IC: Project 1

Universidade de Aveiro
DETI

Daniel Silva, (102537) / João Andrade, (103361)
Tiago Santos, (89356)

Contents

1 Introduction

The goal of this project is to learn about signal modulation through implementing, and testing, a list of programs to process audio and binary files. To test these programs and report the results we used the provided samples.

2 Part I

2.1 WAVHist Class

We altered the WAVHist class to support the histograms of the MID channel or the SIDE channel.

General use:

```
../wav_hist <input> <channel> [bin size]
```

where the the channel can be "left", "right", "mid" or "side", and bin size is the exponent of 2 of the desired bin size (for example, if we use 3 as an argument, each bin will hold 8 values).

2.1.1 Channel Histograms

For the histograms we used a vector of maps (each map corresponding to a channel) that count the occurrence of each sample value.

2.1.2 Coarser Bins

The coarser bins are used to pack some of the sample values together, each bin packing several samples according to the specified value.

2.1.3 Results

After experimenting with different bin sizes over all the samples we confirmed our hypothesis that the histogram would hold higher counts for frequencies closer to 0 (i.e. the dimension of the count for each sample is inversely proportional to its distance from 0).

However, there is an instance of a different scenario, with sample07, where there are peaks for both the highest and lowest frequencies, which are most likely explained through the nature of its sound/type of music.

The histogram of sample01 can be seen in Figure 1 and the histogram of sample07 can be seen in Figure 2.

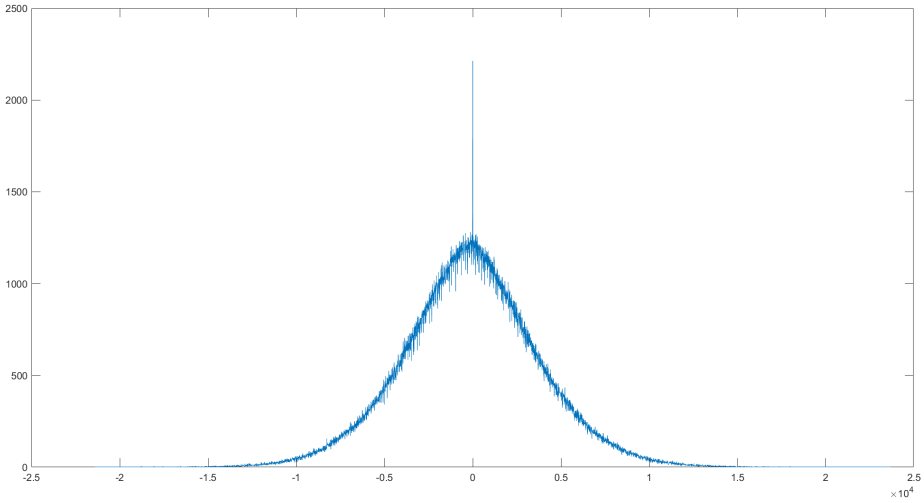

Figure 1: Left Channel Histogram, sample1, bin size = 3

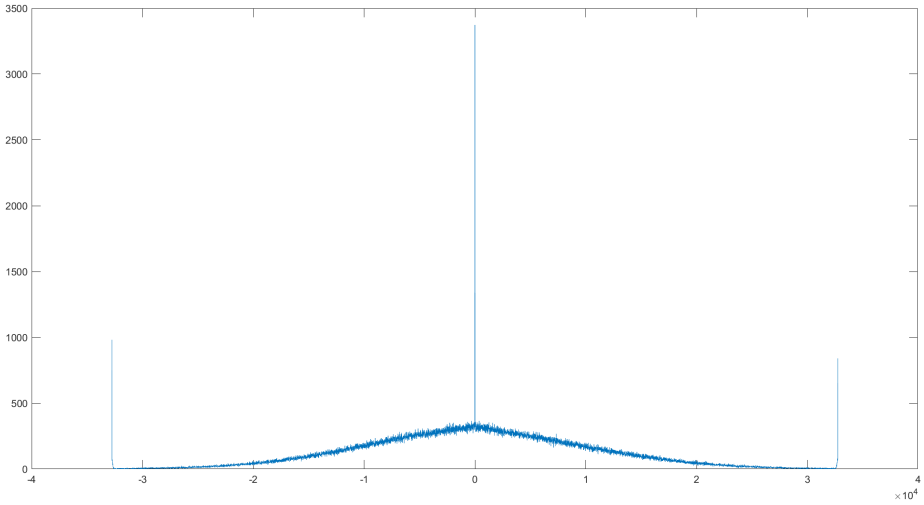

Figure 2: Left Channel Histogram, sample7, bin size = 3

2.2 Error calculations

We use the following expressions to calculate Signal-to-noise-ratio (SNR) and the L2 and L∞ Norms:

$$r(n) = x(n) - \tilde{x}(n), \text{ with } \tilde{x} \text{ being the quantified sample}$$
$$\varepsilon_x = \sum_n |x(n)|^2,\ \varepsilon_r = \sum_n |r(n)|^2$$
$$SNR = 10 \times log_{10} \frac{\varepsilon_x}{\varepsilon_r} \text{dB}$$
$$\varepsilon_{max} = \frac{\Delta}{2}$$
$$|x| = \sqrt{\sum_{k=1}^{n} |x_k|^2}$$
$$||x||_\infty = nsup|x_n|$$

For that we implemented the following functions:

```
double calculateL2Norm(const std::vector<double>& original, const
    std::vector<double>& modified){
    double sum = 0.0;
    size_t numSamples = original.size();

    for (size_t i = 0; i < numSamples; ++i) {
        double diff = original[i] - modified[i];
        sum += diff * diff;
    }

    return std::sqrt(sum / numSamples);
}
double calculateLInfNorm(const std::vector<double>& original, const
     std::vector<double>& modified){
    double maxAbsError = 0.0;
    size_t numSamples = original.size();

    for (size_t i = 0; i < numSamples; ++i) {
        double absError = std::abs(original[i] - modified[i]);
        if (absError > maxAbsError) {
            maxAbsError = absError;
        }
    }

    return maxAbsError;
}
double calculateSNR(const std::vector<double>& original, const std
    ::vector<double>& modified){
    double signalPower = 0.0;
    double noisePower = 0.0;
    size_t numSamples = original.size();

    for (size_t i = 0; i < numSamples; ++i) {
        signalPower += original[i] * original[i];
        noisePower += (original[i] - modified[i]) * (original[i] -
    modified[i]);
    }

    return 10 * std::log10(signalPower / noisePower);
```

```
}
```

2.3 Quantization

Quantization is a way of representing a signal while limiting the number of bits that represent the sample values. For this we truncate the binary value of the samples, therefore achieving the pretended outcome.

2.3.1 Results

We concluded that we can notice noise if we cut away, on average, 8 bits. Some samples would only have noticeable noise by cutting 10 bits (sample0; sample1; sample7), while others could have noticeable noise while only cutting 6 bits or 7 bits (sample6; sample4; sample6).

We believe that, with sample 6, we can only cut up to 5 bits because there's lots of silence in the sample. We're led to this conclusion because of the amount of bits we can freely cut on the samples above (0, 1, and 7) which don't have many silences.

2.4 Effects

For this section, we implemented 3 effects: Echo (Single and Multiple), Delay, and Amplitude Modulation. To make this happen, we used certain calculations that will be described in the following subsections.

2.4.1 Echo and Delay

Echo and Delay (effect 0) were implemented the same way, the only difference being the arguments that the user inputs into the system. For this effect, we used:

$$Volume = Sound * P^R$$

Sound = current volume %, **P** = proportion, **R** = repetition.

We make the **proportion**(P) to the power of the **repetitions**(R) times the volume percentage. With this, we can create a delay and repeat the sounds by making **proportion** equal to 100 (default value), so it repeats with the same **volume** and uses Δ for the space between them and even with repetitions. It is also important to note that Δ is determined by the argument input by the user in seconds and then based on the file's sample rate it determines the number of frames to delay equivalent to the seconds.

For the Echo, we change the proportion to a number in the]0,100[interval so that the percentage of sound is reduced by each repetition, which can be changed, giving the option for single or multiple echoes.

Code for Delay and Echo:

```
case 0: // repetition
    {
      int repetitions = stoi(parameters[0]);
      int delta = stod(parameters[1])*samplerate*channels;
      double proportion = stod(parameters[2])/100;
      created.resize(original.size() + repetitions*delta);
      for (int i = 0; i < int(original.size()); i++) {
        for (int j = 0; j <= repetitions; j++) {
          created[i + j*delta] += short(original[i]*pow(
  proportion, j));
        }
      }
      created.resize(original.size());
      break;
    }
```

2.4.2 Amplitude Modulation

Amplitude Modulation was achieved by multiplying the percentage of sound with the proportion specified by the user. It then creates an audio file by replicating the original with the new volume proportion.

