ChatGPT 4o

Share

Save as PDF

RU

hi i am a university student taking a module on "edge computing and analytics". i need your help guiding me through a lab.

Hey! That sounds interesting — I'd be happy to help you with your "Edge Computing and Analytics" lab. Just let me know:

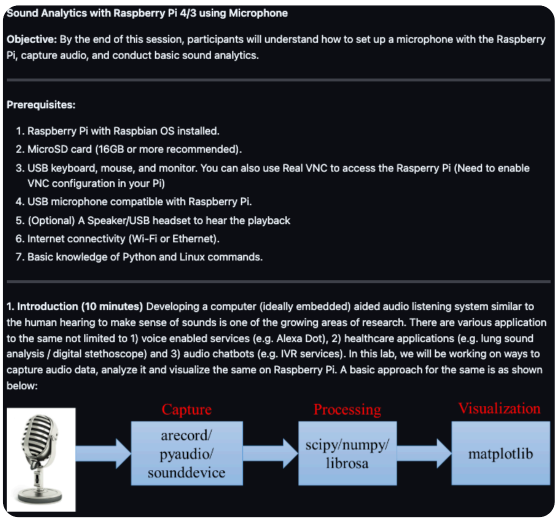
1. What's the lab about (title or topic)?
2. What platform or tools are you using (e.g., Raspberry Pi, AWS Greengrass, Azure IoT, Python, etc.)?
3. Do you have any specific instructions or files?
4. Where are you stuck or what do you want guidance on?

Feel free to copy-paste the lab sheet or describe what you see — and we'll go step by step!

2/2

the lab is on sound analytics. i will be providing you the lab sheet in sections.

Awesome — sound analytics sounds fun and practical, especially in edge computing. Go ahead and send over the first section of the lab sheet when you're ready, and we'll take it from there!



**Sound Analytics with Raspberry Pi 4/3 using Microphone**

**Objective:** By the end of this session, participants will understand how to set up a microphone with the Raspberry Pi, capture audio, and conduct basic sound analytics.

**Prerequisites:**

1. Raspberry Pi with Raspbian OS installed.
2. MicroSD card (16GB or more recommended).
3. USB keyboard, mouse, and monitor. You can also use Real VNC to access the Rasperry Pi (Need to enable VNC configuration in your Pi)
4. USB microphone compatible with Raspberry Pi.
5. (Optional) A Speaker/USB headset to hear the playback
6. Internet connectivity (Wi-Fi or Ethernet).
7. Basic knowledge of Python and Linux commands.

**1. Introduction (10 minutes)** Developing a computer (ideally embedded) aided audio listening system similar to the human hearing to make sense of sounds is one of the growing areas of research. There are various application to the same not limited to 1) voice enabled services (e.g. Alexa Dot), 2) healthcare applications (e.g. lung sound analysis / digital stethoscope) and 3) audio chatbots (e.g. IVR services). In this lab, we will be working on ways to capture audio data, analyze it and visualize the same on Raspberry Pi. A basic approach for the same is as shown below:



Thanks for sharing the first part of the lab sheet! Here's a quick breakdown of what it's covering:

---

## 🔊 Lab Title:

**Sound Analytics with Raspberry Pi 4/3 using Microphone**

---

## 🎯 Objective:

By the end of the lab, you should be able to:

- Set up a microphone with your Raspberry Pi
- Capture audio
- Analyze and visualize sound data

---

## 🧰 Prerequisites:

Make sure you have:

1. A Raspberry Pi with **Raspbian OS**
2. A **16GB+ MicroSD card**
3. A way to interact with the Pi (USB peripherals or **VNC**)
4. A **USB microphone**
5. (Optional) **Speaker/USB headset** for playback
6. **Internet connection**
7. Basic skills in **Python** and **Linux commands**

---

## 📚 Introduction Summary:

- The lab mimics real-world embedded audio systems.
- Applications mentioned:
    1. Voice assistants (e.g., Alexa)
    2. Healthcare tools (e.g., lung sound analysis)
    3. IVR/chatbots

- You'll explore:
  - **Audio Capture:** using `arecord`, `pyaudio`, or `sounddevice`
  - **Processing:** using `scipy`, `numpy`, or `librosa`
  - **Visualization:** using `matplotlib`

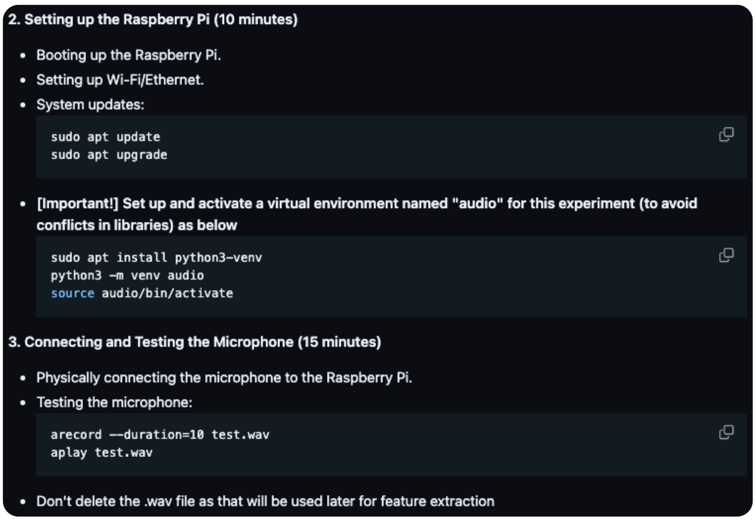
---

## 🧭 Next Steps:

Once you send the next part, we can dive into:

- Microphone setup on Raspberry Pi
- Audio recording (likely with `arecord` or Python libraries)
- Signal processing and visualization

Feel free to send the next section whenever you're ready!



**2. Setting up the Raspberry Pi (10 minutes)**

- Booting up the Raspberry Pi.
- Setting up Wi-Fi/Ethernet.
- System updates:

```
sudo apt update
sudo apt upgrade
```

- **[Important!] Set up and activate a virtual environment named "audio" for this experiment (to avoid conflicts in libraries) as below**

```
sudo apt install python3-venv
python3 -m venv audio
source audio/bin/activate
```

**3. Connecting and Testing the Microphone (15 minutes)**

- Physically connecting the microphone to the Raspberry Pi.
- Testing the microphone:

```
arecord --duration=10 test.wav
aplay test.wav
```

- Don't delete the .wav file as that will be used later for feature extraction

Great — thanks for sending the next section! Here's a step-by-step breakdown of what you're expected to do next:

---

## 🔧 2. Setting up the Raspberry Pi (10 minutes)

### ✅ Steps:

1. **Boot up** your Raspberry Pi and ensure it's connected to **Wi-Fi or Ethernet**.

2. **Update your system packages** by running:

```bash
sudo apt update
sudo apt upgrade
```

🐍 **Create and Activate a Virtual Environment:**

This keeps your Python packages isolated for this project.

1. **Install venv (if not already installed):**

```bash
sudo apt install python3-venv
```

2. **Create a virtual environment called `audio`:**

```bash
python3 -m venv audio
```

3. **Activate it:**

```bash
source audio/bin/activate
```

---

## 🎤 3. Connecting and Testing the Microphone (15 minutes)

🔌 **Connect:**

- Plug your USB microphone into the Raspberry Pi.

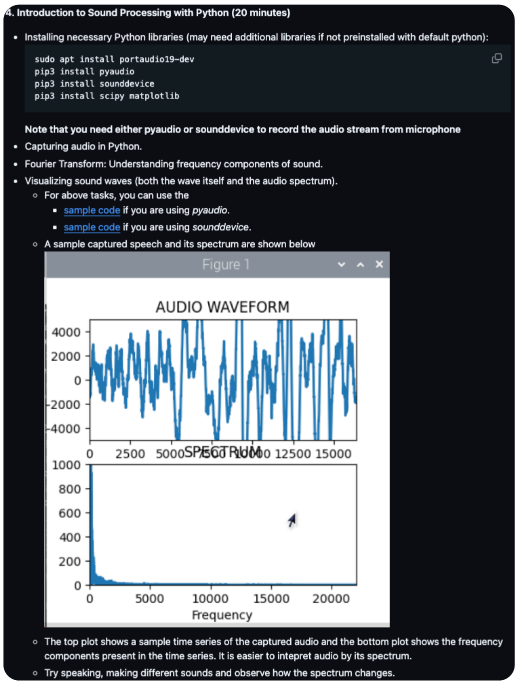
🔊 **Test:**

Record 10 seconds of audio and play it back:

```bash
arecord --duration=10 test.wav
aplay test.wav
```

💡 **Note:** The file `test.wav` will be used later for **feature extraction**, so **don't delete it**!

