Audio Effects Application

Group 10:
Ömer Faruk BAŞARAN - 21050111041
Ali Kayra - 22050111018
Hatice ÇAM - 21050111042
Cihan ATAŞ - 21050111078
Ferit AKYILDIZ - 22050111036

This project focuses on developing an audio application that allows users to load or record audio, apply effects, adjust volume and visualize waveforms in real-time. The application is built with Python, using libraries such as `PyQt5` for the GUI, `pygame` for audio playback, `sounddevice` for recording, `librosa` for MP3 parsing, and `matplotlib` for waveform visualization. The user can choose between original audio, processed audio with an effect, and can pause or stop playback at any point. Waveforms for both original and processed data are plotted in a window.

In our application various digital signal processing techniques are demonstrated, including convolution for echo and reverb, and a low-pass filter for bass. Our application provides an accessible, user-friendly interface for experimenting with real-time audio effects.

1 Introduction

Audio signal processing often works with noise, distortion, and fluctuating input levels, which can distort essential features of the waveform. Implementing real-time effects requires efficient algorithms, since any latency or delay can significantly degrade the user experience. In response to these challenges, our project provides an accessible, interactive application that seamlessly handles both recorded and live audio, giving users the tools to visualize and manipulate signals effectively. This technology is used by many modern applications. From voice calls and music production to synthetic speech. An app like this makes it easy for anyone, from the average person to experts, to use audio and explore the use of audio effects.

2 Methodology

- **GUI:** The application uses the `PyQt5` framework for its GUI.
- **Audio Playback & Recording:** Managed using `pygame.mixer` and `sounddevice`.
- **Effects:** Implemented via digital signal processing techniques. For instance:
  - **Echo:** Convolution with an impulse response that contains a delay.
  - **Bass:** A low-pass Butterworth filter emphasizing lower frequencies.
  - **Reverb:** Convolution with an artificially generated impulse response simulating reflections.
- **Visualization:** `matplotlib` is used to plot waveforms.

2.1 Audio Representation and Basic Notation

Let $x[n]$ be the discrete-time audio signal, where n indexes time in samples, at a sampling rate of f_s. After processing, we obtain $y[n]$, the output signal. Most of the time-domain effects use *discrete convolution*:

$$y[n] = x[n] * h[n] = \sum_{k=-\infty}^{\infty} x[k]\, h[n-k],$$

where $h[n]$ is the impulse response defining the effect.

2.2 Echo Effect

An *echo* is created by adding a delayed, attenuated copy of the original signal. We construct a simple impulse response:

$$h[n] = \begin{cases} 1, & \text{if } n = 0, \\ \alpha, & \text{if } n = d, \\ 0, & \text{otherwise,} \end{cases}$$

where d is the delay in samples and α is the echo attenuation factor. The output is:

$$y[n] = x[n] * h[n].$$

In code, the delayed impulse response vector is convolved with the original data.

2.3 Bass Effect (Low-Pass Filter)

To achieve lower frequencies, a 4th-order Butterworth low-pass filter is applied. A Butterworth filter of order N has the magnitude response:

$$|H(\omega)| = \frac{1}{\sqrt{1 + \left(\frac{\omega}{\omega_c}\right)^{2N}}}.$$

SciPy's butter function computes the discrete-time filter coefficients (b, a). We then filter the audio via the standard linear difference equation:

$$y[n] = \sum_{k=0}^{M} b[k] \cdot x[n-k] \; - \; \sum_{k=1}^{N} a[k] \cdot y[n-k].$$

In code, this is done with:

$$\text{processed} = \text{lfilter}(b, a, x[n]).$$

2.4 Reverb Effect

Reverb simulates the reflection of sound waves in an acoustic environment. The code constructs a custom impulse response $h[n]$ of length `ir_length` $= 0.5 \times f_s$. The impulse response starts with $h[0] = 1$ and each subsequent sample decays exponentially plus a small random amount:

$$h[i] = h[i-1] \cdot \text{decay} + \mathcal{N}(0, \sigma^2).$$

We convolve $x[n]$ with $h[n]$ to get a "wet" signal $y_{\text{wet}}[n]$:

$$y_{\text{wet}}[n] = x[n] * h[n].$$

Finally, the output is a mix of the dry and wet signals:

$$y[n] = \text{dry\_wet} \cdot y_{\text{wet}}[n] + (1 - \text{dry\_wet}) \cdot x[n].$$

2.5 Data and Implementation

The dataset in this project consists of either recorded audio (via microphone) or user-selected audio files (`.wav` or `.mp3`). After loading, the raw audio samples are stored in a NumPy array. Preprocessing steps include:

- **Downmixing to Mono**: If the audio is stereo, the channels are averaged to obtain a single-channel array.
- **Normalization**: The sample values are scaled (divided by 32767 if they were 16-bit integers) to float data type in the range of $[-1.0, +1.0]$.
- **Optional Smoothing or Filtering**: In some scenarios, a median filter or a low-pass filter might be applied to reduce noise before further processing.

