AUDIO ENGINEERING & ACOUSTICS

EE 599

JULY 9, 2024

E/18/334
Narthana Sivalingam

Abstract

This course offers an extensive examination of audio engineering, merging traditional acoustic techniques with advanced machine learning applications. Organized into six detailed modules, it spans topics from the foundational aspects of sound to sophisticated audio analysis, designed to cater to a broad audience ranging from beginners in audio engineering to seasoned professionals keen on integrating artificial intelligence into their practices.

Module 1: Basic Acoustics and Sound Theory begins with essential sound properties—frequency, amplitude, wavelength, and velocity—and extends into human auditory perception. It elaborates on the audio signal flow, comparing analog and digital signals, and explores essential signal processing operations.

Module 2: Synthesis Techniques advances from theoretical underpinnings to practical applications, focusing on synthesizer technologies. It delves into linear and non-linear synthesis, discussing basic waveforms and modulation techniques (FM, AM, RM), and their practical implementations in modern music production.

Module 3: Audio Signal Processing tackles advanced techniques like pitch estimation and manipulation with methods such as Linear Predictive Coding (LPC) and pitch scaling/shifting. It also explores vocal processing techniques including phase cancellation and spectral editing to isolate vocals from music tracks.

Module 4: Advanced Audio Analysis explores high-level methods like subspace filtering for noise reduction and signal enhancement, alongside the physics of reverberation and its digital simulation in audio workstations to enhance audio depth and spatial quality.

Module 5: Audio Data Visualization and Analysis introduces machine learning with techniques such as scalograms and advanced spectral analysis tools. This module emphasizes genre detection and classification, highlighting feature extraction and visualization techniques that aid in understanding complex audio data.

Module 6: Modern Audio Engineering with Machine Learning fully integrates machine learning with audio processing. Covering basic ML concepts and their applications in audio, it delves into specific ML techniques for audio analysis. It concludes with practical projects like designing a genre classification model and creating a speech recognition system.

For comprehensive access to all course materials, including in-depth reports, code snippets, and output demonstrations for each section, participants are encouraged to visit the dedicated **GitHub repository** at https://github.com/Narthanasiva/EE599-Audio-Engineering-E18334. This repository is a vital resource, continuously updated to reflect the latest in audio engineering and machine learning advancements.

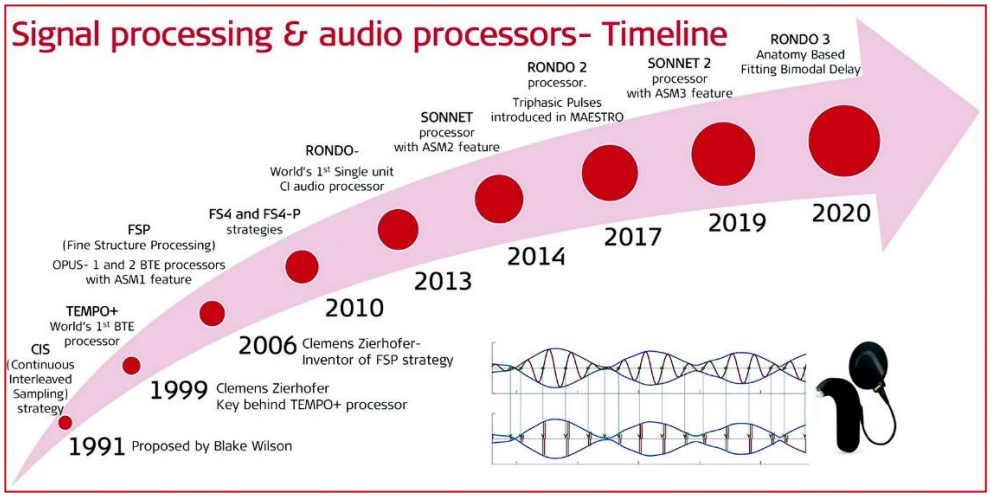
Introduction to Audio Engineering and Acoustics

GitHub Link : https://github.com/Narthanasiva/EE599-Audio-Engineering-E18334

Introduction to Audio Engineering and Acoustics

➢ What is Audio Engineering?

- **Definition of Audio Engineering**: Exploring how sound is recorded, mixed, and produced.
- **Key Components**:
    - **Recording**: Capturing sounds using microphones and other devices.
    - **Editing**: Adjusting sounds to improve quality or alter characteristics.
    - **Mixing**: Combining different sounds to create a final track.
    - **Mastering**: Enhancing the overall sound for final output.



➢ The Role of Acoustics in Audio Engineering

- **Understanding Acoustics**: The science of how sound is produced, controlled, transmitted, and received.
- **Importance in Audio Engineering**:
    - **Sound Quality**: Improving the clarity and detail of the sound.
    - **Sound Control**: Managing how sound behaves in different environments, like studios or concert halls.
    - **Equipment Design**: Developing tools that capture and reproduce sound accurately.

➢ Overview of the Course Structure

Module 1: Basic Acoustics and Sound Theory

- **Fundamentals of Sound**
  - Sound Waves: Frequency, Amplitude, Wavelength, and Velocity
  - The Human Hearing Range and Perception of Sound
- **Audio Signal Flow**
  - Analog vs. Digital Signals
  - Signal Chain and Signal Processing

Module 2: Synthesis Techniques

- **Linear Synthesizer**
  - Basic Waveforms: Sine, Square, Triangle, and Sawtooth
  - ADSR Envelope: Attack, Decay, Sustain, Release
- **Non-linear Synthesizer**
  - Introduction to Modulation Techniques: FM, AM, and RM
  - Practical Uses of Non-linear Synthesis in Modern Music Production

Module 3: Audio Signal Processing

- **Pitch Estimation and Manipulation**
  - Linear Predictive Coding (LPC): Theory and Application
  - Pitch Scaling and Shifting: Techniques and Tools
- **Vocal Processing**
  - Vocal Removal Techniques: Phase Cancellation, Spectral Editing
  - Temporal Separation: Differentiating Between Similar Sounds

Module 4: Advanced Audio Analysis

- **Subspace Filtering**
  - Theory of Subspace Methods
  - Applications in Noise Reduction and Signal Enhancement
- **Reverberation and Spatial Effects**
  - Physics of Reverberation
  - Simulating Reverb in Digital Audio Workstations

Module 5: Audio Data Visualization and Analysis

- **Visualization Techniques**
  - Scalogram: Understanding Wavelet Transforms

    - Advanced Visualization Techniques for Audio Analysis
- **Genre Detection and Classification**
    - Feature Extraction for Genre Classification
    - Practical Applications of Genre Visualization

Module 6: Modern Audio Engineering with Machine Learning

- **Introduction to Machine Learning in Audio**
    - Basics of Machine Learning
    - Applications in Audio Engineering and Acoustics
- **Machine Learning Techniques for Audio Analysis**
    - Supervised and Unsupervised Learning Models
    - Neural Networks and Deep Learning in Audio Classification
- **Practical Machine Learning Projects**
    - Designing a Genre Classification Model
    - Creating a Speech Recognition System

Module 1: Basic Acoustics and Sound Theory

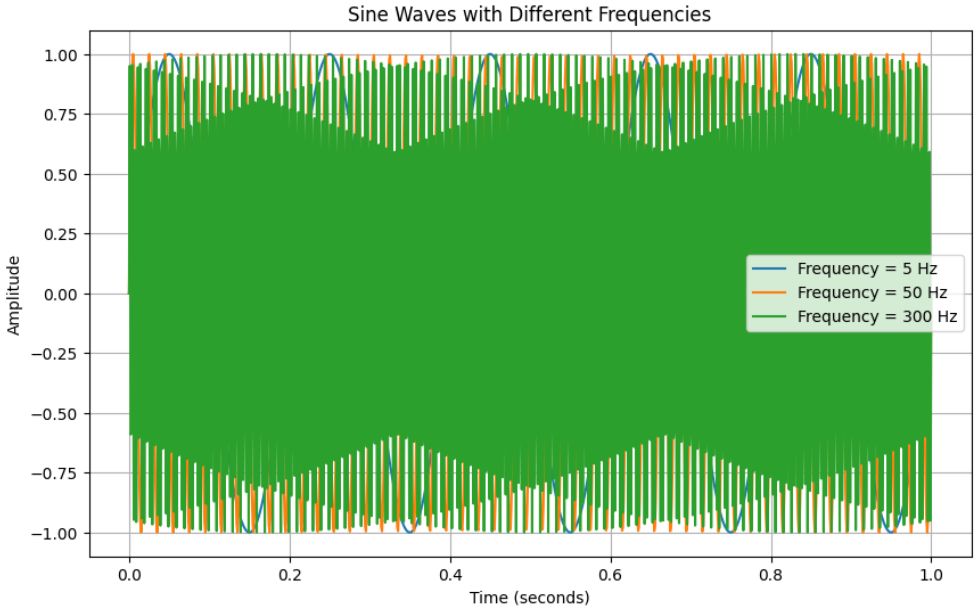
This module is foundational for students studying audio engineering, focusing on the basic properties of sound and how it is manipulated and processed both in analog and digital formats. Here, we'll delve into the fundamentals of sound, covering the nature of sound waves, human hearing, and the basics of audio signal flow.

➢ Fundamentals of Sound

1. **Sound Waves: Frequency, Amplitude, Wavelength, and Velocity**

   - **Sound Waves**: Vibrations that travel through the air or another medium and can be heard when they reach a person's or animal's ear.
   - **Frequency**: The number of complete wave cycles per second, measured in Hertz (Hz). It determines the pitch of the sound.
   - **Amplitude**: The height of the wave, which determines the loudness or volume.
   - **Wavelength**: The distance between corresponding points of two consecutive waves.
   - **Velocity**: The speed at which the sound wave travels through a medium, typically about 343 meters per second in air at room temperature.

2. **Visual Representation of Sound Waves**
   To illustrate these concepts, we can use Python to plot a simple sine wave, which is a basic type of sound wave.



```
import numpy as np
import matplotlib.pyplot as plt

# Parameters for different waves
frequencies = [5, 10, 20]  # frequencies of the waves in Hz
amplitude = 1  # amplitude of the waves
time = np.linspace(0, 1, 1000)  # time vector

# Create sine waves with different frequencies
waves = [amplitude * np.sin(2 * np.pi * f * time) for f in frequencies]

# Plotting the waves
plt.figure(figsize=(10, 6))
for i, wave in enumerate(waves):
    plt.plot(time, wave, label=f'Frequency = {frequencies[i]} Hz')

plt.title('Sine Waves with Different Frequencies')
plt.xlabel('Time (seconds)')
plt.ylabel('Amplitude')
plt.grid(True)
plt.legend()
plt.show()
```

3. **The Human Hearing Range and Perception of Sound**

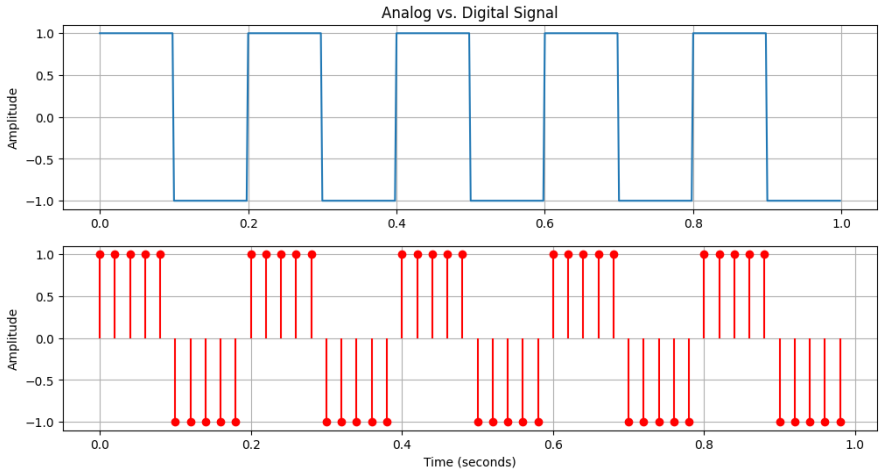
Humans can typically hear frequencies from about 20 Hz to 20,000 Hz. The sensitivity to different frequencies varies by age and individual. Sounds outside this range exist (infrasound and ultrasound) but are not audible to humans.

4. **Audio Signal Flow**

Understanding how audio signals flow from source to output is crucial in audio engineering.

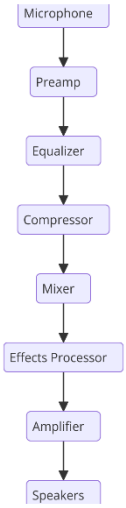
- **Analog vs. Digital Signals**
  - **Analog Signals**: Continuous signals that vary over time and are susceptible to degradation and noise.
  - **Digital Signals**: Represent signals in discrete values (0s and 1s), offering easier

```
from scipy.signal import square

# Generate a square wave (analog)
t = np.linspace(0, 1, 500, endpoint=False)
analog_signal = square(2 * np.pi * 5 * t)

# Sampling the analog signal to convert it to a digital signal
sampling_rate = 50  # 50 samples per second
samples = np.arange(0, 500, 500 // sampling_rate)
digital_signal = analog_signal[samples]

# Plotting both signals
plt.figure(figsize=(12, 6))
plt.subplot(211)
plt.plot(t, analog_signal, label='Analog Signal')
plt.title('Analog vs. Digital Signal')
plt.ylabel('Amplitude')
plt.grid(True)

plt.subplot(212)
plt.stem(samples / 500, digital_signal, 'r', markerfmt='ro', basefmt=" ",
use_line_collection=True, label='Digital Signal')
plt.xlabel('Time (seconds)')
plt.ylabel('Amplitude')
plt.grid(True)
plt.show()
```

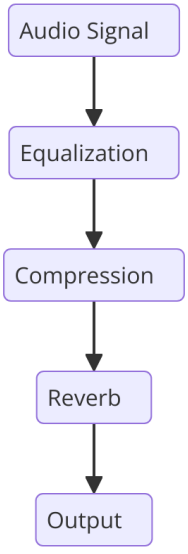


5. **Signal Chain and Signal Processing**

- **Signal Chain**: The path an audio signal takes from the source (microphone) through various processing stages (mixer, amplifier) to the output (speakers).
- **Signal Processing**: The manipulation of these signals to improve quality or alter them creatively. This can include equalization, compression, and reverb.



Signal Chain



Signal Processing

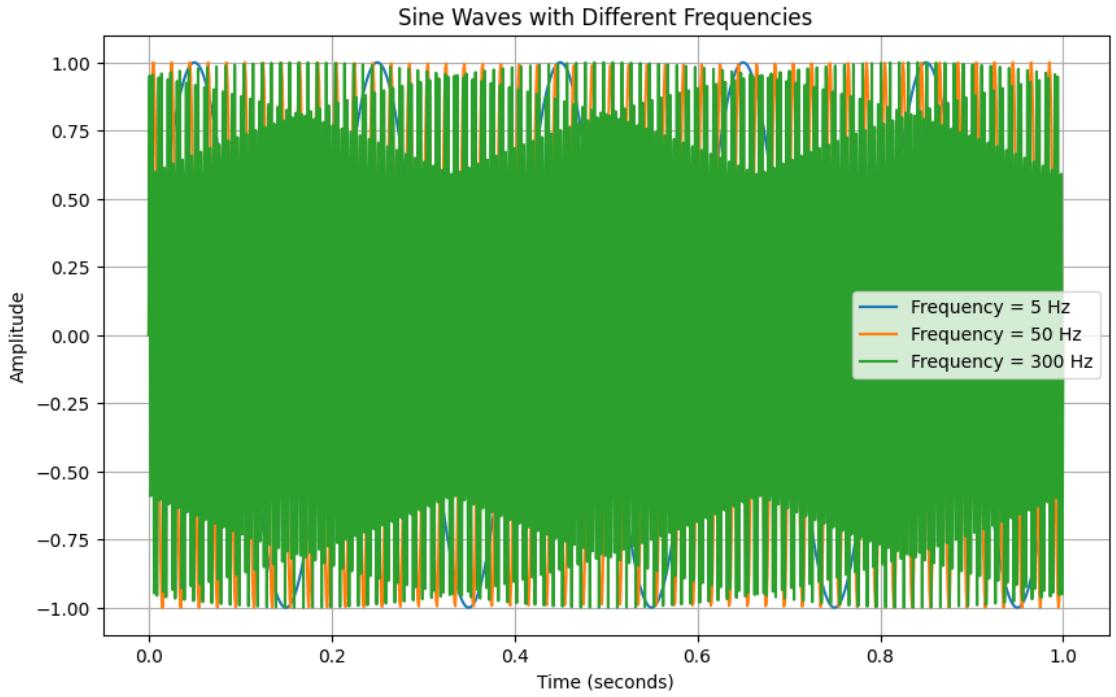
1-basic-acoustics-and-sound-theory

September 7, 2024

1 Visual Representation of Sound Waves

- The output displays three sine waves with frequencies of 5 Hz, 50 Hz, and 300 Hz plotted on the same graph.
- Each wave has the same amplitude but different frequencies, resulting in varying numbers of cycles over one second.
- The higher the frequency, the more wave cycles appear within the time span, illustrating how sound pitch changes with frequency.

