Universidade de Aveiro

Information and Coding

Lab Work nr.1

Guilherme Craveiro (103574),
Nuno Sousa (103664),
Francisco Silva (93400)

Departamento de Eletrônica, Telecomunicações e Informática

29 October, 2023

Índice

Introdução

Este trabalho consiste na resolução de exercícios da cadeira Informação e Codificação em foi necessário desenvolver uma solução para manipulação de ficheiros de áudio utilizando a linguagem *C++*.

O código está disponível neste link: https://github.com/GulhermeC/IC-T2G5-2324

Todos os membros contribuíram de forma igual, com uma participação de 33%.

De seguida apresentamos as soluções desenvolvidas para cada um dos exercícios em que se detalha o código, a sua utilização e os resultados obtidos.

Exercício 1

Código

No exercício 1 foi nos pedido que alterássemos a class WAVHist de modo a calcular as versões:

- Mono (Mid channel) $(L + R) / 2$
- Difference (Side channel) $(L - R) / 2$

A classe usa mapas para representar os diferentes histogramas obtidos.

```
void update(const std::vector<short>& samples) {
        size_t n { };

        for(auto s : samples) {
                counts[n++ % counts.size()][s]++;
                // Only makes sense to calculate MID and SIDE for stereo
                // For every two samples (left and right) calculate mid and side
                if (counts.size() == 2 && (n % 2) == 1)
                {
                        average[(samples[n-1] + samples[n]) / 2]++;     // average[(left+right)/2]
                        difference[(samples[n-1] - samples[n]) / 2]++;  // difference[(left-right)/2]
                }
        }
}
```

A função update é chamada à medida que novas amostras são lidas do ficheiro de entrada. Quando se trata de um ficheiro stereo são calculadas as novas amostras e guardadas nos respectivos mapas.

A classe também suporta a criação de *coarser bins*.

A função updateBins atua assumindo que que os histogramas counts já foram calculados.

Em função do tamanho dos bins pretendidos, para cada bin é calculada a média das amostras que vão pertencer ao bin, e que vai ser o valor que representa o bin em questão.

```
void updateBins(int binSize) {
        for (uint i = 0; i < counts.size(); i++) {

                int sum = 0;            // used to calculate average of values inside the consider
                int total = 0;          // keep track of the counters of the values in the bin
                int x = 0;                      // iterate over bins

                for(auto [value, counter] : counts[i]) {
                        sum += value;
                        total += counter;
                        x++;
                        if (x == binSize) {     // create new bin
                                bins[i][(sum / binSize)] = total;       // use average as index
                                sum = 0;
                                total = 0;
                                x = 0;
                        }
                }
                if (x % binSize != 0)   // handle last potential uncompleted bin
                        bins[i][(sum / binSize)] = total;
```

Utilização do código

Para testar a classe temos o programa wav_hist com a seguinte utilização:

```
Usage: ./wav_hist <input file> <channel> [OPTIONS]
OPTIONS:
        -s    --- bin size (default 4)
```

O programa recebe um ficheiro de áudio como entrada e o canal para o qual vão ser gerados os histogramas. É ainda possível especificar o tamanho dos bins pretendido.

Para cada histograma é criado um ficheiro que é guardado na pasta *histograms*.

