University of Aveiro

DETI

Information and Coding Project nº1

universidade de aveiro

Gonçalo Silva (103244) goncalolsilva@ua.pt
Samuel Teixeira (103325) steixeira@ua.pt
Tiago Alves (104110) tiagojba9@ua.pt

October 29, 2023

Index

List of Figures

Chapter 1

Introduction

The report here conducted follows the structure of the work guide. Part I approaches exercises 1-4. Part II handles exercises 5-6 and Part III describes Exercise 7. This project was developed using the GitHub platform for version control, it can be accessed at `https://github.com/detiuaveiro/ic_gts`

Chapter 2

Part I

2.1 Exercise 1

2.1.1 Modifications or Insertions in the code

To store and operate the additional information needed to execute the program (select channel and bin size), the following aspects of the main function located in the wav_hist.cpp file needed to be altered. Mainly the save and check of the selected channel and bin size:

```cpp
if (argc < 4) {
    cerr << "Usage: " << argv[0] << " <input file> <channel> <binSize>\n";
    return 1;
}

SndfileHandle sndFile{argv[1]};
(...)

std::cout << "Number of channels: " << sndFile.channels() << '\n';
int channel{stoi(argv[2])};
if (channel == 1 && channel != 0) {
        cerr << "Error: invalid channel requested\n";
        return 1;
} else if (channel >= sndFile.channels() + 2) {
    cerr << "Error: invalid channel requested\n";
    return 1;
}

size_t bFactor = stoi(argv[3]);
if (bFactor == 0 || (bFactor % 2 != 0 && bFactor != 1)) {
    cerr << "Error: binningFactor needs to be positive, even or 1\n";
    return 1;
}
```

Listing 2.1: Modification in main

Regarding the WAVHist class, it was decided to incorporate the Mid and Side channels into the already defined counts vector map. This meant that we needed to modify the constructor to increase the vector size, to accommodate the new channels. While at it, the the bin size of the histogram was also saved here, since it's a parameter that isn't supposed to change with the execution of the program. The constructor is as follows:

```cpp
WAVHist(const SndfileHandle& sfh, size_t bFactor = 1) {
    if (sfh.channels() > 2)
        throw std::invalid_argument(
            "WAVHist can only handle 1 or 2 channels");

    // Accounting for Mid and Side channels
    counts.resize(sfh.channels() + 2);
    nChannels = sfh.channels();

    binningFactor = bFactor;
}
```

Listing 2.2: WAVHist Constructor

The update function also needed to be altered. Besides saving the counter of the sample, this function is now able to calculate the Mid and Side channel, by adding and subtracting the left and right samples, respectfully. When it comes to binning the samples in the histogram, since we are dealing with a counter map, there is no need to resize it to accommodate the binned samples. When it comes to the index of the sample, we decided to just shift the value, which in our view simplifies the calculations since the binning Factor can only be even. The code is as follows:

```cpp
// The vector is in format: LR LR LR...
void update(const std::vector<short>& samples) {
    /*
        Constant value of bits to shift by: is (2^15-(-2^15))/
    binningFactor

        numBins = l = (INT16_MAX - INT16_MIN) / binningFactor
    */

    // take into account that when 1 is introduced, no shifts
    should be done
    short binSize = binningFactor - 1;

    size_t n{};
    for (size_t i = 0; i < samples.size(); i++) {
        short index = samples[i] >> binSize;
        counts[n++ % nChannels][index]++;
        if ((i > 0 && i % 2 != 0) && (nChannels > 1)) {
            short left = samples[i - 1] >> binSize;
            short right = samples[i] >> binSize;
            counts[2][(left - right) / 2]++;  // Side
            counts[3][(left + right) / 2]++;  // Mid
        }
    }
}
```

Listing 2.3: WAVHist update

Minor changes were needed on the dump function, mainly just the option to output the chosen channel:

```cpp
void dump(const size_t channel) const {
    if (nChannels == 1)
        std::cout << "Value" << '\t' << "Mono Count" << '\n';
    else {
        switch (channel) {
            case 0:
                std::cout << "Value" << '\t' << "Left Count" << '\n';
                break;
            (...)
        }
    }
    for (auto [value, counter] : counts[channel])
        std::cout << value << '\t' << counter << '\n';
}
```

Listing 2.4: WAVHist dump

For checking the correct values of the graphics, we created a Matlab script to view the data in a bar plot. The script is as follows:

```matlab
fileName = '../sndfile-example-bin/val.txt';
T1 = readtable(fileName,'VariableNamingRule','preserve');
header = T1.Properties.VariableNames;
splitString = strsplit(header{2}, ' ');
name = ['Plot for the ', splitString{1}, ' channel'];
yLeg = splitString{2};
xLeg = header{1};
values = T1{:,1};
counts = T1{:,2};

figure(1);
bar(values, counts);
xlabel(xLeg);
ylabel(yLeg);
title(name);
```

Listing 2.5: Histogram Visualization

2.1.2 Usage

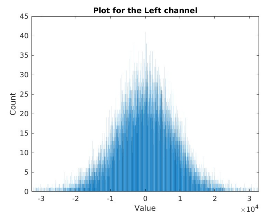
The program can be launched using the following command:

```
./wav_hist <input_file> <channel> <bin_size>
```

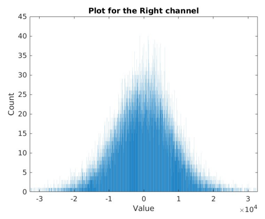
where $< input\_file >$ is the path to the .wav file (which will be tested for format and compatibility). The $< channel >$ argument corresponds to an integer representing the chosen channel, which can range from 0 to the number of channels of the file plus the Mid and Side channel (only one channel for mono audio files). Lastly, the $< bin\_size >$ argument is the number of samples that each bin will have in the histogram.

2.1.3 Results

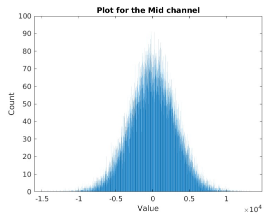
To generate the results, we used the provided sample.wav file and exported the contents to a file (redirected the outputs of the terminal to a text file) which we then loaded onto our Matlab script. The results are as follows:



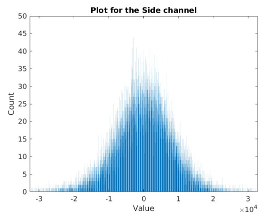
(a) Left channel



(b) Right channel



(c) Mid Channel

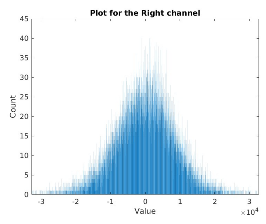


(d) Side channel

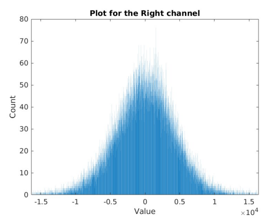
Figure 2.1: Combination of histograms for sample.wav file

The Figure 2.1 show the all the histograms generated for the sample.wav file. We can see that the Mid channel is correct, because of the increase in the counter values (mostly doubled), due to adding the left and right channel. The Side channel, being the subtraction/difference of the right and left channel, is also consistent, since both channel have very similar values, the counter is more concentrated around the 0 value.

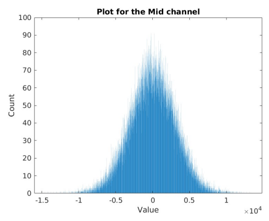
Regarding the binning of samples/coarser bins present in Figure 2.2, the results corroborate the correct functioning of our program, since we can see that in each increase of bin size, we are decreasing the number of bars by a factor of two. That effect can be observed in Figure 2.2b vs. Figure 2.2a, where the number of values is reduced by half, since each bin now corresponds to two values. The same can be said for Figure 2.2d and Figure 2.2d, with this one being a factor of 4, the histogram is much more defined and easy to observe patterns.



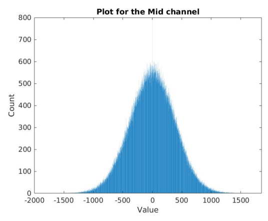
(a) Bin size of 1



(b) Bin size of 2



(c) Bin size of 1



(d) Bin size of 4

Figure 2.2: Binning of sample.wav file samples

2.2 Exercise 2

2.2.1 Modifications or Insertions in the code

This explanation is divided in three sections according to the exercise. First we explain what we did to calculate the mean squared error, than the max sample error and finally the signal-to-noise ratio.

To calculate the mean squared error (MSE) we use the following expression:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(x_i - y_i)^2 \tag{2.1}$$

For each channel we initialize vectors audio1samples and audio2samples to store the audio samples of the original and modified files. After the samples are read we initialize channelError variable that will store the cumulative squared differences between the audio samples. Now we can iterate over the samples and calculate the error difference between each sample of the two files. Then after the value is squared it can be added to channelError. The loop repeats for each channel, and the MSE for each channel is calculated and stored in the channelErrors vector.

```
for(int channel = 0; channel < numChannels; ++channel)
{
    vector<double> audio1_samples(sndFile1.frames());
    vector<double> audio2_samples(sndFile2.frames());

    sndFile1.read(audio1_samples.data(), sndFile1.frames());
    sndFile2.read(audio2_samples.data(), sndFile2.frames());

    double channelError = 0.0; //Or energy Noise

    for (long unsigned int i = 0; i < audio1_samples.size(); ++i)
    {
        double diff = audio1_samples[i] - audio2_samples[i];
        channelError += pow(diff,2);
    }

    channelErrors[channel] = sqrt(channelError / sndFile1.frames())
    ;
}
```

Listing 2.6: WAVCmp MSE for each channel

After computing the MSE for each channel, we calculate the average MSE across all channels by summing the individual channel MSE values and dividing by the total number of channels and store it in the averageError variable.

```
double averageError = 0.0;
for (int channel = 0; channel < numChannels; ++channel)
{
    averageError += channelErrors[channel];
}
averageError /= numChannels;
```

Listing 2.7: WAVCmp Average MSE of all channels

To obtain the maximum per sample absolute error we just need to add a condition statement on the iteration of the samples. If the error difference between each sample of the two files is higher than the value stored in the channelMaxError variable, the value is updated with the higher error value. After the samples loop is finished the maximum error is stored in the channelMaxErrors vector. The process is repeated for each channel, allowing us to determine the maximum per sample absolute error for the average of the channels.

```
for(int channel = 0; channel < numChannels; ++channel)
{
    ...
    double channelMaxError = 0.0;

    for (long unsigned int i = 0; i < audio1_samples.size(); ++i)
    {
        if(diff > channelMaxError){
            channelMaxError = diff;
        }
    }
    channelMaxErrors[channel] = channelMaxError;
}

double averageMaxError = 0.0;
for (int channel = 0; channel < numChannels; ++channel)
{
    averageMaxError += channelMaxErrors[channel];
}
```

```
averageMaxError /= numChannels;
```

Listing 2.8: WAVCmp Maximum per sample absolute error

Finally, to calculate the signal-to-noise ratio (SNR) we use the following expressions:

$$\mathrm{SNR} = 10 \cdot \log_{10} \frac{E_x}{E_r} \quad \mathrm{d}B(decibel) \tag{2.2}$$

$$E_x = \sum_{i=1}^{N} x_i^2 \tag{2.3}$$

$$E_r = \sum_{i=1}^{N} r_i^2 \tag{2.4}$$

The last two equations represent the energy of the signal and of the noise. In the code the energy of the noise is accumulated on the channelError variable and to accumulate the energy of the signal we created a new variable named energySignal. After getting these values the SNR of each channel is calculated. After this the average SNR of the channels can also be determined.

```
double channelError = 0.0; //Or energy Noise
double energySignal = 0.0;

for (long unsigned int i = 0; i < audio1_samples.size(); ++i)
{
    double diff = audio1_samples[i] - audio2_samples[i];
    channelError += pow(diff,2);
    energySignal += pow(audio1_samples[i],2);
}

SNRs[channel] = 10*log10(energySignal/channelError);
```

Listing 2.9: WAVCmp SNR calculation

2.2.2 Usage

The program can be launched using the following command:

```
./wav_cmp <input file> <input2 file>
```

The arguments <input_file> and <input2_file> are paths to .wav files (both will be tested for format and compatibility).

