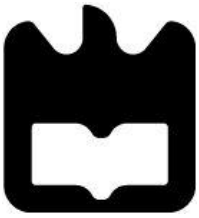
IC - Assignment I

Universidade de Aveiro

Departamento de Eletrónica, Telecomunicações e Informática

Mestrado em Engenharia Informática



Diogo Carvalho nº92969 1/3

Hugo Ferreira nº93093 1/3

Rafael Baptista nº93367 1/3

Repositório: https://github.com/rafaelbaptista13/IC

30 de Outubro de 2022

Índice

1. Introdução

O presente relatório pretende descrever o trabalho realizado na execução do primeiro projeto proposto na unidade curricular de Informação e Codificação, cuja estrutura do enunciado se encontrava dividida em 4 partes distintas.

A primeira fase do projeto continha 5 exercícios que estavam relacionados com a análise do código base fornecido pelo professor para iniciar o trabalho, construção histograma para o canal da versão mono, ou “MID Channel” e canal da diferença ou “SIDE Channel”, implementar programa para efetuar quantização do áudio, cálculo do valor signal-to-noise ratio (SNR) e por fim a implementação de alguns efeitos sonoros no áudio.

Na segunda parte, o enunciado era composto por 2 exercícios, um deles relacionado com a construção de uma BitStream cujo objetivo era ler/escrever bits de/para um ficheiro e o segundo com a utilização desta BitStream efetuar a implementação de um codificador que converte texto de 0’s e 1’s para formato binário e um descodificador que efetua a operação inversa.

A terceira parte do projeto tinha um exercício único cujo objetivo principal era a construção de um “lossy audio codec” que seria baseado na DCT (Discrete Cosine Transform).

Por fim, a última parte do projeto consiste na escrita deste mesmo documento cujo objetivo é explicar as decisões efetuadas e resultados obtidos ao longo do projeto.

2. Metodologia

Para a realização deste projeto, utilizámos a plataforma Github como repositório de código compartilhado. A estrutura do nosso repositório é constituída por duas pastas distintas. A pasta “sndfile-example-src” que contém todo o código desenvolvido ao longo do projeto, e uma pasta “examples” que contém os ficheiros utilizados para testar os diferentes programas que foram desenvolvidos.

No que diz respeito à comunicação entre os elementos do grupo, utilizámos as plataformas Discord e Messenger para efetuar a discussão dos pontos críticos do nosso projeto, facilitando assim a cooperação à distância na realização do mesmo.

Relativamente ao código escrito, o mesmo foi desenvolvido na linguagem de programação C++. A estrutura do código é composta por:

- Programa *wav_hist.cpp* que utiliza a classe WAVHist definida no ficheiro *wav_hist.h* para efetuar a criação do histograma das amplitudes dos diferentes canais de áudio (inclusive MID e SIDE Channel).
- Programa *wav_quant.cpp* que faz uso da classe WAVQuant definida no ficheiro *wav_quant.h* para reduzir o número de bits utilizados para representar cada amostra de áudio através da aplicação quantização escalar uniforme.
- Programa *wav_cmp.cpp* cujo objetivo é efetuar o cálculo do valor do signal-to-noise ratio (SNR) entre um ficheiro de áudio e o ficheiro original.
- Programa *wav_effects.cpp* cujo objetivo é a produção de efeitos sonoros no áudio, efeitos como eco simples, ecos múltiplos, delays de tempo variado entre outros. Para isso, utiliza a classe WAVEffects definida no ficheiro *wav_effects.h*.
- Programa BitStream.cpp cujo objetivo é a utilização de uma “bit stream” definida na classe BitStream no ficheiro *BitStream.h* com a finalidade de converter um texto de 0’s e 1’s em texto binário equivalente bem como o processo inverso.
- Programa *wav_dct_codec.ccp* cujo objetivo é comprimir um ficheiro de áudio e guardá-lo num ficheiro de bits num processo baseado na Discrete Cosine Transform, em que se guarda apenas uma fração das frequências consideradas mais importantes.
- Programa *wav_dct_decoder.ccp* cujo objetivo é realizar o processo inverso do anterior. A partir de um ficheiro de bits (obtido através do programa anterior), este programa é responsável por extrair os coeficientes do ficheiro, e realizar a reconstrução do ficheiro de áudio novamente.

3. Implementação e Modo Utilização Programas

Nesta seção do nosso relatório iremos descrever a implementação que efetuamos para os nossos programas bem como analisar e justificar as decisões tomadas durante esta mesma fase de implementação.

3.1. wav_hist.cpp e wav_hist.h

Vamos iniciar pelo programa desenvolvido no exercício 2 da primeira parte do projeto (wav_hist.cpp). O mesmo tem como objetivo efetuar a construção dos histogramas dos valores da amplitude para os diferentes canais do áudio, inclusive o MID e SIDE Channel.

O MID Channel é calculado a partir da média dos canais e o SIDE Channel é calculado através da diferença entre os canais.

Para correr este programa, é necessário especificar o nome do ficheiro de áudio de input para analisar bem como qual o canal de que se vai criar o histograma.

```
Usage:
./wav_hist <input_file> <channel>
```

Relativamente ao canal, os seguintes valores são permitidos:

- *0, x-1* : Valor inteiro de 0 até x-1 em que x é o número de canais do áudio.
- *SIDE, side* : Strings válidas para calcular o histograma do SIDE Channel.
- *MID, mid* : Strings válidas para calcular o histograma do MID Channel.

```
Usage Examples:
./wav_hist ../examples/sample.wav MID
./wav_hist ../examples/sample.wav SIDE
 ./wav_hist ../examples/sample.wav 0
 ./wav_hist ../examples/sample.wav 1
```

Ao executar o nosso programa, o primeiro passo a ser efetuado é a verificação dos parâmetros fornecidos pelo utilizador. É verificada a existência e formato do ficheiro de input fornecido.

