# Audio and Video Coding - Assignment 2
# Lossless and Lossy Audio Codecs

André Pinho
Email: andre.s.pinho@ua.pt
Nmec: 80313
DETI
UA

Diego Hernandez
Email: dc.hernandez@ua.pt
Nmec: 77013
DETI
UA

Margarida Silva
Email: margaridaocs@ua.pt
Nmec: 77752
DETI
UA

***Abstract*—Report of our implementation of the Assignment 2 of Audio and Video Coding. The proposal was to create a lossy and a lossless predictive audio codec to achieve some level of compression.**

## I. Introduction

In a significant amount of data sources, statistical and perceptual redundancy are present, which makes it possible to compress data. This can be done by transforming one message into another message with less bits, as long as we also provide means for its reconstruction (lossless compression), or by exploiting the imperfections of our senses and eliminating information that has low relevance (lossy compression).

In this assignment we explored this subject by implementing an audio codec.

## II. Compile instructions

The project contains a `CMakeLists.txt` on its root directory. In order to build the sources, a `build/` folder should be created on the same level as the cmake file. Inside the build folder, run `cmake ..`. This will create a `MakeFile` (and other files) that can be executed using `make` to generate the final binary (mad_codec).

## III. Run instructions

The generated binary can encode and decode audio files by providing some arguments. When executing it without any parameters, it will print the usage guide. The valid parameters are stated in Table I;

TABLE I
Minimal flags to get the encoder and decoder to run

| **Mandatory arguments** | | | |
|---|---|---|---|
| **Short flag** | **Long flag** | **Description** | **Valid values** |
| -i | –in | Input file | Any string |
| -o | –out | Output file | Any string |
| -e | –encode | Indicates the program should encode the input file | N.A. |
| -d | –decode | Indicates the program should decode the input file | N.A |
| **Mandatory encoding arguments** | | | |
| -y | –lossy | Employ lossy encoding | N.A. |
| -n | –lossless | Employ lossless encoding | N.A. |

These are the flags that are required. Either the encode or decode flag should be set, as the binary will only behave as one of it. The lossy or lossless flag has to be set for the encoder to work.

TABLE II
Optional flags

| **Optional encoding arguments** | | | |
|---|---|---|---|
| **Short flag** | **Long flag** | **Description** | **Valid values** |
| -w | –window | Window size for which the 'm' will be calculated | >1 |
| -s | –samples | Amount of samples to be skipped before calculating a new 'm' | 0 <= v <w, w = window |
| -a | –auto | Automatically calculate an approximation to the 'ideal' values for the window and skip | 0 <= v <= 4 |
| -p | –predictor | Order of the predictor | 0 <= v <= 3 |
| **Optional encoding arguments for stereo** | | | |
| -b | –windowstereo | Window size for the channel differences | >1 |
| -c | –samplestereo | Amount of samples to be skipped for the channel differences | 0 <= v <w, w = window |
| **Optional arguments for lossy encoding** | | | |
| -q | quantization | Number of bits to be quantized<br><br>in the predictor | 0 <= v <16 |

The parameters in Table II are optional but enable a more fine tuned control over the encoding process. The implementation meaning of these parameters is later explained.

Lossless encoding of a .wav file:

```
mad_codec -i sample01.wav -o sample01.mad -e -n
```

Lossy encoding of a .wav file:

```
mad_codec -i sample01.wav -o sample01.mad -e -y
```

Decoding of a .mad encoded file:

```
mad_codec -i sample01.mad -o sample01.wav -d
```

## IV. Libraries

### A. Bitstream

To aid the read and write operations, a Bitstream class was created, which provides diverse methods for processing a single or several bits or even chars. Different function signatures are available to suit our needs more appropriately, for example, n bits can be set by providing a single 32 bit integer (in case the desired amount of bits is lower than 33) or by sending a pointer to an array of integers. The methods that target chars facilitate the usage of the bitstream to write ordered text.

This class extends fstream, making use of the existing read, write and close methods. When closing the stream, a padding of zeros is added to the end of the file in case it is required to fulfil a byte.

