An audio lossless codec with Golomb coding

João Fonseca, Pedro Silva and Rui Lopes
Departamento de Eletrónica,
Telecomunicações e Informática
Universidade de Aveiro, Portugal
Email: {jpedrofonseca,pedro.mfsilva,ruieduardo.fa.lopes}@ua.pt

***Abstract*—In professional uses of audio, the transmission of contents must save all the data from the original source, available on the files. On transmitting channels, the information must be compressed by proper mechanisms, in order to ensure that the communication are made without causing error by the overuse of bandwidth (in the case of a network) or on the file itself, as a consequence of a propagating error after a copy task.**

In this document we show the design and the implementation of a lossless audio codec which was made through the application of a set of predictors which allowed to establish relations between samples on a stream of audio. Such a codec retrieves the compressed files on a Golomb encoding.

Moreover, we have also implemented a lossy codec working with the residuals of the lossless procedures.

***Index Terms*—audio codec, lossless, lossy, residuals, Golomb**

I. Introduction

In a world full of media content transmissions over a global network such as the Internet, one of the favorite means, the *audio* is by far the most chosen option regarding its uses. Most of them to cast music, audio can then be fitted in the communication channels in one of two shapes, considering its final application: on critical applications, such as a copy of an integral original audio, a *lossless* should be chosen, instead of a *lossy* shape—these are of the entire responsibility of a set of mechanisms called *compression techniques*.

Such compression techniques are important simply because the need of transmission of a given content could produce severe consequences, if a network channel have some difficulties transmitting its data or if the data to be sent is of considerable sizes. Here, someone must be responsible of deciding which one of the methods of compression (that is, lossy or lossless) are meant to be used, given a specific application.

This report relates to the creation of a codec capable of compressing a 16-bits pulse-code modulation (PCM) audio in waveform audio file format (WAV). Using a specific coding technique called *Golomb coding* one could achieve high rates of compression by the simple choice of a parameter which relates to the depth of bits representation. As we only want to maintain, in our compressed files, the relevant data with no redundancy (or its minimum achievable), our design should implement a set of predicting entities which will be capable of only representing relations between samples in the original audio files.

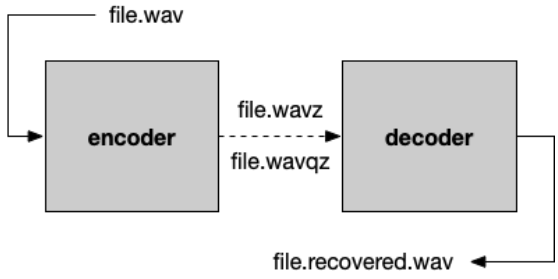

Fig. 1. A high-level system architecture

This work already has its competitors on the market, being parallel to compression codecs such as Free Lossless Audio Codec (FLAC). In fact, part of the compression techniques hereby stated and used are applied on the FLAC procedures.

More than a lossless codec, there is another goal to achieve with this work—develop a lossy codec which can profit from the residuals as evaluated through the prediction development of the lossless codec.

II. System Architecture

As an audio codec our system should be such that receives an audio file in WAV (in the 16-bits PCM format) and retrieves an inaudible compressed file, on an *encoding* phase. In other hand, in a *decoding* phase, the same system must be able to recognize a compressed file and retrieve, by the same (but inverse) techniques, an audio file, in the same format as the original one.

To ease the process of recognizing the files which present compressed content and its types of compression (lossless or lossy) we have defined that, as the input of the encoding task has its name ending on `*.wav`, the compressed lossless files will have their names ending on `*.wavz` and the compressed lossy (which have quantized audio) `*.wavqz`.

In the figure 1 is depicted the global architecture (on a higher level of abstraction) of our system.

As one can verify through figure 1, the system has two different processes: one of encoding and other of decoding. To perform one of the tasks, it is required that another entity (or a user) identifies which task wants to produce results of.

A. The encoding process

If the encoding process is chosen, then a WAV audio file is required to initiate its procedures. As specified by the FLAC documentation and implementation one could apply a combination of predictors to the original samples in order to use relations established between older samples (related to a current seek pointer location on a file) and the current one. More specifically, we have applied the following predictors, as in 1.

$$\begin{cases} \hat{x}_0[n] = 0 \\ \hat{x}_1[n] = x[n-1] \\ \hat{x}_2[n] = 2x[n-1] - x[n-2] \\ \hat{x}_3[n] = 3x[n-1] - 3x[n-2] + x[n-3] \end{cases} \tag{1}$$

An interesting property of the polynomials detailed in the approximations of 1 is that the result residual signals $e_p[n] = x[n] - \hat{x}_p[n]$, can be efficiently computed in the following recursive manner, as in 2, allowing the entire set of predicted samples to be computed more efficiently, without any multiplications.

$$\begin{cases} e_0[n] = x[n] \\ e_1[n] = e_0[n] - e_0[n-1] \\ e_2[n] = e_1[n] - e_1[n-1] \\ e_3[n] = e_2[n] - e_2[n-1] \end{cases} \tag{2}$$

To use all these predictors, one must test each frame (being a set of samples) with each one of the predictors and then estimate its residuals. After performing such action it is now required to choose which of the residuals' mean is more precise (more proximate to zero). Having chosen such predictor, its choice must be save next to the residuals on an output file.

