University of Aveiro

Information and Coding

Lab work n^{o} 1

Henrique Cruz 103442
Diogo Borges 102954
Piotr Bartczak 130327

October 20, 2025

Contents

1 Introduction 2
2 Exercise 1 3
3 Exercise 2 6
4 Exercise 3 9
5 Exercise 4 11
6 Exercise 6 17
7 Exercise 7 20
8 Project Information 23

1 Introduction

This laboratory work focuses on the implementation and analysis of basic audio processing and coding techniques using **C++** and the **libsndfile** library. The main objective was to explore key principles of **digital audio representation**, **quantization**, and **compression**.

2 Exercise 1

In this exercise, the original `WAVHist` program was extended to provide additional histogram functionalities and to support coarser histogram binning.

Objectives

The main goals of this exercise were:

- To compute the histograms of the **MID** and **SIDE** channels for stereo audio signals:
  - MID channel: $(L + R)/2$
  - SIDE channel: $(L - R)/2$
- To allow the use of **coarser bins**, where each bin groups together multiple consecutive sample values (e.g. 2, 4, 8, ..., 2^k values).

Implementation Details

The class `WAVHist` was modified as follows:

1. Added two additional maps:
   - `mid_counts` — stores the histogram of the MID channel.
   - `side_counts` — stores the histogram of the SIDE channel.
2. Introduced a `bin_size` parameter (default value 1) to control histogram granularity. A helper function `quantize()` was implemented to group sample values according to the selected bin size.
3. Added two new methods:
   - `updateMid()` — computes and updates the histogram for the MID channel using $(L + R)/2$.
   - `updateSide()` — computes and updates the histogram for the SIDE channel using $(L - R)/2$.
4. Added corresponding output methods:
   - `dumpMid()` and `dumpSide()` — print the computed histograms to standard output.

5. Modified the `main()` function to accept the following command-line syntax:

   ```
   wav_hist <input_file> <channel|mid|side> [bin_size]
   ```

   where:

   - `channel` can be `0`, `1`, `mid`, or `side`;
   - `bin_size` is optional and specifies the width of each histogram bin.

Example Usage

```
# Left channel histogram
./wav_hist input.wav 0

# MID channel histogram with bin size 8
./wav_hist input.wav mid 8

# SIDE channel histogram
./wav_hist input.wav side
```

Results

The modified program correctly produces histograms for individual channels as well as for the MID and SIDE representations. When a larger bin size is used, the resulting histogram becomes smoother, as several neighboring sample values are grouped together.

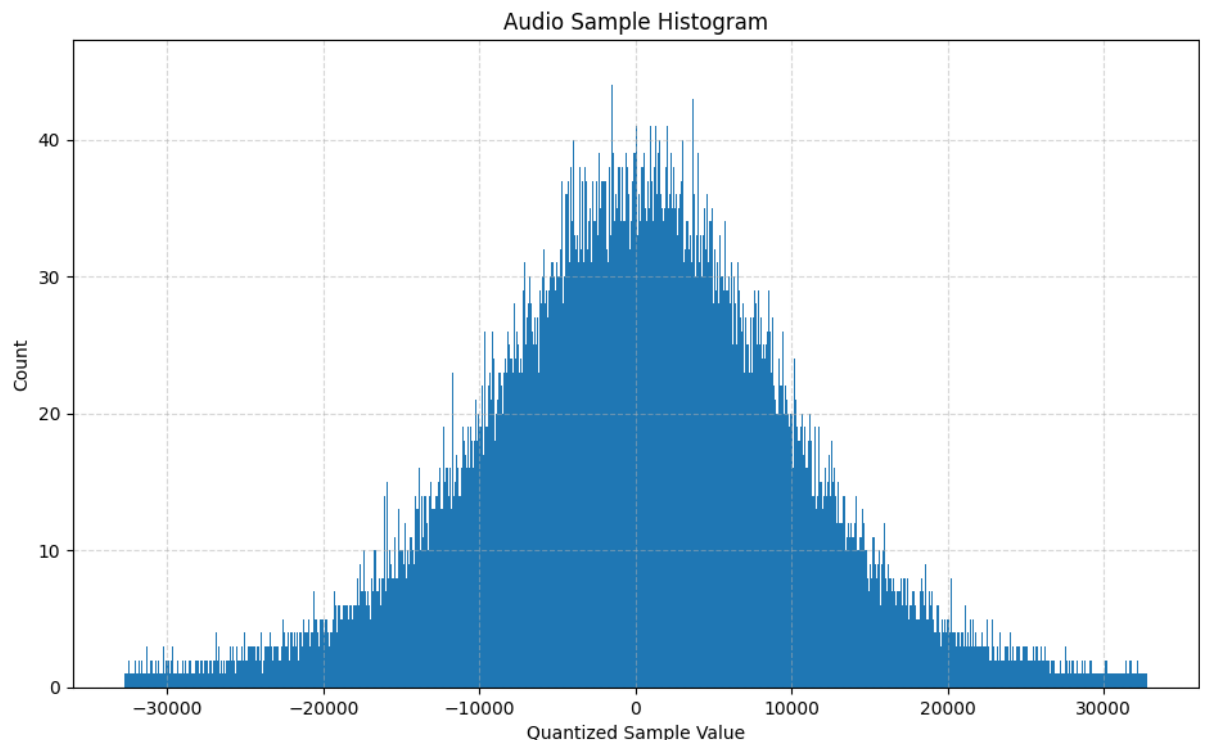

Figure 1: Histogram of a mono audio signal using a bin size of 1. Each sample value has its own bin, resulting in a highly detailed amplitude distribution.

