An audio lossy codec with Vector Quantization (VQ)

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*—Nowadays most of the audio service providers such as Apple Music, Pandora Radio or Spotify distribute their content seamlessly throughout the Internet. This happens due to the power of compression, where a large array of versions of each user requested music track is compressed at a different sample bit length, most of them being made lossy codecs. As lossy codecs are made to exclude data from an audio file which could be identified as redundant, this work shows a construction of a lossy audio codec, based on Vector Quantization (VQ).**

***Index Terms*—audio codec, compression, lossy compression, vector quantization , VQ**

I. Introduction

In most recent years, the amount of music applications used by people in general has been increasing exponentially, due to both lower costs in accessing the contents, and the low delays on the service capabilities of its distribution.

This distribution of audio content can be attained in such a small delay transport since its transmission counts with the help of an advance made back in the history of computing and of information: the compression. In fact, most of today's audio services like Apple Music, Pandora or Spotify use compression to reduce the amount of redundant and dispensable data in audio tracks. The user, unable to distinguish the differences, receives a smaller version of the original record, easing its transmission throughout the networks.

This document relates to the design and the implementation of an audio codec classified as *lossy*, that is, which dispensable and redundant data is lost by the means of compression, and which compressed format is not audible. Some tools were also made in order to evaluate the development of the audio codec, also hereby described.

II. System Architecture

This audio codec is designed to perform compression throughout a mechanism of quantization. In fact, as soon as the input is given to our system, a first module would apply *uniform quantization* to it. This way, given an audio as an input (audio file in waveform audio file format (WAV)) and in 16-bits pulse-code modulation (PCM) format), it is expected this first module tries to remove some of the least significant bits of the samples representation—with this method some of the audio contents are replaced with noise artifacts, nonetheless the media remains audible and understandable to most people.

After this first modification, which would have as an output a `WAV` file with reduced samples (in content, not in length, since we have to restore the original 16-bit samples in order to reproduce the audio on a conventional player), each of these must be given into a module which would reproduce a *codebook*, that is, a set of codewords that allows this module to replace a uniformly quantized sample into the representation of the closest vector within itself. Basically, this module is a bi-dimensional vector with c codewords of b bits, made having in mind the audio object to compress. The representation which will then replace the sample on the compressed file, is the index of the codeword vector which was evaluated to be the closest one in comparison with the given sample.

These described modules would be applied by an entry-point which would also have an encoder and decoder components. The *encoder* would have the function to apply the uniform quantization and codebook to an audio file, generating itself an output being a set of codeword indexes, written with the minimum bits needed to represent them. In the other hand, a *decoder* would have the task of reading those bits and, facing them to the codebook (which should also be given as input), replace the codeword indexes by themselves, obtaining, at the end, an audio file closest to the original.

In figure 1 is depicted the described system architecture.

III. Implementation

In this section we deeply describe the implementation of our lossy audio codec.

A. Uniform Quantization

The first component that we implemented was the `wavquant` module, since is the first to modify the input audio file given to our system.

This module receives a WAV audio file and an integer quantization factor, θ. Reading each sample s from each frame of the audio file, a new audio file will be created after applying a logical shift operation to each s, by a factor of θ, removing the θ least significant bits. Basically, considering that we have an 8-bits sample 11011010, by applying a factor θ of 3 we would get $11011010 \text{ srl } 3 = 11011$.

As we want that this module to generate an audio file as output, it is important to have the new samples similar in length to the originals. That is, before saving our changes in

