Relatório nº1 IC

Edgar Sousa(98757),João Castanheira(97512),
Nuno Fahla(97631)
Turma P1

1.0

Conteúdo

Capítulo 1

Introdução

Neste trabalho prático para a disciplina de IC, foi-nos pedido para, com base na biblioteca **libsndfile**, que é uma biblioteca para C com o objetivo de ler e escrever ficheiros que contenham amostras de som, desenvolver um conjunto de funcionalidades de modo a explorar as capacidades da biblioteca escolhida. Esta biblioteca será usada com um *wrapper* em C++ para as funções, de maneira a que os exercícios possam ser desenvolvidos nessa mesma linguagem. Este relatório irá descrever todos os passos relevantes e decisões tomadas aquando da execução dos items pedidos.

Capítulo 2

Ferramentas utilizadas e recolha de resultados

Para a execução e análise deste trabalho, foram, além da linguagem C++ utilizados outros recursos:

- **gnuplot** - programa da linha de comando que descreve gráficos através de um conjunto de dados.

Para a recolha de resultados dos exercícios produzidos utilizamos o ficheiro *wav_hist.cpp* para criar histogramas em que foram guardados o número de vezes que cada amostra aparece no ficheiro *wav* escolhidos. Estes histogramas foram depois transformados em gráficos através do **gnuplot**. Seguidamente, analisamos estes gráifcos e foi-nos permitido fazer a avaliação do que era possível obter com os exercícios pedidos, assim como verificar se os dados observados correspondiam ao que era pretendido fazer, sempre que possível.

Capítulo 3

Resumo das funcionalidades implementadas

Com a realização deste trabalho prático, as funcionalidades desenvolvidas foram as seguintes:

- Criação de um canal **MID** e **SIDE** para um ficheiro *wav*, assim como a criação de um histograma para esse canal usando como base o ficheiro *wav_hist.h*
- Implementação de uma classe C++ (*wav_quant.h*), assim como um programa associado (*wav_quant.cpp*) que realiza quantização por n bits de cada amostra.
- Implementação de um programa (*wav_cmp.cpp*) que compara um ficheiro de som original com uma cópia que sofreu quantização escalar uniforme e calcula o *signal-to-noise-ratio* assim como o erro máximo encontrado entre as amostras dos ficheiros.
- Implementação de um programa (*wav_effects.cpp*) que introduz efeitos a uma amostra de som.
- Implementação e teste de uma classe (*Bitstream.h*) que processa a leitura e escrita de bits para um ficheiro.
- Implementação de um codec baseado no Discrete Cosine Transform, usando a classe Bitstream para codificar e descodificar as amostras.

Capítulo 4

Descrição e resultados das funcionalidades implementadas

4.1 Exercício 2

Neste exercício procedeu-se à criação do canal **MID**, para, ao usar o programa *wav_ hist.cpp*, mostrar o histograma da média dos canais Esquerdo e Direito. Isto tornou-se possível, criando na classe **WAVHist**, um mapa que guarda o número de vezes que cada amostra aparece no **MID channel**, sendo que as amostras do **MID channel** são dadas pela seguinte fórmula:

$$\frac{\text{Amostras do canal Esquerdo + Amostras do canal Direito}}{2}$$

Para a criação do **SIDE channel**, foram executados exatamente os mesmos passos, à excecção da aplicação da fórmula, que passou a ser a seguinte:

$$\frac{\text{Amostras do canal Esquerdo - Amostras do canal Direito}}{2}$$

Para verificar que funciona corretamente, obtiveram-se os histogramas dos canais MID e SIDE de 2 dos ficheiros *wav* fornecidos pelo professor, sendo os seus gráficos os seguintes:

Figura 4.1: Histograma das amostras do canal MID do ficheiro sample01.wav

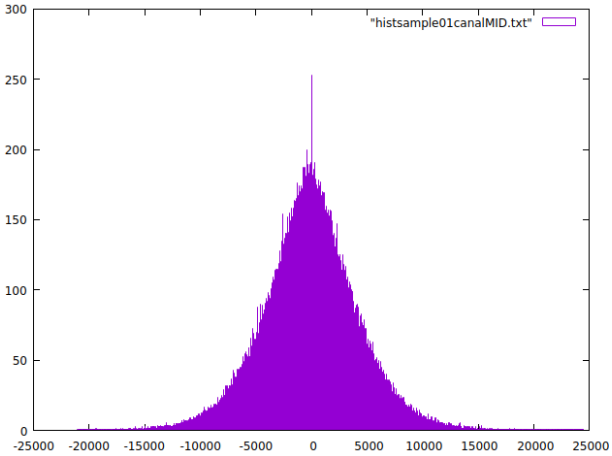


Figura 4.2: Histograma das amostras do canal SIDE do ficheiro sample01.wav

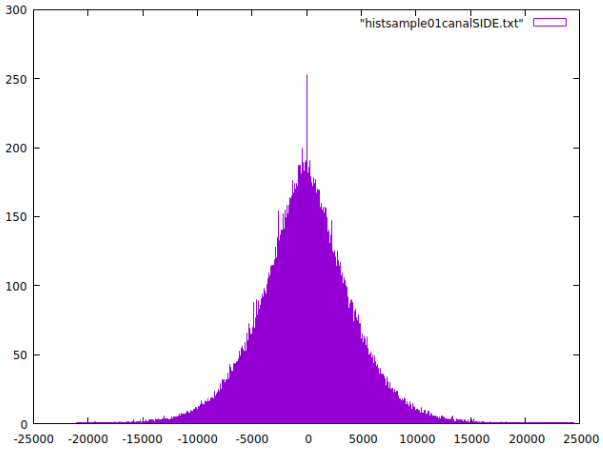


Figura 4.3: Histograma das amostras do canal MID do ficheiro sample02.wav

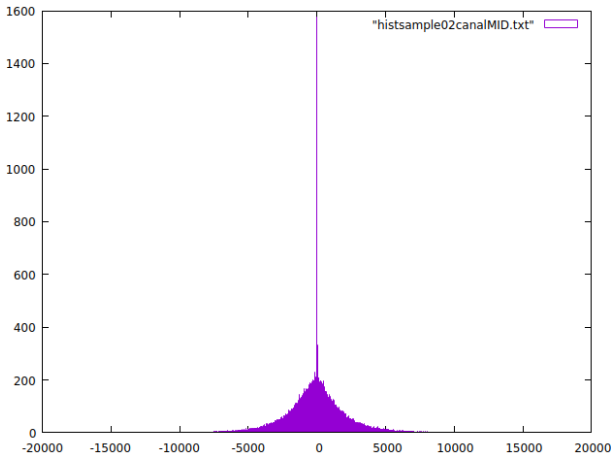
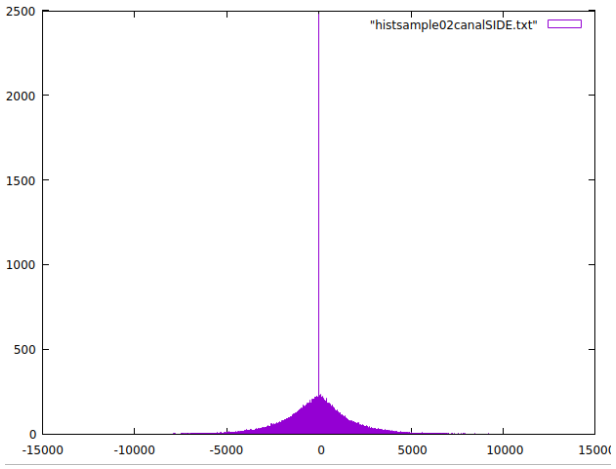


Figura 4.4: Histograma das amostras do canal SIDE do ficheiro sample02.wav



4.2 Exercício 3

O exercício seguinte pedia para realizarmos a quantização de um ficheiro *wav*, utilizando uma classe e um programa para o efeito. Os ficheiros *wav_ quant.cpp* e *wav_ quant.h* nasceram para esse propósito. A classe desenvolvida disponibiliza os métodos (quantSamples) para a quantização e o ficheiro *cpp* permite o seu teste. Assim sendo, a quantização é reduzir o número de bits utilizados para representar cada amostra e é realizada no nosso trabalho ao ir ao valor de cada amostra, fazer um shift-right de n bits, sendo que n é um número introduzido pelo utilizador, precedido por um shift-left de n bits, usado para preencher com 0's os n bits removidos previamente.

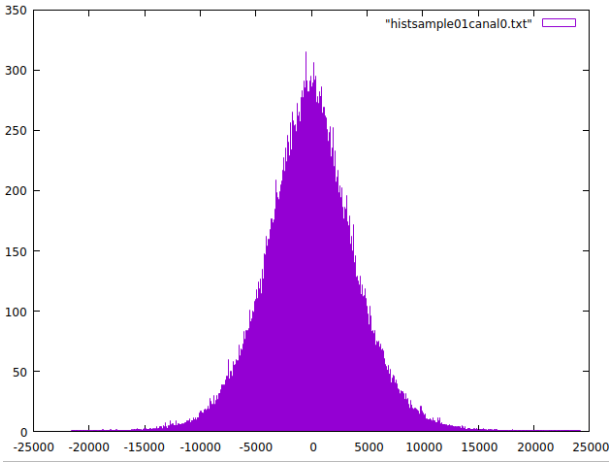
Figura 4.5: Código responsável pela quantização

```
std::vector<short> quantSamples(std::vector<short> samples, size_t quantNumber){
    for(size_t i = 0; i < samples.size(); i++){
        short newsample = samples[i]>>quantNumber;
        newsample = newsample<<quantNumber;
        quantdsamples[i] = newsample;
    }

    return quantdsamples;
}
```