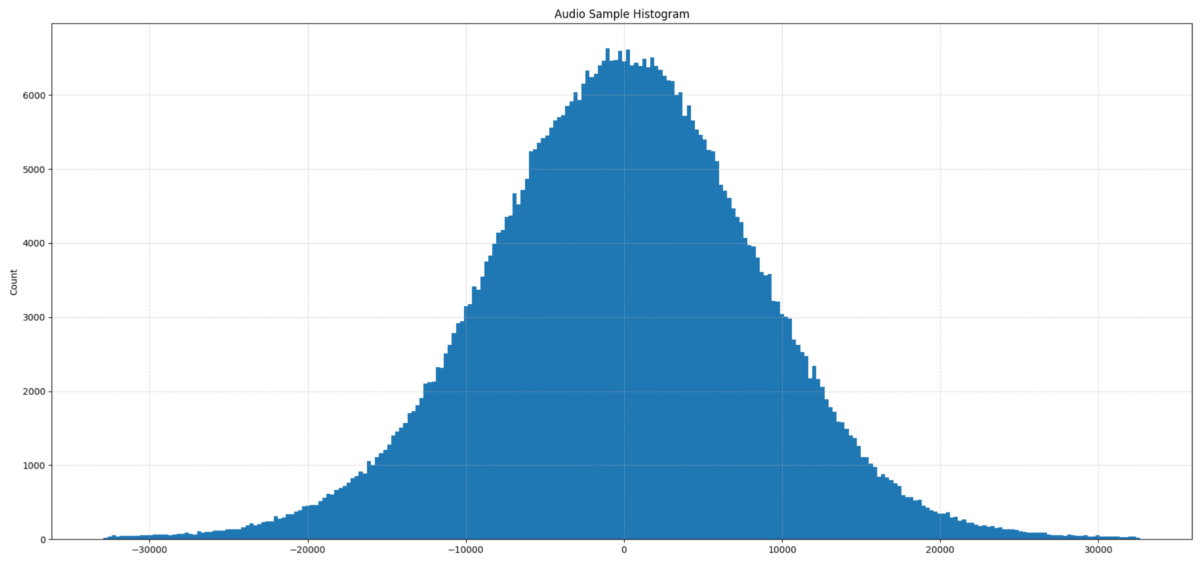

Figure 2: Histogram of the same mono audio signal using a bin size of 256. Coarser binning groups multiple sample values together, producing a smoother overall distribution.

3 Exercise 2

Objective

Quantize PCM samples to a reduced effective bit depth (e.g., 8-bit, 4-bit), measure processing time, and observe the effects on file size (container-level) and on the signal for later comparison.

Procedure

All commands are run from `ic-proj1/sndfile-example/test` after building the tools in `ic-proj1/sndfile-example/src`.

1. Generate quantized versions: 8-bit and 4-bit.
2. Measure wall time for each run.
3. Inspect resulting file sizes (container remains 16-bit PCM, so sizes are expected to be unchanged).

Commands

```
# Quantize to 8-bit and 4-bit with timing
/usr/bin/time -f 'time=%E' ../../sndfile-example/bin/wav_quant -b 8 sample.wav q8.wav
/usr/bin/time -f 'time=%E' ../../sndfile-example/bin/wav_quant -b 4 sample.wav q4.wav

# Another file (optional)
/usr/bin/time -f 'time=%E' ../../sndfile-example/bin/wav_quant -b 8 echo.wav echo_q8.

# File sizes (container-level)
ls -lh *.wav
```

Results

Processing time (wall-clock). Typical observations:

- sample.wav → 8-bit: 0.01s; 4-bit: ≤0.01s
- echo.wav → 8-bit: ≤0.01s

Sizes and container-level ratios. Sizes remained approximately constant:

- sample.wav: 2.1MiB; q8.wav: 2.1MiB; q4.wav: 2.1MiB
- echo.wav: 2.1MiB; echo_q8.wav: 2.1MiB

Thus, the container-level “compression ratio” is 1.0.

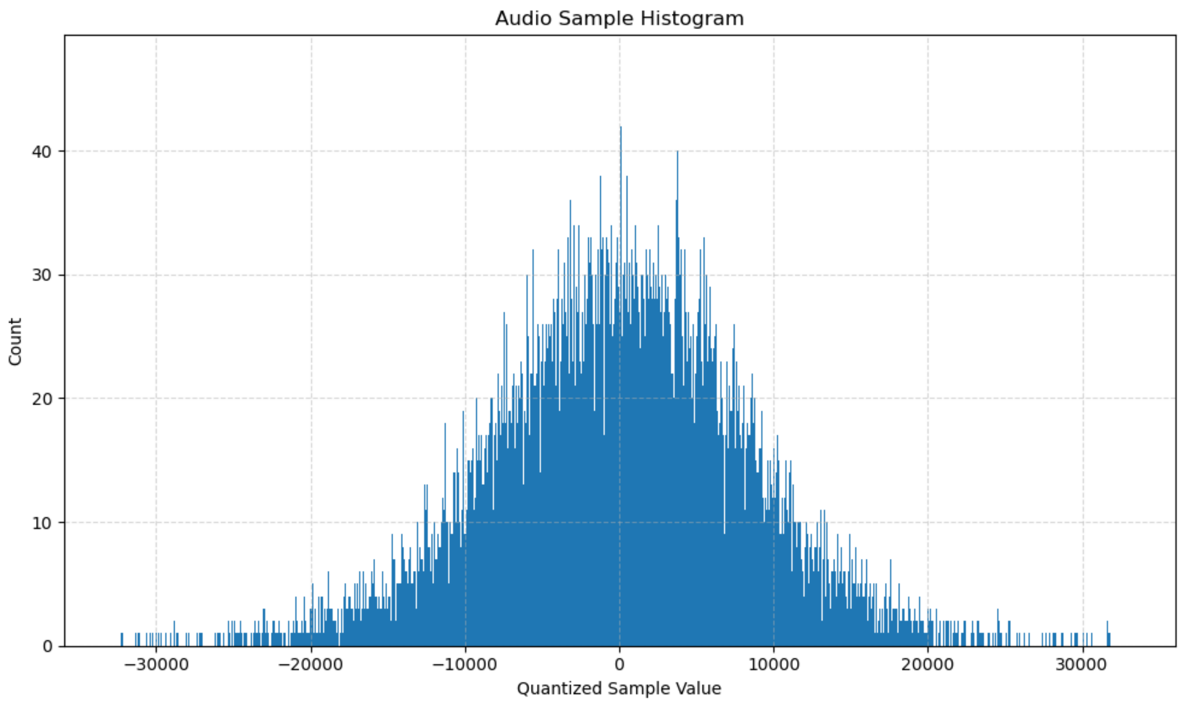

Figure 3: Histogram of the sample.wav file using channel mid and bin size of 1.

