Encrypted FM

Erasmus Traineeship Project

Supervisor: Alex Davidson

Students: Ana-Maria Nanu
Lavinia-Marilena Angan

Table of contents

Introduction

The general idea of this project is to explore the feasibility of using FM radio as a method for transmitting private communications. The primary objective is to determine whether FM radio channels can effectively support secure communication services, particularly in environments where noise increases with distance.

The project is structured into several key phases: devising an experimental methodology to assess noise rates in FM radio channels across various distances, conducting field work to record and analyze FM radio transmissions, and calculating the expected noise for secure communications.

In the initial research phase, the goal is to gain a better understanding of basic information about waveforms, sample visualization, and analyzing data values in hexadecimal and decimal formats.

In the analysis phase, we will need to record several songs from the radio using a receiver and compare them with their original versions. This will allow us to visualize the differences between samples, specifically the noise introduced by radio transmission. With this knowledge, we should be prepared to implement a software script that extracts the components of a WAV audio file, converts the hexadecimal values of the samples to their decimal equivalents, calculates the difference between each aligned sample to isolate the noise, and creates an audio file to hear the generated noise.

Finally, in the solution phase, it was expected to implement a proof-of-concept cryptographic communication protocol and analyze its performance under different noise conditions.

This project also provided opportunities to develop essential skills, including effective communication, working in a multicultural environment, and deepening our understanding of cryptography and error-correcting codes.

The expected outcomes include a comprehensive dataset on noise rates in FM radio transmissions, a scientific report detailing the statistical analysis and its implications for cryptographic communication, and a software implementation of a noise detection and audio files comparison protocol.

WAV Files Format

What is a WAV file?

WAV, short for Waveform Audio File Format, is a subset of Microsoft's Resource Interchange File Format (RIFF) designed for storing digital audio files. This format does not compress the bitstream, preserving the audio in various sampling rates and bitrates.

WAV File Format

The WAV file format, as part of Microsoft's RIFF specification, begins with a file header followed by a sequence of data chunks. A WAV file contains a single "WAVE" chunk, which includes two sub-chunks:

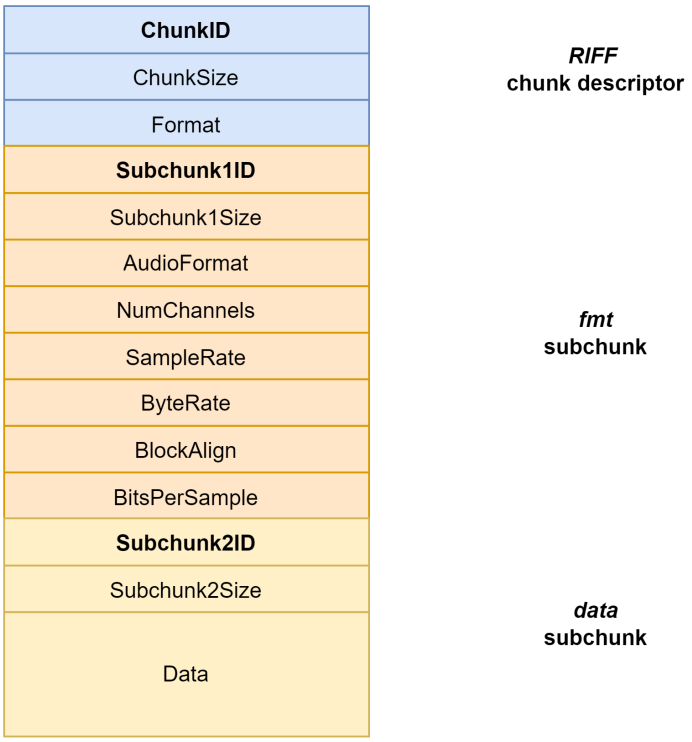
- **"fmt" chunk**: Specifies the data format.
- **"data" chunk**: Contains the actual audio sample data.

Header

The header is the beginning of a WAV (RIFF) file. It provides specifications on the file type, sample rate, sample size, bit size, and overall length. The header of a WAV (RIFF) file is 44 bytes long and follows this format:

Positions	Offset(h)	Size (Bytes)	Description
1-4	00-03	4	RIFF string
5-8	04-07	4	File size minus 8 bytes
9-12	08-0B	4	"WAVE" string
13-16	0C-0F	4	"fmt " string
17-20	10-13	4	Length of format data
21-22	14-15	2	Type of format (1 is PCM)
23-24	16-17	2	Number of channels
25-28	18-1B	4	Sample rate
29-32	1C-1F	4	Byte rate
33-34	20-21	2	Block align
35-36	22-23	2	Bits per sample
37-40	24-27	4	"data" string
41-44	28-2B	4	Size of data chunk

Sample values are given above for a 16-bit stereo source.



What's Bit Size?

Bit size determines how much information can be stored in a file. For most purposes today, a 16-bit size is standard. While 8-bit files are smaller (half the size), they offer less resolution. Bit size affects amplitude:

- **8-bit recordings**: Have 256 amplitude levels (0 to 255).
- **16-bit recordings**: Have 65,536 amplitude levels (-32,768 to 32,767).

Higher bit resolution results in a greater realistic dynamic range. CD-Audio uses 16-bit samples.

What is Sample Rate?

Sample rate is the number of samples per second. CD-Audio has a sample rate of 44,100 Hz, meaning one second of audio contains 44,100 samples. DAT tapes have a sample rate of 48,000 Hz. The highest frequency a file can accurately reproduce is half of the sample rate.

What are Channels?

Channels refer to the number of separate recording elements in the data. One channel is mono, while two channels are stereo. This document discusses both single and dual-channel recordings.

Data

The data consists of individual samples. Each sample size is determined by the bit size and the number of channels. For example:

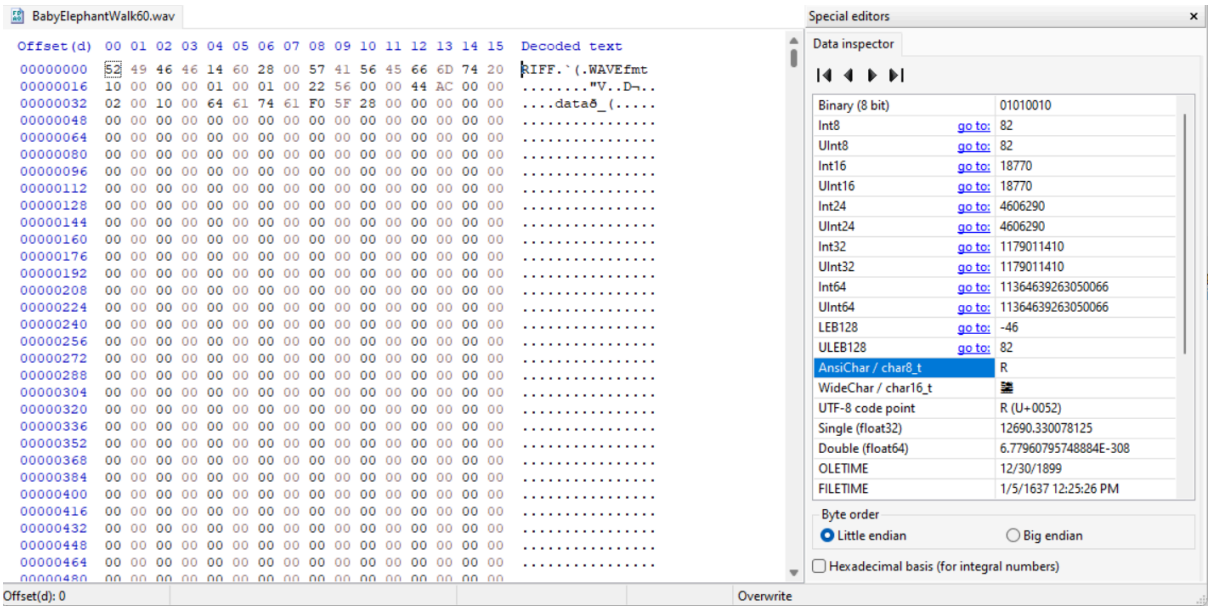
- **Monaural (single channel), 8-bit recording**: 8 bits per sample.
- **Monaural, 16-bit recording**: 16 bits per sample.
- **Stereo, 16-bit recording**: 32 bits per sample (16 bits for each channel).

Samples are placed end-to-end to form the data. For instance, four samples (s1, s2, s3, s4) are arranged as s1s2s3s4.

Examples

First WAV file - BabyElephantWalk60.wav

The WAV file format can be inspected using a Hex Editor, in this case HxD, allowing to view the header values that provide information about the file.



RIFF Header:

Offset 00-03: 52 49 46 46 (ASCII: RIFF) - Marks the file as a RIFF file.
Offset 04-07: 14 60 28 00 - Size of the file minus 8 bytes.
Offset 08-0B: 57 41 56 45 (ASCII: WAVE) - Identifies the file as a WAVE file.

Format Chunk:

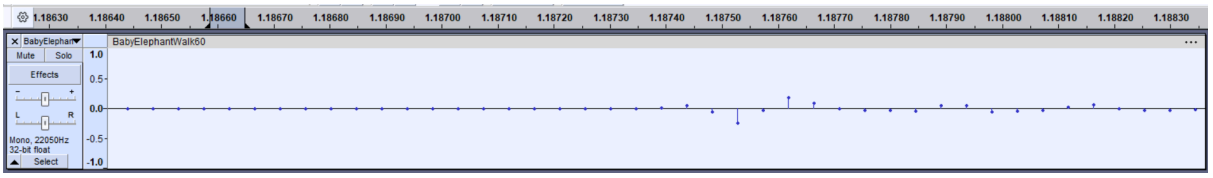
Offset 0C-0F: 66 6D 74 20 (ASCII: fmt) - Format chunk identifier.
Offset 10-13: 10 00 00 00 - Size of the format chunk (16 bytes for PCM).
Offset 14-15: 01 00 - Audio format (1 is PCM - Pulse Code Modulation).
Offset 16-17: 01 00 - Number of channels (1 for mono).

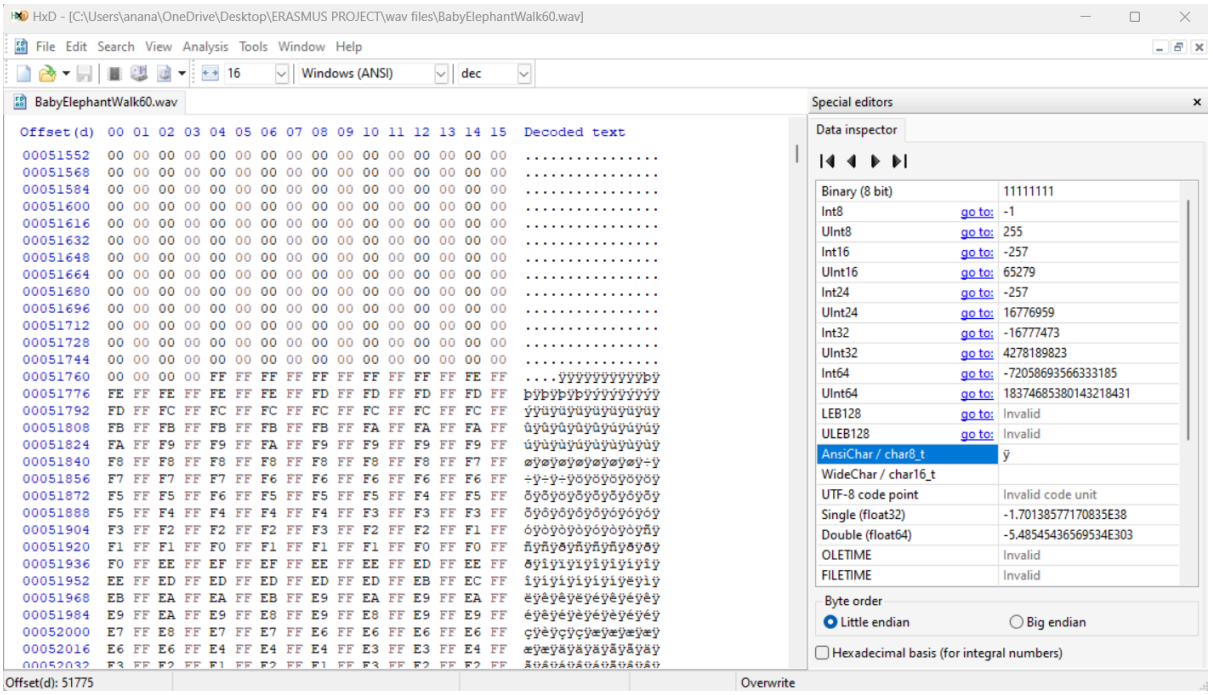
Offset 18-1B: 22 56 00 00 - Sample rate (22050 Hz).
Offset 1C-1F: 44 AC 00 00 - Byte rate (44100 bytes/second).
Offset 20-21: 02 00 - Block align (2 bytes).
Offset 22-23: 10 00 - Bits per sample (16 bits).

Data Chunk:

Offset 24-27: 64 61 74 61 (ASCII: data) - Data chunk identifier.
Offset 28-2B: F0 5F 28 00 - Size of the data section (2,645,872 bytes).

Audio data starts immediately after the SubChunk2 Size field. Samples with a value of 00 00 will appear as a flat line at the center of the waveform display in Audacity. This indicates silence. The amplitude (height) of the waveform corresponds to the absolute value of the sample value. Larger values result in higher peaks or deeper troughs.





Second WAV file - 7seconds.wav

In the first picture we can see the discrete points of the actual wave from Audacity and in the second one is the header and some of the initial data of a WAV file, providing detailed file structure information.. The initial part of the file contains the header information (First 44 bytes), which conforms to the RIFF specification for WAV files.

RIFF Header:

Offset 00-03: 52 49 46 46 (ASCII: RIFF) - Marks the file as a RIFF file.
Offset 04-07: 46 00 00 00 - Size of the file minus 8 bytes.
Offset 08-0B: 57 41 56 45 (ASCII: WAVE) - Identifies the file as a WAVE file.

Format Chunk:

Offset 0C-0F: 66 6D 74 20 (ASCII: fmt) - Format chunk identifier.
Offset 10-13: 10 00 00 00 - Size of the format chunk (16 bytes for PCM).
Offset 14-15: 01 00 - Audio format (1 is PCM - Pulse Code Modulation).
Offset 16-17: 02 00 - Number of channels (2 for stereo).
Offset 18-1B: 44 AC 00 00 - Sample rate (44,100 Hz).
Offset 1C-1F: 10 B1 02 00 - Byte rate (176,400 bytes/second).
Offset 20-21: 04 00 - Block align (4 bytes).
Offset 22-23: 10 00 - Bits per sample (16 bits).

Data Chunk:

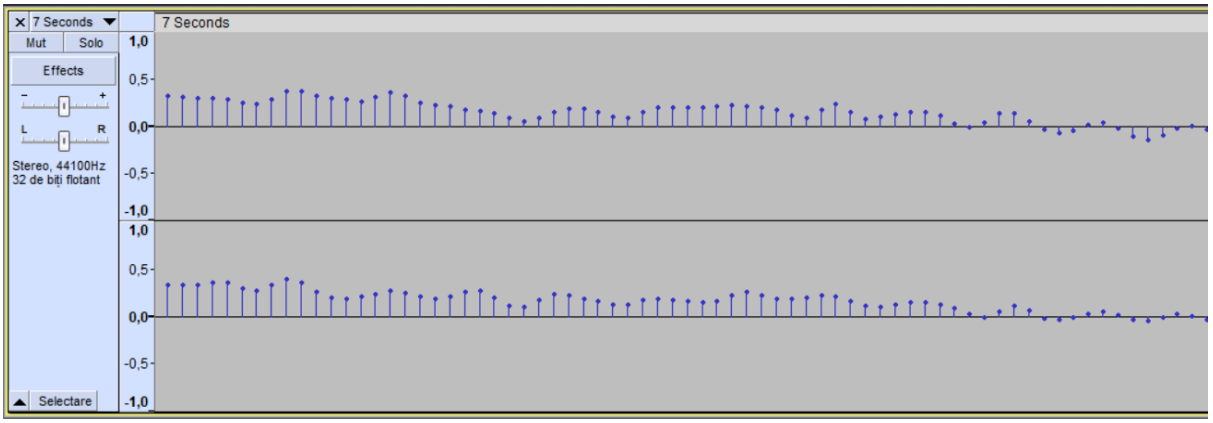
Offset 24-27: 64 61 74 61 (ASCII: data) - Data chunk identifier.
Offset 28-2B: 51 03 00 00 - Size of the data section (849 bytes).

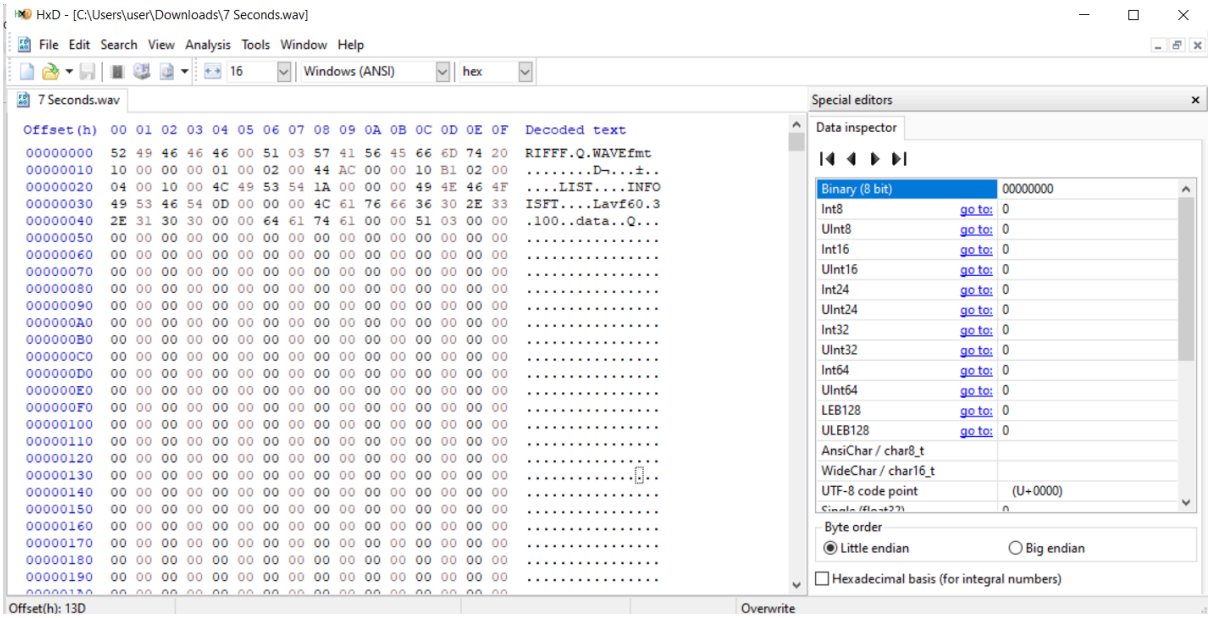
LIST Chunk:

Offset 28-2B: 4C 49 53 54 (ASCII: LIST) - Identifier for a list chunk.
Offset 2C-2F: 1A 00 00 00 - Size of the LIST chunk.
Offset 30-33: 49 4E 46 4F (ASCII: INFO) - Indicates that this chunk contains information.
Offset 34-37: 49 53 46 54 (ASCII: ISFT) - Software identifier.
Offset 38-3B: 4C 61 76 66 36 30 2E 33 (ASCII: Lavf60.3) - Software version that created the file.

Data:

Following the header and metadata chunks, the audio data starts, with bytes corresponding to the actual sound samples. In this screenshot, it appears to start after offset 0x48.





Comparison between modified files

To facilitate a better visualization of each sample, a smaller segment has been extracted from the original file. This approach makes it easier to read the correspondence between each sample and its hexadecimal value.

To verify the accuracy of the representation, we will correlate the values from HxD with the values of each sample in Audacity. The sample values can be approximately read using the scale on the left. These values are normalized to 2^x, where x is the bit size. In the first example, we have a single channel, and the values are represented in 16 bits. Thus, by normalizing to 2^{15}, we obtain sample values that range between 1 and -1.

First example- mono

In the first example, where the .wav file contains a single channel and each sample is read in 16-bit format, we will take 2 bytes from the data chunk for each sample. To simulate the impact of noise on the data, a sample will be modified to observe its effects.
This modification will be made on the second sample.

The values before modification:

Trimmed2.wav 99

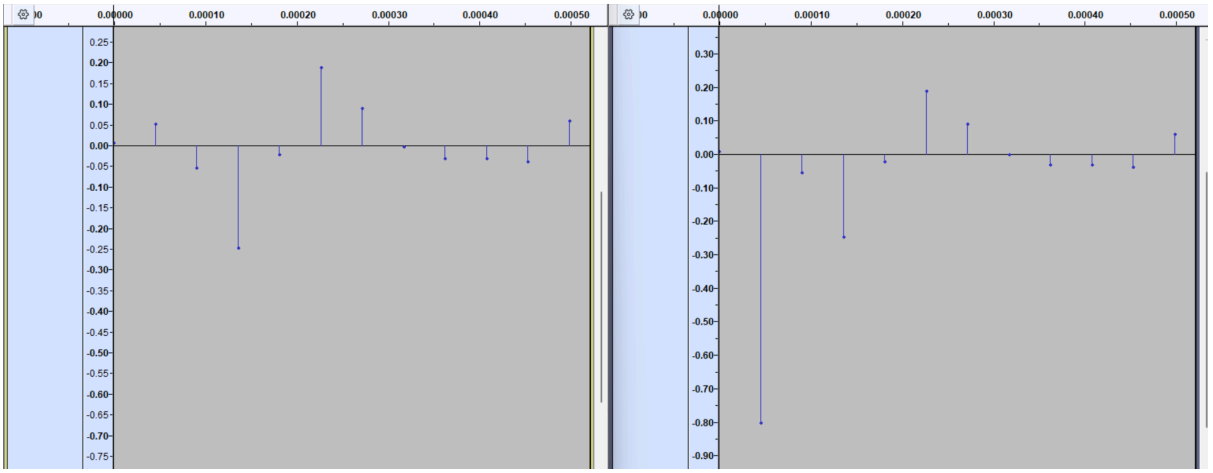
Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	52	49	46	46	58	00	00	00	57	41	56	45	66	6D	74	20	RIFFX...WAVEfmt
00000010	10	00	00	00	01	00	01	00	22	56	00	00	44	AC	00	00	........"V..D¬..
00000020	02	00	10	00	64	61	74	61	34	00	00	00	14	01	BF	06	....data4.....¿.
00000030	3A	F9	86	E0	50	FD	35	18	B4	0B	E0	FF	2E	FC	22	FC	:ùtàPý5.´.àÿ.ü"ü
00000040	1D	FB	A7	07	67	06	50	F9	86	FA	52	FC	1F	04	8F	08	.û§.g.PùtúRü....
00000050	A9	FF	40	FC	AC	FC	04	FE	73	02	EB	00	18	02	45	FF	©ÿ@ü¬ü.þs.ë...Eÿ

The values after modification:

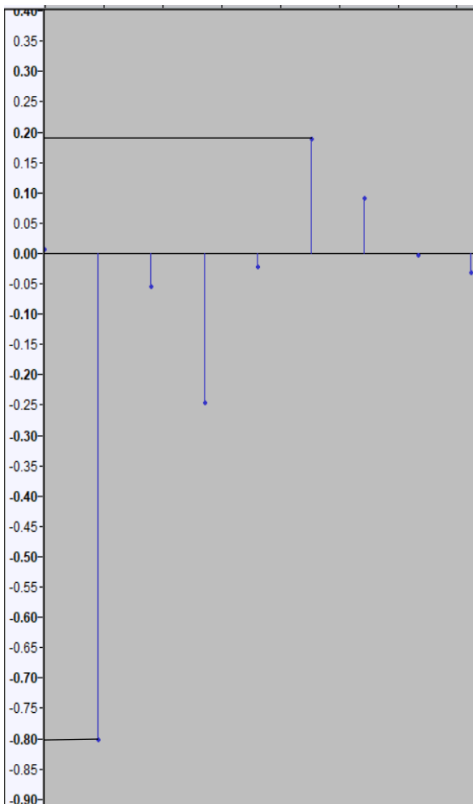
Trimmed2.wav 99

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	52	49	46	46	58	00	00	00	57	41	56	45	66	6D	74	20	RIFFX...WAVEfmt
00000010	10	00	00	00	01	00	01	00	22	56	00	00	44	AC	00	00	........"V..D¬..
00000020	02	00	10	00	64	61	74	61	34	00	00	00	14	01	99	99	....data4.....™™
00000030	3A	F9	86	E0	50	FD	35	18	B4	0B	E0	FF	2E	FC	22	FC	:ùtàPý5.´.àÿ.ü"ü
00000040	1D	FB	A7	07	67	06	50	F9	86	FA	52	FC	1F	04	8F	08	.û§.g.PùtúRü....
00000050	A9	FF	40	FC	AC	FC	04	FE	73	02	EB	00	18	02	45	FF	©ÿ@ü¬ü.þs.ë...Eÿ

Samples before (left) and after (right) modification:



Additionally, verifying the correspondence in Audacity after the modification can be done by reading the amplitude value from the scale. By performing a small calculation, the value 99 99 in hexadecimal corresponds to -26215 in decimal, which will be divided by 2^{15}. This results in -0.8, a value that can be read on the scale in Audacity.



Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	52	49	46	46	58	00	00	00	57	41	56	45	66	6D	74	20	RIFFX...WAVEfmt
00000010	10	00	00	00	01	00	01	00	22	56	00	00	44	AC	00	00	........"V..D¬..
00000020	02	00	10	00	64	61	74	61	34	00	00	00	14	01	99	99	....data4.....™™
00000030	3A	F9	86	E0	50	FD	35	18	B4	0B	E0	FF	2E	FC	22	FC	:ùtàPý5.´.àÿ.ü"ü
00000040	1D	FB	A7	07	67	06	50	F9	86	FA	52	FC	1F	04	8F	08	.û§.g.PùtúRü....
00000050	A9	FF	40	FC	AC	FC	04	FE	73	02	EB	00	18	02	45	FF	©ÿ@ü¬ü.þs.ë...Eÿ

Second sample:

$$(99\ 99)_{16} = (-26215)_{10}$$

$$\frac{-26215}{2^{15}} = -0.8$$

Sixth sample:

$$(35\ 18)_{16} = (6197)_{10}$$

$$\frac{6197}{2^{15}} = 0.189$$

Second example- stereo

I chose to change the bytes for the sixth sample. I noticed previously that one sample consists of 4 bytes for a stereo audio, and when wanting to modify the mentioned sample, I changed the values at positions 40-43. For better observation, I set the value to FF for each of them, which would make the sample value equal to 0. This could be observed afterward when I opened the file in Audacity. The sixth sample now has an amplitude equal to 0.

