universidade de aveiro
Departamento de Eletrónica, Telecomunicações e Informática

Curso:	Mestrado em Engenharia de Computadores e Telemática
Disciplina:	42078 - Informação e Codificação
Ano letivo:	2025/2026

Relatório

Lab work n1

Autores: 103075 — José Jordão
108689 — Gabriel Janicas
107715 — Afonso Rodrigues

Turma: P1

Docente: Armando J. Pinho

Universidade de Aveiro, October 19, 2025

Contents

1 Important Information **4**

2 Part 1 **5**
- 2.1 Exercise 1 . . . 5
  - 2.1.1 Code . . . 5
  - 2.1.2 Usage . . . 6
  - 2.1.3 Results . . . 7
- 2.2 Exercise 2 . . . 9
  - 2.2.1 Code . . . 9
  - 2.2.2 Usage . . . 9
  - 2.2.3 Results . . . 9
- 2.3 Exercise 3 . . . 11
  - 2.3.1 Code . . . 11
  - 2.3.2 Usage . . . 12
  - 2.3.3 Results . . . 12
- 2.4 Exercise 4 . . . 13
  - 2.4.1 Code . . . 13
  - 2.4.2 Usage . . . 14

3 Part 2 **15**
- 3.1 Exercise 6 . . . 15
  - 3.1.1 Introduction . . . 15
  - 3.1.2 Implementation Overview . . . 15
  - 3.1.3 Usage . . . 17
  - 3.1.4 Results . . . 17

4 Part 3 **19**
- 4.1 Exercise 7 . . . 19
  - 4.1.1 Introduction . . . 19
  - 4.1.2 Implementation Overview . . . 19
  - 4.1.3 Header Structure . . . 22
  - 4.1.4 Usage . . . 22
- 4.2 Results . . . 23

List of Figures

2.1 Left Channel Plot 7
2.2 Right Channel Plot 7
2.3 Mid Channel Plot 8
2.4 Side Channel Plot 8
2.5 Histogram of audio file with only 4 bits of resolution 10
2.6 Histogram of audio file with only 6 bits of resolution 10
2.7 Histogram of audio file with only 8 bits of resolution 11

List of Tables

3.1 Compression and performance results for sample01.wav 17
3.2 Compression performance across different audio samples (8-bit quantization) 18

4.1 File Format 22
4.2 DCT-Based Compression Performance for `sample01.wav` 23

Chapter 1

Important Information

In this report, it is described all the work done within the scope of the project "Lab nº 1". For each exercise it is presented the code developed, the theory behind all of it, and, of course, the results. This was a project developed equally by all the 3 contributors mentioned on the title page.

In order to create all the plots presented in this report, there was developed a python script. In order to take advantage of it, use the following command:

```
1
2       python ../src/plot_histogram.py <histogram.txt> <output file>
3
```

Chapter 2

Part 1

2.1 Exercise 1

In this first exercise we were asked to change the *WAVHIST* class in a way which this class would be able to provide this histogram of the average of the channels and the average difference of the channels when the audio is stereo.

2.1.1 Code

First we needed two new data structures to mimic the behaviour of the previous code, but now for the **MID** and **SIDE** channels.

```
1       std::vector<std::map<short, size_t>> mid_values;
2       std::vector<std::map<short, size_t>> side_values;
```

In the same line of thought, we added the functions *mid_dump*, *side_dump*, *mid_update* and *side_update*. Functions with same purpose of the initial ones, but now, channel specific.

By adding/- subtracting the left and right channels and dividing the result by the number of channels, the mid and side channel values were determined. In this instance, the odd channel (2*i + 1) is always on the right, and the even number index (2*i) is always on the left.

```
1
2   void update_mid(const std::vector<short>& samples) {
3         // Check if audio is stereo
4         if(counts.size() != 2) {
5             std::cerr << "Mid processing requires stereo audio (2 channels)
      \n";
6             return;
7         }
8         for(long unsigned int i = 0; i < samples.size()/2; i++) {
9             mid_values[0][(samples[2*i] + samples[2*i+1]) / 2]++;
10        }
11    }
12
13    void update_side(const std::vector<short>& samples) {
14        // Check if audio is stereo
15        if(counts.size() != 2) {
16            std::cerr << "Side processing requires stereo audio (2 channels
      )\n";
17            return;
18        }
```

```
        for(long unsigned int i = 0; i < samples.size()/2; i++) {
            side_values[0][(samples[2*i] - samples[2*i+1]) / 2]++;
        }
    }

    void mid_dump() const {
        if(mid_values[0].empty()) {
            std::cerr << "No mid channel data available\n";
            return;
        }
        for(auto [value, counter] : mid_values[0])
            std::cout << value << '\t' << counter << '\n';
    }

    void side_dump() const {
        if(side_values[0].empty()) {
            std::cerr << "No side channel data available\n";
            return;
        }
        for(auto [value, counter] : side_values[0])
            std::cout << value << '\t' << counter << '\n';
    }

```

In *wav_hist.cpp* we added code to check if the channel argument was selected properly.

```
    int channel { stoi(argv[argc-1]) };
    if(channel >= sndFile.channels() + 2) {
        cerr << "Error: invalid channel requested\n";
        return 1;
    }

    // Check if mid/side channels are requested for mono file
    if((channel == 2 || channel == 3) && sndFile.channels() != 2) {
        cerr << "Error: mid/side channels only available for stereo files\n
    ";
        return 1;
    }

```

2.1.2 Usage

The usage of the program is done following this pattern:

```

        ./wav_hist <input file> <channel> > histogram.txt


```

In this case the input file is an audio file. Running the command above will store the values to a ".txt". In order to plot the results, we developed a python script which receives a .txt as input and creates a png with the results.

```

        python ../src/plot_histograms.py <histogram.txt> Title