This saving action must count with a Golomb coding module, that is, as soon as residuals are ready to be written on a file, they should be given to an Golombs' encoding procedure which, having accounted by a proper initialization value of m, translates 32-bit residuals into a smaller representation, easily readable by the respective decoding method.

As we are mentioning the Golomb encoding, its initialization value of m should be estimated at the beginning of each partition p, considering that $p \supseteq b$ as b represents a *block* (or a frame). This design allows to have the identification of a frame's Golomb initialization value (written in 4-bit Binary Coded Decimal (BCD)) without inducing a severe overhead into the compressed file. In fact, it is important to refer that, as the estimation of the Golomb codes have its basis on a sequence of division operations, if the initialization value of m is a power of base 2, that is $m = 2^k$, by finding exclusively a good k we can apply a brute-force search algorithm which could run a test within a partition p, by performing simple shift logical operations.

In the case of performing a lossy compression one should quantize the residuals with a given factor q. Such a *quantization* must be performed as a uniform quantization, that is, as a removal of the q least significant bits of each sample represented in the file.

In the figure 2 is depicted how does the encoding works.

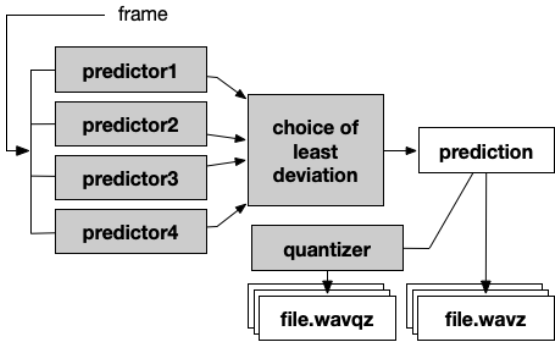

Fig. 2. The encoding process

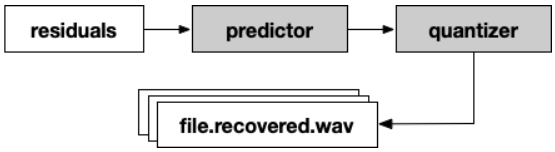

Fig. 3. The decoding process

B. The decoding process

In the decoding process one should give, as input to the system, a `*.wavz` (as a lossless compressed audio file) or a `*.wavqz` (as a lossy compressed audio file). The implemented software must be able to recognize the compression type of the given file and, in the case of the lossy one, be able to retrieve, from its header, the quantization factor q, as encoded by a 4-bit BCD code, since we cannot allow more than 15 bits to quantize—as its action will imply silence as the output.

Independently of the type of compression applied, in this decoding phase one should be able to identify which was the used predictor in each partition p and then apply it to retrieve the original values. Of course that, in case of the lossy type, the output will not give the original values, since it should be reverted the operation of quantization, by the same quantization factor, moving the q logically to the left.

In the figure 3 one can verify the process of decoding depicted.

III. Implementation

Our implementation begun by a simpler module to produce predictions, having in mind a more naive way to present results. Called NAIVEPREDICTOR, its only actions are to PREDICT() and to REVERT(), as we can see below:

Require: p as a vector of predicted residuals
1: {Revert function PREDICT() body}
2: **for** $i = 0$ **to** originalSamples.SIZE() **do**
3: p.ADD(originalSamples[i]−originalSamples[$i + 1$])
4: **end for**

Require: r as a vector of recovered samples
1: {Revert function REVERT() body}
2: **for** $i = 0$ **to** p.SIZE() **do**
3: p.ADD($r[i]-p[i+1]$)
4: **end for**

Such a module only computes the residuals having in mind the differences made from sample to sample. Its creation was executed since we would have a naive predictor to make comparisons to other predictors, to be developed along the time of the project.

