# Information and Coding

Universidade de Aveiro

Sebastian D. González, Catarina Raposo Barroqueiro, Ana Raquel Ângera Conceição

# Information and Coding

Dept. de Eletrónica, Telecomunicações e Informática

Universidade de Aveiro

sebastian.duque@ua.pt(103690), cbarroqueiro@ua.pt(103895),
arconceicao@ua.pt(98582)

October 29, 2023

# Contents

# List of Figures

# Glossário

**DCT** Discrete Cosine Transform.

**SNR** Signal-to-Noise Ratio.

# Chapter 1

# Introduction

In our Information and Coding (IC) project, each member of our group made equal contributions. We've utilized the library libsndfile, designed to facilitate the easy reading and writing of various sound file formats. The project encompasses a software package with multiple components, including a program for copying audio files (in WAV format) block by block, a class C++ (WAVHist) for generating histograms of WAV audio files, and a demonstration of the Discrete Cosine Transform (DCT), which will later be applied for compression purposes.

The project is available at the following link: `https://github.com/ElSebasdg/IC`.

# Chapter 2

# Exercise 1

In Exercise 1, we were tasked with enhancing the functionality of the `WAVHist` class to provide two additional histograms:

1. The histogram of the average of the channels (the mono version, i.e., $(L+R)/2$, also known as the MID channel) when the audio is stereo (i.e., when it contains two channels).
2. The histogram of the difference of the channels (i.e., $(L-R)/2$, also known as the SIDE channel) when the audio is stereo.

In both cases, we used integer division.

## 2.1 Code

Here is the implementation of the `WAVHist` class in the ‘WAVHist.h‘ file, to provide histograms for the mono (MID) and side (SIDE) channels, we have made the following changes to the `WAVHist` class:

- We introduced four member variables, `monoCounts` and `sideCounts`, of type `std::map<short,long>`. These maps will store the histogram data for the left, right, mono and side channels, respectively.

```
    std::map<short, long> leftCounts;  // Histograma canal esquerdo (LEFT)
    std::map<short, long> rightCounts; // Histograma canal direito (RIGHT)
    std::map<short, long> monoCounts;  // Histograma da media dos canais (MID)
    std::map<short, long> sideCounts;  // Histograma da diferenca dos canais (SIDE)
```

Listing 2.1: WAVHist Class in WAVHist.h

The `monoCounts` map stores the histogram data for the average of the channels (MID), calculated as $\frac{L+R}{2}$, while the `sideCounts` map stores the histogram data for the difference of the channels (SIDE), calculated as $\frac{L-R}{2}$.

- In the `update` method, we iterate through the audio samples and calculate the values for the MID and SIDE channels for each sample pair, as the audio is stereo (two channels). We then update the respective histograms (`monoCounts` and `sideCounts`) with these values.

```
for (long unsigned int i = 0; i < samples.size() / 2; i++) {
    // LEFT
    leftCounts[samples[2 * i]]++;

    // RIGHT
    rightCounts[samples[2 * i + 1]]++;

    // MID
    long mid = (samples[2 * i] + samples[2 * i + 1]) / 2;
    monoCounts[mid]++;

    // SIDE
    long side = (samples[2 * i] - samples[2 * i + 1]) / 2;
    sideCounts[side]++;
}
```

Listing 2.2: WAVHist Class in WAVHist.h

The addition of these histograms allows us to analyze the distribution of values in the mono and side channels, providing valuable insights into the audio data.

The exercise also required implementing coarser bins, grouping together multiple sample values. To achieve this, the class retains the original histogram binning for individual sample values. However, you can easily adapt the code to use coarser bins by modifying the `update` method to accumulate values within bins of size $2^k$.

The `saveHistogramData` method can also be adjusted to correctly save the coarser bin histograms.

```
    void saveHistogramData(const std::string& filename, const std::map<short, long>& histogram) const {
        std::ofstream outfile(filename);
        if (outfile.is_open()) {
            for (const auto& [value, counter] : histogram) {
                outfile << value << '\t' << counter << '\n';
            }
            outfile.close();
            std::cout << "Histogram data saved to " << filename << std::endl;
        } else {
            std::cerr << "Error: Unable to open file for saving histogram data." << std::endl;
        }
    }
```
Listing 2.3: WAVHist Class in WAVHist.h

By implementing coarser binning, we can reduce the granularity of the histograms, making it easier to analyze and visualize the data.

These modifications to the `WAVHist` class provide additional functionality for analyzing audio data, making it a more versatile tool for researchers and audio enthusiasts.

The class also includes methods for saving and visualizing the histograms for the mono and side channels. The `saveHistograms`, `dumpMono`, and `dumpSide` methods allow users to store and view the histogram data for further analysis.

```
    // Histograma da media dos canais (MID)
    void dumpMono() {
        std::cout << '\n';
        std::cout << "MID VALUES -----------------------------------" << std::endl;
        for (const auto& [value, counter] : monoCounts) {
            std::cout << "Value:" << std::setw(6) << value << " | Counter:" << std::setw(6) << counter << '\n';
        }
    }

    // Histograma da diferenca dos canais (SIDE)
    void dumpSide() {
        std::cout << '\n';
        std::cout << "SIDE VALUES ----------------------------------" << std::endl;
        for (const auto& [value, counter] : sideCounts) {
            std::cout << "Value:" << std::setw(6) << value << " | Counter:" << std::setw(6) << counter << '\n';
        }
    }
```
Listing 2.4: WAVHist Class in WAVHist.h

These methods facilitate data analysis and visualization, enhancing the utility of the `WAVHist` class for users who want to explore audio data in greater depth.

Overall, these modifications to the `WAVHist` class provide an enhanced and more flexible tool for audio data analysis, making it suitable for various applications that require the analysis of stereo audio data and the use of coarser bins.

## 2.2 Usage Instructions

To use the program, follow these steps:

1. Navigate to the `../sndfile-example-src` directory.
2. Create the executable file by running the following command:

```
g++ -o wav_hist wav_hist.cpp -lsndfile

```

   This command compiles the program source code (`wav_hist.cpp`) and links it with the `libsndfile` library to create an executable file named `wav_hist`.

3. Execute the program using a WAV file for analysis. Replace `sample.wav` with the path to your desired WAV file, and replace `1` with the channel you want to analyze.

```
    ./wav_hist sample.wav 1

```

4. Review the program's output, which may include histograms or other analysis results.

By following these steps, you can compile the program and execute it with your chosen WAV file and channel for analysis. After running the program, you can use Gnuplot to generate the histograms from the saved data files, such as `'mid_histogram.txt'` or `'side_histogram.txt`,' by running the following command:

```
    gnuplot 'plot_file.txt'

```

## 2.3 Results

Upon a thorough examination of 2.1 and 2.2, it becomes clear that the findings presented in 2.3 and 2.4 are in harmony with our initial expectations.

Given that the left and right channels (see 2.1 and 2.2) display analogous amplitude characteristics, we foresee that the mid channel (see 2.3), representing the average of these channels, will likewise inhabit a parallel amplitude domain. Additionally, we anticipate that the frequencies corresponding to various amplitudes will exhibit similarity across all channels.

The side channel, defined as the discrepancy between the left and right channels, should manifest the resemblance in amplitude levels between these channels. Consequently, we project that the side channel's histogram will predominantly gravitate toward the value "0."

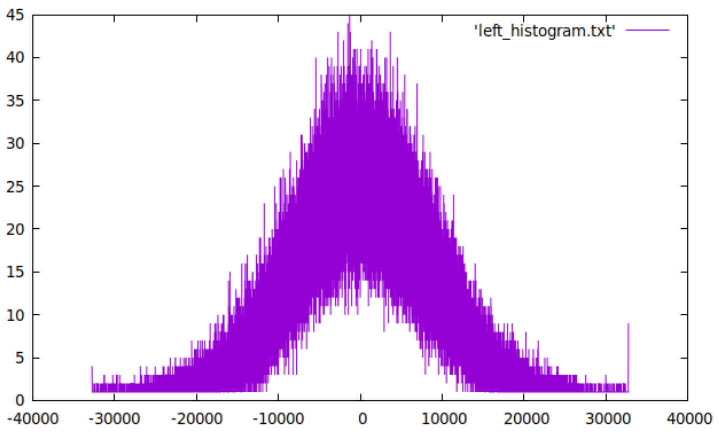

Figure 2.1: Left Channel Histogram

To generate the graphs presented in these figures, we utilized the powerful visualization tool Gnuplot.

