UNIVERSIDADE DE AVEIRO

Project 1 - Group 2

Rui Lameiras 102817, Rafael Santos 98466, Adalberto Vaz do Rosário 105589

2023

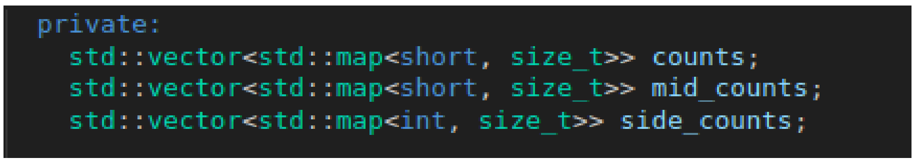
Sumário

1 Information related to the work

Every member of the group participated equaly.
Our code is placed in: <https://github.com/RuiLs1234/IC>

2 Exercise 1

In this exercise we need to create something to store the values so we created vectors to each one of the channels:



```
private:
  std::vector<std::map<short, size_t>> counts;
  std::vector<std::map<short, size_t>> mid_counts;
  std::vector<std::map<int, size_t>> side_counts;
```

Figura 1 – Private vetors used

Then, using the model given by the teachers we built other functions based on the 'dump' function to print the values on the screen and the 'update' function to update the values on our vectors.

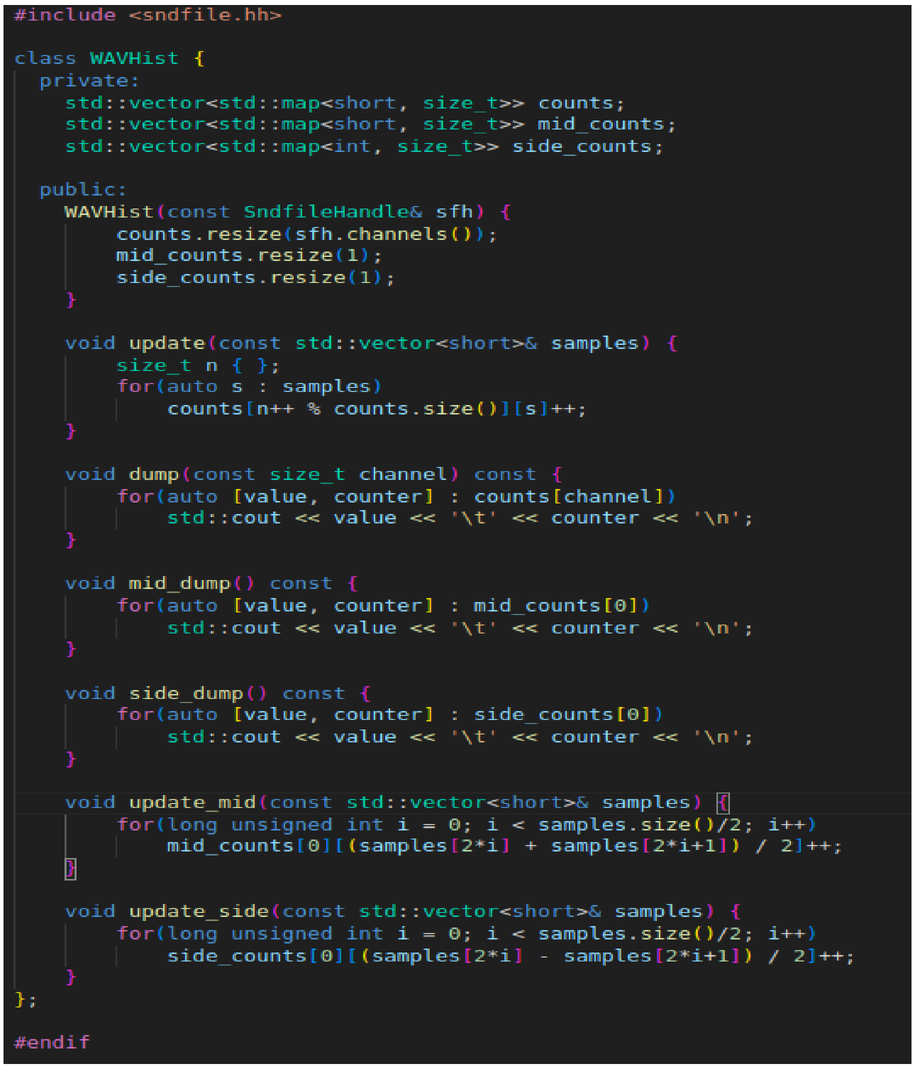
On our Main file we used this functions and made a distintion based on the usage of the program:

2.0.1 Usage

In order to run this program you should do the following: ./wav_hist <input_file> <channel | mid | side>

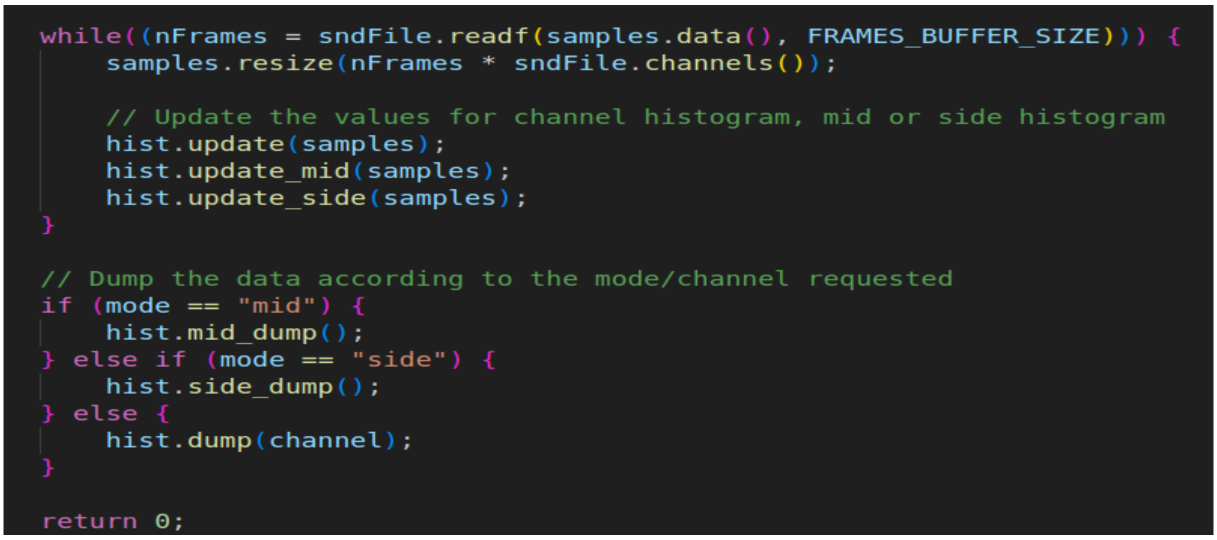
2.0.2 Results

Regarding fig 4,fig 5 and fig 6 we can see that the results are what we expected because left and right channel operate in a similar amplitude, so the mid channel will have similar amplitude because it represents the average of both channels. Looking at fig 7, we can see a not so symmetricall graph and with more concentrated values around the value 0, which is what we expected because its the difference between channels.



```cpp
#include <sndfile.hh>

class WAVHist {
  private:
    std::vector<std::map<short, size_t>> counts;
    std::vector<std::map<short, size_t>> mid_counts;
    std::vector<std::map<int, size_t>> side_counts;

  public:
    WAVHist(const SndfileHandle& sfh) {
        counts.resize(sfh.channels());
        mid_counts.resize(1);
        side_counts.resize(1);
    }

    void update(const std::vector<short>& samples) {
        size_t n { };
        for(auto s : samples)
            counts[n++ % counts.size()][s]++;
    }

    void dump(const size_t channel) const {
        for(auto [value, counter] : counts[channel])
            std::cout << value << '\t' << counter << '\n';
    }

    void mid_dump() const {
        for(auto [value, counter] : mid_counts[0])
            std::cout << value << '\t' << counter << '\n';
    }

    void side_dump() const {
        for(auto [value, counter] : side_counts[0])
            std::cout << value << '\t' << counter << '\n';
    }

    void update_mid(const std::vector<short>& samples) {
        for(long unsigned int i = 0; i < samples.size()/2; i++)
            mid_counts[0][(samples[2*i] + samples[2*i+1]) / 2]++;
    }

    void update_side(const std::vector<short>& samples) {
        for(long unsigned int i = 0; i < samples.size()/2; i++)
            side_counts[0][(samples[2*i] - samples[2*i+1]) / 2]++;
    }
};

#endif
```

Figura 2 – wav_hist.h



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

    // Update the values for channel histogram, mid or side histogram
    hist.update(samples);
    hist.update_mid(samples);
    hist.update_side(samples);
}

// Dump the data according to the mode/channel requested
if (mode == "mid") {
    hist.mid_dump();
} else if (mode == "side") {
    hist.side_dump();
} else {
    hist.dump(channel);
}

