Universidade de Aveiro(UA)

DETI

Information and Coding

Project 1

Eduardo Lopes Fernandes,
João Afonso Ferreira,
Rafael Luís Curado

Teacher: António Neves
Group: P1 - G1

October 2023

Contents

List of Figures

1 Exercise 1 - WAVHist Class Development

1.1 Observations

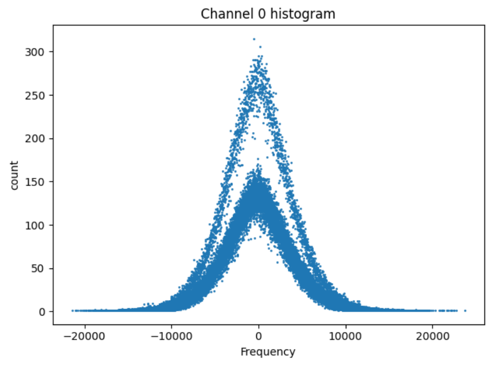
Goal: Change the *WAVHist* class in order to provide the histogram of the average of the channels (the mono version, i.e., $(L + R)/2$, also known as the MID channel) and the difference of the channels (i.e., $(L - R)/2$, also known as the SIDE channel), when the audio is stereo (i.e., when it contains two channels), and also to provide *Coarser bins*, i.e., instead of having a histogram bin for each different sample value, have bins that gather together 2, 4, 8, . . ., 2^k values.

As the exercise description suggests, we expanded the `wav_hist` class. The only alterations made to the .cpp file were to eliminate the need to pass a channel as an argument and to call other `wav_hist` class functions to store the values in four separate files, one for each type of output (channel 0, channel 1, mono, and difference).

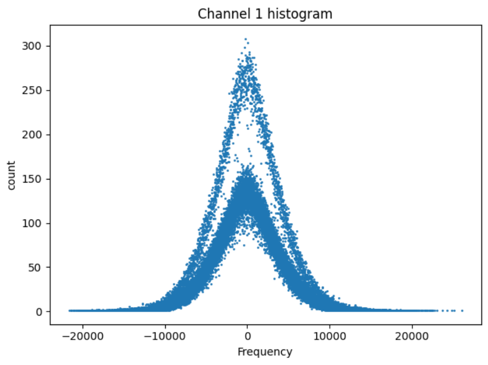
Both the mono and difference histograms are created only when there are two channels. We created a map for each of them and followed the logic of the algorithm that was already applied to determine the channel histograms.

1.2 Results

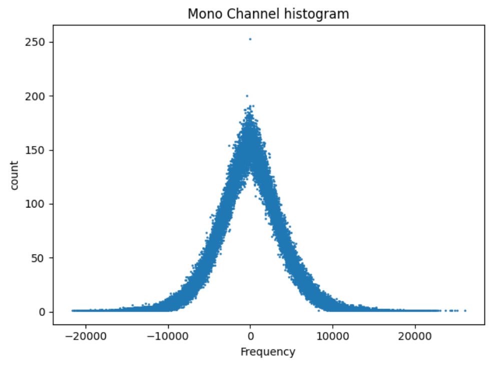
The best way to visualize the results is with graphs, for that it was used a python script with matplotlib [1]. We used sample01.wav available on the elearning and plot a points graph.