In light of our analysis, our results affirm that both the mid (see 2.3) and side channels (see 2.4) exhibit the projected behavior as we initially postulated.

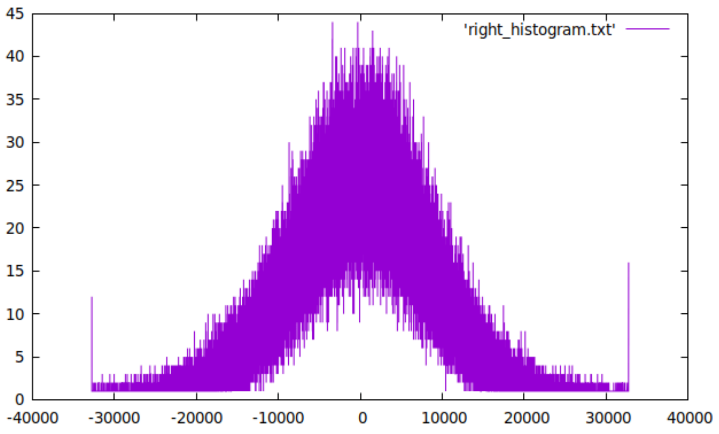

Figure 2.2: Right Channel Histogram

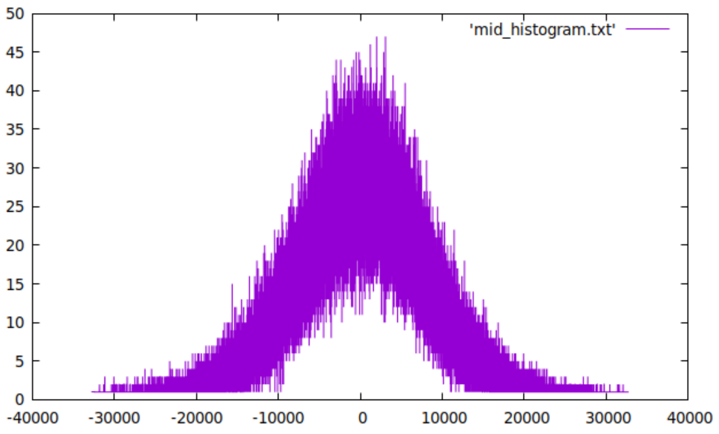

Figure 2.3: Mid Channel Histogram

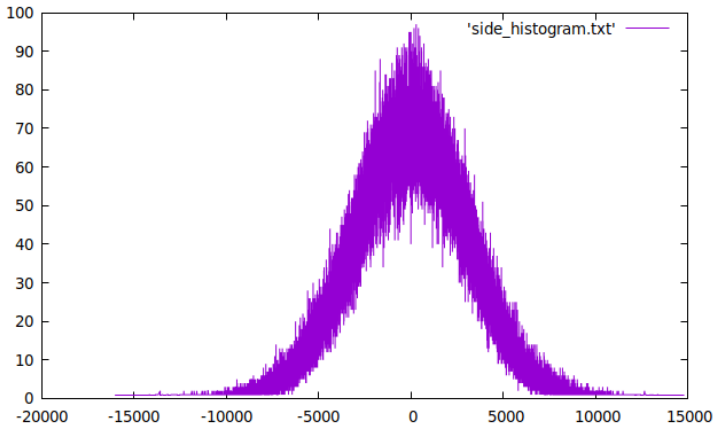

Figure 2.4: Side Channel Histogram

# Chapter 3

# Exercise 2

In Exercise 2, we implemented a program named `wav_cmp` to perform audio file comparisons and analysis. The program calculates various metrics for each channel and the average of the channels, including:

1. **Average Mean Squared Error (L2 Norm):** This metric quantifies the average squared difference between the audio file and its original version, providing insights into the overall distortion.

2. **Maximum Per Sample Absolute Error (L$\infty$ Norm):** This metric measures the maximum absolute difference between corresponding samples of the audio file and its original version. It identifies the worst-case error.

3. **Signal-to-Noise Ratio (SNR):** SNR assesses the quality of the audio file in relation to its original version. It indicates the ratio of the useful signal (original) to noise (distortion).

## 3.1 Code

In the development of the `wavcmp` program, the primary objective was to calculate and report three critical audio quality metrics for each channel and the average of the channels:

1. **Average Mean Squared Error (L2 Norm):** The L2 norm measures the average mean squared error between a given audio file and its original version. This metric helps assess the overall distortion or differences between the two audio signals. It is computed as follows:

$$\text{L2 Norm} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x(i) - \widetilde{x}(i))^2}$$

Where $x(i)$ represents the original audio sample, $\widetilde{x}(i)$ is the corresponding modified sample, and $N$ is the total number of samples.

```cpp
// Calculo da norma L2 (erro medio quadratico)
double calculateL2Norm(const vector<short>& original,
    const vector<short>& modified) {
    if (original.size() != modified.size()) {
        cerr << "Error: Audio files have different sizes
    \n";
        return -1;
    }

    double sum = 0.0;
    for (size_t i = 0; i < original.size(); ++i) {
        double diff = static_cast<double>(original[i]) -
     static_cast<double>(modified[i]);
        sum += diff * diff;
    }

    return sqrt(sum / original.size());
}
```

Listing 3.1: calculateL2Norm function in wav$_c$*mp.cpp*

The code above calculates the L2 Norm. It reads the original and modified audio samples, computes the squared differences, and then returns the square root of the average.

2. **Maximum Per Sample Absolute Error (L$\infty$ Norm):** The L$\infty$ Norm quantifies the maximum per sample absolute error between the original and modified audio files. It identifies the most significant discrepancy between the two signals. The calculation involves finding the maximum absolute difference across all samples:

$$\text{L}\infty \text{ Norm} = \max_{i=1}^{N} |x(i) - \widetilde{x}(i)|$$

```cpp
double calculateLInfinityNorm(const vector<short>&
    original, const vector<short>& modified) {
    if (original.size() != modified.size()) {
        cerr << "Error: Audio files have different sizes
    \n";
        return -1;
    }

    double maxError = 0.0;
    for (size_t i = 0; i < original.size(); ++i) {
        double error = abs(static_cast<double>(original[
    i]) - static_cast<double>(modified[i]));
        maxError = max(maxError, error);
    }

    return maxError;
}
```
Listing 3.2: calculateL2Norm function in wav$_c$*mp.cpp*

The code above calculates the L∞ Norm. It iterates through the samples, computes the absolute differences, and keeps track of the maximum error.

3. Signal-to-Noise Ratio (SNR) is a vital metric that evaluates the quality of an audio file concerning its original version. It indicates the ratio of the signal power to noise power and is often expressed in decibels (dB). The SNR calculation is as follows:

$$\text{SNR (dB)} = 10 \cdot \log_{10}\left(\frac{\sum_{i=1}^{N} x(i)^2}{\sum_{i=1}^{N}(x(i) - \widetilde{x}(i))^2}\right)$$

Where $x(i)$ represents the original audio sample, $\widetilde{x}(i)$ is the corresponding modified sample, and $N$ is the total number of samples.

```
// Calculo da Relacao Sinal-Ruido (SNR)
double calculateSNR(const vector<short>& original, const
     vector<short>& modified) {
    if (original.size() != modified.size()) {
        cerr << "Error: Audio files have different sizes
    \n";
        return -1;
    }

    double signalPower = 0.0;
    double noisePower = 0.0;

    for (size_t i = 0; i < original.size(); ++i) {
        double signalSample = static_cast<double>(
    original[i]);
        double noiseSample = static_cast<double>(
    original[i]) - static_cast<double>(modified[i]);

        signalPower += signalSample * signalSample;
        noisePower += noiseSample * noiseSample;
    }

    return 10.0 * log10(signalPower / noisePower);
}
```
Listing 3.3: calculateL2Norm function in wav$_c$*mp.cpp*

The code above calculates the Signal-to-Noise Ratio. It computes the signal and noise power and then uses these values to calculate SNR in decibels.

