HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

**SCHOOL OF ELECTRICAL AND ELECTRONIC ENGINEERING**



**ASSIGNMENT REPORT**

**Multimedia Compression and Coding**

Name of students: Class code: 157320 *–* Team 3

Lê Hà Hải Vân – 20233886

Bùi Thanh Thảo – 20233876

Phạm Minh Tuyên – 20233885

Đỗ Đại Doanh – 20233837

Name of instructors: Dr. rer. nat. Pham Van Tien

Hanoi, 4/2025

Table of Contents

[LIST OF TABLES 2](#_Toc1427330370)

[1. Assigment overview 3](#_Toc1021387330)

[2. Tasks implementation 4](#_Toc1428851488)

[2.1. Member responsibility 4](#_Toc860739927)

[2.2. Task planning 5](#_Toc1210407565)

[3. Assigment implementation 5](#_Toc1616539695)

[3.1. Audio Recording 6](#_Toc1646919109)

[3.2. Spectrum Analysis 6](#_Toc1057311471)

[3.3. ADPCM Compression 8](#_Toc2096916366)

[3.4. PSNR Analysis of ADPCM and MP3 Compression 8](#_Toc544808380)

[3.5. MIDI & Jazz mixing 12](#_Toc918106269)

[4. Source Code & Repositories 12](#_Toc1586602862)

[5. Conclusions 12](#_Toc857548016)

[5.1. Summary of Key Findings 12](#_Toc526004707)

[5.2. Evaluation of Compression Methods 12](#_Toc1034001240)

[5.3. Lessons Learned and Knowledge Gained 12](#_Toc554802204)

[5.4. Future Development Directions 12](#_Toc602224345)

[6. Appendix: 12](#_Toc626649942)

[6.1. Bùi Thanh Thảo: 12](#_Toc327014817)

[6.2. Lê Hà Hải Vân: 12](#_Toc928129642)

[6.3. Phạm Minh Tuyên: 12](#_Toc1041043796)

[6.4. Đỗ Đại Doanh: 12](#_Toc784265205)

**LIST OF FIGURES**

[Figure 2‑1. Task planning 8](#_Toc195475701)

[Figure 3‑1. Magnitude Spectrum of Audio Signal 10](#_Toc195475702)

[Figure 3‑2. PSNR Calculation of 2 Compression 26](#_Toc195475703)

[Figure 3‑3. PSNR Comparison between 2 codecs 27](#_Toc195475704)

[Figure 3‑4. PSNR per 1-second Segment 27](#_Toc195475705)

# LIST OF TABLES

[Table 3‑1. Alternatives considered 44](#_Toc195475717)

[Table 3‑2. Summary of Libraries & Tools 44](#_Toc195475718)

# Assigment overview

This assigment aims to implement an audio compression and reconstruction pipeline using the **ADPCM (Adaptive Differential Pulse Code Modulation)** codec.

Our group, **Group 3**, was assigned to apply ADPCM to a self-recorded audio clip, analyze its spectrum, compress and compare it with MP3 using **PSNR**, and finally mix it with a generated **MIDI-based Jazz background**.

ADPCM is a predictive codec that compresses audio by encoding the difference between consecutive samples, making it efficient for low-bitrate applications such as voice transmission.

* The Assigment consists of the following key steps:
* Record a 3–4-minute audio clip introducing all group members.
* Use MATLAB to analyze the signal spectrum.
* Compress the audio using ADPCM and compare it with MP3 based on PSNR.
* Generate a MIDI file of the same duration and mix it with the voice audio to form a Jazz track.

This report documents the planning, implementation, analysis, and results of our ADPCM-based compression system.

# Tasks implementation

## Member responsibility

* **Recording Task (All Members):**

All group members participated in recording the final input audio clip, each stating their name, student ID, and individual contribution.

* **Assignment Implementation:**
* **Bùi Thanh Thảo**:
  + Conducted the audio spectrum analysis using MATLAB and added technical comments.
  + Implemented the ADPCM compression program.
* **Lê Hà Hải Vân**:
  + Developed the MP3 compression reference program.
  + Compared ADPCM with MP3 compression using PSNR statistics.
  + Managed the group’s Git repository.
* **Đỗ Đại Doanh**:
  + Wrote the MIDI generation program to create Jazz-style background music.
  + Will edit the presentation video.
* **Phạm Minh Tuyên**:
  + Mixed the MIDI track with the recorded audio to produce the final Jazz mix.
  + Compiled the final group report.

## Task planning

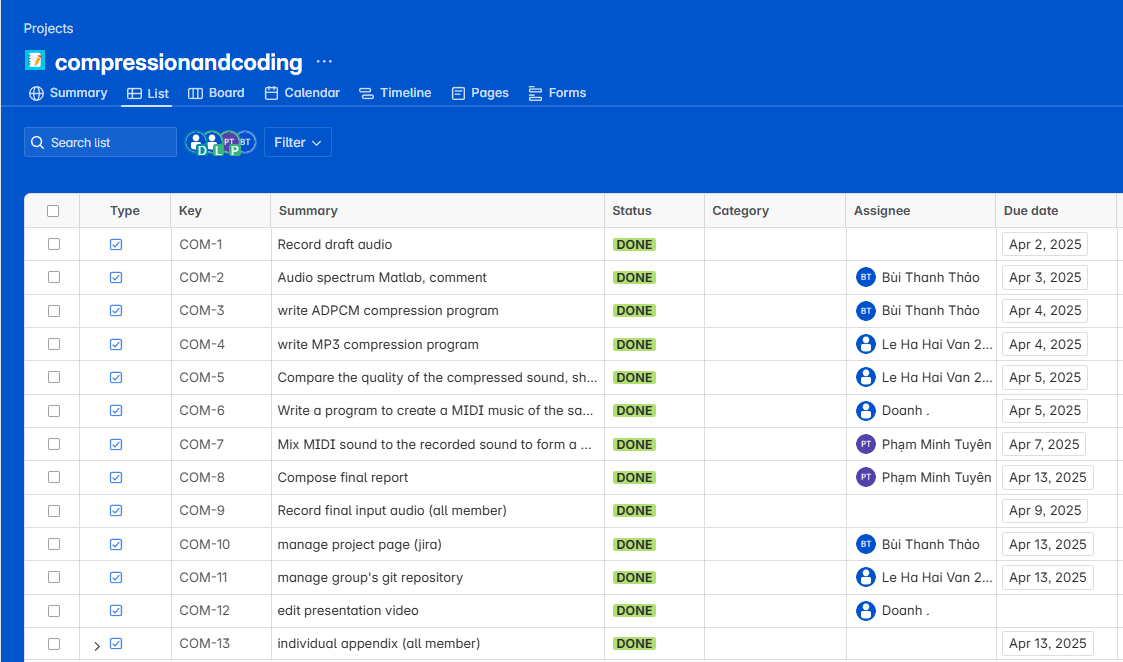


Figure 2‑1. Task planning

# Assigment implementation

## Audio Recording

* The recording is approximately 4 minutes and 03 seconds in length and includes the following:
* Full names of all group members.
* Student IDs.
* Specific tasks assigned to each member.
* Individual technical contributions and achievements.
* Technical Specifications:
* **File Name:** group3.wav
* **Format:** WAV
* **Sample Rate:** 44,100 Hz
* **Bit Depth:** 16-bit
* **Channel Mode:** Mono
* **Duration:** ~4ms
* **Exported:** Full file, no cropping or segment exclusion.

## Spectrum Analysis

* The Fast Fourier Transform (FFT) was applied to convert the time-domain signal into the frequency domain.
* MATLAB program:

% 1. Read the audio file

[audio, fs] = audioread('group3.wav');

% 2. If stereo, convert to mono

if size(audio, 2) == 2

audio = mean(audio, 2); % Average the two channels

end

% 3. Length and FFT

N = length(audio);

Y = fft(audio); % Compute FFT

Y = Y(1:floor(N/2); % Keep only positive frequencies

f = (0:N/2-1) \* fs/N; % Frequency axis

% 4. Compute magnitude spectrum (in dB)

mag = abs(Y);

mag\_db = 20\*log10(mag + eps); % Use eps to avoid log(0)

% 5. Plot spectrum

figure;

plot(f, mag\_db);

xlabel('Frequency (Hz)');

ylabel('Magnitude (dB)');

title('Magnitude Spectrum of Audio Signal');

grid on;

A blue graph with text

AI-generated content may be incorrect.

Figure 3‑1. Magnitude Spectrum of Audio Signal

**General Observations**

* Frequency range: 0 to 22 kHz ⇒ this is likely from an audio file sampled at 44.1 kHz, since the Nyquist frequency is half that.
* Magnitude range: Peaks reach ~90 dB, and the noise floor dips to -70 dB ⇒ this suggests a wide dynamic range in the signal.

