Departamento de Eletrónica, Telecomunicações e Informática
da Universidade de Aveiro

Information and Coding
Lab work nº. 1

Diogo Silva (108212)
Martim Carvalho (108749)
Miguel Marques (103162)

October 19, 2025

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This report details the laboratory work performed as part of the Information and Coding Lab, focusing on the practical application of audio signal processing and data compression techniques using the C programming language. The tasks emphasize proficiency in manipulating WAV audio files, employing libraries such as libsndfile for diverse file I/O operations, and demonstrate core concepts like histogram analysis and quantization.

A significant objective of this lab is the exploration of uniform scalar quantization, where the number of bits used to represent each audio sample is reduced, directly affecting audio fidelity. The work further involves implementing error metrics (mean squared error, maximum absolute error, and signal-to-noise ratio) to rigorously evaluate the impact of quantization and compression on signal integrity. Audio effects, such as single or multiple echoes and amplitude modulation, introduce creative manipulation techniques that reinforce understanding of digital audio processing.

The BitStream class serves as a powerful tool for implementing encoding and decoding algorithms for compressed representations, facilitating both near-lossless and lossy codecs for mono files based on the Discrete Cosine Transform (DCT). We are required to extend these foundational tools to achieve effective compression while maintaining acceptable audio quality, using well-defined quantization and block processing strategies. The report does not present the code in detail but focuses instead on the processes, choices, and outcomes, such as processing times, compression ratios, and measured errors, all demonstrated across several audio files.

In summary, the laboratory work provides valuable practical insights into audio engineering, codec design, and data evaluation, framed within the context of real-world applications and challenges that arise during the implementation and assessment of multimedia processing algorithms.

Chapter 2

Exercises

2.1 Part-I

The first stage focuses on extending the functionality of the **WAVHist** class to compute histograms of audio signals, including MID and SIDE channels as well as coarser grouped bins.

Tools such as `wav_quant` and `wav_cmp` are developed to perform uniform quantization of samples and to compare original versus processed files using error metrics like mean squared error, maximum absolute error, and signal-to-noise ratio.

Additionally, `wav_effects` introduces creative modifications such as single or multiple echoes, enabling practical experimentation with audio transformations.

2.1.1 Exercise 1

In order to address the requirements of the exercise and improve the robustness of the tool, several structural and functional modifications were introduced to the `wav_hist` implementation. These changes can be grouped into five main areas: histogram design, channel handling, configurable binning, output extension, and command-line interface (CLI) corrections.

Code

- **MID/SIDE Histogram Integration:** Additional histograms for MID and SIDE signals were added by defining `midCounts` and `sideCounts` members. MID and SIDE values are computed for stereo input (`numChannels == 2`) using integer arithmetic and binned consistently with the existing histogram logic.

$$MID = \frac{L+R}{2} \qquad SIDE = \frac{L-R}{2}$$

- **Channel Awareness:** Histogram storage now uses a vector of maps, one per channel, and records the channel count during object construction.

- **Configurable Binning:** A binning mechanism, using `binShift_` and `binKey`, maps 16-bit PCM values to bins of size 2^k, reducing sparsity.

- **Extended Output:** The `dump` function now exposes MID/SIDE data as well. Channel indexing is as shown:

Index	Channel
0	Left
1	Right
2	MID
3	SIDE

Table 2.1: Channel Indexing Scheme

- **CLI Bug Fixes:** Arguments are parsed as: input file (`argv[1]`), channel index (`argv[2]`), optional binShift (`argv[3]`). This ties CLI to the binning logic and fixes parsing errors.
- **Verification:** For stereo files, the total MID/SIDE counts match the channel counts. At extremes, MID sample counts should never exceed the minimum of left/right counts.

Usage

The code is run from the command line as follows:

```
./bin/wav_hist <input_file> <channel_index> [<binShift>]
```

Where:

- `<input_file>`: Path to the WAV audio file.
- `<channel_index>`: Channel to process (0: Left, 1: Right, 2: MID, 3: SIDE).
- `[<binShift>]`: (Optional) Histogram bin width as 2^k (default 0, i.e., no binning).

Example (MID channel with a bin grouping of $2^3 = 8$):

```
./bin/wav_hist audio.wav 2 3
```

Results

After running the program, the output histograms confirm correct integration of MID/SIDE and configurable binning. For example, the following figures show results for both original and binned sample distributions:

The figures illustrate that with appropriate binning, histogram sparsity is reduced, and that MID/SIDE channels are now available for analysis alongside originals. The tool correctly handles mono and stereo formats, validates input, and outputs data suitable for interpretation or further processing.