Resultados

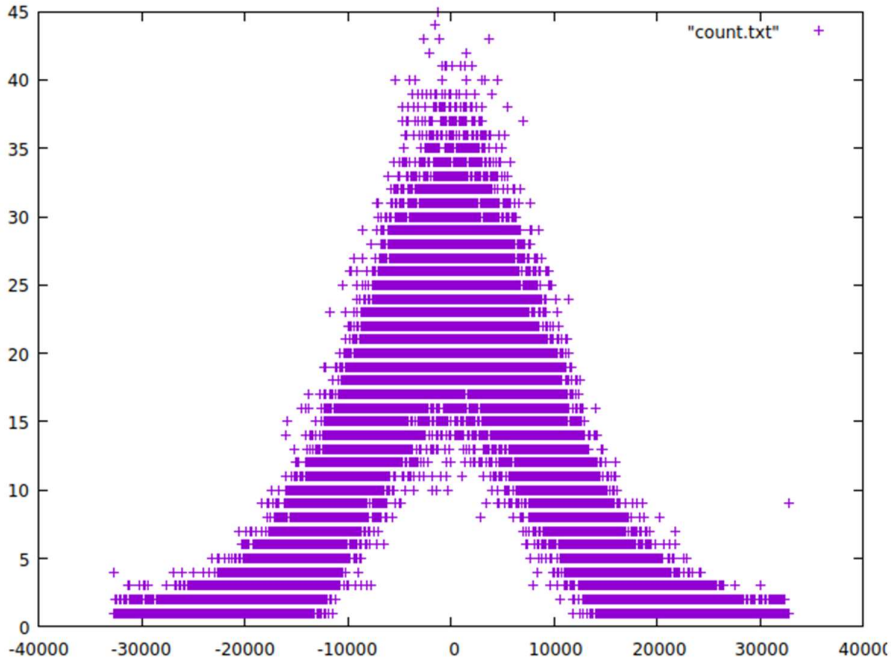

Gráfico 2.1 - Canal esquerdo

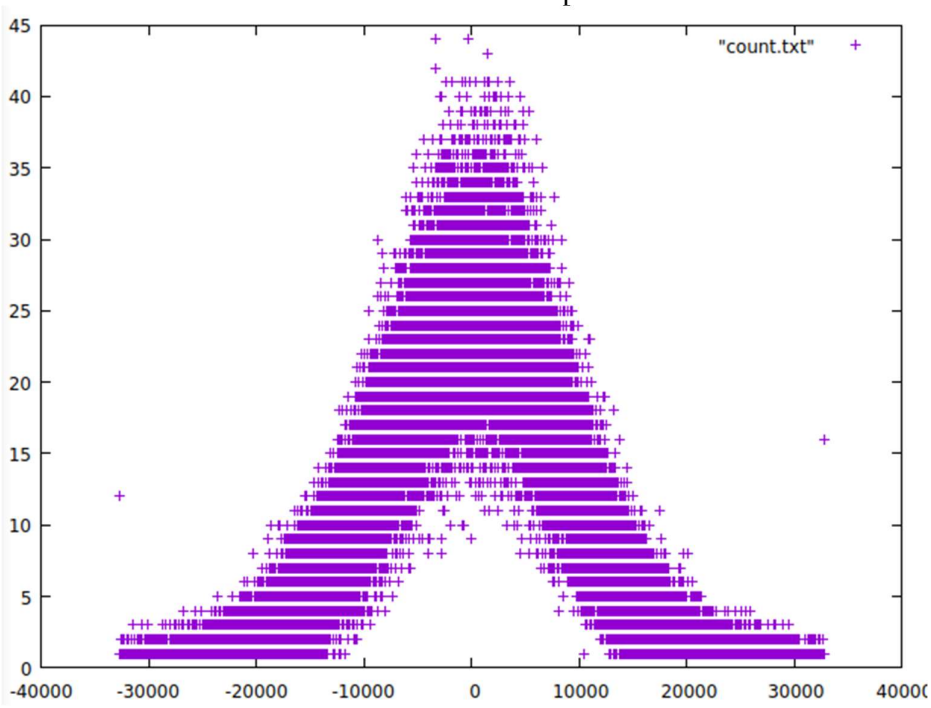

Gráfico 2.2 - Canal direito

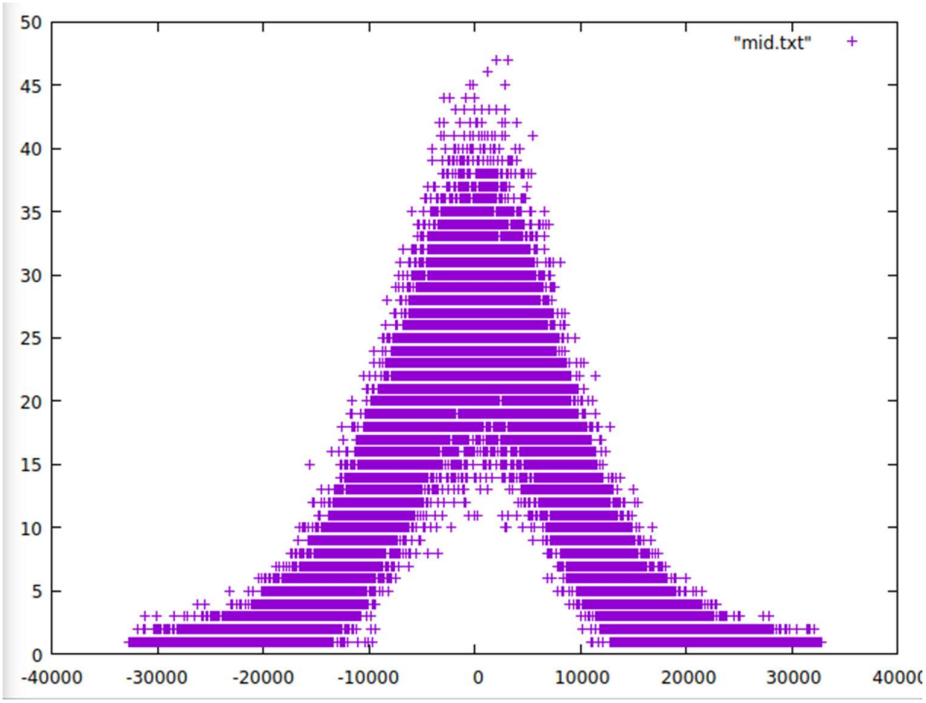

Gráfico 2.3 - Canal MID

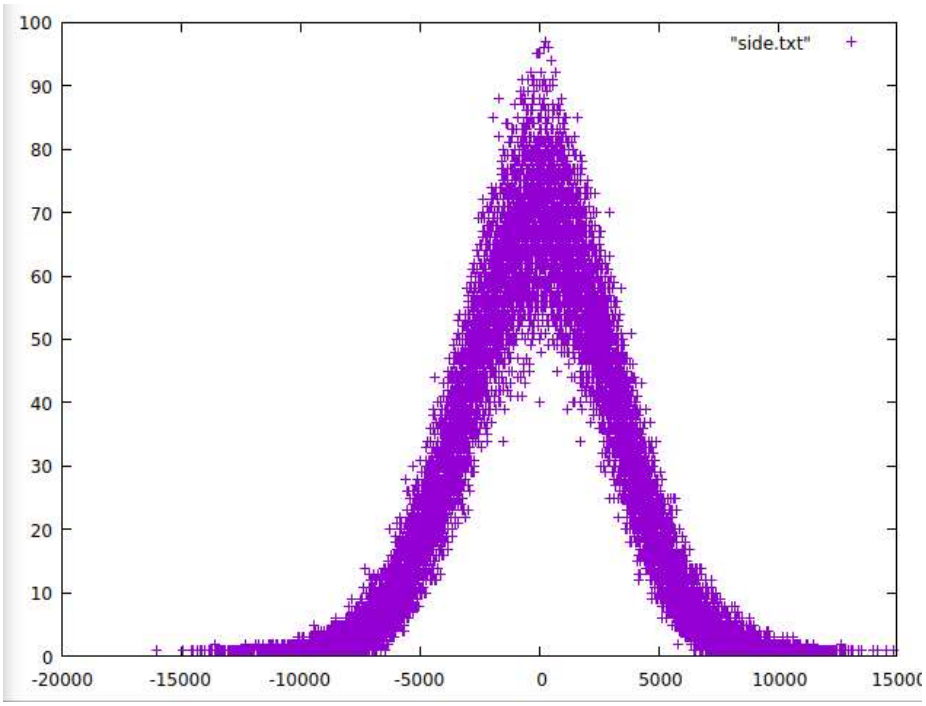

Gráfico 2.4 - Canal SIDE

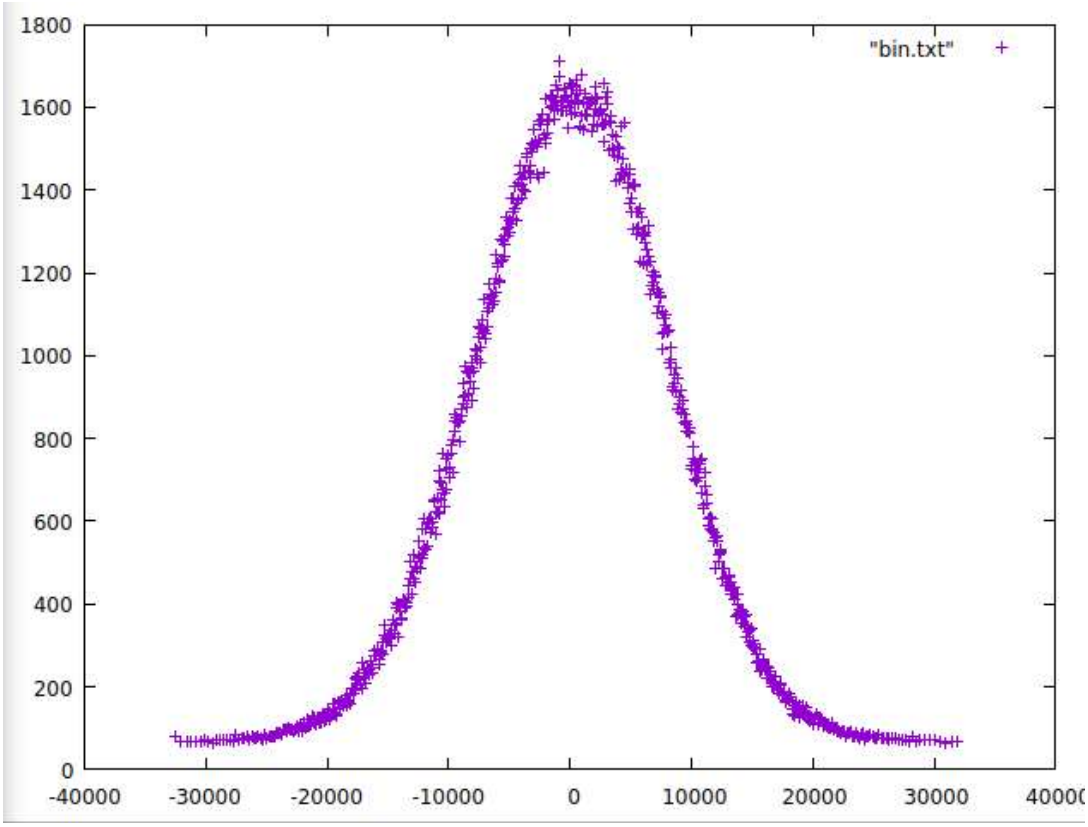

Gráfico 2.4 - Coarser bins canal esquerdo (binSize 64)

Exercício 2

Código

Neste exercício criamos um programa para comparar dois ficheiros de áudio, esta comparação toma forma com o output do "a*verage mean squared error*"(MSE), "*maximum per sample absolute error*"(AE) e "*signal-to-noise ratio*"(SNR).

Estes três parâmetros são calculados e imprimidos da seguinte maneira:

$$MSE = \frac{1}{n}\sum_{i=0}^{n} \quad (Yi - \hat{Y}i)^2$$

```
//MSE
double squaredDifference = pow(CEsamples[i]-OGsamples[i],2);
channelSums[channelIndex] += squaredDifference;

channelCounts[channelIndex]++;

averageSum += squaredDifference;
averageCount++;
```

```
for (size_t channelIndex = 0; channelIndex < OGsndFile.channels(); channelIndex++) {
    double mse = channelSums[channelIndex] / channelCounts[channelIndex];
    cout << "MSE for Channel " << channelIndex << ": " << mse << "\n";
}

double averageMSE = averageSum/averageCount;
cout << "Average MSE: " << averageMSE << "\n\n";
```