Let me know when you're done with this part or if you run into any issues (e.g., microphone not working, errors with `arecord`, etc.). Ready for the next section whenever you are!


**4. Introduction to Sound Processing with Python (20 minutes)**

- Installing necessary Python libraries (may need additional libraries if not preinstalled with default python):

```
sudo apt install portaudio19-dev
pip3 install pyaudio
pip3 install sounddevice
pip3 install scipy matplotlib
```

**Note that you need either pyaudio or sounddevice to record the audio stream from microphone**
- Capturing audio in Python.
- Fourier Transform: Understanding frequency components of sound.
- Visualizing sound waves (both the wave itself and the audio spectrum).
  - For above tasks, you can use the
    - sample code if you are using *pyaudio*.
    - sample code if you are using *sounddevice*.
  - A sample captured speech and its spectrum are shown below



  - The top plot shows a sample time series of the captured audio and the bottom plot shows the frequency components present in the time series. It is easier to intepret audio by its spectrum.
  - Try speaking, making different sounds and observe how the spectrum changes.

pyaudio code:

```
#%% Import the required libraries
import pyaudio # Refer to
https://people.csail.mit.edu/hubert/pyaudio/
import struct  # Refer to
https://docs.python.org/3/library/struct.html (Used for
converting audio read as bytes to int16)
import numpy as np
import matplotlib.pyplot as plt
from scipy.fftpack import fft, fftfreq # Refer to
https://docs.scipy.org/doc/scipy/tutorial/fft.html (Used for
Fourier Spectrum to display audio frequencies)
import time # In case time of execution is required

#%% Parameters
BUFFER = 1024 * 16          # samples per frame (you can
change the same to acquire more or less samples)
FORMAT = pyaudio.paInt16     # audio format (bytes per
sample)
CHANNELS = 1                 # single channel for microphone
RATE = 44100                 # samples per second
RECORD_SECONDS = 30          # Specify the time to record
from the microphone in seconds

#%% create matplotlib figure and axes with initial random plots
```

```
as placeholder
fig, (ax1, ax2) = plt.subplots(2, figsize=(7, 7))
# create a line object with random data
x = np.arange(0, 2*BUFFER, 2)       # samples (waveform)
xf = fftfreq(BUFFER, (1/RATE))[:BUFFER//2]

line, = ax1.plot(x,np.random.rand(BUFFER), '-', lw=2)
line_fft, = ax2.plot(xf,np.random.rand(BUFFER//2), '-', lw=2)

# basic formatting for the axes
ax1.set_title('AUDIO WAVEFORM')
ax1.set_xlabel('samples')
ax1.set_ylabel('volume')
ax1.set_ylim(-5000, 5000) # change this to see more amplitude
values (when we speak)
ax1.set_xlim(0, BUFFER)

ax2.set_title('SPECTRUM')
ax2.set_xlabel('Frequency')
ax2.set_ylabel('Log Magnitude')
ax2.set_ylim(0, 1000)
ax2.set_xlim(0, RATE/2)

# Do not show the plot yet
plt.show(block=False)

#%% Initialize the pyaudio class instance
audio = pyaudio.PyAudio()

# stream object to get data from microphone
stream = audio.open(
    format=FORMAT,
    channels=CHANNELS,
    rate=RATE,
    input=True,
    output=True,
    frames_per_buffer=BUFFER
)

print('stream started')

exec_time = []
for _ in range(0, RATE // BUFFER * RECORD_SECONDS):

    # binary data
    data = stream.read(BUFFER)
```

```
        # convert data to 16bit integers
        data_int = struct.unpack(str(BUFFER) + 'h', data)

        # compute FFT
        start_time=time.time()  # for measuring frame rate
        yf = fft(data_int)

        # calculate time of execution of FFT
        exec_time.append(time.time() - start_time)

        #update line plots for both axes
        line.set_ydata(data_int)
        line_fft.set_ydata(2.0/BUFFER * np.abs(yf[0:BUFFER//2]))
        fig.canvas.draw()
        fig.canvas.flush_events()

    audio.terminate()

    print('stream stopped')
    print('average execution time = {:.0f} milli
    seconds'.format(np.mean(exec_time)*1000))
```

Perfect — this is a key part of the lab where things get interactive. Let's break it down and guide you through it:

---

## 🔍 4. Introduction to Sound Processing with Python (20 minutes)

### 🛠️ Step 1: Install Required Libraries

Run these commands **inside your virtual environment** ( `audio` ) to install the necessary packages:

bash

```bash
sudo apt install portaudio19-dev
pip3 install pyaudio
pip3 install sounddevice
pip3 install scipy matplotlib
```

## 🎙️ Step 2: Capturing and Processing Audio with `pyaudio`

You provided a complete sample script — nice! Here's what it does:

- **Captures live audio** from your mic using `pyaudio`
- **Visualizes** the audio waveform (top plot) and its **frequency spectrum** via FFT (bottom plot)
- **Prints average FFT execution time**

---

## ✅ What You'll Learn from the Script:

| Part | Purpose |
|---|---|
| `pyaudio.open(...)` | Starts the audio stream |
| `stream.read(...)` | Captures buffer of audio data |
| `struct.unpack(...)` | Converts byte stream into integers |
| `fft(data_int)` | Computes frequency components (Fourier Transform) |
| `matplotlib` | Plots waveform + spectrum in real-time |
| `flush_events()` | Keeps the plot window updating live |

---

## 🖼️ Interpreting the Output:

- **Top graph**: Time-domain waveform. Peaks represent loudness.
- **Bottom graph**: Frequency spectrum. Higher peaks = stronger presence of that frequency.
- Try saying different words/sounds (e.g., "ahhh", clapping) and watch how the plots change!

---

## ⚠️ Notes & Tips:

- You can tweak `RECORD_SECONDS` or `BUFFER` size for longer/shorter streams.
- Make sure only **one** of `pyaudio` or `sounddevice` is being used at a time (not both).
- If `pyaudio` doesn't work (common on some OS versions), switch to `sounddevice`.

Let me know when you're ready for the `sounddevice` **version** or the next lab section! Or if you'd like help testing this one.

can you explain the script to me in detail?

Absolutely! Let's go **line-by-line and section-by-section** through the `pyaudio` script to fully understand how it captures and processes audio in real time:

---

## 🎯 Objective of the Script

To:

- Record audio live using a USB microphone,
- Display its **waveform** (volume over time),
- Compute and visualize its **frequency spectrum** using **FFT** (Fourier Transform),
- Track the performance (execution time) of the FFT step.

---

## 🧩 Section-by-Section Breakdown

### 1. Importing Libraries

python Copy Edit

```python
import pyaudio
import struct
import numpy as np
import matplotlib.pyplot as plt
from scipy.fftpack import fft, fftfreq
import time
```

- `pyaudio` : Used to capture audio from your mic.
- `struct` : Converts raw byte data into integers.

- `numpy` : Useful for numeric operations.
- `matplotlib.pyplot` : For plotting graphs.
- `fft` , `fftfreq` (from `scipy.fftpack` ): Compute and label the frequency spectrum.
- `time` : Used to measure FFT execution time.

---

## 2. Parameter Definitions

python

```python
BUFFER = 1024 * 16
FORMAT = pyaudio.paInt16
CHANNELS = 1
RATE = 44100
RECORD_SECONDS = 30
```

- `BUFFER` : Number of samples per audio frame (larger buffer = smoother but slower response).
- `FORMAT` : 16-bit integer audio format.
- `CHANNELS` : Mono audio (1 channel).
- `RATE` : Sample rate (44.1 kHz = CD quality).
- `RECORD_SECONDS` : How long to run the recording loop.

---

## 3. Plot Setup

python

```python
fig, (ax1, ax2) = plt.subplots(2, figsize=(7, 7))
x = np.arange(0, 2*BUFFER, 2)
xf = fftfreq(BUFFER, (1/RATE))[:BUFFER//2]

line, = ax1.plot(x, np.random.rand(BUFFER), '-', lw=2)
line_fft, = ax2.plot(xf, np.random.rand(BUFFER//2), '-', lw=2)
```

- Creates two subplots:
  - `ax1` : for the **waveform**

    - `ax2` : for the **FFT (spectrum)**
- `x` : x-axis for waveform plot (samples)
- `xf` : frequency bins for FFT plot
- `line` and `line_fft` : placeholders for updating plots in real time

**Set axis labels and limits:**

```python
ax1.set_title('AUDIO WAVEFORM')
...
ax2.set_title('SPECTRUM')
...
plt.show(block=False)
```

- Prepares the plot window without blocking script execution (so it can keep updating)

---

## 4. Start Audio Stream

```python
audio = pyaudio.PyAudio()
stream = audio.open(...)
```

- `audio.open(...)` : Opens a real-time audio stream with your mic settings:
    - `input=True` : you're recording from the mic
    - `output=True` : allows playback (optional here)
    - `frames_per_buffer=BUFFER` : how many samples per read

---

## 5. Main Loop – Capturing and Plotting

```python
for _ in range(0, RATE // BUFFER * RECORD_SECONDS):
    data = stream.read(BUFFER)
    data_int = struct.unpack(str(BUFFER) + 'h', data)
```

- `stream.read(BUFFER)` : Reads one chunk of raw audio bytes.
- `struct.unpack(...)` : Converts raw bytes into a list of integers (volume values).

