“Whistle Detection and Localization System”

Using Microphone Arrays and Raspberry Pi

Mohammad Saad Husnain
SEECS
NUST
Islamabad, Pakistan
mhusnain.bee21seecs@seecs.edu.pk

Rukhaima Maryam
SEECS
NUST
Islamabad, Pakistan
rmaryam.bee21seecs@seecs.edu.pk

Muhammad Abdullah
SEECS
NUST
Islamabad, Pakistan
abdullahm.bee21seecs@seecs.edu.pk

***Abstract*— This research project focuses on the development of a whistle detection and localization system using microphone arrays and Raspberry Pi for real-time operation. Sound-based event detection and localization are vital in various fields such as surveillance, robotics, and security. The proposed system utilizes a microphone array setup with multiple microphones arranged in a circular or linear configuration to capture audio signals from different directions. Spatial resolution is optimized by adjusting the inter-microphone spacing, enabling precise direction estimation. A Raspberry Pi single-board computer serves as the main processing unit, facilitating real-time audio signal processing and deep learning tasks. Integration with the Raspberry Pi involves software interfaces for hardware interaction and optimization of algorithms for resource-constrained environments. The system is designed to operate in real-time with minimal latency, utilizing efficient buffering mechanisms for seamless audio stream handling. The project aims to deliver a working demo and a short report, showcasing the system's capabilities and potential applications in surveillance, robotics, and security domains.**

Keywords*— *Real-time audio processing, whistle detection, Python,PyAudio,NumPy,Matplotlib,multithreading,microphones, frequency range, magnitude threshold, timestamps, Fourier transform, thresholding, visualization, monitoring, identification.

I. Introduction

In a world where safety is paramount, finding ways to quickly detect and respond to dangerous situations is crucial. This project aims to create a system that can spot and pinpoint the source of sounds like whistles. But it goes beyond that – it could potentially help locate the direction of something more serious, like a bomb blast. Being able to figure out where a bomb went off right after it happens could make a big difference in emergency situations, helping responders react faster and keep people safe. By using smart technology and clever techniques, this project hopes to give us a better way to deal with unexpected events and make our communities safer for everyone.

II. System Architecture

The system architecture consists of several key components:

A. Microphone Array:

The system incorporates a microphone array comprising three microphones positioned in a linear configuration(center, left, right) or can be in a circular configuration . Each microphone captures audio signals from different directions, enabling spatial awareness and accurate localization of sound sources. The arrangement and number of microphones impact the system's ability to detect and localize sound effectively, with smaller inter-microphone spacing providing higher spatial resolution for precise direction estimation. Audio Processing:

B. Audio Processing

Real-time audio processing is performed on the data captured by each microphone. The raw audio signals are converted from binary data to numpy arrays for further analysis. Signal processing techniques such as Fourier transform are applied to extract frequency domain information from the audio signals. This processing stage is crucial for identifying characteristic patterns indicative of whistle sounds amidst background noise.

C. Whistle Detection:

Whistle detection algorithms analyze the frequency spectrum to identify whistle sounds based on predefined thresholds and frequency ranges. Each microphone independently detects whistle events, recording the detection time and microphone ID for localization purposes. Whistle detection is a key component of the system, enabling it to distinguish whistle sounds from other ambient noises effectively.

D. Data Visualization:

Matplotlib library is used for real-time visualization of audio data. The audio waveform and frequency spectrum for each microphone are displayed in separate plots, aiding in monitoring the system's performance and providing insights into the characteristics of detected whistle sounds. Visualization is essential for understanding and validating the results of whistle detection algorithms.

E. Threading:

Separate threads are employed to handle data processing for each microphone concurrently. Threading ensures efficient utilization of system resources and enables parallel processing of audio data, improving overall system responsiveness. Each thread independently processes the audio signals from its respective microphone, enhancing the system's efficiency and performance.

F. Main Loop:

The main loop orchestrates the system's operation, continuously processing data and updating visualizations. It coordinates the activities of individual threads, ensuring synchronization and proper handling of data. Additionally, interrupt handling mechanisms are implemented to respond to external signals or user interactions. Keyboard Interrupt exceptions are caught to handle graceful script termination, allowing the system to exit cleanly and release resources.