$$AE = max(abs(Yi - \hat{Y}i))$$

```
//Absolute Error
double absoluteError = abs(CEsamples[i] - OGsamples[i]);

if (absoluteError > maxAbsoluteErrors[channelIndex]) {
        maxAbsoluteErrors[channelIndex] = absoluteError;
}

if (absoluteError > maxAbsoluteError) {
    maxAbsoluteError = absoluteError;
}
```

```
for (size_t channelIndex = 0; channelIndex < OGsndFile.channels(); channelIndex++) {
    cout << "Max Absolute Error for Channel " << channelIndex << ": "
    << maxAbsoluteErrors[channelIndex] << "\n";
}

cout << "Max Absolute Error: " << maxAbsoluteError << "\n\n";
```

$$Psignal = \sum_{i=0}^{n} \hat{Y}i^2$$

$$Pnoise = \sum_{i=0}^{n} (Yi - \hat{Y}i)^2$$

$$SNRdb = 10log(\frac{Psignal}{Pnoise})$$

```
//SNR
powerSignal[channelIndex] += pow(OGsamples[i], 2);
powerNoise[channelIndex] += pow(CEsamples[i] - OGsamples[i], 2);

overallSignalpower += pow(OGsamples[i], 2);
overallNoisepower += pow(CEsamples[i] - OGsamples[i], 2);
```

```
for (size_t channelIndex = 0; channelIndex < OGsndFile.channels(); channelIndex++) {
    double channelSNR = 10 * log10(powerSignal[channelIndex] / powerNoise[channelIndex]);
    cout << "SNR for Channel " << channelIndex << ": " << channelSNR << " dB\n";
}

overallAverageSNR = 10 * log10(overallSignalpower / overallNoisepower);
cout << "Average SNR across all channels: " << overallAverageSNR << " dB\n\n";
```

Utilização do código

Para utilizar o programa basta inserir um comando como o abaixo onde "*sample.wav"* é o ficheiro original e "*certain.wav*" o ficheiro que pretendemos comparar.

```
./wav_cmp sample.wav certain.wav
```

Resultados

Correndo o programa com "*sample.wav"* e um "*certain.wav"* criado com o programa "*wav_dct*" obtemos o seguinte resultado.

```
MSE for Channel 0: 1.2324e+06
MSE for Channel 1: 1.70046e+06
Average MSE: 1.46643e+06

Max Absolute Error for Channel 0: 64829
Max Absolute Error for Channel 1: 65153
Max Absolute Error: 65153

SNR for Channel 0: 17.7769 dB
SNR for Channel 1: 16.3994 dB
Average SNR across all channels: 17.0321 dB
```

Exercício 3

Código

A classe WAVQuant e o programa associado permitem quantizar de forma uniforme um ficheiro de áudio.
Recebemos as amostras do ficheiro original e guardamos novas amostras quantizadas.
A classe permite 2 formas de quantizar as amostras.

Uma das hipóteses e a mais óbvia é colocar os n bits menos significativos a zero.

```
// Sets given amount of LSB to zero
void quantBits(const std::vector<short>& samples, uint numBits) {
    for (auto s : samples) {
                    short quantized = s & (0xFFFF << numBits);
        quantSamples.push_back(quantized);
    }
}
```

Para cada sample fazemos n shifts à esquerda, guardando o novo valor com n bits menos significativos a zero.

Outra hipótese é calcular os intervalos em função do número de níveis pretendido.

```
// Quantize by given number of levels
void quantLevels(const std::vector<short>& samples, uint numlevels) {
            uint delta = (1 << 16) / numlevels;
    for (auto s : samples) {
                    short quantized = (s / delta) * delta + delta / 2;
                    quantSamples.push_back(quantized);
    }
}
```

Calculamos: $\Delta = A/2^b$ usando delta para calcular o intervalo ao qual pertence a amostra original.

Utilização do código

Para verificar as funcionalidades da classe temos o programa wav_quant com a seguinte utilização.

```
Usage: ../sndfile-example-bin/wav_quant <input_file> <output_file> [OPTIONS]
OPTIONS:
        -m mode [levels|delete]     --- quantization mode (default: levels)
        -d delete bits              --- number of bits to zero (default: 8)
        -l level bits               --- 2^b levels (default 8)
```

Por exemplo, se quisermos quantizar colocando a 0 os 10 bits menos significativos.

```
../sndfile-example-bin/wav_quant sample.wav quantized.wav -m delete -d 10
```

Ou se quisermos quantizar por níveis com por exemplo 8 bits (2^8 = 256 níveis).

```
../sndfile-example-bin/wav_quant sample.wav quantized.wav -m levels -l 8
```

Resultados

Ao aumentar o número de bits a ser colocados a zero ou ao definir os intervalos temos cada vez menos níveis de representação e podemos observar os efeitos produzidos por estas mudanças.

