

Universidade de Aveiro

# Informação e Codificação

## Projeto 1



João Torrinhos (98435), Diogo Torrinhos (98440), Tiago Bastos  
(97590)

30 de outubro de 2022

# Introdução

Este relatório descreve a resolução do Projeto 1 desenvolvido no âmbito da unidade curricular de Informação e Codificação.

Para a realização do trabalho foi usado o GitHub.

O código desenvolvido para este projeto encontra-se disponível em: <https://github.com/diogotorrinhas/IC>

Os comandos para compilação e alguns resultados encontram-se no README disponível no GitHub e também aqui neste relatório.

## Exercício 2

Para correr o programa, referente a este exercício, basta correr a linha, no terminal:

```
../sndfile-example-bin/ex2 sample.wav Avg > hist.txt
```

```
../sndfile-example-bin/ex2 sample.wav Difference > histDiff.txt
```

Onde *ex2* é o programa referente a este exercício, *sample.wav* é a amostra que vai ser usada para calcular o histograma e o terceiro parâmetro de entrada é o modo como vai ser calculado o histograma. *Avg* é a média dos canais e *Difference* é a diferença dos canais. O resultado por sua vez é apresentado num ficheiro de texto, nos exemplos de cima é apresentado nos ficheiros *hist.txt* e *histDiff.txt*.

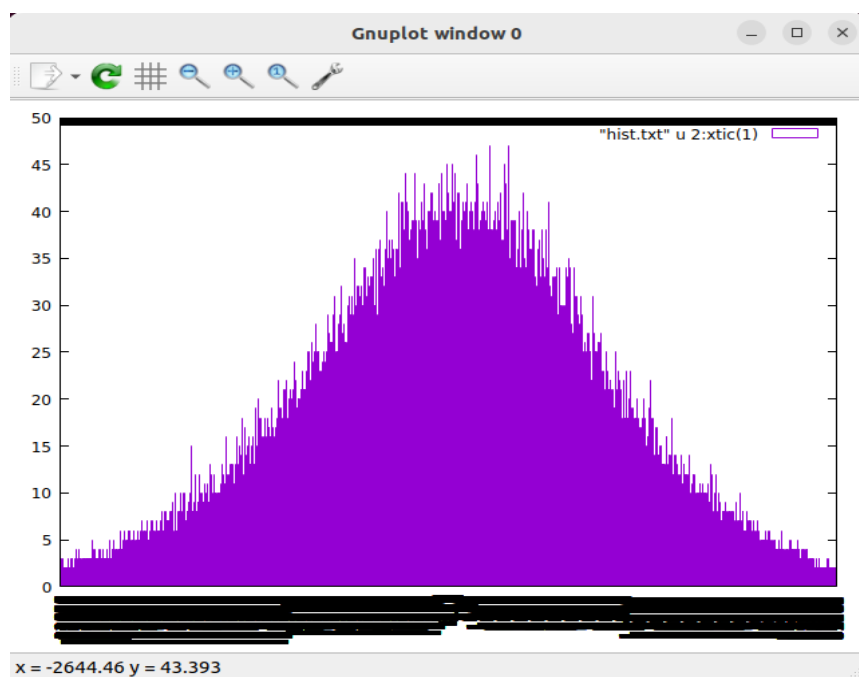
Com recurso à biblioteca *libsndfile* foi possível guardar a informação num vetor onde os índices pares referem-se ao primeiro canal e os índices ímpares referem-se ao segundo canal. Com esta informação criou-se uma estrutura de dados *map* onde as chaves correspondiam aos valores lidos do ficheiro e o valor associado a cada chave representa o número de ocorrências desses valores.

Para visualizar o gráfico dos histogramas usamos o gnuplot, na linha de comandos:

- `gnuplot -p -e 'set sty d hist;set xtic rot; plot "hist.txt" u 2:xtic(1)'`

- `gnuplot -p -e 'set sty d hist;set xtic rot; plot "histDiff.txt" u 2:xtic(1)'`

Obtendo os seguintes resultados:



**Figura1: Histograma para média de canais**

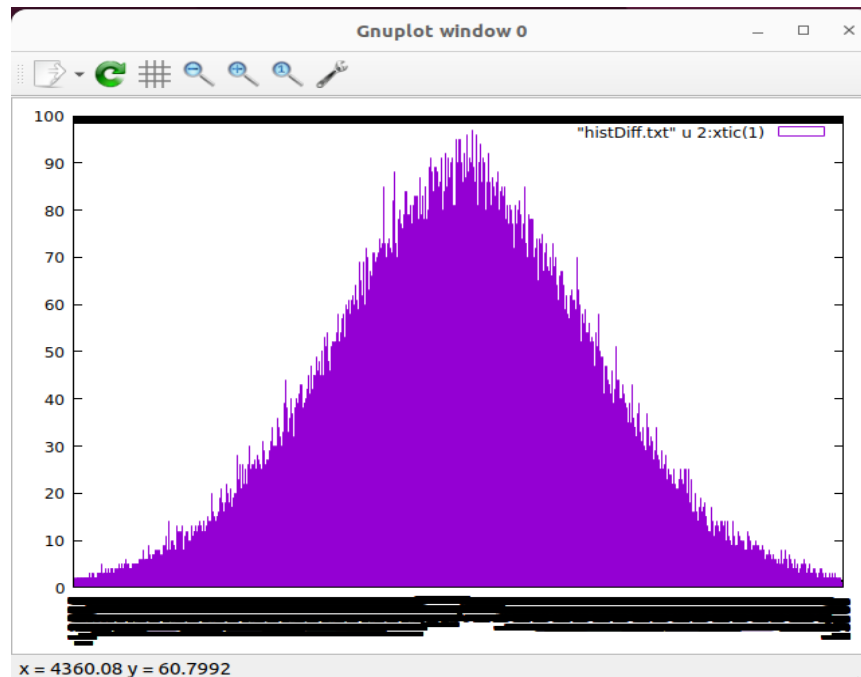


Figura2: Histograma para a diferença de canais

## Exercício 3

Para correr o programa é necessário executar:

```
../sndfile-example-bin/wav_quant sample.wav 10 samplebits.wav
```

Neste exercício, para reduzir o número de bits de um ficheiro de áudio, através das samples de um ficheiro de entrada foi feito um *shift* de *n* bits para reduzir os seus bits.

```
sample = sample >> num_of_bits;
temp = sample << num_of_bits;      //move bits
quant_samples.insert(quant_samples.end(), temp);
```

## Exercício 4

Neste exercício foi utilizada a biblioteca *AudioFile* (recorrendo a uma classe *AudioFile.h*) para manipular ficheiros de áudio.

Para exemplificar o resultado e objetivo deste problema, foi calculada o SNR(signal-to-noise ratio) entre os ficheiros de áudio sample.wav e out.wav e os ficheiros sample.wav e copy.wav.

Como a quantização introduz alguns erros, é importante medir a quantidade de ruído que foi introduzida.

Por isso, para medir a distorção (poder de ruído) introduzida, utilizou-se a seguinte fórmula:

$$- \text{Pr} = D$$

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

Figura3: Fórmula para medir a distorção (poder de ruído) introduzida

Para medir o poder do sinal sem distorção usou-se a fórmula:

$$- P_x = E_x \cdot (1/N)$$

$$E_x = \sum_{n=-\infty}^{\infty} |x(n)|^2.$$

Figura4: Fórmula para medir o poder de sinal sem distorção

Por último, para calcular o SNR (expresso em decibel) foi necessário utilizar a seguinte fórmula (em que utiliza o resultado das duas fórmulas anteriores):

$$\text{SNR} = 10 \log_{10} \frac{P_x}{P_r} \text{ dB},$$

Figura5: Fórmula SNR (Signal-to-noise ratio)

De seguida, segue a demonstração e os resultados de um programa que foi desenvolvido denominado “wav\_cmp.cpp” que dá print ao SNR de um certo ficheiro de áudio em relação ao outro.

Para testar este programa, primeiramente foi calculado o SNR entre os ficheiros de áudio “sample.wav” e “out.wav”

Como comandos para compilação, foram usados os seguintes:

```
g++ wav_cmp.cpp -o wav_cmp
./wav_cmp sample.wav out.wav
```

Figura6: Comandos para compilação ficheiro wav\_cmp.cpp

Obtendo o seguinte resultado:

```
osboxes@osboxes:~/Desktop/IC-main/project1/sndfile-example-src$ ./wav_cmp sample
.wav out.wav
SNR: 17.0321 dB
Maximum per sample absolute error: 1.98831
```

Figura7: SNR e maximum per sample absolute error entre ficheiros “sample.wav” e “out.wav”

