CAV - Developing a lossless audio codec

Nuno Humberto Nuno Miguel Silva

***Abstract* – One of the most used approaches to losslessly compress data takes advantage on the usage of Golomb coding. In lossless audio compression, it is used for encoding audio prediction residuals. The main objective of this assignment is to develop a lossless audio codec that produces a compressed file as small as possible. By dividing the file in smaller partitions and discovering optimal parameters for each partition, the final compressed size can be successfully minimised. The developed codec is able to losslessly compress audio files with higher compression ratios than FLAC at its fastest compression mode. Resulting files, in the best tested case, end up with 45.97% of the original file size.**

***Keywords* – golomb, audio compression, entropy calculation, lossless codec**

I Scenario

I-A Description

In this project, all stages of a lossless audio codec were developed. Achieving this functionality requires several functions with different purposes. This report describes the process of both encoding and decoding techniques. Methods for calculating and displaying histograms were also developed.

For performance and timing analysis, the following audio samples were used:

Sample	Title	Length
A	Rainbow - I Surrender	0:29
B	Carlos Paredes - Verdes Anos	0:15
C	Pink Floyd - Pigs On The Wing	1:25
D	Pink Floyd - Free Four	4:16
E	aivi & surasshu - Nucleus	1:21

Table I: Used audio files

The first two samples are the ones provided in the *eLearning* page, while the other three were either extracted from their CDs or downloaded royalty-free from the internet.

II Encoding

II-A Stage 1 - File Loading

The first stage of performing an encoding operation is the file loading stage – it relies on acquiring every frame of the input audio file. These frames are treated differently depending on the channel they belong to. Left channel samples are stored untouched in a vector while right channel samples are not entirely kept, only their difference to the corresponding left channel sample is stored in a vector. This pre-processing takes advantage of the fact that usually left and right channels share a close correlation. However, when the user requests histogram drawing, all channels are entirely stored as-is for later displaying.

This stage ends with the calculation of the input entropy. This calculation is achieved using the following formula.

$$H = -\sum P(value) \log_2 P(value)$$

In which $value$ is the numeric value of the corresponding sample.
The calculated entropy value is then presented to the user.

II-B Optional Stage - Lossy Encoding

Lossy compression is also available in this codec by the usage of uniform quantization. When this mode is selected, the possible sample values are reduced to a finite set of symbols with tunable dimensions. The typical symbol space of the used audio files has a length of 65536. The user may choose exactly how this symbol space should be reduced by issuing a parameter when launching the codec. This parameter represents the number by which the symbol space should be divided.
For example, issuing a lossy parameter of 2 would cause the symbol space length to be reduced to 32768. Issuing a lossy parameter of 32 would cause the symbol space length to be reduced to 2048.

The resulting symbol space length corresponds to the number of reconstruction levels that will be available at decode time. The symbol decision for a specific sample is given by a decision level, which corresponds to every halfway between two reconstruction levels. If the sample is greater than that value (for positive values) or lower than that value (for negative values), its reconstruction will land in the next reconstruction level.
Since the symbol space can be reduced, samples with originally higher values can now be represented using smaller numbers, allowing smaller file sizes at the cost of a lower resolution.

II-C Stage 2 - Residual Calculation and Comparison

The second stage of the encoding procedure is the residual calculation. In our encoder, residuals are obtained by subtracting the value of each sample with the previous one, obtaining a first order residual. Higher order residuals are calculated using the same method, applied to the residuals of the previous order, until the fourth order is reached.

With the previous 4 residuals calculated, the encoder will now process the residuals in blocks with a fixed size (512 samples). For each set of samples, the sum of the absolute value of their corresponding residuals is calculated. The predictor which yields a lower sum is chosen for that particular block. This introduces an overhead since the encoder

$$\begin{cases} {r_n}^{(0)} = x_n \\ {r_n}^{(1)} = {r_n}^{(0)} - {r_{n-1}}^{(0)} \\ {r_n}^{(2)} = {r_n}^{(1)} - {r_{n-1}}^{(1)} \\ {r_n}^{(3)} = {r_n}^{(2)} - {r_{n-1}}^{(2)} \\ {r_n}^{(4)} = {r_n}^{(3)} - {r_{n-1}}^{(3)} \end{cases}$$

Figure 1: Residual Calculation

is storing two extra bits per block, to indicate the chosen order. However, individually choosing a predictor for each block allows using smaller values on the later Golomb coding phase, enabling better compression results.

The table below depicts the impact of the block size (in samples) in the final compressed file dimensions (in KB).

	Block Size			
Sample	128	256	512	1024
A	3517	3516	3516	3517
B	1747	1747	1746	1749
C	6737	6734	6733	6733
D	29426	29415	29412	29455
E	8391	8389	8389	8399

Table II: Block size impact

Analysing these results, one can observe that using excessively small block sizes causes the introduced overhead to be bigger than the compression acquired by using the ideal predictor.

