Voice and Audio Encoding Systems

“Implementation of an AAC encoder/decoder in MATLAB”

Pol Valls - NIA 184218

Sergi Solà - NIA 182466

Contents

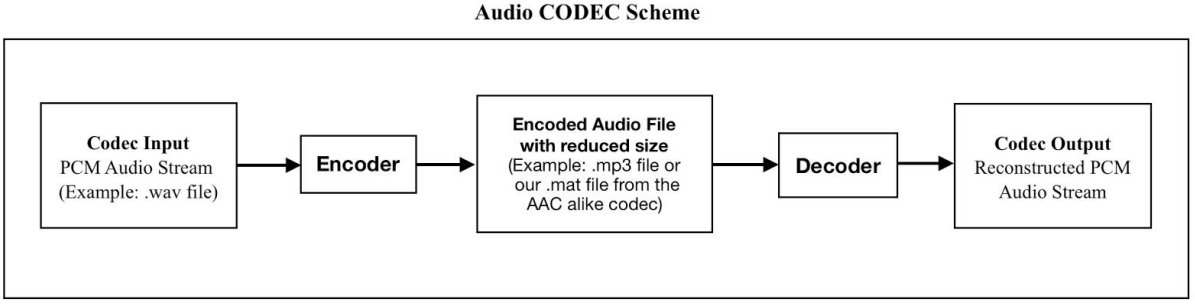
1. Introduction

1.1. Introduction to Audio Encoding.

Audio Encoding is about the storage of audio files in digital format. This digital representation of sound can be *uncompressed* (the exact raw data obtained from the hearing device) or can be *compressed* (a representation of the raw data with a reduced bitstream size). Furthermore, a compressed audio can be *lossless* (having no loss of information in relation to the original data) or can be *lossy* (an irreversible compression of the original data that uses inexact approximations and partial data discarding to represent the original sound).

Moreover, the size of lossless audio files require a relatively large transmission bandwidth and storage space, whereas the smaller size of lossy audio files require much less bandwidth and storage (but suffer the trade-off of losing sound fidelity). Thus, Audio Encoding is an area of knowledge that aims for the creation of digital sound representations that have the most reduced bitstream size while retaining as much sound fidelity as possible.

Currently, the standard method used to digitally represent uncompressed audio signals is PCM (Pulse Code Modulation[1]). Therefore, in order to convert an uncompressed audio file based on PCM (e.g the popular .WAV Format) into a reduced size format and vice versa, an audio compression software known as CODEC (COding and DECoding) is needed. An audio codec follows this simple scheme:



The amount of compression produced by a codec is quantified with the *compression ratio* (the ratio between the input size and the encoded file size) and with the resulting audio *bitrate* (number of bits used in the encoded audio file per second of sound).

The bitrate of a PCM audio stream can be reduced with several simple methods:

1. Lowering the sampling rate.
2. Lowering the number of channels.
3. Lowering the bits/sample (quantization resolution).

But these methods only vary PCM audio properties directly proportional to sound quality, so they have no interest for us. On the other hand, a modern lossy audio codec creates a compressed encoded file by using advanced data discarting and encoding techniques for audio compression.

[1] In a PCM stream, the amplitude of the sound signal is sampled regularly at uniform intervals, and each amplitude sample is quantized with a given number of bits.

This advanced data discarting techniques perform audio compression based on the human hearing perceptual limitations (psychoacoustic modeling, perceptual coding or perceptually driven quantization) and on data encoding methods based on redundancy reduction. In general, modern audio codecs will perform encoding throughout these similar steps:

1. Transform the original data (PCM stream) into a perceptual based representation of the sound. (We will go into further detail in following sections)

2. Perform perceptually driven quantization on the obtained representation that consists of variables in a continuous or large range of values (e.g frequency power spectrums or any desired information to describe the sound).

3. Encode the quantized representation using entropy coding and redundancy reduction to create the final encoded file. (for example with huffman or Arithmetic encoding)

*Note, the decoding will be the "backwards process" of the encoding.

Because of the huge importance of the perceptual based representation of the sound and how it is quantized, it becomes clear that the understanding of human Psycho-Acoustics play a major role when it comes to efficient audio compression.

1.2. Introduction to Psycho-Acoustics.

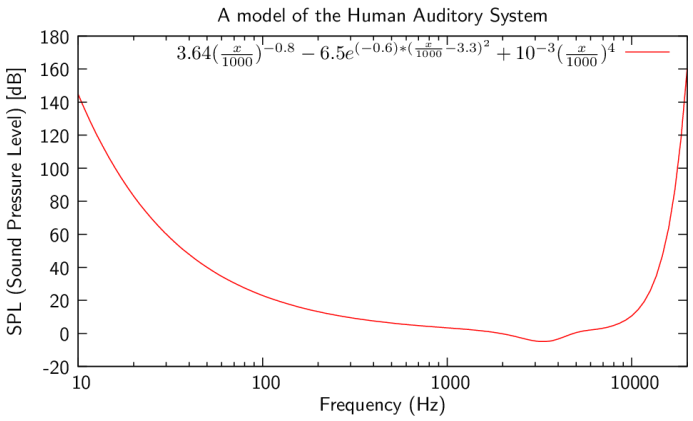
Psycho-acoustics is the field of study of human sound perception. Needless to say, hearing is not a purely mechanical phenomenon of wave propagation, but is also a sensory and perceptual event, so certain differences between waveforms can actually be imperceptible, and perceptual coding takes advantage of this.