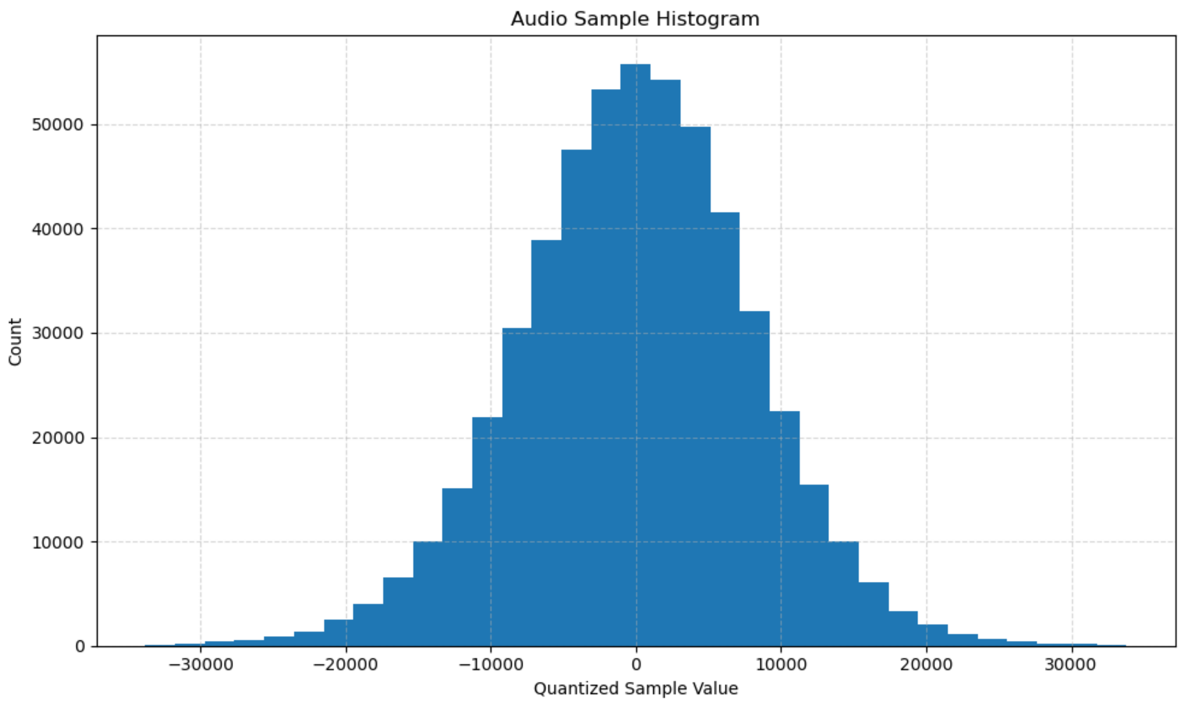

Figure 4: Histogram of the result file after using the wav_quant tool with a target bit of 4 using channel mid and bin size of 1.

Discussion

Uniform scalar quantization with power-of-two steps bounds the per-sample error by half the step size. Lower effective bit depth increases the quantization noise power and will degrade SNR when compared to the original.

4 Exercise 3

Objective

Compare a processed/quantized audio file to its original and report objective error metrics per channel and on the MID signal: Mean Squared Error ("L2 (MSE)"), maximum absolute error (L_∞), and Signal-to-Noise Ratio (SNR).

Procedure

After producing quantized files (with wav_cmp, for example), run wav_cmp for each pair (original vs. processed). The tool compares up to the shortest duration and requires both files to be PCM_16 with matching sample rate and channel count.

Commands

```
# Compare quantized outputs to originals
../../sndfile-example/bin/wav_cmp sample.wav q8.wav
../../sndfile-example/bin/wav_cmp sample.wav q4.wav
../../sndfile-example/bin/wav_cmp echo.wav echo_q8.wav
```

Results

sample.wav vs q8.wav

- Ch 0: MSE ≈ 5453.61, $L_\infty = 128$, SNR ≈ 41.32 dB
- Ch 1: MSE ≈ 5455.16, $L_\infty = 128$, SNR ≈ 41.34 dB
- MID: MSE ≈ 2721.07, $L_\infty = 128$, SNR ≈ 43.75 dB

sample.wav vs q4.wav

- Ch 0: MSE $\approx 1.40 \times 10^6$, $L_\infty = 2048$, SNR ≈ 17.22 dB
- Ch 1: MSE $\approx 1.40 \times 10^6$, $L_\infty = 2048$, SNR ≈ 17.26 dB
- MID: MSE $\approx 6.99 \times 10^5$, $L_\infty = 2046$, SNR ≈ 19.65 dB

echo.wav vs echo_q8.wav

- Ch 0: MSE ≈ 5452.91, $L_\infty = 128$, SNR ≈ 42.25 dB
- Ch 1: MSE ≈ 5461.36, $L_\infty = 128$, SNR ≈ 42.32 dB
- MID: MSE ≈ 2727.69, $L_\infty = 128$, SNR ≈ 44.69 dB

Discussion

As expected, stronger quantization (4-bit) exhibits much larger MSE and lower SNR than the 8-bit case. MID results (averaging L and R) often show slightly higher SNR due to partial cancellation/averaging of uncorrelated noise between channels.

5 Exercise 4

This exercise consisted of implementing a program capable of applying several audio effects to a WAV file, such as echo, multiple echoes, amplitude modulation (tremolo), and time-varying delay (vibrato). The program also computes and outputs the histogram of the audio samples before and after the effect is applied, allowing for a visual analysis of the impact of each transformation. The user can specify the histogram bin size directly from the command line.

Objectives

The goal was to design and implement a program capable of producing a variety of audio effects:

- A single echo
- Multiple echoes
- Amplitude modulation
- Time-varying delay

Each effect operates directly on the waveform samples stored in 16-bit PCM format and outputs the processed signal to a new WAV file.

Implementation Details

The program reads the entire audio file into memory, processes the samples according to the selected effect, and writes the result to an output file. The general command-line syntax is as follows:

```
wav_effects <effect> <input.wav> <output.wav> [parameters...]
```

The supported effects and their corresponding parameters are:

- **Echo:**

  ```
  echo <delay_sec> <decay>
  ```

  Introduces a delayed and attenuated copy of the original signal, producing a single echo. The delay is defined in seconds and the decay factor (typically between 0 and 1) controls the echo's amplitude.

- **Multiple Echoes:**

  ```
  multiecho <delay_sec> <decay> <repeats>
  ```

Generates a series of echoes by repeatedly applying the echo effect. Each subsequent echo is both delayed and decayed relative to the previous one.

- **Amplitude Modulation (Tremolo):**

  ```
  tremolo <freq_Hz> <depth>
  ```

  Modulates the amplitude of the signal at a given frequency (Hz), creating a tremolo effect. The depth parameter controls the modulation intensity, where values near 1 produce stronger amplitude variation.

- **Time-Varying Delay (Vibrato):**

  ```
  vib <max_delay_sec> <freq_Hz>
  ```

  Applies a periodic delay variation to simulate a vibrato effect. The maximum delay (in seconds) determines the depth of the pitch modulation, and the modulation frequency controls the rate of variation.