In addition, to keep the predictor overhead at two bits per block, only a maximum of four predictors can be used. For this codec, two cases were tested. One in which all zero-to-third order predictors were used, with the zero-order residual being the original signal itself, and another in which all first-to-fourth order predictors were used.

$$\begin{cases} {x_n}^{(0)} = 0 \\ {x_n}^{(1)} = x_{n-1} \\ {x_n}^{(2)} = 2x_{n-1} - x_{n-2} \\ {x_n}^{(3)} = 3x_{n-1} - 3x_{n-2} + x_{n-3} \\ {x_n}^{(4)} = 4x_{n-1} - 6x_{n-2} + 4x_{n-3} - x_{n-4} \end{cases}$$

Figure 2: Fixed predictors used

To understand which of the cases normally yields better results, counts were taken of how many times a specific predictor was the 'winner' for every single block. Meaning this specific predictor produced the lowest sum of absolute values among all other predictors.

Notice, in Case 1, the heavy usage of the first and second order predictors and the slight less intense usage of the original signal. With sample A, the original samples are not used at all.

Observing the counters on Case 2, one can notice that the fourth order predictors remain sometimes unused, but still interestingly provide a frequent choice in samples D and E.

	Predictor			
Sample	Original	Order 1	Order 2	Order 3
A	0	1534	2915	607
B	100	874	1483	72
C	192	5674	6965	1817
D	1135	22218	14437	6306
E	105	4539	4867	4486

Table III: Case 1: Zero-to-third order

	Predictor			
Sample	Order 1	Order 2	Order 3	Order 4
A	1534	2915	607	0
B	974	1483	72	0
C	5866	6965	1815	2
D	23353	14437	5759	547
E	4644	4867	2579	1907

Table IV: Case 2: First-to-fourth order

These results leave, for further investigation, interest in analysing the performance of using eight different predictors instead of four, at the cost of an extra overhead bit per block.

II-D Stage 3 - Discovering Optimal Partition Count

The third stage has the function of discovering the best size to use when creating partitions of sample blocks. Individual samples are encoded using Golomb coding. This coding technique relies on a tunable parameter m. Each partition may have a different m used for Golomb coding. To obtain the optimal block per partition ratio, the algorithm firstly experiments only using one block in each partition and calculates the final file size. Then, it makes the same experiment using two sample blocks in each partition and so on. While increasing this value (partition factor) and analysing its performance, the compressed file size is expected to decrease until a point where it starts to grow again. At this point, the last best value acquired is used as the number of blocks per partition.

Since, for every partition, an ideal value for m must be discovered, two approaches were attempted for this purpose. The first and simplest consists in testing every power of 2 up to 8192, which was the maximum value observed for m. The second approach simply attempts increasingly higher values of m until the estimation of the size stops returning decreasing values.

Below is the comparison of the total file size obtained (in KB) using the two distinct approaches.

Observing the results, no difference between the two approaches is found. In the presence of this result, we've eliminated the approach which tests every power of 2 up to 8192, since simply attempting values until the estimation stops decreasing requires, indeed, less processing time and yields the same result.

	M discovery approach	
Sample	Up to 8192	Until it stops decreasing
A	3516	3516
B	1746	1746
C	6733	6733
D	29412	29412
E	8389	8389

Table V: Comparison of two M discovery methods

II-E Stage 4 - Writing encoded file

The final stage of encoding is writing the Golomb Code into a binary file. A header has to be set in the binary file so that in the future a decoder may know how the file was encoded in order to restore its original form.

Listing 1: Encoded file header

```
Number of samples -> 8 bytes
Block size -> 4 bytes
Channels -> 1 byte
Partition factor -> 1 byte
Lossy factor -> 4 bytes
```

This header is composed of the number of samples the audio file has (8 bytes), block size (4 bytes), number of channels (1 byte), number of blocks per partition (partition factor) (1 byte), and the lossy factor (4 bytes). Once the header has been written, it is time to encode the residues to Golomb coding and write them to the file. Firstly, all residuals have to be non-negative, and for this the numbers closest to zero have to remain the closest possible to it when passing from negative to positive. The following equation is used for this first step:

$$\begin{cases} target = x \times 2, \text{ if x is positive} \\ target = -x \times 2 - 1, \text{ if x is negative} \end{cases}$$

Then, it checks if it's time to create a new block, and if that's the case, it can also be the start of a new partition, so these two parameters are crucial when writing the Golomb encoded residuals to the file. If a new partition is to be created, it means that a different m may be going to be used for encoding in this partition. Furthermore, if a new block is to be created, it means a new predictor may be chosen. This information is stored whenever a new partition or block is created because it is crucial for the decoder to know these variables to recover the initial file. The encoder then proceeds to calculate the q and r for each target residual:

$$\begin{cases} q = \lfloor \frac{n}{m} \rfloor \\ r = n - qm \end{cases}$$

The codec also features a *fake* mode, in which every output (file header, predictor choice, m parameters and all Golomb coding) is stored in readable text instead of binary, one per line.

