Music 422 Final Project Report: ROW.mp3 encoder
Colin Raffel, Jieun Oh, Isaac Wang

Introduction & Motivation

Since the mp3's introduction in the mid-90's, the coding format has become widely adopted and somewhat ubiquitous in the world of perceptual audio coders.[1] Despite its widespread use, the mp3 is frequently criticized for low audio quality, and many superior codecs have since been developed. Nevertheless, few have caught on because the mp3 is so widely accepted and implemented, and no new codec can be decoded by a system intended for mp3s. For our final project, we created a codec that is backwards-compatible with the mp3 while offering the potential for higher audio quality. Our codec allows for higher quality without requiring a massive immediate format shift.

Background

In order to improve the quality of the existing mp3 format, we created a coder that codes the error (difference) between the original, lossless audio file and the coded mp3. Ideally, if we simply found the difference between the uncoded and coded files and played it back simultaneously with the mp3, we'd have a lossless audio format. The difference between a lossless file's spectrum and a 64 kbps mp3's is illustrated in figure 1. Unfortunately, this is impractical as it is no better in terms of file size than just using the uncoded data to begin with. This approach has been explored by the mp3HD[2] format, but has not caught on due to the impracticability of dramatically larger files.

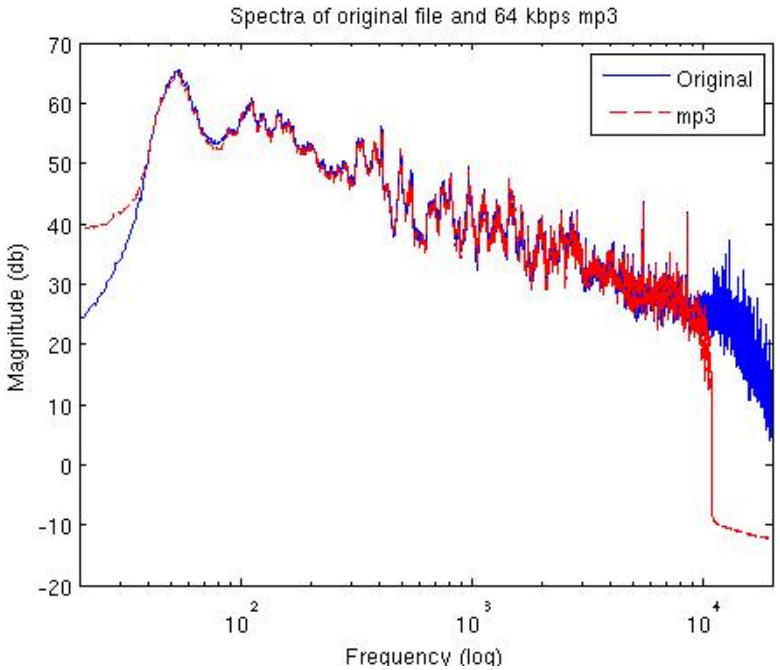

Figure 1: Example spectra of 16-bit PCM and 64 kbps mp3 files.

1. Borland, Jon. "MP3 losing steam?" CNET News, October 15, 2004. <http://news.cnet.com/MP3-losing-steam/2100-1027_3-5409604.html>
2. Hachman, Mark. "Next-Gen MP3 Format Launched by Thomson." PC Magazine, March 19, 2009.

By carefully studying the character of coder error, we decided to exploit features specific to the difference data, such as the fact that the error tends to be fairly noisy. By perceptually coding the error in a carefully chosen low data rate, the row-mp3 codec attempts to increase perceived audio quality without dramatically increasing file size. Ideally, a row-mp3 at a lower bit rate would sound perceptually equivalent to an mp3 at higher bit rate, but at a substantially smaller file size.

In order to store information about the content of an mp3 file, a special section for "metadata" is reserved in each mp3. Common metadata fields for a coded music file include the song's name, artist, and album. The metadata section is comprised of a series of fields, called ID3 tags, which are parsed by the decoder. Because the decoder will ignore fields that are unused, and because each ID3 tag can store up to 16 megabytes of data and have an arbitrary label,[3] the metadata section of the mp3 provides an ideal location for our coded error. With the additional information generated by our encoding process stored in ID3 tags, row-mp3 ignorant players will play the mp3 as usual while proper decoders will play a higher quality file.