Algorithm Overview

Each effect was implemented as a separate function:

`applyEcho()` Creates a delayed version of the signal and mixes it with the original using the specified decay factor.

$$y[n] = x[n] + \alpha \cdot x[n - D]$$

`applyMultiEcho()` Calls `applyEcho()` multiple times with increasing delays and exponentially decreasing amplitudes.

`applyAmplitudeMod()` Modulates each sample by a sinusoidal function of the form:

$$y(t) = x(t) \cdot [1 + d \sin(2\pi f t)]$$

where d is the depth and f is the modulation frequency.

`applyTimeVaryingDelay()` Computes a sinusoidal variation of the delay in samples:

$$\Delta(t) = D_{\max} (0.5 + 0.5 \sin(2\pi f t))$$

and retrieves the delayed samples accordingly to simulate pitch variation.

All functions ensure sample values remain within the valid range $[-32768, 32767]$ to prevent clipping.

Example Usage

```
# Single echo with 0.5 s delay, 0.8 decay, and 128-bin histogram
./bin/wav_effects echo ./test/sample.wav ./data/echo.wav 0.5 0.8 128

# Multiple echoes (3 repetitions)
./bin/wav_effects multiecho ./test/sample.wav ./data/multiecho.wav 0.2 0.6 3 128

# Tremolo with 5 Hz modulation frequency and 0.7 depth
./bin/wav_effects tremolo ./test/sample.wav ./data/tremolo.wav 5 0.7 128

# Vibrato with maximum delay of 0.005 s and modulation frequency of 6 Hz
./bin/wav_effects vib ./test/sample.wav ./data/vibrato.wav 0.005 6 128
```

The value 128 corresponds to the bin size.

Results and Discussion

The resulting audio files demonstrate clear differences between effects:

- The **echo** effect produces a single distinct repetition of the sound.
- The **multi-echo** creates a decaying reverberation effect, similar to sound reflections in a room.
- The **tremolo** introduces periodic amplitude fluctuations perceived as rhythmic volume changes.
- The **vibrato** causes a periodic variation in pitch.

The following histograms illustrate the sample amplitude distribution before and after applying the effects.

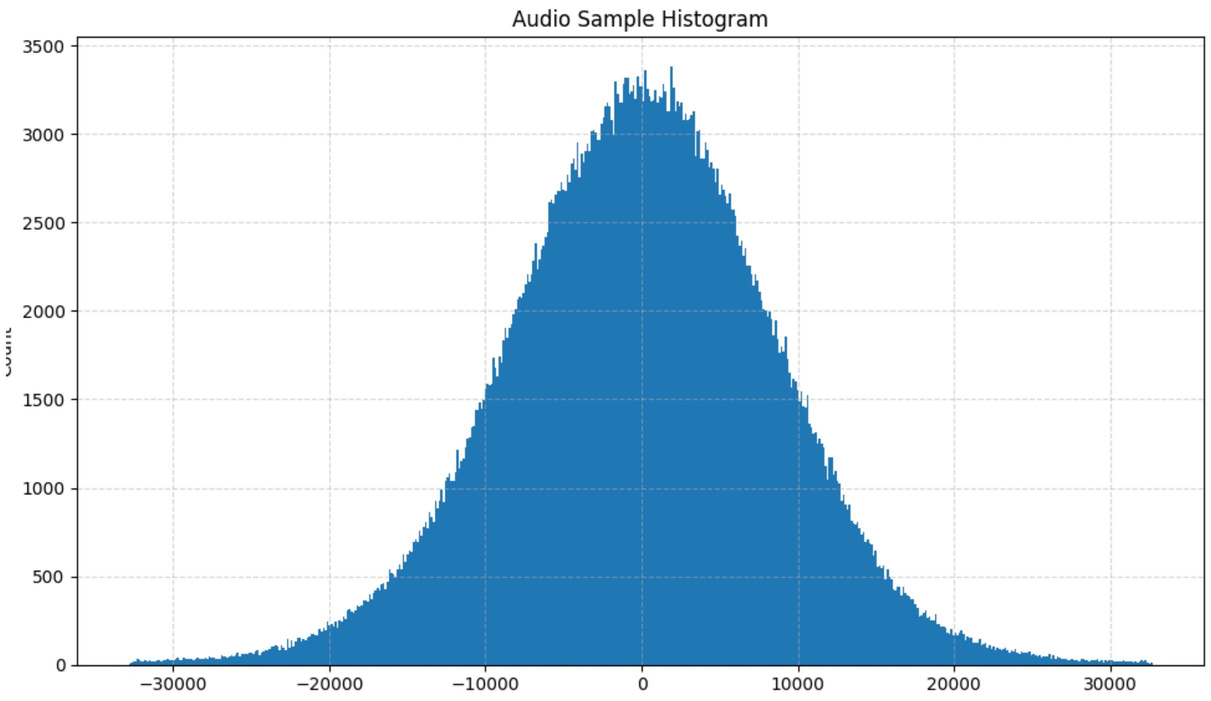

Figure 5: Histogram of the original audio signal (before any effect), 128 bins.