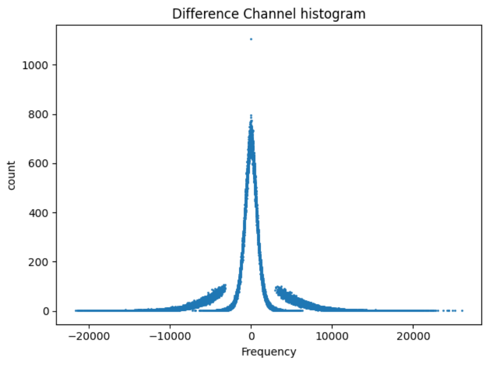
(a) Channel 0 histogram



(b) Channel 1 histogram



(c) Mono histogram

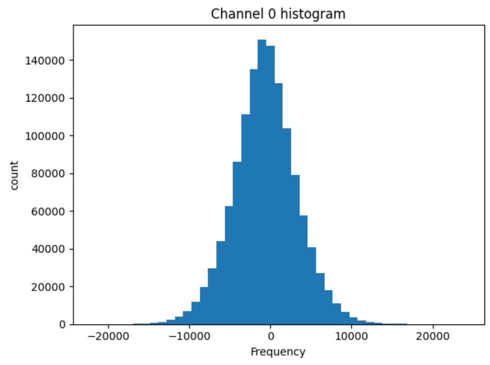


(d) Diff histogram

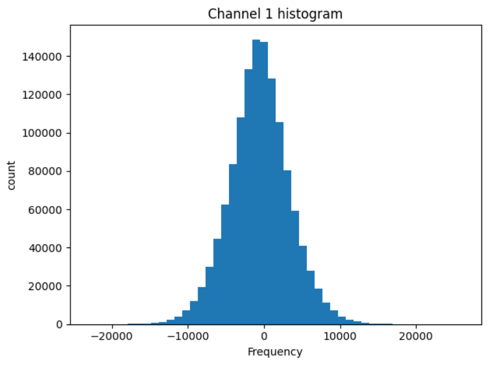
Both channel 0 and channel 1 exhibit unexpected results, as there is a noticeable separation on the graph in both channels. This indicates that certain frequency values occur more frequently than others, possibly due to the way the sample was recorded, likely involving some constant noise that shouldn't be present.

Interestingly, this noise disappears when we combine both channels, resulting in the mono channel. As expected, the graph for the mono channel aligns with our predictions, with lower frequencies being more outstanding than high frequencies.

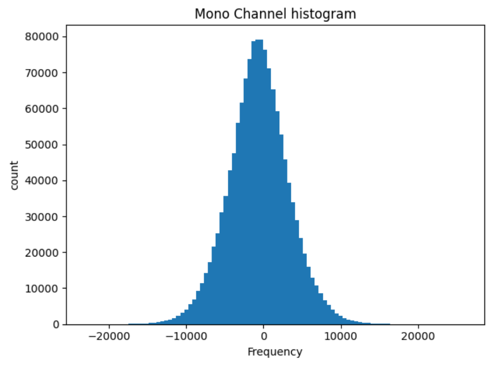
The difference histogram is also intriguing. As anticipated, when both channels are fairly equal, the graphic appears quite sharp. However, there are some points beyond 2 kHz that exhibit higher values than expected. It's possible that these points correspond to the anomalies observed in channel 0 and channel 1.



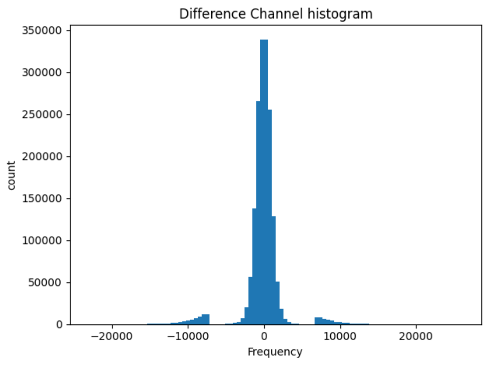
(a) Channel 0 histogram



(b) Channel 1 histogram



(c) Mono histogram



(d) Diff histogram

The primary objective of the second step in the initial exercise was to organize the frequency values into distinct groups represented by bars. In the provided illustration, this was achieved by dividing the data into 2^6 groups, totaling 64 groups. This was possible by leveraging a class developed in the third exercise to quantize the samples. Remarkably, with this quantization as the sole modification, the program began to yield highly satisfactory results.

2 Exercise 2 - Channels Comparison

2.1 Observations

Goal: Compare two WAV audio files computing various quality metrics to assess the similarity between the two files.

Our program reads audio samples from both files in frames and calculates the Mean Squared Error (also known as L2 norm), Maximum Absolute Error (also known as L∞ norm), and Signal-to-Noise Ratio (SNR) to evaluate the difference between the two audio signals.

Mean Squared Error: Obtained by summing the squared differences between corresponding samples and dividing by the total number of samples.
Maximum Absolute Error: Given by the maximum absolute difference between corresponding samples.
Signal-to-Noise Ratio: Calculated using the formula:
SNR = 10 $\times$ log(signal / noise).

Usage:

```
../sndfile-example-bin/wav_cmp <infile.wav> <outfile.wav>
```

2.2 Results

The Mean Squared Error is a measure of the overall difference in signal intensity between two audio files, Maximum Absolute Error is the maximum difference between corresponding samples and the Signal-to-Noise Ratio (SNR) quantifies the quality of the signal compared to the noise, expressed in decibels.

We created a program (wav_cmp_extra) that, given a wav file (we used sample01.wav referenced before), creates 15 other quantized versions of it, from 1 to 16 bits of resolution and stores the data (mean squared error, maxium absolute error and SNR) in 3 diferent text files that were used by a python script to provide the following graphs:

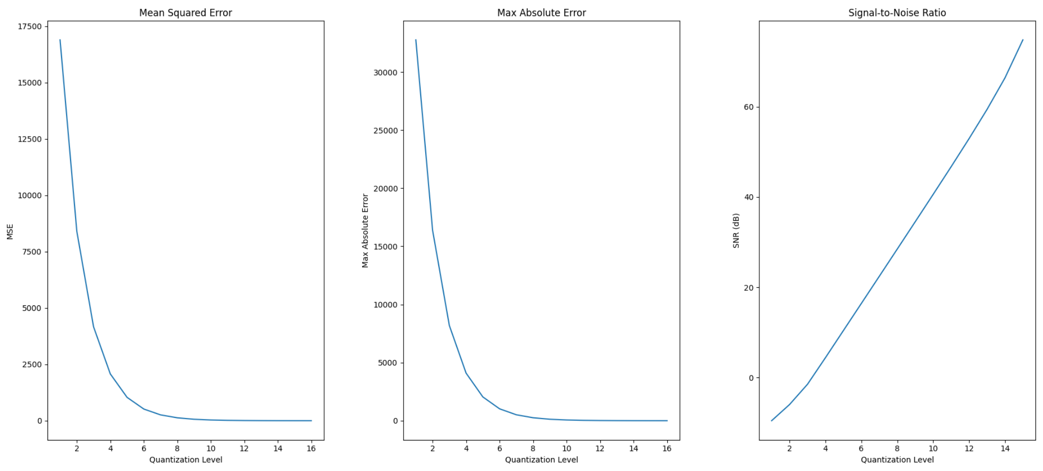


Figure 3: Comparison between a wav file and its various quantized versions

After analysing these graphs, we can conclude that the smaller the quantization, the larger the errors and the smaller the SNR. Also, when the quantization is very small (bellow 3 bits) the SNR is negative, which is what we expected considering that we removed at least 13 bits.

3 Exercise 3 - Reduce bit Number

3.1 Observations

Goal: Create a C++ class called 'Wav_Quant' along with an accompanying program. This class and program should be designed to decrease the bit depth used for encoding each audio sample, effectively implementing uniform scalar quantization on audio data.

Our class is designed for working with WAV audio files and performing quantization reduction. It includes two static member functions:

```
reduce_quantization(short* samples, int nSamples, int newBits)
```

This function takes an array of short audio samples, the total number of samples, and the desired new bit depth as input. It reduces the quantization of the audio samples by applying a bitwise AND operation with a masking value.

```
reduce_quantization(string inFile, string outFile, int newBits)
```

This function works with audio files. It accepts three arguments: the input WAV file path (inFile), the output WAV file path (outFile), and the new desired bit depth. It reads audio data in blocks, reduces the quantization of the samples using the first static function, and writes the modified data to the output file. The purpose of this function is to process entire WAV audio files, making it useful for batch quantization reduction tasks.

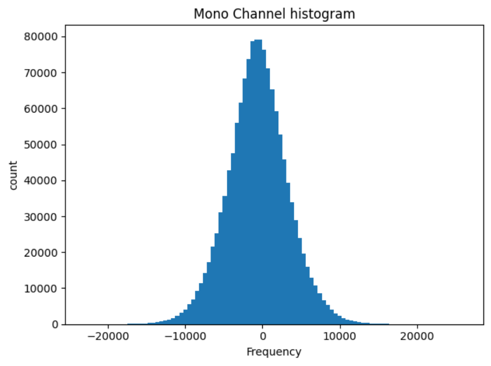
Usage:

```
../sndfile-example-bin/wav_quant <infile.wav> <outfile.wav> <bits_to_keep>
```

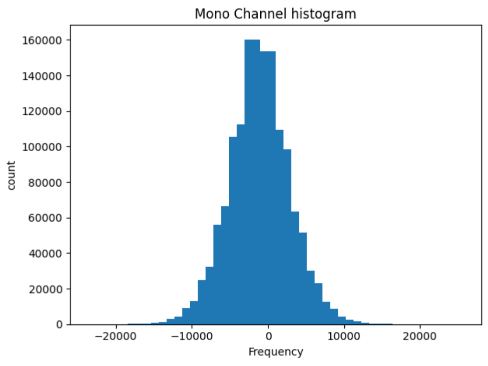
The number of bits to keep should be an integer between 1 and 16.

3.2 Results

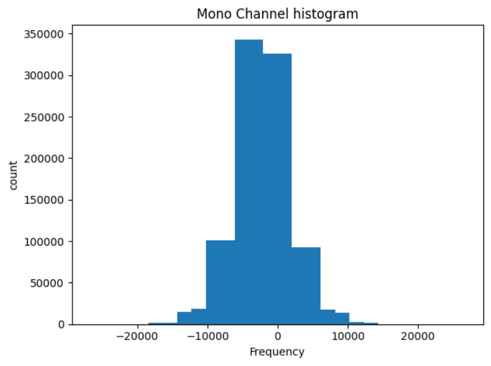
To test our class we used the sample01.wav (from before) for diferent values of quantization: 6,5,4,3. We also used the WAVHIst class and a python script to provide the graphs bellow:



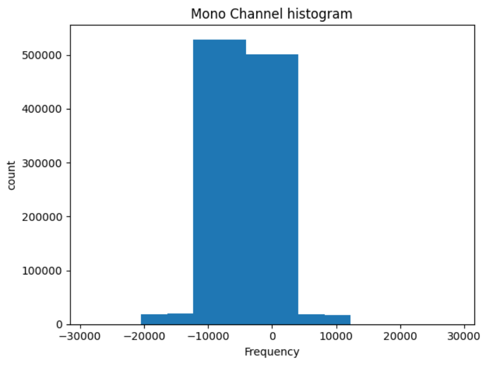
(a) 6 bits quantization



(b) 5 bits quantization



(c) 4 bits quantization



(d) 3 bits quantization

As we can see, as the quantization is smaller, we can notice that the number of bars, which represent diferent values of frequency, is smaller too.

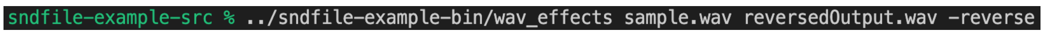
Also, we noticed a decrease in the audio quality for smaller quantizations.

4 Exercise 4 - Audio Effects

4.1 Observations

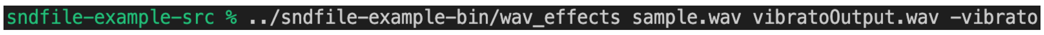
Goal: This exercise aims to create a program capable of applying some audio effects to audio files in the WAV format, allowing for the enhancement and manipulation of audio recordings and part of the goal of this exercise could include the comparison of the different output audio files after applying each audio effect to the original audio. This comparison would allow to demonstrate how each audio effect could modifies the audio as a whole, providing insights into their characteristics and differences.

To achieve the previous mentioned goal, these five distinct **audio effects** including *single echo, bass, reverse, vibrato and amplitude modulation* were chosen by the team to gain a deeper understanding of the various **alterations** in the output when compared to the original audio. The reverse and vibrato effects will be approached next, using them as examples to gain a deeper understanding of their usage and to explore the results they produce.



```
sndfile-example-src % ../sndfile-example-bin/wav_effects sample.wav reversedOutput.wav -reverse
```

Figure 5: Reverse Audio Effect Usage



```
sndfile-example-src % ../sndfile-example-bin/wav_effects sample.wav vibratoOutput.wav -vibrato
```

Figure 6: Vibrato Audio Effect Usage

4.2 Results

In order to enhance the precision of our comparisons with the original audio file and to obtain more reliable results, we chose to conduct external testing involving individuals who were not part of our development team. To facilitate this evaluation, we collaborated with four external participants. after explaining the concept and the meaning of some of these effects, it was asked to evaluate on a scale of one to five how well it was implemented the different audio effects (Fig. 7).

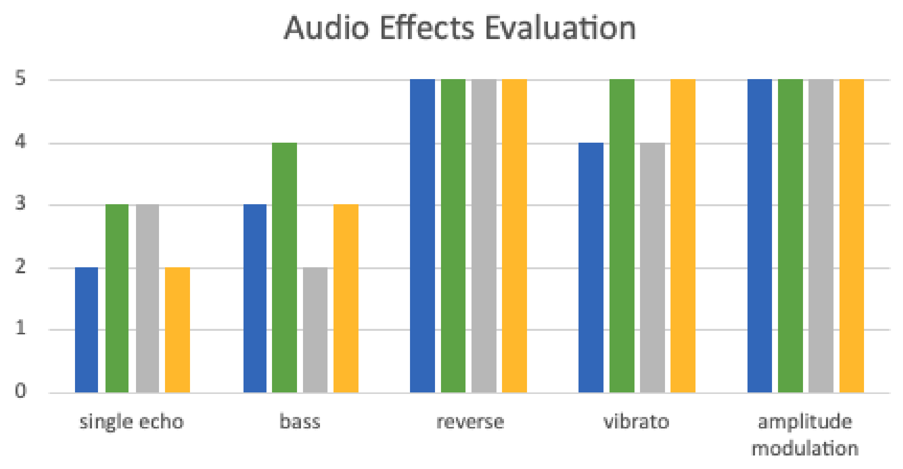


Figure 7: Audio Effects User Evaluation

After carefully analysing the obtained results on this user evaluation, it is possible to notice some flaws concerning the first two audio effects, namely **single echo** and **bass**, which were the hardest ones to implement since they did not quite produce significant modifications to the original audio, therefore were the lowest ranked on our list. On the other hand as we can observe in the figure above, the other 3 effects were quite positive highlighting reverse and amplitude modulation effects. The vibrato effect, is a classical musical concept, on which the subjects A and C were not familiar with, thus, they listen more than twice both samples and after a careful review they could actually noticed the changes.