**FFT + Visualization:**

python Copy Edit

```python
    start_time = time.time()
    yf = fft(data_int)
    exec_time.append(time.time() - start_time)
```

- Computes the **FFT** of the audio signal to get its **frequency spectrum**.
- Measures how long the FFT took to execute (performance metric).

**Update the plots:**

python Copy Edit

```python
    line.set_ydata(data_int)
    line_fft.set_ydata(2.0/BUFFER * np.abs(yf[0:BUFFER//2]))
    fig.canvas.draw()
    fig.canvas.flush_events()
```

- Updates the waveform plot with new audio data.
- Updates the FFT plot with the corresponding spectrum.

---

## 6. Cleanup

python Copy Edit

```python
audio.terminate()
print('stream stopped')
print('average execution time = {:.0f} milli
seconds'.format(np.mean(exec_time)*1000))
```

- Terminates the audio session.
- Prints the **average time** taken per FFT, in milliseconds.

---

## 💡 Notes on Concepts

| Concept | Explanation |
|---|---|
| **Waveform** | Shows changes in air pressure over time (volume amplitude). |
| **FFT (Fast Fourier Transform)** | Breaks the signal into frequencies. Helps analyze pitch, tone, etc. |
| **Execution Time** | Measures how fast FFT is computed — important for real-time edge systems. |

---

Would you like:

- A simplified version of the script?
- Help running or modifying it?
- Or would you like to look at the `sounddevice` version next?

Let me know what you'd like to explore!

yes, here is the sounddevice script:
#%% Import the required libraries
import sounddevice as sd # Refer to https://python-sounddevice.readthedocs.io/en/0.4.6/
import numpy as np
import matplotlib.pyplot as plt
import time

#%% Parameters
BUFFER = 1024 * 16           # samples per frame (you can change the same to acquire more or less samples)
CHANNELS = 1                 # single channel for microphone
RATE = 44100                 # samples per second
RECORD_SECONDS = 30          # Specify the time to record from the microphone in seconds

#%% create matplotlib figure and axes with initial random plots as placeholder
fig, (ax1, ax2) = plt.subplots(2, figsize=(4, 4))
# create a line object with random data
x = np.arange(0, 2*BUFFER, 2)       # samples (waveform)
xf = np.fft.fftfreq(BUFFER,1/RATE)[:BUFFER//2]

```
line, = ax1.plot(x,np.random.rand(BUFFER), '-', lw=2)
line_fft, = ax2.plot(xf,np.random.rand(BUFFER//2), '-', lw=2)

# basic formatting for the axes
ax1.set_title('AUDIO WAVEFORM')
ax1.set_xlabel('samples')
ax1.set_ylabel('volume')
ax1.set_ylim(-5000, 5000) # change this to see more amplitude
values (when we speak)
ax1.set_xlim(0, BUFFER)

ax2.set_title('SPECTRUM')
ax2.set_xlabel('Frequency')
ax2.set_ylabel('Log Magnitude')
ax2.set_ylim(0, 1000)
ax2.set_xlim(0, RATE/2)

# Do not show the plot yet
plt.show(block=False)

#%% Reconrding the sound and constructing the spectrum
exec_time = []
for _ in range(0, RATE // BUFFER * RECORD_SECONDS):

    # Record the sound in int16 format and wait till recording is
done
    data =
sd.rec(frames=BUFFER,samplerate=RATE,channels=CHANN
ELS,dtype='int16',blocking=True)
    data = np.squeeze(data)

    # compute FFT
    start_time=time.time()  # for measuring frame rate
    fft_data = np.fft.fft(data)
    fft_data = np.abs(fft_data[:BUFFER//2])

    # calculate time of execution of FFT
    exec_time.append(time.time() - start_time)

    #update line plots for both axes
    line.set_ydata(data)
    line_fft.set_ydata(fft_data)
    line_fft.set_ydata(2.0/BUFFER * fft_data)
    fig.canvas.draw()
    fig.canvas.flush_events()
```

print('stream stopped')
print('average execution time = {:.0f} milli
seconds'.format(np.mean(exec_time)*1000))

Thanks for sharing the `sounddevice` version! Let's walk through it in detail — and also compare it a bit with the `pyaudio` version so you understand when and why you might prefer one over the other.

---

## 🎧 Real-Time Audio Analysis with `sounddevice` — Line-by-Line Breakdown

---

### 🧩 1. Library Imports

python Copy Edit

```
import sounddevice as sd
import numpy as np
import matplotlib.pyplot as plt
import time
```

- `sounddevice` : Used to **record audio** easily (higher-level & easier than `pyaudio` ).
- `numpy` : For numerical and FFT operations.
- `matplotlib` : For real-time plotting.
- `time` : To track FFT execution time.

---

### ⚙️ 2. Parameter Setup

python Copy Edit

```
BUFFER = 1024 * 16
CHANNELS = 1
```

```
RATE = 44100
RECORD_SECONDS = 30
```

Same as `pyaudio` :

- `BUFFER` : Audio samples per frame
- `CHANNELS` : Mono recording
- `RATE` : Sampling rate (standard: 44.1 kHz)
- `RECORD_SECONDS` : Duration of the experiment

---

## 📊 3. Plot Setup

```python
fig, (ax1, ax2) = plt.subplots(2, figsize=(4, 4))
x = np.arange(0, 2*BUFFER, 2)
xf = np.fft.fftfreq(BUFFER,1/RATE)[:BUFFER//2]
```

- Creates two subplots: time domain + frequency domain.
- `x` : X-axis for the waveform (sample index).
- `xf` : Frequency bins (x-axis for the spectrum plot).

**Initialize plots with random data:**

```python
line, = ax1.plot(x, np.random.rand(BUFFER), '-', lw=2)
line_fft, = ax2.plot(xf, np.random.rand(BUFFER//2), '-', lw=2)
```

Placeholders for live-updating waveform and spectrum.

**Set plot labels and limits:**

```python
ax1.set_title('AUDIO WAVEFORM')
...
ax2.set_title('SPECTRUM')
...
plt.show(block=False)
```

Just like before: prepares and formats the plot window.

---

## 🎙️ 4. Recording and Processing Audio

python

```python
for _ in range(0, RATE // BUFFER * RECORD_SECONDS):
```

Loops for ~30 seconds in real time, based on sample rate and buffer size.

### 🔴 Recording Audio

python

```python
data = sd.rec(frames=BUFFER, samplerate=RATE, channels=CHANNELS,
dtype='int16', blocking=True)
data = np.squeeze(data)
```

- `sd.rec(...)` : Captures `BUFFER` audio frames.
- `blocking=True` : Ensures each recording finishes before moving on.
- `squeeze()` : Converts 2D shape to 1D for easier FFT plotting.

### 📉 Fast Fourier Transform

python

```python
start_time = time.time()
fft_data = np.fft.fft(data)
fft_data = np.abs(fft_data[:BUFFER//2])
exec_time.append(time.time() - start_time)
```

- Computes the **frequency components** using FFT.
- `np.abs(...)` : Gets the magnitude (removes complex values).
- Keeps track of time taken per FFT.

### 🔁 Update the Plots

python

```python
line.set_ydata(data)
line_fft.set_ydata(2.0/BUFFER * fft_data)
```

```
fig.canvas.draw()
fig.canvas.flush_events()
```

- Updates the waveform ( `line` ) and spectrum ( `line_fft` ) with new data in every loop iteration.
- `2.0 / BUFFER * fft_data` : Normalizes the FFT values for display.

---

## ✅ 5. After Recording

```python
print('stream stopped')
print('average execution time = {:.0f} milli seconds'.format(np.mean(exec_time)*1000))
```

Gives you the **average FFT processing time**, useful for analyzing real-time system performance.

