Lab work nº 1

João Branquinho
76543
University of Aveiro
joaobranquinho@ua.pt

Luís Silva
76585
University of Aveiro
lfssilva@ua.pt

Tiago Ramalho
76718
University of Aveiro
t.ramalho22@ua.pt

***Abstract*—Nowadays, the amount of data generated is tremendous. Compression Algorithms are ubiquitous and necessary to maintain an acceptable size of data stored in today's databases and servers spread around the world. In this article, we analyze different techniques of compression. We start with stereo-to-mono, breezed trough uniform scalar quantization and ended with vector quantization compression. We were able to compare the different methods with the help of signal-to-noise ratio analysis. Finally, we draw some conclusions about these lossy compression methods. Their impacts on sound quality and the amount of space that they are able to save.**

I. Introduction

Since the beginning of the technological era, limited disk space has always been a problem. One of the earliest challenges was to find the perfect representation form of information. Being able to store the maximum amount of data into the smallest amount of space. There are several other reasons to pursue better forms of data compression beyond disk space savings, for instance, the reduction of transmitting times.

In this study we aimed to explore some implementations and theoretical approaches to this problem for this we tryed algorithms such as uniform scalar quantization and vector quantization. The measures were based in the signal-to-noise ratio analysis.

A. Stereo to Mono

First, we start by analyzing the histogram of the mono stereo. For this exercise we first had to generate the said channel and then create the histogram of values associated with it. A simple solution to this problem is to sum every channel sample at a given time and divide it by the total number of channels.
Represented in Figure 1 is the mono version of the *sample01.wav* file.

B. Signal-to-Noise Ratio

Signal-to-noise ratio (SNR) is a measure used in engineering in order to compare a given signal and its respective noise.
This measure can be very powerful to compare 2 files where one of them is created after a lossy compression over the other.

The SNR is given by the following formula:

$$SNR_{dB} = 10\log_{10}\left(\frac{\sigma^2_{signal}}{\sigma^2_{noise}}\right)$$

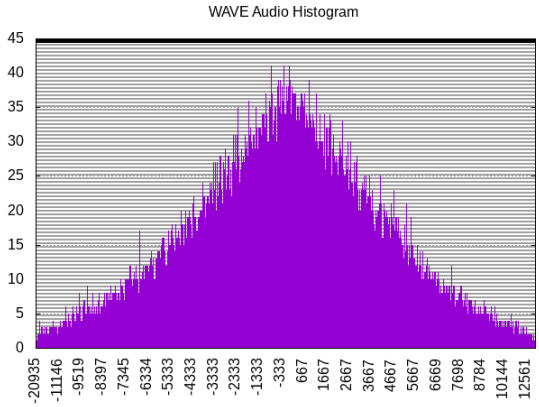

Fig. 1. Mono version of the *sample.wav* file

In our specific case, the objective is to compare 2 audio files. For this to happen both σ^2_{signal} and σ^2_{noise} can be seen as:

$$\sigma^2_{signal} = \sum_{i=0}^{n} (original\_file[i])^2$$

$$\sigma^2_{noise} = \sum_{i=0}^{n} (original\_file[i] - lossy\_file[i])^2$$

where $[i]$ refers to each of the samples of the file.

C. Uniform Scalar Quantization

Uniform Quantization can be seen as a quantization method in which all of the intervals have the same size. One of the challenges presented to the group consisted of the creation of a Uniform Scalar Quantizatior.

1) fstreamBits file:
In compression of data we only want to write the bits who contain information, because of that the group had necessity to create two classes:

READBits: this class have three functions *readHeader*, *readItem* and *readBits*.

WRITEBits this classe have another four functions *writeHeader*, *preWrite*, *writeBits* and *flush*

With this functions the group could manipulate the writing and the reading of binary files to achieve better results in compressing files.

2) Approach by the group:

Our approach the this problem is simple:

Given that each sample of the original audio file is a 16 bit short (in the specific case we worked on) and that we want to ignore n bits in each of the samples, right shift n bits.

The process is simple:

Encode Iterate over the original file right shifting each sample n bits.

Store Store each of the samples on a new file.

Decode Iterate over the stored file left shifting each sample n bits.

The stored file (lossy compressed) is $\frac{sample\ size}{n\ bits}$ times smaller than the original files.

3) Results:

In table I the group exhibits some results of uniform scalar quantization. After table analyzation it is possible to verify that when Compression Rate is lower the SNR is higher. That make sense because when the Compression Rate is bigger the file compressed have less bits with information.

