Advanced Audio Coding Project: Final Report

Audrey Lee, Oliver Johnson

Introduction:

Advanced Audio Coding (AAC) is an audio coding standard for lossy audio compression that is used for storage and transmission of digital audio. It exploits the limitations of human hearing to reduce information in parts of the signal that are less perceivable. This compression model is the successor to the MP3 format, and has especially better audio quality in the 32-80 bitrate range. It is used as the default media format by YouTube, iPhone, iPod, iPad, Apple iTunes, and several other platforms. For this project, we implemented a version of AAC in Python that retains many of the key attributes, but is also simple enough to understand. We also prototyped different parameters and blocks to experiment how they affect the compression and audio quality.

The specifications [1] and tutorials [2] for AAC compression show in detail the design of the AAC codec. An example of a simplified AAC encoder is given by Chang and You [3], where they propose using only long windows for AAC encoding. This is in contrast to the full version of AAC, which automatically switches between 4 different window types depending on the input signal. The psychoacoustic model we implement is based on the standard 3GPP TS 26.403 V17.0.0 [5].

Technical Detail & Implementation:

The full AAC compression algorithm consists of many blocks, but we decided to focus on the essential ones – the psychoacoustic model, block switching / filterbank, scaling, quantization, and Huffman coding – for our implementation as shown in the block diagram in Figure 1. For each of these steps, we allocated a predetermined number of bits to encode and decode in our model in our bitstream formatter which takes in our audio data and outputs our compressed data.

The code for our implementation of the AAC model can be found here, and we have linked the individual blocks with their respective files throughout this section. We have a README file in our repository that explains how to run our model, how to test our model, and how to install any dependencies. We have also slightly modified the main directory's README file of this repository by linking our AAC model's folder.

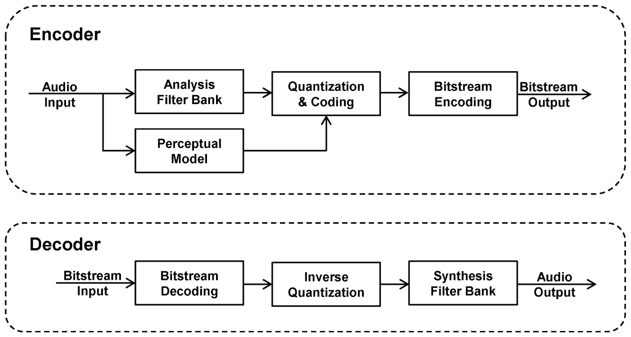


Figure 1: The block diagram of our AAC model [5]

Block switching / filterbank takes blocks of the signal and convolves them with the sine or Kaiser-Bessel Derived window function, performs the MDCT (which ensures that there is time-domain aliasing cancellation), and then uses sets of bandpass filters to get the important spectral components of the signal. The MDCT overlaps the second half of the previous block's signal with the first half of the current one (as shown in Figure 3).

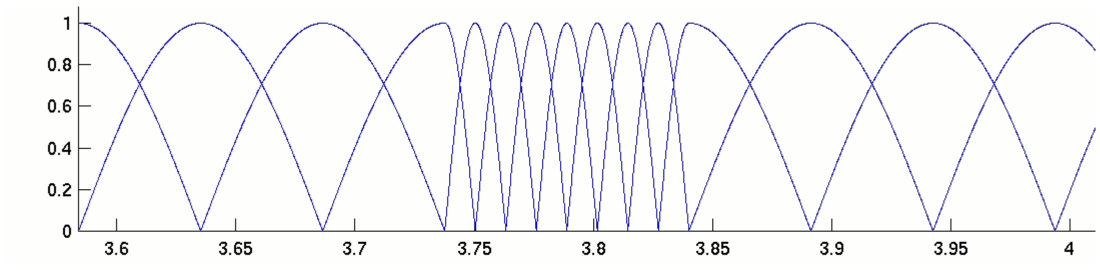


Figure 3: The windowing and block switching representation of the MDCT [6]

It then takes the DCT of this overlapped signal. We used the following equation to compute the DCT

$$y_k = 2\sum_{n=0}^{N-1} x_n \cos\frac{\pi(2k+1)(2n+1)}{4N}$$

Where y_k is the array of forward DCT values, x_n is the array of input of raw data values, *k* is the spectral coefficients, and *N* is the length of the input data. This overlapping and then taking the DCT avoids distortion from block boundaries while also compacting the energy in the signal.