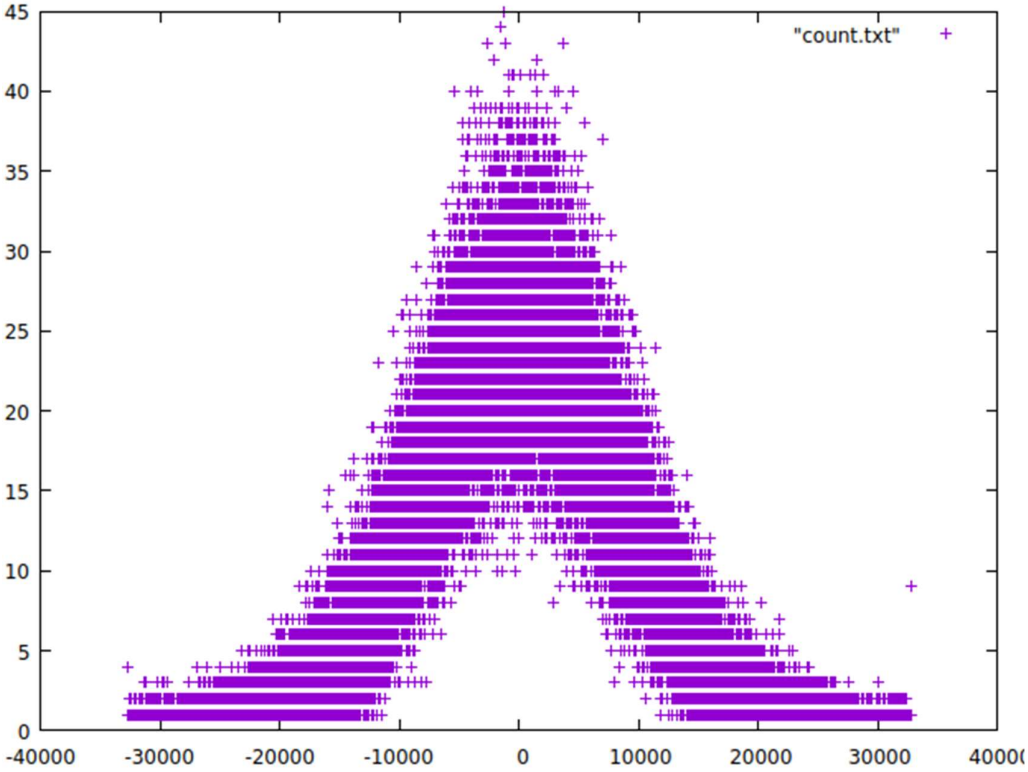

Gráfico 2.1

No gráfico 2.1 temos o histograma do ficheiro original sample.wav.

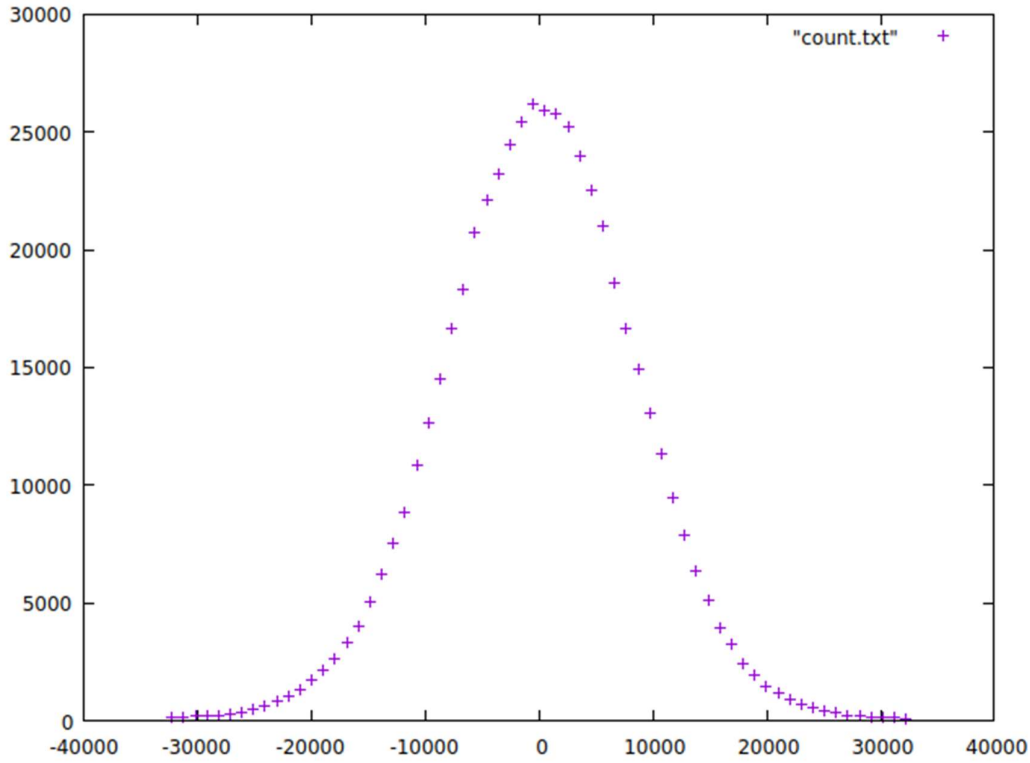

Gráfico 2.2

No gráfico 2.2 podemos observar o histograma do ficheiro quantizado com uma resolução de 6 bits, ou seja 64 níveis.

Comparando os dois ficheiros de áudio conseguimos ouvir um ruído de fundo que aumenta à medida que a resolução diminui.

Exercício 4

Código

Os efeitos implementados incluem: Eco Simples, Ecos Múltiplos, Modulação de Amplitude e a Inversão do som.
Para este fim foram utilizadas as equações abordadas nas aulas teóricas para o Eco Simples, Ecos Múltiplos e Modulação de Amplitude.