The most important perceptual coding techniques used in audio codecs are based on the following perceptual phenomena:

a. **Frequency hearing range**
   Also known as high frequency limit. The human ear can only hear sounds in the frequency range of approximately from 20 Hz to 20,000 Hz, although that the upper limit tends to decrease with age (most adults are unable to hear above 16 kH) and tones between 4 and 16 Hz can be perceived via the body's sense of touch.

b. The absolute threshold of hearing

The absolute threshold of hearing describes the minimum sound level of a pure tone that an average human can hear. It is generally described with the following expression:

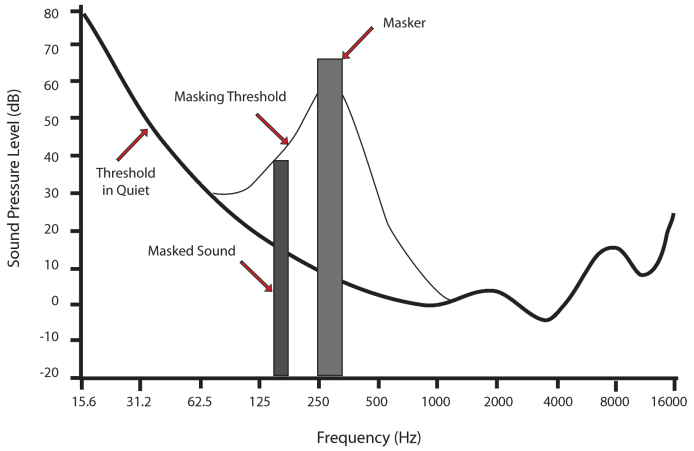


As we can see, the red line is the minimum sound level required for the human auditory system to hear a tone at each given frequency. Note that we hear much better the sounds that range between 2.000Hz and 4.000Hz., which approximately is the frequency range of speech. This information will be used to decide which frequency ranges should have less quantization error.

c. Auditory Masking

c.1 Simultaneous Masking

The frequency resolution of human hearing is limited. The audible frequencies can be grouped into sub-bands with perceptually equal distances, known as barks. Each bark defines the group of frequencies that excite the same cochlear area of the ear, i.e., those frequencies that can be masked by the tone with the highest energy (e.g a powerful spike at 1 kHz will tend to mask out another simultaneous lower-level tone at 1.1 kHz). The following image is a very clear representation of the simultaneous masking and the masking spread cones in the frequency domain:



c.2 Temporal masking

The human auditory system has inertia: sounds are not instantly perceived and remain after they have disappeared, so a non-simultaneous time domain masking can accurr when a sudden stimulus sound makes inaudible other sounds which are present immediately preceding or following the stimulus.

1.3. History of Audio Codecs and perceptual coding

The first ever psychoacoustic masking digital codec was proposed in 1979 by Manfred R. Schroeder from Bell Telephone Laboratories[2] and was addressed for speech coding.

Later on, in the late 80's, when analog audio formats were still predominant, the technological advancements in storage and computing capabilities allowed for the first digital audio formats to appear and be widely used. This formats were based on PCM and would be stored in physical devices specially designed for this purpose (e.g Compact Disc (CD-DA) format). At the same time, a wide variety of perceptual audio compression algorithms were developed, most using auditory masking as part of their fundamental design[3]. Also, in 1988, the Moving Picture Experts Group (MPEG) was established. The most important audio compression codecs of the moment were the Optimum Coding in the Frequency Domain[4] (OCF) and the Perceptual Transform Coding[5] (PXFM).

Once in the early 90's, different organizations such as the International Organization for Standardization / International Electro-technical Comission (ISO/IEC) and International Telecommunications Union (ITU) started integrating different perceptual audio coding standards developed by MPEG and other research groups.

In 1992, the popular lossless audio codec .WAV (*Waveform audio format*) based on PCM was released and became the standard for uncompressed lossless audio files to this day.

Shortly after, in 1993, the Perceptual Masking compression method known as Masking Pattern Universal Sub-band Integrated Coding and Multiplexing (MUSICAM) was released as part of the MPEG-1 audio codec standard. This MPEG-1 release was a game changer and included the first version of the extremely popular and widely used MPEG-1 layer 3 (with extension .MP3) format. Meanwhile, successful companies such as Sony or Dolby also developed their own commercial audio standards like the Sony's Adaptive TRansform Acoustic Coding (ATRAC) or the Dolby Digital Theatre System (DTS) and Dolby's AC-3. The popular MP3 format would soon become the standard for the music and audio file formats on the internet, while AC-3 was the standard used for the encoding of video sound. Although .mp3 is very popular to this day, other codecs have been demonstrated to perform better in low bitrates (e.g the Advanced Audio Coder (AAC), the subject of this work).
In the late 90's, the first surround sound systems appeared and the interest shifted into the multi-channel audio encoding formats. In 1997, the MPEG-2 and MPEG-4 audio and video standardizations were updated and released respectively, establishing the basis for the research on

---

[2] Schroeder, M.R.; Atal, B.S.; Hall, J.L. (December 1979). "Optimizing Digital Speech Coders by Exploiting Masking Properties of the Human Ear". *The Journal of the Acoustical Society of America*. **66** (6): 1647. Retrieved from http://adsabs.harvard.edu/abs/1979ASAJ...66.1647S.

[3] Voice Coding for Communications". *IEEE Journal on Selected Areas in Communications*. **6** (2). February 1988.

[4] Brandenburg, Karlheinz; Seitzer, Dieter (3–6 November 1988). *OCF: Coding High Quality Audio with Data Rates of 64 kbit/s*. 85th Convention of Audio Engineering Society. Retrieved from http://www.aes.org/e-lib/browse.cfm?elib=4721

[5] Johnston, James D. (February 1988). "Transform Coding of Audio Signals Using Perceptual Noise Criteria". *IEEE Journal on Selected Areas in Communications*. **6** (2): 314–323. Retrieved from http://ieeexplore.ieee.org/document/608/

new high quality coding of general audio and video, even in low bit rates. The MPEG-2 and MPEG-4 standards introduced the theoretical successor for the .mp3 format, the Advanced Audio Coding (AAC, with extension .aac, .m4a or .mp4).

