Project IC - First Project

Universidade de Aveiro

Bernardo Marçal, Ricardo Machado, Rui Campos

Project IC - First Project

Informação e Codificação

Universidade de Aveiro

Bernardo Marçal, Ricardo Machado, Rui Campos
(103236) bernardo.marcal@ua.pt, (102737) ricardo.machado@ua.pt
(103709) ruigabriel2@ua.pt

30 de outubro de 2023

Índice

Capítulo 1

Introduction

In the ever-evolving landscape of technology and data, the science of Information and Coding plays a pivotal role in the efficient storage, transmission, and manipulation of information. In this project where we delve into the intricacies of information theory and explore the practical applications of data transformation.

Our project is an exploration of how information can be systematically encoded, manipulated, and decoded to meet various needs. This journey takes us into the world of digital files, where we will learn how to compress, modify, and reconstruct data in various forms, including text, audio, and more. Through hands-on experiences and theoretical understanding, we aim to unravel the mechanisms behind these processes.

The project's repository can be viewed here: Github.

All the command codes to test are available at the README file.

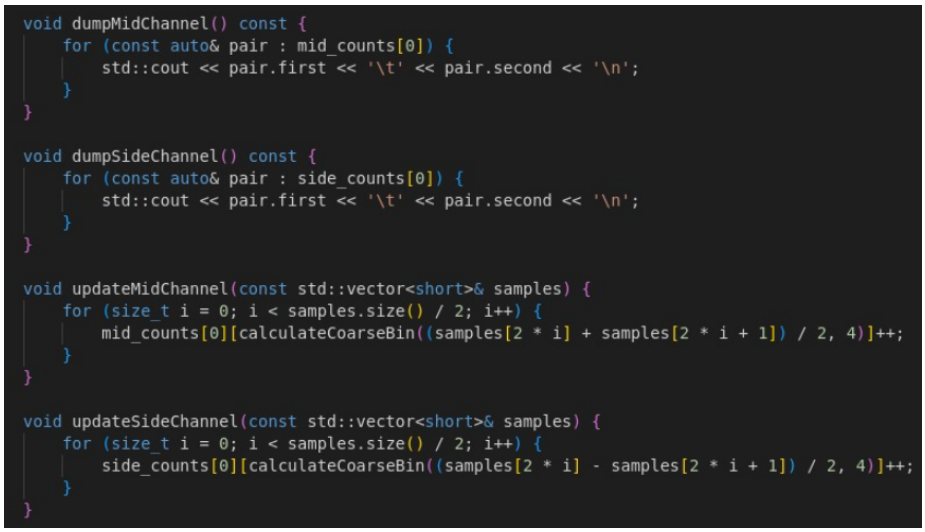
Capítulo 2

Project Workthrough

2.1 Exercise 1

2.1.1 Source Code

For this to work we created a Dump and an Update function for each Mid and Side Channels, the dump function for each basically does the same but in this case it's just in specific for each channel



```
void dumpMidChannel() const {
    for (const auto& pair : mid_counts[0]) {
        std::cout << pair.first << '\t' << pair.second << '\n';
    }
}

void dumpSideChannel() const {
    for (const auto& pair : side_counts[0]) {
        std::cout << pair.first << '\t' << pair.second << '\n';
    }
}

void updateMidChannel(const std::vector<short>& samples) {
    for (size_t i = 0; i < samples.size() / 2; i++) {
        mid_counts[0][calculateCoarseBin((samples[2 * i] + samples[2 * i + 1]) / 2, 4)]++;
    }
}

void updateSideChannel(const std::vector<short>& samples) {
    for (size_t i = 0; i < samples.size() / 2; i++) {
        side_counts[0][calculateCoarseBin((samples[2 * i] - samples[2 * i + 1]) / 2, 4)]++;
    }
}
```

Figura 2.1: Dump and Update for Side and Mid Channels

2.1.2 Results

This is the graphics of each channel we produced:

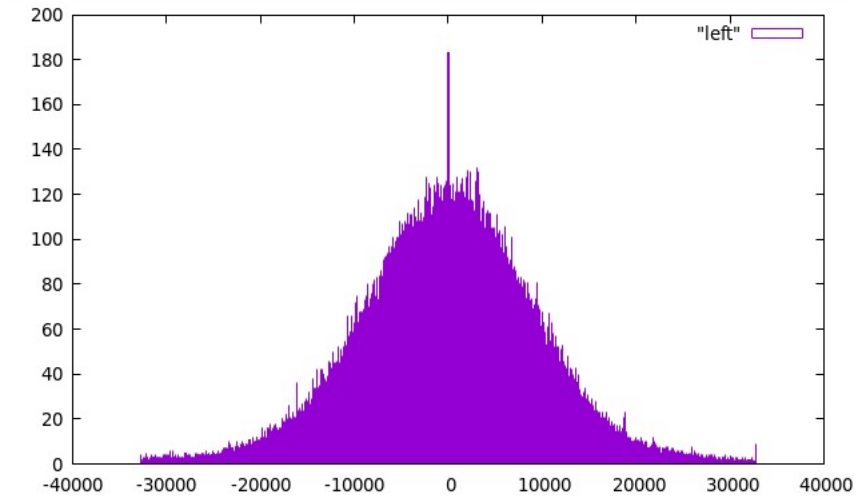


Fig. 2.2: Left Channel Graph

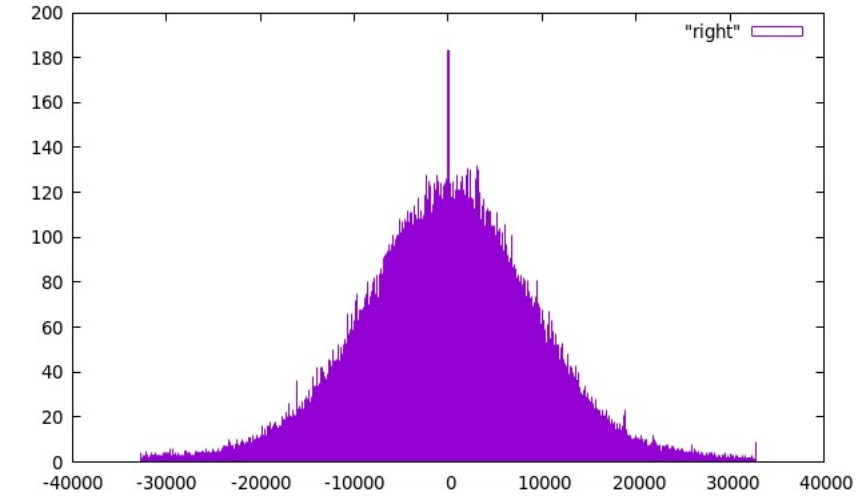


Fig. 2.3: Right Channel Graph

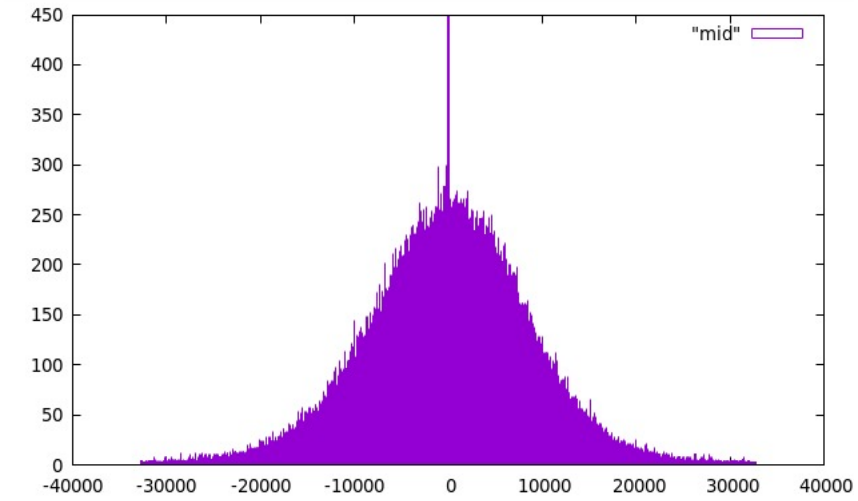


Fig. 2.3: Mid Channel Graph

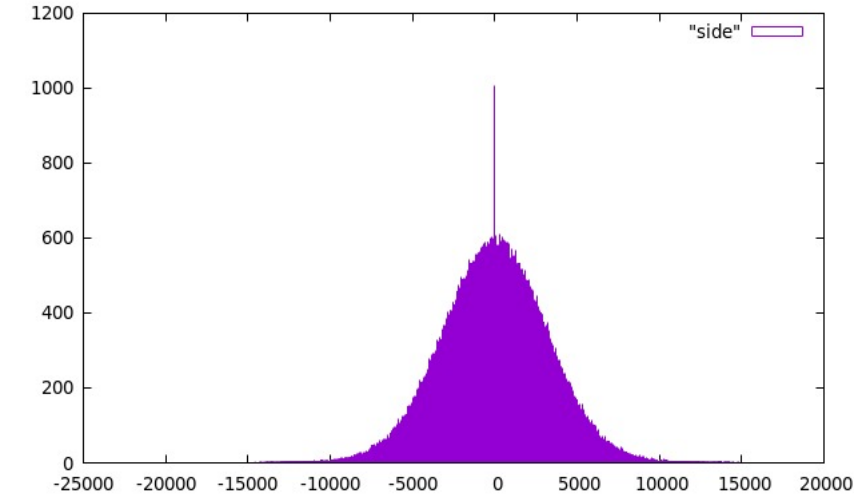


Fig. 2.4: Side Channel Graph

2.1.3 Observations and Conclusions

After looking at the charts for each audio channel, it's clear that the left and right channels are quite similar. This similarity means that when we combine them to create the mid channel, we anticipate the mid channel's chart to also be quite similar to the individual channels. The side channel shows how different the left and right audio channels are. Since both channels have similar volumes, we expect the histogram to have a peak around the value of 0. This peak suggests that, on average, the audio in both channels is in sync or has very little phase difference.

2.2 Exercise 2

2.2.1 Source Code

```
while ((nFrames = sfhIn1.readf(samples_f1.data(), FRAMES_BUFFER_SIZE))) {
    sfhIn2.readf(samples_f2.data(), FRAMES_BUFFER_SIZE);

    samples_f1.resize(nFrames * sfhIn1.channels());
    samples_f2.resize(nFrames * sfhIn2.channels());

    for (long unsigned int i = 0; i < samples_f1.size(); i++) {
        double signal_sample = static_cast<double>(samples_f1[i]);
        double noise_sample = signal_sample - static_cast<double>(samples_f2[i]);

        total_energy_signal += signal_sample * signal_sample;
        total_energy_noise += noise_sample * noise_sample;

        double abs_error = abs(noise_sample);
        max_error = max(abs_error, max_error);

        int channel_index = i % channel_count;
        channel_errors[channel_index] += noise_sample * noise_sample;
        channel_max_errors[channel_index] = max(abs_error, channel_max_errors[channel_index]);
    }
}
```

Fig. 2.5: Principal code for WaveCmp

2.2.2 Results