Overview

In order to produce an error compensation, the row-mp3 encoder first generates an mp3 file at some specified bitrate from the lossless input file. To get the coding error, the time-domain values of the mp3 are subtracted from the input file. This difference is then separated into noisy (stochastic) and tonal (stationary) components. In order to take advantage of the perceptual characteristics of noise, the stochastic component is coded by determining the noise level in each of the 25 critical hearing bands. These noise level values are sent through a Huffman coder to reduce data rate. The tonal component is sent through an MDCT-based perceptual audio coder at a very low bit rate. The two resulting components of the error are then packed into the ID3 tags of the mp3 file, and the ROW.mp3 file is created, which is compatible with any mp3 decoder. A block diagram of the row-mp3 encoder is shown in figure 2.

To properly decode the ROW.mp3 file, the coded error information is first extracted from the ID3 tags. The Huffman-coded noise levels are first sent through the Huffman decoder to obtain the spectral weighting of the mp3 coding error's noise. These values are then used to synthesize a per-critical band noise signal. This signal is then summed with the PAC-decoded tonal component of the error and the mp3 information. A block diagram of the decoder is shown in figure 3.

---

3. Nilsson, Martin. "ID3 tag version 2.3.0." ID3 standards, Febuary 3, 1999. <http://www.id3.org/id3v2.3.0>

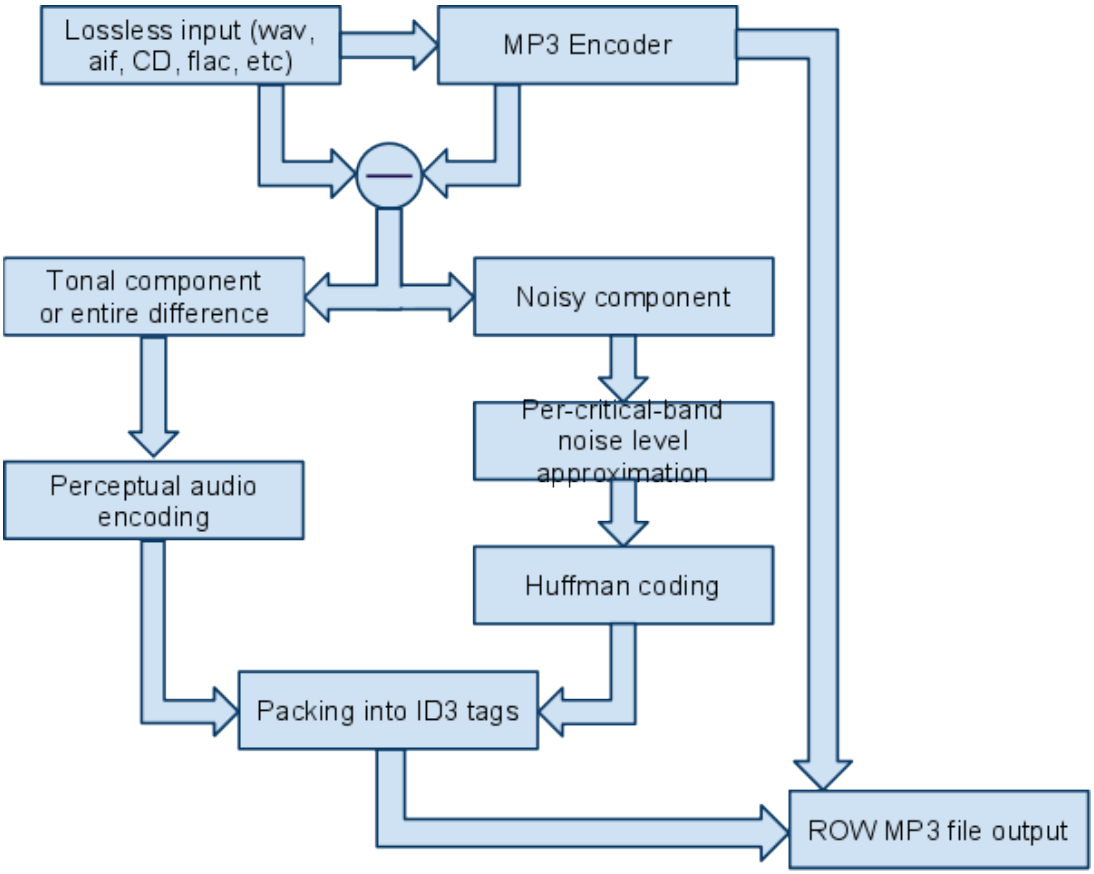

Figure 2: row-mp3 encoding process.