---

## 🔄 Comparison with `pyaudio`

| Feature | `pyaudio` | `sounddevice` |
|---|---|---|
| Low-level access | ✅ | ❌ (higher-level) |
| Easier syntax | ❌ | ✅ |
| Blocking call by default | ❌ | ✅ |
| Better error messages | ❌ | ✅ |
| More flexible audio I/O | ✅ | ❌ (less control) |

If you're just learning or doing quick analytics — `sounddevice` **is cleaner**. But for production-level work (with streaming, event callbacks, etc.), `pyaudio` gives you more control.

---

Let me know if:

- You want to **log/save data** to file,
- Want to detect **pitch or silence**,

- Or want help building this into an **edge application**!

Happy to walk through next steps.

**5. Basic Sound Analytics (40 minutes)**

- Filtering: Removing noise or specific frequencies. The below code illustrates a bandpass filter (only passes audio within a certain frequencies as decided by the user are kept).
- For the filtering task, you can use the
  - sample code if you are using *pyaudio*.
  - sample code if you are using *sounddevice*.
  - Using the audio spectrum visualization, identify the frequency to be kept (e.g. tap sound or some particular sound) and change the above code accordingly.

pyaudio script:

```
#%% Import the required libraries
import pyaudio # Refer to
https://people.csail.mit.edu/hubert/pyaudio/
import struct  # Refer to
https://docs.python.org/3/library/struct.html (Used for
converting audio read as bytes to int16)
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import butter, sosfilt # Refer to
https://docs.scipy.org/doc/scipy/reference/signal.html (Used for
Bandpass filtering)
import time # In case time of execution is required

#%% Parameters
BUFFER = 1024 * 16          # samples per frame (you can
change the same to acquire more or less samples)
FORMAT = pyaudio.paInt16    # audio format (bytes per
sample)
CHANNELS = 1                # single channel for microphone
RATE = 44100                # samples per second
RECORD_SECONDS = 20         # Specify the time to record
from the microphone in seconds

#%% create matplotlib figure and axes with initial random plots
as placeholder
fig, (ax1, ax2) = plt.subplots(2, figsize=(7, 7))
# create a line object with random data
x = np.arange(0, 2*BUFFER, 2)       # samples (waveform)

line, = ax1.plot(x,np.random.rand(BUFFER), '-', lw=2)
line_filter, = ax2.plot(x,np.random.rand(BUFFER), '-', lw=2)

# basic formatting for the axes
ax1.set_title('AUDIO WAVEFORM')
ax1.set_xlabel('samples')
```

```
ax1.set_ylabel('amplitude')
ax1.set_ylim(-5000, 5000) # change this to see more amplitude
values (when we speak)
ax1.set_xlim(0, BUFFER)

ax2.set_title('FILTERED')
ax2.set_xlabel('samples')
ax2.set_ylabel('amplitude')
ax2.set_ylim(-5000, 5000)
ax2.set_xlim(0, BUFFER)

# show the plot
plt.show(block=False)

#%% Function for design of filter
def design_filter(lowfreq, highfreq, fs, order=3):
    nyq = 0.5*fs
    low = lowfreq/nyq
    high = highfreq/nyq
    sos = butter(order, [low,high], btype='band',output='sos')
    return sos

# design the filter
sos = design_filter(19400, 19600, 48000, 3) #change the lower
and higher freqcies according to choice


#%% Initialize the pyaudio class instance
audio = pyaudio.PyAudio()

# stream object to get data from microphone
stream = audio.open(
    format=FORMAT,
    channels=CHANNELS,
    rate=RATE,
    input=True,
    output=True,
    frames_per_buffer=BUFFER
)

print('stream started')
exec_time = []
for _ in range(0, RATE // BUFFER * RECORD_SECONDS):

    # binary data
    data = stream.read(BUFFER)
```

```
        # convert data to 16bit integers
        data_int = struct.unpack(str(BUFFER) + 'h', data)

        # Bandpass filtering
        start_time=time.time()  # for measuring frame rate
        yf = sosfilt(sos, data_int)

        # calculate average frame rate
        exec_time.append(time.time() - start_time)

        #update line plots for both axes
        line.set_ydata(data_int)
        line_filter.set_ydata(yf)
        fig.canvas.draw()
        fig.canvas.flush_events()

    audio.terminate()

    print('stream stopped')
    print('average execution time = {:.0f} milli
    seconds'.format(np.mean(exec_time)*1000))
```

Awesome — you're now entering the **Sound Analytics** phase of the lab, where the focus is on **filtering** and analyzing specific frequency bands in real-time audio.

Let's break down the `pyaudio` **bandpass filter script** step-by-step to make it clear how it works:

---

## 🎯 Goal of the Script

- **Capture audio live**
- **Apply a bandpass filter** (keep only a specific frequency range)
- **Visualize** both the original and filtered audio signals in real-time

---

## 🔎 Detailed Breakdown

---

## 1. Library Imports

python

```python
import pyaudio
import struct
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import butter, sosfilt
import time
```

- `butter`, `sosfilt`: Used to create and apply a **bandpass filter**
- `sosfilt` = more numerically stable than standard `lfilter` (important for real-time audio)

---

## 2. Parameter Setup

python

```python
BUFFER = 1024 * 16
FORMAT = pyaudio.paInt16
CHANNELS = 1
RATE = 44100
RECORD_SECONDS = 20
```

Same as before — defining audio stream settings.

---

## 3. Visualization Setup

python

```python
fig, (ax1, ax2) = plt.subplots(2, figsize=(7, 7))
x = np.arange(0, 2*BUFFER, 2)
line, = ax1.plot(x,np.random.rand(BUFFER))
line_filter, = ax2.plot(x,np.random.rand(BUFFER))
```

- `ax1` = original waveform
- `ax2` = filtered waveform

- Placeholder data added; will be updated live.

---

## 4. Filter Design Function

python

```python
def design_filter(lowfreq, highfreq, fs, order=3):
    nyq = 0.5*fs
    low = lowfreq/nyq
    high = highfreq/nyq
    sos = butter(order, [low,high], btype='band', output='sos')
    return sos
```

- **Butterworth bandpass filter** generator
- `lowfreq` , `highfreq` : Frequency range to preserve
- Output: **sos** (second-order sections) filter coefficients

python

```python
sos = design_filter(19400, 19600, 48000, 3)
```

This sets up a bandpass filter between **19.4 kHz and 19.6 kHz**, ideal for ultrasonic or high-pitched signals.

🔧 **Tip:** You'll usually change these values to match:

- The **dominant frequencies** you saw earlier in your FFT plots (e.g., if tapping shows strong response at 1500 Hz, use 1400–1600 Hz).

---

## 5. Initialize and Open Audio Stream

python

```python
audio = pyaudio.PyAudio()
stream = audio.open(...)
```

Standard `pyaudio` stream setup.

---

## 6. Main Loop — Recording, Filtering, and Plotting

python

```python
for _ in range(0, RATE // BUFFER * RECORD_SECONDS):
```

Loop runs for the duration you set ( `RECORD_SECONDS` ).

### 🔴 Recording & Converting:

python

```python
data = stream.read(BUFFER)
data_int = struct.unpack(str(BUFFER) + 'h', data)
```

- `data` : raw bytes
- `data_int` : list of 16-bit integers representing the sound waveform

### 🧹 Bandpass Filtering:

python

```python
yf = sosfilt(sos, data_int)
```

Applies the bandpass filter to your data — `yf` now contains **only** the frequencies you want to keep.

### 📊 Live Plotting:

python

```python
line.set_ydata(data_int)
line_filter.set_ydata(yf)
fig.canvas.draw()
fig.canvas.flush_events()
```

- Top graph = raw input
- Bottom graph = filtered output

This helps **visualize the noise removal** or focus on specific frequency patterns (e.g., claps, whistles, taps).

---

### 7. Wrap-Up

```python
audio.terminate()
print('average execution time = ...')
```

Cleans up the audio stream and prints how long each FFT/filtering step took.

---

## 💡 What You Can Do

1. **Identify useful frequencies** from your FFT plots (from earlier scripts).
2. Replace the filter's range:

   ```python
   sos = design_filter(900, 1100, RATE, 3)
   ```

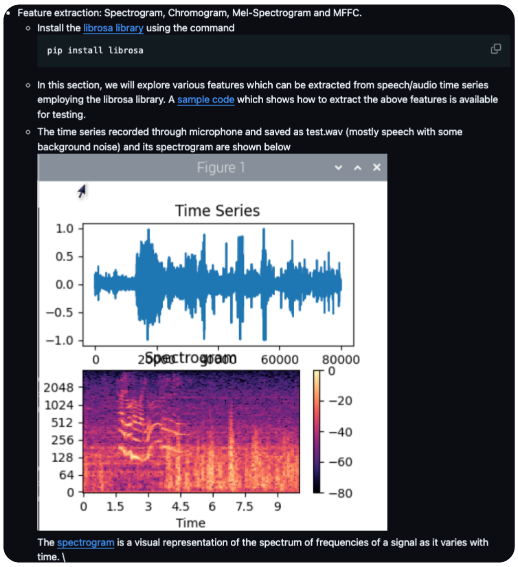
   if you're targeting ~1kHz signals, for example.
3. Try filtering:
   - Background hum (usually ~50–60 Hz)
   - Your voice (~300–3000 Hz)
   - A tap or clap (broad, high-energy burst)

---

Would you like me to:

- Explain how to adapt this for `sounddevice`?
- Help you tweak the filter for a specific use case (e.g., clap detection)?
- Export the filtered audio to a `.wav` file?