return 0;
```

Figura 3 – wav_hist.cpp

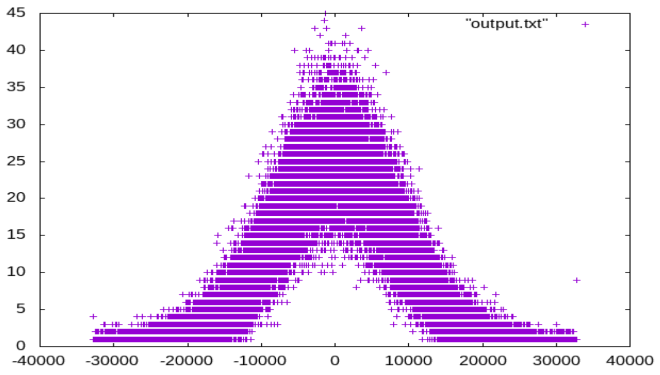


Figura 4 – Right channel

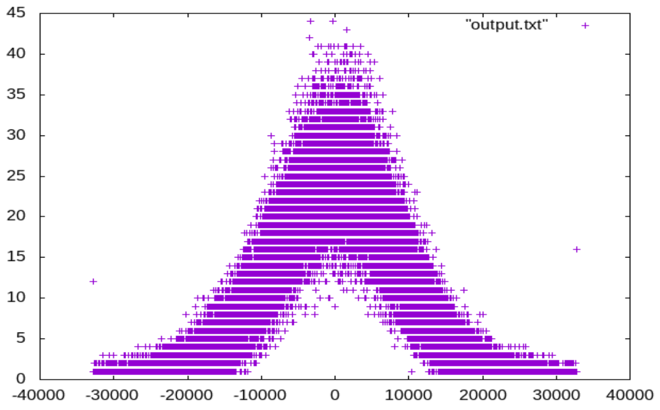


Figura 5 – Left Channel

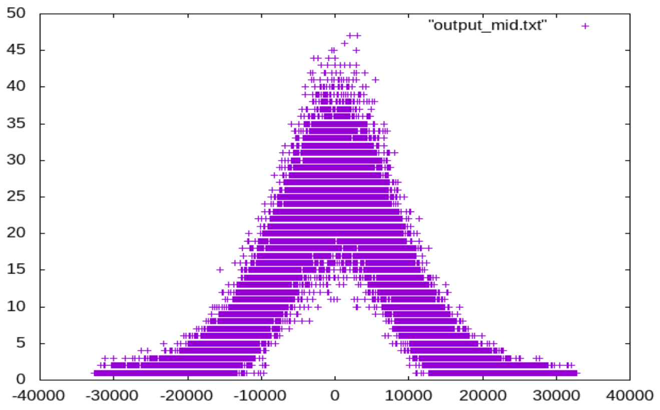


Figura 6 – Mid Channel

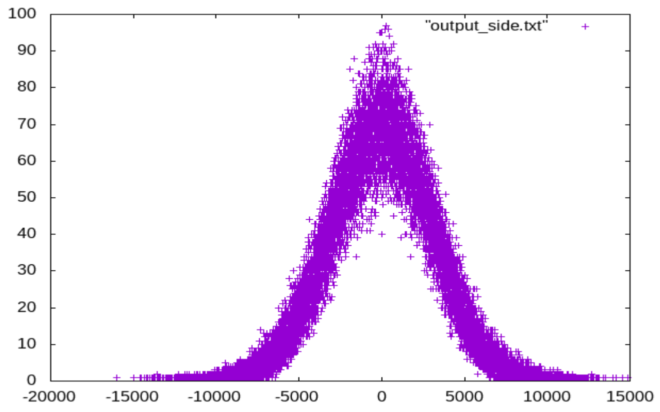
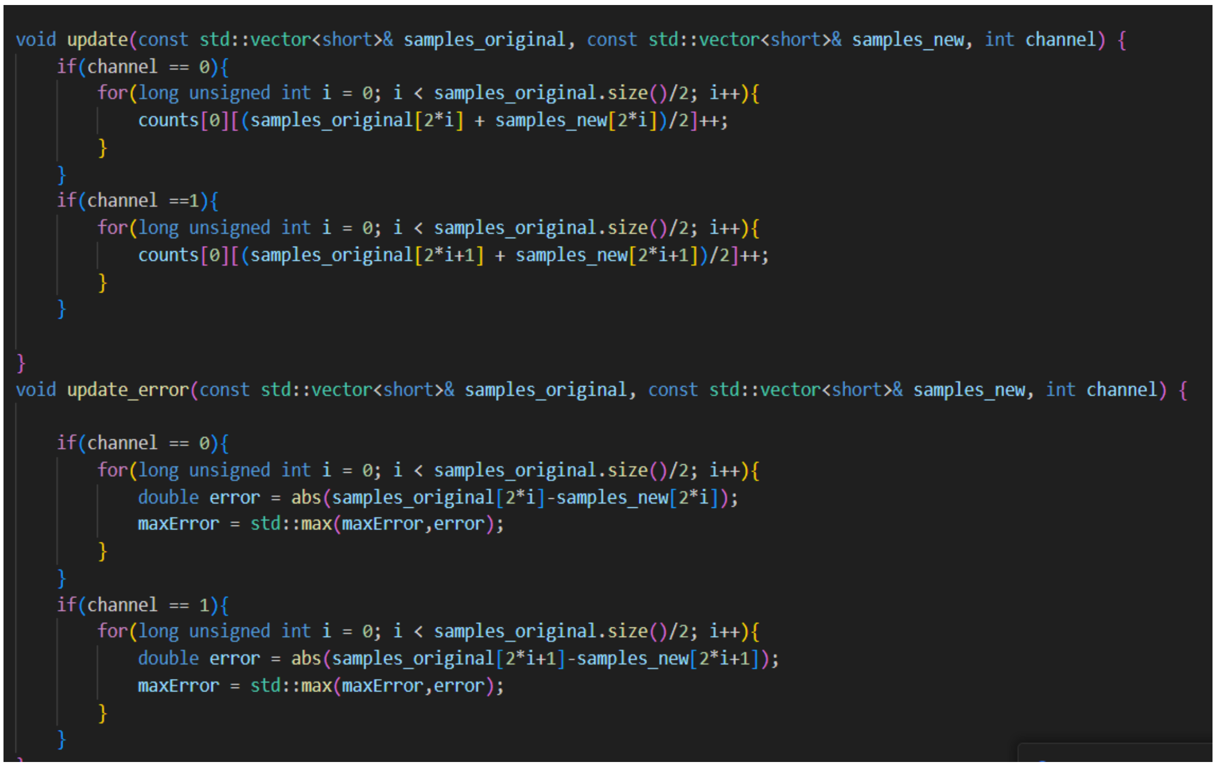


Figura 7 – Side Channel

3 Exercise 2

Using as a base exercise 1, we rebuilt it so that we could perform what is asked. To perform cmp absolute error and average mean squared we changed the update. At the end it ended up like this:



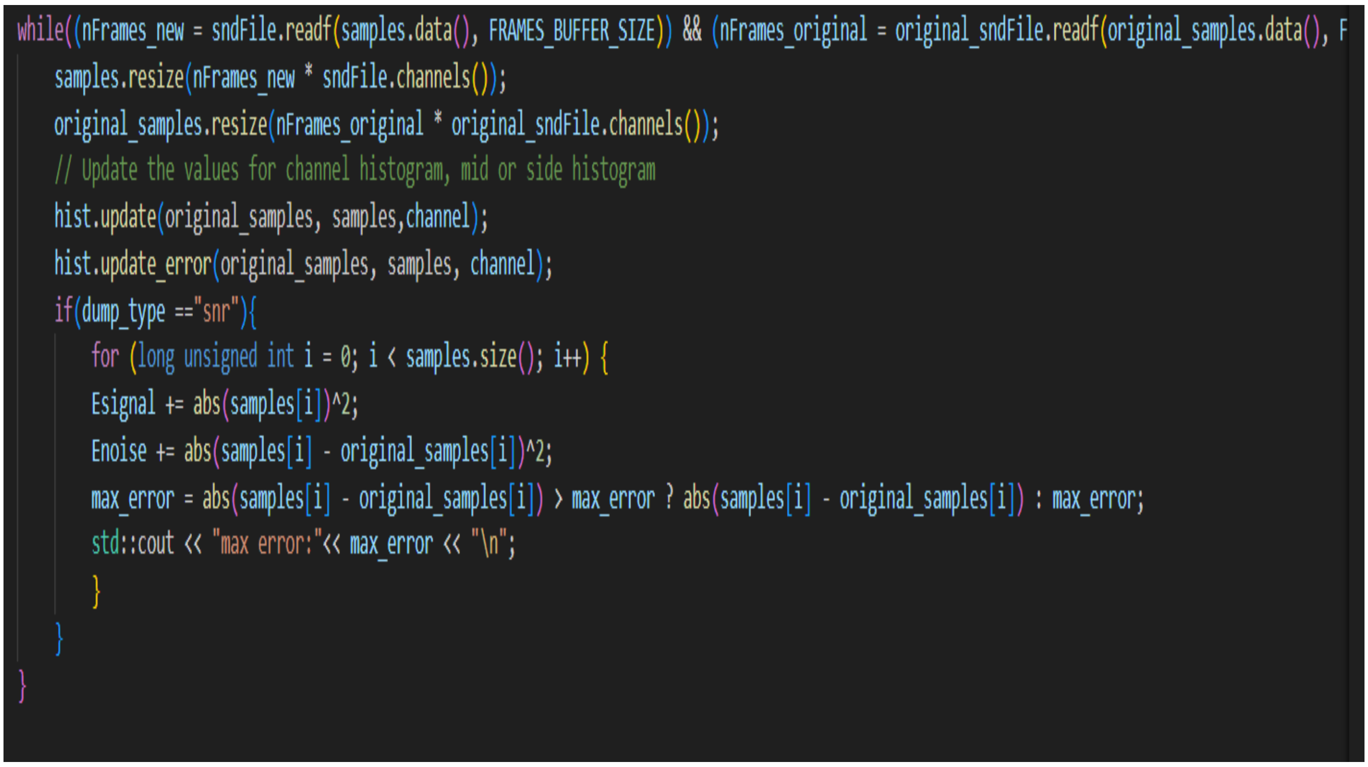
```
void update(const std::vector<short>& samples_original, const std::vector<short>& samples_new, int channel) {
    if(channel == 0){
        for(long unsigned int i = 0; i < samples_original.size()/2; i++){
            counts[0][(samples_original[2*i] + samples_new[2*i])/2]++;
        }
    }
    if(channel ==1){
        for(long unsigned int i = 0; i < samples_original.size()/2; i++){
            counts[0][(samples_original[2*i+1] + samples_new[2*i+1])/2]++;
        }
    }

}
void update_error(const std::vector<short>& samples_original, const std::vector<short>& samples_new, int channel) {

    if(channel == 0){
        for(long unsigned int i = 0; i < samples_original.size()/2; i++){
            double error = abs(samples_original[2*i]-samples_new[2*i]);
            maxError = std::max(maxError,error);
        }
    }
    if(channel == 1){
        for(long unsigned int i = 0; i < samples_original.size()/2; i++){
            double error = abs(samples_original[2*i+1]-samples_new[2*i+1]);
            maxError = std::max(maxError,error);
        }
    }
```

Figura 8 – wav_cmp update function

This is completemented by the use of the wav_hist.cpp 'while'. That 'while' was updated to accommodate the use of the two new updates and to perform the calculation of the SNR.

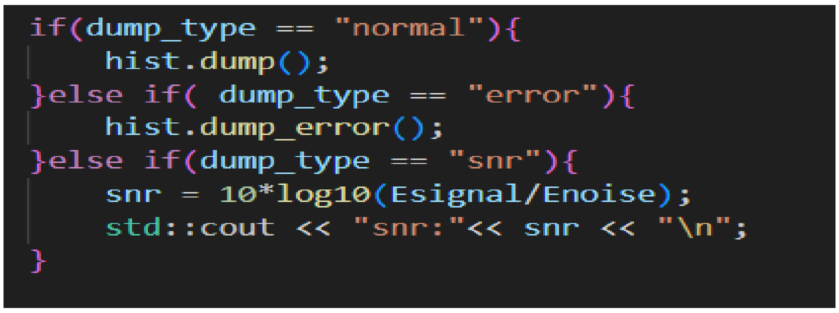
3.0.1 Usage

You run this file by doing ./wav_cmp <modified file> <original file> <channel> <what you want to know> The input: what you want to know, it's basically witch function you want to aplay. We implemented this part so that the program doesn't overflow the screen with information. The print is chosen in the following function:



