Lebanese American University

LAU
School of Engineering

MCE540: Biomechatronics

Section 31

Instructor: Dr. Noel J. Maalouf

OLEVIBRA

Empowering Deaf Awareness:

OLED Display Vibrawearable Bracelet

Razan Hmede - 202103291

Naim David Dankar – 202102165

Mario El Chahni - 202102192

Table of Contents

Table of Figures

Introduction

In an increasingly interconnected world, the importance of accessibility and inclusivity cannot be ignored. Among the various communities benefiting from technological advancements, the deaf community often faces unique challenges in navigating environments heavily reliant on auditory cues.

In response to these challenges, a groundbreaking innovation has emerged: the development of a bracelet designed to enhance accessibility and awareness for the deaf. This innovative device incorporates a color-coded representation system to detect several types of sounds, coupled with vibration features that reflect the intensity of the surrounding auditory environment. As the intensity of sound increases, so does the vibrational feedback from the bracelet, providing vital sensory information to its wearer.

The development of this bracelet represents a significant step forward in the realm of assistive technology, offering a comprehensive approach to enhancing accessibility and awareness for the deaf community. Its integration of visual and tactile feedback mechanisms not only addresses the limitations of traditional auditory-centric solutions but also embodies a commitment to inclusivity and innovation in the pursuit of a more accessible world.

This introduction sets the stage for a deeper exploration into the design, functionality, and potential impact of this remarkable advancement in assistive technology.

Objectives

1. **Utilization of a Color-Coded System for Sound Detection:** Develop a color-coded representation system within the bracelet to visually differentiate between several types of sounds present in the user's environment.

2. **Incorporation of Vibration Feedback Mechanism:** Embed a vibration feature into the bracelet design to provide haptic feedback, serving as an alternative sensory alert mechanism, particularly beneficial for users with hearing impairments. The intensity of vibrations will correspond to the level of surrounding sound, enhancing user awareness and safety.

3. **Integration of Sound Direction Detection:** Develop a sound direction detection feature to enhance spatial awareness and aid users in detecting and identifying nearby sounds. This functionality will provide users with valuable information about the origin and direction of sounds in their environment, further enhancing accessibility and safety.

Through the implementation of these functionalities, the project aims to significantly enhance user awareness, safety, and overall accessibility in diverse environments, with a particular focus on improving the experience for users with hearing disabilities.

General Description

1. **DIY Wristband Creation:** Develop a wristband through (DIY): The band was sewed specifically for this project and made adjustable for different patient to be able to wear it , featuring a built-in microphone array engineered to detect surrounding sounds with a specialized resolution covering 360 degrees.

2. **Audio Processing System:** Design and implement a robust audio processing system where captured audio from the microphone array is forwarded to a preprocessor for refinement and optimization.

3. **Deep Learning Prediction Models:** Train two deep learning models to analyze the preprocessed audio data and make accurate predictions regarding the detected sounds.

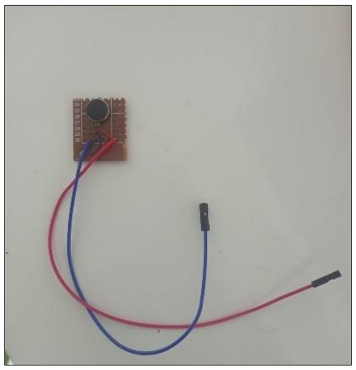
4. **Visual Coding Output:** Develop a visual coding system that generates intuitive visual cues based on the predictions made by the deep learning models**.** These visual cues are specifically tailored to assist hard-of-hearing individuals in understanding and interpreting the surrounding auditory environment.

5. **Vibration Feedback Mechanism:** Implement a vibration feedback mechanism within the wristband to complement the visual coding system. Vibrations are triggered based on the predictions made by the deep learning models, providing additional sensory alerts for hard-of-hearing users.

Needed Hardware

1. Customized wrist band
2. OLED display
3. Microphones k038 for the array and the directionality feature
4. USB microphone for the deep learning and mapping
5. DC coin Vibration Motor Arduino
6. NPN transistor
7. Wires
8. Resistors
9. Power source
10. LEDs
11. Raspberry pi

Bracelet Features



Vibration Feature

The sound intensity captured by the microphone is translated into vibration intensity using Python code. This mapping ensures that the vibration of the DC coin vibrator motor corresponds directly to the intensity of the captured sound. The DC coin motor is affixed to a small PCB board and securely positioned within the wristband, ensuring direct contact with the user's skin.

The circuit used for this small DC coin motor consists of 1 NPN transistor where the base is connected through a resistor to raspberry pi pin 18, both the emitter and the collector are connected to ground.

The VCC of the DC coin motor is connected to 3.3V pin of the raspberry pi and the GND pin is connected as a common ground to the collector of NPN transistor.

The code used to test this DC coin motor is below:

```
import RPi.GPIO as GPIO
import time

# Set the GPIO mode
GPIO.setmode(GPIO.BCM)

# Define the GPIO pin connected to the motor
motor_pin = 18
```

```python
# Set up the GPIO pin as an output
GPIO.setup(motor_pin, GPIO.OUT)

try:
    # Run the motor for 2 seconds
    GPIO.output(motor_pin, GPIO.HIGH)
    print("Motor is ON")
    time.sleep(2)

    # Turn off the motor
    GPIO.output(motor_pin, GPIO.LOW)
    print("Motor is OFF")

except KeyboardInterrupt:
    # Clean up GPIO on CTRL+C exit
    GPIO.cleanup()

finally:
    # Clean up GPIO on normal exit
    GPIO.cleanup()
```

