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
Part 1 4
Exercise 1 4
Exercise 2 4
Exercise 3 5
Exercise 4 6
Part 2 7
Exercise 6 7
Part 3 7
Exercise 7 7

Summary

The project consists of multiple C++ files with the purpose of exploring audio properties and how to read and write various sound file formats. 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 audio files, headers and makefiles to complement the main ones.

Part 1

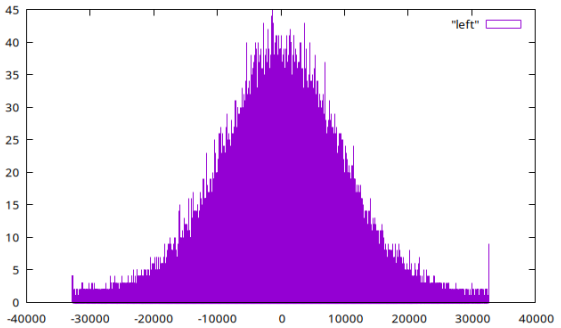
Exercise 1

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, . . . 2^K values.

The output is as follows:



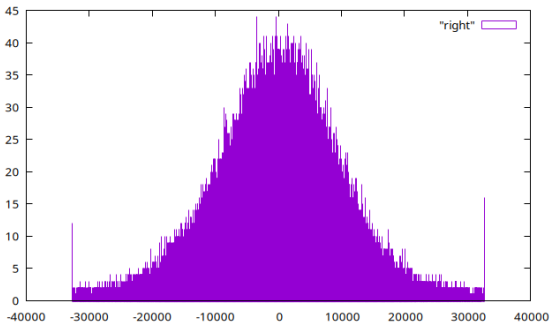
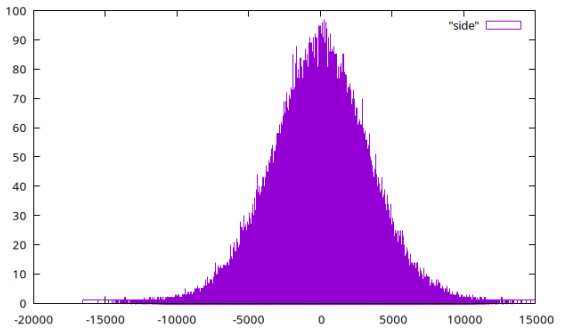


Figure 1 & 2 - Left and Right channels histogram.



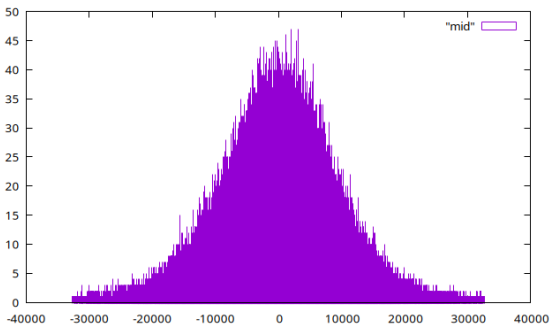


Figure 3 & 4 - SIDE and MID channels histogram.

The results were exactly what we expected, the MID channel has the average of both channels, so it makes sense that the results are similar to right and left, and the SIDE channel shows higher spikes around 0, which makes sense since it's the difference between similar channels.

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:

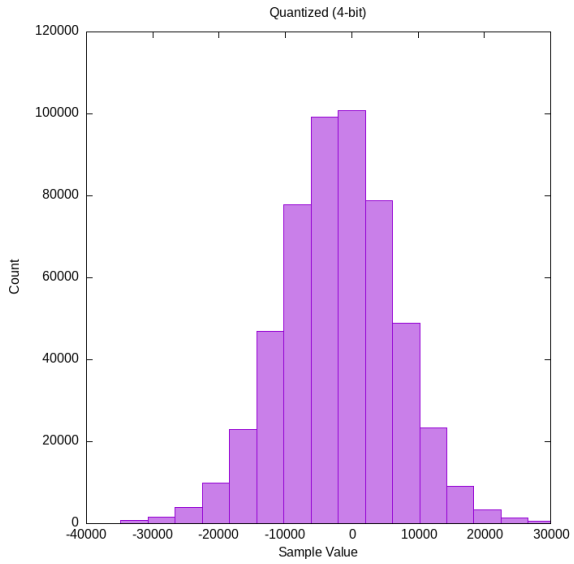


Figure 5 – 4-bit quantized audio histogram.

The histogram shows way less detail compared to the original histogram, which was expected due since this audio only has 4 bits, which equates to 16 bars ($2^4 = 16$).

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 (L^2), the maximum per sample absolute error (L^∞) and the signal-to-noise ratio (SNR). MSE measures the difference between two audio signals, MAE measures the largest absolute difference also between two audio signals and SNR measures of how much desired signal (the clean or original audio) exists relative to unwanted noise.