```

2.1.3 Results

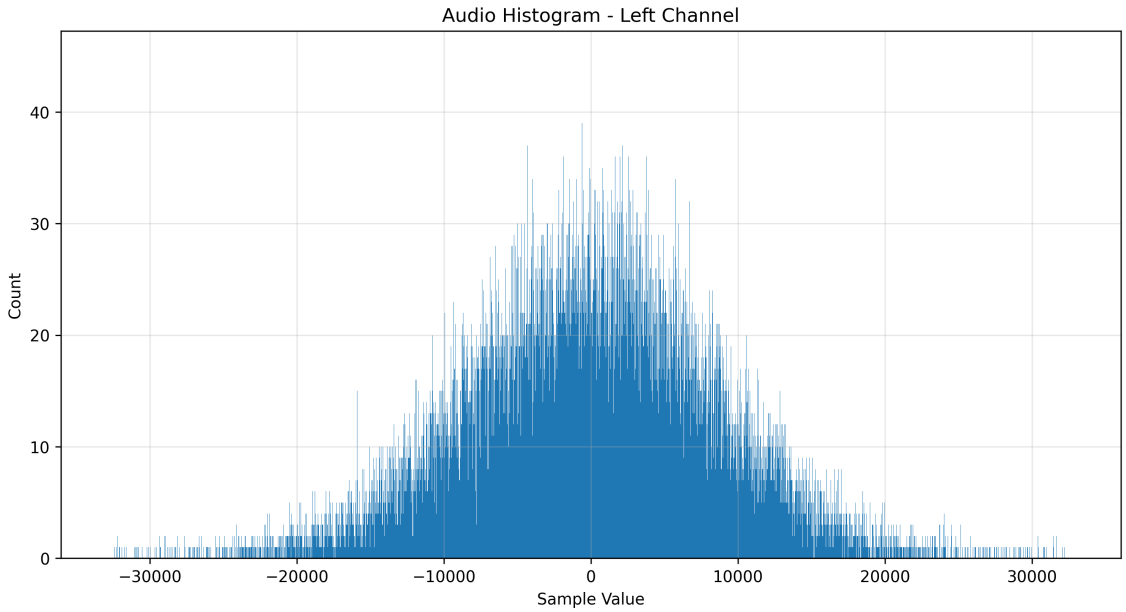

Figure 2.1: Left Channel Plot

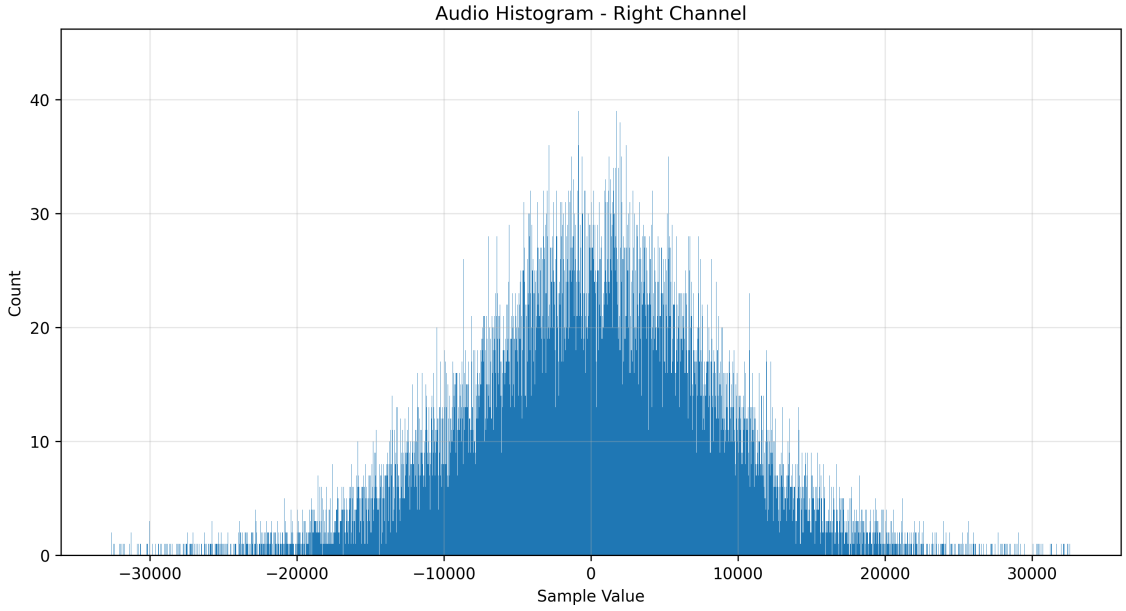

Figure 2.2: Right Channel Plot

By examining Figures 2.1 and 2.2, we can see that the Left and Right audio channels have similar frequency distributions and function across comparable amplitude ranges (about -30000 to +30000), suggesting a well-balanced stereo recording. As a result, the Mid and Side channel results in Figures 2.3 and 2.4 match theoretical predictions.