De seguida, é efetuada a verificação de qual foi o canal escolhido pelo utilizador para a criação do histograma.

Posteriormente são criados um vetor de amostras e um objeto da classe WAVHist onde é passado como argumento o ficheiro de áudio e duas flags que indicam se o canal selecionado foi o canal MID, SIDE ou outro canal.

Finalmente as amostras são lidas do ficheiro de áudio e utilizadas para atualizar o histograma através da chamada ao método update() do objeto da classe WAVHist criado anteriormente. Por fim, os valores do histograma são apresentados ao utilizador através do método dump() do mesmo objeto.

Relativamente ao método update() da classe WAVHist, este método começa por verificar consoante as flags fornecidas no construtor do objeto, qual o canal escolhido para a criação do histograma.

De seguida, o método vai percorrer o vetor de amostras que é fornecido como argumento do método e vai atualizar um mapa com as contagens do número de ocorrência de cada amplitude.

No caso do MID Channel é efetuada a média dos canais e no SIDE Channel é calculada a diferença entre os canais. Estes cálculos seguem as fórmulas fornecidas no enunciado do projeto.

3.2. wav_quant.cpp e wav_quant.h

Vamos prosseguir para a análise do programa desenvolvido no exercício 3 da primeira parte do projeto (wav_quant.cpp). O mesmo tem como objetivo efetuar a redução do número de bits necessários para representar as amostras do áudio através da aplicação da quantização escalar uniforme.

Para executar este programa, é necessário especificar o nome do ficheiro de áudio de input, que vai ser quantizado, nome do ficheiro de output, o número de bits com que se pretende representar cada amostra após o processo de quantização e ainda a versão do processo de quantização que pretendemos utilizar, isto porque foram implementadas duas soluções distintas para comparação e que irão ser explicadas mais à frente.

```
Usage:
./wav_quant <input file> <output file> <num_bits> <version>
```

Relativamente ao valor da versão, os seguintes valores são permitidos:

- *1* : Versão em que a quantização é realizada sem guardar valores de amplitude 0 (será detalhado o processo mais à frente).
- *2* : Versão em que a quantização é realizada a guardar valores de amplitude 0 (será detalhado o processo mais à frente).

```
Usage Examples:
./wav_quant ../examples/sample.wav output.wav 4 1
./wav_quant ../examples/sample.wav output.wav 4 2
./wav_quant ../examples/sample.wav output.wav 8 1
./wav_quant ../examples/sample.wav output.wav 8 2
```

Ao executar o nosso programa, o primeiro passo a ser efetuado é a verificação dos parâmetros fornecidos pelo utilizador. É verificada a existência e formato do ficheiro de input fornecido.

De seguida, é efetuada a verificação de qual foi o número de bits escolhido pelo utilizador e de que o mesmo é válido.

Prosseguimos para a verificação de qual foi a versão escolhida pelo utilizador, bem como se a mesma é válida ou não.

Posteriormente são criados um vetor de amostras e um objeto da classe WAVQuant onde é passado como argumento o número de bits que deve ser utilizado para a quantização e ainda a versão do processo de quantização selecionada pelo utilizador. No construtor desta classe, vai ser construído um mapa com todas as amplitudes, e o respetivo valor quantizado, para que depois seja obtido na função quantization(). A forma de como este mapa é construído depende da versão que o utilizador escolher.

Finalmente as amostras são lidas do ficheiro de áudio, são quantizadas através da chamada ao método quantization() do objeto da classe WAVQuant criado anteriormente, que simplesmente acede ao mapa para ter o respetivo valor quantizado.

Após a quantização, as amostras quantizadas são escritas no ficheiro de output.

Relativamente à criação do mapa na classe WAVQuant, foi implementado de duas formas distintas por forma a verificar as diferenças nos resultados entre ambas, garantindo assim uma melhor aprendizagem dos conteúdos. Posto isto, na primeira versão, o mapa com os valores de amplitude quantizadas vai ser gerado sem ter em conta o valor de 0. Exemplificando, num intervalo de amplitudes de - 8 a 7 (representado com 4 bits), quando se faz a quantização para utilizar apenas 2 bits, as amplitudes quantizadas irão ser convertidas para os valores -6 -2 2 6. Pelo que os valores de amplitude originais do sub intervalo de -8 a -5 são convertidos para o valor -6, do sub intervalo de -4 a -1 são convertidos para o valor -2, do sub intervalo de 0 a 3 são convertidos para o valor 2 e do sub intervalo de 4 a 7 são convertidos para o valor 6. O problema desta versão é que após quantizado, valores que originalmente eram 0 (silêncio), passam a produzir som no áudio quantizado.

No que diz respeito à segunda versão, o mapa com os valores de amplitude quantizadas vai ser gerado a ter em conta o valor 0. Exemplificando, num intervalo de amplitudes de -8 a 7 (representado com 4 bits), quando se faz a quantização para utilizar apenas 2 bits, as amplitudes quantizadas irão ser convertidas para os valores -8 -4 0 4. Pelo que os valores de amplitude originais do sub intervalo de -8 a -6 são convertidos para o valor -8, do sub intervalo de -5 a -2 são convertidos para o valor -4, do sub intervalo -1 a 2 são convertidos para o valor 0, e do sub intervalo 3 a 7 são convertidos para o valor 4. O problema desta versão é que após quantizado, os valores de amplitude -1, 1 e 2 originais (que produziam som) passam a não produzir porque são convertidos para 0. Para além disso, nesta versão os valores não são distribuídos de forma tão uniforme como na primeira versão.