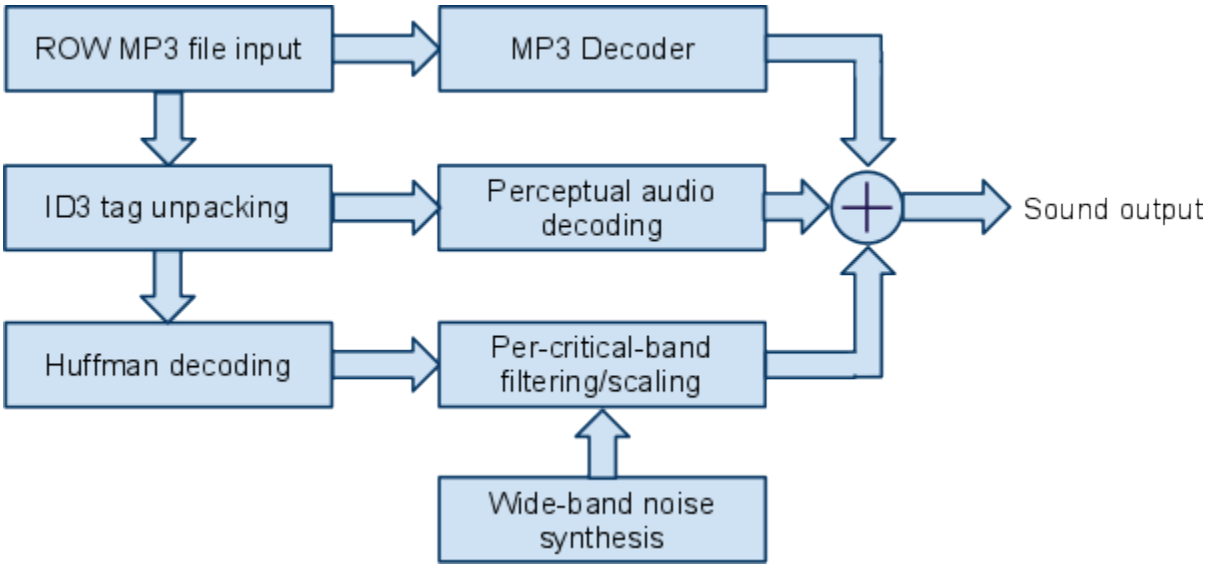

Figure 3: row-mp3 decoder block diagram.

Implementation

The major components of the row-mp3 encoder/decoder include (1) coding the noise level of the error in each critical band, (2) PAC-coding the remaining components of the error, (3) Huffman coding the noise level and the PAC-coded error, and (4) packaging our data in the mp3 header and aligning the values with the mp3 file in the decoding stage.

(1) Noise shaping

We quickly discovered that the error between mp3 and the original, lossless audio file tended to be highly stochastic (that is, generally non-tonal and non-stationary). Because humans cannot perceive the difference between blocks of noise with equivalently weighted spectra, we chose to do perceptual noise shaping on the coding error rather than attempting to perceptually code the noise directly.[4] This allowed us to simply code the level of noise in various parts of the spectrum, which provided drastic benefits in terms of bit rate without sacrificing substantial audio quality.

In order to exploit the human hearing model and to decrease bit rate, we chose to code the noise level in each critical hearing band. This gave a good approximation to the spectral weighting required to make the coded noise sound equivalent to the coding error's stochastic component. Because there are 25 critical bands, only 25 quantized values needed to be passed for each block of samples.