Later, the release of new high storage capacity formats in 1999 such the Super Audio Compact Disk (SACD) motivated researchers to develop new lossless audio encoding algorithm schemas that were not generally adopted, such as the Apple Lossless Audio Coder (ALAC) or the Windows Media Audio 9 (WMA9). Another lossless codec was the Dolby TrueHD, developed by Dolby Labs to be used for high-definition home-entertainment equipment such as the Blueray Disc. Once in the 21st century, the free lossless audio codec (.FLAC) was released and has gained nearly as much popularity as the classic .WAV format in the lossless arena. Meanwhile, the most commonly used lossy formats have continued to be the MP3 and AAC formats.

2. Advanced Audio Coder Implementation in MATLAB

2.1. Introduction to the Advanced Audio Coder (AAC)

The Advanced Audio Coder (AAC) is an audio coding standard. It is based on lossy digital audio compression and it was developed with the collaboration of different companies such as Bell Labs, Fraunhofer Institute, Dolby, Sony and Nokia from 1994 to 1997. Since then, several updated versions have also been added to the standard.

AAC usually achieves better sound quality than other popular coders (such as its predecessor, mp3) at low bit rates and equal quality at average bitrates. The official encoder implementations also requires less resources from the system for coding and decoding. AAC was standardized in April 1997 as an extension of MPEG-2 and the new MPEG-4 standard created by the Moving Picture Experts Group (MPEG). Due to its high performance and quality, it has been used in a wide range of applications, specially those implemented in the internet such as iTunes (Apple), Ahead Nero, Winamp, Youtube and Play Station 3 among many others.

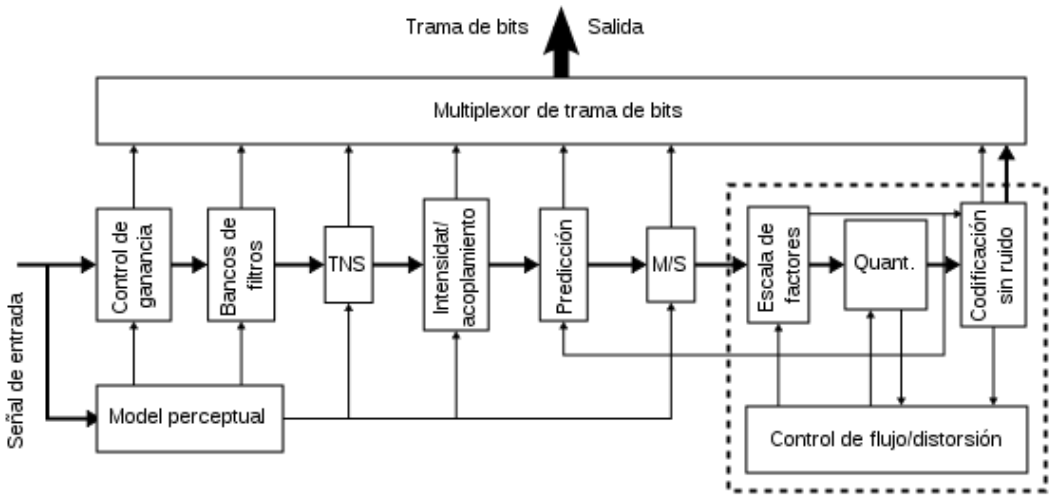
AAC uses a variable bit rate (VBR) coding method that adapts the number of bits used to code the audio data depending on the complexity and perceptual characteristics of the audio signal in that particular moment. The AAC compression algorithm also uses the two different coding strategies mentioned in the 1.1 section: discarding perceptually irrelevant signal and discarding redundancies in the coded audio signal.

In the MPEG-4 audio standard, instead of defining a highly efficient compression scheme, a complex toolbox is used to allow many encoding options, from speech coding to high quality audio coding, with sampling frequencies between 8kHz and 96kHz and any number of channels between 1 and 48.

In the following sections, we will go into further detail about the functioning and encoding strategies of the AAC codec and we will work on a MATLAB implementation of a codec as much similar to AAC as possible.

2.2. Introduction to the AAC encoder modular structure

The AAC follows the steps described in this block diagram to encode the audio data:



In this block diagram we can observe the order and relation between the methods used in AAC to encode sound. The perceptual model is taken into account on each block or step in the encoding process up until after the quantization process. The detailed blocks are:

Gain control of the input sound: To produce the least noisy recording, the gain control should be set as high as possible but without being so high as to clip or distort the signal.

Filterbanks: Based on the Modified Discrete Cosine Transform to convert the input signal from time-domain to frequency-domain.

Temporal Noise Shaping (TNS): A technique to reshape the quantization noise over time with the aim to reduce artifacts in transient and speech signals (It uses perceptual models and divides in subbands).

Intesity/Coupling: Applies data discarding techniques based on auditory masking.

Prediction: Sound fragments/frames are analysed and classified in types, then this information is used to change each frame encoding methods and improve encoding efficiency.

M/S (Mono/Stereo): Finally, some multi-channel encoding techniques are used as AAC can have up to 48 channels. Channel coupling decreases inter-channel redundancy, usually, using prediction techniques.

Quantization and encoding: The resulting data is quantized using the calculated perceptual criteria and encoded with a specific huffman dictionary, then sent to the bit frame multiplexer to obtain the final file bitstream.