3.3. wav_cmp.cpp

O programa desenvolvido para dar resposta ao exercício 4 da primeira parte do projeto (wav_cmp.cpp) tem como objetivo efetuar o cálculo do valor do signal-to-noise ratio (SNR) entre um ficheiro de áudio e o ficheiro original do áudio em questão.

Adicionalmente este programa deve apresentar o valor do máximo erro absoluto por amostra.

Para executar este programa, é necessário especificar o nome dos dois ficheiros de áudio, isto é, ficheiro quantizado e o ficheiro original.

```
Usage:
./wav_cmp <input file> <output file>

Usage Example:
./wav_cmp ../examples/sample.wav output.wav
```

Ao executar o nosso programa, o primeiro passo a ser efetuado é a verificação dos parâmetros fornecidos pelo utilizador. É verificada a existência e formato de ambos os ficheiros fornecidos.

Prosseguindo é efetuada a criação de dois vetores de amostras, um para cada ficheiro. É efetuada a leitura das amostras de ambos os ficheiros para os vetores em questão, de forma simultânea, e é calculado o valor de SNR e o valor do máximo erro absoluto por amostra.

No fim, ambos os valores são apresentados ao utilizador.

No que diz respeito ao cálculo do valor de SNR, é seguida a seguinte abordagem:

$$EQ1: EnergiaAudioOriginal = \sum |x|^2$$

$$EQ2: EnergiaRuído = \sum |x - x_{quantizado}|^2$$

$$EQ3: SNR = 10 * log_{10}\left(\frac{EnergiaAudioOriginal}{EnergiaRuído}\right)$$

3.4. wav_effects.cpp e wav_effects.h

Finalmente, o programa construído para solucionar o último exercício da primeira parte do projeto (wav_effects.cpp) tem como objetivo permitir a produção de efeitos sonoros tais como o eco em ficheiros de áudio.

Os efeitos sonoros disponíveis neste programa são:

- SINGLE : eco de tempo constante.
- MULTIPLE : ecos múltiplos.
- TIME_VARYING : eco de tempo variável.
- AMPLITUDE_MOD : modulação da amplitude.

Para executar este programa, é necessário especificar o nome do ficheiro de áudio de input, a que vai ser aplicado o efeito, nome do ficheiro de output, mode, que será o tipo de efeito a ser aplicado, delay que vai ser o tempo, em segundos, do atraso do eco (no modo SINGLE e MULTIPLE) ou o atraso máximo (no modo TIME_VARYING) e o amplitude_eco que será a amplitude relativa do eco.

```
Usage:
./wav_effects <input file> <output file> <mode> <delay>
<amplitude_eco>

Usage Example:
./wav_effects ../examples/sample.wav output.wav SINGLE 2 0.5
./wav_effects ../examples/sample.wav output.wav MULTIPLE 2 0.5
./wav_effects ../examples/sample.wav output.wav AMPLITUDE_MOD 2 0.5
./wav_effects ../examples/sample.wav output.wav TIME_VARYING 2 0.5
```

Ao executar o nosso programa, o primeiro passo a ser efetuado é a verificação dos parâmetros fornecidos pelo utilizador.

De seguida, é criado um vetor de amostras para as amostras lidas do ficheiro de entrada e é criado o objeto da classe WAVEffects onde são passados como argumentos o SndFileHandle do ficheiro de input para se obter informações como a frequência de amostragem e o número de canais, o tipo de efeito a ser aplicado, o K (número de samples

que existem no tempo de atraso do eco), a amplitude do eco e o tamanho do vetor do ficheiro de output. Na inicialização deste objeto é criado um vetor de output que vai ser utilizado para escrever o ficheiro de output.

Prosseguindo, as amostras são lidas do ficheiro de áudio e depois é aplicada a função applyEffect, que aplica o efeito escolhido às amostras de áudio.

Para o efeito de modulação da amplitude, o valor do sinal é multiplicado pela seguinte função cosseno com frequência de 1.5Hz:

$$\cos\left(2\pi\frac{f}{fs}n\right)$$

Para os restantes efeitos, é criada uma deque que vai ser preenchida inicialmente com as primeiras K amostras, após isso para TIME_VARYING é escolhido a partir de uma função cosseno uma das 200 amostras na frente da deque para ser somada com a amostra e para os restantes é somado à amostra o valor na frente da deque ajustado com a ampEco. Estes resultados formam as amostras do ficheiro de output.

De seguida, o valor na frente da deque é descartado. É adicionado ao fim da deque o da amostra calculada anteriormente para multiple e para os restantes é adicionada a amostra inicial.

Finalmente, o vetor de output é utilizado para escrever o ficheiro de output.

3.5. BitStream.cpp e BitStream.h

Vamos avançar para a descrição do programa desenvolvido para solucionar os dois exercícios correspondentes à segunda parte do projeto (BitStream.cpp). Este programa tem como objetivo a criação de uma “bit stream” que permita efetuar a leitura de bits de um determinado ficheiro de input ou a escrita de bits para um ficheiro de output. Para isso, uma classe BitStream deveria ser definida e construída com os métodos base de get_bit() e write_bit() podendo ser adicionados métodos extra à classe tais como get_n_bits() e write_n_bits(). Posto isto, o objetivo final era a utilização desta “bit stream” para efetuar a conversão de um ficheiro de texto contendo apenas 0’s e 1’s num ficheiro binário equivalente, bem como a utilização da “bit stream” para o processo inverso.