The code used to map the intensity of the surrounding sound to the intensity of vibration of the small DC coin motor:

```python
import pyaudio
import numpy as np
import RPi.GPIO as GPIO

def calculate_rms(audio_data):
    # Convert raw audio data to numpy array
    audio_np = np.frombuffer(audio_data, dtype=np.int16)
    # Ensure non-negative values before squaring
    audio_np = np.abs(audio_np)
    # Calculate RMS amplitude
    rms = np.sqrt(np.mean(np.square(audio_np)))
    return rms

def record_audio(stream, chunk=512):
    # Record audio data from the microphone
    audio_data = stream.read(chunk)
    return audio_data
```

```
def map_intensity(audio_intensity):
    min_audio_intensity = 0
    max_audio_intensity = 255  # Maximum intensity for 8-bit audio
    min_motor_intensity = 0
    max_motor_intensity = 100  # Maximum PWM duty cycle

    return min_motor_intensity + (audio_intensity - min_audio_intensity) *
(max_motor_intensity - min_motor_intensity) / (max_audio_intensity -
min_audio_intensity)

if __name__ == "__main__":
    FORMAT = pyaudio.paInt16
    CHANNELS = 1
    RATE = 44100
    CHUNK = 512


    GPIO.setmode(GPIO.BCM)
    motor_pin = 18
    GPIO.setup(motor_pin, GPIO.OUT)


    audio = pyaudio.PyAudio()
    DEVICE_INDEX = 1

    stream = audio.open(format=FORMAT,
                        channels=CHANNELS,
                        rate=RATE,
                        input=True,
                        frames_per_buffer=CHUNK,
                        input_device_index=DEVICE_INDEX)


    print("Recording...")

    try:

        motor_pwm = GPIO.PWM(motor_pin, 100)
        motor_pwm.start(0)

        while True:
            # Record audio
            audio_data = record_audio(stream, CHUNK)
            rms = calculate_rms(audio_data)

            if not np.isnan(rms):

                motor_intensity = map_intensity(rms)

                motor_pwm.ChangeDutyCycle(motor_intensity)
                print(f"RMS Amplitude: {rms:.2f}, Motor Intensity:
{motor_intensity:.2f}")
            else:
                print("Invalid RMS Amplitude")
    except KeyboardInterrupt:
        GPIO.cleanup()

        motor_pwm.stop()
        GPIO.cleanup()
        stream.stop_stream()
        stream.close()
        audio.terminate()
```

As we can see in this code the mapping is done by getting the distance for the min audio intensity audio intensity – min audio intensity, then multiplying it by the range of motor intensity to map and dividing it by the range of audio intensity to normalize it and then finally adding a bias of min motor intensity so that we don't obtain a negative motor intensity and the mapping starts from 0.

Since after testing the microphone on the terminal of the raspberry pi using arecord I obtained an 8 bit unsigned audio. We used the max audio intensity to be 255 = 2^8.

The RMS amplitude of the audio is calculated using calculate_rms method we defined earlier in order to print the amplitude of the sound.

OLED Display: Sound Direction Feature

An i2c OLED display with a resolution of 128x64 is employed to highlight the output of speech recognition, presenting the recognized words in an easily readable format.

First, we started by getting the specific address of this OLED display which happened to be 0x3C.

Then, a testing code was wrote to test the OLED:

```
import time
```

```
from luma.core.interface.serial import i2c
from luma.oled.device import sh1106
from luma.core.render import canvas
def get_device():
    # Define your I2C interface and device here
    serial = i2c(port=1, address=0x3C)  # Adjust the address based on your OLED
display
    return sh1106(serial)

def main():
    device = get_device()

    with canvas(device) as draw:
        draw.text((10, 10), "WELCOME TO OLEVIBRA!", fill="white")

    time.sleep(10)

if __name__ == "__main__":
    main()
```

Color-Coded System

The captured audio from the microphone array undergoes prediction using the extensive "Urbansound8K dataset," renowned for its ability to distinguish over eight thousand distinct urban sounds, ranging from car honking and dog barking to fire alarms and beyond. Upon making a prediction, the system activates an LED of a corresponding color, uniquely assigned to each sound category.

A comprehensive catalogue accompanying the bracelet provides users with the color code, associating each sound with a specific LED color for easy identification and understanding. This seamless integration of predictive analysis and visual feedback enhances the user experience by providing clear and intuitive cues for several types of urban sounds, contributing to improved awareness and accessibility in urban environments.

Circular Array Configuration

Furthermore, a research initiative was undertaken to evaluate various microphone array alignments, aiming to identify the most effective design for optimal performance. The results are highlighted in the table below:

	LINEAR MICROPHONE ARRAY	**CIRCULAR MICROPHONE ARRAY**
BEAM PATTERN	Narrow beam with low-magnitude side lobes Presence of ambiguity lobe	1 single main lobe +several side lobes No ambiguity lobes
BEAM FORMING PERFORMANCE AT LOW FREQUENCIES	Bad	Bad
SPECTRAL ATTENUATION	Attenuation levels of 20-30 dB consistently along the bandwidth	Good attenuation levels up to 20dB but not consistent throughout the whole bandwidth
SPACE	Takes more space	Does not require much space
SPATIAL COVERAGE	Covers only one dimension	Covers 360 degrees
DIRECTIONALITY	Only one axis	Multi-axis
COMPLEXITY	Simple	More complex

After thorough research, and as a result of the performed comparison, we concluded that the circular array configuration best suited our needs. Consequently, we proceeded to solder eight microphones onto a compact PCB board.

Next as we wanted to determine the number of microphones in question to be soldered, we took into consideration 3 major factors in the design:

The spacing between the microphones: we designed the array such that the microphones are not very close to each other so that we avoid as much as possible the reflections.

The number of microphones is best taken as an even number to obtain an integer when dividing it from 360 to obtain the angle.

Therefore, the number of microphones could not be more than 8 as we used 10 the spacing between the microphones will be much smaller and we decided to not go for 6 microphones since it does not cover as much as if we used 8, the special coverage is much narrower.

Final number of microphone decision after trial and error: 8

- Each microphone is strategically positioned at an angle of 45 degrees apart from one another around the circle, allowing for comprehensive sound detection.

- The radius of the circular arrangement measures 3.5 centimeters, ensuring optimal coverage and sensitivity across various environments.