The code calculates the Signal-to-Noise Ratio (SNR) in decibels (dB). SNR is a measure of the quality of the audio signal relative to the background noise. A higher SNR indicates better audio quality. The result is printed as "SNR: X dB,"where X is the calculated SNR value. The code processes the audio files frame by frame, calculating these metrics in real-time, and then presents the results at the end of the process. These results provide insights into the quality and differences between the two audio files, making it useful for various audio processing and comparison tasks, such as quality assessment and audio analysis.

For two identical audio files, we saw low values for all metrics, indicating high audio quality and no differences. If one audio file is a clean, high-quality recording, and the other is a noisy or degraded version of the same content, we saw expect the SNR of the clean file to be higher, and the L2 Norm and Max Absolute Error values for the noisy file to be higher.

When we put a stereo audio and one channel is significantly distorted or contains noise while the other is clean, we saw higher metrics for the distorted channel in terms of L2 Norm and Max Absolute Error.

As we can see, the results got were expected because the snr for 6bits is higher than snr for 4 and 2 bits. In addition to that, the snr for 2 bits is lower

than zero, which is expected knowing that we removed 14 bits of resolution from the sample.

2.3 Exercise 3

2.3.1 Source Code

```
size_t bits = 16 - leftbits;
vector<short> samples(FRAMES_BUFFER_SIZE * sfhIn.channels());
WAVQuant wavquant { };

size_t framesps;
while((framesps = sfhIn.readf(samples.data(), FRAMES_BUFFER_SIZE))) {
    samples.resize(framesps * sfhIn.channels());
    wavquant.quant(samples, bits);
}

wavquant.toFile(sfhOut);
```

Fig. 2.6: Principal code for WaveQuant

2.3.2 Results

The input audio file will be read and quantized based on the specified number of bits (leftbits), which determines the quantization step size. A smaller number of bits will lead to more aggressive quantization (loss of audio fidelity), while a larger number of bits will result in less quantization (higher audio quality).