Para executar este programa, é necessário indicar o modo de utilização da BitStream, especificar o nome dos dois ficheiros, isto é, o nome do ficheiro de texto e o nome do ficheiro de áudio.

```
Usage: ./BitStream [-r (def)]
                   [-w]
                   <text_file> <bin_file>
```

Relativamente aos modos de utilização da BitStream, as seguintes flags são permitidas:

- *-r* : Para utilizar a BitStream no modo de leitura;
- *-w* : Para utilizar a BitStream no modo de escrita.

```
                    Usage Examples:
./BitStream -r ../examples/output_binary_data ../examples/data
./BitStream -w ../examples/data ../examples/output_binary_data
```

Ao executar o nosso programa, o primeiro passo a ser efetuado é a verificação dos parâmetros fornecidos pelo utilizador. É verificado qual o modo de utilização da BitStream selecionado pelo utilizador. Caso nenhum seja selecionado, por defeito a BitStream é utilizada no modo de leitura.

De seguida, é efetuada a criação do objeto da classe BitStream onde é fornecido o nome do ficheiro com que a BitStream vai interagir, isto é, efetuar operações de escrita ou leitura (consoante o modo de utilização).

No modo de utilização de leitura, o programa vai efetuar chamadas ao método get_bit() da BitStream para obter bits de dados que possa posteriormente escrever para o ficheiro final de output.

No modo de utilização de escrita, o programa vai ler bytes de dados do ficheiro de texto e vai efetuar chamadas ao método write_bit() da BitStream para escrever bits de informação para o ficheiro binário de output.

Adicionalmente foram desenvolvidos os métodos write_n_bits() e get_n_bits() que podem ser utilizados para escrever e ler um número maior de bits de cada vez, respetivamente.

O método get_n_bits() recebe como argumento de entrada, N, o número de bits que se pretende obter, e o seu funcionamento é simplesmente invocar a função get_bit() N vezes, e retornar uma String resultante da concatenação de todos os bits. O método write_n_bits() recebe como argumento de entrada uma String com os bits que se pretende escrever, e o seu funcionamento é simplesmente invocar a função write_bit() para cada bit presente na String de entrada.

3.6. wav_dct_codec.cpp e wav_dct_decoder.cpp

Por fim, vamos descrever os dois programas correspondentes à solução dos exercícios da terceira parte do projeto (wav_dct_codec.cpp e wav_dct_decoder.cpp). O objetivo destes programas são possibilitar a compressão de um ficheiro áudio através da remoção das amplitudes que não são consideradas importantes. Esta gama de amplitudes que são descartadas são definidas através de um argumento do programa na forma de uma fração. Para alcançar este objetivo, era necessário implementar um codificador com perdas de áudio baseado na Discrete Cosine Transform (DCT). Esta codificação seria realizada numa abordagem por blocos, em que cada bloco iria ser convertido usando a DCT, os coeficientes quantizados e escritos para um ficheiro usando a classe BitStream do ponto anterior. O descodificador devia conseguir reconstruir o áudio (com eventuais perdas) baseado no ficheiro binário criado pelo codificador. Para executar o codificador, o utilizador precisa de fornecer o ficheiro para o qual os bits vão ser armazenados (outputCodecFile) e o ficheiro áudio que pretende codificar (wavFileIn). Para além disso tem a possibilidade de escolher o modo verbose (-v), definir o tamanho do bloco (-bs blockSize), e definir a fração de amplitudes que são consideradas importantes (-frac dctFraction).

```
Usage: ./wav_dct_codec  [-v (verbose)]
                        [-bs blockSize (def 1024)]
                        [-frac dctFraction (def 0.2)]
                        <output_codec_file> <wav_file_in>

                        Usage Examples:
  ./wav_dct_codec ../examples/bits_file ../examples/sample.wav
```

Ao executar o programa, o primeiro passo realizado é a verificação dos parâmetros, e consequente escrita dos mesmos para variáveis para mais tarde serem utilizados. De seguida, é criado um objeto do tipo BitStream no modo de escrita para o ficheiro passado como argumento do programa outputCodecFile, e é escrito para o ficheiro utilizando a função write_n_bits(), o formato, os canais, as frames, a sample rate do ficheiro de áudio e a fração escolhida pelo utilizador. É necessário que esta informação seja guardada para que o descodificador consiga reconstruir o áudio. Após isso, são lidas todas as frames do ficheiro de áudio e são declarados um vetor para guardar todos os coeficientes provenientes da DCT, e um vetor para guardar todos os cálculos da DCT. De seguida, começa o processamento do ficheiro de áudio numa abordagem por blocos, sendo que para cada bloco, primeiramente é realizado o cálculo da DCT para todos os elementos do bloco, e de seguida são calculados os coeficientes para serem escritos para disco, sendo que são apenas calculados valores até à fração definida pelo utilizador.

Para executar o descodificador, o utilizador precisa de fornecer o ficheiro binário proveniente da codificação utilizando o programa anterior, e o ficheiro onde vai ser reconstruído o áudio. Para além disso, o utilizador tem também a possibilidade de definir o tamanho do bloco, sendo que deve corresponder ao mesmo valor que foi utilizado para codificar o ficheiro.

```
Usage: ./wav_dct_decoder [-bs blockSize (def 1024)]
                         <input_codec_file> <wav_file_out>

                       Usage Examples:
       ./wav_dct_decoder ../examples/bits_file output.wav
```

Ao executar o programa, o primeiro passo realizado é a verificação dos parâmetros, e consequente escrita do valor de blockSize para uma variável caso exista. De seguida, é criado um objeto do tipo BitStream no modo de leitura para o ficheiro passado como argumento do programa inputCodecFile, e é lido o formato, os canais, as frames, o sample rate, e a fração escolhida, utilizando a função read_n_bits() para de seguida criar o ficheiro que irá ser usado para reconstruir o ficheiro de áudio. Após isso, são declarados um vetor para guardar todas as samples e um vetor para guardar todos os coeficientes que vão ser obtidos a partir do ficheiro binário. De seguida começa a reconstrução das samples do ficheiro de áudio na mesma abordagem por blocos, sendo que para cada bloco, primeiramente é lido o coeficiente da DCT guardado do ficheiro para todos os elementos do bloco, e de seguida é calculado o valor da sample através da inversa da DCT.