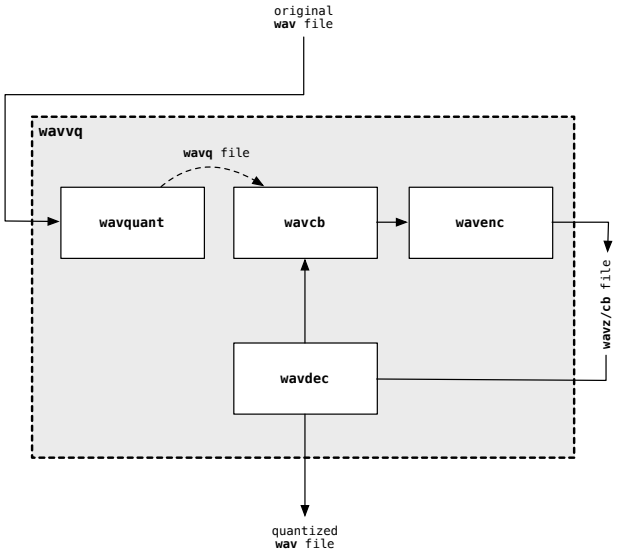

Fig. 1. Lossy audio codec architecture

a new WAV file, it is important to apply the opposite logical shift operation, to revert the length changes and to get the original sample length. Then, this time, we will have a new sample s' which will be the first operation applied with the same operation, in reverse, resulting in the following: $s' = (s \text{ srl } \theta) \text{ sll } \theta$.

The output file is then a WAV audio file which we called, for simplification purposes, a `wavq` file and, by this execution, it will still have the same size as the original `wav` file, since, in length, nothing has changed—only the contents were reduced.

B. Codebook

The codebook is, by far, the most complex entity created in this project. In fact, this entity must be able to receive instructions on how to construct a codebook and then apply an algorithm to generate the proper number of iterations to enhance its utility.

A general description of the implementation of the codebook (as is in `wavcb` module) is described as follows in algorithm, where we try to construct a codebook C with c codewords of b bits, with an overlap factor α and a maximum wanted distortion between iterations ε:

Require: $c > 0$ **and** $b > 0$ **and** $\alpha > 0$ **and** ε
1: GENERATECENTROIDS()
2: $iteration \leftarrow 0$ {the current iteration}
3: $lastDist \leftarrow \infty$ {the distortion between iterations}
4: **while** ¬HASCONVERGED() **or** ¬ISMAXITER() **do**
5: TESTSAMPLESONCENTROIDS()
6: $dist \leftarrow$ EVALUATEDISTORTION()
7: **if** $lastDist - dist < \varepsilon$ **then**
8: $C \leftarrow centroids$
9: MARKHASCONVERGEDAS(**true**)
10: **end if**
11: $lastDist \leftarrow dist$
12: EVALUATENEWCENTROIDS()
13: $iteration \leftarrow iteration + 1$
14: **end while**

Looking to the described algorithm we can verify that the first task we do is a method named GENERATECENTROIDS(). This method is our codebook initialization, which will use the audio file loaded in time of the construction of an object CODEBOOK, where the algorithm above forms a CREATECODEBOOK() method. This generation is made according to the algorithm that follows, where $cent$ is the vector of centroids:

Require: s as samples **and** $c > 0$ **and** $b > 0$ **and** $\alpha > 0$
1: **for** $i = 0$ **to** SIZE(s)$\times b - b$ **do**
2: $cent$.ADD($s(i/16)$ sll (i mod b) **or** ADD(($s(i/16)$+1) srl (b-(i mod b))
3: $cent \leftarrow$ DELETEDUPLICATES($cent$)
4: **if** SIZE($cent$) $< c$ **then**
5: **return** error
6: **end if**
7: $cent \leftarrow$ CHOOSECVECTORS($cent$, c)
8: **end for**

After executing the GENERATECENTROIDS() method we have a vector of c b-bit bit sequences which forms our initial set of centroids to test out our training set.

Now, by considering our samples s as our training set, we try to apply the Linde-Buzo-Gray (LBG) algorithm (also known as generalized Lloyd's algorithm and similar to data clustering's K-means technique), we can, iteration-by-iteration, obtain a good codebook, that is, such a structure that would give us the best set of representations to our audio samples, with which, after the process of decoding, the audio would be the closest possible against the original version.

This algorithm is then expressed on the execution of the TESTSAMPLESONCENTROIDS() method, shown as follows:

Require: s as audio samples **and** $cent$ as centroids
1: **for all** $sample$ in s **do**
2: $\delta = \infty$ {distance}
3: $\delta^\star = \infty$ {temporary distance}
4: $\omega \leftarrow 0$ {minimum distance centroid index}
5: **for** $i = 0$ **to** $cent$.SIZE() **do**
6: $\delta^\star \leftarrow$ DISTANCE($sample$,$cent(i)$)
7: **if** $\delta^\star < \delta$ **then**
8: $\delta = \delta^\star$
9: $\omega = i$
10: **end if**
11: **end for**
12: $samples_{it}\,[\omega]$ .ADD($(sample, \delta)$)
13: **end for**

In the algorithm above there are two items which deserve a little more explanation. Starting by the method DISTANCE(), this is a method to estimate the euclidean distance between points of dimension b. The other item which requires some attention is the structure where we are saving all our results from the LBG method's execution: the $samples_{it}$ vector. This structure is a vector with the size of the $cent$ vector, in which each index has its own vector of tuples, each representing the

tested sample and its estimated distance to the corresponding and chosen centroid.