Reversing an audio (Fig. 5) consists on changing the order of audio samples, which means that unlike the the original audio sample when the audio waveform progresses from left to right, the reversed audio waveform now progresses from right to left, effectively playing the audio in reverse order. When applied ***vibrato*** to an audio sample (Fig. 6), the pitch is modulated (frequency) of the audio signal over time, creating a swinging/oscillation in pitch. In the original audio the audio has a constant pitch, meaning that the frequency of the sound remains constant over time. After applying vibrato, it is introduce a cyclic variation, a swing, in the pitch, where this variation consists on modulating the frequency up and down around the original signal. It was implemented **amplitude modulation** audio effect, that consists in varying the amplitude (volume) of an audio signal over time. To successfully achieved the desirable goal it was used this formula: $m(t) = Acos(2\pi ft)$, , offering user options to control the frequency.

In conclusion, the "wav effects" program has successfully demonstrated its ability to apply a range of audio effects to WAV files. The results indicate that this program can be a valuable tool for audio professionals, offering creative opportunities for audio manipulation and enhancement. Future enhancements could include expanding the number of available audio effects, such as improving the ones already implemented. Additionally, optimizing the code could improve performance further.

5 Exercise 5 - BitStream Class Development

5.1 Observations

When working with binary data, the **BitStream class** is critical to read, write, and for bit-level manipulation. This class is fundamental for encoding and decoding tasks where individual bit manipulation is crucial. For example, in data compression, audio processing and image processing, the ability to efficiently handle bits is important. In this section, we will cover the BitStream class's design, characteristics, and importance, emphasising its ability to read and write individual bits and bit groups quickly as well as strings. The BitStream class developed addresses indeed this problem by providing an efficient way to perform bit operations.

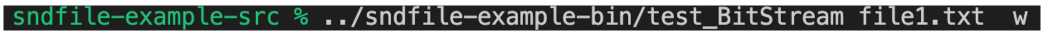
5.2 BitStream Class Overview

This class was designed to offer a flexible solution to read and write bits. In order to perform this efficient solution this class includes key components and methods such as:

1. **Bit Manipulation**: BitStream provides methods to write one bit, read one bit, write N bits and read N bits (with $0 \leq N \leq 64$), with the intent of dealing with various bit sequence lengths.
2. **String Operations**: This class provides methods for reading and writing strings, making it suitable for encoding and decoding textual information (discussed in the sec. 6).
3. **Internal Buffer**: To provide efficiency it uses an internal buffer to optimize bit-level [2].
4. **Input and Output Operations [3]**: In order to handle input and output operations this class provides 2 flags (`enum Flag w, r;`) guaranteeing flexibility whether the operation is for reading (r) or writing (w).

5.3 Results

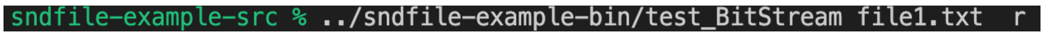
To test the efficiency and to check how suitable it is for the previously described methods in the observations subsection (5.1), the team developed a test program. The test program includes basically writing and reading operations using the flags ”w” (Fig. 8) and ”r” (Fig. 9) as a parameter as mentioned previously.



```
sndfile-example-src % ../sndfile-example-bin/test_BitStream file1.txt  w
```

Figure 8: Writing on a Binary File

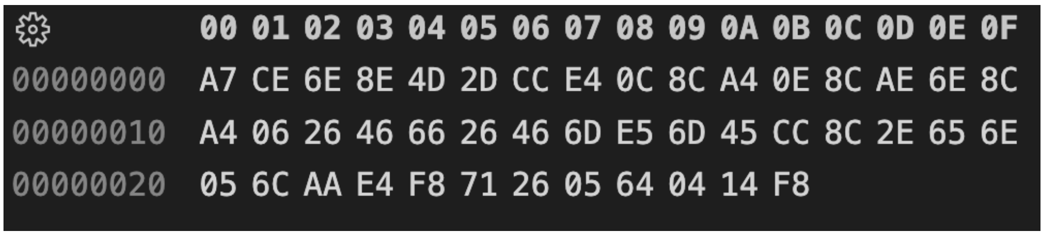
In this writing operation, it is being written to a a text file, considered as a binary file [4], eleven bits of an array of unsigned characters, and it's being initialized with two hexadecimal values: *0xA7* and *0xC3*, a string with 29 characters and then again more eleven bits and use writeBitFlush method from BitStream class that flushes the buffer to ensure all data is written.



```
sndfile-example-src % ../sndfile-example-bin/test_BitStream file1.txt  r
```