[12]:
```
import numpy as np
import matplotlib.pyplot as plt

# Parameters for different waves
frequencies = [5, 50, 300]  # frequencies of the waves in Hz
amplitude = 1  # amplitude of the waves
time = np.linspace(0, 1, 1000)  # time vector

# Create sine waves with different frequencies
waves = [amplitude * np.sin(2 * np.pi * f * time) for f in frequencies]

# Plotting the waves
plt.figure(figsize=(10, 6))
for i, wave in enumerate(waves):
    plt.plot(time, wave, label=f'Frequency = {frequencies[i]} Hz')

plt.title('Sine Waves with Different Frequencies')
plt.xlabel('Time (seconds)')
plt.ylabel('Amplitude')
plt.grid(True)
plt.legend()
plt.show()
```

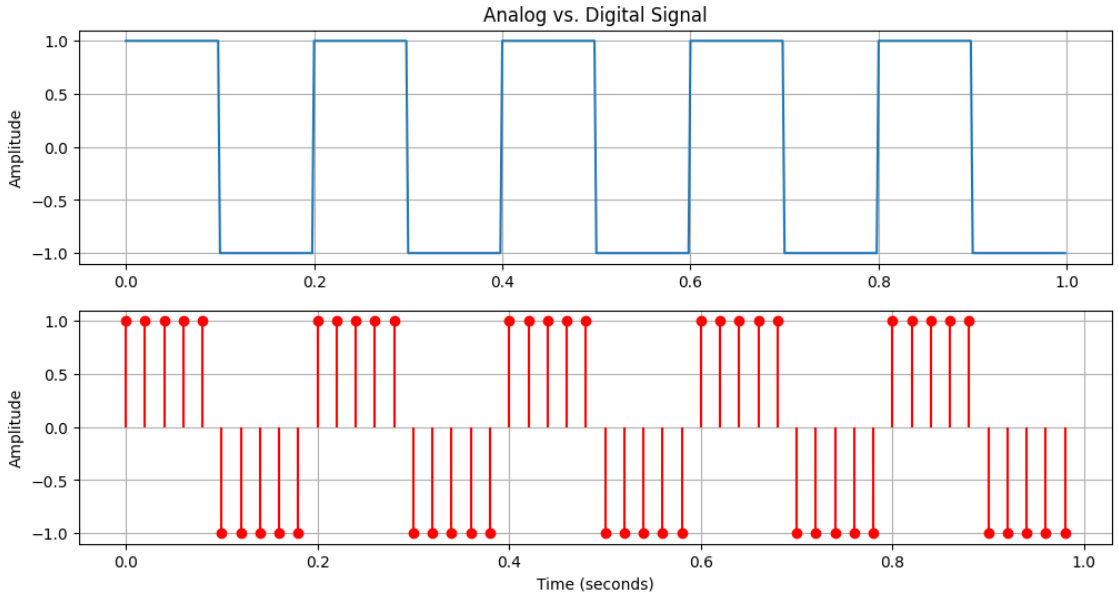


2 Example of Digitalization of an Analog Signal

[13]:
```
from scipy.signal import square

# Generate a square wave (analog)
t = np.linspace(0, 1, 500, endpoint=False)
analog_signal = square(2 * np.pi * 5 * t)

# Sampling the analog signal to convert it to a digital signal
sampling_rate = 50  # 50 samples per second
samples = np.arange(0, 500, 500 // sampling_rate)
digital_signal = analog_signal[samples]

# Plotting both signals
plt.figure(figsize=(12, 6))
plt.subplot(211)
plt.plot(t, analog_signal, label='Analog Signal')
plt.title('Analog vs. Digital Signal')
plt.ylabel('Amplitude')
plt.grid(True)

plt.subplot(212)
```

```
plt.stem(samples / 500, digital_signal, 'r', markerfmt='ro', basefmt=" ",␣
 ↪use_line_collection=True, label='Digital Signal')
plt.xlabel('Time (seconds)')
plt.ylabel('Amplitude')
plt.grid(True)
plt.show()
```

```
<ipython-input-13-adb2c5cab62a>:21: MatplotlibDeprecationWarning: The
'use_line_collection' parameter of stem() was deprecated in Matplotlib 3.6 and
will be removed two minor releases later. If any parameter follows
'use_line_collection', they should be passed as keyword, not positionally.
  plt.stem(samples / 500, digital_signal, 'r', markerfmt='ro', basefmt=" ",
use_line_collection=True, label='Digital Signal')
```



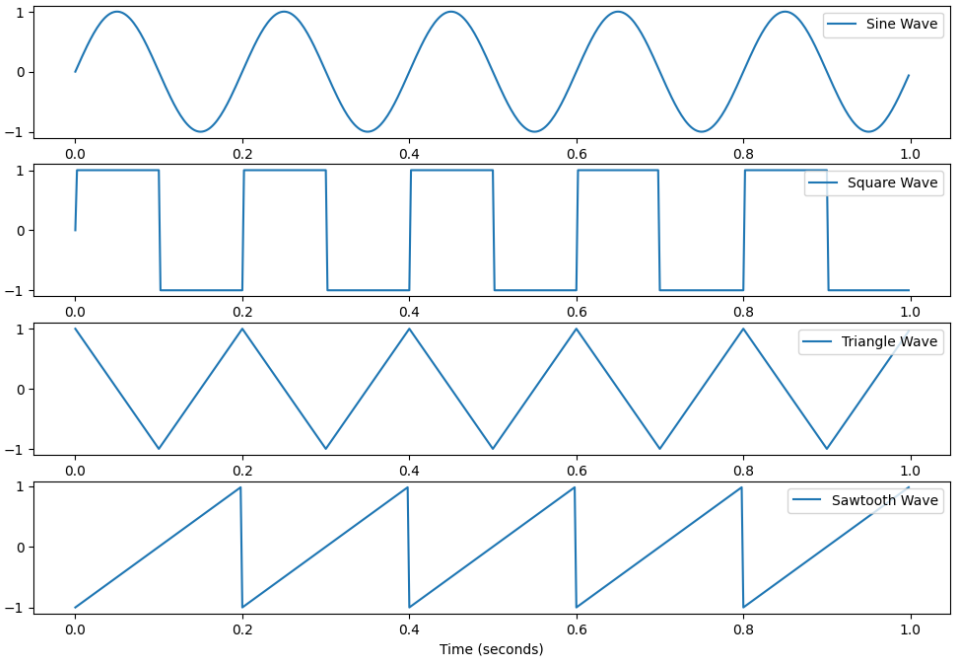
Module 2: Synthesis Techniques

This module explores the core concepts of sound synthesis, a fundamental aspect of audio engineering. Students will learn about the different types of synthesizers, basic waveforms, and envelope controls, along with an introduction to modulation techniques and their applications in modern music production.

➢ Linear Synthesizer

1. **Basic Waveforms**
    - **Sine Wave**: Purest form of waveform, fundamental to other sounds. It has a smooth, periodic oscillation.
    - **Square Wave**: Has a rich, hollow tone; contains odd harmonics. It switches abruptly between high and low amplitude.
    - **Triangle Wave**: Similar to the sine wave but with a linear rise and fall that gives it a sharper tone. Contains only odd harmonics.
    - **Sawtooth Wave**: Contains both odd and even harmonics; has a buzzy quality. It rises linearly but drops abruptly.

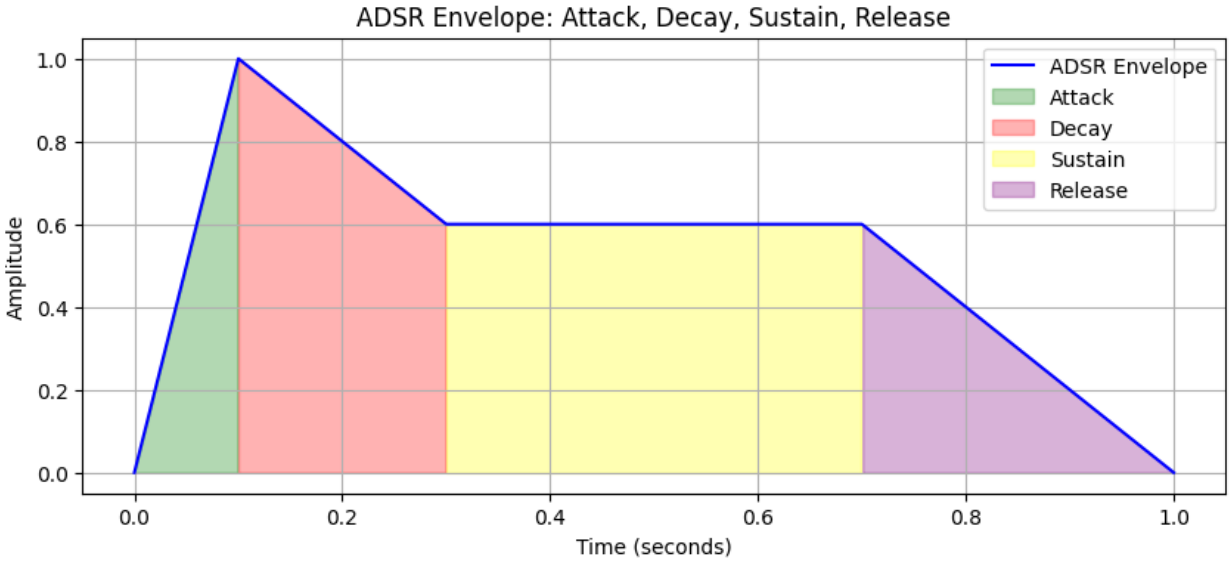
Visualizing Basic Waveforms with Python



2. **ADSR Envelope: Attack, Decay, Sustain, Release**
    - **Attack**: How quickly the sound reaches its maximum level after being triggered.
    - **Decay**: How quickly the sound drops to the sustain level after the initial peak.
    - **Sustain**: The level during the main sequence of the sound's duration, until the key is released.
    - **Release**: How quickly the sound fades when the key is released.

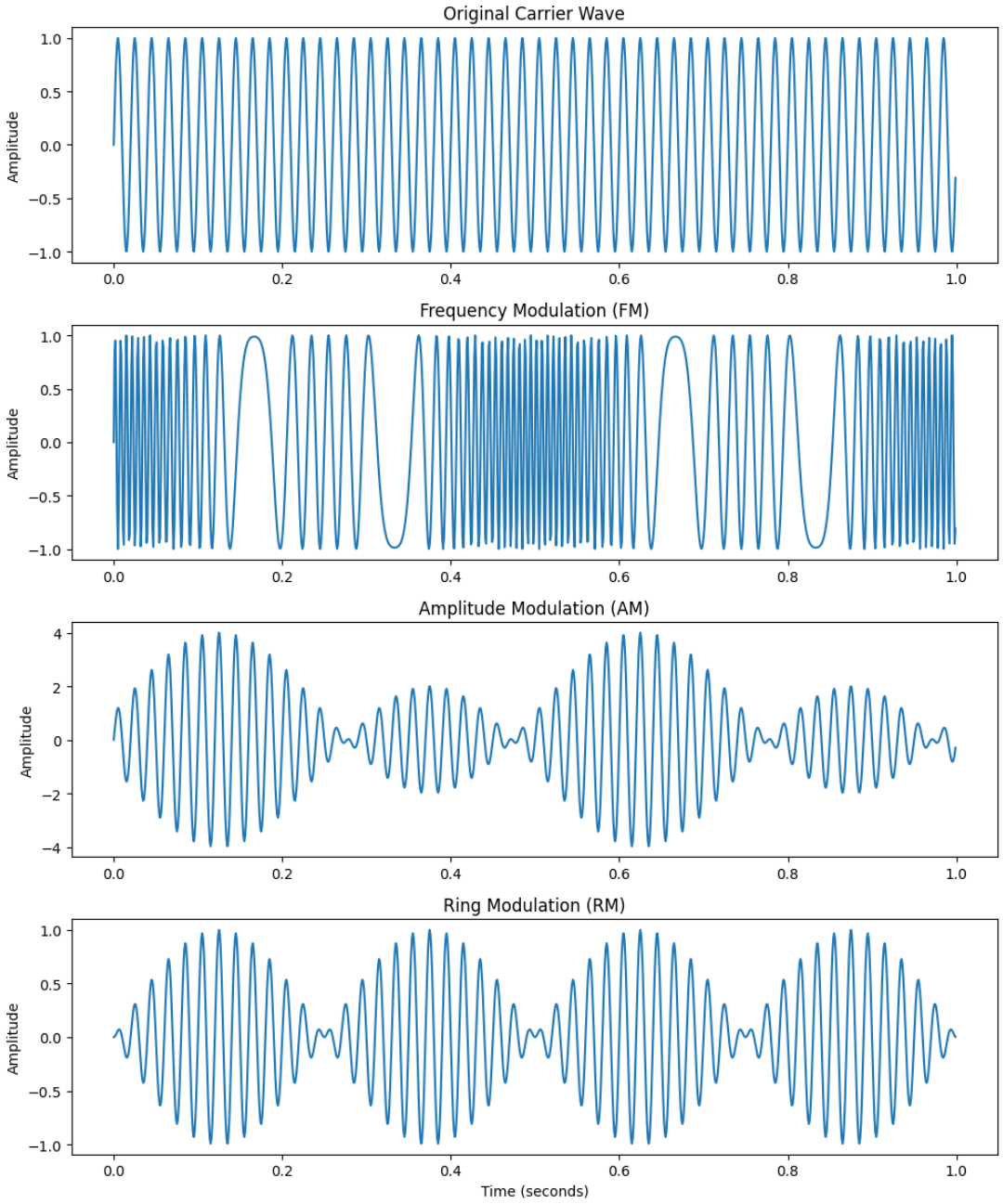
In the graph below, each phase of the ADSR (Attack, Decay, Sustain, Release) envelope is clearly marked with different colors, making it easy to identify:

- **Attack (Green)**: The quick rise at the beginning where the sound reaches its peak. This phase is short and steep.
- **Decay (Red)**: Immediately following the attack, the amplitude decreases to the sustain level. It's a relatively swift drop compared to the attack.
- **Sustain (Yellow)**: This level phase maintains a constant amplitude as long as the note is held. It is visually represented by a flat, extended line.
- **Release (Purple)**: The final phase where the amplitude fades back to zero after the note is released. This tail-off is also quick and concludes the sound's envelope.



➢ Non-linear Synthesizer

1. **Introduction to Modulation Techniques**
    - **Frequency Modulation (FM)**: Modulating the frequency of a waveform with another waveform, resulting in complex harmonic content.
    - **Amplitude Modulation (AM)**: Modulating the amplitude of the carrier wave with another waveform, used to create tremolo effects.
    - **Ring Modulation (RM)**: A type of amplitude modulation but with the modulating signal subtracted as well as added to the carrier.



