Published by :
http://www.ijert.org

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 14 Issue 11, November 2025

Using A Simple Audio Compression as a Learning Tool for CS Students

Patrick McDowell, Kuo-pao Yang, Mark W. Bland
Computer Science Department
Southeastern Louisiana University
Hammond, LA 70402, USA

***Abstract* - In this paper we review a simple FFT-based audio compression algorithm that can be easily adjusted to have better fidelity, or better compression with a loss of fidelity. The proposed algorithm was developed as a teaching tool to facilitate the connection between mathematics, science, and computer science. The algorithm is a lossy compression and was implemented in python.**

Keywords - Fast Fourier Transform (FFT); Audio Compression Algorithm; Python

I. INTRODUCTION

Computational Thinking (CT) was first conceptualized by Papert in 1980 [1] and has since been identified as an important skill for everyone, not just computer scientists [2]. Despite recent attention [3, 4] to the importance of including CT in curriculums as a way to foster the development of problem-solving skills, anecdotal observations support the conclusion that current students often are unable to employ the resulting transfer of training that one might expect. In this paper we discuss the development of an audio compression algorithm that has been used to teach principles of compression and suggest its use as a novel way to bridge the gap between signal processing and computer science in Machine Learning and Pattern Analysis classes for upper level and graduate students. This algorithm can be used with other basic compression techniques such as run-line compression [5] and vector-based compressions [6] to provide students with insight into the interactions of various disciplines including signal processing, computer science, and statistics. We find this topic to be important because of the significant roles it plays in audio and video streaming and still-image compression.

Typical computer science curriculums include significant content in mathematics and science, but unlike fields such as electrical and mechanical engineering, the connection between the supporting mathematics and science courses, and the major computer science courses is not as direct. One of the goals of teaching these topics, and this algorithm was to provide the students with a practical application that teaches and illustrates the intertwined nature of these areas of study.

Our algorithm is a “lossy” compression and is similar to mp3 [7] audio compressions in that it relies heavily on the Fast Fourier Transform (FFT) [8]. One interesting attribute of this algorithm is that it is tunable, meaning that the user can specify how much of the “energy” of the original signal should be kept. When keeping a large percentage of said energy, the compression ratio is in the 7 to 10 range, and the audio clarity is rich. However, when the energy parameter is set lower, the compression ratio increases dramatically with a corresponding drop in the clarity. Interestingly, the audio is still fairly clean at a ratio of 20:1, though listening closely reveals acoustic degradation, notably in some of the higher frequency ranges. At a compression ratio of approximately 35:1, the audio quality becomes similar to an AM radio during stormy weather, in that the artifacts are discernable while the audio is easily recognizable. At a ratio of 65:1, the artifacts are more noticeable while the audio is still easily recognizable, and at a ratio of approximately 133:1, the song is recognizable, but at this point there are dropouts during playback.

The algorithm was implemented in python and tested on an MP3 file that had a broad frequency range and a reasonable dynamic range (most of the testing was done using Wild Cherry’s Play that Funky Music) [9]. Note that in Python, the MP3 file is converted to a complete lossless 2 channel audio signal, and that is the starting place for this compression algorithm. That being said, in this paper we will provide a short background, a discussion of the algorithm including a detailed outline using flow diagrams, and some concluding remarks with suggestions for improvements.

II. BACKGROUND AND MOTIVATION

A popular assignment in machine learning/neural network courses is to compress an image using a Kohonen [10] neural network. This assignment is a fun and easy way to learn about vectorized compressions and Kohonen networks. The assignment can be modified by replacing the neural network with a statistical/histogram approach with gives similar results. While the histogram approach causes the codebook to lose its topological meaning; the codebook entries are no longer arranged in a manner in which similar entries are neighbors, it does have benefits of speed and demonstrates an interesting statistical approach.