**Energy Distribution Over Frequency Axis**

* Dominance in Low Frequencies: The signal exhibits the highest energy concentration below 4 kHz, with the magnitude peaking around 0–1 kHz, which is expected for human voice signals, especially for vowel-rich speech.
* As frequency increases, the spectrum shows a smooth decay in magnitude, which is typical for natural speech signals. This indicates that higher frequencies carry less energy and there is no sign of unwanted high-frequency noise or aliasing.
* Noise Floor and Fluctuation: A fair amount of fine spectral detail/noise can be seen below 0 dB (down to -60 dB). The noise is spread across all frequencies but doesn't overpower the signal, indicating a reasonably clean recording.
* Sharp cutoff at the end of the spectrum ≈ 22 kHz: this is a natural result of digital sampling, where content beyond the Nyquist frequency is not present.

## ADPCM Compression

**3.3.1. Objective:**

The goal of this task is to create a Python program that compresses audio files using ADPCM codec (Adaptive Differential Pulse Code Modulation), which is commonly used for speech compression in low-bitrate scenarios.

**FFmpeg** was initially used to compress audio using the ***adpcm\_ima\_wav*** codec. Later, a custom implementation of the **IMA ADPCM** codec was developed in Python manually, aiming to better understand the internal workings of the compression algorithm.

**3.3.2. ADPCM using FFmpeg**

The implementation leverages the **FFmpeg** multimedia framework through Python to convert uncompressed PCM WAV files into compressed ADPCM-encoded WAV files.

**Environment and Tools Used**

* Language: Python 3
* External Tool: FFmpeg
* Platform: Cross-platform (Windows/Linux/macOS)
* Input Format: Uncompressed WAV (.wav)
* Output Format: Compressed ADPCM WAV (.wav with adpcm\_ima\_wav codec)

**Python program using FFmpeg:**

import subprocess

import os

def compress\_to\_adpcm(*input\_file*, *output\_file*):

    if not os.path.isfile(*input\_file*):

        print("Input file not found")

        return

    try:

        # FFmpeg command to convert WAV to ADPCM

        command = [

            'ffmpeg',

            '-y',  # Overwrite the output file if it exists

            '-i', *input\_file*,

            '-acodec', 'adpcm\_ima\_wav',  # Use ADPCM codec

*output\_file*

        ]

        subprocess.run(command, *check*=True)

        print(f"Successfully compressed: {*output\_file*}")

    except subprocess.CalledProcessError as e:

        print("Error running FFmpeg:", e)

compress\_to\_adpcm("group3.wav", "compressed\_ffmpeg.wav")

**Code Explanation**

* *import subprocess, os*: These modules allow the script to interact with the operating system. subprocess runs external commands (in this case, ffmpeg), and os is used to check file existence.
* *compress\_to\_adpcm(input\_file, output\_file):* This function takes two arguments: the path to the input WAV file and the name for the compressed output file.
* *os.path.isfile(input\_file):* Ensures the input file exists before attempting compression.
* *command = [...]:* Constructs the command to be executed via FFmpeg:
  + -y: Automatically overwrite the output file if it already exists.
  + -i input\_file: Specifies the input WAV file.
  + -acodec adpcm\_ima\_wav: Sets the audio codec to IMA ADPCM.
  + output\_file: Specifies the output filename.
* *subprocess.run(command, check=True):*Executes the command and raises an exception if FFmpeg fails.
* *compress\_to\_adpcm("group3.wav", "compressed\_ffmpeg.wav")*

**Output Specification**

* Output File: compressed\_ffmpeg.wav
* Benefits: Reduced file size, Maintained intelligibility for spoken audio and Broad compatibility with audio players

**3.3.3. Python IMA ADPCM codec implementation**

IMA ADPCM (Interactive Multimedia Association Adaptive Differential Pulse Code Modulation) is a lossy audio compression algorithm that reduces 16-bit PCM audio data to 4 bits per sample.

The core idea is to predict the next sample based on the previous one, then encode only the difference (delta) between them using adaptive quantization.

**Key steps:**

1. **Prediction:** Use the previous sample value as a prediction for the current sample.
2. **Delta calculation:** Compute the difference between the actual sample and the predicted one.
3. **Quantization:** Quantize the delta using a step size selected from a predefined step-size table.
4. **4-bit encoding:** The quantized delta is encoded into a 4-bit "nibble".
5. **Adaptive adjustment:** Update the step size and prediction value based on the result, using index and step tables.

This process is **reversible**: given the initial predicted value and index (stored in the block header), the decoder can reconstruct a close approximation of the original audio.

**Environment and Tools Used**

* Python version: 3.x
* IDE/Editor: Visual Studio Code
* Platform: Windows 10/Linux
* Libraries: wave, struct, os, sys

**Files and Functionality:**

1. ***codec.py*** – Core Codec Logic: This file contains all the core logic for encoding and decoding audio blocks using the IMA ADPCM algorithm.

import struct

# Index change table (for adapting step size)

t\_index = [

    -1, -1, -1, -1, 2, 4, 6, 8,

    -1, -1, -1, -1, 2, 4, 6, 8]

# Step size table (defines quantization levels)

t\_step = [

    7, 8, 9, 10, 11, 12, 13, 14,

    16, 17, 19, 21, 23, 25, 28, 31,

    34, 37, 41, 45, 50, 55, 60, 66,

    73, 80, 88, 97, 107, 118, 130, 143,

    157, 173, 190, 209, 230, 253, 279, 307,

    337, 371, 408, 449, 494, 544, 598, 658,

    724, 796, 876, 963, 1060, 1166, 1282, 1411,

    1552, 1707, 1878, 2066, 2272, 2499, 2749, 3024,

    3327, 3660, 4026, 4428, 4871, 5358, 5894, 6484,

    7132, 7845, 8630, 9493, 10442, 11487, 12635, 13899,

    15289, 16818, 18500, 20350, 22385, 24623, 27086, 29794,

    32767]

\_encoder\_predicted = 0

\_encoder\_index = 0

\_encoder\_step = 7

\_decoder\_predicted = 0

\_decoder\_index = 0

\_decoder\_step = 7

def \_encode\_sample(*sample*):

    # encode one linear pcm sample to ima adpcm neeble

    # using global encoder state

    global \_encoder\_predicted

    global \_encoder\_index

    delta = *sample* - \_encoder\_predicted

    if delta >= 0:

        value = 0

    else:

        value = 8

        delta = -delta

    step = t\_step[\_encoder\_index]

    diff = step >> 3

    if delta > step:

        value |= 4

        delta -= step

        diff += step

    step >>= 1

    if delta > step:

        value |= 2

        delta -= step

        diff += step

    step >>= 1

    if delta > step:

        value |= 1

        diff += step

    if value & 8:

        \_encoder\_predicted -= diff

    else:

        \_encoder\_predicted += diff

    if \_encoder\_predicted < - 0x8000:

        \_encoder\_predicted = -0x8000

    elif \_encoder\_predicted > 0x7fff:

        \_encoder\_predicted = 0x7fff

    \_encoder\_index += t\_index[value & 7]

    if \_encoder\_index < 0:

        \_encoder\_index = 0

    elif \_encoder\_index > 88:

        \_encoder\_index = 88

    return value

def \_decode\_sample(*neeble*):

    # decode one sample from compressed neeble

    # using global decoder state

    global \_decoder\_predicted

    global \_decoder\_index

    global \_decoder\_step

    difference = 0

    if *neeble* & 4:

        difference += \_decoder\_step

    if *neeble* & 2:

        difference += \_decoder\_step >> 1

    if *neeble* & 1:

        difference += \_decoder\_step >> 2

    difference += \_decoder\_step >> 3

    if *neeble* & 8:

        difference = -difference

    \_decoder\_predicted += difference

    if \_decoder\_predicted > 32767:

        \_decoder\_predicted = 32767

    elif \_decoder\_predicted < -32767:

        \_decoder\_predicted = - 32767

    \_decoder\_index += t\_index[*neeble*]

    if \_decoder\_index < 0:

        \_decoder\_index = 0

    elif \_decoder\_index > 88:

        \_decoder\_index = 88

    \_decoder\_step = t\_step[\_decoder\_index]

    return \_decoder\_predicted

def \_calc\_head(*sample*):

    # Calculating ima adpcm block head

    # using global \_encoder\_index

    global \_encoder\_index

    # calculating global \_encoder\_index for first sample

    \_encode\_sample(struct.unpack('h', *sample*)[0])

    # packing header

    head = *sample*                            # Uncompressed sample

    head += struct.pack('B', \_encoder\_index) # Calculated index for sample

    head += struct.pack('B', 0x00)           # Always 0

    return head

def encode\_block(*block*):

    """Encode linear pcm fragment to compressed ima adpcm block.

    Block is a string containing values from wavefile, network, etc.

    Returns a string containing packed compressed ima adpcm block.

