AUDIO FILES COMPARATOR USING WAVELET TRANSFORM AND SIMILARITY METRICS

BY

LEE DA LONG

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF INFORMATION TECHNOLOGY (HONOURS) COMPUTER ENGINEERING

Faculty of Information and Communication Technology

(Kampar Campus)

FEBRUARY 2025

COPYRIGHT STATEMENT

This Final Year Project report is submitted in partial fulfillment of the requirements for the degree of Bachelor of Information Technology (Honours) Computer Engineering at Universiti Tunku Abdul Rahman (UTAR). This Final Year Project report represents the work of the author, except where due acknowledgment has been made in the text. No part of this Final Year Project report may be reproduced, stored, or transmitted in any form or by any means, whether electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author or UTAR, in accordance with UTAR's Intellectual Property Policy.

ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude to everyone who has supported me throughout the completion of my first Final Year Project.

First and foremost, I extend my deepest thanks and appreciation to my supervisor, Mr Lee Heng Yew for giving me the opportunity to engage in this project on audio comparison. His guidance has not only deepened my understanding of Signal Processing but also provided me with the chance to expand my academic horizons. A million thanks to you.

I also wish to express my sincerest appreciation to my parents and family for their unconditional love, support, and encouragement throughout this journey. Their continuous belief in my abilities has been a source of strength for me.

Finally, to my dear friends, thank you for your companionship and for understanding by my side during challenging times. Your support and motivation have been invaluable, and I am grateful to have shared this experience with all of you.

ABSTRACT

This project is a development-based project revolving around signal processing. The aim of this project is to develop a program that utilizes continuous wavelet transform (CWT) for audio similarity recognition. Its primary objective is to identify the similarities among audio files with different information such as file names or formats.

In today's diverse musical landscape, songs undergo various interpretations, covered in different languages, or rendered using a myriad of instruments. Compositions may span the spectrum, ranging from performances with real musical instruments to those composed solely of synthesized sounds, typically electronic dance music (EDM).

Furthermore, songs exhibit versatility in their presentation, ranging from vocal renditions accompanied by instruments to whistling, humming or acapella performances. The evolution of music has also fostered the emergence of mashups and remixes, where distinct tracks seamlessly blend together to create new compositions. Despite these variations, the tunes or pitches of songs remain recognizable to the human ear and even audio detection algorithms. With the proliferation of digital music, people download songs from music applications or the internet, whether for personal listening in vehicles or to play in parties. However, these downloaded songs may vary depending on their file names and formats. Consequently, this project aims to identify identical or akin songs with various information and display out the percentage of differences between the audio files. The project's methodology centres on Python programming, where comparisons of audio similarities will be conducted.

Area of Study (Minimum 1 and Maximum 2): Signal Processing

Keywords (Minimum 5 and Maximum 10): Audio Comparison, Music Cover Detection, Wavelet Transform, MFCC Analysis, Python Programming, User-Friendly Application

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

π	Pi
ϕ	Phi
ψ	Phi
τ	Tau

LIST OF ABBREVIATIONS

CWT	Continuous Wavelet Transform
DFT	Discrete Fourier Transform
DTW	Dynamic Time Warping
FFT	Fast Fourier Transform
GUI	Graphic User Interface
MFCC	Mel-Frequency Cepstral Coefficient
SNR	Signal-to-Noise Ratio
STFT	Short-Time Fourier Transform

Chapter 1
Introduction

1.1 Problem Statement

In this modern era of digital music consumption, users frequently download and store songs from various platforms and devices such as YouTube, SoundCloud, Spotify, and many more. However, these audios often differ significantly in attributes like their format, bitrates and even time-alignment discrepancies. These variations can complicate tasks such as verifying the audio integrity, assessing audio quality, and making informed decisions about song selection. Existing audio comparison tools are limited in their ability to accurately compare and analyze these differences in audio files. Additionally, they often fail to detect more complex scenarios such differentiating songs covered by different instruments, languages, or even covered by different artists. This project addresses these gaps by developing a wavelet-based audio files comparator capable of identifying formats, time-shifts, cover versions, and other content-based differences.

1.2 Motivation

Music is one of the most prominent forms of digital audio content, available in a variety of genres such as classic, folk, pop, and EDM. With the proliferation of digital platforms, users now have access to vast music libraries that allows us to download songs to listen at any place or time [10]. However, users often encounter files with inconsistent quality, timing shifts, or unrecognized covers. For instance, Spotify suggests at least 160kbps while YouTube recommends at least 128kbps for streaming [7]. This variation poses a challenge for users who want to ensure that they are selecting the highest quality audio files or identifying the correct version of the song. The motivation behind this project is to implement a program that not only compares these audio files across various attributes but also delves deeper into content-based comparison, enabling the identification of similarities and differences even in complex cases like covers or remixes.

1.3 Objectives

The primary objective of this project is to develop a program that can analyze and compare audio files from designated folders using wavelet-based pre-processing and similarity metrics. The key functions of the program will include:

1. Detecting differences in time shifts and alignment between audio files.
    - The program will identify temporal discrepancies such as delays and unsynchronized sequences between two audio files.
    - The function is expected to return a higher similarity between two similar audio files despite having different duration or delays.

2. Identifying different cover versions of the same audio file.
    - The program will distinguish between the original and covered versions of two audio files by analyzing variations in artists and languages.
    - The function is expected to return a moderate to high similarity between two similar audio files with different versions.

3. Utilizing Continuous Wavelet Transform (CWT) for pre-processing to enhance comparison accuracy.
    - The Continuous Wavelet Transform (CWT) will be used to extract time and frequency features for comparison.
    - This step ensures more reliable similarity measurements compared to raw signal analysis.

1.4 Project Scope and Direction

This project involves the design, development, and testing of the Python program aimed at comparing audio files using wavelet-based techniques. The scope includes preprocessing audio files using the Mexican Hat wavelet and the use of similarity metrics such as MFCC and cross-correlation. The focus is on developing a reliable tool that has a lightweight computation suitable for a personal PC or laptop to compare songs in different formats, time-shifts, and cover versions. The scope excludes demanding computational methods such as Dynamic Time Warping (DTW) and machine learning due to hardware constraints. The output will include similarity scores and basic visualizations for user-friendly interpretation and analysis of the audio comparison.

1.5 Contributions

The primary contributions of this project include the validation and enhancement of audio comparison techniques, particularly using wavelet-based methods and similarity metrics. The project advances accessible audio comparison by proving the efficiency of the Continuous Wavelet Transform (CWT) and similarity metrics in environments with limited resource capabilities. Therefore, this project will deliver a user-friendly tool that bypasses the need for high-end computation while maintaining accuracy.

1.6 Report Organization

This format of this report is structured to provide a thorough summary of the development progress and findings of the project. Chapter 1 introduces the problem statement, motivations, objectives, project scope, and contributions of the study. Chapter 2, some previous relevant works on audio comparison have been reviewed while highlighting their strengths and limitations. Chapter 3 details the methodology and how the current project aims to address them. Chapter 4 explains the system design and the whole architecture used. Chapter 5 briefs the system implementation based on hardware and software used. Chapter 6 describes the test cases used and their results while determining the project challenges. Finally, Chapter 7 concludes the report's summary of findings, potential recommendations, and directions for further research.

Chapter 2
Literature Review

2.1 Previous Works on Audio Files Comparison

2.1.1 Wavelet Transform Analysis

Wavelet transforms are time-localized wave oscillations with two basic properties which are scale and position. By adjusting the scale parameter, the wavelet transform may identify and locate different frequencies in the signal [1]. By decomposing signals into a series of wavelet coefficients across different scales and frequencies, wavelet analysis captures both time and frequency information with high resolution. This capability enables more robust and efficient methods for audio comparison.

Continuous Wavelet Transform (CWT) allows for the analysis of signals with varying frequencies over time, making it suitable for capturing transient changes and non-stationary behavior in audio recordings. By convolving signals with a continuously varying wavelet function, CWT offers high flexibility and precision in characterizing complex audio features [11]. The equation of CWT is expressed as:

$$T(a,b) = \frac{1}{\sqrt{a}} \int_{-\infty}^{\infty} x(t)\, \psi^* \frac{(t-b)}{a} dt$$

Figure 2.1 Equation of CWT

Where:

- a is the scale parameter to determine the wavelet frequency.
- b is the position parameter that locates the wavelet.
- ψ^* is the complex conjugate of the wavelet function.

2.1.2 Cross-Correlation

Cross-correlation is a similarity metric often used in audio signal processing used to analyze the differences between two various signals by applying a time delay to either one of them [4]. By computing the correlation of samples from two audio segments across varying delays, it identifies the alignment points where patterns output a higher similarity. This approach is particularly useful for detecting shifted or overlapping audio segments.

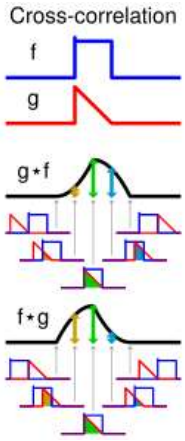


Figure 2.2 Cross-Correlation

2.1.3 Mel-Frequency Cepstral Coefficients (MFCC)

MFCC is one of the methods used to extract features from audio files, especially to analyze different types of music and speech. These are designed to replicate how human ears receive the frequency of sounds. By applying a mel-scale filterbank to the spectrogram of a signal and computing the cepstral coefficients, MFCCs compactly encode the timbral and spectral characteristics of audio analysis tasks such as speech recognition and music genre classification [3]. MFCCs are initially developed for speaker identification and now evolved to modern music analysis such as detecting cover versions of songs.