Right after the audio sample training set testing on the centroids which were created we should analyze the quality of the iteration results. To do so, one should estimate the distortion of the current iteration. This is the meaning of the EVALUATEDISTORTION() method, described below:

Require: s_{it} as the iteration result **and** n samples count
1: $sum \leftarrow 0$
2: **for all** $centroid$ in s_{it} **do**
3: **for all** $sample$ in $centroid$ **do**
4: $sum \leftarrow sum + sample(1)$
5: **end for**
6: **end for**
7: **return** sum/n

Having estimated the distortion, then we only need to calculate the new set of centroids, in order to repeat our algorithm and make a step towards the convergence of the codebook, as we want to. Then, we execute the method EVALUATENEWCENTROIDS(), detailed in the algorithm below:

Require: s_{it} as the iteration result
1: **for all** $centroid$ in s_{it} **do**
2: $point \leftarrow$GETAVERAGECENTROID($centroid$)
3: $centroids.[centroid] = point$
4: **end for**

Our codebook implementation also relies on a set of other functions such as CODEBOOK.SAVE() and CODEBOOK.LOAD(). These functions save and load a `*.cb` file which contains, in binary, the contents of the codebook—the codewords, in a contiguous sequence.

C. Encoder and Decoder

As soon as the codebook is created by the `wavcb` module, the task of giving a sample to test on it is a responsibility of an encoder. Then to reverse the process one should use a decoder.

In our implementation, the encoder is described on the `wavenc` module. This class ENCODER receives, in order to construct such an object, a codebook, an audio file to compress and a file name to give as an output. This output, then, will be a compressed file with unreadable audio, by a conventional media player, since it will be made with the indexes of the codewords which the codebook gave away, after testing each audio sample with a closest vector within itself.

More than the compressed audio samples themselves, the encoder also has the duty of filling a header with the specifications of such audio file. Without it, the decoder would be blind and could not fulfill to create a new audio file without information such as the format, the sample rate or the number of channels, in a generalized solution.

The decoder, then, specified by the class DECODER present on the module `wavdec`, allows us to construct such an object via a file name of a compressed audio file and the respective codebook file.

As the decoder receives a compressed audio file name, it tries to open the file and reads its header, saving useful information about how does the final audio file will look like, as its format, its number of channels and its sample rate. Right after such action, the decoder will start the process of giving its records to the codebook, testing them to get the respective vectors, which will conclude on the production of a WAV audio file, decompressed.

D. Developed tools for evaluation

The system's output WAV file, in comparison to the first one (the original one) should have its content audible but with less sound quality, that is, with a certain amount of noise added to it. This is justifiable due to the quantization processes made in both uniform and vectorized steps. To proof this, we created two tools which allows us to evaluate the compression made.

The first tool, an *WAV histogram viewer*, shows us the number of times each sample appear in each audio channel, of a given audio file. More, it can also give the histogram of a mono channel made out of the average samples between the two available channels in such WAV audio files.

The second tool, a *Signal-to-Noise ratio* estimator, was created to compare two WAV audio files—the first being the natural signal (in other words, the original audio file, which we will use to compare to) and the second being the noisy signal (or the compressed audio file).

Signal-to-Noise ratio means that we want to get a ratio of a desired signal to the level of background noise, in decibels (dBs). To estimate this, SNR_{dB}, we must calculate

$$\text{SNR}_{\text{dB}} = 10 \log_{10} \frac{P_{\text{signal}}}{P_{\text{noise}}}$$

, where P_{signal} is the power of the [original] signal and P_{noise} is the power of the background noise.

Given such information, comparing two equal files must be theoretically infinity, what gives us the conclusion that the bigger the value, the better the compressed audio file is.