2.3. Preliminary search of an AAC implementation in MATLAB

To design and code from scratch an entire AAC implementation in MATLAB would be a huge load of work. Instead, our idea is to examine and understand an already existing implementation of an AAC encoder in MATLAB and then improve on it.

With the information from the introduction sections and having studied the techniques and modular structure that are applied in an AAC Encoder, we searched in the internet for some implementations of AAC alike Encoders in Matlab. We looked into some different options:

1. In the Matlab website's File Exchange section, we could find a very simple AAC implementation made by the user Ravi Lakkundi, but after analyzing a bit what the code was doing we realized that it was not doing the most part of the things that an AAC Encoder does. This implementation pretended to be a real AAC encoder and created encoded files with a fake .AAC extension, but these files were not readable by media players such as VLC, iTunes, Groove, Windows Media, it could only be turned back into a wav file with the provided matlab decoder. We also realized that this fake .aac file actually had a low compression ratio, so we decided that we would not work on that encoder. This implementation can be downloaded from the following website:
   https://es.mathworks.com/matlabcentral/fileexchange/26137-aac-encoder

2. We also thought about looking into the actual Matlab audiowrite function that supports AAC encoding. We looked into the source code of this function and after debugging and some research we found that the function creates "channel" variables that communicate with external audio libraries and plugins to do the encoding and writing of the file. This makes a lot of sense because otherwise the function would have had to compile the whole AAC encoding functions every time that it is called.

3. After discarding other options, we finally stumbled upon the website SearchCode.com where we found an AAC Codec project implemented in Matlab that seemed to be very complete. We could not even download the code from the website, so we had to copy and paste it into matlab. At first we found that there were many duplicated and unfinished functions. Nevertheless, it implemented most of the AAC encoding techniques such as the filter banks, Temporal Noise Shaping, the perceptual model, quantization, standard MPEG huffman encoding, etc. After analyzing this code, understanding most of it and doing minor bug fixes to run it and test it, we were excited about most blocks/functions working properly, even though that the overall encoder had several issues and was awfully structured and without explanatory comments. Despite all of this, we decided to work on the understanding and improvement of this code anyways (because it was the best AAC implementation we could find). This stepping stone code that we have used can be found in the following website:
   https://searchcode.com/file/11693972/trunk/source_tree.txt

2.4. Analysis of the original AAC encoder/decoder implementation.

The original source code we decided to work on was from 2011 and included a README.txt file that explained that the authors developed the code as part of an unspecified academic project. It also mentioned which author had done each part of the code and whether or not this part was finished. About the authors we only know their names: Donovan Fortsworth, Elias K and Giannis A.

2.4.1 General guidelines of original source.

This original source code we decided to use was in fact a combination of 3 "codecs", each adding complexity to the previous one. More specifically, each implemented part of the AAC encoding strategies and stored the transformed data in a matlab variable that could then be used by a given decoder to reconstruct the sound. Each "codec" or level of complexity implements the following AAC tasks/methods:

Level 1: The first level performs a frame segmentation of the input signal. Then it assigns a frametype to each of the frames. Finally it applies filter banks to each frame transforming them to the frequency domain taking into account the frame types.

Level 2: The second level performs the same steps as level 1. Then, it applies a psychoacoustic model based on simultaneous masking. Afterwards, it applies TNS (Temporal Noise Shaping) to obtain a coefficients that indicate how to reshape quantization noise.

Level 3: The third level of complexity would replicate the level 1 and level 2 steps and then perform perceptually based quantization and huffman encoding to obtain the "encoded stream" of bits.

*Notice that levels 1 and 2 have no loss of information and no compression at all, compression is only accomplished in the level 3 quantization and huffman encoding.

For further detail see function explanations below.

2.4.2. Original code function tree:

Level 1	Level 2	Level 3
|-- SSC.m	|-- TNS.m	|-- AACquantizer.m
|-- filterbank.m	|-- iTNS.m	|-- iAACquantizer.m
|-- iFilterbank.m	|-- psycho.m	|-- huffman.m
|-- AACoder1.m	|-- AACoder2.m	|-- iHuffman.m
|-- iAACoder1.m	|-- iAACoder2.m	|-- AACoder3.m
|-- demoAAC1.m	|-- demoAAC2.m	|-- iAACoder3.m
		|-- demoAAC3.m

2.4.3. Detailed function explanations. (Recommended to skip to 2.4.4)

In this subsection we have described in detail what each function does. We have written it to better understand the code ourselves and the explanations are long and dense, feel free to skip to section 2.5.

The explanations are the fruit of our deductions on the code and our knowledge about AAC, the original code included few comments. The original code functions were:

AACoder1.m/iAACoder1.m, AACoder2 .m/iAACoder2, AACoder3.m/iAACoder3

These were the main encoder and decoders of Level 1, 2, and 3. Each would use the previous one. The iAACoder1.m, iAACoder2.m and the iAACoder3.m are the decoders for the encoded data.

AACoder1.m starts by reading the input stereo audio file and dividing the data samples into frames and generate the struct that will contain the output signal. This struct has four different fields which are the frametype, the wintype and the frame data of the two channels (left and right). Then the code calls the function SSC to know the type of each frame. Finally each frame is filtered using the function filterbank.m. The obtained data is saved in the struct generated before.

AACoder2.m starts doing the same as AACoder1.m as it reads the input file and divides the array of data samples to encode into frames. Then the code generates the struct that will contain the output signal data and has the same fields as the struct from AACoder1.m but adding two new fields: the TNS coefficients and the acoustic threshold. After that, the code calls the function SSC.m to determine the type of every frame. Then in a new for, frames are filtered using the function filterbanks, the psychoacoustical model is applied to compute the acoustic threshold and the TNS function is called to compute the TNS coefficients. Then, all this data is saved in the struct generated before.