We note a substantial drop in audio quality we can expect to only get 2N different values where N is the number of bits kept in the audio file. For a sample with 6bits we expect to have $26 = 64$ bars.

2.4 Exercise 4

2.4.1 Source Code

The effects that we implemented were: Single Echo, Multiple Echos and Amplitude Modulation.

```
if (wanted_effect == "single_echo" || wanted_effect == "multiple_echo") {
    while((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) {
                if (wanted_effect == "single_echo") {
                    sample_out = (samples.at(i) + gain * samples.at(i - delay)) / (1 + gain);
                } else if (wanted_effect == "multiple_echo") {
                    sample_out = (samples.at(i) + gain * samples_out.at(i - delay)) / (1 + gain);
                }
            } else {
                sample_out = samples.at(i);
            }

            samples_out.insert(samples_out.end(), sample_out);
        }
    }
```

Fig. 2.7: Code for Single and Multiple Echos

```
else if (wanted_effect == "amplitude_modulation") {
  while((nFrames = sfhIn.readf(samples.data(), FRAMES_BUFFER_SIZE))) {
      samples.resize(nFrames * sfhIn.channels());

      for (int i = 0; i < (int)samples.size(); i++) {
          sample_out = samples.at(i) * cos(2 * M_PI * (freq/sfhIn.samplerate()) * i);
          samples_out.insert(samples_out.end(), sample_out);
      }
  }
```

Fig. 2.8: Code for Amplitude Modulation

2.4.2 Results

We created a file with single echo, 4 of gain and 40000Hz of delay (singleecho.wav), one audio file with multiple echos, 1 of gain and 40000Hz of delay (multipleecho.wav), another file with amplitude modulation of 3Hz (ampmod.wav) and finally we created an audio file that is the reverse of sample.wav (reverse.wav).

Then we started by listening the audio files, we can clearly see that singleecho.wav and multipleecho.wav have an echo effect. Finally, we note that multiple echos creates a more distorted audio file.

2.5 Exercise 5

2.5.1 Source Code

We've implemented an efficient approach for writing files, ensuring that it scales linearly with the file size. The writing procedure relies on an integer values array, which is initialized and initially filled with zeros and ones, typically representing binary data.

When the "writeBit"and "writeBits"functions are employed, this array accumulates information from the input file until it reaches its capacity, which is 8 units, with each unit representing a single bit. At this point, these collected bits are converted into a byte and appended to the end of the output file.

To conclude the writing process and ensure that the last bits are correctly saved, the "close()"function takes care of writing the remaining bits present in the array, even if only a single bit remains. It then pads the remaining space with zeros and transforms them into a byte. Additionally, this function is responsible for closing the open file, be it in read or write mode, as needed.

```
void writeBits(std::vector<int> bits) {
    if (modef != "w") {
        std::cout << "File is not open for writing" << std::endl;
        return;
    }

    for (int i = 0; i < bits.size(); i++) {
        if (currentBitPos == 0) {
            bitArray = std::vector<int>(8);
        }
        bitArray[currentBitPos] = bits[i];
        currentBitPos = (currentBitPos + 1) % 8;

        if (currentBitPos == 0) {
            char byte = bitArrayToByte(bitArray);
            file.write(&byte, 1);
        }
    }
}

void writeBit(int bit) {
    if (modef != "w") {
        std::cout << "File is not open for writing" << std::endl;
        return;
    }

    if (currentBitPos == 0) {
        bitArray = std::vector<int>(8);
    }
    bitArray[currentBitPos] = bit;
    currentBitPos = (currentBitPos + 1) % 8;

    if (currentBitPos == 0) {
        char byte = bitArrayToByte(bitArray);
        file.write(&byte, 1);
    }
}
```

Fig. 2.9: Code for BitStream write mode

To enhance efficiency in terms of both memory and execution time when working with an object of the BitStream type, you must specify the mode "r"(read) or "w"(write) based on the intended operation.