In order to calculate the power of a signal we applied the following algorithm, where S is a vector of the original samples and M is the vector of the compressed audio samples, after decoding:

Require: S as original samples **and** M as modified samples
1: $\text{sum}_{\text{original}} \leftarrow 0$
2: **for all** value in S **do**
3: $\text{sum}_{\text{original}} \leftarrow \text{sum}_{\text{original}} + \text{value} \times \text{value}$
4: **end for**
5: $\text{sum}_{\text{modified}} \leftarrow 0$
6: **for** $i = 0$ **to** S.SIZE() **do**
7: $\text{sum}_{\text{modified}} \leftarrow \text{sum}_{\text{modified}} + (S[i] - M[i])^2$
8: **end for**
9: **return** $10 \times \log_{10} (\text{sum}_{\text{original}}/\text{sum}_{\text{modified}})$

IV. EVALUATION

As our implementations produced some results, in this section we establish an analysis of them.

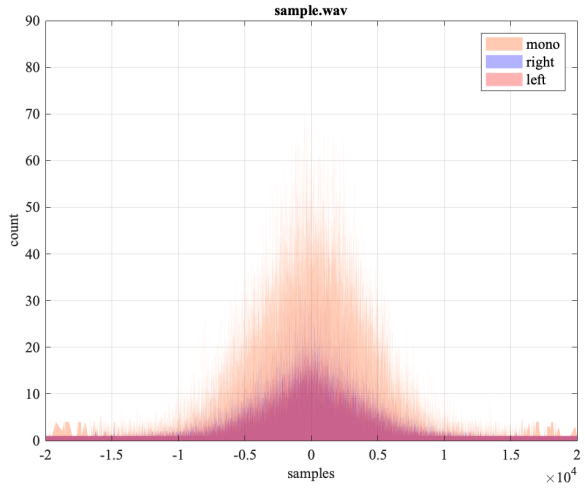

Fig. 2. Left, right and mono channels histogram (`sample.wav`, 16-bit audio)

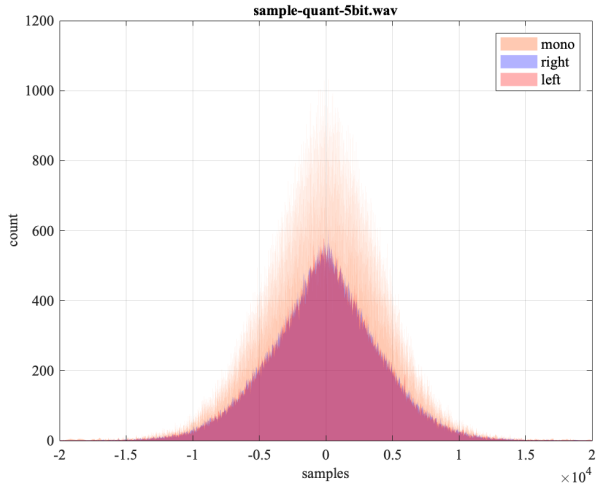

Fig. 3. Left, right and mono channels histogram (`sample.wav`, 16-bit audio quantized in 5 bits)

A. *Evaluation tools*

Our developed tools to evaluate the lossy audio codec provided meaningful results. Starting by the WAV audio histogram plotter, this tool is able to provide us plots on the count of samples per channel on an audio sample set and the average channel histogram as equal.

In the figure 2 we can verify the histogram of left, right and mono channels of the sample audio file given by the class' teachers—file `sample.wav`.

As we can analyze from the figure 2 while both left and right channel's samples are quite similar in terms of number of appearances on the audio file, when we produce the average channel, more samples appear equal, since the probability of being so is almost the double than before.

In terms of our other tool—the Signal-to-Noise ratio estimator—this also gives useful results, as expected by its theoretical development. When we presented its implementation, we stated that if two equal audio files are submitted into such a comparison, then a value of infinity should arise. In fact, in our implementation, when we do so, the IEEE 754 value of `inf` is printed as output.