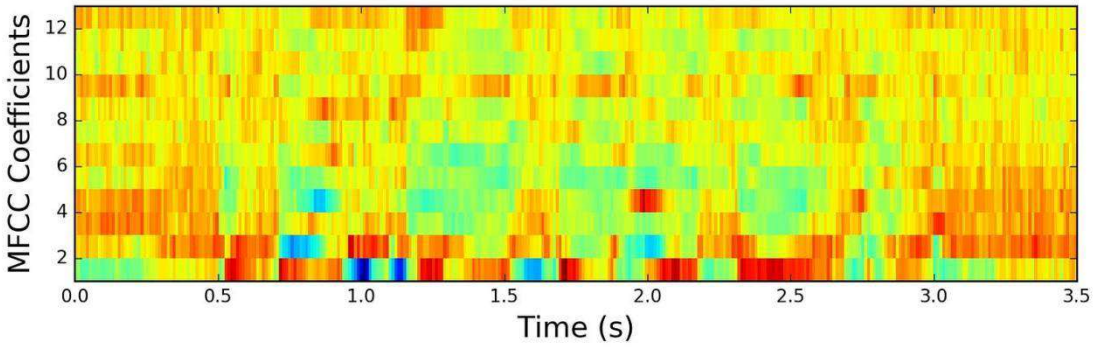


Figure 2.3 Sample of MFCC Visualization

2.1.4 Fast Fourier Transform (FFT)

Fast Fourier Transform (FFT) has been mostly used in audio comparison projects due to its efficiency in analyzing frequency components of audio signals. Its application involves the time and frequency domains of the audio signals, enabling spectral analysis and facilitating comparison between different audio recordings. Common applications of FFT-based techniques include pitch detection, spectrum analysis, and spectral characteristics comparison between audio files [9].

FFT can also compute Discrete Fourier Transform (DFT), which is crucial for analyzing the frequency components of discrete-time signals like digital audio waveforms. The DFT equation is expressed as:

$$X(k) = \sum_{n=0}^{N-1} x(n)\, e^{-j\left(\frac{2\pi}{N}\right)nk}$$

Figure 2.4 Equation of DFT

Where:

- N is the fundamental period.
- j is the imaginary unit.

2.1.5 Dynamic Time Warping (DTW)

Dynamic Time Warping (DTW) is an algorithm that is used to analyze the similarity and difference between two signals that might have different speed or tempo. It is particularly useful in aligning time series data, such as wavelet coefficients that are extracted from audio signals. DTW is widely used in applications like speech recognition and audio classification, which are essential for precise alignment of temporal data [13].

The key idea behind DTW is to find the most optimal alignment between two sequences such as audio waveforms by non-linearly warping them in the time dimension [12]. This allows the algorithm to match the similar elements between the sequences, even if they occur at different pitch, timing or tempo. The algorithm computes a cost matrix to find the most minimum distance path between signals to accommodate temporal distortions without requiring uniform signal lengths.

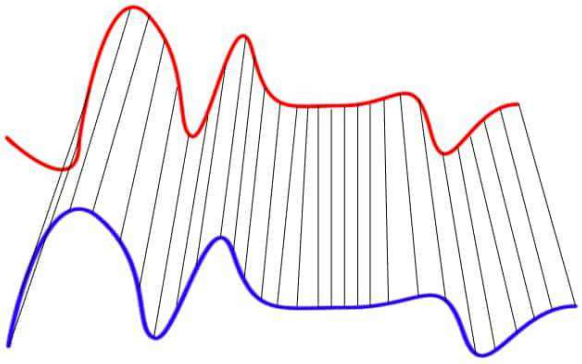


Figure 2.5 Dynamic Time Warping

2.2 Strengths of Previous Works

Wavelet transform provides a high multiscale resolution which differentiates coarse and fine scales. The coarse scales capture broad spectral trends, while fine scales isolate abrupt changes. CWT is efficient in analyzing non-stationary signals such as audio, as it provides a time-frequency representation that adapts to variations within the signal over time. CWT is also noise-resistant to noise due to their ability to decompose signals into different resolution levels. The noise reduction allows selective suppression while preserving the signal's structure [6].

The simplicity of cross-correlation proves to be a widely used tool for time-domain audio comparison. Its computational efficiency allows real-time implementation, even for longer recordings. Studies have demonstrated its robustness in noisy environments, as shown in its ability to accurately detect signal arrival times in underwater acoustic positioning and neutrino telescope calibration (ANTARES/KM3NeT), despite the low SNR and reverberation interference [2].

MFCCs excels in audio comparison due to their computational efficiency. Their mel-scale compression emphasizes mid-frequency bands where human hearing is the most sensitive, which reduces redundancy in raw spectra. The low dimensionality of MFCCs enables real-time processing, which makes them practical for analysis of longer audio files.

2.3 Limitations of Previous Studies

Despite the widespread use of FFT, it struggles with handling non-stationary signals and capturing transient changes in audio characteristics. This is because FFT assumes stationarity in the audio signals analysed, which can lead to overlooking important temporal variations in audio signals that leads to distorted analysis and result visualization. Recent hybrid approaches such as STFT and Constant-Q Transform aim to mitigate these issues but still introduced a high computational overhead due to their complex calculations and adjustments [8].

DTW may have more versatility but still suffers from a high computational complexity of $O(N^2)$, which is very extensive for long signals and limits real-time applications. The excessive warping flexibility forces the alignment of two completely different signals, leading to high similarity scores. Constraints like the Sakoe-Chiba band are often applied to mitigate these problems by limiting the warping path to a predefined region to prevent over-warping, however it only restricts the warping path width to only 10% of the signal length and still cannot cater longer signals [14].

Chapter 3

System Methodology/Approach

3.1 Methods and Approaches

3.1.1 Continuous Wavelet Transform (CWT)

CWT is a robust method to analyze non-stationary signals such as audio files by preprocessing them into time-frequency representations. This method is done using a mother wavelet that is adjusted according to the scales [11].

For audio decomposition, Mexican Hat wavelet, also known as the Ricker wavelet, is chosen as the mother wavelet due to its single oscillation that aligns with transient audio signals. Additionally, it offers a balanced time-frequency localization, which makes it possible to accurately represent audio with both short high-frequency and lengthy low-frequency components. While other wavelets such as Morlet (morl) and Complex Morlet (cmor) offer higher quality in frequency resolution for components, their values tend to be complex which complicates the interpretation for real-world audio processing.

The Mexican Hat wavelet is derived as:

$$W_{(t,\tau)} = \frac{1}{\sqrt{2\pi\tau}}(1 - \frac{t^2}{2\tau^2})e^{-\frac{t^2}{4\tau^2}}$$

Where

- t is the time variable
- τ is the scale parameter

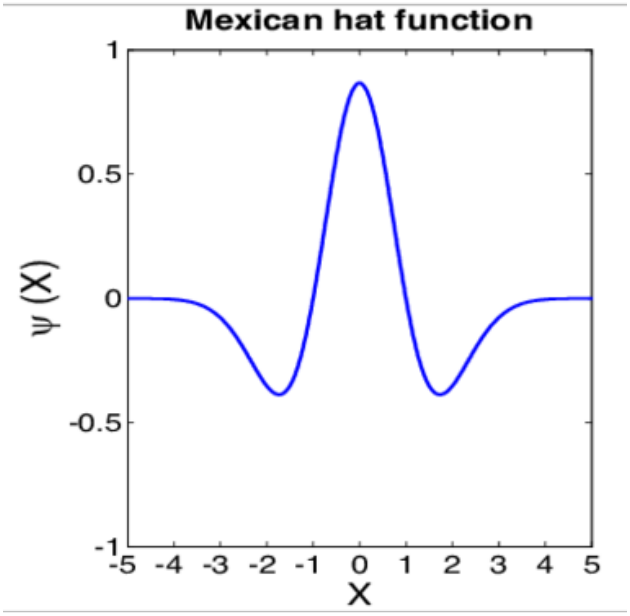


Figure 3.1 Mexican Hat Wavelet

3.1.2 Cross-Correlation

Cross-correlation is a statistical method used to analyze the similarity and differences between two audio signals using a time lag applied to one of them. Unlike autocorrelation which compares a signal to itself, cross-correlation identifies aligned patterns between two distinct signals, which makes it suitable for audio alignment and rhythm analysis [5]. This method will reveal hidden periodic structures by measuring the similarity between a signal and its delayed counterpart. The mathematical expression for cross-correlation is expressed as:

$$R_{xy}[\tau] = \sum_{n=0}^{N-1} x[n] \cdot y[n+\tau]$$

Where

- τ represents the time lag
- x[n] represents the signal at time, t
- N represents the total samples

3.1.3 Mel-Frequency Cepstral Coefficients (MFCC)

MFCCs are designed to estimate the perception and frequency response of the human ear. They are effective for comparing different covers of the same song since they can capture the timbre of songs while reducing pitch and harmonic redundancies. The extraction process begins with computing STFT to obtain the spectrogram of the audio signal. Then, a mel-scale filterbank is applied to warp frequencies into bands with an approximate perception of the human ear. The logarithm of the spectrogram is then computed to decorrelate its features using Discrete Cosine Transform (DCT) [15].

However, practical implementation requires careful parameter selections such as the number of mel bands and computational cost. The DCT step retains the first 12 to 13 coefficients to discard high-order spectral fluctuations.

