Table of Contents

Executive Summary

My project was to create a program to reduce the storage space requirement for audio data while having little perceivable impact on the output sound, so that portable devices supporting the format would be able to store more music in a comparable amount of memory than devices reading compression schemes such as MP3 and WMA. The program reads input from a file, splits it into units called frames, calculates the frequency spectrum of the sound, eliminates frequencies the program deems inaudible, scales the rest of the frequencies to take up half the space they otherwise would, and sends them to gzip, a lossless compression scheme, for further compression.

Introduction

Project
My project was to create a program to efficiently store audio data (such as music) by eliminating imperceivable qualities in the original recording.

Problem
The problem is choosing data that can be eliminated, as to produce an output that is perceived as high quality in spite of the reduced actual fidelity of the signal.

Reason for choosing
I chose this project because, owning a portable MP3/WMA player, I was interested in compressing songs in a way that would fit a couple hours of music on its 32MB of Flash memory. However, this requires a bitrate of 32kbps, certainly not enough for a stereo CD-quality sound. Even at 64kbps, CD-quality was impossible with the formats the device supported. MP3 would drop the upper pitches in a signal, and WMA's sound became "muddy" at anything below 96kbps, as well as introducing audible high-pitched noise into the output. This was less than satisfactory, so I decided to create a new format that would be able to store sound more efficiently.

Description

Scope
I decided the program would target near CD-quality (as defined by my ears and listening tests on a sampling of the general population) for a single-channel signal at 24-32kbps. The program does not currently handle stereo signals or correctly handle samplerates other than 44100Hz. It reads signed 16-bit native-endianness PCM data (Pulse Code Modulation—samples are taken at a set rate, and the amplitude of the sound at each point is recorded and stored as a number—in this case, a 16-bit signed integer) with no header, and writes an output file in its own format. The decoder reverses the process, writing raw PCM as output.

Framing
The first step of the compression process is to split the input PCM data into frames. In choosing the frame size, there is a trade-off between frequency resolution and ease of controlling echo effects because of limited time resolution. The frame size used is 16384 samples, chosen because it allowed enough frequency resolution to make eliminating the clicking noises generated between frames much easier.

Fourier Transform
The next step is to transform this PCM data into the frequency domain, as this makes compressing based on features of human perception possible. This transform is accomplished via the Fourier Transform, which outputs the level at which various frequencies are present within a frame. The equation for the DFT (Discreet Fourier Transform, the discreet version of the Fourier Transform) is:

$$\sum_{n=0}^{N-1} P(n) e^{\frac{-2\pi n x i}{N}}$$ [1]

N is the number of samples in each frame. P(n) is the PCM data at sample n within a frame. x is the frequency for which the function is being evaluated. The reverse transform is very similar:

$$\sum_{n=0}^{N-1} F(n) e^{\frac{2\pi n x i}{N}}$$ [1]

F(n) is the coeffient for frequency n (on a scale determined by the number of samples in each frame and the samplerate) and x describes the point on the output waveform whose amplitude is being computed. The transform is produced by FFTW (the Fastest Fourier Transform in the West), as it provided an efficient implementation of the DFT.

Magnitude calculation
The data returned by the FFT is still unsuitable for further processing, as it provides a complex coefficient. However, magnitude and phase information can be extracted from the FFT output as follows:

$$m = \sqrt{r^2 + a^2}$$

$$p = \tan^{-1}(\frac{a^2}{r^2})$$

where m is the magnitude, p is the phase, r is the real part of the FFT output, and a is the imaginary part. Only the magnitude is used by the program, as the output is a compressed form of the FFT output.

Masking calculation

There are two forms of masking the program uses to determine frequencies that are safe to eliminate. The first kind is threshold masking, which eliminates frequencies that the program deems inaudible. The second type uses the fact that quieter sounds similar in pitch to a louder sound are masked by the louder sound. Several sets of equations were tried to see what produced the best results. The current expression to calculate the mask is:

$$(1.20107 \cdot (\ln(|\ln(Nx)| - 10.0011)) + \ln(|\ln(Nx)| - 18.2951)) - 1.93618)) \cdot tr + .01(x - y)t - t$$

where N is the number of frequencies output by the FFT, x is the logarithmically scaled magnitude for the frequency in question, y is the logarithmically scaled convoluted magnitude for the frequency in question, t is a user-specified multiplier, and r is the RMS (Root Mean Square) of the magnitude of the frame. The convolution is performed by transforming a spreading function using the FFT, multiplying the FFT output of the magnitude by it, and then transforming back using the IFFT (Inverse Fast Fourier Transform). The RMS is the square root of the mean of the squares of the magnitudes ($\sqrt{\bar{x}}$). This implements both forms of masking in one expression, producing an array of minimum amplitudes each frequency must have in order to be included in the output.

Application of mask

The application of this combined mask is relatively simple. The program simply compares each magnitude with the mask, and, if the magnitude is less than the mask, both the real and imaginary portions of the respective points on the FFT output are zeroed.

Quantization

The FFT coefficients are stored on an 8-bit exponiential scale. This allows for more accurate representation of lower volumes than higher ones, thus acheiving a 2:1 compression from this step alone, with no perceivable quality loss.

Lossless compression

As the final step in compression, the data is sent to zlib to be compressed using the lossless gzip compression scheme.

Results

Compression ratio/quality comparison

The program currently acheives 8:1 compression with almost no perceivable quality loss. This is, however, short of my goal. Some sets of equations have produced as much as 40:1 with lesser but still almost acceptable quality, so I'm working on merging the features of several sets to acheive better compression.

Conclusion

Max useable compression for purpose

As determined by listening tests, for the purpose for which this project was created, 12:1 compression (64kbps) is currently the maximum useable compression without significant quality loss.

Recommendations

Implement tonality measure

Tonality is the measure of how long each frequency persists in the sound file. A measure of how tonal signals are within a frame would allow for more intelligent masking, as tonal signals are hard to mask but mask other signals well.

Implement variable quantization

Quantization is the process of reducing the accuracy of a value to reduce the number of bits it takes up in storage. Currently, the quantization scheme allocates a static number of bits to each coefficient. This should be improved by allowing for bands where different numbers of bits are used, giving more attention to important frequencies.

Pack output before lossless compression

The program currently passes on eliminated frequencies (replaced with 0s) to the lossless compression. Fixing this would improve the compression ratio with no quality loss.

Add stereo support

The program currently only works on monaural signals. With a stereo signal, it could also take advantage of redundancy between channels to further compress the signal, as well as allow for a better listening experience.

Handle Range Limitations

The range on PCM samples is –32768 to 32768. When a result of a calculation outputting PCM data goes out of range, the computer will start wrapping back to the opposite end of the range, causing audible clicks in these areas. To compensate, the program detects these out of range values before they are converted to 16-bit integers and snaps them to the nearest extreme. However, this is not done for the FFT output as it is quantized, making it possible to have the same problem in the frequency domain.

Lack of Scientific Data

The testing done on this program has been entirely subjective. Objective tests are only partially useful, as the goal is to produce output that is percieved by the human listener as high quality, but objective testing, such as measuring output frequency response, should be done.

Bibliography

[1] "FFTW Reference", http://www.fftw.org/doc/fftw_3.html