Let me know!



- Feature extraction: Spectrogram, Chromogram, Mel-Spectrogram and MFFC.
    - Install the librosa library using the command

```
pip install librosa
```

    - In this section, we will explore various features which can be extracted from speech/audio time series employing the librosa library. A sample code which shows how to extract the above features is available for testing.
    - The time series recorded through microphone and saved as test.wav (mostly speech with some background noise) and its spectrogram are shown below



The spectrogram is a visual representation of the spectrum of frequencies of a signal as it varies with time. \

script:

```
#%% Import the required libraries

import numpy as np
import matplotlib.pyplot as plt

import librosa # Used for speech feature extraction:
https://librosa.org/doc/

y, sr = librosa.load("test.wav", sr=None) #Save the microphone
recording as test.wav

#%% Compute the spectrogram magnitude and phase
S_full, phase = librosa.magphase(librosa.stft(y))

#%% Plot the time series and the frequency-time plot
(spectrogram)
fig, (ax1, ax2) = plt.subplots(2, figsize=(7, 7))
ax1.plot(y)
ax1.set_xlabel('samples')
ax1.set_ylabel('volume')
img = librosa.display.specshow(librosa.amplitude_to_db(S_full,
ref=np.max),
                    y_axis='log', x_axis='time', sr=sr, ax=ax2)
fig.colorbar(img, ax=ax2)
ax1.set(title='Time Series')
ax2.set(title='Spectrogram')
plt.show()

#%% Chroma Estimation
S = np.abs(librosa.stft(y, n_fft=4096))**2
chroma = librosa.feature.chroma_stft(S=S, sr=sr)
fig, ax = plt.subplots(nrows=2, sharex=True)
```

```
img = librosa.display.specshow(librosa.amplitude_to_db(S,
ref=np.max),
                    y_axis='log', x_axis='time', ax=ax[0])
fig.colorbar(img, ax=[ax[0]])
ax[0].label_outer()
img = librosa.display.specshow(chroma, y_axis='chroma',
x_axis='time', ax=ax[1])
fig.colorbar(img, ax=[ax[1]])
ax1.set(title='Power Spectrogram')
ax2.set(title='Chromogram')
plt.show()

#%% Compute Mel-Spectrogram
S_mel = librosa.feature.melspectrogram(y=y, sr=sr,
n_mels=128,
                        fmax=8000)
fig, ax = plt.subplots()
S_mel_dB = librosa.power_to_db(S_mel, ref=np.max)
img = librosa.display.specshow(S_mel_dB, x_axis='time',
                  y_axis='mel', sr=sr,
                  fmax=8000, ax=ax)
fig.colorbar(img, ax=ax, format='%+2.0f dB')
ax.set(title='Mel-frequency spectrogram')
plt.show()

#%% Compute MFCC
mfccs = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=40)
S = librosa.feature.melspectrogram(y=y, sr=sr, n_mels=128,
                        fmax=8000)
fig, ax = plt.subplots(nrows=2, sharex=True)
img = librosa.display.specshow(librosa.power_to_db(S,
ref=np.max),
                    x_axis='time', y_axis='mel', fmax=8000,
                    ax=ax[0])
fig.colorbar(img, ax=[ax[0]])
ax[0].set(title='Mel spectrogram')
ax[0].label_outer()
img = librosa.display.specshow(mfccs, x_axis='time', ax=ax[1])
fig.colorbar(img, ax=[ax[1]])
ax[1].set(title='MFCC')
plt.show()
```

You're now at the final and **most feature-rich part** of the lab — **Feature Extraction** using the `librosa` library. This part is where sound becomes **data** you can use for machine learning, classification, or analytics.

Let's go step-by-step through the script and explain what's going on in detail.

---

## 📦 Step 0: Install `librosa`

bash Copy Edit

```bash
pip install librosa
```

---

## 🎧 Step 1: Load the Audio

python Copy Edit

```python
y, sr = librosa.load("test.wav", sr=None)
```

- `y` : waveform (array of amplitudes)
- `sr` : sampling rate (auto-detected from the file)
- `test.wav` : should be the file you recorded earlier

---

## 🔍 Step 2: Spectrogram

### Compute and Plot

python Copy Edit

```python
S_full, phase = librosa.magphase(librosa.stft(y))
```

- `stft(y)` : Short-Time Fourier Transform — breaks the signal into overlapping chunks and applies FFT.
- `magphase()` : Separates **magnitude** and **phase** (only magnitude is visualized here).

### Visualize

```python
img = librosa.display.specshow(librosa.amplitude_to_db(S_full, ref=np.max),
                               y_axis='log', x_axis='time', sr=sr, ax=ax2)
```

- Converts raw amplitude to decibels for better visual range.
- `y_axis='log'` : Shows frequency scale logarithmically (like human hearing).
- Result: A **spectrogram** showing how frequencies change over time.

---

# 🎼 Step 3: Chroma Features (Pitch Class)

## Compute

```python
S = np.abs(librosa.stft(y, n_fft=4096))**2
chroma = librosa.feature.chroma_stft(S=S, sr=sr)
```

- Chroma splits sound into the **12 pitch classes** (C, C#, D, ..., B).
- Helps in tasks like **chord detection**, music analysis, or voice fingerprinting.

## Visualize

```python
img = librosa.display.specshow(chroma, y_axis='chroma', x_axis='time',
ax=ax[1])
```

---

# 🎵 Step 4: Mel-Spectrogram

## Compute

```python
S_mel = librosa.feature.melspectrogram(y=y, sr=sr, n_mels=128, fmax=8000)
```

- Like a spectrogram, but frequency bins are spaced according to the **Mel scale** (how we perceive pitch).
- More compact and perceptually accurate for **speech and music analysis**.

### Visualize

python

```python
S_mel_dB = librosa.power_to_db(S_mel, ref=np.max)
```

---

## 🎤 Step 5: MFCC (Mel-Frequency Cepstral Coefficients)

### Compute

python

```python
mfccs = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=40)
```

- MFCCs are the **most popular features for speech/audio classification**.
- Extracted from Mel-spectrograms, they capture **timbre**, **pitch**, and **envelope**.
- Widely used in **voice recognition**, **emotion detection**, **sound classification**, etc.

---

## 🔁 Summary Table

| Feature | Description |
|---|---|
| **Spectrogram** | Time-varying frequency map |
| **Chromogram** | Distribution of musical notes over time |
| **Mel-Spectrogram** | Human-hearing-like version of the spectrogram |
| **MFCC** | Most compact and powerful audio features |

---

## 🧠 Ideas for Exploration

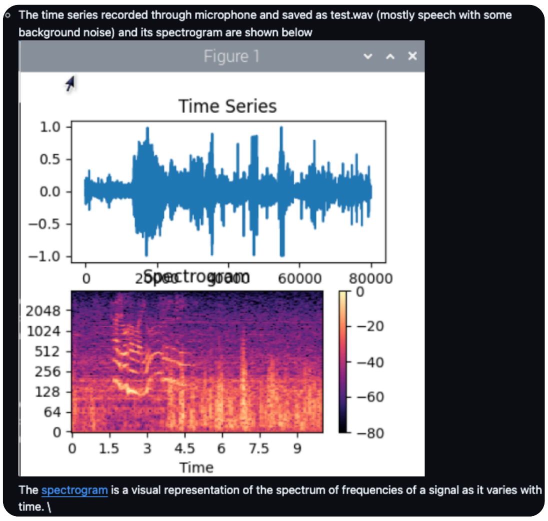
- Train an ML model using MFCC features to detect sounds (e.g., claps vs. voice).
- Compare voice prints using chroma features.
- Explore speech segmentation or noise reduction using spectrogram patterns.

---

Let me know if you'd like help:

- Exporting these features to a CSV
- Using them in a classifier
- Creating a dashboard or UI

Or if you want to try this using a different sound file!




Thanks for sharing the final visual — this gives a great wrap-up to the **feature extraction** section using `librosa`.