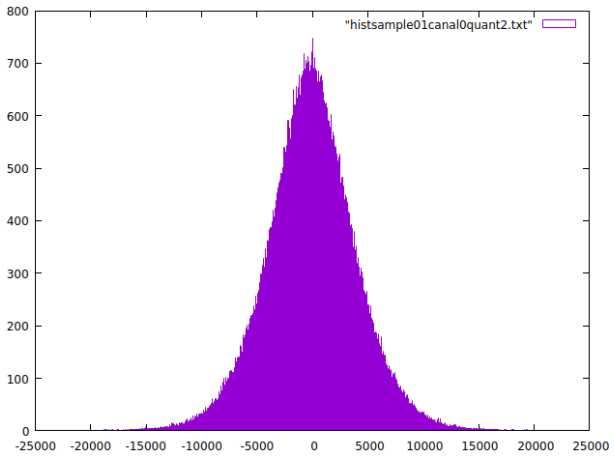
De seguida, são apresentado os histogramas de um ficheiro de aúdio não quantizado e de um ficheiro de áudio quantizado:

Figura 4.6: Histograma do canal 0 do ficheiro sample01.wav



Atentando nas figuras 4.6 e 4.7 podemos reparar que a quantização foi efetuada com sucesso, uma vez que apesar da forma do gráfico ser semelhante para ambos os histogramas, a escala de valores do eixo dos Y's da figura 4.7 utiliza números superiores que os valores do eixo dos Y's da figura 4.6. Isto acontece, pois ao retirar os bits menos significativos de cada amostra (neste caso 2, pois foi feita uma quantização de 2 bits), vamos ter um conjunto de amostras com

Figura 4.7: Histograma do canal 0 do ficheiro sample01.wav após sofrer uma quantização de 2 bits



valores mais semelhantes e por consequência, se repetem mais vezes. Usando as capacidades de zoom do **gnuplot**, pudemos também verificar que é necessário um número menor de zooms até encontrar espaços entre as barras do histograma 4.7, o que significa que existem valores que não têm nenhuma entrada no histograma, após a quantização, o que satisfaz o objetivo do exercício.

4.3 Exercício 4

Ao efectuar a quantização, são introduzidos erros provocados pela imprecisão criada pela redução do número de bits usado para representar uma amostra. Assim sendo, e para verificar isto foi desenvolvido um programa que calcula o signal-to-noise-ratio que é dado pela seguinte fórmula:

$$(10 * log10 * \frac{\text{Energia do Sinal}}{\text{Energia do Noise}})$$

A imagem seguinte mostra o código responsável por este cálculo, sendo que a variável **Ex** mostra o cálculo realizado para obter a energia do sinal, a variável **noise** computa o cálculo realizado para obter a energia do sinal, e na variável **snr** o cálculo do signal-to-noise-ratio. Existe também uma variável maxError, que ao longo da iteração pelo ficheiro, guarda o erro máximo dado pela subtração da amostra quantizada menos a amostra nao quantizada:

Figura 4.8: Código responsável pelo cálculo do signal-to-noise-ratio e do erro máximo

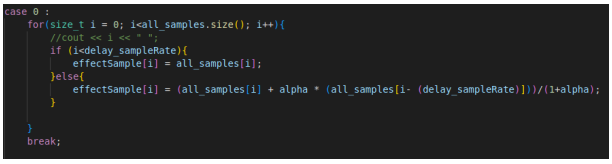
Figura 4.9: signal-to-noise-ratio e erro máximo associado à comparação entre os ficheiros sample04.wav e sample04quant.wav que é o ficheiro sample04 quantizado em 4 bits

4.4 Exercício 5

Com a biblioteca **libsndfile**, o acesso às amostras e a sua consequente manipulação torna-se algo simples. Assim sendo, com algumas operações com as amostras, é possível aplicar efeitos de som aos ficheiros de áudio. Para este exercício, foram desenvolvidos os 2 efeitos seguintes:

- Single Echo - para obtermos este efeito, cria-se um novo vector com as amostras, sendo que esse vector vai ser igual ao inicial, mas a cada número de amostras, que é calculado a partir do valor de delay escolhido pelo utilzador, é concatenada a amostra original com uma amostra volume reduzido por um alpha, também escolhido pelo utilizador.