Since both original channels have similar features, the Mid channel ((L+R)/2), which is the average of the Left and Right channels, maintains a similar amplitude range and frequency distribution. As a result, the histogram roughly resembles the individual channels'

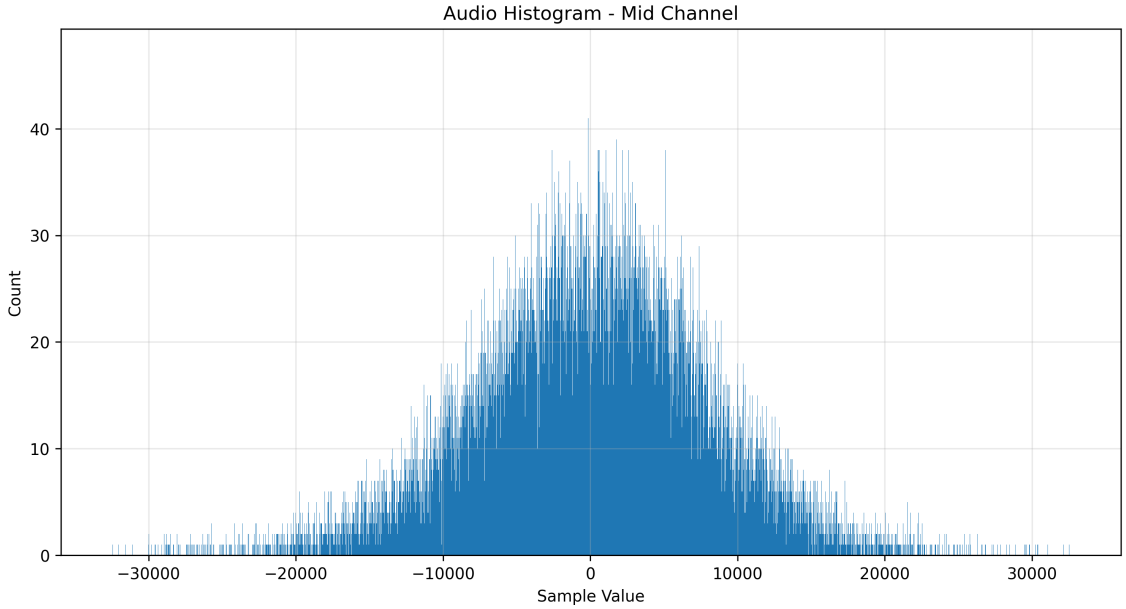

Figure 2.3: Mid Channel Plot

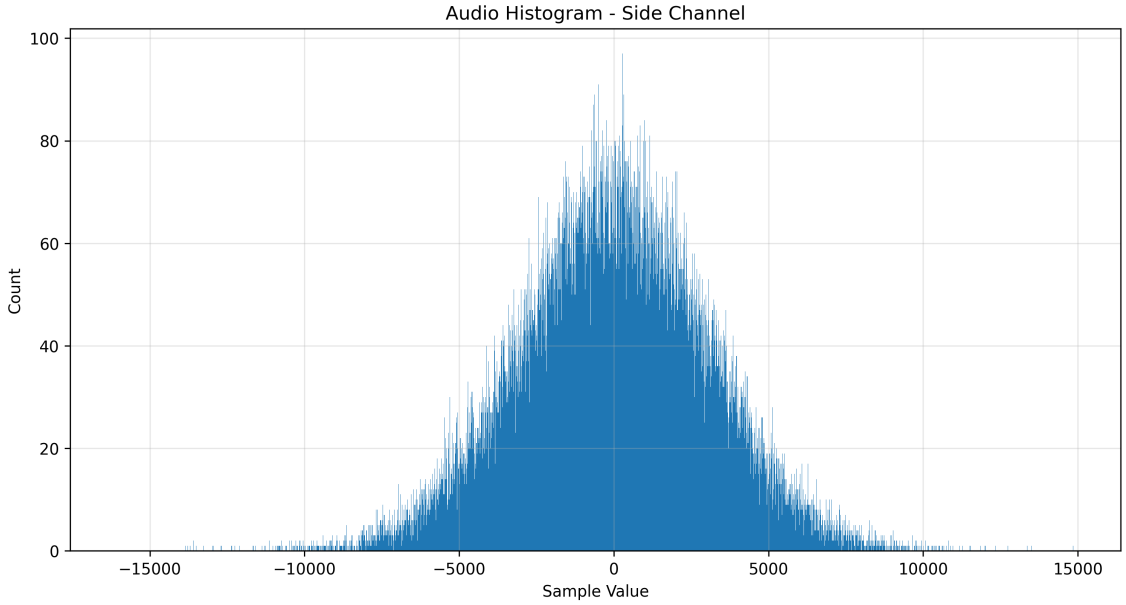

Figure 2.4: Side Channel Plot

histograms.

The difference between the two channels is represented by the Side channel ((LR)/2), which exhibits a strong concentration around zero and a reduced amplitude range (around -15000 to +15000). When the Left and Right channels are similar, this behavior is predicted, indicating that the stereo mix is well-balanced and that the recording has good mono compatibility.

Overall, the results show that Mid-Side processing behaves as expected.

2.2 Exercise 2

In this exercise we were instructed to implement a program, named wav quant, to reduce the number of bits used to represent each audio sample. We were asked to perform uniform scalar quantization.

2.2.1 Code

To quantify the sample, we first shift the desired number of bits to the right, and then we shift the same number to the left, "replacing" the previous values with zeros.

```
1
2     class WAVQuant {
3 public:
4     WAVQuant() {}
5
6     void quant(std::vector<short>& samples, size_t num_bits) {
7         for (auto& sample : samples) {
8             sample = (sample >> num_bits) << num_bits;
9         }
10    }
11 };
12
13
```

The quantified samples are written to an output audio file when we just utilize the quant function in *wav_quant.cpp*.

```
1
2         while((nFrames = sfhIn.readf(samples.data(), FRAMES_BUFFER_SIZE)))
    {
3         quant.quant(samples, bits_to_cut);
4
5         sfhOut.writef(samples.data(), nFrames);
6     }
7
```

2.2.2 Usage

```
1
2     ./wav_quant <input file> <Bits to Keep>  <output file>
3
```

In this case, the bits are kept as integers and the input/output files are audio files (tested with.wav).