```
while((nFrames_new = sndFile.readf(samples.data(), FRAMES_BUFFER_SIZE)) && (nFrames_original = original_sndFile.readf(original_samples.data(), F
    samples.resize(nFrames_new * sndFile.channels());
    original_samples.resize(nFrames_original * original_sndFile.channels());
    // Update the values for channel histogram, mid or side histogram
    hist.update(original_samples, samples,channel);
    hist.update_error(original_samples, samples, channel);
    if(dump_type =="snr"){
        for (long unsigned int i = 0; i < samples.size(); i++) {
        Esignal += abs(samples[i])^2;
        Enoise += abs(samples[i] - original_samples[i])^2;
        max_error = abs(samples[i] - original_samples[i]) > max_error ? abs(samples[i] - original_samples[i]) : max_error;
        std::cout << "max error:"<< max_error << "\n";
        }
    }
}
```

Figura 9 – wav_cmp max error

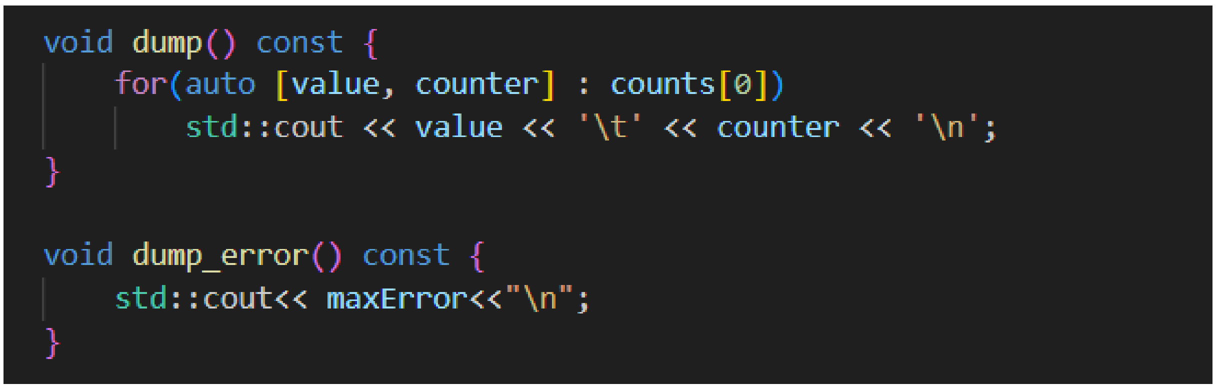


```
if(dump_type == "normal"){
    hist.dump();
}else if( dump_type == "error"){
    hist.dump_error();
}else if(dump_type == "snr"){
    snr = 10*log10(Esignal/Enoise);
    std::cout << "snr:"<< snr << "\n";
}
```

Figura 10 – Modes available to use on wav_cmp

This will then play the function 'dump' or it will play an immediate print in the SNR case, as showned in the picture above.

The modes are most explicit on what they do: type 'snr' is SNR (signal-to-noise ratio) and type 'error' is the maximum per sample absolute error. But the average mean squared error between a certain audio file and its original version it's type 'normal'. We considered normal since it's the first thing that the exercise asked for and we considered the rest as other functions that the program can do.

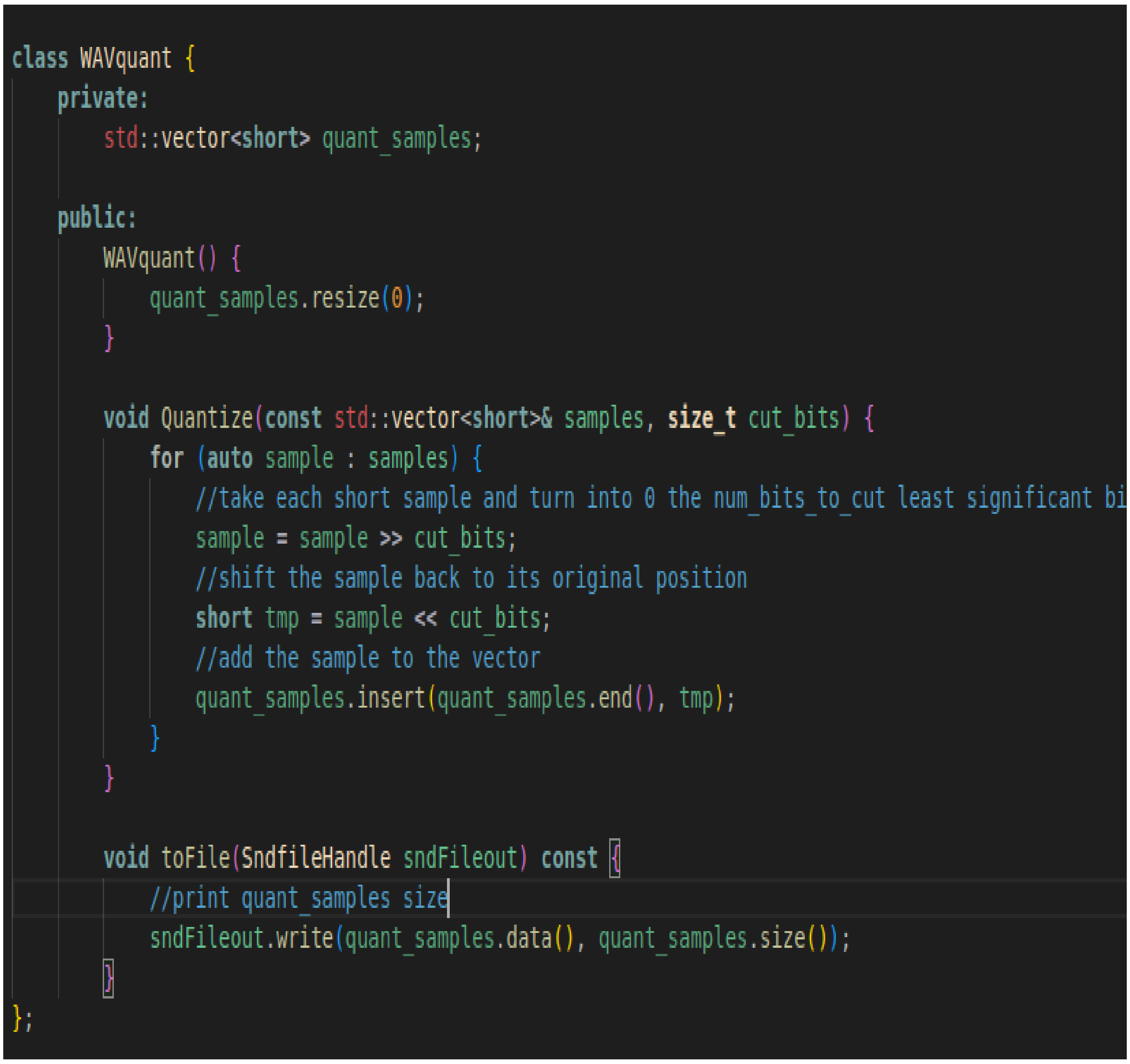


```
void dump() const {
    for(auto [value, counter] : counts[0])
        std::cout << value << '\t' << counter << '\n';
}

void dump_error() const {
    std::cout<< maxError<<"\n";
}
```

Figura 11 – dump functions on wav_cmp

4 Exercise 3

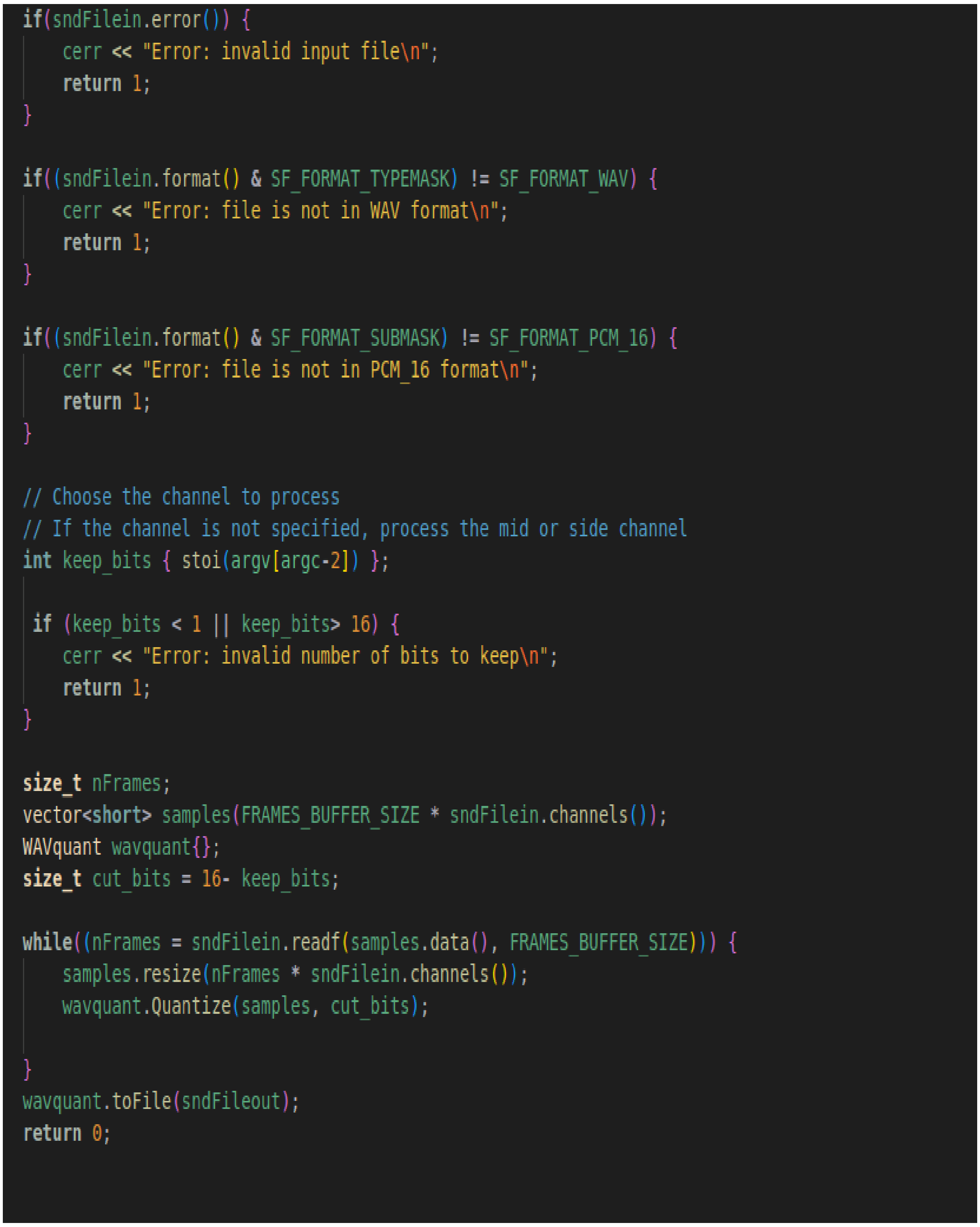
This is a simple class to make the Quantization. The function is composed by two void functions: Quantize(const std::vector<short>&Samples, size_t cut_bits) function to reduce the number of bits(cut_bits) used to represent each audio sample(Samples) and keep it in a vector(quant_samples), and toFile(SndfileHandle sndFileout) to write the quant_samples size in the a file. The wav_quant program , see 13, uses the previous