II-F Optional Stage - Calculating and Displaying Histograms

When the user requests histogram calculation, the codec will keep a count of every occurrence of each possible sample value on left and right channels. This counting allows histograms to be later drawn. Histogram drawing for left channel, right channel, channel average (mono) and channel sum are supported.

Upon data acquisition, the counts for each value are extracted and placed in an OpenCV matrix. The matrix is then normalised so that it is possible to draw a histogram with a fixed height, which peak always represents the highest count present in the original matrix. Since drawing the histogram in its full would require an horizontal resolution of 65536, the actual matrix is condensed into a fixed width matrix by subsequently averaging adjacent values.

II-F.1 Example #1 - Noise with a normal distribution

This example shows the histogram calculation and drawing when the input is a file which contains noise with a Gaussian distribution.

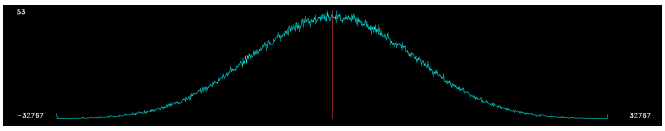

Figure 3: Gaussian distribution audio file histogram

II-F.2 Example #2 - Noise with a uniform distribution

This example shows the histogram calculation and drawing when the input is a file which contains noise with a uniform distribution.

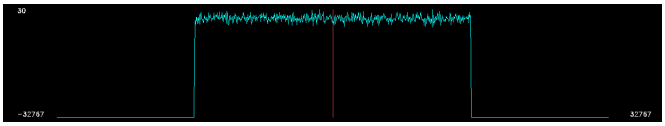

Figure 4: Uniform distribution audio file histogram

II-F.3 Example 3 - Original vs Residuals

This example shows the histogram calculation and drawing when sample A is used. It focuses on demonstrating the difference in the range of values between the original file and its residuals.

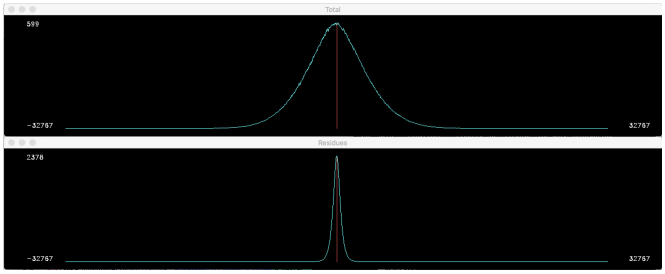

Figure 5: Range difference between an audio file and its residuals

II-G Encoding results

In this section it's presented the result of the developed encoder. Here follows a table with some results:

Sample	Original Size (KB)	Compressed Size (KB)	Encoding Time (s)
A	5176	3600	2.204
B	2589	1788	0.865
C	14999	6894	3.851
D	44095	29412	14.22
E	14332	8590	4.316

Table VI: Encoder Compression Results

Sample	Compress Ratio	FLAC level 0 Compress Ratio
A	69.56%	70.41%
B	69.08%	65.85%
C	45.97%	49.11%
D	66.71%	66.73%
E	59.94%	62.10%

Table VII: Comparison between our codec and FLAC

After these results we can conclude for these five sample an average compression ratio of 62.25%. As a challenge from the teachers for achieving better compression than FLAC, we can compare our results with the average compression ratio of 62.8% from FLAC.

The encoder with lossy features was also tested and the following table shows the results:

Symbol Space Divisions	Final Size (KB)	Compression Ratio
2	5993	39.95%
4	5131	34.20%
8	4358	29.05%
16	3719	24.79%
128	2866	19.10%

Table VIII: Lossy encoding of sample C

One can infer from this results that the compression ratio shows better outcomes from the regular encoding at the cost of having less precision when decoding the file. An experiment with 1024 symbol space divisions was made where the decoded audio file presented a sounds very much similar to the original file where the human ear can badly distinguish the difference.

III Decoding

Decoding an audio file encoded with the developed codec is a faster and simpler process when in comparison to encoding.

III-A Stage 1 - Header Reading

The first stage of the decoding process is the parsing of the encoded file header. This header follows the already mentioned structure and provides all the needed information for the decoding process to take place.

