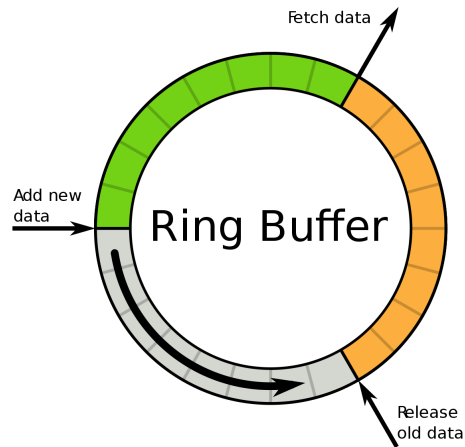
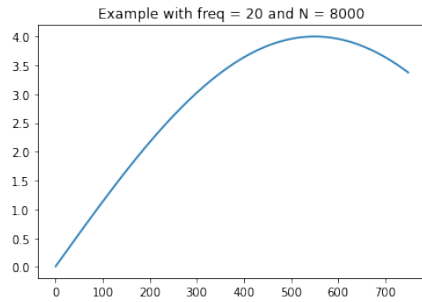


Figure 2: Schematic of a ring buffer [from [RingBuffer](#)]. The green section represents the new data, the orange section the data to be processed and the grey section the data that have been processed.

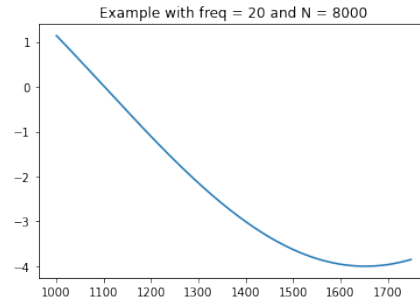
To create a ring buffer in Python, one can use the function `deque`⁴ that creates a deque object that behaves like a ring buffer. This means that it saves data until all its slots have been filled. Then when new data arrives, it shifts its content to remove the oldest sample at the end of the queue and saves the new one at the beginning.

1. Complete `create_ringbuffer` so that it makes use of deque to create a ring buffer of maximum length `maxlen` samples.
2. To verify the efficiency of your implementation, first create a ring buffer of `maxlen 750` and use the `extend` function to save your sine wave signal one sample at the time (i.e: in a *for* loop). Then with `MATPLOTLIB`, display the buffer content once it reaches its maximal size and then once the 1000th sample has been saved (cf. Figure 3).

⁴from Python library `COLLECTIONS`



(a) Buffer content from 0 to 750 samples



(b) Buffer content from 1000 to 1750 samples

Figure 3: Verification example for a sine wave signal of $A_{pp} = 8$, $f = 20\text{Hz}$ and $N = 8000$ at $f_s = 44.1\text{ kHz}$.

1.3 Pre-processing

Most of the time data from sensors may require pre-processing to fit a format that increases the efficiency of the application. In this section, you are asked to modify some aspects of the signals like the amplitude or the amount of samples.

1.3.1 Normalisation

The signal amplitude scales between its maximum and minimum values that depends on the sensor characteristics and the signal intensity. To normalise the amplitude between $[-1, 1]$, **implement the function `normalise` so that it outputs the normalised form of the input signal**. Apply it on your sine wave signal and show both before and after normalisation in the same figure.

1.3.2 Downsampling

Sensors like microphones have their own sampling frequency specifications that may not suit your requirements (e.g. data provided in **LocateClaps** are sampled at 44.1 kHz). By examining the spectrogram of a signal, you can see which part of the frequency spectrum is occupied. If the useful frequencies are low enough to respect Shannon's theorem, one can downsample the signal to process less data (meaning less computation) without important loss of information. As stated in Chapter 3 of Signal Processing, downsampling is performed in two steps: the anti-aliasing filter first eliminates frequency components higher than half the new sampling frequency, so as to avoid aliasing during the subsequent decimation step. This second step only keep one sample out of M (M being the decimation factor) (Cf. Figure 4).