```
class WAVquant {
    private:
        std::vector<short> quant_samples;

    public:
        WAVquant() {
            quant_samples.resize(0);
        }

        void Quantize(const std::vector<short>& samples, size_t cut_bits) {
            for (auto sample : samples) {
                //take each short sample and turn into 0 the num_bits_to_cut least significant bi
                sample = sample >> cut_bits;
                //shift the sample back to its original position
                short tmp = sample << cut_bits;
                //add the sample to the vector
                quant_samples.insert(quant_samples.end(), tmp);
            }
        }

        void toFile(SndfileHandle sndFileout) const {
            //print quant_samples size
            sndFileout.write(quant_samples.data(), quant_samples.size());
        }
};
```

Figura 12 – wav_quant class

class 12 to make the quantization.



```
if(sndFilein.error()) {
    cerr << "Error: invalid input file\n";
    return 1;
}

if((sndFilein.format() & SF_FORMAT_TYPEMASK) != SF_FORMAT_WAV) {
    cerr << "Error: file is not in WAV format\n";
    return 1;
}

if((sndFilein.format() & SF_FORMAT_SUBMASK) != SF_FORMAT_PCM_16) {
    cerr << "Error: file is not in PCM_16 format\n";
    return 1;
}

// Choose the channel to process
// If the channel is not specified, process the mid or side channel
int keep_bits { stoi(argv[argc-2]) };

 if (keep_bits < 1 || keep_bits> 16) {
    cerr << "Error: invalid number of bits to keep\n";
    return 1;
}

size_t nFrames;
vector<short> samples(FRAMES_BUFFER_SIZE * sndFilein.channels());
WAVquant wavquant{};
size_t cut_bits = 16- keep_bits;

while((nFrames = sndFilein.readf(samples.data(), FRAMES_BUFFER_SIZE))) {
    samples.resize(nFrames * sndFilein.channels());
    wavquant.Quantize(samples, cut_bits);

}
wavquant.toFile(sndFileout);
return 0;
```

Figura 13 – wav_quant program

4.0.1 Usage

To run this file you should do : ./wav_quant <input file> <output file> <bits_to_keep>

4.0.2 Results

Looking to the next pictures we can see that we achieved the expected results noticing that the more bits we take from the stream the more distorted the sound gets.

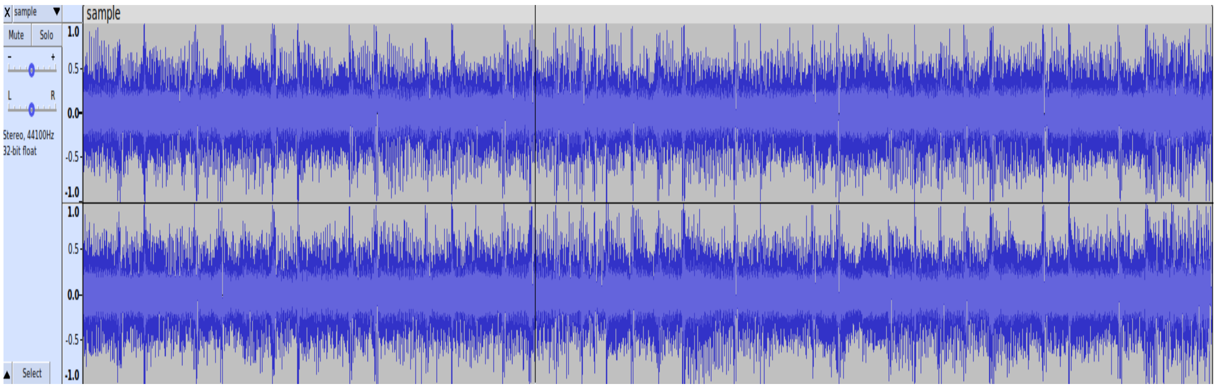


Figura 14 – normal wav

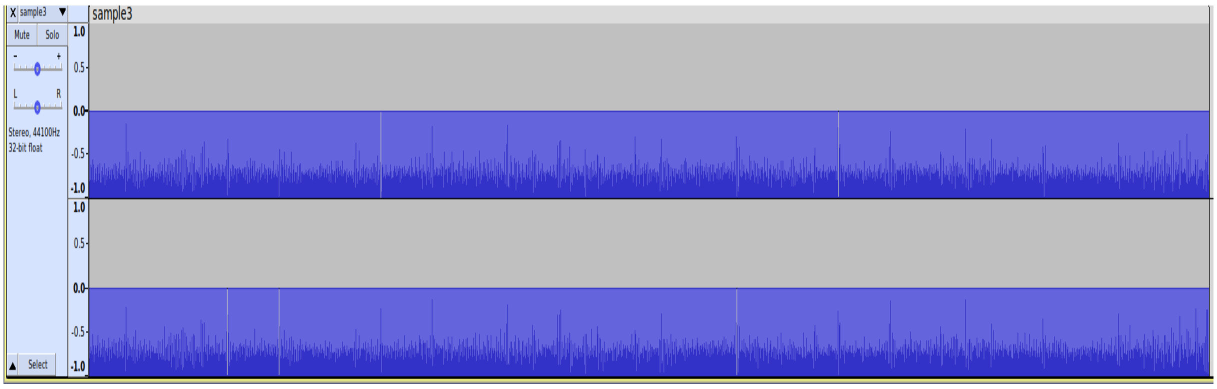


Figura 15 – 1 bit kept wav

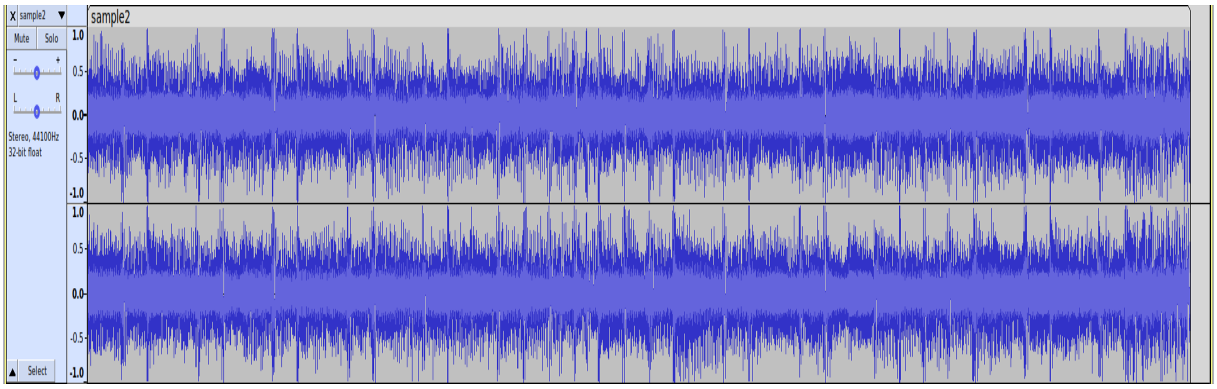


Figura 16 – 6 bits kept wav

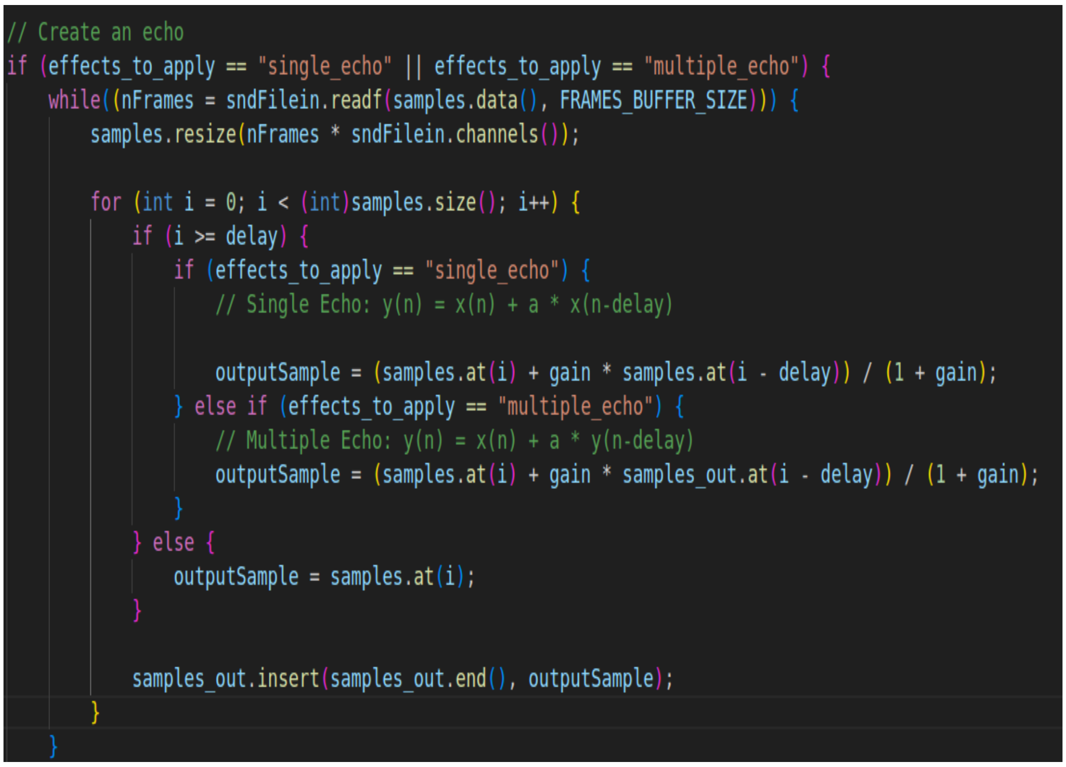
5 Exercise 4

In this exercise we implemented 3 effects : single echo, multilple echo and amplitude modulation. For these effects we used equations that were given to us in the theoretical class. Those formulas are:

$$Single\ Echo : y(n) = x(n) + \alpha \cdot x(n - \mathrm{delay}) \tag{5.1}$$

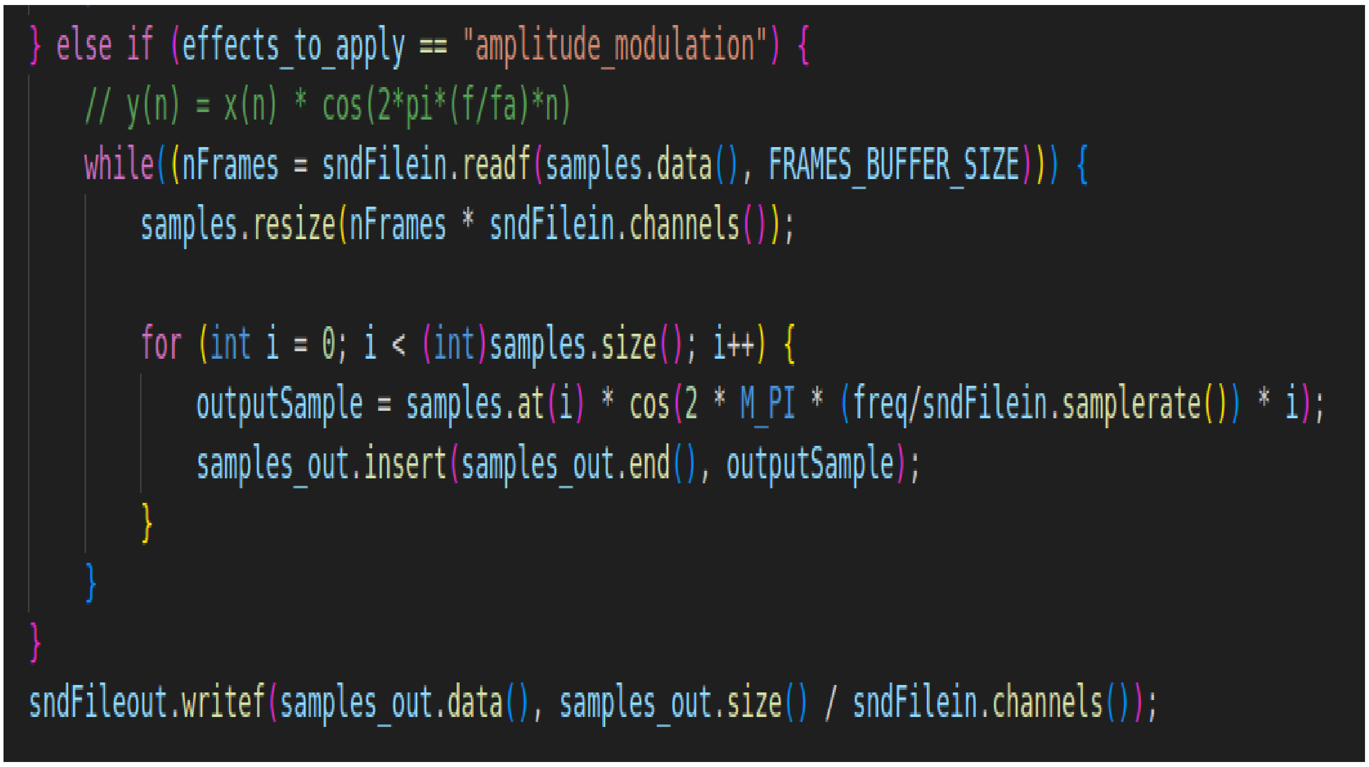
$$Multiple\ Echo : y(n) = x(n) + \alpha \cdot y(n - \mathrm{delay}) \tag{5.2}$$

$$Amplitude\ modulation : y(n) = x(n) \cdot \cos\left(2 \cdot \pi \cdot \left(\frac{f}{f_a}\right) \cdot n\right) \tag{5.3}$$