Before modification:

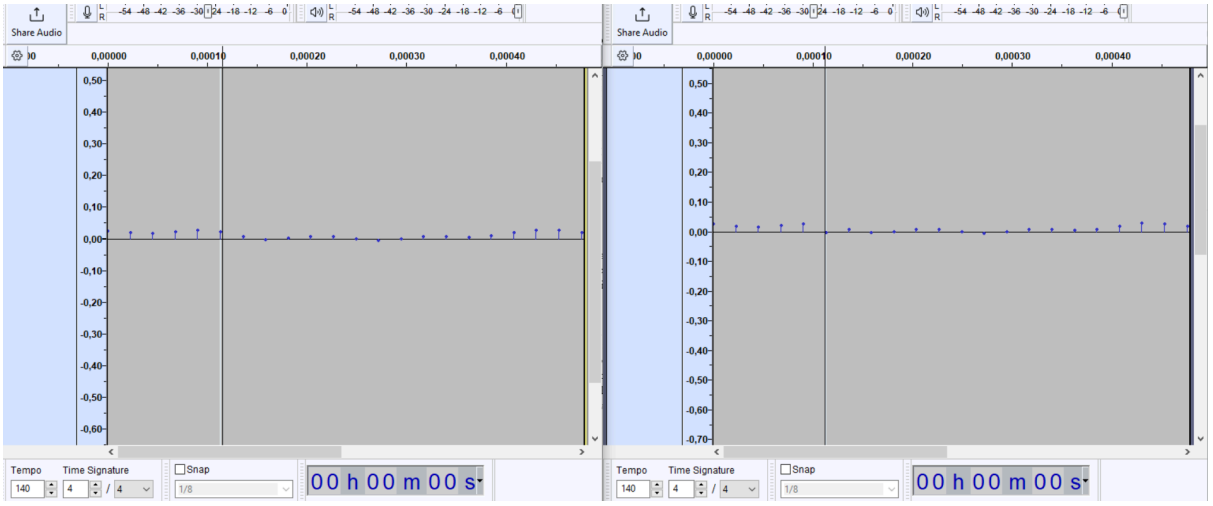
7 Seconds-trimBYTESMODIFICATI.wav | 7 Seconds-trim.wav

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	52	49	46	46	24	01	00	00	57	41	56	45	66	6D	74	20	RIFF$...WAVEfmt
00000010	10	00	00	00	01	00	02	00	44	AC	00	00	10	B1	02	00	........D¬...±..
00000020	04	00	10	00	64	61	74	61	98	00	00	00	6A	03	60	04	....data....j.`.
00000030	AD	02	E5	04	51	02	06	05	FD	02	79	04	BC	03	42	03	..Í.Q...ý.y.L.B.
00000040	13	03	DC	01	24	01	DF	00	DB	FF	7F	00	53	00	6E	00	..Ü.$.ß.Ű˙..S.n.
00000050	47	01	33	00	1B	01	A0	FF	11	00	E8	FE	9C	FF	6C	FE	G.3... ˙..čţś˙lţ
00000060	51	00	66	FE	38	01	C2	FE	46	01	57	FF	E9	00	31	00	Q.fţ8.ÂţF.W˙é.1.
00000070	57	01	4A	01	AA	02	37	02	C0	03	6A	02	A9	03	D3	01	W.J.Ş.7.Ŕ.j.©.Ó.
00000080	A3	02	09	01	C8	01	B9	00	C0	01	EC	00	F2	01	02	01	Ł...Č.ą.Ŕ.ě.ň...
00000090	70	01	83	00	44	00	A2	FF	42	FF	D1	FE	F7	FE	28	FE	p...D.˘˙B˙Ńţ÷ţ(ţ
000000A0	FE	FE	80	FD	87	FE	02	FD	9B	FD	23	FD	4B	FD	13	FE	ţţ€ý‡ţ.ý›ý#ýKý.ţ
000000B0	24	FE	61	FF	5B	FF	5F	00	F8	FF	DA	00	F2	FF	3D	01	$ţa˙[˙_.ř˙Ú.ň˙=.
000000C0	FA	FF	F6	01	4C	49	53	54	2E	00	00	00	49	4E	46	4F	ú˙ö.LIST....INFO
000000D0	49	53	46	54	22	00	00	00	4C	61	76	66	36	30	2E	33	ISFT"...Lavf60.3
000000E0	2E	31	30	30	20	28	6C	69	62	73	6E	64	66	69	6C	65	.100 (libsndfile
000000F0	2D	31	2E	30	2E	33	31	29	00	00	69	64	33	20	2A	00	-1.0.31)..id3 *.
00000100	00	00	49	44	33	03	00	00	00	00	00	20	54	58	58	58	..ID3..... TXXX
00000110	00	00	00	16	00	00	00	53	6F	66	74	77	61	72	65	00	.......Software.
00000120	4C	61	76	66	36	30	2E	33	2E	31	30	30					Lavf60.3.100

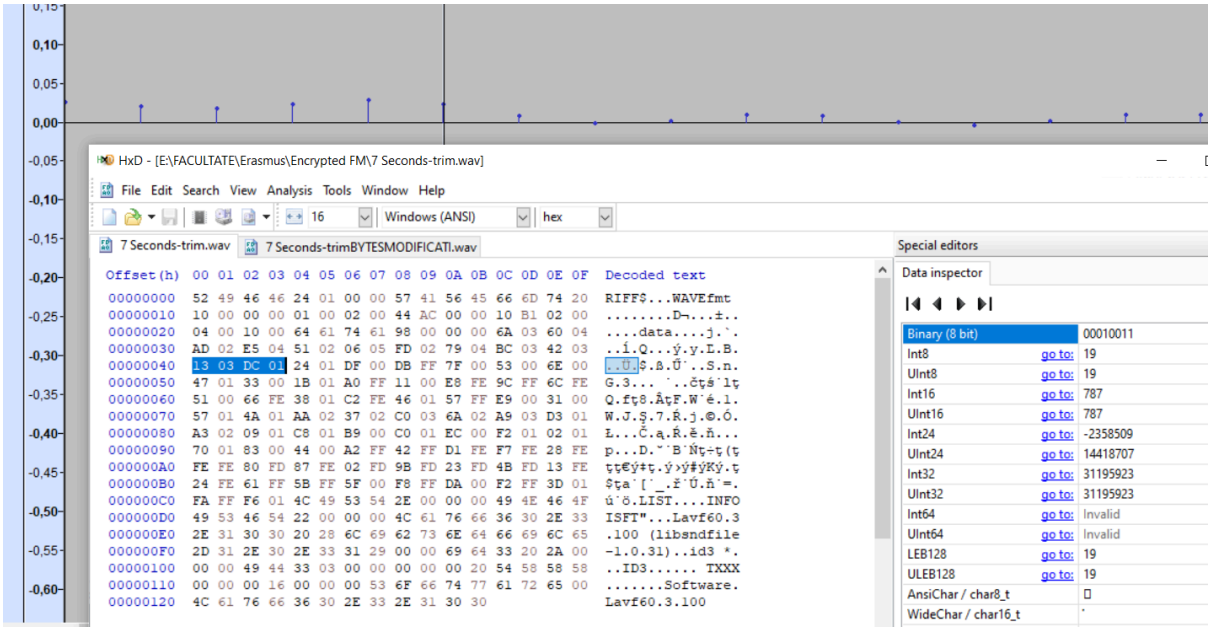
After modification:

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	52	49	46	46	24	01	00	00	57	41	56	45	66	6D	74	20	RIFF$...WAVEfmt
00000010	10	00	00	00	01	00	02	00	44	AC	00	00	10	B1	02	00	........D¬...±..
00000020	04	00	10	00	64	61	74	61	98	00	00	00	6A	03	60	04	....data....j.`.
00000030	AD	02	E5	04	51	02	06	05	FD	02	79	04	BC	03	42	03	..Í.Q...ý.y.L.B.
00000040	FF	FF	FF	FF	24	01	DF	00	DB	FF	7F	00	53	00	6E	00	˙˙˙˙$.ß.Ű˙..S.n.
00000050	47	01	33	00	1B	01	A0	FF	11	00	E8	FE	9C	FF	6C	FE	G.3... ˙..čţś˙lţ
00000060	51	00	66	FE	38	01	C2	FE	46	01	57	FF	E9	00	31	00	Q.fţ8.ÂţF.W˙é.1.
00000070	57	01	4A	01	AA	02	37	02	C0	03	6A	02	A9	03	D3	01	W.J.Ş.7.Ŕ.j.©.Ó.
00000080	A3	02	09	01	C8	01	B9	00	C0	01	EC	00	F2	01	02	01	Ł...Č.ą.Ŕ.ě.ň...
00000090	70	01	83	00	44	00	A2	FF	42	FF	D1	FE	F7	FE	28	FE	p...D.˘˙B˙Ńţ÷ţ(ţ
000000A0	FE	FE	80	FD	87	FE	02	FD	9B	FD	23	FD	4B	FD	13	FE	ţţ€ý‡ţ.ý›ý#ýKý.ţ
000000B0	24	FE	61	FF	5B	FF	5F	00	F8	FF	DA	00	F2	FF	3D	01	$ţa˙[˙_.ř˙Ú.ň˙=.
000000C0	FA	FF	F6	01	4C	49	53	54	2E	00	00	00	49	4E	46	4F	ú˙ö.LIST....INFO
000000D0	49	53	46	54	22	00	00	00	4C	61	76	66	36	30	2E	33	ISFT"...Lavf60.3
000000E0	2E	31	30	30	20	28	6C	69	62	73	6E	64	66	69	6C	65	.100 (libsndfile
000000F0	2D	31	2E	30	2E	33	31	29	00	00	69	64	33	20	2A	00	-1.0.31)..id3 *.
00000100	00	00	49	44	33	03	00	00	00	00	00	20	54	58	58	58	..ID3..... TXXX
00000110	00	00	00	16	00	00	00	53	6F	66	74	77	61	72	65	00	.......Software.
00000120	4C	61	76	66	36	30	2E	33	2E	31	30	30					Lavf60.3.100

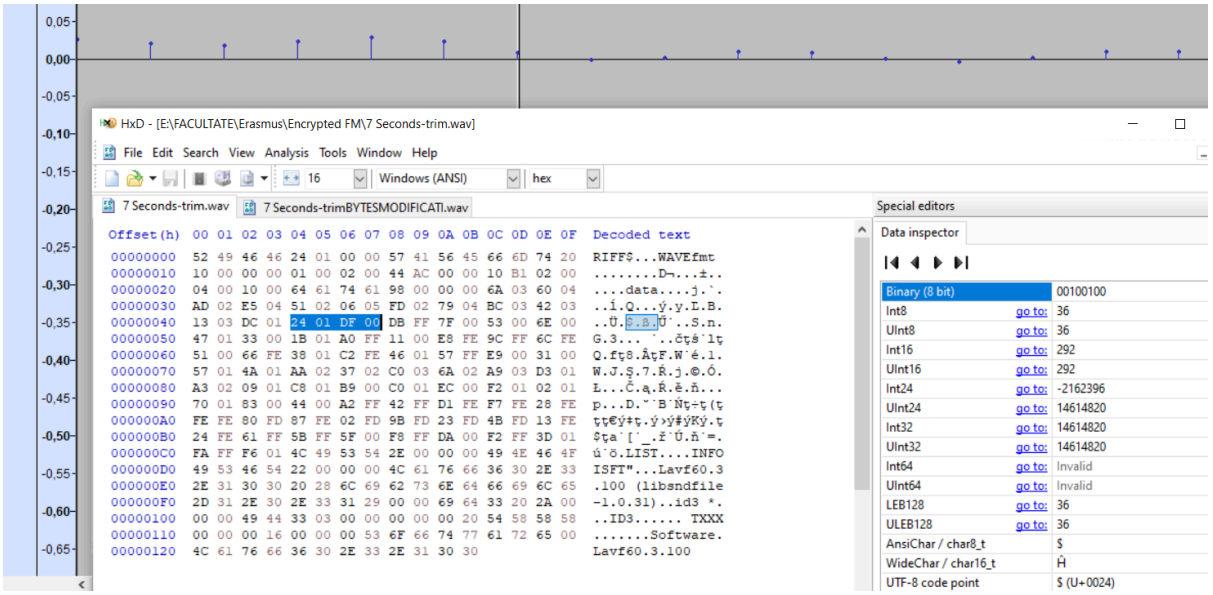
Comparison in Audacity:



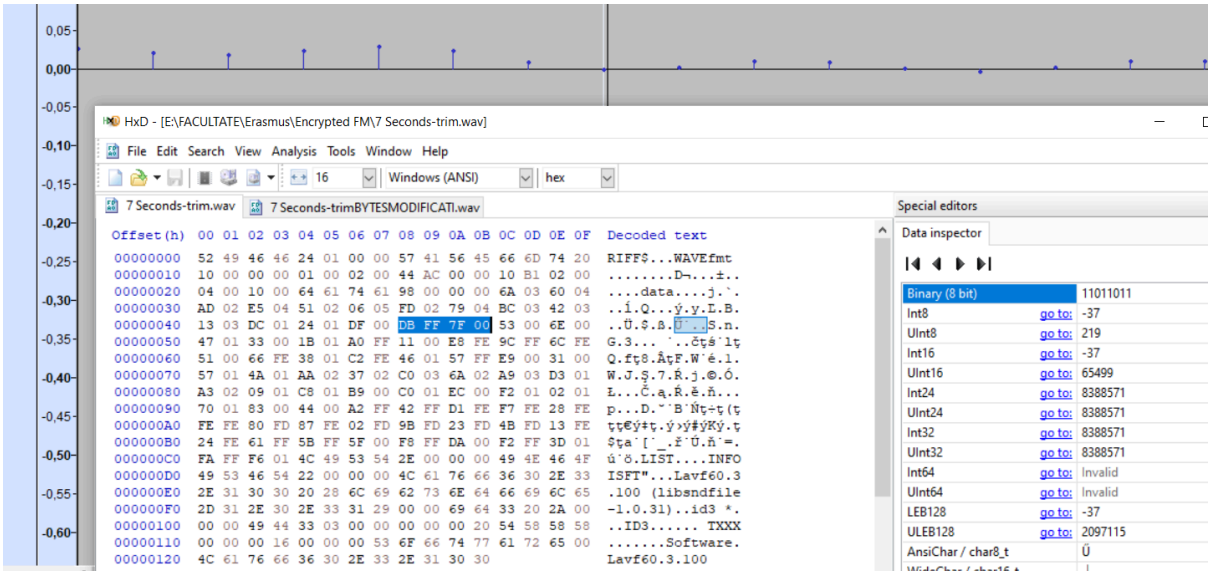
To estimate the sample values, we first took the 4 bytes for the sixth sample and divided the hexadecimal value by 2^{15}. It can be easily observed that, as a result of the calculation, the result matches the value on the axis in Audacity. We did the same with the next two samples to get a clearer picture of the corresponding sample values.



$$\frac{787}{2^{15}} = 0.02401$$



$$\frac{292}{2^{15}} = 0.00891$$



$$\frac{-37}{2^{15}} = -0.00112$$

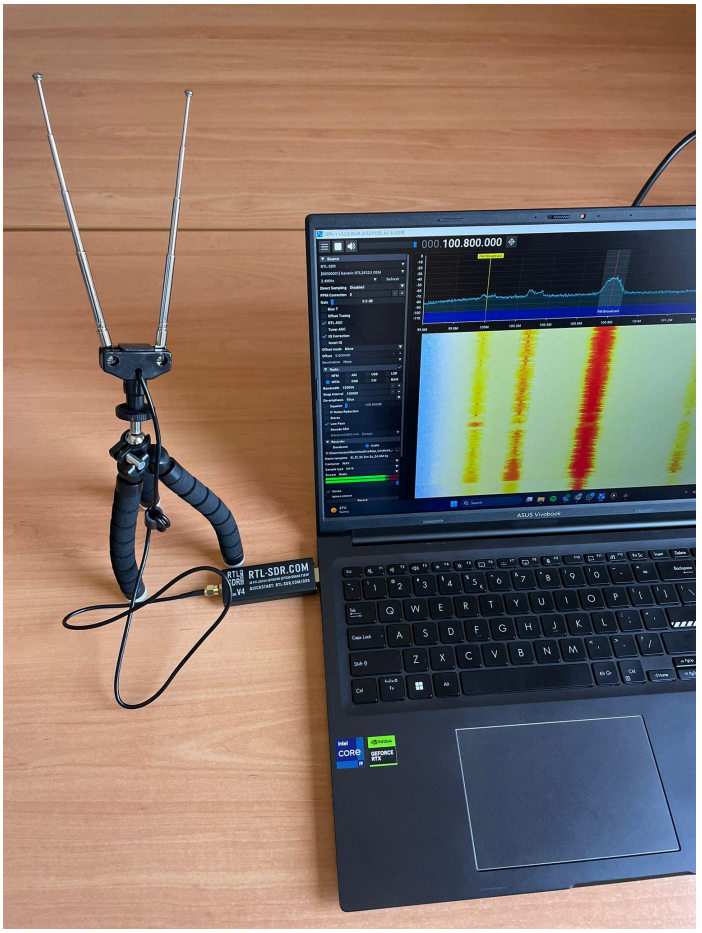
RTL-SDR Device Instructions

The RTL-SDR is an affordable software-defined radio that utilizes DVB-T TV tuners equipped with RTL2832U chips. This device functions as a wideband radio scanner and is appealing to ham radio enthusiasts, hardware hackers, tinkerers, and anyone interested in radio frequency technology. The RTL-SDR V4 model allows direct tuning from 500 KHz to 1.7 GHz without any frequency gaps or the need for offset tuning, which was necessary in the past.

The advantages of this device are its reduced cost and improved usability. However, because the RTL-SDR V4 is a recent release, it faces driver compatibility issues across all platforms.While these problems will diminish over time, users must adjust their system configurations to effectively utilize this device and harness its full capabilities.

Hardware components

The hardware part of this device has been improved since its initial releases, and the version that is about to be presented is the newest one, released in 2024. The RTL-SDR kit includes two antennas of different sizes, antenna supports, a cable to connect the antennas to the device, and the device itself, which connects to a laptop via a USB connection.



Software Instructions

Regarding the software aspect, you need to start by downloading the open-source SDR++ receiver program following the steps from this link: https://arachnoid.com/software_defined_radios_II/. The next step is to download the necessary drivers for the SDR++ program to work and this step can often be challenging. The drivers downloaded from the link https://www.rtl-sdr.com/v4/ need to be placed in the same directory as the executable to ensure that the SDR++ program functions properly. Then you need to make a few settings and modifications within the program to be able to hear different radio waves from the transmitters in the area.

Listening to FM Radio with SDR++

Launch SDR++:

- Open SDR++ and ensure the SDR device is connected and recognized.

Configure the Device:

1. Select the SDR device from the available list.

Select Demodulation Mode:

2. Choose **WFM** (Wideband Frequency Modulation) for standard FM radio stations.

Tune to Desired Frequency:

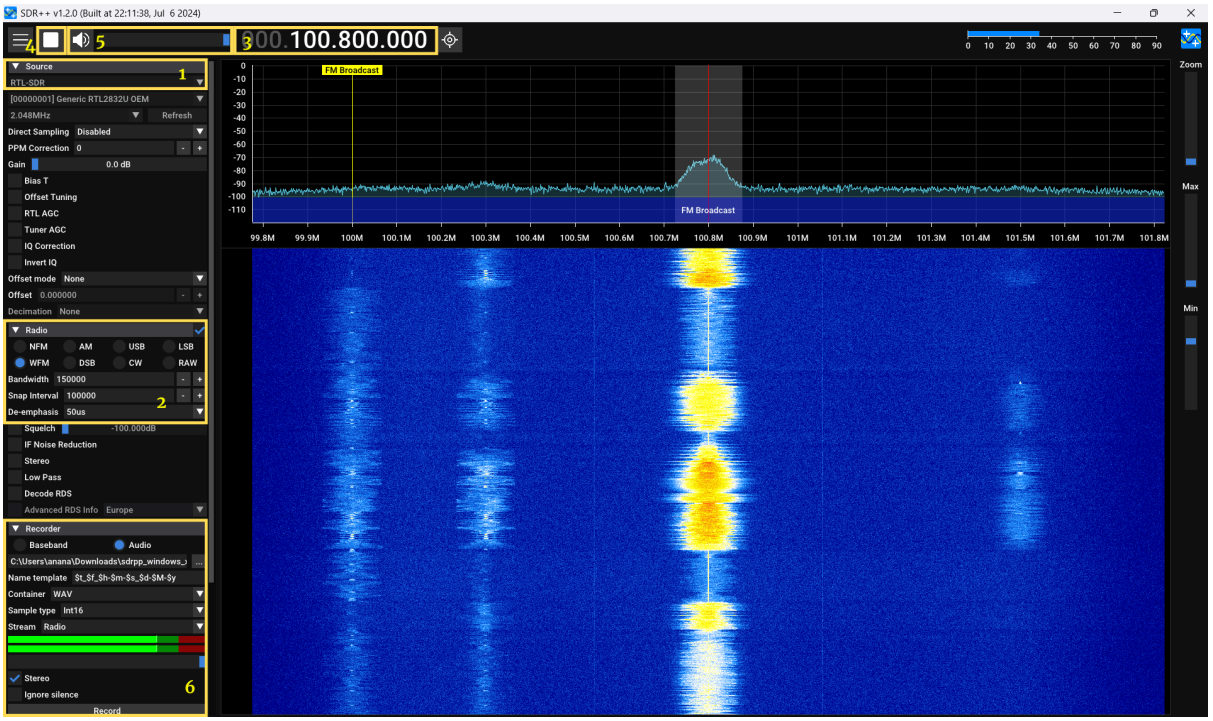
3. Enter the frequency of the FM radio station you want to receive (typically between 88 MHz and 108 MHz).

Activate Audio:

4. Ensure audio output is enabled by clicking the "Start" or "Play" button in SDR++.
5. Check the volume in both SDR++ and your operating system.

Start recording:

6. Click on the folder icon or input field to choose the directory where the recordings will be saved and start recording.



Code Explanation

Python script for histogram of the differences/noise waveform

The following Python script is designed to compare two WAV audio files by analyzing their headers, extracting and normalizing their sample data, and computing the differences between the samples. The script can output either a histogram of the differences or a noise waveform, and it can also generate a new WAV file containing the differences.

1. Import Statements

```
import wave
import binascii
import struct
import numpy as np
import matplotlib.pyplot as plt
import argparse
import os
```

wave: Module for reading and writing WAV files.

binascii: Module for converting between binary data and ASCII.

struct: Module for working with C-style data structures in Python.

numpy (np): Used for numerical operations, specifically for histogram calculations.

matplotlib.pyplot (plt): Used for plotting histograms and waveforms.

argparse: Module for parsing command-line arguments.

os: Module for interacting with the operating system, checking file existence, etc.

2. Functions

get_wav_header_info(wav_file)

```
def get_wav_header_info(wav_file):
    with wave.open(wav_file, 'rb') as wf:
        header_info = {
            "Number of Channels": wf.getnchannels(),
            "Sample Width (bytes)": wf.getsampwidth(),
            "Frame Rate (samples/sec)": wf.getframerate(),
            "Number of Frames": wf.getnframes(),
            "Compression Type": wf.getcomptype(),
            "Compression Name": wf.getcompname()
        }
    return header_info
```

Reads and extracts header information from a WAV file, such as the number of channels, sample width, frame rate, etc.

print_wav_header_info(header_info, wav_file)

```
def print_wav_header_info(header_info, wav_file):
    print(f"File: {wav_file}")
    for key, value in header_info.items():
        print(f"{key}: {value}")
    print()
```

Prints the header information in a readable format.

read_wav_data(wav_file)

```
def read_wav_data(wav_file):
    with wave.open(wav_file, 'rb') as wf:
        raw_data = wf.readframes(wf.getnframes())
        return raw_data
```

Reads the raw audio data (samples) from a WAV file.

get_samples(raw_data, sample_width)

```
def get_samples(raw_data, sample_width):
    num_samples = len(raw_data) // sample_width
    format_char = '<h' if sample_width == 2 else '<b'

    samples = []
    for i in range(num_samples):
        sample = struct.unpack(format_char, raw_data[i*sample_width:(i+1)*sample_width])[0]
        samples.append(sample)

    return samples
```

Converts the raw byte data into individual audio samples using the appropriate format based on the sample width (8-bit or 16-bit).

hex_to_normalized_decimal(raw_data, sample_width)

```
def hex_to_normalized_decimal(raw_data, sample_width):
    num_samples = len(raw_data) // sample_width
    format_char = 'h' if sample_width == 2 else 'b'
    max_value = 2**(8*sample_width - 1)

    normalized_values = []
    for i in range(num_samples):
        sample = struct.unpack('<' + format_char, raw_data[i*sample_width:(i+1)*sample_width])[0]
        normalized_value = sample / max_value
        normalized_values.append(normalized_value)

    return normalized_values
```

Converts the raw byte data into normalized decimal values, scaling them between -1 and 1 based on the sample width.

calculate_differences(normalized_values1, normalized_values2)

```
def calculate_differences(normalized_values1, normalized_values2):
    num_samples = min(len(normalized_values1), len(normalized_values2))
    normalized_differences = []

    for i in range(num_samples):
        norm_value1 = normalized_values1[i]
        norm_value2 = normalized_values2[i]
        normalized_difference = norm_value1 - norm_value2
        normalized_differences.append(normalized_difference)

    return normalized_differences
```

Computes the difference between corresponding samples of two normalized audio data arrays.

write_difference_wav(differences, output_wav_file, sample_rate, sample_width)

```python
def write_difference_wav(differences, output_wav_file, sample_rate, sample_width):
    # Determine max amplitude based on sample width
    if sample_width == 1:
        # 8-bit audio
        max_amplitude = 2**7 - 1  # Range for 8-bit audio is from -127 to 127
    elif sample_width == 2:
        # 16-bit audio
        max_amplitude = 2**15 - 1  # Range for 16-bit audio is from -32768 to 32767
    else:
        raise ValueError("Unsupported sample width. Only 1-byte and 2-byte samples are supported.")

    # Scale factor to convert normalized difference to the appropriate amplitude range
    scale_factor = max_amplitude / max(abs(min(differences)), abs(max(differences)))
    print(scale_factor)


    # Write the WAV file with the differences
    with wave.open(output_wav_file, 'w') as wf:
        wf.setnchannels(2)
        wf.setsampwidth(sample_width)
        wf.setframerate(sample_rate)

        for diff in differences:
            # Scale the difference and clamp to the valid range
            scaled_diff = int(diff * scale_factor)
            scaled_diff = max(-max_amplitude, min(max_amplitude, scaled_diff))  # Clamping

            # Write the frame
            if sample_width == 1:
                # For 8-bit audio, range is from 0 to 255
                wf.writeframes(struct.pack('<B', scaled_diff + 128))  # Add 128 to shift range from [-128, 127] to [0, 255]
            elif sample_width == 2:
                # For 16-bit audio, range is from -32768 to 32767
                wf.writeframes(struct.pack('<h', scaled_diff))
```

Creates a new WAV file from the differences between the two audio files. It scales the differences to fit within the appropriate amplitude range for the given sample width.

plot_histogram(differences) & plot_noise_wave(differences)

```python
def plot_histogram(differences):
    bins = np.arange(-2, 2, 0.1)
    hist, edges = np.histogram(differences, bins=bins)

    plt.hist(differences, bins=bins, edgecolor='black')
    plt.xlabel('Difference Intervals')
    plt.ylabel('Frequency')
    plt.title('Histogram of Differences')
    plt.xticks(bins)
    plt.show()

    return hist, edges

def plot_noise_wave(differences):
    plt.plot(differences)
    plt.xlabel('Sample Index')
    plt.ylabel('Difference')
    plt.title('Noise Waveform')
    plt.show()
```

Generates and displays a histogram of the differences between the samples of the two WAV files. Plots a waveform showing the differences between the samples.

compare_wav_files(file1, file2, output_type)

```
def compare_wav_files(file1, file2, output_type):
    header_info1 = get_wav_header_info(file1)
    header_info2 = get_wav_header_info(file2)

    print("Header Information for File 1:")
    print_wav_header_info(header_info1, file1)

    print("Header Information for File 2:")
    print_wav_header_info(header_info2, file2)

    raw_data1 = read_wav_data(file1)
    raw_data2 = read_wav_data(file2)

    sample_width1 = header_info1["Sample Width (bytes)"]
    sample_width2 = header_info2["Sample Width (bytes)"]

    samples1 = get_samples(raw_data1, sample_width1)
    samples2 = get_samples(raw_data2, sample_width2)

    normalized_values1 = hex_to_normalized_decimal(raw_data1, sample_width1)
    normalized_values2 = hex_to_normalized_decimal(raw_data2, sample_width2)

    differences = calculate_differences(normalized_values1, normalized_values2)

    print(f"First 10 hexadecimal samples for {file1}: {[binascii.hexlify(struct.pack('<' + ('h' if sample_width1 == 2 else 'b'), sample)).decode('utf-8') for sample in samples1][:10]}")
    print(f"First 10 hexadecimal samples for {file2}: {[binascii.hexlify(struct.pack('<' + ('h' if sample_width2 == 2 else 'b'), sample)).decode('utf-8') for sample in samples2][:10]}")

    print(f"First 10 normalized decimal values for {file1}: {normalized_values1[:10]}")
    print(f"First 10 normalized decimal values for {file2}: {normalized_values2[:10]}")
```

```
if output_type == "histogram":
    hist, edges = plot_histogram(differences)

    with open("histogram_data.txt", "w") as f:
        f.write("Difference Interval\tFrequency\n")
        for i in range(len(edges) - 1):
            interval = f"{edges[i]:.2f} to {edges[i+1]:.2f}"
            frequency = hist[i]
            f.write(f"{interval}\t{frequency}\n")

    with open("histogram_data.txt", "r") as f:
        print(f.read())
elif output_type == "difference":
    plot_noise_wave(differences)

    with open("differences_data.txt", "w") as f:
        f.write("Sample Index\tDifference\n")
        for index, difference in enumerate(differences):
            f.write(f"{index}\t{difference:.6f}\n")

    # Call the function to write the differences to a WAV file
    output_wav_file = r"C:\Users\anana\OneDrive\Desktop\compare_wav\differences.wav"
    # Use the sample rate from one of the header infos (assuming both files have the same sample rate)
    sample_rate = header_info1["Frame Rate (samples/sec)"]
    # Use the sample width from one of the header infos (assuming both files have the same sample width)
    sample_width = sample_width1

    print(f"Attempting to write differences to: {output_wav_file}")

    write_difference_wav(differences, output_wav_file, sample_rate, sample_width)
    print(f"Differences written to {output_wav_file}")

     # Check if file was created
    if os.path.exists(output_wav_file):
        print(f"File {output_wav_file} successfully created.")
    else:
        print(f"File {output_wav_file} was not created.")
```

The main function that drives the comparison. It performs the following steps:

1. Extracts and prints header information from both files.
2. Reads the raw audio data.
3. Converts the raw data into samples and normalized decimal values.
4. Computes the differences between the samples.

5. Depending on the output_type, either plots a histogram of the differences or plots the noise waveform.
6. If the output_type is "difference", it writes the differences to a new WAV file and checks if the file was successfully created.