2.2.3 Results

We can anticipate only receiving 2N distinct values, where N is the amount of bits retained in the audio file, in addition to a noticeable decline in audio quality. We anticipate $2^6 = 64$ bars (different values) for a sample with 6 bits, as shown in Figure 2.6. We anticipate $2^4 = 16$ bars for a sample with 4 bits, as shown in Figure 2.5, and $2^8 = 256$ bars for a sample with 8 bits, as shown in Figure 2.7.

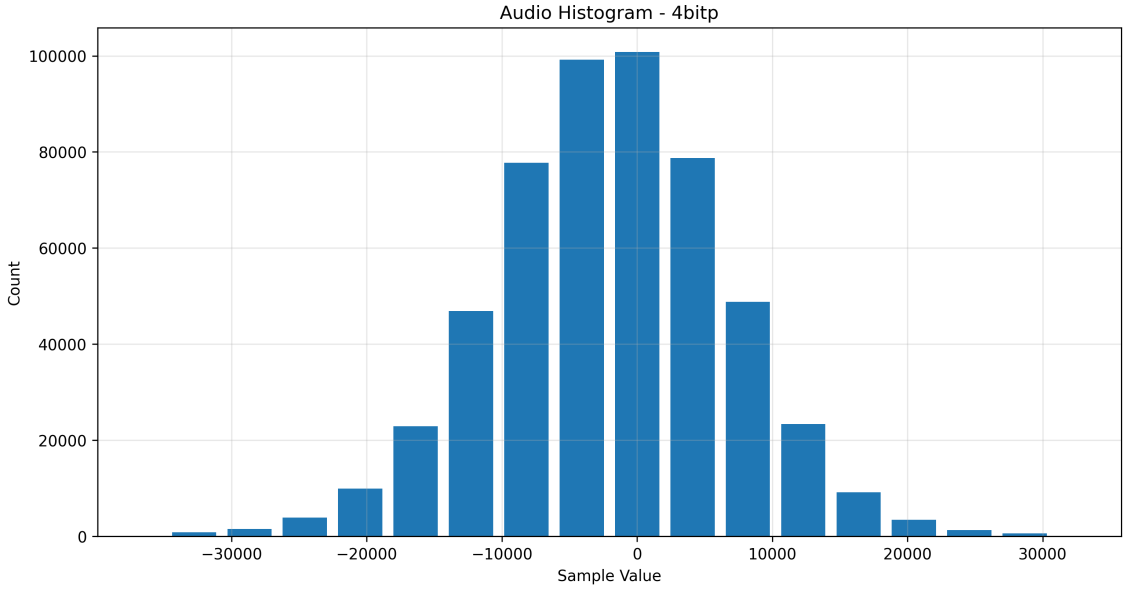

Figure 2.5: Histogram of audio file with only 4 bits of resolution

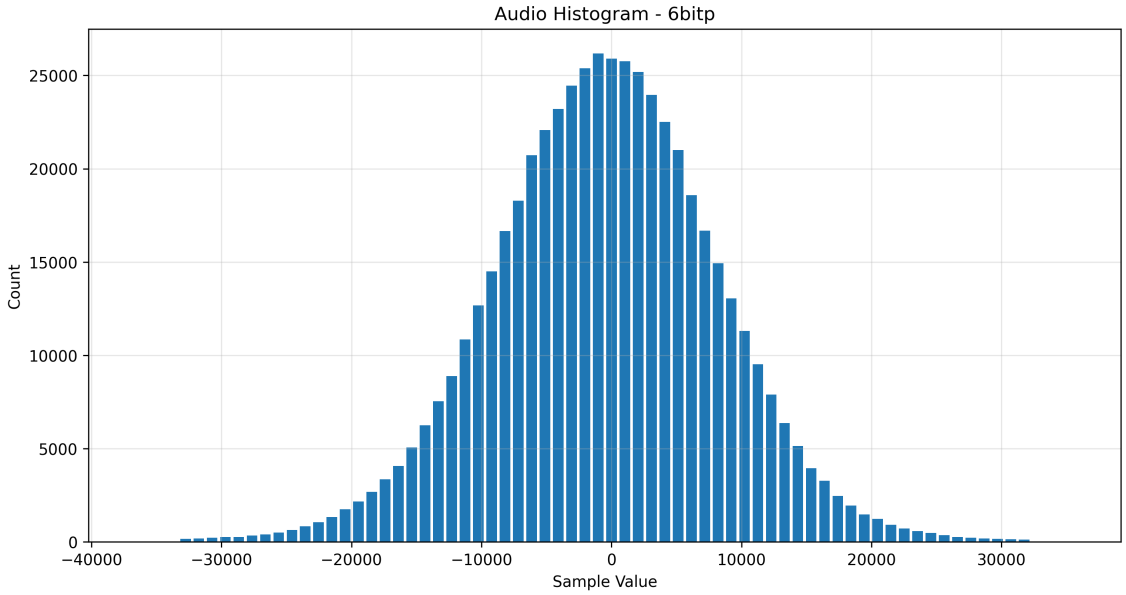

Figure 2.6: Histogram of audio file with only 6 bits of resolution

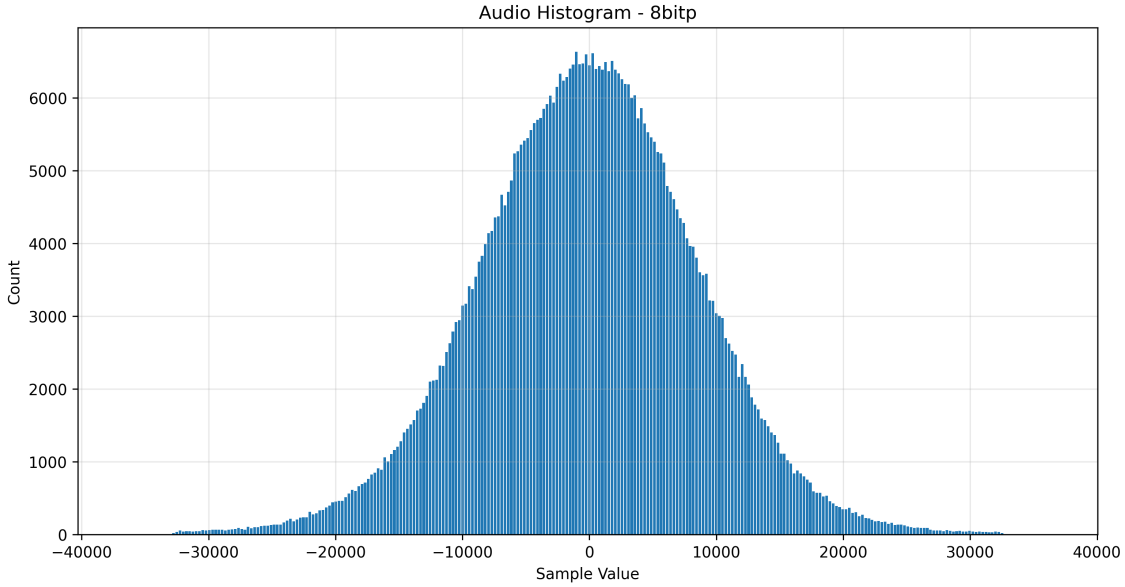

Figure 2.7: Histogram of audio file with only 8 bits of resolution

2.3 Exercise 3

This program is a command-line utility designed to perform a quantitative comparison between two WAV audio files. It verifies that the files have identical properties (format, channels, length) and then calculates several key quality and error metrics on a per-channel basis, as well as providing an overall summary for multi-channel files.

2.3.1 Code

To calculate Signal-to-noise-ratio **(SNR)** and the maximum per sample absolute error, we use the following expressions:

$$r(n) = x(n) - \tilde{x}(n), \text{ where } \tilde{x} \text{ is the quantified sample}$$
$$\mathcal{E}_x = \sum_n |x(n)|^2, \quad \mathcal{E}_r = \sum_n |r(n)|^2$$
$$SNR = 10 \times \log_{10} \frac{\mathcal{E}_x}{\mathcal{E}_r} \text{ dB (decibel)}$$

To Calculate the Mean Squared Error **(MSE)**, we use the following expression:

$$MSE = \frac{1}{N} \sum_n |r(n)|^2 = \frac{\mathcal{E}_r}{N}$$

And finally, to calculate the Maximum Absolute Error **(L Norm)**, we use the following expression:

$$E_{max} = \max_n |r(n)| = \max_n |x(n) - \tilde{x}(n)|$$

```
1
2           energy_signal[channel] += s1 * s1;
3               energy_noise[channel] += diff * diff;
```

```
4
5             if (abs_diff > max_error[channel]) {
6                 max_error[channel] = abs_diff;
7             }
8
9         ...
10
11        double snr = (energy_noise[c] == 0) ? INFINITY : 10 * log10(
    energy_signal[c] / energy_noise[c]);
12        double mse = energy_noise[c] / sfhIn1.frames();
13
14    }
15
```

2.3.2 Usage

```
1
2    ./wav_cmp <input file> <input file 2>
