Universidade de Aveiro

Information and Coding

Lab Work nr.1

André Clérigo (98485), João Amaral (98373), Pedro Rocha (98256)

Departamento de Eletrónica, Telecomunicações e Informática

October 30, 2022

Contents

List of Figures

Chapter 1

General Information

The contributions between each member of the group were equal.
The project's repository can be viewed here: `https://github.com/PedroRocha9/IC`
In a side note, it's important to inform that this repository was public throughout the development of the project.

Chapter 2

Exercise 2

2.1 Code

For wav_hist.h we added the following variables to replicate the existing behaviour but adding the feature for mid and side channels.

```
std::vector<std::map<short, size_t>> mid_counts;
std::vector<std::map<int, size_t>> side_counts;
```

In the same line of thought, we added update_mid, update_side, mid_dump and side_dump functions, which replicate the update and dump functions but channel specific.

```
void mid_dump() const {
    for(auto [value, counter] : mid_counts[0])
        std::cout << value << '\t' << counter << '\n';
}

void side_dump() const {
    for(auto [value, counter] : side_counts[0])
        std::cout << value << '\t' << counter << '\n';
}

void update_mid(const std::vector<short>& samples) {
    for(long unsigned int i = 0; i < samples.size()/2; i++)
        mid_counts[0][(samples[2*i] + samples[2*i+1]) / 2]++;
}

void update_side(const std::vector<short>& samples) {
    for(long unsigned int i = 0; i < samples.size()/2; i++)
        side_counts[0][(samples[2*i] - samples[2*i+1]) / 2]++;
}
```

The mid and side channel values were calculated by doing the adding/-subtracting of left and right channel, and diving the result by the number of channels. In this case, the left channel is always the even number index (2*i) and the right channel is always the odd channel (2*i + 1).

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

```
string mode { argv[argc-1] };
int channel {};
if (mode != "mid" && mode != "side") {
    try {
        channel = stoi(argv[argc-1]);
    } catch(exception &err) {
        cerr << "Error: invalid mode requested\n";
        return 1;
    }

    if(channel >= sndFile.channels()) {
        cerr << "Error: invalid channel requested\n";
        return 1;
    }
}
```

Additionally we added the following lines inside the given while loop to get the mid and side values, and finally we only dumped the desired channel values.

```
while(...) {
    ...
    hist.update_mid(samples);
    hist.update_side(samples);
}

if (mode == "mid") {
    hist.mid_dump();
} else if (mode == "side") {
    hist.side_dump();
} else {
    hist.dump(channel);
}
```

2.2 Usage

The usage of the program is done following this pattern:

```
./wav_hist <input file> <channel | mid | side>
```

Where the input file is an audio file (tested with .wav) and the channel is an integer.
Additionally we redirected the program's output to a file with the > operand, and then used gnuplot to plot the desired histogram (plot "<filename>" with boxes).

2.3 Results

The results we got with gnuplot were the following:

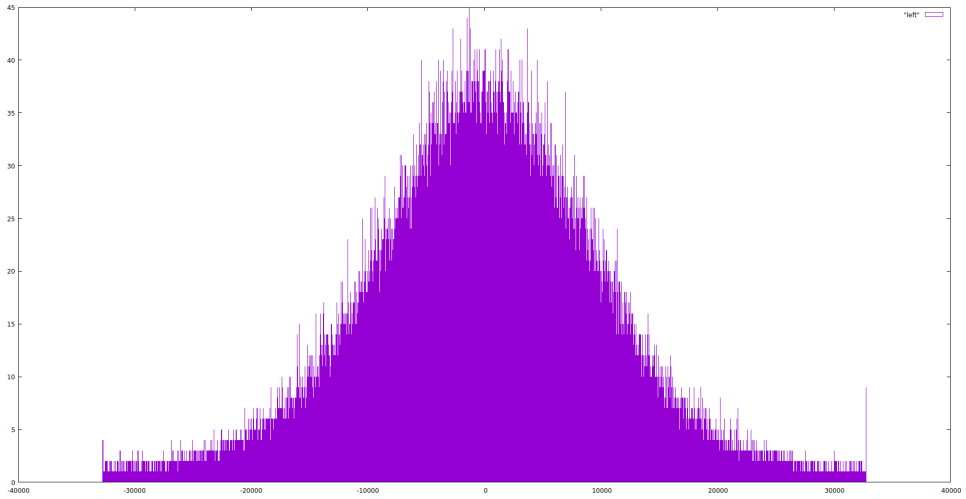

Figure 2.1: Gnuplot histogram for left channel histogram

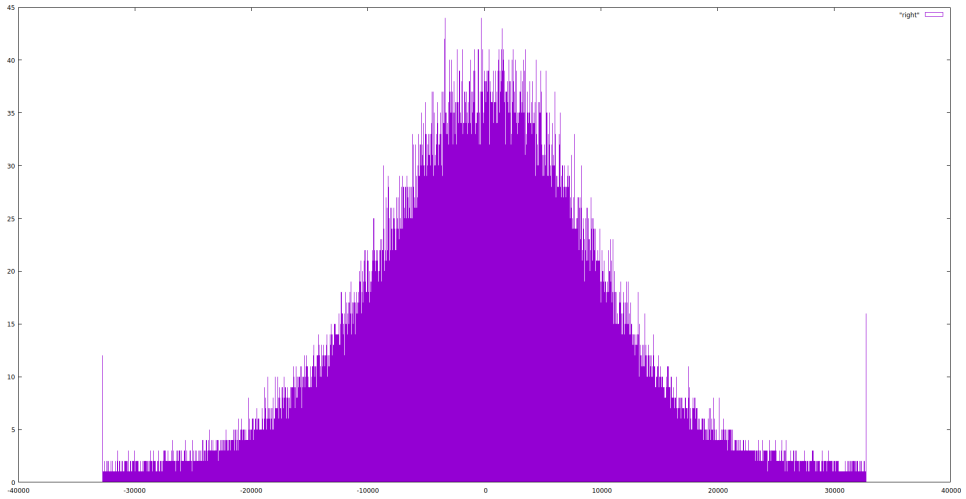

Figure 2.2: Gnuplot histogram for right channel histogram

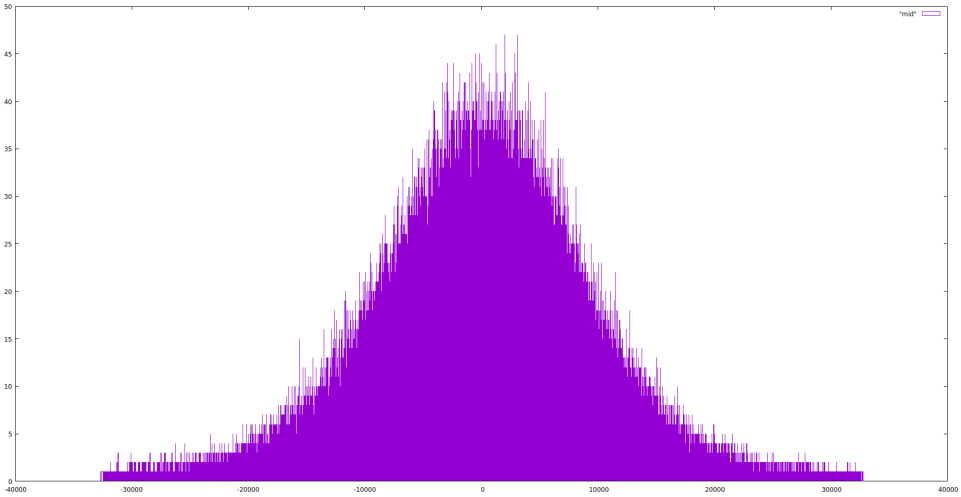

Figure 2.3: Gnuplot histogram for mid channel histogram

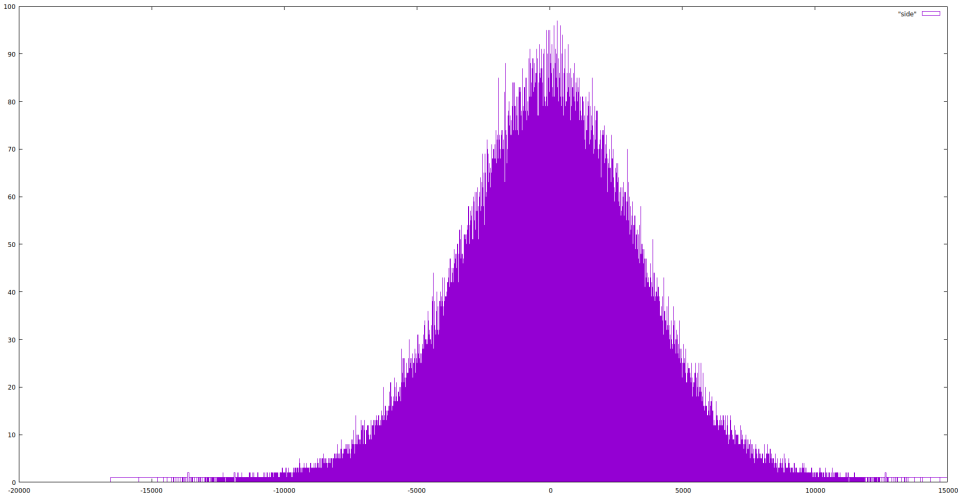

Figure 2.4: Gnuplot histogram for side channel histogram

Analyzing the Figure 2.1 and Figure 2.2 we can see that the results got on Figure 2.3 and Figure 2.4 were expected.

Since, left and right channel operate within similar amplitudes, we expect that the mid channel (which is an average of the two) has a similar amplitude domain has these channels, also, we expect that the frequency of each amplitude reached to be similar. The side channel is the difference between the two channels, with that said, and knowing the left and right channel operate within similar amplitudes, we expect the histogram to be concentrated around the value ”0”.

Our results show the expected behaviour for mid and side channel.

Chapter 3

Exercise 3

3.1 Code

Our wav_quant.h has a private vector that stores the quantified sample values and we also have a function (quant) to quantify a given sample. We quantify the sample by shifting right the number of bits that we want to cut, and then shift the same amount left, this way, we are "replacing" the old values with zeros.

```
private:
    std::vector<short> quant_samples;

public:
    ...

    void quant(const std::vector<short>& samples, size_t
    num_bits_to_cut) {
        for (auto sample : samples) {
            sample = sample >> num_bits_to_cut;
            short tmp = sample << num_bits_to_cut;
            quant_samples.insert(quant_samples.end(), tmp);
        }
    }
```

In wav_quant.cpp we simply use the quant function and then write the quantified samples to an output audio file.

```
while(...) {
    quant.quant(samples, num_bits_to_cut);
}

quant.toFile(sfhOut);
```

3.2 Usage

The usage of the program is done following this pattern:

```
./wav_quant <input file> <bits_to_keep> <output_file>
```

Where the input/output files are audio files (tested with .wav) and the bits_to_keep an integer.
In addition to playing the quantified audio file we used wav_hist to make sure that the quantization was done correctly.

3.3 Results

The results we got with gnuplot were the following:

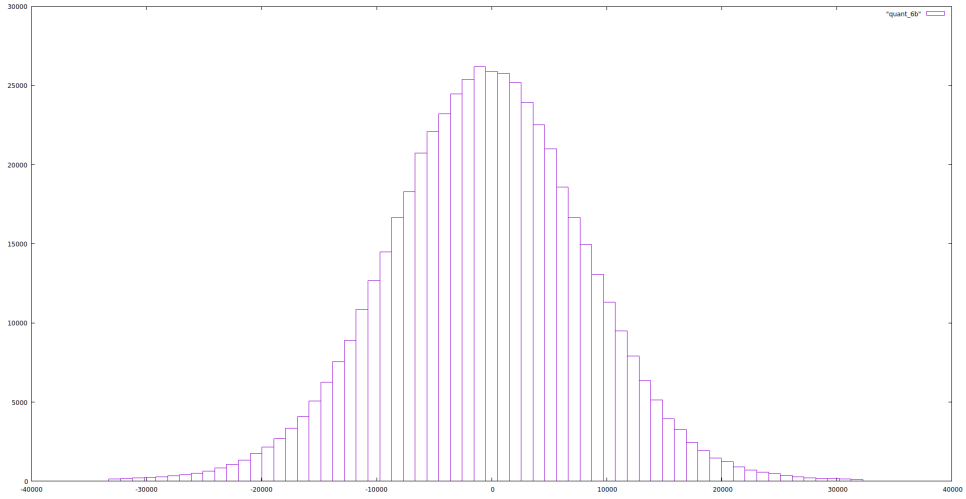

Figure 3.1: Gnuplot histogram for audio file with only 6bits of resolution

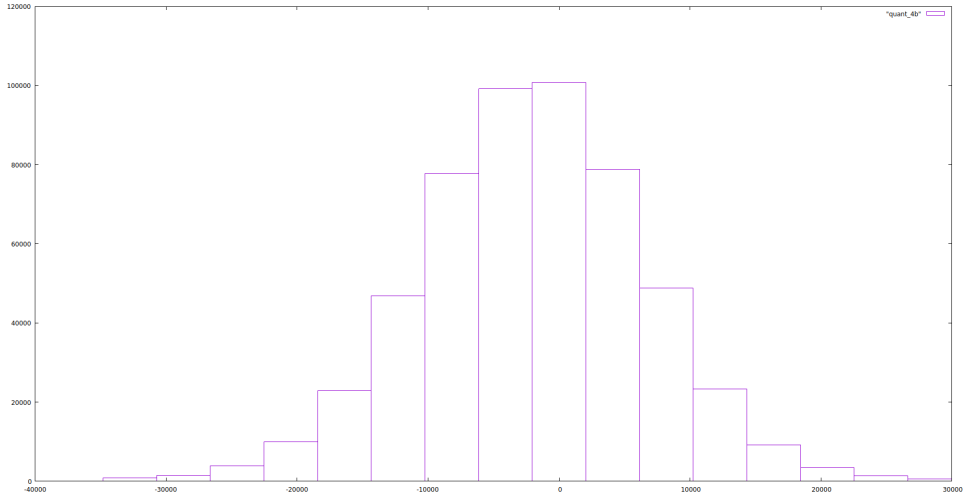

Figure 3.2: Gnuplot histogram for audio file with only 4bits of resolution

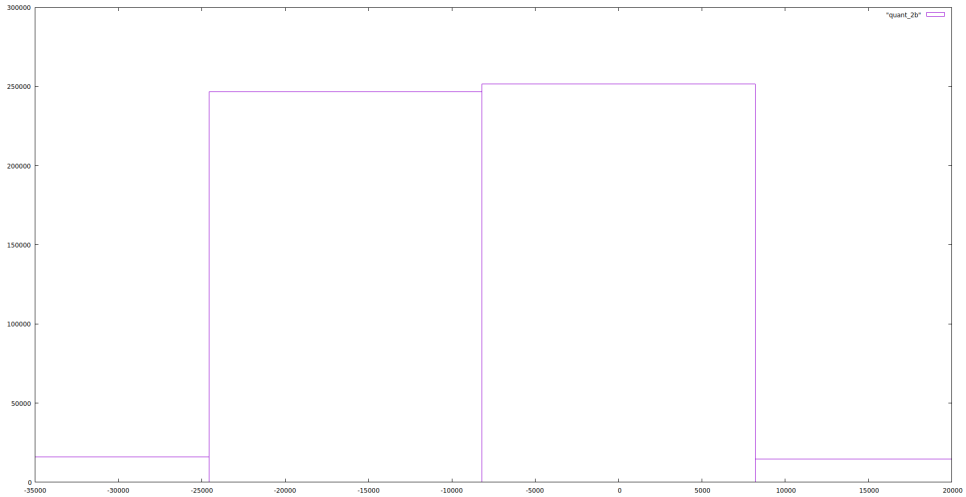

Figure 3.3: Gnuplot histogram for audio file with only 2bits of resolution

In addition to notice a substantial drop in audio quality we can expect to only get 2^N different values where N is the number of bits kept in the audio file. For a sample with 6bits we expect to have $2^6 = 64$ bars (different values), which can be viewed on the on Figure 3.1. For a sample with 4bits we expect to have $2^4 = 16$ bars, which can be viewed on the on Figure 3.2. For a sample with 2bits we expect to have $2^2 = 4$ bars, which can be viewed on the on Figure 3.3.

To further confirm our results we added the wav_quant6b.wav, wav_quant4b.wav and wav_quant2b.wav audio tracks to Audacity and got the follwing results.

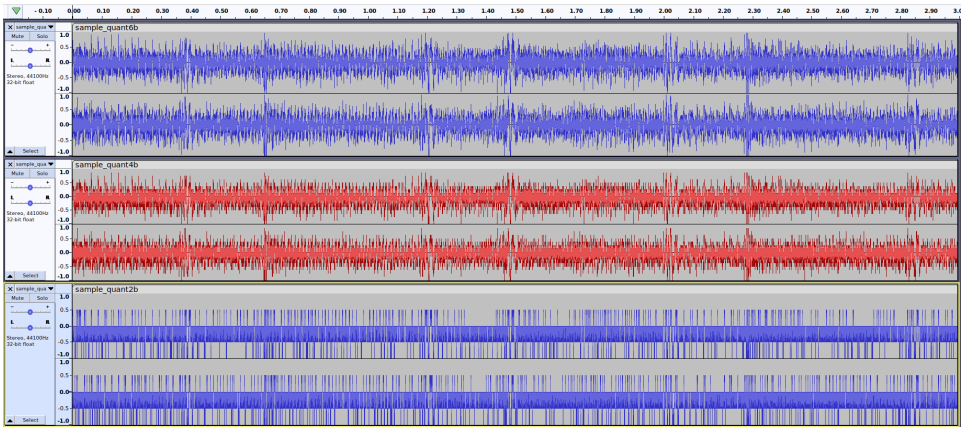

Figure 3.4: Audacity waveforms for quantified audio files

We can see on Figure 3.4 that the more bits we remove from the original file, the greater the loss of signal quality.