We found the noise level in each critical band by finding the flux for that portion of the spectrum. The spectral flux is defined as the difference between the magnitude spectrum of two successive frames. To convert the flux value in each critical band to a noise level, we used the scaled RMS level of the flux. In other words, the noise level in critical band i for the the N point spectrum n is given by

$$\text{Noise Level} = \frac{\sqrt{\sum_{\text{Band i}} (|X[n]| - |X[n-1]|)^2}}{N}$$

This gave a highly accurate value with which to scale portions of the spectrum when re-synthesizing the noise.

On the decoding end, we needed to take the 25 noise levels and turn them into a weighted noise spectrum. First, random complex numbers are generated for half of the necessary points in the spectrum. Then, a scaling mask is generated by linearly interpolating between the 25 noise levels. The interpolation helped increase sound quality by avoiding scaling discontinuities between adjacent critical bands. The random complex numbers are then scaled by the mask and duplicated to create a symmetric spectrum. The inverse Fourier transform of our noise-shaped spectrum is then found and output. Figure 4 confirms that the synthesized error spectrum closely matches the stochastic component of the overall error.

---

4. Schulz, Donald. "Improving Audio Codecs by Noise Substitution." JAES Volume 44 Issue 7/8, July, 1996.

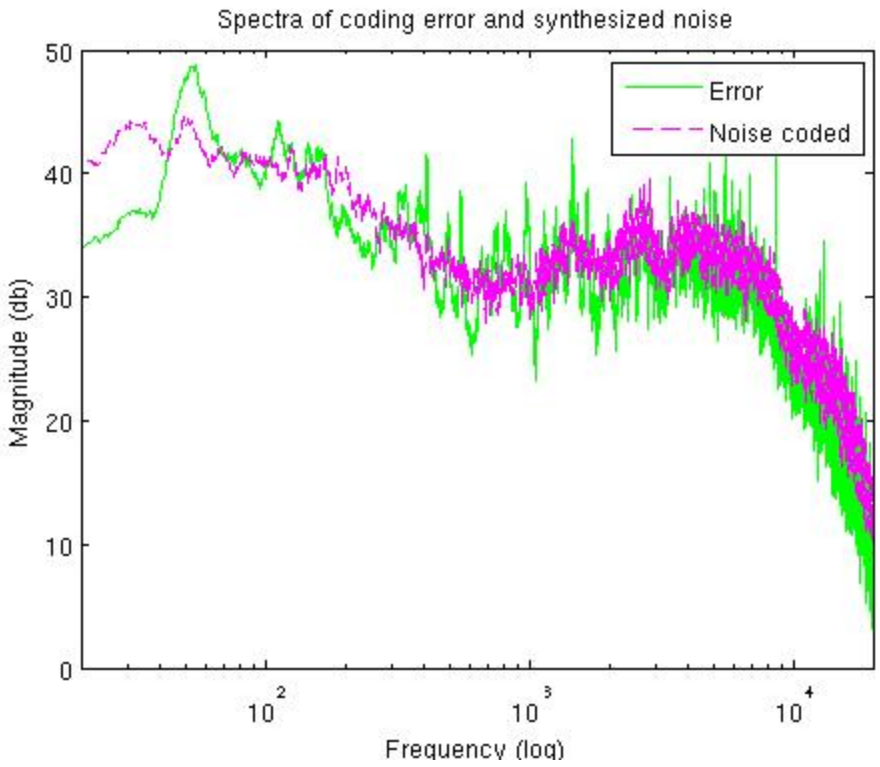

Figure 4: Spectra of mp3 coding error and the per-critical-band synthesized noise representation.

Because our noise coding scheme deals with samples on a per-block basis, a number of techniques were developed to ensure that the output had no obvious discontinuities. On the encoding side, a leaky integrator is used so that the noise levels in each critical band can be made to change slowly. The integration speed is parametrized so that an appropriate value can be chosen on a per-signal basis. On the decoding side, we quickly found that any immediate change in noise level resulted in clicks at each sample block boundary. To avoid this, we took advantage of the non-periodic nature of noise and implemented an overlap-add technique. For each incoming block of data, two blocks of shaped noise are synthesized. The double-sized block is windowed and overlap-added with the previous block. This prevents discontinuities and makes the resulting noise sound much more smooth.