Vectorized image compression is typically performed using a 3-pixel by 3-pixel vector (such as a dither pattern). At this vector size a 9 to 1 compression of the image is realized if the codebook is not included. Here, the codebook is the collection of ideal vector patterns used to compress the image. Codebooks can be one dimensional or two dimensional. Typical sizes for one dimensional code books are 128 to 1024 vectors; larger sizes usually aid in more faithful image reproductions. The function of the Kohonen network is to find the collection of vectors that can best duplicate the image. Larger vector sizes result in higher compression ratios; however, image artifacts are more visible with larger vector sizes.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 14 Issue 11, November 2025

Run-line schemes are lossless, such that when the image is decompressed, it is identical to the original. Run-line schemes work very well with images composed primarily of single colors. However, if the image is a mix of colors (i.e. consistent blue sky vs hazy, cloudy sky) the efficiency of run-line schemes breaks down. Where the run-line scheme performs poorly with "mixes" of colors, a vectorized scheme performs well because the various tone and hues can be captured in the vectors. Along with the aforementioned tones and hues, edges and other shapes comprising images can be present in a well-developed codebook.

One way to enhance the vectorized compression is to couple it with a run-line scheme. In this way large areas of non-primary colors can be compressed by specifying the number of instances a codebook entry is to be placed in a row in the decompression routine. Thus, instead of replicating a single pixel several times, a codebook entry is replicated that number of times.

How do these ideas work with audio data? As it turns out, the vector approach works and is a very interesting exercise. With that said, the resulting audio can be "muddy"; like listening to a well-used 8-track tape. While the audio is easily recognizable, the quality of the reproduction is not ideal.

MP3 compressions are extremely effective, to the point where many people cannot tell the difference between them and a non-compressed recording. A typical MP3 compression ratio [11] is about 10:1, depending on the bit rate/sampling rate of the recording. The MP3 algorithm uses a Psychoacoustic Model which identifies what sounds people are less likely to perceive. These are the sounds that are targeted for removal from the compressed recording. For reference, typical human hearing range is from 20hz to 20khz, with the most sensitive range being from 2khz to 5khz [12]. A voice line [13] (telephone) has a frequency range from 300hz to about 3.4khz. Regarding the frequency ranges of musical instruments [14], most do not emit sounds below about 35hz or above 17khz. By analyzing what can be heard, the frequency ranges of the sounds of interest, and other factors, scientists can identify what frequencies to keep, and which to discard when designing a compression algorithm.

The MP3 algorithm is extremely effective in both its compression ratio and its faithfulness to the original recording, as perceived by listeners, and it has a significant amount of scientific work behind it. However well it works, teaching the basics of compression can be overwhelming for students.

The motivation behind this work is to demonstrate that while vectorized compression can be used for audio, the basic algorithm is not ideal. However, a basic algorithm with some similarities to the MP3 algorithm can be useful for creating some impressive compression ratios with improved playback performance.

III. APPROACH

Our approach is based on the energy content of the signal: if the user specifies that they want to retain 80% of the energy, the lowest 20% of the signal's energy is discarded. However, the energy content is arranged by frequency. Thus, an FFT is taken of the signal, the bins are sorted by strength from highest to lowest, and in the case of the above example, the bins that hold 80% of the signal energy are kept and remaining low energy bins are discarded. Upon playback, the signal is then reconstructed from high energy bins.

The idea here is that when listening to an audio signal, there are several low energy frequency bands that are stored that the listener may not be able to detect. These undetectable bands still take up storage space in the recording, but if you can't hear them, why keep them? The other thing of note is that unwanted sounds like fuzz and hiss (undesirable background noises) many times have lower energy than the audio, and those noises will be filtered out.

IV. SIMPLE LOSSY COMPRESSION ALGORITHM

The algorithm works by repeatedly collecting "packets" of 2 channel audio data. The data for each stream is comprised of sound intensity values sampled at a typical rate of 44,100 samples per second. The packet size is an FFT friendly size of 4096.

Once a packet is retrieved, the FFT is calculated. As an aside, the FFT is a reversible process that transforms the input signal from the time/intensity domain to the frequency/intensity domain. The FFT data (frequency/intensity pairs) is then sorted to from the order of high to low based on intensity, that is the frequency bins with highest intensities are placed at the beginning of the sorted data.

