

Universidade de Aveiro

# Informação e Codificação



Bruno Lemos 98221, Tiago Marques 98459, João  
Viegas 98372

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>1</b>
<b>2</b>	<b>Parte I</b>	<b>3</b>
2.1	Exercício 1 - Análise Inicial . . . . .	3
2.2	Exercício 2 - Histograma . . . . .	3
2.2.1	Argumentos . . . . .	3
2.2.2	Algoritmo . . . . .	4
2.2.3	Resultados . . . . .	4
2.3	Exercício 3 - Quantização . . . . .	5
2.3.1	Argumentos e Requisitos . . . . .	6
2.3.2	Resultados . . . . .	6
2.4	Exercício 4 - Signal to noise [SNR] . . . . .	7
2.4.1	Argumentos e Requisitos . . . . .	8
2.4.2	Resultados . . . . .	8
2.5	Exercício 5 - Efeitos de áudio . . . . .	9
2.5.1	Reverse . . . . .	9
2.5.2	Mute Left e Right . . . . .	10
2.5.3	Eco . . . . .	10
2.5.4	Amplitude Loop . . . . .	11
<b>3</b>	<b>Parte II</b>	<b>13</b>
3.1	Exercício 6 - Leitura/Escrita de bits . . . . .	13
3.1.1	Ler um bit . . . . .	13
3.1.2	Escrever um bit . . . . .	14
3.1.3	Ler N bits . . . . .	14
3.1.4	Escrever N bits . . . . .	15
3.2	Exercício 7 -Encoder/Decoder . . . . .	16
3.2.1	Encoder Decoder . . . . .	16
<b>4</b>	<b>Parte III</b>	<b>17</b>
4.1	Exercício 8 - Discrete Cosine Transform [DCT] . . . . .	17
4.1.1	Argumentos e Requisitos . . . . .	18
4.1.2	Resultados . . . . .	19

# Lista de Figuras

2.1	Execução do exercício 2. Exemplo : samples.wav . . . . .	3
2.2	Exemplo do MID_channel.txt . . . . .	4
2.3	Resultado gráfico do Side Channel . . . . .	5
2.4	Resultado gráfico do Mid Channel . . . . .	5
2.5	Gráfico do sinal original . . . . .	6
2.6	Gráfico do sinal com 8 bits . . . . .	7
2.7	Gráfico do sinal com 4 bits . . . . .	7
2.8	Execução do exercício , Exemplo : samples.wav . . . . .	8
2.9	Valores do sinal reduzido para 8 bits . . . . .	8
2.10	Valores do sinal reduzido para 4 bits . . . . .	8
2.11	Explicação Algortimo reverse . . . . .	9
2.12	Áudio original do sample2.wav . . . . .	9
2.13	Áudio sample2.wav com reverse . . . . .	10
2.14	Áudio sample2.wav com single echo . . . . .	11
2.15	Áudio sample2.wav com triple echo . . . . .	11
2.16	Áudio após efeito Amplitude loop . . . . .	12
2.17	Função módulo de seno . . . . .	12
3.1	Conteúdo do ficheiro . . . . .	14
3.2	Representação do primeiro bit lido do ficheiro . . . . .	14
3.3	Representação do primeiro bit lido do ficheiro . . . . .	14
3.4	Representação dos bits do ficheiro escrito . . . . .	14
3.5	Representação do conteúdo do ficheiro . . . . .	15
3.6	Representação dos bits do 10 bits lidos do ficheiro . . . . .	15
3.7	Representação do conteúdo do ficheiro . . . . .	15
3.8	Representação do conteúdo lido do ficheiro de 8 em 8 bits . . . . .	15
3.9	Representação do conteúdo lido do ficheiro de 10 em 10 bits . . . . .	16
4.1	Bits necessários para representar coeficientes inteiros <i>DCT</i> . . . . .	17
4.2	Fluxo das operações realizadas sobre os coeficientes <i>DCT</i> até codificação . . . . .	18
4.3	Execução do exercício 8 . . . . .	18
4.4	Gráfico do sinal original . . . . .	19
4.5	Gráfico do sinal de coeficientes DCT . . . . .	19
4.6	Valores do <i>SNR</i> e erro máximo absoluto . . . . .	19

# Capítulo 1

## Introdução

O presente relatório visa descrever a resolução do Projeto 1 desenvolvido no âmbito da unidade curricular de Informação e Codificação.

O código desenvolvido para o projeto encontra-se disponível em : [https://github.com/brunolemos06/IC\\_Project1](https://github.com/brunolemos06/IC_Project1).

Para este projeto foram utilizadas algumas bibliotecas como cmath, map e algorithm.