2.2.3 Results

First we used the original audio file and a quantified audio file that keeps 8 bits. Also is important to notice that the file used for testing is stereo meaning there are two channels. Below is the output obtained.

```
Channel 1 MSE: 147.347
Channel 1 Max sample error: 255
Channel 1 SNR: 35.1971
Channel 2 MSE: 147.409
Channel 2 Max sample error: 255
Channel 2 SNR: 35.4513
Average MSE of the channels: 147.378
Average Max sample error fo the channels: 255
Average SNR of the channels: 35.3242
```

On the next one we chose to only keep 5 bits for the quantified file.

```
Channel 1 MSE: 1181.91
Channel 1 Max sample error: 2047
Channel 1 SNR: 17.1122
Channel 2 MSE: 1181.36
Channel 2 Max sample error: 2047
Channel 2 SNR: 17.3742
Average MSE of the channels: 1181.63
Average Max sample error fo the channels: 2047
Average SNR of the channels: 17.2432
```

And finally only keeping 3 bits.

```
Channel 1 MSE: 4728.06
Channel 1 Max sample error: 8191
Channel 1 SNR: 5.07025
Channel 2 MSE: 4732.53
Channel 2 Max sample error: 8191
Channel 2 SNR: 5.31989
Average MSE of the channels: 4730.3
Average Max sample error fo the channels: 8191
Average SNR of the channels: 5.19507
```

It is not possible to do quantization without introducing irreversible errors. As evidenced by the values above, when we decrease the number of bits kept the SNR reduces. This reduction is due to the presence of more noise relative to the signal. A SNR value close to 0 dB represents a poor quality audio file with lots of noise. On the other hand, if we keep more bits, we can obtain a good SNR, often reaching or surprassing 20 dB. Furthermore, as the number of bits is reduced, the level of noise in the signal raises, resulting in higher values for both MSE and Maximum Sample Error.

To conduct a more in-depth analysis we generated three distinct plots: one illustrating the average Mean Squared Error (2.3), another for the average Maximum Sample Error (2.4), and a third displaying the average SNR (2.5). These three plots highlight the explanation given above.

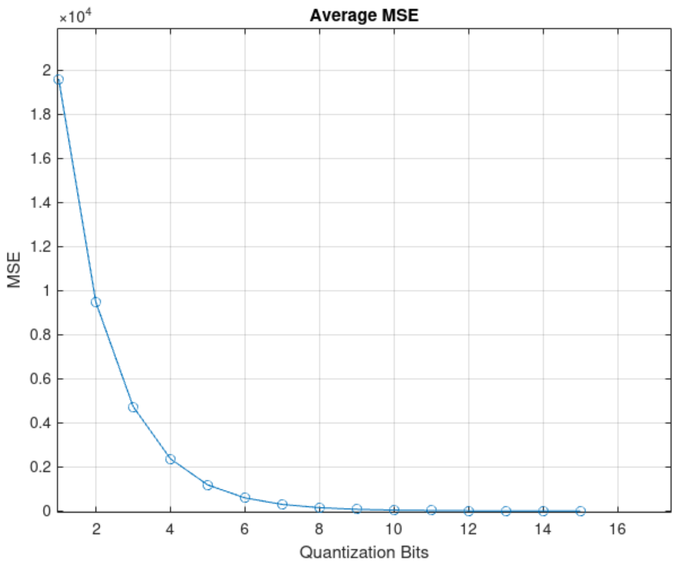


Figure 2.3: Average MSE of the channels

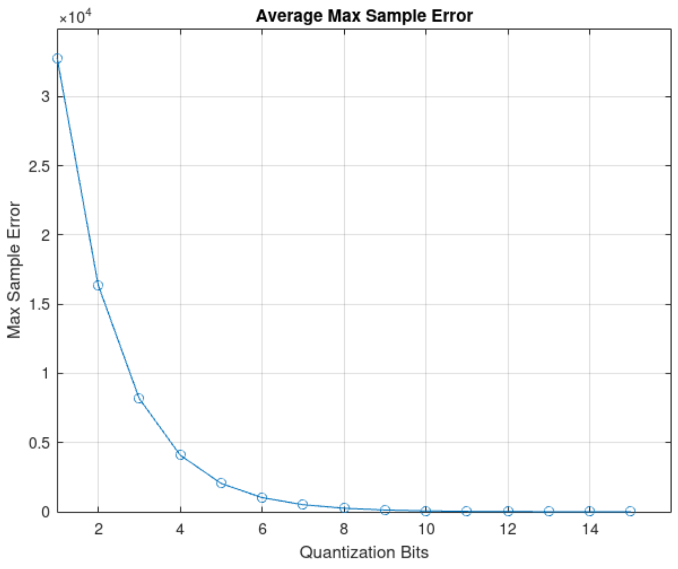


Figure 2.4: Average Max Sample Error of the channels

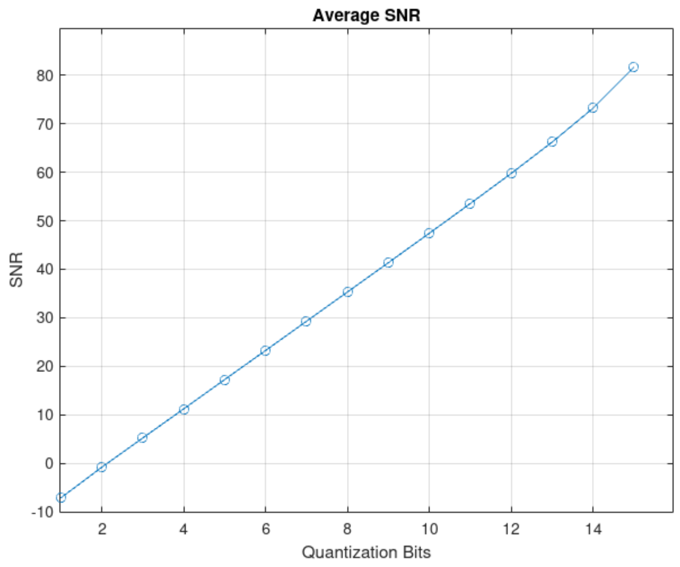


Figure 2.5: Average SNR of the channels

2.3 Exercise 3

2.3.1 Modifications or Insertions in the code

This exercise is divided in a class and an associated program. The "wav_quant" class has two methods: one for performing quantization by replacing the least significant bits with zeros and saving the quantized samples, and other to write these quantized samples to an output file.

```
void quantize(const vector<short> &samples)
{
    int cutBits = 16 - quantBits;
    for (auto sample : samples){
        sample >>= cutBits; // bits menos significativos a 0
        short tmp = sample << cutBits; // posicao original
        quantSamples.insert(quantSamples.end(),tmp);
    }
}

void toFile(SndfileHandle sfhOut) {
    sfhOut.write(quantSamples.data(), quantSamples.size());
}
```

Listing 2.10: WAVCmp SNR calculation

In "wav_quant.cpp" we just use the two functions from the class.

```
...
WAVQuant quant {quantBits};
while((nFrames = sfhIn.readf(samples.data(), FRAMES_BUFFER_SIZE)))
{
    samples.resize(nFrames * sfhIn.channels());
    quant.quantize(samples);
}
quant.toFile(sfhOut);
```

Listing 2.11: WAVCmp SNR calculation

2.3.2 Usage

The program can be launched using the following command:

```
./wav_quant <input file> <quantization bits> <output file>
```

Both <input file> and <output file> are paths to .wav files (both will be tested for format and compatibility). The bits kept for quantization must be between 1 and 15.

2.3.3 Results

Quantization is used to reduce the number of bits necessary to represent samples values. In order to test our code, we chose an audio file, 'sample.wav', and performed three different quantizations keeping 8, 5 and 3 bits for each one. The first observation was the noticeable increase in noise when listening to the output file with fewer bits. This happens because the value of the sample before represented as a value with more bits, which possessed more precision, is now approximated to the nearest value of the possible 2^N values, where N represents the number of bits kept for quantization. To confirm the results we generated distinct plots for each one of the three quantizations.

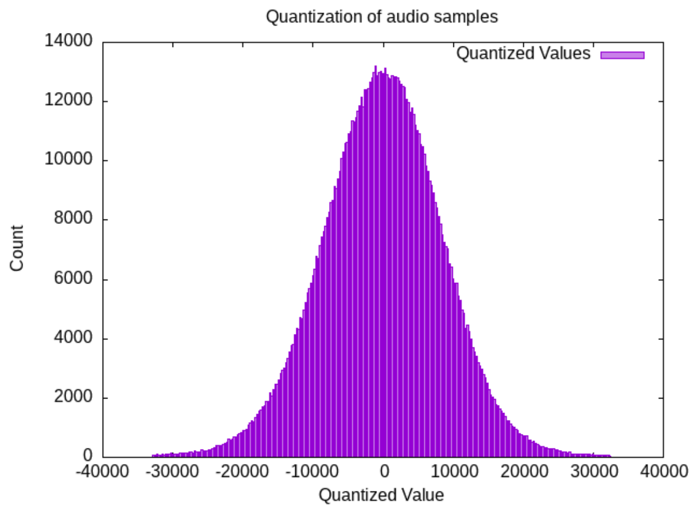


Figure 2.6: 8-bit Quantization

As we can see by figure 2.6, retaining 8 bits leads to values of samples still very distinct from each other, resulting in a sound that seems very close to the original. This is due to the presence of a total of $2^8 = 256$ possible values for each sample.