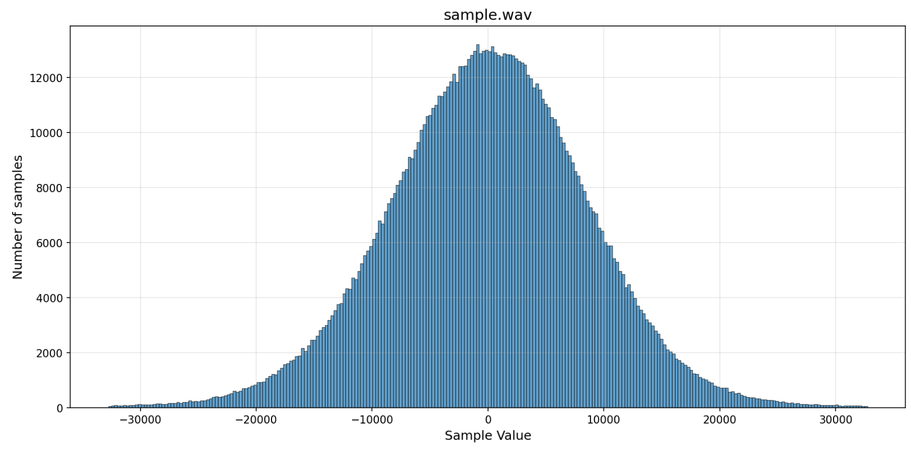


Figure 2.1: Original Sample Histogram

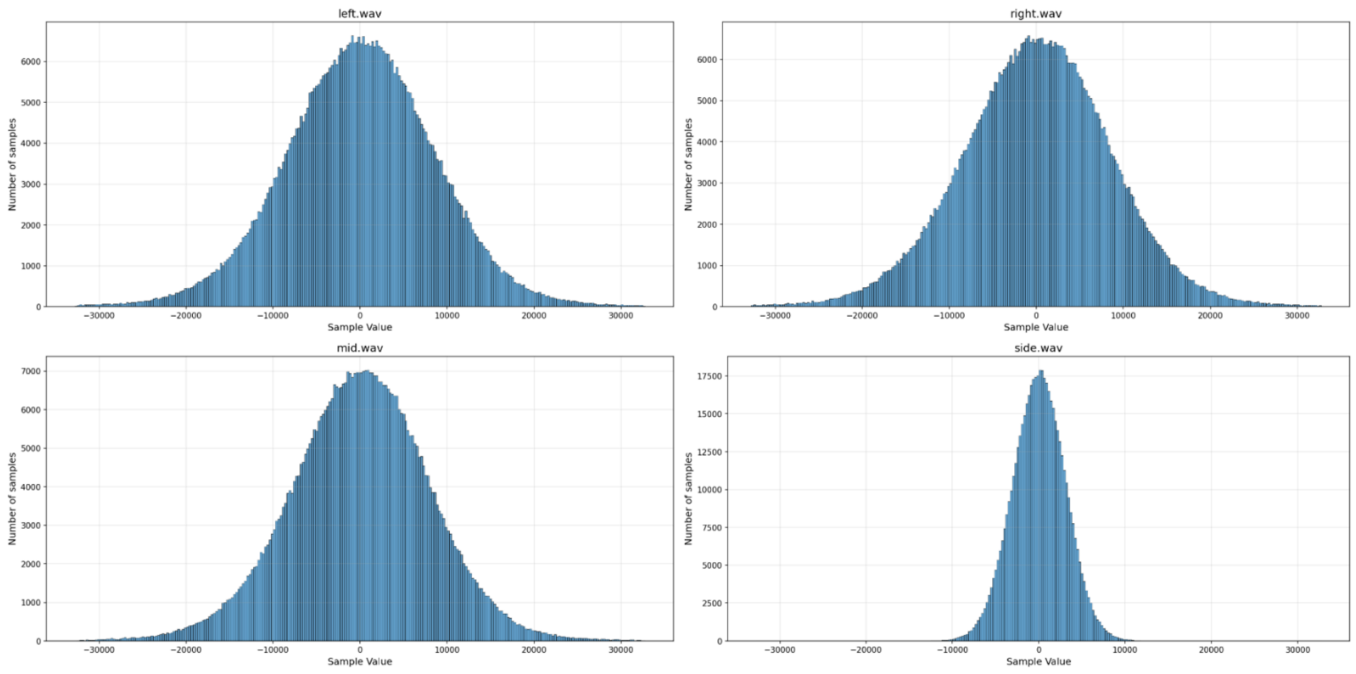


Figure 2.2: Left, Right, MID, SIDE Channel Histograms

2.1.2 Exercise 2

In this exercise, we were asked to develop a program called `wav_quant` with the goal of reducing the number of bits used to represent an audio file (Uniform Scalar Quantization). The resulting output must be presented in audio file format.

Code

In the developed scalar uniform quantization process, a private vector was created to store the samples after transformation, where each value read is modified through bit shifts that remove the excess precision and replace the cleared spaces with zeros to preserve the same bit size per sample. Afterwards, the quantized samples are used to generate the output audio file. Finally, the program ensures the validation of the number of bits provided by the user and, after fully reading the input file, applies quantization to each set of samples before producing the final result.

Usage

The code can be tested using the command below, where `input_file` is the name of the received file, `num_bits` is the number of bits to keep, and `output_file` is the name of the output file.

```
../bin/wav_quant <input_file> <num_bits> <output_file>
```

The command below saves the values generated by the program in the file output_histogramFile so that the histogram can be generated later.

```
../bin/wav_quant <input_file> <num_bits> <output_file>
      <output_histogramFile>
```

Results

After the above, we tested our code with the same `input_file` that uses 16 bits and different numbers of bits to represent the audio. In figures 2.14, we can observe the histograms obtained from these, and we can conclude that the lower the number of bits, the lower the quality of the signal obtained.

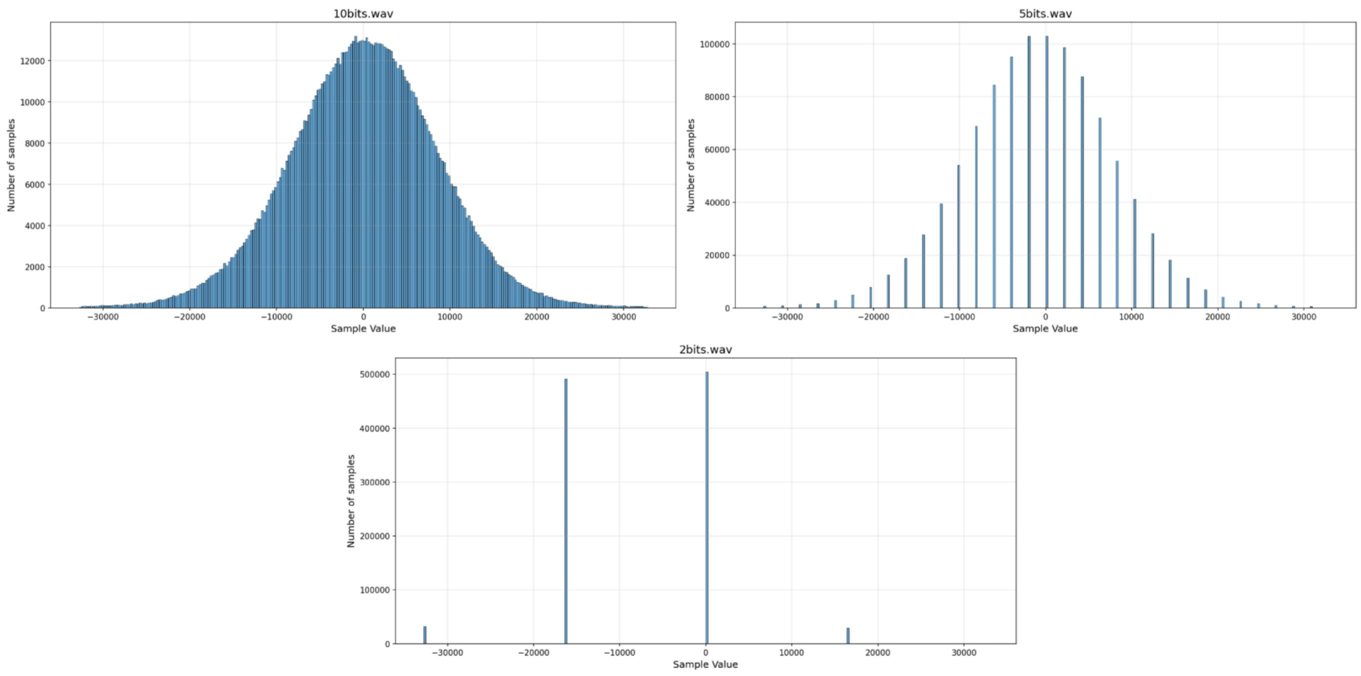


Figure 2.3: Histogram of 10, 5 and 2 bits respectively

The comparison between the original file and the files with 10 and 5 bits does not show any noticeable changes in Audacity, so figure 2.4 shows the comparison reduction between the original file and the 2 bit file.

