UNIVERSIDADE DE AVEIRO

DEPARTAMENTO DE ELETRÓNICA, TELECOMUNICAÇÕES E INFORMÁTICA

42078 - Informação e Codificação

Eduardo Fernandes 98512
João Ferreira 103625
Pedro Monteiro 97484

Lab Work nº1

2023

Contents

List of Code Blocks

1 Exercise 1

1.1 Exercise Description

This exercise asks to change the *WAVHist* class in order to achieve some goals. The *WAVHist* class is responsible for outputting the histogram of a WAV audio file. We changed the class in order to:

- Produce the histogram of the average of the channels (known as the mono version of a audio file) and the difference of the channels (know as the SIDE channel);
- Change the bins of the histogram so that it gather together 2^k instead of a bin for each different sample value.

1.2 Code

1.2.1 wav_hist.h

To produce the histogram of the two different versions of the audio we added the following structures in *wav_hist.h*. Each one responsible for storing the counting of each version.

Code Block 1: Wav Hist new structures

```
std::map<short, size_t> avg_counts; // average of channels
std::map<short, size_t> diff_counts; // difference of channels
```

To update the counting of each structure we changed the *WAVHist* class and introduced two new functions.

Code Block 2: Wav Hist new update functions

```
void update_avg(const std::vector<short>& samples, int num_channels)
   {
    if (num_channels == 2){
      for (size_t i = 0; i < samples.size(); i += 2) {
        short left = samples[i];
        short right = samples[i + 1];
        short avg = adjustForCoarseness((left + right) / 2);
        avg_counts[avg]++;
      }
    }
}
void update_diff(const std::vector<short>& samples, int num_channels)
    {
    if (num_channels == 2){
      for (size_t i = 0; i < samples.size(); i += 2) {
        short left = samples[i];
        short right = samples[i + 1];
        short diff = adjustForCoarseness((left - right) / 2);
        diff_counts[diff]++;
      }
    }
}
```

Both of the mid and side versions of the audio are calculated by adding or subtracting, respectively, the left with the right channel and dividing by the number of the channels. We then store this new value as a key of the structure mentioned before. This new value is calculated taking a coarseness factor into the equation. A new function called *adjustForCoarseness* is used to approach this taks. For a coarseness factor of 2k, the sample value can be divided by 2k and then multiplied by 2k, effectively rounding it down to the nearest multiple of 2k.

Code Block 3: Function adjustForCoarseness

```
short adjustForCoarseness(short sample) const {
    return (sample / coarseness_factor) * coarseness_factor;
}
```

To dump the results the provided *dump* function in the *WAVHist* class was changed as it is shoiwn below.

Code Block 4: Wav Hist dump function changes

```
void dump(const size_t channel, const std::string version) const {
    if (version == "avg"){
      std::cout << "Average (MID) channel:\n";
      for(auto [value, counter] : avg_counts)
      std::cout << value << '\t' << counter << '\n';
    }
    else if (version == "diff"){
      std::cout << "Difference (SIDE) channel:\n";
      for(auto [value, counter] : diff_counts)
        std::cout << value << '\t' << counter << '\n';
    }
    else{
      std::cout << "Normal (Stereo) channel:\n";
      for(auto [value, counter] : counts[channel])
      std::cout << value << '\t' << counter << '\n';
    }
}
```

1.2.2 wav_hist.cpp

Also, the *wav_hist.cpp* while loop was modified to update the structure for the pretended version. At the end the *dump* function is called to return the intended results.

Code Block 5: While loop changes

```
while((nFrames = sndFile.readf(samples.data(), FRAMES_BUFFER_SIZE))){
    samples.resize(nFrames * sndFile.channels());
    if (version == "avg")
        hist.update_avg(samples, sndFile.channels());
    else if (version == "diff")
        hist.update_diff(samples, sndFile.channels());
    else
        hist.update(samples);
}

hist.dump(channel, version);
```

1.3 Usage

It should be provided to the program the input file, the channel or the version of the audio we want to create the histogram from and the coarseness factor that later will be initialized in the *WAVHist* class as an exponent of base 2.

```
$ ./wav_hist <input file> <channel | version> <coarseness>
```

1.4 Results

Below are some of the results of the program. We made this plots with *gnuplot* by targeting the values from the program to a file so that we use this file to plot the histograms.

Figure 1: Histogram for left channel of the audio.

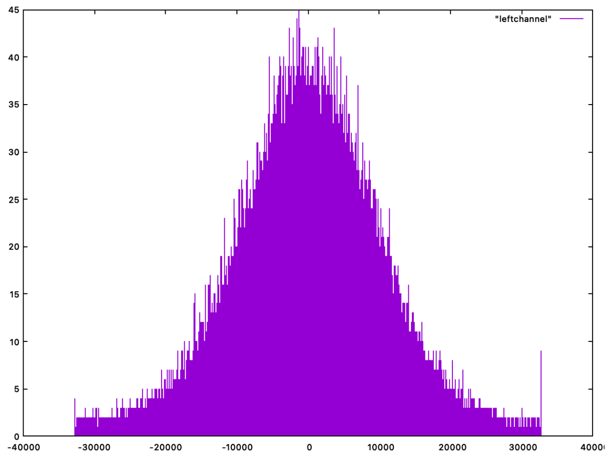


Figure 2: Histogram for right channel of the audio.

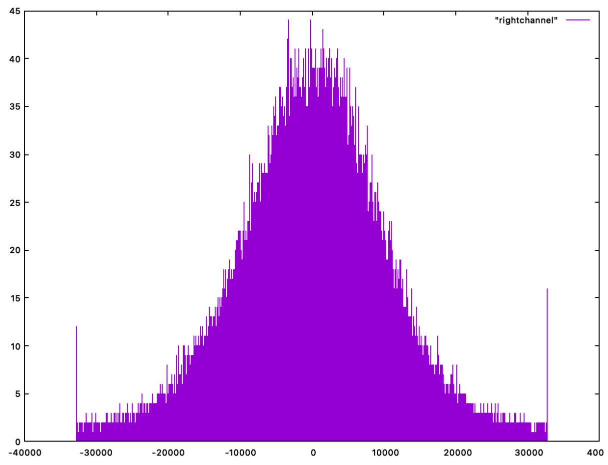


Figure 3: Histogram for mid channel of the audio.

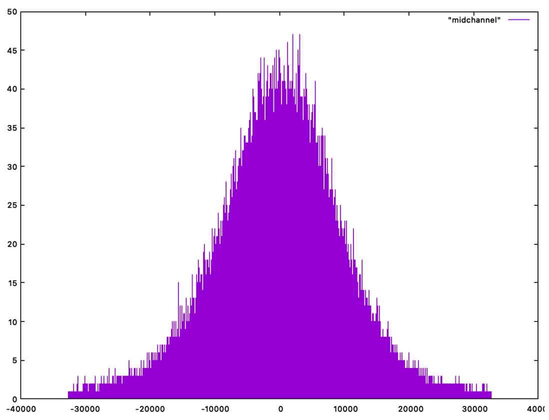


Figure 4: Histogram for side channel of the audio.

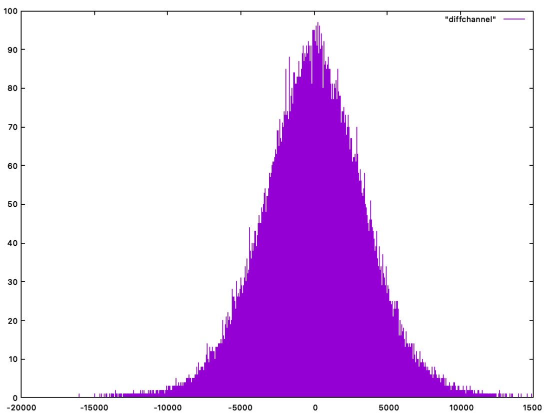


Figure 5: Histogram for mid channel of the audio with a coarseness factor of 2^7.