Para executar os comandos seguintes é necessário estar no diretório:

```
IC_Project_1/sndfile_example_src
```

Para compilar todos os programas

```
make all
```

Para eliminar os executáveis

```
make clean
```

Executar o exercício 1

```
../sndfile_example_bin/wav_cp <-v(verbose)> <audio.wav> <out.wav> Caso  
'-v' : remete as informação das frames, samples/s e número de channels.
```

Executar o exercício 2

```
../sndfile_example_bin/wav_hist <audio.wav> <channel>
```

Executar o exercício 3

```
../sndfile_example_bin/wav_quant <audio.wav> <out.wav> <bits_to_cut>
```

Executar o exercício 4

```
../sndfile_example_bin/wav_cmp <audio.wav> <out.wav>
```

Executar o exercício 5

```
../sndfile_example_bin/wav_effects <audio.wav> <out.wav> <effect>
```

Executar o exercício 6

```
../sndfile_example_bin/BitStream
```

Executar o exercício 7

```
../sndfile_example_bin/wav_encoder_decoder <file_to_encode.txt> <file_decoded.txt>
```

Executar o exercício 8

```
../sndfile_example_bin/wav_dct <-v(verbose)> <-f(blocksize)> <-frac(dctFraction)>  
<audio.wav> <out.wav>
```

Importante notar que os exercícios compilados de cada programa são guardados na pasta **sndfile-example-bin** e o código fonte na pasta **sndfile-example-src**.

# Capítulo 2

## Parte I

### 2.1 Exercício 1 - Análise Inicial

Neste exercício através do elearning instalámos o software *sndfile-example.tar.gaz*. Foi neste exercício onde tivemos o primeiro contacto com o projeto. Analisámos os ficheiros a baixo para podermos continuar o projeto.

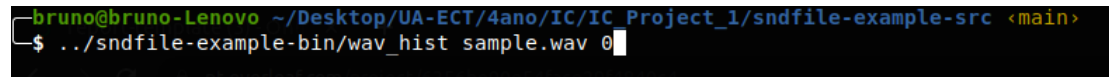
- `wav_cp` : Ficheiro que faz uma cópia de um ficheiro de áudio.
- `wav_hist` : Ficheiro original cria o histograma de um ficheiro de áudio.

### 2.2 Exercício 2 - Histograma

Neste exercício foi nos proposto alterar a classe *wav\_hist* para fornecer um histograma das médias dos canais. Foram criados dois histogramas: MID Channel e SIDE Channel.

#### 2.2.1 Argumentos

Os argumentos necessários para executar o programa é um ficheiro de audio do género *\*.wav*, no caso usámos o *samples.wav* como se verifica na figura a baixo. Também é necessário o número do canal.

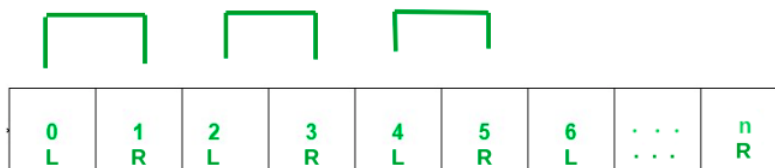
A terminal window with a black background and green text. The prompt is 'bruno@bruno-Lenovo' followed by the path '~/Desktop/UA-ECT/4ano/IC/IC Project\_1/sndfile-example-src' and the shell '<main>'. The command entered is '\$ ../sndfile-example-bin/wav\_hist sample.wav 0' followed by a cursor.

```
bruno@bruno-Lenovo ~/Desktop/UA-ECT/4ano/IC/IC Project_1/sndfile-example-src <main>
$ ../sndfile-example-bin/wav_hist sample.wav 0
```

Figura 2.1: Execução do exercício 2. Exemplo : *samples.wav*

### 2.2.2 Algoritmo

Num ciclo é percorrido as samples de duas em duas iterações e é efetuado a contagem de quantas vezes ocorreu certo valor de acordo com o histograma:



```
Mid Channel -> MidChannel[ ( L + R ) / 2 ] ++  
Side Channel -> SideChannel[ ( L - R ) / 2 ] ++
```

### 2.2.3 Resultados

Os resultados obtidos são dois ficheiros , Mid\_Channel.txt e Side\_Channel.txt, onde em cada linha desses ficheiros podemos observar que está associado a contagem desse elemento,  $(L + R) / 2$  e  $(L - R) / 2$  respetivamente por cada ficheiro.

Por exemplo:

```
12492 7  
12493 4  
12494 10  
12495 11  
12496 7  
12497 5  
12498 7  
12499 8  
12500 10  
12501 6  
12502 9  
12503 5  
12504 5  
12505 4  
12506 9  
12507 7  
12508 6  
12509 4
```

Figura 2.2: Exemplo do MID\_channel.txt

Para uma visualização mais gráfica usámos Gnuplot, um programa em linha de comando que através de um conjunto de dados que a class wavhist produziu cria um gráfico.