4. Análise de resultados

Nesta seção do relatório iremos apresentar algumas experiências efetuadas com os programas desenvolvidos e analisar os resultados obtidos.

4.1. wav_hist.cpp

Relativamente ao programa wav_hist.cpp, efetuamos a execução do mesmo para cada um dos canais de áudio do ficheiro de áudio fornecido pelo professor (sample.wav), incluindo para o MID e SIDE Channel.

Posto isto, seguem-se as imagens obtidas dos histogramas para cada um dos canais enumerados:

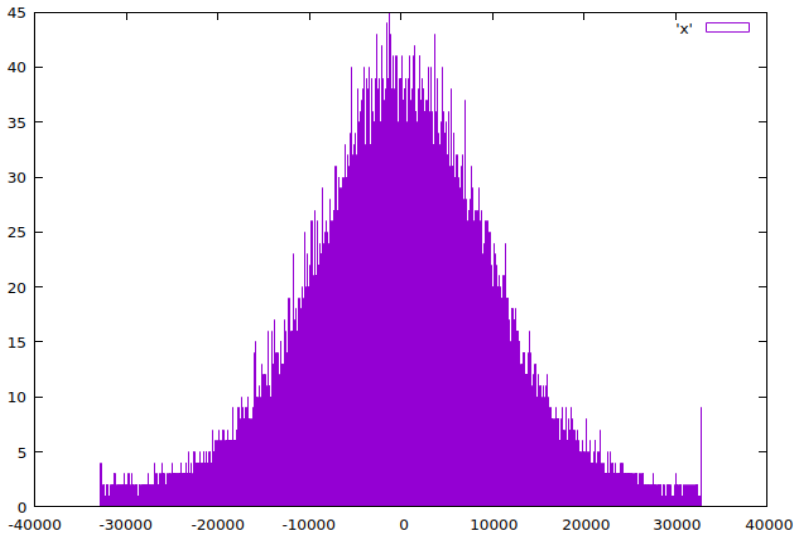


Figura 1: Histograma do canal de áudio 0

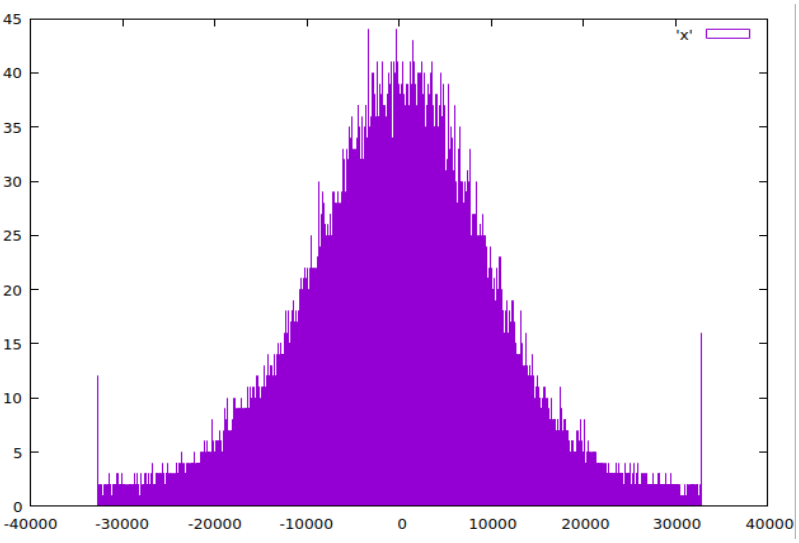


Figura 2: Histograma do canal de áudio 1

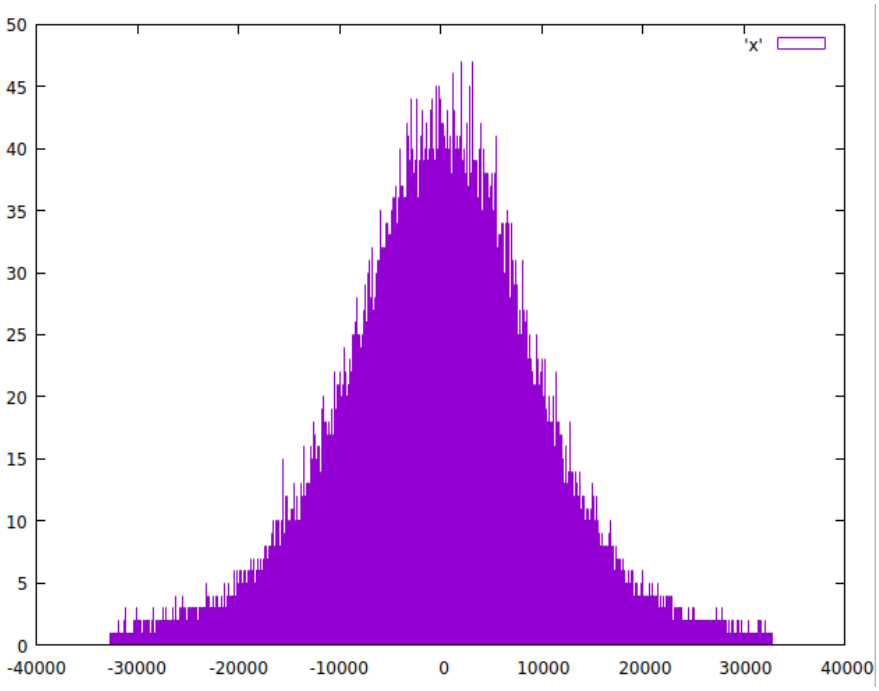


Figura 3: Histograma do MID Channel

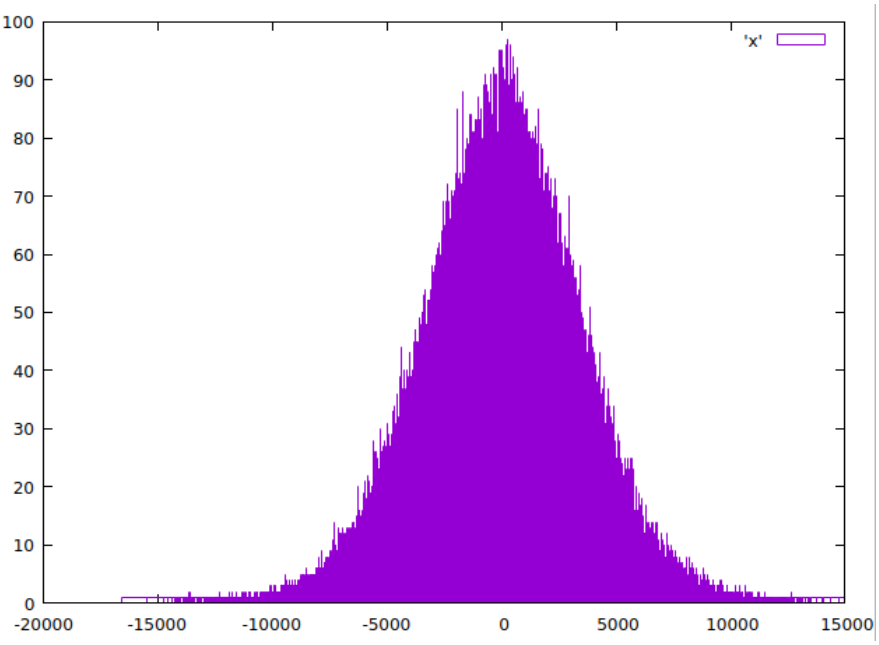


Figura 4: Histograma do SIDE Channel

No eixo das abscissas dos diferentes gráficos temos os valores da amplitude das amostras dos canais de áudio. No eixo das ordenadas temos o número de vezes que esse valor de amplitude foi verificado.

Os histogramas obtidos estão de acordo com aqueles que foram apresentados ao longo das aulas práticas pelo professor. Podemos ver que, por exemplo, para o histograma do SIDE Channel, visto que o mesmo resulta da diferença entre os valores de amplitude das amostras dos canais, o intervalo de amplitudes registadas é mais reduzido e ao mesmo tempo o número de vezes que essas amplitudes são registadas é maior.

4.2. wav_quant.cpp

O programa wav_quant.cpp, como já descrito, permite efetuar a quantização das amostras de um ficheiro de áudio permitindo assim representar essas mesmas amostras com um determinado número de bits inferior ao número de bits original e selecionado pelo utilizador. A forma de utilização deste programa é descrita na secção 3.2 deste relatório.