For our filterbank, we decided to implement the case where there is only a "LONG" window. This means that the length of each window will be 2048 samples, and we don't have to worry about switching to another window type that has fewer than 2048 samples when overlapping. This windowed and filtered data is then sent to be scaled and quantized. For our implementation, we ran into a couple issues when testing the filterbank and block switching. At first, we tried many different approaches to manually calculate the MDCT (and the DCT within this function as well). We toiled with numpy arrays, for loops, tensorflow, padding, testing, windowing, etc. And although we learned a lot about how MDCT and block switching works, we decided in the interest of time and our continued learning to incorporate an already implemented version of MDCT with tensorflow. This design decision helped us focus on other parts of the AAC model.

The psychoacoustic model essentially masks the behavior of the human auditory system mathematically, so that we could process it. It generates thresholds by calculating the maximum distortion energy that is masked by the signal energy. The main component of the psychoacoustic model is how it determines the masking thresholds. Louder sounds (i.e. the masker), will tend to suppress softer sounds in some spectral or temporal neighborhood of the masker. This is shown in Figure 2 [5].

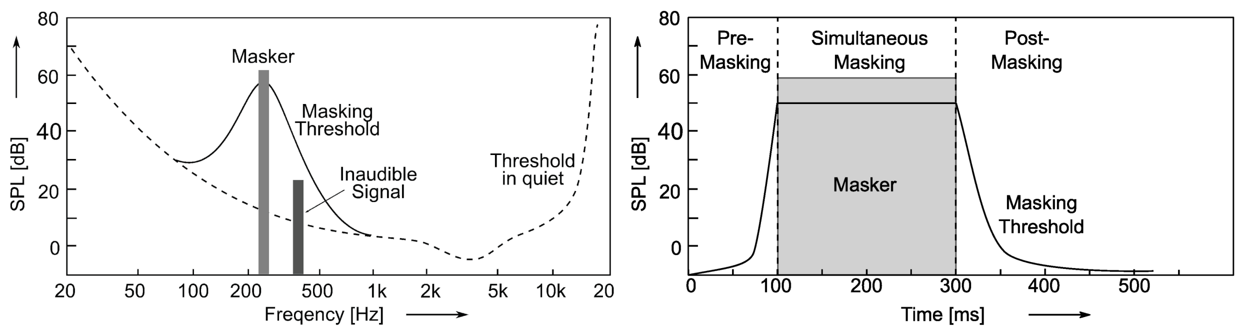


Figure 2: Psychoacoustic model hearing thresholds [5]. The left shows the spectral masking effects and the right shows the temporal masking effects. It is interesting to note the non-causal effect premasking whereby loud sounds will still mask a soft sound even if it happens before the loud sound.

It is described in detail in [1], however this version is extremely finetuned and complex, so we will implement a much simpler version that contains the essence of the model. The version that we have chosen to implement is based on the FastEnc implementation of AAC, which is standardized through the 3rd Generation Partnership Project (3GPP) [4,5]. In this FastEnc implementation, the psychoacoustic model is calculated directly on the MDCT filterbank, and the threshold calculation is based on a simpler set of assumptions. We implemented a simplified version of the psychoacoustic model that determines the thresholds for each block. A series of calculations is done based on section 5.4.2 in the 3GPP TS 26.403 standard. This was implemented in psychoacoustic_model.py.

In the full version of the AAC model, these thresholds would be fed into a non-linear quantization module, and adjusted as to achieve a desired average bitrate using a bit reservoir. Ideally we would have implemented this but decided it was too complicated to implement in the time we were given. However, we decided to use an approach that was similar to the quantization in JPEG where less important frequencies in the spectrum have a larger quantization. We therefore scaled the thresholds to give a sensible range of integer values that the spectrum could be divided by to allow quantization. The threshold scaling step was also implemented in psychoacoustic_model.py, and the quantization was implemented in quantization.py.