```cpp
// Create an echo
if (effects_to_apply == "single_echo" || effects_to_apply == "multiple_echo") {
    while((nFrames = sndFilein.readf(samples.data(), FRAMES_BUFFER_SIZE))) {
        samples.resize(nFrames * sndFilein.channels());

        for (int i = 0; i < (int)samples.size(); i++) {
            if (i >= delay) {
                if (effects_to_apply == "single_echo") {
                    // Single Echo: y(n) = x(n) + a * x(n-delay)

                    outputSample = (samples.at(i) + gain * samples.at(i - delay)) / (1 + gain);
                } else if (effects_to_apply == "multiple_echo") {
                    // Multiple Echo: y(n) = x(n) + a * y(n-delay)
                    outputSample = (samples.at(i) + gain * samples_out.at(i - delay)) / (1 + gain);
                }
            } else {
                outputSample = samples.at(i);
            }

            samples_out.insert(samples_out.end(), outputSample);
        }
    }
```

Figura 17 – Single and multiple echos



```
} else if (effects_to_apply == "amplitude_modulation") {
    // y(n) = x(n) * cos(2*pi*(f/fa)*n)
    while((nFrames = sndFilein.readf(samples.data(), FRAMES_BUFFER_SIZE))) {
        samples.resize(nFrames * sndFilein.channels());