In this library there is also a bitstream wrapper class, used by the parametizer so it is possible to count the number of bits that would be written without writing them to disk, but also with the option to call the bitstream methods and actually writing.

### B. Golomb

The Golomb coding method is an algorithm for generating variable size codes. The main principle is to (integer) divide a number by a given value $m$, and represent it with unary code, and get the reminder of the division, and store it in natural binary code. One of the most important features of Golomb codes is that they are optimum for codes that follow a geometric distribution. Audio (almost) follows a geometric distribution of values, but it's reflected on zero. To solve this, we used the method where negative numbers are represented in odd positive numbers and positive numbers are represented in even positive numbers.

Since the beginning of the project, we were interested in creating an efficient backward adaptive codec. This means that the codec only has to make one pass on the data (thus being faster). This implementation of the algorithm works with a sliding window over the samples. Since the algorithm's performance relies on choosing the right $m$, we calculate the $m$ of the window (at a certain rate), and use it to encode new values. The decoder can follow the same steps of the encoder and will be able to replicate the same $m$s. This means that the only information that needs to be stored in the file are the size of the window, the amount of samples between each new approximation of $m$, and the samples themselves (or more precisely, the residual of the prediction of that sample).

The calculation of $m$ is done using the mean of the geometric distribution. Internally, the sum of all values of the window is kept updated to cut down on calculation times. Using that sum, we divide by the size of the window to calculate the mean, and apply (explained on code on file `src/lib/golomb/golomb.cpp`, function: `predict_m`):

$$m = \left\lceil \frac{-1}{log_2(\frac{mean}{mean+1.0})} \right\rceil$$

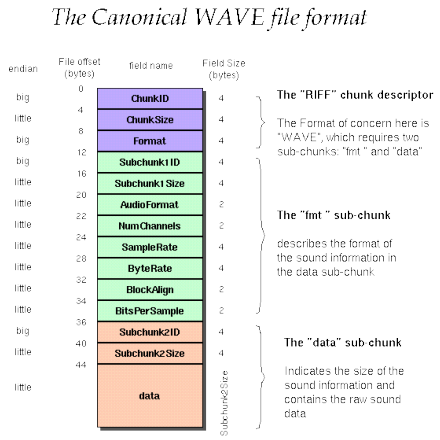


Fig. 1. WAV data format.

An auxiliary module called `golomb_bitstream` was created to serve as an interface to the bitstream file writer module.

### C. Wav

A library focused on reading and manipulating WAVEform audio format (WAV) sound information has been implemented. Hence, no existent sound library, such as sox, has been used for the development of the assignment in question.

It was crucial to understand the WAV file structure, its header, 1, the data structure, and finally the endianness of the data samples.

With this library, the programmer can load a WAV file, read the WAV sound header and data. Besides that, the programmer is able to create a WAV file without the need to load one. The header can be created, and its parameters can be modified. The WAV data can also be created from start, extending the necessary space to register its values. They can be altered and selected. Moreover, it is possible to separate the channels data and store them in C++ vectors. Finally, after all the insertions and modifications, the WAV data in consideration can be written into a WAV file.

Summing up, with this library, the programmer is capable to do almost all manipulation operations to wav sound files, their headers and data.

### D. Predictor

The predictor class is another independent module that has been developed, fundamental for predictive coding. The following polynomial predictor was chosen:

$$\begin{cases} \hat{x}_n{}^{(0)} = 0 \\ \hat{x}_n{}^{(1)} = x_{n-1} \\ \hat{x}_n{}^{(2)} = 2x_{n-1} - x_{n-2} \\ \hat{x}_n{}^{(3)} = 3x_{n-1} - 3x_{n-2} + x_{n-3} \end{cases} \quad (1)$$

This module is adaptable for both lossy and lossless compression, having the necessary buffers and methods to implement the linear predictors, calculate the residuals and

to reconstruct the expected values in order to decompress the WAV sound file. The order of the predictor is specified on its instantiation, allowing the algorithms to choose the best one.