Code for Amplitude Modulation:

```
  case 1: // multiply amplitude
    {
      double proportion = stod(parameters[0])/100;
      created.resize(original.size());
      for (int i = 0; i < int(original.size()); i++) {
        created[i] = short(original[i]*proportion);
      }
      break;
    }
```

2.4.3 Results

General Usage:

```
../wav_effects inputFile.wav outputFile.wav <effect> <arguments>
```

Echo and Delay usage:

```
../wav_effects inputFile.wav outputFile.wav 0 <repetitions> <
  delta> <proportion>
In case of Delay, proportion = 100, otherwise [0,100]
```

Amplitude Modulation usage:

```
../wav_effects inputFile.wav outputFile.wav 1 <proportion>
```

We processed to test the effects in this order: delay, echo, and amplitude modulation. We then proceed to compare waveforms between the original and the modified file.

The delay was tested with only one repetition and a 2-second delay using sample01.wav, echo with 4 repetitions, a 1-second delay, and 50% proportion using sample05.wav. Finally, for the amplitude modulation, we used 40% proportion in sample02.wav. All of this was tested with 44100 Hz.

Looking at image 3 we can see that after the 2-second delay, the audio volume in effect_delay.wav increases exponentially representing the delay effect, therefore the beginning of the sample.

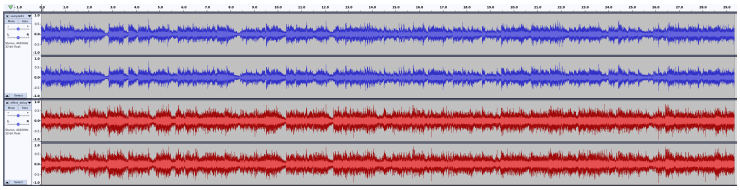

Figure 3: Waveforms comparison between sample01 and effect_delay

In image 4 the echo effect can be seen to occur after 1 second, and can also be easily heard in its audio file: effect_echo.wav.

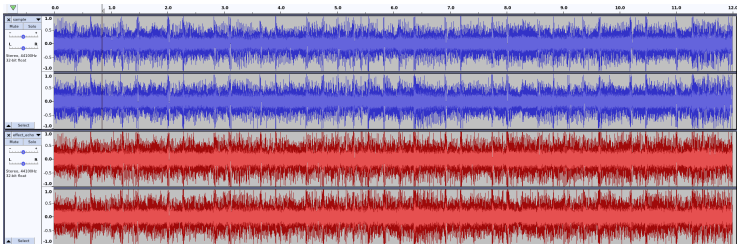

Figure 4: Waveforms comparison between sample05 and effect_echo

Finally by analyzing image 5 we can easily see the small decrease in the audio volume, more precisely 50% of its original size.

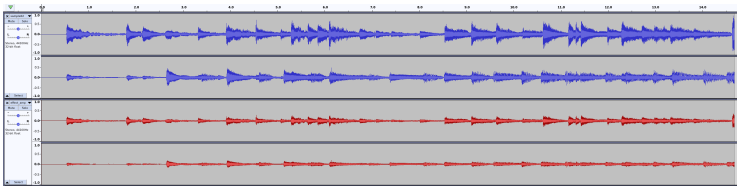

Figure 5: Waveforms comparison between sample02 and effect_amp(amplitude modulation)

It is important to note that all of these modified audio files ended up with the expected results, only having a small difference in quality in audio in effect_delay.wav.

3 Part II

3.1 BitStream

To write and read bits from a file, we created the BitStream class. This class provides methods to read several bits from 1 to 64 and to write an arbitrary number of bits. It can also read some bytes and return them as a string, and write a string to a file. This class was used in the exercises that followed.

When reading several bits that are not a multiple of 8, the program will discard some read-only bits to return the number of bits that were asked. This creates a problem where the discarded bits are lost. Although we could fix this issue, it didn't prove to be a problem in the exercises that were used in this class.

3.2 Encoder / Decoder

To test the functions in Bitstream we created two programs, an encoder and a decoder.

The encoder is responsible for reading a file, assumed only to contain ones and zeros and can transform it into a binary file. To do so, the program uses the "write" functions from the Bitstream class. The decoder does the opposite, by using the binary file and returning it to its original form (ones and zeros).