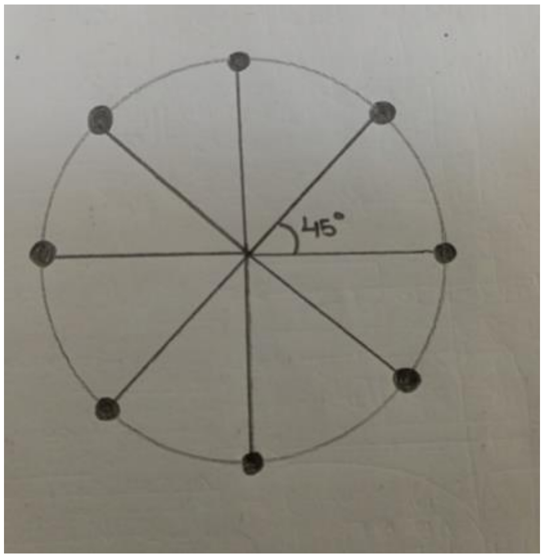


Figure 1: Preliminary design of the array on paper

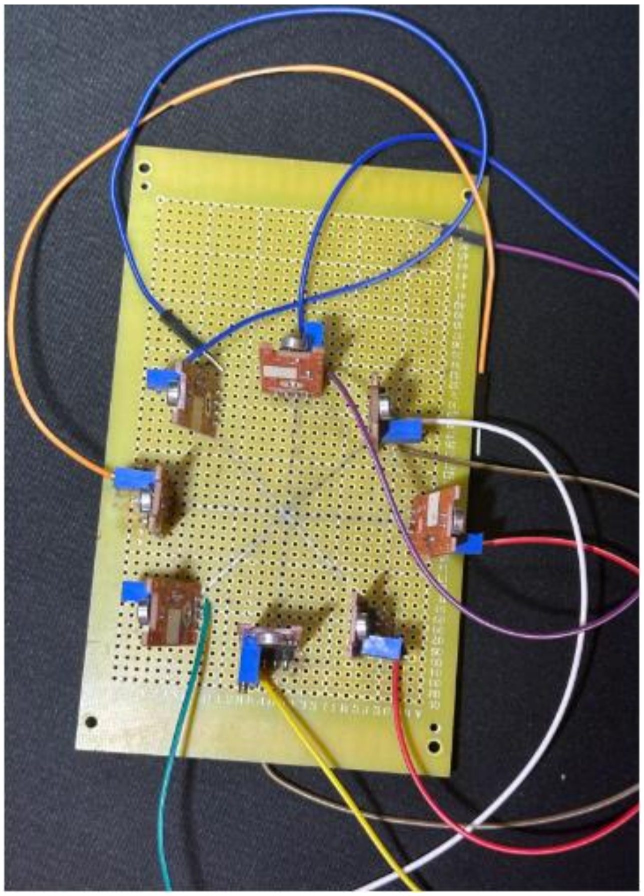
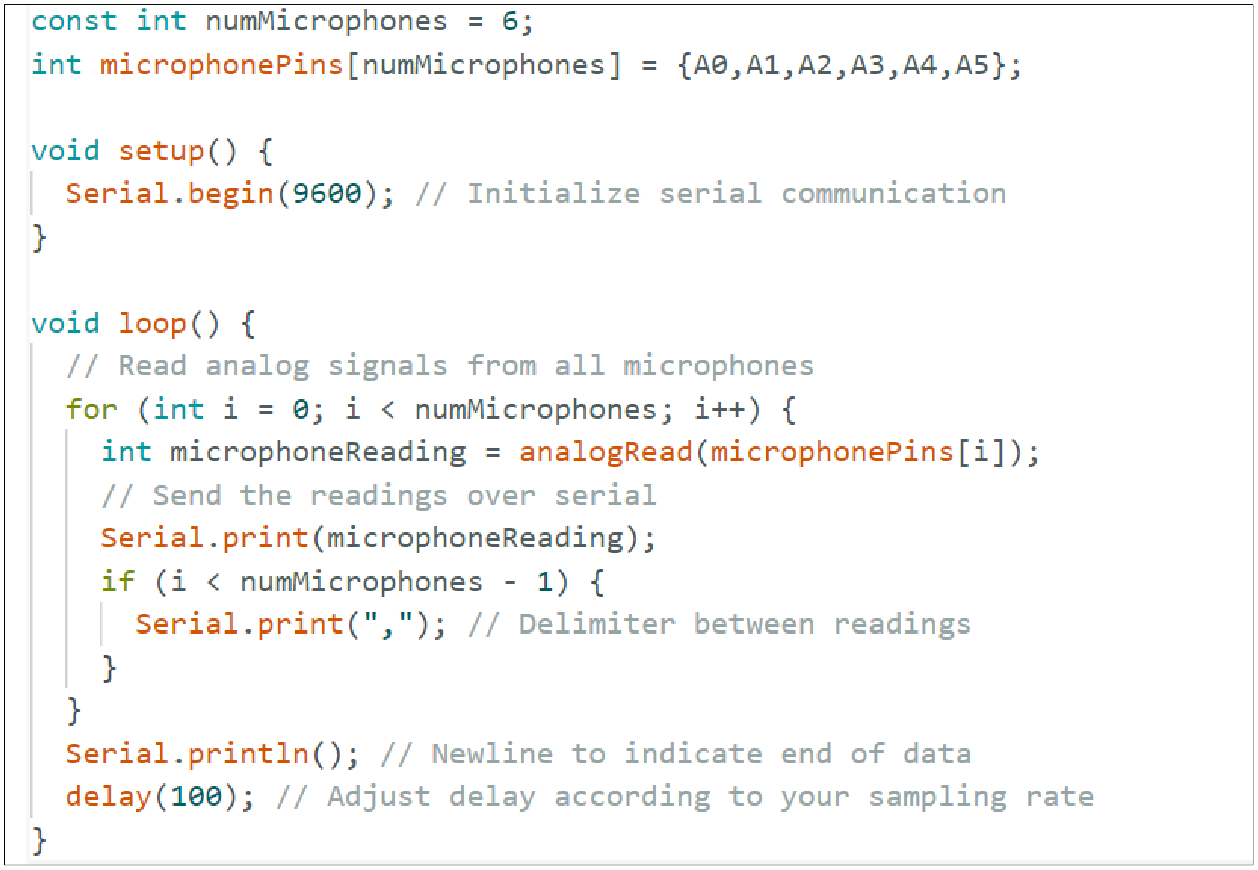


Figure 2: Obtained array after soldering

After we designed this circular array and we tried the code for the microphone sensor used, we discovered that it only detects if there is sound, the intensity as well as it can be used for directionality purposes but it cannot give useful info about the audio signal to be preprocessed after and used in the deep learning models we used.