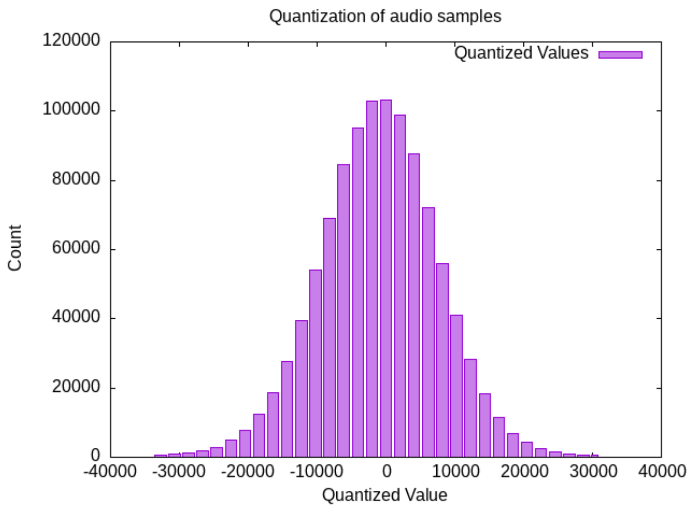


Figure 2.7: 5-bit Quantization

As it's shown by figure 2.7, lowering the bits kept to 5 only gives us $2^5 = 32$ possible values. This already introduces a noticeable difference from the original values, with more noise being perceptible.

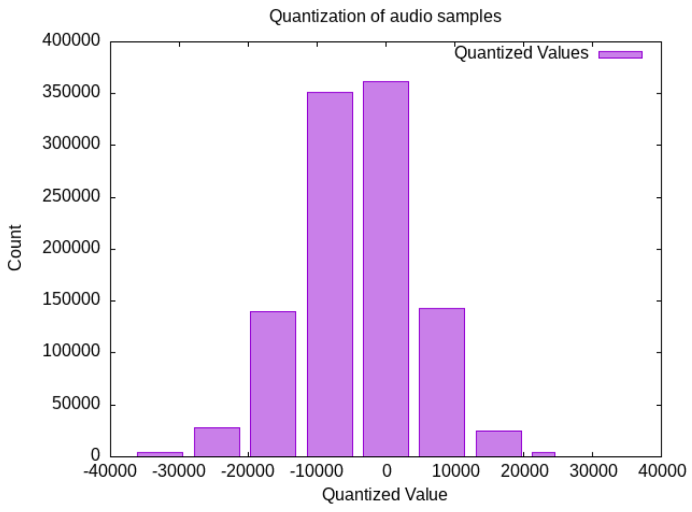


Figure 2.8: 3-bit Quantization

Finally, 3-bit quantization only gets us 8 possible values as seen in the figure 2.8. With this amount of bits is impossible not to get big differences from the original values and lots of noise.

2.4 Exercise 4

In this exercise, we implemented the following effects: echo (single and multiple lines), amplitude modulation, delay, advance, reverse, speed up, slow down, invert, mono and restrict to Left or Right channels.

2.4.1 Usage

The program can be launched using the following command:

```
./wav_effects [OPTIONS] [EFFECTS] <inputFile>
```

To check the available options and effects, use the -h option, like:

```
./wav_effects -h
```

This command will generate the following text in the console:

```
Usage: %s [OPTIONS] [EFFECTS] <inputFile>
  OPTIONS:
  -h, --help          --- print this help
  -o, --output        --- set output file name (default: output.wav)
  EFFECTS:
  -e n d g            --- apply echo effect, [number_echoes], [delay
    ], [gain] (suggested: 1, 1.0, 0.5)
  -a f                --- apply amplitude modulation effect, f is
    frequency (default: 1)
  -d time             --- apply delay, time in seconds (default: 1)
  -f time             --- advance in music, time in seconds (default:
     1)
  -r                  --- apply reverse effect
  -s x                --- apply speed up effect (default: 10%)
  -b x                --- apply slow down effect (default: 10%)
  -i                  --- apply invert effect
  -m                  --- apply mono effect (convert all channels to
    one)
  -l                  --- play only left channel
  -p                  --- play only right channel
```

Listing 2.12: wav_effects arguments

The program will check for correct options introduced by the user and possible arguments and check their validity. The default arguments will be applied if no value is supplied in the Effects that require it. Note that the program converts all arguments into doubles and it's up to when the individual function is called to check for the arguments validity. So if an invalid argument is introduced, for example a 0 where it doesn't accept one, the error will be detected when the first sample segment is to be processed.

2.4.2 Modifications or Insertions in the code

The Effects enum class serves as control logic in the code to represent the possible effects:

```cpp
enum Effects {
    NONE,
    ECHOE,
    AMPLITUDE_MODULATION,
    TIME_VARYING_DELAYS,
    DELAY,
    FORWARD,
    REVERSE,
    SPEED_UP,
    SLOW_DOWN,
    INVERT,
    MONO,
    LEFT_CHANNEL_ONLY,
    RIGHT_CHANNEL_ONLY
};
```

Listing 2.13: Possible effects enumerator

We use the EffectsInfo namespace to store configurable values gathered from the user input:

```cpp
namespace EffectsInfo {
    string outputFileName = "output.wav";
    string inputFileName = "sample.wav";
    Effects effect = NONE;
    vector<double> param;
    bool effectChosen = false;
};  // namespace EffectsInfo
```

Listing 2.14: EffectsInfo namespace

Regarding the main file, the wav_effects.cpp file. We have three main functions to deal with checking user inputs and setting variables. The setEffect function is used to set the user desired effect and associated parameters. If there isn't any associated parameter, the val pointer should be null and no value will be saved:

```cpp
void setEffect(Effects effect, char* val, double arg) {
    EffectsInfo::effect = effect;
    if (val == nullptr)
        return;
    if (!isdigit(*val)) {
        cerr << "Error: Expecting numerical value, but received "
    << val
             << " instead" << endl;
        exit(1);
    }
    try {
        EffectsInfo::param.push_back(arg);
    } catch (std::invalid_argument& e) {
        std::cerr << "Error: Invalid argument for option." << std::
    endl;
        exit(1);
    }
}
```

Listing 2.15: setEffects function

The check_wav_file is just a function to check if the file is valid and returns true if so, false otherwise:

```cpp
int check_wav_file(SndfileHandle& musicFile) {
    if (musicFile.error()) {
        cerr << "Error: invalid input file\n";
        return -1;
    }

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

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

    return 0;
}
```

Listing 2.16: check_wav_file function

The process_arguments function receives the user arguments and process them, dealing with making sure that the values options and effects are correct and setting the values for the configurable parameters:

```cpp
int process_arguments(int argc, char* argv[]) {
    for (int i = 1; i < argc; i++) {  // Start for at 1, to skip
    program name
        if (strcmp(argv[i], "-h") == 0 || strcmp(argv[i], "--help")
         == 0) {
            print_usage();
            return 1;
        } else if (strcmp(argv[i], "-o") == 0 ||
                   strcmp(argv[i], "--output") == 0) {
            i++;  // Move to the next argument for the output file
    name
            if (i < (argc - 1)) {
                EffectsInfo::outputFileName = argv[i];
            } else {
                std::cerr << "Error: Missing argument for -o/--
    output option."
                          << std::endl;
                return -1;
            }
        (...)
        } else if (!EffectsInfo::effectChosen && strcmp(argv[i], "-
    p") == 0) {
            setEffect(RIGHT_CHANNEL_ONLY, nullptr, INT32_MAX);
            EffectsInfo::effectChosen = true;
            // checks if the user introduced something unknown that
     starts with a '-'
        } else if (argv[i][0] == '-') {
            std::cerr << "Error: Unknown option or argument: " <<
    argv[i]
                      << std::endl;
            return -1;
        }
    }
    return 0;
}
```

Listing 2.17: process_arguments function

The main function of the program is described next:

```
int main(int argc, char* argv[]) {
    clock_t startTime = clock();
    (...)
    int ret = process_arguments(argc, argv);
    (...)

    /* Check .wav files */
    EffectsInfo::inputFileName = argv[argc - 1];
    SndfileHandle sfhIn{EffectsInfo::inputFileName};
    if (check_wav_file(sfhIn) < 0)
        return 1;

    int channels = sfhIn.channels();
    if (EffectsInfo::effect == MONO)
        channels = 1;

    SndfileHandle sfhOut{EffectsInfo::outputFileName, SFM_WRITE,
    sfhIn.format(),
                         channels, sfhIn.samplerate()};

    if (sfhOut.error()) {
        cerr << "Error: problem encountered generating file "
             << EffectsInfo::outputFileName << endl;
        return 1;
    }

    // Create a buffer for reading and writing audio data
    std::vector<short> inputSamples(FRAMES_BUFFER_SIZE * sfhIn.
    channels());
    std::vector<short> outputSamples;
    std::vector<short> reverseBuffer;
    size_t nFrames;

    // Effects class
    WAVEffects effects{EffectsInfo::effect, EffectsInfo::param,
                       sfhIn.samplerate()};

    // Read and process audio data in chunks
    while ((nFrames = sfhIn.readf(inputSamples.data(),
    FRAMES_BUFFER_SIZE))) {

        inputSamples.resize(nFrames * sfhIn.channels());
        // Apply the desired effect to the input buffer
        switch (EffectsInfo::effect) {
            case ECHOE:
                effects.effect_echo(inputSamples, outputSamples);
                break;
            (...)
            case REVERSE:
                reverseBuffer.insert(reverseBuffer.end(), inputSamples.begin(),
                                     inputSamples.end());
                break;
            case RIGHT_CHANNEL_ONLY:
                effects.effect_merge_right_channel(inputSamples, outputSamples);
                break;
            default:
                cerr << "The specified effect is not supported.\n";
                return 1;
        };
    }

    if (EffectsInfo::effect == REVERSE)
        effects.effect_reverse(reverseBuffer, outputSamples);

    // Write the modified audio data to the output file
    //  (divide by the number of channels, since they all get mixed up)
    if (EffectsInfo::effect == MONO)
        sfhOut.writef(outputSamples.data(), outputSamples.size());
    else
        sfhOut.writef(outputSamples.data(),
                      outputSamples.size() / sfhIn.channels());

    (...)
}
```

Listing 2.18: main function

Here we start by reading the input file parameters and creating the output file based on them (with the exception of the Mono effect, which only requires one channel, even if the input file has more). Then we read each set of samples and perform the necessary operations, using the WAVEffects class. After processing all samples of the input file, we write the modified samples in the output file. In the next subsection we will be describing the algorithm for each of the effects. For each one of them, the inputSamples corresponds to the original audio signal and the outputSamples to the output audio signal with the desired effect.

Echo

The **echo effect** is an audio processing technique that adds delayed copies of the original signal. The algorithm for applying an echo effect to an audio signal is described as follows:

- **Arguments**:
  - arg - A vector of three arguments:
    - arg[0] - The number of echoes, denoted as $nLines$.

    * arg[1] - The delay in seconds, denoted as *delay*.
    * arg[2] - The gain or decay factor, denoted as *decay*.

- **Equation**:
  - Single Line:
$$y[n] = x[n] + \text{decay} \cdot y[n - \text{Delay}]$$
  - Multiple Lines:
$$y[n] = x[n] + \text{decay}_1 \cdot y[n - \text{D}_1] + \text{decay}_2 \cdot y[n - \text{D}_2]...$$

- **Results**:
  - Usage:

```
              ./wav_effects -o echo.wav -e 1 1.0 0.5 sample.
    wav
```

  - Execution time - Took 90.417 ms.
  - As we can see by the comparison in Figure 2.9, the signal duration stayed the same, but there was a slight change in the signal amplitude, however it's difficult to notice out-wright the inclusion of the echo effect.