The noise coding and decoding is done in `noisecode.py`. The extraction of noisy and tonal (described below) components is handled by the `noiseTonalSeparator` class. An instance of this class is initialized with the sample block size, sampling rate, and parameters which control the amount of leakiness in successive sets of noise levels. To analyze data, the class takes in a block of samples in the `tick` method. This function keeps track of the current and previous spectra and takes the FFT of the incoming data. To obtain a set of noise levels, the `noiseCode` function is called, which implements the method described above and returns a level for each of the 25 critical bands. To decode these values, the `noiseDecoder` class is used. This class is similarly initialized with the block size and sampling rate. Calling this class's `noiseDecode` function with a set of noiseLevels will return a weighted noisy spectrum overlapped and added with the previous spectrum.

(2) Non-Stochastic Error Coding

Although we are able to code the noise with high perceptual accuracy at very low cost, the noise representation is not effective at representing stationary components of the spectrum. To counteract this, we simultaneously create an error representation that focuses on the tonal aspects of the spectrum.

Ideally, each component of the spectrum that could not be represented in our noise coding could be coded in an efficient, perceptually-informed manner. Unfortunately, because the synthesized noise doesn't necessarily have any per-sample relation to the error in the the time or frequency domain, it is impossible to find an informative difference between the coding error and the noise representation. Instead, we chose to find the tonal component through a method that effectively looked for the components of the spectrum that are ignored by the flux-based representation described above. In this way, we can ideally find two separate components of the error that can be coded in ways that take advantage in their structure.

Because the stochastic component-finding algorithm searches for changes in the spectrum, we find our tonal component by looking for similarities between subsequent spectra. We first tried to simply test whether the value in each FFT bin was close to its previous value, but we were unable to obtain reliable results in this way. Instead, we first find the maximum and minimum values of the current and previous spectra. Then, we divide the minimum values by the maximum values to create a scale value for each bin. In this way, if the value in a bin does not change much between the two spectra, the scale value for that bin will be close to one. The scale values are then raised to some power in order to parametrically control the weighting of scale values. This causes scale values close to one to remain close to one, while smaller values can either be made much smaller when the power is much greater than one or much greater when the power is much smaller than one. Finally, the calculated scaled values are applied to the current spectrum, and the result is sent through an inverse Fourier transform to synthesize the tonal output. The output value for FFT bin n can be expressed based on the current spectrum X_0, previous spectrum X_{-1}, and power a by

$$X_{out}[n] = X_0[n]\left(\frac{\min(X_0[n], X_{-1}[n])}{\max(X_0[n], X_{-1}[n])}\right)^a$$

This parametrized form goes from a simple pass-though when $a = 0$ to an extreme weighting on tonal components when $a \gg 1$.

The main drawback of synthesizing the tonal component was dealing with the results of differences in subsequent calculated spectra. Because the scale factors could change dramatically and phase was rarely maintained, discontinuities between synthesized tonal blocks occurred. To remedy this, we implemented an overlap-add routine similar to the one used in our noise synthesizer. Unfortunately, because of the lack of randomness, the benefits were not as great for our tonal representation, and the resulting output still fluctuated between subsequent spectra in an audible way.

As mentioned above, the tonal component extraction is included in `noisecode.py`'s `noiseTonalSeparator` class. The `noiseTonalSeparator` class takes in the similarity scaling power at initialization, which is used when calculating the tonal component. After using the

`tick` method as described above, a block of tonal data can be obtained by calling the `getTonal` function. This method compares the current and previous spectra and returns the time-domain tonal component with some amount of user-settable leakiness.

After obtaining some form of the tonal representation of our error, we utilized a standard perceptual audio coder to decrease datarate. In order to obtain a sufficiently small file size increase for our row-mp3, we were only able to allocate a fraction of a bit per sample, which is at the very limits of what is possible in perceptual audio coding. However, because the tonal component had a relatively simple spectrum, the perceptual audio coder was able to effectively determine the necessary information even at extremely low bit rates. We experimented with various scale powers and found that the ideal value depended on the mp3 data rate as well as the characteristics of the input file. After many trials on different combination of parameters, we found that using 3 scale bits and 2 mantissa bits at average 0.2~0.3 bits per sample was the "sweet spot", where we were able to capture the most (including the high frequency components) for a small resulting PAC file size of 600~900KB for the original difference file of 42.4MB.