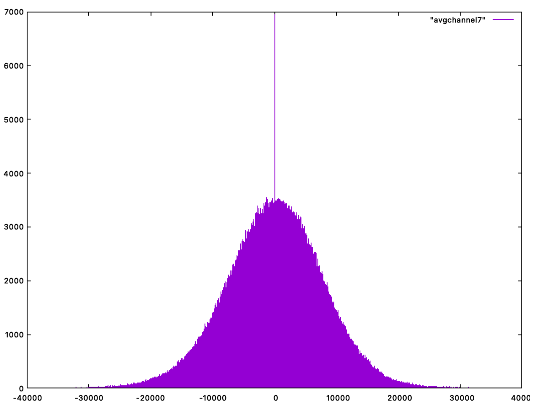
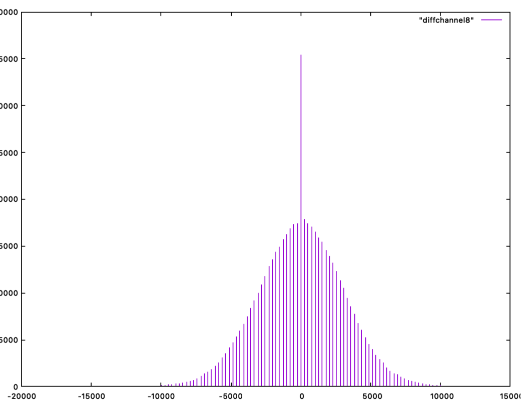


Figure 6: Histogram for side channel of the audio with a coarseness factor of 2^8.



Looking at the figure we can tell that the results are correct. The fact that the mid channel is the average of the two channels we expect that his amplitude would be similar to this channels. The side channel is a bit different. As the side channel is the difference of the two channels we expect that his amplitude would go around 0 because the left and right channel are in a similar amplitude domain. When we used the histogram with the coarseness factor, it made it easier to see patterns in the values. We grouped similar values together, and our results matched what we expected at the beginning.

2 Exercise 2

2.1 Exercise Description

This exercise asks to implement a new program name *wav_cmp.cpp* that given two audio files prints:

- The average mean squared error;
- The maximum per sample absolute error;
- The signal-to-noise ratio (SNR).

2.2 Code

To achieve the goal of this exercise we needed the formulas to calculate this three metrics. Mean Squared Error is a metric used to measure the average squared difference between the values of two audio signals, typically the original and the processed versions. The equation calculates the MSE by summing the squared differences between each sample of the original audio signal (x_i) and its corresponding sample in the processed audio signal ($\hat{x}_i$), and then dividing this sum by the total number of samples (N).

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(x_i - \hat{x}_i)^2 \tag{1}$$

The Maximum Per Sample Absolute Error measures the maximum absolute difference between corresponding samples of two audio signals. The equation calculates the maximum absolute error by finding the maximum absolute difference between each sample of the original audio signal (x_i) and its corresponding sample in the processed audio signal ($\hat{x}_i$).

$$Max\ Absolute\ Error = \max_{i=1}^{N} |x_i - \hat{x}_i| \tag{2}$$

Signal-to-Noise Ratio is a metric that quantifies the ratio of the power of the original audio signal to the power of the noise introduced during processing. The equation calculates SNR by taking the ratio of the power of the original audio signal (sum of squared samples) to the power of the difference between the original and processed audio signals (sum of squared errors). The result is then scaled by 10 and the logarithm base 10 is taken to obtain SNR in dB.

$$SNR = 10 \cdot \log_{10}\left(\frac{\sum_{i=1}^{N} x_i^2}{\sum_{i=1}^{N}(x_i - \hat{x}_i)^2}\right) \tag{3}$$

Where:

$$\begin{aligned} N &: \text{Total number of samples} \\ x_i &: \text{Original audio sample at index } i \\ \hat{x}_i &: \text{Processed audio sample at index } i \end{aligned}$$

2.2.1 wav_cmp.cpp

A WAV file is not just a series of raw audio samples. It begins with a header that contains meta-information about the audio data that follows.
The *WAVHeader* structure provides necessary details like sample rate, bit depth, number of channels, etc. This header was designed to be used in the *readWAV* function that its purpose is to read the samples from a given WAV file. By reading the WAV header, the function ensures that it's dealing with a valid WAV file and understands the format and meta-information of the audio. The actual audio comparison (between original and processed audio) then works purely with the sample data and does not need to concern itself with the details of the WAV file format.

Code Block 6: WAVHeader and readWav function

```
struct WAVHeader {
    char riff[4];
    uint32_t chunkSize;
    char wave[4];
    char fmt[4];
    uint32_t subChunkSize;
    uint16_t audioFormat;
    uint16_t numChannels;
    uint32_t sampleRate;
    uint32_t byteRate;
    uint16_t blockAlign;
    uint16_t bitsPerSample;
};

vector<int16_t> readWAV(const string &filename) {
    WAVHeader header;
    ifstream file(filename, ios::binary);

    if (!file.read((char*)&header, sizeof(WAVHeader))) {
        throw runtime_error("Error reading WAV header");
    }

    vector<int16_t> samples;
    int16_t sample;
    while (file.read((char*)&sample, sizeof(int16_t))) {
        samples.push_back(sample);
    }

    return samples;
}
```

In the main of the program there is a for loop that will cycle through each sample in the original and processed vectors. After the loop completes, these metrics have been calculated based on all the samples in the audio files. The subsequent calculations outside the loop normalize the MSE and compute the SNR using the signal and noise power. In the end, the results are printed to the console.

Code Block 7: metrics calculation

```
for (size_t i = 0; i < original.size(); i++) {
    double error = original[i] - processed[i];
    mse += error * error;
    maxError = max(maxError, abs(error));
    signalPower += original[i] * original[i];
    noisePower += error * error;
}

mse /= original.size();
snr = 10 * log10(signalPower / noisePower);
```

2.3 Usage

It should be provided to the program the two audio files to be processed.

```
$ ./wav_cmp <original_file> <processed_file>
```

2.4 Results

To test this program we used the original *sample.wav* file and two versions being the quantified versions of this one with 6-bit and 2-bit. This were the results we got:

Results	6-bit	2-bit
Average Mean Squared Error:	349223	8.945e+07
Maximum Per Sample Absolute Error:	1023	16383
Signal-to-Noise Ratio (dB):	23.2638	-0.820976

As the bits of the quantified file decrease from 6 to 2 bits, it is possible to notice significant changes in the results we obtain. We can see that the Mean Squared Error and Maximum Per Sample Absolute Error increase. On the other hand the Signal-to-Noise Ratio decreases.
A higher MSE indicates more distortion and a more dissimilar match between the signals. The results we obtained support this fact because with fewer bits, the quantified version deviates more from the original, leading to higher error. The same happens with maximum per sample absolute error that represents a higher value if more samples have a large deviation from the original ones.
The SNR is the only metric that decreases, meaning that the noise is getting more stronger than the signal. As the quantified versions of the original sample present more noise, we conclude that we obtained the expected results.

3 Exercise 3

3.1 Exercise Description

The primary goal of this exercise is to develop class and program that allow to compress the size of a WAV file by reducing its bit depth via uniform scalar quantization, thereby achieving data economy without significantly compromising on audio quality.

3.2 Code

The *wav_quant.h* header file offers a structured approach to representing and quantizing WAV audio files. At its core, the code aims to bridge the gap between raw WAV file data and the potential for data compression through bit depth modification, all without causing a drop in audio fidelity.

3.2.1 wav_quant.h

The WavQuant class provides the core functionality of quantize WAV audio files. It manages the reading of the input WAV file, using the libsndfile library, the quantization process, and writing the quantized audio to an output file, also with the libsndfile library.