The program is designed to implement these calculations for audio files and provide detailed metrics for each channel and an overall average for multiple channels.

## 3.2 Usage Instructions

To run Example 2, follow these steps within the `../sndfile-example-src` directory:

1. Compile the program to create an executable file using the following command:

   ```
   g++ -o wavcmp wav_cmp.cpp -lsndfile

   ```

   This command compiles the program source code (`wav_cmp.cpp`) and links it with the `libsndfile` library to create an executable file named `wavcmp`.

2. Execute the program using audio files for comparison. You should specify the path to the sample audio file (`sample.wav`) and the corresponding output audio file (`out.wav`). The `-v` option is used for verbosity to display detailed information about the comparison.

   ```
       ./wavcmp -v sample.wav out.wav

   ```

   Replace `sample.wav` with the path to your sample audio file and `out.wav` with the path to the corresponding output audio file.

3. Review the program's output, which will include the calculated metrics for each channel and the average of the channels. The verbosity option (`-v`) provides detailed information about the comparison, including the L2 norm, L$\infty$ norm, and SNR.

By following these steps, you can use the `wavcmp` program to compare audio files and analyze various quality metrics, with the option to display detailed information about the comparison.

## 3.3 Results

To illustrate the program's functionality, here are sample results from tests conducted using the `wavcmp` program:

- **Test 1: Original vs. Modified Fast Version** In this test, the program compared an original audio file with a modified fast version. The following results were obtained:

```
Overall Metrics:
L2 Norm: 0.0671408
L$\infty$ Norm: 54604
SNR: -1.75897e-05 dB
```

These results provide valuable insights into the comparison between the original and modified audio files. Here are the key conclusions:

- **L2 Norm:** The L2 norm, which quantifies the average mean squared error between the two audio signals, is relatively low at 0.0671408. This low value suggests that the differences between the original and modified audio are minimal, indicating a high degree of similarity.
- **L$\infty$ Norm:** The L$\infty$ Norm, measuring the maximum per sample absolute error, reports a value of 54604. This high value indicates that there are individual samples in the audio that exhibit significant differences between the original and modified versions.
- **SNR:** SNR is a critical metric for audio quality assessment. In this test, the SNR value is approximately -1.75897e-05 dB. A positive SNR value indicates that the signal power (original audio) is greater than the noise power (differences or distortions). However, in this case, the negative SNR value suggests that the noise power may slightly exceed the signal power, which is unusual. It's important to note that a small negative SNR value could result from quantization or rounding errors in the modified audio.

In summary, the L2 Norm and SNR values suggest a high level of similarity between the original and modified audio, while the high L$\infty$ Norm value indicates the presence of a few samples with significant differences. The small negative SNR value might be attributed to minor quantization or rounding errors in the modified audio. Further analysis and adjustments may be needed to improve the audio quality.

- **Test 2: Original vs. Single Echo Audio** In this test, a Single Echo modified audio file was compared to the original. The results were as follows:

```
Overall Metrics:
L2 Norm: 0.0747546
L$\infty$ Norm: 26110
SNR: 9.98695e-05 dB
```

These results provide valuable insights into the comparison between the original and the Single Echo modified audio files. Here are the key conclusions:

- **L2 Norm:** The L2 norm, which quantifies the average mean squared error between the two audio signals, is notably higher at 0.0747546 compared to Test 1. This increase in L2 norm indicates substantial differences between the original and the Single Echo modified audio. It suggests that the modification has introduced more distortion, resulting in a lower degree of similarity.

– **L$\infty$ Norm:** The L$\infty$ Norm, measuring the maximum per sample absolute error, reports a value of 26110, which is considerably lower than the value in Test 1. However, it is still relatively high, signifying that there are individual samples in the audio that exhibit significant differences between the original and modified versions.

– **SNR:** The Signal-to-Noise Ratio (SNR) is a crucial metric for audio quality assessment. In this test, the SNR value is approximately 9.98695e-05 dB, which is higher than the value in Test 1. This positive SNR value suggests that the signal power (original audio) is slightly greater than the noise power (differences or distortions). It indicates a relatively better audio quality compared to Test 1, but it's still a small positive SNR value.

In summary, the higher L2 Norm value indicates a decrease in similarity between the original and Single Echo modified audio. While the L$\infty$ Norm remains relatively high, the SNR is positive, suggesting a better audio quality than in Test 1 but still with noticeable differences. The modifications in the form of a Single Echo effect have introduced some level of distortion to the audio. Further analysis and adjustments may be needed to improve the audio quality.

Chapter 4

# Exercise 3

In Example 3, we implemented a C++ class and program named `wavquant` to perform uniform scalar quantization on audio samples. This process involves reducing the number of bits used to represent each audio sample.

## 4.1 Usage Instructions

To run Example 3, follow these steps within the `../sndfile-example-src` directory:

1. Compile the program to create an executable file using the following command:

   ```
   g++ -o wavquant wav_quant.cpp -lsndfile

   ```

   This command compiles the program source code (`wav_quant.cpp`) and links it with the `libsndfile` library to create an executable file named `wavquant`.

2. Execute the program using audio files for quantization. You should specify the path to the sample audio file (`sample.wav`) and the path for the corresponding output audio file (`out.wav`). Additionally, provide an integer value as an argument to specify the number of bits for quantization. For example, to quantize to 4 bits:

   ```
   ./wavquant sample.wav out.wav 4

   ```

   Replace `sample.wav` with the path to your sample audio file, `out.wav` with the path for the output file, and `4` with the desired number of bits.

3. Test the quantization with different bit values and listen to the resulting audio files. You can use the same `wavquant` program with different bit

values to observe the impact on audio quality. For example, to quantize to 3 bits:

```
./wavquant sample.wav out.wav 3

```

And similarly for 2 bits and 1 bit:

```
./wavquant sample.wav out.wav 2
./wavquant sample.wav out.wav 1

```

Listening to the output files for different quantization bit values will help you understand the effect of quantization on audio quality.

By following these steps, you can use the `Wavquant` program to perform uniform scalar quantization on audio samples and observe the results with different quantization bit values.

## 4.2 Code

The class called WavQuant performs audio quantization. Has said before, in this case, the input values are audio samples, and the output values are quantized audio samples. The class has the following public member functions:

- `WavQuant:`Its the constructor that takes an integer argument numBits, which specifies the number of bits to use for quantization. It initializes the numBits and quantizationLevels member variable.
- `short quantizeSample:`Takes a vector of audio samples as input and returns a vector of quantized audio samples.
- `std::vector<short> quantizeSamples:`Takes a single audio sample as input and returns the quantized version of that sample.
- `short getQuantizationLevels:`Takes a single audio sample as input and returns the quantized version of that sample.
- `void saveQuantizedSamplesToTextFile:` Converts the quantized audio samples to a text file to analyse in Gnuplot.

```cpp
class WavQuant {
public:
    WavQuant(int numBits);
    short quantizeSample(short sample);
    std::vector<short> quantizeSamples(const std::vector<short>& samples);
    short getQuantizationLevels() const;
    void saveQuantizedSamplesToTextFile(const std::vector<short>& quantizedSamples, const std::string& filename);
private:
    int numBits;
    short quantizationLevels;
};
WavQuant::WavQuant(int numBits) {
    this->numBits = numBits;
    this->quantizationLevels = static_cast<short>(pow(2, numBits) - 1);
}
short WavQuant::quantizeSample(short sample) {
    return static_cast<short>(round(static_cast<double>(sample) * quantizationLevels / 32767.0));
}
std::vector<short> WavQuant::quantizeSamples(const std::vector<short>& samples) {
    std::vector<short> quantizedSamples;
    quantizedSamples.reserve(samples.size());
    for (short sample : samples) {
        quantizedSamples.push_back(quantizeSample(sample));
    }
    return quantizedSamples;
}
short WavQuant::getQuantizationLevels() const {
    return quantizationLevels;
}
void WavQuant::saveQuantizedSamplesToTextFile(const std::vector<short>& quantizedSamples, const std::string& filename) {
    std::ofstream outFile(filename);
    if (outFile.is_open()) {
        for (short sample : quantizedSamples) {
            outFile << sample << std::endl;
        }
        outFile.close();
    } else {
        std::cerr << "Error: Unable to open the output text file for writing." << std::endl;}}
```
Listing 4.1: wav_quant.h Class