We also experimented with a lower threshold of hearing curve to see if that would result in our PAC capturing more high-frequency materials that the MP3 omitted, but we discovered that increasing the average bits per sample was much more important than lowering the threshold curve for capturing the high-frequency content. Also, we tried encoding the PAC using a smaller block size (512 MDCT lines) with hopes of reducing smeared transients, but found the frequency domain resolution and time domain resolution trade-off to be not convincing enough for this change, and ended up keeping the original block size of 1024.

(3) Huffman Coding & Decoding

In the lossless coding stage, Huffman coding could potentially be applied to (i) the noise level data and (ii) mantissas and scale factors when creating the PAC file. We decided to apply Huffman coding to the noise level data only, as the compression offered by Huffman coding was by far the most powerful for the noise level data, reducing the size by almost half of the original. This ratio assumes that we generate a Huffman table specific to each given sound file, so the compression ratio would not be nearly as powerful if we used the same table for different sound files. But since the huffman table is not very big, we decided to pass the table specifically generated for each audio file in the file's meta-data.

The module `huffmanCode.py` creates a Huffman binary tree given a list of dictionary data (symbol, frequency) pairs. Other useful methods, such as `Symbol2Code` and `Code2Symbol`, were written to create dictionaries for quick look-up of symbols and their corresponding Huffman codes. A huffman table is generated by creating an instance of the Huffman class: `h = Huffman(dicData)`, where `dicData` is a list of (key, value) pairs in the dictionary storing quantized symbols and their respective frequency of occurrence. To code and decode using the Huffman table created fromdicData, we call `h.Symbol2Code[_symbol]` and `h.Code2Symbol[_code]`, respectively, where `_symbol` is the quantized value and `_code` is the huffman code.

(3.1) noise level data

Noise level data consists of 25 floating-point numbers (representing the noise level for each critical band) per block of 1024 samples as described above in (1). We created a single Huffman table from the noise level data of the entire file, stored the table in the header, and used it to Huffman encode/ decode the mantissas of floating-point block-

quantized numbers using 3 scale bits and 5 mantissa bits. Huffman coding the noise level cut the file size storing the noise level data by about half, from approximately 2MB to 1MB.

(3.2) entire difference file

When we tried out Huffman coding on the entire difference file as it was being made into a PAC file, we did not divide the frequency spectrum into subregions (i.e. "big_values", "count1", and "rzero" regions, as is done in MPEG Layer III). The rationale was that the error file is quite noisy and has a much flatter spectrum than that of typical wave files. We simply took the entire spectrum and performed Huffman coding/decoding on the block of difference data, again using a single Huffman table generated from the entire file; that is, one table for the scale bits and another table for the mantissa bits were created from an entire difference file of a given song, with the intention of storing these tables in the header.

Huffman coding the mantissas also offered good compression, but for the scale factors it wasn't as useful: coding at 0.3bits/sample using 3 scale bits and 2 mantisas bits resulted in a mantissas coding that's about 70% of original and scale factors coding that's about 90% of the original. Though we ended up not including the Huffman coding for the mantissas or scale factors in our final ROW coder (for technical and time constraints), it is definitely a good addition to have for future work.

(4) Using the MP3 Header: Compute, Store, and Decode Error

In order to find the error between an mp3 file and it's wave file, we use both the LAME mp3 coder and mpg123 libraries. In our python module `mp3wav.py`, we are able to compare the wav file with it's mp3 coded file and accurately find the difference between the two. The main caveat with this comparison is that the LAME mp3 coder injects a variable delay of samples into the signal. To compensate for this, we simply calculate and strip out the silence at the beginning of the mp3 file, which corrects the offset. This difference is shown in figure 5.

Once we have the error, we use our PAC encoder to encode the error. However, unlike the normal PAC process, we do not create a PAC file but rather store the scale factors, mantissas, bit allocations, and overall scale factors as arrays of data.

We then use the ID3 tags of the mp3 to store both the Huffman Encoded noise/table and the PAC coded error data arrays. row.mp3s use ID3v2.x tags since ID3v1 tags are too small in file size and number[5].