Chapter 4
System Design

4.1 Graphic User Interface (GUI)

The GUI is an essential part of the Audio Files Comparator, providing users with an intuitive interface to interact with the system. The main interface has buttons that allow users to load audio files, preview audio analysis and initiate comparison. The GUI also includes playback features such as the Play and Stop button for preview. The figure below shows the interface design of the audio files comparator.

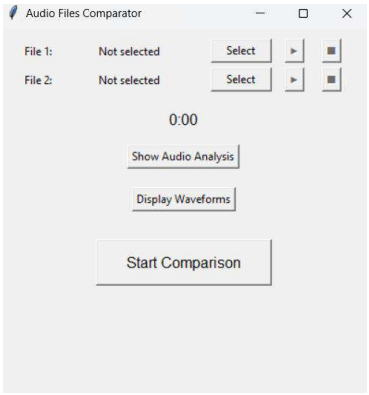


Figure 4.1 GUI of Audio Files Comparator

4.1.1 Interface Design

The GUI is designed using the Tkinter library from Python. The main window features some key elements:

1. Playback Controls

- Two 'Select' buttons allow users to load audio files of a variety of formats such as MP3, WAV, OGG and FLAC files.
- The Play and Stop buttons allow users to enable audio preview for the audio files.
- A timer to display the current playback position of either audio file in MM:SS format.

2. Audio Analysis Tools

- The ‘Show Audio Analysis’ button enables users to display the metadata of both audio files that consists of their duration and sample rate.
- The ‘Display Waveforms’ button will plot the raw waveforms of both audio files using Matplotlib library.

3. Comparison and Results

- The ‘Start Comparison’ button initiates the comparison analysis of both audio files.
- The window will change to display a progress bar that updates during preprocessing.
- The results will be displayed at the last window.

The key elements also featured error handling whenever audio files are not loaded into the program. The program will prompt error messages to remind users to upload their desired audio files for comparison.

4.1.2 Audio Preprocessing

Once the program has been started, users will have the option to load their preferred audio files for comparison. The audio files are loaded using the ‘librosa.load’ function without any fixed sample rate so that the audio files can be loaded with their original sample rate. Each entire audio is stored as a NumPy array, and the sample is captured. The program currently supports only WAV, MP3, OGG and FLAC files as they are the more common audio file formats.

The pygame.mixer module handles the playback functions for the loaded audio files with real-time timer. The playback can be stopped with the Stop button but will also be done automatically when the comparison starts or the window closes.

The loaded audio will then be preprocessed to ensure its suitability for analysis. The loaded audio will first be divided into smaller segments based on a predefined segment duration of 1 second. The segment size is calculated by multiplying the segment duration by the sample rate. The total segments are then determined by dividing the total samples depending on the audio length.

After segmentation, each segment is processed using CWT with Mexican Hat Wavelet (mexh) as the mother wavelet. The CWT then preprocesses the audio segment into wavelet coefficients across a range of scales. These coefficients capture both time and frequency information, which are critical for comparing audio files. These coefficients for all segments are then collected into a 3D array for subsequent analysis.

Once both audio files have been successfully preprocessed into wavelet coefficients collected in 3D arrays, the arrays are then flattened and normalized in the comparison metrics.

The GUI can also catch exceptions during the processes within the audio preprocessing by displaying user-friendly error messages via tk.messagebox.

4.1.3 Data Visualization

After audio preprocessing and comparing using metrics, the similarity percentages will be displayed via the GUI. The results of pattern consistency, timbre similarity and final comparison results are presented with different colors and messages based on the similarity score. A high similarity percentage ranging from 90% to 100%, is shown in green, indicating that the files are the same song. Percentages between 40% to 90% are displayed in orange, suggesting that the songs are likely similar. For scores below 40%, the text is shown in red, signifying that the songs are completely different.

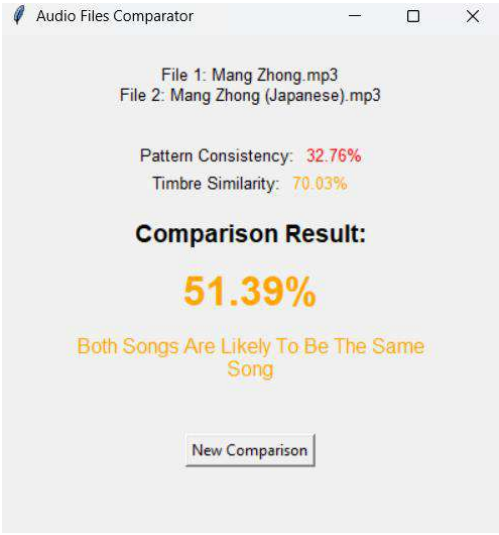


Figure 4.2 Visualization of Similarity Scores

The GUI also enables functions to preview audio analysis such as audio properties, raw waveforms, and wavelet coefficients. The 'Audio Analysis' button enables users to preview the audio properties such as the file name, duration and sample rate.

Figure 4.3 Audio Analysis Window

The 'Display Waveforms' button enables users to preview the raw waveform of both audio files loaded. The raw waveforms are downsampled to balance the computational efficiency and optimized visualization. This allows users to quickly observe the audio waveforms before starting the comparison.

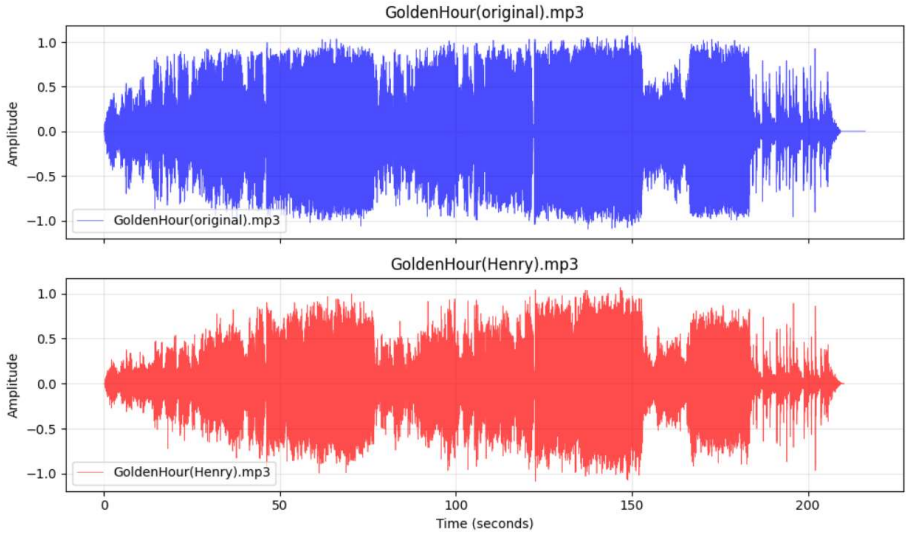


Figure 4.4 Raw Waveform Visualization

4.2 System Architecture Diagram

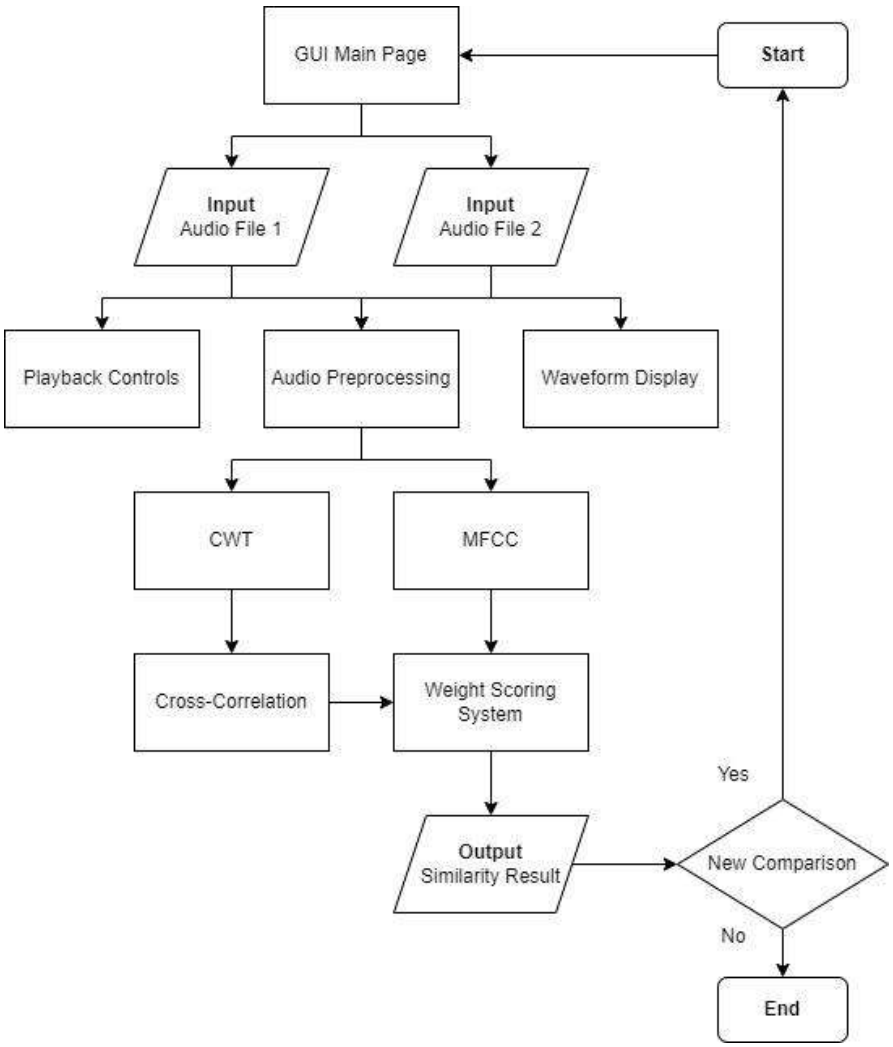


Figure 4.5 Flowchart of Audio Files Comparator

Chapter 5
System Implementation

5.1 Hardware Setup

The hardware involved in this project is only a personal laptop as the project primarily revolves around software implementation.