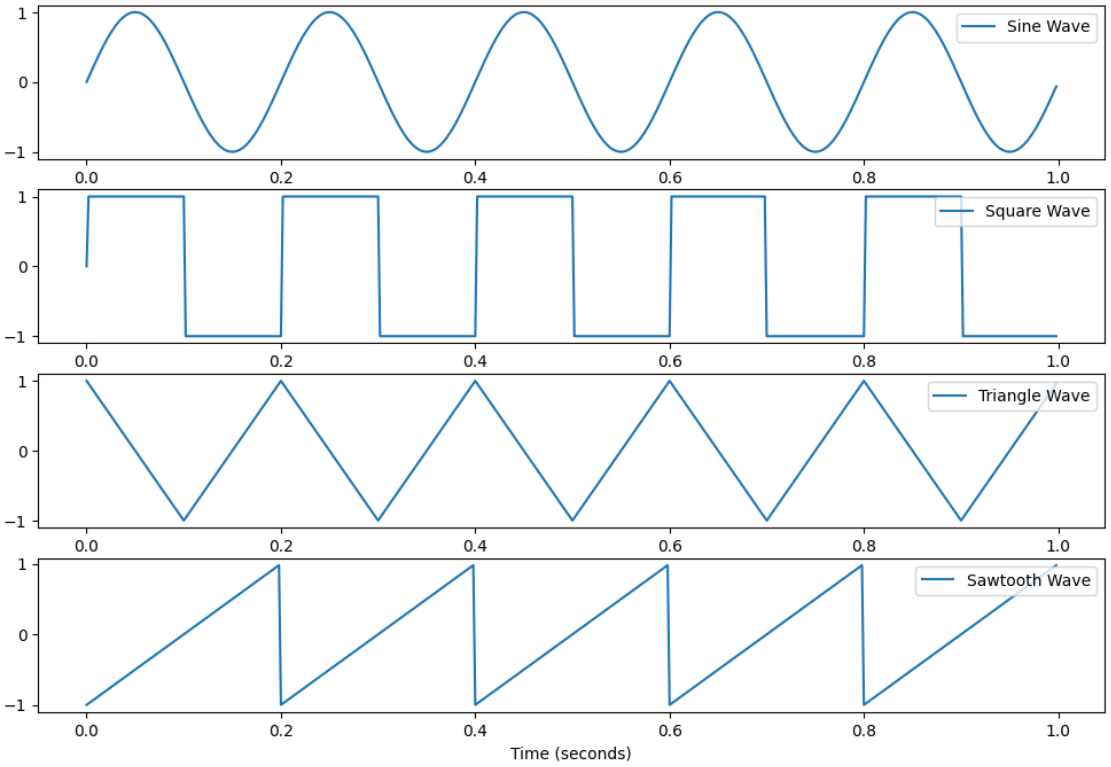
2. **Practical Uses of Non-linear Synthesis in Modern Music Production**
    - **Texture and Depth**: Non-linear synthesizers can add rich textures and depth to music, making it more vibrant and expressive.
    - **Sound Design**: Widely used in sound design to create unique sounds for movies, games, and electronic music.
    - **Dynamic Effects**: Employing modulation techniques can dynamically alter the timbre of sounds during a performance or recording.

02-module-2-synthesis-techniques

September 7, 2024

1 Visualizing Basic Waveforms of Linear Synthesiser with Python

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

# Time vector
t = np.linspace(0, 1, 500, endpoint=False)

# Generating basic waveforms
sine_wave = np.sin(2 * np.pi * 5 * t)
square_wave = np.sign(np.sin(2 * np.pi * 5 * t))
triangle_wave = 2 * np.abs(2 * (t * 5 % 1) - 1) - 1
sawtooth_wave = 2 * (t * 5 % 1) - 1

# Plotting the waveforms
plt.figure(figsize=(12, 8))
plt.subplot(411)
plt.plot(t, sine_wave, label='Sine Wave')
plt.legend(loc='upper right')

plt.subplot(412)
plt.plot(t, square_wave, label='Square Wave')
plt.legend(loc='upper right')

plt.subplot(413)
plt.plot(t, triangle_wave, label='Triangle Wave')
plt.legend(loc='upper right')

plt.subplot(414)
plt.plot(t, sawtooth_wave, label='Sawtooth Wave')
plt.legend(loc='upper right')

plt.xlabel('Time (seconds)')
plt.show()
```

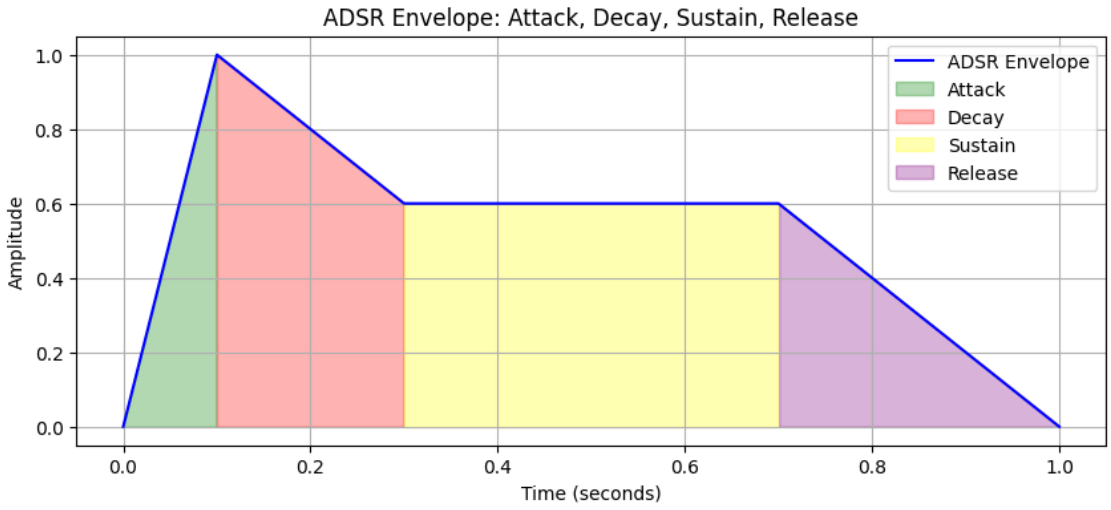


2 In the graph, each phase of the ADSR (Attack, Decay, Sustain, Release) envelope is clearly marked with different colors, making it easy to identify:

- Attack (Green): The quick rise at the beginning where the sound reaches its peak. This phase is short and steep.
- Decay (Red): Immediately following the attack, the amplitude decreases to the sustain level. It's a relatively swift drop compared to the attack.
- Sustain (Yellow): This level phase maintains a constant amplitude as long as the note is held. It is visually represented by a flat, extended line.
- Release (Purple): The final phase where the amplitude fades back to zero after the note is released. This tail-off is also quick and concludes the sound's envelope.

[2]:
```
import numpy as np
import matplotlib.pyplot as plt

# Time vector for the entire ADSR cycle
t = np.linspace(0, 1, 1000)

# ADSR parameters
attack_time = 0.1
```

```
decay_time = 0.2
sustain_level = 0.6
release_time = 0.3
total_duration = 1  # Total duration of 1 second

# Generate ADSR envelope
adsr_envelope = np.zeros_like(t)

# Attack phase
attack_indices = t < attack_time
adsr_envelope[attack_indices] = (t[attack_indices] / attack_time)

# Decay phase
decay_indices = (t >= attack_time) & (t < attack_time + decay_time)
adsr_envelope[decay_indices] = ((1 - sustain_level) * (1 - (t[decay_indices] -␣
 ↪attack_time) / decay_time)) + sustain_level

# Sustain phase
sustain_indices = (t >= attack_time + decay_time) & (t < total_duration -␣
 ↪release_time)
adsr_envelope[sustain_indices] = sustain_level

# Release phase
release_indices = t >= total_duration - release_time
adsr_envelope[release_indices] = sustain_level * (1 - (t[release_indices] -␣
 ↪(total_duration - release_time)) / release_time)

# Plotting the ADSR envelope
plt.figure(figsize=(10, 4))
plt.plot(t, adsr_envelope, label='ADSR Envelope', color='blue')
plt.fill_between(t, 0, adsr_envelope, where=attack_indices, color='green',␣
 ↪alpha=0.3, label='Attack')
plt.fill_between(t, 0, adsr_envelope, where=decay_indices, color='red', alpha=0.
 ↪3, label='Decay')
plt.fill_between(t, 0, adsr_envelope, where=sustain_indices, color='yellow',␣
 ↪alpha=0.3, label='Sustain')
plt.fill_between(t, 0, adsr_envelope, where=release_indices, color='purple',␣
 ↪alpha=0.3, label='Release')
plt.title('ADSR Envelope: Attack, Decay, Sustain, Release')
plt.xlabel('Time (seconds)')
plt.ylabel('Amplitude')
plt.grid(True)
plt.legend()
plt.show()
```

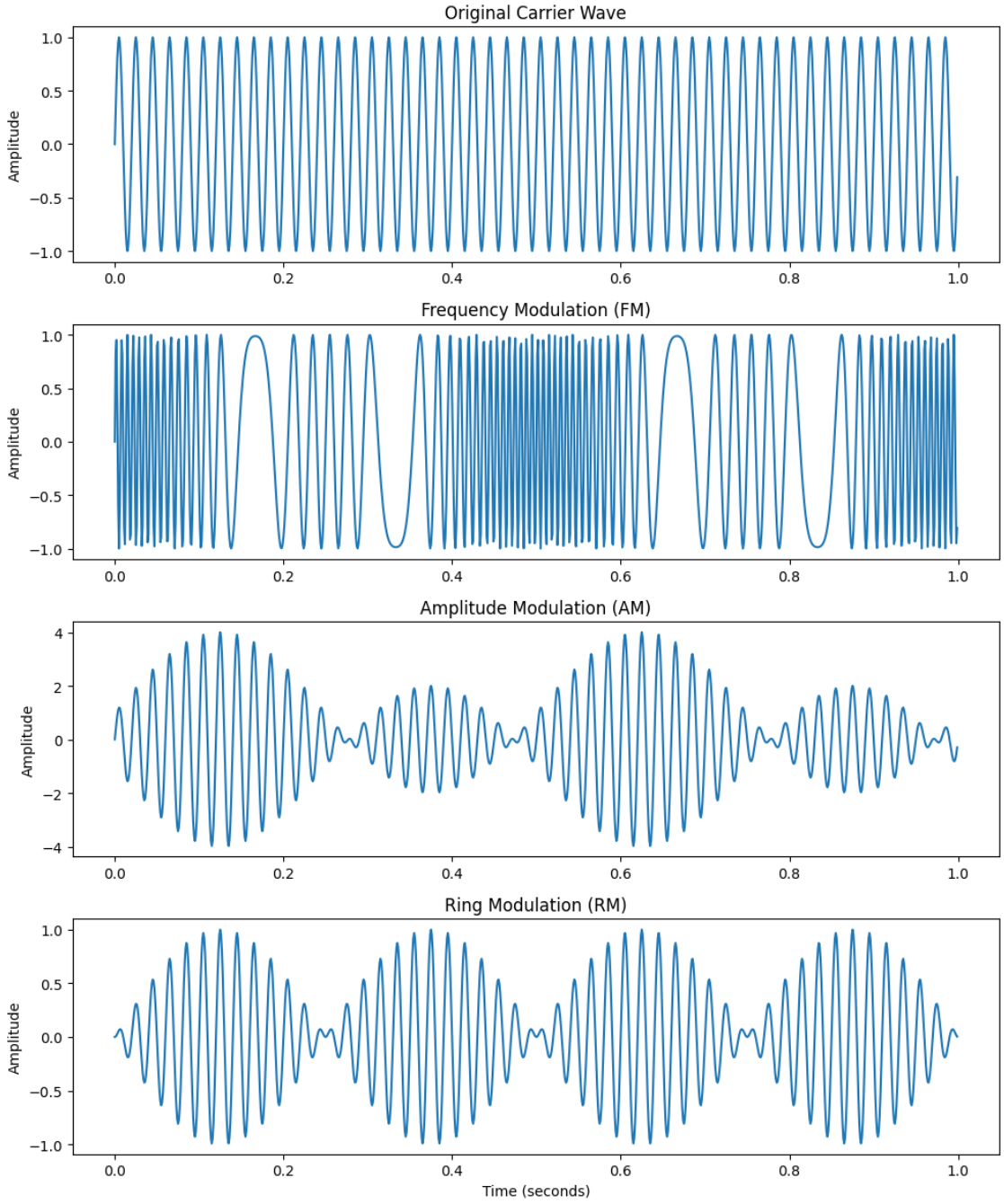


3 Non Linear Synthesisers

- Original Carrier Wave: This is the base sine wave at 50 Hz, used as the starting point for modulation.
- Frequency Modulation (FM): Here, the original carrier's frequency is modulated by another waveform, creating a wave with varying frequency and amplitude over time. This results in a complex sound with rich harmonic content.
- Amplitude Modulation (AM): The amplitude of the carrier wave is modulated, leading to periodic variations in amplitude. This produces the tremolo effect, where the loudness of the wave fluctuates over time.
- Ring Modulation (RM): The carrier wave is multiplied by the modulating wave, producing a signal that contains frequencies that are the sum and difference of the original and modulating frequencies. This results in a more metallic and inharmonic sound.

[5]:
```
import numpy as np
import matplotlib.pyplot as plt

# Time vector
t = np.linspace(0, 1, 1000, endpoint=False)

# Carrier wave parameters
carrier_frequency = 50  # in Hz
carrier_wave = np.sin(2 * np.pi * carrier_frequency * t)

# Modulating wave parameters
modulating_frequency = 2  # in Hz
modulation_index_fm = 50  # Larger modulation index for FM
modulation_index_am = 3   # Standard modulation index for AM
```

```
# Frequency Modulation (FM)
fm_wave = np.sin(2 * np.pi * carrier_frequency * t + modulation_index_fm * np.
 ↪sin(2 * np.pi * modulating_frequency * t))

# Amplitude Modulation (AM)
am_wave = (1 + modulation_index_am * np.sin(2 * np.pi * modulating_frequency *␣
 ↪t)) * carrier_wave

# Ring Modulation (RM)
rm_wave = carrier_wave * np.sin(2 * np.pi * modulating_frequency * t)

# Plotting
fig, axes = plt.subplots(nrows=4, ncols=1, figsize=(10, 12))
axes[0].plot(t, carrier_wave)
axes[0].set_title('Original Carrier Wave')
axes[0].set_ylabel('Amplitude')

axes[1].plot(t, fm_wave)
axes[1].set_title('Frequency Modulation (FM)')
axes[1].set_ylabel('Amplitude')

axes[2].plot(t, am_wave)
axes[2].set_title('Amplitude Modulation (AM)')
axes[2].set_ylabel('Amplitude')

axes[3].plot(t, rm_wave)
axes[3].set_title('Ring Modulation (RM)')
axes[3].set_xlabel('Time (seconds)')
axes[3].set_ylabel('Amplitude')