The corresponding residuals are computed as follows:

$$\begin{cases} \hat{r}_n{}^{(0)} = x_n \\ \hat{r}_n{}^{(1)} = r_n{}^{(0)} - r_{n-1}{}^{(0)} \\ \hat{r}_n{}^{(2)} = r_n{}^{(1)} - r_{n-1}{}^{(1)} \\ \hat{r}_n{}^{(3)} = r_n{}^{(2)} - r_{n-1}{}^{(2)} \end{cases} \tag{2}$$

### E. Quantization

A simple module that makes available two types of quantization, Midrise and Midtread Quantization. More detail about this quantizers are referenced in the first assignment report by the same authors of this report.

## V. Compression

### A. Stereo redundancy coding

Lossless and lossy compression modules have a specific manner to compress and decompress stereo WAV sound files. Calculating and writing the residuals for each channels' samples is clearly not the most efficient manner of compressing this type of files. For a more efficient compression of stereo files, the data compression algorithm has been based on how data sound is recorded on a vynil.

First, in the file encoding process, the module calculates two types of values. The mean sample value (X) and the difference (Y) between both channel's sample values. Then, the residual is obtained based on the calculated values. If the compression is lossy, the same residual values are quantized with the midtread quantization method. The values are finally written to a file in a sequential manner, respectively.

$$X = \frac{L + R}{2} \tag{3}$$

$$Y = L - R \tag{4}$$

At the file decoding process, right after the reconstruction process, previously mentioned X and Y values derived from the respective channel's samples are obtained. The sample value for each channel is calculated with the following formula:

$$\begin{cases} R = \frac{2*X+(Y\%2)+Y}{2} \\ L = \frac{2*X+(Y\%2)-Y}{2} \end{cases} \tag{5}$$

When calculating the X value, the sum between the value L (sample value of the Left audio channel) and R (sample value of the Right audio channel) can be an odd number (if one and only one of the sample values is odd), and thus X could be a non integer. As such, because we only write Integer values to the compressed file, the fractional part (0.5) is truncated. To work around this situation, and knowing that this happens consequently when Y value is also an odd number, we increment at the dividend the module of Y by 2 at both equations with the purpose of recovering the real sample value of both channels.

Using X and Y values and resorting to the linear prediction techniques and variable-length coding, Golomb code, we achieved a significant level of compression, whether using the lossy or lossless compression techniques.

### B. Lossless audio codec

This module handles the lossless compression and decompression of our codec. It starts by reading a wav file and constructing the headers of our compressed file. If there are two channels, two `predictors/golomb_bitstream` pairs are instantiated (for the same `bitstream`), otherwise only one pair is created. All the samples pass through the predictor, but the first sample of the wav is sent in natural binary, since the predictor can't guess the first number. This makes it inefficient to store in a Golomb code. The rest of the samples are stored by the `golomb_bitstream`. The decoding process follows symmetric pipeline.

There was an effort to implement an offline codec (file `src/lossless/offline_lossless.cpp`). This kind of codec pre-processes the data to get the best $m$ and related attributes. Unfortunatly, the timeframe we had didn't allow us to finish this implementation, but we did implement a similar behaviour with the Parametizer module, making the codec capable of being a hybrid (offline and online) codec.

The Table III represents the header of this format.

TABLE III
HEADER FOR LOSSLESS MAD

| Attribute | Size (in bits) |
|---|---|
| Magic number (CAVN+) | (5+1)*8 |
| Stereo indicator | 1 |
| Predictor order | 2 |
| Number of samples | 32*8 |
| Initial $m$ | 32*8 |
| Window size | 32*8 |
| $m$ calculation interval | 32*8 |
| Initial $m$ for stereo | 32*8 |
| Window size for stereo | 32*8 |
| $m$ calc int for stereo | 32*8 |

### C. Lossy audio codec

The lossy encoding module has a similar high-level flow as the lossless one, except before sending the residuals, the codec quantizes the residual, re-sends it to the predictor (so it can synchronize with the error). Without this the decoder wouldn't synchronize with the coder.