3. Main Execution

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Compare two WAV files and output a histogram or noise wave.")
    parser.add_argument("-audio1", required=True, help="Path to the first audio file")
    parser.add_argument("-audio2", required=True, help="Path to the second audio file")
    parser.add_argument("-output", required=True, choices=["histogram", "difference"], help="Type of output: histogram or difference")

    args = parser.parse_args()
    compare_wav_files(args.audio1, args.audio2, args.output)
```

Command-line Arguments:

-audio1: Path to the first audio file.

-audio2: Path to the second audio file.

-output: Type of output ("histogram" or "difference").

The script runs the compare_wav_files function based on the provided arguments.

4. Key Points

Normalization: The script normalizes the sample values between -1 and 1 to allow for meaningful comparison.

Difference Calculation: The differences between the corresponding samples of the two WAV files are computed to identify any noise or amplitude variation.

WAV File Generation: The script can generate a new WAV file containing only the differences, which can be useful for analyzing noise or other distortions.

Histograms and Waveforms: Visualization is provided through histograms and waveforms, helping to understand the nature of the differences.

This script is highly useful for audio analysis, particularly in detecting and visualizing differences between two audio signals. It's well-structured, allowing for flexible output options and clear reporting of the results.

Python script for a file with the average noise

This Python script is designed to average multiple WAV audio files and produce a single output file that represents the average of the input files. It uses the numpy library for numerical operations and scipy.io.wavfile for reading and writing WAV files.

1. Import Statements

```
import numpy as np
import scipy.io.wavfile as wavfile
import sys
```

numpy (np): Provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.

scipy.io.wavfile (wavfile): Module used to read and write WAV audio files.

sys: Module that provides access to system-specific parameters and functions, particularly for command-line arguments in this script.

2. Functions

average_wav_files(file_list)

```
def average_wav_files(file_list):
    if len(file_list) < 2:
        raise ValueError("You must specify at least two WAV files.")

    sample_rate, data = wavfile.read(file_list[0])
    num_channels = data.shape[1] if len(data.shape) > 1 else 1
    min_samples = len(data)

    total_data = np.zeros((min_samples, num_channels), dtype=np.float64)

    for file in file_list:
        sr, d = wavfile.read(file)
        if sr != sample_rate:
            raise ValueError("All WAV files must have the same sample rate.")
        num_samples = len(d)
        min_samples = min(min_samples, num_samples)
        d = d[:min_samples]

        if len(d.shape) > 1:
            if d.shape[1] != num_channels:
                raise ValueError("All WAV files must have the same number of channels.")
            total_data[:min_samples] += d.astype(np.float64)
        else:
            if num_channels != 1:
                raise ValueError("WAV files must have the same number of channels.")
            d_stereo = np.stack([d, d], axis=1)
            total_data[:min_samples] += d_stereo.astype(np.float64)

    average_data = total_data / len(file_list)

    average_data = np.clip(average_data, -32768, 32767)
    average_data = average_data.astype(np.int16)

    return sample_rate, average_data
```

Purpose: Averages the audio data from multiple WAV files.

Parameters: file_list: A list of paths to the WAV files that need to be averaged.

Functionality:

Input Validation: Checks that at least two WAV files are provided.

Reading the First File: The sample rate and data of the first WAV file are read using wavfile.read. The number of channels is determined (mono or stereo). Initializes an array (total_data) to accumulate the audio data from all files.

Iterating Over the Files: Each file in the list is read, and it is validated that all files have the same sample rate and number of channels. Audio data is added to the total_data array. If the file has fewer samples than previously read files, only the minimum number of samples is used to avoid mismatches. Stereo data is handled by stacking mono data to match the required shape if necessary.

Averaging: The accumulated data is divided by the number of files to calculate the average.

Clipping and Type Conversion: The resulting average data is clipped to fit within the valid range for 16-bit audio (-32768 to 32767). The data is then converted back to 16-bit integers (np.int16), suitable for writing to a WAV file.

Return Value: The function returns the sample rate and the averaged audio data, ready to be saved.

main()

```
def main():
    if len(sys.argv) < 3:
        print("python script.py file1.wav file2.wav [file3.wav ...]")
        sys.exit(1)

    input_files = sys.argv[1:]

    sample_rate, averaged_data = average_wav_files(input_files)

    output_filename = 'output_averaged.wav'
    wavfile.write(output_filename, sample_rate, averaged_data)
    print(f"The resulting WAV file was saved as'{output_filename}'.")
```

Purpose: Manages the overall script execution.

Functionality:

Command-Line Argument Handling: Checks that at least two WAV files are provided as arguments. If not, prints usage instructions and exits.

Averaging Process: Calls average_wav_files with the provided file list. Receives the averaged data and sample rate.

Writing the Output File: Saves the averaged audio data to a new WAV file named output_averaged.wav. Prints a confirmation message indicating where the file was saved.

3. Main Execution