Following such a creation we featured the creation of an advanced predictor. Called ADVANCEDPREDICTOR, this one applies the theoretical mentioned in the section before. Having described all the predictors and residual estimators on the INTEGERFINITEIMPULSERESPONSECOEFFICIENTS module, here, the implementation of the PREDICT() function follows this algorithm, as presented below:

Require: p as a vector of predicted residuals
1: originalSamples.INSERTATBEGIN(lastThreeSamples)
2: **for** $i = 3$ **to** originalSamples.SIZE() **do**
3: predictor0.ADD($\hat{x}_0[n]$) {Predictor 0}
4: deviation0.ADD($e_0[n]$)
5: predictor1.ADD($\hat{x}_1[n]$) {Predictor 1}
6: deviation1.ADD($e_1[n]$)
7: predictor2.ADD($\hat{x}_2[n]$) {Predictor 2}
8: deviation2.ADD($e_2[n]$)
9: predictor3.ADD($\hat{x}_3[n]$) {Predictor 3}
10: deviation3.ADD($e_3[n]$)
11: **end for**
12: lastThreeSamples ← originalSamples[-3 .. -1]
13: $k \leftarrow$ GETINDEXOFBESTPREDICTOR()
14: **if** k is 0 **then**
15: residuals.APPEND(deviation0)
16: **else if** k is 1 **then**
17: residuals.APPEND(deviation1)
18: **else if** k is 2 **then**
19: residuals.APPEND(deviation2)
20: **else if** k is 3 **then**
21: residuals.APPEND(deviation3)
22: **end if**

Similarly as before, we also did create the REVERT() method, which allowed us to recover the samples from the residuals. The function itself is described below:

Require: r as a vector of recovered samples
1: r.INSERTATBEGIN({0, 0, 0})
2: **for** $i = 1$ **to** residuals.SIZE() **do**
3: **if** currentPredictor is 0 **then**
4: predictorResult = $\hat{x}_0[n]$
5: **else if** currentPredictor is 1 **then**
6: predictorResult = $\hat{x}_1[n]$
7: **else if** currentPredictor is 2 **then**
8: predictorResult = $\hat{x}_2[n]$
9: **else if** currentPredictor is 3 **then**
10: predictorResult = $\hat{x}_3[n]$
11: **end if**
12: r.ADD(residuals[i] + predictorResult)
13: **end for**
14: r.ERASETHREEFIRSTELEMENTS()

Our implementation also have, as stated before, a module named GOLOMB able to encode and further decode a bitstream into Golomb coding. Such a module is positioned after the execution of the PREDICT() method of the ADVANCEDPREDICTOR module—right before saving the residuals to file—and before the execution of the same module's REVERT() method—right after reading values from a compressed file.

In the section before we designed a solution having in mind the re-initialization of the m value of the Golomb coding, throughout the compression execution, by partitions. Unfortunately, as we could not implement a method to get a viable number of blocks per partition, given some project's time organization issues, we did implement the opportunity to estimate such m value globally as equal throughout all blocks or as a specific m value per block. At the time of implementation we already estimated that the overhead was to be heavy when m was pre-estimated on a per-block basis, that is, because its value should be saved at the header of each block, granting 4 new bits per-block of overhead.

TABLE I
SAMPLES USED ON PERFORMED TESTS

ID	Title	Duration	Size
1	Rainbow, *I Surrender*	0:29s	5.2MB
2	LSD + Max Planck, *Receive, Return*	4:05s	43.1MB
3	Samuel Barber, *Adagio for Strings*	8:02s	85.1MB
4	Vangelis, *Spiral*	0:24s	4.2MB

IV. EVALUATION

After the implementation phase we did some tests and tried to retrieve some knowledge from it. The runnable code could be achieved by entering the `src` directory and then execute both CMake and further Makefile, by running the following commands in the shell:

```
$ cmake .
$ make
```

After doing so, in the root of the project a `bin` directory should appear with an executable inside called `losslesscodec`—this is our runnable code. All the examples and results reported hereby comes from the execution of such executable with the following samples, as described in table I.

The choice of samples was made having in mind both scenarios of strong variations (scenario granted with samples 1, 2 and 4), and of low variations (as sample 3). The level of silences is also very high in the third sample, as this is a characteristic of classical music genre. It is then expected to obtain strong compression rates in samples such as the third and lower on more *complex* tracks, such as 2, which is of electronic genre.

TABLE II
FIRST RUN RESULTS—LOSSLESS CODEC

ID	Original Size	Compressed Size
1	5.2MB	4.5MB
2	43.1MB	40MB
3	85.1MB	66.5MB
4	4.2MB	3.5MB

A. Lossless Codec

The first test we made was to run the lossless codec in all our samples, which will generate a `*.wavz` file. To do so, simply run the following, considering that the file `file.wav` is in the same directory:

```
$ ./losslesscodec file.wav
```

The following results were obtained, as described in the table II.