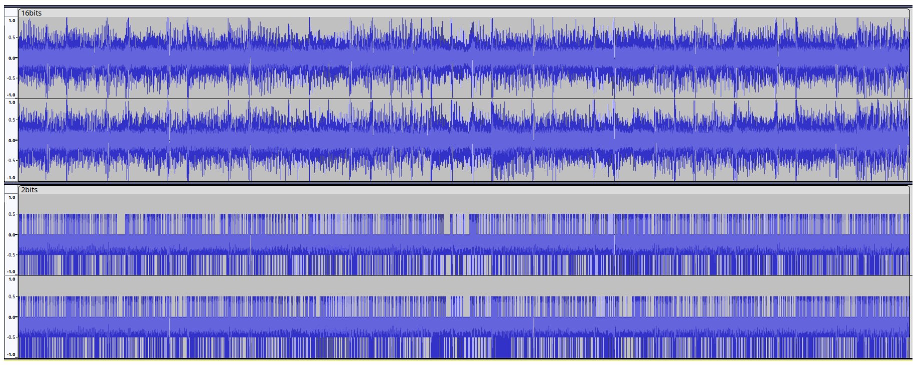


Figure 2.4: Comparison between 16 bits (up) and 2 bits (down)

2.1.3 Exercise 3

In the `wav_cmp` program, we compare each channel and the average of the channels between the original and modified audio files using three key metrics: average squared error (mean squared error), maximum absolute error per sample, and signal-to-noise ratio (SNR).

1. Average Squared Error (Mean Squared Error)

The **mean squared error (MSE)** quantifies the average squared difference between each sample in the original and the modified audio signal. It provides an overall measure of how much the signal has been distorted by processing steps such as quantization or compression. For discrete signals x_n (original) and y_n (processed), MSE is defined as:

$$\text{MSE} = \frac{1}{N}\sum_{n=1}^{N}(x_n - y_n)^2$$

where N is the total number of samples. A lower MSE reflects higher fidelity to the original signal and is widely used in signal processing to assess reconstruction quality.

2. Maximum Absolute Error Per Sample

The **maximum absolute error** (also called L_∞ norm) captures the largest difference between any pair of samples, highlighting extreme discrepancies which might produce perceptible artifacts.

$$\text{MaxAbsError} = \max_n |x_n - y_n|$$

This metric is essential to ensure that the transformation does not introduce unacceptable errors at any sample, which could affect perceived audio quality.

3. Signal-to-Noise Ratio (SNR)

The **signal-to-noise ratio (SNR)** measures the strength of the original signal relative to the introduced noise (error). It is reported in decibels (dB) and is computed as:

$$\mathrm{SNR} = 10 \log_{10} \left(\frac{\sum_{n=1}^{N} x_n^2}{\sum_{n=1}^{N} (x_n - y_n)^2} \right)$$

A higher SNR indicates that the signal has less distortion or noise relative to its overall strength, resulting in better perceptual quality.

Code

The codebase offers utilities for working with WAV audio files, primarily focusing on file copying and comparison. The file copy utility reads from an input WAV file and writes to an output WAV file with optional verbose statistics on frames, sample rate, and channels. The WAV comparison interface defines a statistics structure to calculate error metrics such as mean squared error, maximum absolute error, and signal-to-noise ratio per audio channel. The implementation aims to support precision-focused audio analysis and validation of PCM 16-bit WAV data. The tools prioritize error handling and format validation to ensure reliable audio file operations.

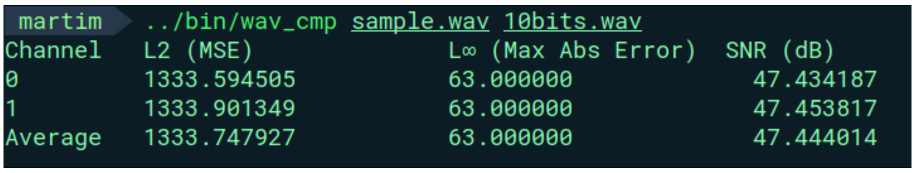
Usage

The code can be tested using the command below, where `original_file` is the name of the original file and `modified_file` is the name of the modified file.

```
    ../bin/wav_quant <original_file> <modified_file>
```

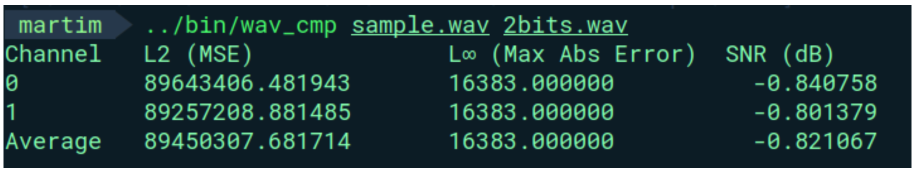
Results

After the explanation above we can conclude that the smaller the number of bits used in the WAV file, the greater the absolute error, the greater the mean square error between the modified file and its original version and also the lower the noise-to-noise ratio (SNR).



```
 martim  ../bin/wav_cmp sample.wav 10bits.wav
Channel    L2 (MSE)             L∞ (Max Abs Error)  SNR (dB)
0          1333.594505          63.000000             47.434187
1          1333.901349          63.000000             47.453817
Average    1333.747927          63.000000             47.444014
```

Figure 2.5: Comparison between original file and audio file with 10 bits



```
 martim  ../bin/wav_cmp sample.wav 2bits.wav
Channel    L2 (MSE)             L∞ (Max Abs Error)  SNR (dB)
0          89643406.481943      16383.000000          -0.840758
1          89257208.881485      16383.000000          -0.801379
Average    89450307.681714      16383.000000          -0.821067
```

Figure 2.6: Comparison between original file and audio file with 2 bits

2.1.4 Exercise 4

This module implements a set of digital audio effects applied directly to sampled audio data. Each effect manipulates the signal in a distinct way to achieve different auditory characteristics:

1. **Single Echo (`singleEcho`)**
   Introduces a delayed copy of the signal mixed back with the original (dry) signal. Produces a single echo with configurable delay time and decay factor.

2. **Multiple Echo (`multipleEcho`)**
   Extends the single echo by adding several repeated echoes. Each repetition is attenuated by the decay factor, simulating a reverberant environment. Balances the dry and wet signal using fixed mix ratios.

3. **Amplitude Modulation (`amplitudeModulation`)**
   Applies periodic modulation of the signal's amplitude using a low-frequency oscillator (LFO). The depth controls how strongly the amplitude fluctuates, while the modulation frequency sets the speed of variation. Produces a tremolo-like effect.

4. **Time-Varying Delay (`timeVaryingDelay`)**
   Introduces a dynamic delay that varies sinusoidally over time. Interpolates between delayed samples for smoother transitions. Creates a chorus- or flanger-like effect depending on parameters (base delay and modulation frequency).

5. **Bass Boosted (`bassBoosted`)**
   Enhances the low-frequency (bass) components of audio by applying a low-pass filter and amplifying those frequencies before mixing them back into the signal. This increases the power and depth of the bass, giving the audio a fuller and richer sound. It is commonly used to make the bass more prominent without distorting the overall signal.

In short:

- *Echo/Multiple Echo* simulate sound reflections.
- *Amplitude Modulation* alters loudness periodically (tremolo).
- *Time-Varying Delay* introduces a sweeping delay for modulation effects (chorus/flanger).
- *Bass Boosted* accentuates and amplifies low frequencies, making the bass in audio more prominent and powerful.