2.6 Python Implementation

Our application uses a combination of Python libraries including `NumPy`, `SciPy`, `SoundDevice`, `PyGame`, and `librosa` for audio I/O and signal processing. The main steps are:

1. **Load Audio**: Depending on the file extension, we either load a `.wav` with Python's `wave` module or an `.mp3` via `librosa`.
2. **Store to NumPy**: The loaded samples are converted into a NumPy array, ensuring a common data type (`float32`).
3. **Process or Apply Effects**: We convolve the signal with an impulse response for echo/reverb or apply infinite impulse response filters for bass enhancement.
4. **Convert for Playback**: The processed NumPy array is converted to 16-bit samples for `PyGame`'s `Sound` objects.

```
Import numpy as np
Import pygame

data = np.loadtxt(data.csv)

normalized_data = data.astype(np.float32) / 32767.0

int16_data = (normalized_data * 32767).astype(np.int16)
sound_array = pygame.sndarray.make_sound(int16_data)

sound_array.play()
```

Listing 1: Key Python Code

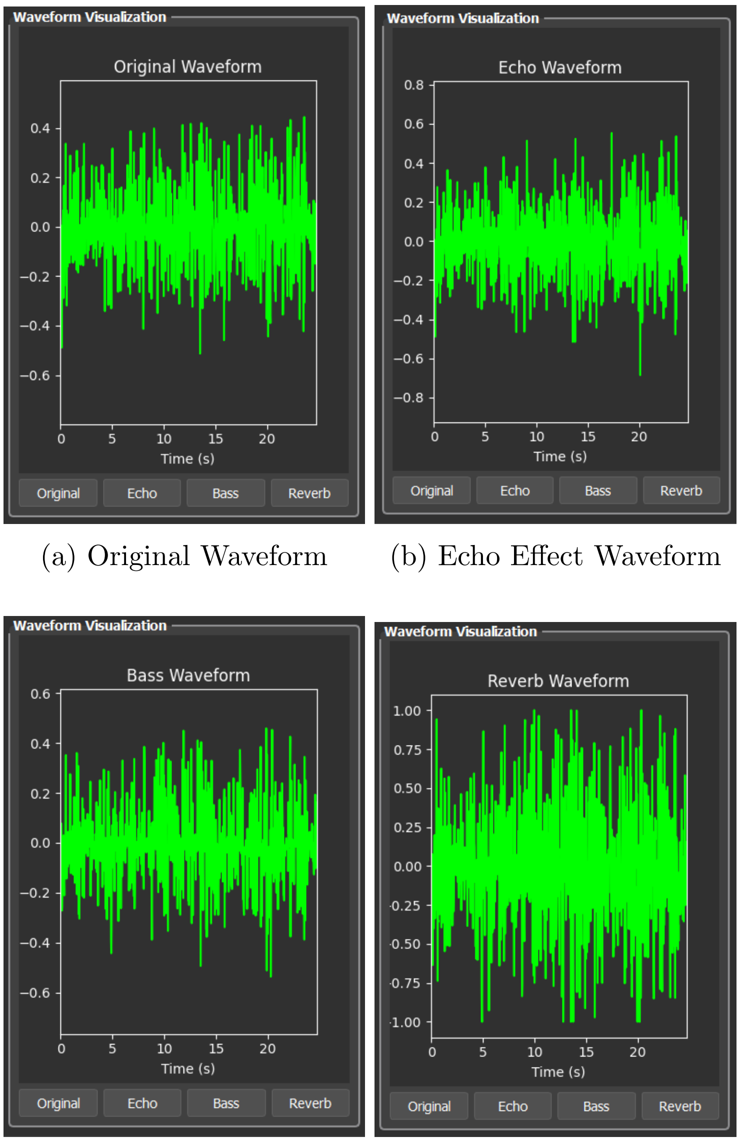
In the code above, `np.loadtxt` shows how numeric data can be read from a CSV file. In the actual application, this mechanism is extended to loading, normalizing, and processing any audio signal prior to playback or visualization.

3 Results and Discussion

The application displays real-time waveforms for both original and processed audio. Users can switch between the following effects:

- **Echo:** Noticeable delays in the signal.
- **Bass:** Enhanced low frequencies due to the low-pass filter.
- **Reverb:** More spacious sound.

Recording functionality also allows quick capture of audio data through a microphone and provides immediate playback or effect application.



(a) Original Waveform (b) Echo Effect Waveform

(c) Bass Effect Waveform (d) Reverb Effect Waveform

Figure 1: Waveform Visualizations for Different Audio Effects

4 Conclusion

In this project, we developed an Audio Effects Application that has a user interface with powerful audio processing capabilities. Built using Python and making use of libraries such as `PyQt5`, `pygame`, `sounddevice`, and `matplotlib`, the application allows users to load or record audio, apply effects like echo, bass enhancement, and reverb, and visualize waveforms in real-time. Overall, this application demonstrates the versatility of Python in audio signal processing.

References

- https://www.pygame.org/docs/ref/mixer.html

  - https://python-sounddevice.readthedocs.io/
  - https://matplotlib.org/stable/users/index
  - https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.butter.html
  - https://www.sciencedirect.com/topics/engineering/audio-signal-processing
  - https://www.electronics-tutorials.ws/filter/filter$_8.html$

- GitHub Repository