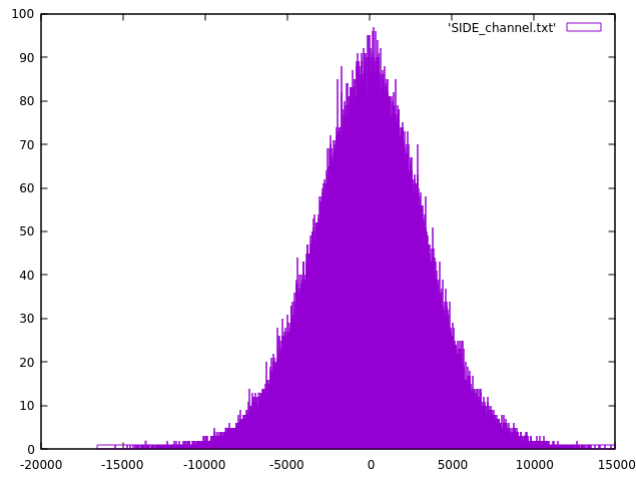


Figura 2.3: Resultado gráfico do Side Channel

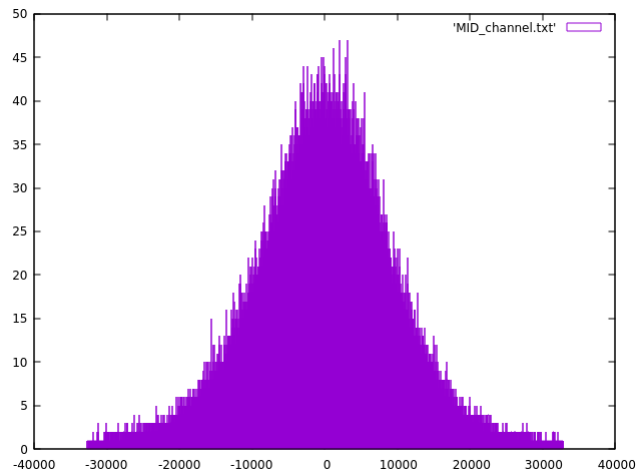


Figura 2.4: Resultado gráfico do Mid Channel

## 2.3 Exercício 3 - Quantização

Para este exercício foi nos proposto a implementação de uma classe *wav\_quant* e um programa associado que fizesse uso da mesma. Esta classe tem como objetivo reduzir o número de bits usados para representar cada *sample* de um ficheiro de áudio, usando quantização.

Em termos de processamento de sinais, o processo de quantização representa a



transformação de um sinal contínuo num sinal discreto. Realizar esta técnica de compressão de informação implica perda de informação, pelo que é impossível reverter o processo e assim sendo torna inviável reconstruir o sinal original.

### 2.3.1 Argumentos e Requisitos

Os argumentos necessários para executar o programa é um ficheiro de audio do género *\*.wav* em formato *PCM16*, o nome do ficheiro de saída (terminado em *\*.wav*) e o número de bits a reduzir, sendo que o número de bits pertence a [1, 15].

### 2.3.2 Resultados

Ao fazer uso do programa e da classe desenvolvidos obtemos um ficheiro de áudio (*\*.wav*) cujo sinal é o resultado da quantização do sinal do ficheiro de áudio original.