Description	Specifications
Model	ASUS TUF Dash F15 FX516PC
Processor	Intel Core i5-11300H
Operating System	Windows 11
Graphic	NVIDIA GeForce RTX 3050 Laptop GPU
Memory	16GB RAM
Storage	NVME 512GB SSD

Table 5.1 Specifications of Laptop

5.2 Software Setup / Programming Language

This project utilizes Python 3.12 as the primary programming language for developing the audio files comparator. Python 3.12 offers enhanced performance, improved syntax features, and a variety of libraries, making it well suited for audio processing and wavelet analysis tasks. Key libraries such as *librosa* for audio analysis, *pywt* for wavelet transforms, *numpy* for numerical operations, and many more are integrated into the project to facilitate efficient processing and analysis of audio data.

5.3 Code Implementation

5.3.1 CWT Implementation

The CWT is implemented using a discretized version of the equation from Figure 2.1. The audio signal is first segmented into consecutive non-overlapping segments with a duration of 1 second. For each segment, the CWT is computed using the Mexican Hat wavelet at multiple

scales ranging from 1 to 127, resulting in a set of wavelet coefficients. The coefficients are stored in an array with dimensions (segments, scales, time).

```
num_segments = len(audio_data) // segment_samples

coefficients = []
scales = np.arange(1, 128)

# Process each segment
for seg in range(num_segments):
    start_idx = seg * segment_samples
    end_idx = start_idx + segment_samples
    segment = audio_data[start_idx:end_idx]

    # Perform CWT
    cwt_segment, _ = pywt.cwt(segment, scales, 'mexh')
    coefficients.append(cwt_segment)
```

Figure 5.1 Code Snippet of CWT Implementation

```
[[[-1.78188202e-06 -7.64097204e-04 -5.28277305e-05 ... -6.57626707e-03
   -5.16710896e-03  3.04613076e-02]
  [-6.58309669e-04  8.61601613e-04  1.35850321e-04 ... -1.39223589e-02
    4.21697386e-02 -4.31769900e-02]
  [-6.17861879e-05  2.70212744e-03 -3.74591589e-04 ...  7.23373592e-02
    2.39468105e-02 -9.20117646e-02]
  ...
  [-2.63988599e-02 -2.50439849e-02 -2.37157661e-02 ... -1.20155418e+00
   -1.20097172e+00 -1.24636161e+00]
  [-2.33331490e-02 -2.20856853e-02 -2.22662482e-02 ... -1.20229900e+00
   -1.20406127e+00 -1.23767221e+00]
  [-2.31317692e-02 -2.11666804e-02 -2.04105861e-02 ... -1.21734047e+00
   -1.21933675e+00 -1.26176190e+00]]

 [[ 3.62978876e-03  2.19826791e-02 -1.28137590e-02 ... -3.18521354e-03
   -1.18369830e-03  3.03430832e-03]
  [-4.69566043e-03 -4.29789051e-02  4.80904914e-02 ...  5.65040903e-03
    3.69474903e-04 -5.29112574e-03]
  [-2.86923219e-02 -7.31382146e-02  4.49403301e-02 ...  1.62616856e-02
    7.06273830e-03 -1.37292966e-02]
  ...
  [-1.61621702e+00 -1.64667606e+00 -1.63683128e+00 ... -2.27084197e-02
   -2.53911801e-02 -2.94917058e-02]
  [-1.59408510e+00 -1.62903440e+00 -1.61957121e+00 ... -2.22584698e-02
   -2.58511584e-02 -3.01313046e-02]
  [-1.56054533e+00 -1.58841097e+00 -1.57554352e+00 ... -2.11587045e-02
   -2.25326866e-02 -2.57674754e-02]]
```

Figure 5.2 Sample Set of CWT Coefficients of a Song

5.3.2 Cross-Correlation Implementation

In the context of Audio Files Comparator, cross-correlation is used to compare the wavelet coefficients from the two audio signals and measure their similarity. While this may be useful for audio files comparison, the similarity results are often either too high or too low, such that the results are either 100% or 0% when comparing very different or almost identical signals.

In the Python script, the ‘cross-correlate’ function computes a normalized cross-correlation between the two sets of wavelet coefficients obtained from the CWT. For each scale, it computes the maximum correlation value normalized by signal energy and averages the results across all scales and segments. This approach helps identify high similarities even for two songs with different time shifts.

```python
# Perform Autocorrelation
def cross_correlate(coeffs1, coeffs2):
    try:
        correlations = []

        for seg1, seg2 in zip(coeffs1, coeffs2):
            min_len = min(seg1.shape[1], seg2.shape[1])
            seg1 = seg1[:, :min_len]
            seg2 = seg2[:, :min_len]

            scale_corrs = []

            for scale in range(seg1.shape[0]):
                corr = correlate(seg1[scale], seg2[scale], mode='same', method='fft')

                norm = np.sqrt(np.sum(seg1[scale]**2) * np.sum(seg2[scale]**2))
                scale_corrs.append(np.max(corr)/norm if norm > 0 else 0)

            correlations.append(np.mean(scale_corrs))

        print(f"Optimized Autocorrelation: {np.mean(scale_corrs):.4f}")
        return np.mean(correlations) * 100

    except Exception as e:
        raise Exception(f"Error computing cross-correlation: {str(e)}")
```

Figure 5.3 Code Snippet for Cross-Correlation

5.3.3 MFCC Implementation

In the context of Audio Files Comparator, MFCCs are used to compare the timbre of the two different audio signals. MFCCs can capture perceptual features by approximating the frequency heard by the human ear, which makes them effective to detect different variations of songs like different artists and languages.

Unlike the wavelet preprocessing, the loaded audio files are extracted using 13 MFCCs. The means of each MFCC vector are taken and the Euclidean distance between the means are measured. Larger Euclidean distance results in a higher difference. The distance is normalized to a 0-100% similarity score to avoid negative values when the distance exceeds the maximum value.

```python
# Perform MFCC Comparison
def mfcc_compare(file1, file2):
    try:
        y1, sr1 = librosa.load(file1, sr=44100)
        y2, sr2 = librosa.load(file2, sr=44100)

        mfcc1 = librosa.feature.mfcc(y=y1, sr=sr1, n_mfcc=13)
        mfcc2 = librosa.feature.mfcc(y=y2, sr=sr2, n_mfcc=13)

        dist = np.linalg.norm(mfcc1.mean(axis=1) - mfcc2.mean(axis=1))

        max_dist = 100
        similarity = max(0, 100 - (dist/max_dist)*100)

        return similarity

    except Exception as e:
        raise Exception(f"Error computing MFCC comparison: {str(e)}")
```

Figure 5.4 Code Snippet for MFCC Implementation

5.3.4 Weighted Scoring System

The two distinct comparison metrics are called to evaluate the similarity between audio files which are Cross-Correlation and MFCC. Each of these metrics offers a different perspective on the audio's similarity, and their combined analysis offers a comprehensive assessment.

From the code snippet in Figure 5.5, the weightages of both comparison metrics are equal to 50% each. The balanced weights reflect the equal strengths of each method, where cross-correlation can detect time shifts and MFCCs can detect timbral differences between two songs, suitable for cover detection.

```
# Function to compare coefficients
def compare_coefficients(self, coeffs1, coeffs2):
    try:

        print("\nComputing Autocorrelation...")
        cor_similarity = cross_correlate(coeffs1, coeffs2)
        print("\nComputing MFCC Similarity...")
        mfcc_similarity = mfcc_compare(self.file1_path, self.file2_path)

        self.cor_similarity = cor_similarity
        self.mfcc_similarity = mfcc_similarity

        weights = {'cross_corr': 0.5, 'mfcc': 0.5}
        total_similarity = (weights['cross_corr'] * cor_similarity + weights['mfcc'] * mfcc_similarity)

        self.total_similarity = total_similarity
        similarity_text = f"{total_similarity:.2f}%"

        print(f"Cross Correlation Similarity: {cor_similarity:.2f}%")
        print(f"MFCC Similarity: {mfcc_similarity:.2f}%")
        print(f"Total Similarity: {total_similarity:.2f}%")
        return similarity_text

    except Exception as e:
        raise Exception(f"Error in comparison: {str(e)}")
```

Figure 5.5 Code Snippet of Combining Comparison Metrics

5.4 Implementation Issues and Challenges

There is only one major issue encountered in the implementation of the project, which is the computational demand for comparisons of longer songs. While comparison of longer songs can be successfully computed, the program or hardware may crash, especially if the charger is not connected. The preprocessing process takes approximately 3 minutes while the comparison metrics take about 2 minutes. Although the computation time has improved a lot compared to the preliminary version of the program which computes 20 minutes for a longer audio file, the program cannot be guaranteed to work efficiently without any overheating concerns.