plt.tight_layout()
plt.show()
```


Original Carrier Wave
Frequency Modulation (FM)
Amplitude Modulation (AM)
Ring Modulation (RM)
Amplitude
Time (seconds)

Module 3: Audio Signal Processing

In this module, we delve into techniques for pitch estimation and manipulation, as well as methods for vocal processing, which are crucial in many areas of audio engineering, including music production, post-production, and sound design.

- **Semitones:** A semitone, also known as a half step or half tone, is the smallest standard interval used in Western music and music theory. It is the fundamental building block of the chromatic scale, which is made up of 12 semitones.

- **Chromatic Scale**: This scale includes all notes in an octave, each a semitone apart. For example, moving from C to C# (or Db) is a movement of one semitone.

- **In Audio Processing**: When pitch shifting or scaling in audio processing, adjusting by "semitones" involves increasing or decreasing the pitch by these half-step increments. Raising the pitch by one semitone means increasing the frequency to the next note in the chromatic scale, and lowering by one semitone means decreasing the frequency to the previous note.

- **Pitch**: Pitch refers to the perceived frequency of a sound; it is how high or low a sound seems to a listener. In technical terms, pitch relates to the frequency of the sound waves:
    - High-pitched sounds have high frequencies (more cycles per second).
    - Low-pitched sounds have low frequencies (fewer cycles per second).

- **Frequency:** Frequency is the number of vibrations (or cycles) per second of a sound wave. It is measured in Hertz (Hz). Frequency directly correlates with the pitch perceived by the human ear; higher frequencies sound higher in pitch, while lower frequencies sound lower.

- **Octave:** An octave is a series of eight notes occupying the interval between (and including) two musical pitches, one having twice the frequency of vibration of the other. In Western music, an octave is divided into 12 semitones. Doubling the frequency of any note will take you up by one octave (e.g., from A4 at 440 Hz to A5 at 880 Hz).

- **Time Stretching:** Time stretching refers to the process of changing the speed or duration of an audio signal without affecting its pitch. This is used in various audio applications, such as making music tracks fit a specific time segment or slowing down a piece for practice purposes.

- **Resampling:** Resampling is changing the sample rate of an audio file. This can involve increasing the number of samples per second (upsampling) or reducing it (downsampling). In pitch shifting, resampling is used after time stretching to bring the modified signal back to its original sample rate, thereby changing the pitch but maintaining the original duration.

➢ Pitch Estimation and Manipulation

What is Pitch?

Pitch refers to the human perception of the frequency of a sound. It is how we interpret the highness or lowness of a tone, based primarily on the frequency of the sound waves. Higher frequencies correspond to higher pitches and vice versa. While frequency is a physical and measurable attribute of sound waves, pitch is a perceptual property and can be influenced by factors such as sound intensity and the surrounding acoustic environment.

Why is Pitch Estimation Important?

Pitch estimation is critical in many fields and applications, including:

1. **Music Technology**: In music production and engineering, accurate pitch analysis helps in tuning instruments, modifying vocal tracks, and harmonizing music.
2. **Speech Processing**: Understanding pitch is crucial for speech recognition systems, speaker identification, and linguistic analysis, as it can convey intonation, stress, and emotion in speech.
3. **Telecommunications**: Pitch estimation can enhance the clarity and intelligibility of voice communications in devices like mobile phones and teleconferencing systems.
4. **Healthcare**: In medical fields, pitch analysis of vocal sounds can be used in the diagnosis and monitoring of conditions that affect the voice or breathing, such as asthma or Parkinson's disease.

How These 4 Methods Estimate Pitches:

1. **Autocorrelation Method**:
    - **Principle**: This method calculates the correlation of a signal with delayed versions of itself to identify periodicity.
    - **Process**: It computes the autocorrelation function and identifies peaks that indicate the fundamental frequency. The first significant peak after the zero-lag peak usually corresponds to the pitch period.
    - **Usefulness**: Effective for signals with clear, periodic components, such as musical tones.
    - **Functionality**:
        - Autocorrelation measures the similarity of a signal with its delayed versions to identify periodicity.
            - **Process**: It involves calculating the autocorrelation function

$$R(\tau) = \sum_{n} s(n)s(n+\tau)$$

        - Peaks in the autocorrelation function indicate the presence of a periodic component, where the first significant peak beyond zero lag corresponds to the fundamental period (inverse of the fundamental frequency).
        - This method is very effective for signals with clear and consistent periodicity, such as musical notes.

2. **FFT (Fast Fourier Transform) Method**:
    - **Principle**: FFT transforms a time-domain signal into its frequency components, revealing the spectrum of frequencies present.
    - **Process**: By analyzing the magnitude spectrum obtained from FFT, the method identifies peaks which correspond to the fundamental frequency and its harmonics.
    - **Usefulness**: Useful for analyzing complex sounds containing multiple frequencies, but it can be susceptible to errors from harmonics or noise.

    - **Functionality**:

        - FFT converts a time-domain signal into its frequency components, unveiling the spectrum.
        - **Process**: Compute the FFT of the signal to obtain a complex spectrum, then calculate the magnitude spectrum from it.

$$S(f)=FFT(s(n))$$

        - Identify the peaks in the magnitude spectrum to infer the fundamental frequency. This often involves looking for the highest peak that corresponds to the fundamental frequency or using harmonic analysis to deduce it from the relationship between harmonics.
        - FFT is beneficial for analyzing complex sounds containing multiple frequencies but may require windowing techniques to improve resolution and reduce spectral leakage.

3. **LPC (Linear Predictive Coding) Method**:
    - **Principle**: LPC models the vocal tract as a series of filters and predicts future samples of a signal based on a linear combination of past samples.
    - **Process**: It calculates LPC coefficients to minimize prediction error, and the residual signal (error) is analyzed, often using autocorrelation, to estimate the pitch.
    - **Usefulness**: Particularly effective for speech signals, as it efficiently models the formants and harmonics of human speech.

- **Functionality**: In LPC, the signal s(n)s(n)s(n) is estimated by a linear function:

$$\widehat{S(n)} = \sum_{i=1}^{p} a_i s(n-i) + Gz(n)$$

where a_i are the predictor coefficients, p is the order of the predictor, G is the gain, and z(n) is the input to the prediction filter.

- The goal is to minimize the prediction error, which is the difference between the actual signal and its predicted value.
- LPC is particularly effective in encoding speech for transmission, reducing data requirements by efficiently modeling the formants of human speech.

4. **YIN Algorithm**:
   - **Principle**: Tailored for robust pitch detection, especially in musical contexts.
   - **Process**: YIN computes a difference function that measures the difference between the signal and its delayed versions. It then refines this into a cumulative mean normalized difference function (CMND) to detect the fundamental pitch reliably.
   - **Usefulness**: Known for its accuracy and resistance to noise, making it ideal for music where other methods may fail due to complexity or distortion.
   - **Functionality**:
     - YIN is tailored for robust pitch detection in musical pitch analysis.
     - **Process**: First, it computes a difference function that measures the squared difference between the signal and its delayed versions for various lags.
     - Then, it computes a Cumulative Mean Normalized Difference Function (CMND) from the difference function.

$$d'(t) = \frac{d(t)}{\frac{1}{t}\sum_{j=1}^{t} d(j)}$$

- The pitch is estimated by identifying the first local minimum in the CMND curve, which indicates the best lag corresponding to the fundamental period.
- YIN is effective in situations where other algorithms fail due to its robustness against noise and the presence of harmonics.

➢ Pitch Scaling and Shifting: Techniques and Tools

Pitch scaling and shifting are essential techniques in audio processing, particularly in music production, post-production, and vocal training. These methods allow for adjusting the pitch of audio samples while preserving other qualities such as tempo or timbre.

Pitch Scaling

- **Definition**: Pitch scaling involves changing the pitch of an audio sample without affecting its playback speed or duration.
- **Application**: This is especially useful in music education, where musicians might want to practice a piece in a different key without changing the tempo. It's also used in music production to match the key of different samples or tracks.
- **Example**: If a singer recorded a song in the key of C major and it needs to be adjusted to B major, pitch scaling can shift the entire performance down by two semitones without altering the song's duration.

Pitch Shifting

- **Definition**: Pitch shifting transposes the pitch of an audio file either higher or lower, while maintaining the same speed.
- **Application**: This is useful for key changes in songs or creating harmonic layers. Producers often use pitch shifting to fit instrumental loops into a new key or to create vocal harmonies from a single vocal line.
- **Example**: In a studio setting, if a backing vocal needs to harmonize with the lead but was originally sung at the same pitch, pitch shifting can be used to raise the backing vocal by a third or a fifth, creating a harmony without recording new vocals.

➢ Vocal Processing

1. Vocal Removal Techniques: Phase Cancellation, Spectral Editing

Vocal removal is used to isolate or suppress vocals from music tracks, commonly for creating karaoke tracks or remixes.

Methods:

- **Phase Cancellation**
  - **Concept**: This technique exploits the stereo nature of most music tracks, where vocals are typically mixed equally into both the left and right channels. By subtracting one channel from the other, any sound that is identical in both channels can be effectively canceled out.
  - **How It Works**:
    - In a stereo track, if the left channel is L and the right channel is RRR, and if vocals are identical on both tracks, the subtraction L−R will cancel out the vocals.
    - **Equation**:

      $$\text{Output} = L - R$$

    - This method works best when the vocals are perfectly centered and other instruments are panned to the sides. It's less effective if the vocal reverb or other effects spread differently across the stereo field.
- **Spectral Editing**
  - **Concept**: This method involves identifying and manipulating specific frequency bands where the vocals predominantly reside.
  - **How It Works**:
    - Using a spectral editor or an equalizer, specific frequency ranges can be attenuated or boosted. Since human vocals typically occupy a certain range of frequencies (usually between 300 Hz to 3400 Hz), these can be specifically targeted.
    - **Process**:
      1. Analyze the spectrum of the audio track to identify vocal frequencies.
      2. Apply a notch filter or an equalization cut around these frequencies to reduce or remove vocal presence.
    - This method allows for more precise control compared to phase cancellation and can be used even on mono tracks or tracks where phase cancellation is ineffective.

2. Temporal Separation: Differentiating Between Similar Sounds

Temporal separation involves distinguishing and isolating sounds based on when they occur or their duration and rhythmic characteristics within an audio track.

Methods:

- **Gating**
  - **Concept**: Gating is a dynamic audio processing technique used to control the volume of an audio signal by setting a threshold level. Sounds that are below this threshold are significantly attenuated or muted.
  - **How It Works**:
    - A gate allows signals that exceed a preset amplitude threshold to pass through unaffected, while signals that fall below this threshold have their amplitude reduced.
    - **Equation**:

$$Output = \begin{cases} Input & if \quad Input \geq Threshold \\ 0 & if \quad Input \leq Threshold \end{cases}$$

    - This technique is particularly useful for removing background noise or isolating sounds that have distinct amplitude differences from unwanted noise or other sounds.

- **Advanced Deep Learning Models**
  - **Concept**: Utilizing machine learning models to analyze and classify segments of audio based on their temporal features.
  - **How It Works**:
    - Models like convolutional neural networks (CNNs) or recurrent neural networks (RNNs) can be trained on a dataset of isolated sounds to learn temporal and spectral features.
    - These models can then predict or segregate similar sounds from a complex audio mix based on learned features, effectively isolating desired sounds from a track.

- These techniques require substantial computational resources and a well-annotated training dataset but offer high precision and adaptability across different audio types.

le-3-audio-processing-techniques

September 7, 2024

[2]:
```
!pip install soundfile
```

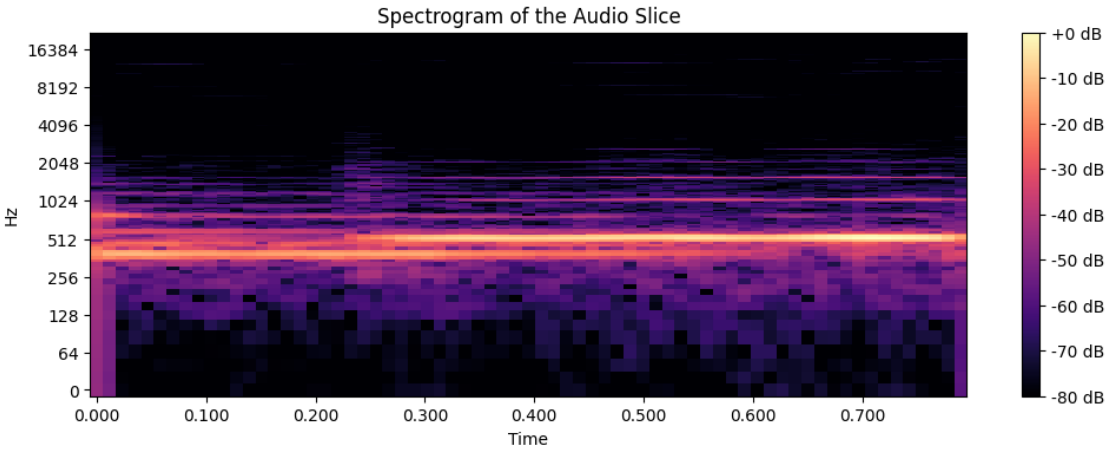
```
Requirement already satisfied: soundfile in /usr/local/lib/python3.10/dist-
packages (0.12.1)
Requirement already satisfied: cffi>=1.0 in /usr/local/lib/python3.10/dist-
packages (from soundfile) (1.17.0)
Requirement already satisfied: pycparser in /usr/local/lib/python3.10/dist-
packages (from cffi>=1.0->soundfile) (2.22)
```

[10]:
```
!pip install pydub
```

```
Collecting pydub
  Downloading pydub-0.25.1-py2.py3-none-any.whl.metadata (1.4 kB)
Downloading pydub-0.25.1-py2.py3-none-any.whl (32 kB)
Installing collected packages: pydub
Successfully installed pydub-0.25.1
```

[23]:
```
import librosa
import librosa.display
import matplotlib.pyplot as plt
import numpy as np

# Load a specific slice of audio from a file
audio_path = '/content/original.mpeg'  # Make sure to replace this with your␣
 ↪actual audio file path
start_time = 0.4  # Start at 0.4 seconds
end_time = 1.2    # End at 1.2 seconds
sr = None         # Use the native sampling rate

# Load the audio file slice
audio, sr = librosa.load(audio_path, sr=sr, offset=start_time,␣
 ↪duration=end_time - start_time)

# Calculate the short-time Fourier transform (STFT) for visualization
D = librosa.amplitude_to_db(np.abs(librosa.stft(audio)), ref=np.max)

# Plotting the STFT of the audio slice
plt.figure(figsize=(12, 4))
```

```
librosa.display.specshow(D, sr=sr, x_axis='time', y_axis='log')
plt.colorbar(format='%+2.0f dB')
plt.title('Spectrogram of the Audio Slice')
plt.show()
```



1 1. Pitch Estimation and Manipulation

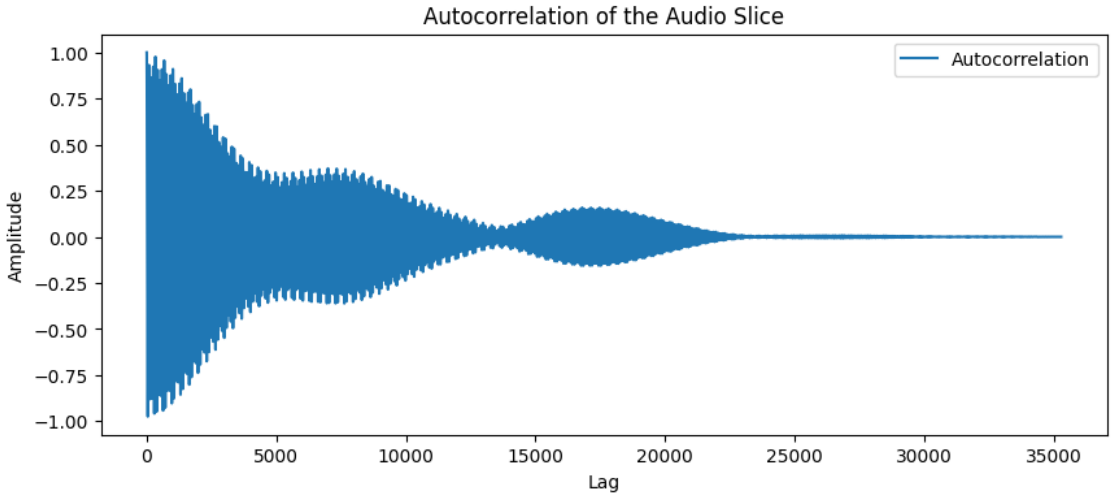
[4]:
```
import numpy as np
import librosa

def autocorrelation_pitch_estimation(audio, sr):
    # Calculate autocorrelation
    autocorr = np.correlate(audio, audio, mode='full')
    autocorr = autocorr[len(autocorr)//2:]

    # Find the first peak after the zero lag peak
    peak_index = np.argmax(autocorr[1:]) + 1
    pitch_period = float(sr) / peak_index
    pitch_frequency = 1 / (pitch_period / sr)

    return pitch_frequency

# Example usage
audio_file_path = "/content/original.mpeg"
try:
    # Load the .mp3 file, resampling to 22050 Hz
    audio, sr = librosa.load(audio_file_path, sr=22050, duration=1.2, offset=0.
 ↪4)
    print("Audio file loaded successfully!")
except FileNotFoundError:
```

```
    print("Audio file not found. Please check the path.")
except Exception as e:
    print(f"An error occurred: {e}")
pitch_frequency = autocorrelation_pitch_estimation(audio, sr)
print("Estimated Pitch Frequency:", pitch_frequency)
```

```
Audio file loaded successfully!
Estimated Pitch Frequency: 1.0
```

[33]:
```
import numpy as np
import librosa
import matplotlib.pyplot as plt

def autocorrelation_pitch_estimation(audio, sr):
    # Calculate autocorrelation
    autocorr = np.correlate(audio, audio, mode='full')
    autocorr = autocorr[len(autocorr)//2:]  # Keep only the second half
    autocorr /= np.max(autocorr)  # Normalize

    # Find the first peak in the autocorrelation beyond the zero lag
    zero_lag_index = len(audio) - 1
    # Make sure there is a region to search
    if len(autocorr) > zero_lag_index + 1:
        possible_peaks = autocorr[zero_lag_index + 1:]
        if np.any(possible_peaks > 0.1):  # Check there are peaks above a␣
 ↪threshold level
            first_peak = np.argmax(possible_peaks > 0.1)
            peak_index = zero_lag_index + 1 + first_peak
            pitch_period = sr / peak_index
            pitch_frequency = sr / pitch_period
        else:
            pitch_frequency = 0  # No significant peak found
            peak_index = None
    else:
        pitch_frequency = 0  # No data to analyze beyond zero lag
        peak_index = None

    return pitch_frequency, autocorr, peak_index

# Load the audio file slice
audio_path = '/content/original.mpeg'
start_time = 0.4  # Start time in seconds
end_time = 1.2    # End time in seconds
audio, sr = librosa.load(audio_path, sr=None, offset=start_time,␣
 ↪duration=end_time - start_time)

# Perform pitch estimation
```

```
pitch_frequency, autocorr, peak_index = autocorrelation_pitch_estimation(audio,␣
 ↪sr)