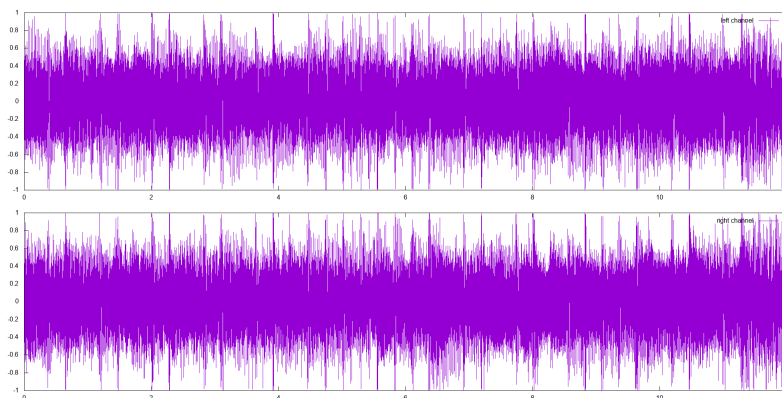


Figura 2.5: Gráfico do sinal original

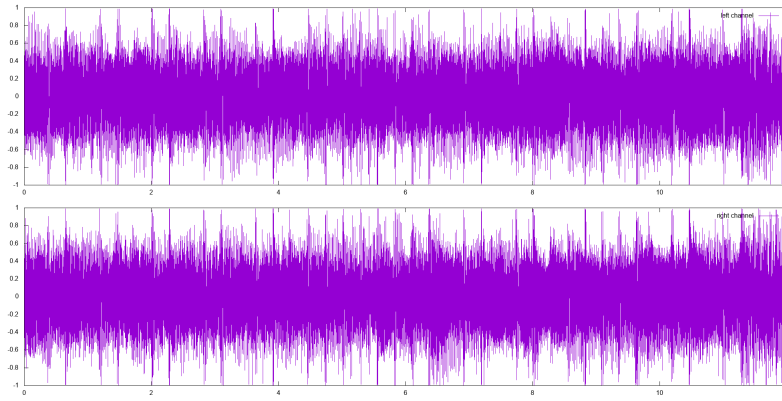


Figura 2.6: Gráfico do sinal com 8 bits

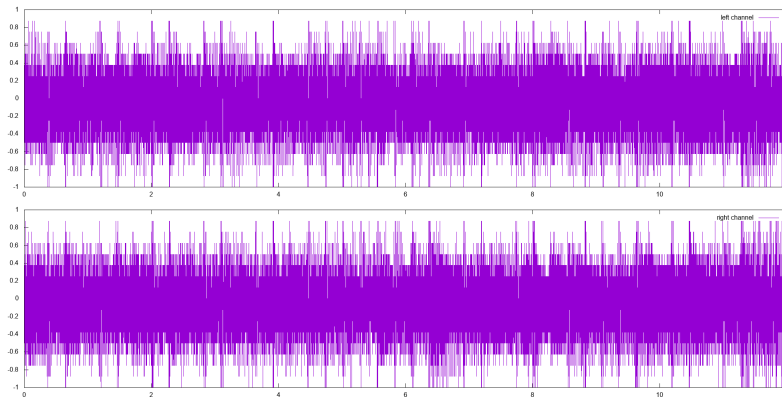


Figura 2.7: Gráfico do sinal com 4 bits

## 2.4 Exercício 4 - Signal to noise [SNR]

Como a técnica de quantização introduz erros é importante medir a quantidade de ruído introduzida. Para medir o *signal-to-noise ratio* usou-se a seguinte fórmula:

$$SNR = 10 \cdot \log_{10} \left( \frac{S}{D} \right)$$

Para fazer uso da fórmula acima, previamente calculamos a distorção usando:

$$D = \frac{1}{N} \cdot \sum_{n=1}^N (x_n - \hat{x}_n)^2$$

E concorrentemente calculamos o poder do sinal original usando:

$$S = \frac{1}{N} \cdot \sum_{n=1}^N (x_n)^2$$

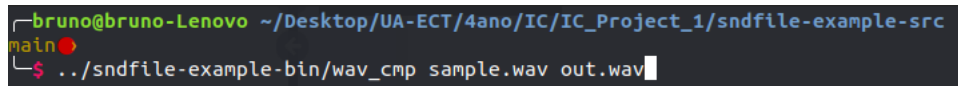
Adicionalmente o erro máximo absoluto foi calculado enquanto estavam a ser percorridas todas as *samples* sequencialmente, para o cálculo de D e de S. Foi usada a seguinte fórmula:

$$Err_{max} = |x_n - \hat{x}_n|$$

Durante a leitura sequencial das *samples* havia uma variável *maxError* que armazenava o valor do erro máximo absoluto caso necessário.

### 2.4.1 Argumentos e Requisitos

Os argumentos necessários para executar o programa são dois ficheiros de áudio do género *\*.wav*, dos quais um seja 'original' e o outro uma versão comprimida do 'original'. Ambos os ficheiros têm de possuir o mesmo número de *samples* e estar em formato *PCM16*.

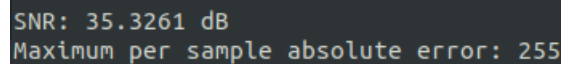


```
bruno@bruno-Lenovo ~/Desktop/UA-ECT/4ano/IC/IC_Project_1/sndfile-example-src
main
$ ./sndfile-example-bin/wav_cmp sample.wav out.wav
```

Figura 2.8: Execução do exercício , Exemplo : samples.wav

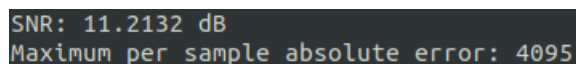
### 2.4.2 Resultados

Usamos os ficheiros de áudio obtidos nos resultados do exercício 3 e verificamos o seu *signal-to-noise ratio* e erro máximo absoluto.



```
SNR: 35.3261 dB
Maximum per sample absolute error: 255
```

Figura 2.9: Valores do sinal reduzido para 8 bits



```
SNR: 11.2132 dB
Maximum per sample absolute error: 4095
```

Figura 2.10: Valores do sinal reduzido para 4 bits

Ao observar os resultados obtidos podemos concluir que o valor de SNR é inversamente proporcional ao ruído de um ficheiro de áudio e que o erro máximo absoluto aumenta conforme o ruído introduzido.