Chapter 6

System Evaluation and Discussion

6.1 Test Case Setup

Several test cases were conducted to assess the performance of the developed techniques in real-world circumstances to validate the feasibility of the Audio Files Comparator program. The test cases were selected to cover a range of audio comparison challenges as listed below:

1. Both Same Songs
2. Same Song, Different Formats
3. Same Song, Different Duration
4. Same Song, Different Artists
5. Same Song, Different Languages
6. Both Different Songs

6.2 Results of Test Cases

6.2.1 Test Case 1: Both Same Songs

For this test case, only one audio file will be loaded twice, which is a chorus of a song titled "Blue" by Yung Kai about 18 seconds long. Based on the results in Figure 6.1, it is proven that both pattern consistency and timbre similarity result in 100% which also results in 100% overall similarity for completely identical songs.

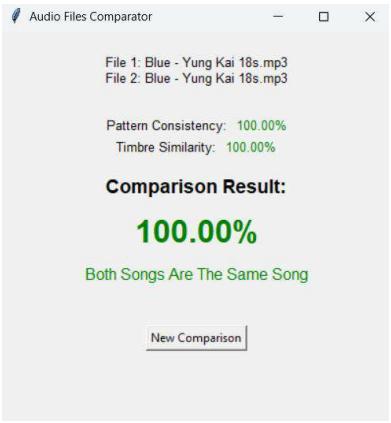


Figure 6.1 Comparison between Same Songs

6.2.2 Test Case 2: Same Song, Different Formats

For this test case, an audio file “primepodcast” of 49 seconds is tested with different formats: MP3 and OGG. Based on the results in Figure 6.2, the high similarity score indicates that the system effectively recognized the same song despite differences in formats. Most of the similarity scores resulted 100% except OGG files which are considered lossless formats.

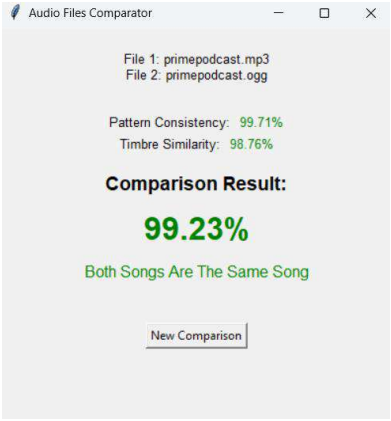


Figure 6.2 Comparison between MP3 and OGG Files

6.2.3 Test Case 3: Same Song, Different Duration

For this test case, two versions of a song titled “Mang Zhong” by Zhao Fangjing. The original song is 3 minutes 42 seconds while the second version is trimmed 1 second at the intro. Based on the results in Figure 6.3, the system effectively identifies both songs have a higher similarity in timbre similarity. However, the pattern consistency of both songs has a moderate score due to a smaller segment duration for CWT preprocessing. When the segment duration increases to 2 seconds, the pattern consistency increases from 40.71% to 55.07% as shown in Figure 6.4.

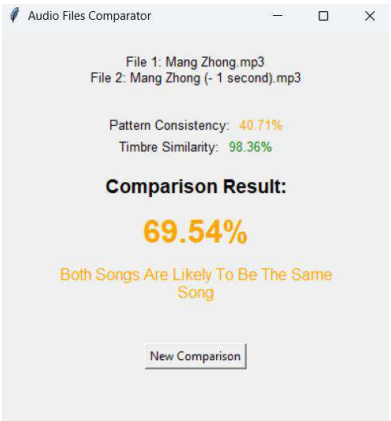


Figure 6.3 Comparison between Songs of Different Duration (Using 1-second segment)

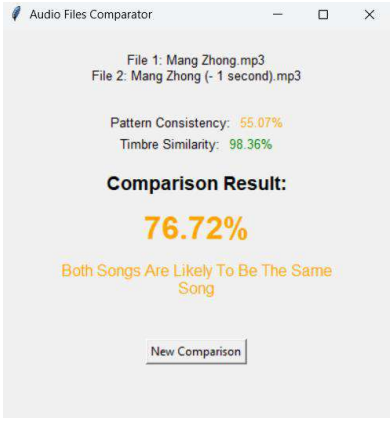


Figure 6.4 Comparison between Songs of Different Duration (Using 2-second segment)

6.2.4 Test Case 4: Same Song, Different Artists

For this test case, two songs "Golden Hour" originally composed by JVKE and covered by Henry Lau respectively are tested. Based on the results in Figure 6.5, the similarity score is lower compared to the previous test cases, reflecting the differences in vocal style and instrumentation between two artists. The timbre similarity is usually higher than the pattern consistency since different covers tend to have similar melodies. This test case demonstrates the system's ability to recognize the same type of song even with variations in performance.

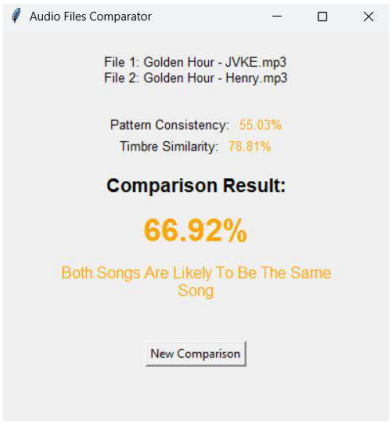


Figure 6.5 Comparison between Different Covers

6.2.5 Test Case 5: Same Song, Different Languages

For this test case, a song titled "Glow" composed by CORSAK is loaded using two different versions: English and Mandarin. Based on the results in Figure 6.6, similar to Test Case 4, the similarity score is relatively moderate, reflecting the differences in not only vocal

instrumentation but language as well. Both pattern consistency and timbre similarity are balanced to maintain the moderate similarity score.

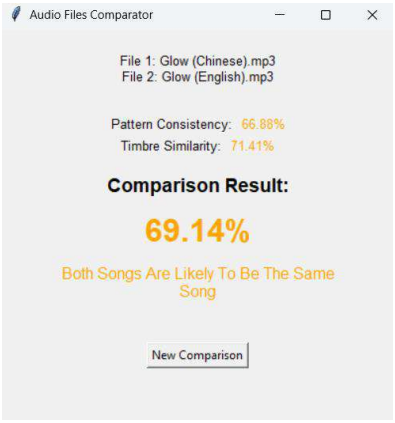


Figure 6.6 Comparison between Different Languages

6.2.6 Test Case 6: Both Different Songs

For this test case, two completely different songs are tested. Based on the results in Figure 6.7, the pattern consistency has a very low score. However, the timbre similarity still results in a moderate score due to MFCCs mostly capture the texture of the songs instead of their whole melody. Eventually the averaged low similarity score confirms that the system has successfully differentiates between entirely different songs, validating its effectiveness in distinguishing non-matching audio files.

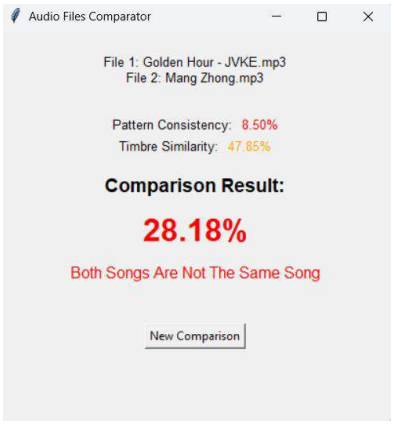


Figure 6.7 Comparison between Different Songs

6.3 Project Challenges

Audio File 1	Audio File 2	Cross-Correlation (%)	MFCC (%)	Total Similarity (%)
Blue – Yung Kai 18s.mp3	Blue – Yung Kai 18s.mp3	100.00	100.00	100.00
Golden Hour - JVKE.mp3	Golden Hour - JVKE.mp3	97.22	100.00	98.61
prime podcast.mp3	prime podcast.wav	100.00	100.00	100.00
prime podcast.mp3	prime podcast.ogg	99.71	98.76	99.23
Mang Zhong.mp3	Mang Zhong (-1 second).mp3	40.71	98.36	69.54
Golden Hour - JVKE.mp3	Golden Hour - Henry.mp3	55.03	78.81	66.92
Glow (Chinese).mp3	Glow (English).mp3	66.88	71.41	69.14
Mang Zhong.mp3	Mang Zhong (Japanese).mp3	32.76	70.03	51.39
Blue – Yung Kai 18s.mp3	Blue (with MINNIE).mp3	91.39	88.59	89.99
Blue – Yung Kai 18s.mp3	Blue – Yung Kai (UIIA version).mp3	68.79	56.18	62.48
Golden Hour - JVKE.mp3	Mang Zhong.mp3	8.50	47.85	28.18

Table 6.1 Test Cases & Results

Differences	Cross-Correlation (%)	MFCC (%)	Total Similarity (%)
Same Songs	High	High	**90-100**
Different Formats (e.g. mp3, wav)	High	High	**90-100**
Different Duration	Medium	High	**40-89**
Different Artists	Medium	Medium	**40-89**
Different Languages	Medium	Medium	**40-89**
Different Songs	Low	Medium	**0-39**

Table 6.2 Feasibility of Methods

Based on Table 6.1, both cross-correlation and MFCCs show varying levels of feasibility depending on the type of audio comparison being performed. Although the program seemed to work well for some test cases, the program still faced some challenges during the comparison process.

Cross-correlation is supposed to obtain a higher similarity result since it can potentially detect time shifts. However, its capabilities are limited due to the adjustments of the segment size for CWT preprocessing. If the segment size is smaller than the duration offset, the similarity score will be lower, and some information may be missing when comparing the CWT coefficients.

MFCCs have proven to capture high timbral similarities for similar songs with minor differences. However, songs that are completely different will still result in a moderate similarity score since they are designed to capture the general texture of the songs, which might include same instruments used in the songs. This implies that MFCCs do not specifically compare the melody of the songs.