Our main Class using the 4.2 `wavquant.h` class to quantize the input audio samples using a specified number of bits and then save the quantized samples into a text file for later analysis.

```cpp
WavQuant quantizer(numBits);

    vector<short> samples(FRAMES_BUFFER_SIZE * sfh.channels());
    vector<short> quantizedSamples;

    SndfileHandle sfhOut(outputFile, SFM_WRITE, sfh.format(), sfh.channels(), sfh.samplerate());

    while (size_t nFrames = sfh.readf(samples.data(), FRAMES_BUFFER_SIZE)) {
        // Quantize the samples
        quantizedSamples = quantizer.quantizeSamples(samples);

        // Save the quantized samples to a text file
        const std::string textOutputFile = "quantized_samples.txt";
        quantizer.saveQuantizedSamplesToTextFile(quantizedSamples, textOutputFile);

        // Write the quantized samples to the WAV file
        sfhOut.writef(quantizedSamples.data(), nFrames);
    }

    cout << "Quantization completed with " << numBits << " bits per sample.\n";

    return 0;
```

Listing 4.2: wav_quant.cpp main

## 4.3 Results

We tested the results for three different examples: 8bits, 4 bits and 2 bits. And analysed it in Gnuplot. This were the histograms we obtained for each individual example:

- `8 bits`

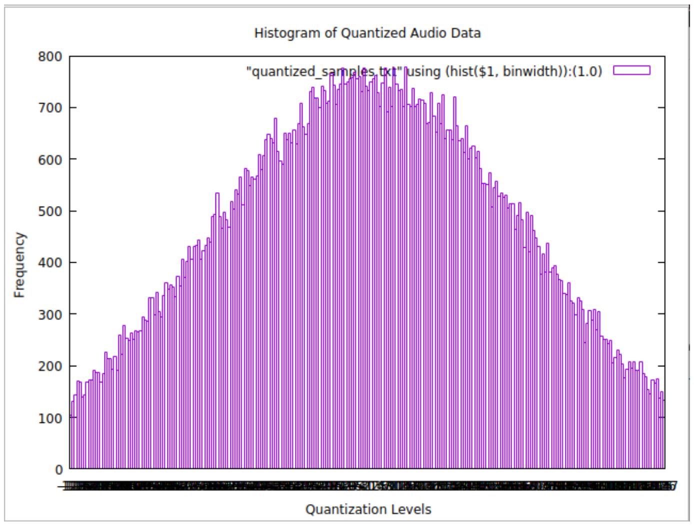

Figure 4.1: 8bits resolution audio file

- `4 bits`

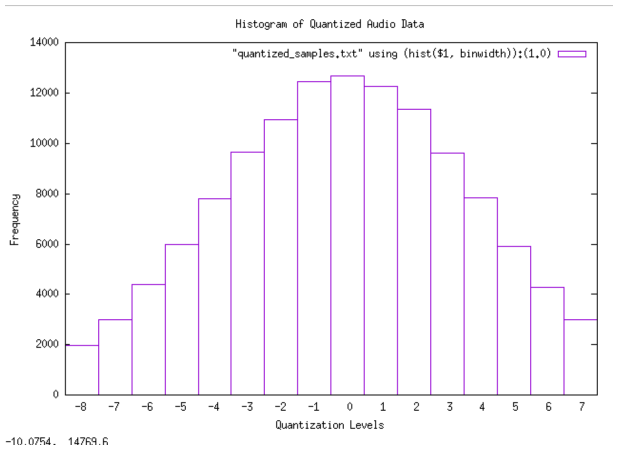

Figure 4.2: 4bits resolution audio file

- `2 bits`

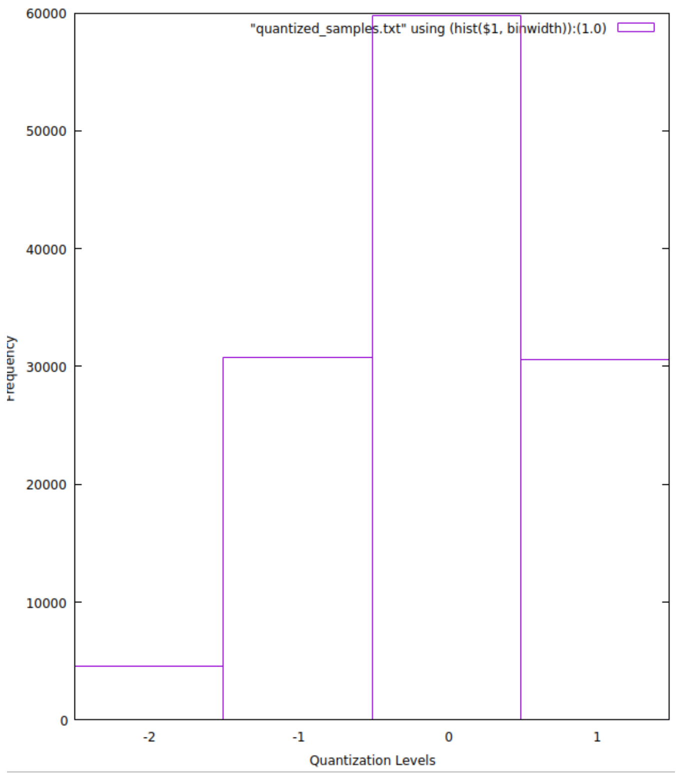

Figure 4.3: 2bits resolution audio file

Seeing the results, we can conclude that reducing the number of bits also reduces the accuracy of the audio signal, which can result in a loss of quality.

When an audio signal is quantized, each sample is rounded to the nearest value that can be represented with the available number of bits. For example, if an audio signal is quantized to 8 bits 4.1, each sample can have one of 256 possible values. If the same signal is quantized to 4 bits 4.2, each sample can have one of only 16 possible values. This means that the quantized signal will be less accurate and will have a lower dynamic range than the original signal.

Audio quantization can be useful in some situations, such as when the audio signal is being compressed for storage or transmission. However, it is important to be aware of the potential loss of quality that can result from quantization, and to use the appropriate number of bits for the specific use and application.

Chapter 5

# Exercise 4

In Example 4, we implemented a program named `wav_effects` to apply various audio effects to audio files. These effects can include single and multiple echoes, amplitude modulation, and time-varying delays.

## 5.1 Code

The program includes the necessary header files and libraries and defines a constant `FRAMES_BUFFER_SIZE` for processing audio data. The `main` function serves as the entry point of the program and opens the input WAV file and the output file for writing.

It prompts the user to choose an audio effect from the menu and collects relevant parameters (e.g., delay, gain or frequency). The available effects in the menu are as follows:

- **Single Echo (Option 1)**: Adds a single echo to the audio with user-defined delay and gain.
- **Multiple Echo (Option 2)**: Adds multiple echoes to the audio with user-defined delay and gain.
- **Amplitude Modulation (Option 3)**: Modulates the audio amplitude with a user-defined frequency.
- **Time-Varying Delays (Option 4)**: Introduces time-varying delays to the audio based on a list of delay times.
- **Slow Motion (Option 5)**: Slows down the audio playback.
- **Fast Forward (Option 6)**: Speeds up the audio playback.