The class Constructor initializes the class with the paths of the input and output WAV files and the desired bit depth for quantization and the quantize() function drives the entire quantization process – from reading the audio samples to writing the quantized data back

Code Block 8: WavQuant Class

```
#ifndef WAV_QUANT_H
#define WAV_QUANT_H

#include <sndfile.hh>
#include <vector>

class WavQuant {
public:
    // constructor
    WavQuant(const std::string &inputPath, const std::string &
       outputPath, int newBitDepth);
    void quantize();

private:
    std::string inputPath; // store the input WAV file path
    std::string outputPath; // store the output WAV file path
    int newBitDepth; // store the desired bit depth for quantization

    void quantizeSample(short &sample); // function to quantize an
       individual audio sample
};

#endif
```

The private section of the *WavQuant* class encapsulates the internal mechanisms essential for the quantization process, safeguarding them from any unintended external interferences or modifications.

Among the private attributes, it has *inputPath*, *outputPath*, and *newBitDepth*. These store crucial details such as the source of the audio file, where the quantized output should be saved, and the desired bit depth for quantization, respectively.

These attributes are important to the class's operation, and it's vital that they remain immutable after the object's instantiation.

The *quantizeSample(short sample)* function is a private method that carries out the actual bit modification on a single audio sample. The reason this function is kept private is its specialized nature. Externally, another class should be concerned with quantizing an entire audio file rather than individual samples.

By making *quantizeSample* private, the *WavQuant* class hides this granular detail and ensures that users don't inadvertently misuse it.

On the other hand, the quantize() function is public since it provides the complete functionality of reading an audio file, processing all its samples, and then writing the quantized output.

3.2.2 wav_quant.cpp

The main function begins by ensuring the correct number of command-line arguments are provided, guiding users on the proper usage if the check fails, and also it then checks if the desired bit depth is within acceptable limits (between 1 and 16).

After the validation process, a WavQuant object, quantizer, is created using the input and output paths and the desired bit depth, and then the quantize() method is invoked.

So, the input WAV file is opened for reading, a new output WAV file is prepared, retaining the format, channels, and sample rate of the input. All audio samples from the input file are read into a vector and each audio sample in the vector is passed through the quantizeSample method, effectively reducing its bit-depth, writting the quantized audio samples to the output file.

Code Block 9: Scalar Quantization Main Program

```cpp
int main(int argc, char *argv[]) {
    if (argc != 4) { // check number of command-line arguments
        std::cerr << "Usage: " << argv[0] << " <input.wav> <output.
           wav> <bitsToKeep>" << std::endl;
        return 1;
    }

    // parse command-line arguments
    std::string inputPath = argv[1];
    std::string outputPath = argv[2];
    int bitsToKeep = std::stoi(argv[3]);

    if (bitsToKeep <= 0 || bitsToKeep > 16) { // check bit range
        std::cerr << "Please provide a valid number of bits to keep
           between 1 and 16." << std::endl;
        return 1;
    }

    WavQuant quantizer(inputPath, outputPath, bitsToKeep); // create
       a WavQuant object
    quantizer.quantize();

    std::cout << "Quantization completed." << std::endl;
    return 0;
}
```

In the main program is also the implementation of the header methods.

The constructor, WavQuant::WavQuant, is responsible for initializing a WavQuant object with three arguments: the input WAV file path (inputPath), the path where the quantized output WAV file should be saved (outputPath), and the desired bit depth for quantization (newBitDepth).

The method WavQuant::quantize is the core part of this quantization operation. It begins by attempting to open the input WAV file using the SndfileHandle library, and if there's an error in this process, it exits with an error message. If reading is successful, it sets up another SndfileHandle object for writing the output WAV file, using the format, channels, and sample rate from the input file. The audio samples from the input file are then read into a vector. Each of these samples undergoes a quantization operation, achieved by invoking the quantizeSample method. Once all samples in the vector are quantized, they are written to the output file, completing the quantization process.

The WavQuant::quantizeSample function is a utility method used to quantize individual audio samples. The number of bits to be discarded is computed based on the difference between the original bit depth and the desired bit depth (newBitDepth). Bitwise operations then effectively "truncate" the least significant bits of the sample, producing a quantized version of it. This quantization process effectively reduces the dynamic range of the audio sample, which can lead to a reduction in file size but at the potential cost of audio quality.

Code Block 10: WavQuant Header Methods Implementation

```
// Constructor: initializes the WavQuant object with inputPath,
   outputPath and desired bit depth
WavQuant::WavQuant(const std::string &inputPath, const std::string &
   outputPath, int newBitDepth)
    : inputPath(inputPath), outputPath(outputPath), newBitDepth(
       newBitDepth) {}

void WavQuant::quantize() { // main quantization function
    SndfileHandle inFile(inputPath); // open input WAV file for
       reading
    if (inFile.error()) {
        std::cerr << "Error reading input file." << std::endl;
        return;
    }

    // create output WAV file for writing, copying the format,
       channels, and sample rate from the input file
    SndfileHandle outFile(outputPath, SFM_WRITE, inFile.format(),
       inFile.channels(), inFile.samplerate());
    if (outFile.error()) {
        std::cerr << "Error creating output file." << std::endl;
        return;
    }

    std::vector<short> samples(inFile.frames() * inFile.channels());
       // read audio samples from the input file into a vector
    inFile.read(samples.data(), samples.size());

    for (short &sample : samples) { // quantize each sample in the
       vector
        quantizeSample(sample);
    }

    outFile.write(samples.data(), samples.size()); // write the
       quantized samples
}

void WavQuant::quantizeSample(short &sample) { // quantize an
   individual sample
    int numBitsToCut = 16 - newBitDepth; // get how many bits to
       discard based on desired bit depth

    // bitwise operations to quantize the sample
    sample = sample >> numBitsToCut;
    sample = sample << numBitsToCut;
}
```

3.3 Usage

In order to run the program, it is necessary to provide the input audio sample, the name of the output sample and also the desired number of bits for the final sample:

```
$ ./wav_quant <input.wav> <output.wav> <newBits>
```

3.4 Results

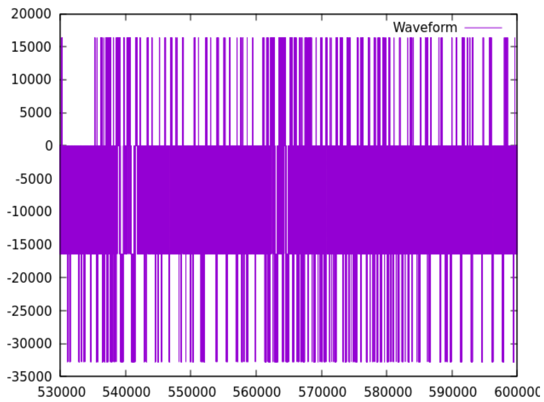
Below are presented the results, obtained when performing uniform scalar quantization with different bit values (2,4,6 and 8), using gnuplot, for the same audio sample, provided in the statement.

Scalar uniformization refers to the process of quantizing audio samples to fit within a defined range or scale. In this context, the audio samples are quantized to bit depths of 2, 4, 6, and 8 bits.

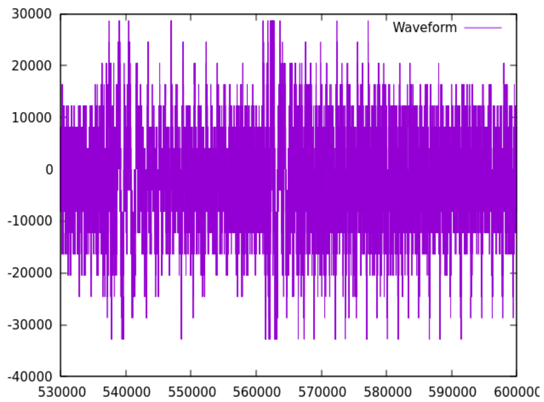
Regarding Figure 7a, its observed a highly simplified waveform with only four unique amplitude levels. This simplification is indicative of 2-bit quantization, which inherently represents just $2^2 = 4$ levels. While the general waveform shape is maintained, many details have been sacrificed due to the significant decrease in amplitude resolution.