    Only 1010 bytes size linear mono 16 bit fragment supported."""

    if len(*block*) != 1010:

        raise ValueError('Unsupported sample quantity in block. Should be 505.')

        return

    result = \_calc\_head(*block*[0:2])

    for i in range(2, len(*block*)):

        if (i + 1) % 4 == 0:

            sample2 = \_encode\_sample(struct.unpack('h', *block*[i - 1:i + 1:])[0])

            sample = \_encode\_sample(struct.unpack('h', *block*[i + 1:i + 3:])[0])

            result += struct.pack('B', (sample << 4) | sample2)

    return result

def decode\_block(*block*):

    if len(*block*) != 256:

        raise ValueError('Unsupported block size. Should be 256.')

    global \_decoder\_predicted

    global \_decoder\_index

    global \_decoder\_step

    result = bytearray()

    \_decoder\_predicted = struct.unpack('h', *block*[0:2])[0]

    \_decoder\_index = struct.unpack('B', *block*[2:3])[0]

    \_decoder\_step = t\_step[\_decoder\_index]

    result += *block*[0:2]

    for i in range(4, len(*block*)):

        original\_sample = struct.unpack('B', *block*[i:i+1])[0]

        second\_sample = original\_sample >> 4

        first\_sample = original\_sample & 0x0F

        result += struct.pack('h', \_decode\_sample(first\_sample))

        result += struct.pack('h', \_decode\_sample(second\_sample))

    return result

\_\_all\_\_ = ["encode\_block", "decode\_block"]

* **Lookup tables:**
  + t\_index: ADPCM index table
  + t\_step: ADPCM step size table
* **Global state variables:** These variables maintain the current predicted value and step size index. They are updated continuously during encoding/decoding to reflect the current state of the algorithm.
  + Encoder state: \_encoder\_predicted, \_encoder\_index
  + Decoder state: \_decoder\_predicted, \_decoder\_index, \_decoder\_step
* **Key Functions:**
  + \_encode\_sample(sample): Encodes a single PCM sample (16-bit) into a 4-bit ADPCM value.
  + \_decode\_sample(neeble): Decodes a 4-bit ADPCM value into a predicted PCM sample.
  + \_calc\_head(sample): Generates a block header containing the initial PCM sample and index.
  + encode\_block(block): Takes a 1010-byte PCM block (505 samples) and compresses it into a 256-byte ADPCM block.
  + decode\_block(block): Takes a 256-byte ADPCM block and reconstructs the original 1010-byte PCM block.

**Input/Output of codec.py functions:**

| **Function** | **Input** | **Output** |
| --- | --- | --- |
| encode\_block() | 1010 bytes PCM (16-bit mono) | 256 bytes ADPCM |
| decode\_block() | 256 bytes ADPCM | * + 1. es PCM (16-bit mono) |

1. ***run.py*** – Driver Script: This file is the executable script used to test the codec.

import wave

from codec import encode\_block, decode\_block

# --- Đọc file WAV gốc ---

with wave.open("group3.wav", "rb") as wav\_in:

    if wav\_in.getsampwidth() != 2 or wav\_in.getnchannels() != 1:

        raise ValueError("File WAV phải là 16-bit PCM mono.")

    framerate = wav\_in.getframerate()

    pcm\_data = wav\_in.readframes(wav\_in.getnframes())

# --- Encode PCM → ADPCM ---

adpcm\_data = bytearray()

for i in range(0, len(pcm\_data), 1010):

    block = pcm\_data[i:i+1010]

    if len(block) < 1010:

        block += b'\x00' \* (1010 - len(block))  # Padding nếu block cuối thiếu

    adpcm\_data += encode\_block(block)

with open("group3.adpcm", "wb") as f:

    f.write(adpcm\_data)

print("Đã nén xong → group3.adpcm")

# --- Decode ADPCM → PCM ---

decoded\_pcm = bytearray()

for i in range(0, len(adpcm\_data), 256):

    block = adpcm\_data[i:i+256]

    decoded\_pcm += decode\_block(block)

with wave.open("group3\_decoded.wav", "wb") as wav\_out:

    wav\_out.setnchannels(1)

    wav\_out.setsampwidth(2)

    wav\_out.setframerate(framerate)

    wav\_out.writeframes(decoded\_pcm)

print(" Đã giải nén lại → group3\_decoded.wav")

**Functionalities:**

* Opens a ***.wav*** file (must be mono, 16-bit PCM)
* Reads audio data and splits it into blocks of 1010 bytes
* Encodes each block using ***encode\_block()*** and writes the result to a ***.adpcm*** file
* Reads back the ***.adpcm*** file, decodes each block using ***decode\_block()***
* Saves the decoded data as a ***.wav*** file to verify the result

## PSNR Analysis of ADPCM and MP3 Compression

* + 1. Objective:

In this section, we quantitatively evaluate the audio quality of the compressed files using the **Peak Signal-to-Noise Ratio (PSNR)** metric. PSNR is commonly used to measure the difference between an original signal and a reconstructed or compressed signal. A higher PSNR value typically indicates better quality and less distortion.

* + 1. MP3 compression:

1. Overview:

**MP3 (MPEG-1 Audio Layer III)** is one of the most widely used audio compression formats today. It uses **lossy compression**, meaning that some of the original audio data is permanently discarded in order to significantly reduce the file size.

The core idea of MP3 compression is based on **psychoacoustic modeling** — exploiting limitations in human hearing. It removes or reduces audio components that are less likely to be perceived, such as:

* Frequencies masked by louder sounds (temporal and frequency masking),
* Sounds outside the human hearing range,
* Redundant or irrelevant information in the signal.

Through this model, MP3 can compress audio to about **1/10th the size** of the original WAV file while still maintaining relatively good sound quality for casual listening.

1. MP3 Encoding Implementation:

In this project, we implemented MP3 compression using the **LAME (LAME Ain't an MP3 Encoder)** library — a high-quality open-source encoder. The original audio file (group3.wav) was encoded into MP3 using a custom C program, mp3\_compression.c.

The key steps in the implementation are:

**1). Encoder Initialization:**

* + The LAME encoder is initialized using lame\_init().

**2). Setting Parameters:**

* + **Sampling Rate:** 44.1 kHz (CD-quality).
  + **Bitrate:** 128 kbps, balancing quality and file size.
  + **Channels:** Mono.
  + **Mode & Quality:** MONO mode and quality setting 2 for high quality.

**2). File Processing:**

* + The WAV file is read in chunks of PCM data, skipping the 44-byte header.
  + Each chunk is passed to lame\_encode\_buffer() to be encoded into MP3.
  + The MP3 data is written to the output file compressed\_mp3.mp3.

**4). Finalization:**

* + After all PCM data is processed, the remaining data in the encoder buffer is flushed using lame\_encode\_flush().
  + All files are closed and the encoder is cleaned up.

1. Code:

#include <stdio.h>

#include <stdlib.h>

#include <lame/lame.h>

int main() {

    lame\_global\_flags \*gfp = lame\_init();

    if (gfp == NULL) {

        fprintf(stderr, "Failed to initialize LAME encoder\n");

        return 1;

    }

    // Set LAME parameters

    lame\_set\_num\_channels(gfp, 1);       // Mono

    lame\_set\_in\_samplerate(gfp, 44100);  // 44.1 kHz

    lame\_set\_brate(gfp, 128);            // 128 kbps

    lame\_set\_mode(gfp, MONO);            // MONO mode

    lame\_set\_quality(gfp, 2);            // Quality setting

    // Initialize the encoder

    if (lame\_init\_params(gfp) < 0) {

        fprintf(stderr, "Failed to initialize LAME parameters\n");

        lame\_close(gfp);

        return 1;

    }

    FILE \*wav\_file = fopen("group3.wav", "rb");

    if (wav\_file == NULL) {

        perror("Failed to open WAV file");

        lame\_close(gfp);

        return 1;

    }

    // Skip WAV header (44 bytes)

    fseek(wav\_file, 44, SEEK\_SET);

    FILE \*mp3\_file = fopen("compressed\_mp3.mp3", "wb");

    if (mp3\_file == NULL) {

        perror("Failed to open MP3 file");

        fclose(wav\_file);

        lame\_close(gfp);

        return 1;

    }

    const int PCM\_SIZE = 8192;

    const int MP3\_SIZE = 8192;

    short pcm\_buffer[PCM\_SIZE];

    unsigned char mp3\_buffer[MP3\_SIZE];

    int read, write;

    do {

        // Read PCM data (16-bit signed mono)

        read = fread(pcm\_buffer, sizeof(short), PCM\_SIZE, wav\_file);

        if (read == 0) {

            // Encode the remaining data

            write = lame\_encode\_flush(gfp, mp3\_buffer, MP3\_SIZE);

        } else {

            // Encode mono PCM to MP3

            write = lame\_encode\_buffer(gfp, pcm\_buffer, NULL, read, mp3\_buffer, MP3\_SIZE);

        }

        // Write MP3 data

        fwrite(mp3\_buffer, write, 1, mp3\_file);

    } while (read != 0);

    fclose(wav\_file);

    fclose(mp3\_file);

    lame\_close(gfp);

    printf("Compression complete. Output: compressed\_mp3.mp3\n");

    return 0;

}

1. Compression result:

The resulting MP3 file is significantly smaller in size than the original WAV file due to the use of perceptual coding and bitrate reduction. This file was later used in the **PSNR analysis** section to evaluate the fidelity of MP3 compression in comparison to ADPCM and the original signal.

This implementation not only demonstrates how MP3 encoding works at a low level but also highlights the trade-offs between compression ratio and audio quality in multimedia applications.

* + 1. Methodology:

PSNR (Peak Signal-to-Noise Ratio) is a commonly used metric to measure the quality of a compressed signal compared to the original signal, especially popular in image and audio processing fields. PSNR is calculated based on the Mean Squared Error (MSE), which is the average squared error between the original signal and the compressed signal.