## 2.5 Exercício 5 - Efeitos de áudio

Neste exercício foi nos proposto implementar um programa designado *wav\_effects* que produz alguns efeitos de áudio. No nosso programa estão disponíveis os seguintes efeitos descritos nas subsecções abaixo.

O output deste programa é um ficheiro de áudio de acordo com os efeito disponíveis no nosso programa e descrito nos parâmetros de entrada.

### 2.5.1 Reverse

Este efeito tem o propósito de produzir o ficheiro de áudio original de trás para a frente.

Para tal foi necessário inverter as posições das samples em grupos de 2 elementos tal como mostra na figura a baixo,

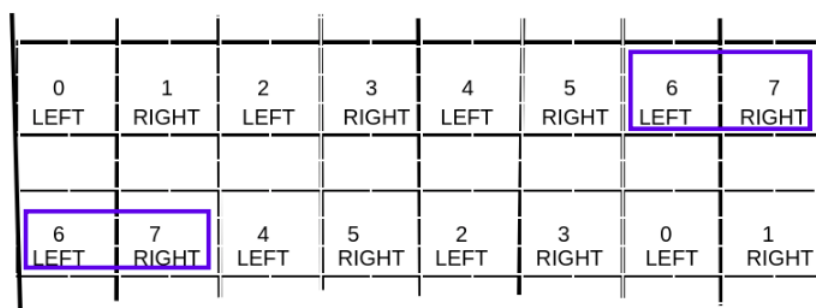


Figura 2.11: Explicação Algoritmo reverse

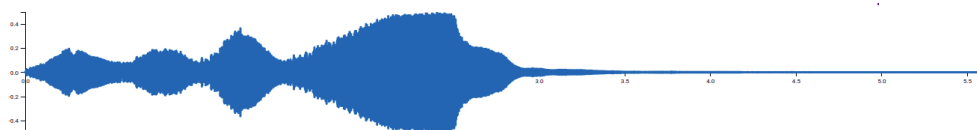


Figura 2.12: Áudio original do sample2.wav

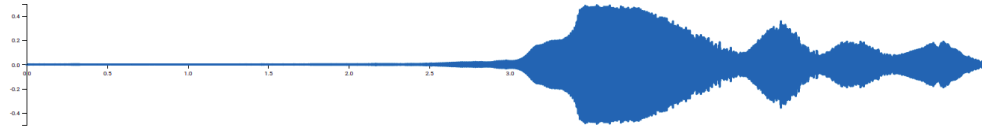


Figura 2.13: Áudio sample2.wav com reverse

### 2.5.2 Mute Left e Right

Este efeito produz som apenas de um dos lados, ou do lado esquerdo ou do direito. Para tal é necessário mudar o valor da sample para 0 de acordo com o canal.

É inicializado um counter que ao longo do programa vai aumentando variando entre número par e ímpar o que nos dá a possibilidade de fazer comutação entre o valor '0' e o `samples[index_atual]`.

- Left

$$samples[i] = (value \% 2) * samples[i]$$

L	R	L	R	L	R	L	R
<code>samples[0]</code>	0	<code>samples[2]</code>	0	<code>samples[4]</code>	...	<code>samples[2n]</code>	0

- Right

$$samples[i] = (value + \% 2) * samples[i]$$

L	R	L	R	L	R	L	R
0	<code>samples[1]</code>	0	<code>samples[3]</code>	0	...	0	<code>samples[2n-1]</code>

### 2.5.3 Eco

O eco caracteriza-se quando o som refletido chega aos nossos ouvidos após um determinado tempo, logo podemos perceber distintamente o som emitido e o som refletido.

No nosso programa realizámos múltiplos ecos, tal como único eco, duplo eco e triplo echo.

#### Single Echo:

$$samples[i] = (samples[i] + samples[i - delay]) * \alpha$$



Figura 2.14: Áudio sample2.wav com single echo

#### Double Echo:

$$samples[i] = (samples[i] + samples[i - delay] + samples[i - delay * 2]) * \alpha$$

#### Triple Echo:

$$samples[i] = (samples[i] + samples[i - delay] + samples[i - delay * 2] + samples[i - delay * 3]) * \alpha$$



Figura 2.15: Áudio sample2.wav com triple echo

#### N Echos:

$$samples[i] = \alpha \sum_{k=0}^N [samples[i - delay * N]]$$

### 2.5.4 Amplitude Loop

Este efeito produz um som em que a amplitude é uma função módulo sinusoidal como podemos ver nas figuras mais a baixo.

$$samples[i] = samples[i] * \alpha : \alpha \in [0, 1]$$



Figura 2.16: Áudio após efeito Amplitude loop

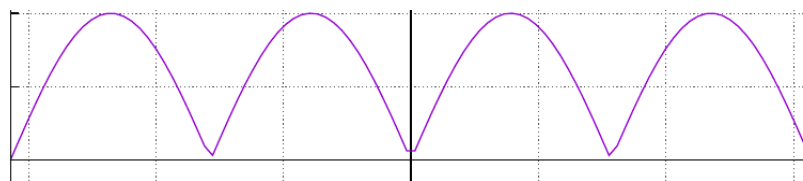


Figura 2.17: Função módulo de seno

Nesta Função sinusoidal, Y é a amplitude sonora e X o tempo.