Now that the data is in this form, the sum of the intensities of all the bins is calculated so that based on the "keep this much signal energy" parameter can be applied. Next, starting at the first bin in the sorted bins, the intensities are summed in order (high to low) until the sum reaches the specified energy threshold.

Fig. 1 below shows the cumulative FFT data of channels 1, 2 (top 2 subplots) vs the thinned-out data (bottom 2 subplots).

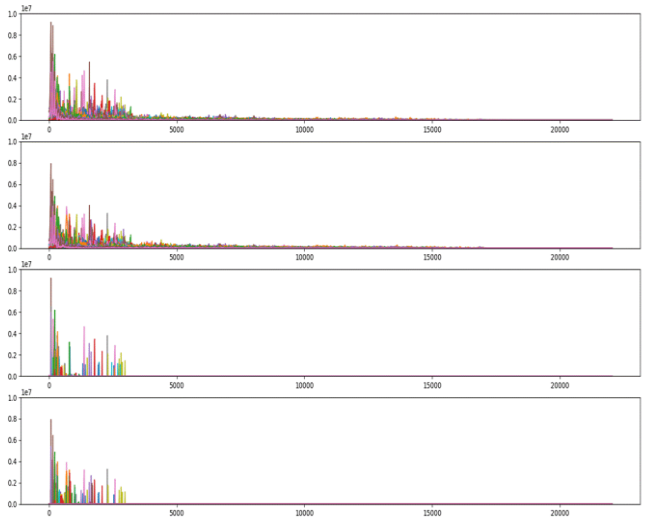

Fig. 1. Figure 1 illustrates the total energy, displayed in the frequency/intensity range of the first 10 seconds of a song. The top two subplots show the total energy, the lower two show about 1 thousandth of the energy. Note the overall shape of the plots is the same, but the lower 2 are very sparse.

This data is for 10 seconds of the test song, done with an energy parameter of 5%. The resulting compression was about 0.000935 of the size of the original; 410 bins out of 438272

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 14 Issue 11, November 2025

total bins were kept. As a side note, this was done to show how the data will still hold the shape of the original (imagine an envelope going around the FFT results) but be very sparse. Interestingly, the audio is still somewhat recognizable, even though it has only about one thousandth of the original signal.

Once this sorting process is complete, the high energy subset of the sorted frequency intensity pairs can be written to storage. The algorithm is illustrated in Fig. 2 below.

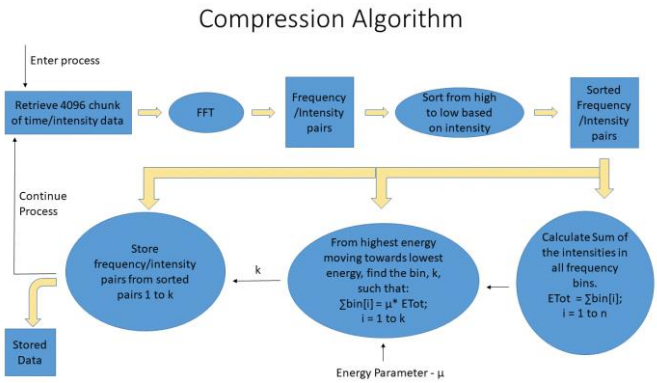

Fig. 2. Figure 2 illustrates the compression algorithm. The key part of the algorithm is sorting the frequency/intensity data to facilitate keeping the strongest components of the signal.

This process is done for each channel, for the entire song. To play the audio of the compressed data, each group of frequencies is read, a complex array of 0 value bins is created, and the intensity data that is read in is loaded into the correct frequency locations. Once this is done, the inverse FFT is taken, recreating the time/intensity signal, and the sound is ready to be listened to. Fig. 3 below illustrates the decompression algorithm.