Therefore, the circular array is used along with an Arduino microcontroller to predict the direction of the sound coming from the surroundings.

Sound Direction Feature



```
const int numMicrophones = 6;
int microphonePins[numMicrophones] = {A0,A1,A2,A3,A4,A5};

void setup() {
  Serial.begin(9600); // Initialize serial communication
}

void loop() {
  // Read analog signals from all microphones
  for (int i = 0; i < numMicrophones; i++) {
    int microphoneReading = analogRead(microphonePins[i]);
    // Send the readings over serial
    Serial.print(microphoneReading);
    if (i < numMicrophones - 1) {
      Serial.print(","); // Delimiter between readings
    }
  }
  Serial.println(); // Newline to indicate end of data
  delay(100); // Adjust delay according to your sampling rate
}
```

This Arduino code is designed to read analog signals from multiple microphones and send these readings over serial communication to be processed further using a MATLAB code we designed. Here's a breakdown of the Arduino code provided above:

- **Initialization**: sets up serial communication at a baud rate of 9600 bits per second.
- **Main Loop:** Continuously reads analog signals from each microphone pin specified in the microphonePins array.
- **Reading Microphones:** iterates through each microphone pin, reads the analog signal from it using analogRead(), and sends the reading over serial.
- **Serial Communication:** Each reading is sent over serial communication with a comma (,) separating readings. After all readings are sent, a newline character is added to indicate the end of the data.

- **Delay:** It includes a delay of 100 milliseconds between each loop iteration to control the sampling rate. Adjust this delay according to the desired sampling rate.

The following is the MATLAB code used to process the received signals and estimate the direction of sound:

```
% Initialize serial port communication
s = serial('COM4', 'BaudRate', 9600); % Adjust COM port as necessary
fopen(s);

% Parameters
numMicrophones = 6;
samplingRate = 1000; % Adjust according to Arduino code delay
frameSize = 6; % Number of samples to read per frame

while true
    % Read data from Arduino
    data = fscanf(s, '%d', [numMicrophones, frameSize]);

    beamformedSignal = sum(data, 1);

    [maxValue, maxIndex] = max(beamformedSignal);
    estimatedDirection = (maxIndex - 1) * (360 / frameSize); % Convert index to degrees

    % Display estimated direction
    fprintf('Estimated direction of sound source: %.2f degrees\n', estimatedDirection);
end

% Clean up
fclose(s);
delete(s);
clear s;
```

This code continuously listens to sound signals from the implemented microphones, combines them to focus on the direction of the sound source, and predicts where the sound is coming from.

Here's a breakdown of the provided MATLAB code:

- **Initialization:** sets up communication with a serial port connected to an Arduino, which is collecting data from the microphones.
- **Parameters:** defines parameters like the number of microphones, sampling rate, and frame size.
- **Main Loop:** continuously reads data from the Arduino, where each data read represents samples from all microphones for a short time period (a frame). Then it sums up the samples from all microphones to create a single "beamformed" signal.
- **Direction Estimation:** finds the index of the maximum value in the beamformed signal, which indicates the direction of the sound source. This index is converted to degrees.
- **Display:** prints out the estimated direction of the sound source in degrees.
- **Cleanup:** After the loop ends (which might not happen as it's set to run indefinitely), it closes the serial connection and cleans up resources.

For the rest of the project, we used a USB mini microphone for Raspberry pi:

The USB microphone is inserted in one of the ports of the raspberry pi board.

ALSA library is then downloaded as long as with Pyaudio to be able to identify the device. Using the alsa configuration we were able to edit the files to make this device the default device as well getting the card number specific for it and the device index to be used later on in the codes.

The microphone testing code we used:

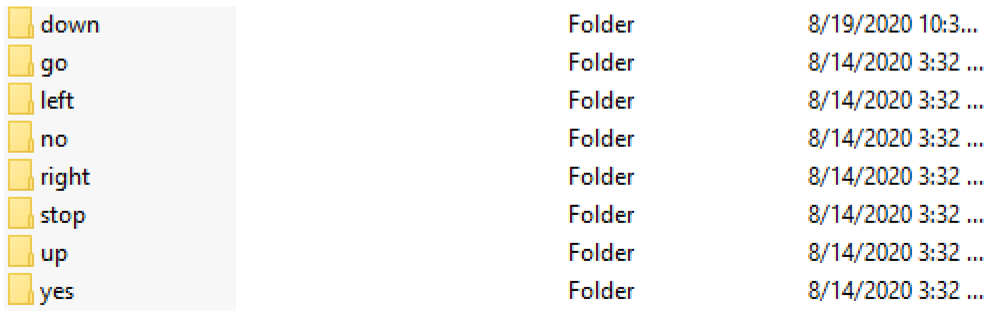
```
import pyaudio

pa = pyaudio.PyAudio()

# Get the index of the default input device
default_device_index = pa.get_default_input_device_info()['index']

# Get the number of available audio devices
num_devices = pa.get_device_count()

print("Available audio devices:")
for i in range(num_devices):
    device_info = pa.get_device_info_by_index(i)
    print(f"Device {i}: {device_info['name']}")
    print(f"  Max Input Channels: {device_info['maxInputChannels']}")
    print(f"  Default Sample Rate: {device_info['defaultSampleRate']}")
    print(f"  Host API: {device_info['hostApi']}")  # Optional, depending on your needs
    print("")
```

```
pa.terminate()
```

Datasets Used

Speech Recognition

In Tensor Flow Lite, the dataset comprises audio clips categorized into eight distinct folders, each representing a specific speech command: no, yes, down, go, left, up, right, and stop.
The mini speech command is downloaded and used later on to train the model.



down	Folder	8/19/2020 10:3...
go	Folder	8/14/2020 3:32 ...
left	Folder	8/14/2020 3:32 ...
no	Folder	8/14/2020 3:32 ...
right	Folder	8/14/2020 3:32 ...
stop	Folder	8/14/2020 3:32 ...
up	Folder	8/14/2020 3:32 ...
yes	Folder	8/14/2020 3:32 ...