## Capítulo 3

# Parte II

Descreve os resultados obtidos.

### 3.1 Exercício 6 - Leitura/Escreita de bits

Neste exercício o objetivo era criar uma classe bitstream com métodos de escrita e leitura de bits de um para outro ficheiro.

#### 3.1.1 Ler um bit

Começamos por fazer um método para ler 1 bit do ficheiro. Para ler o primeiro bit do ficheiro, nós lemos o primeiro byte completo para um vetor bitset. Posteriormente guardamos os valores convertidos em string numa variável e escrevemos o primeiro character da string, que corresponde ao bit mais significativo, no ficheiro de output.

#### Teste

Para testar o método da leitura de um bit de um ficheiro optamos por ler o primeiro byte do ficheiro e depois representar num outro ficheiro o bit mais significativo desse byte. Como podemos ver nas figuras a baixo, o ficheiro de onde é lida a informação está o seguinte byte '00100011'. Como o nosso computador consegue representar este conjunto de bits em ascii, então é representado um '#'.

Para testar o método, inicializamos a classe com:

```
BitStream bs("test.txt", 'r');
```

Chamamos o método para leitura de bits:

```
bs.readbit();"
```



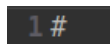


Figura 3.1: Conteúdo do ficheiro

Na figura 3.2 está representado o caracter equivalente ao bit mais significativo do byte lido.

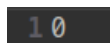


Figura 3.2: Representação do primeiro bit lido do ficheiro

### 3.1.2 Escrever um bit

Para escrever um bit no ficheiro, lê-mos um caracter com valor 0 ou 1 de um ficheiro e escrevemos um bit com o valor do caracter no ficheiro.

#### Teste

Para testar se o método de escrita funcionava corretamente, lê-mos o caracter '0' ou '1' do ficheiro e num ciclo escrevemos 8 vezes esse bit no ficheiro. Desta forma, se o caracter do ficheiro for " " como não existe uma representação gráfica em ascii para o valor "11111111" então vai ser apresentado em hexadecimal, ou seja, 'FF'.

Para testar o método, inicializamos a classe com:

```
BitStream bs("test.txt", 'w');
```

Chamamos o método para leitura de bits:

```
for(int i = 0; i < 8; i++){bs.writebit("decoder.txt");}"
```

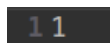


Figura 3.3: Representação do primeiro bit lido do ficheiro

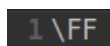


Figura 3.4: Representação dos bits do ficheiro escrito

### 3.1.3 Ler N bits

Para ler N bits de um ficheiro, sabemos que o ficheiro têm que ser lido byte a 1 byte. Se N não for múltiplo de 8, temos de calcular quantos bits temos de ler do ultimo byte para assim poder representar o numero de bits pedidos.

### Teste

Para testar o método para ler  $n$  bits utilizamos o seguinte método. Num ficheiro com conteúdo "001000110010001100100011" que são representados no ficheiro com "#", lê-mos apenas os primeiros 10 primeiros bits, ou seja, "0010001100". Para testar o método, inicializamos a classe com:

```
BitStream bs("test.txt", 'r');
```

Chamamos o método para leitura de bits:

```
bs.readNbits(10);
```

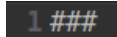


Figura 3.5: Representação do conteúdo do ficheiro

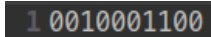


Figura 3.6: Representação dos bits do 10 bits lidos do ficheiro

### 3.1.4 Escrever $N$ bits

Para escrever  $n$  bits, criámos um método que lê de um ficheiro os caracteres com valores 0 ou 1 para um vetor de bits e escrevemos no ficheiro o vetor de  $n$  em  $n$  bits do vetor para o ficheiro.

### Teste

Para testar o método de escrever no ficheiro de  $n$  em  $n$  bits podemos ver as diferenças de output da escrita entre 8 bits e 10 bits. Para testar o método, inicializamos a classe com:

```
BitStream bs("decoder.txt", 'r');
```

Chamamos o método para leitura de bits:

```
bs.writeNbits("test.txt", 8);
```

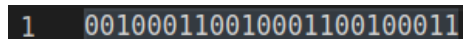


Figura 3.7: Representação do conteúdo do ficheiro

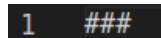


Figura 3.8: Representação do conteúdo lido do ficheiro de 8 em 8 bits



Figura 3.9: Representação do conteúdo lido do ficheiro de 10 em 10 bits

## 3.2 Exercício 7 -Encoder/Decoder

Neste exercício, o objetivo era fazer um programa *Encoder* e *Decoder* para converter um ficheiro de texto com zeros e uns num ficheiro com o seu binário equivalente e realizar o processo inverso de volta a obter o conteúdo do ficheiro original.