3.2.1 Results

Encoder usage:

```
../encoder <inputFile> <outputFile>
```

Decoder usage:

```
../decoder <inputFile> <outputFile>
```

To test these programs we used message.txt and determined the times it takes to process it. After several tests on encoding a file, we reached an average of 2,8601ms and for the decoding 4,8314ms.

4 Part III

4.1 Lossy Codec

A lossy audio codec was implemented, which is composed of both a lossy encoder and a lossy decoder, as the name suggests. The encoder takes an audio file as input and produces a binary file, smaller than the original. Some information is lost during this process, and the newly created file is not readable by any audio player. The decoder takes this binary file and turns it into an audio file with the same size as the original, but with reduced bit depth.

The encoder works by employing the DCT lossy compression technique to represent the original audio information in sets of blocks. Some specifics of this process can be modified with parameters when executing the program, these being the block size and the fraction of low frequencies to keep. All the values resulting from the DCT are truncated and lose their decimal digits. The encoder calculates, based on the minimum and maximum value resulting from the DCT process, the minimum amount of bits necessary to represent every value. This information will be written in the header of the binary file, along with the block size, number of channels, number of frames, number of blocks, and sample rate. All of this information is necessary and sufficient to turn the compressed file into an audio file that resembles the original. After writing the header, the values resulting from the DCT calculations will be written, all of them represented with the same number of bits.

The decoder works by reversing the logic used in the encoder. It starts by reading the header and storing it in the appropriate variables for later use. Then, the DCT values are read and expanded to 32 bits such that they can be stored in a standard C integer variable. After inverse DCT calculations are made with these values, they are written to a file with the same configuration as the original (number of channels and sample rate).

The DCT and its inverse calculations were written by adapting the wav_dct program that was provided with the project template.

The resulting file after the process of encoding is never bigger than the original, but the amount of compression varies between files and parameters. We tested different values of block size and compared the sizes of the resulting binary files.

4.1.1 Results

We weren't able to find an accurate pattern when analyzing the results for some of the samples.

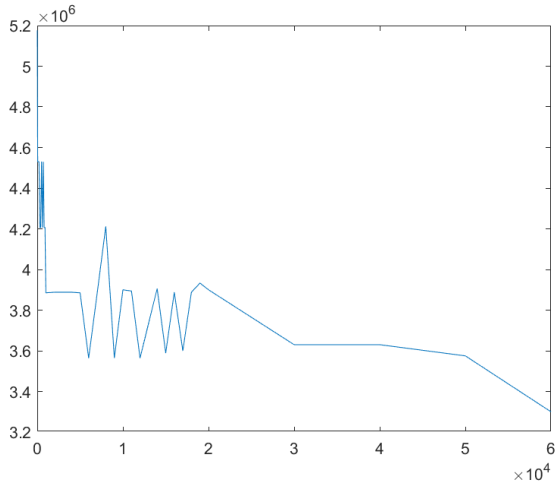

Figure 6: Size in bytes of output file with lossy_encoder for sample01.wav at different bs values

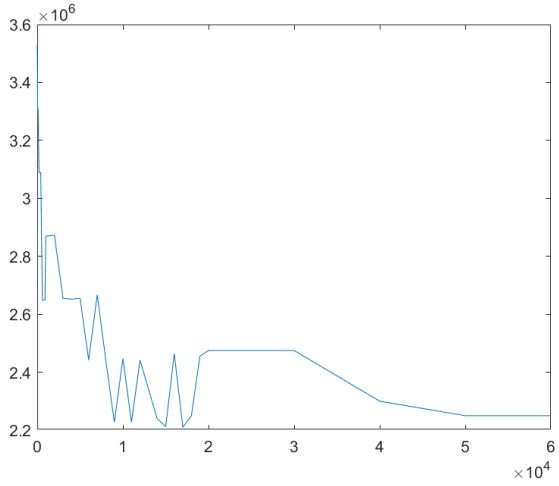

Figure 7: Output file size with lossy_encoder for sample03 at different bs values

5 Conclusion

We managed to test our methods with the various samples and believe that the results are in accordance with what we expected a priori, based on signal processing and file compression theory. We have deepened our understanding of both the subject material and the C++ language.

Some other data collected during this project can be found on our GitHub Repository: `https://github.com/jrpfaria/IC`

6 Group Contribution

We consider this project's development to have been shared equally amongst all members of the group.