The information itself is stored in the ID3 user text frames; whatever is written there (user URLs, etc.) may be replaced by the data, depending on how much information is already stored in the user URLs (since each frame can hold maximum 16 MB). ID3 tags can only store unicode strings as information, so we use the pickle python module in order to convert our data into strings and vice versa:

---

5. Nilsson, Martin. "ID3 tag version 2.3.0." ID3 standards, Febuary 3, 1999. <http://www.id3.org/id3v2.3.0>

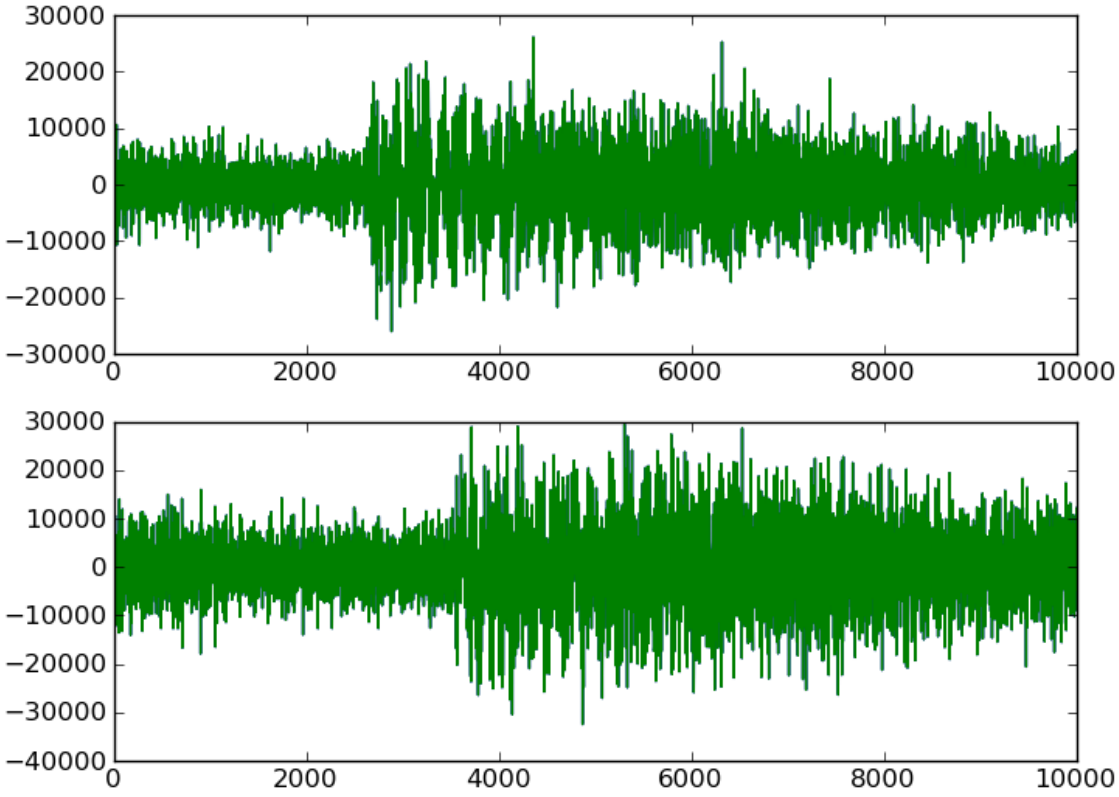

Figure 5: Offset from coding using LAME mp3 encoder.

```
import pickle

# Store data as a string
strData = pickle.dumps(data)

# Load the data back as a string
reloadedData = pickle.loads(strData)
```

In order to modify the ID3 tags, the eyeD3 python module is used (see **Running ROW.mp3**):

```
import eyeD3

# Create a tag to link to an mp3 file
tag = eyeD3.Tag()

# Link with mp3
tag.link('file/of/mp3')
```

```
# Set version of ID3 tag
tag.setVersion(eyeD3.ID3_V2)

# Add user text frame
tag.addUserTextFrame('description',pickle.dumps(data))
...

# Retrieve information from user text frame
for frame in tag.getUserTextFrames():
        if frame.description == 'description':
            reloadedData = pickle.loads(str(frame.text))
```

After the data is retrieved from the ID3 tags, it is passed back into the Decoder where the wav file is reconstructed from the PAC-coded error, noise envelope, and mp3 file itself. We then combine the mp3 signal, error signal, and noise envelope into one wav file as our reconstructed wav.