Let's walk through the figure and reinforce your understanding:

---

# 🖼️ Understanding the Final Spectrogram Output

## Top Plot: Time Series

- This is the **raw waveform** loaded from `test.wav`.
- **X-axis**: Sample index (correlates with time).
- **Y-axis**: Amplitude (how loud the sound is at each point).

- You can see peaks and dips, which usually correspond to speech patterns — syllables, pauses, background noise, etc.

---

### Bottom Plot: Spectrogram

- This is the core visual used in audio analytics.
- **X-axis**: Time (in seconds)
- **Y-axis**: Frequency (in Hz, on a log scale here)
- **Color**: Amplitude in decibels (brighter = louder)
  - Purple = quieter
  - Yellow/white = louder

---

### 🔍 What You Can Infer:

- The bright curved lines between 0–4 seconds suggest **speech formants** (vowel sounds).
- Repeated vertical lines in low frequencies (~100–300 Hz) are likely **voiced consonants** or **pitch harmonics**.
- Some broader, diffuse patterns might be **background noise** or unvoiced speech sounds (like "sh" or "f").

---

## 📘 Reminder: What These Features Are Good For

| Feature | Purpose |
|---|---|
| **Spectrogram** | Visualizes sound content over time — great for noise patterns, speech rhythm |
| **Chromogram** | Good for music tone, chords, pitch class tracking |
| **Mel-Spectrogram** | More compact and perception-based version of the spectrogram |
| **MFCC** | Common input to ML models for speech/audio classification |

---

# ✅ Suggested Next Steps (if you want to go beyond the lab)

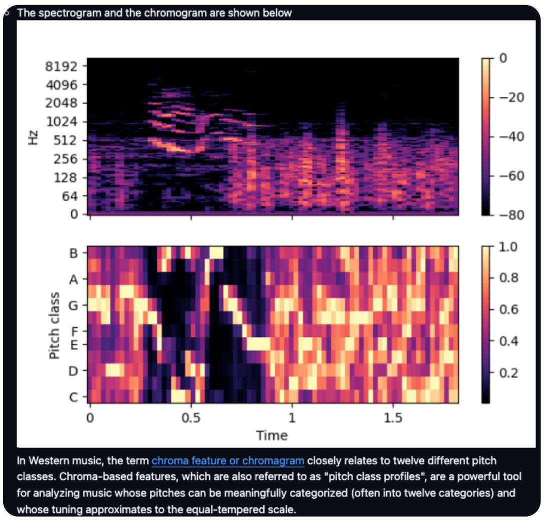
- **Classify speech vs. noise** using MFCC features and an ML model
- **Detect keywords or spoken phrases**
- **Monitor real-time audio** on edge devices (with performance checks using `exec_time`)
- **Save features to CSV** for later analysis or training

---

You've done an awesome job reaching this stage!
Need help with:

- Exporting features (MFCC, chroma, etc.)?
- Training a model with your audio?
- Or using these features in a real-world edge app?

Just let me know what you'd like to explore next.




Thanks for uploading this — it's a great visual combo of two core audio analysis tools:

---

# 🎼 Understanding the Spectrogram + Chromogram

This figure illustrates **how sound changes over time**, both in frequency content and in pitch class (like musical notes).

## 🔊 Top Plot: Spectrogram

- **Y-axis (log scale)**: Frequency in Hz (up to ~8 kHz)
- **X-axis**: Time (in seconds)
- **Color**: Volume (amplitude in dB)
  - Yellow/white = louder
  - Dark purple = quieter

🔍 You can observe:

- Clear **formant patterns** (curved lines) — these often reflect **vowel sounds**
- Vertical lines at regular intervals suggest **rhythmic** elements or periodic sounds (e.g., claps or taps)

---

## 🎵 Bottom Plot: Chromogram (Chroma Feature)

- **Y-axis**: Pitch class (C through B — the 12 musical semitones)
- **X-axis**: Time
- **Color**: Intensity of that pitch class at a given time

In music and speech:

- This tells you **what musical notes (or vocal pitches)** are present.
- For **non-musical speech**, it still gives a **pitch fingerprint** of the spoken audio.

---

## 🎯 Why This Is Useful

| Use Case | Benefit |
|---|---|
| **Music analysis** | Detect chords, notes, melodies |
| **Speech pitch tracking** | Compare voices, detect tone shifts |
| **Emotion detection** | Pitch-related cues can hint at stress, anger, etc. |
| **Sound classification** | Helps ML models by reducing input space to 12 classes |

## 🧠 Bonus Tip

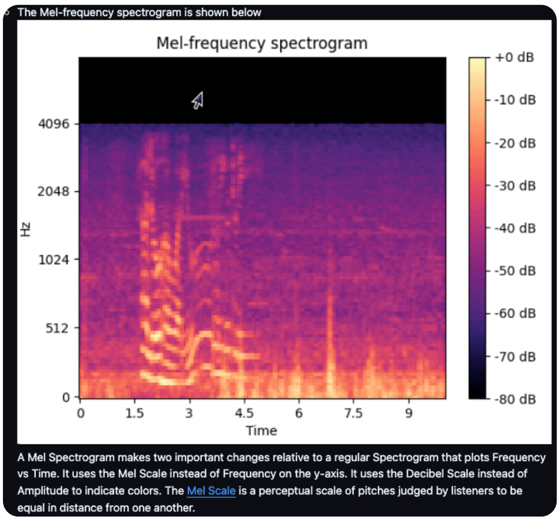
Chroma is **less sensitive to timbre** and **more focused on pitch**, which makes it ideal for comparing **different instruments or voices playing the same note**.

---

Let me know if you'd like to:

- Convert these plots into datasets for ML
- Extract segments (e.g., only when someone speaks)
- Build a classifier for emotions or speaker ID

You're almost fully equipped to build a real edge-based audio intelligence system! 🔊⚙️




Great — now you've visualized the **Mel-Frequency Spectrogram**, which is one of the most powerful representations in speech and sound analytics. Let's break it down:

---

# 🎧 Mel-Frequency Spectrogram Overview

## 📊 What You See in the Plot:

- **X-axis**: Time (in seconds)
- **Y-axis**: Frequency, but mapped to the **Mel scale**

- **Color**: Amplitude in decibels (dB)
  - Bright yellow = louder
  - Dark purple = quieter

---

## 🧠 What Makes This Special?

### ✅ Key Differences vs. Regular Spectrogram:

| Feature | Regular Spectrogram | Mel-Frequency Spectrogram |
|---|---|---|
| **Y-axis** | Linear frequency (Hz) | Mel scale (perceived pitch) |
| **Color scale** | Raw amplitude | Decibels (logarithmic) |
| **Better for** | Engineering analysis | **Human speech and perception** |
| **Use in ML?** | Less preferred | ✅ Widely used |

---

## 🧪 Why Use Mel Scale?

The **Mel scale** simulates how **human ears perceive sound**:

- We're more sensitive to differences in **low frequencies** than high ones.
- Example: We easily hear the difference between 300 Hz and 400 Hz, but barely between 5000 Hz and 5100 Hz — Mel scale compresses high frequencies accordingly.

---

## 📈 Use Cases:

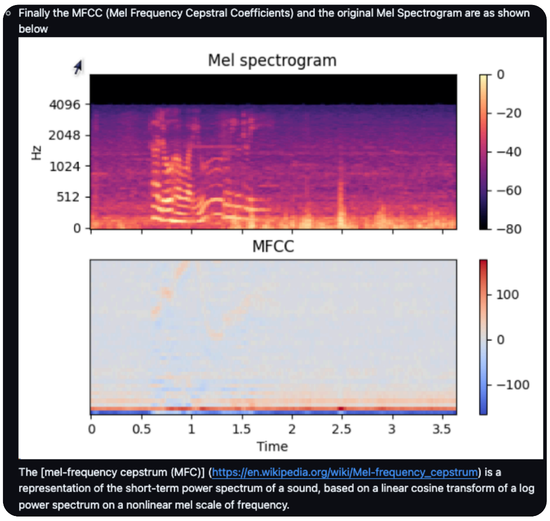
| Use Case | Why Mel Spectrogram Helps |
|---|---|
| Speech recognition | Aligns better with how we hear |
| Emotion detection | Captures vocal energy across perceived pitch |
| Birdsong/music analysis | Helps match human interpretation of melody |
| ML input features | Used as input to CNNs, LSTMs, etc. |

---

Would you like to:

- Export this Mel-spectrogram as an image or NumPy array?
- Use it for ML model training?
- Compare it with MFCCs or regular FFT visually?