AACoder3.m starts by reading the stereo audio input file and dividing it into frames of fixed size and defining the structs with different fields such as the frametype, the wintype, the TNS coefficients, the acoustic threshold, the stream of bits generated to encode each channel of the frame and the gain. Then, the function declares the type of the first frame to use it when we call the SSC function for the following frame. Inside a for loop we call the SSC function for every frame. Then AACoder3 calls the filterbanks function, the psycho function, the TNS function, the quantizer function and finally the huffman encoding function for every frame and saves all the data to the struct that has been created in the beginning.

SSC.m (Sequence Segmentation Control)

This function receives as a parameter a time domain frame and the type of the previous frame and returns the type of this frame that can take the following values: OLS for Only Long Sequence (1), LSS for Long Start Sequence (2), ESH for Eight Short Sequence (3) and LPS for Long Stop Sequence(4).

For the two channels of the frame, the first thing the function has to do is to generate the high-pass filter and filtering the frame with this filter. Then it computes the energy of the signal in this frame by squaring the frame, the attack values of this frame by summing all energy values of the frame and dividing for the length of this sum. Then using some defined conditions the type of the frame for each channel is chosen.

Filterbank.m / iFilterbank.m
Filterbank.m receives as parameters the current time domain frame, the type of this frame and the type of window that we are using (Sinusoidal window or Kaisser window). Then, for every channel, if the type of the frame is ESH, it divides the frame into eight smaller frames and applies the windowing and the MDCT for each of this smaller frames. If the type of the frame is not ESH, it only applies windowing and the MDCT to the frame.

The iFilterbank.m function is used in the decoding process so we will not explain how does it work.

Win.m, KBDWindow.m, sinWindow.m
This three functions are all related to determining and creating windows. The Win function receives as parameters the length of the window, the type of window we want to create and an alpha value for the KBD window. Then, if the wintype is the sinusoidal window it calls the sinWindow function. If the wintype is not the sinusoidal, it calls the KBDWindow function. The KBDWindow function receives the length of the window and the parameter alpha as parameters and generates a KBD window using this parameters. The sinWindow function receives the length of the window and generates a sinusoidal window.

mdctv.m / imdctv.m
This function was created by the researcher from the Columbia University, Marios Athineos, and it is used to compute the Modified Discrete Cosine Transform in a vectorized way of an input vector. The imdctv.m function does the inverse process for the decoding part. These two functions can be found on the internet and we will not explain them any further.

psycho.m
This function receives the current frame's frequency domain representation and the frametype as parameters and it is used to compute the psychoacoustic thresholds for every band and the energy spectrum of every band, which are the values that this function returns.

The first thing that this function does is to compute the hearing threshold in quiet depending on the frametype and then call the filterbank function using the KBD Window. Then, for each channel, the function computes the power spectrum what is multiply the frame by itself for every band and sum all the values. Then, it is important to call the Temporal Noise Shaping (TNS) function and compute the power spectrum again after applying TNS. Having done that, the function computes the acoustic threshold and the threshold spread for each band. The final threshold, for each band, is the maximum between the threshold in quiet and the threshold spread.

TNS.m
This function receives a filtered frame, the type of this frame and the power spectrum computed in the psycho function and implements the Temporal Noise Shaping. The first thing this function does is to define the bands in barks for long and short (ESH) frames. Depending on the type of the frame it does the Temporal Noise Shaping in different ways.

For OLS, LSS or LPS frames, computes the square root of every band and smoothing the results by suming two of them and dividing them by two. Then, all the values of the frame, for each channel, are

divided by the value that it is obtained from the step before. After that, the function computes the LPC coefficients of the frame values that has been obtained in the step before and create a filter that has to be stable. Then, for every coefficient, if the number is not between the coding range it is moved so that it is then between this range. After that, the TNS coefficients are computed based on the coefficients that were obtained in the previous steps for each channel. Finally, if the frame is filtered by some coefficients, a new frame is obtained as the result of applying the Temporal Noise Shapping.

For ESH frames the function computes the output frame and the TNS coefficients as explained before but having into account that an ESH is a frame that has been divided in 8 shorter frames of the same length.

AACquantizer.m

This function receives the current frame, the type of this frame and the acoustic threshold and it is used to quantize this frame. This function returns the vector of symbols from the MDCT coefficients quantization of the frame, the gains for each band and the global gain of the frame.

The first thing that this function does is to declare the Bark bands. Then, if the type of the frame is OLS, LSS or LPS, then the quantization of the frame is done by doing the sum of the square roots of the values of the frame for each band and for each channel. Then, the symbol can be obtained by applying some rules that can be seen in the code. Each symbol will be then coded using a Huffman code. After that, for each channel and for each band, the function computes the quantization noise (difference between the quantized value and the original value) and the power of the quantization error (multiply each quantization error value for itself and sum the obtained values) and convert this value to dB. If the quantization error is below the acoustic threshold in that band the function adds one unit to the amplitude and subtract one unit if it is not. Having into account this changes, the obtained symbols are computed again.

The Global Gain (G) for each channel is the maximum of the amplitudes. The gain of each band can be obtained by subtracting each amplitude to the global gain and then subtract the value of the previous one to the value of the current one.

For ESH frames the function quantizes the frame as explained before but having into account that an ESH is a frame that has been divided in 8 shorter frames of the same length.

Huffman.m, ihuffman.m

The huffman function the symbols and the frame type and it is used in the coding phase of the encoder to code the data using a Huffman code. This function returns the stream of bits of the frame coded, the codebook used and the stream length. The ihuffman function is the inverse function and it is used during the decoding process so we will no explain how does it work.