3
```

In this case, both input files are audio files, the files must have the same format (Wav) and the same frames.

2.3.3 Results

Signal-to-Noise Ratio (SNR) measures signal power relative to noise power in decibels. Values above 0 dB indicate more signal than noise. We compared the original audio file against quantized versions:

8-bit quantization: SNR = 35.32 dB, MSE = 21,720.38, Max Error = 255
6-bit quantization: SNR = 23.26 dB, MSE = 349,224.07, Max Error = 1,023
4-bit quantization: SNR = 11.21 dB, MSE = 5,599,532.21, Max Error = 4,095

The results follow the expected pattern: SNR decreases by approximately 6 dB per bit removed, consistent with uniform quantization theory. The Maximum Absolute Error quadruples with each 2-bit reduction ($\times 4 = 2^2$), and MSE increases dramatically as bit depth decreases. The 8-bit version maintains excellent quality (>35 dB), while the 4-bit version shows noticeable degradation but still preserves more signal than noise (> 0 dB).

2.4 Exercise 4

In this exercise we were challenged to build a program that would produce some audio effects, such as, Single Echo, Multiple echos, Amplitude Modulation and Reverse.

2.4.1 Code

Single Echo: $y(n) = x(n) + \alpha \times x(n - \text{delay})$
Multiple Echo: $y(n) = x(n) + \alpha \times y(n - \text{delay})$

```
1
2       if (wanted_effect == "single_echo" || wanted_effect == "
    multiple_echo") {
3       vector<short> delay_buffer(delay_samples, 0);
4       size_t delay_cursor = 0;
5
6       while((nFrames = sfhIn.readf(buffer.data(), FRAMES_BUFFER_SIZE))) {
7           size_t nSamples = nFrames * num_channels;
8           for(size_t i = 0; i < nSamples; ++i) {
9               short delayed_sample = delay_buffer[delay_cursor];
10              short current_sample = buffer[i];
11
12              // Add the echo
13              buffer[i] = current_sample + gain * delayed_sample;
14
15              // Update the delay buffer for the next iteration
16              short feedback_sample = (wanted_effect == "multiple_echo")
    ? buffer[i] : current_sample;
17              delay_buffer[delay_cursor] = feedback_sample;
18
19              delay_cursor = (delay_cursor + 1) % delay_samples;
20          }
21          sfhOut.writef(buffer.data(), nFrames);
22      }
23  }
24
25  }
26
```