PSNR is defined by the following formula:

Where:

* MAX is the peak value the signal can have (for normalized audio, MAX = 1),
* MSE is the Mean Squared Error between the original signal and the compressed signal.

A higher PSNR value indicates that the quality of the compressed signal is closer to the original signal, i.e., better quality. In practice:

* PSNR > 30 dB: Good quality.
* PSNR 20–30 dB: Acceptable quality.
* PSNR < 20 dB: Low quality.
  + 1. Code implementation:

To compare the quality of the signals compressed with ADPCM and MP3, we used a MATLAB script to calculate the PSNR between the following files:

* group3.wav: The original audio file,
* decoded\_adpcm.wav: The file compressed and decompressed using ADPCM,
* compressed\_mp3.mp3: The file compressed using MP3.

Matlab code:

function PSNR(original\_file, compressed\_file, mp3\_file)

    % Read audio files

    [x\_org, fs\_orig] = audioread(original\_file)*;*

    [x\_adpcm, fs\_adpcm] = audioread(compressed\_file)*;*

    [x\_mp3, fs\_mp3] = audioread(mp3\_file)*;*

    % Resample if needed

    if fs\_adpcm ~= fs\_orig

        x\_adpcm = resample(x\_adpcm, fs\_orig, fs\_adpcm)*;*

    end

    if fs\_mp3 ~= fs\_orig

        x\_mp3 = resample(x\_mp3, fs\_orig, fs\_mp3)*;*

    end

    % Convert stereo to mono

    if size(x\_org,2) == 2

        x\_org = mean(x\_org, 2)*;*

    end

    if size(x\_adpcm,2) == 2

        x\_adpcm = mean(x\_adpcm, 2)*;*

    end

    if size(x\_mp3,2) == 2

        x\_mp3 = mean(x\_mp3, 2)*;*

    end

    % Normalize amplitudes (optional)

    x\_org = x\_org / max(abs(x\_org))*;*

    x\_adpcm = x\_adpcm / max(abs(x\_adpcm))*;*

    x\_mp3 = x\_mp3 / max(abs(x\_mp3))*;*

    % Match signal lengths

    min\_len = min([length(x\_org), length(x\_adpcm), length(x\_mp3)])*;*

    x\_org = x\_org(1:min\_len)*;*

    x\_adpcm = x\_adpcm(1:min\_len)*;*

    x\_mp3 = x\_mp3(1:min\_len)*;*

    % Compute PSNR values

    psnr\_adpcm = calc\_psnr(x\_org, x\_adpcm)*;*

    psnr\_mp3 = calc\_psnr(x\_org, x\_mp3)*;*

    % Display results

    fprintf('PSNR (ADPCM): %.2f dB\n', psnr\_adpcm)*;*

    fprintf('PSNR (MP3): %.2f dB\n', psnr\_mp3)*;*

    % Plot bar chart with colors

    figure*;*

    b = bar([psnr\_adpcm, psnr\_mp3], 'FaceColor', 'flat')*;*

    b.CData(1,:) = [135 206 235] / 255;  % Skyblue

    b.CData(2,:) = [255 165 0] / 255;    % Orange

    set(gca, 'XTickLabel', {'ADPCM', 'MP3'})*;*

    ylabel('PSNR (dB)')*;*

    title('PSNR Comparison Between Codecs')*;*

    grid on*;*

end

function psnr\_val = calc\_psnr(ref, test)

    mse = mean((ref - test).^2)*;*

    if mse == 0

        psnr\_val = Inf*;*

    else

        psnr\_val = 10 \* log10(1 / mse); % Reference is normalized to [-1,1]

    end

end

Additionally, we also developed a code that can calculate the PSNR for each segment with a duration of 1 second.

function PSNR\_segments(original\_file, adpcm\_file, mp3\_file)

    % Read audio files

    [x\_org, fs\_org] = audioread(original\_file)*;*

    [x\_adpcm, fs\_adpcm] = audioread(adpcm\_file)*;*

    [x\_mp3, fs\_mp3] = audioread(mp3\_file)*;*

    % Resample if needed

    if fs\_adpcm ~= fs\_org

        x\_adpcm = resample(x\_adpcm, fs\_org, fs\_adpcm)*;*

    end

    if fs\_mp3 ~= fs\_org

        x\_mp3 = resample(x\_mp3, fs\_org, fs\_mp3)*;*

    end

    % Convert stereo to mono if needed

    if size(x\_org,2) == 2

        x\_org = mean(x\_org, 2)*;*

    end

    if size(x\_adpcm,2) == 2

        x\_adpcm = mean(x\_adpcm, 2)*;*

    end

    if size(x\_mp3,2) == 2

        x\_mp3 = mean(x\_mp3, 2)*;*

    end

    % Normalize amplitudes

    x\_org = x\_org / max(abs(x\_org))*;*

    x\_adpcm = x\_adpcm / max(abs(x\_adpcm))*;*

    x\_mp3 = x\_mp3 / max(abs(x\_mp3))*;*

    % Match signal lengths

    min\_len = min([length(x\_org), length(x\_adpcm), length(x\_mp3)])*;*

    x\_org = x\_org(1:min\_len)*;*

    x\_adpcm = x\_adpcm(1:min\_len)*;*

    x\_mp3 = x\_mp3(1:min\_len)*;*

    % Compute PSNR for each 1-second segment

    segment\_len = fs\_org;  % 1 second per segment

    num\_segments = floor(min\_len / segment\_len);

    psnr\_adpcm = zeros(1, num\_segments);

    psnr\_mp3 = zeros(1, num\_segments);

    for i = 1:num\_segments

        idx\_start = (i - 1) \* segment\_len + 1;

        idx\_end = i \* segment\_len;

        seg\_org = x\_org(idx\_start:idx\_end);

        seg\_adpcm = x\_adpcm(idx\_start:idx\_end);

        seg\_mp3 = x\_mp3(idx\_start:idx\_end);

        psnr\_adpcm(i) = calc\_psnr(seg\_org, seg\_adpcm);

        psnr\_mp3(i) = calc\_psnr(seg\_org, seg\_mp3);

    end

    % Plot the results

    figure;

    p1 = plot(1:num\_segments, psnr\_adpcm, '-o', 'DisplayName', 'ADPCM');

    hold on;

    p2 = plot(1:num\_segments, psnr\_mp3, '-x', 'DisplayName', 'MP3');

    % Set colors for the plots

    p1.Color = [135 206 235] / 255;  % Skyblue

    p2.Color = [255 165 0] / 255;    % Orange

    title('PSNR per 1-second Segment');

    xlabel('Time (seconds)');

    ylabel('PSNR (dB)');

    legend show;

    grid on;

end

function psnr\_val = calc\_psnr(ref, test)

    mse = mean((ref - test).^2);

    if mse == 0

        psnr\_val = Inf;

    else

        psnr\_val = 10 \* log10(1 / mse);

    end

end

* + 1. Result:

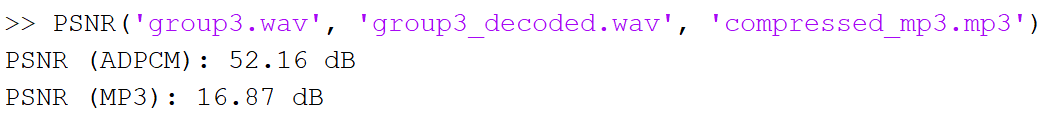


Figure 3‑2. PSNR Calculation of 2 Compression

A screenshot of a graph

AI-generated content may be incorrect.

Figure 3‑3. PSNR Comparison between 2 codecs

The chart clearly illustrates the substantial quality difference between the two compression methods, with ADPCM achieving 52.1598 dB compared to MP3's 16.8663 dB. The visual representation emphasizes the approximately 3x higher PSNR value of ADPCM, highlighting the significant quality advantage of this codec in preserving the original audio signal.

A graph with blue and orange lines

AI-generated content may be incorrect.

Figure 3‑4. PSNR per 1-second Segment

This figure displays a detailed temporal analysis of PSNR values measured at 1-second intervals throughout the audio sample. This time-series plot reveals several critical insights:

* **Range Differentiation**: The ADPCM codec (blue line with circles) operates predominantly in the 45-65 dB range, with occasional peaks reaching approximately 80 dB. In contrast, the MP3 codec (orange line with x-markers) generally operates in the 15-25 dB range, with occasional peaks reaching approximately 57 dB.
* **Performance Stability**: The ADPCM graph shows moderate variability but maintains consistently high values throughout the recording. The MP3 graph exhibits similar patterns of variability but at a significantly lower quality level.
* **Peak Analysis**: Both codecs show synchronized peaks at several time points (particularly notable around the 100-second mark), indicating certain audio passages that are inherently easier to encode regardless of the compression algorithm used.
* **Quality Floor**: ADPCM rarely drops below 40 dB throughout the entire recording, while MP3 frequently approaches 10-15 dB in its lower-quality segments, representing a substantial difference in worst-case performance.