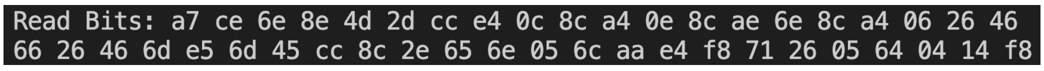
Figure 9: Reading from a Binary File

On the other hand, the reading operation opens the same file but now for reading, it reads the total of written bits from the file and then creates an array called *outBuffer* to store the data being read and calculates the *bitCount* based on the length of the string and other bits written. After reading, it prints into the console the hexadecimal values from the binary file (Fig. 10b).



```
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  A7 CE 6E 8E 4D 2D CC E4 0C 8C A4 0E 8C AE 6E 8C
00000010  A4 06 26 46 66 26 46 6D E5 6D 45 CC 8C 2E 65 6E
00000020  05 6C AA E4 F8 71 26 05 64 04 14 F8
```

(a) file1.txt - Hex Editor



```
Read Bits: a7 ce 6e 8e 4d 2d cc e4 0c 8c a4 0e 8c ae 6e 8c a4 06 26 46
66 26 46 6d e5 6d 45 cc 8c 2e 65 6e 05 6c aa e4 f8 71 26 05 64 04 14 f8
```

(b) file1.txt - Read Operation

Figure 10: Writing and Reading Operations Comparison

With the intention of verifying the results and further the efficiency of this class, there was made a comparison between writing and reading operations. This evaluation was accomplished by reviewing carefully the binary data from file's *Hex Editor* (Fig. 10a) along with the output after the reading operation, and as we can see on Fig. 10, it is possible to understand that both contain the same binary data.

In conclusion, the team was able to develop the BitStream class requested as well as to recognize how valuable this class can be when working with binary data. Its flexibility, efficiency solution for tasks involving bit manipulation. It simplifies complex encoding and decoding processes and enhances performance of applications that requires accurate control over binary data.

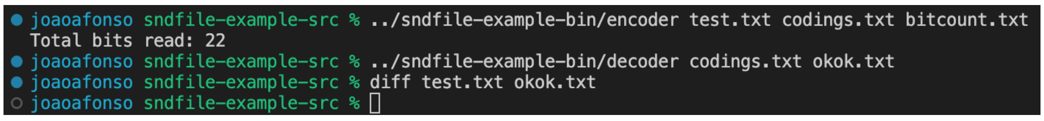
6 Exercise 6 - Encoder and Decoder BitStream

6.1 Observations

Goal: The output of both programs must be exactly the same as the original text file when executed correctly. At the end use the command-line utility *'diff'* to compare both.

For this exercise, we faced some challenges due to the fact that we couldn't assure that the **number of bits read** on the text file, **when not divisible by 8**, was **the same after** the execution of encoder and decoder programs. The encoder program reads textual data and converts it into binary, it writes bits appending unwanted bits to complete the eight bit groups, and when decoding it was reading and decoding these unwanted bits which was not the point of this exercise. For this, ensuring that both the encoder and decoder agree on how to **handle the bit count number read** and how data is encoded was crucial.

Usage:

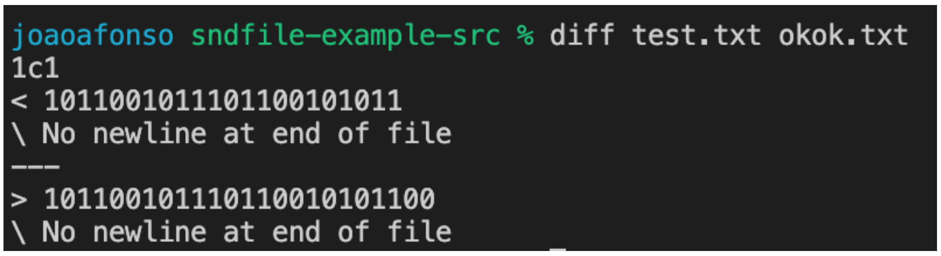


```
● joaoafonso sndfile-example-src % ../sndfile-example-bin/encoder test.txt codings.txt bitcount.txt
  Total bits read: 22
● joaoafonso sndfile-example-src % ../sndfile-example-bin/decoder codings.txt okok.txt
● joaoafonso sndfile-example-src % diff test.txt okok.txt
○ joaoafonso sndfile-example-src % 
```

Figure 11: Encoding and Decoding Usage

6.2 Results

- Without BitCount Handling:

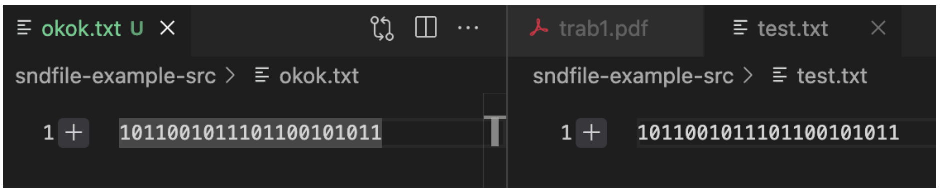


```
joaoafonso sndfile-example-src % diff test.txt okok.txt
1c1
< 1011001011101100101011
\ No newline at end of file
---
> 101100101110110010101100
\ No newline at end of file
```