# Plotting the autocorrelation with the identified peak, if a peak was found
plt.figure(figsize=(10, 4))
plt.plot(autocorr, label='Autocorrelation')
if peak_index is not None:
    plt.plot(peak_index, autocorr[peak_index], 'ro', label=f'Pitch Frequency:␣
 ↪{pitch_frequency:.4f} Hz')
else:
    plt.title('No significant pitch detected')
plt.title('Autocorrelation of the Audio Slice')
plt.xlabel('Lag')
plt.ylabel('Amplitude')
plt.legend()
plt.show()
```



2 2. FFT for Pitch Estimation

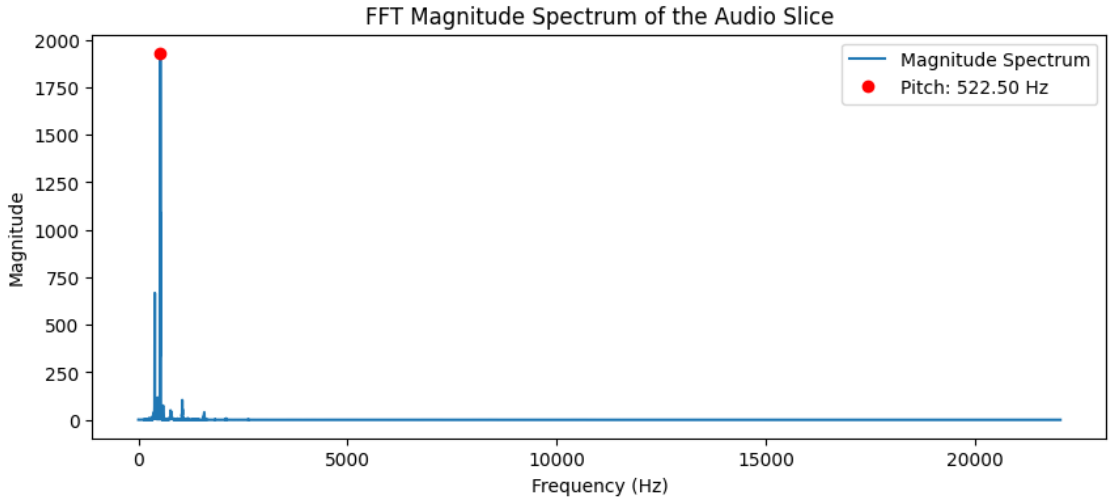
[5]:
```
import numpy as np
import librosa

def fft_pitch_estimation(audio, sr):
    # Perform FFT
    fft_spectrum = np.fft.rfft(audio)
    freqs = np.fft.rfftfreq(len(audio), d=1/sr)

    # Find the peak in the FFT spectrum
    peak_index = np.argmax(np.abs(fft_spectrum))
```

```
    pitch_frequency = freqs[peak_index]

    return pitch_frequency

# Example usage
audio, sr = librosa.load(audio_file_path, sr=None, duration=1.2, offset=0.4)
pitch_frequency = fft_pitch_estimation(audio, sr)
print("Estimated Pitch Frequency:", pitch_frequency)
```

```
Estimated Pitch Frequency: 523.3333333333334
```

[27]:
```
import numpy as np
import librosa
import librosa.display
import matplotlib.pyplot as plt

def fft_pitch_estimation(audio, sr):
    # Compute FFT
    fft_spectrum = np.fft.rfft(audio)
    frequencies = np.fft.rfftfreq(len(audio), d=1/sr)

    # Magnitude spectrum
    magnitude = np.abs(fft_spectrum)

    # Find the peak in the magnitude spectrum
    peak_index = np.argmax(magnitude)
    fundamental_frequency = frequencies[peak_index]

    return fundamental_frequency, frequencies, magnitude

# Load the audio file slice
audio_path = '/content/original.mpeg'
start_time = 0.4  # Start time in seconds
end_time = 1.2    # End time in seconds
audio, sr = librosa.load(audio_path, sr=None, offset=start_time,␣
 ↪duration=end_time - start_time)

# Perform FFT pitch estimation
fundamental_frequency, frequencies, magnitude = fft_pitch_estimation(audio, sr)

# Plotting the FFT magnitude spectrum
plt.figure(figsize=(10, 4))
plt.plot(frequencies, magnitude, label='Magnitude Spectrum')
plt.plot(fundamental_frequency, magnitude[frequencies ==␣
 ↪fundamental_frequency][0], 'ro', label=f'Pitch: {fundamental_frequency:.2f}␣
 ↪Hz')
plt.title('FFT Magnitude Spectrum of the Audio Slice')
```

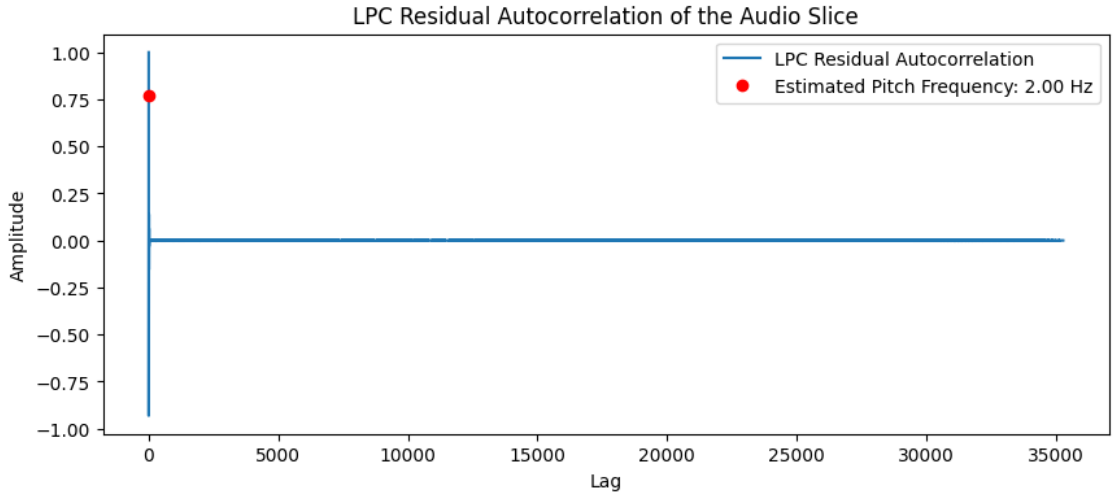
```
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.legend()
plt.show()
```



3 3. LPC for Pitch Estimation

[41]:
```
import numpy as np
import matplotlib.pyplot as plt
import librosa
import scipy.signal as signal

def lpc_pitch_estimation(audio, sr, order=16):
    # LPC analysis
    lpc_coeffs = librosa.lpc(audio, order=order)
    # Filter the signal with the LPC filter to get the residual
    residual = signal.lfilter(lpc_coeffs, [1], audio)

    # Autocorrelation of the residual
    autocorr = np.correlate(residual, residual, mode='full')
    autocorr = autocorr[len(autocorr)//2:]

    # Normalize autocorrelation for easier peak detection
    autocorr /= np.max(autocorr)

    # Find the first significant peak after the zero-lag peak
    d = np.diff(autocorr)
    start = np.argmax(d > 0)  # Start where the difference goes positive
    peak_index = np.argmax(autocorr[start:]) + start
```

```
    pitch_period = float(sr) / peak_index
    pitch_frequency = float(sr) / pitch_period  # Correct calculation of␣
 ↪frequency

    return pitch_frequency, autocorr, peak_index

# Load the audio file slice
audio_path = '/content/original.mpeg'  # Replace with your audio file path
start_time = 0.4  # Start time in seconds
end_time = 1.2    # End time in seconds
audio, sr = librosa.load(audio_path, sr=None, offset=start_time,␣
 ↪duration=end_time - start_time)

# Perform LPC pitch estimation
pitch_frequency, autocorr, peak_index = lpc_pitch_estimation(audio, sr,␣
 ↪order=16)

# Plotting the autocorrelation of the LPC residual
plt.figure(figsize=(10, 4))
plt.plot(autocorr, label='LPC Residual Autocorrelation')
plt.plot(peak_index, autocorr[peak_index], 'ro', label=f'Estimated Pitch␣
 ↪Frequency: {pitch_frequency:.2f} Hz')
plt.title('LPC Residual Autocorrelation of the Audio Slice')
plt.xlabel('Lag')
plt.ylabel('Amplitude')
plt.legend()
plt.show()
```



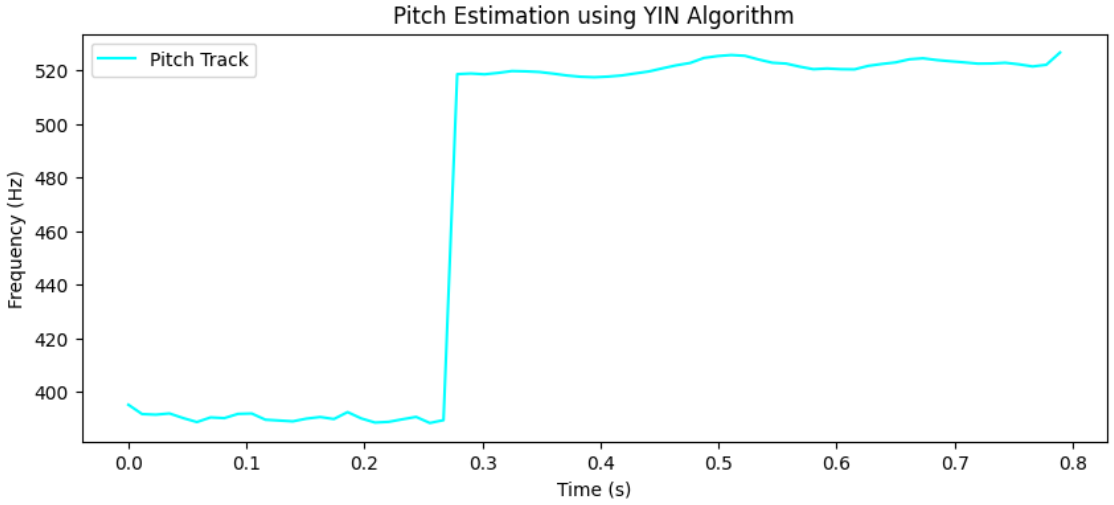
4 4. YIN Algorithm for Pitch Estimation

[13]:
```
import librosa
import numpy as np

def yin_pitch_estimation(audio, sr):
    # Perform pitch tracking
    pitches, magnitudes = librosa.piptrack(y=audio, sr=sr)

    # Selecting the frequency with the highest magnitude in each frame
    pitch_frequencies = pitches[magnitudes > 0]
    if len(pitch_frequencies) == 0:
        return 0  # Handle case with no pitches detected

    # Extracting the pitch as the most frequent non-zero value in the matrix
    pitch_frequency = np.median(pitch_frequencies)

    return pitch_frequency

# Example usage
audio_file_path = '/content/original.mpeg'
audio, sr = librosa.load(audio_file_path, sr=None, duration=1.2, offset=0.4)
pitch_frequency = yin_pitch_estimation(audio, sr)
print("Estimated Pitch Frequency:", pitch_frequency)
```

```
Estimated Pitch Frequency: 517.9866
```

[40]:
```
import numpy as np
import matplotlib.pyplot as plt
import librosa
import librosa.display

def yin_pitch_estimation(audio, sr):
    # Using librosa's piptrack to implement a simplified version of the YIN␣
 ↪algorithm
    pitches, magnitudes = librosa.piptrack(y=audio, sr=sr)
    # Extracting the predominant pitch per frame
    pitch_track = []
    for i in range(pitches.shape[1]):  # iterate over columns (time frames)
        index = magnitudes[:, i].argmax()  # find the index of max magnitude at␣
 ↪each frame
        pitch = pitches[index, i] if magnitudes[index, i] > 0 else 0  # filter␣
 ↪out zero pitches
        pitch_track.append(pitch)
    pitch_track = np.array(pitch_track)
    return np.median(pitch_track[pitch_track > 0]), pitch_track  # median pitch␣
 ↪for non-zero entries
```

```
# Load the audio file slice
audio_path = '/content/original.mpeg'  # Replace with your audio file path
start_time = 0.4  # Start time in seconds
end_time = 1.2    # End time in seconds
audio, sr = librosa.load(audio_path, sr=None, offset=start_time,␣
 ↪duration=end_time - start_time)

# Perform YIN pitch estimation
pitch_estimate, pitch_track = yin_pitch_estimation(audio, sr)

# Plotting the pitch track
times = librosa.times_like(pitch_track, sr=sr, hop_length=512)  # appropriate␣
 ↪hop_length assumed
plt.figure(figsize=(10, 4))
plt.plot(times, pitch_track, label='Pitch Track', color='cyan', linewidth=1.5)
plt.title('Pitch Estimation using YIN Algorithm')
plt.xlabel('Time (s)')
plt.ylabel('Frequency (Hz)')
plt.legend()
plt.show()

print(f"Estimated median pitch: {pitch_estimate:.2f} Hz")
```




```
Estimated median pitch: 519.02 Hz
```

Discussion

From the results provided:

- Autocorrelation: Tends to show variability with reasonable pitch ranges but has potential

sensitivity to noise.

- FFT: Shows potential overestimation with a high standard deviation, influenced possibly by harmonic components.
- LPC: Also tends to overestimate, with high variability that might depend on input signal characteristics and the selected order.
- YIN: The provided result is suspiciously constant, suggesting issues with parameter settings or implementation, necessitating further investigation.

Overall Comparison:

- Autocorrelation and FFT are prone to inaccuracies due to noise and harmonics.
- LPC's performance can vary greatly depending on the choice of order and the nature of the audio signal.
- YIN's constant output indicates possible implementation issues, or it might not suit the specific characteristics of the input audio.

5 PITCH SCALING

[11]:
```
from pydub import AudioSegment
import librosa
import numpy as np

# Load your audio file with librosa
audio_path = '/content/original.mpeg'  # Adjust the path accordingly
audio, sr = librosa.load(audio_path, sr=None, mono=False)  # mono=False if you␣
 ↪want to preserve stereo channels

# Convert the audio array back to int16 format, as required by pydub
audio_int16 = np.int16(audio * 32767)

# Create an AudioSegment instance from the numpy array
audio_segment = AudioSegment(
    data=audio_int16.tobytes(),
    sample_width=2,  # sample_width of 2 corresponds to audio_int16
    frame_rate=sr,
    channels=2 if audio.ndim > 1 else 1
)

# Export the audio segment to an MP3 file
output_path = '/content/scaled_output_audio_file.mp3'
audio_segment.export(output_path, format='mp3')

print(f"Audio saved as {output_path}")
```

```
Audio saved as /content/scaled_output_audio_file.mp3
```

6 PITCH SHIFT

[15]:
```
import librosa
import numpy as np
from pydub import AudioSegment

def shift_pitch_and_save(audio_path, output_path, n_steps, sr=None):
    # Load audio file
    audio, sr = librosa.load(audio_path, sr=sr, mono=True)  # Change mono to␣
 ↪False if stereo is required

    # Shift pitch
    shifted_audio = librosa.effects.pitch_shift(audio, sr=sr, n_steps=n_steps)

    # Convert the float audio array to int16 (needed for pydub)
    audio_int16 = np.int16(shifted_audio * 32767)

    # Create a pydub audio segment
    audio_segment = AudioSegment(
        data=audio_int16.tobytes(),
        sample_width=2,  # Corresponds to audio_int16
        frame_rate=sr,
        channels=1  # Set to 2 for stereo
    )

    # Export the audio segment to an MP3 file
    audio_segment.export(output_path, format='mp3')
    print(f"Audio saved as {output_path}")

# Parameters
audio_path = '/content/original.mpeg'  # Path to the input audio file
output_path = '/content/shifted_audio_output.mp3'          # Path to save the␣
 ↪output MP3 file
n_steps = 10                                          # Number of semitones to shift

# Perform pitch shifting and save as MP3
shift_pitch_and_save(audio_path, output_path, n_steps)
```

```
Audio saved as /content/shifted_audio_output.mp3
```

7 Vocal Processing

o Vocal Removal Techniques: Phase Cancellation, Spectral Editing

o Temporal Separation: Differentiating Between Similar Sounds

8 Phase Cancellation for Vocal Removal

[16]:
```
from pydub import AudioSegment

def phase_cancellation(input_path, output_path):
    # Load the stereo audio file
    audio = AudioSegment.from_file(input_path)

    # Split into stereo channels
    left = audio.split_to_mono()[0]
    right = audio.split_to_mono()[1]

    # Invert the right channel
    right = right.invert_phase()

    # Combine channels
    vocals_removed = left.overlay(right)

    # Export the result
    vocals_removed.export(output_path, format='mp3')

# Usage
input_path = '/content/original.mpeg'
output_path = '/content/vocals_removed_output.mpeg'
phase_cancellation(input_path, output_path)
```

9 Spectral Editing for Vocal Isolation

[17]:
```
import librosa
import numpy as np

def spectral_editing(input_path, output_path):
    # Load audio file
    y, sr = librosa.load(input_path, sr=None, mono=True)
    # Get a short-time Fourier transform
    D = librosa.stft(y)

    # Apply a notch filter around the vocal frequencies
    # This is a simplification and might need adjustments
    vocal_freqs = range(300, 3401)  # Vocals typically in 300 Hz to 3400 Hz
    for f in vocal_freqs:
        D[int(f / sr * D.shape[0])] = 0