In contrast, Figure 7b showcases a waveform with $2^4 = 16$ unique amplitude levels. There's a marked enhancement in its representation when juxtaposed with the 2-bit version. The waveform offers more intricacies, though some subtleties from the original audio may remain absent.

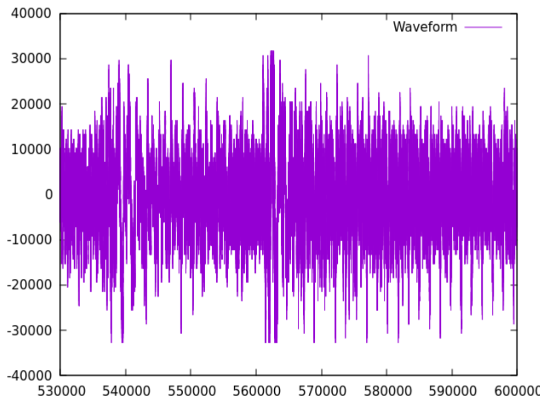
Turning to the 6-bit representation in Figure 7c, the waveform possesses $2^6 = 64$ unique amplitude levels. This rendition is notably more refined than its 4-bit counterpart. A greater



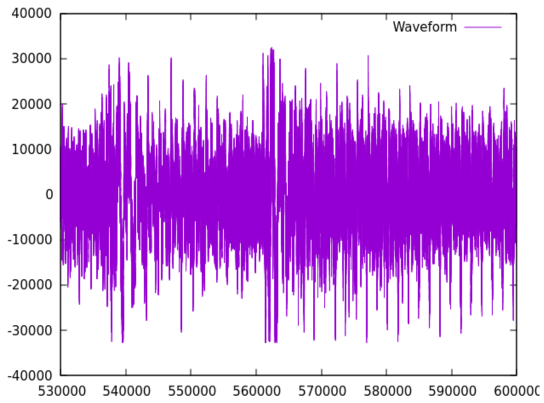
(a) Gnuplot Audio with 2 Bits



(b) Gnuplot Audio with 4 Bits



(c) Gnuplot Audio with 6 Bits



(d) Gnuplot Audio with 8 Bits

Figure 7: Gnuplot Audio Samples After Uniform Scalar Quantization

number of the audio sample's details are discernible, implying a representation more congruent with the original audio.

Lastly, the waveform in the final representation boasts $2^8 = 256$ unique amplitude levels. The 8-bit quantized waveform stands out as the most detailed among all versions presented. It bears a strong resemblance to the original audio, especially in the aspects of amplitude variations and overall waveform shape.

Quantizing audio samples to lower bit depths leads to a significant reduction in the audio's quality and fidelity. While this can result in smaller data sizes, it may not be suitable for all applications. Note the fact that in the sample with the lowest number of bits, 2 bits, the audio is almost imperceptible, with a lot of noise.

4 Exercise 4

4.1 Exercise Description

The primary objective of this exercise is to create a class and program called ”wav effects” to implement various audio effects such as reverse, echo, amplitude modulation, etc.

4.2 Code

4.2.1 Reverse

This effect is intended to produce the original audio file in reverse.

Code Block 11: Effects Function

```

if (effect == "reverse")
{
    while ((nFrames = sfhIn.readf(samples.data(), FRAMES_BUFFER_SIZE)
       ))
    {
        i++;

        // short aux = 0;
        for (size_t i = 0; i < samples.size(); i++)
        {
        samples_out.push_back(samples[samples.size() - 1 - i]);
        }
    }
    sfhOut.writef(samples_out.data(), FRAMES_BUFFER_SIZE * i);
}
```

4.2.2 Mute Left and Right

This effect produces sound from only one side, either the left or the right side. To achieve this, it is necessary to set the sample value to 0 based on the channel. A counter is initialized and throughout the program it incrementally between even and odd numbers. This allows us to toggle between the value ’0’ and the samples, providing the option to switch between them.

- Left

$$samples[i] = (++\text{value}\%2) \cdot samples[i]$$

- Right

$$samples[i] = (value++\%2) \cdot samples[i]$$

Code Block 12: Effects Function

```

else if (effect == "left")
{
    while ((nFrames = sfhIn.readf(samples.data(), FRAMES_BUFFER_SIZE)
       ))
    {
        for (size_t i = 0; i < samples.size(); i++)
        {
            echo_out[i] = (++value % 2) * samples[i];
        }

        sfhOut.writef(echo_out.data(), nFrames);
    }
}
else if (effect == "right")
{
    while ((nFrames = sfhIn.readf(samples.data(), FRAMES_BUFFER_SIZE)
       ))
    {
        for (size_t i = 0; i < samples.size(); i++)
        {
            echo_out[i] = (value++ % 2) * samples[i];
        }
        sfhOut.writef(echo_out.data(), nFrames);
    }
}
```

4.2.3 Eco

An echo is characterized by the delayed arrival of reflected sound at our ears after a certain amount of time. This allows us to distinctly perceive the original sound and the reflected sound. In our program, we implemented multiple echoes, including single, double, and triple echoes, thus creating a variety of echo effects.

- Single Echo

$$samples[i] = (samples[i] + samples[i - \text{delay}]) \cdot \alpha$$

- Double Echo

$$samples[i] = (samples[i] + samples[i - \text{delay}] + samples[i - \text{delay} \cdot 2]) \cdot \alpha$$

- Triple Echo

$$samples[i] = (samples[i]+samples[i-\text{delay}]+samples[i-\text{delay}\cdot 2]+samples[i-\text{delay}\cdot 3])\cdot\alpha$$

Code Block 13: Effects Function

```
else if (effect == "single_echo")
{
    while ((nFrames = sfhIn.readf(samples.data(), FRAMES_BUFFER_SIZE)
       ))
    {
           for (size_t i = 0; i < samples.size(); i++)
        {
            echo_out[i] = (samples[i] + samples[i - 44000]) * 0.5;
        }
        sfhOut.writef(echo_out.data(), nFrames);
    }
}
else if (effect == "double_echo")
{
    while ((nFrames = sfhIn.readf(samples.data(), FRAMES_BUFFER_SIZE)
       ))
    {
        for (size_t i = 0; i < samples.size(); i++)
        {
            echo_out[i] = (samples[i] + samples[i - 44100] + samples[
               i - 88200]) * 0.4;
        }
        sfhOut.writef(echo_out.data(), nFrames);
    }
}
else if (effect == "triple_echo")
{
    while ((nFrames = sfhIn.readf(samples.data(), FRAMES_BUFFER_SIZE)
       ))
    {
        for (size_t i = 0; i < samples.size(); i++)
        {
            echo_out[i] = (samples[i] + samples[i - 44100] + samples[
               i - 88200] + samples[i - 132300]) * 0.2;
        }
        sfhOut.writef(echo_out.data(), nFrames);
    }
}
```

4.2.4 Amplitude Loop

This effect creates audio with an amplitude pattern that resembles a sinusoidal function.

$$samples[i] = samples[i] \cdot \alpha, \quad \alpha \in [0, 1]$$

Code Block 14: Effects Function

```
else if (effect == "amplitude_loop")
```

```
{
    while ((nFrames = sfhIn.readf(samples.data(), FRAMES_BUFFER_SIZE)
       ))
    {
        i++;
        for (size_t i = 0; i <= samples.size(); i++)
        {
            samples_out.push_back(samples[i] * (double(value) /
               double(samples.size())));
            if (flag == 1)
            {
                value++;
            }
            else
            {
                value--;
            }
        }
        flag = !flag;
    }
    sfhOut.writef(samples_out.data(), FRAMES_BUFFER_SIZE * i);
}
```

4.3 Usage

```
$ ./wav_effects ../sndfile-example-src/sample.wav out.wav <reverse>
```

4.4 Results

The results of this exercise will not be shown here. However, if you want to check the effects, they are available on the folder named ”results”. Each file’s name is the corresponding effect.