Code

The implemented code defines a modular framework for applying digital audio effects to time-domain sampled signals. Each effect is registered in a lookup table and can be invoked dynamically. Processing operates directly on the audio buffer (`Samples`), which is assumed to contain interleaved multi-channel data. The following algorithms are provided:

1. **Single Echo (`singleEcho`)** This effect implements a finite impulse response (FIR) delay line with a single delayed copy of the signal. The delay offset is computed as
$$N_d = \lfloor delaySec \cdot f\_s \rfloor \cdot C$$
where f_s is the sampling rate and C is the number of channels. The delayed sample is attenuated by a decay factor and summed with the direct signal, producing a single reflection comparable to early echoes in a reverberant environment.

2. **Multiple Echo (`multipleEcho`)** This effect generalizes the single echo by introducing repeated echoes. The output at time n is defined as
$$y[n] = \alpha x[n] + \sum_{r=1}^{R} \beta \cdot d^{r-1}\, y[n - rN_d]$$
where R is the number of repetitions, d the decay factor, and α, β the dry/wet mix coefficients. This recursive formulation approximates the reverberant tail of a room impulse response.

3. **Amplitude Modulation (`amplitudeModulation`)** This effect applies periodic modulation of the signal amplitude using a low-frequency oscillator (LFO). The instantaneous gain is given by
$$g(t) = (1 - D) + D \cdot \frac{\sin(2\pi f_m t) + 1}{2}$$
where D is the modulation depth and f_m the modulation frequency. The result is a tremolo effect, characterized by cyclic loudness variation without altering spectral content.

4. **Time-Varying Delay (`timeVaryingDelay`)** This effect implements a modulated delay line, where the delay varies sinusoidally over time, producing chorus- or flanger-like effects. The delay length is defined as
$$N_d(t) = N_0 + \Delta N \cdot \sin(2\pi f_m t)$$
where N_0 is the base delay in samples, ΔN is the modulation amplitude, and f_m the modulation frequency. Fractional delays are handled via linear interpolation between adjacent samples, ensuring smooth transitions without discontinuities.

5. **Bass Boosted(`bassBoosted`)** This effect enhances the low-frequency content of the signal, resulting in a pronounced bass or "bass boosted" sound.

   The effect is achieved by extracting the low-frequency components with a first-order low-pass filter and adding an amplified version of this bass signal back to the original audio. The process accentuates bass without significantly distorting higher frequencies, creating a deep, powerful low-end.

   The algorithm operates as follows:
$$b[n] = b[n-1] + \alpha(x[n] - b[n-1])$$
   where $\alpha = dt/RC + dt$ with $RC = 1/2\pi fc$, **fc** as the cutoff frequency (Hz) and $dt = 1/fs$ with **fs** as a sample rate.

The final output is then computed bu adding an amplified bass component to the original:

$$y[n] = x[n] + G.b[n]$$

where **G** is bassboost gain.

The Bass Boosted effect is widely used in modern audio production to emphasize depth and impact, particularly in genres like dance, hip-hop, and pop music.

Implementation Notes

- All functions are channel-aware, explicitly computing frame and channel indices.
- Processed samples are written to a separate output buffer and swapped at the end to avoid in-place overwriting.
- Sample values are saturated using a `clamp16` helper, ensuring results remain within the valid 16-bit PCM range.

Usage

The `wav_effects` program can be used through the following command. It requires the name of the input file (`original_file`), the name of the output file (`out_file`), and the effect to be applied (`wav_effect`).

```
    ../bin/wav_effects <original_file> <out_file> <wav_effect>
```

The available effects are:

- `none`
- `singleEcho`
- `multipleEcho`
- `amplitudeModulation`
- `timeVaryingDelay`
- `bassBoosted`

Results

To evaluate the implemented effects, several tests were performed using an original audio sample and applying each of the available transformations. The resulting signals were analyzed both in the time and frequency domains, using amplitude histograms and frequency spectrum visualizations obtained through *Audacity*.

The comparison between the original and processed signals allows us to verify how each effect modifies the waveform and its spectral characteristics. While the original sample presents a clean and stable amplitude distribution, each processed version introduces distinct alterations that are consistent with the expected theoretical behavior of the implemented algorithms.

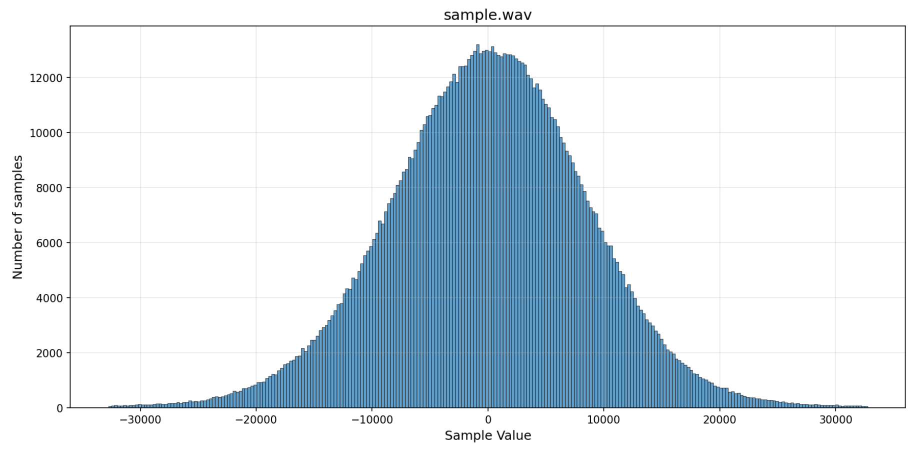


Figure 2.7: Histogram of Original Sample

- **Single Echo** - The `singleEcho` effect introduces a delayed version of the signal, producing a secondary reflection visible in the waveform as a repeated amplitude pattern. In the frequency domain, small constructive and destructive interferences appear, slightly modifying the spectral envelope due to phase overlap between the direct and delayed components.

- **Multiple Echo** - When multiple echoes are applied, the waveform exhibits successive attenuated repetitions. The amplitude histogram becomes more spread due to the cumulative superposition of delayed signals, and the frequency spectrum shows mild comb-filtering effects, typical of recursive delay systems.

- **Amplitude Modulation** - The `amplitudeModulation` effect produces periodic variations in signal amplitude, clearly observable as oscillations in the waveform envelope. In the frequency spectrum, additional sidebands appear around the original frequency components, corresponding to the modulation frequency and its harmonics.

- **Time-Varying Delay** - The `timeVaryingDelay` effect introduces a continuously changing delay time, resulting in a smooth detuning or "thickening" of the sound. This can be seen as a subtle fluctuation in the waveform and a moving interference pattern in the frequency plot, characteristic of chorus or flanger effects.

- **Bass Boosted** - The `bassBoosted` effect enhances the low-frequency content of the audio signal, resulting in a visibly larger amplitude for low-frequency waveforms in the time domain. In the waveform plot, regions of the signal with strong bass appear more pronounced, with wave peaks and troughs in these sections becoming noticeably more prominent after the effect is applied. This visual increase in amplitude reflects the boost provided to bass components during processing. In the frequency spectrum, there is a clear rise in the magnitude of frequency components below the cutoff point (around 200 Hz). The boosted spectrum shows a stronger response in these frequencies compared to the original, confirming effective enhancement of the bass range.