The ADPCM-compressed file achieved an overall PSNR of 52.16 dB, clearly visualized in Figure 1's blue bar. As Figure 2 demonstrates, this high performance is maintained consistently across the entire audio sample. The temporal pattern reveals characteristic oscillations in PSNR values, with remarkably high peaks (approaching 80 dB) occurring at regular intervals throughout the recording, suggesting excellent handling of specific audio features.

The blue line in Figure 2 exhibits notable stability, with the majority of data points concentrated in the 50-60 dB range. Even during performance drops, the PSNR rarely falls below 40 dB, confirming ADPCM's reliability in maintaining high-quality audio reproduction across varying audio content.

The MP3-compressed file exhibited a substantially lower overall PSNR value of 16.87 dB as shown by the orange bar in Figure 1. Figure 2's temporal analysis (orange line) reveals that this lower quality is consistent throughout the recording, with most segments falling between 15-20 dB.

The graph shows that MP3 compression occasionally achieves moderate PSNR peaks exceeding 30 dB, with a maximum of approximately 57 dB. However, these high-quality moments are sparse and brief. The predominant pattern shows significant signal degradation with PSNR values consistently below 25 dB, indicating substantial deviation from the reference signal.

* + 1. Conclusion:

This analysis demonstrates that ADPCM significantly outperforms MP3 in terms of signal preservation as measured by PSNR for the specific audio sample tested. The overall PSNR values (52.16 dB for ADPCM vs. 16.87 dB for MP3) represent a difference of approximately 35.29 dB, while the segment-by-segment analysis confirms this superiority is maintained consistently throughout the recording.

These findings suggest that ADPCM would be the preferred choice for applications requiring high fidelity audio reproduction, such as professional audio recording, archival purposes, or detailed audio analysis. Conversely, MP3 compression at the tested parameters would be more suitable for applications where storage or bandwidth constraints are paramount and some quality degradation is acceptable.

The temporal analysis provides particularly valuable insights for applications where consistent quality throughout the audio stream is essential, highlighting ADPCM's superior stability in addition to its higher overall fidelity.

## MIDI & Jazz mixing

3.5.1. MIDI Jazz File

The MIDI composition was programmatically generated using a Python script, designed to match the length and musical mood of the recorded group3.wav audio file. The process involved extracting audio duration, calculating beat structure, and layering multiple jazz instruments.

**Overview of the process:**

* The group3.wav file was first loaded to determine its duration in seconds.
* Using a fixed tempo of 65 BPM, the program calculated how many musical beats and full 4-beat bars fit within the audio length.
* Five MIDI instrument tracks were then programmatically generated and assigned to different MIDI channels to avoid overlap.

**Instruments:**

* **Piano Melody (Channel 0):**  
   Randomly selected note sequences (phrases) simulate an improvised melody. Timing and dynamics vary slightly to imitate human expression.
* **Piano Chords (Channel 1):**  
   Jazz-style triads (3-note chords) are repeated every 4 beats, forming the harmonic base.
* **Acoustic Bass (Channel 2):**  
  A walking bass line is generated using four notes per bar, derived from the root of each chord. The pattern [root, root + 3, root + 5, root + 2] is played in the lower octave, with each note lasting 1 beat. Velocity is at 60 to give the bass line enough presence in the mix. This deeper tone provides a strong rhythmic and harmonic foundation typical of jazz styles.
* **Drums (Channel 9):**  
   A light swing rhythm is applied by placing a ride cymbal hit on beats 2 and 4 of each 4-beat bar. Percussion instruments use Channel 9 per General MIDI standard.

Cymbal hits are placed on beats b % 2 == 1 (i.e., beats 2 and 4) to create swing feel.

Each hit uses note 51 (ride cymbal) with a soft velocity of 40 and short sustain (~0.6 beats).

* **String Pad (Channel 3):**  
   Sustained background harmonies are created with simple two-note chords (root + fifth), transposed up an octave and played softly (velocity 15) to add texture without distraction.

**Algorithm Details:**

* **Audio Duration Calculation:**  
   The total duration is extracted using librosa.get\_duration() on the input .wav file.  
   → Example: 4 minutes = 240 seconds.
* **Beat & Bar Conversion:**  
  Beats are computed as:

At 65 BPM, ~260 beats are calculated, then rounded to complete bars (4 beats per bar).

* **Melody Generation:**  
   Each phrase (e.g., [72, 74, 76]) is randomly selected. Notes are spaced with rest durations between 1.0 to 4.0 beats to simulate phrasing.
* **Channel Assignment:**  
   Each instrument is written to a separate MIDI channel to avoid overlap.

**Code:**

import random

import librosa

from mido import MidiFile, MidiTrack, Message, MetaMessage, bpm2tempo

# === CONFIG ===

audio\_path = "group3.wav"

output\_file = "jazz.mid"

ticks\_per\_beat = 480

bpm = 65

# === LOAD AUDIO AND MATCH LENGTH ===

y, sr = librosa.load(audio\_path)

duration\_sec = librosa.get\_duration(y=y, sr=sr)

beats\_total = int((bpm / 60) \* duration\_sec)

bars = beats\_total // 4

total\_beats = bars \* 4

# === JAZZ CONTENT ===

melody\_phrases = [[72, 74], [76, 74, 72], [72], [74, 76]]

chords = [[60, 64, 67], [62, 65, 69], [59, 63, 67], [60, 64, 69]] \* (bars // 4)

bass\_roots = [36, 38, 35, 36] \* (bars // 4) # Acoustic bass range

ride\_cymbal = 51 # GM percussion note for ride

pad\_chords = [[60, 67], [62, 69], [59, 67], [60, 67]] \* (bars // 4)

mid = MidiFile(ticks\_per\_beat=ticks\_per\_beat)

# Piano Melody (channel 0)

melody\_track = MidiTrack()

melody\_track.append(MetaMessage('track\_name', name='Acoustic Grand Piano', time=0))

melody\_track.append(MetaMessage('instrument\_name', name='Piano Melody', time=0))

melody\_track.append(MetaMessage('set\_tempo', tempo=bpm2tempo(bpm)))

melody\_track.append(Message('program\_change', program=0, channel=0, time=0))

melody\_time = 0

beat = 0

while beat < total\_beats:

if random.random() < 0.5:

phrase = random.choice(melody\_phrases)

for note in phrase:

vel = random.randint(35, 45)

melody\_track.append(Message('note\_on', note=note, velocity=vel, time=melody\_time, channel=0))

melody\_track.append(Message('note\_off', note=note, velocity=vel, time=int(ticks\_per\_beat \* 1.2), channel=0))

melody\_time = 0 # reset after first note

rest = int(ticks\_per\_beat \* random.uniform(1.0, 3.0))

melody\_time = rest

beat += len(phrase) + rest / ticks\_per\_beat

else:

rest = int(ticks\_per\_beat \* random.uniform(2.0, 4.0))

melody\_time = rest

beat += rest / ticks\_per\_beat

if melody\_time > 0:

melody\_track.append(Message('note\_on', note=0, velocity=0, time=melody\_time, channel=0))

mid.tracks.append(melody\_track)

# Piano Chords (channel 1)

chord\_track = MidiTrack()

chord\_track.append(MetaMessage('track\_name', name='Acoustic Grand Piano', time=0))

chord\_track.append(MetaMessage('instrument\_name', name='Piano Chords', time=0))

chord\_track.append(Message('program\_change', program=0, channel=1, time=0))

for chord in chords:

for i, note in enumerate(chord):

chord\_track.append(Message('note\_on', note=note, velocity=50, time=0 if i > 0 else 0, channel=1))

for i, note in enumerate(chord):

chord\_track.append(Message('note\_off', note=note, velocity=50, time=int(ticks\_per\_beat \* 4) if i == 0 else 0, channel=1))

mid.tracks.append(chord\_track)

# Acoustic Bass (channel 2)

bass\_track = MidiTrack()

bass\_track.append(MetaMessage('track\_name', name='Acoustic Bass', time=0))

bass\_track.append(MetaMessage('instrument\_name', name='Acoustic Bass', time=0))

bass\_track.append(Message('program\_change', program=32, channel=2, time=0))

for root in bass\_roots:

pattern = [root, root + 3, root + 5, root + 2]

for i, note in enumerate(pattern):

bass\_track.append(Message('note\_on', note=note, velocity=60, time=0 if i == 0 else 0, channel=2))

bass\_track.append(Message('note\_off', note=note, velocity=60, time=ticks\_per\_beat, channel=2))

mid.tracks.append(bass\_track)

# Drums (channel 9)

drum\_track = MidiTrack()

drum\_track.append(MetaMessage('track\_name', name='Standard Drum Kit', time=0))

drum\_track.append(MetaMessage('instrument\_name', name='Drums', time=0))

drum\_track.append(Message('program\_change', program=0, channel=9, time=0))

for b in range(total\_beats):

if b % 2 == 1:

drum\_track.append(Message('note\_on', note=ride\_cymbal, velocity=40, time=0, channel=9))

drum\_track.append(Message('note\_off', note=ride\_cymbal, velocity=40, time=int(ticks\_per\_beat \* 0.6), channel=9))

else:

drum\_track.append(Message('note\_on', note=0, velocity=0, time=int(ticks\_per\_beat), channel=9))

mid.tracks.append(drum\_track)

# Pad (channel 3)

pad\_track = MidiTrack()

pad\_track.append(MetaMessage('track\_name', name='String Ensemble 1', time=0))

pad\_track.append(MetaMessage('instrument\_name', name='String Pad', time=0))

pad\_track.append(Message('program\_change', program=48, channel=3, time=0))

for chord in pad\_chords:

for i, note in enumerate(chord):

pad\_track.append(Message('note\_on', note=note + 12, velocity=15, time=0 if i > 0 else 0, channel=3))

for i, note in enumerate(chord):

pad\_track.append(Message('note\_off', note=note + 12, velocity=15, time=int(ticks\_per\_beat \* 4) if i == 0 else 0, channel=3))

mid.tracks.append(pad\_track)

# save

mid.save(output\_file)

print(f" full jazz combo saved as: {output\_file}")

**Code explaination:**

1. Duration and Bar Calculation

Code:  
 y, sr = librosa.load(audio\_path)  
 duration\_sec = librosa.get\_duration(y=y, sr=sr)  
 beats\_total = int((bpm / 60) \* duration\_sec)  
 bars = beats\_total // 4  
 total\_beats = bars \* 4  
   
Explanation:  
 - Loads the input audio and measures its length.  
 - Converts that length into total musical beats.  
 - Rounds the total number of beats to the nearest multiple of 4 to ensure all bars are complete.