5 Exercise 5

5.1 Exercise Description

This exercise involves designing a class named BitStream that enables efficient reading and writing of individual bits or sequences of bits to and from a file.

5.2 Code

5.2.1 bit_stream.h

The private section of the BitStream class encapsulates the crucial components and operations needed for the class's functionality. By designating these members and methods as private, the class ensures that they cannot be directly accessed or modified from outside the class, preserving data integrity.

The *buffer* temporarily holds bits as they are being read from or written to the *file*. The *currBitPos* keeps track of the current position within the *buffer*, indicating where the next bit should be placed or read from. *flushBuffer()* checks if there are any bits in the buffer and, if so, writes the current *buffer* to the file, ensuring that the buffer is written to the file whenever it's full or when deemed necessary. After writing the *buffer* to the file, it resets the *buffer* and *currBitPos* to their initial states, preparing them for the next set of bit operations.

Code Block 15: Bit Stream Class Private Methods

```cpp
class BitStream {
private:
    std::fstream file;      // file stream for reading/writing
    unsigned char buffer;  // current byte buffer, holds the current byte
    int currBitPos;            // current position in the buffer [0, 7]

    void flushBuffer() { // write current buffer to the file if there are any bits in it
        if (currBitPos == 0) return;
        file.write(reinterpret_cast<char*>(&buffer), 1);

        // reset buffer and currBitPos
        buffer = 0;
        currBitPos = 0;
    }
```

The public section of the BitStream class provides the primary interface for users to interact with the class.

The class Constructor initializes the BitStream object by opening the specified file in binary mode, given the name of the file to be opened and the mode in which the file should be opened (read or write).

Code Block 16: BitStream Constructor

```cpp
public:
    // constructor
    BitStream(const std::string& filename, std::ios::openmode mode)
```

```
        : buffer(0), currBitPos(0) {
        file.open(filename, mode | std::ios::binary); // open file in
            binary mode
        if (!file.is_open()) {
            throw std::runtime_error("Failed to open file"); // throw
                exception if fails
        }
    }
```

The *writeBit* and *readBit* allow users to write and read a single bit to and from the file, respectively. The *writeBit* method adds a bit to the internal buffer, which accumulates until it completes a byte, at which point it writes to the file. Conversely, *readBit* extracts a single bit from the buffer. If the buffer is empty at the call time, it reads the next byte from the file, ensuring a continuous flow of data.

Code Block 17: ReadBit and WriteBit Functions

```

    void writeBit(bool bit) { // write a single bit to the buffer
        if (bit) {
            buffer |= (1 << (7 - currBitPos)); // create a byte where
                only one bit is set, in the currBitPos
            // if currBitPos=3, 7-3 = 4, 1 << 4 => 0001000
        }
        currBitPos++; // go to next position
        if (currBitPos == 8) { // if buffer is full, flush the buffer
            to file
            flushBuffer();
        }
    }

    bool readBit() { // read a single bit from the buffer
        if (currBitPos == 0) { // if buffer is empty, read a new byte
            from the file.
            if (!file.read(reinterpret_cast<char*>(&buffer), 1)) {
                throw std::runtime_error("Failed to read bit");
            }
        }
        bool bit = (buffer & (1 << (7 - currBitPos))) != 0; //
           extract bit at the current position from  buffer.
                                // (1 << (7 - currBitPos)) ->
                                   generates a byte where only one
                                   bit is set
                                // corresponding to the current
                                   position, others are 0
        currBitPos = (currBitPos + 1) % 8; // increment position
           pointer
        return bit;
    }
```

For handling multiple bits, the class offers *writeBits* and *readBits* methods. These methods are designed to write or read a sequence of bits, specified by the user, to or from the file.

The *writeBits* method is designed to write a sequence of n bits from a given value to the file. The method iterates through the specified number of bits in the value argument, from the most significant to the least significant. For each bit, it uses the *writeBit* method to write the bit to the internal buffer.

readBits retrieves a sequence of n bits from the file, returning them as a 64-bit integer. This method initializes a 64-bit integer, value, to accumulate the bits read from the file. For each iteration, it uses the *readBit* method to fetch a single bit, then shifts this bit into the appropriate position in value.

Code Block 18: ReadBits and WriteBits Functions

```
void writeBits(uint64_t value, int n) { // Writes n bits to the file.
    if (n < 0 || n > 64) { // check if n is in the range
        throw std::invalid_argument("Number of bits out of range");
    }
    for (int i = n - 1; i >= 0; --i) {
        writeBit((value >> i) & 1); // (value >> i) -> shift the value to the right by 1 position
                                    // & 1 -> check if the least significant bit is 1 or 0
    }
}

uint64_t readBits(int n) { // read n bits from file
    if (n < 0 || n > 64) {
        throw std::invalid_argument("Number of bits out of range");
    }
    uint64_t value = 0; // var to store bits from file
    for (int i = 0; i < n; ++i) { // process 1 bit from the file
        value = (value << 1) | readBit(); // (value << 1) -> shift the current value to the left by 1position
    }                               // | readBit() -> set least significant bit to the bit read from the file
    return value;
}
```

Additionally, the class caters to string data through its *writeString* and *readString* methods. These methods interpret strings as sequences of bits, allowing them to be written to or read from the file.

The *writeString* method iterates over each character in the provided string. For every character, it casts the character to a 64-bit integer and then uses the *writeBits* method to write the character's bit representation to the file.

The *readString* method initializes an empty string, result, to accumulate the characters read from the file. For each character to be read, it uses the *readBits* method to fetch 8 bits (representing a character) and then casts these bits back to a character type. This character is then appended to the result string. The process continues until the entire string of the specified length is reconstructed.

Code Block 19: ReadString and WriteString Functions

```
    void writeString(const std::string& str) { // write each
       character of str as 8 bits to file
        for (char c : str) {
            writeBits(static_cast<uint64_t>(c), 8);
        }
    }

    std::string readString(int len) { // read len characters (8 bits
       each) from file and return as a string.
        std::string result;
        for (int i = 0; i < len; ++i) {
            result.push_back(static_cast<char>(readBits(8))); //
               append to result casted 8 bits readed from file
        }
        return result;
    }
```

5.2.2 bit_stream.cpp

In order to test the previously described *BitStream* class, a simple demonstration is provided in the form of a *main* function.

The program starts by testing the *writeBit* and *readBit* functions, writing bits one by one to a file, *testfile.bin*. After writing, the *readBit* function is used to read the bits one by one from the created file. Finally, the close function is called to clear any possible bits that are still in the buffer.

Code Block 20: Main Program ReadBit and WriteBit Test

```
    BitStream writer("testfile.bin", std::ios::out);
    writer.writeBit(1);
    writer.writeBit(0);
    writer.writeBit(1);
    writer.close();

    BitStream reader("testfile.bin", std::ios::in);
    std::cout << reader.readBit() << std::endl;
    std::cout << reader.readBit() << std::endl;
    std::cout << reader.readBit() << std::endl;
    reader.close();
```

Next, two new instances of the *BitStream* class are created, writer2 and reader2, in order to test the *writeBits* and *readBits* functions. The value 5 is then written in the *testfile2* file, with 3 bits, which corresponds to 101, and this value is then read by the *readBits* function, producing the value 5 in the terminal, as expected.

Subsequently, two new instances of the *BitStream* class are created: writer2 for writing and reader2 for reading, to demonstrate the functionality of the *writeBits* and *readBits* methods. The number 5 is encoded into *testfile2* using just three bits, represented as 101 in binary. When these bits are read back using the *readBits* method, the output displayed on the terminal is the expected value of 5.