### 3.2.1 Encoder Decoder

Na execução do programa é iniciado o processo de *encode* que lê de um ficheiro os caracteres com valores de zeros ou uns, agrupa esses caracteres em grupos de 8, formando assim um byte, e escreve esses novos bytes obtidos num ficheiro chamado `encodedFile.txt`. Na eventualidade de o numero de bits não for múltiplo de oito, chamamos a função *flush*, onde o ultimo byte a ser escrito é preenchido com bits a zero nas posições mais significativas do byte.

Na parte do *decoder*, programa converte um ficheiro com *bytes* em caracteres equivalentes de zeros ou uns. Para converter os *bytes* em string, lê-mos o ficheiro byte a byte e cada um dos bytes são passados a valores inteiros dentro de um *bitset*. Posteriormente transforma-mos o *bitset* de inteiros em numa *string* binária que por sua vez irá ser escrita no ficheiro final.

#### Teste

Para testar o programa, num ficheiro com conteúdo "0010001100100011001000110010001100100011" depois de executado a parte do programa correspondendo à parte do *encode* é gerado um ficheiro intermédio com "#####". Este ficheiro adquire esta representação porque os bits escritos conseguem ser interpretados em caracteres ascii pelo computador. Na realização do processo de *decode* passamos um ficheiro com o conteúdo de "#####", aplicamos o processo de *decode* no qual obtemos noutra ficheiro os caracteres de zeros e uns com os seguintes valores: "0010001100100011001000110010001100100011", iguais aos valores do ficheiro original.

## Capítulo 4

## Parte III

### 4.1 Exercício 8 - Discrete Cosine Transform [DCT]

Neste exercício é nos pedido a implementação de um *codec* de áudio com perda baseado na *Discrete Cosine Transform DCT*.

Para uma implementação correta deste *codec*, em primeiro lugar, precisamos processar o áudio de bloco a bloco e posteriormente cada bloco processado irá a ser convertido em coeficientes usando o *DCT*.

Ao termos todos os blocos já processados e convertidos é preciso quantizar os coeficientes resultantes, a nossa solução passou por uma conversão de tipos na linguagem *C++*, de *double*, tipo original dos coeficientes com casas decimais, para *int*, fazendo assim os valores dos coeficientes perderem a sua parte decimal. Na escrita dos valores dos coeficientes para um ficheiro de caracteres com valor '1' ou '0', houve a necessidade de criar uma norma para a escrita/leitura desses tais valores inteiros, assim como a necessidade de saber quantos bits eram necessários para os representar.

Para resolver a questão do número de bits para representação dos valores inteiros, percorremos os valores inteiros do *DCT* até encontrar o maior valor positivo e menor valor negativo e calculamos o número de bits necessários para representar essa gama de valores.

```
min_value: -1642
bits needed to represent min_value: 11
max_value: 2126
bits needed to represent max_value: 12
bits needed to represent [min_value, max_value]: 13
max bin: 0100001001000
min bin: 1100110010110
```

Figura 4.1: Bits necessários para representar coeficientes inteiros *DCT*

Como podemos observar iremos precisar de 13 bits para representar os coeficientes quantizados.

Para a criação da norma para a escrita/leitura dos valores decidimos escrever primeiro os bits, neste ficheiro inicial cada bit é um carácter, do coeficiente do canal esquerdo e depois os bits do coeficiente do canal direito, alternando assim

até termos todos os coeficientes *DCT* escritos. Depois da criação deste ficheiro, com caracteres de '0's e '1's, este ficheiro é dado a classe BitStream que com a função *encoder* lê o ficheiro e faz a codificação dos bits presentes no ficheiro inicial para um novo ficheiro onde cada carácter representa o valor de 8 bits, ex:00100011 passa a #.