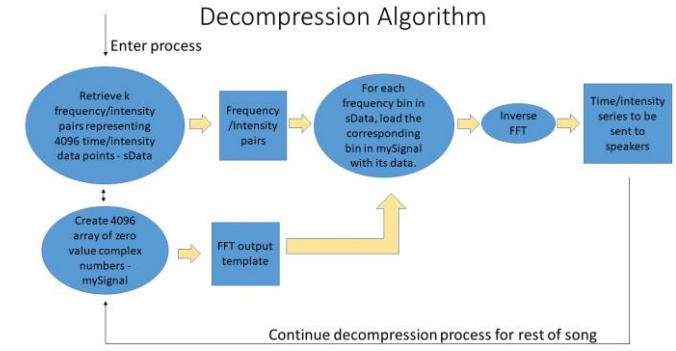

Fig. 3. Figure 3 illustrates the decompression process. An FFT template of zero value complex numbers is populated with the saved frequency/intensity pairs from the decompression algorithm. The resulting data set is then sent to the inverse FFT routine which generates a time/intensity signal which can then be listened to.

V. RESULTS, CONCLUSIONS, AND FUTURE WORK

The algorithm effectively compresses the audio data and because it is so easily adjustable students can explore the relationships between degree of compression and fidelity. More importantly, this algorithm is an easy way to provide hands on experience for students with signal processing topics, such as practical uses of the FFT.

There are many improvements that can be made, including:

- Using a better/consistent measure of the degree of compression. The “energy” parameter is intuitive, but it does not lead to a constant number of bins kept for each sample of the signal.
- The reproduced signals fidelity could be increased, perhaps without decreasing the amount of compression, by dividing the signal into separate frequency ranges, like bass, treble and midrange, and compressing them separately.

There are other interesting things that can be done here, that probably would not help the fidelity of the signal and would likely be computationally expensive. Namely vectorization of the signal, using the FFT. This process would likely result in a “wonky” reproduction of the song, which may not be worthwhile as an accurate compression, but may lend itself as a sound effect.

REFERENCES

[1] M. Lodi and S. Martini, “Computational Thinking, Between Papert and Wing,” Science & Education, 30(4): 883–908, https://doi.org/10.1007/s11191-021-00202-5, April 2021.

[2] J. M. Wing, “Computational Thinking,” Communications of the ACM, 49(3): 33-35, 2006.

[3] S. I. Swaid, “Bringing Computational Thinking to STEM Education,” Procedia Manufacturing, vol. 3: 3657–3662. https://doi.org/10.1016/j.promfg.2015.07.761, 2015.

[4] T. C. Hsu, S. C. Chang, and Y. T. Hung, “How to learn and how to teach computational thinking: Suggestions based on a review of the literature,” Computers and Education, vol. 126, 296–310. https://doi.org/10.1016/j.compedu.2018.07.004, November 2018.

[5] Run-length encoding - https://api.video/what-is/run-length-encoding.

[6] Vector based compression - https://docs.weaviate.io/weaviate/concepts/vector-quantization.

[7] MP3 compression - https://en.wikipedia.org/wiki/MP3.

[8] Fourier Transform - https://en.wikipedia.org/wiki/Fourier_transform.

[9] Wild Cherry – Play that Funky Music - https://en.wikipedia.org/wiki/Play_That_Funky_Music.

[10] C. Amerijckx, M. Verleysen, P. Thissen, and J. D. Legat, “Image compression by self-organized Kohonen map,” IEEE Transactions on Neural Networks, 9(3): 503 – 507, May 1998.

[11] MP3 compression ratio - https://www.sciencebuddies.org/science-fair-projects/project-ideas/Music_p025/music/mp3-how-much-compression-is-too-much.

[12] Hearing range - https://en.wikipedia.org/wiki/Hearing_range.

[13] Voice line bandwidth - https://physics.stackexchange.com/questions/620272/why-are-traditional-telephone-lines-limited-to-about-3-4-khz-bandwidth.

[14] Musical instrument frequency range - https://www.sweetwater.com/insync/music-instrument-frequency-cheatsheet.