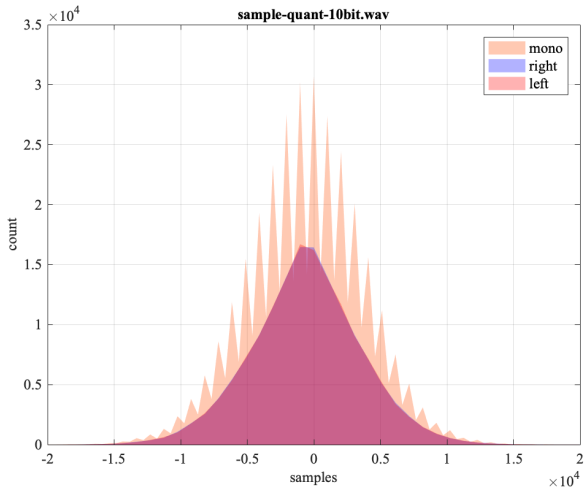

Fig. 4. Left, right and mono channels histogram (`sample.wav`, 16-bit audio quantized in 10 bits)

B. *Uniform quantization*

If we apply uniform quantization to our `sample.wav` file, then we should verify that both left and right channels appear more often with the same samples, as we just decreased the size of the possible combinations of samples in this audio. In figure 3 such a scenario is possible to be proofed, where we produced a quantization of 5 bits.

As we obtained, by hearing, the maximum quantization in which this file is submitted, maintaining useful information for us humans, is of 10 bits. In figure 4 is depicted such a set of histograms.

In the figure 4, in fact, is very clear to see that as we just decreased our range of representation of each sample to just 6 bits, our average channel starts to show its discrete values easily—the values have always been discrete, but the histogram resolution did not allows us to see it.

If we increase even more, the quantization (to 14 bits, i.e.) we can see that the histogram is heavily saturated, with which we can, by itself, guess that the audio would be completely inaudible (figure 5).

In terms of Signal-to-Noise ratio, the following results were obtained, as we can see in table I.

As we can see by the results, it is proved that the closer the SNR is to infinity, the better the sound quality is, since the quantization factor is smaller. We can also notice that if the quantization is maximum, then the value will be closer to 0 dB.

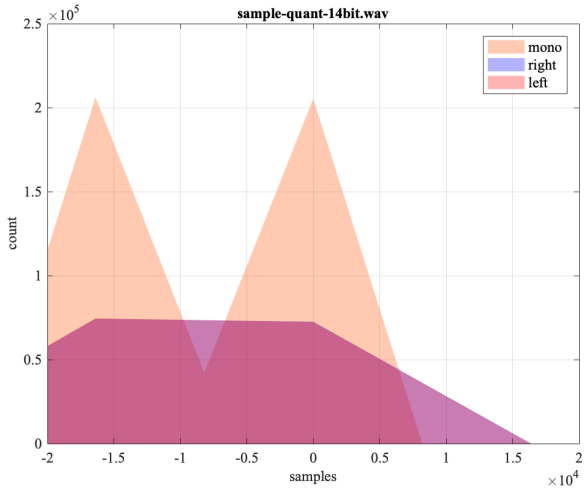

Fig. 5. Left, right and mono channels histogram (`sample.wav`, 16-bit audio quantized in 14 bits)

TABLE I
SIGNAL-TO-NOISE RATIO BY QUANTIZATION FACTOR

Quantization	SNR
0	∞ dB
2	66.8418 dB
4	53.3973 dB
6	41.0444 dB
8	28.9237 dB
10	16.8549 dB
12	4.77507 dB
14	-7.70631 dB
15	-14.3015 dB
16	0 dB

C. Vector quantization

Unfortunately, we could not produce any result with vector quantization at the time of this work delivery. This happened because of the implementation of the codebook, which did not let us test its unit in such a short time after we have completed it.

V. Conclusions

This work not only has showed us the effects of lossy compression on an audio file, but also gave us the comprehension we needed in terms of analysis of its histograms to better understand the effects on the quantity range of representation of each sample present on an audio file.

While we have learned a large number of concepts, misfortunately we could not attend the delivery deadline with a complete work, letting such a space for fixing the codebook issues and to implement some other features.

If this work had already been done, one could study some different strategies of implementing codebooks, with different algorithms, basing its initialization on different features, such as the average sample of a given file to compress or other. Moreover, one could also juxtapose both encoded file and codebook and create one media player which would be able to read the media.

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 — 10% of total time;
- Pedro Silva — 10% of total time;
- Rui Lopes — 80% of total time.