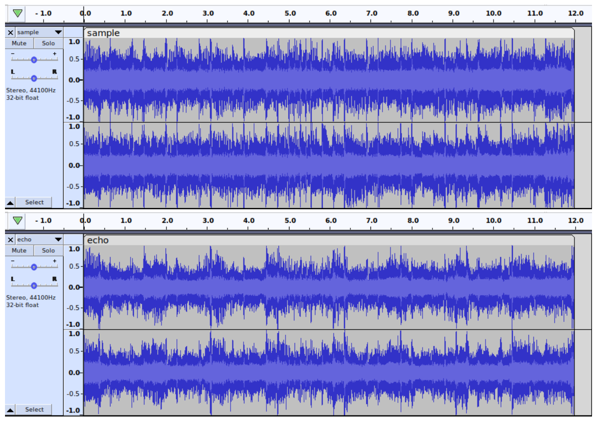


Figure 2.9: Comparison between the music without modifications (top) and with a single echo, delay of 1 second and 0.5 decay (bottom)

1. Calculate the delaySamples by converting the *delay* in seconds to the corresponding number of samples based on the sample rate. For example, 44100 samples for 1 second at a 44.1 kHz sample rate.
2. Iterate through each sample in the inputSamples:
   (a) Initialize *echoSample* with the current sample ($x[n]$).

(b) If the current sample index (i) is greater than or equal to delaySamples, apply the echo effect by:

   i. Calling the feedback_lines function that implements the Echo equation, both for single or multiple lines. This function updates *echoSample* and the outputSamples.

   ii. Dividing *echoSample* by $(1 + \text{decay})$ to attenuate it.

(c) Append the final *echoSample* to the outputSamples.

The echo effect algorithm creates *nLines* echoes with a specified *delay* and attenuates the signal to guarantee that the echoed samples don't exceed the maximum value.

```cpp
static int feedback_lines(const std::vector<short>& inputSamples,
    std::vector<short>& outputSamples, uint8_t numLines, double
    decay, int sampleDelay, int sample, int iter) {
    if (numLines == 0)
        throw std::invalid_argument(
            "The number of lines needs to be greater than 0");
    else if (numLines == 1) {
        // Single echoe: y[n] = x[n] + decay * y[n - Delay]
        sample += decay * inputSamples[iter - sampleDelay];
    } else {
        for (int i = 0; i < numLines; i++) {
            // Multiple echoe: y[n] = x[n] + decay1 * y[n - D1] +
    decay2 * y[n - D2]...
            sample += decay * outputSamples[iter - sampleDelay];
        }
    }

    return sample;
}

void effect_echo(const std::vector<short>& inputSamples, std::
    vector<short>& outputSamples) {

    // arguments
    if (arg.size() != 3)
        throw std::invalid_argument(
            "Expected 3 arguments for echo effect (number of echoes
    , delay "
            "and gain/decay)");

    uint8_t nLines = static_cast<uint8_t>(arg[0]);
    if (nLines == 0)
        throw std::invalid_argument(
            "Invalid number of lines (needs to be > 0)");
    double delay = arg[1];
    if (delay == 0)
        throw std::invalid_argument("Invalid delay (needs to be >
    0)");
    double decay = arg[2];
    if (decay <= 0 || decay >= 1)
        throw std::invalid_argument(
            "Invalid gain/decay (needs to be between 0 < x < 1)");

    int delaySamples = static_cast<int>(
        delay * this->sampleRate);  // 1 * 44100, 2 * 44100...

    for (long i = 0; i < (long)inputSamples.size(); i++) {
        int echoSample = inputSamples[i];  // x[n]
        if (i >= delaySamples) {
            echoSample = feedback_lines(inputSamples, outputSamples
    , nLines,
                                              decay, delaySamples,
    echoSample, i);
            echoSample /= (1 + decay);
        }
        outputSamples.push_back((short)echoSample);
    }
}
```

Listing 2.19: effect_echo function

Amplitude Modulation

The **amplitude modulation (AM)** effect is an audio processing technique that varies the amplitude (strength) of a carrier wave in proportion to the waveform being sent. The algorithm for applying an AM effect to an audio signal is described as follows:

- **Arguments**:
  - arg - A vector containing a single argument:
    - arg[0] - The modulation frequency, denoted as $freq$.
- **Results**:
  - Usage:

    ```
                 ./wav_effects -o amp_mod.wav -a 1 sample.wav
    ```

  - Execution time - Took 82.88 ms.
  - Present in Figure 2.10 is the comparison between the regular music signal and the one that's amplitude modulated. The modulated signal clearly shows the pattern of a high-frequency sinusoidal carrier wave, where the high points in amplitude mostly coincide with the loudest/peak segments of the music.

1. Initialize $freq$ as the provided modulation frequency.
2. Iterate through each sample in the inputSamples:
   (a) Calculate the modulated sample ($modulatedSample$) by multiplying the current input sample ($inputSamples[i]$) by the cosine of a sinusoidal carrier wave. The carrier wave has a frequency of $\frac{freq}{\text{sampleRate}}$ and is represented as $\cos(2\pi \cdot \frac{freq}{\text{sampleRate}} \cdot i)$.

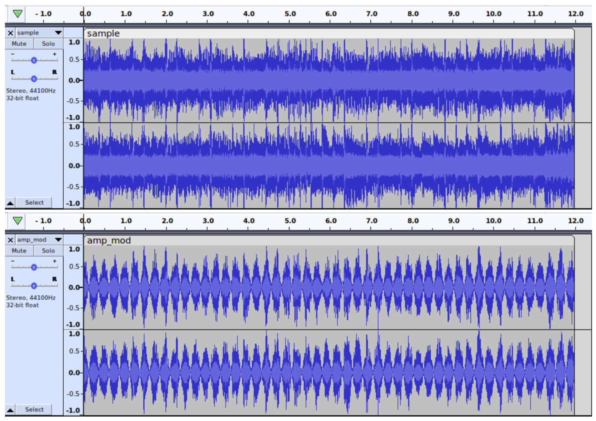


Figure 2.10: Comparison between the music without modifications (top) and with a modulation frequency of 1 second (bottom)

(b) Append the *modulatedSample* to the outputSamples.

The AM effect algorithm varies the amplitude of the input signal based on a sinusoidal carrier wave with the specified modulation frequency *freq*. This results in a modulated audio signal that carries the characteristics of the carrier wave.

The equation for the AM effect is as follows:

$$\text{modulatedSample} = \text{inputSamples}[i] \cdot \cos\left(2\pi \cdot \frac{\text{freq}}{\text{sampleRate}} \cdot i\right)$$

```
/*! In AM, the amplitude (strength) of a carrier wave is varied in
    proportion to the waveform being sent.*/
void effect_amplitude_modulation(const std::vector<short>&
    inputSamples, std::vector<short>& outputSamples) {

    // arguments
    if (arg.size() != 1)
        throw std::invalid_argument(
            "Expected 1 argument for Amplitude Modulation effect "
            "(frequency)");
    double freq = arg[0];
    if (freq == 0)
        throw std::invalid_argument("Invalid frequency (needs to be
     > 0)");

    for (long i = 0; i < (long)inputSamples.size(); i++) {
        // C(t) = A_c * cos(2pi * f_c * t) where f_C is 1/f = t
        short modulatedSample =
            inputSamples.at(i) * cos(2 * M_PI * (freq / sampleRate)
     * i);
        outputSamples.push_back(modulatedSample);
    }
}
```

Listing 2.20: effect_amplitude_modulation function

Delay

The **delay effect** is an audio processing technique that introduces a time delay to an audio signal, increasing it's duration. The algorithm for applying a delay effect to an audio signal is described as follows:

- **Arguments**:
  - arg - A vector containing a single argument:
    - arg[0] - The delay in seconds, denoted as *delay*.
- **Results**:
  - Usage:

```
./wav_effects -o delay.wav -d 5 sample.wav
```

  - Execution time - Took 46.831 ms.
  - Present in Figure 2.11 is the delayed audio. This effect delayed the music by adding 5 seconds of silence at the start. The music itself wasn't altered since the total time is 17 seconds (5 seconds of delay + 12 seconds of the original music),

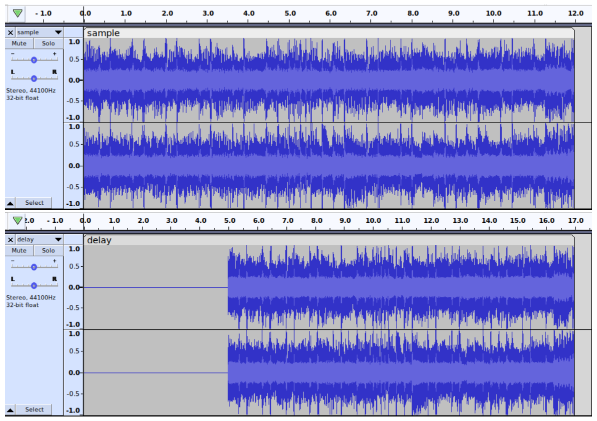


Figure 2.11: Comparison between the music without modifications (top) and with a delay of 5 seconds (bottom)

The delay effect algorithm is implemented as follows:

1. Calculate the number of delaySamples by converting the *delay* in seconds to the corresponding number of samples based on the sample rate. For example, 44100 samples for 1 second at a 44.1 kHz sample rate.
2. Fill the output with 0 samples for the extra time at the start to account for the delay provided by the user
3. Iterate through each sample in the inputSamples and append them to the end of outputSamples

```cpp
void effect_delay(const std::vector<short>& inputSamples, std::
    vector<short>& outputSamples) {

    if (arg.size() != 1)
        throw std::invalid_argument(
            "Expected 1 argument for delay effect (seconds)");

    uint8_t delay = static_cast<uint>(arg[0]);
    if (delay <= 0)
        throw std::invalid_argument("Invalid delay (needs to be >
    0)");

    long delaySamples = static_cast<long>(
        delay * this->sampleRate);  // 1 * 44100, 2 * 44100...

    // Fill the extra time at the start with 0s
    for (; aux < delaySamples * 2; aux++)
        outputSamples.push_back(0);

    for (long i = 0; i < (long)inputSamples.size(); i++)
        outputSamples.push_back(inputSamples[i]);
}
```

Listing 2.21: effect_delay function

Advance

The **Advance effect** is an audio processing technique that advances an audio signal, shortening it. The algorithm is described as follows:

- **Arguments**:
  - arg - A vector containing a single argument:
    - arg[0] - The advance in seconds, denoted as *time*.
- **Results**:
  - Usage:
    ```
                ./wav_effects -o advance.wav -f 5 sample.wav
    ```
  - Execution time - Took 36.868 ms.

– As present in Figure 2.12, the advanced music is shorter in time than the original music. Since we chose to advance in 5 seconds, the original music was slashed 5 seconds at the start, resulting in a length of 7 seconds.