D. Vector Quantization

Vector quantization is a quantization technique that allows signal processing.

This technique gives the possibility to model density functions based on the distribution of prototype vectors. Moreover, specifically on data compression, vector quantization is frequently used in data compression. This works by encoding multidimensional vectors into a finite set of values of a lower dimension.

Our implementation of this algorithm is based on the **Linde–Buzo–Gray algorithm** [1] which is similar to the k-means method.

Our implementation of this algorithm takes in account inputs given by the user: the vector size of each of the codebook entries, the codebook size, the overlap of each of the vectors in the training set and the maximum error associated each iteration ($1 + \epsilon$). In our implementation the codebook size is always a power of 2, this is because of the way we generate new vectors for our codebook. The implementation goes as follows:

Generate the Training Set We iterate over the samples and group them in groups the size of the block_size we want for our codebook. The overlap dictates the skips we make in our iteration.

Calculate the initial Vector With our training set we calculate the medium vector, this will be our starting point for the generation of the codebook.

Split the Codebook With our starting vector we generate two new ones that are separated by some distance.

Convergence of the Codebook With our new codebook we try to reach a convergence this is done by moving the vectors step by step to the center of the points in their proximity. For this, we calculate the nearest vector for each group in our dataset. And then update the codebook with this new values calculating the new center of this group of samples that share the same vector.

Split the codebbok again After the codebook convergence we split the codebbok again and so on until we have the size requested.

1) Results:

Our implementation of the **Linde–Buzo–Gray algorithm** was based on a python implementation of this algorithm [2]. We had a great deal of fun. We made a great deal of experimentation.

We started by analyzing the impact of the overlap on the dataset and we noticed no significant differences. Something interesting that we noticed, with the samples provided by the teacher, was that a codebook generated for one of them was decent to use to compress and decompress the other samples which we found very interesting since it is possible to generate a codebook generalized. Another interesting phenomenon we noticed was that with some samples our implementation did not work very well, especially when the music was composed of very loud sounds. We conclude that this was due to the way we initialized the first vector. As this vector is generated from the average of all values this would have an influence on the generated codebook as the average is sensitive to extremes.

II. Conclusion

In this study, we took a closer look at some lossy compression methods. These techniques are based on the assumption that some original data will be lost. However, the results obtained are impressive. Even with basic coding techniques such as uniform scalar quantization, it is possible to reduce the size of the file by half without noticing a quality loss in the file after decompressing. The implementation of this method is very straightforward and easy to implement. And it makes for a very convenient solution for simple compression necessities.

As for the other technique implemented it was a bit short of what we expected. Not that it is a bad technique, we believe that it was our implementation that failed. After analyzing the results, the values obtained are not as good as they could be. Although the codebook generation is fast compared to other implementations, the final results something to be desired in terms of sound quality produced after file decompression. But here, the savings in space are even more drastic.

With this, we conclude that the careful analysis of the type of data that we want to compress is indispensable. Sometimes what's good for one file is not the best for another.

Acknowledgment

The authors would like to thank to their families for all the love and time wasted on raising them, to the teachers Armando Pinho and António Neves for their availability to answer questions.

A big thank you to all these people.

References

[1] Linde, Y.; Buzo, A.; Gray, R. (1980). *An Algorithm for Vector Quantizer Design.* IEEE Transactions on Communications. 28: 84.

[2] Markus Konrad, py-lbg, (2015), GitHub repository, https://github.com/internaut/py-lbg

TABLE I

Audio Sample	Nº bits	Encode Time	Decode Time	SNR channel 0	SNR channel 1	Compression Ratio
sample.wav	8	0.020 s	0.228 s	34.96 db	34.99 db	2:1
sample.wav	10	0.025 s	0.266 s	46.98 db	47.05 db	8:5
sample.wav	14	0.056 s	0.270 s	70.55 db	70.64 db	8:7
sample02.wav	10	0.073 s	0.663 s	43.85 db	41.68 db	8:5

TABLE II

Audio Sample	Block Size	Overlap Size	Codebook Size	Time Generating Codebook	SNR channel 0	SNR channel 1
sample.wav	4	3	1024	18.73 s	10.47 db	9.90 db
sample.wav	2	1	1024	36.50 s	20.31 db	19.18 db
sample.wav	2	1	512	29.13 s	21.62 db	20.05 db
sample02.wav	2	1	512	4 min 41 s	19.81 db	27.37 db
sample02.wav	4	3	1024	3 min 16 s	12.58 db	18.80 db