Relativamente a resultados obtidos com este programa, executamos o mesmo para três ficheiros de áudio distintos, incluindo o ficheiro original distribuído pelo professor. Para cada ficheiro efetuamos a quantização para diferentes números de bits, e medimos o tempo de execução do programa.

Os resultados obtidos para o tempo de execução do programa com a versão 1 do processo de quantização estão presentes na seguinte tabela:

Tabela 1: Tempo de Execução do programa wav_quant para Versão 1

num_bits	*Áudio Número 1*	*Áudio Número 2*	*Áudio Número 3*
1	0.240 s	0.362 s	0.233 s
2	0.236 s	0.372 s	0.244 s
4	0.237 s	0.365 s	0.229 s
6	0.238 s	0.358 s	0.232 s
8	0.240 s	0.363 s	0.241 s
10	0.245 s	0.361 s	0.224 s
12	0.238 s	0.359 s	0.245 s
14	0.235 s	0.357 s	0.243 s
15	0.237 s	0.367 s	0.228 s

Os resultados obtidos para o tempo de execução do programa com a versão 2 do processo de quantização estão presentes na seguinte tabela:

Tabela 2: Tempo de Execução do programa wav_quant para a versão 2

num_bits	*Áudio Número 1*	*Áudio Número 2*	*Áudio Número 3*
1	0.235 s	0.361 s	0.238 s
2	0.241 s	0.368 s	0.246 s
4	0.241 s	0.370 s	0.234 s

6	$0.240\ s$	$0.353\ s$	$0.228\ s$
8	$0.245\ s$	$0.358\ s$	$0.236\ s$
10	$0.239\ s$	$0.368\ s$	$0.228\ s$
12	$0.238\ s$	$0.362\ s$	$0.244\ s$
14	$0.242\ s$	$0.357\ s$	$0.229\ s$
15	$0.239\ s$	$0.365\ s$	$0.231\ s$

```
Áudio Número 1: sample.wav
Áudio Número 2: sample2.wav
Áudio Número 3: sample3.wav
```

Os resultados apresentados foram cálculos através da média de cinco ensaios independentes para cada experiência.