The first thing that the huffman function does is loading all the Huffman codebooks from the .mat file using the loadLUT function. Then another function is called, called encodeHuff that receive the symbols and the data imported from the .mat file and returns the stream of bits and the codebook used to code the frame. Then, this function calls another function called pushstream, which adds to the end of the stream variable the stream of bits obtained from the previous step. Finally, this function computes the length of the final stream of bits that it is used to code this frame.

EncodeHuff.m, DecodeHuff.m

The function EncodeHuff.m receives the quantized values and the table that contains the Huffman codebooks imported with the function loadLUT as parameters and return a binary string corresponding to the stream of bits used to code the input values and the number of codebook from the table that has been used.

The first thing that the function EncodeHuff.m does is to compute the maximum value in absolute terms of the input values. Then a switch is generated to select a Huffman codebook depending on this value and code the value using this codebook with the huffLUTCode1 function. If the maximum input value in absolute terms is 0, then the function used to code is the huffLUTCode0 function.If this number is not between 0 and 15 then the huffLUTCodeESC is called. The function huffLUTCode0 only returns an empty string while the others make complex actions.

The huffLUTCode1 function is used to encode all the symbols except if the symbol is 0 or bigger than 15. This function receives the codebook we have to use to code the symbols and the symbols and returns a stream of bits that are the coded symbols.

The huffLUTCodeESC function is used to encode symbols that are bigger than 15. It receives and returns the same parameters and variables as the huffLUTCode1 function and works in a very similar way determining an string of bits depending on the value of the symbol and the Huffman codebook and returns an stream of bits that is the concatenation of the strings generated for every symbol in the order that the symbols are coded.

As the DecodeHuff.m is used in the decoding process, we will not explain how this function works.

loadLUT.m

This function imports the huffman codebooks from two .mat files and puts each codebook into a different struct. As this function has been extracted from the internet we will not explain how does it works.

2.4.4. Original Source Code Issues

Finally, we identified the purpose and understood the code of each given function, but the code presented some serious issues:

a. The code was from 2011 and used old matlab functions such as wavread, etc.

b. We needed some .mat files that were not available in the searchcode.com repository. For example, the standard MPEG huffman dictionary huffCodebooks.mat imported in loadLUT.m. We had to find it in a greek university forum that also used loadLUT.

c. Some functions were duplicated and we would not know which was the definitive version.

d. The code was extremely poorly organized and structured. (Probably because there were many authors that had to coordinate).

e. The general code quality (how ideas and solutions were implemented) was not very efficient nor nicely done. For example, Level 3 would execute the Level 1 and Level 2 and copy their results or repeat the calculations itself depending on the part of the code. Also, the code had minimal comments and was hard to understand.

2.5. Our AAC encoder/decoder. Changes and improvements applied to source code:

After analysing the source code and identifying its problems, we decided that we were going to create our own AAC encoder and decoder based on the source code.

2.5.1 Improvements of our AAC encoder/decoder in relation to the original source code

1. We have unified the 3 levels into a unique codec.

2. Our codec does not only work with matlab variables. It creates an encoded audio file in the form of a .mat file, which the decoding function then reads/loads and turns into a reconstruction of the original audio.

3. We have added the option to compute codec performance information:
    a. The quality of the compression with the SNR between original and decoded sound.
    b. Compression ratio between original and encoded file.
    c. Compare old and new reduced bitrate.

4. Finally, our code, presents a much simpler function organization and is nicely commented and explained.

2.5.2 Our final function-tree is:

```
Our AAC encoder / decoder

|--main_AAC.m
              |--AAC_encoder.m
                              |--SSC.m
                              |--Filterbank.m
                                              |--mdctv.m
                                              |--sinwin.m/kbdwin.m

                              |--TNS.m
                              |--psycho.m
                              |--AACquantizer.m
                              |--encodeHuff.m

              |--AAC_decoder.m
                              |--decodeHuff.m
                              |--iAACquantizer.m
                              |--iTNS.m
                              |--iFilterbank.m
                                              |--imdctv.m
                                              |--sinwin.m/kbdwin.m
              |--DisplayInfo.m
```

2.5.3 Explanation of our functions and their differences from original source

main_AAC.m

This function is used to run and configure the encoding and decoding process. In this function we set the name of the input file, the encoded output file and the decoded output file. Finally we enter this names as parameters to the encoding and decoding functions and the codec is executed. Lastly, we have also created the DisplayInfo.m function to display codec performance information.

AAC_encoder.m / AAC_decoder.m

These are the main encoder and decoder functions. The encoder saves the encoded audio file as a .mat file with reduced size that the decoder function can read/load and decode back into a representation of the original audio.

The first thing AAC_encoder does is to initialize the frame length, window types and the default data to work. Then the function reads the audio file it wants to encode and applies zero padding so that the signal is perfectly divisible in segments of 2048). After that the encoder declares the struct that will contain the encoded data. This struct has four main fields: the frametype, the wintype and the data from each channel. The fields that contain the data from each channel has another struct inside formed by six fields: the TNS coefficients, the acoustic threshold, the stream of bits to codify the frame, the number of codebook we have used to encode the frame and the scale factor gains. Next, we import the

data from the huffCodebooks.mat file which contains all the information for the coding of frames using the loadLUT function.