Observations

- The **Echo** and **Multiple Echo** effects primarily alter the signal in the time domain, creating delayed reflections.
- The **Amplitude Modulation** effect modifies the amplitude envelope and introduces low-frequency components in the spectrum.
- The **Time-Varying Delay** effect combines both time and frequency domain alterations, leading to dynamic phase and pitch modulation.
- The **Bass Boosted** effect primarily strengthens frequencies below the cutoff, making bass notes stand out more dramatically. This enrichment is especially noticeable in sections with musical bass or drums, contributing a sense of fullness and depth.

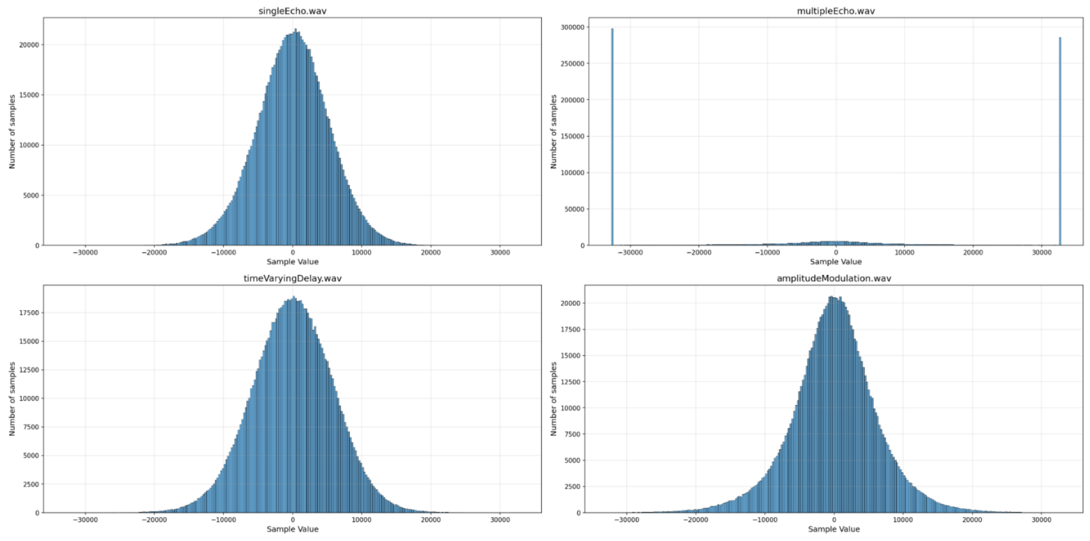


Figure 2.8: Histograms of SingleEcho, MultipleEcho, TimeVaryingDelay and AmplitudeModulation (respectively)