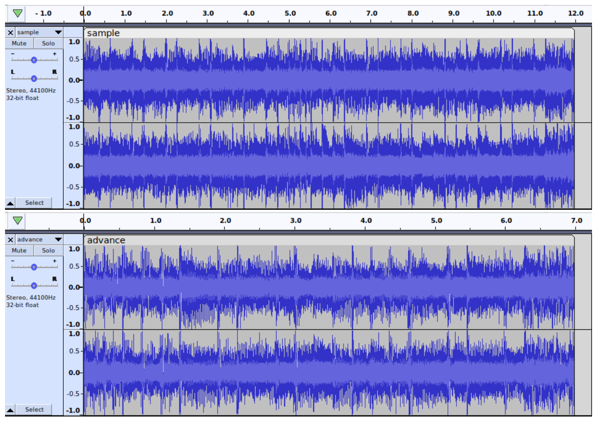


Figure 2.12: Comparison between the music without modifications (top) and with an advance of 5 seconds (bottom)

The advance effect algorithm is implemented as follows:

1. Calculate the number of delaySamples by converting the *delay* in seconds to the corresponding number of samples based on the sample rate. For example, 44100 samples for 1 second at a 44.1 kHz sample rate.
2. Iterate through each sample in the inputSamples:
   (a) If the *aux* counter exceeds 2 times the delaySamples, append each input sample (inputSamples[i]) to the outputSamples, resulting in selectively passing the audio signal forward in time.

```cpp
void effect_forward(const std::vector<short>& inputSamples, std::
    vector<short>& outputSamples) {

    if (arg.size() != 1)
        throw std::invalid_argument(
            "Expected 1 argument for forward effect (seconds)");

    uint time = static_cast<uint>(arg[0]);
    if (time <= 0)
        throw std::invalid_argument("Invalid time (needs to be > 0)
    ");

    int delaySamples = static_cast<int>(
        time * this->sampleRate);  // 1 * 44100, 2 * 44100...
    for (long i = 0; i < (long)inputSamples.size(); i++) {
        if (aux++ > delaySamples * 2)
            outputSamples.push_back(inputSamples[i]);
    }
}
```

Listing 2.22: effect_forward function

Reverse

The **reverse effect** is an audio processing technique that reverses the order of audio samples in an audio signal, effectively playing the audio signal backward. The algorithm is described as follows:

- **Results**:
  - Usage:
    ```
                    ./wav_effects -o reverse.wav -r sample.wav
    ```
  - Execution time - Took 52.21 ms.
  - Present in Figure 2.13 is the comparison between the original music and the reversed one. Looking closely, we can see the reverse of the original music, with the signal remaining the same in amplitude and duration.

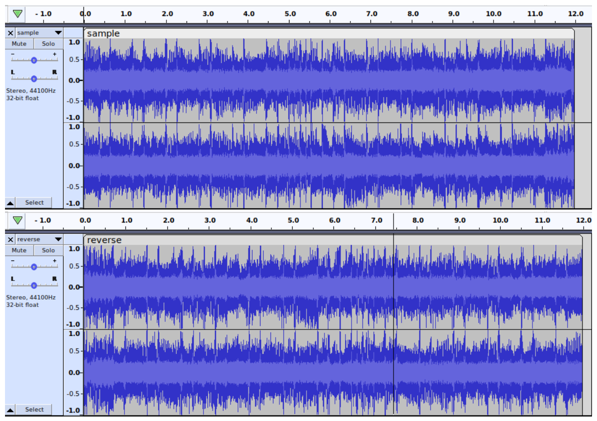


Figure 2.13: Comparison between the music without modifications (top) and with reverse effect (bottom)

1. Iterate in reverse order through each sample in the inputSamples and append it to outputSamples.

Note that as present in the code, the Reverse effect has a modification in main to allow it to buffer all samples into a vector, that is then provided to the function after all frames are read.

```cpp
void effect_reverse(const std::vector<short>& inputSamples, std::
    vector<short>& outputSamples) {
    for (long i = ((long)inputSamples.size() - 1); i >= 0; i--)
        outputSamples.push_back(inputSamples[i]);
}
```

Listing 2.23: effect_reverse function

Speed Up

The **speed-up effect** is an audio processing technique that increases the playback speed of an audio signal. The algorithm is described as follows:

- **Arguments**:
  - arg - A vector containing a single argument:
    - arg[0] - The speed-up percentage, denoted as *speedUp*.
- **Results**:
  - Usage:

    ```
                ./wav_effects -o speedUp.wav -s 20 sample.wav
    ```

  - Execution time - Took 40.818 ms.
  - The speed-up effect present in Figure 2.14 was one of the trickiest ones to implement. It's still not perfect since we should have implemented techniques for quantization or normalization, since a slight humming was introduced to the music (we think this is due to the casting of a division to an integer and then to short), specially at lower speed-up values, i.e 20%. Analyzing the signal, it's clear that the music was accelerated, since the duration was reduced, in this case, by 20%.

1. Calculate the $speedUpFactor$ as $\frac{100-\text{speedUp}}{100.0}$, representing the factor by which the playback speed will be increased.
2. Calculate the number of numOutputSamples based on the original input sample count and the $speedUpFactor$. This represents the length of the sped-up audio.
3. Iterate through each sample in the inputSamples:
   (a) Calculate the corresponding input index for the sped-up output by dividing the current output index (i) by the $speedUpFactor$.
   (b) Append the input sample (inputSamples[$inputIndex$]) to the outputSamples.

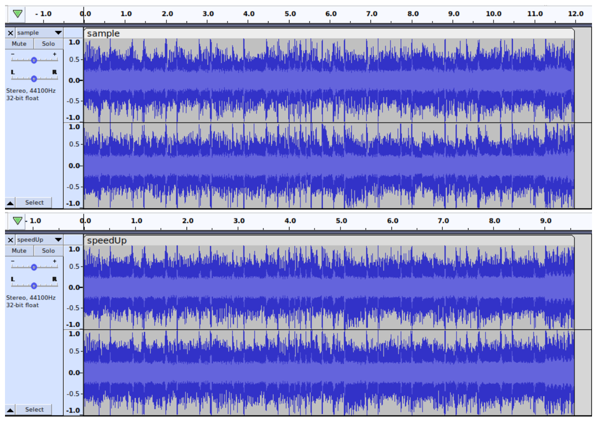


Figure 2.14: Comparison between the music without modifications (top) and the version speed-up by 20% (bottom)

```cpp
void effect_speed_up(const std::vector<short>& inputSamples, std::
    vector<short>& outputSamples) {

    if (arg.size() != 1)
        throw std::invalid_argument(
            "Expected 1 argument for speed up effect (percentage)")
    ;

    uint speedUp = static_cast<uint>(arg[0]);
    if (speedUp <= 0)
        throw std::invalid_argument(
            "Invalid speed up percentage (needs to be x > 0)");

    float speedUpFactor = (100 - speedUp) / 100.0;

    size_t numOutputSamples =
        static_cast<size_t>(inputSamples.size() * speedUpFactor);

    for (size_t i = 0; i < numOutputSamples; ++i) {
        size_t inputIndex = static_cast<size_t>(i / speedUpFactor);
        outputSamples.push_back(inputSamples[inputIndex]);
    }
}
```

Listing 2.24: effect_speed_up function

Slow down

The **slow-down effect** is an audio processing technique that decreases the playback speed of an audio signal. The algorithm is similar to the speed-up, with the exception of the *speedUpFactor* where we now sum 100 instead of subtracting. The new one is calculated as $\frac{100+\text{speedUp}}{100.0}$.

- **Results**:

– Usage:

```
./wav_effects -o slowDown.wav -b 20 sample.wav
```

– Execution time - Took 47.606 ms.

– Since this effect is similar in implementation to the speed-up, we also encountered the same problem with the sound quality. But apart from that it's working as intended and as present in Figure 2.15, the slow-downed music is 20% larger in duration than the original music, and it's especially noticeable for the high frequencies around the middle of the music, that they were extended in duration.

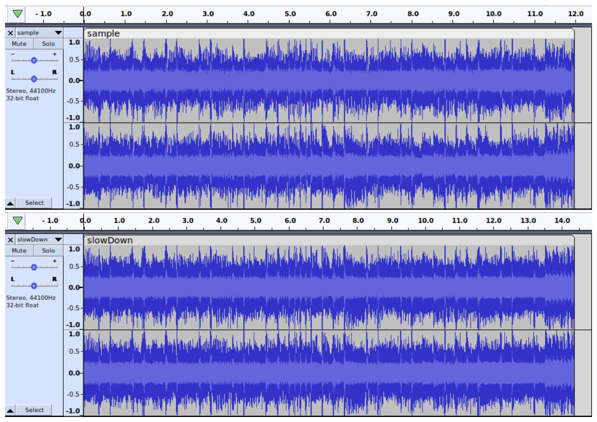


Figure 2.15: Comparison between the music without modifications (top) and the version slow-down by 20% (bottom)

```
void effect_slow_down(const std::vector<short>& inputSamples, std::
    vector<short>& outputSamples) {

    if (arg.size() != 1)
        throw std::invalid_argument(
            "Expected 1 argument for speed up effect (percentage)")
    ;

    uint speedUp = static_cast<uint>(arg[0]);
    if (speedUp <= 0)
        throw std::invalid_argument(
            "Invalid speed up percentage (needs to be x > 0)");

    float speedUpFactor = (100 + speedUp) / 100.0;

    size_t numOutputSamples =
        static_cast<size_t>(inputSamples.size() * speedUpFactor);

    for (size_t i = 0; i < numOutputSamples; ++i) {
        size_t inputIndex = static_cast<size_t>(i / speedUpFactor);
        outputSamples.push_back(inputSamples[inputIndex]);
    }
}
```

Listing 2.25: effect_slow_down function

Invert

The **Invert effect** is an audio processing technique that changes the polarity of an audio signal, effectively inverting it's amplitude. The algorithm is described as follows:

1. Iterate through each sample in inputSamples and invert the signal by multiplying -1 and append it to outputSamples.

- **Results**:
  - Usage:

    ```
                   ./wav_effects -o invert.wav -i sample.wav
    ```

  - Execution time - Took 56.563 ms.
  - This effect is one of the clearest to spot, just looking at Figure 2.16, it's clearly visible that the music was inverted, with positive peaks transformed into negative peaks and vice-versa. However, listening to the music doesn't make any noticeable effect since for the human ear, negative or positive frequencies generate the same result.

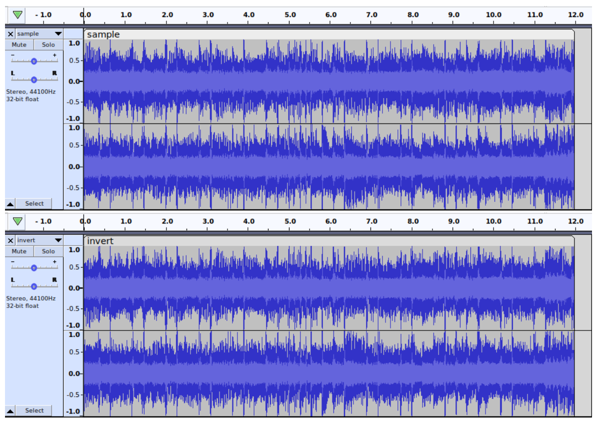


Figure 2.16: Comparison between the music without modifications (top) and with the invert effect (bottom)

```
void effect_invert(const std::vector<short>& inputSamples, std::
    vector<short>& outputSamples) {

    for (size_t i = 0; i < inputSamples.size(); i++) {
```

```
        outputSamples.push_back(-inputSamples[i]);
    }
}
```
Listing 2.26: effect_invert function

Mono

The **mono effect** is an audio processing technique that converts stereo audio (with separate left and right channels) into mono audio by calculating the average of the left and right channel samples. The algorithm is described as follows:

1. Iterate through each sample in inputSamples with a step of 2, as stereo audio samples are interleaved (left, right, left, right).
    (a) Calculate the mono sample by averaging the left channel sample (inputSamples[i]) and the right channel sample (inputSamples[$i+1$]).
    (b) Append the mono sample to the outputSamples.