Como podemos observar pelos resultados demonstrados na tabela acima, o tempo de execução do programa não foi afetado pelo número de bits utilizado no processo de quantização, nem pela versão do processo de quantização escolhida em nenhum dos três ficheiros de áudio utilizados.

Os ficheiros de áudio quantizados produzidos durante esta experiência foram guardados para utilização no teste do programa *wav_cmp.cpp.*

4.3. wav_cmp.cpp

O programa wav_cmp.cpp deve ser utilizado para efetuar o cálculo do valor do signal-to-noise ratio (SNR) entre um ficheiro de áudio e o ficheiro original.

Para avaliar a performance do programa efetuamos uma experiência com os ficheiros de áudio quantizado produzidos na secção anterior. Comparámos esses ficheiros com os áudios originais e registamos os valores de SNR e máximo erro absoluto por amostra retornados.

Os resultados obtidos demonstrados na seguinte tabela são referentes aos ficheiros de áudio quantizado com a versão 1 do processo de quantização e são os valores de SNR registados:

Tabela 3: Valores de SNR obtidos com a Versão 1 do processo quantização

num_bits	*Áudio Número 1*	*Áudio Número 2*	*Áudio Número 3*
1	$-2.12354\ dB$	$-10.3581\ dB$	$-8.93363\ dB$
2	$5.15654\ dB$	$-3.15304\ dB$	$-1.69889\ dB$
4	$17.2391\ dB$	$9.8751\ dB$	$11.9831\ dB$

6	29.2755 dB	22.0875 dB	24.2059 dB
8	41.3175 dB	34.3642 dB	36.2613 dB
10	53.3581 dB	46.6445 dB	48.3005 dB
12	65.3709 dB	58.8883 dB	60.3066 dB
14	76.9322 dB	70.6189 dB	71.8749 dB
15	81.7094 dB	75.4416 dB	76.6445 dB

Os resultados obtidos representados na tabela abaixo são referentes aos ficheiros de áudio quantizado com a versão 2 do processo de quantização e são os valores de SNR registados:

Tabela 4: Valores de SNR obtidos com a Versão 2 do processo de quantização

num_bits	*Áudio Número 1*	*Áudio Número 2*	*Áudio Número 3*
1	0.514778 dB	0.187292 dB	0.0988178 dB
2	5.17142 dB	1.78364 dB	2.5716 dB
4	17.2276 dB	11.7432 dB	12.3634 dB
6	29.276 dB	23.6832 dB	24.2311 dB
8	41.3258 dB	35.5661 dB	36.2596 dB
10	53.3631 dB	47.4483 dB	48.3001 dB
12	65.3662 dB	59.293 dB	60.3121 dB
14	76.9378 dB	70.7487 dB	71.8713 dB
15	81.7007dB	75.4805 dB	76.6441 dB

Através da análise dos valores de SNR representados em ambas as tabelas anteriores, podemos verificar que o valor de SNR diminui à medida que o número de bits utilizado para representar as amostras do áudio diminui também, visto que a informação que é perdida é maior, aumentando a quantidade de ruído introduzido no áudio e por isso diminuindo o valor de SNR.

Na comparação entre as duas versões de quantização utilizadas, podemos verificar que para valores de números de bits superiores a 4, as diferenças não são significativas. Contudo, para números de bits mais baixos, a versão 2 possui valores de SNR superiores, o que indica que introduz menos ruído no áudio. De relembrar que esta versão 2 do processo de quantização, é a

versão que tem em conta o valor de 0 de amplitude. Provavelmente esta redução na introdução de ruído, está relacionada com o facto de manter o áudio em silêncio (amplitude 0) quando no áudio original existe silêncio também.

Os resultados obtidos demonstrados na seguinte tabela são referentes aos ficheiros de áudio quantizado com a versão 1 do processo de quantização e são os valores do máximo erro absoluto por amostra registados:

Tabela 5: Valores do Máximo Erro Absoluto por Amostra com a Versão 1 do processo de quantização

num_bits	*Áudio Número 1*	*Áudio Número 2*	*Áudio Número 3*
1	16384	16384	16384
2	8192	8192	8192
4	2048	2048	2048
6	512	512	512
8	128	128	128
10	32	32	32
12	8	8	8
14	2	2	2
15	1	1	1

Os resultados obtidos demonstrados na seguinte tabela são referentes aos ficheiros de áudio quantizado com a versão 2 do processo de quantização e são os valores do máximo erro absoluto por amostra registados:

Tabela 6: Valores do Máximo Erro Absoluto por Amostra com a Versão 2 do processo de quantização

num_bits	*Áudio Número 1*	*Áudio Número 2*	*Áudio Número 3*
1	32767	32767	32059
2	16383	16383	15675
4	4095	4095	3387
6	1023	1023	512
8	255	255	128
10	63	63	32

12	15	15	8
14	3	3	2
15	1	1	1

Através da análise dos valores de máximo erro absoluto representados em ambas as tabelas anteriores, podemos verificar que a versão 2 apresenta um maior erro absoluto devido ao facto de que nesta versão o último intervalo de valores quantizados é superior aos restantes intervalos, algo que não acontece na versão 1 em que todos os intervalos são iguais.

4.4. wav_effects.cpp

Em termos dos resultados obtidos no wav_effects, para o modo SINGLE foram obtidos bons resultados no eco sendo este bem claro. Para o modo MULTIPLE foi obtido o resultado esperado, isto é, o som tornou-se uma confusão após algum tempo devido à natureza de realimentação deste tipo de eco. Já para o modo TIME_VARYING, os resultados obtidos não foram muito satisfatórios pois não se consegue notar que o eco é de intervalo variável, pois apenas usamos as primeiras 200 amostras da deque o que efetivamente faz com que a diferença entre os diferentes atrasos seja de no máximo 4.5 milissegundos, para uma frequência de amostragem de 44100 Hz. Finalmente, para o modo AMPLITUDE_MOD, os resultados são satisfatórios pois podemos realmente ouvir o volume/amplitude a variar com a frequência colocada de 1.5 Hz, ou seja, a variar de um máximo a outro após 1.5 segundos.