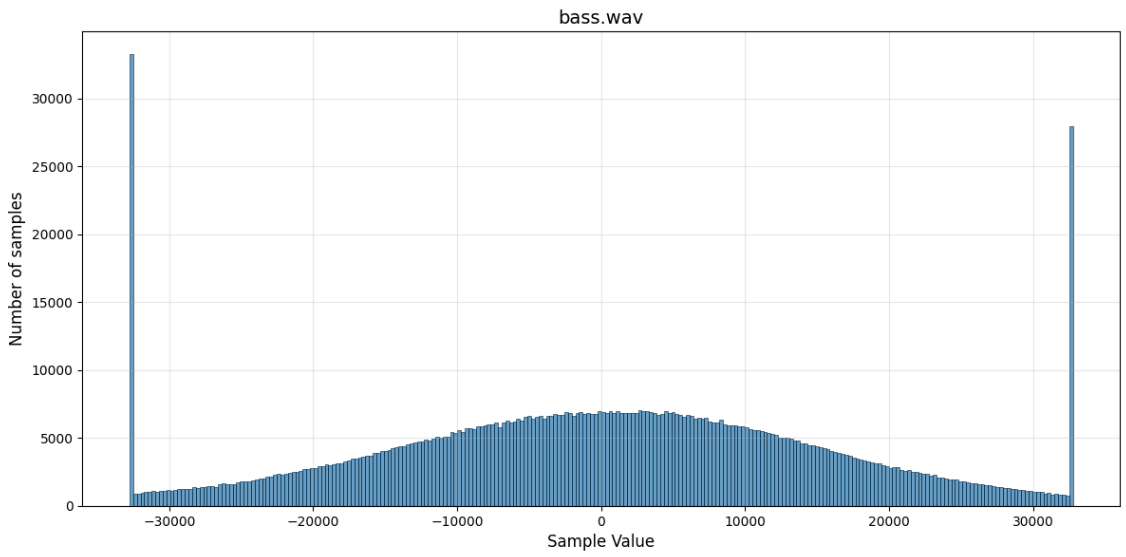


Figure 2.9: Bass Boosted Histogram

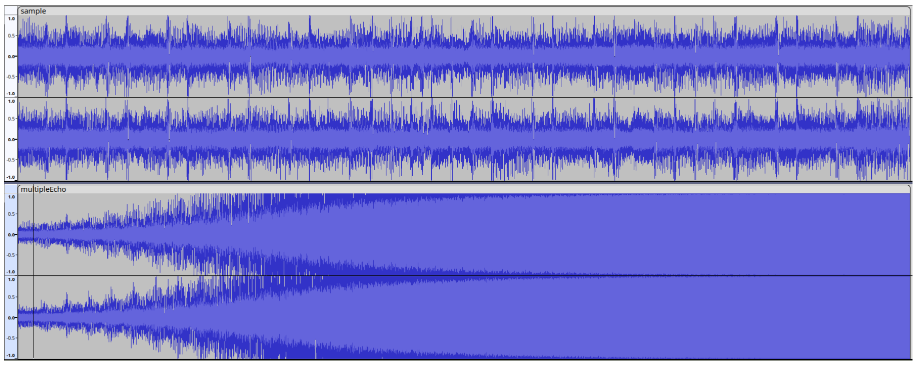


Figure 2.10: Comparison between Original Sample and Multiple echo

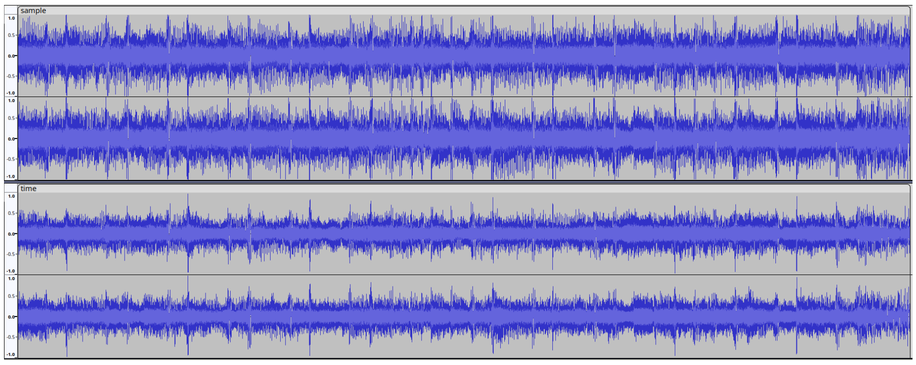


Figure 2.11: Comparison between Original Sample and Time varying delay

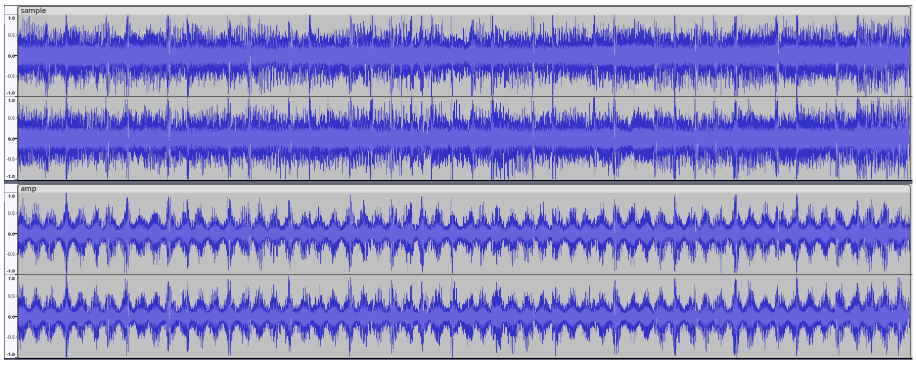


Figure 2.12: Comparison between Original Sample and Amplitude modulation

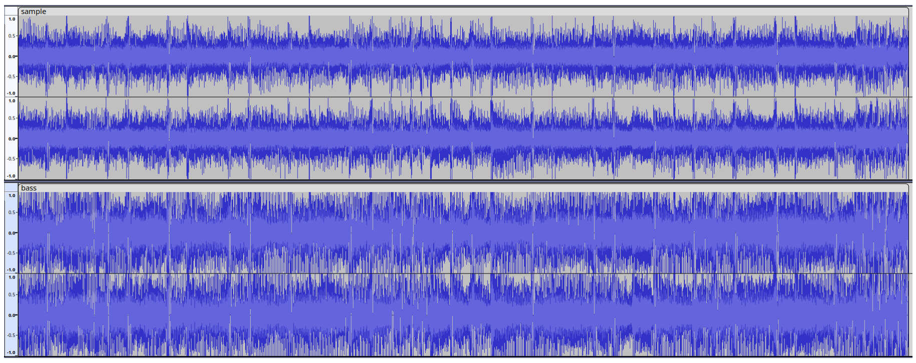


Figure 2.13: Comparison between Original Sample and Bass Boosted modulation

Overall, the visual and spectral analysis confirms that each implemented effect behaves according to its intended design, with clear and measurable differences compared to the original signal.

2.2 Part-II

The second stage introduces the **BitStream** class, which allows direct manipulation of bits when reading and writing files. Building upon this, an encoder (`wav_quant_enc`) and decoder (`wav_quant_dec`) are implemented to pack quantized audio samples efficiently and later recover them. This step marks the transition from high-level WAV file manipulation to lower-level bitstream representation, bridging the gap between quantization and coding.

2.2.1 Exercise 6

In this exercise, the **BitStream** class was introduced, allowing direct bit-level operations for encoding and decoding quantized audio. The main objective was to develop an encoder (`wav_quant_enc`) and a decoder (`wav_quant_dec`) capable of storing quantized audio samples compactly, improving efficiency compared to simple writing in conventional WAV format.

Code

Encoder The encoder reads samples from a WAV file, performs uniform quantization (controlled reduction of the number of bits per sample), and writes the results to a binary file using low-level operations with BitStream. Before the samples, the file header stores the following metadata: number of bits per sample, number of channels, sampling frequency, and total number of frames.

The per-sample process consists of:

1. Bit-shifting to the right to remove precision, keeping only the specified bits.
2. Storing each sample using only the strictly necessary bits, avoiding the typical redundancy of PCM16.
3. The compact storage results in a significantly smaller file for the same content.

Decoder The decoder does the opposite: it reads the binary file generated by the encoder, reconstructs the PCM samples from the number of bits originally used, and rewrites a valid WAV file. The decoder reads the header to recover all necessary parameters and expands each sample to its predictable value, maintaining compatibility with standard audio players.

Usage

The code can be tested using the command below, where `input.wav` is the original audio file to be encoded, `num_bits` represents the number of bits to keep, and `output.pack` is the compressed file generated:

```
./wav_quant_enc <input.wav> <num_bits> <output.pack>
```

The following command performs the inverse operation, i.e., decoding the packed file and reconstructing the corresponding WAV file:

```
./wav_quant_dec <output.pack> <reconstructed.wav>
```

Results

The implementation of uniform quantization allowed us to standardize bit depth across samples, offering direct control over the balance between audio quality and compression. Reducing the number of bits per sample significantly decreases file size, but at the cost of introducing noticeable distortion. By employing a BitStream for bitwise manipulation, only the required number of bits per sample are written, eliminating redundancies typically present in PCM formats and improving storage efficiency.

Compatibility and robustness were ensured through validation checks and proper handling of metadata, which guarantees that reconstructed files retain the original sampling rate and channel configuration. This means that, despite the reduced bit depth, the resulting WAV files remain valid and interoperable.

The compression benefits are clear: smaller bit depths such as 4 or 8 bits yield substantial savings in file size compared to PCM16. However, this also reduces the diversity of values available in the MID and SIDE channels, as illustrated in histograms, where discrete levels become more clustered under stronger quantization.