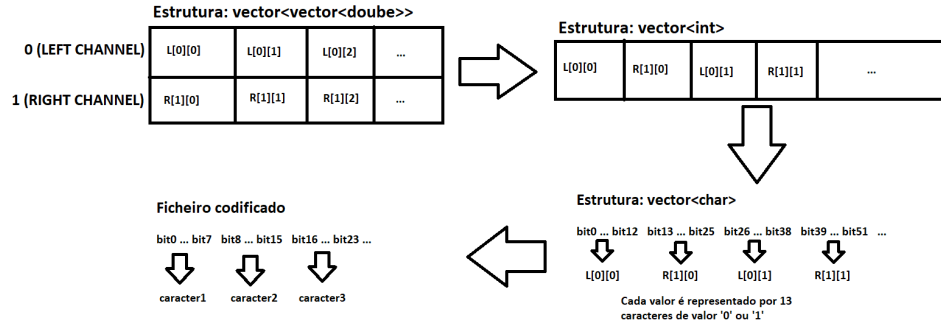


Figura 4.2: Fluxo das operações realizadas sobre os coeficientes *DCT* até codificação

Esse ficheiro codificado é agora alimentado a função *decoder*, também da classe BitStream, que por sua vez, faz o inverso da função *encoder*. Este novo ficheiro decodificado, com caracteres de zeros e uns, é lido pelo programa principal, que agrupada os caracteres em grupos de 13 formando uma *string* binária e converte-a para o seu equivalente em número inteiro. Já com todos os valores inteiros processados estes serão colocados na estrutura original, vetor de vetor, é calculado o *DCT* Inverso que, por sua vez, resulta no output de um ficheiro de áudio construído a partir dos coeficientes quantizados iniciais \*.wav.

#### 4.1.1 Argumentos e Requisitos

Os argumentos necessários para executar o programa é um ficheiro de áudio do género \*.wav em formato *PCM16* e o nome do ficheiro de saída (terminado em \*.wav).

```
bruno@bruno-Lenovo ~/Desktop/UA-ECT/4ano/IC/IC_Project_1/sndfile-example-src :main
$ ../sndfile-example-bin/wav_dct -v -b 512 -freq 0.4 sample.wav out.wav
```

Figura 4.3: Execução do exercício 8

### 4.1.2 Resultados

Usamos o ficheiro de áudio obtido na execução do programa e comparamo-lo ao seu original.

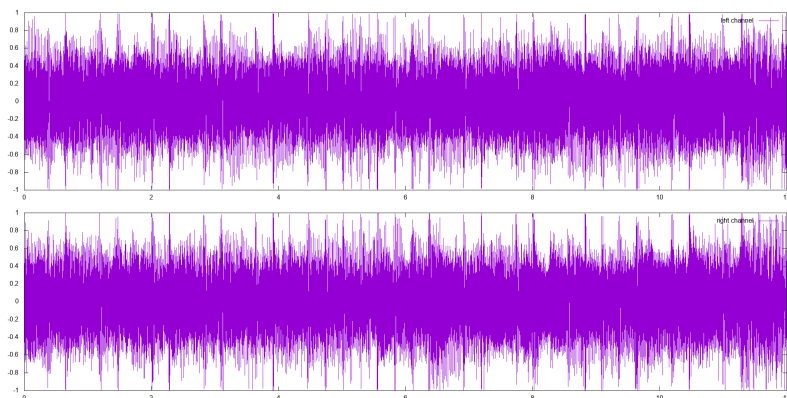


Figura 4.4: Gráfico do sinal original

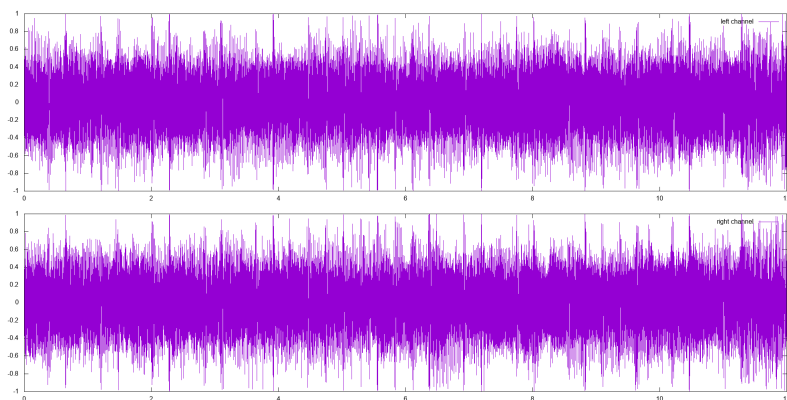


Figura 4.5: Gráfico do sinal de coeficientes DCT

Podemos observar pelo comparação dos gráfico dos sinais, que o sinal resultante do programa apesar de possuir algumas diferenças, nenhuma é de grande discrepância. Por fim verificamos o seu *signal-to-noise ratio* e erro máximo absoluto.

```
SNR: 17.0321 dB  
Maximum per sample absolute error: 65153
```

Figura 4.6: Valores do *SNR* e erro máximo absoluto

# Contribuições dos autores

Todos os autores participaram de forma igual na divisão, desenvolvimento e discussão deste trabalho pelo que a percentagem de contribuição para cada aluno fica:

- Bruno Lemos - 33.3%
- Tiago Marques - 33.3%
- João Viegas - 33.3%