In read mode, the file is read in binary format, allowing it to interpret only binary values, which are typically zeros and ones. When using the "readBit"or "readBits"functions, one byte of the file's content is read into a temporary array, serving as a buffer. This byte is then converted by splitting it into 8 individual bits.

Every time the array is fully read and processed, the next byte of the file content is extracted to the buffer, doing this process consecutively. This way, the program becomes more efficient in terms of memory because it will only read another byte when necessary.

```
std::vector<int> readBits(int n) {
    if (modef != "r") {
        std::cout << "File not open for reading" << std::endl;
        return std::vector<int>();
    }

    std::vector<int> outBits;

    while (n > 0) {
        if (currentBitPos == 0) {
            char byte;
            file.read(&byte, 1);
            bitArray = byteToBitArray(byte);
        }

        outBits.push_back(bitArray[currentBitPos]);
        currentBitPos = (currentBitPos + 1) % 8;
        n--;
    }

    return outBits;
}

int readBit() {
    if (modef != "r") {
        std::cout << "File not open for reading" << std::endl;
        return -1;
    }

    if (currentBitPos == 0) {
        char byte;
        file.read(&byte, 1);
        bitArray = byteToBitArray(byte);
    }

    int bit = bitArray[currentBitPos];
    currentBitPos = (currentBitPos + 1) % 8;

    return bit;
}
```

Fig. 2.10: Code for BitStream read mode

2.6 Exercise 6

2.6.1 Source Code

The purpose of this task is to validate the proper implementation of the BitStream class using two programs: an encoder and a decoder. These programs operate on a text file that exclusively contains binary values (zeros and ones).

Here's how the encoder works:

The encoder begins by reading the input file provided as an argument. It assumes that the file contains only binary values (0s and 1s) and does not include other characters such as letters, spaces, or newlines.

It creates an instance of the BitStream object in write mode, specifying the output file's name as an argument.

For each value it reads from the input file, whether it is zero or one, the encoder converts it to an integer and appends it to the bits array. This array accumulates all the bits from the file in the order they were read.

After collecting all the bits, the encoder employs the "writeBits"function to write this information to a binary file.

To finalize the file writing process, the encoder invokes the "close"method of the BitStream class to ensure that the file is properly closed.

The encoder is responsible for converting a text file containing binary values into a binary file with the assistance of the BitStream class. The binary file will accurately represent the same data in binary format.

```
BitStream outputFile (outputFileName, "w") ;

vector<int> bits;
for (int i = 0; i < line.length(); i++){
    bits.push_back(line[i] - '0');
}
outputFile.writeBits(bits);
outputFile.close();

return 0;
```

Fig. 2.11: Code for BitStream output

The decoder takes the encoded file as an argument and specifies the desired output file name. It starts by creating a new BitStream object configured for reading mode.

To determine the size of the encoded file in bytes, the decoder employs the "getFileSize"method, which helps establish the total number of bits required to read the entire content of the file.

With this information about the file's size in terms of bits, the decoder invokes the "readBits"method, passing the determined size as an argument. This method retrieves the binary data and writes it to a text file, effectively reversing the encoding process and restoring the original content.

```
BitStream inputFile (argv [1], "r") ;

ofstream outputFile (argv [2], ios::out) ;
if (! outputFile) {
    cerr << "Invalid output file " << argv [2] << ".\n" ;
    return 1 ;
}

vector<int> bits;
bits = inputFile.readBits(inputFile.getFileSize() * 8);
inputFile.close();

for (int i = 0; i < bits.size(); i++){
    outputFile << bits[i];
}
outputFile.close();

return 0;
```

Fig. 2.12: Code for BitStream input

2.6.2 Results

To test encoder.cpp and decoder.cpp we started with a text file "texto.txt". Then, we used decoder.cpp to convert the text.txt to a file that contains the binary equivalent using the characters "1" and "0".

With the results that we has, we can say that the decode/enconde processing time depends of the file size, and we see that this correlation is linear viewing the trending line.

2.7 Exercise 7

2.7.1 Source Code

In this exercise, we've got two programs, just like the last one.