Urban Sound Classification

We used a labeled dataset comprising a diverse collection of 8,732 sound clips, each lasting 4 seconds, spread across ten different classes:

- Air conditioner noise
- Car horn signals
- Sounds of children playing
- Barking of dogs
- Drilling noises
- Engine idling sounds
- Gunshot sounds
- Jackhammer noises
- Siren alerts
- Street music ambiance

This dataset was downloaded and saved as csv file to be later on used in the deep learning model.

Training the models

Google Colab was used for the first deep learning model.

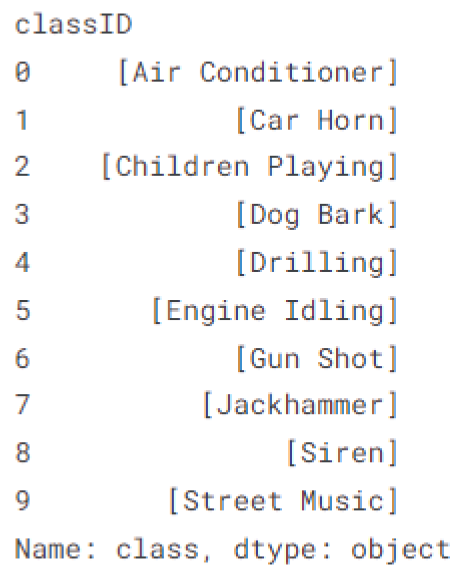
First, we upload the mini speech command dataset zip file into the COLAB and we run the model.

After training we obtained an accuracy of 83% approximately, which is a good accuracy so we proceed. The input for the training is audio signal saved as waveforms for each command.

The model was then saved in the colab as a .h5 file, downloaded and added to the workspace of our project.

This model is based on CNN, therefore in the model we convert the audio waveforms into spectrograms to be able to predict based on these.

Kaggle was used for the second model, with an accuracy

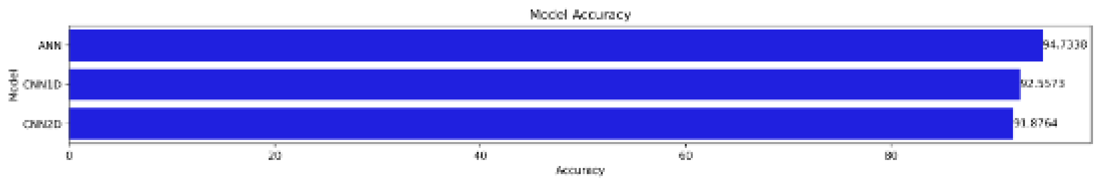


```
classID
0     [Air Conditioner]
1            [Car Horn]
2    [Children Playing]
3            [Dog Bark]
4            [Drilling]
5       [Engine Idling]
6            [Gun Shot]
7          [Jackhammer]
8               [Siren]
9        [Street Music]
Name: class, dtype: object
```

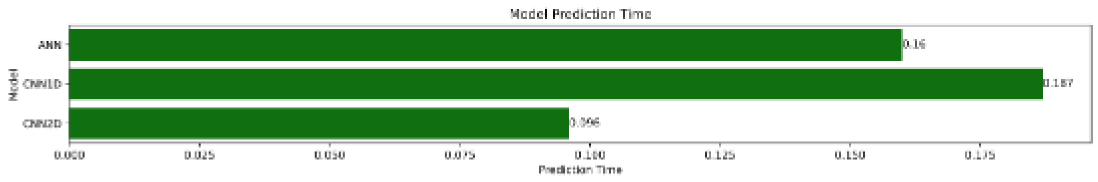
The model utilizes MFCC in order to predict the type of sound using librosa library.

The model in kaggle compares between 3 types of models: ANN/CNN1D/CNN2D

And based on the metrics results after training we used the first model which is the ANN:



The accuracy of the ANN model is 94.333 which is the highest.



Also for the model prediction time the first model took less time to predict the type of sound.

The model is downloaded in .h5 file and used in the main code in Vscode.

Note: the 2 models are included in the python folder that contains all the python codes.

Raspberry pi and remote connection on VScode

In order to establish connection between raspberry pi and the local machine in this case my laptop.

We started by downloading the raspberry pi OS 64bit on the SD Card along with the Wi-Fi configuration as a text file and an empty ssh file.

Next, we got the IP address of the raspberry pi and established a remote connection using t hip address on VScode.

To further view the pi system we used REAL VNC VIEWER to install the libraries using the terminal of the pi.

Codes for first deep learning model: Speech Recognition

This project consists of 3 helper codes and 1 main code:

Recording the audio python file to record the audio using the USB microphone:

```
import pyaudio
import scipy.signal
import numpy as np

def upsample_audio(audio, target_length=16000):
    original_length = len(audio)
    # Upsample the audio using linear interpolation
    upsampled_audio = scipy.signal.resample(audio, target_length)
    return upsampled_audio

def record_audio(stream, chunk=512):
    # Record audio data from the microphone
    audio_data = stream.read(chunk)
    return audio_data

if __name__ == "__main__":
    # Set parameters for audio recording
    FORMAT = pyaudio.paInt16
    CHANNELS = 1
    RATE = 16000
    CHUNK = 512

    # Initialize PyAudio
    audio = pyaudio.PyAudio()
    DEVICE_INDEX = 1

    # Open audio stream
    stream = audio.open(format=FORMAT,
                        channels=CHANNELS,
                        rate=RATE,
                        input=True,
                        frames_per_buffer=CHUNK,
                        input_device_index=DEVICE_INDEX)

    print("Recording...")

    frames = []
    seconds = 1
    for i in range(0, int(RATE / CHUNK * seconds)):
        data = stream.read(CHUNK)
        frames.append(data)



    stream.stop_stream()
    stream.close()

    recorded_audio = np.frombuffer(b''.join(frames), dtype=np.int16)
    print("Recording stopped")
    print("Length of recorded audio:", len(recorded_audio))

    # Terminate PyAudio
    audio.terminate()
```