Running ROW.mp3

Requirements

The following libraries and tools must be installed in order to run ROW.mp3:

- **LAME**: http://sourceforge.net/projects/lame/files/lame/
- **mpg123**: http://www.mpg123.org/
- **eyeD3**: http://eyed3.nicfit.net/
- **scipy** version 0.8.0: (mp3wav's findDifference relies on this version of scipy)

Running

To code and decode in the ROW.mp3 format, one should download all the files for row-mp3 (which, in addition to the standard modules needed for perceptual audio coding, includes huffmanCode.py, mp3wav.py, noisecode.py, rowfile.py, and trainData.py). The wav file to be encoded and decoded should also be placed in the same folder.

Encoding a ROW.mp3 file can be done with the following command:

```
python rowfile.py INPUT.WAV OUTPUT BITRATE
```

Please note that the input file name must have ".wav" extension, while the output name excludes the extension.

Results

In order to evaluate the performance of our codec, we created a series of MUSHRA tests[6].

---

6. ITU-R recommendation BS.1534

Each test encodes a different style of music or instrument in the following formats:

- Reference file (lossless, 44.1khz 16 bit PCM)
- 3.5 khz low-pass filtered reference (as required by MUSHRA)
- 128 kbps mp3
- 128 kbps row-mp3
- 64 kbps mp3
- 64 kbps row-mp3
- 320 kbps mp3

To ensure that the row-mp3 was effective, we chose a wide variety of audio sources including recordings from the EBU SQAM disc as well as popular music recordings from a number of genres. The MUSHRA tests are also structured such that the test-taker gets immediate feedback as to what their choices were, making it a challenge as to whether they can perceive the differences between the reference and various audio codings. The test is also hosted online, so that it can be taken by as many people as possible. Our series of tests can be found here:

https://ccrma.stanford.edu/~craffel/mp3challenge/

Because the test is ongoing, we do not have definitive results as to the effectiveness of the row-mp3 codec, but results are quickly accumulating and reliable data will be available shortly. Preliminary results based on our own listening suggest that the row-mp3 coding is effective on wide-band, dynamic music, and in particular on low-bitrate mp3s. For example, we found the sound of a 64 kbps row-mp3 to be significantly better than a 64 kbps mp3, although not as perceptually close to the reference as the 128 kbps mp3. One major drawback of the row-mp3 is that it is very poor when dealing with transients. This is due to the fact that a transient will create a relatively flat frequency response when analyzed, so the row-mp3 will synthesize wide-band noise for a block of samples, which is hardly impulsive.

Conclusion

We have developed a perceptual audio codec that codes the error between a lossless audio file and a coded mp3 version. By storing the coded error information in the ID3 tag of the mp3, our codec is backwards-compatible with the mp3. In order for our codec to be effective, we ensured that the storage for coded error was very small in comparison to the mp3's file size. To obtain a very low bit rate, we exploited the noisy nature of the error and simply passed quantized, per-critical band noise level values. We also Huffman coded these values, as they tended to stay in a relatively small range, making the coding very effective. For the remainder of the error, we performed a basic tonal extraction and used a standard perceptual audio coder to decrease filesize. Preliminary results suggest that our data is effective at increasing the quality of low-bitrate mp3s with a very low bit cost. We created a large-scale MUSHRA test to more accurately determine the improvements made by ROW.mp3. With some potential improvements, the row-mp3 codec could provide a viable, backwards-compatible alternative to the highly pervasive mp3.

Future Work

Some ideas for future features and applications of the row-mp3 include:

- An intelligent algorithm which analyzes an mp3 file and predicts the error in absence of the original lossless file
- Noise synthesis in the time domain with a scaled filter bank rather than using random complex numbers in the frequency domain
- Block switching when extracting the noisy component to deal with poor coding of transients
- Direct coding of missing transients in the time domain
- A more intelligent tonal algorithm with better reconstruction in the time domain
- A perceptual audio codec for the tonal component which is especially well suited for low data rates and coding highly tonal sound
- Application of Huffman coding for the perceptual audio coder component to further reduce the file size