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;`
- `MAE = 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:

```
Channel 0 MSE: 8.96434e+07
Channel 1 MSE: 8.92572e+07
Average MSE: 8.94503e+07

Channel 0 Max Abs Error (L∞): 16383
Channel 1 Max Abs Error (L∞): 16383
Average Max Abs Error (L∞): 16383

Channel 0 SNR: -0.840758 dB
Channel 1 SNR: -0.801379 dB
Average SNR: -0.821067 dB
```

```
Channel 0 MSE: 3.85857e+08
Channel 1 MSE: 3.8272e+08
Average MSE: 3.84288e+08

Channel 0 Max Abs Error (L∞): 32767
Channel 1 Max Abs Error (L∞): 32767
Average Max Abs Error (L∞): 32767

Channel 0 SNR: -7.17984 dB
Channel 1 SNR: -7.12376 dB
Average SNR: -7.15182 dB
```

Figure 6 & 7 – MSE, MAE & SNR for two different audio samples.

The results were what we expected according to the samples we used, the SNR is lower than zero since there's less noise than original audio, and the MSE and MAE are both higher when the audio samples are less similar to each other (or have more changes done to them).

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: y(n) = x(n) + gain * x(n - delay)`

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: y(n) = x(n) + gain^k * x(n - k*delay)`

It 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: y(n) = x(n) * (1 + depth * sin(2 * pi * f * t))`

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: y(n) = x(n) + 0.3 * x(n - d(n)) [where d(n) varies with LFO]`

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 results were the audio samples of each choice, which came out as predicted.

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 the WAV file into an encoded bit-packed representation, while keeping all the necessary information of the original file. It writes the data into a binary file using the **`bit_stream`** file and for each frame the bits get reduced from 16 to 4 (best example to showcase this exercise) by iteratively discarding the least significant bits, creating the less precise audio file.

The decoder recovers the encoded file into a new 16-bit WAV file. To create the new output, it needs to get the frames, channels and such data to accurately build it, so it once again uses **`bit_stream`** to get this information, this time to read the binary content. After checking if the data is valid the file sets up the samples to use them in a loop, where each sample gets written.

The output is as follows:

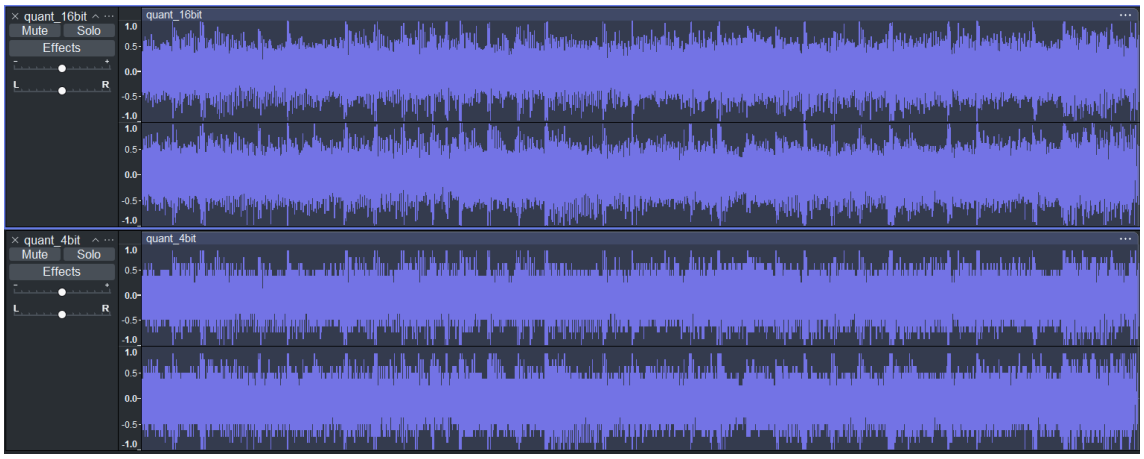


Figure 8 – Spectrogram of a 16-bit and 4-bit audio.

The results showed what we expected, a bin file was created every time the encoder ran with the audio sample, and when the decoder used that file a copy of the original audio was created, but with the amplitude slightly limited.

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, but 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`**, the encoder starts by 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 file 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 it's a high-frequency coefficient, to simplify values, and gets written to the bitstream output. Only the lower frequencies get saved to the output, as the higher ones get discarded.

The decoder is once again similar to **`wav_quant_dec`** but centered on rebuilding the compressed WAV file without noticeably degrading the audio too much. Besides reading the usual frame and block information, the file also uses **`bit_stream`** to recover the quantized DCT coefficient, essential to maintain the compression. Afterwards each coefficient gets dequantized and each block goes through inverse DCT (inverse of previous formula) to recover the necessary frequency to then write the output audio.

The results came out as expected, the encoder successfully created the bin file representing the audio file and the decoder created the new audio sample with a smaller size and without significant changes in audio quality.