1. Piano Melody (Channel 0)

Code:  
 if random.random() < 0.5:  
 phrase = random.choice(melody\_phrases)  
 for note in phrase:  
 melody\_track.append(Message('note\_on', ...))  
 melody\_track.append(Message('note\_off', ...))  
 rest = int(ticks\_per\_beat \* random.uniform(1.0, 3.0))  
 else:  
 rest = int(ticks\_per\_beat \* random.uniform(2.0, 4.0))  
   
Explanation:  
 - Randomly selects whether to play a melody or insert a pause.  
 - If playing: selects a short phrase and plays each note with 1.2 beat duration.  
 - Adds a short rest (1.0–3.0 beats) between phrases.  
 - If skipping the phrase, a longer rest (2.0–4.0 beats) is inserted.  
 - This makes the melody sound natural.

1. Piano Chords (Channel 1)

Code:  
 for chord in chords:  
 for note in chord:  
 chord\_track.append(Message('note\_on', ...))  
 for note in chord:  
 chord\_track.append(Message('note\_off', ...))  
   
Explanation:  
 - Plays three-note chords (triads) every 4 beats.  
 - Notes start together and last one full bar.  
 - Channel 1 is used for separation from other instruments.

1. Bass (Channel 2)

Code:  
 pattern = [root, root + 3, root + 5, root + 2]  
 for note in pattern:  
 bass\_track.append(Message('note\_on', ...))  
 bass\_track.append(Message('note\_off', ...))  
   
Explanation:  
 - Creates a four-note walking bass line based on the chord root.  
 - Each note lasts 1 beat, filling the entire bar.  
 - Channel 2 is used for acoustic bass.

1. Drums (Channel 9)

Code:  
 if b % 2 == 1:  
 drum\_track.append(Message('note\_on', note=51, ...))  
 drum\_track.append(Message('note\_off', note=51, ...))  
   
Explanation:  
 - Adds ride cymbal hits on beats 2 and 4 of every bar.  
 - Uses Channel 9, reserved for percussion.  
 - Empty beats are spaced using silent notes to maintain rhythm.

1. String Pad (Channel 3)

Code:  
 for chord in pad\_chords:  
 for note in chord:  
 pad\_track.append(Message('note\_on', note=note+12, ...))  
 for note in chord:  
 pad\_track.append(Message('note\_off', note=note+12, ...))  
   
Explanation:  
 - Plays soft background harmonies using root and fifth notes.  
 - Each chord is transposed one octave up and held for 4 beats.  
 - Channel 3 is used to keep it separate from other instruments.

1. Saving the MIDI File

Code:  
 mid.tracks.append(track)  
 mid.save(output\_file)  
   
Explanation:  
 - After all tracks are created, they are combined into one MIDI object.  
 - The complete file is saved as a “jazz.mid”

**Result:**

* **Techincal consideration:**
* Duration matched using Librosa and fixed BPM.
* Each instrument assigned a separate MIDI channel.
* Melody uses randomized phrasing; results vary without seed.
* Velocity ranges were adjusted to balance instruments.
* Mido used for timing (480 ticks/beat); no overlap issues.
* Ran under Ubuntu for compatibility.
* **Output MIDI file:**

The generated jazz.mid file successfully simulates a complete jazz song that matches duration of the recorded audio. Each instrument contributes uniquely:

* The melody mimics improvisation with random phrases and natural rests.
* Piano chords and walking bass establish harmonic and rhythmic structure.
* Ride cymbals emphasize beats 2 and 4, reinforcing the jazz feel.
* A soft string pad adds background warmth without overpowering the lead instruments.

**Tools used:**

|  |  |
| --- | --- |
| **Tool** | **Purpose** |
| Python | Programming the MIDI generator |
| Librosa | Extracting .wav duration |
| Mido | Writing MIDI messages and organizing channels |
| Ubuntu | Running the Python code with better compability |

**3.5.2. Jazz Song Mixing Process**

The goal of this step was to enrich the recorded audio (group3.wav) with a dynamic jazz backing track. This was accomplished by converting a generatively created MIDI file (jazz.mid) into a waveform and combining it with the vocal audio track. The mixing process was fully automated using a Python-based pipeline.

1. **Code**

from pydub import AudioSegment

import subprocess

import os

def mix\_audio(wav1, wav2, output\_file="final\_mix.wav", gain\_db=20):

audio1 = AudioSegment.from\_file(wav1)

audio2 = AudioSegment.from\_file(wav2).apply\_gain(gain\_db) # tăng âm lượng MIDI

mixed = audio1.overlay(audio2)

mixed.export(output\_file, format="wav")

print(f"✅ Mixed audio saved to: {output\_file}")

def convert\_midi\_to\_wav(midi\_file, output\_wav, soundfont\_path):

"""Chuyển đổi file MIDI sang WAV sử dụng FluidSynth"""

# Kiểm tra xem file soundfont có tồn tại không

if not os.path.exists(soundfont\_path):

print(f"❌ Không tìm thấy file soundfont tại: {soundfont\_path}")

return False

cmd = f'fluidsynth -ni "{soundfont\_path}" "{midi\_file}" -F "{output\_wav}" -r 44100'

print(f"Đang chạy lệnh: {cmd}")

result = subprocess.run(cmd, shell=True)

if os.path.exists(output\_wav) and os.path.getsize(output\_wav) > 0:

print(f"✅ Chuyển đổi thành công! File WAV đã được tạo: {output\_wav}")

return True

else:

print("❌ Chuyển đổi thất bại. Lỗi khi chạy FluidSynth.")

return False

if \_\_name\_\_ == "\_\_main\_\_":

# Các đường dẫn file

original\_wav = "group3.wav" # đổi tên file ghi âm của bạn

midi\_file = "jazz.mid"

midi\_wav = "jazz.wav" # sẽ tạo ra từ MIDI

# Đường dẫn đến file soundfont - thay đổi cho đúng

# Thay đổi đường dẫn tới file soundfont trên Ubuntu

# soundfont\_path = "/usr/share/sounds/sf2/FluidR3\_GM.sf2"

soundfont\_path = r"FluidR3\_GM.sf2"

# Kiểm tra nếu người dùng đã sao chép file vào thư mục hiện tại

if os.path.exists("FluidR3\_GM.sf2"):

soundfont\_path = "FluidR3\_GM.sf2"

print("Đã tìm thấy file soundfont trong thư mục hiện tại.")

print("\n🎵 Đang chuyển đổi MIDI sang WAV...")

success = convert\_midi\_to\_wav(midi\_file, midi\_wav, soundfont\_path)

if success:

# Bước 2: Mix 2 file WAV lại

if os.path.exists(original\_wav):

input("\n👉 Nhấn Enter để tiếp tục mix hai file WAV...")

mix\_audio(original\_wav, midi\_wav, "jazz\_mix.wav", gain\_db=20)

else:

print(f"❌ Không tìm thấy file ghi âm: {original\_wav}")

else:

print("Vui lòng kiểm tra đường dẫn đến file soundfont hoặc sao chép file FluidR3\_GM.sf2 vào thư mục hiện tại.")