Chapter 7
Conclusion and Recommendations

7.1 Conclusion

This project aims to effectively detect durations or time shifts and identify different covers of songs made by different artists or composed with different languages. This project also leverages CWT in audio preprocessing to enhance comparison accuracy.

The program demonstrates that traditional signal processing techniques can effectively compare audio files without relying on machine learning. While this approach works reasonably well for checking alignments and detecting covers, this is still a mechanical and rule-based method, which lacks the adaptability of modern machine learning solutions. This project is unable to fulfil that approach due to the limitations of hardware used.

Despite the computational constraints for longer files, the project successfully provides a functional alternative to compare audio files systematically. Unlike commercial software that hides behind paywalls, this tool offers a clear and user-friendly comparison method without complicated algorithms.

7.2 Recommendation

To further improve this program, future work could implement machine learning models to enhance audio comparisons with better hardware that can handle high computational demands. Additional features such as pitch and tempo variation analysis can also be integrated for more comprehensive comparison. Ultimately, this project could evolve into a web-based application, making it more accessible to users without the need to be installed into their hardware.

REFERENCES

[1] A. Dutt (2021) "Audio Classification using Wavelet Transform and Deep Learning" Available at: https://adityadutt.medium.com/audio-classification-using-wavelet-transform-and-deep-learning-f9f0978fa246 (Accessed: Mar. 19, 2024)

[2] Adrián-Martínez *et al.*, "Acoustic signal detection through the cross-correlation method in experiments with different signal to noise ratio and reverberation conditions," *arXiv.org*, 2015. https://arxiv.org/abs/1502.05038 (Accessed: Apr. 10, 2024)

[3] A. P. D. Huang & M. M. S. Chen & B. Y. Wang, "A review of mel-frequency cepstral coefficients (MFCCs): Features, algorithms and applications" *Signal Processing Magazine, IEEE*, vol. 30, no. 3, pp. 359-389, May 2013.

[4] D. W. Boyd, "Stochastic Analysis," *Systems Analysis and Modeling*, pp. 211–227, 2001, doi: https://doi.org/10.1016/b978-012121851-5/50008-3.

[5] J. D. Guzman (2018). "Fast and Efficient Pitch Detection: Bitstream Autocorrelation". Available at: https://www.cycfi.com/2018/03/fast-and-efficient-pitch-detection-bitstream-autocorrelation/ (Accessed: Aug. 19, 2024)

[6] J. Zhu, C. Gao, and H. Liu, "A wavelet transform: Research on noise reduction of music signals," *Noise & Vibration Worldwide*, vol. 56, no. 1–2, pp. 120–125, Dec. 2024, doi: https://doi.org/10.1177/09574565241306334

[7] Kaustubh. (2024) "A Complete Guide on Audio Bitrate". Available at: https://www.gumlet.com/learn/audio-bitrate/ (Accessed: Mar. 16, 2025)

[8] Lilia, K. Lajmi, B. Blum, Eng, Marc-André, and Tucholke, "Introduction and evaluation of a new time-frequency transformation based on the CQT," doi: https://doi.org/10.26271/opus-1254

[9] M. A. Nguyen (2016). “The Method of Comparing Tow Audio Files”. Available at: https://biomedicalsignalandimage.blogspot.com/2016/03/the-method-of-comparing-two-audio-files.html?m=1 (Accessed: Mar. 19, 2024)

[10] M. Greentree (2023) “Why You Should Download Music from your Preferred Streaming Service”. Available at: https://www.subjectivesounds.com/tips/why-you-should-download-music-from-your-preferred-streaming-service (Accessed: Mar. 16, 2025)

[11] MathWorks (2016) “Wavelet Transforms in MATLAB”. Available at: https://www.mathworks.com//discovery/wavelet-transforms.html (Accessed: Mar. 19, 2024)

[12] N. S. Chauhan (2022). “Dynamic Time Warping(DTW) Algorithm in Time Series”. Available at: https://www.theaidream.com/post/dynamic-time-warping-dtw-algorithm-in-time-series (Accessed: Aug. 20, 2024)

[13] P. Mora (2021). “Dynamic Time Warping: Explanation and extensive testing on audio and tabular data”. Available at: https://paul-mora.com/classification/time-series/clustering/python/Dynamic-Time-Warping-Explanation-and-Testing-on-Audio-and-Tabular-Data/ (Accessed: Aug. 20, 2024)

[14] Sakoe, H. & Chiba, S. (1978). Dynamic programming algorithm optimization fro spoken word recognition. IEEE Trans. Acoustics, Speech, and Signal Proc., Vol. ASSP-26. pp. 43-49

[15] S. Davis and P. Mermelstein (1980). “Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 28, no. 4, pp. 357–366, Aug. 1980, doi: https://doi.org/10.1109/tassp.1980.1163420.