The formulas studied in the theory lessons were used for the first three options:

$$\texttt{Single echo:} \quad y(n) = x(n) + \alpha \times x(n - delay)$$

$$\texttt{Multiple echos:} \quad y(n) = x(n) + \alpha \times y(n - delay)$$

with $y$ being the new modified sample and $x$ the original sample, in addition we divide the modified sample by $(1 + gain)$ to guarantee that the modified sample does not go above the maximum value.

$$\texttt{Amplitude modulation:} \quad y(n) = x(n) \times \cos(2 \times \pi \times \frac{f}{fa}) \times n)$$

with $f$ being the user-defined frequency and $fa$ the frequency of the original sample

```cpp
void applySingleEcho(SndfileHandle& sfhIn, SndfileHandle&
    sfhOut, int delay, float gain) {
    vector<short> samples(FRAMES_BUFFER_SIZE * sfhIn.
    channels());
    vector<short> echoBuffer;

    while (size_t nFrames = sfhIn.readf(samples.data(),
    FRAMES_BUFFER_SIZE)) {
        samples.resize(nFrames * sfhIn.channels());

        for (int i = 0; i < (int)samples.size(); i++) {
            if (i >= delay) {   //y(n) = x(n) + a * x(n - k)
                echoBuffer.push_back((samples.at(i) + gain *
     samples.at(i - delay)) / (1 + gain));
            } else {
                echoBuffer.push_back(samples.at(i));
            }
        }
    }

    sfhOut.writef(echoBuffer.data(), echoBuffer.size() /
    sfhIn.channels());
}
```

Listing 5.1: Single echo in wav_effects.cpp

```cpp
void applyMultipleEchoes(SndfileHandle& sfhIn, SndfileHandle& sfhOut, int delay, float gain) {
    vector<short> samples(FRAMES_BUFFER_SIZE * sfhIn.channels());
    vector<short> echoBuffer;

    while (size_t nFrames = sfhIn.readf(samples.data(), FRAMES_BUFFER_SIZE)) {
        samples.resize(nFrames * sfhIn.channels());

        for (int i = 0; i < (int)samples.size(); i++) {
            if (i >= delay) {   // y(n) = x(n) + a * y(n - k)
                echoBuffer.push_back((samples.at(i) + gain * echoBuffer.at(i - delay)) / (1 + gain));
            } else {
                echoBuffer.push_back(samples.at(i));
            }
        }
    }

    sfhOut.writef(echoBuffer.data(), echoBuffer.size() / sfhIn.channels());
}
```

Listing 5.2: Multiple echos in wav_effects.cpp

```cpp
void applyAmplitudeModulation(SndfileHandle& sfhIn, SndfileHandle& sfhOut, float freq) {
    vector<short> samples(FRAMES_BUFFER_SIZE * sfhIn.channels());
    vector<short> outBuffer;

    while (size_t nFrames = sfhIn.readf(samples.data(), FRAMES_BUFFER_SIZE)) {
        samples.resize(nFrames * sfhIn.channels());

        for (int i = 0; i < (int)samples.size(); i++) {
            // y(n) = x(n) * cos(2*pi * (f/fa) * n)
            outBuffer.push_back(samples.at(i) * cos(2 * M_PI * (freq / sfhIn.samplerate()) * i));
        }
    }

    sfhOut.writef(outBuffer.data(), outBuffer.size() / sfhIn.channels());
}
```

Listing 5.3: Amplitude modulation in wav_effects.cpp

The `Time-Varying Delays` function reads an audio file and adds a dynamic echo effect to it. It does this by creating multiple delayed copies of the audio, each with varying delay times specified in seconds. These delayed copies are then mixed with the original audio.

```cpp
void applyTimeVaryingDelays(SndfileHandle& sfhIn, SndfileHandle& sfhOut, vector<double> delayTimesSeconds, double sampleRate) {
    vector<short> samples(FRAMES_BUFFER_SIZE * sfhIn.channels());
    size_t nFrames;

    while (nFrames = sfhIn.readf(samples.data(), FRAMES_BUFFER_SIZE)) {
        samples.resize(nFrames * sfhIn.channels());

        // Atrasos Variaveis
        vector<vector<short>> delayedSamples(delayTimesSeconds.size(), vector<short>(nFrames, 0));

        for (size_t i = 0; i < delayTimesSeconds.size(); ++i) {
            size_t delaySamples = static_cast<size_t>(delayTimesSeconds[i] * sampleRate);

            for (size_t j = delaySamples; j < nFrames; ++j) {
                delayedSamples[i][j] = samples[j - delaySamples];
            }
        }

        // Misturar as amostras com atraso
        for (size_t i = 0; i < nFrames; ++i) {
            for (size_t j = 0; j < delayTimesSeconds.size(); ++j) {
                samples[i] += delayedSamples[j][i];
            }
        }

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

Listing 5.4: Time-Varying Delays in wav_effects.cpp

The `Slow Motion` function achieves the slow motion effect by reducing the sample rate of the audio and adding extra interpolated samples between the original samples to extend the audio duration. The extent of slowdown is controlled by the `slowdownFactor` parameter.

```cpp
void applySlowMotion(SndfileHandle& sfhIn, SndfileHandle&
    sfhOut, double slowdownFactor) {
    vector<short> samples(FRAMES_BUFFER_SIZE * sfhIn.
    channels());
    vector<short> slowedSamples;

    size_t nFrames;

    while (nFrames = sfhIn.readf(samples.data(),
    FRAMES_BUFFER_SIZE)) {
        samples.resize(nFrames * sfhIn.channels());

        // Reduz a taxa de amostragem
        for (size_t i = 0; i < nFrames; ++i) {
            slowedSamples.push_back(samples[i]);

        // Adicionar amostras extras para prolongar o audio
            for (int j = 1; j < slowdownFactor; ++j) {
                short interpolatedSample = (samples[i] +
    samples[i + 1]) / 2;
                slowedSamples.push_back(interpolatedSample);
            }
        }
    }

    sfhOut.writef(slowedSamples.data(), slowedSamples.size()
     / sfhIn.channels());
}
```

Listing 5.5: Slow Motion in wav_effects.cpp

The `Fast Forward` function achieves the fast forward effect by skipping audio samples based on the `speedupFactor` parameter. The higher the `speedupFactor`, the faster the audio playback.

```cpp
void applyFastForward(SndfileHandle& sfhIn, SndfileHandle& sfhOut, double speedupFactor) {
    vector<short> samples(FRAMES_BUFFER_SIZE * sfhIn.channels());
    vector<short> spedUpSamples;

    size_t nFrames;
    while (nFrames = sfhIn.readf(samples.data(), FRAMES_BUFFER_SIZE)) {
        samples.resize(nFrames * sfhIn.channels());

        // Salta amostras com base no fator de aceleracao
        for (size_t i = 0; i < nFrames; i += speedupFactor) {
            spedUpSamples.push_back(samples[i]);
        }
    }


    sfhOut.writef(spedUpSamples.data(), spedUpSamples.size() / sfhIn.channels());
}
```

Listing 5.6: Fast Forward in wav_effects.cpp

## 5.2 Usage Instructions

To run Example 4, follow these steps within the `../sndfile-example-src` directory:

1. Compile the program to create an executable file using the following command:

   ```
   g++ -o wav_effects wav_effects.cpp -lsndfile

   ```

   This command compiles the program source code (`wav_effects.cpp`) and links it with the `libsndfile` library as well as the math library to create an executable file named `wav_effects`.

2. Execute the program using audio files and choose an effect by providing an integer value (1-6). For example, to apply the first effect:

```
    ./wav_effects sample.wav out.wav
    Select an effect:
    1. Single Echo
    2. Multiple Echo
    3. Amplitude Modulation
    4. Time Varying Delays
    5. Slow Motion
    6. Fast Forward
    Enter your choice (1-6): 1

```

   For the first two effects, choose values for the delay and gain. For the third effect choose the frequency:

```
    ./wav_effects sample.wav out.wav
    Select an effect:
    1. Single Echo
    2. Multiple Echo
    3. Amplitude Modulation
    4. Time Varying Delays
    5. Slow Motion
    6. Fast Forward
    Enter your choice (1-6): 1
    Enter the delay (e.g., 44100): 44100
    Enter the gain (e.g., 0.8): 0.8
    Effect applied successfully.

```

   Replace `sample.wav` with the path to your sample audio file, `out.wav` with the path for the output file, and `1` with the desired effect code.

3. Test the program with different effect values and listen to the resulting audio files. You can use the same `wav_effects` program with different effect codes to explore and experience various audio effects. For example, to apply the third effect:

```
    ./wav_effects sample.wav out.wav
    Select an effect:
    1. Single Echo
    2. Multiple Echo
    3. Amplitude Modulation
    4. Time Varying Delays
    5. Slow Motion
    6. Fast Forward
    Enter your choice (1-6): 3
    Enter the frequency (e.g., 1Hz): 1
    Effect applied successfully.

```

And similarly for effect codes 4, 5 and 6 without specifying any values:

```
    ./wav_effects sample.wav out.wav
    Select an effect:
    1. Single Echo
    2. Multiple Echo
    3. Amplitude Modulation
    4. Time Varying Delays
    5. Slow Motion
    6. Fast Forward
    Enter your choice (1-6): 6
    Effect applied successfully.

```

Listening to the output files with different effect values allows you to experience and assess the audio effects.

By following these steps, you can use the `wav_effects` program to apply various audio effects to audio files and experiment with different effect codes to achieve different results.

## 5.3 Results

We created 6 audio files for each of the effects: single echo with 0.8 of gain and 44100Hz of delay (`out_single.wav`), multiple echos with 0.8 of gain and 44100Hz of delay (`out_multiple.wav`), amplitude modulation of 1Hz (`out_amplitude.wav`), time-varying delays, slow motion and fast forward. When listening to the audio files, we can clearly see that both `out_single.wav` and `out_multiple.wav` have an echo effect, but the latter creates a more distorted audio file.

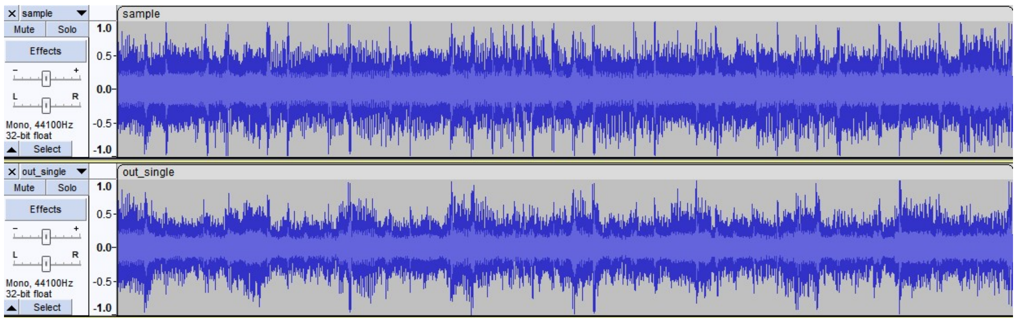

Figure 5.1: Comparing sample audio with the single echo output (0.8 of gain and 44100Hz of delay)

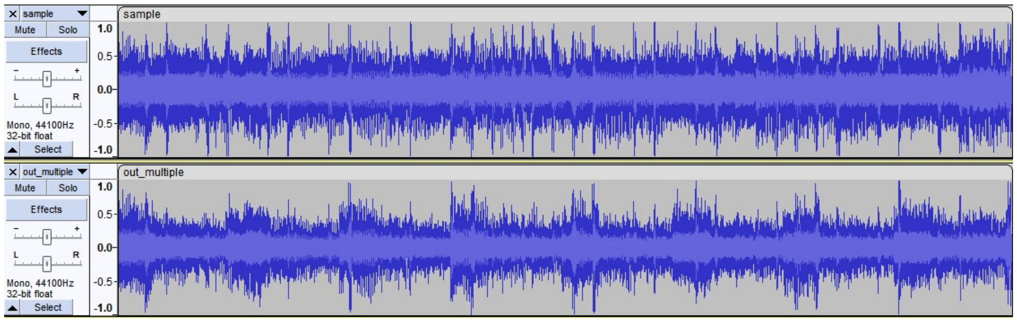

Figure 5.2: Comparing sample audio with the multiple echos output (0.8 of gain and 44100Hz of delay)

We can detect in 5.1) and 5.2) that the audios have been modified.

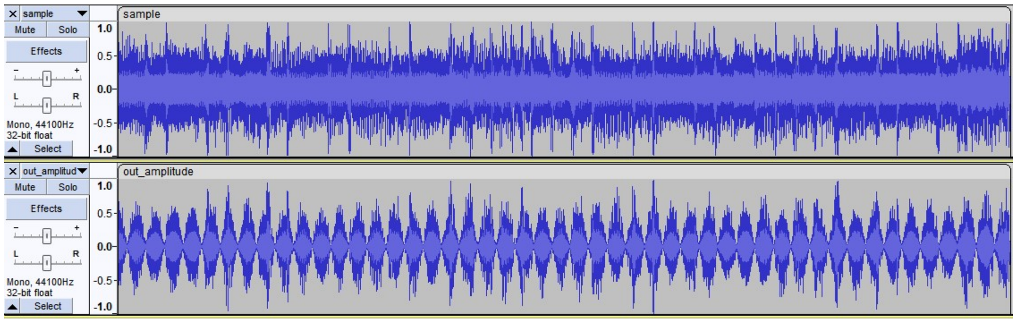

Figure 5.3: Comparing sample audio with the amplitude modulation output (of 1Hz)

As expected, the output audio for amplitude modulation (`out_multiple.wav` in 5.3) demonstrates variations in amplitude over time, following the 1Hz harmonic wave.

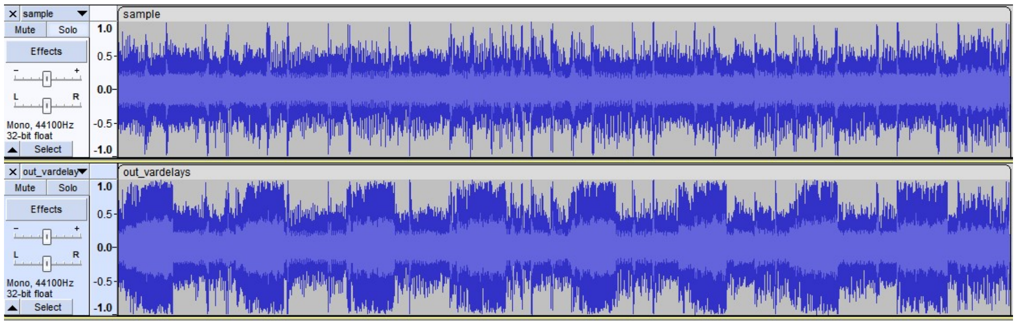

Figure 5.4: Comparing sample audio with the time-varying delays output

The waveform for the time-varying delays effect is characterized by a repeating pattern of echoes with varying delays

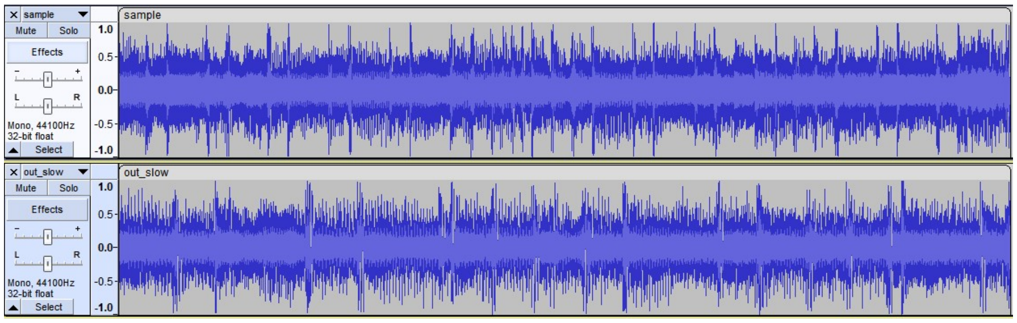

Figure 5.5: Comparing sample audio with the slow motion output

The waveform of the output audio with the slow motion effect (`out_slow.wav` in 5.5) shows a lower-pitched and elongated representation of the original audio.

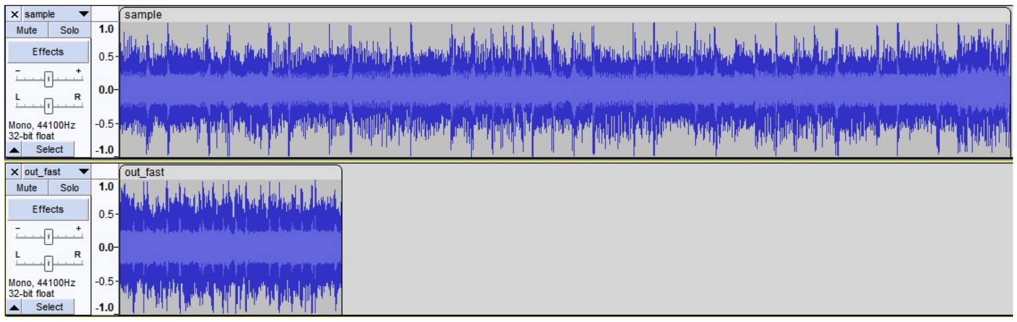

Figure 5.6: Comparing sample audio with the fast forward output

The output audio with the fast forward effect (`out_fast.wav` in 5.6) shows compression and shorter duration. It visually represents the faster playback of the audio, creating a distinctive and recognizable pattern on the waveform.

# Chapter 6

# Exercise 5

In Example 5, we developed a C++ class named `BitStream`, to facilitate the reading and writing of binary data at the bit level from and to a file.

## 6.1 Code

The `BitStream` class encapsulates a set of functionalities for working with binary data. It allows users to manipulate individual bits, sequences of bits, and strings in a file. The class ensures efficient bit-level reading and writing while managing file operations and error handling. To effectively use the `BitStream` class, create an instance of the `BitStream` class with the desired filename and mode, write (true) or read (false):