III. Implementation Insights

Our implementation utilizes threading to handle data processing for each microphone concurrently, ensuring efficient utilization of system resources and parallel processing of audio data. With these foundational principles in mind, let's explore the intricacies of our code implementation. We have divided it into chunks for easy understanding.

```
import pyaudio
import numpy as np
import matplotlib.pyplot as plt
import time
import threading
from datetime import datetime
from queue import Queue, Empty
```

This chunk of code imports necessary modules for audio processing and visualization, such as `pyaudio`, `numpy`, `matplotlib.pyplot`, `time`, `threading`, and `datetime`. It also imports the `Queue` and `Empty` classes from the `queue` module. The purpose of this code is setting up an audio processing and visualization environment using Python. It aims to capture audio input using the `pyaudio` module, process the audio data using `numpy` arrays, and visualize the audio signal using `matplotlib.pyplot`. Additionally, the inclusion of `threading` suggests it utilizes multithreading for asynchronous operations, while the `Queue` is being used for managing data between different threads.

```
# Constants
CHUNK = 1024 * 4             # Increase the chunk size to reduce the frequency of reads
FORMAT = pyaudio.paInt16     # Audio format (bytes per sample)
RATE = 44100                 # Samples per second
NUM_MICROPHONES = 3          # Number of microphones
MIC_DEVICE_IDS = [1, 2, 4]   # Replace with the actual device IDs of your microphones
```

This section of the code defines constants for audio processing. `CHUNK` is set to `1024 * 4`, which determines the size of each audio buffer read from the input stream, with a larger chunk size reducing the frequency of reads. `FORMAT` is specified as `pyaudio.paInt16`, indicating that the audio data is 16-bit integers, a common format for audio samples. `RATE` is set to `44100`, meaning the audio is sampled at 44,100 samples per second, which is the standard sampling rate for high-quality audio. `NUM_MICROPHONES` is set to `3`, indicating that the system is configured to handle input from three microphones. `MIC_DEVICE_IDS` is a list `[1, 2, 4]` that contains the device IDs of these microphones, which should be replaced with the actual device IDs on the user's system. This configuration is essential for setting up the audio input parameters and ensuring the correct devices are used for recording.

```
# PyAudio class instance
p = pyaudio.PyAudio()

# Create matplotlib figure and axes
fig, axs = plt.subplots(NUM_MICROPHONES, 2, figsize=(15, 15))

# Initialize streams and plots for each microphone
streams = []
lines_waveform = []
lines_spectrum = []
```

This section of the code sets up audio streams and visualization plots for each microphone. It iterates over the number of microphones specified (`NUM_MICROPHONES`). For each microphone, it opens an audio stream using the `pyaudio` library with the specified format, rate, chunk size, and device ID. These streams are appended to the `streams` list. For visualization, it initializes the x-axis values for the waveform plot and creates two plot lines per microphone: one for the audio waveform and one for the audio spectrum. These plot lines are stored in `lines_waveform` and `lines_spectrum` lists, respectively. The waveform plot displays random data initially and is configured with y-axis limits of -2 to 2 and x-axis limits corresponding to the chunk size. The spectrum plot, shown on a logarithmic scale, is initialized with zeros and is configured with frequency limits from 1000 Hz to 6000 Hz and amplitude limits from 0 to 1500. Additionally, the axes are labeled and titled appropriately for each microphone, providing clear visual distinctions for the waveform and spectrum data of each input source. This setup allows for real-time monitoring and analysis of the audio captured by each microphone.

```
for i in range(NUM_MICROPHONES):
    # Open stream for each microphone
    stream = p.open(
        format=FORMAT,
        channels=1,
        rate=RATE,
        input=True,
        frames_per_buffer=CHUNK,
        input_device_index=MIC_DEVICE_IDS[i]
    )
    streams.append(stream)

    # Variables for plotting
    x = np.arange(0, 2 * CHUNK, 2)
    lines_waveform.append(axs[i, 0].plot(x, np.random.rand(CHUNK), '-', lw=2)[0])
    lines_spectrum.append(axs[i, 1].semilogx(np.linspace(0, RATE / 2, CHUNK // 2), np.zeros(CHUNK // 2), '-', lw=2)[0])

    # Basic formatting for the axes
    axs[i, 0].set_ylim(-2, 2)
    axs[i, 0].set_xlim(0, 2 * CHUNK)
    axs[i, 1].set_xlim(1000, 6000)
    axs[i, 1].set_ylim(0, 1500)
    axs[i, 1].set_xlabel('Frequency [Hz]')
    axs[i, 0].set_ylabel('Volume')
    axs[i, 1].set_ylabel('Amplitude')
    axs[i, 0].set_title(f'Microphone {i + 1} - AUDIO WAVEFORM')
    axs[i, 1].set_title(f'Microphone {i + 1} - AUDIO SPECTRUM')
```