Next, we have the tensor flow helper code for audio preprocessing and converting the audio to spectrograms:

```
import numpy as np
import tensorflow as tf


# Set the seed value for experiment reproducibility.
seed = 42
tf.random.set_seed(seed)
np.random.seed(seed)


def get_spectrogram(waveform):
    # Zero-padding for an audio waveform with less than 16,000 samples.
    input_len = 16000
    waveform = waveform[:input_len]
    zero_padding = tf.zeros(
        [16000] - tf.shape(waveform),
        dtype=tf.float32)
    # Cast the waveform tensors' dtype to float32.
    waveform = tf.cast(waveform, dtype=tf.float32)
    # Concatenate the waveform with `zero_padding`, which ensures all audio
    # clips are of the same length.
    equal_length = tf.concat([waveform, zero_padding], 0)
    # Convert the waveform to a spectrogram via a STFT.
    spectrogram = tf.signal.stft(
        equal_length, frame_length=255, frame_step=128)
    # Obtain the magnitude of the STFT.
    spectrogram = tf.abs(spectrogram)
    # Add a `channels` dimension, so that the spectrogram can be used
    # as image-like input data with convolution layers (which expect
    # shape (`batch_size`, `height`, `width`, `channels`).
    spectrogram = spectrogram[..., tf.newaxis]
    return spectrogram
```

```
def preprocess_audiobuffer(waveform):
    """
    waveform: ndarray of size (16000, )

    output: Spectogram Tensor of size: (1, `height`, `width`, `channels`)
    """
    waveform = np.frombuffer(waveform, dtype=np.int16)
    waveform = waveform.astype(np.float32)

    # Convert waveform to floating-point type

    # Scale waveform to the range [-1, 1]
    waveform = waveform / 32767
    #  normalize from [-32768, 32767] to [-1, 1]
    waveform =  waveform/32768

    waveform = tf.convert_to_tensor(waveform, dtype=tf.float32)

    spectogram = get_spectrogram(waveform)

    # add one dimension
    spectogram = tf.expand_dims(spectogram, 0)

    return spectogram
```

Then, the printing file on the OLED display to print the commands predicted on the OLED:

```
import time
from luma.core.interface.serial import i2c
from luma.oled.device import sh1106
from luma.core.render import canvas

def get_device():
    # Define your I2C interface and device here
    serial = i2c(port=1, address=0x3C)  # Adjust the address based on your OLED
display
    return sh1106(serial)

def right():
    device = get_device()
    with canvas(device) as draw:
        draw.text((10, 10), "RIGHT", fill="white")
    time.sleep(20)

def up():
```

```python
    device = get_device()
    with canvas(device) as draw:
        draw.text((10, 10), "UP", fill="white")
    time.sleep(20)

def left():
    device = get_device()
    with canvas(device) as draw:
        draw.text((10, 10), "LEFT", fill="white")
    time.sleep(20)

def down():
    device = get_device()
    with canvas(device) as draw:
        draw.text((10, 10), "DOWN", fill="white")
    time.sleep(20)

def yes():
    device = get_device()
    with canvas(device) as draw:
        draw.text((10, 10), "YES", fill="white")
    time.sleep(20)

def no():
    device = get_device()
    with canvas(device) as draw:
        draw.text((10, 10), "NO", fill="white")
    time.sleep(20)

def go():
    device = get_device()
    with canvas(device) as draw:
        draw.text((10, 10), "GO", fill="white")
    time.sleep(20)

def stop():
    device = get_device()
    with canvas(device) as draw:
        draw.text((10, 10), "STOP", fill="white")
    time.sleep(20)

def printOLED(command):
    if command == 'up':
        up()
    elif command == 'down':
        down()
    elif command == 'left':
        left()
    elif command == 'right':
        right()
    elif command == 'go':
        go()
    elif command == 'stop':
        stop()
    elif command == 'yes':
        yes()
    elif command == 'no':
        no()
```

Lastly, the main python code where we call all the codes mentioned earlier to predict the command:

```
import numpy as np
import h5py
import tensorflow as tf
from tensorflow import keras
from recording_helper import record_audio
from tfhelper import preprocess_audiobuffer
import time
from luma.core.interface.serial import i2c
from luma.oled.device import sh1106
from luma.core.render import canvas
import pyaudio
import scipy.signal  # Import the scipy module for resampling

commands = ['stop' ,'up' ,'yes', 'down' ,'no' ,'left' ,'right' ,'go']
loaded_model = keras.models.load_model('/home/pi/Desktop/OLEVIBRA/my_model.h5')
loaded_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

def get_device():
    # Define your I2C interface and device here
    serial = i2c(port=1, address=0x3C)  # Adjust the address based on your OLED
display
    return sh1106(serial)

def start():
    device = get_device()
    with canvas(device) as draw:
        draw.text((10, 10), "WELCOME TO OLEVIBRA!", fill="white")
    time.sleep(10)

def upsample_audio(audio, target_length=16000):
    original_length = len(audio)
    # Upsample the audio using linear interpolation
    upsampled_audio = scipy.signal.resample(audio, target_length)
    return upsampled_audio

def predict_mic():
    audio = record_audio()
    # Upsample the audio to 16,000 samples
    audio_upsampled = upsample_audio(audio)
    spec = preprocess_audiobuffer(audio_upsampled)
    prediction = loaded_model.predict(spec)
    label_pred = np.argmax(prediction, axis=1)
    label_index = label_pred[0]
    print("Predicted label index:", label_index)
    command = commands[label_index]
    print("Predicted command:", command)
    command = commands[label_index]
    print("Predicted label:", command)
    return command

if __name__ == "__main__":
    from Oledprintcom import printOLED
    FORMAT = pyaudio.paInt16
    CHANNELS = 1
    RATE = 16000
    CHUNK = 512

    p = pyaudio.PyAudio()
    stream = p.open(format=FORMAT,
                    channels=CHANNELS,
                    rate=RATE,
                    input=True,
                    frames_per_buffer=CHUNK)

    while True:
        start()
        command = predict_mic()
        printOLED(command)
```

After we run the code, the command displayed on the OLED display is always the last command in the command list after we tried to switch the order of the commands in the command list.