APPENDIX

Python Script

```
import tkinter as tk
from tkinter import filedialog, messagebox
from tkinter import ttk
import threading
import time
import librosa
import librosa.display
import numpy as np
import pywt
import os
from scipy.signal import correlate
import matplotlib.pyplot as plt
from functools import partial
import pygame
from pygame import mixer


# Function to process audio and compute wavelet coefficients
def process_audio(file_path, update_progress, segment_duration=1):
    # Load the audio file
    audio_data, sample_rate = librosa.load(file_path, sr=None)
    segment_samples = int(segment_duration * sample_rate)
    num_segments = len(audio_data) // segment_samples

    coefficients = []
    scales = np.arange(1, 128)

    # Process each segment
    for seg in range(num_segments):
        start_idx = seg * segment_samples
        end_idx = start_idx + segment_samples
        segment = audio_data[start_idx:end_idx]

        # Perform CWT
        cwt_segment, _ = pywt.cwt(segment, scales, 'mexh') # Mexican Hat wavelet used
        coefficients.append(cwt_segment)

        # Update the progress bar
        update_progress(1)

    print(f"All segments of '{os.path.basename(file_path)}' processed.")
    return np.array(coefficients)

# Perform Cross-Correlation
def cross_correlate(coeffs1, coeffs2):
    try:
        correlations = []
```

```
        for seg1, seg2 in zip(coeffs1, coeffs2):
            min_len = min(seg1.shape[1], seg2.shape[1])
            seg1 = seg1[:, :min_len]
            seg2 = seg2[:, :min_len]

            scale_corrs = []

            for scale in range(seg1.shape[0]):
                corr = correlate(seg1[scale], seg2[scale], mode='same', method='fft')

                norm = np.sqrt(np.sum(seg1[scale]**2) * np.sum(seg2[scale]**2))
                scale_corrs.append(np.max(corr)/norm if norm > 0 else 0)

            correlations.append(np.mean(scale_corrs))

        print(f"Optimized Cross-correlation: {np.mean(scale_corrs):.4f}")
        return np.mean(correlations) * 100

    except Exception as e:
        raise Exception(f"Error computing Cross-correlation: {str(e)}")

# Perform MFCC Comparison
def mfcc_compare(file1, file2):
    try:
        y1, sr1 = librosa.load(file1, sr=None)
        y2, sr2 = librosa.load(file2, sr=None)

        mfcc1 = librosa.feature.mfcc(y=y1, sr=sr1, n_mfcc=13)
        mfcc2 = librosa.feature.mfcc(y=y2, sr=sr2, n_mfcc=13)

        dist = np.linalg.norm(mfcc1.mean(axis=1) - mfcc2.mean(axis=1))

        max_dist = 100
        similarity = max(0, 100 - (dist/max_dist)*100)

        return similarity

    except Exception as e:
        raise Exception(f"Error computing MFCC comparison: {str(e)}")



# Function to compare coefficients
def compare_coefficients(self, coeffs1, coeffs2):
    try:

        print("\nComputing Cross-Correlation...")
        cor_similarity = cross_correlate(coeffs1, coeffs2)
        print("\nComputing MFCC Similarity...")
        mfcc_similarity = mfcc_compare(self.file1_path, self.file2_path)
```

```
        self.cor_similarity = cor_similarity
        self.mfcc_similarity = mfcc_similarity

        weights = {'cross_corr': 0.5, 'mfcc': 0.5}
        total_similarity = (weights['cross_corr'] * cor_similarity + weights['mfcc'] *
mfcc_similarity)

        self.total_similarity = total_similarity
        similarity_text = f"{total_similarity:.2f}%"

        print(f"Cross-Correlation Similarity: {cor_similarity:.2f}%")
        print(f"MFCC Similarity: {mfcc_similarity:.2f}%")
        print(f"Total Similarity: {total_similarity:.2f}%")
        return similarity_text

    except Exception as e:
        raise Exception(f"Error in comparison: {str(e)}")

# Function to preprocess both audio files concurrently
def preprocess_both_files(file1_path, file2_path, update_progress):
    coeffs1 = process_audio(file1_path, update_progress)
    coeffs2 = process_audio(file2_path, update_progress)
    return coeffs1, coeffs2

class GUI:
    def __init__(self, root):
        self.root = root
        self.root.title("Audio Files Comparator")
        self.root.geometry("400x400")
        self.root.resizable(True, True)

        self.file1_path = None
        self.file2_path = None

        self.wavelet1 = None
        self.wavelet2 = None

        self.init_ui()

        # Initialize pygame mixer for playback
        pygame.init()
        mixer.init()

        self.playback_position = 0
        self.song_length = 0
        self.is_playing = False
        self.update_interval = 100
        self.currently_playing = None

        self.root.protocol("WM_DELETE_WINDOW", self.on_close)

    def on_close(self):
```

```
        try:
            self.stop_audio()
        except:
            pass  # Ignore errors during cleanup
        try:
            mixer.quit()
            pygame.quit()
        except:
            pass  # Ignore errors during cleanup
        self.root.destroy()

    def init_ui(self):
        # Clear any existing widgets
        for widget in self.root.winfo_children():
            widget.destroy()

        # Frame for file selection
        self.file_frame = tk.Frame(root)
        self.file_frame.pack(pady=10)

        self.file_frame.grid_columnconfigure(0, minsize=80)
        self.file_frame.grid_columnconfigure(1, minsize=200)
        self.file_frame.grid_columnconfigure(2, minsize=40)
        self.file_frame.grid_columnconfigure(3, minsize=40)

        self.file1_label = tk.Label(self.file_frame, text="File 1:", anchor="w")
        self.file1_label.grid(row=0, column=0, sticky='w', padx=5, pady=5)
        self.file1_name_label = tk.Label(self.file_frame, text="Not selected", anchor="w",
width=25)
        self.file1_name_label.grid(row=0, column=1, sticky="w", padx=5)
        self.file1_button = tk.Button(self.file_frame, text="Select",
command=self.select_file1, width=8)
        self.file1_button.grid(row=0, column=1, sticky="e", padx=5)
        self.play1_button = tk.Button(self.file_frame, text="▶", command=lambda:
self.play_audio(1), state=tk.DISABLED)
        self.play1_button.grid(row=0, column=2, padx=5)
        self.stop1_button = tk.Button(self.file_frame, text=" ■ ", command=lambda:
self.stop_audio(1), state=tk.DISABLED)
        self.stop1_button.grid(row=0, column=3, padx=5)

        self.file2_label = tk.Label(self.file_frame, text="File 2:", anchor="w")
        self.file2_label.grid(row=1, column=0, sticky="w", padx=5, pady=5)
        self.file2_name_label = tk.Label(self.file_frame, text="Not selected", anchor="w",
width=25)
        self.file2_name_label.grid(row=1, column=1, sticky="w", padx=5)
        self.file2_button = tk.Button(self.file_frame, text="Select",
command=self.select_file2, width=8)
        self.file2_button.grid(row=1, column=1, sticky="e", padx=5)
        self.play2_button = tk.Button(self.file_frame, text="▶", command=lambda:
self.play_audio(2), state=tk.DISABLED)
        self.play2_button.grid(row=1, column=2, padx=5)
```

```
        self.stop2_button = tk.Button(self.file_frame, text=" ■ ", command=lambda: self.stop_audio(2), state=tk.DISABLED)
        self.stop2_button.grid(row=1, column=3, padx=5)

        self.timer_frame = tk.Frame(self.root)
        self.timer_frame.pack(pady=5)
        self.timer_label = tk.Label(self.timer_frame, text="0:00", font=("Arial", 12))
        self.timer_label.pack()

        self.analysis_button = tk.Button(self.root, text="Show Audio Analysis", command=self.show_analysis)
        self.analysis_button.pack(pady=10)

        self.display_button = tk.Button(self.root, text="Display Waveforms", command=self.display_waveforms)
        self.display_button.pack(pady=10)

        # Button to start comparison
        self.compare_button = tk.Button(root, text="Start Comparison", command=self.start_comparison, width=20, height=2, font=("", 12))
        self.compare_button.pack(pady=20)
        self.root.bind('<Return>', self.on_enter_pressed)

    def play_audio(self, file_num):
        file_path = self.file1_path if file_num == 1 else self.file2_path
        if file_path:
            try:
                mixer.music.stop()
                mixer.music.load(file_path)
                mixer.music.play()
                self.is_playing = True
                self.currently_playing = file_num

                # Get song length
                audio_data, sr = librosa.load(file_path, sr=None)
                self.song_length = librosa.get_duration(y=audio_data, sr=sr)
                self.playback_position = 0

                # Update timer
                self.update_timer()

            except Exception as e:
                messagebox.showerror("Playback Error", f"Could not play audio: {e}")

    def stop_audio(self, file_num=None):
        try:
            mixer.music.stop()
            self.is_playing = False
            self.currently_playing = None
            if hasattr(self, 'timer_label') and self.timer_label.winfo_exists():
                self.timer_label.config(text="0:00")
```

```
        except:
            pass

    def update_timer(self):
        if not self.is_playing:
            return

        # Get current playback position
        self.playback_position = mixer.music.get_pos() / 1000  # convert to seconds

        # Update timer label
        current_time = self.format_time(self.playback_position)
        self.timer_label.config(text=current_time)

        # Schedule next update
        self.root.after(self.update_interval, self.update_timer)

    def format_time(self, seconds):
        minutes = int(seconds // 60)
        seconds = int(seconds % 60)
        return f"{minutes}:{seconds:02d}"

    def on_enter_pressed(self, event):
        # Trigger the button's command when Enter is pressed
        self.start_comparison()

    def show_progress(self):
        self.stop_audio()

        # Clear any existing widgets
        for widget in self.root.winfo_children():
            widget.destroy()

        self.progress = ttk.Progressbar(root, orient="horizontal", length=300,
mode="determinate")
        self.progress.pack(pady=20)

        self.progress_label = tk.Label(root, text="Loading... 0%")
        self.progress_label.pack(pady=10)

        self.root.protocol("WM_DELETE_WINDOW", self.on_close)

    def update_progress(self, step):
        self.progress['value'] += step
        percentage = min(99, int((self.progress['value'] / self.progress['maximum']) * 100))
        self.progress_label.config(text=f"Loading... {percentage}%")

    def select_file1(self):
        self.file1_path = filedialog.askopenfilename(filetypes=[("Audio Files", "*.wav
*.mp3 *.ogg *.flac")])
        if self.file1_path:
            short_name = os.path.basename(self.file1_path)
```

```
        if len(short_name) > 25:
            short_name = short_name[:22] + "..."
        self.file1_name_label.config(text=short_name)
        self.play1_button.config(state=tk.NORMAL)
        self.stop1_button.config(state=tk.NORMAL)

    def select_file2(self):
        self.file2_path = filedialog.askopenfilename(filetypes=[("Audio Files", "*.wav
*.mp3 *.ogg *.flac")])
        if self.file2_path:
            short_name = os.path.basename(self.file2_path)
            if len(short_name) > 25:
                short_name = short_name[:22] + "..."
            self.file2_name_label.config(text=short_name)
            self.play2_button.config(state=tk.NORMAL)
            self.stop2_button.config(state=tk.NORMAL)

    def show_analysis(self):
        if not self.file1_path or not self.file2_path:
            tk.messagebox.showerror("Error", "Please select both audio files.")
            return

        analysis_window = tk.Toplevel(self.root)
        analysis_window.title("Audio Analysis")

        audio_data1, sr1 = librosa.load(self.file1_path, sr=None)
        audio_data2, sr2 = librosa.load(self.file2_path, sr=None)

        analysis_text = (f"File 1: {os.path.basename(self.file1_path)}\n"
                         f"  - Duration: {librosa.get_duration(y=audio_data1, sr=sr1):.2f}
seconds\n"
                         f"  - Sample Rate: {sr1} Hz\n\n"
                         f"File 2: {os.path.basename(self.file2_path)}\n"
                         f"  - Duration: {librosa.get_duration(y=audio_data2, sr=sr2):.2f}
seconds\n"
                         f"  - Sample Rate: {sr2} Hz\n")

        analysis_label = tk.Label(analysis_window, text=analysis_text, font=("Arial", 12),
anchor="w", justify="left")
        analysis_label.pack(pady=20, padx=20)

    def display_waveforms(self):
        if not self.file1_path or not self.file2_path:
            tk.messagebox.showerror("Error", "Please select both audio files.")
            return

        audio_data1, sr1 = librosa.load(self.file1_path, sr=None)
        audio_data2, sr2 = librosa.load(self.file2_path, sr=None)

        duration1 = len(audio_data1) / sr1
        duration2 = len(audio_data2) / sr2
```