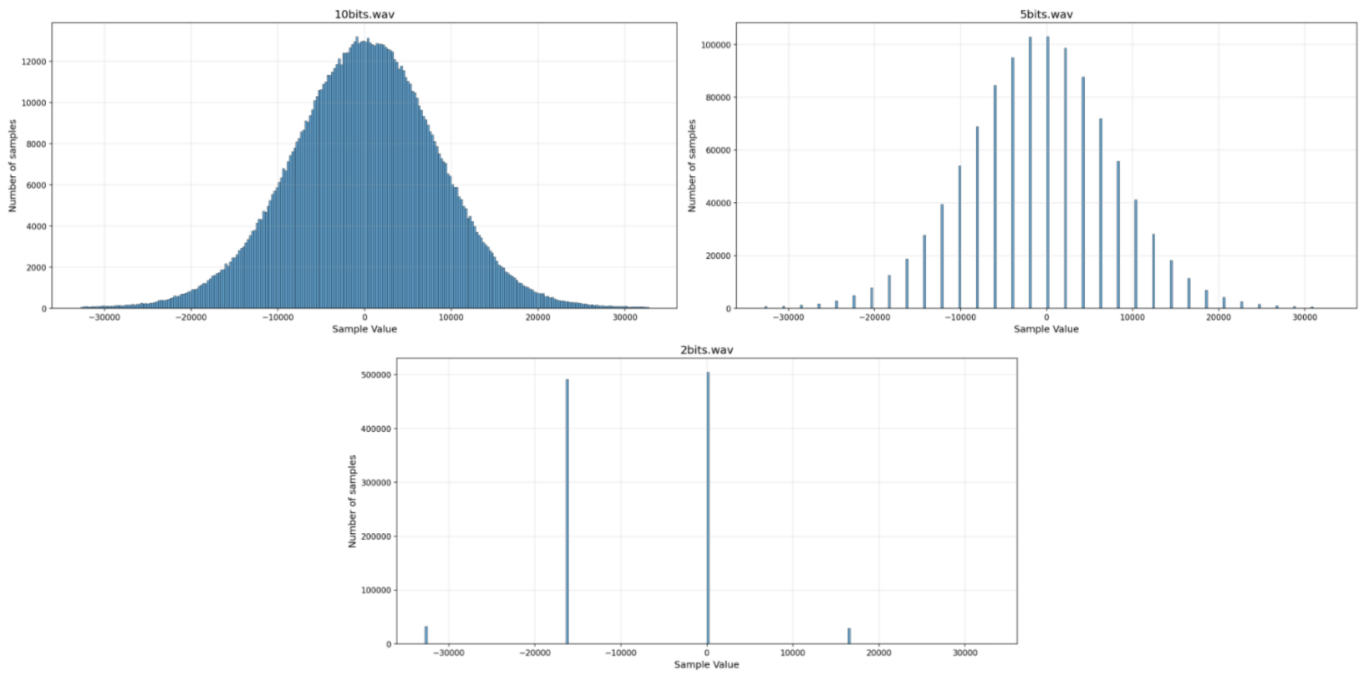


Figure 2.14: Histogram comparing the original and 10 bit file

Objective measurements - including MSE, maximum absolute error, and SNR - confirm this tradeoff, showing how higher compression consistently comes with increased distortion.

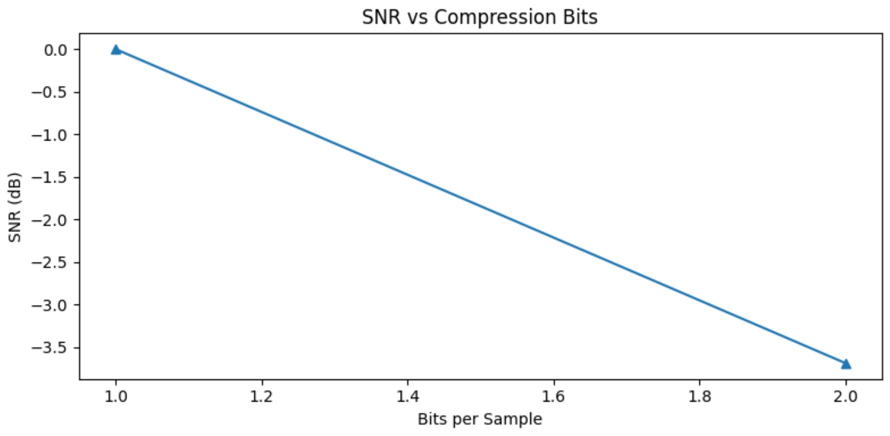


Figure 2.15: SNR vs Compression Bits

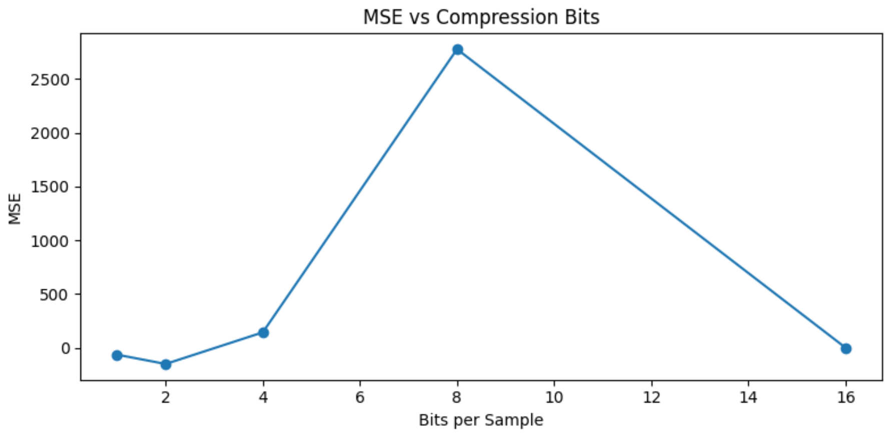


Figure 2.16: MSE vs Compression Bits

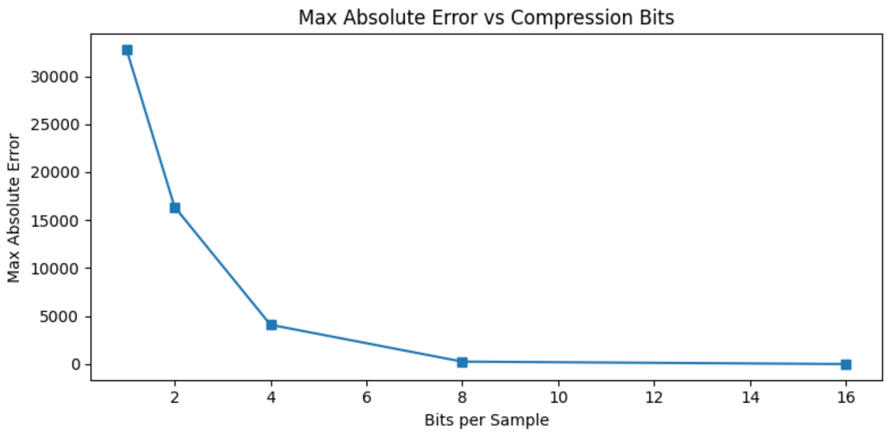


Figure 2.17: Max Absolute Error vs Compression Bits

Overall, reducing bit depth proves to be an effective method when storage space or transmission capacity is constrained. While it inevitably compromises audio quality, the BitStream-based approach maximizes efficiency in packing and unpacking data, providing a practical and lightweight solution for embedded applications and low-bandwidth audio transmission.

2.3 Part-III

In the third stage, a lossy codec for mono audio is implemented using the Discrete Cosine Transform (DCT). Audio is processed block by block, transformed into frequency coefficients, quantized, and stored using the BitStream class. A decoder reconstructs the signal from this encoded data, producing an approximate version of the original audio. The main goal is to achieve effective compression ratios while maintaining acceptable audio quality, exploring the balance between efficiency and fidelity.

2.3.1 Exercise 7

This exercise develops an audio encoder and decoder that use the **Discrete Cosine Transform (DCT)** on fixed-size blocks of PCM audio samples for lossy compression and reconstruction. Instead of direct sample quantization (as in Exercise 6), this approach operates in the **frequency domain**, where energy compaction allows more efficient storage of audio data.

Code

1. **Encoder**

   The encoder works by processing a mono WAV file in blocks of 1024 samples. Each block is transformed from the time domain into the frequency domain using the **DCT-II**, which concentrates most of the signal's energy in the lower-frequency coefficients. Once the transform is applied, the frequency coefficients are **quantized**: each coefficient is divided by a user-defined quantization step (‘qstep‘) and rounded to the nearest integer. This step effectively reduces the precision of the data, allowing for compression while keeping the most perceptually relevant information intact.