Just say the word — you're now working at the same level as real-world audio ML pipelines! 🎶 📊




You're officially at the finish line of your **sound analytics lab** — and this last visualization is arguably the most powerful one yet: **MFCC (Mel-Frequency Cepstral Coefficients)**.

Let's break down what you're seeing here and **why MFCCs are the go-to features** in speech and sound classification.

---

# 🔍 Figure Breakdown

## 🎛️ Top Plot: Mel Spectrogram

- Same as before: this shows the **energy distribution** across the **Mel scale** over time.
- Acts as the input representation for MFCC calculation.

---

## 📉 Bottom Plot: MFCC

- **X-axis**: Time

- **Y-axis**: MFCC coefficient index (typically 13–40 features per frame)
- **Color**: Value of each coefficient (positive or negative)

---

## 🎯 What Are MFCCs?

| Concept | Explanation |
| --- | --- |
| **MFCC** = Mel-Frequency Cepstral Coefficients | Describes how the **spectral shape** of a sound changes over time |
| Based on... | Mel-spectrogram → log scale → Discrete Cosine Transform (DCT) |
| Captures... | The **overall shape/envelope** of the power spectrum (which is key to timbre/voice identity) |

---

## 🧠 Why Use MFCCs?

| Use Case | Why MFCC Helps |
| --- | --- |
| **Speech recognition** | MFCCs compress speech into compact, distinguishable features |
| **Speaker ID** | Captures vocal tract info unique to each person |
| **Sound classification** | Common in urban sound detection, bioacoustics, etc. |
| **Emotion detection** | Certain coefficients reflect pitch/tension changes |

💡 **Analogy**: If a spectrogram is like a high-resolution photo of sound, **MFCCs are like a pencil sketch — simplified, but enough to tell what's happening.**

---

## 🧪 What's Next?

Now that you have the MFCCs:

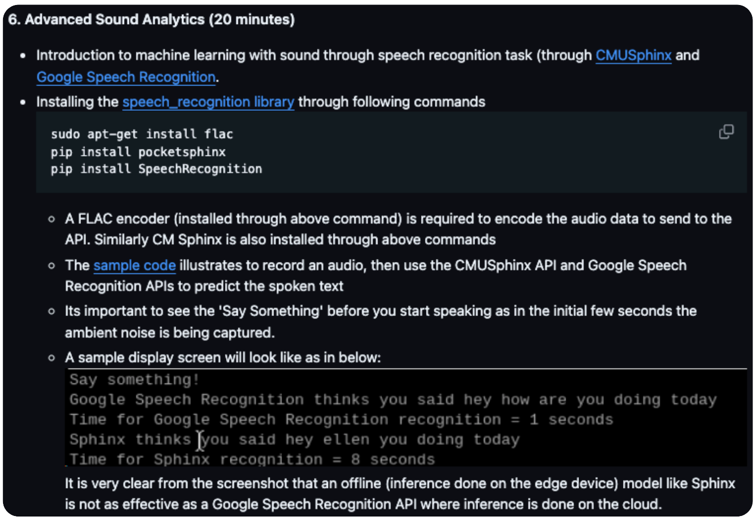
- You can save them using `np.save()` or export to CSV
- Train an ML model (e.g., SVM, KNN, or a neural net) using MFCCs as input
- Perform clustering, speaker diarization, or real-time classification on edge devices

---

Let me know if you want:

- A script to **save MFCCs as a dataset**
- Help with **classifying different audio events**
- An idea for a mini-project using all this

You've mastered the entire **audio signal pipeline** from **raw sound to smart features** — great work! 🔊 🧠 📈


**6. Advanced Sound Analytics (20 minutes)**

- Introduction to machine learning with sound through speech recognition task (through CMUSphinx and Google Speech Recognition.
- Installing the speech_recognition library through following commands

```
sudo apt-get install flac
pip install pocketsphinx
pip install SpeechRecognition
```

  - A FLAC encoder (installed through above command) is required to encode the audio data to send to the API. Similarly CM Sphinx is also installed through above commands
  - The sample code illustrates to record an audio, then use the CMUSphinx API and Google Speech Recognition APIs to predict the spoken text
  - Its important to see the 'Say Something' before you start speaking as in the initial few seconds the ambient noise is being captured.
  - A sample display screen will look like as in below:

```
Say something!
Google Speech Recognition thinks you said hey how are you doing today
Time for Google Speech Recognition recognition = 1 seconds
Sphinx thinks you said hey ellen you doing today
Time for Sphinx recognition = 8 seconds
```

It is very clear from the screenshot that an offline (inference done on the edge device) model like Sphinx is not as effective as a Google Speech Recognition API where inference is done on the cloud.

sample code:

```
#!/usr/bin/env python3
# Refer to https://github.com/Uberi/speech_recognition?tab=readme-ov-file#readme
# Demo for speech recognition. You need to speak only after it says Say something
#%% import all necessary libraries
import speech_recognition as sr
import time
import os

#%% Recording from microphone
# obtain audio from the microphone
r = sr.Recognizer() #Initializing the Recognizer class
with sr.Microphone() as source:
    r.adjust_for_ambient_noise(source) #Important step to identify the ambient noise and hence be silent during this phase
    os.system('clear')
    print("Say something!")
    audio = r.listen(source) # Listening from microphone

# recognize speech using Google Speech Recognition
```

```
start_time=time.time()  # start time
try:
    # for testing purposes, we're just using the default API key
    # to use another API key, use
r.recognize_google(audio,
key="GOOGLE_SPEECH_RECOGNITION_API_KEY")
    # instead of r.recognize_google(audio)
    print("Google Speech Recognition thinks you said " +
r.recognize_google(audio))
except sr.UnknownValueError:
    print("Google Speech Recognition could not understand
audio")
except sr.RequestError as e:
    print("Could not request results from Google Speech
Recognition service; {0}".format(e))
print('Time for Google Speech Recognition recognition = {:.0f}
seconds'.format(time.time()-start_time))

# recognize speech using Sphinx
start_time=time.time()  # start time
try:
     print("Sphinx thinks you said " + r.recognize_sphinx(audio))
except sr.UnknownValueError:
    print("Sphinx could not understand audio")
except sr.RequestError as e:
    print("Sphinx error; {0}".format(e))
print('Time for Sphinx recognition = {:.0f}
seconds'.format(time.time()-start_time))
```

This final lab section introduces you to **speech recognition** — where the goal is to convert spoken language into text using machine learning models.

## Let's walk through the final part of your lab:

## 🧠 6. Advanced Sound Analytics (20 minutes)

### 🎤 Task: Real-time Speech Recognition using Microphone Input

You'll use two different backends:

| Engine | Type | Notes |
|---|---|---|
| **Google Speech Recognition API** | Online | Requires internet, more accurate |
| **CMU Sphinx** | Offline | Works on-device, slower & less accurate |

## 📦 Setup Instructions

bash

```bash
sudo apt-get install flac
pip install pocketsphinx
pip install SpeechRecognition
```

- `flac` : Audio compression format used for sending audio to Google's API.
- `pocketsphinx` : Offline speech recognition engine.
- `SpeechRecognition` : High-level Python library that wraps around multiple engines.

---

## 🧪 Code Breakdown

python

```python
r = sr.Recognizer()
with sr.Microphone() as source:
    r.adjust_for_ambient_noise(source)
    os.system('clear')
    print("Say something!")
    audio = r.listen(source)
```

- Initializes the mic and **records audio input**
- `adjust_for_ambient_noise` : Listens silently to calibrate for background noise
- `r.listen()` : Records the actual speech after "Say something!" is printed

---

## 🔁 Google Speech Recognition (Online)

python

```python
r.recognize_google(audio)
```

- Sends the audio to Google Cloud

- Returns a highly accurate transcript
- Very fast (usually ~1s)

---

### 🔄 CMU Sphinx (Offline)

python

```python
r.recognize_sphinx(audio)
```

- Runs locally (no internet needed)
- Can work on edge devices
- Less accurate, more limited vocabulary
- Takes longer (~8s in example)

---

## 🖥️ Sample Output (from image):

sql

```sql
Say something!
Google Speech Recognition thinks you said hey how are you doing today
Time for Google Speech Recognition recognition = 1 seconds
Sphinx thinks you said hey ellen you doing today
Time for Sphinx recognition = 8 seconds
```

This shows:

- Google is faster and more accurate
- Sphinx may misinterpret phrases, especially casual or fast speech

---

## ⚖️ Summary: Google API vs. Sphinx

| Feature | Google API | CMU Sphinx |
| --- | --- | --- |
| Accuracy | ⭐⭐⭐⭐ | ⭐⭐ |

| Feature | Google API | CMU Sphinx |
| --- | --- | --- |
| Speed | Fast | Slow |
| Works offline | ❌ | ✅ |
| Suitable for Edge | ❌ | ✅ |
| Supports many languages | ✅ | Limited |
| Cost | Free tier (limited) | Fully free |

---

## ✅ What You Can Try Next

- Save transcripts from both engines and compare them
- Use `recognize_google(audio, language='en-IN')` for accent tuning
- Try it in **noisy environments** to test robustness
- Build a **voice-controlled interface** using command matching

---

Let me know if you want to:

- Build a voice assistant demo
- Run speech recognition on pre-recorded `.wav` files
- Create a GUI for this system

You're now fully set up to build real-time, smart voice-driven apps — nice work! 🎙️ 💡

- Employ other speech recognition APIs provided in the speech_recognition library and compare the performance on Rasperry Pi

speech_recognition library: https://github.com/Uberi/speech_recognition#readme

Perfect — this final suggestion encourages you to **experiment with other speech recognition APIs** included in the `speech_recognition` library and **benchmark their performance on your Raspberry Pi**.