Eco Simples: $y(n) = x(n)+\alpha \times x(n-delay)$
Onde x é a amostra de som original e y é a amostra de som modificada.

```
vector<short> single_echo(SndfileHandle sndFile, int delay, float gain)
{
    vector<short> smples_out;
    vector<short> smples(FRAMES_BUFFER_SIZE * sndFile.channels());
    size_t nFrames;
    short sample;
    smples_out.resize(0);
    while((nFrames = sndFile.readf(smples.data(), FRAMES_BUFFER_SIZE)))
    {
        smples.resize(nFrames * sndFile.channels());

        for(int n = 0; n < (int)smples.size(); n++)
        {
            if(n >= delay)
            {
                // Y(N) = X(N) + A * X(N - delay)
                sample = (smples.at(n) + gain * smples.at(n - delay)) / (1 + gain);
            }else{
                sample = smples.at(n);
            }
            smples_out.insert(smples_out.end(), sample);
        }
    }

    return smples_out;
}
```

Ecos Múltiplos: $y(n) = x(n)+\alpha \times y(n-delay)$

```
vector<short> multiple_echos(SndfileHandle sndFile, int delay, float gain)
{
    vector<short> smples_out;
    vector<short> smples(FRAMES_BUFFER_SIZE * sndFile.channels());
    size_t nFrames;
    short sample;
    smples_out.resize(0);

    while((nFrames = sndFile.readf(smples.data(), FRAMES_BUFFER_SIZE)))
    {
        smples.resize(nFrames * sndFile.channels());

        for(int n = 0; n < (int)smples.size(); n++)
        {
            if(n >= delay)
            {
                // Y(N) = X(N) + A * Y(N - delay)
                sample = (smples.at(n) + gain * smples_out.at(n - delay)) / (1 + gain);
            }else{
                sample = smples.at(n);
            }
            smples_out.insert(smples_out.end(), sample);
        }
    }

    return smples_out;
}
```

Modulação de Amplitude: $y(n) = x(n) \times cos(2 \times \pi \times \frac{f}{fa} \times n)$

```
vector<short> amplitude_mod(SndfileHandle sndFile, float freq)
{
    vector<short> smples_out;
    vector<short> smples(FRAMES_BUFFER_SIZE * sndFile.channels());
    size_t nFrames;
    short sample;
    smples_out.resize(0);

    while((nFrames = sndFile.readf(smples.data(), FRAMES_BUFFER_SIZE)))
    {
        smples.resize(nFrames * sndFile.channels());

        for(int n = 0; n < (int)smples.size(); n++)
        {
            // Y(N) = X(N) * cos(2*pi*(F/Fa)*N)
            sample = smples.at(n) * cos(2 * M_PI * (freq/sndFile.samplerate()) * n);

            smples_out.insert(smples_out.end(), sample);
        }
    }

    return smples_out;
}
```

Para a implementação do efeito da Inversão do som, foi criada uma função com o objetivo de escrever as amostras num novo ficheiro por ordem inversa, criando assim o efeito de ouvir a amostra de som original ao contrário.

```
vector<short> reverse_snd(SndfileHandle sndFile)
{
    vector<short> smples_out;
    vector<short> smples(FRAMES_BUFFER_SIZE * sndFile.channels());
    size_t nFrames;
    smples_out.resize(0);

    while((nFrames = sndFile.readf(smples.data(), FRAMES_BUFFER_SIZE)))
    {
        smples.resize(nFrames * sndFile.channels());

        for (int n = (int)smples.size() - 1; n >= 0; n--)
        {
            // Add Sample to end
            smples_out.insert(smples_out.end(), smples.at(n));
        }

    }

    return smples_out;
}
```

Utilização do código

Para a utilizar o programa vamos inserir um comando com esta estrutura

```
./wav_effect sample.wav output.wav <effect>
```

Para Eco Simples:

```
./wav_effect sample.wav output.wav single_echo
```

Para Ecos Múltiplos:

```
./wav_effect sample.wav output.wav multiple_echos
```

Para Modulação de Amplitude:

```
./wav_effect sample.wav output.wav amplitude_modulation
```

Para a Inversão de Som:

```
./wav_effect sample.wav output.wav reverse
```

Resultados

Através dos comandos criamos diferentes ficheiros onde aplicamos diferentes efeitos ao ficheiro *sample.wav,* tal como um ficheiro com eco simples com ganho de 0.8 e um atraso de 44100 Hz, outro ficheiro com múltiplos ecos com ganho de 0.8 e atraso de 44100 Hz, um ficheiro com o efeito de modulação de amplitude de 1Hz, e por último um ficheiro que é o inverso da sample utilizada como entrada.