        for (int i = 0; i < (int)samples.size(); i++) {
            outputSample = samples.at(i) * cos(2 * M_PI * (freq/sndFilein.samplerate()) * i);
            samples_out.insert(samples_out.end(), outputSample);
        }
    }
}
sndFileout.writef(samples_out.data(), samples_out.size() / sndFilein.channels());
```

Figura 18 – Amplitude Modulation

5.0.1 Usage

To run the program you should do: ./wav_effects <input file> <output_file> <amplitude_modulation>.

5.0.2 Results

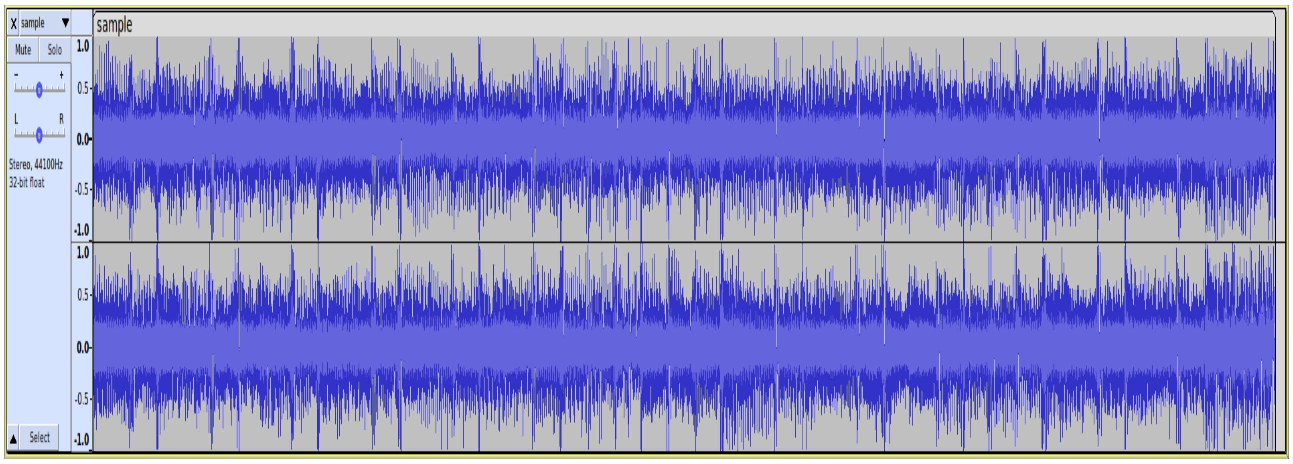


Figura 19 – sample.wav

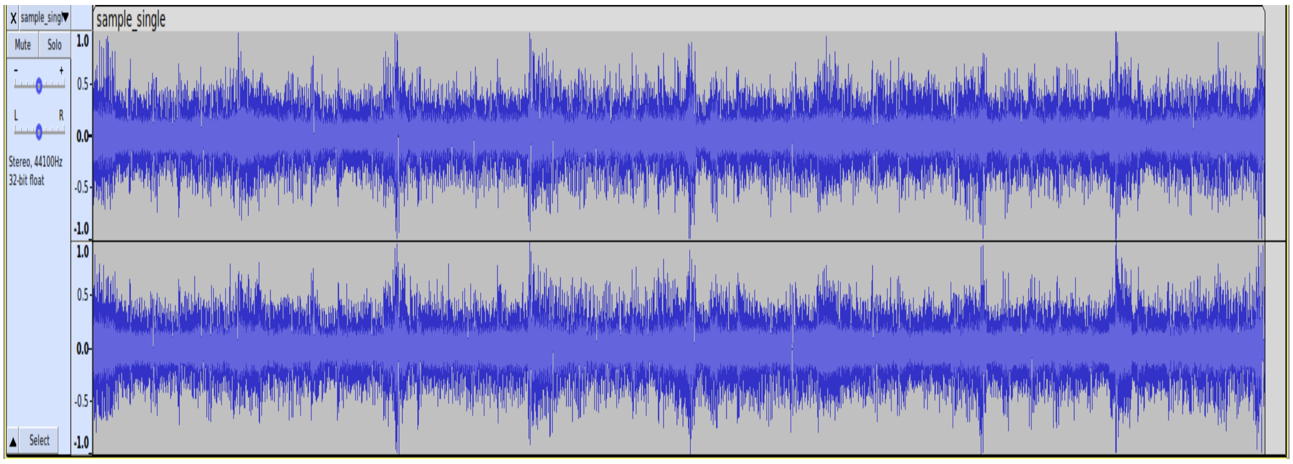


Figura 20 – single_echo

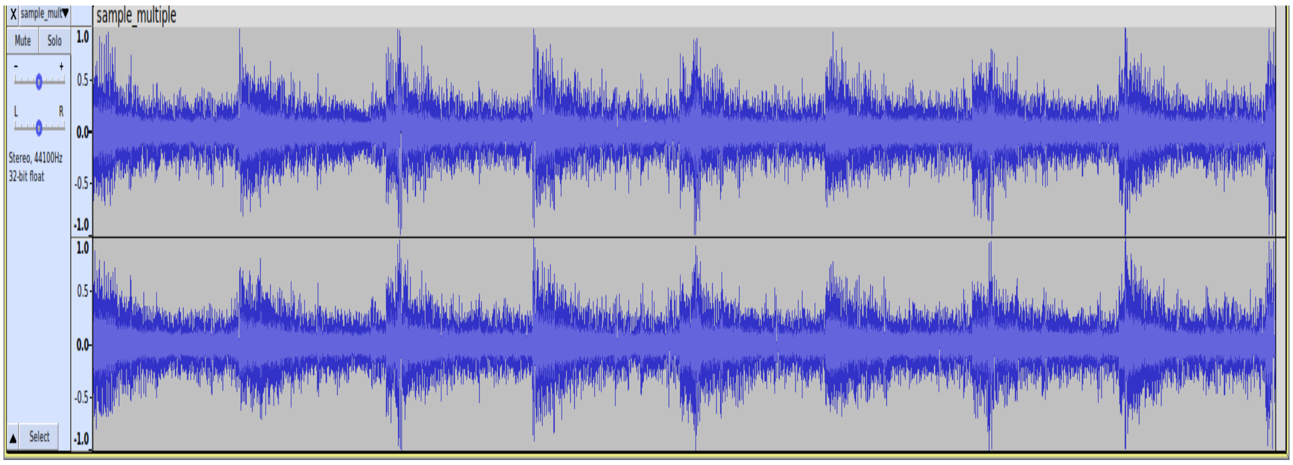


Figura 21 – multiple_echo

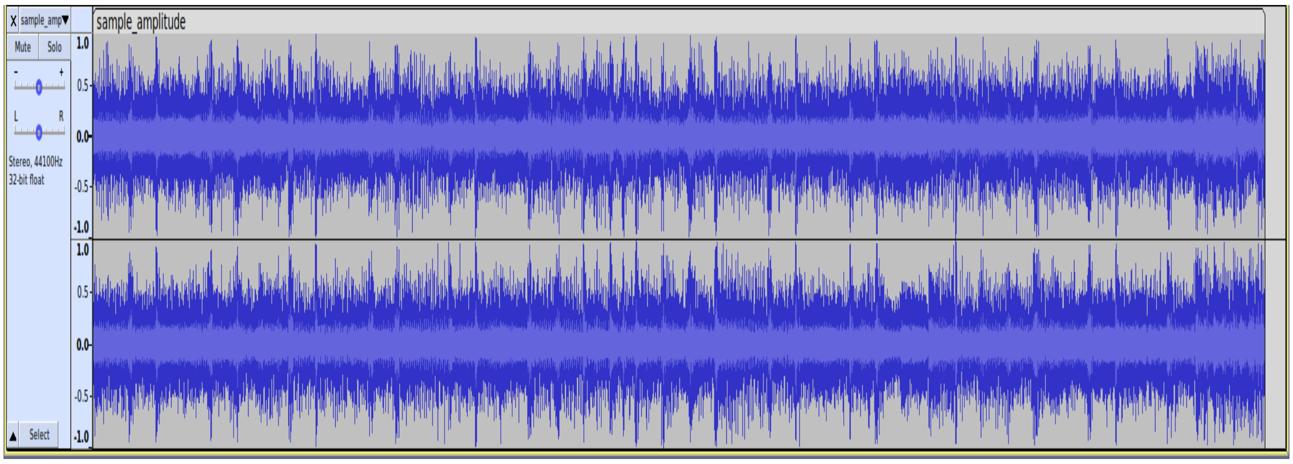
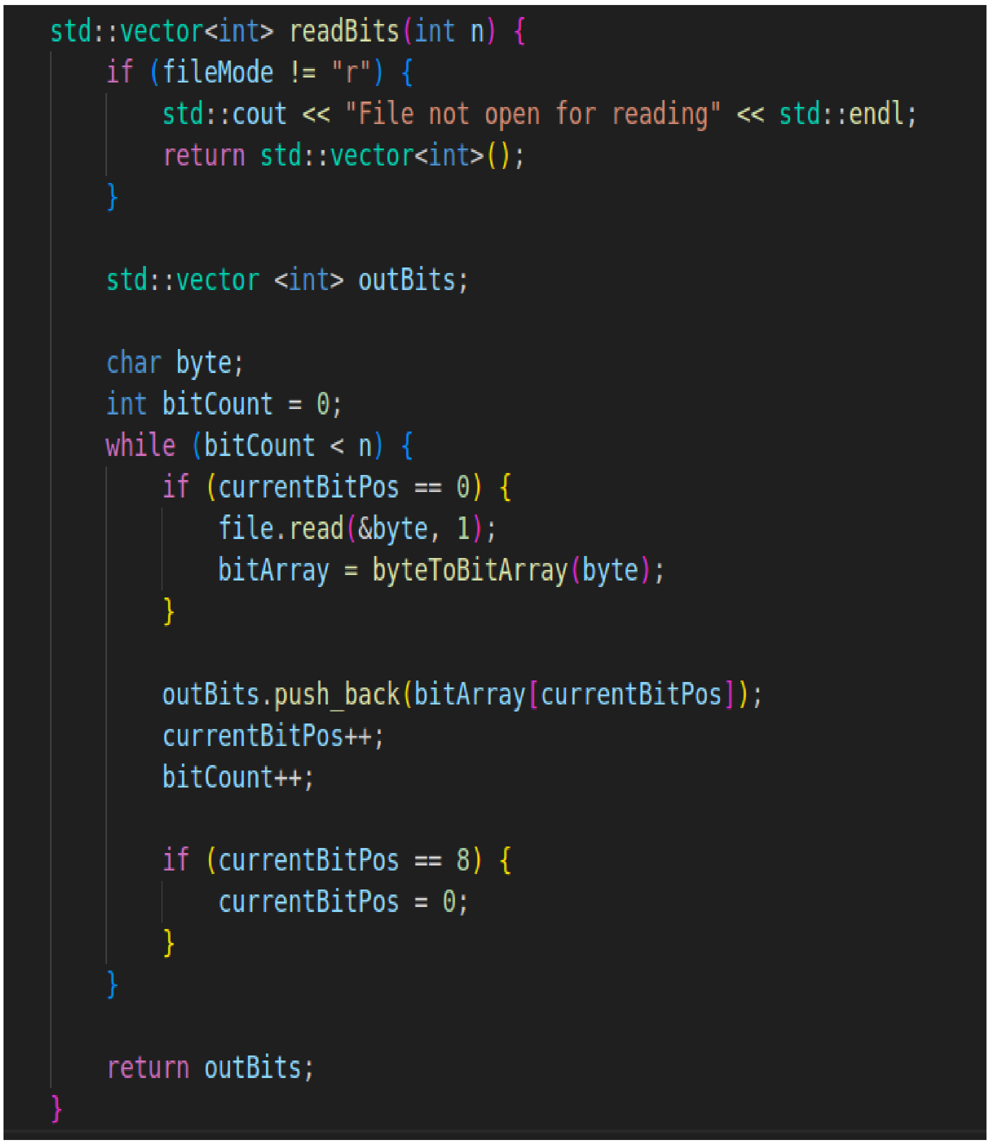


Figura 22 – amplitude modulation

Between this four figures above we can see that are clearly changes in the audio. We applied a gain of 1.5 and a frequency of 20000Hz to the files from single and multiple echo, the higher the frequency the higher the delay of echo and the higher the gain equals a more audible echo. On the modulation one we only applied a frequency of 20000Hz.

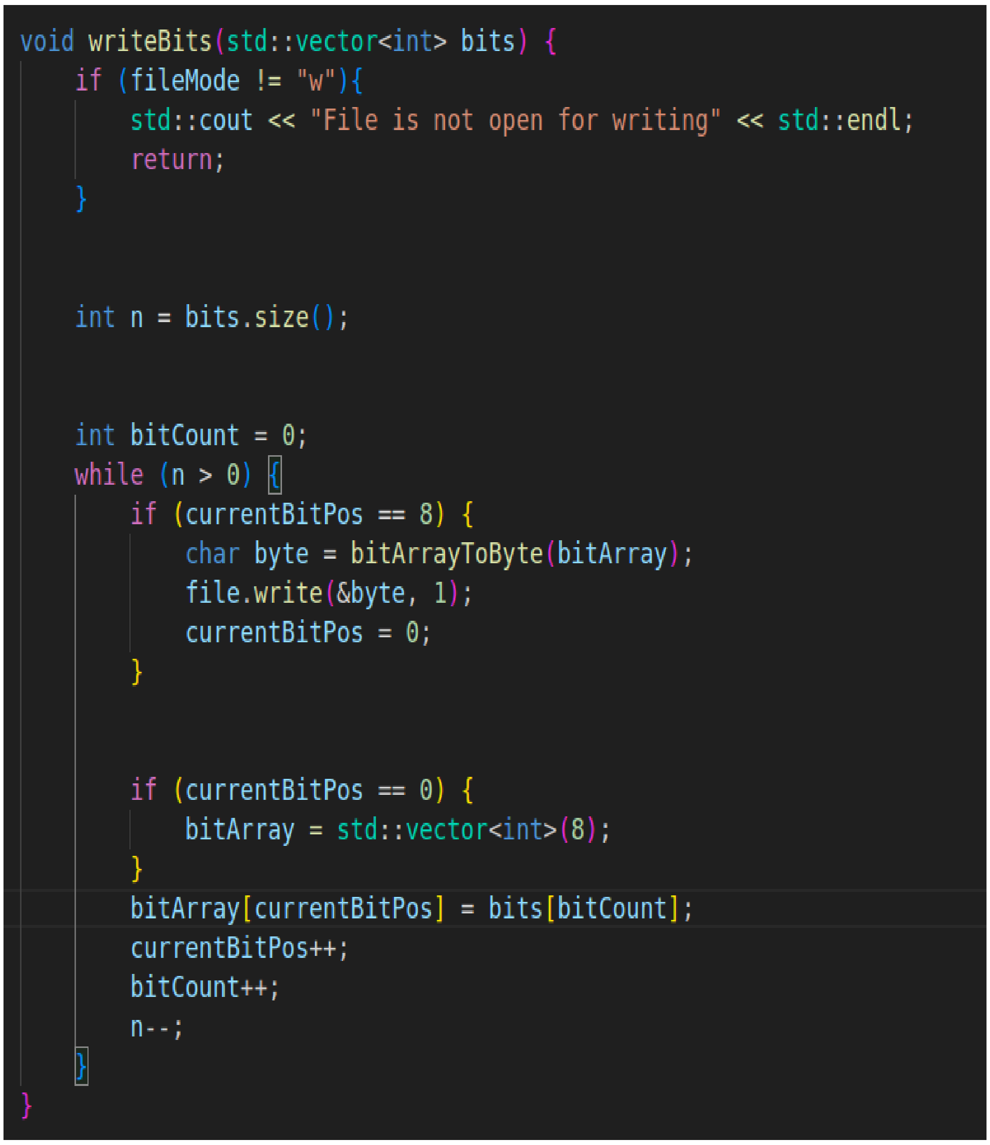
6 Exercise 5

In this exercise we implement the BitStream class. It reads bits from the file and writes them to another file. This class allows you to write one byte at a time, if the original length of the file is not 1 byte (8 bits) before writing it will be completed with some zeros at the end. The first time we implemented a version of readBits and writeBits without a vector, but it didn't work very well, so we implemented another one with a vector.