- **Results**:
    - Usage:
      ```
      ./wav_effects -o mono.wav -m sample.wav
      ```
    - Execution time - Took 51.156 ms.
    - This simple effect combined the two channels into one and as present in Figure 2.17, we can see that the resulting mono channels is a mix of the two channels.

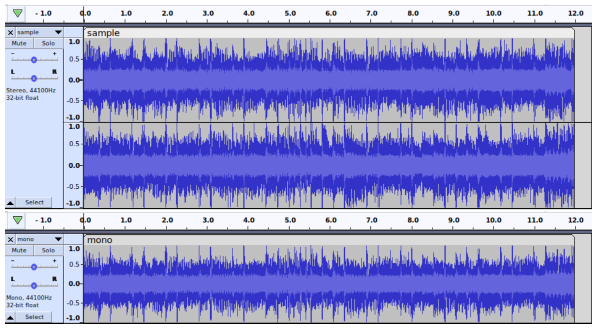


Figure 2.17: Comparison between the music without modifications (top) and with the mono effect (bottom)

Then, as described above, in the main program we need to create the output file to only account for one channel.

```cpp
void effect_mono(const std::vector<short>& inputSamples, std::
    vector<short>& outputSamples) {

    for (size_t i = 0; i < inputSamples.size(); i += 2) {
        // Calculate the average of the left and right channels
        int sample = (inputSamples[i] + inputSamples[i + 1]) / 2;
        outputSamples.push_back(sample);
    }
}
```

Listing 2.27: effect_mono function

Merge to Right Channel

The **merge right channel effect** is an audio processing technique that takes a stereo audio signal and creates a new audio signal by merging the right channel into the left channel (effectively leaving the left channel silenced). The algorithm is described as follows:

1. Iterate through each sample in the inputSamples with a step of 2, as stereo audio samples are interleaved (left, right, left, right).
   (a) Calculate the merged sample by averaging the left channel sample (inputSamples[i]) and the right channel sample (inputSamples[$i+1$]).
   (b) Append a zero to the outputSamples to represent the left channel.
   (c) Append the merged sample to the outputSamples to represent the right channel.

- **Results**:
  - Usage:
    ```
                ./wav_effects -o right_channel.wav -p sample.
        wav
    ```
  - Execution time - Took 55.214 ms.
  - In Figure 2.18, the left channel is completely absent of samples.

```cpp
void effect_merge_right_channel(const std::vector<short>&
    inputSamples, std::vector<short>& outputSamples) {

    for (size_t i = 0; i < inputSamples.size(); i += 2) {
        int sample = int((inputSamples[i] + inputSamples[i + 1]) /
    2);
        outputSamples.push_back(0);
        outputSamples.push_back(sample);
    }
}
```

Listing 2.28: effect_merge_right_channel function

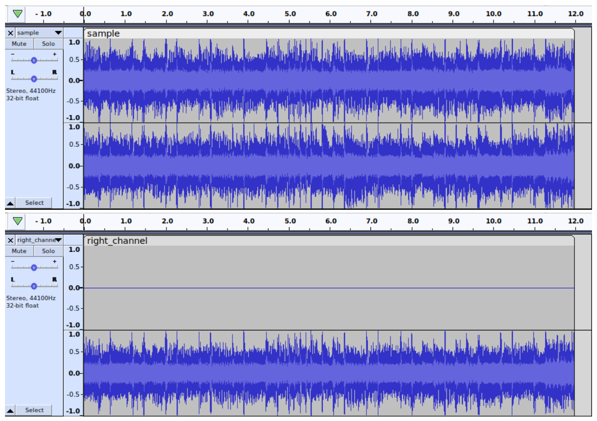


Figure 2.18: Comparison between the music without modifications (top) and with the merge to right channel (bottom)

Merge Left Channel

The **merge left channel effect** is similar to the **merge right channel effect**, however the left channel will be the one with the audio and the right one staying silent.

- **Results**:
  - Usage:

    ```
                ./wav_effects -o left_channel.wav -l sample.
        wav
    ```

  - Execution time - Took 77.528 ms.
  - In Figure 2.19, the right channel is completely absent of samples.

```cpp
void effect_merge_left_channel(const std::vector<short>&
    inputSamples, std::vector<short>& outputSamples) {
    for (size_t i = 0; i < inputSamples.size(); i += 2) {
        int sample = int((inputSamples[i] + inputSamples[i + 1]) /
    2);
        outputSamples.push_back(sample);
        outputSamples.push_back(0);
    }
}
```

Listing 2.29: effect_merge_left_channel function

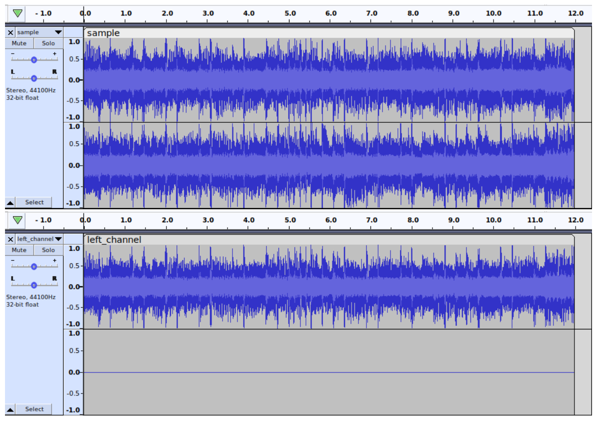


Figure 2.19: Comparison between the music without modifications (top) and with the merge to left channel (bottom)

Chapter 3

Part II

3.1 Exercise 5

3.1.1 Class Methods

For the BitStream class, we have implemented methods to write various types of data in the most compressed way possible.

The methods of the class are the following:

- Constructor - The Constructor creates a std::fstream file (can be read-/written), the mode of the buffer (read/write), and associates a buffer of 8 bits to manage the operations, along with a default index for the buffer
- writeBit - writes a bit to the buffer/file (writes a byte to the file if buffer goes full)
- readBit - reads a bit from the file to the buffer (returns a byte as output if the buffer goes full)
- writeNBits/readNBits - writes/reads N bits (N <= 64)
- writeInt/writeChar/writeString - Aux functions to write data types
- readInt/readChar/readString - Aux functions to read data types
- Deconstructor - The Deconstructor has the purpose of flushing any remaining data from the buffer into the file.

3.1.2 Performance

Before going into examples, there are few aspects that are noticeable from the code. The first one is that some data types are not compressible, such as chars/strings. Also, negative numbers are not compressed because we didn't implement a way to handle two's complement representation. Finally, doubles/float types cannot be used as well because we didn't handle floating point representation. But the goal of the BitStream class is to compress positive integers.

We have created a test file for this class, and we've created a few data to compress and decompress.

```
    BitStream writer('w', example.bin);

    writer.writeBit(0);

    writer.writeBit(0);

    writer.writeBit(0);

    writer.writeNBits(42994, 11);  //A7C0

    writer.writeChar('A');

    writer.writeString("Info Cod");

    writer.writeInt(300);

    writer.~BitStream();

    BitStream reader('r', "example.bin");

    int bit1 = reader.readBit();
    int bit2 = reader.readBit();
    int bit3 = reader.readBit();

    std::cout << "Bit1: " << bit1 << " , Bit2: " << bit2 << " ,
    Bit3: " << bit3
              << std::endl;

    long readNums = reader.readNBits(11);
    std::cout << "Read Nums: " << readNums << std::endl;

    char readCharacter = reader.readChar();
    std::cout << "Read Character: " << readCharacter << std::endl;

    std::string readString = reader.readString(64);
    std::cout << "Read String: " << readString << std::endl;

    int readInteger = reader.readInt(9);
    std::cout << "Read Int: " << readInteger << std::endl;

    reader.~BitStream();
```

Listing 3.1: data being written to a file and then read

This are the results of the program:

```
Bit 1: 0
Bit 2: 0
Bit 3: 0
Read Vector/Number: 1343
Read Character: A
Read String: Info Cod
Read Integer: 300
```

If the number of bits to be read is incorrectly specified (that is, reading more or less bits that the ones that were written to the file), the data in return is incorrect and it can result in a waterfall of random data being read. Taking for example this inputs:

```cpp
    BitStream writer('w', "example.bin");

    writer.writeBit(0);

    writer.writeBit(0);

    writer.writeBit(0);

    writer.writeNBits(42994, 11);  //A7C0

    writer.writeChar('A');

    writer.writeString("Info Cod");

    writer.writeInt(300);

    writer.~BitStream();

    BitStream reader('r', "example.bin");

    int bit1 = reader.readBit();
    int bit2 = reader.readBit();
    int bit3 = reader.readBit();

    std::cout << "Bit1: " << bit1 << " , Bit2: " << bit2 << " ,
    Bit3: " << bit3
              << std::endl;

    long readNums = reader.readNBits(16);
    std::cout << "Read Nums: " << readNums << std::endl;

    char readCharacter = reader.readChar();
    std::cout << "Read Character: " << readCharacter << std::endl;

    std::string readString = reader.readString(64);
    std::cout << "Read String: " << readString << std::endl;

    int readInteger = reader.readInt(9);
    std::cout << "Read Int: " << readInteger << std::endl;

    reader.~BitStream();
```

Listing 3.2: Data being read (data for the first writeNBits function doesn't correspond to the number of bits that were written

We get the following results:

Bit 1: 0
Bit 2: 0
Bit 3: 0
Read Vector/Number: 42984
Read Character:)
Read String: -m (2 undetected ascii characters before m and one after the m)
Read Integer: 389

We can see that the fact that we read 16 bits from the corresponding writeNBits written (11 bits were written using this fucntion) caused the program to read the data from the original 11 bits plus other 5 bits that belonged to the data that was written after 42994. Not only did we get the wrong output from that data, but also from the data after, because the extra bits that were read are no longer readable (the buffer_index will no longer point to that data and the buffer will get another byte from the file, erasing the previous one).

If the last integer on the previous example was being read with more bits than its representation, the outcome for the integer would be random as well.

```
    writer.writeInt(300);

    ...

    int readInteger = reader.readInt(15);
    std::cout << "Read Int: " << readInteger << std::endl;
```

Listing 3.3: Integer 300 was written with the necessary 9 bits and 15 bits were read

The output for the integer is 19211. After reaching the last character's end, the next bits will be random, because the file.get function will retrieve the EOF (end of file) value, which is a value (not a character) that can vary according to different systems. So the result is unpredictable.