1. **MIDI-to-WAV Conversion Using FluidSynth**

* MIDI files do not store audio data; they are symbolic representations (like sheet music), specifying *which* notes to play, *when*, and *how* (velocity, duration, etc.). To play MIDI as sound, we need a **synthesizer** and a **sound library**.
* **FluidSynth** is a real-time software synthesizer that generates audio by interpreting MIDI instructions using **SoundFont 2 (.sf2)** files.
* It supports low-latency rendering, multiple voices, and is cross-platform.
* Code breakdown:

fluidsynth -ni "FluidR3\_GM.sf2" "jazz.mid" -F "jazz.wav" -r 44100

|  |  |
| --- | --- |
| **Flag** | **Meaning** |
| -ni | Non-interactive mode (run without GUI) |
| FluidR3\_GM.sf2 | High-quality GM SoundFont |
| -F "jazz.wav" | Output rendered audio as WAV |
| -r 44100 | Sample rate (44.1kHz = CD quality) |

* Windows: FluidSynth must be installed or run through bundled .exe tools.
* Ubuntu/Linux: Available via package managers:

sudo apt install fluidsynth  
sudo apt install fluid-soundfont-gm

* SoundFont Paths:
* Linux default: /usr/share/sounds/sf2/FluidR3\_GM.sf2
* Windows: You may need to download and place FluidR3\_GM.sf2 in the project folder manually.
* The code checks if the .sf2 file exists and validates the output WAV using:

os.path.exists(output\_wav) and os.path.getsize(output\_wav) > 0

1. **Audio Mixing with PyDub**

* Library Used: [pydub](https://github.com/jiaaro/pydub)

PyDub provides a high-level API for manipulating audio in Python. It wraps around ffmpeg for decoding/encoding and allows for:

* Overlaying tracks
* Applying gain/volume changes
* Slicing, joining, and exporting
* Mixing code and Algorithm used:
* The mixing uses **waveform overlay**:

mixed = audio1.overlay(audio2)

* This is equivalent to **vector addition** of two audio samples at each time point:

mixed\_sample[i] = audio1[i] + audio2[i]

=> It aligns samples starting at **time = 0s**, ensuring synchronous playback.

* Gain Adjustment :

audio2 = AudioSegment.from\_file(wav2).apply\_gain(20)

=> Gain is used to amplify the MIDI-generated audio because it tends to be lower in volume compared to human voice recordings. +20 dB increases amplitude by a factor of ~10.

* Export output :

mixed.export(output\_file, format="wav")

=> This saves the result as uncompressed PCM audio (.wav), preserving quality.

1. **Additional Technical Considerations**

* Latency and Alignment:
* overlay() assumes perfect time alignment (no latency).
* If the MIDI file starts with a rest or the original recording has silence, misalignment may occur. In our case, timing was manually ensured during MIDI generation.

Table 3‑1. Alternatives considered

|  |  |  |
| --- | --- | --- |
| **Tool** | **Pros** | **Cons** |
| Audacity | GUI-based, intuitive | Manual, not scalable |
| ffmpeg | Powerful CLI tool | Complex syntax |
| DAWs (Ableton, FL Studio) | High control | Non-programmatic |

We chose **PyDub + FluidSynth** due to their scriptability, open-source nature, and ease of integration in a Python workflow.

1. **Final Outcome**

* The final product (jazz\_mix.wav) contains synchronized voice and jazz instrumentation.
* No clipping, drop-outs, or desynchronization was observed.
* The method is repeatable, adjustable (e.g., gain), and cross-platform

1. **Summary of Libraries & Tools**

Table 3‑2. Summary of Libraries & Tools

|  |  |  |
| --- | --- | --- |
| **Component** | **Library / Tool** | **Purpose** |
| pydub | PyDub (Python) | Audio manipulation: loading, overlaying, gain control, export |
| subprocess | Python standard lib | Execute shell commands to call FluidSynth |
| os | Python standard lib | Check file existence, validate WAV file output |
| FluidSynth | External Synth Engine | Convert MIDI to waveform audio using SoundFont rendering |
| FluidR3\_GM.sf2 | SoundFont file | High-quality General MIDI instrument samples |

**Final Remarks.**

This module demonstrates a practical, scriptable audio engineering workflow using open-source tools instead of traditional DAWs. It is fully automatable, platform-independent, and easy to integrate into broader multimedia processing pipelines. The process ensures audio quality, synchronization, and flexibility for different use cases, including educational or creative production environments.

# Source Code & GitHub Repositories

* **Github links & Code**:

https://github.com/haimayoi/ADPCM-and-MP3-compression

# Conclusions

## Summary of Key Findings

Our project has successfully implemented a complete ADPCM-based audio compression and reconstruction pipeline, with comparative analysis against MP3 compression. The primary findings of our work include:

1. **Spectral Analysis**:

* Our recorded speech signal showed characteristic concentration in the lower frequency bands (below 4 kHz), with energy peaks between 0-1 kHz. This frequency distribution is typical for human speech and explains why ADPCM, which is optimized for speech signals, performed exceptionally well on our dataset.

1. **ADPCM Implementation**:

* We developed both an FFmpeg-based ADPCM implementation and a custom Python IMA ADPCM codec. The custom implementation provided deeper insights into the adaptive prediction mechanisms that make ADPCM effective for speech compression.

1. **Compression Performance**: The quantitative analysis revealed significant differences between the two codecs:

* ADPCM achieved a PSNR of 52.16 dB
* MP3 compression at 128 kbps yielded a PSNR of 16.87 dB
* This represents a difference of 35.29 dB in favor of ADPCM

1. **Temporal PSNR Analysis**: The segment-by-segment PSNR measurements showed that ADPCM maintained consistently high quality throughout the recording:

* ADPCM PSNR ranged between 40-80 dB across all segments
* MP3 PSNR fluctuated between 10-57 dB, with most segments below 25 dB
* Both codecs displayed similar patterns of quality variation across time segments

1. **MIDI Generation and Audio Mixing**:

* We successfully created an algorithmically generated jazz accompaniment that matched our 4-minute 3-second recording and mixed it with the voice recording using an automated Python pipeline.

## Evaluation of Compression Methods

**1. ADPCM Compression**

ADPCM (Adaptive Differential Pulse Code Modulation) demonstrated exceptional performance in our testing:

* **Signal Fidelity**: With a PSNR of 52.16 dB, ADPCM preserved the original signal characteristics with minimal perceptible degradation. This high PSNR value indicates near-transparent quality reproduction.
* **Compression Efficiency**: While not achieving the extreme reduction ratios of perceptual codecs like MP3, ADPCM still provided substantial file size reduction compared to uncompressed PCM.
* **Consistency**: The temporal analysis showed that ADPCM quality remained consistently high throughout the recording, rarely dropping below 40 dB even in challenging signal segments.
* **Implementation**: Our custom Python IMA ADPCM codec successfully compressed 1010 bytes of PCM data (505 samples) into 256 bytes, achieving a compression ratio of approximately 4:1 while maintaining excellent signal quality.

**2. MP3 Compression**

MP3 compression (MPEG-1 Audio Layer III) showed the following characteristics:

* **Compression Ratio**: The MP3 implementation achieved superior compression ratios, reducing file sizes to approximately 1/10th of the original WAV file (at 128 kbps).
* **Quality Trade-off**: This higher compression came at a significant cost to signal fidelity, with a PSNR of only 16.87 dB.
* **Temporal Variation**: The segment-by-segment analysis revealed that MP3 quality varied considerably throughout the recording, with occasional peaks of acceptable quality (up to 57 dB) but predominantly operating below 25 dB.
* **Implementation Complexity**: The LAME encoder implementation required more parameters and configuration than ADPCM, reflecting the greater complexity of perceptual audio coding.

**3. Comparative Analysis and Practical Applications**

Our results demonstrate a clear quality-size tradeoff between the two codecs:

* **ADPCM is superior for**: Speech preservation, professional recordings, archival purposes, and applications requiring high fidelity, particularly for voice content.
* **MP3 is preferable for**: Applications where extreme file size reduction is crucial, such as streaming services, mobile applications, and general music distribution where some quality loss is acceptable.
* **Quantitative Advantage**: The 35.29 dB PSNR advantage of ADPCM over MP3 represents a substantial quality difference that would be immediately noticeable to listeners in critical applications.

## Lessons Learned and Knowledge Gained

Through this project, our team has gained valuable experience and insights in several areas:

1. **Audio Compression Techniques**:

* Understanding of predictive coding principles (ADPCM) vs. perceptual coding (MP3)
* Implementation experience with both approaches
* Practical knowledge of codec performance characteristics and their suitability for different audio content

1. **Digital Signal Processing**:

* Application of Fast Fourier Transform for spectral analysis
* Understanding frequency characteristics of human speech signals
* Experience in quantitative quality assessment using PSNR

1. **Programming and Software Tools**:

* Multi-language programming experience (Python, C, MATLAB)
* Proficiency with multimedia libraries (librosa, pydub, LAME, mido)
* Cross-platform development considerations for audio processing tools
* Effective team collaboration through Git version control

1. **Practical Applications**:

* Complete audio processing pipeline from recording to compression to mixing
* Integration of multiple technologies (MIDI, ADPCM, FFmpeg, FluidSynth)
* Understanding of the digital music production workflow

## Future Development Directions

Based on our experiences and results, we identify several promising directions for future work:

1. **Algorithm Optimization**: Enhance our custom ADPCM implementation for better performance, and explore adaptive bitrate options for both ADPCM and MP3.
2. **Extended Codec Comparison**: Include additional codecs such as AAC, Opus, and FLAC in future analyses to provide a more comprehensive understanding of modern audio compression techniques.
3. **User Interface Development**: Create a graphical interface for non-technical users to easily compress, decompress, and mix audio without requiring programming knowledge.
4. **Machine Learning Applications**: Investigate how modern machine learning algorithms could improve compression efficiency and audio quality through adaptive parameter selection.
5. **Real-time Processing**: Explore the feasibility of implementing these compression and mixing techniques in real-time applications for live audio streaming or teleconferencing.