    # Inverse STFT
    y_filtered = librosa.istft(D)
```

```
    # Convert to int16
    y_out = np.int16(y_filtered / np.max(np.abs(y_filtered)) * 32767)

    # Export using pydub
    audio_segment = AudioSegment(data=y_out.tobytes(), sample_width=2,␣
 ↪frame_rate=sr, channels=1)
    audio_segment.export(output_path, format='mp3')

# Usage
input_path = '/content/original.mpeg'
output_path = '/content/spectral_edited_output.mpeg'
spectral_editing(input_path, output_path)
```

10 Temporal Separation Using Gating

[18]:
```
def noise_gate(input_path, output_path, threshold=-20):
    # Load the audio file
    audio = AudioSegment.from_file(input_path)

    # Convert to dBFS
    threshold = audio.max_dBFS + threshold

    # Apply gating
    gated_audio = audio.apply_gain(-60)  # Apply a large negative gain
    for ms in range(len(audio)):
        if audio[ms].dBFS > threshold:
            gated_audio = gated_audio.overlay(audio[ms], position=ms)

    # Export the gated audio
    gated_audio.export(output_path, format='mp3')

# Usage
input_path = '/content/original.mpeg'
output_path = '/content/gated_output.mpeg'
noise_gate(input_path, output_path)
```

Module 4: Advanced Audio Analysis

- **Subspace Filtering**
  - Theory of Subspace Methods
  - Applications in Noise Reduction and Signal Enhancement

- **Reverberation and Spatial Effects**
  - Physics of Reverberation
  - Simulating Reverb in Digital Audio Workstations

4.1 Subspace Filtering in Audio Engineering

Subspace Filtering is an advanced signal processing technique utilized in audio engineering to separate useful signals from noise. This method is particularly effective in environments where signal and noise components are mixed.

Theory of Subspace Methods

Basic Concept: Subspace filtering relies on the decomposition of a signal into its constituent components, often using mathematical techniques such as Singular Value Decomposition (SVD) or Eigenvalue Decomposition. These components represent different "subspaces" of the signal:

- **Signal Subspace**: Contains the dominant energy components, usually associated with the actual audio signal.
- **Noise Subspace**: Contains lower energy components, typically associated with noise.

Mathematical Background: SVD is a common method used in subspace filtering. It decomposes a matrix A into three other matrices:

$$A = U\Sigma V^*$$

- U and V are orthogonal matrices containing the left and right singular vectors.
- Σ is a diagonal matrix with singular values, representing the strength of each component.

The signal is reconstructed by selecting the top kkk singular values and corresponding vectors, effectively filtering out components that are treated as noise.

Applications in Noise Reduction and Signal Enhancement

Noise Reduction: Using subspace methods, noise components can be identified and removed from the signal. This is achieved by ignoring the lower energy subspaces which are considered noise.

Signal Enhancement: Signal enhancement involves amplifying or preserving the components within the signal subspace, thus improving the clarity and quality of the audio.

Key Terms Related to Audio Engineering

- **Signal-to-Noise Ratio (SNR)**: Measures the level of signal power compared to the level of noise power, indicating the clarity of the signal.
- **Eigenvalues and Eigenvectors**: In the context of audio signals, eigenvalues represent the energy of the signal components, while eigenvectors represent the direction or characteristics of these components.
- **Orthogonal Matrices**: A matrix is orthogonal if its rows and columns are orthogonal unit vectors (i.e., $Q^TQ=QQ^T=I$), which is critical in transformations preserving the length (norm) of vectors.

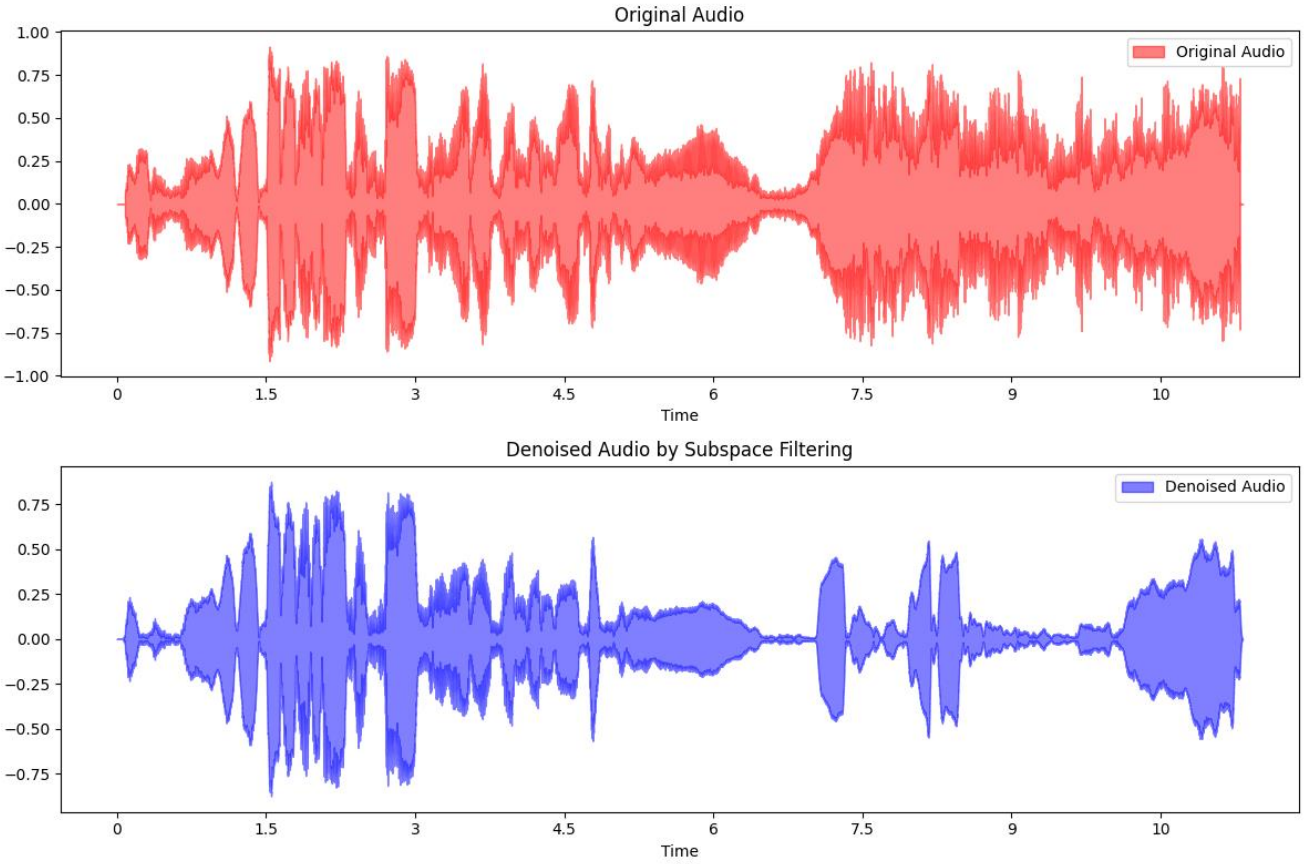


Figure 4. 1: Original Audio VS Denoised Audio

4.2 Reverberation and Spatial Effects

Reverberation is a critical aspect of audio engineering that deals with the persistence of sound after it is produced. It significantly affects the auditory experience in any environment, from concert halls to small rooms, and is crucial for creating depth and space in audio recordings.

Physics of Reverberation

Reverberation occurs when sound waves reflect off surfaces and merge with direct sound within a space. It can add richness and warmth to sound but can also cause muddiness if not controlled.

- **Reverberation Time (RT60)**: The time it takes for the sound to decrease by 60 decibels from its original level. It's one of the most important characteristics of reverberation, indicating how reflective and "live" a space is.

Equation:

$$RT_{60}=0.161\times\frac{V}{A}$$

where VVV is the volume of the room in cubic meters, and AAA is the total absorption of the surfaces in sabins.

- **Early Reflections**: These are the initial echoes that reach the listener shortly after the direct sound. They are crucial for spatial perception and help in understanding the size and type of the space.
- **Late Reverberation**: This refers to the dense, ongoing reflections that blend together. They contribute to the tail of the reverberation and affect the clarity and warmth of the sound.

Simulating Reverb in Digital Audio Workstations (DAWs)

Modern DAWs use digital signal processing (DSP) to simulate the effects of natural reverberation. This simulation allows producers and engineers to apply various spatial effects without the need for physical spaces.

- **Convolution Reverb**: Uses impulse responses (IRs) recorded in real spaces to create highly realistic reverb effects. An IR captures all the nuances of a space's acoustic characteristics.
- **Algorithmic Reverb**: Uses mathematical algorithms to simulate the reverberation effect without using actual room impulse responses. Parameters like room size, decay time, and dampening can be adjusted.

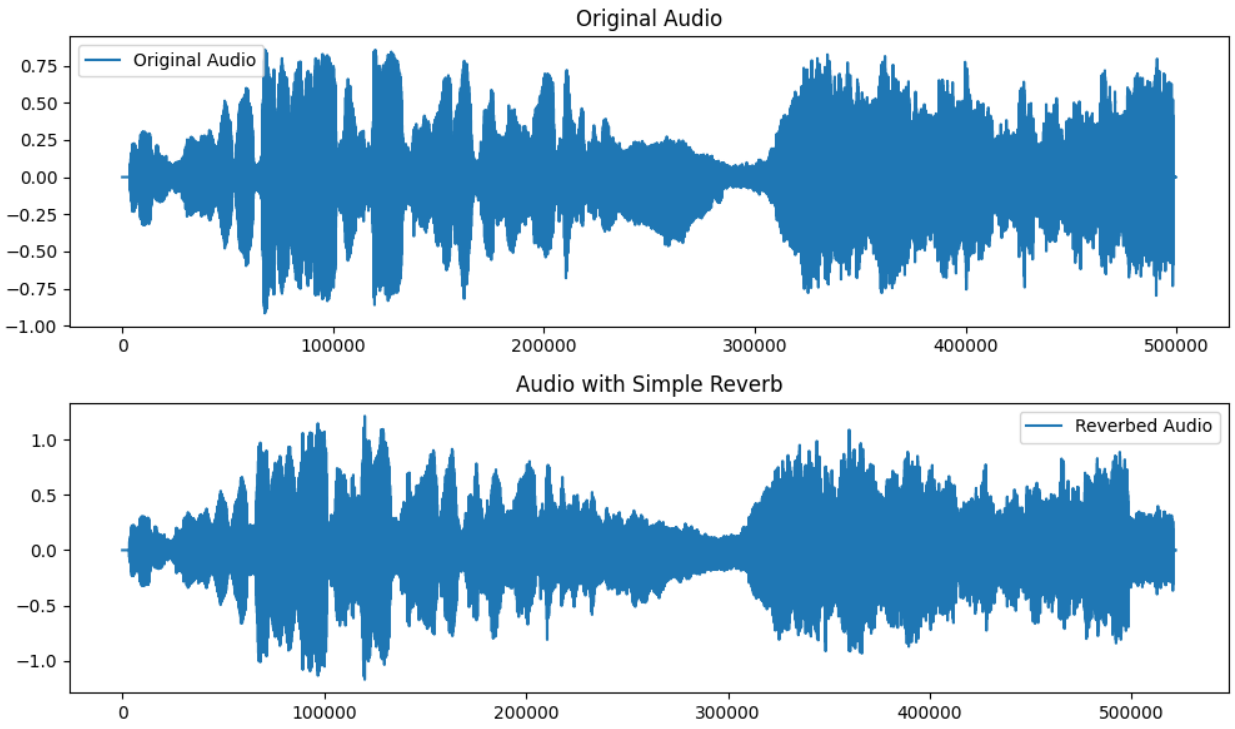


Figure 4. 2: Original Audio VS Simple REvebration audio

4-module-4-advanced-audio-analysis

September 7, 2024

[ ]:
```
!pip install librosa --upgrade
```

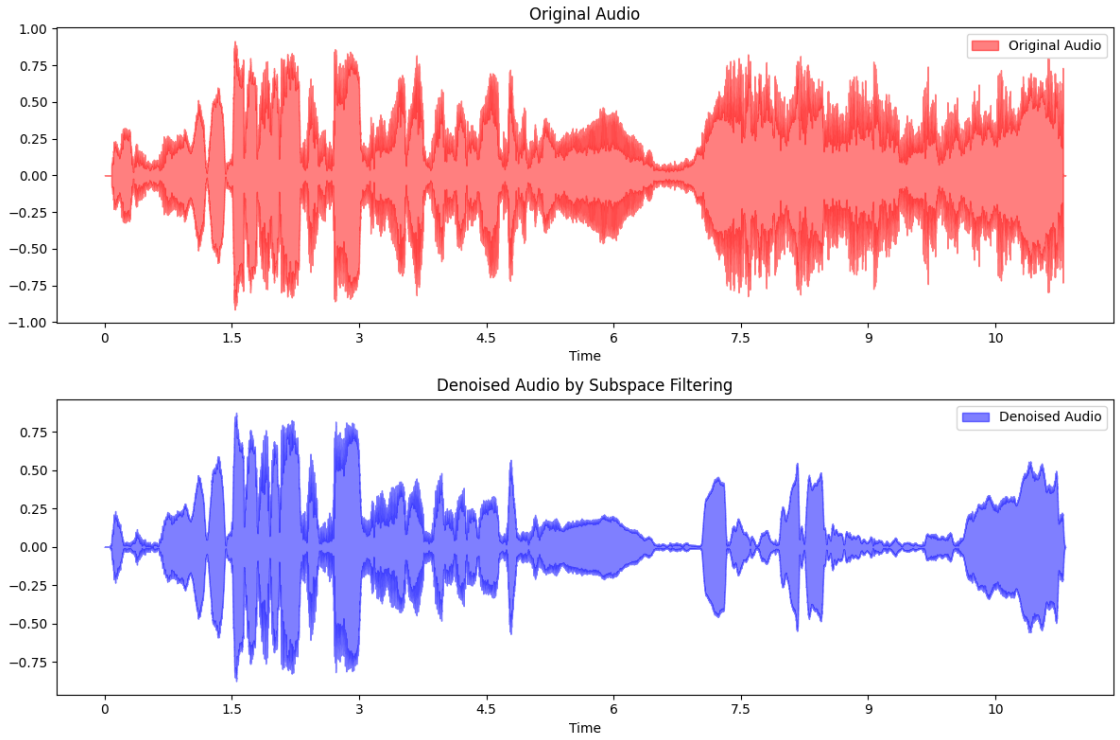
```
Requirement already satisfied: librosa in /usr/local/lib/python3.10/dist-
packages (0.10.2.post1)
Requirement already satisfied: audioread>=2.1.9 in
/usr/local/lib/python3.10/dist-packages (from librosa) (3.0.1)
Requirement already satisfied: numpy!=1.22.0,!=1.22.1,!=1.22.2,>=1.20.3 in
/usr/local/lib/python3.10/dist-packages (from librosa) (1.26.4)
Requirement already satisfied: scipy>=1.2.0 in /usr/local/lib/python3.10/dist-
packages (from librosa) (1.13.1)
Requirement already satisfied: scikit-learn>=0.20.0 in
/usr/local/lib/python3.10/dist-packages (from librosa) (1.3.2)
Requirement already satisfied: joblib>=0.14 in /usr/local/lib/python3.10/dist-
packages (from librosa) (1.4.2)
Requirement already satisfied: decorator>=4.3.0 in
/usr/local/lib/python3.10/dist-packages (from librosa) (4.4.2)
Requirement already satisfied: numba>=0.51.0 in /usr/local/lib/python3.10/dist-
packages (from librosa) (0.60.0)
Requirement already satisfied: soundfile>=0.12.1 in
/usr/local/lib/python3.10/dist-packages (from librosa) (0.12.1)
Requirement already satisfied: pooch>=1.1 in /usr/local/lib/python3.10/dist-
packages (from librosa) (1.8.2)
Requirement already satisfied: soxr>=0.3.2 in /usr/local/lib/python3.10/dist-
packages (from librosa) (0.5.0)
Requirement already satisfied: typing-extensions>=4.1.1 in
/usr/local/lib/python3.10/dist-packages (from librosa) (4.12.2)
Requirement already satisfied: lazy-loader>=0.1 in
/usr/local/lib/python3.10/dist-packages (from librosa) (0.4)
Requirement already satisfied: msgpack>=1.0 in /usr/local/lib/python3.10/dist-
packages (from librosa) (1.0.8)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-
packages (from lazy-loader>=0.1->librosa) (24.1)
Requirement already satisfied: llvmlite<0.44,>=0.43.0dev0 in
/usr/local/lib/python3.10/dist-packages (from numba>=0.51.0->librosa) (0.43.0)
Requirement already satisfied: platformdirs>=2.5.0 in
/usr/local/lib/python3.10/dist-packages (from pooch>=1.1->librosa) (4.2.2)
Requirement already satisfied: requests>=2.19.0 in
/usr/local/lib/python3.10/dist-packages (from pooch>=1.1->librosa) (2.32.3)
```