Exercício 5

Código

Neste exercício foi criado um programa *bitStream.h*, para melhorar a eficiência deste programa as funções foram divididas em termos da operação que vão realizar, ler (*read*) ou escrever (*write*).
Em adição, as funções de tradução de entre bits e byte, são feitas entre 1 byte e 8 bits depois disto o array é enviado e processado. Repetindo este processo a eficiência da bitStream vai ser melhorada.

```
vector<int> byteToBit(char byte)
{
    vector<int> bitArr;
    for (int i = 7; i>= 0; i--)
    {
        bitArr.push_back((byte & (1 << i))? 1 : 0);
    }

    return bitArr;
}
```

```
char bitToByte(vector<int> bitArr)
{
    char byte = 0;

    for (int i = 0; i < 8; ++i) {
        byte |= (bitArr[i] << (7 - i));
     }
    return byte;
}
```

A bitStream também inclui uma função para traduzir uma string para o Array de bits.

```
void stringtoBit(string txt)
{

    for(unsigned long int i=0;i<txt.size();i++)
    {
        char chr=txt[i];
        vector<int> bitArr=byteToBit(chr);
        for(unsigned long int k=0;k<bitArr.size();k++)
            bitArray.push_back(bitArr[k]);
    }

}
```

Estas funções são utilizadas dentro das funções principais, tal como a função para ler um ficheiro texto que vai ler um arquivo de texto e coleta uma quantidade limitada de bits.

```
void ReadTxtFile(string fileName, int maxbit)
{
    std::ifstream file(fileName);
    std::string line;

    //ficheiro de saida

    if(file.is_open()) {
        while (getline(file, line)){
            if(max < maxbit){
               // std::cout << line << std::endl;
                for(unsigned long int i=0;i<line.size();i++)
                {
                    if(max < maxbit){
                        bitArray.push_back(line[i]-'0');
                        max++;
                    }else{
                        break;
                    }

                }
            }else{
                max = 0;
                break;

            }

        }
        max = 0;
        file.close();
```

A função de ler um ficheiro binário que lê um arquivo binário e converte uma quantidade limitada de bytes desse arquivo em bits, adicionando-os a um vetor de bits.

```
void ReadBinFile(string fileName, int maxbit)
{
    std::ifstream file(fileName,std::ios::binary);
    char byte;
    if(file.is_open())
    {
        while(file.read(&byte,1)){
            std::cout << byte << std::endl;
            if (max < maxbit){
                //gravar no ficheiro de texto
                vector<int> bitsConvertidos=byteToBit(byte);
                for(unsigned long int i=0;i<bitsConvertidos.size();i++){
                    if(max < maxbit){
                        max++;
                        bitArray.push_back(bitsConvertidos[i]);
                    }else{
                        break;
                    }

                }
            }else{
                max = 0;

                break;
            }

        }
    }
    max = 0;
    file.close();
```

Uma função para escrever um ficheiro texto que vai escrever os bits armazenados no array num ficheiro de texto

```
void WriteTxTFile(string fileName)
{

    //ficheiro de saida
    std::ofstream fileOut(fileName);
    if (!fileOut.is_open())
    {
        std::cerr << "Erro ao criar ficheiro texto" << std::endl;
        return;
    }
    if(fileOut.is_open())
    {
        for(unsigned long int i=0;i<bitArray.size();i++){
            fileOut << bitArray[i];
            if ((i+1) % 8 ==0)
                fileOut << std::endl;
        }

    }
    fileOut.close();
```

A função onde os bits armazenados no array vai ser escritos num ficheiro binário, sendo que cada 8 bits são agrupados em 1 byte antes de serem escritos.

```
void WriteBinFile(string fileName)
{

    //ficheiro de saida
    std::ofstream fileOut(fileName,std::ios::binary);
    if (!fileOut.is_open())
    {
        std::cerr << "Erro ao criar ficheiro binário" << std::endl;
        return;
    }
    if(fileOut.is_open()) {
        vector<int> umbyte;
        for(unsigned long int i=0;i<bitArray.size();i++)
        {
            umbyte.push_back(bitArray[i]);
            if ((i+1) % 8 ==0){

                char convertido= bitToByte(umbyte);
                fileOut.write(&convertido,1);
                umbyte.clear();
            }

        }
        fileOut.close();
```

Por final foi criada uma função para ler strings dentro de um ficheiro de texto sendo que têm de ser tratadas de maneira ficheiro pois têm mais símbolos do que zeros e uns

```
void readStringFile(string fileName){
    std::ifstream inputFile(fileName);
    if(!inputFile.is_open())
    {
        std::cerr << "Erro ao abrir o ficheiro" << fileName << std::endl;
        return;
    }

    std::string line;
    while(getline(inputFile, line))
    {
        lines.push_back(line);
    }
    inputFile.close();
    std::cout << lines.size() << " Lines read" << std::endl;

    for(unsigned long int i=0;i<lines.size();i++)
    {
        stringtoBit(lines[i]);
    }
```

Exercício 6

Código

No Exercício 6 foi criado o programa *encoder*, este programa foi utilizado de maneira a verificar o código dentro da função bitStream, usando um ficheiro de texto que só contém zeros e uns.

O encoder tem 2 funções primárias, uma para dar encode e outra para dar decode.
No caso da função para dar encode, o ficheiro vai ser assumido como que só tenha zeros e uns e nenhum outro caracter. Assim vai ser passado para a função *ReadTxtFile* do bitStream, e seguida para a função *WriteBinFile*, para finalizar a transição retornando um ficheiro binário.

```
if (fileMode=="e")
{
    bts.ReadTxtFile(fileName, maxbit);
    bts.WriteBinFile(fileNameOut);
```

Para realizar o decode, o ficheiro é enviado para a função *ReadBinFile*, e de seguida para o *WriteTxTFile* transformando assim o ficheiro binário num ficheiro de texto.

```
}else if(fileMode=="d"){
    bts.ReadBinFile(fileName, maxbit);
    bts.WriteTxTFile(fileNameOut);
```