Chapter 4

Exercise 4

4.1 Code

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

$$r(n) = x(n) - \tilde{x}(n), \quad \text{where } \tilde{x} \text{ is the quantified sample}$$

$$\mathcal{E}_x = \sum_n |x(n)|^2, \quad \mathcal{E}_r = \sum_n |r(n)|^2$$

$$SNR = 10 \times \log_{10} \frac{\mathcal{E}_x}{\mathcal{E}_r} \text{ dB (decibel)}$$

$$\mathcal{E}_{max} = \frac{\Delta}{2}$$

We implement these expressions with the following algorithm:

```
while(...) {
    for (long unsigned int i = 0; i < samples_f1.size(); i++) {
        energy_signal += abs(samples_f1[i])^2;
        energy_noise += abs(samples_f1[i] - samples_f2[i])^2;
        max_error = abs(samples_f1[i] - samples_f2[i]) >
    max_error ? abs(samples_f1[i] - samples_f2[i]) : max_error;
    }
}

snr = 10 * log10(energy_signal / energy_noise);
```

4.2 Usage

The usage of the program is done following this pattern:

```
./wav_cmp <input_file> <input_file_2>
```

Where both input files are audio files (tested with .wav).

4.3 Results

A Signal-to-noise ratio is a measure of the amount of background noise with respect to the primary input signal. It is formally defined as the ratio of signal power to the noise power, and is often expressed in decibels. A ratio higher than 1:1 (greater than 0 dB) indicates more signal than noise, with that in mind, we expect to have values greater than zero when using a file with a few bits lost and a value bellow zero when using a file with a lot of bits lost.

We tested the program giving the first file argument as the original audio file, and the second file a quantified audio file.

When running the program with file 2 being a quantified version keeping 6bits we get: SNR = 11.215 dB and Maximum Absolute Error = 1023.

When running the program with file 2 being a quantified version keeping 4bits we get: SNR = 5.187 dB and Maximum Absolute Error = 4095.

When running the program with file 2 being a quantified version keeping 2bits we get: SNR = -0.822 dB and Maximum Absolute Error = 16383.

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.

Chapter 5

Exercise 5

5.1 Code

The effects that we implemented were: Single Echo, Multiple Echos, Amplitude Modulation and Reverse. We used the formulas studied in the Theoretical classes for Single Echo, Multiple Echos and Amplitude Modulation.

Single: $y(n) = x(n) + \alpha \times x(n - delay)$, Multiple: $y(n) = x(n) + \alpha \times y(n - delay)$

Where y is the new modified sample and x is the original sample

```
if (wanted_effect == "single_echo" || wanted_effect == "
    multiple_echo") {
    while(...) {
        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);
        }
    }
}
```

After 10 runs, we got an average processing time of 44,40 ms for single echo and 70,67 ms for multiple echos. In addition we divide the modified sample by (1 + gain) to guarantee that the modified sample doesn't go above the maximum value.

$$\text{Amplitude Modulation: } y(n) = x(n) \times \cos(2 \times \pi \times \frac{f}{f_a} \times n)$$

```
else if (wanted_effect == "amplitude_modulation") {
    while(...) {
        for (int i = 0; i < (int)samples.size(); i++) {
            sample_out = samples.at(i) * cos(2 * M_PI * (1.0/
    sfhIn.samplerate()) * i);
            samples_out.insert(samples_out.end(), sample_out);
        }
    }
}
```

After 10 runs, we got an average processing time of 104,25 ms. In addition we can listen to the audio file and ear the constant pulsation in the music.

For the reverse effect the only thing we do is to write the samples to the new file in the reverse order using the index value.

```
else if(wanted_effect == "reverse") {
    while(...) {
        for (int i = (int)samples.size() - 1; i >= 0; i--)
    samples_out.insert(samples_out.end(), samples.at(i));
    }
}
```

After 10 runs, we got an average processing time of 30,86 ms.

5.2 Usage

The usage of the program is done following this pattern:

```
./wav_effects <input_file> <output_file> <wanted_effect> [delay
    | freq] [gain]
```

For single/multiple echo we should run the program like:

```
./wav_effects <input file> <output_file> single/multiple_echo <
    delay > <gain>
```

For amplitude modulation we should run the program like:

```
./wav_effects <input file> <output_file> amplitude_modulation <
    freq >
```

For revese we should run the program like:

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

Where the input and output files are audio files (tested with .wav), the delay/freq are integers and the gain is a double/float.

5.3 Results

We created a file with single echo, 2 of gain and 44100Hz of delay (single_echo_2_44100.wav), one audio file with multiple echos, 0.8 of gain and

44100Hz of delay (multiple_echo_2_44100.wav), another file with amplitude modulation of 2Hz (amp_mod_2hz.wav) and finally we created an audio file that is the reverse of sample.wav (reverse.wav) and another the reverse of reverse.wav (reverse_of_reverse.wav).

Then we started by listening the audio files, we can clearly see that single_echo_2_44100.wav and multiple_echo_2_44100.wav have an echo effect, noting that multiple echos creates a more distorted audio file.

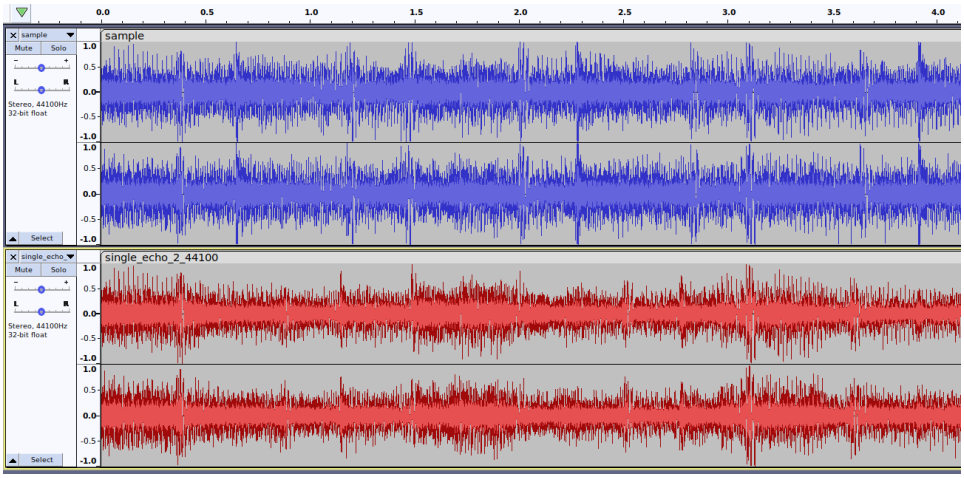

Figure 5.1: Comparing waveforms between sample.wav and a single echo with delay of 44100Hz and gain of 2