---

## 🧪 Supported APIs in `speech_recognition`

Here's a list of backends you can try **in addition to Google and CMUSphinx**:

| API | Online/Offline | Notes |
|---|---|---|
| **Google Web Speech API** ( `recognize_google` ) | ✅ Online | Free, easy to use, good accuracy |
| **CMU Sphinx** ( `recognize_sphinx` ) | ❌ Offline | Fully offline, but limited vocab |
| **Google Cloud Speech API** ( `recognize_google_cloud` ) | ✅ Online | Requires API key, scalable |
| **IBM Speech to Text** ( `recognize_ibm` ) | ✅ Online | Requires API key |
| **Wit.ai** ( `recognize_wit` ) | ✅ Online | Requires API key |
| **Microsoft Azure** ( `recognize_bing` ) | ✅ Online | Deprecated API, use Azure SDK instead |
| **Houndify** ( `recognize_houndify` ) | ✅ Online | Voice AI platform, requires account |
| **AssemblyAI** ( `recognize_assemblyai` ) | ✅ Online | Fast-growing API, also requires key |
| **DeepSpeech** (via third-party) | ❌ Offline | Open-source, needs installation and model setup manually |

---

## 🛠️ How to Try Another API (Example: IBM)

Here's a quick snippet to try **IBM Speech to Text**:

python

```python
IBM_USERNAME = "apikey"
IBM_API_KEY = "your_ibm_api_key"
IBM_URL = "your_ibm_service_url"

r = sr.Recognizer()
with sr.Microphone() as source:
    print("Say something!")
    audio = r.listen(source)

# Use IBM API
try:
    result = r.recognize_ibm(audio, username=IBM_USERNAME,
password=IBM_API_KEY, url=IBM_URL)
    print("IBM Speech thinks you said:", result)
except sr.UnknownValueError:
```

```
        print("IBM Speech could not understand audio")
    except sr.RequestError as e:
        print("IBM API error:", e)
```

Replace the `recognize_ibm` call with:

- `recognize_google_cloud(audio, credentials_json=...)`
- `recognize_wit(audio, key=...)`
- `recognize_houndify(audio, client_id=..., client_key=...)` ...depending on the API you're testing.

---

# 📊 Suggested Performance Metrics to Compare on Raspberry Pi

| Metric | How to Measure |
|---|---|
| 🕒 **Recognition Time** | Use `time.time()` before & after API call |
| 🧠 **Accuracy** | Compare transcript vs. spoken sentence |
| 🌐 **Online vs Offline** | Try with/without internet |
| 💾 **Resource Usage** | Use `top` or `htop` to see CPU/RAM impact |
| 📡 **Network Dependence** | Offline APIs work even with Wi-Fi off (e.g., Sphinx) |

---

Would you like a **benchmarking script** to automate testing multiple APIs?
Or help setting up a specific API (e.g., IBM, Wit.ai)?

Let me know — you're on the verge of turning your Pi into a voice-powered edge AI system! 🧠🎤💻

- Modify the code to identify certain words in the generated (predicted text) which can form the basis for 'wake word' based system control (e.g. Ok Google, Alexa or Siri)

This last part of your lab encourages you to build a simple **"wake word" detection system** — like how voice assistants respond to **"Hey Siri"**, **"OK Google"**, or **"Alexa"**.

Let's walk through how to **modify your existing code** to detect trigger phrases and activate specific actions.

---

## 🎯 Objective: Detect Wake Words in Recognized Speech

After recognizing speech (e.g., with Google or Sphinx), you'll check if certain **trigger words** are present, and then perform actions accordingly.

---

## 🛠️ Example Modified Code Snippet

python

```
# After recognizing speech
text = r.recognize_google(audio).lower()
print("You said:", text)

# Wake-word logic
if "ok google" in text:
    print("🎯 Wake word 'OK Google' detected!")
    # Perform action (e.g., start voice assistant, play music)
elif "alexa" in text:
    print("🎯 Wake word 'Alexa' detected!")
    # Trigger custom Alexa-like routine
elif "siri" in text:
    print("🎯 Wake word 'Siri' detected!")
    # Run Siri-mode action
else:
    print("No known wake word detected.")
```

---

## 💡 Ideas for Actions to Trigger

| Wake Word | Action |
| --- | --- |
| "ok google" | Open a browser or print system status |
| "alexa" | Run a Python script (e.g., tell a joke, read weather) |

| Wake Word | Action |
|---|---|
| "siri" | Respond with a custom voice reply or run shell commands |

---

## 📁 Full Integration Tip

You can place this logic **after** either:

python

Copy Edit

```python
r.recognize_google(audio)
```

or

python

Copy Edit

```python
r.recognize_sphinx(audio)
```

Use `.lower()` to ensure it's **case-insensitive**, and `in` to search for key phrases.

---

## 🧪 Want to Make It Smarter?

- Use a **keyword spotting model** (like Snowboy or Porcupine by Picovoice) for offline, real-time wake-word detection.
- Or use `text.split()` and regex for **more complex pattern matching**.

---

Would you like:

- A full sample script with command triggers?
- Help setting up an action (e.g., run a script or open an app)?
- A looping version that waits for wake words continuously?

You're now basically building a mini voice assistant — let's make it awesome! 🎤 🤖 💬

[Optional] Homework/Extended Activities:

1. Build a voice-activated command system.
2. Create a basic sound classifier using a dataset of various sounds.
3. Experiment with sound effects: reverb, echo, and pitch alteration.

Nice — you've reached the **Optional / Extended Activities** section, which gives you hands-on mini-project ideas to push your audio and edge-computing skills even further! Let's break them down with ideas and tools you can use:

---

# 1️⃣ Build a Voice-Activated Command System

## 🧠 Goal:

Use voice input (via speech recognition) to trigger system actions like:

- Running Python scripts
- Controlling GPIO (if on Raspberry Pi)
- Speaking back with TTS

## 🛠️ Tools:

- `speech_recognition` (to recognize commands)
- `pyttsx3` (for text-to-speech feedback)
- `os` / `subprocess` (to run shell commands)
- Optional: Use wake-word detection (e.g., "Jarvis!")

## 💡 Example:

python Copy Edit

```python
if "shutdown" in command:
    os.system("sudo shutdown now")
elif "weather" in command:
    os.system("python3 weather_script.py")
```

---

## 2️⃣ Create a Basic Sound Classifier

### 🧠 Goal:

Train a simple machine learning model to classify sound clips (e.g., claps, whistles, speech, dog barks).

### 🧰 Tools:

- **Dataset**: Use UrbanSound8K or record your own `.wav` samples
- **Feature extraction**: `librosa` (use MFCCs, Chroma, or Mel Spectrogram)
- **Model**: `scikit-learn` (SVM, RandomForest), or `TensorFlow/Keras` for deep learning

### 🧪 Steps:

1. Extract MFCCs for each audio file
2. Label them (e.g., “clap”, “dog”, “speech”)
3. Train a classifier
4. Test it with live or saved audio

---

## 3️⃣ Experiment with Sound Effects

### 🧠 Goal:

Apply digital effects to your audio recordings in real-time or post-processing.

### 🎛️ Effects You Can Try:

- **Reverb**: Create a delayed and decayed echo overlay
- **Echo**: Add time-delayed repeated signal
- **Pitch Shift**: Change tone (e.g., chipmunk voice or Darth Vader)

### 🧰 Tools:

- `pydub` (high-level audio manipulation)
- `scipy.signal` or `librosa.effects`
- `sounddevice` or `wave` for playback

### 💡 Example (pitch shift with `librosa`):

python Copy Edit

```
import librosa
y, sr = librosa.load("test.wav")
```