The same thing that happens if the number of bits read from a type of data that was written is bigger than the size of the actual data will happen if the number of bits read is smaller.

```
    BitStream writer('w', "example.bin");

    writer.writeBit(0);

    writer.writeBit(0);

    writer.writeBit(0);

    writer.writeNBits(42994, 11);  //A7C0

    writer.writeChar('A');

    writer.writeString("Info Cod");

    writer.writeInt(300);

    writer.~BitStream();

    BitStream reader('r', "example.bin");

    int bit1 = reader.readBit();
    int bit2 = reader.readBit();
    int bit3 = reader.readBit();

    std::cout << "Bit1: " << bit1 << " , Bit2: " << bit2 << " ,
    Bit3: " << bit3
              << std::endl;

    long readNums = reader.readNBits(9);
    std::cout << "Read Nums: " << readNums << std::endl;

    char readCharacter = reader.readChar();
    std::cout << "Read Character: " << readCharacter << std::endl;

    std::string readString = reader.readString(64);
    std::cout << "Read String: " << readString << std::endl;

    int readInteger = reader.readInt(9);
    std::cout << "Read Int: " << readInteger << std::endl;

    reader.~BitStream();
```

Listing 3.4: Data being read (data for the first writeNBits function is smaller than the correspond to the number of bits that were written

We get the following outputs:

Bit 1: 0
Bit 2: 0
Bit 3: 0
Read Vector/Number: 335
Read Character: undetected ascii character
Read String: R[followed by 4 ascii characters not represented
Read Integer: 75

The reason behind those outputs is similar to the reason why this waterfall effect happens on the previous example. In fact, here less bits will be read for the number 42994, which will give us the number 335. Those unread bits that belonged to this data will be read for the other data because the buffer_index is pointing at them. The buffer_index will always point X bits before where it needed to point, being X the difference between the size of data written and the wrong amount read (in this case, X = 11 - 9 = 2, so it will point always two bits before the correct bit to get the correct output). All the data read will be compromised.

The program also works if we want to represent Nbits in a greater number of bits that the minimum we need to represent it. If we use writeNbits to represent the number 3 in a 4 bit representation, we get the number 3 as output (as long as we specifie the correct number of bits to be read and assuming that there was no mistake in the number of bits to be read for other data before the number 3).

The results show that the data is being compressed into the file and returned the same values that were inserted, as long as the data being read has the same size as the one that was written.

3.2 Exercise 6

3.2.1 Modifications or Insertions in the code

On the encoder the characters read from a text file are grouped into blocks of 8. These blocks of characters are converted to bits which are stored on a long integer. Then we use the writeNbits function from the BitStream class to write the long value to the binary file, also in blocks of 8, each representing a byte.

```
BitStream outF{'w', outFile};

// Convert chuncks of 8 bits to a long value
// Then write each byte
long longValue = 0;
int bitsWritten = 0;
for (char c : fileBits) {
    longValue = (longValue << 1) + (c - '0');
    bitsWritten++;
    if (bitsWritten == 8) {
        outF.writeNBits(longValue, 8);
        longValue = 0;
        bitsWritten = 0;
    }
}
```

Listing 3.5: Encoder reading and writing

On the decoder, the binary data is read from the input file using the readNbits function. The goal is to represent the long integer returned as a sequence of 0s and 1s (binary) in the output text file. We do this by converting the long value into an 8-bit binary representation.

```
BitStream inputF{'r', inFile};
std::ofstream outFile(argv[2],ios::out);

int fileSize = inputF.fileSizeBytes();
    for (int i = 0; i < fileSize; i++) {
        long longValue = (inputF.readNBits(8));
        std::string binaryValue = "";

        // Convert longValue to a binary string with 8 bits
        for (int j = 7; j >= 0; j--) {
            int bit = (longValue >> j) & 1;
            binaryValue += std::to_string(bit);
        }
        outFile << binaryValue;
    }
```

Listing 3.6: Encoder reading and writing

3.2.2 Usage

We can run the encoder and the decoder by using the following commands:

```
./encoder <input file> <output file>
./decoder <input file> <output file>
```

On the encoder <input_file> is a text file and the <output_file> is a binary file. The decoder accepts a binary file for <input_file> and a text file for the <output_file>. The input file for the encoder is tested if it only has 0s and 1s.

3.2.3 Results

A text file contains information represented as characters, with each one occupying 2 bytes of storage space. By encoding we can compress this data reducing the 8 characters in the text file (which represent 8 bits) into a single byte in the resulting binary file. This compression results in a more efficient storage of the data.

While testing we used a text file containing only zeros and ones sequences in multiples of 8. We ran the encoder and encoder separated and with a script that executed both at the same time. The binary file produced by the encoder had the compressed information as expected. The result from the decoder was a text file with the decompressed information and, if the script was used, this output was exactly the same as the original text file.

Chapter 4

Part III

4.1 Exercise 7

This exercise is focused on implementing a lossy codec for mono audio files. Two different programs were constructed, the *encoder_lossy* and *decoder_lossy*, for encoding and decoding a music, respectively.

4.1.1 Encoded file structure

Since the requirements forced the decoder program in only relying in the encoded file to obtain the parameters, we needed to embed the necessary information into the file. With this in mind, we chose to include all the information at the start of the file and with fixed size. This could involve some truncating of values, but according to our testing no such occurrences were found. Just a small note, the Number of channels was included for future use if needed, but it's not a value that is used rigorously in our program, since we supposed to be dealing only with mono audio files.

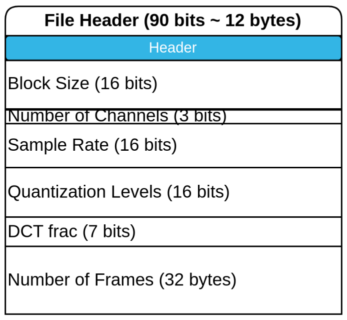

File Header (90 bits ~ 12 bytes)
Header
Block Size (16 bits)
Number of Channels (3 bits)
Sample Rate (16 bits)
Quantization Levels (16 bits)
DCT frac (7 bits)
Number of Frames (32 bytes)

Figure 4.1: File Header structure

4.1.2 Code Explanation

Lossy Encoder

The *Options* class condensates all the various configurable parameters needed thought the code into a shared data structure. Here we can see the parameters that have default values and the ones that are obtained from processing the file (the ones that aren't initialized).

```cpp
// configurable parameters
namespace Options {
string musicName = "sample.wav";
string encodedName = "encodedSample";
size_t blockSize = 1024;
size_t quantizationLevels = 8;  // or 256 levels
double dctFrac = 0.2;
size_t nChannels;  // should be always 1, but will leave it for
    future upgrades
size_t nFrames;
size_t sampleRate;
}  // namespace Options
```

Listing 4.1: Options namespace

The *create_file_header* creates a header in a file, by writing the necessary contents of the *Options* namespace into the encoded file according to the structure defined in subsection 4.1.1.

```cpp
void create_file_header(BitStream& outputStream) {
    // Store Block Size (16 bits)
    outputStream.writeNBits(Options::blockSize, 16);

    // Store Number of channels -> technically should be always 1
    (3 bits)
    outputStream.writeNBits(Options::nChannels, 3);

    // Store SampleRate (16 bits)
    outputStream.writeNBits(Options::sampleRate, 16);

    // Store Quantization Levels (16 bits)
    outputStream.writeNBits(Options::quantizationLevels, 16);

    // Store DCT Fraction (7 bits > max 100)
    outputStream.writeNBits(int(Options::dctFrac * 100), 7);

    // Store Total Number of frames (32 bits)
    outputStream.writeNBits(Options::nFrames, 32);
}
```

Listing 4.2: Create File Header

Since our lossy encoder and decoder should only handle mono audio files, the *transform_music_mono* function creates a new mono audio file based on the stereo version supplied by the user.

```cpp
void transform_music_mono(SndfileHandle& sfhIn, SndfileHandle&
    sfhOut) {

```

```
    std::vector<short> inputSamples(FRAMES_BUFFER_SIZE * sfhIn.
    channels());
    std::vector<short> outputSamples;
    vector<double> arg;
    size_t nFrames;

    WAVEffects effects{MONO, arg, sfhIn.samplerate()};

    while ((nFrames = sfhIn.readf(inputSamples.data(),
    FRAMES_BUFFER_SIZE))) {
        inputSamples.resize(nFrames * sfhIn.channels());

        effects.effect_mono(inputSamples, outputSamples);
    }

    sfhOut.writef(outputSamples.data(), outputSamples.size());
}
```

Listing 4.3: Transform music mono

The *quantize_dct_coefficients* function converts the floating-point values obtained by the application of the Direct Cosine Transformation (DCT) on the samples into integer values quantified to the desired number of bits. Generally the quantization occurs in the 16 bits and below range. However in our testing, we needed to have the option for 32 bits, not because of this function, since the results will be the same as using 16 bits. But because later in the code we are able to save the values into the encoded file, without truncating. So the 32 bits here present is just a condition introduced for compatibility in the main function, when writing the values into the encoded file since the same variable, $Options :: quantizationLevels$ is used. This function also transforms the DCT coefficients, that arrive in a vector of blocks, into a single vector with all the values, since this is the best way to save the values into a file.

```
// Function to quantize DCT coefficients
std::vector<int> quantize_dct_coefficients(
    const vector<vector<double>>& dct_blocks) {

    uint cutBits;
    if (Options::quantizationLevels <= 16)
        cutBits = 16 - Options::quantizationLevels;
    else
        cutBits = 32 - Options::quantizationLevels;
    std::vector<int> quantizedCoefficients;
    for (const std::vector<double>& dctCoefficients : dct_blocks) {
        for (const double coefficient : dctCoefficients) {
            int quantizedCoefficient = int(coefficient);
            quantizedCoefficient >>= cutBits;  // LSB at 0
            quantizedCoefficient = quantizedCoefficient
                                   << cutBits;  // Revert the
    position change
            quantizedCoefficients.push_back(quantizedCoefficient);
        }
    }

    return quantizedCoefficients;
}
```