As we can verify from the results above, our lossless compression achieved the expected results, where the file with more repeated sample-shapes had the higher compression. This is due to the amount of redundant data which exists throughout the track (the sample 3). In the other hand, the track with more variations (lower redundancy to remove) had a lower compression rate.

To test out if the lossless compression is, in fact, working as it should, we can decompress the files by running the following code:

```
$ ./losslesscodec -d file.wavz
```

The output of such execution generates a file called `*.recovered.wav`. To verify the authenticity of the lossless property of the codec, we can compare its hashes, by executing the following:

```
$ shasum file.wav file.recovered.wav
```

A proper result should be like the following, being the output of sample 1:

```
$ shasum 1.wav 1.recovered.wav
2029(...)0fad8961  1.wav
2029(...)0fad8961  1.recovered.wav
```

B. Lossless Codec with Global m Golomb Value

By default, the execution of the executable `losslesscodec` has the estimation of the Golomb's m value on a per-block basis. Notwithstanding, we can also test out if there should be a difference on the same execution, but with a global m value. The results were the following, as revealed in table III.

The obtained results were achieved as expected, where the differences in comparison with the same test but with a dynamic m value estimation were small. This happens due to an overhead made by 4-bits of the m value encoding per-block of 512 samples on the file. With the creation of partitions, where one could install several blocks inside (that amount depends on each file to compress) this overhead could be taken down, granting better and more useful results.

TABLE III
SECOND RUN RESULTS—LOSSLESS CODEC WITH GLOBAL m VALUE

ID	Original Size	Per-block m Size	Global m Size
1	5.2MB	4.5MB	4.7MB
2	43.1MB	40MB	41.4MB
3	85.1MB	66.5MB	71.7MB
4	4.2MB	3.5MB	3.7MB

TABLE IV
THIRD RUN RESULTS—LOSSY CODEC

ID	Original	$q = 5$	$q = 7$	$q = 12$	Audible
1	5.2MB	2.7MB	2MB	1.3MB	Y Y N
2	43.1MB	25.5MB	19.3MB	10.9MB	Y Y Y
3	85.1MB	38.5MB	28.5MB	21.4MB	Y Y N
4	4.2MB	2MB	1.5MB	1.1MB	Y Y N

C. Lossy Codec

Finally, the lossy codec could be ran by executing the following, where `FACTOR` is the wanted factor of quantization:

```
$ ./losslesscodec -l FACTOR file.wav
```

The output of this execution is a `*.wavqz` file which includes, encoded, the quantization factor on its header, to further decode without giving the parameter back. In the table IV we can verify the following results, with different q (quantization factor).

It is understandable that the bigger the quantization factor q, the bigger are the losses of data, then the noisier is the sound. In these tests we are applying a dynamic m Golomb value. This way, and with the Golomb coding, we can expect a lot more compression of data, since after cutting some bits of the samples, the amount of redundant data increases.

D. Choice of Predictors

In terms of choice of predictors we can verify, by the figure 4, that both second and third predictors are the most favorite to use. This is due to the variability of the sounds. In fact, the smaller the predictor index is, the minor variations the sound has on its track—this conducts to more redundancy.

V. CONCLUSIONS

The level of compression, although limited in lossless circumstances, proofed to be very powerful with our implementation. Although not achieving the compression rates of codecs such as FLAC's, it was important to us to verify all the possibilities we had implemented and others which could also be developed.

The increment of techniques of compression showed us, also, that with the loss of redundancy we could get ourselves in scenarios of overheading data with the auxiliary information that helps us to get the original file. With such a scenario,

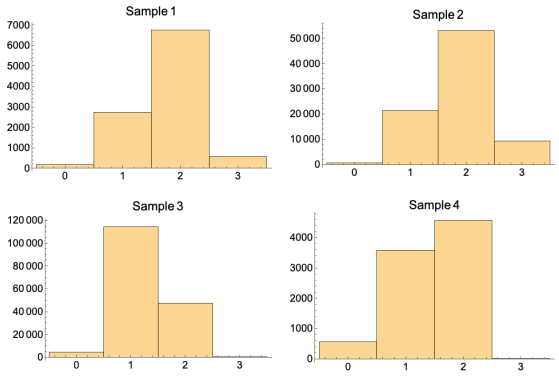

Fig. 4. The choice of predictors

one should be careful of adjusting the number of techniques applied, in order to avoid ending with files way bigger than the original ones.

VI. Group Collaboration Assessment

After some deliberation within the work group we have decided to distribute the working time by each member in the following percentages:

- João Fonseca — 30% of total time;
- Pedro Silva — 15% of total time;
- Rui Lopes — 55% of total time.