Figure 12: Not Successful - Original Text File and Output file Comparison

Observing the Fig. 12 it is possible to recognize the comparison made between the two files, concluding that it **does not** satisfies the goal of this problem, since there are some**differences** between them. When carefully reviewed the differences, there are two zeros in **excess**. In the early stages of this exercise's development, the data was being encoded and decoded without taking into account the extra zero bits added when the text file did not have a bit count **divisible by 8**. In other words, zeros were appended to complete the eight bit groups. Consequently, this led to a **deviation** in the outcome in comparison to the original text file, due to the fact that when decoding, it **was reading more bits than the ones originally on the text file**.

- With BitCount Handling:


okok.txt U
trab1.pdf
test.txt
sndfile-example-src > okok.txt
sndfile-example-src > test.txt

```
1011001011101100101011
```

```
1011001011101100101011
```

Figure 13: Successful - Original Text File and Output file Comparison

After carefully reviewing the problem and with the guidance of our teacher, we discovered an **alternative approach for managing bit counting** to prevent the occurrence of **unwanted bits**. As a result, to address this bit counting challenge, we came across a **solution** that involves storing a **variable equal to the number of bits read** during the encoder's execution and writing it into a new file called "bit-count.txt". This ensures that, when decoding, the **same number of bits read are faithfully decoded**.

To conclude, it was possible to achieve the goal of the problem since there are **no differences** between those two files as it is possible to observe in Fig. 11 & Fig. 13 when running the command-line utility *diff*.

7 Exercise 7 - Lossy Codec for Mono Audio Files

7.1 Observations

Goal: Implement a *lossy codec* for mono audio files (i.e., with a single channel), based on the Discrete Cosine Transform (DCT). The audio should be processed on a block by block basis. Each block will have to be converted using the DCT, the coefficients appropriately quantize, and the bits written to a file, using the BitStream class. Note: the idea is to get good compression without degrading too much the audio.

This last exercise summarizes the project in a number of different ways, there was a need to deeply understand how to read and write .wav files, read and write binary files and quantize different types of data. The *dct* part was provided and was used as base. For reading and writing files we used our own class created on a previous exercise as well as the already developed quantization class.

7.2 Approach

To compress the audio file, it was used a pair of tools: *Discrete Cosine Transform and quantization.* The DCT transforms a block of frequency samples into a sum of cosine waves. These cosine waves are calculated in such a way that the most relevant ones tend to dominate initially. Consequently, we can discard a portion of those values and still experience minimal audio quality loss during the inverse transformation. The selected cosine waves are represented by floating-point numbers, and these values can be quantized.

Quantization allows us to eliminate the excess bits from our output file, which occurs because numbers are typically designed to store a certain range (e.g., a short int has 16 bits, ranging from negative $(2^{15} - 1)$ to positive 2^{15}). However, we may only need to store a smaller range. In such cases, dropping the most significant bits saves space without any adverse effects.

After analyzing several .wav files and determining the minimum and maximum values of the cosine waves, we concluded that we would need 14 bits (a range of -8192 to 8191). This means that a file with 1000 KB, without removing unnecessary values from the DCT, would only be reduced to $875KB$ (using a bit ratio of (14/16), which accounts for the quantization). While this might not seem like a significant reduction, another solution we explored involved quantizing the double values returned by the DCT.

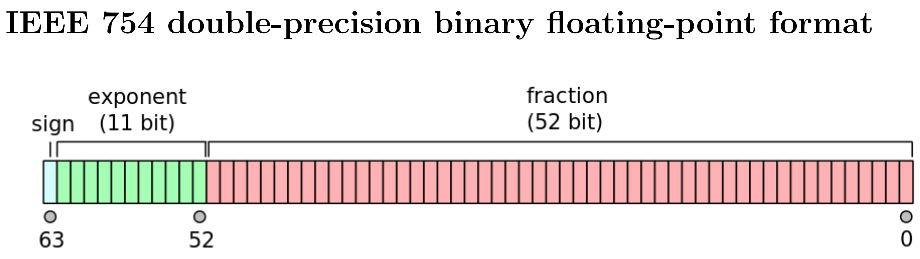


Figure 14: IEEE 754 Double Floating Point Format

A double value is stored in three parts: the first bit is responsible for storing the sign of the number, the next 11 bits store the exponent, and the remaining bits represent the mantissa. Both the exponent and the mantissa can be significantly reduced. The required exponent range is just 4, and the mantissa can be more dynamically reduced; for example, we can use as few as two bits. However, using such a small number of mantissa bits may introduce some noise into the output audio. In this way, we can represent each wave with just $1 + 4 + 2 = 7$ bits, reducing the size of our 1000 KB file to 437.5 KB.