```
56
57  if __name__ == "__main__":
58      main()
59
```

Command-line Arguments:

The script is designed to be run from the command line, where you provide the paths to the WAV files as arguments.

The script calculates the average of the WAV files specified and writes the output to output_averaged.wav.

4. Key Points

Handling Stereo and Mono Data: The script is careful to ensure that stereo data is preserved and properly handled. If any mono files are included, they are converted to stereo by duplicating the channel.

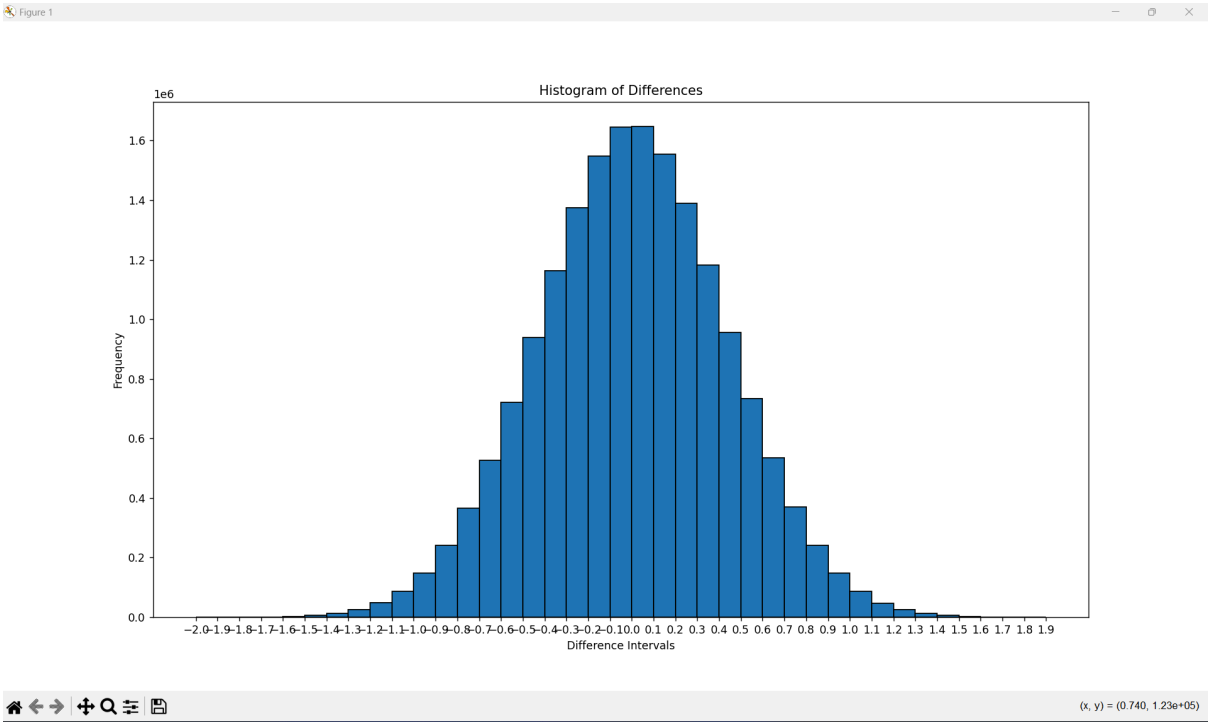
Sample Rate Consistency: It checks that all input files have the same sample rate, as mixing different sample rates would lead to distortion.

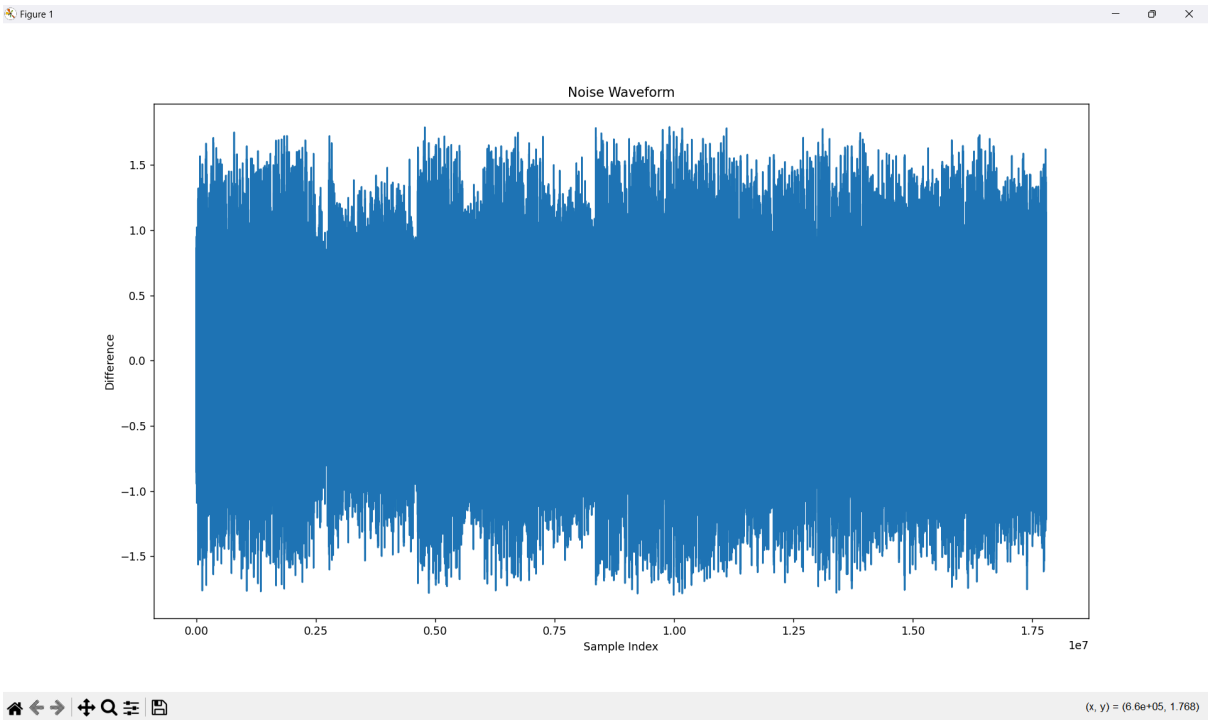
Sample Size Matching: The script works with the smallest number of samples found across all files, ensuring that no mismatches occur due to differing file lengths.

Data Clipping and Conversion: Before saving, the averaged data is clipped to ensure it fits within the valid range for 16-bit audio and then converted to the appropriate format.

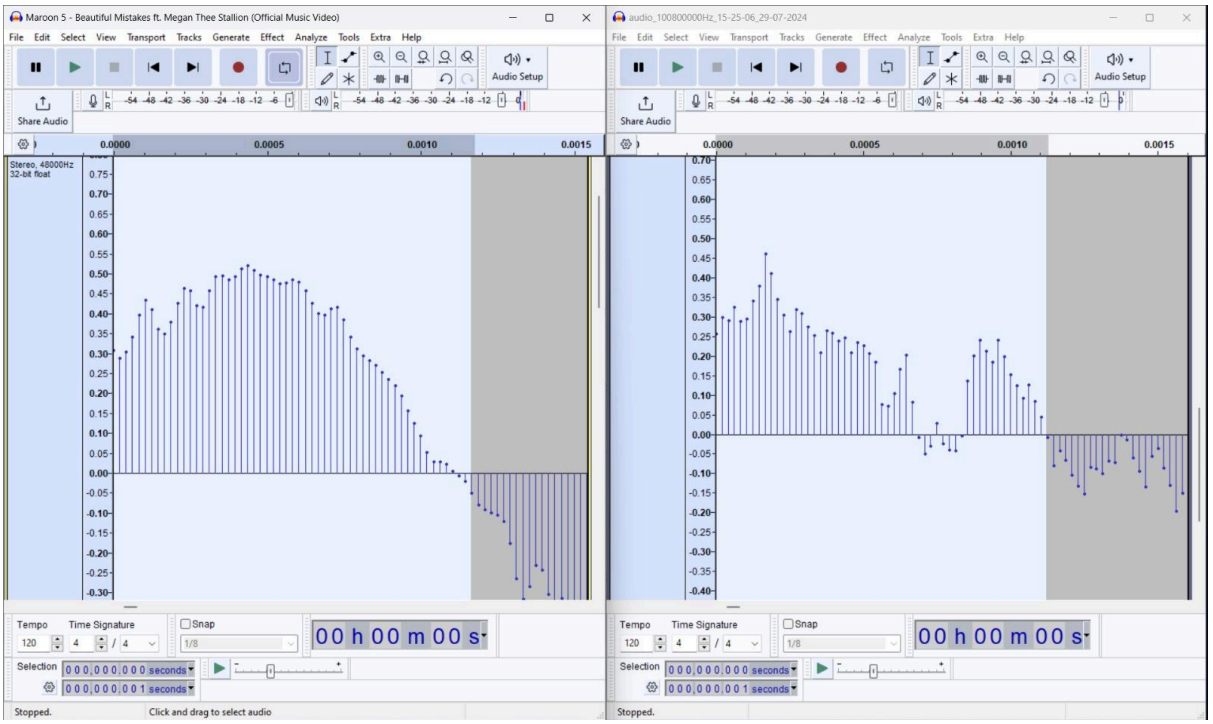
Data analysis of experimental data

1. Radio Station: Radio Sudoeste Lisbon - 100,8 FM (Radio MEO Music)
Location:Rua Viriato nº 25 6º - 105-234 LISBOA
Song: Beautiful Mistakes
VS Code Output:





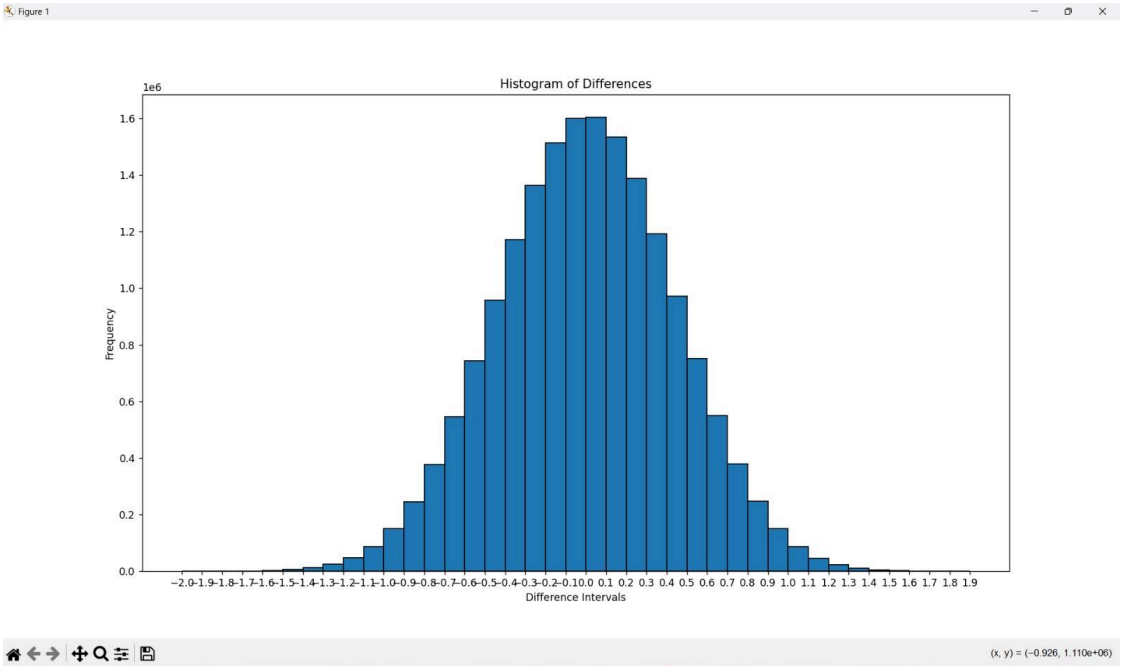
Alignment of samples between the downloaded (MP3 to WAV) and recorded versions of the song:

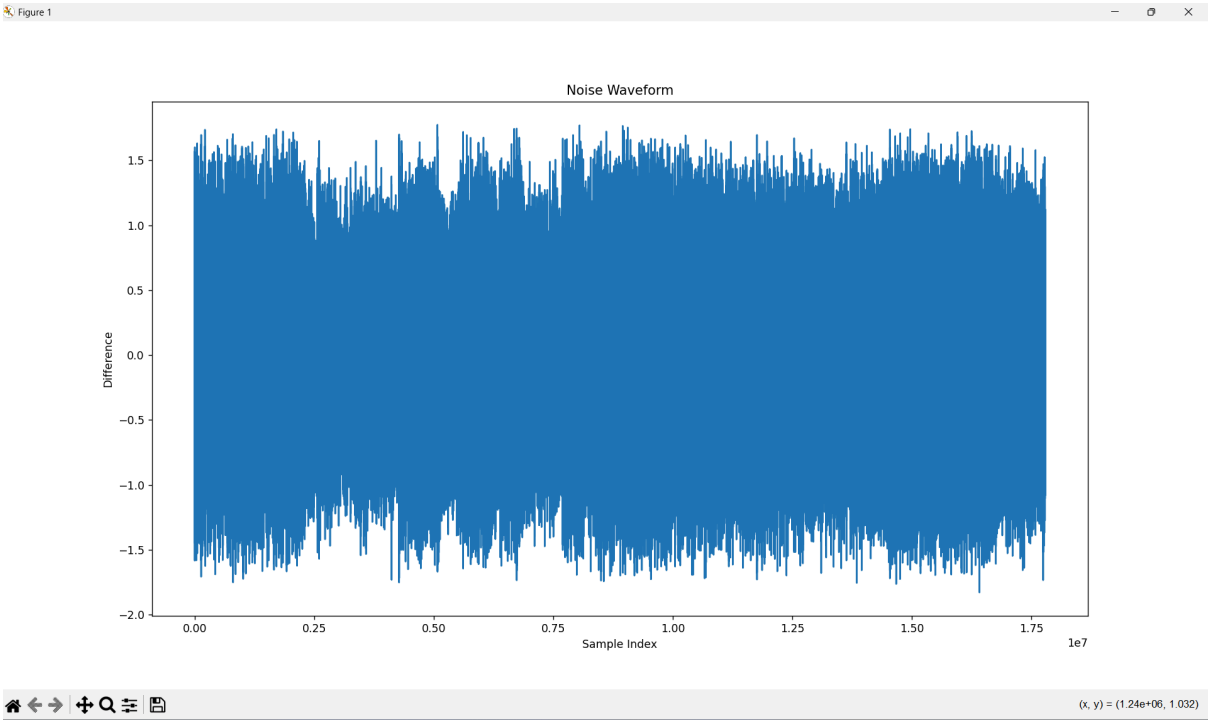


2. Downloaded version & Converted Flac to WAV version

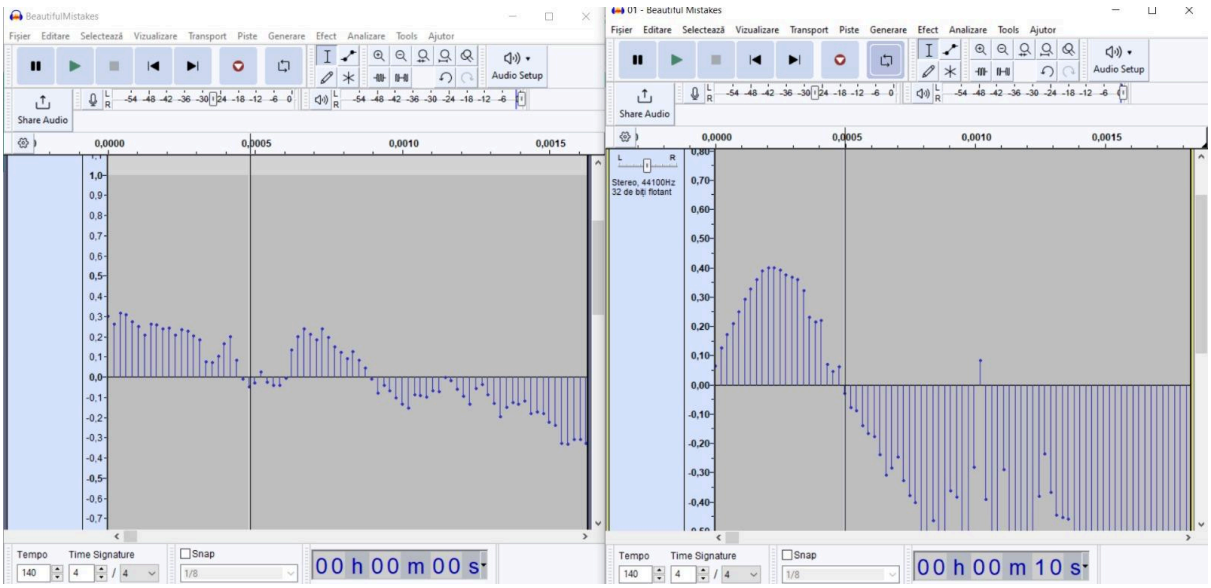
Song: Beautiful Mistakes

VS Code Output:

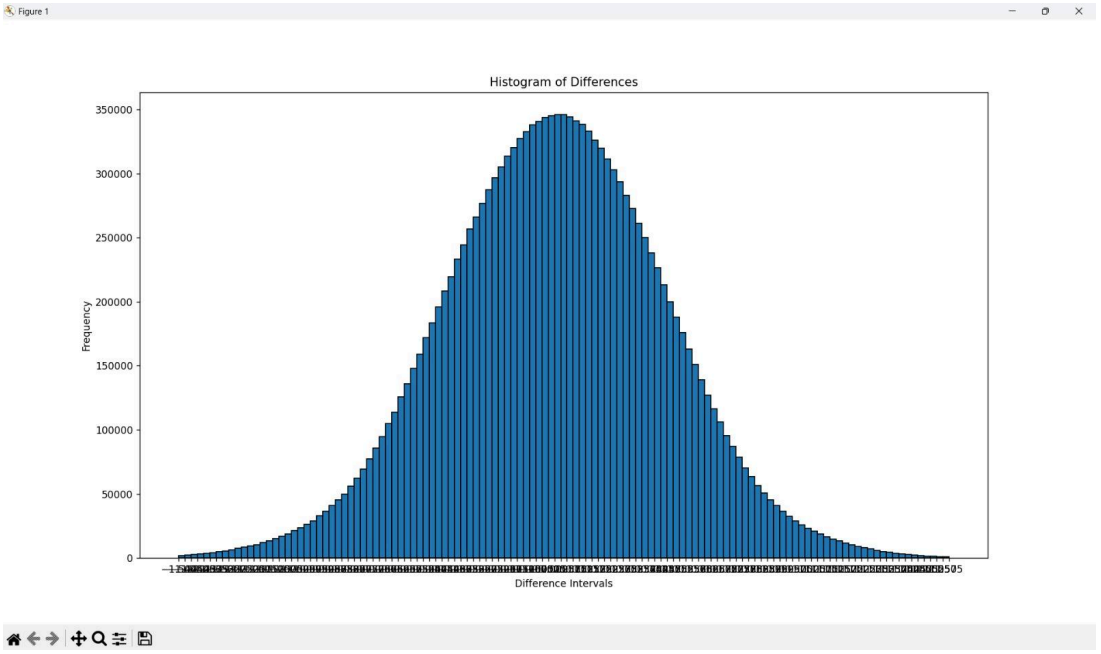




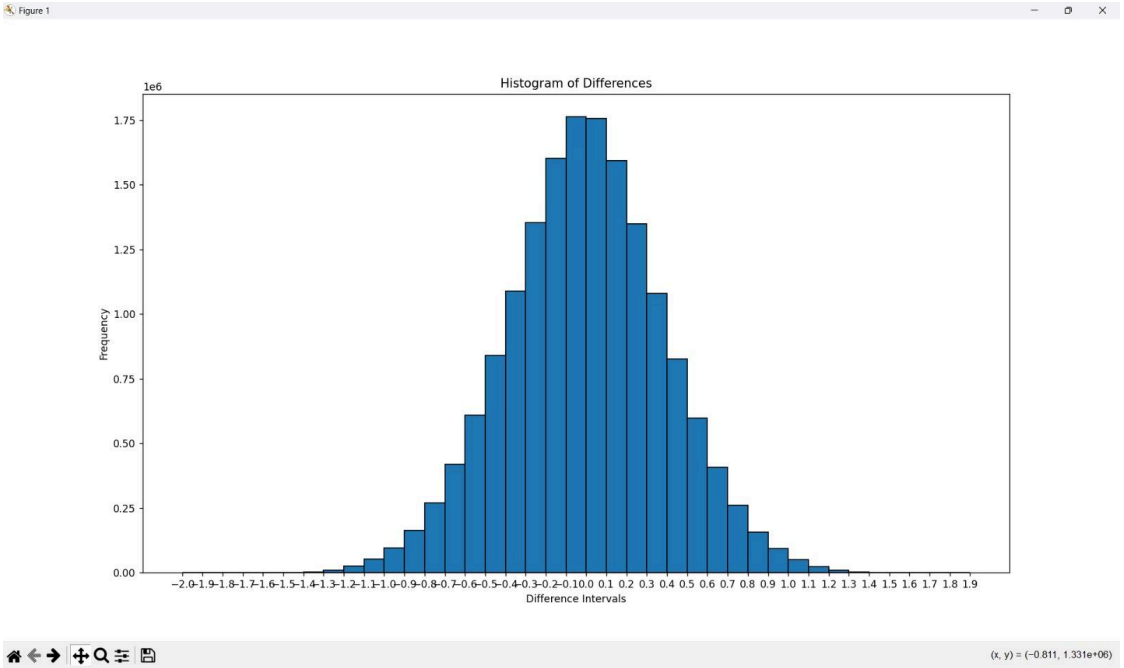
Alignment of samples between the downloaded(Flac to WAV) and recorded versions of the song:

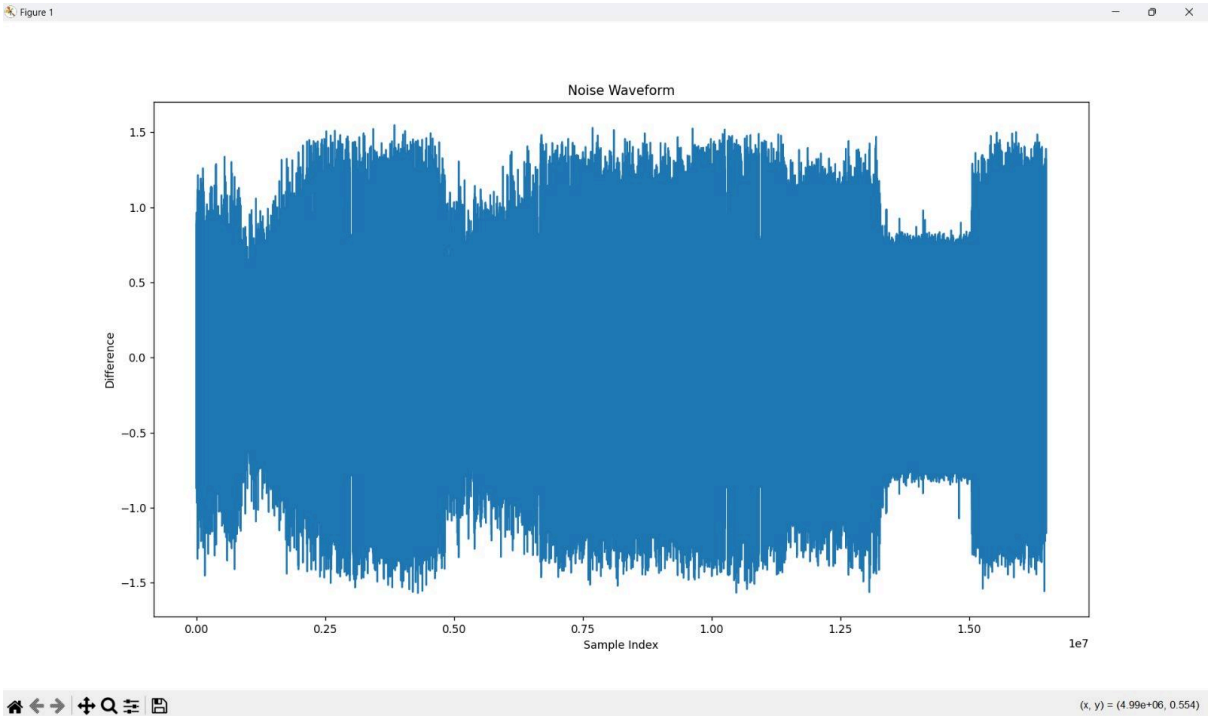


3.Two Radio Stations: Radio Sudoeste Lisbon - 100,8 FM and RFM - 93,2 FM
Locations: Rua Viriato nº 25 6º - 105-234 LISBOA and Quinta do Bom Pastor, Estrada da Buraca 8 - 12 Lisboa
Songs (two different songs): (Beautiful Mistakes-recorded and A bar song - downloaded)
VS Code Output:

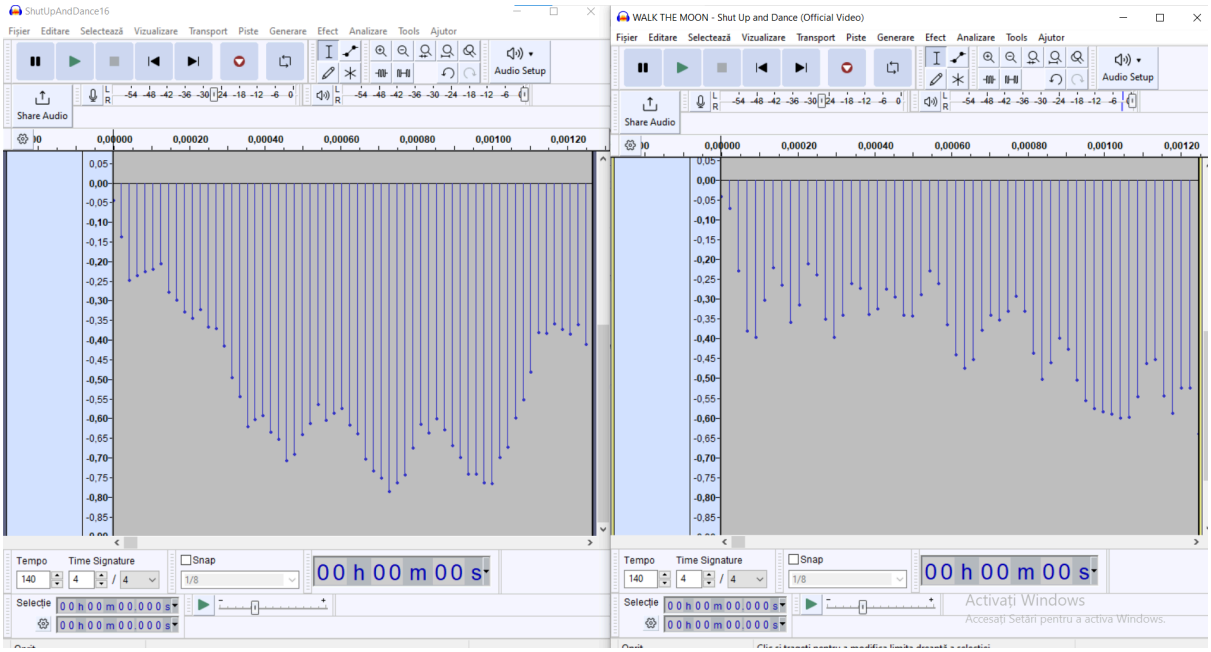


4. Radio Station: RFM - 93,2 FM
Location: Monsanto Forest Park - Monsanto Communications Tower
Song: Shut up and dance-WALK THE MOON
VS Code Output:

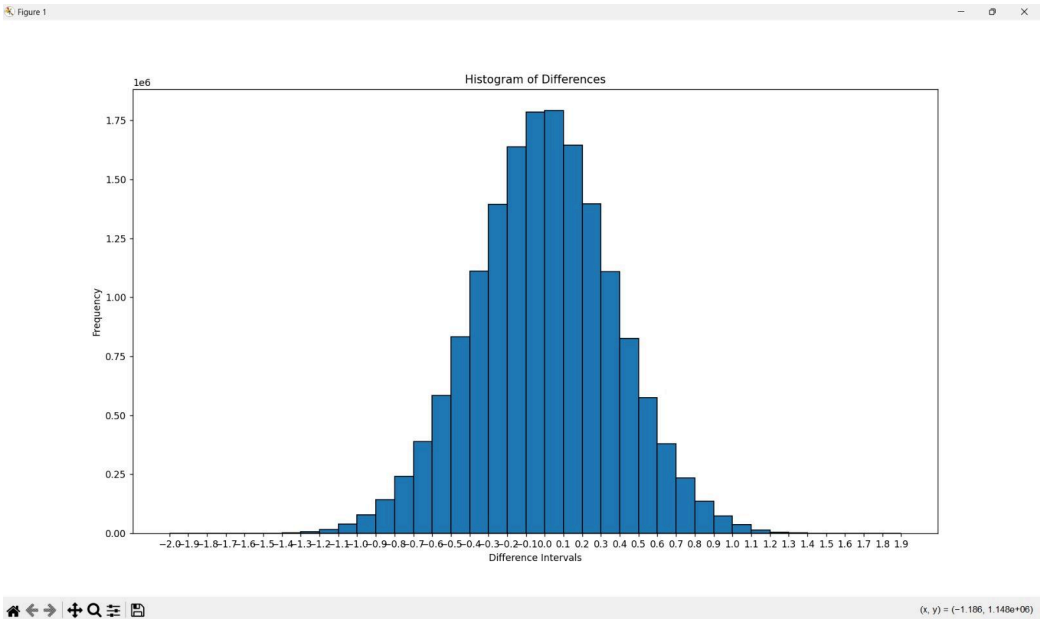




Alignment of samples between the downloaded(directly to WAV) and recorded versions of the song:

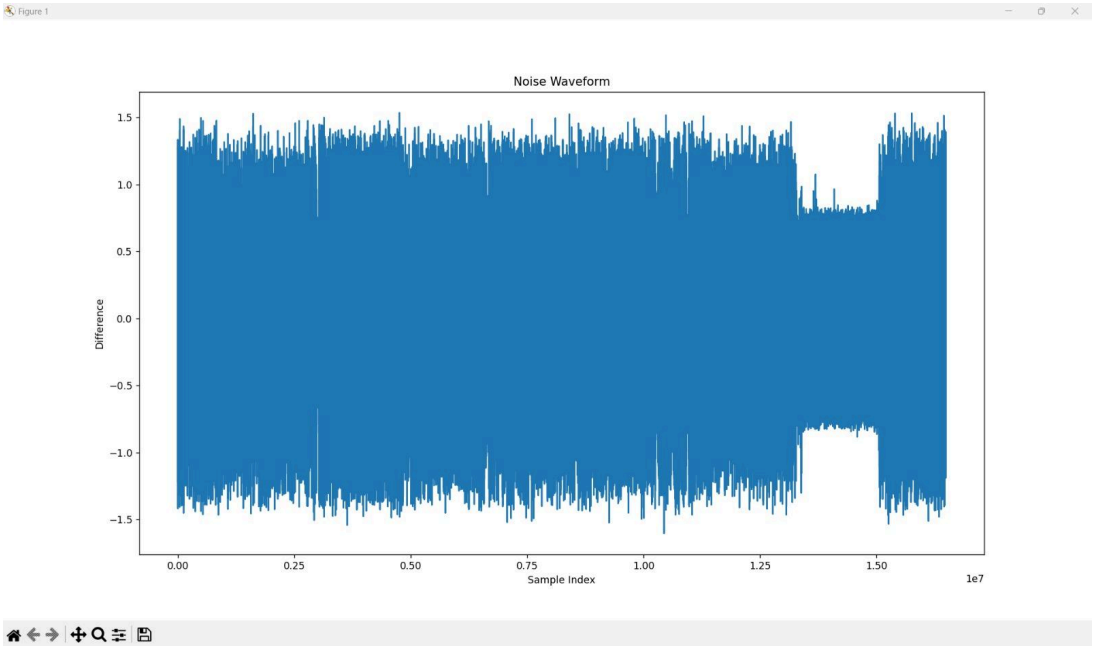


5. Radio Station: RFM - 93,2 FM
Location:Monsanto Communications Tower
Song: Shut up and dance-WALK THE MOON
VS Code Output:



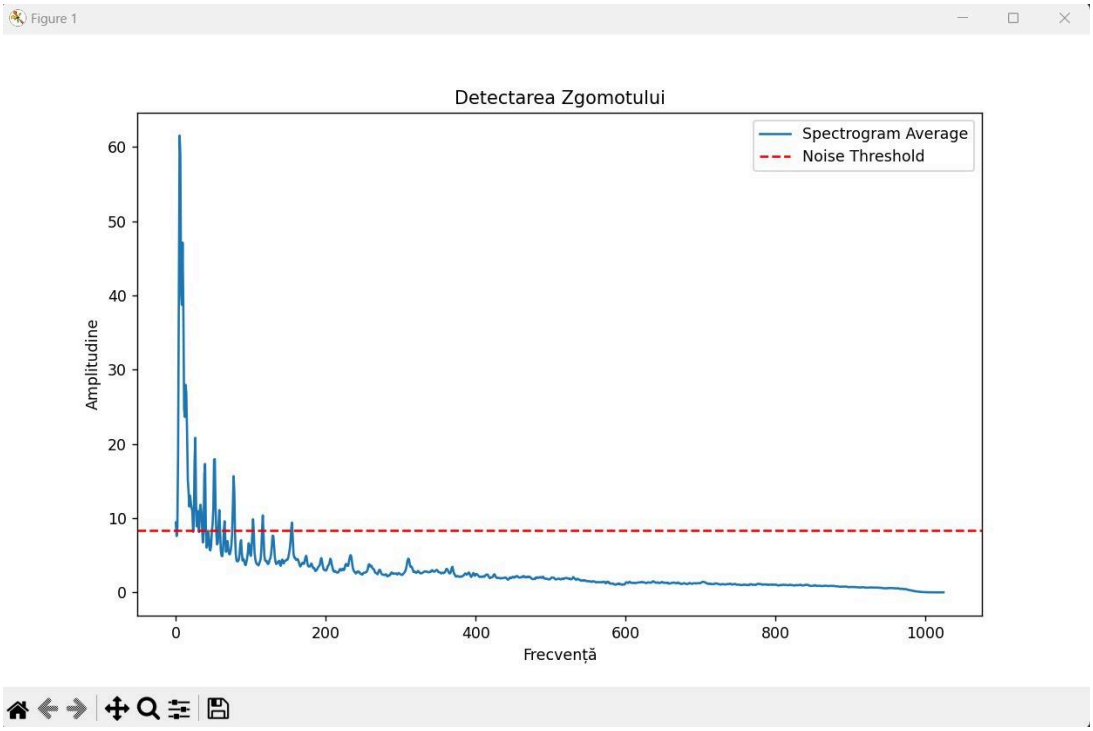
Noise Waveform

It shows the amplitude difference between two audio signals over time. The consistent waveform indicates a constant noise level, and the dip might show a period of reduced noise. This plot gives a visual indication of the noise's consistency over the entire duration of the audio sample, showing how much one signal deviates from another.

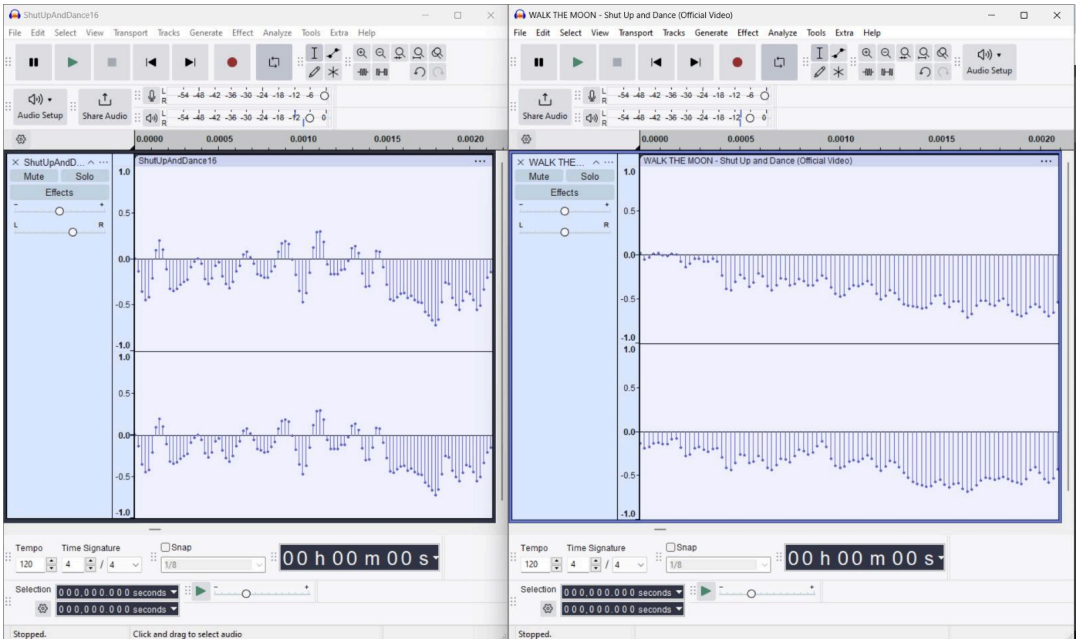


Spectrogram Noise Detection

It shows the average amplitude of different frequencies in the audio signal. The red dashed line represents a noise threshold, helping to identify which parts of the signal exceed this threshold and might be considered noise. This plot is useful for detecting noise across different frequency bands. Frequencies with amplitudes above the threshold might be considered as containing noise.

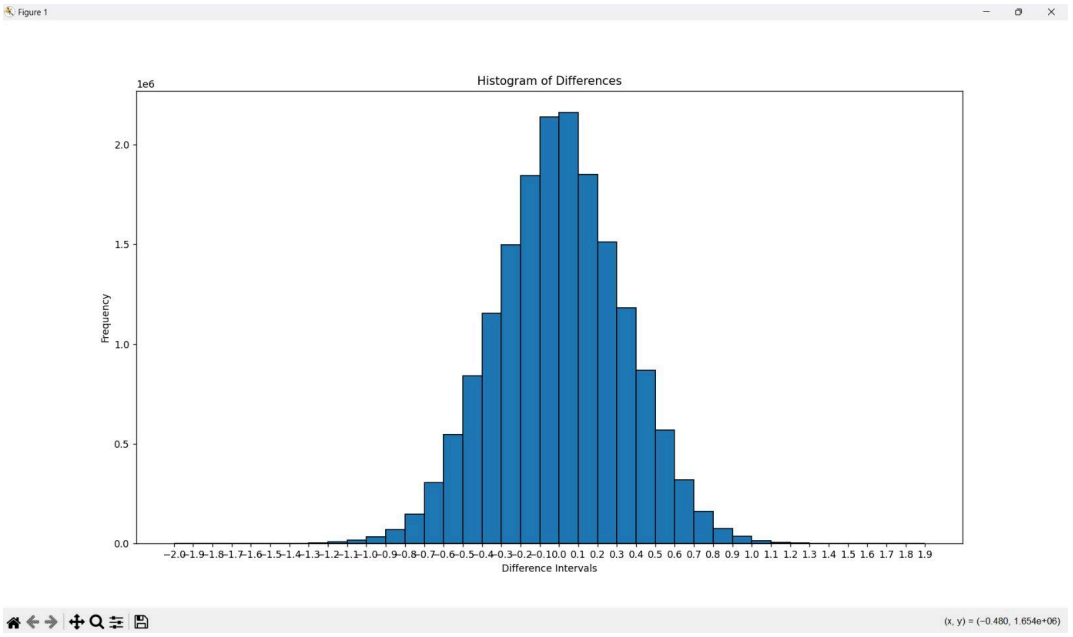


Alignment of samples between the downloaded(Flac to WAV) and recorded versions of the song:



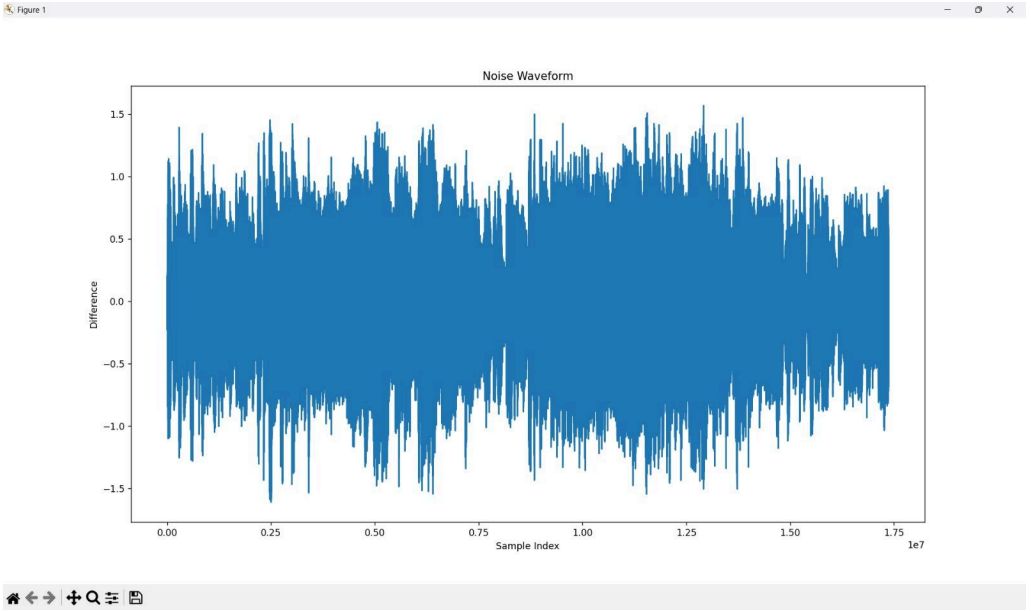
6. Radio Station: RFM - 93,2 FM
Location:Monsanto Communications Tower
Song: What Was I Made For-Billie Eilish

VS Code Output:



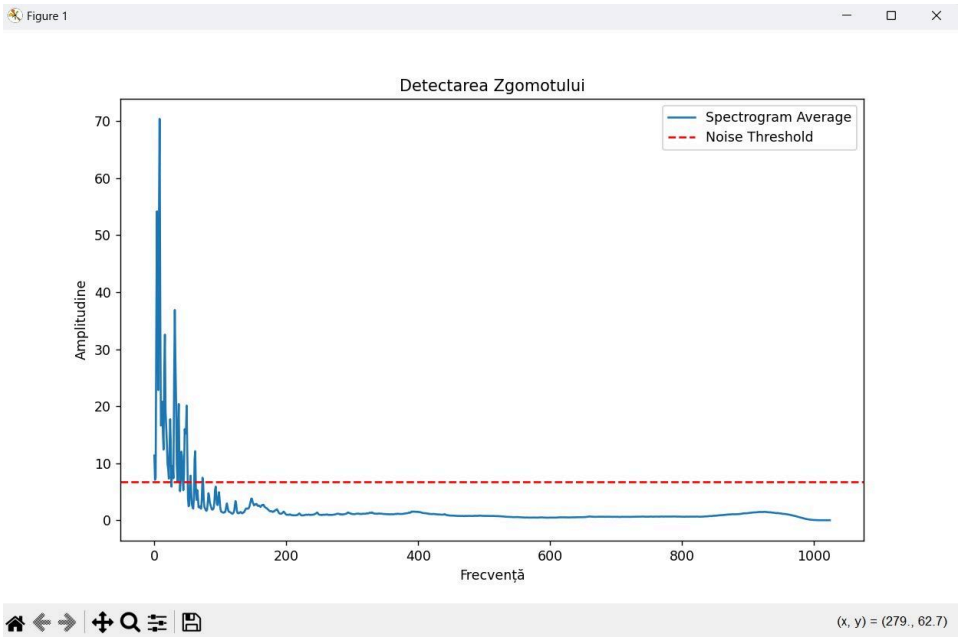
Noise Waveform

It shows the amplitude difference between two audio signals over time. The consistent waveform indicates a constant noise level, and the dip might show a period of reduced noise. This plot gives a visual indication of the noise's consistency over the entire duration of the audio sample, showing how much one signal deviates from another.

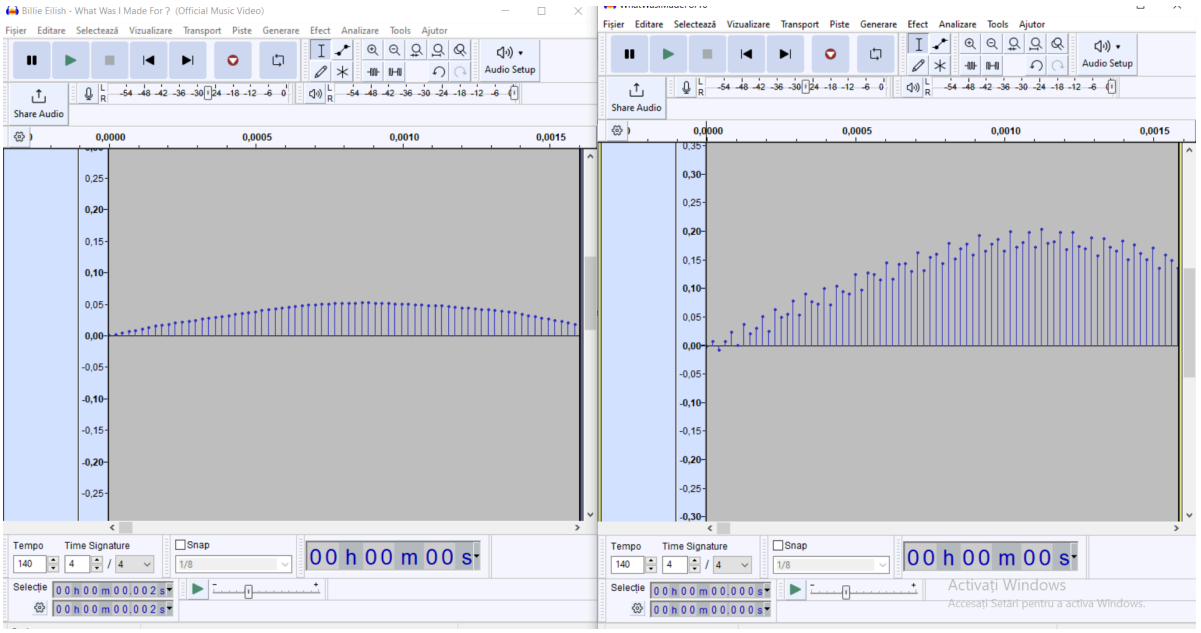


Spectrogram Noise Detection

It shows the average amplitude of different frequencies in the audio signal. The red dashed line represents a noise threshold, helping to identify which parts of the signal exceed this threshold and might be considered noise. This plot is useful for detecting noise across different frequency bands. Frequencies with amplitudes above the threshold might be considered as containing noise.