```
std::vector<int> readBits(int n) {
    if (fileMode != "r") {
        std::cout << "File not open for reading" << std::endl;
        return std::vector<int>();
    }

    std::vector <int> outBits;

    char byte;
    int bitCount = 0;
    while (bitCount < n) {
        if (currentBitPos == 0) {
            file.read(&byte, 1);
            bitArray = byteToBitArray(byte);
        }

        outBits.push_back(bitArray[currentBitPos]);
        currentBitPos++;
        bitCount++;

        if (currentBitPos == 8) {
            currentBitPos = 0;
        }
    }

    return outBits;
}
```

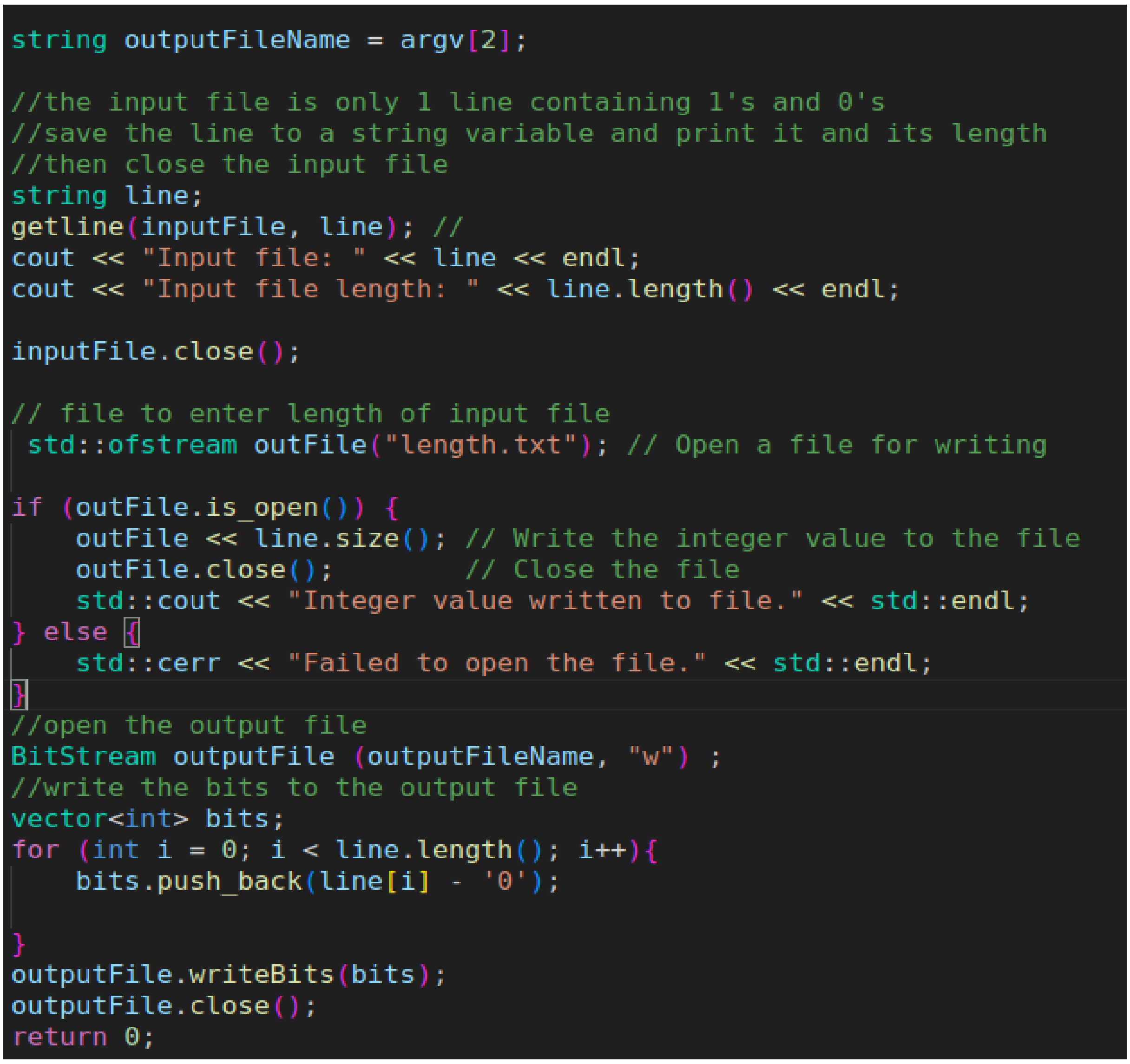
Figura 23 – BitStream :readBits



```cpp
void writeBits(std::vector<int> bits) {
    if (fileMode != "w"){
        std::cout << "File is not open for writing" << std::endl;
        return;
    }


    int n = bits.size();


    int bitCount = 0;
    while (n > 0) {
        if (currentBitPos == 8) {
            char byte = bitArrayToByte(bitArray);
            file.write(&byte, 1);
            currentBitPos = 0;
        }


        if (currentBitPos == 0) {
            bitArray = std::vector<int>(8);
        }
        bitArray[currentBitPos] = bits[bitCount];
        currentBitPos++;
        bitCount++;
        n--;
    }
}
```

Figura 24 – BitStream :writeBits

7 Exercise 6



```
string outputFileName = argv[2];

//the input file is only 1 line containing 1's and 0's
//save the line to a string variable and print it and its length
//then close the input file
string line;
getline(inputFile, line); //
cout << "Input file: " << line << endl;
cout << "Input file length: " << line.length() << endl;

inputFile.close();

// file to enter length of input file
 std::ofstream outFile("length.txt"); // Open a file for writing

if (outFile.is_open()) {
    outFile << line.size(); // Write the integer value to the file
    outFile.close();        // Close the file
    std::cout << "Integer value written to file." << std::endl;
} else {
    std::cerr << "Failed to open the file." << std::endl;
}
//open the output file
BitStream outputFile (outputFileName, "w") ;
//write the bits to the output file
vector<int> bits;
for (int i = 0; i < line.length(); i++){
    bits.push_back(line[i] - '0');