```cpp
BitStream myBitStream("example.bin", true); // For writing
BitStream myBitStream("example.bin", false); // For reading
```

The `BitStream` class offers a trio of essential read functions, namely *readBit*, *readNBits*, and *readString*. The *readBit* function efficiently retrieves a single bit from the file and refills its internal buffer when necessary, ensuring that bits are read one at a time while managing the buffer. On the other hand, the *readNBits* function reads a specified number of bits (up to 64) from the file and accumulates them, from the most significant to the least significant. Lastly, *readString* serves as a tool to reconstruct strings from binary data in the file. It reads 8-bit segments and assembles them into a string, which is particularly useful for reading textual information encoded as binary data in the file.

```
bool readBit() {
        if (bufferLength == 0 && !eofFlag) {
            readByte();
        }

        bool bit = (buffer >> (bufferLength - 1)) & 1;
        bufferLength--;
        return bit;
}

uint64_t readNBits(uint8_t numBits) {
    if (numBits <= 64) {
        uint64_t result = 0;
        for (int i = numBits - 1; i >= 0; i--) {
            result = (result << 1) | readBit();
        }
        return result;
    } else {
        std::cerr << "Numero de bits para ler excede 64" << std::endl;
        return 0;
    }
}

std::string readString(size_t length) {
    std::string result;
    for (size_t i = 0; i < length; i++) {
        char c = static_cast<char>(readNBits(8));
        result += c;
    }
    return result;
}
```

Listing 6.1: Read Functions in BitStream.h

Similarly to the reading process, the `BitStream` class also includes a set of write functions to complement its read capabilities, *writeBit*, *writeNBits*, and *writeString*. The *writeBit* function efficiently writes individual bits, allowing users to compose binary data one bit at a time. This function manages an internal buffer and flushes it to the file when a complete byte is ready for writing. The *writeNBits* function enables the composition of binary data by writing a specified number of bits (up to 64) from a given value. It carefully arranges these bits, from the most significant to the least significant, before appending them to the file. Furthermore, the *writeString* function offers a convenient way to store strings as binary data. It encodes each character in the string as an 8-bit segment and appends them to the file.

```cpp
void writeBit(bool bit) {
    buffer |= (bit << (7 - bufferLength));
    bufferLength++;

    if (bufferLength == 8) {
        file.put(buffer);
        buffer = 0;
        bufferLength = 0;
    }
}

void writeNBits(uint64_t data, uint8_t numBits) {
    if (numBits <= 64) {
        for (int i = numBits - 1; i >= 0; i--) {
            writeBit((data >> i) & 1);
        }
    } else {
        std::cerr << "Numero de bits para escrever excede 64
    " << std::endl;
    }
}

void writeString(const std::string& str) {
    for (char c : str) {
        writeNBits(c, 8);
    }
}
```

Listing 6.2: Write Functions in BitStream.h

This dual capability to read and write both individual bits and multibit sequences makes the `BitStream` class exceptionally versatile, suitable for a broad spectrum of binary data manipulation tasks, ranging from low-level bit operations to encoding and decoding more complex data, such as integers and textual information, in binary format.

## 6.2 Usage Instructions

To test Example 5, follow these steps within the `../sndfile-example-src` directory:

1. Compile the program `test_ex5` to create an executable test file using the following command:

   ```
   g++ -o test_ex5 test_ex5.cpp

   ```

   The BitStream class, if implemented correctly, when given only two bytes, with hexadecimal values A7 and C0 and asked to return the first 11 bits from this file, should return 10100111110. The program `test_ex5` checks if the BitStream class is working, running this exact test.

2. Execute the program:

```
    ./test_ex5

```

3. Review the program's output, if the first 11 bits read match the expected value, it returns "Test passed!", if not, "Test failed!".

# Chapter 7

# Exercise 6

In this exercise we had to develop and implement an encoder and a decoder using the bitStream.h class.

The encoder should be able to convert a text file containing only 0s and 1s into the binary equivalent (each byte of the binary file should represent eight of the bits in the text file).

First, the program receives the `.txt` file and checks if the file actually contains only 0s and 1s and then performs the encoding operation.Afterwards the `BitStream` object is created with the output bin file name mentioned as an argument and followed by the flag true(writeMode).

```
void encoder(const std::string& inputFileName, const std::
    string& outputFileName) {
std::ifstream inputFile(inputFileName, std::ios::in);
if (!inputFile.is_open()) {
    std::cerr << "Error opening input file." << std::endl;
    return;
}
BitStream outputBitStream(outputFileName, true);
char bit;
while (inputFile.get(bit)) {
    if (bit != '0' && bit != '1') {
        std::cerr << "Input file should contain only 0s and
    1s." << std::endl;
        return;
    }
    outputBitStream.writeBit(bit == '1');
}
inputFile.close();
outputBitStream.close();
}
```

Listing 7.1: File Encoder

The decoder implements the opposite operation, converts a text file into its 0s and 1s equivalent.

First, the program receives the `.bin` and then performs the decoding operation.The `BitStream` object is created with the output txt file name mentioned as an argument and followed by the flag false(ReadMode).