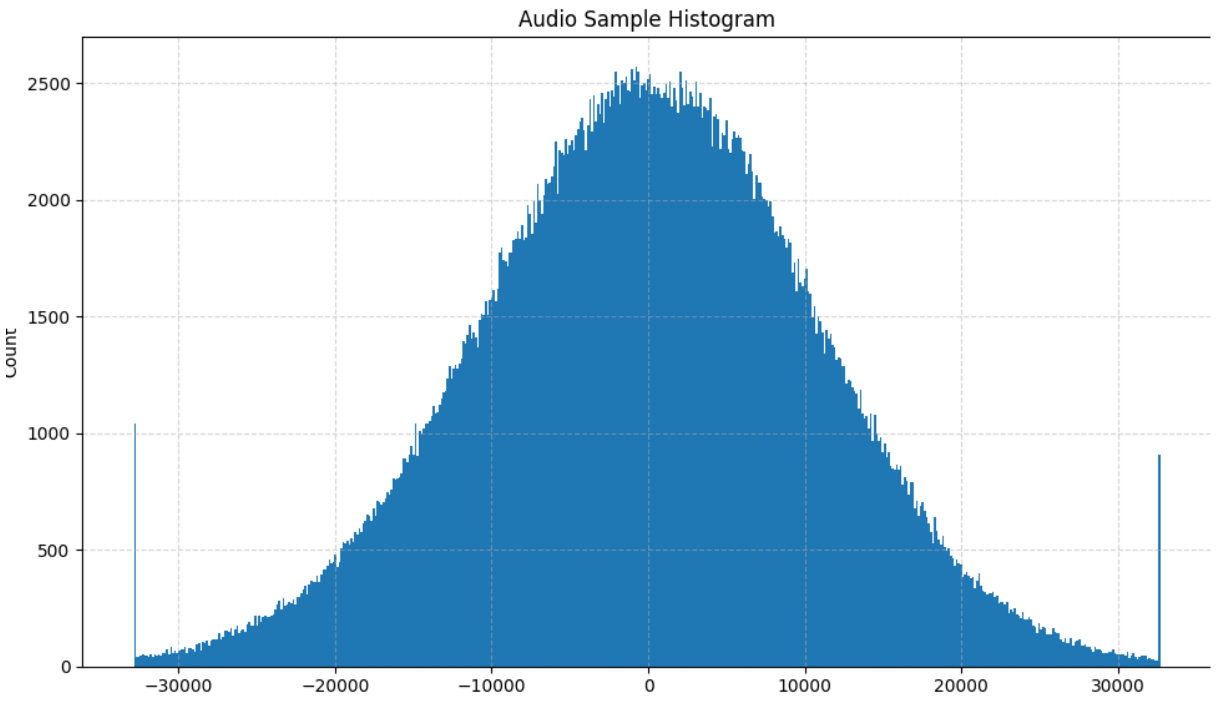

Figure 6: Histogram of the audio signal after applying the echo effect (0.5 s delay, 0.8 decay), 128 bins.

When comparing the histograms, it is possible to observe a noticeable change in the shape of the amplitude distribution after the echo effect is applied. Before the effect (Figure 5), the histogram is centered around zero where most samples have small amplitudes.

After applying the echo (Figure 6), the histogram becomes slightly wider and less peaked. This happens because the echo introduces delayed copies of the original signal, which overlap with the main waveform. As a result, the amplitude values at certain points increase when the original and delayed signals add up. This produces a broader spread in the histogram and sometimes small spikes at the extremes, reflecting higher amplitude occurrences.

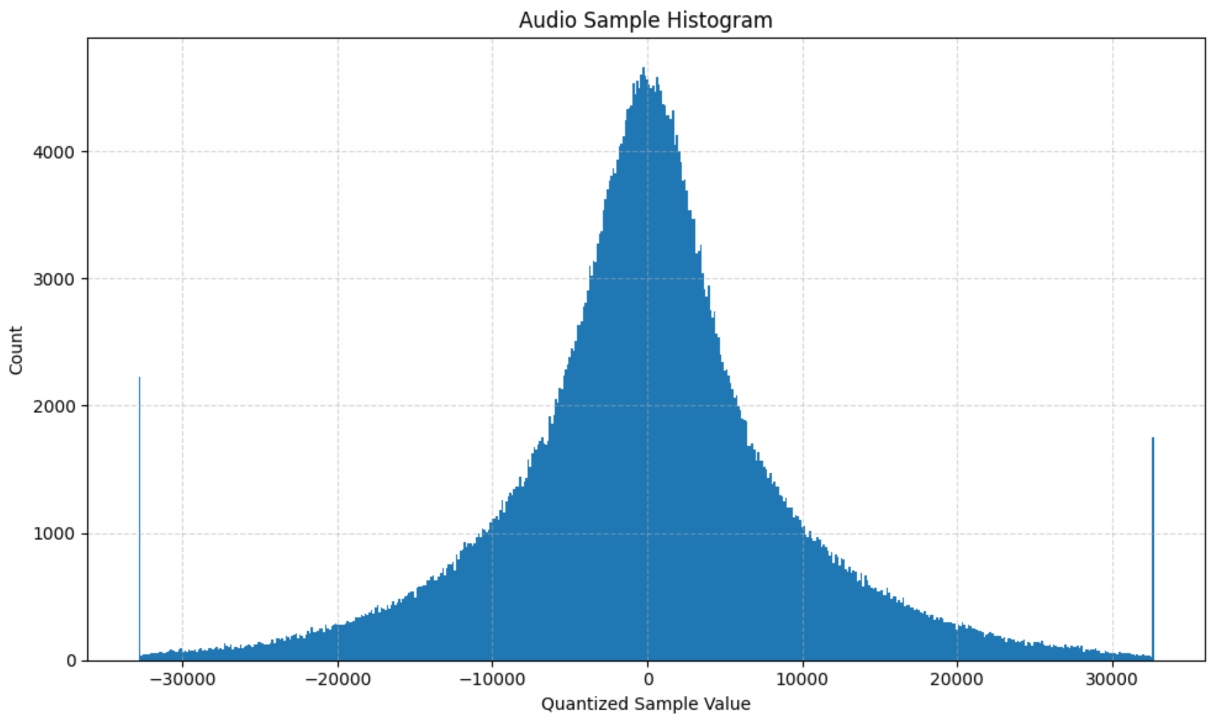

Figure 7: Histogram of the audio signal after applying the tremolo effect (20 Hz frequency, 0.7 depth), 128 bins.

For the **tremolo** effect (Figure 7), the histogram becomes much thinner compared to the original distribution. This occurs because the periodic amplitude modulation introduces more low-amplitude samples and reduces the occurrence of mid-range values, resulting in a narrower distribution.