}
outputFile.writeBits(bits);
outputFile.close();
return 0;
```

Figura 25 – Encoder

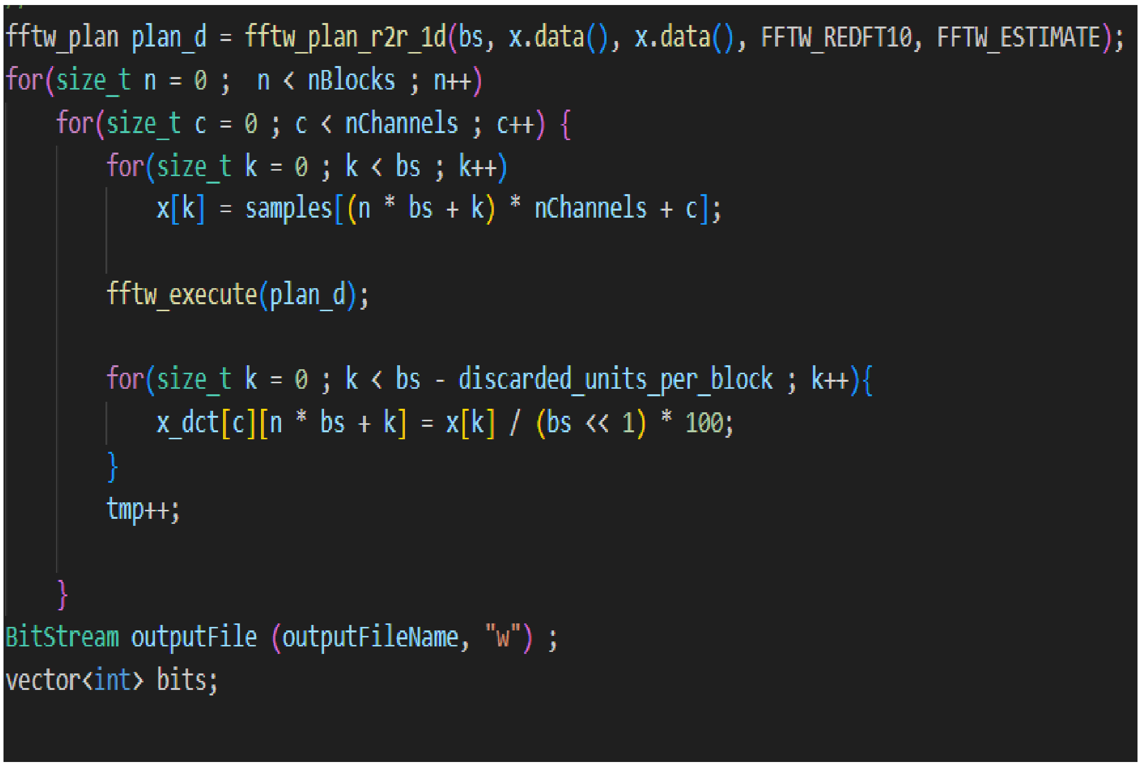
So that the contents of the encoder and decoder were equal we had to take the extra zeros we had to put to fullfill a byte, whenever that was the case. Our solution was to store in a file the number of bits that were in the file before we fullfil the whole byte. Then, at the decoder part we read the file we created on the encoder and take the number written in that file. While writing on the decoder output file it will only write the number of bits that were written initially on the input file of the encoder.

7.0.1 Usage

To use the encoder we have to do: ./encoder <input file> <output file>. To use the decoder we have to do: ./decoder <input file> <output file>.

8 Exercise 7

To do exercice 7 we created two programs. One encode and one decoder, following the meganic from before. The encoder is call lossy encoder. Lossy encoder recives an audio file via arguments. That will also recive via arguments the e output binary file, the blocksize and the discarded units per block. Afther running the program, it will directly apply the Discrete Cosine Transform (DCT) to the samples of the input audio file, keeping just blockSize-discarded units per block, which means it will only keep the most important frequencies of the audio file, in this case the lower ones.

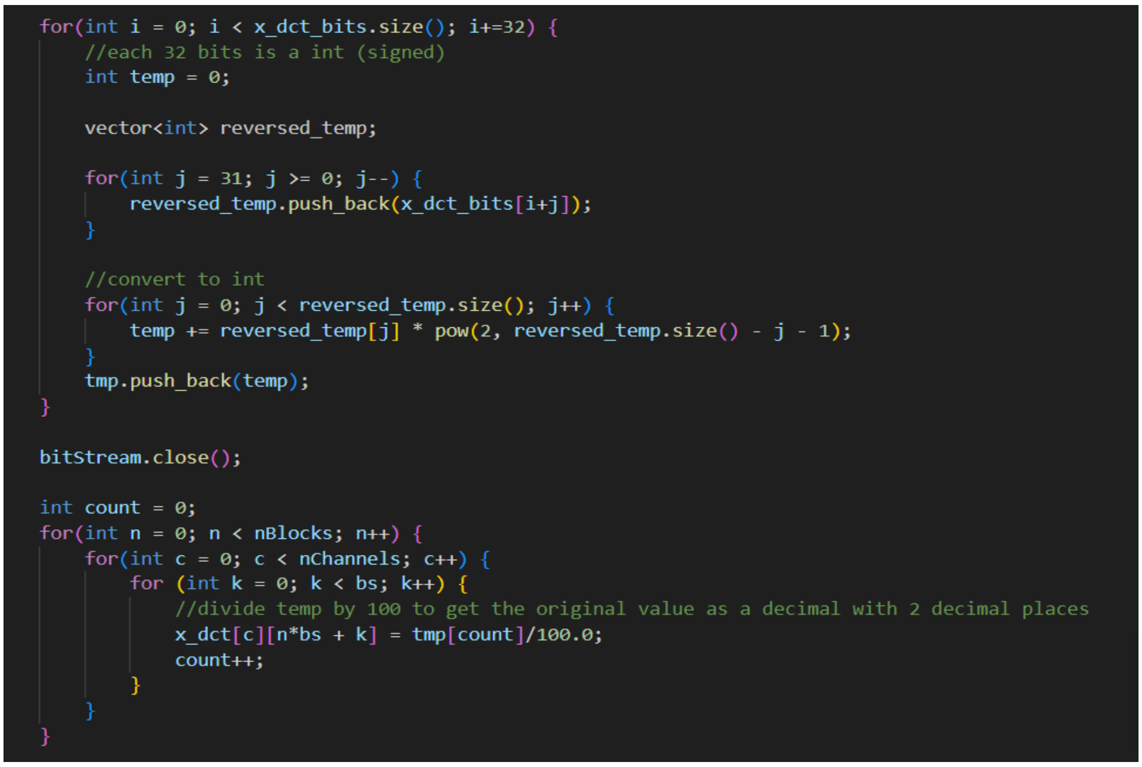


```
fftw_plan plan_d = fftw_plan_r2r_1d(bs, x.data(), x.data(), FFTW_REDFT10, FFTW_ESTIMATE);
for(size_t n = 0 ;  n < nBlocks ; n++)
    for(size_t c = 0 ; c < nChannels ; c++) {
        for(size_t k = 0 ; k < bs ; k++)
            x[k] = samples[(n * bs + k) * nChannels + c];

        fftw_execute(plan_d);

        for(size_t k = 0 ; k < bs - discarded_units_per_block ; k++){
            x_dct[c][n * bs + k] = x[k] / (bs << 1) * 100;
        }
        tmp++;

    }
BitStream outputFile (outputFileName, "w") ;
vector<int> bits;
```

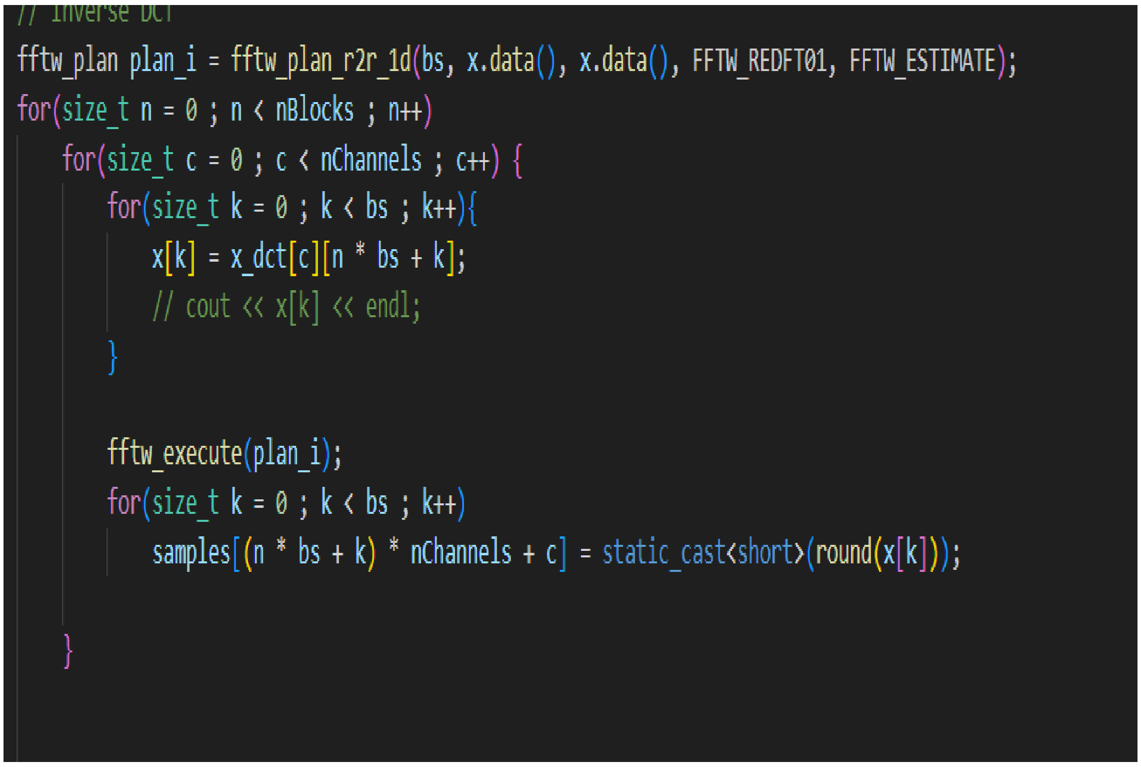
The decoder file is call lossy decoder. So that we can use it to decode the binary file we will need more informations that we need to keep besides the coefficients of the DCT (x dct). For example the number of blocks, the number of channels, the sampleRate of the original file and the number of frames.

This values are kept as a header of the binary file, before the storing process of the coefficients. For each of the kept values, either part of the header or the x_dct, there must occur a correct conversion to binary (having in mind the representation resolution: 16 bits for header values, except the number of frames, and 32 bits for the DCT coefficients). It's also relevant that this coefficients are saved after a multiplication by 100, so that, in the decode process, the inverse DCT is made with a greater resolution (numbers with 2 decimal places) when compared to integer values.

Lossy decoder recives arguments like lossy encoder. It receives the encoded binary file and a output audio file that's supposed to be created and written after the decode process. The first step it needs to preform is to read the initial bits of the header in the binary file and convert them into the respective integers (blockSize, nBlocks, nChannels, sampleRate and nFrames). After that, before starting the decoding, it must be created an output file with the given name, number of channels and sample rate indicated.


```
for(int i = 0; i < x_dct_bits.size(); i+=32) {
    //each 32 bits is a int (signed)
    int temp = 0;

    vector<int> reversed_temp;

    for(int j = 31; j >= 0; j--) {
        reversed_temp.push_back(x_dct_bits[i+j]);
    }

    //convert to int
    for(int j = 0; j < reversed_temp.size(); j++) {
        temp += reversed_temp[j] * pow(2, reversed_temp.size() - j - 1);
    }
    tmp.push_back(temp);
}

bitStream.close();

int count = 0;
for(int n = 0; n < nBlocks; n++) {
    for(int c = 0; c < nChannels; c++) {
        for (int k = 0; k < bs; k++) {
            //divide temp by 100 to get the original value as a decimal with 2 decimal places
            x_dct[c][n*bs + k] = tmp[count]/100.0;
            count++;
        }
    }
}
```

All of the remaining bits of the file belong to the encoded x dct, therefore they are read e and converted to their respective values (32 bits to integer and, after that, to doubles with 2 decimal places for a bigger resolution). Having the x dct and the rest of the values of the header, it's possible to make the inverted DCT and recontruct the samples array in order to write it in the output audio file.



```
// inverse DCT
fftw_plan plan_i = fftw_plan_r2r_1d(bs, x.data(), x.data(), FFTW_REDFT01, FFTW_ESTIMATE);
for(size_t n = 0 ; n < nBlocks ; n++)
    for(size_t c = 0 ; c < nChannels ; c++) {
        for(size_t k = 0 ; k < bs ; k++){
            x[k] = x_dct[c][n * bs + k];
            // cout << x[k] << endl;
        }

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

    }
```

8.0.1 Usage

The lossy encoder can be used like this: ./lossy_encoder <input file> <output file> <blockSize> < discard units per Block>. Where the input file is an audio file (tested with .wav), the output file is a binary file, the blockSize and discarded units per block are both integers.

The lossy decoder can be used like this: ./lossy_decoder <input file> <output file>. Where the input file is a binary file and the output is an audio file (tested with .wav).