This compression model uses Huffman coding on the quantized spectra. Our compression model uses the Stanford Compression Library's Huffman coding module on our quantized spectra, and generates a codebook based on the values. With the official AAC model, the codebook is used to represent *n*-tuples (*n* is either two or four for the tuple size) of the quantized spectral coefficients. There are eleven codebooks that the Huffman code can draw from [1]. However, for our implementation, we decided to simplify the number of codebooks, and just create one codebook for the WAV file we are compressing. This codebook is compressed along with the audio file, which adds a small overhead. However, this design decision also means we don't have to store those eleven codebooks with our model, thus saving space.

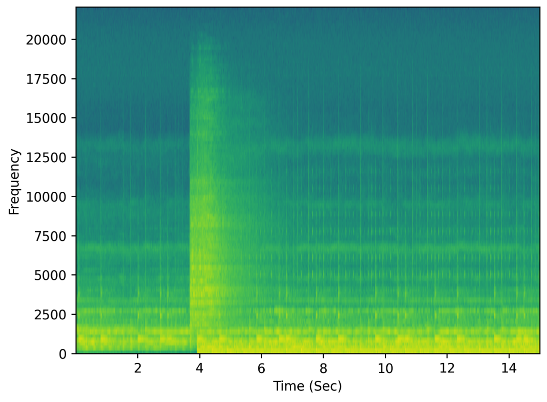
We were able to create a rudimentary AAC model that compresses a WAV file to a compressed file based on AAC. We qualitatively assessed our results by conducting listening tests to compare our AAC compression with the reference WAV files, as well as the output from a commercial AAC compressor software. We also performed a quantitative assessment with the Mean Squared Error (MSE) and looked at a spectrogram between the original and AAC-compressed files, which may correlate more closely with human perception rather than directly assessing output waveform files.

Results:

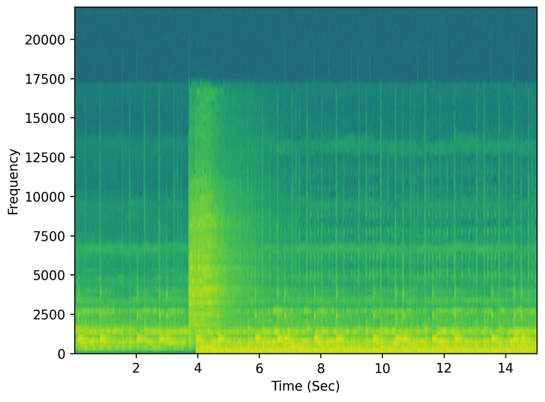
With our completed AAC model, we wanted to see how it would fare against different quantization levels and against a more complex AAC model. Table 1 shows the MSE and bitrate for various different levels of quantization of our AAC model. Even though MSE is not a perceptual metric for audio, we still included it as to provide a quantitative point of comparison between the different scenarios.

Table 1: AAC Compression MSE and bitrate performance

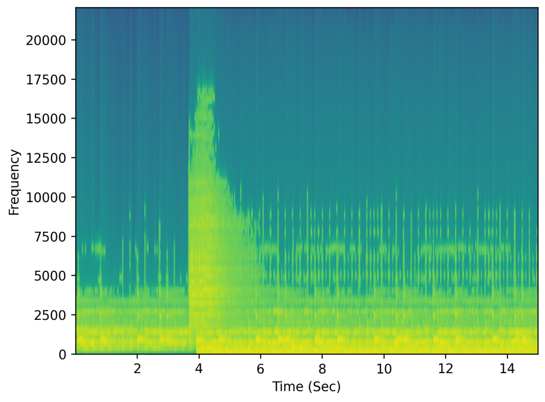
Level of Quantization	Bitrate (kbps)	MSE
1	78.8	2.32×10^{-5}
3	71.1	$8.2\ 3\times 10^{-5}$
7	65.6	3.02×10^{-4}
10	63.9	5.20×10^{-4}



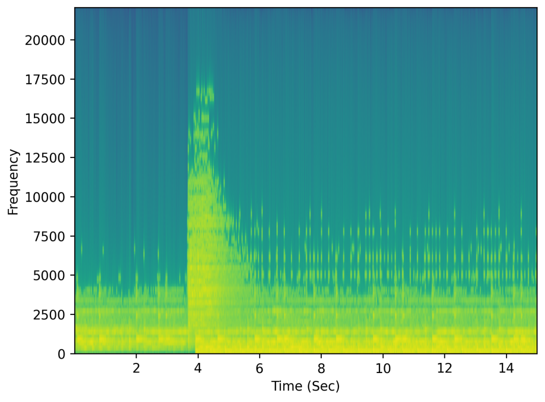
(a) Original Audio



(b) MPEG AAC



(c) Quantization level of 1



(d) Quantization level of 10

Figure 4: Shows the spectrograms of our original input file (a), the spectrogram of the MPEG AAC professional model (b), and the output after encoding and decoding with our AAC-encoder with quantization level of 1 (c) and quantization level of 10 (d).