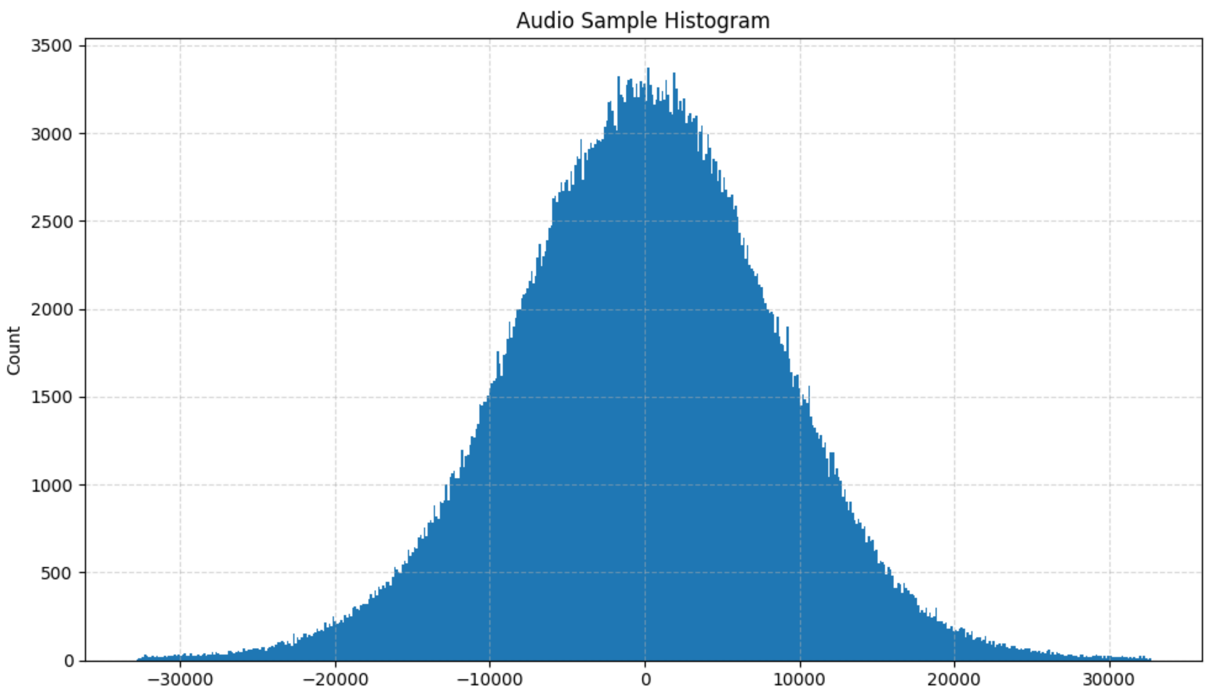

Figure 8: Histogram of the audio signal after applying the vibrato effect (0.005 s max delay, 20 Hz frequency), 128 bins.

For the **vibrato** effect (Figure 8), the histogram becomes slightly wider than the original. This is because the periodic pitch variation shifts sample amplitudes up and down, spreading the distribution slightly while maintaining overall its original shape.

6 Exercise 6

Objective

The aim of this task was to implement a simple audio codec using uniform quantization and bitstream packing. The encoder (wav_quant_enc) compresses quantized samples into a compact binary format, while the decoder (wav_quant_dec) restores them into a standard WAV file. The goal was to demonstrate efficient bit-level data representation and reconstruction of quantized audio signals.

Procedure

Executing the program is practically running 2 commands, first:

```
wav_quant_enc -b <bits> <input.wav> <output.qnt>
```

to reduce the number of bits per audio sample using uniform bit quantization, and then pack the result into a binary format, and then running:

```
wav_quant_dec <input.qnt> <output.wav>
```

to recover the playable WAV format from the binary file.

Results and Discussion

The histogram after 4-bit quantization is sparse because the signal is limited to only 16 possible amplitude levels ($2^4 = 16$). Many original values are mapped to the same quantized level, so only a few discrete bars appear. The gaps between bars show the loss of intermediate values. This confirms correct quantization but also illustrates the loss of detail and increased quantization noise.

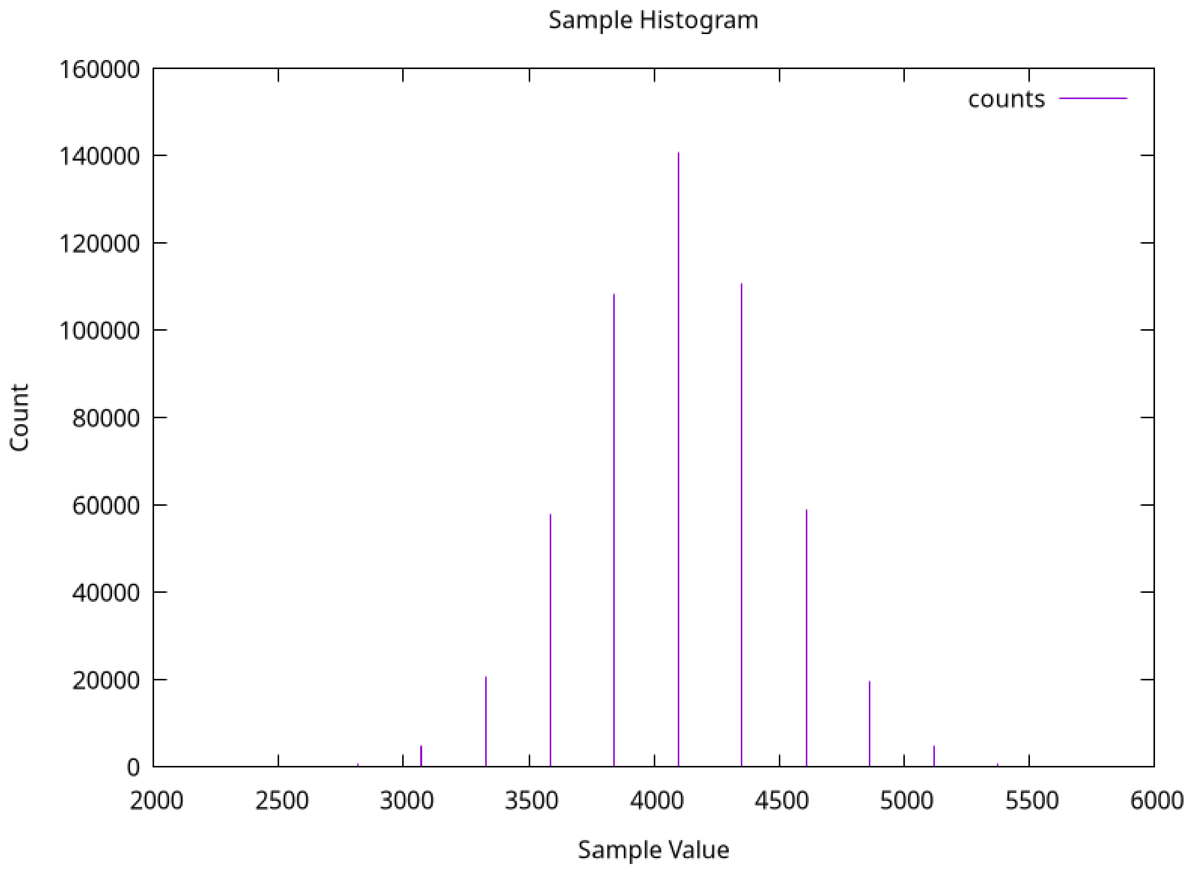

Figure 9: Histogram of the audio signal after packing and unpacking the results of performing the uniform quantization, 4-bit samples

In the 8-bit quantized signal, the histogram becomes much denser compared to the 4-bit case, since there are 256 possible quantization levels. The distribution appears almost continuous, preserving the general shape of the original waveform.

The very high bar in the center likely results from many samples having values close to the mean (around zero or mid-amplitude) before quantization. During quantization, these values are rounded to the same central level, producing a noticeable peak. This indicates that the audio signal contains a large number of samples with small amplitudes, which is typical for natural sounds.