In conclusion, this project has successfully met all the initial requirements while providing our team with an opportunity to apply modern audio processing techniques in a practical context. By combining theoretical understanding with hands-on implementation, we have gained deep insights into audio compression and reconstruction processes, particularly the effectiveness of ADPCM in preserving speech signal quality. The acquired knowledge and skills form a solid foundation for more complex projects in multimedia and digital signal processing.

# Appendix:

## 6.1 Bùi Thanh Thảo:

* + 1. Task assigned

In this project, I was responsible for the following tasks:

* Managing the group’s Jira project page includes creating tasks, assigning responsibilities, and tracking progress throughout the development process.
* Using Matlab to show spectrum of the recorded audio signal and comments on its energy distribution over the frequency axis.
* Researching the IMA ADPCM compression algorithm and its specifications.
* Using FFmpeg t
* Implementing the ADPCM codec in Python, including both encoding and decoding functions.
* Contributing to the final report, especially section 3.3 – ADPCM compression
  + 1. Technical contribution

My specific contributions include:

* Managed the team’s Jira board, created a clear task structure, and maintained workflow transparency across all project stages.
* Created a MATLAB script to analyze the audio signal’s frequency spectrum using FFT. This allowed us to observe how ADPCM compression affects energy distribution across different frequency bands.
* Wrote two Python-based audio compression tools:
* The first script used FFmpeg to automate .wav to .adpcm conversion.  
  The second script reimplemented IMA ADPCM manually, including prediction, delta calculation, quantization, and nibble encoding. The decoder reversed this process to recover 16-bit PCM samples.
* Tested the manually compressed output by comparing it with FFmpeg’s result, ensuring that our implementation followed the ADPCM standard. I also assisted teammates in integrating this codec into the overall processing pipeline.
  + 1. Tools and skills gained

**Tools Used:**

* Python: codec implementation, audio block processing
* MATLAB: FFT analysis, signal visualization
* FFmpeg: command-line audio compression
* Jira: team collaboration, task management

**Skills Gained:**

* Understanding of ADPCM predictive compression techniques
* Integration of external tools (FFmpeg) in Python-based workflows
* Hands-on experience implementing audio codecs from scratch
* Frequency domain analysis and spectrum interpretation
* Project coordination and workflow tracking using Jira
* Integration of external tools (FFmpeg) in Python-based workflows

## 6.2 Lê Hà Hải Vân:

* + 1. Task assigned:

In this project, I was responsible for the following tasks:

* Researching the MP3 compression technique and its characteristics.
* Implementing **MP3 compression** in C using the **LAME encoder**.
* Writing **MATLAB scripts to calculate PSNR** between the original, ADPCM, and MP3 audio files.
* **Plotting PSNR bar charts** and signal spectrum comparisons.
* Contributing to the final report, especially section **3.4 – PSNR Analysis of ADPCM and MP3 Compression**.
* **Managing the project’s GitHub repository**: uploading files, writing README, and organizing folder structure.
  + 1. Technical contribution:

My specific contributions include:

* **Implemented MP3 audio compression** using the LAME library in C:
  + Set encoder parameters (bitrate, sampling rate, channels, etc.)
  + Processed raw PCM data and encoded it to MP3 format
* **Developed MATLAB code to calculate PSNR**, which includes:
  + Reading audio data from .wav, .mp3, and .adpcm files
  + Synchronizing sample rate and audio length
  + Computing MSE and converting it to PSNR values
* **Plotted PSNR comparison charts** to visualize and compare the compression quality
* **Managed the team’s GitHub repository**:
  + Uploaded and organized code and audio files
  + Wrote README and added project documentation
  + Helped resolve file conflicts and version control issues
* Analyzed the results and wrote the evaluation section in the report
  + 1. Tools and skills gained:

**Tools Used:**

* **MATLAB**: audio signal processing, PSNR calculation, data visualization
* **LAME MP3 Encoder** (in C): for compressing .wav to .mp3
* **GitHub**: version control, team collaboration, and documentation
* **PSNR analysis**

**Skills Gained:**

* Understanding and applying **audio compression techniques** (ADPCM and MP3)
* Programming audio **codec algorithms** in C and MATLAB
* Analyzing and visualizing signal quality using **PSNR metrics**
* **Teamwork & time management** in a technical group project
* Managing a collaborative **GitHub repository**
* Writing **technical documentation and structured reports** for multimedia processing

## 6.3 Phạm Minh Tuyên:

**6.3.1. Task Assigned:**

In this project, I was responsible for the following tasks:

* Converting a programmatically generated MIDI file into an audible WAV format using **FluidSynth**.
* Writing a complete Python script to **mix the MIDI-converted audio with the recorded speech audio** (group3.wav) using the **pydub** library.
* Ensuring correct synchronization, gain balancing, and export of the final product (jazz\_mix.wav).
* Handling **cross-platform compatibility** (Windows vs. Ubuntu) in terms of SoundFont configuration and tool installation.
* Contributing to the final report, especially **section 3.5.2 – MIDI to Audio Mixing**.

**6.3.2. Technical Contribution:**

My specific technical contributions include:

* Developed a Python script (mix\_jazz.py) which:
  + Converts MIDI to WAV using fluidsynth via the subprocess module.
  + Checks for valid .sf2 SoundFont file, with fallback logic.
  + Uses pydub.AudioSegment to load and amplify the MIDI-generated WAV by **+20 dB**.
  + Overlays the MIDI WAV and voice WAV synchronously with .overlay() and exports the result.
* Handled compatibility and environment setup:
  + Provided instructions and validation logic for both **Ubuntu** (using apt install fluidsynth) and **Windows** (manual setup, environment variables).
  + Identified common issues like missing DLLs or volume imbalance and resolved them in code.
* Validated and tested the final mix:
  + Auditory validation to ensure no clipping or alignment errors.
  + Empirically tuned the gain factor for optimal mix balance.
* Wrote detailed explanation and analysis in **Section 3.5.2** of the report.

**6.3.3. Tools and Skills Gained:**

**Tools Used:**

* **Python**: Core scripting language.
* **PyDub**: For audio processing and overlay.
* **FluidSynth**: Synthesizer to convert MIDI to WAV.
* **SoundFont (FluidR3\_GM.sf2)**: GM instrument bank.
* **GitHub**: Code hosting and documentation.
* **OS terminal (Windows & Ubuntu)**: Environment setup and CLI tools.

**Skills Gained:**

* Understanding the distinction between symbolic MIDI data and waveform audio.
* Automating audio conversion and mixing processes using Python.
* Cross-platform software setup and dependency resolution.
* Signal processing concepts: gain adjustment, synchronization.
* Writing technical documentation and collaborating on a structured multimedia report.
* Delivering reproducible and scalable audio workflows.

## Đỗ Đại Doanh:

* + 1. **Task Assigned:**

In this project, I was responsible for the following tasks:

* Writing a Python script to generate a multi-track jazz-style MIDI file based on the duration of audio input.
* Using the **librosa** library to calculate the total duration and convert it into beats and bars for synchronization.
* Creating structured instrument tracks: piano melody, piano chords, acoustic bass, ride cymbal, and pad chords using the **mido** library.
* Managing MIDI channel assignment and velocity control to prevent overlap.
* Verifying the output ensuring alignment, correct phrasing.
* Contributed to the report section 3.5.1

**6.4.2. Technical Contribution:**

My specific technical contributions include:

* Developed a Python script (jazz.py) which:
  + Loads a .wav file and extracts its duration using **librosa.get\_duration()**.
  + Calculates total beats and bars using a fixed BPM and maps them into musical time.
  + Generates multiple instrument tracks (melody, chords, bass, drums, pad) using the **mido** library.
  + Assigns each instrument to a different **MIDI channel** to prevent overlap.
  + Implements randomized melody phrasing with controlled velocity and rest intervals.
  + Structures all note-on and note-off events.
  + Exports the final synchronized multi-track arrangement as a jazz.mid file.
* Verified the MIDI file to ensure:
  + Correct synchronization with the original voice track.
  + Clear separation of instruments by channel.
  + No overlapping notes or timing errors.
  + Musical consistency in phrasing and rhythm.

**6.4.3. Tools and Skills Gained:**

**Tools Used:**

* **Python:**  
  Used as the main programming language for writing and executing the MIDI generation script.
* **Librosa library:**  
  Applied to load .wav files and extract the audio duration to calculate the musical structure (beats and bars).
* **Mido library:**  
  Used to build and write MIDI files, manage MIDI messages (note\_on, note\_off), assign channels, and control timing (ticks per beat).
* **Ubuntu:**  
  Provided an environment compatible with audio libraries like Librosa and Mido, ensuring smoother execution of the script.

**Skills Gained:**

* Learned how to translate audio duration into musical timing using BPM and bar calculations.
* Gained experience generating multi-instrument MIDI tracks using programmatic logic.
* Understood how MIDI channels, note velocities, and timing work.
* Applied randomized logic to simulate phrasing in melody generation.
* Developed and tested code in Ubuntu.
* Practiced validating output audio and adjusting the rhythm.