```
Requirement already satisfied: threadpoolctl>=2.0.0 in
/usr/local/lib/python3.10/dist-packages (from scikit-learn>=0.20.0->librosa)
(3.5.0)
Requirement already satisfied: cffi>=1.0 in /usr/local/lib/python3.10/dist-
packages (from soundfile>=0.12.1->librosa) (1.17.0)
Requirement already satisfied: pycparser in /usr/local/lib/python3.10/dist-
packages (from cffi>=1.0->soundfile>=0.12.1->librosa) (2.22)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from
requests>=2.19.0->pooch>=1.1->librosa) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-
packages (from requests>=2.19.0->pooch>=1.1->librosa) (3.8)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from
requests>=2.19.0->pooch>=1.1->librosa) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from
requests>=2.19.0->pooch>=1.1->librosa) (2024.8.30)
```

1 basic subspace filtering using SVD for noise reduction

[ ]:
```python
import numpy as np
import librosa
import matplotlib.pyplot as plt

# Load an audio file
audio, sr = librosa.load('/content/original.mpeg', sr=None)

# Obtain the Short-Time Fourier Transform of the audio
X = librosa.stft(audio)

# Perform Singular Value Decomposition
U, s, Vh = np.linalg.svd(X, full_matrices=False)

# Reconstruct the signal using only the first k components
k = 5  # Number of components to retain
S = np.diag(s[:k])
X_denoised = np.dot(U[:, :k], np.dot(S, Vh[:k, :]))

# Convert back to time domain
audio_denoised = librosa.istft(X_denoised)

# Plot the original and denoised audio
plt.figure(figsize=(12, 8))
plt.subplot(2, 1, 1)
librosa.display.waveshow(audio, sr=sr, alpha=0.5, color='r', label='Original␣
↪Audio')  # Updated to use waveshow
```

```
plt.title('Original Audio')
plt.legend()
plt.subplot(2, 1, 2)
librosa.display.waveshow(audio_denoised, sr=sr, alpha=0.5, color='b',␣
 ↪label='Denoised Audio')  # Updated to use waveshow
plt.title('Denoised Audio by Subspace Filtering')
plt.legend()
plt.tight_layout()
plt.show()
```



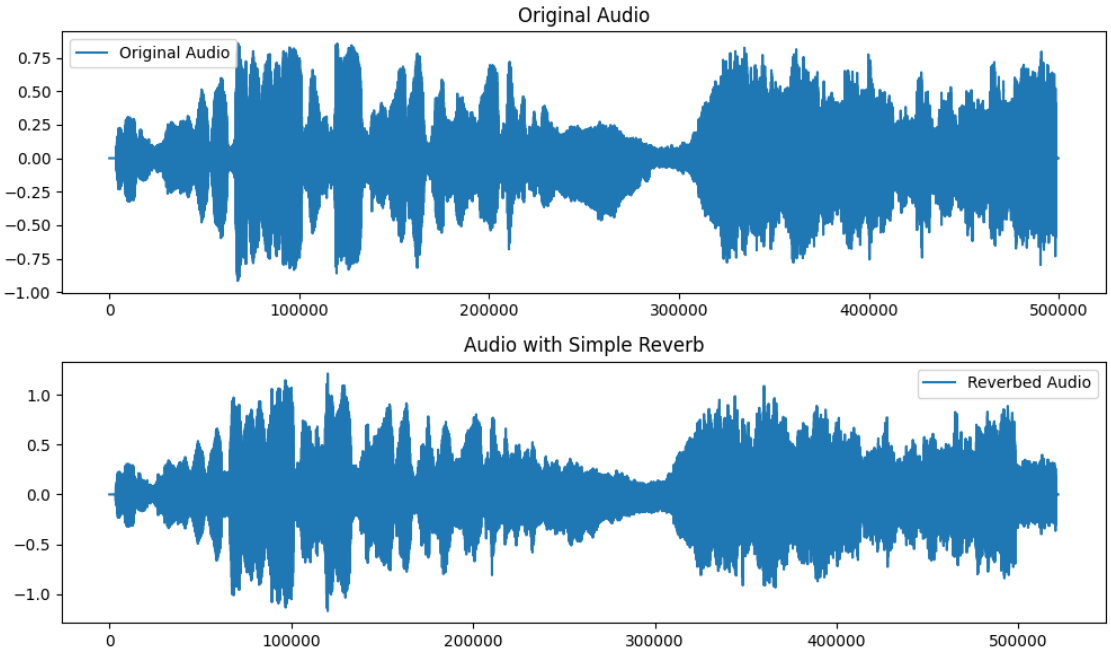
2 basic example of algorithmic reverb using Python

[ ]:
```
!pip install pydub
```

```
Collecting pydub
  Downloading pydub-0.25.1-py2.py3-none-any.whl.metadata (1.4 kB)
Downloading pydub-0.25.1-py2.py3-none-any.whl (32 kB)
Installing collected packages: pydub
Successfully installed pydub-0.25.1
```

[17]:
```
import numpy as np
import librosa
```

```
import matplotlib.pyplot as plt
import soundfile as sf

def simple_reverb(signal, sr, delay_ms, decay):
    """
    Apply a simple reverb effect to the signal
    :param signal: Input audio signal
    :param sr: Sampling rate
    :param delay_ms: Delay of the reverb in milliseconds
    :param decay: Decay factor of the reverb tail
    :return: Signal with reverb applied
    """
    delay_samples = int(sr * delay_ms / 1000)
    output = np.zeros(len(signal) + delay_samples)
    output[:len(signal)] = signal

    # Apply reverb effect
    for i in range(len(signal)):
        output[i + delay_samples] += signal[i] * decay

    return output

# Load an audio file
audio, sr = librosa.load('/content/original.mpeg', sr=None)

# Apply simple reverb
reverbed_audio = simple_reverb(audio, sr, delay_ms=500, decay=0.5)

# Plot the original and reverbed audio
plt.figure(figsize=(10, 6))
plt.subplot(2, 1, 1)
plt.plot(audio, label='Original Audio')
plt.title('Original Audio')
plt.legend()
plt.subplot(2, 1, 2)
plt.plot(reverbed_audio, label='Reverbed Audio')
plt.title('Audio with Simple Reverb')
plt.legend()
plt.tight_layout()
plt.show()
```



[ ]:
```python
import librosa
import numpy as np
import soundfile as sf
from pydub import AudioSegment

def simple_reverb(signal, sr, delay_ms, decay):
    delay_samples = int(sr * delay_ms / 1000)
    output = np.zeros(len(signal) + delay_samples)
    output[:len(signal)] = signal

    # Apply reverb effect
    for i in range(len(signal)):
        output[i + delay_samples] += signal[i] * decay

    return output

# Load an audio file
audio, sr = librosa.load('/content/original.mpeg', sr=None)

# Apply simple reverb
reverbed_audio = simple_reverb(audio, sr, delay_ms=500, decay=0.5)

# Convert the float audio array to int16 format
audio_int16 = np.int16(reverbed_audio / np.max(np.abs(reverbed_audio)) * 32767)

# Create a pydub audio segment
```

```
audio_segment = AudioSegment(
    data=audio_int16.tobytes(),
    sample_width=2,  # Corresponds to audio_int16
    frame_rate=sr,
    channels=1  # or 2 for stereo
)

# Export the audio segment to an MP3 file
output_path = '/content/reverbed_audio.mpeg'
audio_segment.export(output_path, format='mpeg')
```

[ ]: <_io.BufferedRandom name='/content/reverbed_audio.mpeg'>

Module 5: Audio Data Visualization and Analysis

This module covers essential techniques and methods for visualizing and analyzing audio data, emphasizing the application of these techniques in genre detection and classification. It provides a comprehensive overview of the tools and methods used to extract meaningful information from audio signals.

5.1 Visualization Techniques

1. Scalogram: Understanding Wavelet Transforms

- **Scalogram**: A visual representation of a wavelet transform that shows how the frequency content of a signal changes over time. It is particularly useful for analyzing non-stationary signals where frequency components vary over time.
- **Wavelet Transform**: Unlike Fourier transforms, which only offer frequency information, wavelets provide both frequency and time information, making them ideal for audio analysis. A wavelet transform decomposes a signal into a series of wavelets, unlike Fourier transforms that use sine and cosine waves.

  Mathematical Foundation:

$$CWT(t,s) = \int x(\tau).\frac{1}{\sqrt{s}}\,\psi^*(\frac{\tau - t}{s})d\tau$$

  where CWT is the continuous wavelet transform, x(τ) is the signal, ψ is the mother wavelet, and s is the scale factor.

- **Applications**: Scalograms are used in music to analyze rhythms, detect transient features, and identify musical notes that vary over time.

2. Advanced Visualization Techniques for Audio Analysis

- **Spectrogram**: Displays frequency content over time by showing intensity (amplitude squared) at various frequencies, providing a 3D surface within a 2D view.
- **Sonogram**: Similar to a spectrogram but primarily visualizes the intensity of frequencies using different colors rather than surface height.

5.2 Genre Detection and Classification

1. Feature Extraction for Genre Classification

- **Feature Extraction**: The process of deriving parameters suitable for distinguishing between different types of audio content, particularly different music genres.

  Key Audio Features:

  - **Spectral Features**: Such as spectral centroid, spectral bandwidth, spectral flatness, and spectral rolloff.
  - **Rhythm Features**: Beat, tempo, and rhythm patterns.
  - **Timbral Texture Features**: Zero-crossing rate, mel-frequency cepstral coefficients (MFCCs).
- **Feature Vector**: A numerical representation of an audio file's characteristics, used to train machine learning models for classification tasks.

2. Practical Applications of Genre Visualization

- **Visualization in Machine Learning**: Visualizing feature distributions and classification boundaries helps understand how well different genres are separated in the feature space.
- **Confusion Matrix**: A useful tool in classification tasks to visualize the performance of an algorithm, showing how often predictions match the actual genre labels.

Algorithms and Process Methods

1. Classification Algorithms

- **k-Nearest Neighbors (k-NN)**: A simple, non-parametric method used for classification by comparing the feature vectors of k nearest neighbors.
- **Support Vector Machines (SVM)**: Effective for high-dimensional spaces, SVMs are used for both classification and regression tasks but are particularly popular in classification problems.
- **Deep Learning Models**: Such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), which can capture complex patterns in spectral and temporal features of audio data.

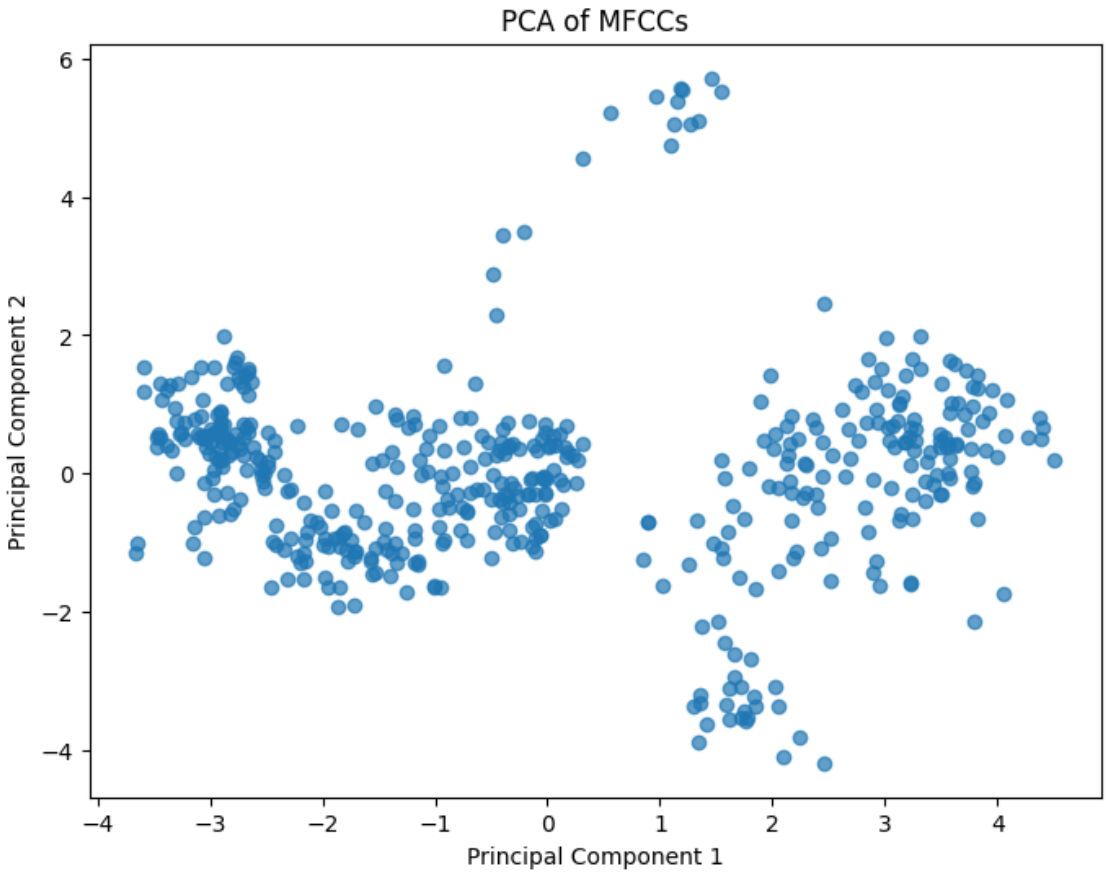
2. Process Workflow

1. **Data Preprocessing**: Convert audio files into a uniform format, sample rate, and duration.
2. **Feature Extraction**: Compute relevant features from each audio segment.
3. **Data Visualization**: Use techniques like PCA (Principal Component Analysis) or t-SNE (t-Distributed Stochastic Neighbor Embedding) to reduce dimensionality for visualization.
4. **Model Training**: Train classification models using the extracted features.
5. **Evaluation**: Use metrics like accuracy, precision, recall, and F1-score to evaluate model performance.
6. **Visualization of Results**: Use plots like confusion matrices and error rate diagrams to visualize and interpret the results.

io-data-visualization-and-analysis

September 7, 2024

[1]:
```
import librosa
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

# Load audio file
audio, sr = librosa.load('/content/original.mpeg')

# Extract features
mfccs = librosa.feature.mfcc(y=audio, sr=sr, n_mfcc=13)

# Scale features
scaler = StandardScaler()
mfccs_scaled = scaler.fit_transform(mfccs.T)

# PCA for visualization
pca = PCA(n_components=2)
mfccs_pca = pca.fit_transform(mfccs_scaled)

# Plot
plt.figure(figsize=(8, 6))
plt.scatter(mfccs_pca[:,0], mfccs_pca[:,1], alpha=0.7)
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('PCA of MFCCs')
plt.show()
```

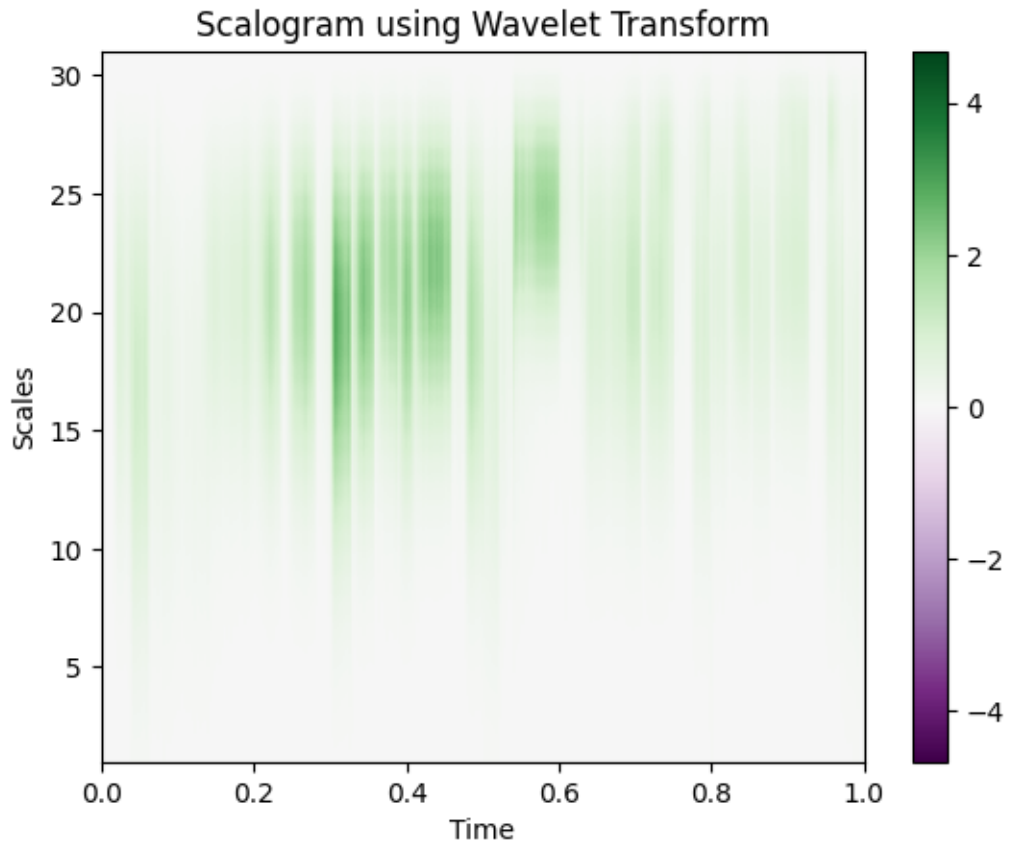


1 1. Creating a Scalogram using Wavelet Transforms

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
import librosa
from scipy.signal import cwt, ricker

# Load an audio file
audio, sr = librosa.load('/content/original.mpeg', duration=5.0)  # Adjust the␣
 ↪path and duration as needed

# Define wavelet widths
widths = np.arange(1, 31)

# Perform the Continuous Wavelet Transform (CWT)
cwtmatr = cwt(audio, ricker, widths)

# Plotting the scalogram
```