In the case of the **Reverse Effect**, we reversed all the samples and swapped the left and right channels back.

```
1
2        else if (wanted_effect == "reverse") {
3       // For reverse, we must read the whole file first.
4       sfhIn.seek(0, SEEK_SET); // Rewind file handle
5       vector<short> all_samples(sfhIn.frames() * num_channels);
6       sfhIn.readf(all_samples.data(), sfhIn.frames());
7
8       // Reverse the entire vector of samples
9       std::reverse(all_samples.begin(), all_samples.end());
10
11      // If stereo, swap left and right channels back
12      if(num_channels > 1) {
13          for(size_t i = 0; i < all_samples.size(); i += num_channels) {
14              std::reverse(all_samples.begin() + i, all_samples.begin() +
    i + num_channels);
15          }
16      }
```

```
        sfhOut.writef(all_samples.data(), sfhIn.frames());
    }

```

Amplitude Modulation: $y(n) = x(n) \times \cos(2\pi f t)$

```
    else if (wanted_effect == "amplitude_modulation") {
        long long total_samples_processed = 0;
        double sample_rate = sfhIn.samplerate();

        while((nFrames = sfhIn.readf(buffer.data(), FRAMES_BUFFER_SIZE))) {
            size_t nSamples = nFrames * num_channels;
            for(size_t i = 0; i < nSamples; ++i) {
                // Use a global sample counter for continuous phase
                double time = (double)(total_samples_processed + i /
num_channels) / sample_rate;
                buffer[i] *= cos(2 * M_PI * freq * time);
            }
            sfhOut.writef(buffer.data(), nFrames);
            total_samples_processed += nSamples;
        }
    }

```

2.4.2 Usage

For this program, the command to use, will depend on the effect to apply. For the **single and multiple echo**, the commands are basically the same, with the same parameters, changing only the effect name.

```
./wav_effects <input file> <output file> single_echo <delay(ms)> <gain>

./wav_effects <input file> <output file> multiple_echo <delay(ms)> <gain>


```

If the effect to apply is **Amplitude Modulation**, the command is the following:

```
./wav_effects <input file> <output file> amplitude_modulation <freq(Hz)>

```

For the reverse effect, there are no parameters; only the effect name will work.

```
./wav_effects <input file> <output file> reverse

```

In all this commands, the input and output files, are audio files.

Chapter 3

Part 2

3.1 Exercise 6

3.1.1 Introduction

This section describes the implementation and evaluation of a codec based on uniform scalar quantization with bit-packing. Unlike the WAV quantizer from Part I, this codec uses the **BitStream** class to efficiently pack quantized samples, achieving significant file size reduction.

3.1.2 Implementation Overview

Encoder (wav_quant_enc)

The encoder reads a standard 16-bit WAV file and produces a compressed binary format through three main steps:

```
// Extract WAV metadata
uint32_t sample_rate = *reinterpret_cast<uint32_t*>(&header[24]);
uint16_t num_channels = *reinterpret_cast<uint16_t*>(&header[22]);
uint32_t num_samples = data_size / (sizeof(int16_t) * num_channels);
```

Listing 3.1: Extracting WAV metadata

Step 2: Custom Header Generation The encoder writes a compact 15-byte header containing all necessary decoding information:

```
// Write 15-byte header:
// magic (4) + rate (4) + channels (2) + bits (1) + frames (4)
ofs.write("WQ01", 4);
ofs.write(reinterpret_cast<const char*>(&sample_rate), 4);
ofs.write(reinterpret_cast<const char*>(&num_channels), 2);
ofs.write(reinterpret_cast<const char*>(&qbits), 1);
ofs.write(reinterpret_cast<const char*>(&num_samples), 4);
```

Listing 3.2: Writing custom binary header

```
const int levels = 1 << quant_bits;  // 2^n levels
for (size_t i = 0; i < samples.size(); ++i) {
    // Normalize to [0, 1)
    float normalized = (samples[i] + 32768.0f) / 65536.0f;


    int q_index = static_cast<int>(normalized * levels);


    bs.write_n_bits(q_index, quant_bits);
}
```

Listing 3.3: Uniform quantization and bit-packing

Decoder (wav_quant_dec)

The decoder reverses the process, reconstructing a playable WAV file from the compressed format:

```

char magic[5] = {0};
ifs.read(magic, 4);
if (string(magic, 4) != "WQ01") {
    cerr << "Invalid file format\n";
    return 1;
}
```

Listing 3.4: Validating compressed file format

Step 2: Dequantization The decoder implements a reconstruction to minimize quantization error:

```
const int levels = 1 << quant_bits;
for (size_t i = 0; i < total_samples; ++i) {

    int q_index = bs.read_n_bits(quant_bits);

    float normalized = (q_index + 0.5f) / levels;

    // Denormalize to [-32768, 32767]
    int16_t sample = static_cast<int16_t>(
        normalized * 65536.0f - 32768.0f
    );
    samples.push_back(sample);
}
```

Listing 3.5: Dequantization with mid-point reconstruction

```
// Build standard 44-byte WAV header
ofs.write("RIFF", 4);
ofs.write(reinterpret_cast<char*>(&file_size_wav), 4);
ofs.write("WAVE", 4);

```

```
ofs.write("data", 4);
ofs.write(reinterpret_cast<char*>(&data_size), 4);
ofs.write(reinterpret_cast<char*>(samples.data()), data_size);
```

Listing 3.6: Building standard WAV header

Quantization Formula

The quantization process follows a well-defined mathematical mapping:

Encoding:

$$\text{normalized} = \frac{x + 32768}{65536} \in [0, 1) \tag{3.1}$$

$$q = \lfloor \text{normalized} \times 2^n \rfloor \in [0, 2^n - 1] \tag{3.2}$$

where x is the original 16-bit sample and n is the number of quantization bits.