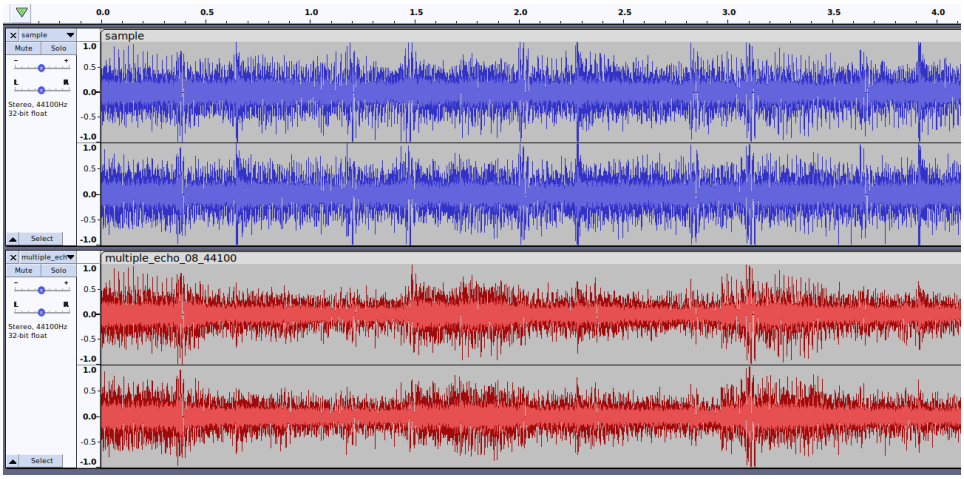

Figure 5.2: Comparing waveforms between sample.wav and multiple echos with delay of 44100Hz and gain of 0.8

Analyzing Figure 5.1 and Figure 5.2 we can see clearly the audio files have been modified.

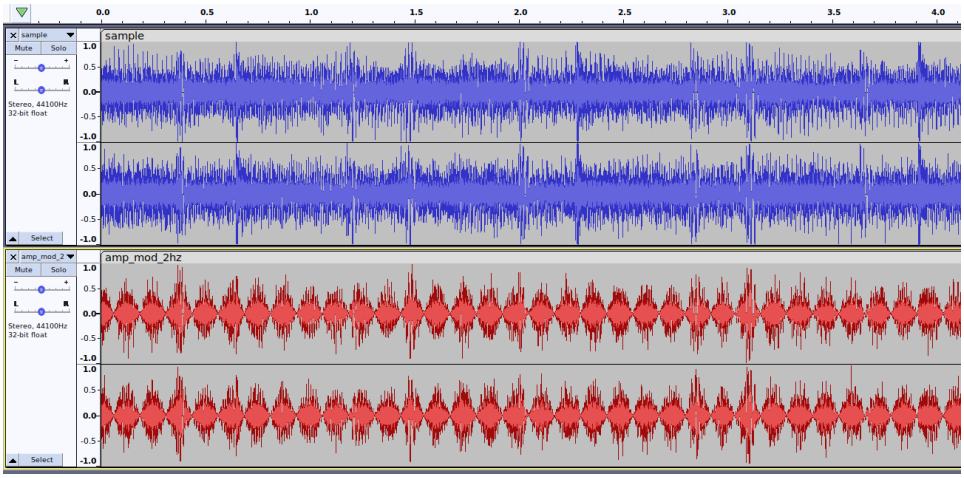

Figure 5.3: Comparing waveforms between sample.wav and amplitude modulation of 2Hz

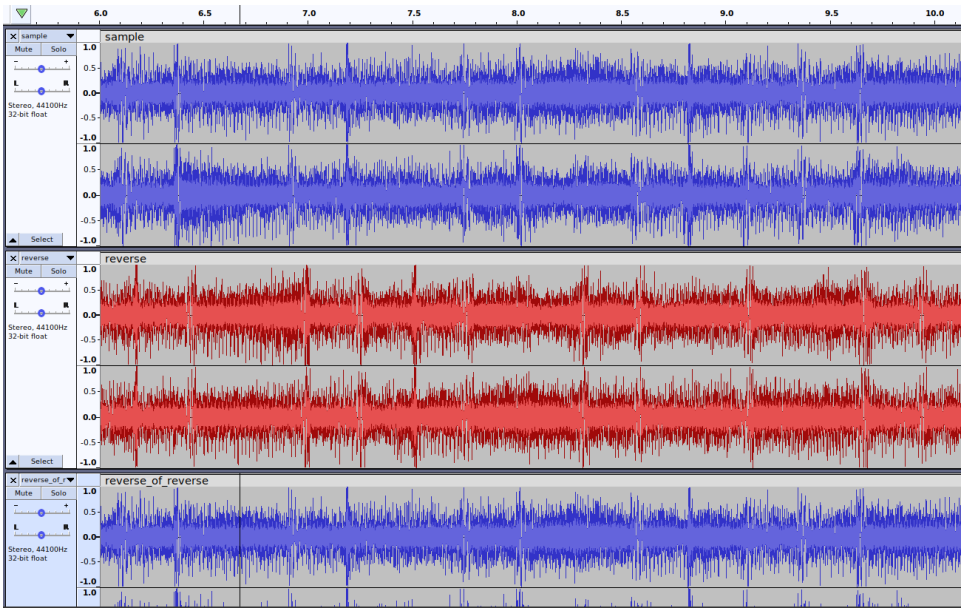

Figure 5.4: Comparing waveforms between sample.wav and reverse.wav and reverse_of_reverse.wav

On Figure 5.3 we can see that the audio tends to follow the 2Hz harmonic wave, as expect. On Figure 5.4 we can see that the reverse of reverse is exactly the same as the original sample, this situation clearly shows that the reverse function is working properly.