Listing 4.4: Quantization Function

The *main* function handles all the processing logic and program flow. We want to highlight the checking of the audio file for mono channel and the ability to create a new one that is. The file header is saved into the file before all the processing occurs. Then the DCT is applied and it's values quantified. The resulting integer values will be saved into the encoded file, with the defined size.

```
int main(int argc, char* argv[]) {
    int ret = process_arguments(argc, argv);
    (...)

    /* Check .wav files */
    SndfileHandle sfhTmp{Options::musicName};
    if (check_wav_file(sfhTmp) < 0)
        return 1;

    int channels = sfhTmp.channels();
    // Check if the music is mono channel and ask the user if we
    want's to transform it
    if (channels > 1) {
        cout << "This program can only process music in mono
    channel, do you "
                "wish to transform it? (y/n): ";

        (...)

        cout << " - Music transformed into mono and saved in " <<
    newMusicName
                  << endl;)
    }
    clock_t startTime = clock();

    /* Check .wav files */
    SndfileHandle sfhIn{Options::musicName};
    if (check_wav_file(sfhIn) < 0)
        return 1;

    BitStream outputBitStream{'w', Options::encodedName};

    (....)

    create_file_header(outputBitStream);

    (...)

    // Direct DCT
    fftw_plan plan_d = fftw_plan_r2r_1d(Options::blockSize, x.data
    (), x.data(),
                                                FFTW_REDFT10, FFTW_ESTIMATE
    );
    for (size_t n = 0; n < nBlocks; n++) {
        for (size_t c = 0; c < Options::nChannels; c++) {
            for (size_t k = 0; k < Options::blockSize; k++)
                x[k] = inputSamples[(n * Options::blockSize + k) *
```

```
                                        Options::nChannels +
                                    c];

            fftw_execute(plan_d);
            // Keep only "dctFrac" of the "low frequency"
    coefficients
            for (size_t k = 0; k < Options::blockSize * Options::
    dctFrac; k++)
                x_dct[c][n * Options::blockSize + k] =
                    x[k] / (Options::blockSize << 1);
        }
    }

    std::vector<int> quantizedCoefficients =
    quantize_dct_coefficients(x_dct);

    std::cout << "Number of quantized coefficients: "
              << quantizedCoefficients.size() << std::endl;

    for (int sample : quantizedCoefficients)
        outputBitStream.writeNBits(sample, Options::
    quantizationLevels);

    (...)

    return 0;
}
```

Listing 4.5: main function

Lossy Decoder

The Decoder follows the reversed logic of the encoder. We start by using the *read_file_header* function to read the header of the file, according to the previously established format, in subsection 4.1.1, into the same *Options* namespace explained in the Encoder.

```
void read_file_header(BitStream& inputStream) {
    // Store Block Size (16 bits)
    Options::blockSize = static_cast<size_t>(inputStream.readNBits
    (16));

    // Store Number of channels -> technically should be always 1
    (3 bits)
    Options::nChannels = static_cast<size_t>(inputStream.readNBits
    (3));

    // Store SampleRate (16 bits)
    Options::sampleRate = static_cast<size_t>(inputStream.readNBits
    (16));

    // Store Quantization Levels (16 bits)
    Options::quantizationLevels =
        static_cast<size_t>(inputStream.readNBits(16));

    // Store DCT Fraction (7 bits > max 100)
    Options::dctFrac = double(inputStream.readNBits(7)) / 100.0;

    // Store Total Number of frames (32 bits)
    Options::nFrames = static_cast<size_t>(inputStream.readNBits
    (32));
}
```

Listing 4.6: Read header function

The main function will create the resulting decoded audio file, by reading the data in the file by bits and according to the size of each value specified in the header. Then, the values will be divided into blocks, as if they just resulted from a Direct Cosine Transformation (DCT) and the Inverse DCT will be applied, resulting in a set of frames that will be then saved to the decoded audio file.

```
int main(int argc, char* argv[]) {
    (...)

    BitStream inputBitStream{'r', Options::encodedName};

    read_file_header(inputBitStream);

    (...)

    SndfileHandle sfhOut{Options::musicName, SFM_WRITE,
                         SF_FORMAT_WAV | SF_FORMAT_PCM_16,
                         static_cast<int>(Options::nChannels),
                         static_cast<int>(Options::sampleRate)};
    if (sfhOut.error()) {
        cerr << "Error: problem generating output .wav file\n";
        return 1;
    }

    // Read all the values in the encoded file
    std::vector<int> quantizedCoefficients;

    while (!inputBitStream.check_eof())
        quantizedCoefficients.push_back(
            inputBitStream.readNBits(Options::quantizationLevels));

    (...)

    // Vector for holding all DCT coefficients, channel by channel
    vector<vector<double>> x_dct(Options::nChannels,
                                 vector<double>(nBlocks * Options::
    blockSize));

    // Divide the coefficients vector into blocks
    int index = 0;
    for (size_t n = 0; n < nBlocks; n++) {
        for (size_t c = 0; c < Options::nChannels; c++) {
            for (size_t k = 0; k < Options::blockSize; k++) {
                x_dct[c][n * Options::blockSize + k] =
                    double(quantizedCoefficients[index++]) / 100.0;
            }
        }
    }

    // Vector for holding DCT computations
    vector<double> x(Options::blockSize);

    std::vector<short> outputSamples(Options::nChannels * Options::nFrames);
    // Do zero padding, if necessary
    outputSamples.resize(nBlocks * Options::blockSize *
                         Options::nChannels);

    // Inverse DCT
    fftw_plan plan_i = fftw_plan_r2r_1d(Options::blockSize, x.data(), x.data(),
                                        FFTW_REDFT01, FFTW_ESTIMATE);
    for (size_t n = 0; n < nBlocks; n++)
        for (size_t c = 0; c < Options::nChannels; c++) {
            for (size_t k = 0; k < Options::blockSize; k++)
                x[k] = x_dct[c][n * Options::blockSize + k];

            fftw_execute(plan_i);
            for (size_t k = 0; k < Options::blockSize; k++)
                outputSamples[(n * Options::blockSize + k) *
                                  Options::nChannels +
                              c] = static_cast<short>(round(x[k]));
        }

    sfhOut.writef(outputSamples.data(), Options::nFrames);

    (...)
}
```

Listing 4.7: Read header function

4.1.3 Usage

Lossy Encoder

The program can be launched using the following command:

```
./encoder_lossy [OPTIONS]
```

To check the available options, use the -h option, like:

```
./encoder_lossy -h
```

This command will generate the following text in the console:

```
Usage: %s [OPTIONS]
  OPTIONS:
  -h, --help         --- print this help
  -i, --input        --- set music file name (default: sample.wav)
  -o, --output       --- set encoded file name (default: encodedSample)
  -b, --blockSize    --- set block size (default: 1024)
  -l, --levels       --- set Quantization Levels (default: 8)
  -d, --dctFrac      --- set Dct Frac (default: 0.2)
```

Listing 4.8: wav_effects arguments

Lossy Decoder

The program can be launched using the following command:

```
./decoder_lossy [OPTIONS]
```

To check the available options, use the -h option, like:

```
./decoder_lossy -h
```

This command will generate the following text in the console:

```
Usage: %s [OPTIONS]
  OPTIONS:
  -h, --help        --- print this help
  -i, --input       --- set encoded file name (default:
    encodedSample)
  -o, --output      --- set encoded file name (default:
    decodedSample.wav)
```

Listing 4.9: wav_effects arguments

4.1.4 Results

In the following subsection we'll be approaching the results of several executions conducted with 4 files, as to evaluate various aspects of our program. Only the files changed, the parameter stayed the same. Which were:

```
 - Block Size: 1024
 - Number of Channels: 1
 - Sample Rate: 44100
 - Quantization Levels: 8
 - Dct Frac: 0.2
```

Performance

As predicted, the file size of the audio music contributes to the time of encoding and decoding, with linear correlation. As the size of the file increase, the amount of times to encode and decode it also increases. This can be observed in Figure 4.2.

Reduced file size

The encoded file resulting from executing the *encoder_lossy* program, resulted in an average compressing of 75% compared with the initial file and after executing the *decoder_lossy*, we obtained an audio file that was 49% smaller, in average, than the original audio file. The trend can be observed in Figure 4.3, where we can see the increase in file sizes on a linear rate.

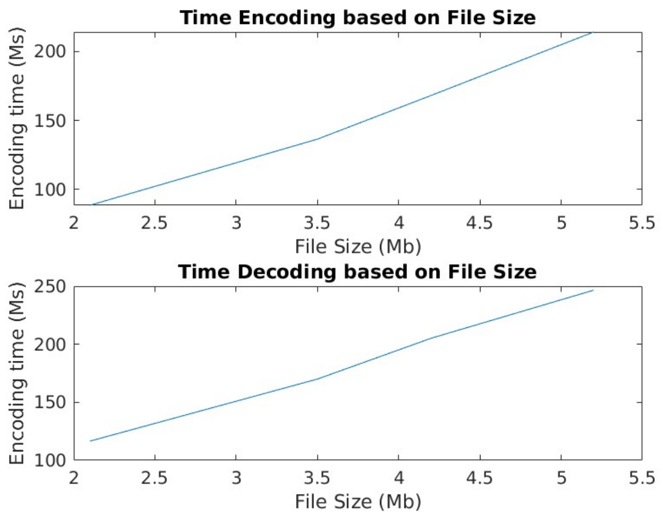


Figure 4.2: Time Encoding/Decoding based on File Size

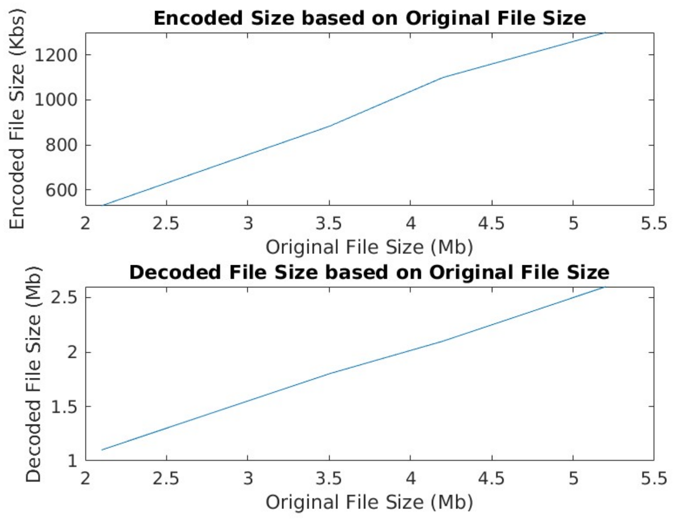


Figure 4.3: Encoded/Decoded Size based on Original File Size

Noise

Our testing showed a some levels of noise in the various steps of the program (applying the DCT, quantizing and storing the values into the encoded file). The parameter $dctFrac$ also allows to increase/reduce the amount of errors when applying the DCT, but this can be mitigated by using the value 1.0, which will not discard any blocks. However, our biggest introduction of noise is located in the quantization (as expected) and the amount of bits to store in the file (sort of unexpected). We can justify this by observing the quantization function, which will essentially eliminate as many LSB's as the number of levels supplied by the user. But this function will still retain the position of each bit in a coefficient. After this processing, the coefficient is then stored in the file, which will save the values according to the specified bits. This means that for example, a 32 bits signed integer like 10110...110 will look like this after quantized 10110...000 and will be stored prioritizing the MSBs, and when the decoder reads the value, it doesn't alter it into a 32 bit signed integer, by performing the necessary shift, it retains the number of levels specified by the user. This will result in values that are lower. What does this mean? Means that if we use a quantization variable of 32 bits (acts like 16 bits in our quantization function) which will store and load the values in 32 bits, we get almost no noise, however the sound level of the music get's substantially reduced. Lower numbers in bits than that will introduce a substantially amount of noise. However this can be sort of atenuated by adjusting the $blockSize$ and $dctFrac$, but the truncation and quantization of values is still there. In our approach it makes sence, since if we want to quantize an audio file with 8 bits. This means that each coefficient in the encoded file should be 8 bits and each sample should be loaded with 8 and not 32 bits.

Contributions

The work carried out in this project was equally distributed between all members, so the evaluation is the same for all.