Alignment of samples between the downloaded(directly to WAV) and recorded versions of the song:



After analyzing the experimental data and comparing the MP3/Flac and Wav audio files, we concluded that the radio stations use the MP3 format to transmit the audio files.

The RFM radio station in Lisbon uses the MP3 format for its digital streams, with a typical bitrate of 128 kbps. This format is widely used for online radio broadcasts due to its balance between sound quality and data efficiency.

For FM broadcasts, the format used is standard analog FM transmission, which is common for terrestrial radio stations.

Radio distance analysis

Monsanto Forest Park is one of the main locations for FM transmissions in Lisbon. Various radio stations broadcast from this site due to its elevated position, which provides good coverage over the city.

Monsanto Forest Park in Lisbon hosts several radio frequencies and some of the known radio frequencies transmitted from the Monsanto Communications Tower:

1. **RFM** on 93.2 MHz
2. **Rádio Renascença** on 98.7 MHz
3. **Antena 1** on 99.1 MHz
4. **TSF** on 89.5 MHz

The Monsanto Communication Tower is also known as the Monsanto South Telecommunication Tower (Altice Cell Tower). Here are some key details about it:

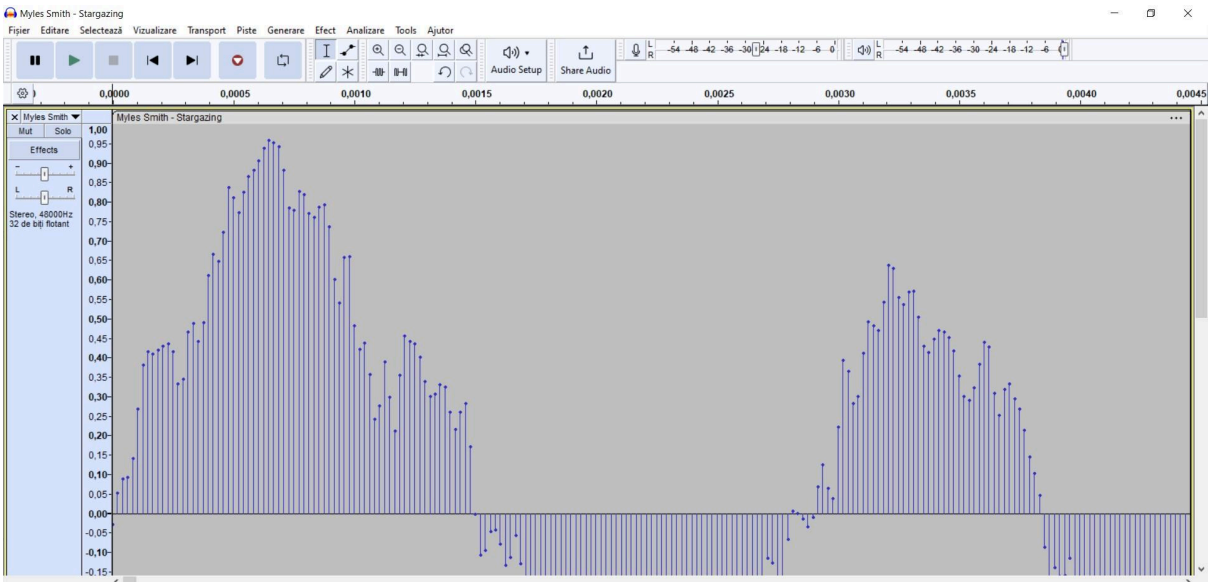
- The tower stands at 117 meters tall and is primarily used for telecommunication purposes by Portugal Telecom
- It is situated in the Monsanto Forest Park area, a significant green space in Lisbon
- The tower is designed for communication purposes, including broadcasting radio and television signals across the region
- The tower is a built structure, and it has been serving its communication purposes effectively since its completion

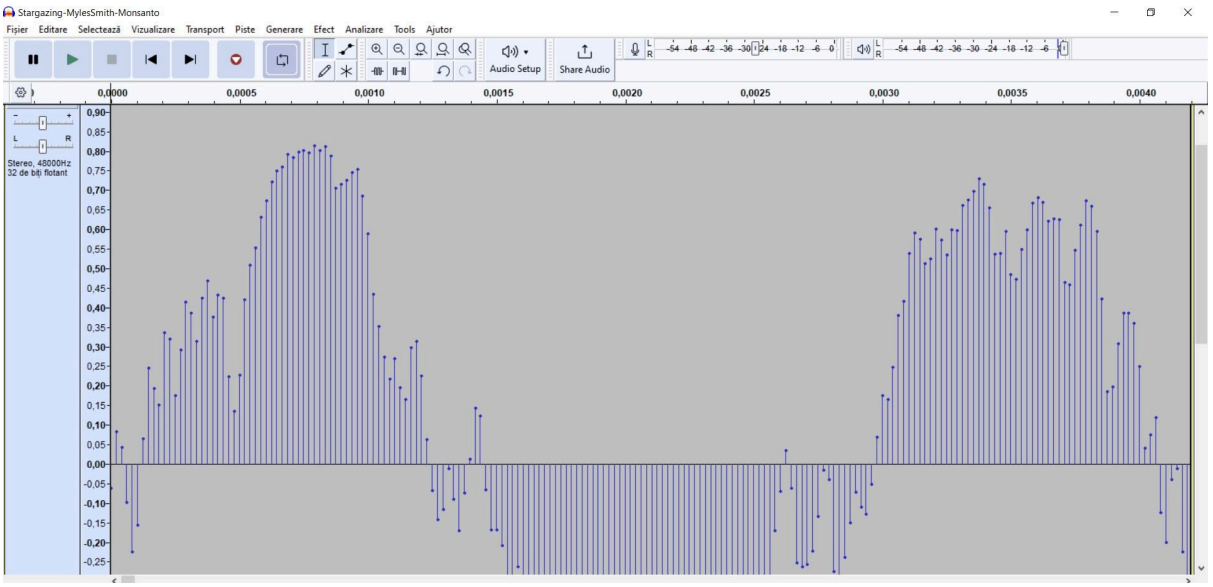
The next recording examples will be based on recordings from **Radio RFM** on the **93.2 MHz** frequency.

First Location: Monsanto Forest Park, near Altice Cell Tower (70 meters away)

1.Song: Stargazing- Myles Smith

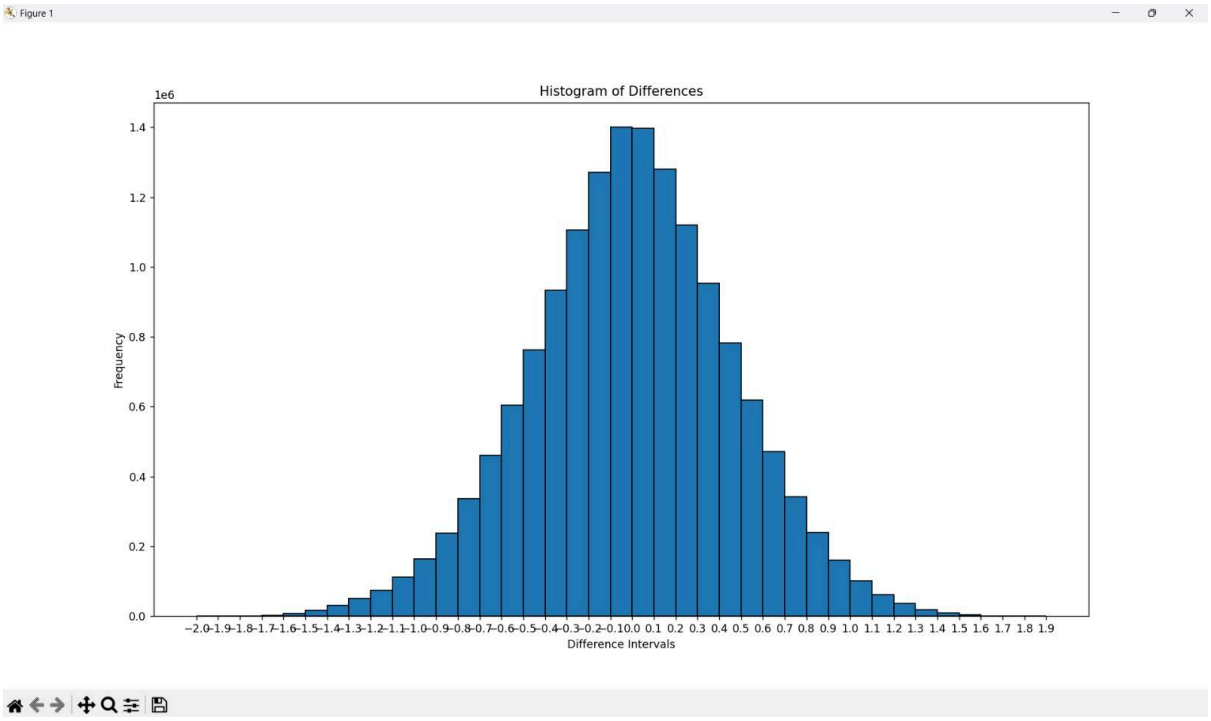
Alignment of samples between the downloaded (MP3 to WAV) and recorded versions of the song:

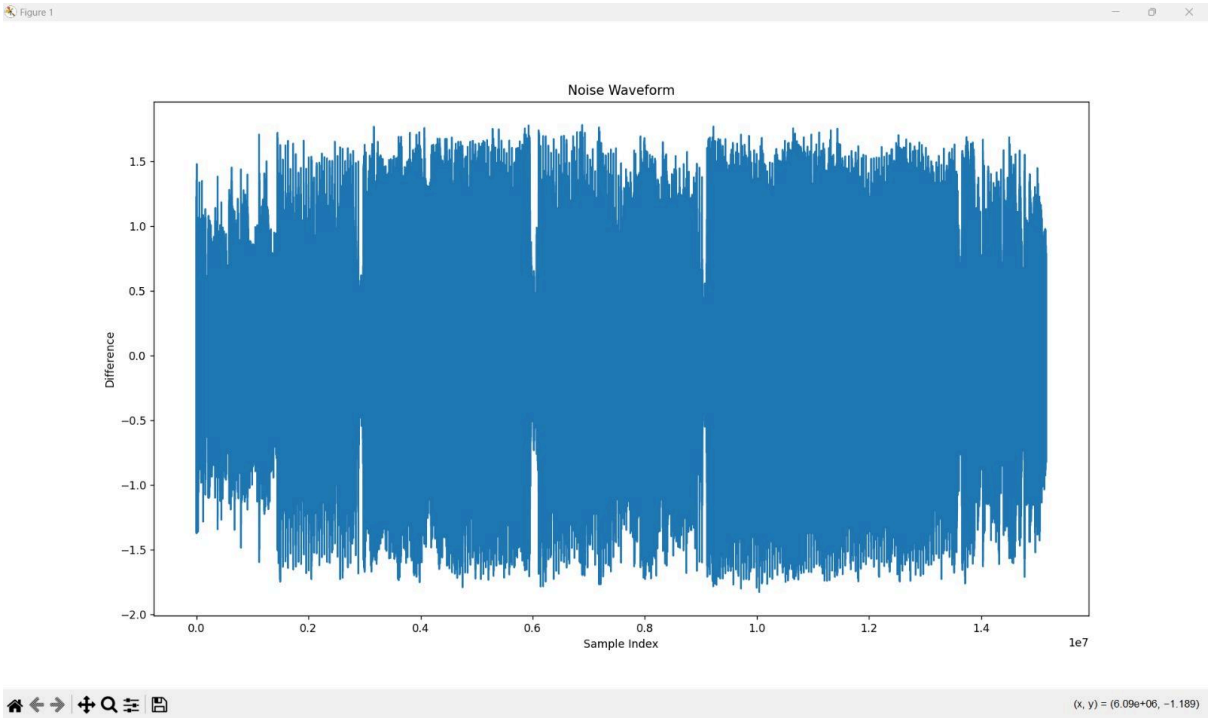




Scaling Factor:17917.547868168545

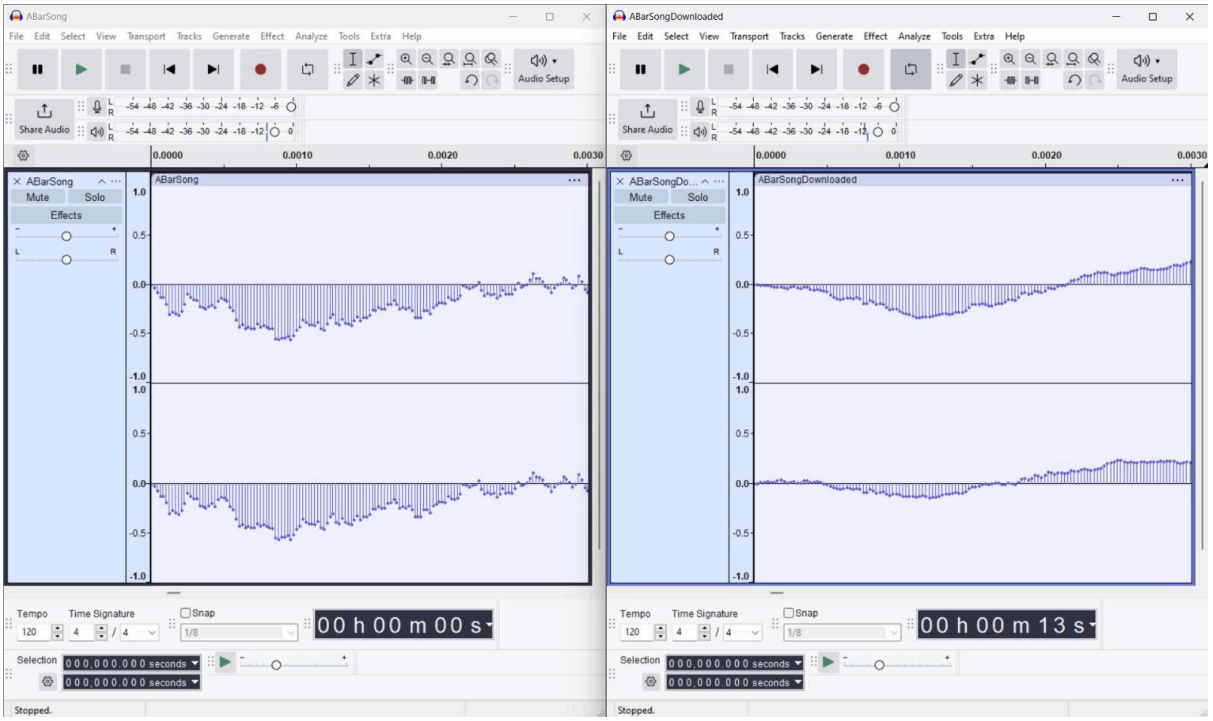
VS Code Output:





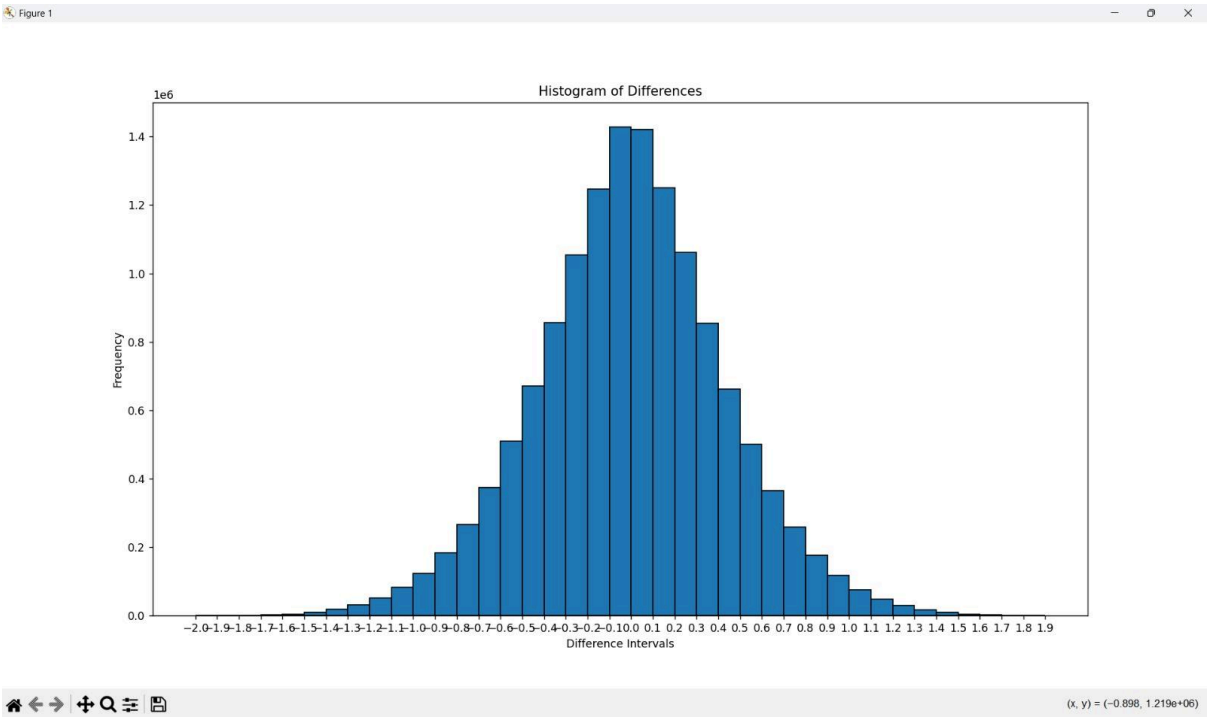
2.Song: A bar song - Shaboozey

Alignment of samples between the downloaded (MP3 to WAV) and recorded versions of the song:



Scaling Factor: 17846.667486661237

VS Code Output:



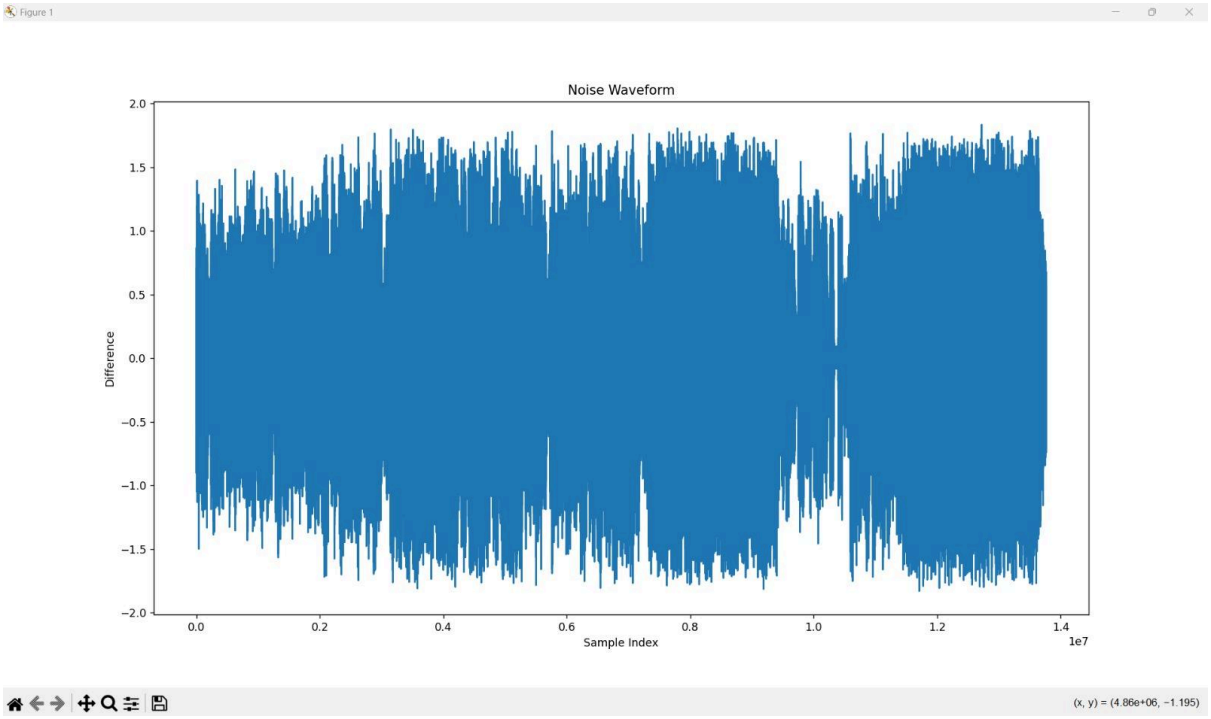
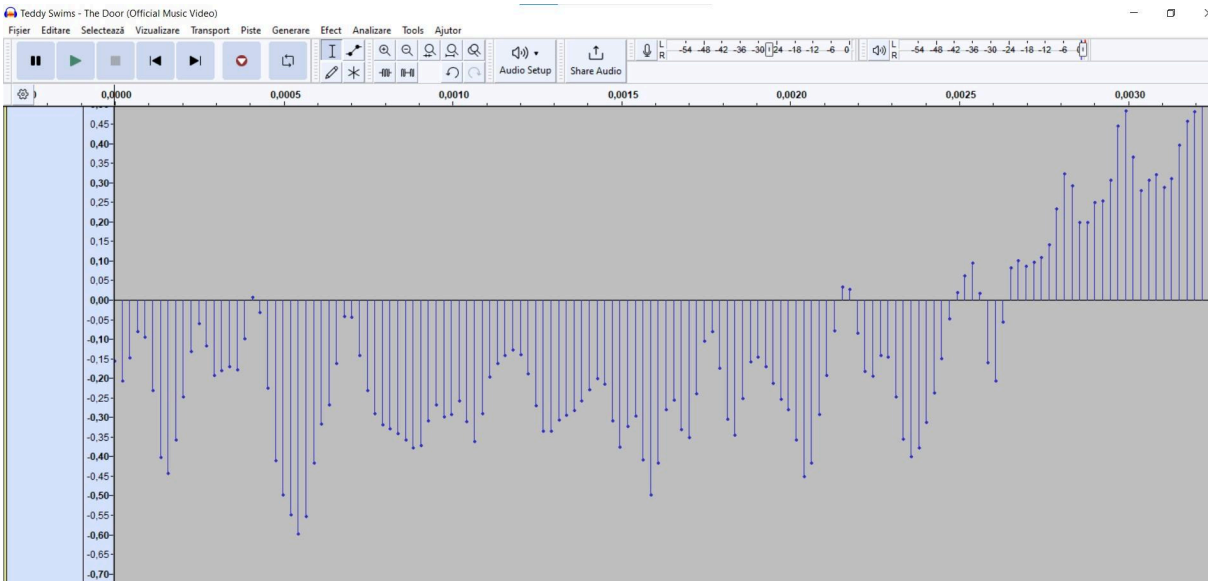
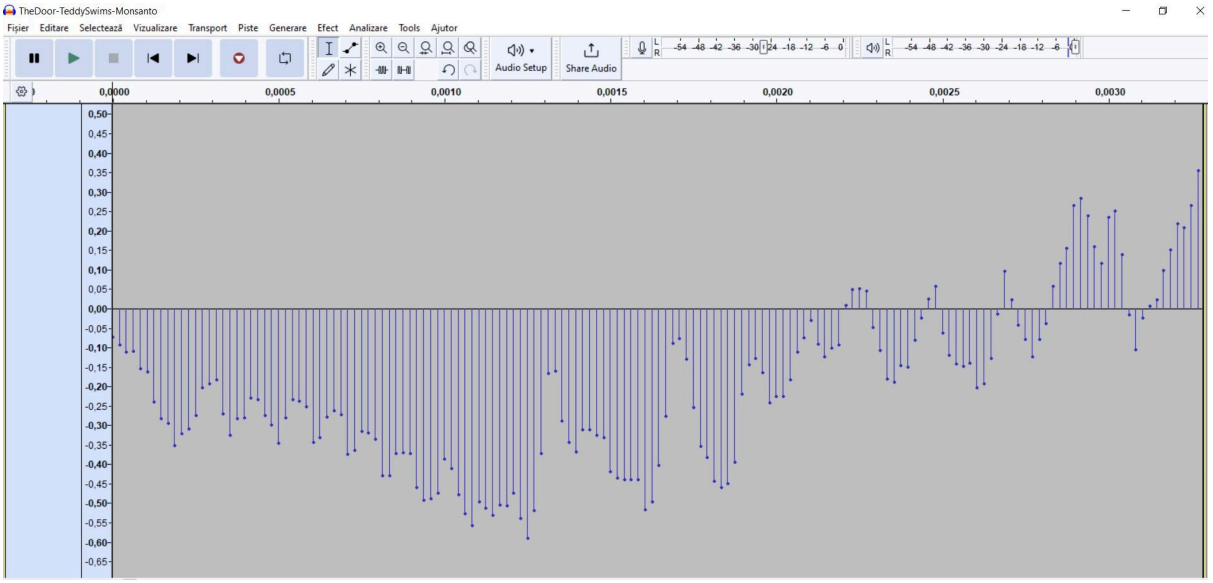


Figure 1
Noise Waveform
Difference
Sample Index
1e7
(x, y) = (4.86e+06, −1.195)

3.Song: The Door - Teddy Swims

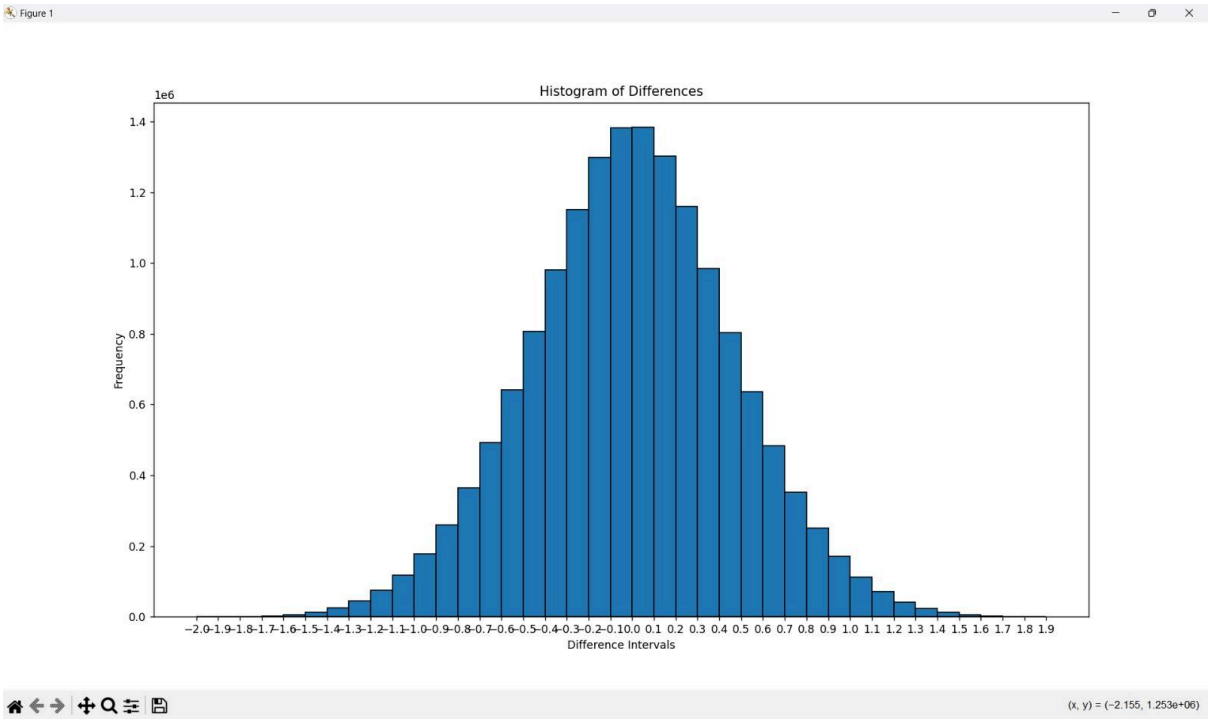
Alignment of samples between the downloaded (MP3 to WAV) and recorded versions of the song:

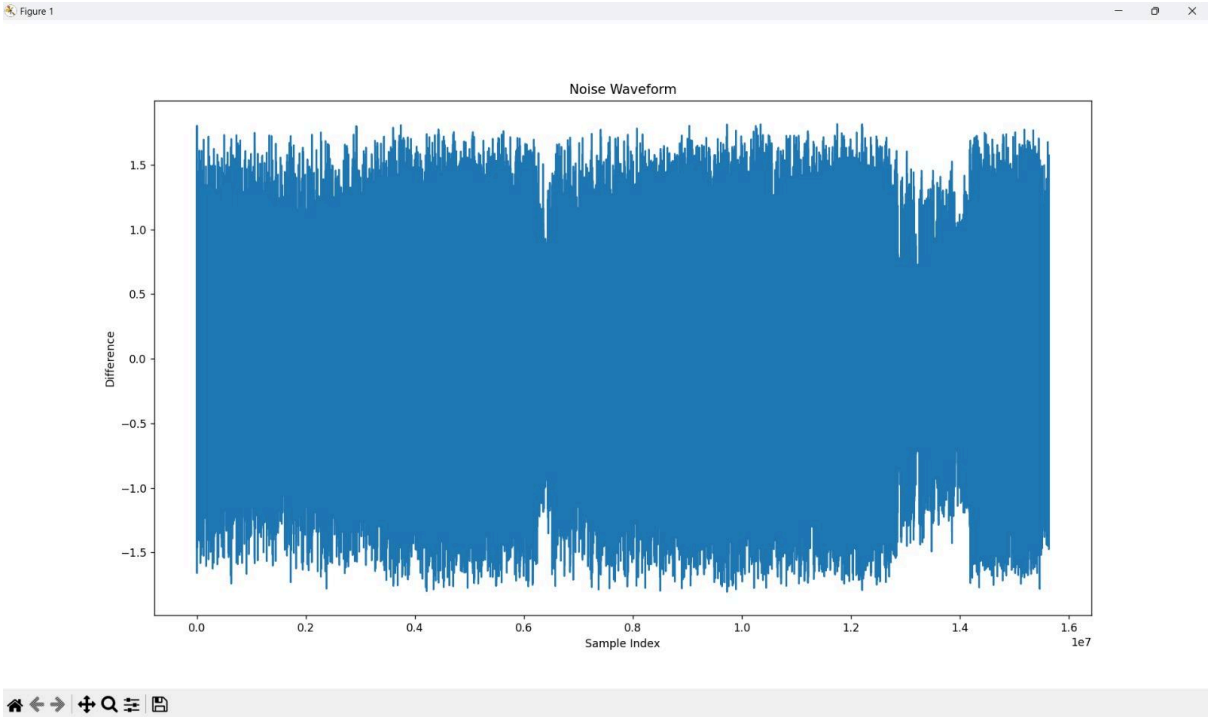




Scaling Factor: 18030.37877413938

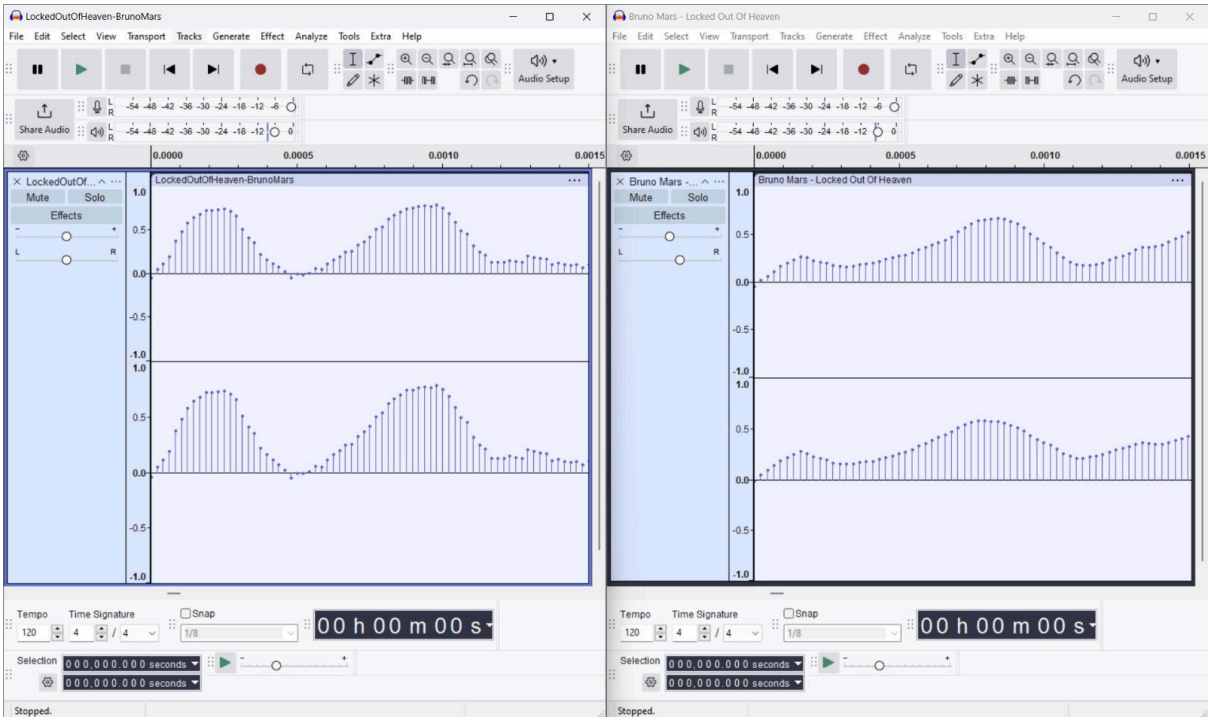
VS Code Output:





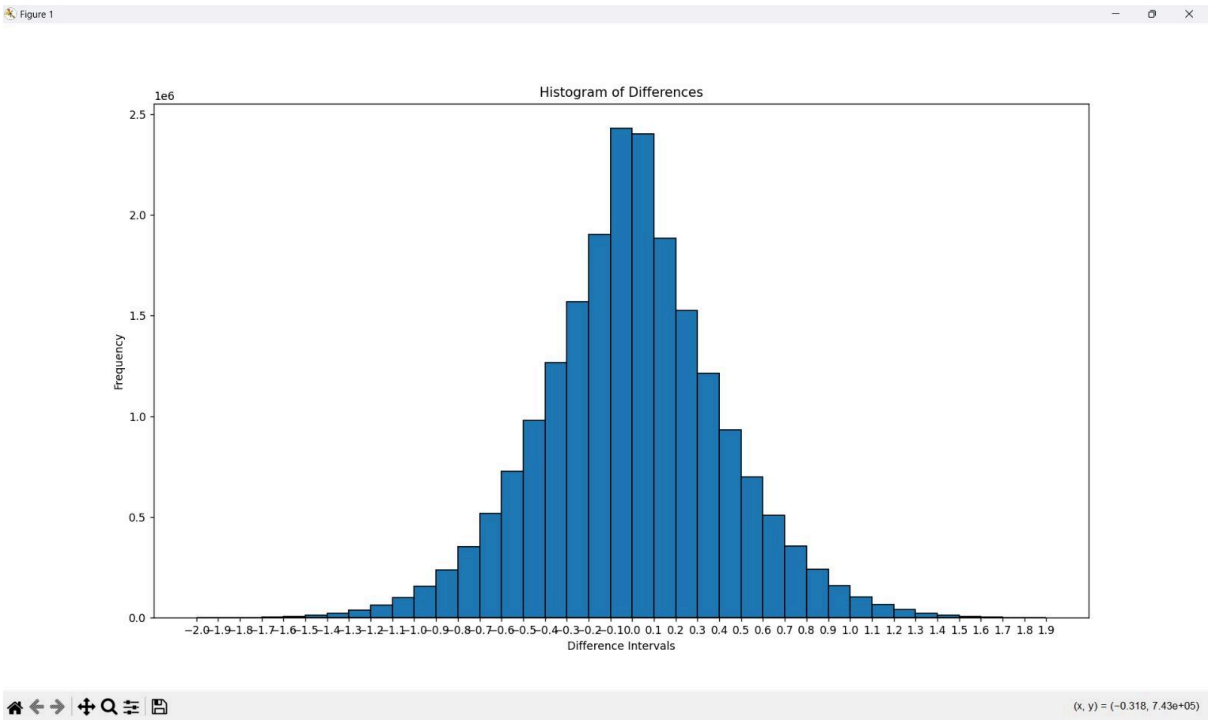
4.Song: Locked out of heaven-Bruno Mars

Alignment of samples between the downloaded (MP3 to WAV) and recorded versions of the song:



Scaling Factor: 17979.05318151373

VS Code Output:



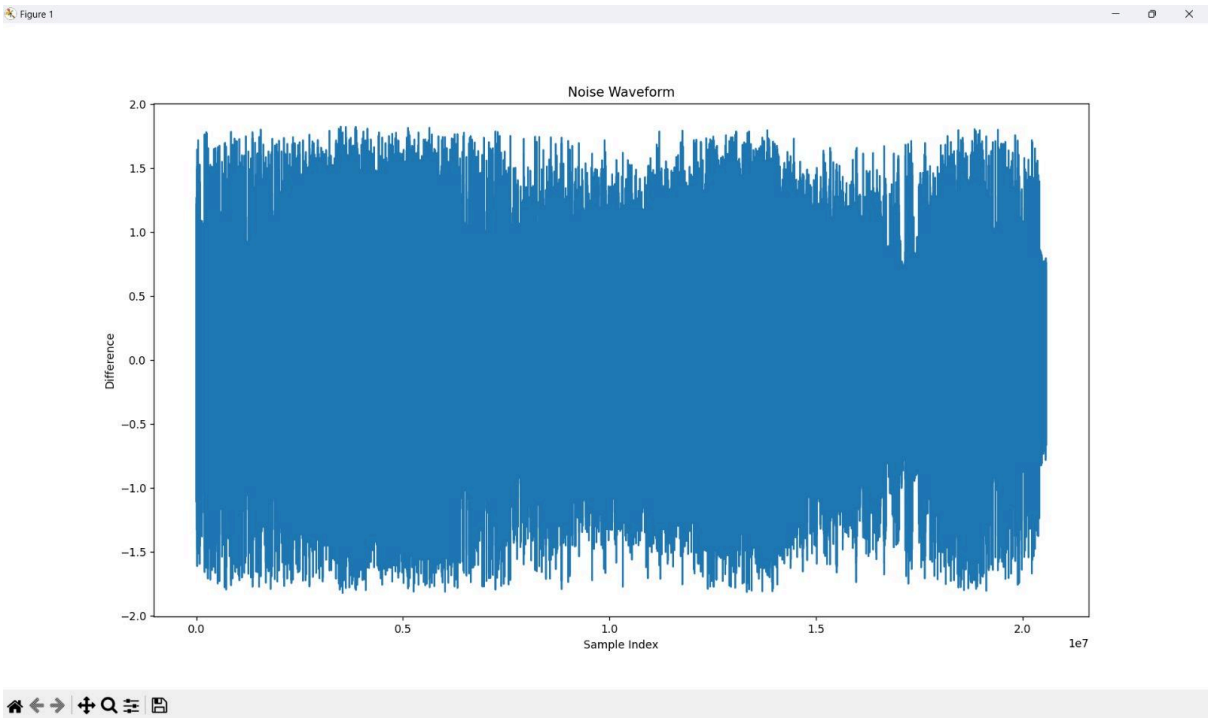


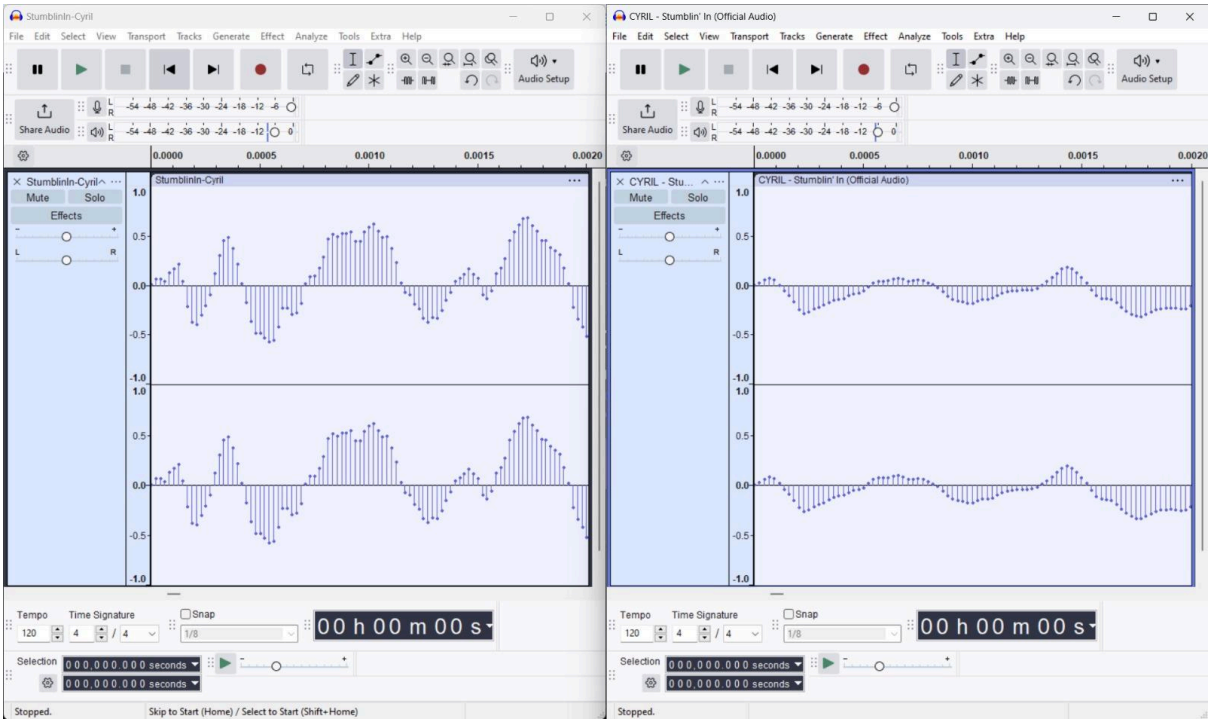
Figure 1
Noise Waveform
Difference
Sample Index
1e7
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0.0
0.5
1.0
1.5
2.0

Second Location: 5km away

- **Fabrica Coffee Roasters**

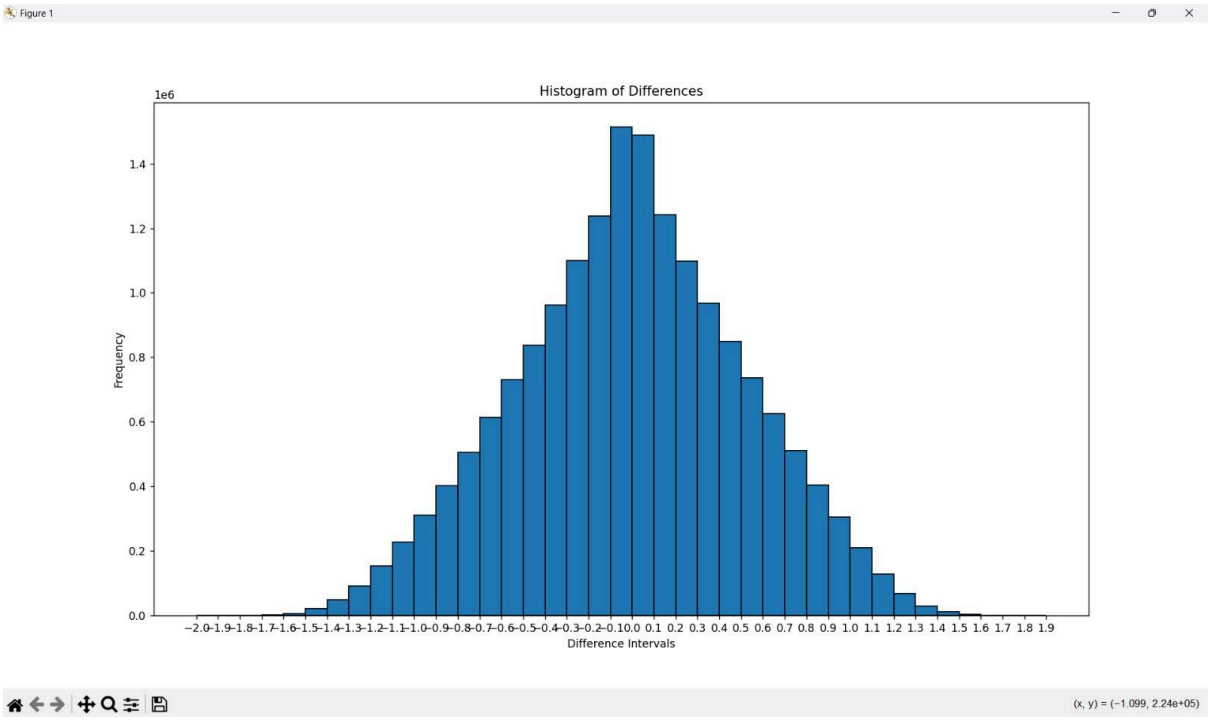
1.Song: Stumblin in - Cyril

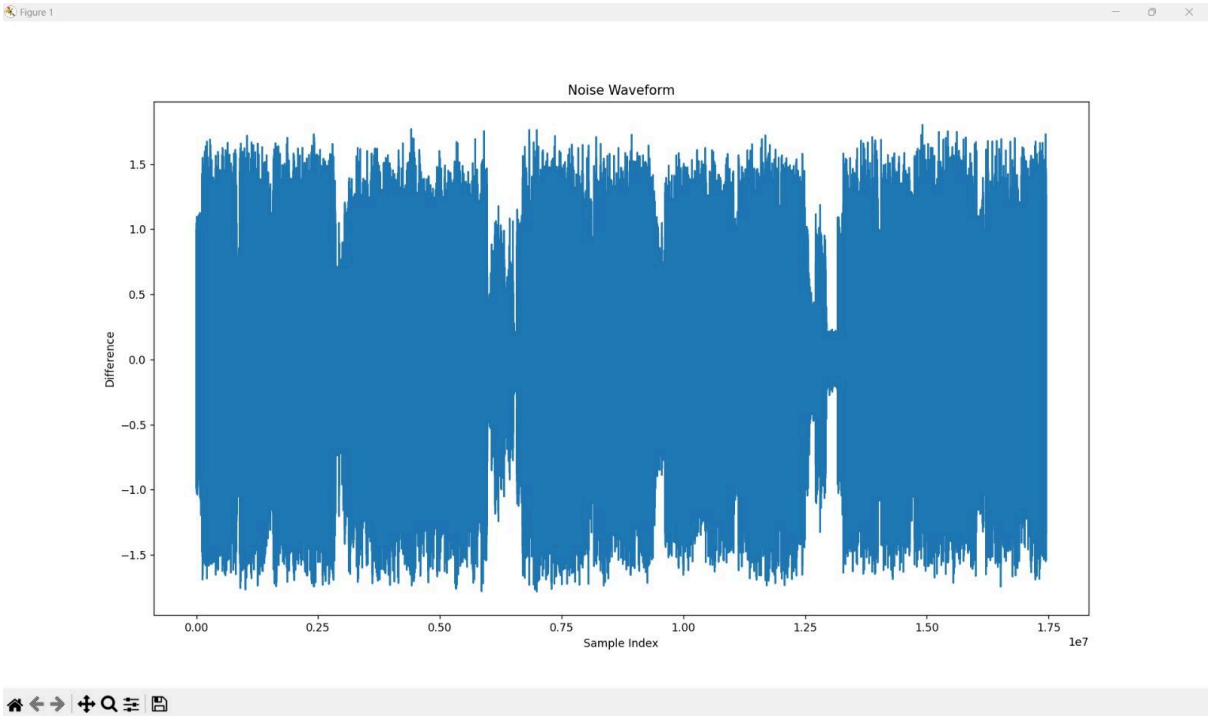
Alignment of samples between the downloaded (MP3 to WAV) and recorded versions of the song:



Scaling Factor: 18167.35852185242

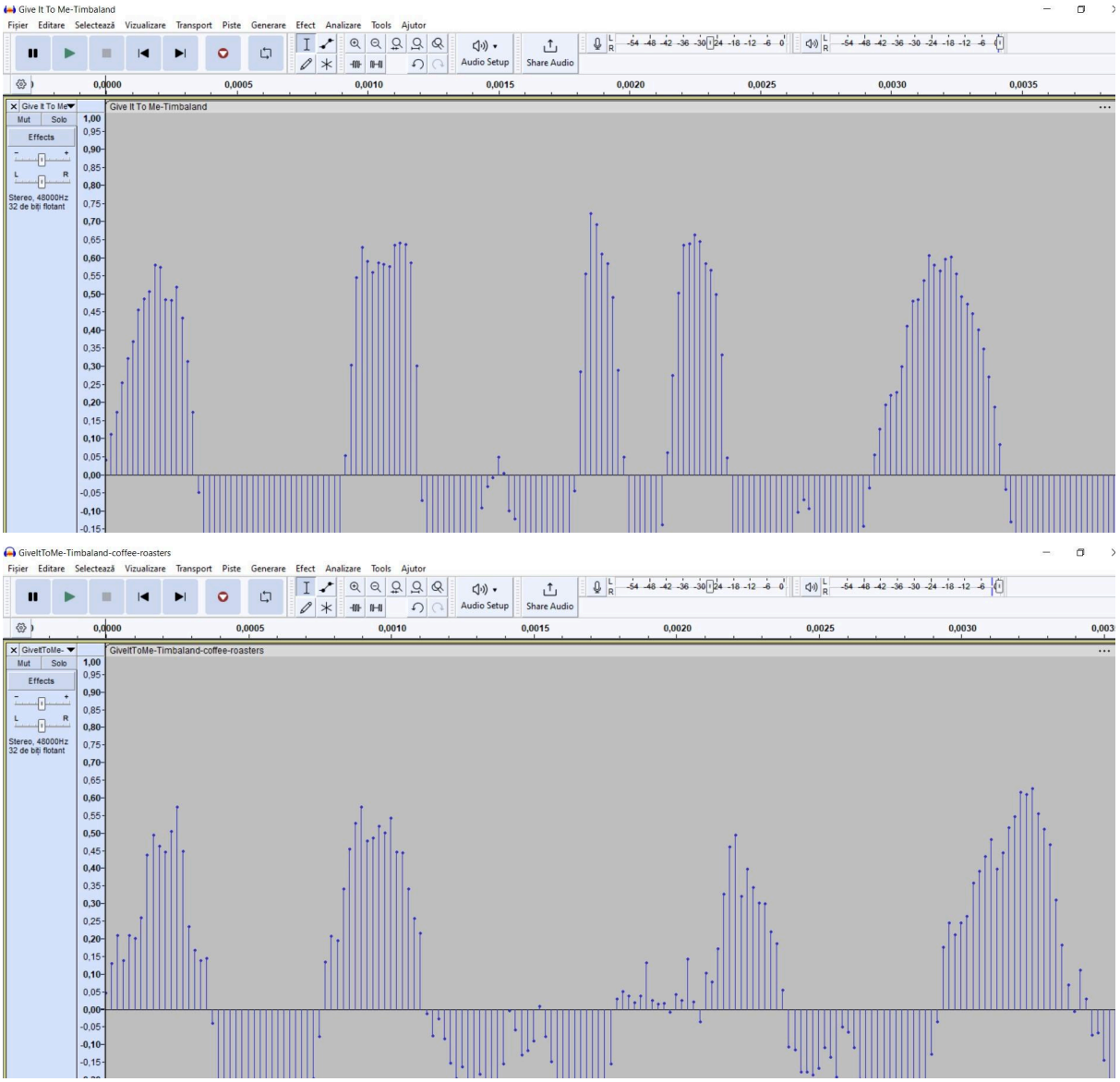
VS Code Output:





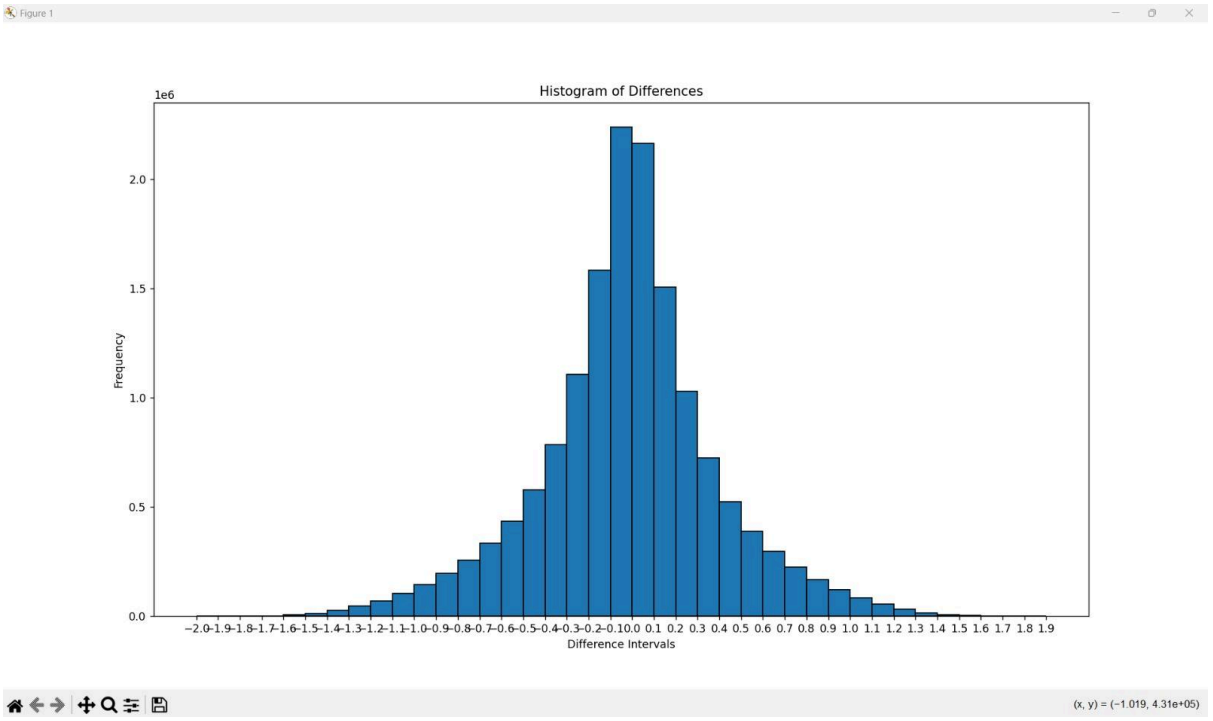
2.Song: Give it to me - Timbaland

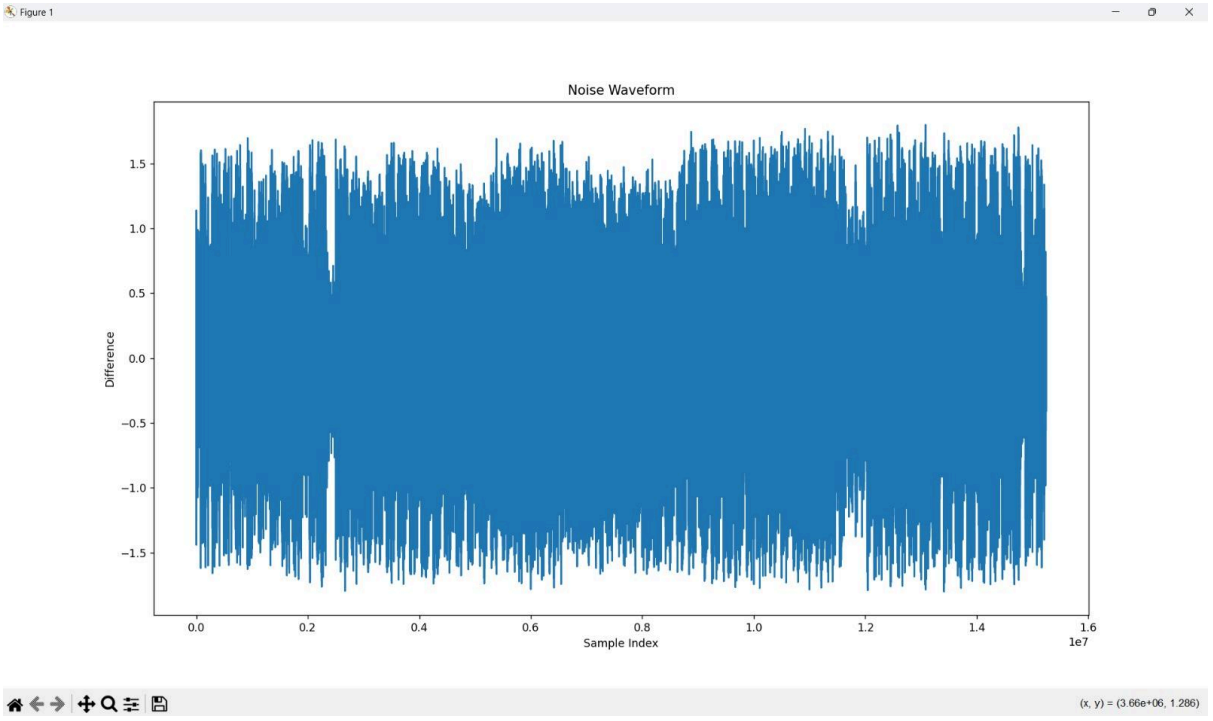
Alignment of samples between the downloaded (MP3 to WAV) and recorded versions of the song:



Scaling Factor: 18201.852141924766

VS Code Output:

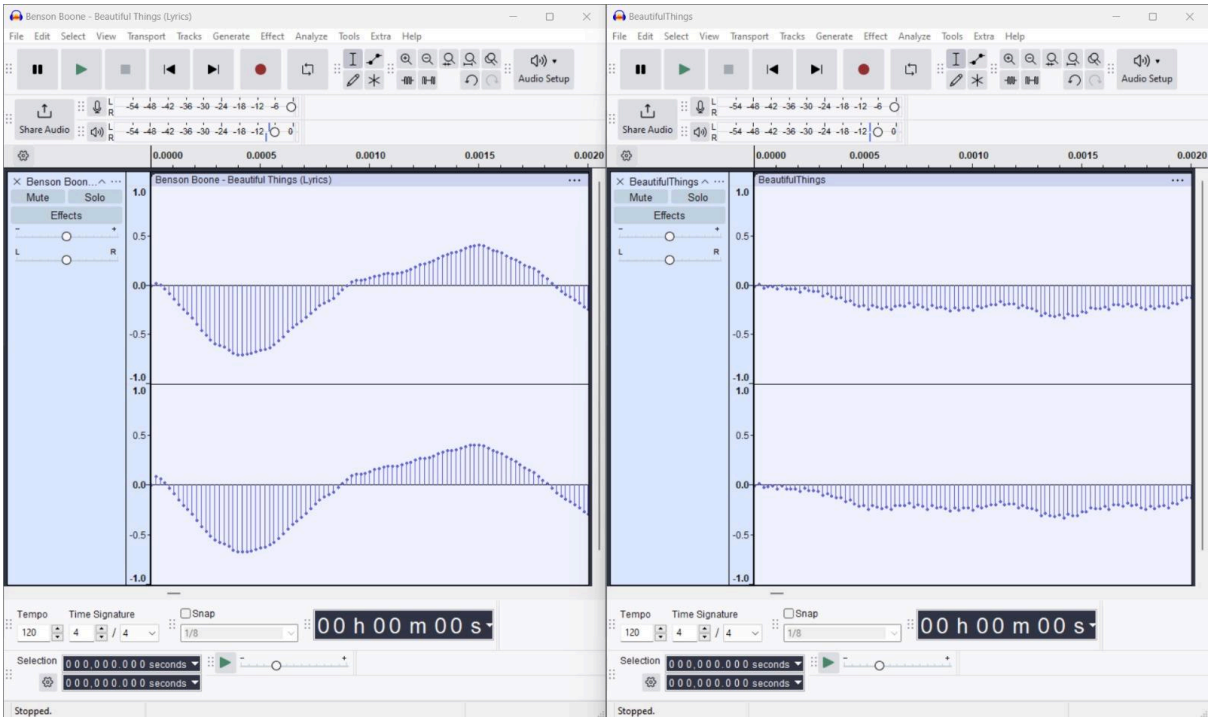




- **Jardim do Torel**

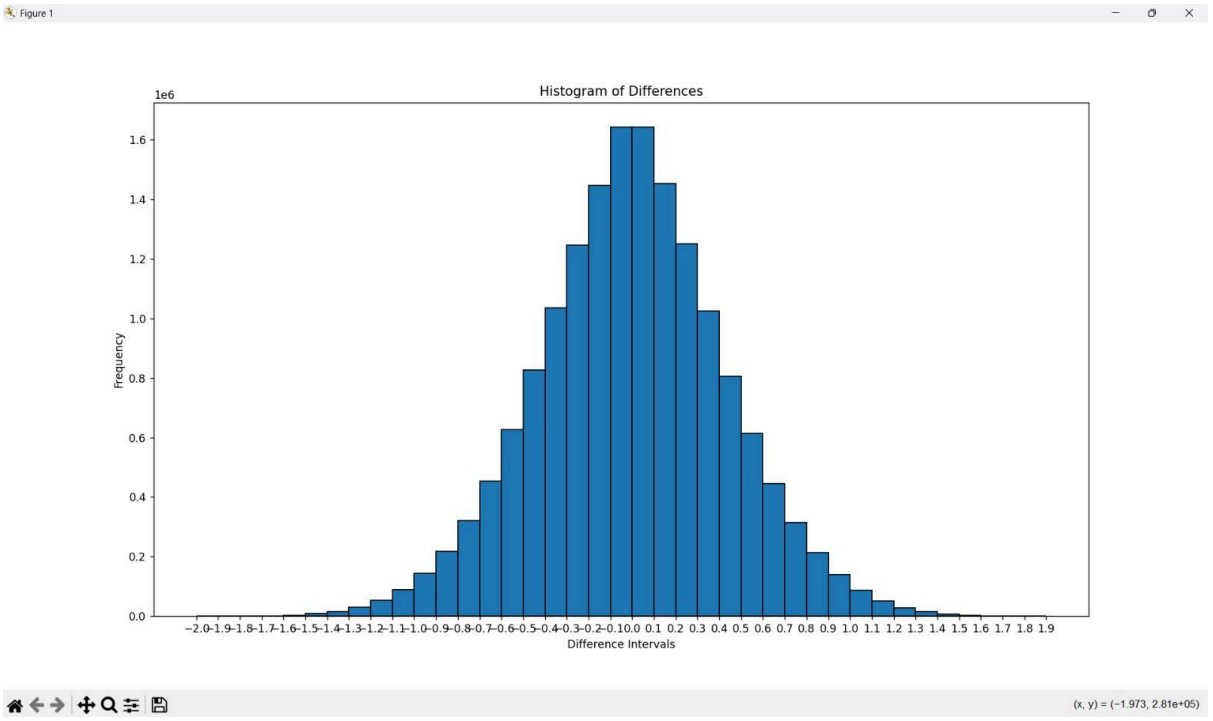
3.Song: Beautiful things - Benson Boone

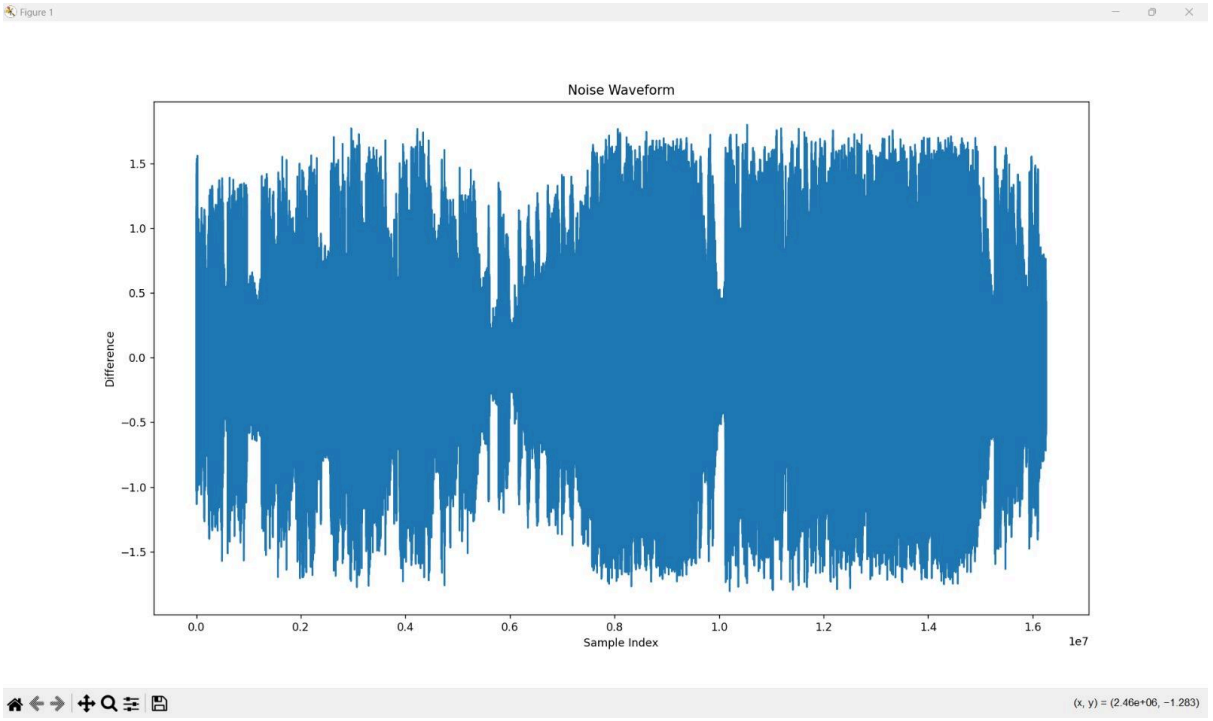
Alignment of samples between the downloaded (MP3 to WAV) and recorded versions of the song:



Scaling Factor: 18141.88050824547

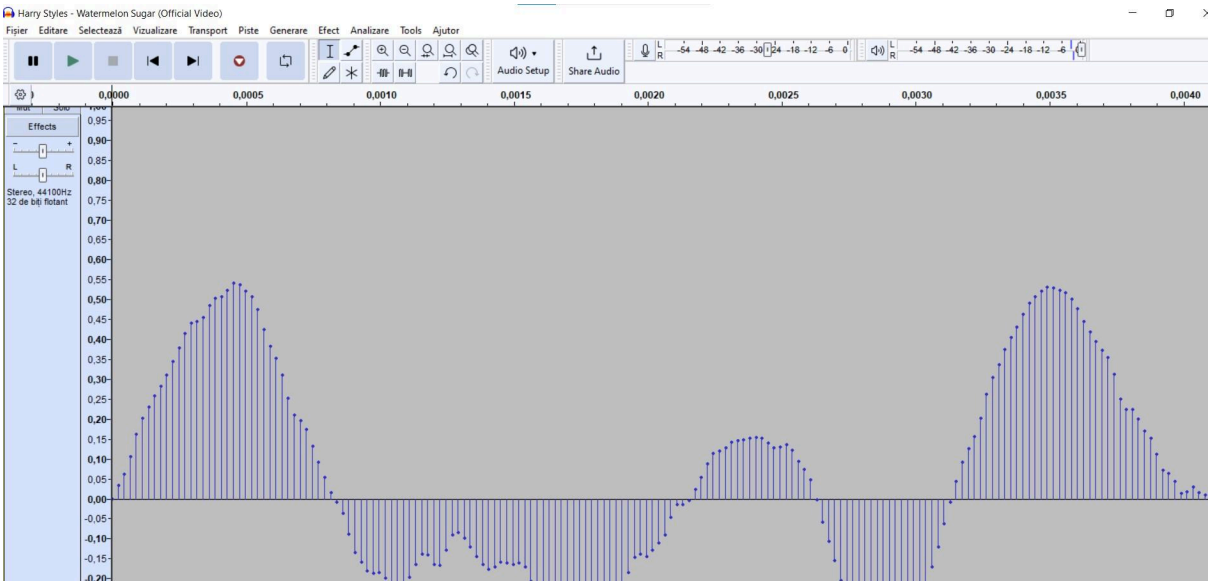
VS Code Output:

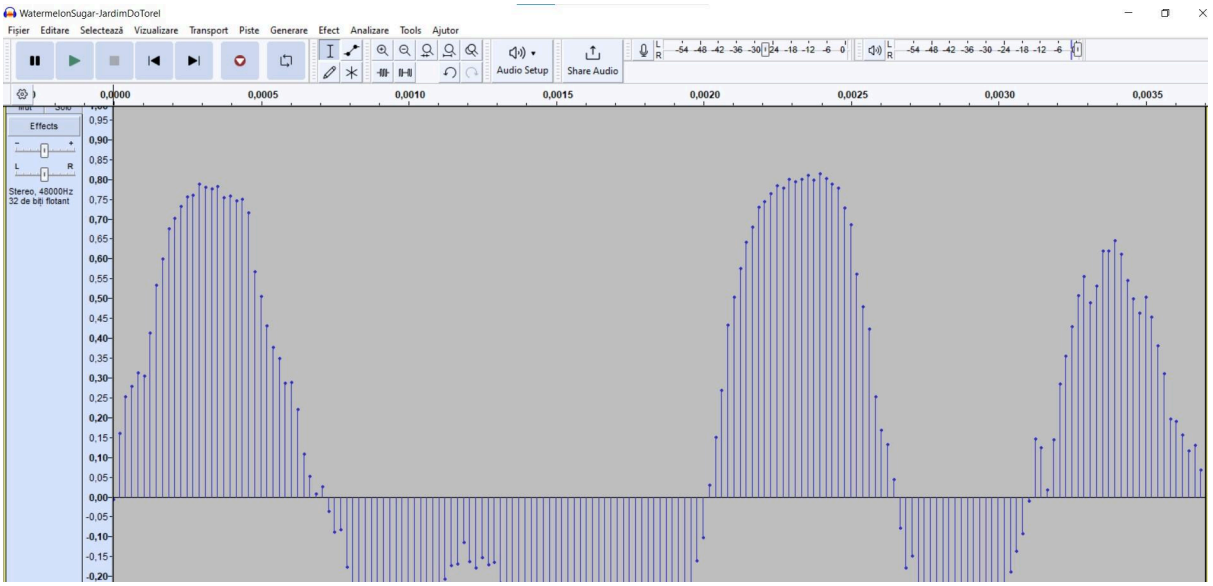




4.Song: Watermelon Sugar - Harry Styles

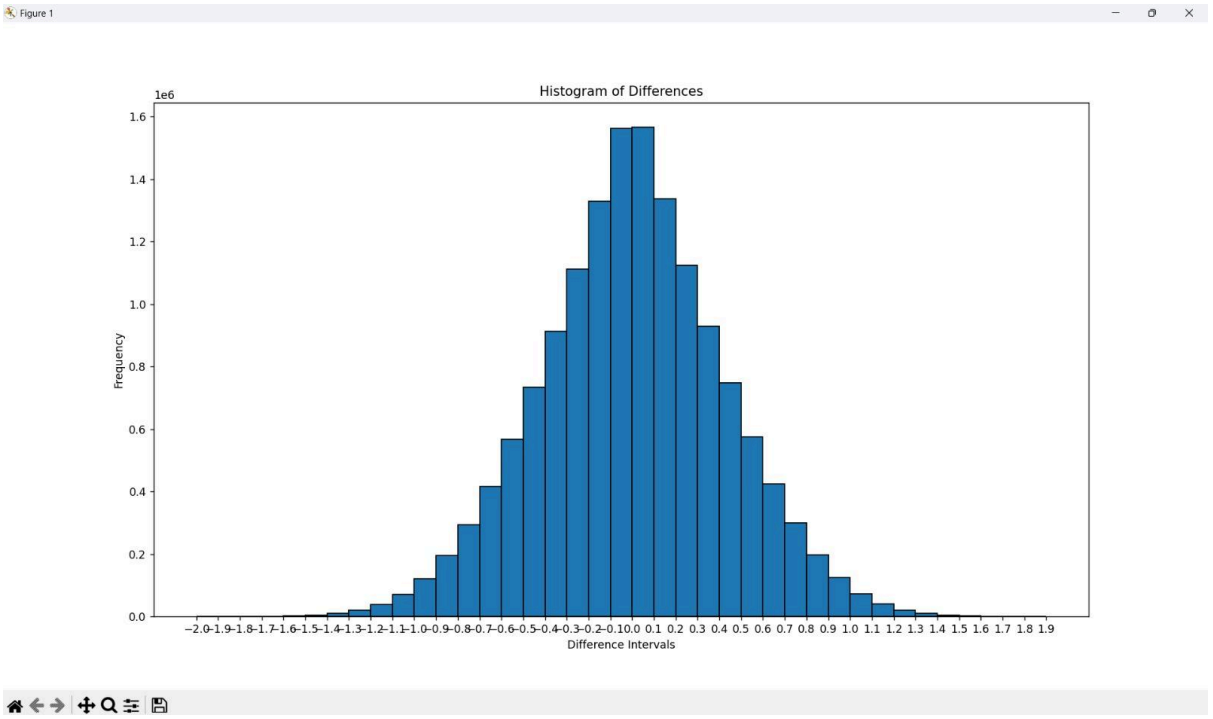
Alignment of samples between the downloaded (MP3 to WAV) and recorded versions of the song:

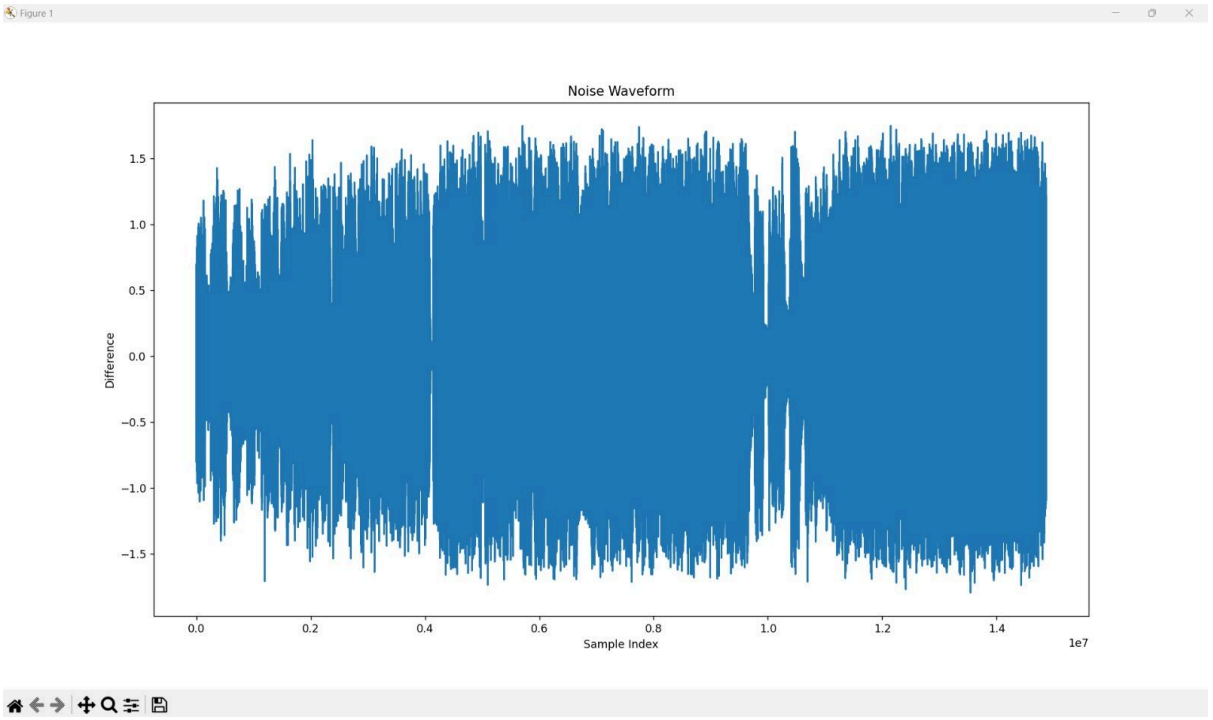




Scaling Factor: 18263.153475872157

VS Code Output:

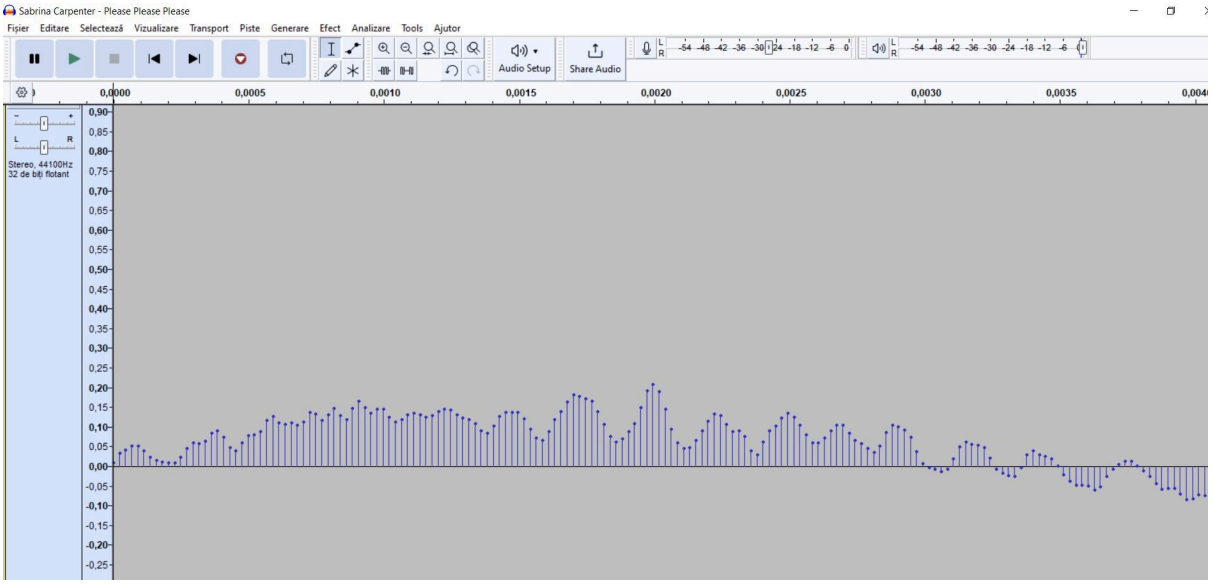


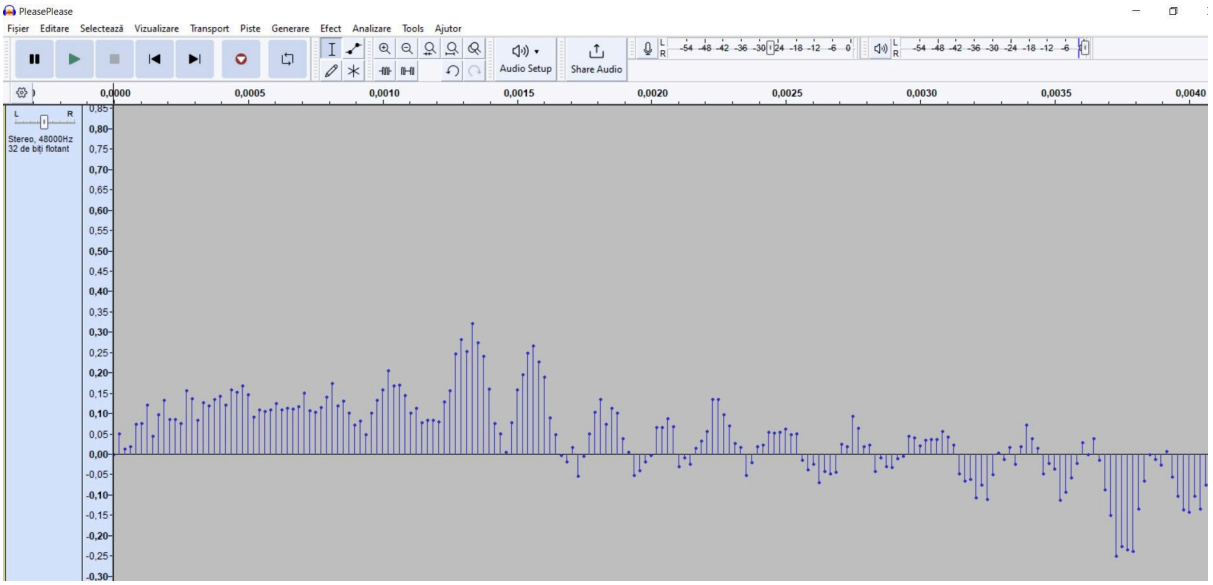


Third Location: Jardins do Palácio Marquês de Pombal (15 km away)

1.Song: Please, please, please - Sabrina Carpenter

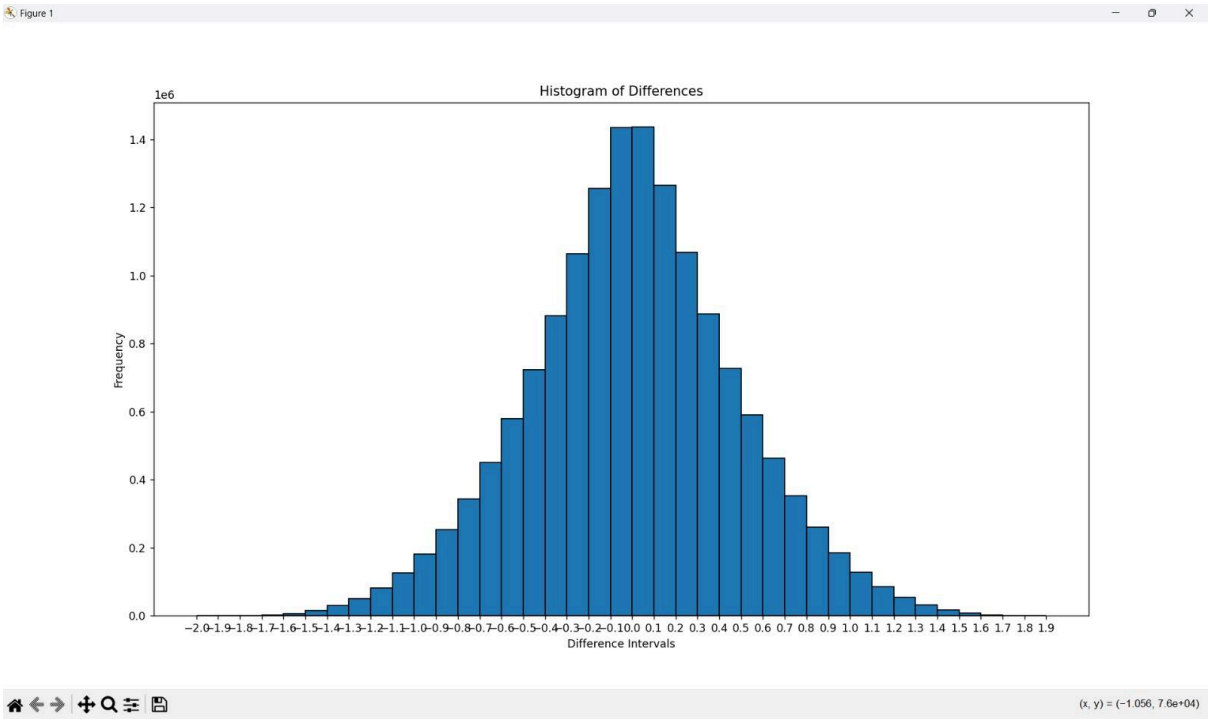
Alignment of samples between the downloaded (MP3 to WAV) and recorded versions of the song:

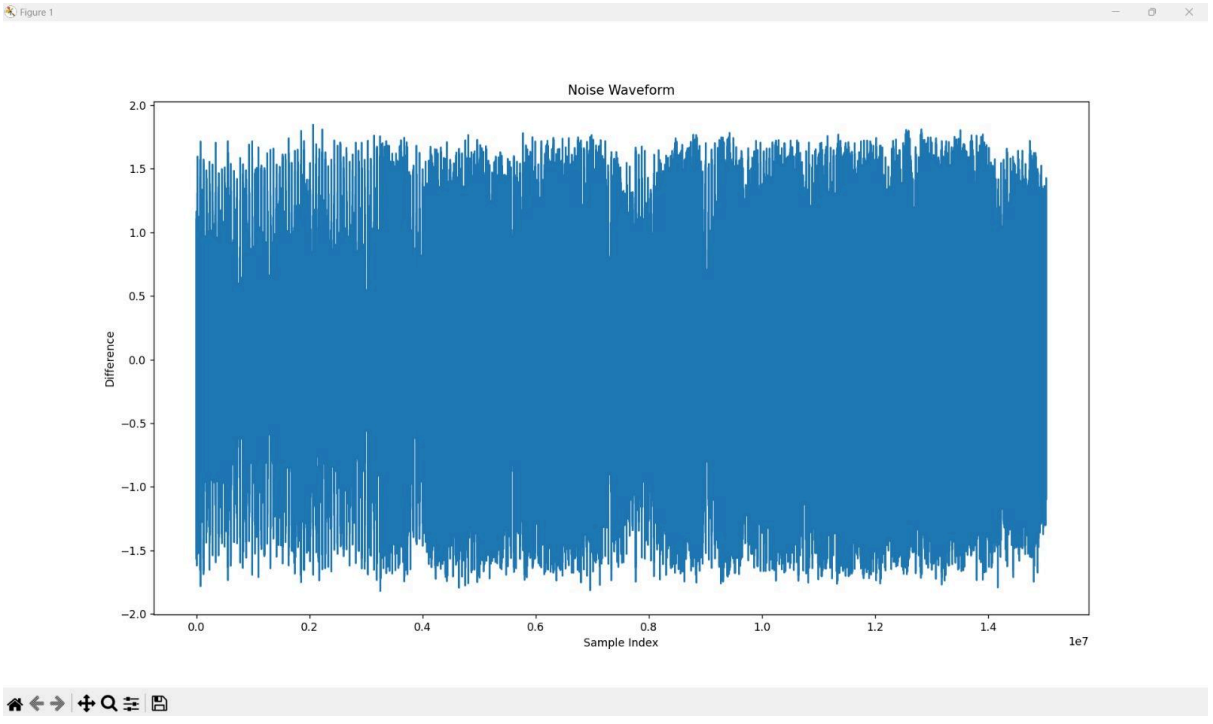




Scaling Factor: 17736.117083484754

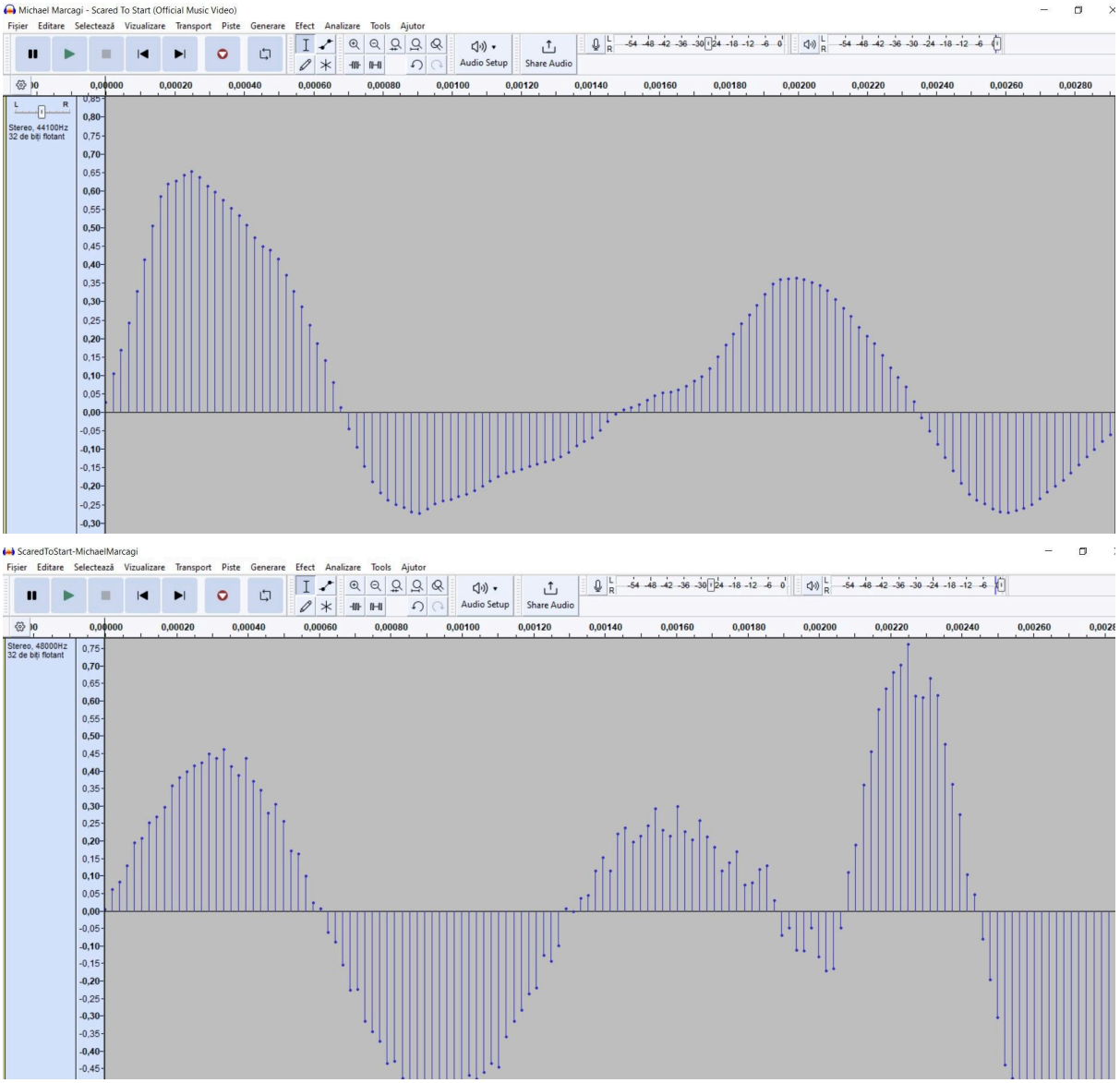
VS Code Output:





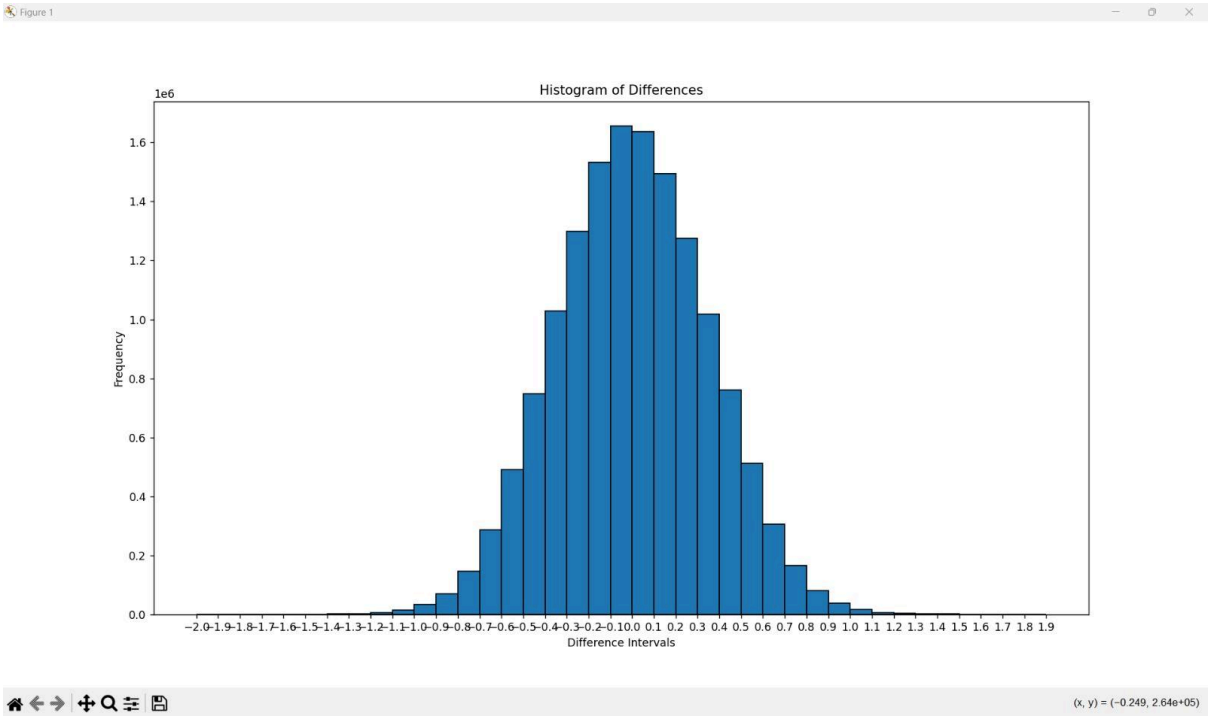
2.Song: Scared to start - Michael Marconi

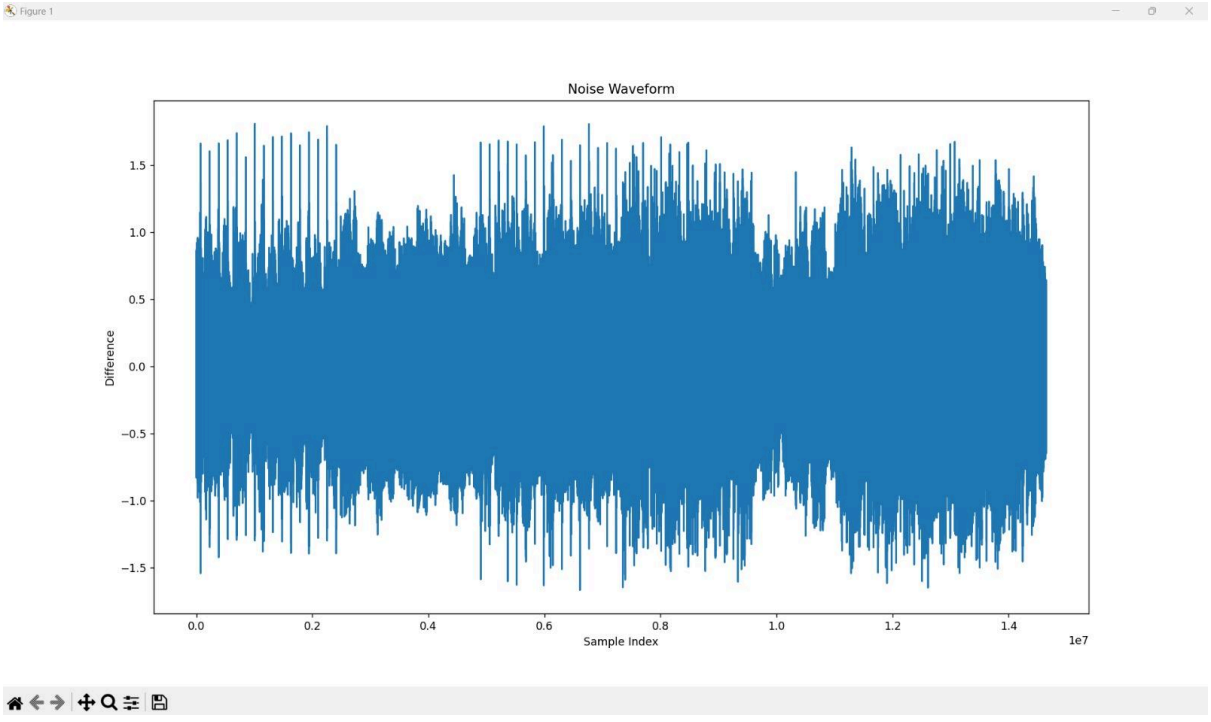
Alignment of samples between the downloaded (MP3 to WAV) and recorded versions of the song:



Scaling Factor: 18081.998248568543

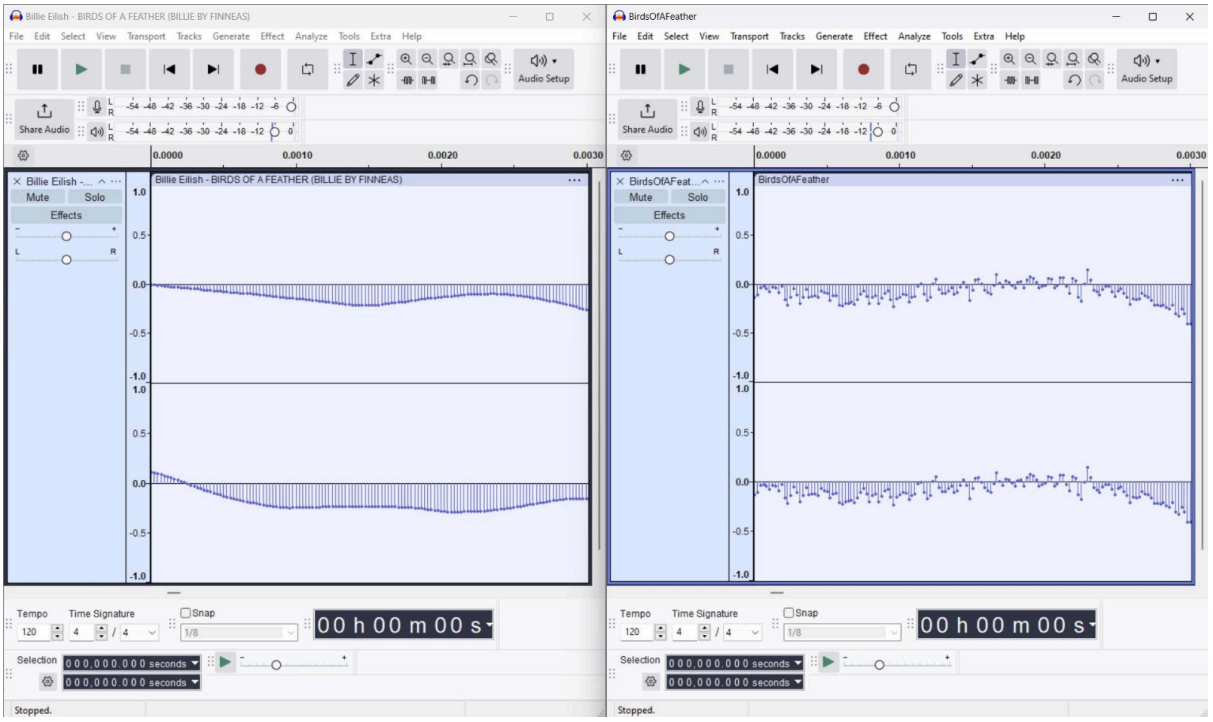
VS Code Output:





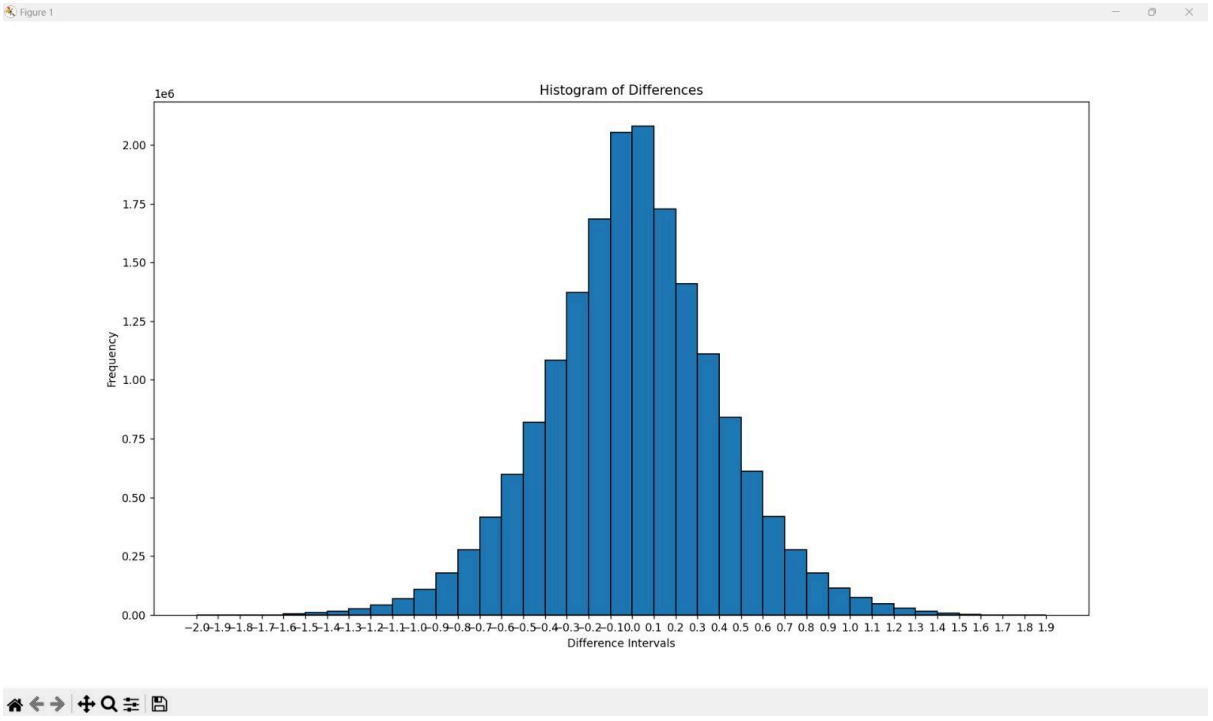
3.Song: Birds of a feather-Billie Eilish

Alignment of samples between the downloaded (MP3 to WAV) and recorded versions of the song:



Scaling Factor: 18163.363264201373

VS Code Output:



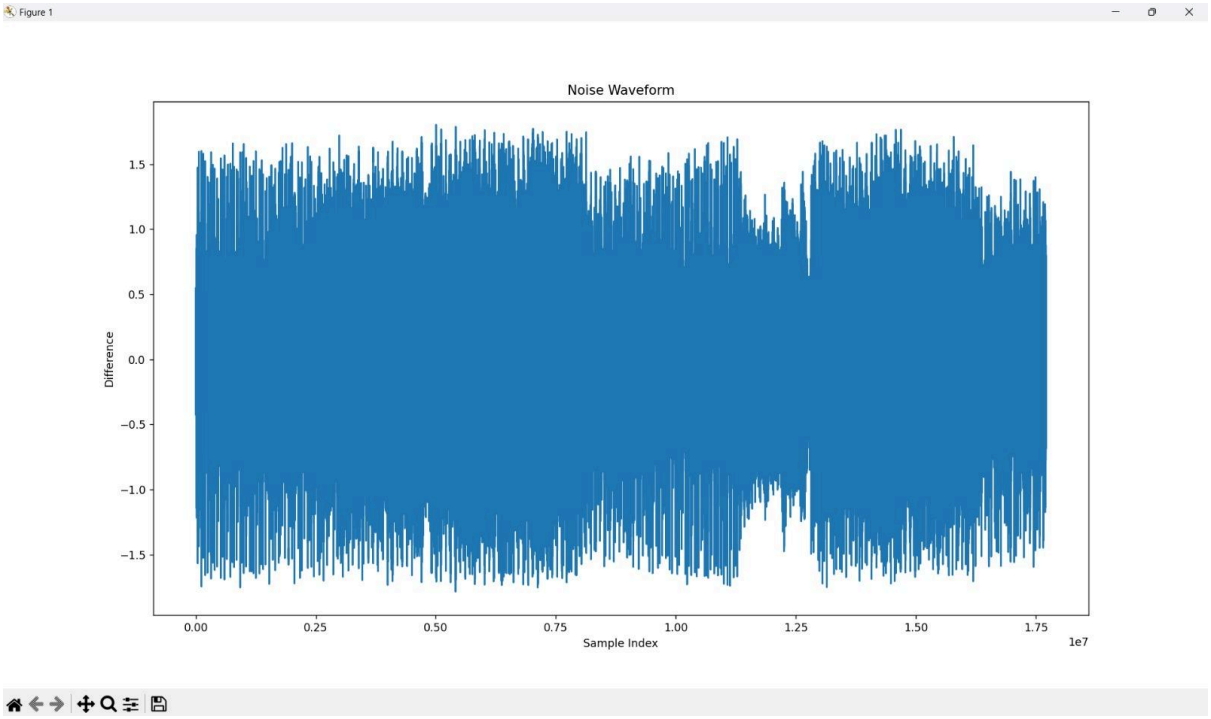
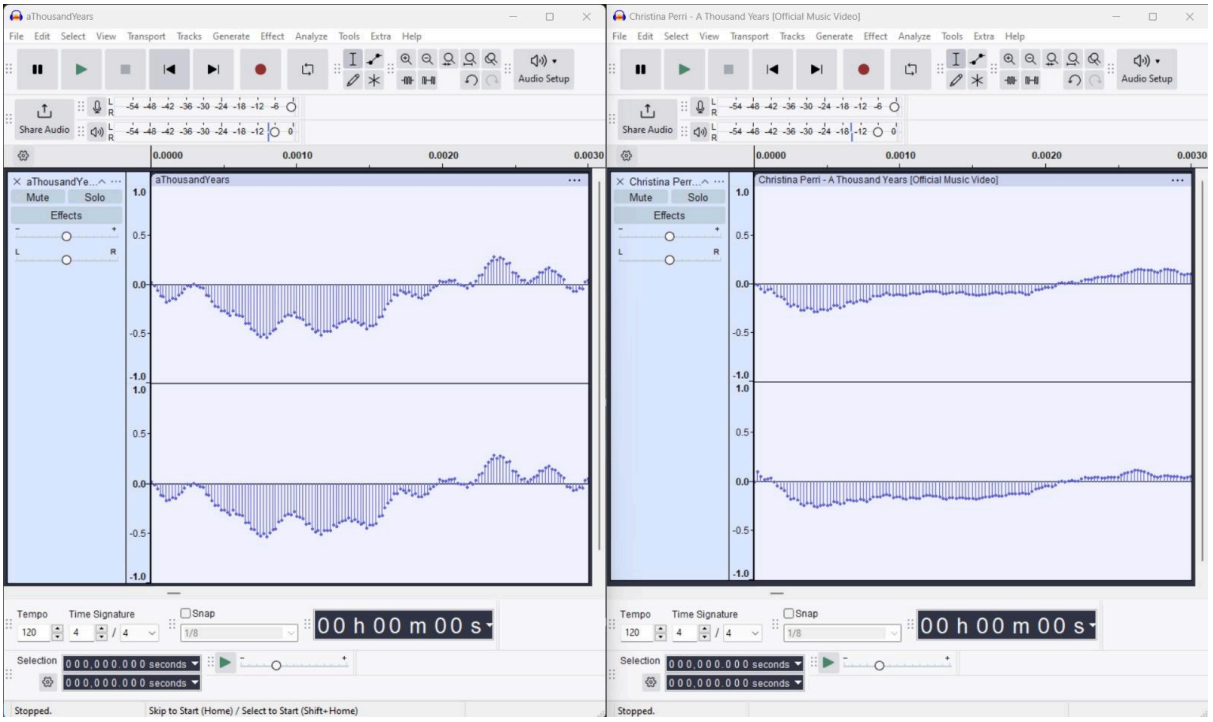


Figure 1
Noise Waveform
Difference
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
Sample Index
1e7

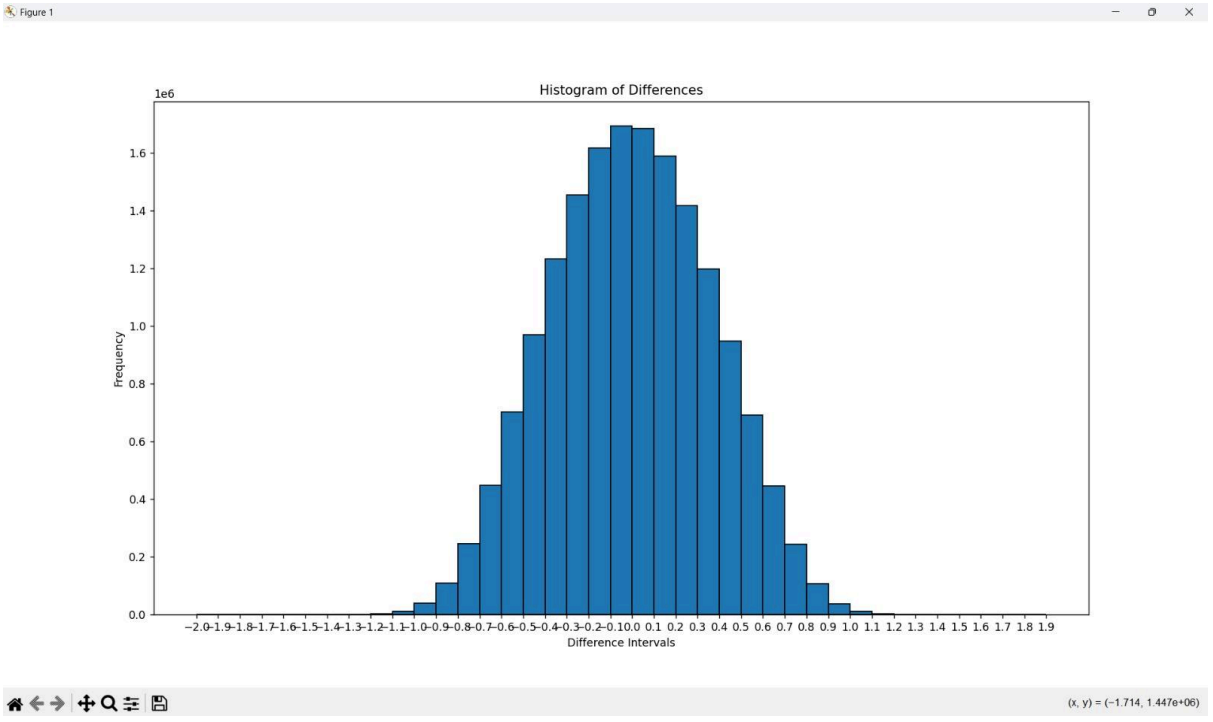
4.Song: A thousand years - Christina Perri

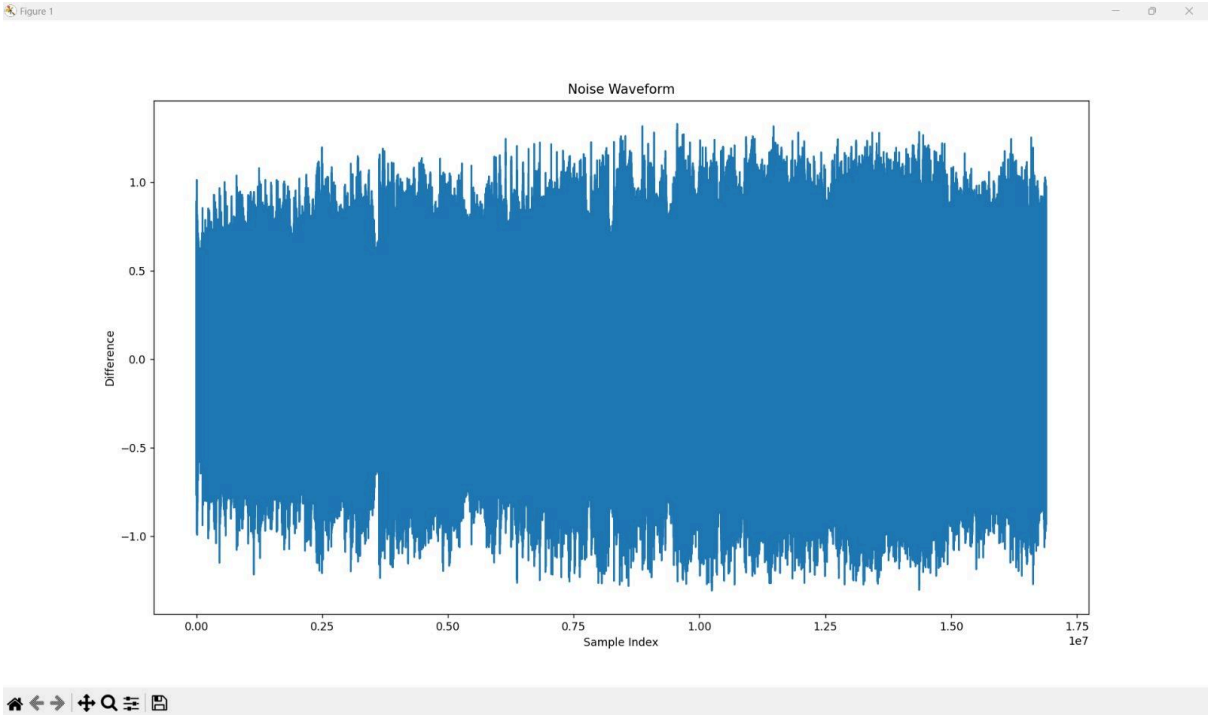
Alignment of samples between the downloaded (MP3 to WAV) and recorded versions of the song:



Scaling Factor: 24613.9346201458

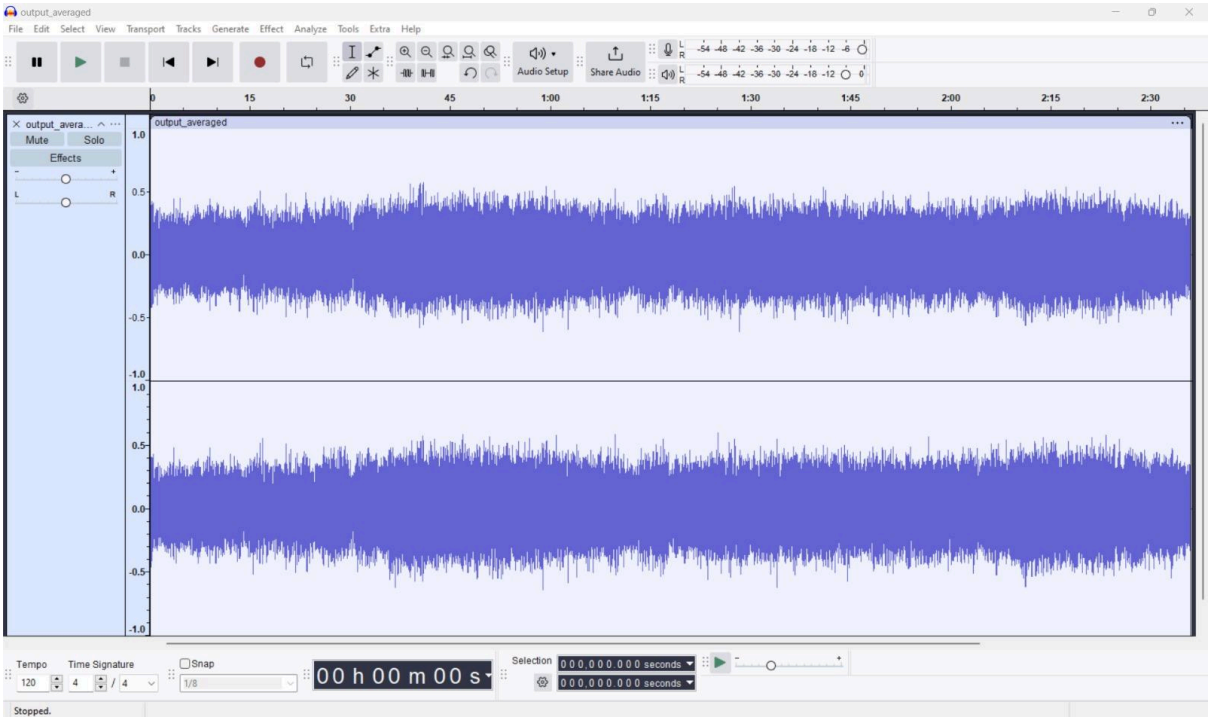
VS Code Output:



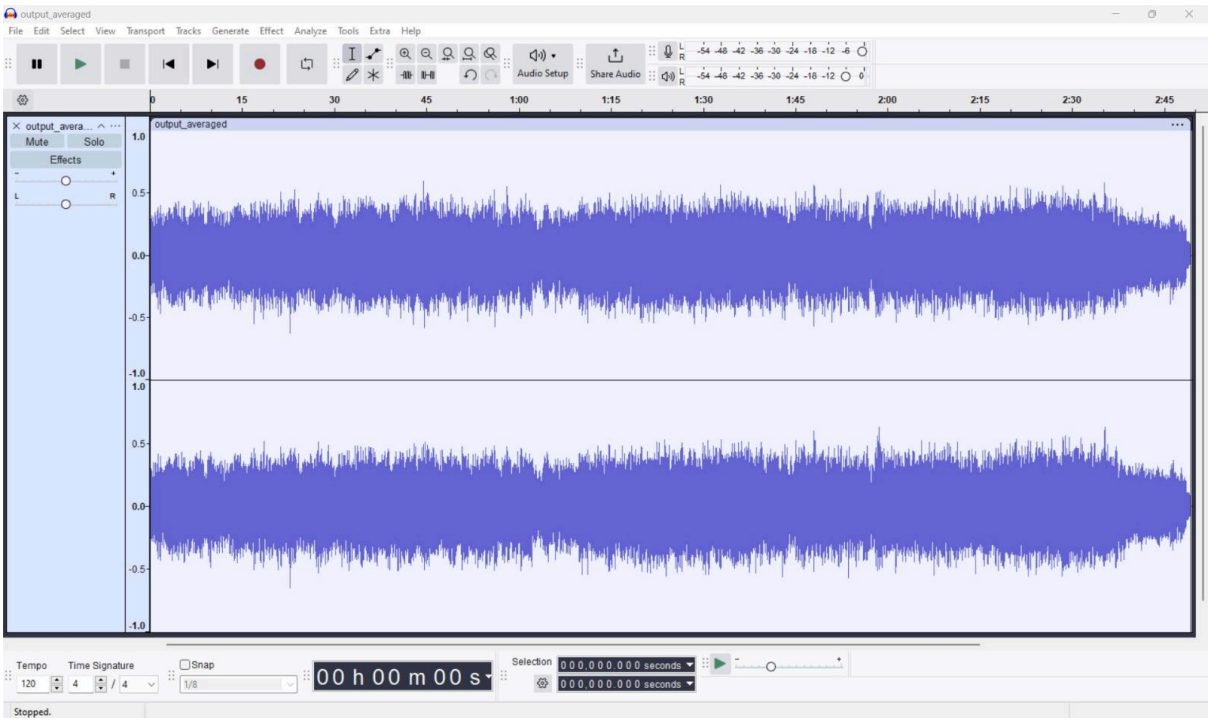


The average noise from the mentioned locations

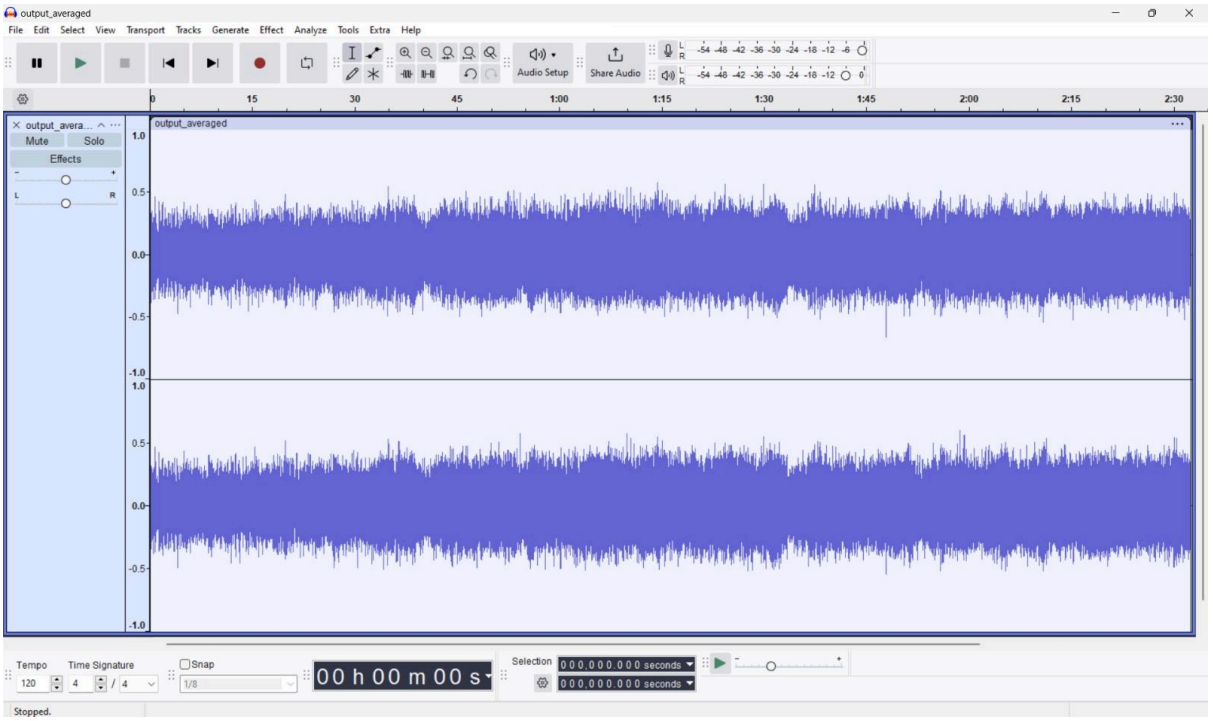
First Location: Monsanto Forest Park, near Altice Cell Tower (70 meters away)



Second Location: Fabrica Coffee Roasters and Jardim do Torel (5km away)



Third Location: Jardins do Palácio Marquês de Pombal (15 km away)



Sample Scaling

The scaling factor is a value calculated to adjust the samples of the recorded song to the same amplitude as those of the downloaded song. This factor varies from one song to another and is influenced by the noise introduced by the FM transmission.

In nearby locations (70 meters), the scaling factors are quite uniform (between 17,846 and 18,030), indicating relatively constant and low noise.

In more distant locations (5 km and 15 km), the scaling factors increase and vary more (reaching up to 24,613 in some cases), suggesting an increase in noise and a greater variation in signal quality.

Differences in Signal Quality

First Location (70 meters):

The histogram analysis reveals a concentrated distribution of differences centered around zero. This indicates a high degree of signal fidelity, with the recorded audio closely mirroring the original signal and exhibiting minimal noise or distortion. The relatively uniform and modest scaling factors further corroborate the low levels of deviation, suggesting that the signal remains largely unaltered at this short range.

Second Location (5 km):

At an approximate distance of 5 km, there is observable broadening in the histograms, indicating an increase in the variability of differences between the recorded and original signals. This widening distribution suggests that the signal is more susceptible to noise and interference, leading to a gradual decline in fidelity. While the signal integrity is moderately preserved, the presence of these broader differences highlights the impact of distance and environmental factors on signal quality. The scaling factors, while larger than those at 70 meters, still indicate that the recorded signal remains relatively consistent with the original, albeit with greater variability.

Third Location (15 km):

The histograms corresponding to recordings made at a distance of 15 km demonstrate a significant increase in variability, with a more pronounced spread of differences. This suggests a degradation in signal quality, likely attributable to the increased effects of distance, environmental interferences, and potential amplification discrepancies in the recorded signal. The wide range of scaling factors, including one notably elevated value, further supports the conclusion that the signal undergoes considerable distortion at this distance. This high variability points to the challenges in maintaining signal integrity over extended distances.