These results show good compression results for a reasonable bitrate. We chose to compare our model against the MPEG AAC model. Unfortunately, we were unable to test the MSE of the MPEG AAC model since the output array sizes were different from the original audio and it was not clear how the MPEG AAC model modified the signal temporally. However, we were able to compare it with our model using a spectrogram as shown in Figure 4.

Qualitatively, the audio quality seems good for the 1 and 3 quantization levels. However, especially at the quantization level of 10 (corresponding to a 63.9 kbps) some aspects of the signal, especially the symbol crash are distorted. In comparison to listening to the MPEG AAC compressed file at a similar bit rate, the symbol crash is still clean, and overall the audio is more pleasing to listen to. This model incorporates more building blocks than our model such as M/S scaling, the bit reservoir that allows allocation of bits from less complex parts of the signal to the more complex parts, and the non-linear quantization method, so we expected it to perform better in a perceptual listening test.

The original audio and the MPEG AAC spectrograms (figures 4(a) and 4(b)) look really similar, but with the MPEG AAC spectrogram showing a clear cutoff frequency and looking slightly “blocky”. The results show that our AAC implementation, while resulting in a significant compression ratio, does not perform as well as FFMPEG AAC. This is to be expected since our implementation does not include many of the features in AAC.

For our AAC compression model, we experimented with different quantization levels and plotted their spectrograms. Figures 4(c) and 4(d) are two plots that we thought would highlight the differences in quantization levels and how they affect not only the noise we hear in the audio, but visually as well. Our AAC model seems blockier and not as smooth as the original nor MPEG AAC. This is especially noticeable in Figure 4(c) where the information at the 4 - 5 second range appears as little blocks instead of a smooth gradient which lines up with the distorted symbol crash we hear in its respective audio file.

Conclusion:

Our AAC model was successful in compressing the lossless WAV file into a format that was significantly smaller, but still retained the aspects of the signal that were most important. While our AAC model did not perform as well as a commercial AAC implementation, at higher bitrates it was still somewhat difficult to differentiate the original WAV file from our AAC compressed version. As part of this project we were able to learn about audio encoding, filterbank design, the human hearing system and quantization methods. We also gained experience in reading and understanding the complex standards that define AAC.
For future work, we would implement more of the optional AAC blocks into our system, conduct further fine-tuning of the parameters of our system to adjust the bitrate and quality and compare performance with more types of audio.

References:

[1] ISO/IEC 13818-7:2006 Information technology — Generic coding of moving pictures and associated audio information — Part 7: Advanced Audio Coding (AAC).

[2] Brandenburg, Karlheinz. "MP3 and AAC explained." Audio Engineering Society Conference: 17th International Conference: High-Quality Audio Coding. Audio Engineering Society, 1999.

[3] Chang, FM., You, S.D. (2004). Using Only Long Windows in MPEG-2/4 AAC Encoding. In: Aizawa, K., Nakamura, Y., Satoh, S. (eds) Advances in Multimedia Information Processing - PCM 2004. PCM 2004. Lecture Notes in Computer Science, vol 3333. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-30543-9_20

[4] 3GPP TS 26.403 V17.0.0, 3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; General audio codec audio processing functions; Enhanced aacPlus general audio codec; Encoder specification; Advanced Audio Coding (AAC) part (Release 17)

[5] Herre, J.; Dick, S. Psychoacoustic Models for Perceptual Audio Coding—A Tutorial Review. Appl. Sci. 2019, 9, 2854. https://doi.org/10.3390/app9142854

[6] Cuadra, P.; Liu Y.; Traube, C. ; Perceptual Audio Coder using Window Switching. March 2000. https://ccrma.stanford.edu/~pdelac/422/project/report.html