Overall, the 8-bit quantization provides better resolution and less distortion than the 4-bit case, while still reducing data size compared to the original high-bit-depth signal.

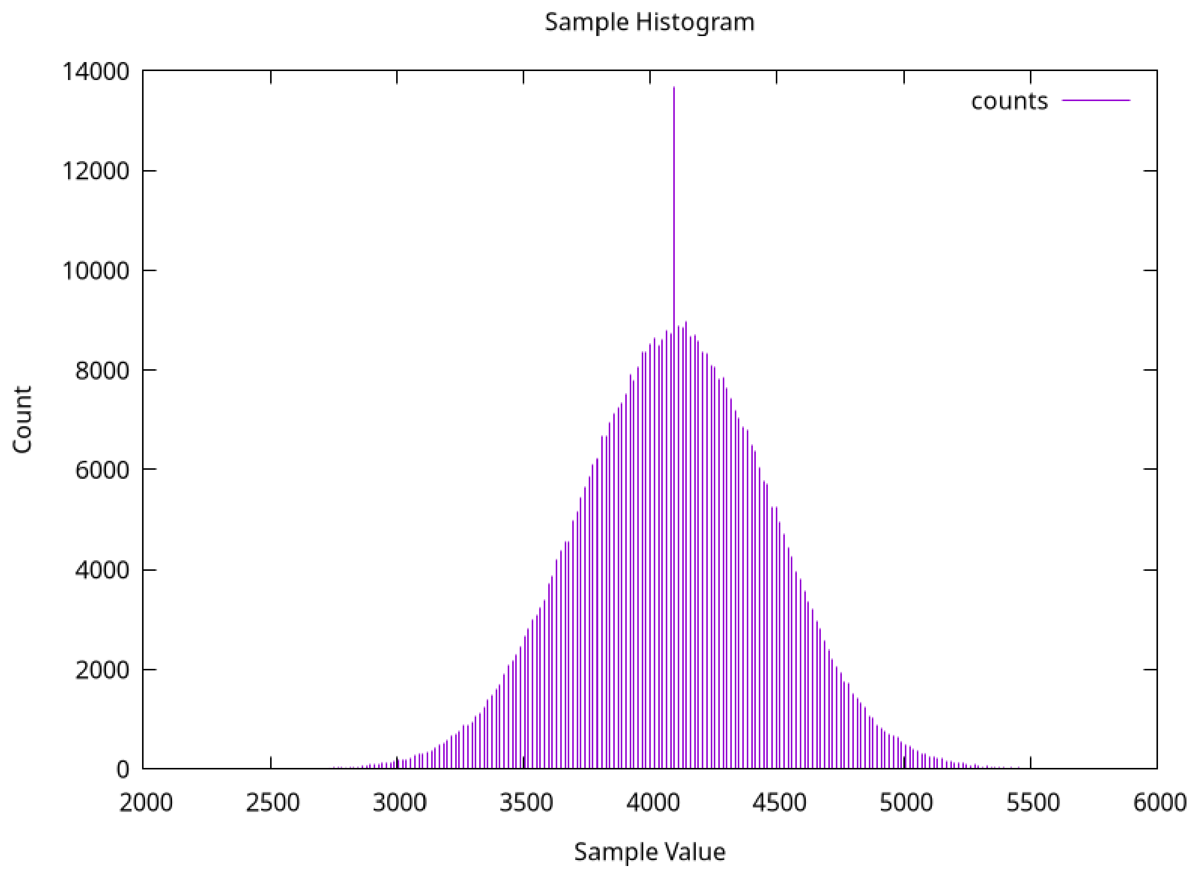

Figure 10: Histogram of the audio signal after packing and unpacking the results of performing the uniform quantization, 8-bit samples

7 Exercise 7

Objective

Design and evaluate a lossy mono audio codec based on block Discrete Cosine Transform (DCT) with uniform quantization and a compact bitstream stored via `BitStream`. The decoder must reconstruct audio using only the binary file.

Procedure

The pipeline consists of: (1) optional downmix to mono, (2) encoding with DCT and quantization, (3) decoding with inverse DCT, and (4) objective evaluation against the original mono signal.

1. Downmix to mono when necessary:

   ```
   ../../sndfile-example/bin/wav_to_mono sample.wav sample_mono.wav
   ```

2. Encode (block DCT + quantization) to a compact bitstream:

   ```
   ../../sndfile-example/bin/dct_enc [ -v ] \
     -bs 1024 -k 256 -b 12 -q 8.0 \
     sample_mono.wav result.dct
   ```

   where: `-bs` is block size (N), `-k` the number of low-frequency coefficients kept per block (K), `-b` bits per coefficient, and `-q` the uniform quantization step.

3. Decode the bitstream to WAV (inverse DCT):

   ```
   ../../sndfile-example/bin/dct_dec [ -v ] result.dct result_dec.wav
   ```

4. Evaluate distortion with the comparison tool:

   ```
   ../../sndfile-example/bin/wav_cmp sample_mono.wav result_dec.wav
   ```

Bitstream format (encoder output)

The binary file begins with a small header containing magic (`"DCT1"`), version, samplerate, total frames, block size (N), kept coefficients (K), bits per coefficient, and quantization step. Then, for each block, only the first K quantized coefficients are written (remaining $N-K$ are implicitly zero). This enables decoding without side information beyond the file itself.

Commands

```
# End-to-end example (mono pipeline)
../../sndfile-example/bin/wav_to_mono sample.wav sample_mono.wav
../../sndfile-example/bin/dct_enc -v -bs 1024 -k 256 -b 12 -q 8.0
sample_mono.wav result.dctls -lh result.dct  # typically << WAV size
../../sndfile-example/bin/dct_dec -v result.dct result_dec.wav
../../sndfile-example/bin/wav_cmp sample_mono.wav result_dec.wav
```

Results

On the provided `sample.wav` downmixed to mono, using `-bs 1024`, `-k 256`, `-b 12`, `-q 8.0` we obtained the following indicative results:

- Bitstream size: `result.dct` $\approx$ `194 KiB`, versus `sample_mono.wav` $\approx$ `1.1 MiB`.
- Distortion (Channel 0): MSE $\approx 5.76 \times 10^5$, $L_\infty \approx 15550$, SNR ≈ 20.5 dB.

This demonstrates a substantial reduction of storage compared to PCM WAV, at the cost of moderate distortion.