1. SIGNAL ANALYSIS: First **analyse the useful frequencies of the `LocateClaps` data using `spectrogram`**⁵ Based on what you observed,

⁵from MATPLOTLIB.PYPILOT

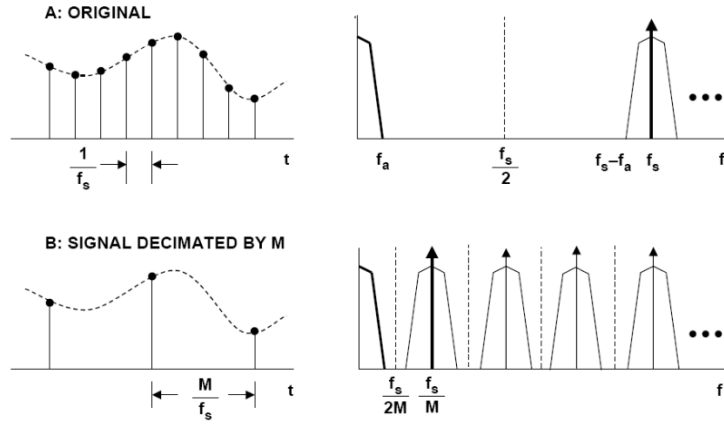


Figure 4: Decimation by a factor $M = 4$. [From Signal Processing, Chapter 3, Fig. 3.18, p. 13]

choose the most appropriate sampling frequency f_s between the following: 8k, 16k, 24k or 44.1k Hz and justify;

2. **ANTI-ALIASING FILTER:** Then investigate Chebychev and Cauer filters to eliminate all frequency components above $f_s/2$ (f_s being the final sampling frequency). The filter should filter out the frequencies after $f_s/2$ Hz. The filter you keep should have minimal delay and minimal deformation on the signal. **Test both filters on a signal that contains a sine wave with amplitude 1000 V and frequency 8500 Hz and an other sine wave with amplitude 20 and frequency 7500 Hz.**
3. **DECIMATION:** As stated, decimation is the action of keeping one sample out of M . In python, you can perform decimation easily with the following syntax:

```
decimated_vector = your_vector [::M]
```

Verify the decimation with a signal that contains a sine wave with amplitude 1000 V and frequency 8500 Hz and a sine wave with amplitude 20 V and frequency 7500 Hz with a decimation factor $M = 3$. In addition to that, show in the frequency domain the difference when we use the anti-aliasing filter has on a decimated signal and when we don't.

1.4 Cross-correlation

Cross-correlation is a method to measure the similarity of two signals. Figure 5 shows that cross-correlation is a variation of the convolution. In 2D signals, it computes the common surface between two signals when we slide the first one on the second (signal red on signal blue). While convolution

flips the second signal, cross-correlation keeps it as is. Due to that, take care that convolution has the commutativity property but cross-correlation depends on the order of the signals.

Equations 1 and 2 shows the Fourier Transform formulas of convolution and cross-correlation respectively. The advantage of using the Fourier transform is that the Fourier Transform of a convolution becomes a simple multiplication of the Fourier Transform of each signal⁶.

$$\text{Convolution: } \{g * h\} = \mathcal{F}^{-1}\{\mathcal{F}\{g\} \cdot \mathcal{F}\{h\}\} \quad (1)$$

$$\begin{aligned} \text{Cross-correlation: } \{g(t) \star h(t)\}(t) &= g(t) * h(-t) \\ &= \mathcal{F}^{-1}\{\mathcal{F}\{g\} \cdot \overline{\mathcal{F}\{h\}}\} \end{aligned} \quad (2)$$

where \mathcal{F} is the Fourier Transform and \bar{h} the complex conjugate of h .

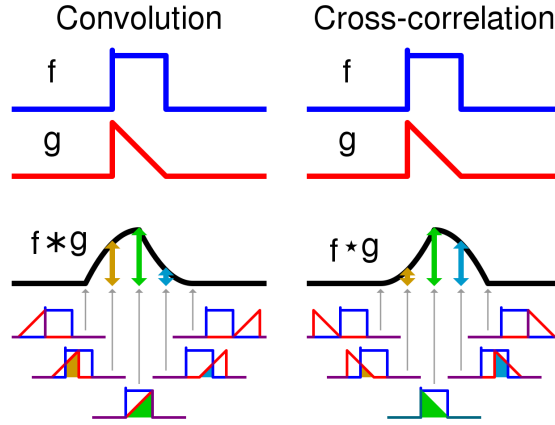


Figure 5: Comparison between convolution (left) and cross-correlation (right). The difference between both is that the signal g is not reversed in cross-correlation. [Modified from Wikipedia]

Based on Equation 2, implement the function `fftxcorr` in such a way that it outputs the correlation of the two signals. Use the library `numpy.fft` (or `np.fft`) to obtain the Fourier Transform of your signals. Verify your implementation, compare your output with that of the function `FFTCONVOLVE`⁷ when computing the auto-correlation⁸ of your sine wave signal. Remember that you need to counter the flip that convolution does on the second signal.