We tried to up sample the audio since the input audio coming from the USB microphone is not 16 bit as the one used in the model but 8 bit.

Even after up sampling the audio to 16 bit, the result obtained remained the same.

Next, to solve this issue I rerun the Google Colab and tried saving the model in 3 different ways

.h5 / saved_model/.keras and for the three ways the result remained the same.

We also tried to change the seed number in the preprocessing code.

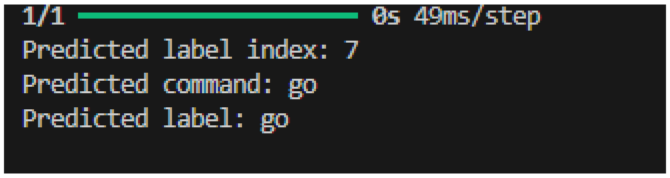
Furthermore, we tried to compile the model after loading it in the code and we still had the same results.

After, thorough research and reading through stack overflow and Github discussion forms we noticed that almost everyone that used this model had the same issue in the prediction: it's always predicting the same command over and over again even without speaking.

This is most likely due that there is no ambient noise aspect taken into consideration in the model, which is leading to this constant error in the prediction. According, to people who used this model and had the same error I am getting, the issue is in the model incapability to respond to ambient noise.

After multiple trials, we achieved a dead end in this error even after editing the preprocessing method to match it with the one used in the model.

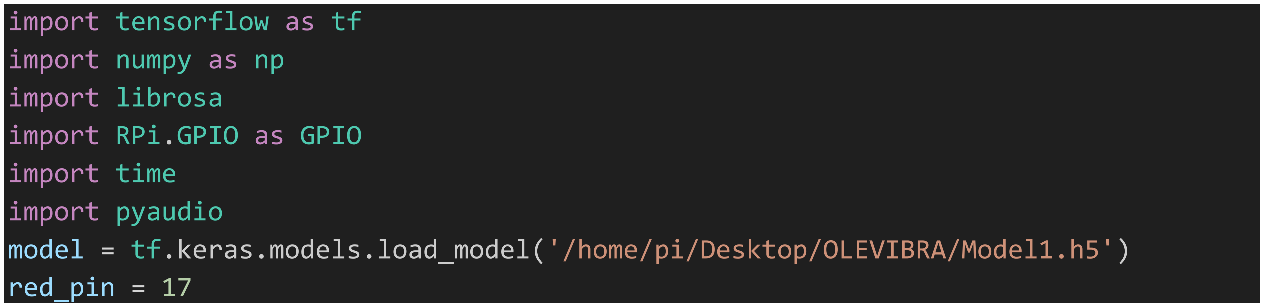
Result:



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 49ms/step
Predicted label index: 7
Predicted command: go
Predicted label: go
```

Second Deep Learning model: Urban sound classification

Code used for this model:



```
import tensorflow as tf
import numpy as np
import librosa
import RPi.GPIO as GPIO
import time
import pyaudio
model = tf.keras.models.load_model('/home/pi/Desktop/OLEVIBRA/Model1.h5')
red_pin = 17
```

```python
yellow_pin = 22
blue_pin = 27
white_pin=10
GPIO.setmode(GPIO.BCM)
GPIO.setup(red_pin, GPIO.OUT)
GPIO.setup(yellow_pin, GPIO.OUT)
GPIO.setup(blue_pin, GPIO.OUT)
GPIO.setup(white_pin, GPIO.OUT)
def classify_audio(audio_data):

    audio_data = audio_data.reshape(1, -1)
    predictions = model.predict(audio_data)
    return np.argmax(predictions)


def control_led(prediction):
    #siren
    if prediction == 8:
        # Class 0 prediction, turn on red LED
        GPIO.output(red_pin, GPIO.HIGH)
        GPIO.output(yellow_pin, GPIO.LOW)
        GPIO.output(blue_pin, GPIO.LOW)
        GPIO.output(white_pin, GPIO.LOW)
    elif prediction == 6:
        #gunshot
        GPIO.output(red_pin, GPIO.LOW)
        GPIO.output(yellow_pin, GPIO.HIGH)
        GPIO.output(blue_pin, GPIO.LOW)
        GPIO.output(white_pin, GPIO.LOW)
    elif prediction == 1:
        #carhorn
        GPIO.output(red_pin, GPIO.LOW)
        GPIO.output(yellow_pin, GPIO.LOW)
        GPIO.output(blue_pin, GPIO.HIGH)
        GPIO.output(white_pin, GPIO.LOW)
    elif prediction == 3:
        #Dogbark
        GPIO.output(red_pin, GPIO.LOW)
        GPIO.output(yellow_pin, GPIO.LOW)
        GPIO.output(blue_pin, GPIO.LOW)
        GPIO.output(white_pin, GPIO.HIGH)

# Function to continuously record audio from USB microphone
def record_audio(stream, CHUNK):
    while True:
        data = stream.read(CHUNK)
        audio_data = np.frombuffer(data, dtype=np.int16)
        prediction = classify_audio(audio_data)
        control_led(prediction)

# Setup USB microphone stream
CHUNK = 512
FORMAT = pyaudio.paInt16
CHANNELS = 1
RATE = 44100
p = pyaudio.PyAudio()
stream = p.open(format=FORMAT,
                channels=CHANNELS,
                rate=RATE,
                input=True,
                frames_per_buffer=CHUNK)