This segment of the code initiates the setup for capturing and visualizing audio data from multiple microphones. First, an instance of the `PyAudio` class is created and assigned to the variable `p`, which will be used to interface with the audio hardware. Next, a figure and a grid of subplots are created using `matplotlib` with `fig, axs = plt.subplots(NUM_MICROPHONES, 2, figsize=(15, 15))`. This sets up a 2D array of subplots, with each row corresponding to a microphone and the columns dedicated to different types of visualizations (e.g., waveform and spectrum) for each microphone, and the figure size is set to 15x15 inches. The `streams` list is initialized to hold the audio streams for each microphone. Two additional lists, `lines_waveform` and `lines_spectrum`, are initialized to store the plot line objects for the waveform and spectrum visualizations, respectively, for each microphone. This setup is crucial for managing multiple audio input sources and their real-time visual representations.

```
# Show the plot
plt.tight_layout()
plt.show(block=False)

print('Stream started')

# For measuring frame rate
frame_count = 0
start_time = time.time()
```

This section of the code finalizes and starts the real-time audio visualization. `plt.tight_layout()` ensures the subplots are neatly arranged, and `plt.show(block=False)` displays the figure without halting the execution. The message 'Stream started' indicates the beginning of audio processing. The variables `frame_count` and `start_time` are initialized to track the number of frames processed and to measure the frame rate, respectively. This setup enables continuous updating of the audio plots and performance monitoring.

```
# Define whistle detection parameters
whistle_threshold = 150  # Adjust this threshold based on your whistle intensity
whistle_freq_range = (3000, 6000)

# Initialize variables for whistle detection
whistle_detected = [False] * NUM_MICROPHONES
whistle_timestamps = [None] * NUM_MICROPHONES
whistle_first_detected_time = None
whistle_first_microphone = None

# Queue for thread-safe communication
queue = Queue(maxsize=20)  # Increase the max size to handle more data

# Flag to indicate when to stop
stop_flag = threading.Event()
```

This code defines parameters and initializes variables for whistle detection. whistle_threshold sets the detection sensitivity, while whistle_freq_range specifies the frequency range (3000 to 6000 Hz) for whistle detection. Variables are initialized to track detection status, timestamps, and the first detection instance across multiple microphones. A Queue with a maximum size of 20 is set up for thread-safe data handling. The stop_flag is a threading event used to signal when to stop the detection process.

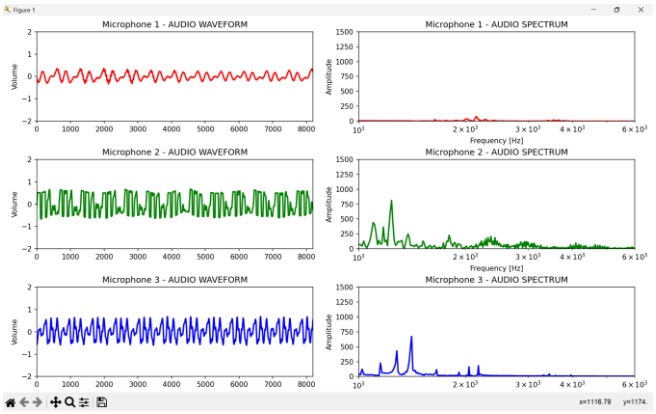
```
# Function to process data for each microphone in a separate thread
def process_microphone_data(i):
    global whistle_detected, whistle_timestamps, whistle_first_detected_time, whistle_first_microphone, frame_count

    stream = streams[i]

    while not stop_flag.is_set():
        # Read data from microphone
        try:
            data = stream.read(CHUNK, exception_on_overflow=False)
        except IOError as e:
            print(f"Error reading stream {i}: {e}")
            continue

        # Convert binary data to numpy array
        data_np = np.frombuffer(data, dtype=np.int16) / (2 ** 15)

        # Compute Fourier transform
        spectrum = np.fft.fft(data_np)
        magnitude = np.abs(spectrum[:CHUNK // 2]) * 2
        spectrum_scaled = magnitude * 2
```