⁶cf. Signal Processing, chapter 4

⁷from Python library `SCIPY.SIGNAL`.

⁸i.e. the cross-correlation of the same signal

1.5 Localisation

In this section, you will implement two functions for your localisation system: a TDOA function based on cross-correlation and a solver for the localisation equations.

1.5.1 TDOA

As stated in the introduction, TDOA is based on the time shift between the same signal recorded by different sensors. One way to measure this time shift is to use the cross-correlation seen previously and get the index of the highest amplitude sample. **Implement the function `TDOA` that takes as input the correlation vector of two signals and outputs the time shift value.**

1.5.2 Localisation equation system

The time shift would allow us to understand which sensor gets the signal first. We can combine two time shifts to triangulate the source of the signal in a 2 dimensional space. Figure 6 shows the project configuration: three microphones (in blue) are installed at the corners of a triangle, whose center is the origin (in red). A sound is produced at a random location src (in green). The distance between microphone i and the sound location is given by the sound velocity in the air ($v \approx 343$ m/s) multiplied by the time taken to arrive (t_i). This distance is also the hypotenuse of a right-angled triangle as depicted in the figure. While we know the position of each microphone, the distance between two (mics i and j) can also be characterised by their time difference to detect the sound ($v \times (t_i - t_j) = v \times \tau_{i,j}$).

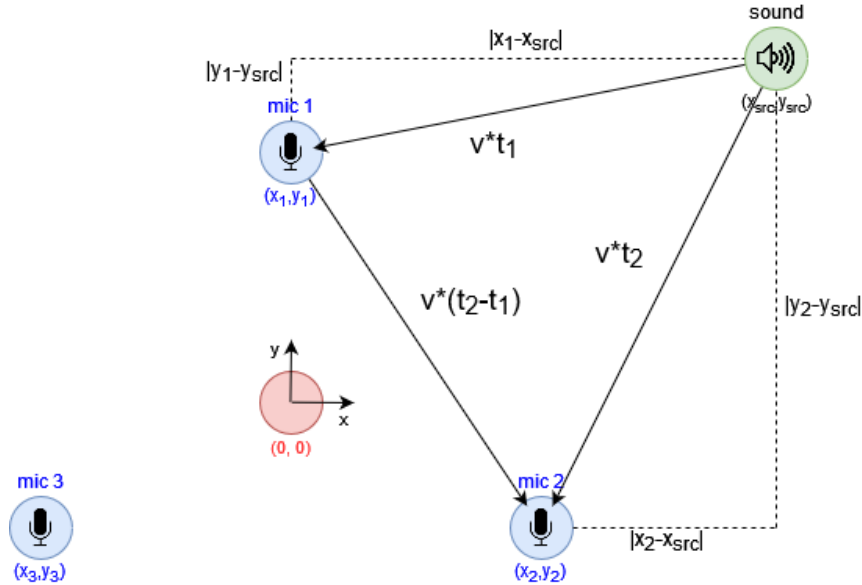


Figure 6: Schematic of the microphone positions.

Based on this geometry, the triangulation can be written as:

$$\begin{aligned} v \times \tau_{2,1} &= \sqrt{(x_2 - x_{src})^2 + (y_2 - y_{src})^2} + \sqrt{(x_1 - x_{src})^2 + (y_1 - y_{src})^2} \\ v \times \tau_{3,1} &= \sqrt{(x_3 - x_{src})^2 + (y_3 - y_{src})^2} + \sqrt{(x_1 - x_{src})^2 + (y_1 - y_{src})^2} \end{aligned} \quad (3)$$

with $\tau_{i,j}$ the TDOA time shift between sensors i and j , x_k , y_k the x and y coordinates of sensor k and v the sound velocity in the air.

Given Figure 6 and microphone coordinates $(0, 0.0487)$, $(0.0425, -0.025)$ and $(-0.0425, -0.025)$ for microphones #1, #2 and #3 respectively, you should be able to triangulate the source of a signal. In this project, we will mainly focus on predicting the angle between the x -axis and the source position, based on the predicted coordinates. We provide you with the function `localise_sound` that outputs the coordinates of the source when given two TDOAs. **Using those coordinates, implement the function `source_angle` so that it outputs the angle for given coordinates.**

1.6 Systems accuracy and speed

Online systems rely on the small computational cost of their internal operations to give the fastest and most accurate outputs. During the implementation of your system, you were asked to verify the outputs of your functions. This section aims to measure the accuracy and the speed of your system.

The way we will measure the accuracy is by comparing the difference between the expected value (the **ground truth**, which refers to a value that is known to be true) and the predicted value to a threshold.