# Record and classify audio
try:
    print("Recording started...")
    record_audio(stream, CHUNK)
except KeyboardInterrupt:
    print("Recording stopped by user.")
finally:
    # Turn off all LEDs
    GPIO.output(red_pin, GPIO.LOW)
    GPIO.output(yellow_pin, GPIO.LOW)
    GPIO.output(blue_pin, GPIO.LOW)
    GPIO.output(white_pin, GPIO.LOW)

    # Cleanup GPIO and stream
    GPIO.cleanup()
    stream.stop_stream()
    stream.close()
    p.terminate()
```

For this code, and based on the prediction:

We only used the dataset to predict the 4 sounds:

When the sound detected is the sound of a siren → Red LED is ON

When the sound detected is the sound of a gunshot → Yellow LED is ON

When the sound detected is the sound of a car horn → Blue LED is ON

When the sound detected is the sound of a dog bark → White LED is ON

Final Design

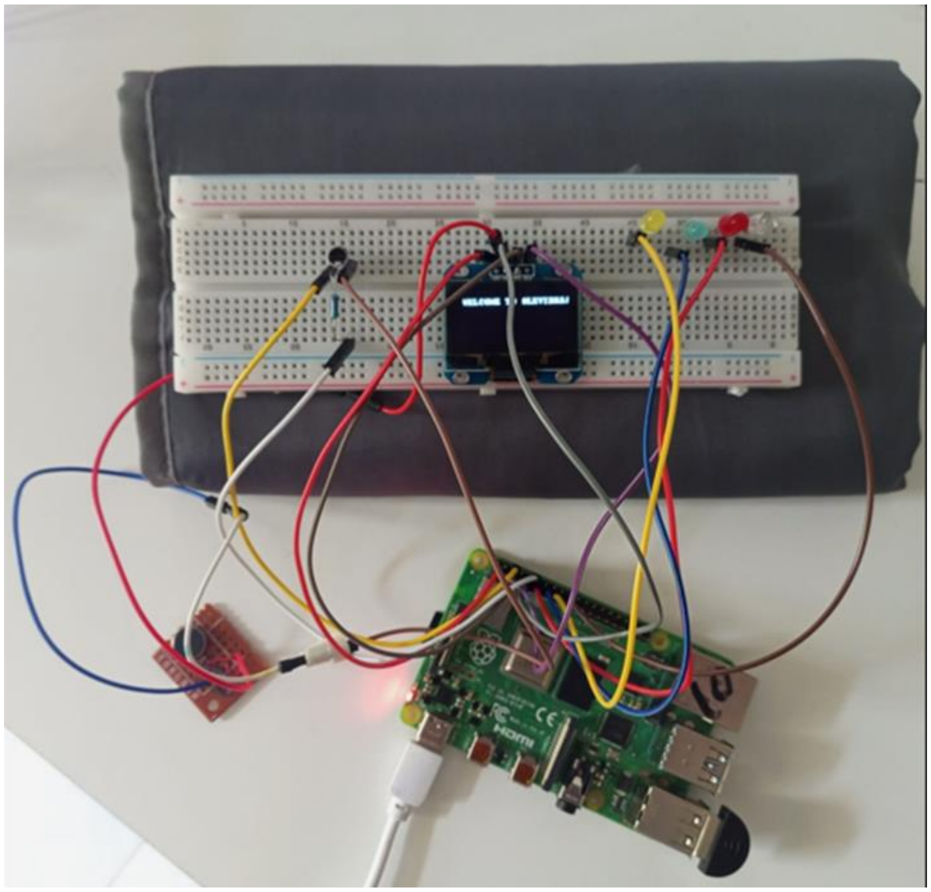
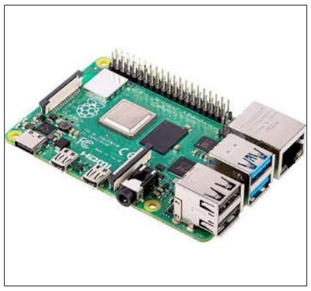


Figure 3: Final Design

Limitations

Limitation with Raspberry pi



- Arduino BLE Sense is not accessible. Therefore, we converted all codes from C++ to Python.
- The Raspberry Pi lacks integrated analog pins for connecting the microphone module. Consequently, we used a USB microphone connected to the Raspberry Pi.

Circular Array Issues

Two fundamental issues arise from using the microphone circular array:

- **Spatial Resolution:** Spatial resolution refers to the array's ability to discern the angles of incoming sound waves. A key factor influencing this capability is the distance between the microphones: the greater the separation, the more precise the measurement of delays between sound arrivals. Considering the limitations imposed by the size of the bracelet, we aimed to achieve the most optimal spatial resolution possible. By strategically arranging the microphones within these constraints, we aimed to strike a balance between spatial accuracy and practicality, ensuring effective sound localization and detection within the device's compact design.

- **Reflections:** The most favorable environment for using beam forming technology is free space, where sounds travel free from reflectors. Reflectors present multiple different incoming directions in spatial dimension in addition to the different time-of-arrivals in the time domain.

Deep learning model Issues

Issues discussed in the deep learning part of the project that limited our ability to see correct output after writing the code.

Conclusion

In our increasingly interconnected society, ensuring accessibility and inclusivity is paramount. Among the various communities benefiting from technological advancements, the deaf community faces unique challenges in environments reliant on auditory cues. To address these challenges, a pioneering innovation has emerged: a bracelet designed specifically to enhance accessibility and awareness for the deaf.

This innovative device incorporates a color-coded representation system to detect several types of sounds, alongside vibration features that reflect the intensity of the surrounding auditory environment. As sound intensity increases, the bracelet provides corresponding vibrational feedback to its wearer, offering vital sensory information.

The invention of this bracelet is a big step forward in helping deaf people. It uses both sight and touch to give feedback, making it better than previous solutions that only rely on hearing. This shows a commitment to making the world easier to access and understand for everyone, especially the deaf.