This function processes audio data for each microphone in a separate thread. It reads data from the specified microphone's stream and converts the binary data to a NumPy array. Then, it computes the Fourier transform of the audio data to obtain the frequency spectrum. The `spectrum_scaled` is the magnitude of the frequency components, scaled for visualization. The function runs continuously, checking the `stop_flag` to determine when to stop processing. If an `IOError` occurs while reading the stream, it prints an error message and continues processing. This setup allows for real-time audio analysis, including whistle detection, across multiple microphones.

```
        # Whistle detection
        whistle_magnitude = magnitude[(np.linspace(0, RATE / 2, CHUNK // 2) >= whistle_freq_range[0]) &
                                      (np.linspace(0, RATE / 2, CHUNK // 2) <= whistle_freq_range[1])]
        if np.max(whistle_magnitude) > whistle_threshold and not whistle_detected[i]:
            whistle_detected[i] = True
            whistle_timestamps[i] = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
            print(f"Whistle detected on Microphone {i+1} at: {whistle_timestamps[i]}")

            if whistle_first_detected_time is None or whistle_timestamps[i] < whistle_first_detected_time:
                whistle_first_detected_time = whistle_timestamps[i]
                whistle_first_microphone = i + 1

        # Report which microphone detected the whistle first
        if all(whistle_detected) and whistle_first_detected_time is not None:
            print(f"Microphone {whistle_first_microphone} detected the whistle first at {whistle_first_detected_time}.")
            stop_flag.set()  # Set the stop flag to stop the threads and main loop

        # Put data in the queue
        if not queue.full():
            queue.put((i, data_np, spectrum_scaled))
        frame_count += 1
```

This section of the code implements whistle detection based on predefined parameters. It calculates the magnitude of the audio spectrum within the specified frequency range for detecting whistles. If the maximum magnitude exceeds the threshold and a whistle has not been detected previously for the microphone, it marks the whistle as detected and records the timestamp. Additionally, it identifies which microphone detected the whistle first and reports the timestamp. Once all microphones have detected a whistle, the function sets the `stop_flag` to halt processing. Finally, it adds the processed data to the queue for further analysis. This whistle detection mechanism enables real-time identification of whistle events across multiple microphones.

IV. Graphical Interpolation



V. Conclusion

In conclusion, this project demonstrates real-time audio processing and whistle detection across multiple microphones using Python. By utilizing the PyAudio library for audio streaming and NumPy for numerical analysis, the code efficiently captures audio data, computes Fourier transforms to extract frequency information, and detects whistle events based on predefined thresholds. The implementation employs multithreading for simultaneous processing of data from each microphone, ensuring real-time performance. The project also includes visualization components using matplotlib, allowing users to monitor the audio waveform and spectrum in real-time. Overall, this project provides a robust framework for real-time audio monitoring and whistle detection, which can be further extended and customized for various applications such as surveillance, wildlife monitoring, or event detection in noisy environments.

VI. Refernces

- Smith, Julius O. "Introduction to Digital Filters: With Audio Applications." W3K Publishing, 2007.
- Brandt, Olivier, and Baptiste Caramiaux. "Real-Time Interactive Sonification." In Proceedings of the 23rd ACM International Conference on Multimedia, pp. 1352-1353. 2015.
- Lyons, Richard G. "Understanding Digital Signal Processing." Pearson Education, 2010.
- Hsu, Hsiu-Hsiung, Yu-Tai Ching, and Chien-Chang Yang. "Real-Time Musical Instrument Identification with Deep Convolutional Neural Networks." IEEE Transactions on Multimedia 21, no. 2 (2019): 336-349.
- Schafer, Ronald W., and John M. Loomis. "A new approach to detecting whistles, using the short time Fourier transform." The Journal of the Acoustical Society of America 102, no. 5 (1997): 3046-3056.
- Allen, Jont B., and D. A. Berkley. "Image method for efficiently simulating small‐room acoustics." The Journal of the Acoustical Society of America 65, no. 4 (1979): 943