Chapter 6

Exercise 6

6.1 Code

In order to be more efficient in terms of memory and execution time, when an object of the type BitStream is created, according to the operation that's pretended to be done, it must be specified the mode "r" or "w", read or write, respectively. In reading mode, the file is read in binary mode, which means it can only read zeros and ones. Whenever you use the "readBit" or "readBits" functions, it's loaded 1 byte of the file content to a temporary array (working as a buffer), which is converted, by separating it, to 8 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. For instance, even if the file to be read has a great size, the execution time won't be affected, therefore it will have a linear performance has depicted in section 7.3.

```
1  std::vector<int> readBits(int n) {
2      ...
3
4      std::vector <int> outBits;
5      char byte;
6      int bitCount = 0;
7      while (bitCount < n) {
8          if (currentBitPos == 0) {
9              file.read(&byte, 1);
10             bitArray = byteToBitArray(byte);
11         }
12
13         outBits.push_back(bitArray[currentBitPos]);
14         currentBitPos++;
15         bitCount++;
16
17         if (currentBitPos == 8) {
18             currentBitPos = 0;
19         }
```

```
    }

    return outBits;
}

int readBit() {
    ...

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

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

    return bit;
}
```

Similarly to the reading process, we implemented an efficient way of writing files that would be linear to the size of the file. The writing process is done with the help of an integer values array (but only zeros and ones), initialized and populated with zeros. When the "writeBit" and "writeBits" functions are used, that array is filled with the information from the input file until it reaches its limit (8 units, each one representing a bit) and then converts those bits to a byte and writes it at the end of the output file.

In order to end the writing process, to ensure that the last bits are correctly written, the "close()" function writes the remaining bits on the array (even if it only contains 1 bit), populates the remaining space with zeros and converts them to a byte. This function also closes the file that is open, either for reading or writing, if necessary.

```
void writeBits(std::vector<int> bits) {
    ...

    int n = bits.size();
    int bitCount = 0;

    while (n > 0) {
        if (currentBitPos == 8) {
            char byte = bitArrayToByte(bitArray);
            file.write(&byte, 1);
            currentBitPos = 0;
        }

        if (currentBitPos == 0) {
            bitArray = std::vector<int>(8);
        }

        bitArray[currentBitPos] = bits[bitCount];
        currentBitPos++;
```

```
20          bitCount++;
21          n--;
22      }
23  }
24
25  void writeBit(int bit) {
26      ...
27
28      if (currentBitPos == 8) {
29          char byte = bitArrayToByte(bitArray);
30          file.write(&byte, 1);
31          currentBitPos = 0;
32      }
33
34      if (currentBitPos == 0) {
35          bitArray = std::vector<int>(8);
36      }
37
38      bitArray[currentBitPos] = bit;
39      currentBitPos++;
40  }
```

It's also important to be aware that this class works with the bits order in which we'd normally read a non binary number, for instance, the number 00110100 in a file will be written bit by bit as (0, 0, 1, 1, 0, 1, 0, 0) or populating an array like [0, 0, 1, 1, 0, 1, 0, 0] and using the function writeBits with that array as an argument.

Chapter 7

Exercise 7

7.1 Code

Exercise 7 is supposed to verify the correct development of the bitStream.h class, using two programs (encoder and decoder), with the help of a text file that contains only zeros and ones.

The encoder, firstly, reads the file (argument) and assumes that the text only contains zeros and ones and not other characters, like letters, spaces or newlines. Besides that, a BitStream object is created in writing mode, with the output file name mentioned as an argument. Secondly, for each value read, either zero or one, it casts it to an integer and writes them in the bits array, which contains all the bits of the file in the order they were read. After that, applying the "writeBits" function on that array and the class will write that information on a binary file. Finally, just to close the file, invoke the method "close" of the bitStream.

```
1 BitStream outputFile (outputFileName, "w") ;
2
3 vector<int> bits;
4 for (int i = 0; i < line.length(); i++){
5     bits.push_back(line[i] - '0');
6 }
7 outputFile.writeBits(bits);
8 outputFile.close();
```

The decoder receives as an argument the encoded file and that name for the output file. A BitStream object is created again but this team on reading mode. Using the method "getFileSize" we'll be able to verify its size in bytes, and find out how many bits are needed to read all of the file's content. Knowing this, the method "readBits" will be invoked with this size passed as an argument and the result will be written on a text file.

```
1 BitStream inputFile (argv [1], "r") ;
2 ofstream outputFile (argv [2], ios::out) ;
3 if (! outputFile) {
```

```
    cerr << "Error: could not open 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();
```

7.2 Usage

The usage of the encoder and decoder programs are done following this pattern:

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

Or

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

Where both input files are audio files (tested with .wav).

7.3 Results

To test encoder.cpp and decoder.cpp we started with a text file "lusiadas.txt" Table 7.1. Then, we used decoder.cpp to convert the lusiadas.txt to a file that contains the binary equivalent using the characters "1" and "0" Figure 7.1. After that we used encoder.cpp on the decoded file to get the initial text, seen on Table7.2.

As armas e os barões assinalados,
Que da ocidental praia Lusitana,
Por mares nunca de antes navegados,
Passaram ainda além da Taprobana,
Em perigos e guerras esforçados,
Mais do que prometia a força humana,
E entre gente remota edificaram
Novo Reino, que tanto sublimaram;