Conclusion

This project aimed to explore the impact of distance on the quality of audio signals transmitted via FM radio, and the process involved several key steps that contributed to the final analysis.

1. Understanding the Structure of .WAV Files
To effectively work with the data, we began by understanding the structure of .wav files. This step was crucial for detecting essential parameters needed for processing and comparing the files, such as sample rate, number of channels, and sample width. This foundational knowledge allowed us to become familiar with tools like HxD Editor and Audacity, which proved invaluable in representing and understanding audio signals.

2. Data Collection Using RTL-SDR Equipment
With the help of an RTL-SDR receiver, we were able to collect data in the form of recorded songs at various distances from radio mast. Learning to use this equipment and understanding its structure, as well as installing the necessary drivers, provided us with a deeper insight into the transmission and reception of signals, and more broadly, into analog-digital communication.

3. Development of the Comparison Software
To draw meaningful conclusions, we developed software in Python capable of comparing two .wav files by calculating the amplitude differences between them and isolating noise in amplitude. Additionally, we created a tool that averages noise by processing multiple files containing amplitude differences from the first application. This software allowed us to analyze the real-world data collected through recordings at various distances from the main antenna broadcasting radio stations in Lisbon.

4. Data Collection and Analysis
We gathered real data by recording songs at different distances from the antenna: 70 meters, 5 kilometers (from different locations), and 15 kilometers enabling us to reach conclusions on how distance impacts sound quality and reception. The results of our analysis led to several key observations:

- Types of Noise: Noise can manifest in various forms, including amplitude and frequency, and is influenced by numerous factors such as environmental interference, signal reflections, and equipment quality. For example, buildings, weather conditions, and even the type of receiver used can significantly affect the noise level in a recorded signal.
- Multifactorial Influence: It became evident that noise cannot be isolated purely based on distance without considering other factors that may cause distortions. The complexity of these interactions means that a comprehensive analysis must account

for multiple variables, including transmission power, environmental conditions, and receiver sensitivity.

- Impact of File Compression: Radio stations typically use MP3 files rather than WAV files. The compression and modification of the original file during transmission can introduce errors and differences that affect the final recorded signal. This compression can lead to artifacts that might be mistaken for noise or distortion, complicating the analysis.

Influence of Noise and Possible Alignment Errors

It is important to note that manual alignment of the signals might introduce errors, which could influence the calculated scaling factors. Additionally, variations in background noise and recording conditions, such as post-recording amplification, could affect the results. These aspects should be considered when interpreting the data.

General Conclusion

While it is expected that noise increases as the distance from the antenna increases, the results do not show a perfect correlation between distance and added noise. This can be explained by the variability in recording conditions, possible signal amplifications, and errors introduced by manual alignment. The results suggest that while distance is an important factor, other variables can also influence perceived noise and, consequently, the scaling factor.

Throughout the project, we experienced periods of smooth progress where everything functioned as expected, as well as slower phases where challenges and unforeseen obstacles slowed down the pace of our work. Despite these fluctuations, the conclusions we reached are highly relevant to the project's objectives. They provide significant insights into how distance affects signal quality and offer valuable implications for future work, particularly in the analysis of data and the potential use of radio communication for private communication networks.

ANNEX 1

Comparison Software