Code Block 21: Main Program ReadBits and WriteBits Test

```
    BitStream writer2("testfile2.bin", std::ios::out);
    writer2.writeBits(5, 3);  // write multiple bits to a file -> 5
       in binary is 101
    writer2.close();

    BitStream reader2("testfile2.bin", std::ios::in);
    std::cout << reader2.readBits(3) << std::endl;  // should print 5
    reader2.close();
```

Finally, the *writeString* and *readString* methods are put to the test. Two new *BitStream* objects, writer3 and reader3, are instantiated for this purpose. The string ”Hello” is written to *testfile3.bin* for demonstration. Subsequently, this string is retrieved using the *readString* method and its value is displayed on the terminal

Code Block 22: Main Program ReadString and WriteString Test

```
    BitStream writer3("testfile3.bin", std::ios::out);
    writer3.writeString("Hello"); // write a string to a file
    writer3.close();

    BitStream reader3("testfile3.bin", std::ios::in);
    std::cout << reader3.readString(5) << std::endl;  // should print
        "Hello"
    reader3.close();
```

The test files are available on github repository[1].

[1]⟨https://github.com/pedromonteiro01/ic/tree/main/trab1/ex5_test_files⟩

6 Exercise 6

6.1 Exercise Description

This exercise involves the implementation of a program that consists of both an encoder and a decoder. The main purpose of these programs is to utilize a custom *BitStream* class, which appears to have been developed separately, to perform encoding and decoding operations on text files containing only binary data represented as 0s and 1s.

6.2 Code

6.2.1 encoder.cpp

The encoder's main function is to take as input a text file containing sequences of 0s and 1s. It converts these binary sequences into a binary equivalent where each byte in the output binary file represents eight bits from the input text file. Essentially, it takes the textual representation of binary data and packs it into a binary file format where each byte is a group of eight bits.
First, we establish two essential file streams: *outStream* for writing to the output file and *inputFile* for reading from the input file. To know the length of the binary data to be further decoded we need the number of bits in the input text file first. This is accomplished using a while loop that iterates through the characters in the *inputFile*. If the character is indeed '0' or '1', it increments the totalBits counter. Once the total number of bits in the input data has been calculated, the code proceeds to write this information as a 32-bit integer to the output file using *writeBits* method.

Code Block 23: total bits in input file

```
BitStream outStream(outputFilename, ios::out);
ifstream inputFile(inputFilename);

// Calculate total number of bits first.
while (inputFile.get(c)) {
    if (c == '0' || c == '1') {
        totalBits++;
    }
}

// Write the total number of bits to the output file as a 32-bit
   integer.
outStream.writeBits(static_cast<uint64_t>(totalBits), 32);

// Reset input file and start encoding.
inputFile.clear();
inputFile.seekg(0, ios::beg);
```

In the next step follows the primary operation. Begins with a while loop that iterates through the characters in the *inputFile*. For each character read from the input file, it extracts its numeric value (0 or 1) and appends it to the value variable. At the same time, a count variable is incremented to keep track of the number of bits read. If the count has reached 8, it writes the 8-bit value to the output file using the *writeBits* method. Following this, both the count and value variables are reset to 0 to prepare for the next set of 8 bits. The loop continues to process characters from the input file until there are no more characters left. At the end, any remaining bits that are less than 8 are also written to the output file after the loop concludes.

Code Block 24: bits to output file

```
// Read each character from the input file
while (inputFile.get(c)) {
    // If the character is not a 0 or 1, ignore it
    if (c == '0' || c == '1') {
        value = (value << 1) | (c - '0');  // Shift and append bit
        count++;                           // Increment bit count
    }

    // If we have 8 bits, write them to the output file
    if (count == 8){
        outStream.writeBits(value, count);
        count = 0;
        value = 0;
    }
}

// Write any remaining bits to the output file
if (count > 0) {
    outStream.writeBits(value, count);
    outStream.close();
}
```

6.2.2 decoder.cpp

The decoder's main function is to perform the reverse operation of the encoder. It takes as input the binary file created by the encoder (where each byte represents eight bits). It converts this binary data back into its original textual form, which consists of sequences of 0s and 1s. The decoding process basically unpacks the binary data and produces the original textual representation of binary information.
As it is in the encoding process, the code begins by establishing two file streams: *inStream* for reading from the input file and *outputFile* for writing to the output file. The first operation in this process is to acknowledge the number of bits to read from the input file. For that, the code reads a 32-bit value from the input file, which is the total number of bits that need to be decoded. The next step is the actual decoding process. It starts with a *while* loop that continues as long as there are bits left to decode (*totalBitsToDecode¿*0). Then, calculates the number of bits to read in each batch, ensuring that it doesn't exceed 64 bits. After reading the bits from the input file it writes in output one. In the end of processing a batch of bits, the *totalBitsToDecode* is decremented by the number of bits read, ensuring accurate tracking of the remaining bits to decode.

Code Block 25: decoding process

```cpp
BitStream inStream(inputFilename, ios::in);
ofstream outputFile(outputFilename);

// Read the total number of bits from the input file
uint64_t totalBitsToDecode = inStream.readBits(32);

// Read each bit from the input file
while (totalBitsToDecode > 0) {
    // Determine the number of bits to read in this batch (up to 64)
    uint64_t bitsToRead = min(totalBitsToDecode, static_cast<uint64_t
       >(64));

    // Read totalBitsToDecode bits from the input file
    uint64_t byteValue = inStream.readBits(bitsToRead);

    // Write each bit to the output file
    for (int i = bitsToRead - 1; i >= 0; --i) {
        outputFile << ((byteValue >> i) & 1);  // Extract each bit
           and write to file
    }

    totalBitsToDecode -= bitsToRead;
}
```

6.3 Usage

For each program it should be provided to the program the two files.

```
$ ./encoder <input_file> <output_file>
$ ./decoder <input_file> <output_file>
```

In the encoder program the first file it has to be the text file containing the binary data and the second one the output binary file. The decoder program is the reverse of this one.

6.4 Results

Initially we tested the program for a input text file containing a small number of bits. For a more tough test we used a lorem ipsum text. First we converted the text to binary textual data with a online translator. After calculating the encoding and decoding with this data, we check again that the bits generated by the decoding program form the initial lorem ipsum text. The files used in these programs are stored in the results folder in the project root.

Lorem ipsum dolor sit amet consectetur adipisicing elit. Maxime mollitia, molestiae quas vel sint commodi repudiandae consequuntur voluptatum laborum numquam blanditiis harum quisquam eius sed odit fugiat iusto fuga praesentium optio, eaque rerum! Provident similique accusantium nemo autem. Veritatis obcaecati tenetur iure eius earum ut molestias architecto voluptate aliquam nihil, eveniet aliquid culpa officia aut! Impedit sit sunt quaerat, odit, tenetur error, harum nesciunt ipsum debitis quas aliquid. Reprehenderit, quia. Quo neque error repudiandae fuga? Ipsa laudantium molestias eos sapiente officiis modi at sunt excepturi expedita sint? Sed quibusdam recusandae alias error harum maxime adipisci amet laborum. Perspiciatis minima nesciunt dolorem! Officiis iure rerum voluptates a cumque velit quibusdam sed amet tempora. Sit laborum ab, eius fugit doloribus tenetur fugiat, temporibus enim commodi iusto libero magni deleniti quod quam consequuntur! Commodi minima excepturi repudiandae velit hic maxime doloremque. Quaerat provident commodi consectetur veniam similique ad earum omnis ipsum saepe, voluptas, hic voluptates pariatur est explicabo fugiat, dolorum eligendi quam cupiditate excepturi mollitia maiores labore suscipit quas? Nulla, placeat. Voluptatem quaerat non architecto ab laudantium modi minima sunt esse temporibus sint culpa, recusandae aliquam numquam totam ratione voluptas quod exercitationem fuga. Possimus quis earum veniam quasi aliquam eligendi, placeat qui corporis!