Table 7.1: lusiadas.txt

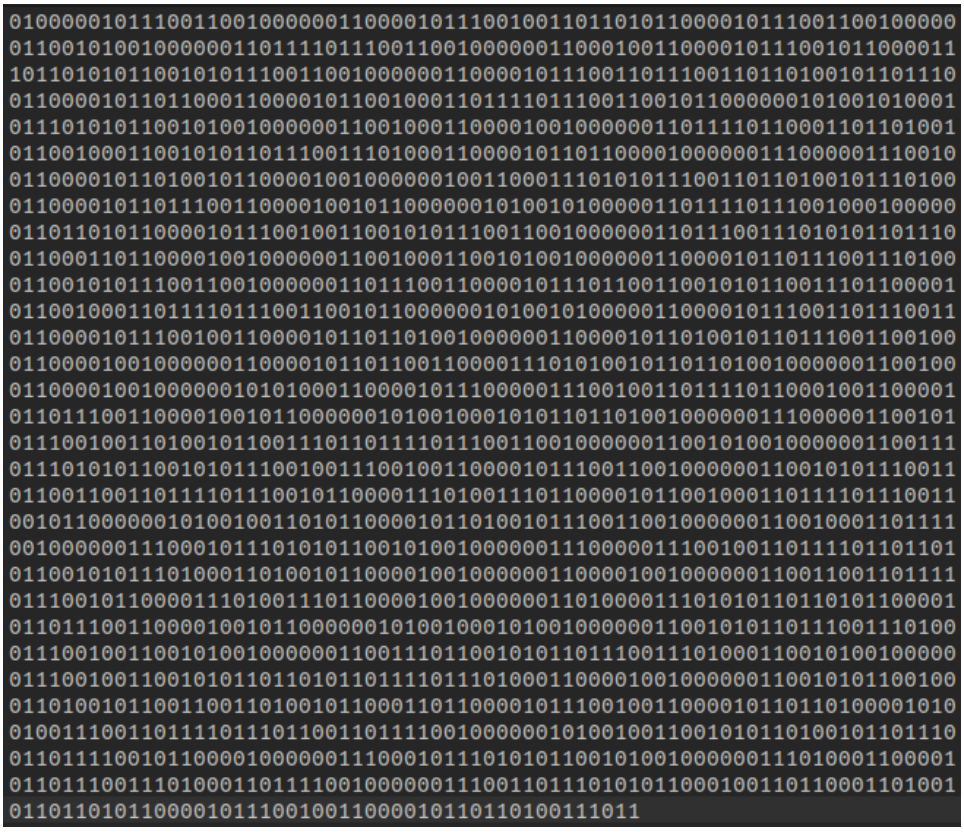

Figure 7.1: lusiadas_decoded.txt

Table 7.2: lusiadas_encoded.txt

To test the code's performance we created two files (generate_str.py and generate_bin_str.py) the first creates a random string with N given characters and the latter creates a random string (of ones and zeros) with N characters. We used these files to test encoder and decoder programs using string with 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536 and 131072 bytes.

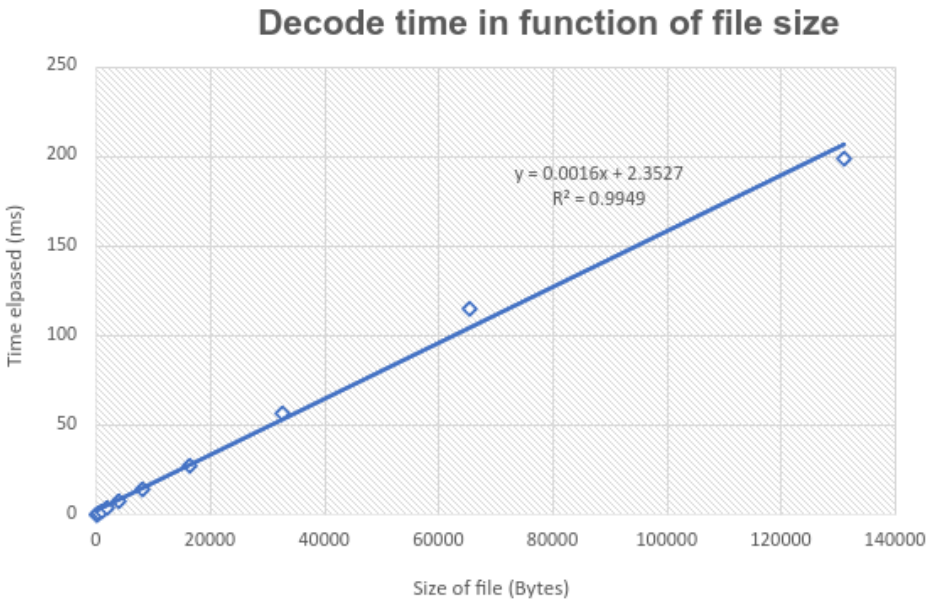

Figure 7.2: Graph for decoder performance

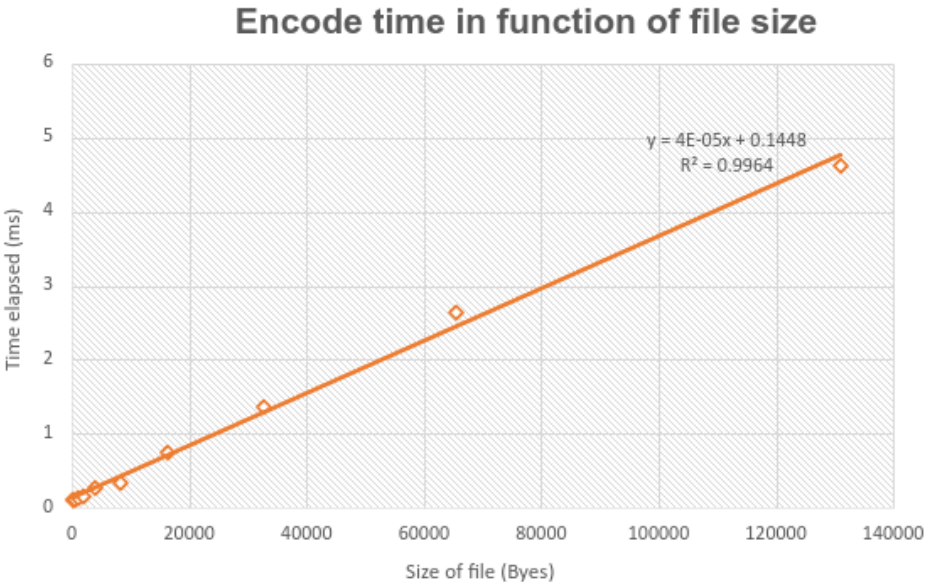

Figure 7.3: Graph for encoder performance

Viewing Figure 7.2 and Figure 7.3 we can see that the R Squared value is very close to one, which means that our data shows a correlation dependency. With that said, we can safely affirm that the decode/enconde processing time depends of the file size, and we see that this correlation is linear viewing the trending line and it's equation.

The generate files as well as the results we got are stored in the "extras" folder inside the project's root.

Chapter 8

Exercise 8

8.1 Code

In exercise 8, we've got two programs.