Decoding:

$$\hat{\text{normalized}} = \frac{q + 0.5}{2^n} \tag{3.3}$$

$$\hat{x} = \hat{\text{normalized}} \times 65536 - 32768 \tag{3.4}$$

3.1.3 Usage

```

#Encoding
./wav_quant_enc input.wav output.bin <quant_bits>

#Decoding
./wav_quant_dec output.bin decoded.wav

```

3.1.4 Results

In order to test both the developed encoder and decoder, we used the different audio files provided with the assignment. From these tests, we obtained the following results.

Table 3.1: Compression and performance results for sample01.wav

Quant. Bits	Encoded Size (KB)	Compression Ratio	Space Savings	Total Time (ms)	Throughput (MB/s)
4	1,264	4.00×	75.0%	66	74.9
6	1,896	2.67×	62.5%	74	66.8
8	2,528	2.00×	50.0%	83	59.6
10	3,160	1.60×	37.5%	102	48.5
12	3,792	1.33×	25.0%	115	43.0
14	4,424	1.14×	12.5%	126	39.3
16	5,055	1.00×	0.0%	154	32.1

Table 3.1 presents the compression results for sample01.wav.The codec demonstrates consistent performance across different quantization levels, with compression ratios ranging from 4.00× (4-bit quantization) to 1.00× (16-bit, no compression).

Table 3.2: Compression performance across different audio samples (8-bit quantization)

Sample	**Original (KB)**	**Encoded (KB)**	**Ratio**	**Enc. Time (ms)**	**Dec. Time (ms)**
sample01.wav	5,055	2,528	2.00×	41.8	40.5
sample02.wav	2,529	1,264	2.00×	24.2	24.3
sample03.wav	3,448	1,724	2.00×	33.2	31.4
sample04.wav	2,299	1,150	2.00×	20.8	23.4
sample05.wav	3,563	1,781	2.00×	33.7	33.2

The results confirm the theoretical alignment: all samples achieve 2.00× compression with 8-bit quantization. Processing time scales linearly with file size (∼30 ms per MB), and encoding/decoding times are balanced, demonstrating the codec's consistent behaviour regardless of audio content or duration.

Chapter 4

Part 3

4.1 Exercise 7

4.1.1 Introduction

Following the codec implementation in Part 2, this section details the development of a lossy codec for mono audio files, as required by Exercise 7. This codec is based on the Discrete Cosine Transform (DCT), a fundamental technique in modern compression standards for audio and video. The core principle is to transform audio from the time domain to the frequency domain, where energy is typically concentrated in a few low-frequency coefficients. By selectively quantizing and discarding less significant high-frequency information, we can achieve substantial compression while aiming to minimize the perceptual loss of audio quality. The entire process is managed on a block-by-block basis, and the final compressed data is efficiently packed using the **BitStream** class.

4.1.2 Implementation Overview

Encoder (`wav_dct_enc`)

The encoding process follows these key steps:

1. **Input and Pre-processing**: The encoder reads a 16-bit PCM WAV file. If the input audio is stereo, it is first downmixed to a single mono channel by averaging the left and right channels (`(L+R)/2`). This ensures compatibility with the mono-centric design.

```
for(size_t i = 0 ; i < nFrames ; ++i) {
    // L is at i*2, R is at i*2 + 1
    samples[i] = (raw_samples[i * 2] + raw_samples[i * 2 + 1]) / 2;
}

```

Listing 4.1: Downmixing to Mono (In case the file is Stereo)

2. **Header Generation**: Before processing the audio data, a custom header is written to the output file. This header contains all the necessary metadata for the decoder to correctly reconstruct the audio, such as sample rate, block size, and quantization parameters.

```
bsOut.write_n_bits(static_cast<uint64_t>(sampleRate), 32);
bsOut.write_n_bits(static_cast<uint64_t>(bs), 16);
bsOut.write_n_bits(static_cast<uint64_t>(nDctCoeffsPerBlock), 16);
```

```
bsOut.write_n_bits(static_cast<uint64_t>(N_BITS_QUANT), 8);
bsOut.write_n_bits(static_cast<uint64_t>(nFrames), 32);
bsOut.write_n_bits(nChannelsIn, 8);

```

Listing 4.2: Write Encoder Header

3. **Blocking and Padding**: The mono audio stream is segmented into fixed-size blocks (e.g., 1024 samples). If the total number of samples is not a multiple of the block size, the final block is zero-padded to ensure uniform processing.

```
size_t nBlocks = static_cast<size_t>(ceil(static_cast<double>(nFrames)
    / bs));
samples.resize(nBlocks * bs * nChannelsOut);

```

Listing 4.3: Ensurence of Uniform Processing

4. **DCT Transformation**: Each block is transformed into the frequency domain using the DCT-II algorithm (`FFTW_REDFT10`). This transformation concentrates the signal's energy into the lower-frequency coefficients.

```
fftw_plan plan_d = fftw_plan_r2r_1d(bs, x.data(), x.data(),
                                    FFTW_REDFT10, FFTW_ESTIMATE);
// ... inside loop ...
fftw_execute(plan_d);

```

Listing 4.4: DCT Processing and Encoding

5. **Coefficient Selection**: Compression is achieved by retaining only a specified fraction of the initial (low-frequency) DCT coefficients. The remaining high-frequency coefficients, which contribute less to the perceptual quality of the audio, are discarded.

```
for(size_t k = 0 ; k < nDctCoeffsPerBlock ; k++) {
    double dct_coeff = x[k];

    long q_val = lround(dct_coeff);

    bsOut.write_n_bits(static_cast<uint64_t>(q_val), N_BITS_QUANT);
}

```

Listing 4.5: Quantization and Writing to BitStream

6. **Quantization and Bit-Packing**: The selected coefficients are uniformly quantized by rounding them to the nearest integer. These quantized values are then written to the output file using the `BitStream` class, which packs them efficiently according to the specified number of bits.

```
double dct_coeff = x[k];
long q_val = lround(dct_coeff);
bsOut.write_n_bits(static_cast<uint64_t>(q_val), N_BITS_QUANT);