```
import wave
import binascii
import struct
import numpy as np
import matplotlib.pyplot as plt
import argparse
import os

def get_wav_header_info(wav_file):
    with wave.open(wav_file, 'rb') as wf:
        header_info = {
            "Number of Channels": wf.getnchannels(),
            "Sample Width (bytes)": wf.getsampwidth(),
            "Frame Rate (samples/sec)": wf.getframerate(),
            "Number of Frames": wf.getnframes(),
            "Compression Type": wf.getcomptype(),
            "Compression Name": wf.getcompname()
        }
    return header_info

def print_wav_header_info(header_info, wav_file):
    print(f"File: {wav_file}")
    for key, value in header_info.items():
        print(f"{key}: {value}")
    print()

def read_wav_data(wav_file):
    with wave.open(wav_file, 'rb') as wf:
        raw_data = wf.readframes(wf.getnframes())
        return raw_data

def get_samples(raw_data, sample_width):
    num_samples = len(raw_data) // sample_width
    format_char = '<h' if sample_width == 2 else '<b'

    samples = []
    for i in range(num_samples):
        sample = struct.unpack(format_char,
raw_data[i*sample_width:(i+1)*sample_width])[0]
```

```
        samples.append(sample)

    return samples

def hex_to_normalized_decimal(raw_data, sample_width):
    num_samples = len(raw_data) // sample_width
    format_char = 'h' if sample_width == 2 else 'b'
    max_value = 2**(8*sample_width - 1)

    normalized_values = []
    for i in range(num_samples):
        sample = struct.unpack('<' + format_char,
raw_data[i*sample_width:(i+1)*sample_width])[0]
        normalized_value = sample / max_value
        normalized_values.append(normalized_value)

    return normalized_values

def calculate_differences(normalized_values1, normalized_values2):
    num_samples = min(len(normalized_values1), len(normalized_values2))
    normalized_differences = []

    for i in range(num_samples):
        norm_value1 = normalized_values1[i]
        norm_value2 = normalized_values2[i]
        normalized_difference = norm_value1 - norm_value2
        normalized_differences.append(normalized_difference)

    return normalized_differences

def write_difference_wav(differences, output_wav_file, sample_rate,
sample_width):
    if sample_width == 1:
        max_amplitude = 2**7 - 1
    elif sample_width == 2:
        max_amplitude = 2**15 - 1
    else:
        raise ValueError("Unsupported sample width. Only 1-byte and
2-byte samples are supported.")

    max_diff = max(differences)
    min_diff = min(differences)
```

```
    if max_diff > 1 or min_diff < -1:
        scale_factor = max_amplitude / max(abs(min_diff),
abs(max_diff))
    else:
        scale_factor = max_amplitude

    print(f"Scale factor: {scale_factor}")

    with wave.open(output_wav_file, 'w') as wf:
        wf.setnchannels(2)
        wf.setsampwidth(sample_width)
        wf.setframerate(sample_rate)

        for diff in differences:

            scaled_diff = int(diff * scale_factor)
            scaled_diff = max(-max_amplitude, min(max_amplitude,
scaled_diff))

            if sample_width == 1:
                wf.writeframes(struct.pack('<B', scaled_diff + 128))
            elif sample_width == 2:
                wf.writeframes(struct.pack('<h', scaled_diff))

def plot_histogram(differences):
    bins = np.arange(-2, 2, 0.1)
    hist, edges = np.histogram(differences, bins=bins)

    plt.hist(differences, bins=bins, edgecolor='black')
    plt.xlabel('Difference Intervals')
    plt.ylabel('Frequency')
    plt.title('Histogram of Differences')
    plt.xticks(bins)
    plt.show()

    return hist, edges

def plot_noise_wave(differences):
    plt.plot(differences)
    plt.xlabel('Sample Index')
    plt.ylabel('Difference')
    plt.title('Noise Waveform')
    plt.show()
```

```
def compare_wav_files(file1, file2, output_type):
    header_info1 = get_wav_header_info(file1)
    header_info2 = get_wav_header_info(file2)

    print("Header Information for File 1:")
    print_wav_header_info(header_info1, file1)

    print("Header Information for File 2:")
    print_wav_header_info(header_info2, file2)

    raw_data1 = read_wav_data(file1)
    raw_data2 = read_wav_data(file2)

    sample_width1 = header_info1["Sample Width (bytes)"]
    sample_width2 = header_info2["Sample Width (bytes)"]

    samples1 = get_samples(raw_data1, sample_width1)
    samples2 = get_samples(raw_data2, sample_width2)

    normalized_values1 = hex_to_normalized_decimal(raw_data1,
sample_width1)
    normalized_values2 = hex_to_normalized_decimal(raw_data2,
sample_width2)

    differences = calculate_differences(normalized_values1,
normalized_values2)

    print(f"First 10 hexadecimal samples for {file1}:
{[binascii.hexlify(struct.pack('<' + ('h' if sample_width1 == 2 else
'b'), sample)).decode('utf-8') for sample in samples1][:10]}")
    print(f"First 10 hexadecimal samples for {file2}:
{[binascii.hexlify(struct.pack('<' + ('h' if sample_width2 == 2 else
'b'), sample)).decode('utf-8') for sample in samples2][:10]}")

    print(f"First 10 normalized decimal values for {file1}:
{normalized_values1[:10]}")
    print(f"First 10 normalized decimal values for {file2}:
{normalized_values2[:10]}")

    if output_type == "histogram":
        hist, edges = plot_histogram(differences)
```

```
        with open("histogram_data.txt", "w") as f:
            f.write("Difference Interval\tFrequency\n")
            for i in range(len(edges) - 1):
                interval = f"{edges[i]:.2f} to {edges[i+1]:.2f}"
                frequency = hist[i]
                f.write(f"{interval}\t{frequency}\n")

        with open("histogram_data.txt", "r") as f:
            print(f.read())
    elif output_type == "difference":
        plot_noise_wave(differences)

        with open("differences_data.txt", "w") as f:
            f.write("Sample Index\tDifference\n")
            for index, difference in enumerate(differences):
                f.write(f"{index}\t{difference:.6f}\n")

        output_wav_file =
r"C:\Users\anana\OneDrive\Desktop\compare_wav\differences.wav"
        sample_rate = header_info1["Frame Rate (samples/sec)"]
        sample_width = sample_width1

        print(f"Attempting to write differences to: {output_wav_file}")
        write_difference_wav(differences, output_wav_file, sample_rate,
sample_width)
        print(f"Differences written to {output_wav_file}")

        if os.path.exists(output_wav_file):
            print(f"File {output_wav_file} successfully created.")
        else:
            print(f"File {output_wav_file} was not created.")

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Compare two WAV files
and output a histogram or noise wave.")
    parser.add_argument("-audio1", required=True, help="Path to the
first audio file")
    parser.add_argument("-audio2", required=True, help="Path to the
second audio file")
    parser.add_argument("-output", required=True, choices=["histogram",
"difference"], help="Type of output: histogram or difference")
    args = parser.parse_args()
    compare_wav_files(args.audio1, args.audio2, args.output)
```

ANNEX 2

Software for noise averaging

```
import numpy as np
import scipy.io.wavfile as wavfile
import sys

def average_wav_files(file_list):
    if len(file_list) < 2:
        raise ValueError("You must specify at least two WAV files.")

    sample_rate, data = wavfile.read(file_list[0])
    num_channels = data.shape[1] if len(data.shape) > 1 else 1
    min_samples = len(data)

                total_data    =    np.zeros((min_samples,    num_channels),
dtype=np.float64)

    for file in file_list:
        sr, d = wavfile.read(file)
        if sr != sample_rate:
              raise ValueError("All WAV files must have the same sample
rate.")
        num_samples = len(d)
        min_samples = min(min_samples, num_samples)
        d = d[:min_samples]

        if len(d.shape) > 1:
            if d.shape[1] != num_channels:
                    raise ValueError("All WAV files must have the same
number of channels.")
            total_data[:min_samples] += d.astype(np.float64)
        else:
            if num_channels != 1:
                  raise ValueError("WAV files must have the same number
of channels.")
            d_stereo = np.stack([d, d], axis=1)
            total_data[:min_samples] += d_stereo.astype(np.float64)

    average_data = total_data / len(file_list)

    average_data = np.clip(average_data, -32768, 32767)
```

```
    average_data = average_data.astype(np.int16)

    return sample_rate, average_data

def main():
    if len(sys.argv) < 3:
        print("python script.py file1.wav file2.wav [file3.wav ...]")
        sys.exit(1)

    input_files = sys.argv[1:]

    sample_rate, averaged_data = average_wav_files(input_files)

    output_filename = 'output_averaged.wav'
    wavfile.write(output_filename, sample_rate, averaged_data)
    print(f"The resulting WAV file was saved as'{output_filename}'.")

if __name__ == "__main__":
    main()
```