```cpp
void decoder(const std::string& inputFileName, const std::string& outputFileName) {
    BitStream inputBitStream(inputFileName, false);
    if (!inputBitStream.is_open()) {
        std::cerr << "Error opening the input file." << std::endl;
        return;
    }
    std::ofstream outputFile(outputFileName, std::ios::out);
    if (!outputFile.is_open()) {
        std::cerr << "Error creating the output file." << std::endl;
        return;
    }
    // escrever no ficheiro destino
    while (!inputBitStream.eof()) {
        bool bit = inputBitStream.readBit();
        if (inputBitStream.eof()) break;
        if (bit) {
            outputFile << '1';
        } else {
            outputFile << '0';
        }
    }
    inputBitStream.close();
    outputFile.close();
}
```

Listing 7.2: File Decoder

The main goal of these two programs is to test the BitStream class developed before, although they might also be used in the future for debugging the output of other codecs.

## 7.1 Usage Instructions

To test Example 6, follow these steps within the `../sndfile-example-src` directory:

1. Compile the program `decoder_encoder.cpp` to create an executable test file using the following command:

   ```
   g++ decoder_encoder.cpp -o decoder_encoder

   ```

   Depending in what you wanna use, the `Decoder` or the `Encoder` you have to specify it when executing the program.

2. If you want to use the `Encoder` you execute the program followed by the `.txt` file you want do encode plus the `.bin` file to store the final 0s and 1s output:

   - Execute the program:

     ```
     ./decoder_encoder encoder 01_file.txt encoded.bin

     ```

3. If you want to use the `Decoder` you execute the program followed by the `.bin` file you want do decode flowed by the `.txt` file to store the final output translated:

   - Execute the program:

     ```
     ./decoder_encoder decoder encoded.bin 01_file.txt

     ```

## 7.2 Results

The results are very straightforward. For example, lets first use the `Encoder`. In this case the file we are going to decode(01_file.txt) contains the following sequence of 0s and 1s:

- "0100100001100101011011000110110001101111001000000101
  0111011011110111001001101100011001000010000100001010"

The result after executing the Encoder will be the following String in the encoded.bin:

- "Hello World!"

The opposite operation will occur if we apply the `Decoder` to the encoded.bin file.

# Chapter 8

# Exercise 7

In exercise number 7 we had to implement a lossy codec for mono audio files (i.e., with a single channel), based on the Discrete Cosine Transform.

First, the audio is processed on a block by block basis. Each block is converted using the DCT, the coefficients appropriatelly quantized, and the bits written to a file, using the BitStream class.

To create the compressed audio correctly, first, you have to run the encoder on the desired audio you to create the binary file, for later the decoder to reconstruct an approximate version of the original audio compressed.

## 8.1 Usage Instructions

Depending if you want to use the Dct_encoder or the Dct_decoder you need to choose. For that, we created two different files for the Decoder and the Encoder.

1. Compile the program `dct_encoder.cpp` to create an executable test file using the following command:

```
g++ -o dct_encoder dct_encoder.cpp -lfftw3 -lsndfile

```

2. To use encoder to generate binary file of the audio:

```
./dct_encoder -v -bs 1024 -frac 0.2 sample.wav
    encoded_output.dat

```

   The arguments are by order: -v (verbose), blockSize (def 1024), dctFraction (def 0.2), the audio file to convert and finally the encoded file output.

3. Compile the program `dct_decoder.cpp` to create an executable test file using the following command:

```
g++ -o dct_decoder dct_decoder.cpp -lfftw3 -lsndfile

```

4. To use the decoder to reconstruct the binary file of the audio, where the argv[1]("2" in this example) is the number of samples:

```
./dct_decoder 2 encoded_output.dat output_audio.wav

```

The arguments are, by order: -v (verbose), blockSize (def 1024), dctFraction (def 0.2), the audio file to convert and, finally, the encoded file output.

## 8.2 Code

The `dct_encoder.cpp` is a program designed for audio compression,using the Discrete Cosine Transform (DCT) encoding technique. It takes into account mono audio files, processes audio data in blocks, and retains the most relevant DCT coefficients for efficient compression. It provides a robust solution for audio compression, allowing for command-line arguments to control its behavior, such as enabling verbose mode, specifying the block size (-bs), and the DCT fraction (-frac). If no arguments are provided, default values are used.

```
vector<short> samples(nChannels * nFrames);
    sfhIn.readf(samples.data(), nFrames);
    size_t nBlocks{static_cast<size_t>((nFrames) / bs)};
    // Do zero padding, if necessary
    samples.resize(nBlocks * bs * nChannels);
    // Vector for holding all DCT coefficients, channel by
   channel
    vector <vector<double>> x_dct(nChannels, vector<double>(
   nBlocks * bs));
    // Vector for holding DCT computations
    vector<double> x(bs);
    // Direct DCT
    fftw_plan plan_d = fftw_plan_r2r_1d(bs, x.data(), x.data
   (), FFTW_REDFT10, FFTW_ESTIMATE);
    for (size_t n = 0; n < nBlocks; n++){
        for (size_t k = 0; k < bs; k++)
            x[k] = samples[(n * bs + k)];
        fftw_execute(plan_d);
        // Keep only "dctFrac" of the "low frequency"
   coefficients
        for (size_t k = 0; k < bs * dctFrac; k++) {
            x_dct[0][n * bs + k] = x[k] / (bs << 1);
        }
    }
    BitStream bitStream{argv[argc - 1], true}; //writing
    bitStream.writeNBits(static_cast<uint64_t>(nFrames),24);
    bitStream.writeNBits(static_cast<uint64_t>(nChannels)
   ,16);
    bitStream.writeNBits(static_cast<uint64_t>(bs), 16);
    bitStream.writeNBits(static_cast<uint64_t>(sfhIn.
   samplerate()),16);
    bitStream.writeNBits(static_cast<uint64_t>(round(dctFrac
    * 100)),16);
    for (size_t n = 0; n < nBlocks; n++){

        // Keep only "dctFrac" of the "low frequency"
   coefficients
        for (size_t k = 0; k < bs * dctFrac; k++) {
            auto value = round(x_dct[0][n * bs + k]);
            bitStream.writeNBits(static_cast<uint64_t>(value), 16);
        }
    }
    bitStream.close(); // Close the bitstream
```
Listing 8.1: File DCT Encoder

A vector called samples is created to store the audio samples, with the size determined by the number of channels and frames. The number of blocks needed for the DCT computation is calculated based on the number of frames and a block size (bs). The samples vector is resized to accommodate zero padding if necessary. Two-dimensional vectors x_dct and x are created to hold the DCT coefficients and computations, respectively. The DCT is performed on the input audio samples in a nested loop, with the outer loop iterating over the blocks, the middle loop iterating over the channels, and the inner loop iterating over the DCT coefficients.

The resulting DCT coefficients are stored in the x_dct vector. The inverse DCT is then applied to the DCT coefficients, and the resulting audio samples are stored in the samples vector. Finally, the samples vector is written to an output file using the SndfileHandle class.

```cpp
vector<short> samples(nChannels * nFrames);
  size_t nBlocks{static_cast<size_t>(ceil(static_cast<double>(nFrames) / bs))};
  // Do zero padding, if necessary
  samples.resize(nBlocks * bs * nChannels);
  // Vector for holding all DCT coefficients, channel by channel
  vector <vector<double>> x_dct(nChannels, vector<double>(nBlocks * bs));
  // Vector for holding DCT computations
  vector<double> x(bs);
  cout << x_dct[0][0];
  for (size_t n = 0; n < nBlocks; n++) {
    for (size_t c = 0; c < nChannels; c++) {
      for (size_t k = 0; k < bs * dctFrac; k++) {
        int16_t value = static_cast<int16_t>(bitStream.readNBits(16));
        cout << value << endl;
        x_dct[c][n * bs + k] = static_cast<double>(value);
      }
    }
  }
  SndfileHandle sfhOut{argv[argc - 1], SFM_WRITE, 65538,
            nChannels, sr};
  cout << "FORMAT " << sfhOut.format() << endl;
  cout << "Channels " << sfhOut.channels() << endl;
  cout << "SR " << sfhOut.samplerate() << endl;
  // Inverse DCT
  fftw_plan plan_i = fftw_plan_r2r_1d(bs, x.data(), x.data(), FFTW_REDFT01, FFTW_ESTIMATE);
  for (size_t n = 0; n < nBlocks; n++)
    for (int c = 0; c < nChannels; c++) {
      for (size_t k = 0; k < bs; k++) {
        x[k] = x_dct[c][n * bs + k];
```

```
            }
            fftw_execute(plan_i);
            for (size_t k = 0; k < bs; k++)
                samples[(n * bs + k) * nChannels + c] =
    static_cast<short>(round(x[k]));
        }
    sfhOut.writef(samples.data(), nFrames);
```

Listing 8.2: File DCT Decoder

## 8.3 Results

Listening the `output_audio.wav` obtained after using the Encoder and the Decoder we can notice an obvious loss in sound quality due to the removal of certain audio information.