```

Listing 4.6: Construct header metadata

Decoder (`wav_dct_dec`)

The decoding process reverses the steps taken by the encoder:

1. **Header Parsing**: The decoder first reads the metadata from the custom header of the encoded file. This information is crucial for initializing the decoder with the correct parameters (e.g., sample rate, block size).

```
int sampleRate = static_cast<int>(bsIn.read_n_bits(32));
size_t bs = static_cast<size_t>(bsIn.read_n_bits(16));
size_t nDctCoeffsPerBlock = static_cast<size_t>(bsIn.read_n_bits(16));
int N_BITS_QUANT = static_cast<int>(bsIn.read_n_bits(8));
sf_count_t nFrames = static_cast<sf_count_t>(bsIn.read_n_bits(32));

```

Listing 4.7: Read Metadata from BitStream

2. **Data Unpacking**: The decoder reads the packed data block by block, using the `BitStream` class to unpack the quantized DCT coefficients.

```
// Clear the DCT vector
for (size_t k = 0; k < bs; k++)
    x[k] = 0.0;

// Read quantized coefficients and place them in the vector
for(size_t k = 0 ; k < nDctCoeffsPerBlock ; k++) {
    uint64_t raw_val = bsIn.read_n_bits(N_BITS_QUANT);
    int32_t q_val = static_cast<int32_t>(raw_val);
    x[k] = static_cast<double>(q_val);
}

```

Listing 4.8: De-quantization and IDCT Input Setup

3. **Inverse DCT (IDCT)**: The reconstructed block of coefficients is transformed back into the time domain using the IDCT-II algorithm (`FFTW_REDFT01`). To restore the original signal's amplitude, the output of the IDCT is scaled by a factor of `2 * block_size`.

```
// Inverse DCT plan (FFTW_REDFT01 is IDCT-II)
fftw_plan plan_id = fftw_plan_r2r_1d(bs, x.data(), x.data(),
                                     FFTW_REDFT01, FFTW_ESTIMATE);
// ... inside loop ...
fftw_execute(plan_id);
double scale = 2.0 * bs;
long scaled_sample = lround(x[k] / scale);

```

Listing 4.9: Scaling and storing the reconstructed time-domain samples

4. **Output Generation**: The reconstructed time-domain blocks are concatenated to form the complete audio stream, which is then written to a standard 16-bit PCM WAV file.

```
SndfileHandle sfhOut {
    argv[argc-1],
    SFM_WRITE,
    (SF_FORMAT_WAV | SF_FORMAT_PCM_16),
    (int)nChannels,
    sampleRate
```

```
};
sfhOut.writef(samples.data(), nFramesToWrite);

```

Listing 4.10: Output WAV File Setup and Writing

4.1.3 Header Structure

The encoder prepends a 13-byte header to the compressed **BitStream**. This header is essential for the decoder to understand the structure of the file and the parameters used during encoding. The format is detailed in Table 4.1.

Table 4.1: File Format

Field	Size	Description
Sample Rate	4	The audio sample rate in Hz (e.g., 44100).
Block Size	2	The number of samples in each processing block.
Kept DCT Coefficients	2	The number of DCT coefficients stored per block.
Quantization Bits	1	The number of bits used for each quantized coefficient.
Total Frames	4	The original number of frames in the audio file.
Number of Channels	1	Number of channels of the original audio file.

Total header size: 14 bytes;

4.1.4 Usage

Encoding

The encoder takes an input WAV file and produces a compressed binary file. The compression level can be controlled via optional parameters.

```
./wav_dct_enc [options] <inputFile.wav> <outputFile.bin>

# Example: Compress 'sample.wav' with a block size of 2048 and 10% of DCT
    coefficients
/bin/wav_dct_enc -bs 2048 -frac 0.1 /test/sample.wav /test/
    sample_compressed.bin
```

Encoder Options:

- `-bs <size>`: Sets the block size (default: 1024).
- `-frac <fraction>`: Sets the fraction of DCT coefficients to keep (default: 0.2).
- `-qbits <bits>`: Sets the bits for quantization (default: 32).
- `-v`: Enables verbose mode for detailed output.

Decoding

The decoder takes a compressed binary file and reconstructs the WAV audio file.

```
./wav_dct_dec [-v (verbose)] <inputFile.bin> <outputFile.wav>

# Example: Decompress 'sample_compressed.bin' into 'reconstructed.wav'
/bin/wav_dct_dec /test/sample_compressed.bin /test/reconstructed.wav
```

4.2 Results

To test out the compression and performance results of the developed encoder and decoder, we put a single audio file through a couple tests varying different variables.

Table 4.2: DCT-Based Compression Performance for `sample01.wav`

QBits	**Ratio (:1)**	**Space Saved (%)**	**SNR (dB)**	**MSE ($\times 10^{-6}$)**	**Max Err**	**Enc Time (s)**	**Dec Time (s)**
32	2.51	60.15	0.00	0.00	0.00	0.04	0.04
16	5.02	80.08	0.00	0.00	0.00	0.04	0.04
8	10.04	90.04	0.00	0.00	0.00	0.03	0.04
4	20.07	95.02	0.00	0.00	0.00	0.03	0.03

Input File: `sample01.wav` (Original Size: 2588420 bytes)

DCT Block Size: 1024, DCT Kept Fraction: 0.2

Our DCT codec is highly effective at compression and the lossy nature of the process is currently 100% driven by the truncation of the DCT coefficients (the -frac parameter), not by the quantization bit depth (the -qbits parameter). Changing the number of QBits only affects the size of the output file, not its measured quality (in this test).