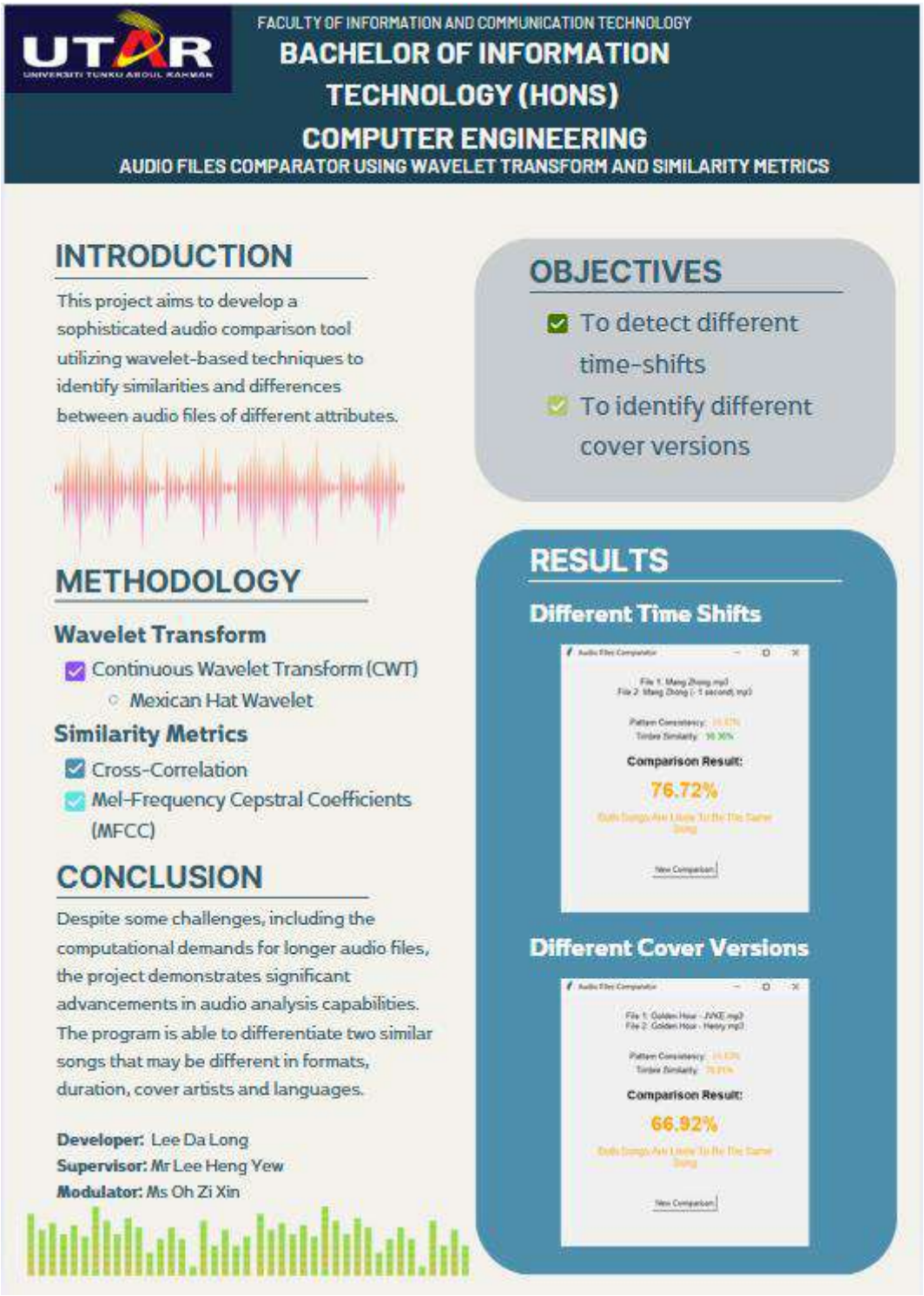
```
        # Downsampling
        max_samples = 10000 # per seconds
        max_points = min(int(max_samples * max(duration1, duration2)), 500000)

        # Create time vector for x-axis
        time1 = np.linspace(0, duration1, len(audio_data1))
        time2 = np.linspace(0, duration2, len(audio_data2))

        def decimate_waveform(time, data, max_points):
            if len(data) <= max_points:
                return time, data
            step = int(len(data) / max_points)
            return time[::step], data[::step]

        time1, audio_data1 = decimate_waveform(time1, audio_data1, max_points)
        time2, audio_data2 = decimate_waveform(time2, audio_data2, max_points)

        fig, axs = plt.subplots(2, 1, figsize=(10, 6), sharex=True)
        fig.canvas.manager.set_window_title("Audio Waveforms")

        axs[0].plot(time1, audio_data1, color='b', alpha=0.7, linewidth=0.5)
        axs[0].set_title(f"{os.path.basename(self.file1_path)}")
        axs[0].legend([f"{os.path.basename(self.file1_path)}"])
        axs[0].set_ylabel("Amplitude")
        axs[0].grid(True, alpha=0.3)

        axs[1].plot(time2, audio_data2, color='r', alpha=0.7, linewidth=0.5)
        axs[1].set_title(f"{os.path.basename(self.file2_path)}")
        axs[1].legend([f"{os.path.basename(self.file2_path)}"])
        axs[1].set_ylabel("Amplitude")
        axs[1].grid(True, alpha=0.3)

        plt.xlabel("Time (seconds)")
        plt.tight_layout()
        plt.show()

    def start_comparison(self):
        if not self.file1_path or not self.file2_path:
            messagebox.showerror("Error", "Please select both audio files.")
            return

        self.stop_audio()
        self.show_progress()

        # User threading to prevent GUI from freezing
        threading.Thread(target=self.preprocess_compare,                args=(self.file1_path,
self.file2_path)).start()

    def show_result(self, similarity_text):
        # Clear any existing widgets
        for widget in self.root.winfo_children():
            widget.destroy()
```

```
        # Create a frame for the result display
        result_frame = tk.Frame(self.root)
        result_frame.pack(pady=20)

        # Display the result with formatted text
        total_similarity = float(similarity_text.strip('%'))

        def get_metric_color(value):
            if value >= 90:
                return "green"
            elif value < 40:
                return "red"
            else:
                return "orange"

        if total_similarity >= 90:
            similarity_color = "green"
            interpretation = "Both Songs Are The Same Song"
        elif total_similarity < 90:
            if total_similarity < 40:
                similarity_color = "red"
                interpretation = "Both Songs Are Not The Same Song"
            else:
                similarity_color = "orange"
                interpretation = "Both Songs Are Likely To Be The Same Song"

        info_frame = tk.Frame(result_frame)
        info_frame.pack()

        tk.Label(info_frame,
                 text=f"File 1: {os.path.basename(self.file1_path)}\nFile 2:
{os.path.basename(self.file2_path)}",
                 font=("Arial", 10)).pack(pady=(0, 5))

        metrics_frame = tk.Frame(result_frame)
        metrics_frame.pack(pady=(0, 20))

        # Pattern Consistency
        pattern_frame = tk.Frame(result_frame)
        pattern_frame.pack()
        tk.Label(pattern_frame,
                 text="Pattern Consistency: ",
                 font=("Arial", 10)).pack(side=tk.LEFT)
        tk.Label(pattern_frame,
                 text=f"{self.cor_similarity:.2f}%",
                 font=("Arial", 10),
                 fg=get_metric_color(self.cor_similarity)).pack(side=tk.LEFT)

        # Timbre Similarity
        timbre_frame = tk.Frame(result_frame)
        timbre_frame.pack()
```

```
        tk.Label(timbre_frame,
                 text="Timbre Similarity: ",
                 font=("Arial", 10)).pack(side=tk.LEFT)
        tk.Label(timbre_frame,
                 text=f"{self.mfcc_similarity:.2f}%",
                 font=("Arial", 10),
                 fg=get_metric_color(self.mfcc_similarity)).pack(side=tk.LEFT)

        tk.Frame(result_frame, height=15).pack()

        # Overall similarity
        result_section = tk.Frame(result_frame)
        result_section.pack()

        tk.Label(result_section,
                 text="Comparison Result:",
                 font=("Arial", 14, "bold")).pack()

        tk.Label(result_frame, text=similarity_text,
                 font=("Arial", 24, "bold"),
                 fg=similarity_color).pack(pady=(10, 0))

        tk.Label(result_frame, text=interpretation,
                 font=("Arial", 12),
                 fg=similarity_color,
                 wraplength=300).pack(pady=10)

        new_compare_button = tk.Button(self.root,
                                       text="New Comparison",
                                       command=self.init_ui)
        new_compare_button.pack(pady=10)



    def preprocess_compare(self, file1_path, file2_path):
        try:
            # Set the maximum value for the progress bar
            self.progress['maximum'] = 2 * (len(librosa.load(file1_path, sr=None)[0]) //
(44100))

            print("Starting concurrent audio processing for both audio files...")
            start_preprocessing = time.time()
            coeffs1, coeffs2 = preprocess_both_files(file1_path, file2_path, lambda step:
self.update_progress(step))
            end_preprocessing = time.time()
            print("\nProcessing complete!")
            print(f"Processing time: {end_preprocessing - start_preprocessing:.2f} seconds")

            # Manually set progress to 99% before comparison starts
            self.progress['value'] = self.progress['maximum'] - 1
            self.update_progress(0)
```

```
            self.wavelet1 = coeffs1
            self.wavelet2 = coeffs2

            print("\nStarting comparison...")
            start_comparison = time.time()
            result_text = compare_coefficients(self, coeffs1, coeffs2)
            end_comparison = time.time()
            print("\nComparison complete!")
            print(f"Comparison time: {end_comparison - start_comparison:.2f} seconds")
            self.show_result(result_text)

        except Exception as e:
            messagebox.showerror("Error", f"Error during preprocessing: {e}")

if __name__ == '__main__':

    root = tk.Tk()
    app = GUI(root)
    root.mainloop()
```

POSTER


FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY
BACHELOR OF INFORMATION TECHNOLOGY (HONS) COMPUTER ENGINEERING
AUDIO FILES COMPARATOR USING WAVELET TRANSFORM AND SIMILARITY METRICS
INTRODUCTION
This project aims to develop a sophisticated audio comparison tool utilizing wavelet-based techniques to identify similarities and differences between audio files of different attributes.
OBJECTIVES
To detect different time-shifts
To identify different cover versions
METHODOLOGY
Wavelet Transform
Continuous Wavelet Transform (CWT)
Mexican Hat Wavelet
Similarity Metrics
Cross-Correlation
Mel-Frequency Cepstral Coefficients (MFCC)
RESULTS
Different Time Shifts
Comparison Result:
76.72%
Different Cover Versions
Comparison Result:
66.92%
CONCLUSION
Despite some challenges, including the computational demands for longer audio files, the project demonstrates significant advancements in audio analysis capabilities. The program is able to differentiate two similar songs that may be different in formats, duration, cover artists and languages.
Developer: Lee Da Long
Supervisor: Mr Lee Heng Yew
Modulator: Ms Oh Zi Xin