Depois, foi calculado o SNR entre os ficheiros de áudio “sample.wav” e “copy.wav” que são iguais, logo o maximum per sample absolute error iria ser 0, o que foi comprovado:

```
osboxes@osboxes:~/Desktop/IC-main/project1/sndfile-example-src$ ./wav_cmp sample
.wav copy.wav
SNR: inf dB
Maximum per sample absolute error: 0
```

Figura7: SNR e maximum

per sample absolute error entre ficheiros “sample.wav” e “out.wav”

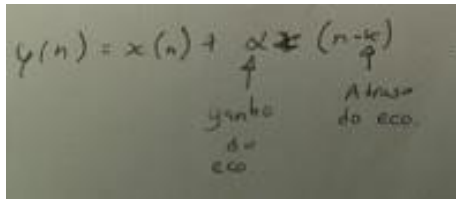
## Exercício 5

Neste exercício foi utilizada a biblioteca *AudioFile* (recorrendo a uma classe *AudioFile.h*) para manipular ficheiros de áudio.

Foi implementado um programa denominado “wav\_effects.cpp” onde é possível produzir dois efeitos de áudio, um Eco único e múltiplos

Ecos, em relação a um ficheiro de áudio original (“sample.wav”) e produzir um ficheiro de áudio destino contendo esses mesmos efeitos.

Para produzir um Eco único, foi utilizada a seguinte fórmula aplicada a cada sample (a começar na segunda) do ficheiro de áudio original (no código implementado o ganho do eco está predefinido como 0.8 e o atraso do eco como 4410 que corresponde a 100ms (44100/100=4410)):


$$y(n) = x(n) + \alpha x(n-k)$$

↑                      ↑  
ganho                      Atraso  
do eco                      do eco

Figura8: Fórmula para calcular um único Eco

Para produzir múltiplos Ecos, foi utilizada a seguinte fórmula aplicada a cada sample (a começar na segunda) do ficheiro de áudio original (foi usado o mesmo ganho e atraso):

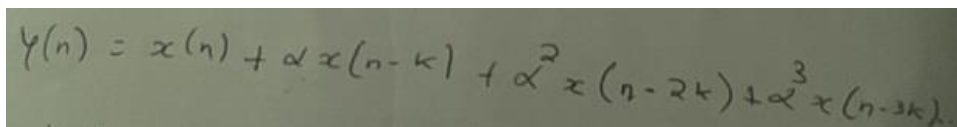
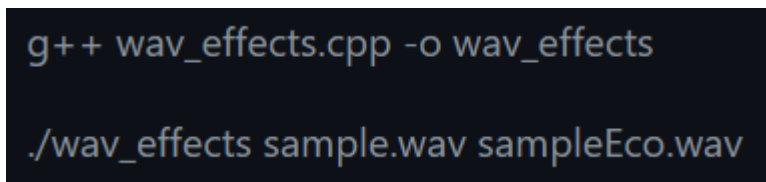

$$y(n) = x(n) + \alpha x(n-k) + \alpha^2 x(n-2k) + \alpha^3 x(n-3k)$$

Figura9: Fórmula para calcular múltiplos Ecos

Em ambos os casos, foi dividido o resultado de cada uma das fórmulas por (1+atraso\_do\_eco) para tudo dar corretamente e não haver problemas de amplitude.

Como comandos para compilação, foram usados os seguintes:



```
g++ wav_effects.cpp -o wav_effects
./wav_effects sample.wav sampleEco.wav
```

Figura10: Comandos para compilação programa “wav\_effects.cpp”

Como resultados temos um ficheiro de áudio denominado “sampleEco.wav” em que:

- Se o objetivo for produzir Eco único, tem de se descomentar a parte "Eco", linha 60 no ficheiro *wav\_effects* e correr os comandos de compilação.
- Se o objetivo for produzir múltiplos ecos, tem de se descomentar a parte "Eco Múltiplo", linha 63 no ficheiro *wav\_effects* e correr os comandos de compilação.

Para verificar os resultados basta reproduzir o ficheiro de áudio destino ("sampleEco.wav") e verificar auditivamente o Eco único ou os múltiplos Ecos.

## Exercício 6

No exercício 6, pede-nos para criar uma *bitstream* para ler e escrever bits num ficheiro de texto e por isso, criámos a função *writeBit* para escrever um bit no ficheiro e o *readBit* para ler um bit do ficheiro.