The lossy_encoder, that receives as arguments the audio file (.wav) that will be modified, the output binary file, the blocksize and the discarded units per block, will directly apply the Discrete Cosine Transform (DCT) to the samples of the input audio file, keeping just *blockSize-discarded_units_per_block*, which means it will only keep the most important frequencies of the audio file, in this case the lower ones.

```
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;
```

To decode the file, there's information that's important to keep besides the coefficients of the DCT (x_dct). With this in mind, we kept the number of blocks, the number of channels, the sampleRate of the original file and the number of frames. All of the previous are critical for the reconstruction of the new audio file. This values are kept as a header of the binary file, before

the storing process of the coefficients. For each of the kept values, either part of the header or the x_dct, there must occur a correct conversion to binary (having in mind the representation resolution: 16 bits for header values, except the number of frames, and 32 bits for the DCT coefficients). It's also relevant this coefficients are saved after a multiplication by 100, so that, in the decode process, the inverse DCT is made with a greater resolution (numbers with 2 decimal places) when compared to integer values.
The lossy_decoder needs, as arguments, the encoded binary file and a output audio file that's supposed to be created and written after the decode process.

The first step is to read the initial bits of the header in the binary file and convert them into the respective integers (blockSize, nBlocks, nChannels, sampleRate and nFrames). After that, before starting the decoding, it must be created an output file with the given name, number of channels and sample rate indicated.

```
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++;
        }
    }
}
```

All of the remaining bits of the file belong to the encoded x_dct, therefore they are read e and converted to their respective values (32 bits to integer and, after that, to doubles with 2 decimal places for a bigger resolution). Having the x_dct and the rest of the values of the header, it's possible to make the inverted DCT and recontruct the samples array in order to write it in 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);
```

8.2 Usage

The usage of the lossy_encoder and lossy_decoder programs are done following this pattern:

```
./lossy_encoder <input_file> <output_file> <blockSize>
    <discarded_units_per_block>
```

Where the input file is an audio file (tested with .wav), the output file is a binary file, the blockSize and discarded_units_per_block are both integers.

Or

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

Where the input file is a binary file and the output is an audio file (tested with .wav).

8.3 Results

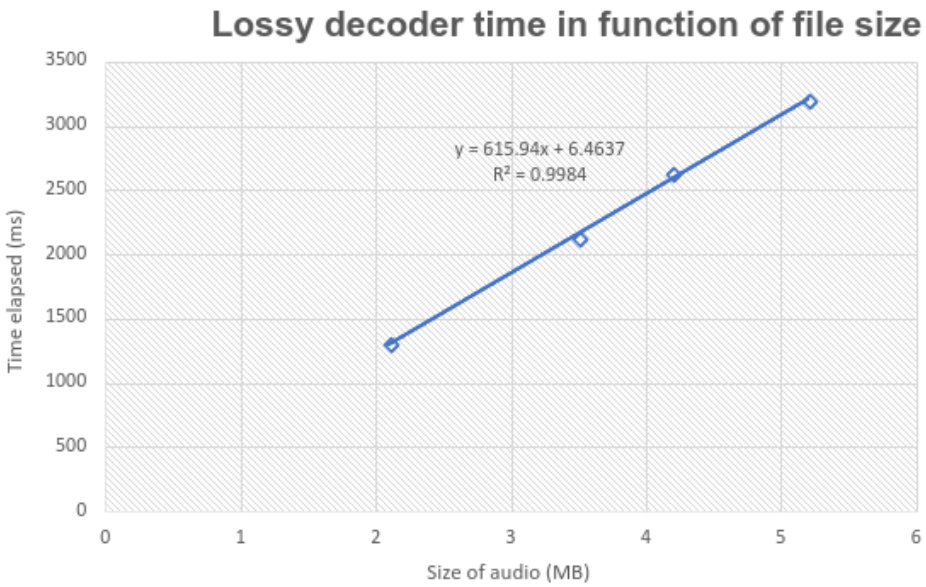

Figure 8.1: Graph for lossy encoder performance

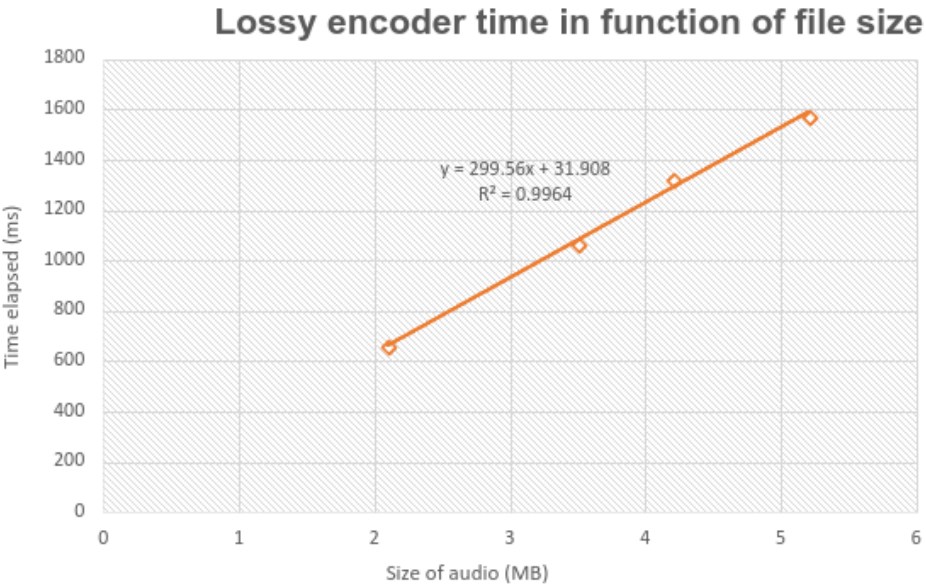

Figure 8.2: Graph for lossy encoder performance

Viewing Figure 8.1 and Figure 8.2 we can see that the R Squared value is very close to one, which means that our data shows a correlation dependency. With that said, we can safely affirm that the decode/enconde processing time depends of the file size, and we see that this correlation is linear viewing the trending line and it's equation. This results are expected knowing that lossy_decoder and lossy_encoder use the decode and encode functions respectively, when proven on Section 7.3 that decode and encode depend linearly from the file size, we can expect that lossy_decoder and lossy_encoder depend too.