   To make the data compact and easily storable, the encoder then **packs the coefficients into a binary stream**. A header is included at the beginning of the file, containing essential metadata such as the block size, sampling frequency, number of channels (always 1 for mono), the quantization step, and the total number of samples. Since the quantized coefficients can take on negative values, they are shifted by **+32768**, ensuring they fit neatly into unsigned 16-bit integers before being written to the file.

   This frequency-domain representation, together with quantization, enables a significant reduction in file size while preserving the most important aspects of the original signal.

2. **Decoder**

   The decoder takes the compressed binary file and carefully reverses the encoding process to reconstruct the original audio signal. It begins by reading the metadata header, which provides all the necessary parameters for decoding, such as the block size, quantization step, and total number of samples. With this information, the decoder restores the quantized frequency coefficients by subtracting the earlier offset of 32768, bringing them back to signed integer form. These coefficients are then **dequantized** by multiplying them with the quantization step, recovering approximate values of the original DCT outputs.

Once the frequency-domain data is restored, the decoder applies the **inverse DCT (IDCT)** to transform each block back into time-domain audio samples. To ensure playback compatibility, the samples are **clipped** so that no values exceed the valid PCM16 range. Finally, the reconstructed blocks are written to a new WAV file, which retains the original sampling rate and channel configuration of the input audio.

The result is a standard WAV file that, although not bit-identical to the original, remains perceptually similar and fully playable, with compression efficiency determined by the choice of quantization step.

Usage

The code can be tested using the command below, where `input_file` is the name of the WAV file to be compressed, `quant_step` is the quantization step and `output_file` is the compressed file generated.

```
../bin/wav_dct_enc <input_file> <quant_step> <output_file>
```

The following command performs the inverse operation, i.e., decoding the previously compressed file and reconstructing a new WAV file:

```
../bin/wav_dct_dec <input_file> <output_file>
```

Results

The choice of quantization step (`qstep`) plays a central role in balancing audio quality, but in the current implementation, **the compressed file size remains constant regardless of the qstep value**. This occurs because all DCT coefficients are encoded using 16 bits each, even when many coefficients are quantized to zero with larger qstep values.

To achieve actual compression, the encoder would need to implement:

- **Run-length encoding** for consecutive zero coefficients
- **Huffman coding** or other entropy coding methods
- **Selective coefficient discarding** based on energy concentration

One of the main strengths of the DCT approach is that it concentrates most of the signal's energy into a few low-frequency coefficients. With the current implementation, this energy concentration is not exploited for size reduction - all coefficients are stored regardless of their significance.

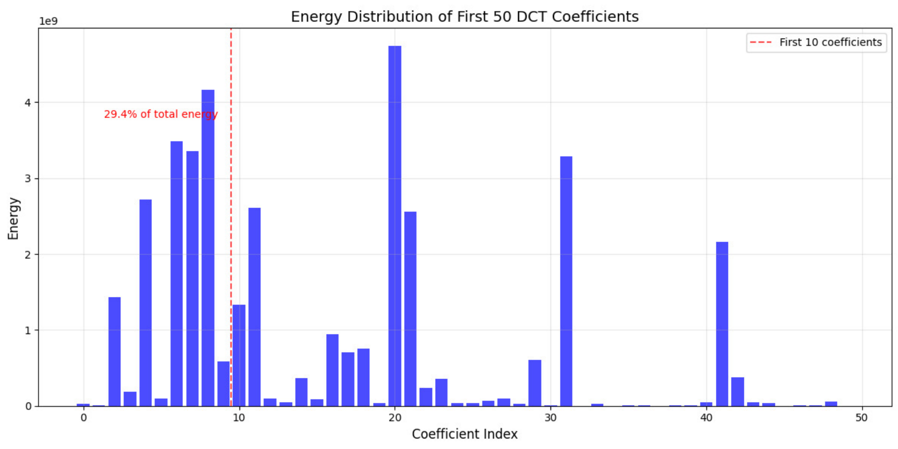


Figure 2.18: Energy Distribution of First 50 DCT Coefficients

Quantizing in the frequency domain aligns with human auditory perception and could allow for data reduction with minimal perceptual impact. However, without proper entropy coding, the potential compression benefits are not realized in the file size.

The reconstructed output remains a standard WAV file with proper metadata, ensuring compatibility with common audio players.

To evaluate performance, both objective and subjective measures can be applied:

- **Signal fidelity**: Mean squared error (MSE) and signal-to-noise ratio (SNR) quantify reconstruction quality
- **Perceptual quality**: Listening tests reveal how different quantization settings affect the audio experience
- **Compression potential**: While current file sizes remain constant, the concentration of energy in few coefficients indicates significant compression opportunities with improved encoding

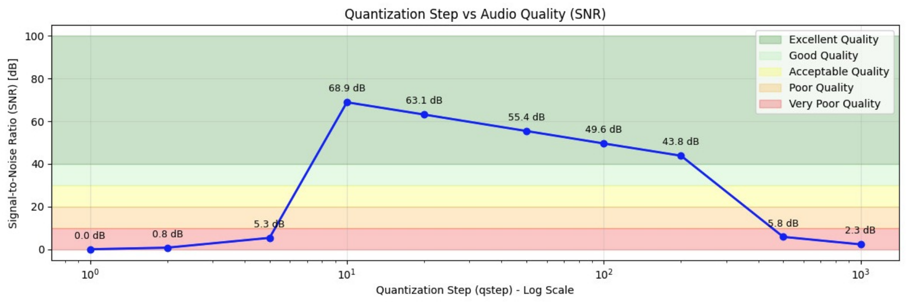


Figure 2.19: Quantization Step vs Audio Quality

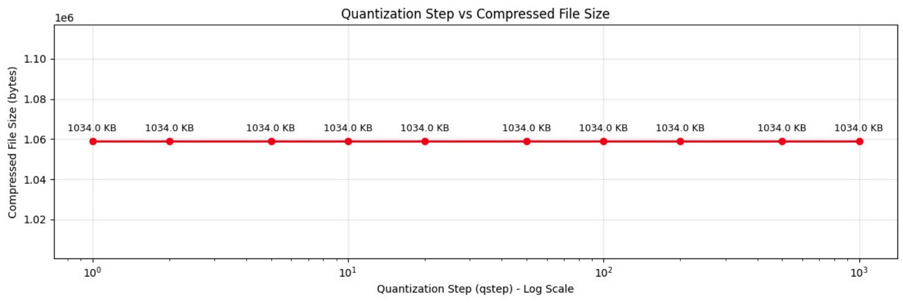


Figure 2.20: Quantization Step vs Compressed File Size

Chapter 3

Conclusion

This lab project provided hands-on experience with the development of audio processing tools in C++, combining practical coding skills with theoretical insights into digital audio systems.

The group successfully extended and improved standard audio manipulation functionalities by implementing efficient histogram generation, quantization techniques, codec design, and sound effects processing-all integrated with robust command-line interfaces and careful validation.

Throughout the assignment, emphasis was placed on producing reliable code, creative experimentation, and in-depth testing, resulting in tools that are adaptable to different audio formats and capable of delivering meaningful analysis on sound data.

The work highlighted the importance of structured design, modular programming, and iterative debugging to solve real-world engineering problems in Information and Coding.