III-B Stage 2 - Residual Reading

After the structural information of the encoded file has been extracted from the file header, the codec initiates the residual reading process by obtaining every residual individually. Since it parsed the file header, it has enough information to ascertain whenever each block or partition starts, proceeding to read their correspondent predictor choice or m parameter. Acquiring the correct m parameter allows the Golomb-encoded residuals to be decoded to their original form.

$$\begin{cases} n = qm + r \\ residual = \frac{n}{2} \text{ if n is even} \\ residual = -\frac{n+1}{2} \text{ if n is odd} \end{cases}$$

III-C Optional Stage - Lossy Decoding

When the lossy factor read from the encoded file header is different than zero, the decoded detects that lossy compression was used at encoding time. Decoding audio files that were encoded using lossy compression add very little extra complexity to the normal decoding process.

Each sample is approximated to its reconstruction level. To accomplish this, each sample simply needs to be multiplied by a delta value.

$$\begin{cases} \Delta = \frac{SHORT\_MAX - SHORT\_MIN}{\frac{SHORT\_MAX - SHORT\_MIN}{L}} \\ L = \text{Lossy Factor} \\ SHORT\_MAX = 32768 \\ SHORT\_MIN = -32767 \end{cases}$$

III-D Stage 3 - WAV Reconstruction

When the original residual readings are obtained, the reconstruction of the actual samples may take place. This is done by using the actual predictor (periodically parsed from every block) to recover each sample using its residual. For this effect, direct substitution of the variables on the polynomials described in the Encoding section is performed.
After this process is repeated for every single residual, the actual left and right channels can be recovered – the left channel, which was stored as-is and the right channel, by adding the differences which residuals stored in the encoded file after the left channel residuals.

IV Program Usage

This section will describe the usage of our program on command line. The program options are the following: *-i* or *–input* follows the input file to be used, *-d* or *–decode* specifies the decode operation and follows the output file name, *-e* or *–encode* specifies the encoding operation and follows the output file name, *–draw* draws histograms, *-l* or *–lossy* followed by a number of symbol space divisions, enables lossy encoding or recovery/decoding of lossy encoded audio file, *-f* or *–fake* generates/decodes textual files and *–help* shows the help menu.

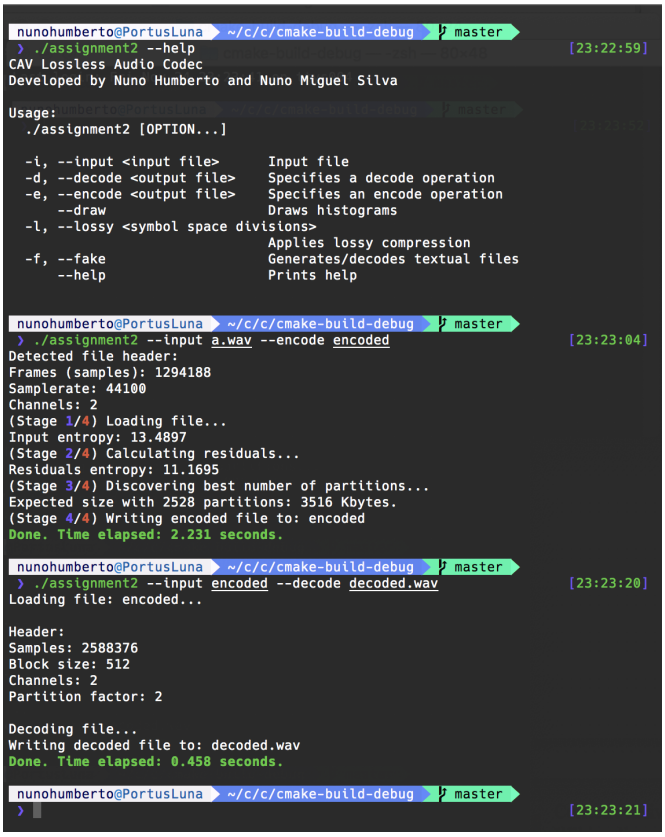

Figure 6: Program usage demonstration

V Conclusion

The main objectives were achieved successfully. It was an interesting project to deepen our knowledge in the C++ programming language and also to study the purpose of audio file compression algorithms, the usage of Golomb coding, residual calculation and polynomial predictors.

The main issues in the development process of this project were based on finding strategies to work around the problems we had, for example a way to convert negative numbers into non-negative in order to proceed to Golomb encoding and finding a polynomial equation for the fourth prediction order. In general, and how these problems were resolved, having worked around these somewhat unfamiliar obstacles helped our growth as future engineers.

References

https://xiph.org/flac/format.html#prediction
http://mi.eng.cam.ac.uk/reports/svr-ftp/auto-pdf/robinson_tr156.pdf
http://www.hpl.hp.com/techreports/1999/HPL-1999-144.pdf