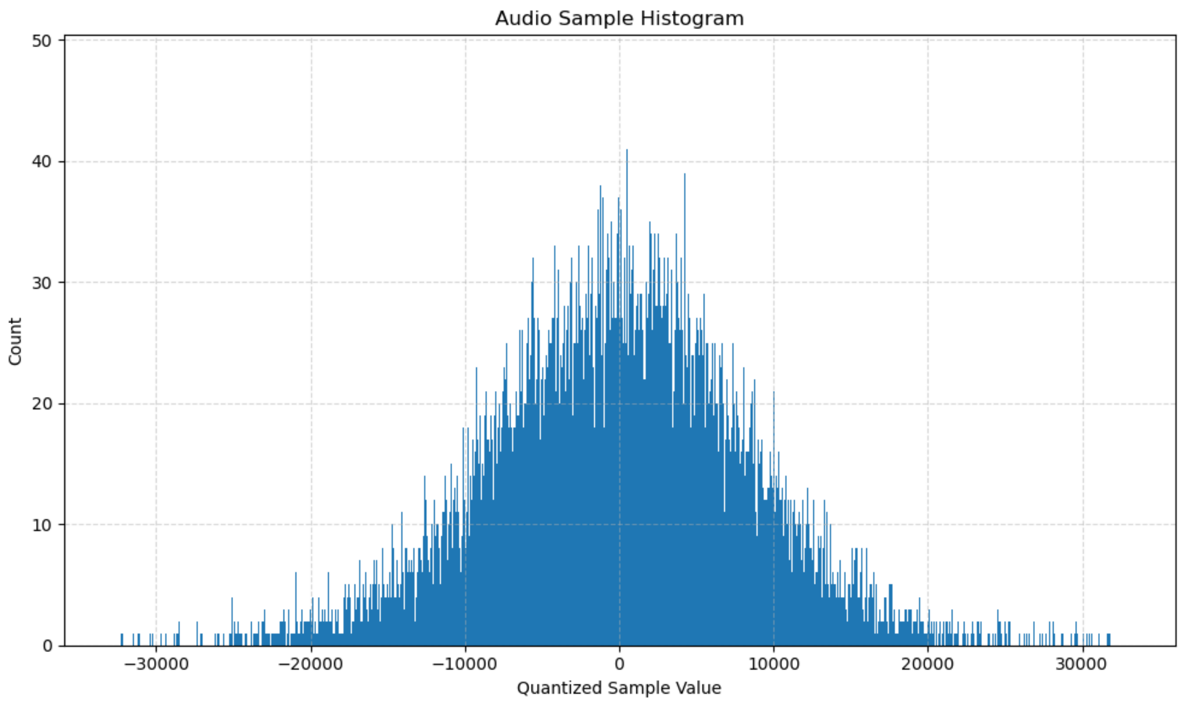

Figure 11: Histogram of the sample_mono.wav file using channel mid and bin size of 1.

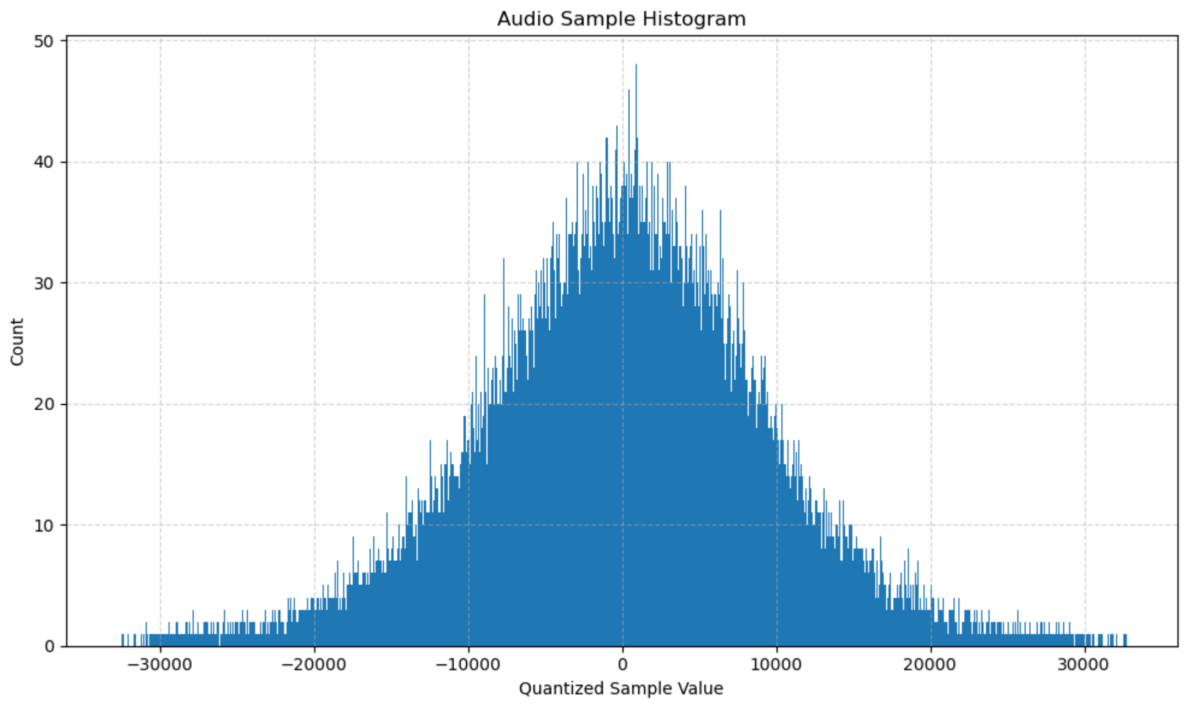

Figure 12: Histogram of the result file after encoding and decoding the sample_mono.wav file through the dct_enc and dct_dec tools.

Discussion

Quality/compression can be tuned via K, q, b, and N:

- Increase K (keep more low-frequency DCT coefficients) to preserve more detail at higher rate.
- Decrease q (finer quantization) to reduce quantization noise at the same bit allocation (if b is sufficient to avoid clipping).
- Increase b (bits per coefficient) to avoid clipping of large coefficients when needed.
- Adjust N (block size) for a balance between spectral efficiency (larger N) and temporal artifacts (smaller N).

In practice, settings such as `-bs 1024 -k 384 -b 14 -q 3.0` can enhance SNR, while `-k 256 -b 12 -q 2.0` offers a lower-rate alternative with improved noise characteristics compared to a coarse step.

8 Project Information

Division of Work

All group members contributed equally to the development of the project.

Repository

The complete source code, including all modules, examples, and test files, is available at the following repository:

`https://github.com/hmecruz/ic-proj1`

How to Build and Test

In order to build and test the project there is a README file in the repository.