For the speed evaluation, Python is not known as the most performing language in term of processing speed. We will base our measure on the time library, which provides the function `time_ns`. This function returns the time when it was called in ns, which might not be precise enough (and returns 0 ns) if the function is well optimised.

1. Implement the function `accuracy` in such a way that it compares the difference between the predicted angle and the ground truth angle with a threshold to detect wrong predictions⁹. The function should output *True* if the difference between both angle is below the threshold; it should return *False* otherwise.
2. Use `accuracy` on all 12 angles available in the `LocateClaps` dataset and report the results (for each angle and give also the average accuracy).

⁹REMINDER: the ground truth angle is written at the end of the file name, before the extension: `M1_30.wav` stands for 30°.

3. The function `time_delay` prints out the processing speed of a given function. It is a wrapper function, which means that you provide to it a function and its inputs and the wrapper outputs will be the same as those of the function you provided. For example, when we call the following code:

```
def my_function(a, b):
    return a*b

def wrapper(func, args):
    print("Wrapper here")
    out = func(*args)
    return out

print(my_function(10, 2))
print(wrapper(my_function, [10, 2]))
```

the result is as follows:

```
20

Wrapper Here
20
```

Use the wrapper `time_delay` on the following functions (with one file from `LocateClaps` as your default input signal):

- `normalise`,
- `downsampling`,
- `fftxcorr` (and compare with `fftconvolve`),
- `TDOA`,
- `localise_sound`,
- `source_angle`

Report the speed when the signal is downsampled and when it is not (do not evaluate downsampling function in this case). Explain the difference you observe and which situation is the fastest.

2 Real-time localisation

To perform the detection in real-time, specific hardware is required that embeds multiple microphones. In addition to that we also need to manage data as a continuous stream. This section describes the different processes implemented to obtain a real-time application.

2.1 Hardware equipment

The hardware on which your implementation will be uploaded consists of a Raspberry Pi 3 model B and a Respeaker 6-Mic Circular Array (cf. Figure 7 and Figure 8):

Raspberry Pi 3B The Raspberry is a single-board computer the size of a credit card. In addition to the ports you can find on most computers (HDMI, USB, 3.5 mm jack), it also includes 40 GPIO pins that allows one to control hardware modules (such as LEDs). The Raspberry runs a specific Linux operating system called Raspberry Pi OS and contains a Python IDE.

Respeaker Mic Array The 6-Mic circular array designed by Seeed Studio to extend the Raspberry Pi. It contains 6 microphones designed for audio applications as well as 12 uniformly separated LEDs in a circular shape. These features can be controlled with Python 3 since they are recognised as standard audio input. Its recording specifications are:

- $\max f_s = 48kHz$,
- Omnidirectional sensitivity,
- $SNR = 59dB$

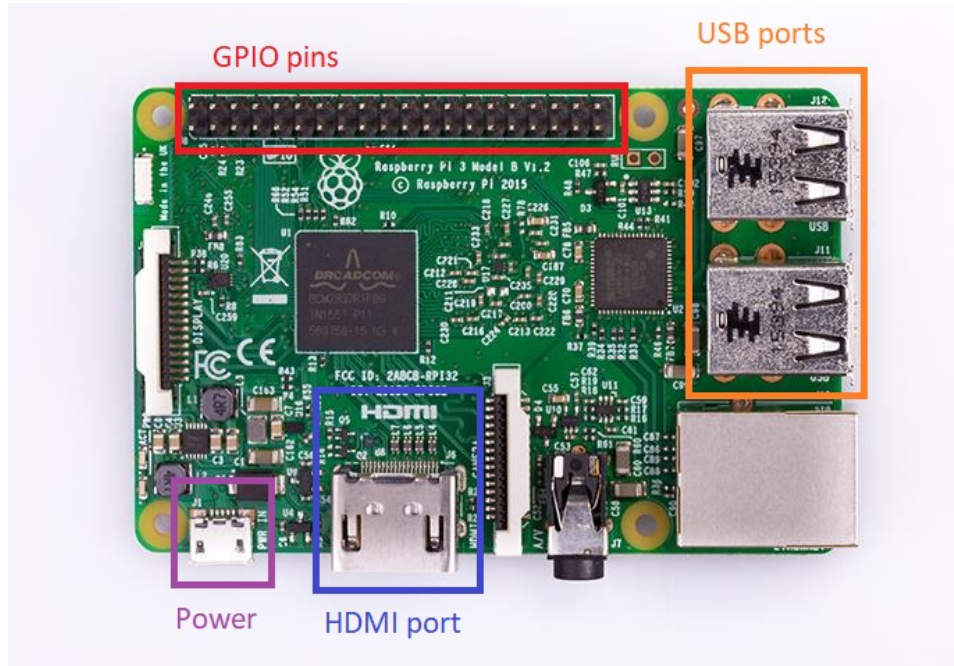


Figure 7: Raspberry PI 3B, with highlighted zones [modified from [Mouser](#)].