The Table IV represents the header of this format.

### D. Parametizer

The intention of the Parametizer module is to calculate near optimum values of the window size and window skip of the stereo audio samples.

Parameters work as follows: between a possible range of window size values that can be used for the Golomb procedure, the module iterates through a finite number of window sizes and window skips and other Golomb module parameters

TABLE IV
HEADER FOR LOSSY MAD

| Attribute | Size (in bits) |
|---|---|
| Magic number (CAVN-) | (5+1)*8 |
| Stereo indicator | 1 |
| Predictor order | 2 |
| Number of samples | 32*8 |
| Num Quantized bits | 32*8 |
| Initial $m$ | 32*8 |
| Window size | 32*8 |
| $m$ calculation interval | 32*8 |
| Initial $m$ for stereo | 32*8 |
| Window size for stereo | 32*8 |
| $m$ calc int for stereo | 32*8 |

values that belongs to the respective domain, and applies the variable coding procedure with such parameters to the sound data. It retrieves the total number of bits of the encoded data, this is the total number of bits of the compressed audio file. As the following number keeps getting lower with other window sizes that are part of the selected domain, the module records the parameters, such as window size, window skip and the linear prediction order in order to obtain such compression of the sound file. It also keeps on record the total number of bits to compare with other window size cases and substitutes the mentioned parameters by others if other window size is capable of generating a more compressed file.

Summing up, every time the "hypothetical" output tends to get a lower size during the iteration and application of the variable coding algorithm, the parameters that were able to accomplish it are registered at a dedicated module structure.

Parametizer has been tested with several different WAV audio files, using different and finite numbers of window sizes, window skip values and other parameter values from the Golomb module. Furthermore, we have verified that an extensive search and analysis of the different possible parameters tends to be useless and time-punishing. This is, the Golomb module parameters that are responsible for generating the most compressed file can be approximately obtained by just analysing a small set of window sizes and window skips values, leading to a faster determination of the mentioned parameters, with no need of doing an extensive search for the optimum parameters.

However, a more extensive search and analysis of the possible Golomb module parameters are more likely to correctly lead to the determination of parameters that can generate the most compressed file correspondent to the sound data file in consideration.

We have decided to create various presets responsible to search the optimum parameters. Some do extensive search and calculations and others do a lighter and faster search. This presets are mapped with a degree of motivation. As mentioned, it has been verified that a light and fast can generally come up with a good valid compression length value.

## VI. RESULTS

### A. Compression ratio and times

To check the correctness and efficiency of our algorithm, we compared the results of our codec with different parameters and also FLAC. The used parameters were both lossy and lossless compression with the default values and with algorithmically fitted values with a motivation value of 0.

TABLE V
COMPARISON OF THE COMPRESSION RATIO AND TIMES

| Compression Method | Sample | Compression Time (s) | Compression Rate |
|---|---|---|---|
| **Lossless** | sample01 | 0.277 | 1.313 |
| **Lossless Auto** | | 9.678 | 1.361 |
| **Lossy** | | 0.264 | 2.655 |
| **Lossy Auto** | | 9.828 | 2.869 |
| **FLAC** | | 0.119 | 1.499 |
| **Lossless** | sample02 | 0.149 | 1.351 |
| **Lossless Auto** | | 5.274 | 1.410 |
| **Lossy** | | 0.132 | 2.754 |
| **Lossy Auto** | | 5.150 | 3.032 |
| **FLAC** | | 0.084 | 1.588 |
| **Lossless** | sample03 | 0.187 | 1.534 |
| **Lossless Auto** | | 6.943 | 1.555 |
| **Lossy** | | 0.179 | 3.499 |
| **Lossy Auto** | | 6.771 | 3.724 |
| **FLAC** | | 0.093 | 1.780 |

The lossless compression was able to reach compression rates close to those observed in FLAC, although slower. Using the auto flag, the rates slightly improved but to a great cost of compression time. As expected, the lossy approaches held significantly better results.