Figura 4.10: Código responsável pela alteração das samples para criar o efeito de echo



- Multiple Echo - para obtermos este efeito, a lógica utilizada é a mesma, só que é criado um eco sobre ecos anteriores, ou seja há a introdução de amostras cada uma com um volume mais reduzido ao longo do tempo.

Figura 4.11: Código responsável pela alteração das samples para criar o efeito de múltiplo echo

```
for(size_t i = 0; i<all_samples.size(); i++){
    if(i<delay_sampleRate){
        effectSample[i] = all_samples[i];
    }
    else{
        short new_sample = (all_samples[i] + alpha * (all_samples[i- (delay_sampleRate)]))/(1+alpha);
        effectSample[i] = new_sample;
        short new_sample2 = (all_samples[i] + alpha * (effectSample[i- (delay_sampleRate)]))/(1+alpha);
        effectSample[i] = new_sample2;
    }

}
break;
```

4.5 Exercício 6 & 7

No exercício 6 & 7 o objetivo era a criação de uma classe denominada de BitStream que irá conter métodos para, a partir de sequências binárias descritas em texto simples de 8 bits. Também com a habilidade de traduzir ficheiros binários para este mesmo formato de texto.

Um dos requisitos necessários de estes métodos era a eficiência de memória especialmente no processo de descomprimir de binário para texto, pois por cada byte lido irão ser ocupados 8 bytes.

Para testar estes métodos utilizamos sistemas de logs e ficheiro teste. A verificação do encode e decode foi realizado com a comparação de ficheiros através do terminal linux.

Realizando testes temporais o tempo de execução do processamento encoding e decoding do ficheiro sample.wav resultou em 9.145 sec.

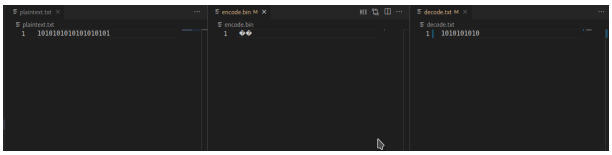
Figura 4.12: Resultados eficiência- temporal

```
$time ../sndfile-example-bin/codec sample.wav 1024
sizeOffile : 2326569
real    0m9.145s
user    0m8.785s
sys     0m0.103s
```

Na figura seguinte podemos ver os resultados de execução dos comandos:

```
1  ../sndfile-example-bin/encoder plaintext.txt 10
2  ../sndfile-example-bin/decoder encode.bin 10
```

Figura 4.13: Exemplo do uso do bitsream txt->bin->txt



Aqui podemos ver a transição dos bits do ficheiro .txt para ficheiro .bin e de novo a .txt.

4.6 Exercício 8

Neste exercício, foi-nos pedido para desenvolver um codec baseado no Discret Cosine Transformation. Assim sendo, a partir do momento em que se corre o programa com o nome de um ficheiro de som como argumento associado do blocksize, este vai passar por vários passos:

- Cálculo dos coeficientes DCT, com base no código fornecido pelos professores em *wav_ dct.cpp*
- Conversão dos coeficientes DCT para binário e escrita num ficheiro de texto.
- Codificação do ficheiro de texto para um ficheiro binário, usando a classe BitStream desenvolvida no exercício anterior.
- Descodificação do ficheiro binário para um ficheiro de texto, usando o método decode da classe BitStream
- Conversão dos coeficientes de DCT de representação binária para float.
- Reconstrução do vetor x_dct (que guarda os coeficientes DCT previamente codificados num ficheiro binário)
- Cálculo da inversa dos coeficientes DCT e reconstrução do vetor de amostras
- Reconstrução do ficheiro de som, com a escrita do vetor de amostras para um novo ficheiro de som.

Apesar de no nosso ficheiro *codec.cpp* estar descrita uma ação sequencial de codificação e descodificação de um ficheiro de somo, estamos cientes que podia ser implementado em dois programas distintos, sendo que um dos ficheiros podia ser usado para dar encode e o outro para dar decode.

Ouvindo o ficheiro de áudio gerado, reparamos que existe um efeito de compressão que foi introduzido pelo nosso codec, assim como era esperado.

Capítulo 5

Conclusão

Com a realização de todos estes exercícios, conseguimos obter uma maior fluidez no uso da linguagem C++, mas mais importante que isso, permitiu-nos consolidar vários conceitos na área da codificação de informação. Permitiu-nos perceber como alguns programas que usamos diariamente geram efeitos de som, assim como interiorizar, de uma forma mais prática, conceitos como a quantização, o que são amostras, canais e manipulação de amostras.

O trabalho realizado foi igualmente distribúido por todos os participantes deste grupo, sendo que fomos todos ativos em todas as fases do projeto, assim como na elaboração do relatório.

Link do repositório com o código desenvolvido

Acrónimos

IC Informação e Codificação