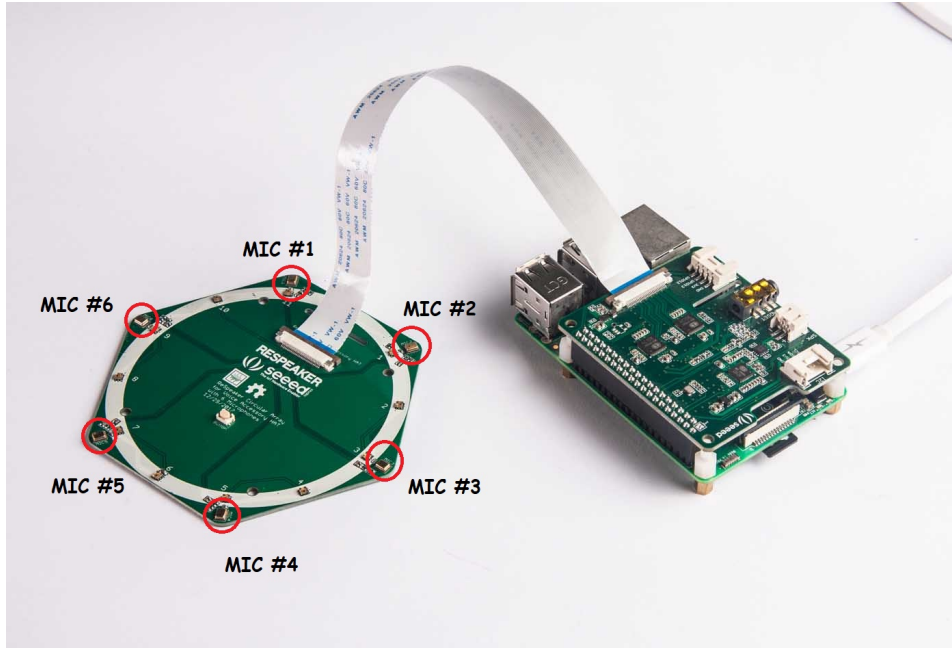


Figure 8: Respeaker 6-Mic Circular Array Kit [from [Seeed Studio](#)].

In this section, you are asked to search for an practical application with Raspberry Pi. Explain briefly how the chosen application works and its general use in your report.

2.2 Data acquisition and processing

Callback The hardware used for the project sends in a stream the concatenation of data from multiple microphones each sampled at 44.1 kHz. In this project, three microphones only are required while six are available on the audio card (cf. Figure 8, which means that we have to save only half of the stream. As we intend to keep the same triangular shape formed by the mics (cf. Figure 6), we will select data from microphones #1, #3 and #5.

To save data without stopping the process, we implemented a [callback](#) function. A callback function can be seen as a function called every time the hardware is able to record a new set of samples. The callback we wrote saves data in separate buffers, each related to one microphone.

Stream management To collect data from the audio card, we have to create a stream object linked to this hardware. Functions [init_stream](#) and [close_stream](#) manages the creation and the closure respectively of a stream for you. Processing signals in real-time means to use the stream data while it is active. In this project, the function [detection](#) calls the functions you implemented above to detect an angle in real-time.

Visual feedback In order to verify your system, we used functions that control the LED lights of the card to provide a visual feedback on the predicted angle. Since there is 12 LEDs, we adapted the intensity of each LED to provide more information on how close the predicted angle is compared to a LED position: if a color is bright on a particular LED, the source is in the same direction; if two or three adjacent LEDs are light up, the source is in a direction between those.

For this section, **you are asked to report a subjective evaluation on two aspects of the online implementation:**

1. the accuracy of the direction provided by the visual feedback;
2. the real-time impression of the system (mention how the delay between the sound creation and its detection by the system)

Compare the results with different buffer sizes (1s, 2s and 3s) or sampling frequencies (16 kHz or 24 kHz).

References

- [1] C. Köppl, “Evolution of sound localisation in land vertebrates,” *Current biology*, vol. 19, no. 15, pp. R635–R639, 2009.