4.5. BitStream.cpp

Com o objetivo de verificar se a BitStream funciona corretamente, decidimos criar um programa em que é possível testar ambos os modos de utilização. Deste modo, começamos por executar o programa BitStream.cpp com o modo de leitura para inicializar a BitStream no modo leitura com um ficheiro em que o conteúdo era “Hello World!”, e o objetivo era saber se o ficheiro foi lido corretamente. Neste teste, o programa iria ler bit a bit através da função get_bit() da BitStream, e iria escrever os bits que ia recebendo para um ficheiro. No final, o esperado era que o ficheiro que o programa escreveu, seria igual ao ficheiro que foi passado à BitStream.

```
Comando Executado:
./BitStream -r output_binary_data ../examples/data
```

Os resultados que obtivemos foram um sucesso uma vez que o ficheiro que o programa escreveu a partir dos bits que recebeu da BitStream foi exatamente o mesmo que o ficheiro introduzido na BitStream.

Por forma a testar o modo escrita, executamos o programa BitStream.cpp com o modo escrita para inicializar a BitStream no modo escrita com um ficheiro. Neste teste, o programa iria ler o conteúdo de um ficheiro bit a bit em que o conteúdo era “Hello World!”, e para cada bit invocava a função write_bit() para a BitStream escrever esse bit no ficheiro associado. No

final era esperado que o conteúdo do ficheiro que a BitStream criou seria igual ao conteúdo do ficheiro que o programa leu.

```
Comando Executado:
./BitStream -w ../examples/data output_binary_data
```

4.6. wav_dct_codec.cpp e wav_dct_decoder.cpp

Por último, para testar os nossos programas wav_dct_codec.cpp e wav_dct_decoder.cpp, decidimos verificar a taxa de conversão para cada um dos ficheiros de áudio de exemplo variando a fração de valores de amplitude considerados relevantes. Os resultados estão apresentados na seguinte tabela:

Os ficheiros de áudio têm originalmente os seguintes tamanhos:

- *Áudio Número 1*: 2116844 bytes.
- *Áudio Número 2*: 6257742 bytes.
- Áudio Número 3: 10406738 bytes.

Tabela 7: Taxas de compressão com diferentes frações para cada áudio

fração	*Áudio Número 1*	*Áudio Número 2*	*Áudio Número 3*
0.1	$213024\ bytes\ (-89.9\%)$	$629556\ bytes\ (-89.9\%)$	$1046912\ bytes\ (-89.9\%)$
0.2	$423960\ bytes\ (-79.9\%)$	$1252980\ bytes\ (-79.9\%)$	$2083640\ bytes\ (-79.9\%)$
0.3	$636964\ bytes\ (-69.9\%)$	$1882516\ bytes\ (-69.9\%)$	$3130532\ bytes\ (-69.9\%)$
0.4	$847900\ bytes\ (-59.9\%)$	$2505940\ bytes\ (-59.9\%)$	$4167260\ bytes\ (-59.9\%)$
0.5	$1058836\ bytes\ (-49.9\%)$	$3129364\ bytes\ (-49.9\%)$	$5203988\ bytes\ (-49.9\%)$
0.7	$1482776\ bytes\ (-29.9\%)$	$4382324\ bytes\ (-29.9\%)$	$7287608\ bytes\ (-29.9\%)$
0.9	$1906716\ bytes\ (-9.9\%)$	$5635284\ bytes\ (-9.9\%)$	$9371228\ bytes\ (-9.9\%)$

Analisando o tamanho dos ficheiros comprimidos, conseguimos verificar que os mesmos foram comprimidos com as taxas de compressão esperadas, ou seja, quando utilizamos uma fração de 0.1, apenas as 10% frequências mais relevantes por bloco do áudio são mantidas pelo que a taxa de compressão vai ser de aproximadamente 90%.

Relativamente à qualidade do áudio, esta só é possível ser medida através do ouvido humano, pelo que à medida que o valor de fração diminui, a qualidade do áudio vai piorando, contudo, dá para perceber à mesma (com menos qualidade) a maior parte do conteúdo do áudio o que é um resultado bastante bom, uma vez que reduzir o tamanho de um ficheiro em 90%, e conseguir manter as amplitudes de áudio mais criticas (mais importantes) torna este programa bastante interessante e com diversos casos de uso possíveis.

5. Conclusão

A realização deste projeto permitiu-nos colocar em prática os conhecimentos obtidos nas aulas teóricas e aplicar procedimentos utilizados no processamento de sinal de áudio, tais como a análise de canais do áudio através de histogramas, a redução do número de bits utilizados para representar as amostras através da quantização escalar uniforme, cálculo do valor de SNR e ainda a aplicação de efeitos sonoros no áudio.

Adicionalmente, um aspeto que consideramos bastante vantajoso foi termos desenvolvido o projeto com a linguagem de programação C++, especialmente na criação e manipulação de ficheiros ao nível do bit, como fizemos na classe BitStream, e por último termos aplicado a Discrete Cosine Transform num contexto útil como a compressão de ficheiros de áudio mantendo as frequências mais relevantes ao ouvido humano.

Concluindo, na nossa opinião este projeto foi um desafio realizado com sucesso pelo que foi útil para o desenvolvimento das nossas competências individuais.