Universidade de Aveiro

**Information and Coding**

**lab work nº1**

****

Diogo Martins (108548) | Luís Sousa (108583) | Tomás Viana (108445)

Departamento de Eletrónica, Telecomunicações e Informática

19/10/2025

Index

[Summary 3](#_Toc211522783)

[Part 1 4](#_Toc211522784)

[Exercise 1 4](#_Toc211522785)

[Exercise 2 4](#_Toc211522786)

[Exercise 3 4](#_Toc211522787)

[Exercise 4 5](#_Toc211522788)

[Part 2 5](#_Toc211522789)

[Exercise 6 5](#_Toc211522790)

[Part 3 6](#_Toc211522791)

[Exercise 7 6](#_Toc211522792)

# Summary

The project consists of multiple C++ files with the purpose of exploring sound file’s properties and transforming them. The original repository came with a few pre-made programs such as **wav\_cp.cpp** which creates a copy of a given sound file, **wav\_hist.cpp** that outputs a basic histogram of the sound file, a **wav\_dct.cpp** which represents an example of use of the Discrete Cosine Transform (DCT) and **bit\_stream.cpp** that read and writes bits from/to a file. On top of that, there’s also some examples of sound files, libraries and makefiles to help development.

# Part 1

## Exercise 1

For this exercise, the idea was to modify **wav\_hist.h** and **wav\_hist.cpp** so that it can also output the histogram of the average of the channels (channel MID) and the average difference of the channels (channel SIDE).

To output channel MID, update() in **wav\_hist.h**was updated to calculate the expression (L+R)/2, which represents the addition between channels divided by two, returning the average of the channels. The same was done for channel SIDE with the expression (L-R)/2 returning the average difference. The necessary dump functions for each new channel were also added similar to the functions of channels 0 and 1.

The file **wav\_hist.cpp** was updated to detect the new channels and to properly dump them. Additionally, the histogram bins meant for each different sample were replaced with coarser bins, that gather together 2, 4, 8, . . . 2K values.

The output is as follows:

* input
* output

## Exercise 2

For this exercise, we implemented a program named **wav\_quant.cpp** to reduce the number of bits used to represent a given audio sample, in other words, to perform uniform scalar quantization. This means that the program receives an audio sample and creates a worse sounding copy, since the amplitude will be stored with less precision.

After retrieving the input of the number of bits to keep, the program calculates the bits to remove (assuming we’re working with 16-bit audio). Then the creation of the new audio sample proceeds as usual, except that in each frame the sample is shifted according to the bits to remove, discarding the least significant ones, resulting in the less precise audio.

The output is as follows:

* input
* output

## Exercise 3

The third exercise consists of creating a program named **wav\_cmp.cpp** with the purpose of calculating and outputting, for each channel and their average, the average mean squared error (L2), the maximum per sample absolute error (L∞) and the signal-to-noise ratio (SNR).

The three expressions are calculated comparing an audio sample to its modified copy (if both samples are the same, the expressions will output 0 and if the audio samples are completely different the results will lose their meaning). Inside two “for” loops, the difference between the samples for each frame is acquired and used to calculate the expressions:

* MSE = (1/N) \* Σ[n=1 to N] of (x[n] - x̂[n])²;
* MSA = max(|x[n] - x̂[n]|), for 1 ≤ n ≤ N;
* SNR = 10\*log10((Σ[n=1 to N] of x[n]^2) / (Σ[n=1 to N] of (x[n]-x̂[n])^2));

The output is as follows:

* input
* output

## Exercise 4

For this exercise, we created a program named **wav\_effects.cpp** that produces and applies simple audio effects, such as echoes, amplitude modulation and time-varying delays. The effect applied will be according to the input, to which there are four choices:

- Echo creates a single echo in the audio sample, according to the delay and gain given. The program acquires delay samples and, for each sample, adds more artificial delay.

- Multi\_echo is similar to the previous one except it applies multiple echoes to the sample. Same process as before, but the delay is added again in different intervals of time.

- Amplitude\_mod varies the amplitude (loudness) of the sample. The program calculates the modulation factor determined by the quantity of the amplitude changes and, for each sample, multiplies the original sample by that factor.

- Varying\_delay takes the audio sample and mixes it with a delayed version of itself. Using the given max delay and sample rate it computes the time-varying delay for each frame and adds the mix to the original audio, in which 70% is the original sample and 30% is the delayed sample.

The output is as follows:

* input
* output

# Part 2

## Exercise 6

For this exercise, we implemented a codec consisting of two components: an encoder and a decoder (**wav\_quant\_enc** and **wav\_quant\_dec**) that can pack the result of the reduction of the number of bits by uniform quantization, essentially decreasing the overall amplitude of the signal.

The encoder converts a WAV file into an encoded bit-packed audio, which means it decreases the number of bits according to the bits to keep. Through importing functions from **bit\_stream** the encoder packs the input audio file bit by bit, to then turn them into full bytes to create the output data. Finally, for each frame the bits are reduced from 16 to 8 (for the specific audio files used in this project) by iteratively discarding the least significant bit, creating the less precise output file.

The decoder

The output is as follows:

* input
* output

# Part 3

## Exercise 7

The final exercise consisted of implementing a lossy codec only for mono audio files based on the Discrete Cosine Transform (DCT). This file also uses **bit\_stream** to compress a sound file bit by bit, however it creates and outputs the data in blocks (chunks of the waveform) within a binary file representing the compressed sound. While **wav\_quant\_enc** manipulates audio on the amplitude domain, this codec works on the frequency domain and through DCT decreases its quality.

Similarly to **wav\_quant\_enc**, after checking the input the encoder starts packing the audio file data bit by bit using the **bit\_stream** functions, this time also operating with the given block size. After this the process of creating the blocks begins, where for each block the dct() function computes the block according to the formula:

* DCT = α(k) × Σ[n=0 to N−1] of [x(n) × cos((π/N) × (n+0.5) × k)]

Each DCT coefficient gets rounded up to an integer and gets discarded if a highest-frequency coefficient, to simplify values, and get written to the output bitstream.

The output is as follows:

* input
* output