### B. Lossy codec SNR

In order to encode a WAV sound file with lossy audio codec the following command must be executed:

```
mad_codec -i sample01.wav -o sample01_q1.mad -e -y
-q 1
```

In this example we compress sample01.wav applying lossy audio codec, removing the first least significant bit of the samples' values at the encoding process. The compressed file is written with the na,e sample01_q1.mad.

To decompress the file and retrieve the correponding wav file, the following command must be executed:

```
mad_codec -i sample01_q1.mad -o sample01_d1.wav -d
```

We have compressed the sample04.wav with the lossy codec. According to the number of least significant bits removed at the quantization module, it has been obtained the SNR values presented on Table VI.

It can be verified that applying a quantization that removes more than 11 LSB of the samples is dominated mostly by noise, easily for the user notice elevated noise at the reproduction of the sound WAV file.

TABLE VI
SNR LOSSY CODEC

| N$^o$ bits | SNR |
|---|---|
| 1 | 75.4223 |
| 2 | 67.6268 |
| 3 | 60.7263 |
| 4 | 54.2724 |
| 5 | 48.0461 |
| 6 | 41.9183 |
| 7 | 35.8044 |
| 8 | 29.71 |
| 9 | 23.6467 |
| 10 | 17.5892 |
| 11 | 11.5492 |
| 12 | 5.51178 |
| 13 | -0.556405 |
| 14 | -6.87945 |
| 15 | -11.7444 |

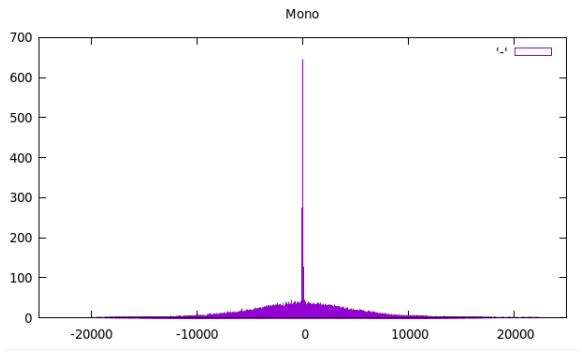


Fig. 2. Histogram of the samples' values of sample04.wav.

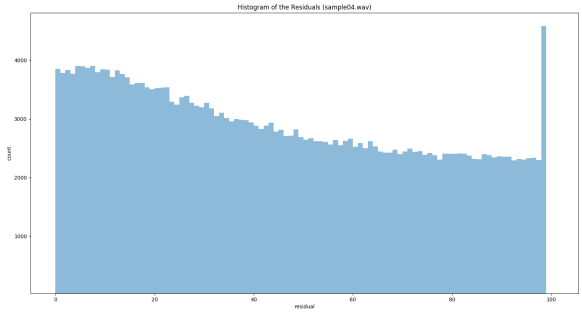


Fig. 3. Histogram of the residuals of sample04.wav.

### C. Histograms

The Figure 3 represents the histogram of residual of the WAV audio file sample04.wav. With the predicted value $(\hat{x})$ and the current value of the sample $x$, the residual is calculated as following:

$$r_n = x_n - \hat{x}_n \tag{6}$$

Polynomial predictors are used in the audio encoders, in order to compute the predicted value and the residuals. These are calculated by using the equations (1) and (2) respectively.

## VII. CONCLUSIONS

After ending this assignment, we observed that with a combination of simple algorithms it was possible to achieve compression rates similar to the ones FLAC presents. On the other hand, the compression and decompression times from these algorithms proved to be significantly better than ours.

When using more complex approaches on lossless compression to get parameters closer to the "ideal" values, some improvement could be noticed, but the compression times had to be highly sacrificed for this.

Employing strategies to tackle perceptual redundancy, we could reach higher levels of compression without a significantly noticeable difference to our hearing and with a relatively low compression time.

This proves the usefulness of lossy compression mechanisms, with a good balance between audio quality and amount of data it is an appropriate solution in a bandwidth critical scenario.