Once we have all the variables initialized, it is time to start encoding the frames. All the encoding process takes places into a for loop with an iteration for each frame of the signal. Inside the for loop, the current frame and next following frame are selected from the signal. Then the function SSC determines the type of the current frame and once we know the type of the frame we can apply the filter banks calling the corresponding function. The following step is to apply Temporal Noise Shaping by calling the TNS function. Then, for every channel, we have to apply the psychoacoustic model to obtain the Signal-to-Mask Ratio (SMR) that will be used in the quantizer (AACQuantizer.m), the next function to be called. Then the encodeHuff function is called to obtain the frame encoded and saved in the struct. Finally, we save this struct to a .mat file.

As the AAC_decoder.m function is used in the decoding process we will not explain it. Yet, we have made many changes to the decoder in order to merge the three original decoders into one.

SSC.m

SSC.m has not been changed. Just like in the original source code, this function implements the Sequence Segmentation Control step, meaning that it determines the type of a frame depending on the type of the previous frame and the power distribution of the following frame. The frame types are the same as the original.

Filterbank.m / iFilterbank.m

The filterbank function implements the filterbanks and receives the current frame in time domain, the type of the current frame and the type of window that will be used and returns the current frame in frequency domain after applying the Modified Discrete Cosine Transformation.

The main changes we have made to this functions are that we have eliminated the if condition that analyzed the type of window before creating the window in every case of the switch creating a new function called window that receive the window type as a parameter and with only one if (instead of one for each case of the switch) we can create the window we want. Another thing we have changed is that in the code we had before the MDCT was applied in every case of the switch, but now the MDCT is only applied at the for loop located at the end of the function.

The iFilterbank function is only used in the decoding process so, although we have made some changes to adapt it to the new code and solve some problems, we will not explain how it works.

TNS.m / iTNS.m

The TNS function implements the Temporal Noise Shaping to the frequency domain frame we got from filterbanks. Its input parameters are the current frame and the frametype. It returns a filtered frame and the TNS coefficients.

The main changes made to the TNS function are:
We have simplified the initial if conditions into just one that directly chooses the correspondent bark scale using the frame type. We have also simplified a lot the rest of the function taking advantage of

how Matlab works with matrices. For more info on what the function does and we have changed, see the original and our new commented functions.
Although iTNS have had some changes, we don't explain it.

Sinwin.m / Kbdwin.m
The sinwin function is used to create a sinusoidal window and receives the length of the window we have to create as a parameter. This function return the window for the two channels (left and right). In the code we had before the window was only created for one channel so we had to call this function twice to obtain a window for the two channels. So we changed so that this function creates the window for each channel of the frame doing each step twice (once for channel).

The kbdwin is used to create a KBD window of a determined length and alpha parameter that the function receives as parameters. As in the sinwin function, we have also made that this function returns the KBD window for each channel of the frame saying that the window of one channel will have to be the same as the window for the other one. Another thing that changes in that function is that instead of writing the formula by hand in the code we call to a Matlab implemented function called kaiser() passing the correct parameters and making the respective modifications afterwards.

Psycho.m
This function implements the psycoacoustical model and has changed quite a lot from the one we had before. This new function receives the current frame in frequency domain, the type of the current frame, the previous frame and the twice previous frame and returns the Signal-to-Mask Ratio of this frame. For further information look into the new commented code.

Mdctv.m / imdctv.m
It is exactly the same as before, so we will not explain how this function works.

AACQuantizer.m / iAACQuantizer.m
The AACQuantizer function does the quantization of a frame. This function receives the frame in the current frequency domain representation, the type of this frame and the Signal-to-Mask Ratio as parameters and returns the corresponding quantization symbols (s), the scaling factor gains (quantization allocation) for each band and the global gain.

The function first computes the power spectrum of the frame as a squared sum of the frame, and then divides this power spectrum by the SMR to obtain the acoustic threshold (mask). Later, it uses this acoustic threshold to perform a sort of subband quantization allocation and obtain the final variables S, sfc and G.

encodeHuff.m / decodeHuff.m
The encodeHuff function is nearly the same as the original. Only now it can be called with two (like the original) or three parameters and depending on this an if will be used to call the rest of the exactly equal functions huffLUTCode1, huffLUTCode0 and huffLUTCodeESC.

loadLUT.m
It is exactly the same as the previous one, so we will not explain how this function works.

DisplayInfo.m

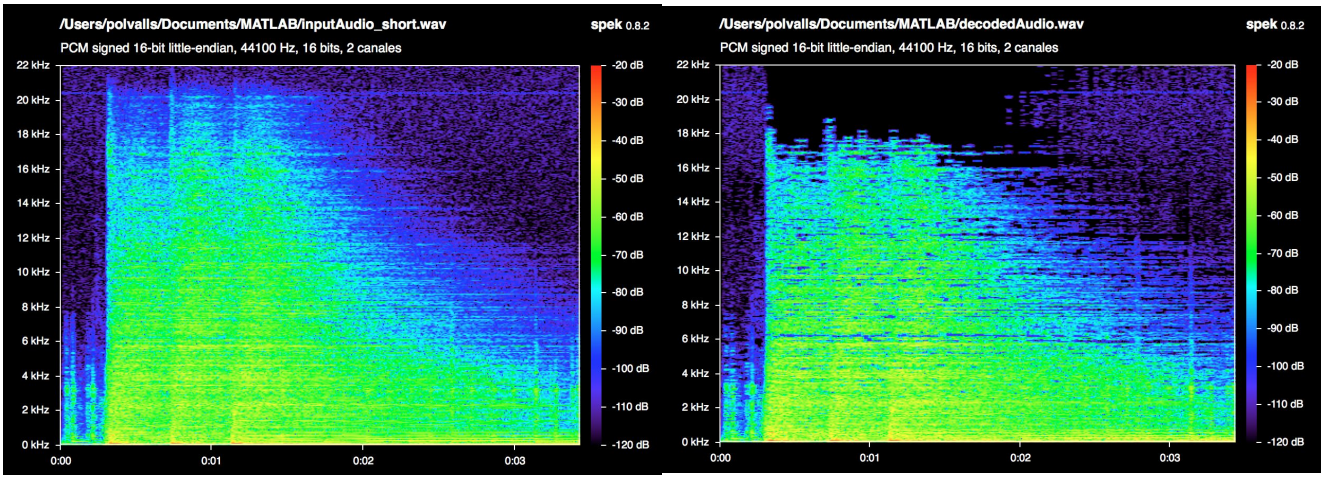
This new function we have created displays in the Matlab console the codec performance information such as:

a. The quality of the compression with the SNR between original and decoded sound.
b. Compression ratio between original and encoded file. (encoded_size/input_size)
c. Compare old and new reduced bitrate. (size/duration)

3. Presentation of results.

We selected a small portion of the song Eye of The Tiger by Survivor in lossless .wav format. Then we encoded it with our unified AAC alike encoder and when the process finished, we obtained the compressed EncodedAudio.mat audio file. Then, we executed our decoder with this EncodedAudio.mat and the name 'DecodedAudio.wav' as input parameters and we obtained a file named DecodedAudio.wav that was the reconstruction of the original audio.

If we listen to the original and the reconstructed audio files, almost no difference is noticeable. Yet, we tried plotting their spectrograms with the Speck (Free Acoustic Spectrum Analyzer) software and we obtained the following figures:



If we analyze the images above we see how the encoding/compression process has produced a loss of information that traduces in a loss of "resolution" in the spectrogram and in a reduction of the maximum frequencies that the sound reaches in the original and the encoded file (From 20kHz to aprox. 18kHz.)

Finally, using the DisplayInformation.m function that we have created we can also obtain information to evaluate our codec's performance. The function provides us with the following information:

```
Command Window
  Channel 1 SNR: 3.8788 dB
  Channel 2 SNR: 4.1301 dB
  Uncompressed audio size: 0.57817 MB
  Uncompressed audio bitrate: 172.6986 KB/s
  Compressed .mat file size: 0.11396 MB
  Compressed .mat file bitrate: 34.0388 KB/s
  Compression ratio: 80.29 %.
  The encoded file is 5.07 times smaller than the original file.
fx >>
```

We can observe that the codec has accomplished a compression ratio of 80.29%, meaning that the encoded audio file is more than 5 times smaller than the original audio file. (The input audio has a bitstream size of 0.578 MB and the EncodedAudio.mat compressed audio file has a bitstream size of only 0.114 MB).

Finally, if we take into account the huge compression ratio accomplished and the little amount of quality/sound fidelity lost, we can conclude that the results of our AAC alike coder and decoder are excellent.

4. Conclusions

In conclusion, we have had a very useful introduction to audio encoding and audio codecs, its relation to the crucial study of psychoacoustics and the idea of perceptual coding. We have been through the history of the digital audio representation and audio codecs and we have studied the particular case of the modern AAC codec structure and functioning. Finally, we have also been able to take an already existing implementation of AAC encoding and decoding methods in MATLAB and solve its issues, optimize it, improve it and extend its capabilities, obtaining a fully complete and functioning codec that portrays excellent results in terms of compression vs fidelity loss.

As a final note, this final project for Voice and Audio Encoding Systems has demanded much more time, work and effort from us than what we initially expected. When we first thought of implementing AAC, we would have never imagined how challenging it would be. Yet, this has been a highly interesting, educational and fun project to work on, and this motivation has been the key component for making this work possible. We are thankful for having had the opportunity to work on a relatively big project in a field that we liked and for having the opportunity to apply and interact with many of the contents we have seen during the course and also to learn many others.

5. References

Advanced Audio Coding - **https://es.wikipedia.org/wiki/Advanced_Audio_Coding**

Kaiser Window - **https://en.wikipedia.org/wiki/Kaiser_window**

Differential Pulse Code Modulation - **https://ca.wikipedia.org/wiki/DPCM**

Noise shaping - **https://en.wikipedia.org/wiki/Noise_shaping**

Transformada Discreta del Cosinus Modificat (TDCM - MDCT)
https://ca.wikipedia.org/wiki/Transformada_directa_del_cosinus_modificat

Lutzky, M.; Schuller, G.; Gayer, M; Krämer, U.; Wabnik, S.; *A Guideline to Audio Codec Delay.* Fraunhofer Institute for Integrated Circuits IIS and Fraunhofer Institute for Digital Media Technology IDMT. Retrieved from
https://www.iis.fraunhofer.de/content/dam/iis/de/doc/ame/conference/AES-116-Convention_guideline-to-audio-codec-delay_AES116.pdf

Kurniawati, E.; Lau, C.T.; Premkumar, B.; Absar, J.; George, S.; *New implementation techniques of an efficient MPEG advanced audio coder.* IEEE Transactions on Consumer Electronics, May 2004, Vol. 50(2), pp.655-665.

Sreenivas, T.V. ; Dietz, M. *Vector quantization of scale factors in advanced audio coder (AAC).* Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98, 0 1998, Vol. 6, pp.3641-3644

Chen, S; Xiong, N; Hyuk P., J.; Chen, M.; Hu, R.; *Spatial parameters for audio coding: MDCT domain analysis and sinthesis*. Multimedia Tools and Applications, 2010, Vol. 48(2), pp.225-246

Dimkoviae, I.; Milovanoviae, D.; Bojkoviae, Z.; *Fast software implementation of MPEG advanced audio coder*. 2002 14th International Conference Signal Processing Proceedings 2002, Vol. 2, pp.839-843