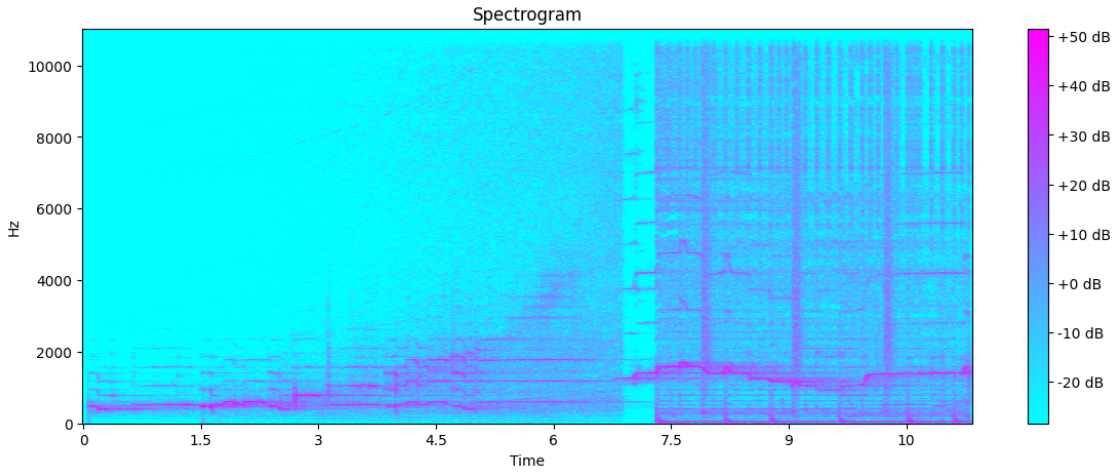
```
plt.imshow(np.abs(cwtmatr), extent=[0, 1, 1, 31], cmap='PRGn', aspect='auto',
           vmax=abs(cwtmatr).max(), vmin=-abs(cwtmatr).max())
plt.title('Scalogram using Wavelet Transform')
plt.ylabel('Scales')
plt.xlabel('Time')
plt.colorbar()
plt.show()
```

```
<ipython-input-2-704ad93d3565>:13: DeprecationWarning: scipy.signal.cwt is
deprecated in SciPy 1.12 and will be removed
in SciPy 1.15. We recommend using PyWavelets instead.

  cwtmatr = cwt(audio, ricker, widths)
```



2 2. Advanced Visualization Techniques for Audio Analysis

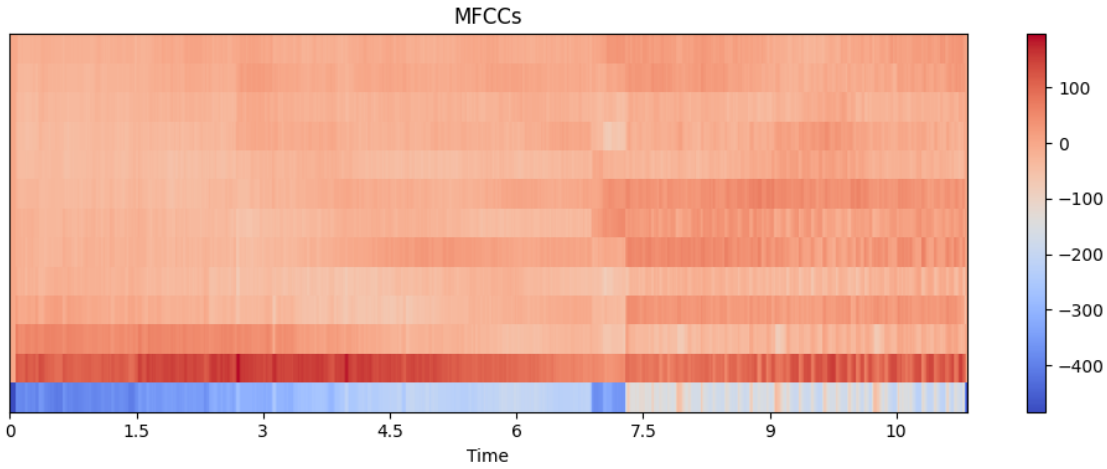
[3]:
```
import librosa.display

# Load audio
audio, sr = librosa.load('/content/original.mpeg')

# Compute the Short-Time Fourier Transform (STFT)
X = librosa.stft(audio)

# Convert the magnitude to decibel units
Xdb = librosa.amplitude_to_db(abs(X))

# Plot the spectrogram
plt.figure(figsize=(14, 5))
librosa.display.specshow(Xdb, sr=sr, x_axis='time', y_axis='hz', cmap='cool')
plt.colorbar(format='%+2.0f dB')
plt.title('Spectrogram')
plt.show()
```



3 3. Feature Extraction for Genre Classification

[5]:
```
import librosa
import matplotlib.pyplot as plt
import librosa.display

# Load an audio file
audio_path = '/content/original.mpeg'  # Make sure to use the correct path to␣
 ↪your audio file
```

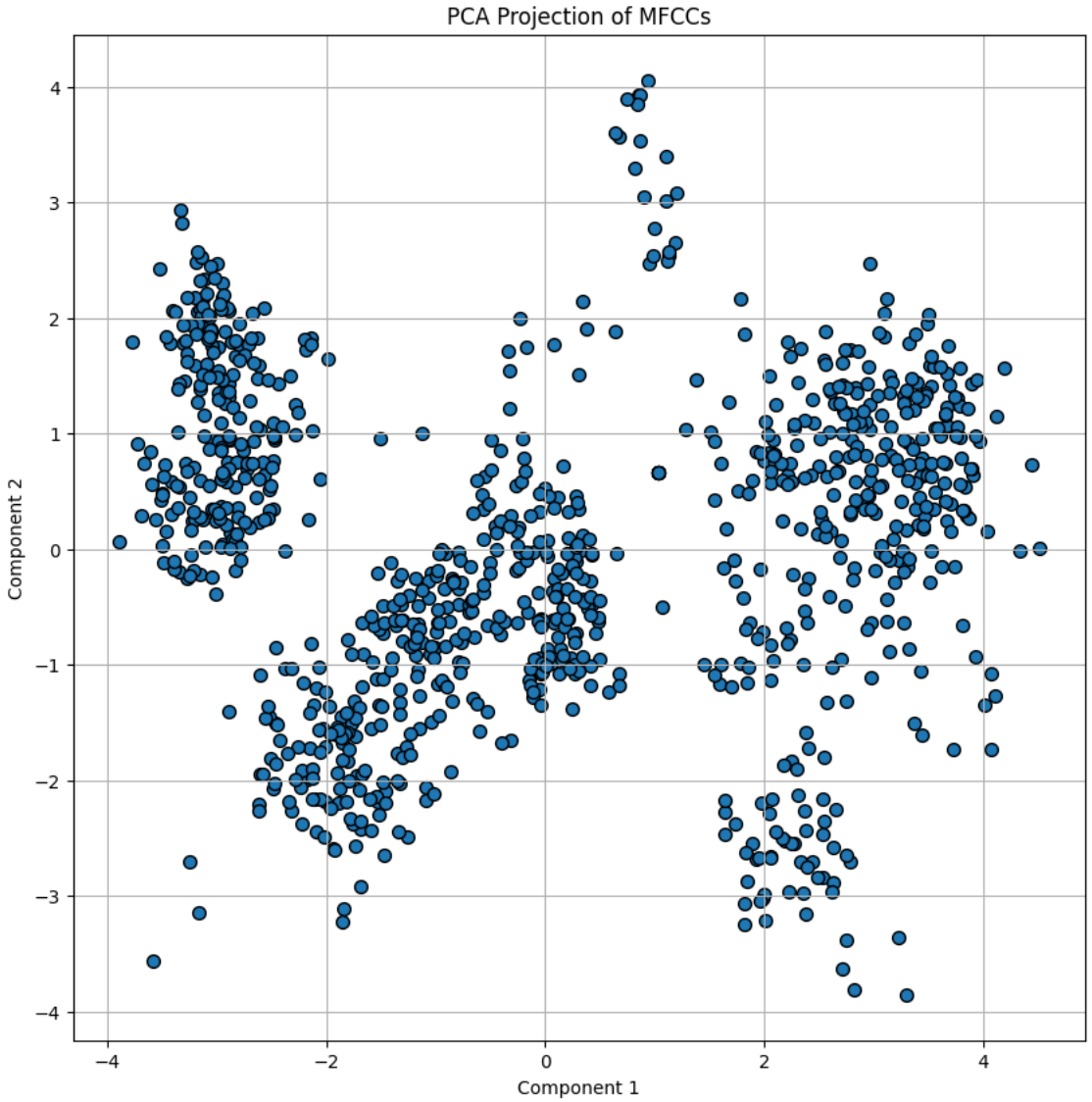
```
audio, sr = librosa.load(audio_path, sr=None)  # Automatically uses the native␣
↪sampling rate

# Extract MFCCs using keyword arguments
mfccs = librosa.feature.mfcc(y=audio, sr=sr, n_mfcc=13)

# Visualizing MFCCs
plt.figure(figsize=(10, 4))
librosa.display.specshow(mfccs, sr=sr, x_axis='time')
plt.colorbar()
plt.title('MFCCs')
plt.tight_layout()
plt.show()
```



4 4. Practical Applications of Genre Visualization

[6]:
```
from sklearn.decomposition import PCA

# Assuming 'mfccs' is extracted as shown above
mfccs_scaled = StandardScaler().fit_transform(mfccs.T)

# Apply PCA
pca = PCA(n_components=2)
projected = pca.fit_transform(mfccs_scaled)

# Plot
plt.figure(figsize=(10, 10))
plt.scatter(projected[:, 0], projected[:, 1], edgecolor='k', s=50)
plt.xlabel('Component 1')
```

```
plt.ylabel('Component 2')
plt.title('PCA Projection of MFCCs')
plt.grid(True)
plt.show()
```



Let's delve into the practical implications and significance of the four processes I described: creating a scalogram using wavelet transforms, generating advanced audio visualizations (like spectrograms), extracting audio features for genre classification, and utilizing dimensionality reduction for genre visualization. Each of these processes serves specific purposes in audio analysis and engineering, enhancing our understanding and manipulation of sound.

1. Scalogram using Wavelet Transforms Output Explanation:

A scalogram provides a time-frequency representation of a signal, displaying how different frequency

components evolve over time. The output is a visual plot where the x-axis represents time, the y-axis represents scales (or frequencies), and the color or intensity represents the amplitude of the wavelet coefficients. Importance and Applications:

- Scalograms are used to analyze non-stationary signals where frequency components vary over time, such as in speech, music, and other natural sounds.
- In music production to detect rhythmic patterns, in speech analysis to understand prosody and intonation, and in environmental sound analysis to identify and classify natural sounds.
- Importance: They allow for the precise analysis of signals whose spectral properties change over time, providing critical insights that are not possible with traditional Fourier transforms.

2. Advanced Visualization Techniques (Spectrogram) Output Explanation:

A spectrogram is another time-frequency representation, but it uses Fourier transform methods to show how the intensity (power) of various frequencies varies over time. The output visualizes frequencies on the y-axis, time on the x-axis, and the energy or power of frequency components using color intensity. Importance and Applications:

- To visually analyze the spectral density of signals. Spectrograms are crucial for identifying how sound spectra evolve in music, speech, and ambient sounds.
- In audio editing and engineering for enhancing recordings, in speech recognition technologies to improve accuracy, and in acoustic ecology to study environmental sounds.
- Importance: Spectrograms provide a detailed view of the signal's spectral richness and time-evolving behavior, facilitating the identification of harmonics, noise, and other spectral characteristics crucial for audio processing.

3. Feature Extraction for Genre Classification (MFCCs) Output Explanation:

Mel Frequency Cepstral Coefficients (MFCCs) capture the timbral aspects of sound. They summarize the power spectrum of a signal based on a linear cosine transform of a log power spectrum on a nonlinear mel scale of frequency. Importance and Applications:

- MFCCs are effective for capturing the basic properties of sound, such as timbre, which are crucial for distinguishing between different musical genres or different speakers in speech analysis.
- In music information retrieval (MIR) for genre classification, in speech recognition to distinguish between different speakers, and in sound synthesis to model sound textures.
- Importance: MFCCs reduce the complexity of the signal without losing key spectral properties, making them highly efficient for training machine learning models in classification tasks.

4. Dimensionality Reduction for Genre Visualization (PCA) Output Explanation:

Principal Component Analysis (PCA) reduces the dimensionality of the feature space while attempting to preserve the variance (information) of the dataset. The output is a 2D or 3D scatter plot that highlights how instances (audio tracks) cluster based on their genre. Importance and Applications:

- PCA is used to simplify the complexity in high-dimensional data (like MFCCs) to two or three principal components that can be easily visualized and analyzed.
- In exploratory data analysis to find patterns in music datasets, in machine learning to reduce overfitting by eliminating noisy or unnecessary features, and in user interfaces to visualize differences in music tracks based on their acoustic features.

- Importance: By reducing the dimensionality of data, PCA helps in visualizing and understanding the underlying structure of complex datasets, facilitating more informed decision-making in audio classification and other machine learning tasks.

Module 6: Modern Audio Engineering with Machine Learning

This module explores the integration of machine learning with audio engineering, focusing on the fundamental concepts, applications, and specific machine learning techniques for audio analysis. It also outlines some practical projects that leverage machine learning to solve common problems in audio processing.

Introduction to Machine Learning in Audio

Basics of Machine Learning:

- **Definition**: Machine learning (ML) is a branch of artificial intelligence that involves teaching computers to learn from data and make decisions or predictions based on that data without being explicitly programmed.
- **Key Concepts**:
  - **Training**: The process of teaching a machine learning model using labeled data.
  - **Inference**: Applying the trained model to new data to make predictions.
  - **Overfitting and Underfitting**: Common issues in machine learning where a model is too complex or too simple for the data.

Applications in Audio Engineering and Acoustics:

- **Noise Reduction**: Machine learning models can automatically identify and remove unwanted noise from audio signals.
- **Echo Cancellation**: Using adaptive filters in ML to enhance communication in real-time audio applications.
- **Sound Localization and Tracking**: Implementing ML algorithms to determine the position of sound sources in a space.

Machine Learning Techniques for Audio Analysis

- **Supervised and Unsupervised Learning Models:**

  - **Supervised Learning**: Involves training a model on a labeled dataset, where the input data and the corresponding output are known. Commonly used in tasks like genre classification or speech recognition.
  - **Unsupervised Learning**: Works with datasets without labeled responses, aiming to find the underlying patterns or structures. Used for clustering similar sounds or identifying unique sound features.

- **Neural Networks and Deep Learning in Audio Classification:**

  - **Neural Networks**: Composed of layers of nodes that mimic the human brain's neurons, used extensively in audio processing to capture complex patterns.
  - **Deep Learning**: A subset of ML based on artificial neural networks with multiple layers (deep networks), excellent for handling large datasets and complex audio processing tasks like feature extraction and classification.
  - **Applications**: From identifying music genres to detecting emotions in speech, deep learning models are reshaping how we interact with audio content.