The encoder, which is now referred to as "encoderloss"accepts several arguments: the input audio file (.wav) to be modified, the output binary file, the block size, and the number of discarded units per block. In this case, it directly applies the Discrete Cosine Transform (DCT) to the samples of the input audio file, retaining only "blockSize - discarded"units per block. This selective retention of units preserves the most crucial frequencies of the audio file, specifically the lower frequencies.

```
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 c = 0 ; c < nChannels ; c++) {
        for(size_t k = 0 ; k < bs ; k++)
            x[k] = samples[(n * bs + k) * nChannels + c];

        fftw_execute(plan_d);

        for(size_t k = 0 ; k < bs - discarded_units_per_block ; k++){
            x_dct[c][n * bs + k] = x[k] / (bs << 1) * 100;
        }
        tmp++;

    }
BitStream outputFile (outputFileName, "w") ;
vector<int> bits;
```

Fig 2.13: Code for Encoder Loss

To facilitate the decoding process, certain critical information must be retained alongside the DCT coefficients (denoted as x dct). This essential data includes the number of blocks, the number of channels, the original file's sample rate, and the number of frames. All of these are indispensable for reconstructing the new audio file. These values serve as the header of the binary file, and each of them must be correctly converted into binary format. It's important to consider the representation resolution: 16 bits for header values (except the number of frames), and 32 bits for the DCT coefficients. Additionally, these coefficients are saved after being multiplied by 100. This ensures that during the decoding process, the inverse DCT is performed with higher resolution (resulting in numbers with two decimal places) compared to integer values.

The lossy decoder requires two arguments: the encoded binary file and the name of the output audio file to be created and written after the decoding process.

The decoding process begins by reading the initial bits of the header from the binary file. These bits are then correctly converted into their respective

integers, which include blockSize, nBlocks, nChannels, sampleRate, and nFrames. Subsequently, before initiating the decoding operation, an output audio file is created with the provided name, the specified number of channels, and the designated sample rate.

```
for(int i = 0; i < x_dct_bits.size(); i+=32) {
    int temp = 0;

    vector<int> reversed_temp;

    for(int j = 31; j >= 0; j--) {
        reversed_temp.push_back(x_dct_bits[i+j]);
    }
    for(int j = 0; j < reversed_temp.size(); j++) {
        temp += reversed_temp[j] * pow(2, reversed_temp.size() - j - 1);
    }
    tmp.push_back(temp);
}

bitStream.close();

int count = 0;
for(int n = 0; n < nBlocks; n++) {
    for(int c = 0; c < nChannels; c++) {
        for (int k = 0; k < bs; k++) {
            x_dct[c][n*bs + k] = tmp[count]/100.0;
            count++;
        }
    }
}
```

Fig. 2.14: Code for convert

The remaining bits in the file exclusively pertain to the encoded x dct. Consequently, they are read and subsequently converted to their respective values. These values involve a conversion from 32-bit integers to doubles, ensuring a higher resolution with two decimal places. With the x dct coefficients and the header's remaining values, it becomes possible to perform the inverse DCT, reconstruct the samples array, and then write it to the output audio file.

```
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(size_t 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);
```

Fig. 2.15: Code for Decoder Loss

2.7.2 Results

Upon executing these two codes, we can anticipate obtaining an output audio file, which should ideally represent a decoded rendition of the original audio file. However, the outcome may entail some intricacies. The quality of the resulting audio is intricately linked to the chosen encoding parameters, particularly the block size and the number of discarded units. The extent of quality variation is influenced by a combination of encoding settings and the inherent attributes of the input audio file.

It is important to note that the encoding process is inherently lossy, meaning that some level of data fidelity is inevitably sacrificed to achieve compression. Consequently, the audio quality of the decoded output may not be an exact replica of the original. The degree of quality degradation, if present, is contingent upon the chosen settings and the characteristics of the audio material. The trade-off between compression and audio fidelity must be carefully considered when selecting encoding parameters. Therefore, it is advisable to fine-tune the encoding settings to strike an optimal balance between file size reduction and audio quality preservation.

Capítulo 3

Repository and Contributions

If there is any doubts about any part of the code that was put into this report, here is the github repository that has every part of the code that we have worked on and every file:

https://github.com/BennyMarcal/IC_Project

All the compilation methods are inserted in the Readme in the main page of the repository The contributions of the group work to this project are the following:

Bernardo Marçal (103236) - 33%
Ricardo Machado (102737) - 33%
Rui Campos (103709) - 33%