Figure 8: lorem_ipsum_encoded.txt

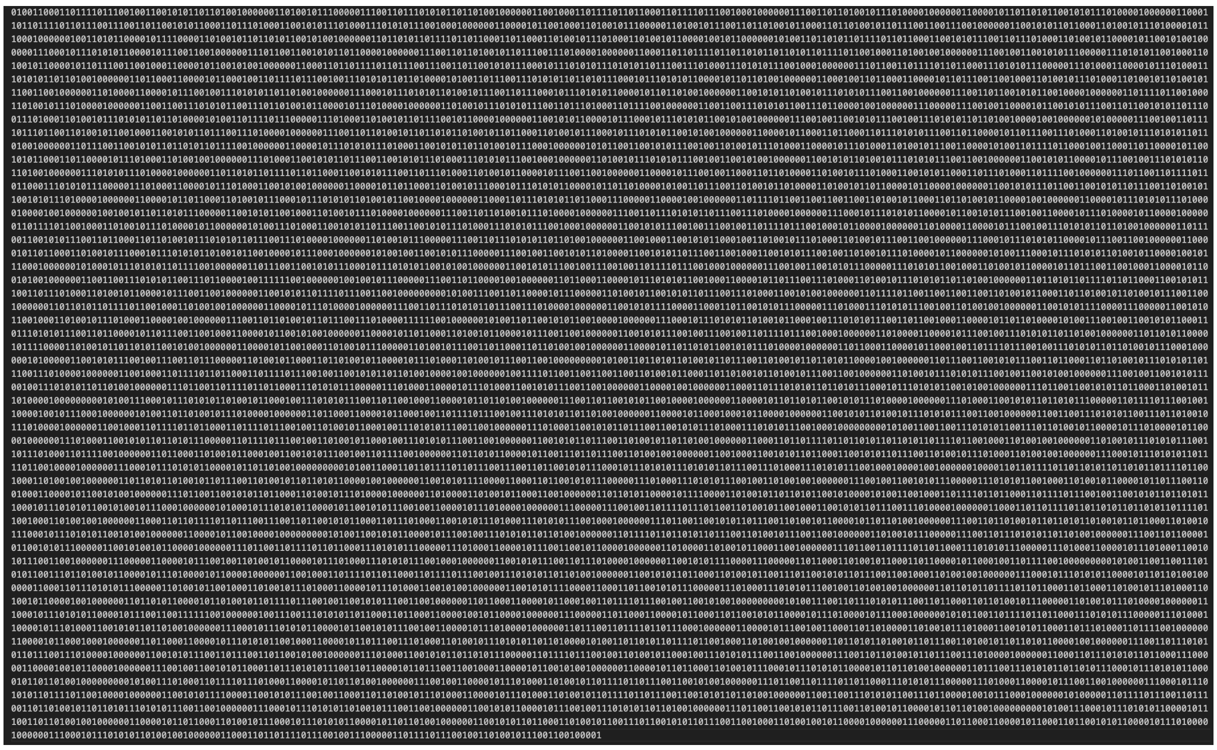


Figure 9: lorem_ipsum_decoded.txt

7 Exercise 7

7.1 Exercise Description

The main goal of this exercise is to implement a lossy codec for mono audio files with a single channel. It uses the Discrete Cosine Transform (DCT) to process audio on a block-by-block. Each audio block is transformed through DCT, followed by appropriate quantization of coefficients. The quantized data is then written to a binary file. The decoder, created to solely rely on the binary file generated by the encoder, attempts to recreate an approximate rendition of the audio. The objective is to achieve effective data compression while preserving audio quality.

7.2 Code

7.2.1 lossy_encoder.cpp

This code takes an input WAV file, divides it into blocks, and performs a Discrete Cosine Transform (DCT) on each block. It then quantizes the transformed coefficients, discards a specified number of coefficients, and writes the remaining coefficients to an output file.

The program starts defining a *quantization_factor*, which is used to control the level of quantization applied to the DCT coefficients. A higher factor reduces the accuracy of the representation.

Code Block 26: Quantization Factor

```cpp
const size_t QUANTIZATION_FACTOR = 10; // constant for quantization
                                       // higher values reduce
                                          representation accuracy
```

The program is expected to receive 4 arguments, namely the input audio file, the name of the output file, the block size, which specifies the number of samples per block, and number of DCT coefficients to discard, therefore it is necessary to check the number of arguments, returning error if it is not correct.

Code Block 27: Command Line Arguments Check Lossy Encoder

```cpp
    if (argc < 5) // check command line arguments
    {
        cerr << "Usage: " << argv[0] << " <input WAV file> <output
           binary file> <block size> <discarded units per block>\n";
        return 1;
    }
```

Then the input file is opened, checking that there are no errors, and also that it is really a mono audio file, that is, just one channel.

Code Block 28: Mono Audio Check Lossy Encoder

```cpp
    SndfileHandle sfhIn(inputFileName);
    if (sfhIn.error())
    {
        cerr << "Error: Could not open the input file or there was an
            issue with its format.\n";
        return 1;
    }

    if (sfhIn.channels() != 1) // // check if the WAV file is mono (
       only one channel)
    {
        cerr << "Error: The input file is not a mono file. It has "
           << sfhIn.channels() << " channels.\n";
        return 1;
    }
```

Two vectors *x* and *x_dct* are then created to store original samples and their DCT coefficients, respectively. An FFTW plan for DCT computation is created using the *fftw_plan_r2r_1d* function. This plan will later be executed for each block of samples.

Code Block 29: DCT Vectors

```
    // vectors to store original and DCT transformed data
    vector<double> x(block_size);
    vector<double> x_dct(block_size);

    // compute dct
    fftw_plan plan_d = fftw_plan_r2r_1d(block_size, x.data(), x_dct.
       data(), FFTW_REDFT10, FFTW_ESTIMATE);
```

After this, the program opens a binary file using the custom BitStream class to write the compressed data. Metadata (like block size, number of blocks, sample rate, and discarded units) is written to the binary file. This metadata will be crucial for the decoder to reconstruct the audio from the compressed format.

Code Block 30: Write Metadata Lossy Encoder

```
    BitStream outStream(outputFileName, ios::out);

    // write metadata to the bitstream to be read by the decoder (
       also using bitstream methods)
    outStream.writeBits(block_size, 16);
    outStream.writeBits(samples.size() / block_size, 16);
    outStream.writeBits(sfhIn.samplerate(), 16);
    outStream.writeBits(discarded_units, 16); // passed this to
       dinamically iterate in decoder
```

The code below outlines the core of a lossy audio compression algorithm.

It operates by dividing the audio data into fixed-size blocks and applying the Discrete Cosine Transform (DCT) to each block, transforming the time-domain samples into the frequency domain.

To achieve compression, the algorithm employs two main techniques: quantization and coefficient discarding. Quantization reduces the precision of the DCT coefficients, enabling them to be represented with fewer bits. Coefficient discarding omits the least significant coefficients, capitalizing on the fact that DCT often concentrates the energy of the signal into a few major coefficients.

Both techniques result in data reduction but also introduce errors, making the compression lossy. This means that the decompressed audio won't be an exact replica of the original but will be a close approximation, with the trade-off being a significantly smaller file size.