Em seguida, criámos mais duas funções, *readNBit* e *writeNBits*, que chama as funções anteriores, para ler e escrever vários bits num ficheiro.

Por fim, criámos outras duas funções, a *EOF* para sabermos quando o ficheiro termina e a *close* para fechar o ficheiro.

## Exercício 7

Neste exercício foi utilizada a classe *BitStream.h* desenvolvida no exercício anterior para desenvolver os programas "encoder.cpp" e "decoder.cpp".

O programa "encoder.cpp" recebe um ficheiro .txt contendo 0s e 1s e converte-o para um ficheiro binário correspondente. O "decoder.cpp" faz a operação oposta, recebe um ficheiro binário e converte-o num ficheiro .txt correspondente.

No desenvolvimento destes dois programas foram usados os métodos *readBit()* e *writeBit()* da classe *BitStream.h*.



Como exemplo de uma demonstração do funcionamento de ambos os programas, existe um ficheiro .txt, por exemplo:

```
project1 > sndfile-example-src > ex7.txt
1 0110
```

Figura11: Ficheiro ex7.txt

Tendo o ficheiro .txt que vai ser convertido para um ficheiro binário através do “encoder.cpp”, utiliza-se os seguintes comandos de compilação:

```
g++ BitStream.cpp encoder.cpp -o ex7encoder
./ex7encoder ex7.txt ex7convertido.bin
```

Figura12: Comandos de compilação do programa “encoder.cpp”

Após feita a compilação, é feito o print no terminal do conteúdo do ficheiro .txt em binário, e é originado um ficheiro ex7convertido.bin contendo esse mesmo conteúdo:

```
osboxes@osboxes:~/Desktop/IC-main/project1/sndfile-example-src$ ./ex7encoder ex7
.txt ex7convertido.bin
valores do conteúdo do ficheiro ex7.txt em binário:
00110000
00110001
00110001
00110000
Arquivo Criado
```

Figura13: Compilação e print dos valores do conteúdo do ficheiro ex7.txt em binário

```
project1 > sndfile-example-src > ex7convertido.bin
1
```

Figura14: Ficheiro binário originado após usado o “encoder.cpp”

Tendo esse ficheiro binário originado, irá agora usar-se o “decoder.cpp” para o desconverter e originar um ficheiro .txt em que o conteúdo terá de ser igual ao ficheiro ex7.txt originalmente convertido pelo “encoder.cpp”.

Para isso são utilizados os seguintes comandos de compilação:

```
g++ BitStream.cpp decoder.cpp -o ex7decoder  
./ex7decoder ex7convertido.bin ex7desconvertido.txt
```

Figura15: Comandos de compilação do programa “decoder.cpp”

Após a compilação, irá ser originado um ficheiro .txt, neste caso ex7desconvertido.txt, que irá conter o seguinte conteúdo:

```
project1 > sndfile-example-src > ex7desconvertido.txt  
1 0110
```

Figura16: Ficheiro ex7desconvertido.txt

Pode-se verificar que o conteúdo do ficheiro ex7desconvertido.txt é igual ao conteúdo do ficheiro ex7.txt.

## Exercício 8

Neste exercício, após termos convertido os valores do áudio, usando o *DCT*, tivemos de guardar esses valores num ficheiro de texto para, posteriormente, transformarmos esses valores para binário.

No entanto, ao verificarmos que o ficheiro, que continha os valores do *DCT*, era muito grande, cerca de 1 milhão de linhas, tivemos de ler 1024 valores a 1024 valores da conversão do *DCT*, e à medida que liamos esses valores, fazíamos a conversão para binário. Por falta de tempo, não conseguimos dar *decode* ao ficheiro binário e, portanto, não conseguimos ver o resultado final.

Para este exercício, tivemos também de criar outra *BitStream* porque ao usar a *Bitstream* do exercício 6 não estávamos a conseguir obter o tamanho do ficheiro de entrada e por isso, no código do *encode* e *decoder* do exercício 7 estávamos a predefinir o tamanho máximo que um ficheiro poderia ter.

Como no exercício 8, ao usarmos o DCT, obtivemos um ficheiro com 1 milhão de linhas, tivemos de criar outra *BitStream* e o respetivo *enconder*.

## Contribuição dos Autores

Todos os autores participaram de igual forma na divisão, desenvolvimento e discussão deste projeto.