O programa também inclui mais 2 funções não primárias que foram utilizadas de forma a verificar o código dentro da função bitStream.
Estes modos têm o objetivo de traduzir uma string dentro de um ficheiro texto para um ficheiro binário

```
}else if(fileMode=="wr"){
    bts.readStringFile(fileName);
    bts.WriteBinFile(fileNameOut);
```

e traduzir uma string dentro de um ficheiro binário em uma string num ficheiro texto.

```
}else if(fileMode=="wd"){
    bts.ReadBinFile(fileName,maxbit);
    bts.WriteBinFile(fileNameOut);
}
```

Utilização do código

Para utilizar as funções dentro do programa encoder utilizamos comandos com o padrão

```
./encoder input_file output_file op
```

Para o encode

```
./encoder input_file.txt output_file.bin e maxbits
```

sendo *maxbits* um número entre 0 e 64, com o valor default 64

Para o decode

```
./encoder input_file.bin output_file.txt d maxbits
```

sendo *maxbits* um número entre 0 e 64, com o valor default 64

Para converter uma string texto em binário

```
./encoder input_file.txt output_file.bin wr
```

Para converter uma string binária em formato texto

```
./encoder input_file.bin output_file.txt wd
```

Resultados

Para testar as funções foi criado o ficheiro testfile.txt com o valor binário dos números hexadecimais A7 e C0.
Utilizando o comando

```
./encoder testfile.txt resultado.bin e
```

Vai resultar na criação de um ficheiro binário com a tradução destes valores em binário.
Utilizando o comando

```
./encoder resultado.bin decode.txt d
```

Vai ser criado um ficheiro texto com os valores traduzidos do ficheiro binário.
Em ambos estes comandos podemos incluir um valor de bits máximos que queremos que sejam traduzidos.

Exercício 7

Código

No exercício 7 criamos um *lossy codec* e *decoder* baseados em *Discrete Cosine Transform* (DCT).

```
fftw_plan plan_d = fftw_plan_r2r_1d(BlockSize, x.data(), x.data(), FFTW_REDFT10, FFTW_ESTIMATE);
for(size_t n = 0 ; n < nBlocks ; n++){
    for(size_t k = 0 ; k < BlockSize ; k++)
        x[k] = samples[(n * BlockSize + k) * nChannels + 0];

    fftw_execute(plan_d);
    for(size_t k = 0 ; k < BlockSize ; k++)
        x_dct[0][n * BlockSize + k] = x[k] / (BlockSize << 1);
}
```

Depois quantiza-mos os coeficientes e eles são escritos em binário num ficheiro binário usando a classe BitStream:

```
for (int i = 0; i < x_dct[0].size(); i++){
    int quantizedCoe = round((x_dct[0][i] / delta));
    bts.writeBits(intToBin(quantizedCoe,sampleBits));
}
```

São escritos também os parâmetros necessários, como tamanho dos blocos para DCT, delta usado para quantização, e quantos bits ocupa cada valor, para descodificar o ficheiro binário para som.

```
BitStream bts(outputFile,"w");
bts.writeBits(intToBin(sndFile.samplerate(),17));
bts.writeBits(intToBin(BlockSize,16));
bts.writeBits(intToBin(delta,16));
bts.writeBits(intToBin(sampleBits,16));
```

No *decoder* usamos a classe BitStream para extrair os parâmetros necessários para a descodificação:

```
int sampleRate = binToInt(bts.readBits(17));
int blockSize = binToInt(bts.readBits(16));
int delta = binToInt(bts.readBits(16));
int sampleBits = binToInt(bts.readBits(16));
```

Revertemos a quantização:

```
for(size_t n = 0 ; n < numberBlocks ; n++){
    for(size_t k = 0 ; k < blockSize ; k++)
        x_dct[0][n * blockSize + k] = (binToInt(bts.readBits(sampleBits))*delta);
}
```

Depois invertemos a DCT e escrevemos para um ficheiro de som:

```
fftw_plan plan_i = fftw_plan_r2r_1d(blockSize, x.data(), x.data(), FFTW_REDFT01, FFTW_ESTIMATE);
for(size_t n = 0 ; n < numberBlocks ; n++){
    for(size_t k = 0 ; k < blockSize ; k++)
        x[k] = x_dct[0][n * blockSize + k];

    fftw_execute(plan_i);
    for(size_t k = 0 ; k < blockSize ; k++)
        samples[(n * blockSize + k)] = static_cast<short>(round(x[k]));
}

sndFileOut.writef(samples.data(),samples.size());
```

Utilização do código

Para usar o codec com o comando abaixo, o ficheiro de som que se pretende comprimir no lugar de "*sample.wav*", e o ficheiro destino para a compressão no lugar de "*codedBin.bin*". Há também três parâmetros opcionais, tamanho de bloco que será usado no cálculo do DCT, o delta usado para a quantização e quantos bits cada valor ocupará no ficheiro destino.

```
./lossy_codec sample.wav codedBin.bin [-s BlockSize (256)] [-d delta (500)] [-b bits per sample (7)]
```

No decoder o comando como primeiro argumento o ficheiro destino do codec e o segundo nome do novo ficheiro de som.

```
./sound_decoder codedBin.bin newSample.wav
```

Resultados

Usando este programa com "*sample.wav*" obtemos um ficheiro binário codificado com um rácio de compressão de 4.5. Usando o decoder obtemos de volta um ficheiro de som com um nível de degradação aceitável.