Code Block 31: Lossy Encoder Process Blocks

```
    // process blocks of samples
    for (size_t i = 0; i < samples.size(); i += block_size) // iterate over samples vector
    {
        for (size_t j = 0; j < block_size; ++j) // populate ‘x‘ vector with samples for DCT
        {
            x[j] = (i + j < samples.size()) ? samples[i + j] : 0.0; // padding with 0s if necessary
        }

        fftw_execute(plan_d); // apply DCT on the block of samples

        // quantize and discard coefficients (arg from command line)
        for (size_t j = 0; j < block_size - discarded_units; ++j)
        {
            int quantized = x_dct[j] * QUANTIZATION_FACTOR;
            outStream.writeBits(quantized, 32); // 32 bits per coefficient
        }
    }
```

To conclude, the DCT plan is destroyed, reventing potential memory leaks, and the program closes the output *bitstream*, ensuring that all data written during the compression process is saved correctly to the output file.

Code Block 32: Destroy DCT and Close File

```
    fftw_destroy_plan(plan_d); // clean fftw
    outStream.close(); // close the output bitstream
    return 0;
```

7.2.2 lossy_decoder.cpp

This code reads the encoded binary data, performs an inverse DCT on the quantized coefficients to reconstruct the audio samples, and then writes them to a WAV file. The process is lossy due to the quantization and discarding of certain coefficients during encoding.

As with the encoder, the variable for dequantization is also defined, and this value must match the quantization factor used during the encoding phase. It ensures that the decoding process correctly reverses the quantization applied during encoding.

Code Block 33: Dequantization Factor

```
const size_t QUANTIZATION_FACTOR = 10; // constant for dequantization
                                       // has to be the same as used in the encoder
```

The code below is responsible for initializing variables and gathering metadata from a binary file, which holds compressed audio data. Starting off, the code retrieves file paths from the command line arguments, with *inputFileName* capturing the path to the input binary file and *outputFileName* designating the path for a potential output file. It then establishes that the audio being worked upon is mono, a format with just a single channel, as indicated by *numChannels* being set to 1.

To read the compressed audio, a BitStream object named *inStream* is created, pointing to the input file for reading. Following this initialization, the code dives into searching metadata from this binary file. It retrieves four key pieces of information: *block_size* (indicating the number of samples in each audio block), *nBlocks* (total number of these blocks), *sampleRate* (the frequency at which audio samples were taken), and *discarded_units* (representing the number of coefficients that were intentionally discarded during an earlier compression phase).

Code Block 34: Lossy Decoder Get Metadata

```
    string inputFileName = argv[1]; // get input file
    string outputFileName = argv[2]; // get output file
    int numChannels = 1; // mono audios only have 1 channel

    BitStream inStream(inputFileName, ios::in); // open bin file

    // get metadata from bin file
    size_t block_size = inStream.readBits(16); // each sample block
       size
    size_t nBlocks = inStream.readBits(16); // get number of blocks
    int sampleRate = inStream.readBits(16);
    size_t discarded_units = inStream.readBits(16); // discarded
       units
```

A vector named *outputSamples* is initialized to accumulate the decoded audio samples. Its size is computed as the product of *block_size* and *nBlocks*, ensuring it has enough capacity to hold all the reconstructed audio data.

Then, two vectors, *x_dct* and x, are created. *x_dct* is designed to hold the Discrete Cosine Transform (DCT) coefficients, which are essentially the transformed values representing audio in the frequency domain. Conversely, x is created to store the inverse-transformed audio samples, essentially representing audio in the time domain, after the inverse DCT is applied.

The final line crafts an FFTW plan, a framework provided by the FFTW library. A plan named *plan_id* is made using the *fftw_plan_r2r_1d* function. This plan is configured to compute the inverse DCT on a 1D real array. The two main inputs for this function are pointers to the data regions of *x_dct* and *x*. When executed, this plan will transform the data in *x_dct* using inverse DCT and place the result in x. The *FFTW_ESTIMATE* flag suggests that the planner should aim for a balanced trade-off between planning time and execution time.

Code Block 35: Lossy Decoder FFTW

```
    vector<short> outputSamples(block_size * nBlocks); // vector to
       store the decoded audio samples

    // vectors for the DCT coefficients and their inverse
       transformation
    vector<double> x_dct(block_size);
    vector<double> x(block_size);

    fftw_plan plan_id = fftw_plan_r2r_1d(block_size, x_dct.data(), x.
       data(), FFTW_REDFT01, FFTW_ESTIMATE);
```

The outer loop below works as the core mechanism for decoding the audio, iterating over the number of audio blocks in the binary input.

Within this loop, the nested loop's primary task is to reconstruct these frequency domain representations. For each coefficient within a block, the program checks if it was one of the retained coefficients during the encoding process or if it was discarded.

If it was retained, the coefficient is read from the binary input stream, where it is stored as a 32-bit integer.

This coefficient is then dequantized by dividing it by the *quantization_factor*, essentially reversing the quantization process applied during encoding. The resultant value is stored in the *x_dct* vector.

On the other hand, if the coefficient was amongst those discarded during encoding, its value is explicitly set to zero in the *x_dct* vector.

Post this coefficient handling, the code employs the inverse DCT on the *x_dct* vector. This is a crucial step that takes the data from the frequency domain and transforms it back to the time domain.

The final nested loop within the main loop has the responsibility of storing the reconstructed audio samples. It ensures that each sample remains within the range suitable for 16-bit audio representation. Once processed, these samples are stored in the *outputSamples* vector.

Code Block 36: Lossy Decoder Core Process

```
    for (size_t i = 0; i < nBlocks; i++) // process blocks of
       coefficients
    {
        // read and dequantize coefficients
        for (size_t j = 0; j < block_size; ++j) // read and
           dequantize coefficients
        {
            if (j < block_size - discarded_units)
            {
                int quantized = inStream.readBits(32);
                x_dct[j] = quantized / static_cast<double>(
                   QUANTIZATION_FACTOR);
            }
            else
            {
                x_dct[j] = 0; // zero out the discarded coefficients
            }
        }

        fftw_execute(plan_id);  // get back the original audio
           samples with inverse DCT

        for (size_t j = 0; j < block_size; ++j) // save reconstructed
            samples, within the 16-bit range
        {
            outputSamples[i * block_size + j] = min(max(static_cast<
               int>(round(x[j])), -32768), 32767);
        }
    }
```

After processing the audio samples. it's important to ensure proper resource management, link in the encoder. So, the first line of the code below, fftw_destroy_plan(plan_id), is dedicated to this purpose.

Next, the input *bitstream*, from which the compressed audio data was read, is closed using the *inStream.close()* method.

Following the resource cleanup, the focus shifts to writing out the reconstructed audio data. Once the *SndfileHandle* instance is correctly set up, the reconstructed audio data stored in the *outputSamples* vector is written to the output file using the *writef* method. This method transfers the audio data from the vector to the file.

Code Block 37: Lossy Decoder FFTW Destroy and File Write

```
    fftw_destroy_plan(plan_id);
    inStream.close();

    // write output WAV file
    SndfileHandle sfhOut(outputFileName, SFM_WRITE, SF_FORMAT_WAV |
       SF_FORMAT_PCM_16, numChannels, sampleRate);
    sfhOut.writef(outputSamples.data(), outputSamples.size());

    return 0;
```

7.3 Usage

In order to execute the encoder, *lossy_encoder*, the command is as follows:

```
$ ./lossy_encoder <input_file> <output_file> <block_size> <
  discarded_units>
```

On the other hand, to run the decoder, *lossy_decoder*, the command is the one below:

```
$ ./lossy_decoder <input_file> <output_file>
```

7.4 Results

Within the *ex7_test_files* folder of the GitHub repository[2], you'll discover all the essential files for this exercise. The encoder's input file, *mono.wav*, and its corresponding binary output, output.bin, are both provided. Additionally, the folder contains the decoder's output audio file named *reconstructed_sample.wav*.

[2]⟨https://github.com/pedromonteiro01/ic/tree/main/trab1/ex7_test_files⟩

8 Workload and General Information

The repository for the project is ⟨https://github.com/pedromonteiro01/ic⟩.
Each team member made an equitable contribution to the project's resolution.