7.3 Results

To accomplish the tasks mentioned above, we developed wav_dct.cpp. This file is divided into two main sections: one for compressing the audio file and the other for decompressing it. To execute the specific section you desire, you can use flags. Additionally, there are flags available for adjusting variables as needed.

```
void showhelp() {
    std::cerr << "Usage: wav_dct [ -c file to compress ]\n";
    std::cerr << "               [ -d file to decompress ]\n";
    std::cerr << "               [ -bs blockSize (def 1024) ]\n";
    std::cerr << "               [ -frac dctFraction (def 0.2) ]\n";
    std::cerr << "               [ -qe quantizationExponent (def 11) ]\n";
    std::cerr << "               [ -qm quantizationMantissa (def 5) ]\n";
    std::cerr << "               outputfile\n";
}
```

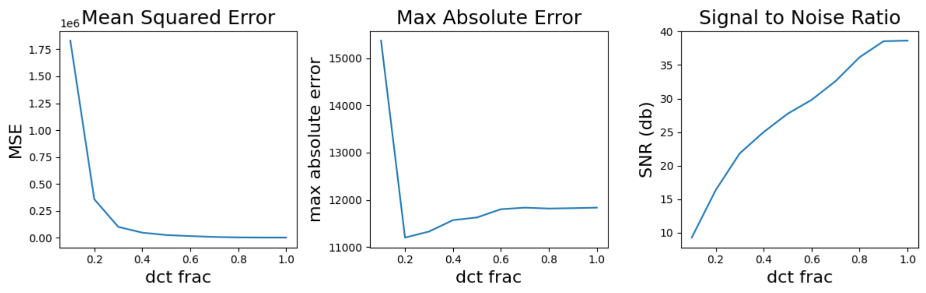


Figure 15: DCT Fraction of Values Used

The impact of discarding less significant values returned by the DCT is illustrated in Figure 15, which is based on values generated using sample01.wav. The figure demonstrates that after 0.3, the squared error becomes close to 0. However, to our ears, the audio only becomes sufficiently close to the original at around 0.5.

In terms of the Signal-to-Noise Ratio (SNR), we do not observe the value reaching infinity. This is because these tests were conducted using only 10 bits in the mantissa, which proves to be precise enough, as indicated in the next figure.

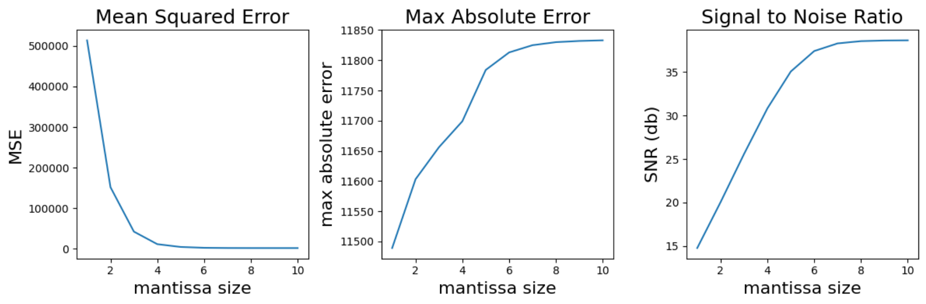


Figure 16: Mantissa Size

As previously mentioned, even when using only a few bits for the mantissa, the compression remains remarkably accurate. It appears that six bits are generally sufficient for nearly 100% recovery of the audio.

7.4 Future Work

We didn't have the chance to complete the code for quantizing the exponent part of the double. Nevertheless, we anticipate that it will not perform in a similar way to mantissa quantization. There should be a range where it simply won't work, and then a range where it performs perfectly every time. This occurs when there is a loss of a single bit in the exponent that can significantly impact the value itself, and the ideal number of bits for quantization would be 5.

8 Acronyms

DCT Discrete Cosine Transform

SNR Signal-to-Noise Ratio

9 Bibliography

References

[1] Matplotlib, *Matplotlib: Visualization with python*, `https://matplotlib.org`, Accessed: [10/2023], 2023.

[2] Tutorials point, *Buffering*, `https://shorturl.at/eBGQ6`, Accessed: [10/2023], 2023.

[3] C plus plus, *Input/output with files*, `https://cplusplus.com/doc/tutorial/files/`, Accessed: [10/2023], 2023.

[4] Tech Target, *Binary file*, `https://shorturl.at/JR078`, Accessed: [10/2023], 2023.