Investigating Data Compression of Sound Files

Sarah Edge Mann and Priya V. Prasad

December 18, 2009

Abstract

The fast advances in technology has allowed us to capture digital media which contains vast amounts of information, but accordingly requires vast amounts of storage space. There are a wide array of algorithms that will take a raw digital sound file and write it in a different format using less space. For this RTG project the authors researched and implemented a lossy compression algorithm for sound files based on the discrete cosine transform. In this paper we will discuss some key principles of sound, digital sound representation, and compression theory as well as the lossy compression algorithms we developed.

Contents

List of Figures

List of Tables

1 Introduction to Sound Compression

In this age of digital media, we have the ability to digitally capture sound and images at astonishingly high fidelity. However, this high-fidelity digital media can require such vast amounts of storage space that it is prohibitive for wide spread use. For example, the raw digital recording of a three minute song might have a file size 29.5 MB.[1] A 2GB Apple iPod Shuffle could hold a mere 68 songs of this size, whereas Apple claims it can hold 500 songs. The reason for this discrepancy is that Apple assumes that the sound files put on the iPod will be in a compressed format and will thus require only about 4 MB of space per song. Similarly, 15 terabytes are required to capture a mere 15 minutes of raw digital video. This is the equivalent of 250 DVDs worth of data, and yet, in practice, we are able to put an entire 2 hour movie on a single DVD. Again, this feat is achieved through data compression.

There are a wide array of algorithms that will take a raw digital sound file and write it in a different format using less space. MP3 and AAC files are examples of sound files that have been digitally compressed. There are two categories of algorithms that compress sound (or any other) files. The first is called lossless compression: the compression algorithm attempts to represent the file in a more efficient way without losing any information. When a file is compressed using a lossless compression algorithm, it may be decompressed to give the exact original file. The second type of algorithm is called lossy compression: the compression algorithm creates a smaller file, but in so doing loses some information. When a file is compressed using a lossy compression algorithm, it may not be decompressed to give the exact original file, however it may come close[2] to achieving this property.

For this RTG project the authors researched and implemented a lossy compression algorithm for sound files based on the discrete cosine transform[3]. In this paper we will discuss some key principles of sound, digital sound representation, and compression theory as well as the lossy compression algorithm we developed.

[1] We assume 16-bit stereo encoding, with a sampling frequency of 44100 Hz. See Section 2.2 for further discussion of file structure and size.

[2] See Section 3.3 for a more precise discussion of "close".

[3] The discrete cosine transform forms the basis of many popular compression algorithms such as the MPEG2 video format. MPEG2 is the compression algorithm used for digital television.

2 Sound

2.1 The Physics of Sound

Physically, sound is a traveling wave of pressure through a medium. Although most noise we encounter is quite complex, it is easy to mathematically describe a sound wave of a single frequency as

$$S(t) = A\cos(2\pi ft + \phi), \tag{1}$$

where t is time, A is the amplitude of the sound wave corresponding to the volume of the sound, f is the sound frequency corresponding to the pitch of the note, and ϕ is a phase shift that has no relevant physical meaning.

2.2 Digital Sound Recording

Before we can begin to discuss compression, it is important to understand how sound is recorded digitally and the structure of a raw sound file. During recording, a microphone is set near the source sound. The air pressure inside the microphone is then sampled many times per second and recorded as a fixed point number. For *stereo* recordings, the sound is sampled at two positions in space, corresponding to a right and left sound channel, and the sampled amplitudes are stored separately.

The number of times per second the sound is sampled is called the *sampling frequency*. The *precision* of the recording refers to the allowed dynamic range of the pressure readings. For example, if the air pressure at each sample is stored as an 8-bit signed floating point number, then there would be 2^8 possible values that a sampled pressure could attain. If we instead use a 16-bit signed floating point number to represent the sampled pressure, then we would have a dynamic range of 2^{16}.

The more bits we use to represent the pressure, and the higher the sampling frequency used, the better quality a digital recording we produce. However, we pay for this improvement in sound quality with an increase in file size. In fact,

$$\text{file size} = \frac{F_s \times t \times d}{8} \text{ bytes},$$

where F_s is the sampling frequency, t is the length of the recorded sound in seconds, and the amplitude is recorded using d-bits. We divide by 8 to convert from bits to bytes (1 byte = 8 bits) since file size is typically measured in bytes.

The example sound files used in this project were typically recorded with a sampling frequency of F_s = 48,000 Hz or 44,100 Hz, using 16-bit signed floating point numbers.

3 Data Compression

Before considering how to compress sound files specifically, we will discuss some basic principles of data compression. Data compression consists two components: statistical modeling and source coding[3], usually done in that order. However, we will discuss source coding first, because it motivates some of the choices we made in terms of statistical modeling.

3.1 Source Coding

General purpose source coding algorithms typically look for patterns in the input data and exploit these patterns to represent the information more compactly.

If we consider a file X as a sequence of characters from some alphabet $\mathcal{A}$, such that $X = \{x_i\}_{i=1}^{n}$, $x_i \in \mathcal{A}$, then we may define

$$p(x_i) = \frac{|\{j : x_j = x_i, x_j \in \mathcal{A}\}|}{|X|}$$

and think of $p(x_i)$ as the frequency with which the character x_i occurs in the file X. Given these frequencies for an input file, we might then compress the file by denoting characters or sequences of characters that occur frequently using a small number of bits, and use a relatively large number of bits to denote characters or sequences of characters that occur relatively infrequently. In a raw sound file, each character in the file corresponds to a sampled sound amplitude, and the alphabet can be represented as a d-bit signed fixed point number.

3.1.1 Entropy

The *entropy* H of a file X is a measure of the amount of information that the file contains and is defined by

$$H(X) = -\sum_{i=1}^{m} p(x_i) \log_2 p(x_i)$$

where m is the number of characters in the alphabet used. Entropy is measured in bits and can be considered to be the "information rate" in bits per sample. Shannon's source coding theorem states that regardless of the algorithm used, we cannot represent an input file X in less than $|X| \cdot H(X)$ bits[2], where $|X|$ is the cardinality of X. This gives an explicit lower bound on how much compression of a file is possible.

Let us now calculate the entropy for a few example source files. First, consider a file X whose characters are drawn from some alphabet $\mathcal{A}$ with $|\mathcal{A}| = m$ such that the frequencies of all the characters in the alphabet are the same, that is, $p(x) = \frac{1}{m}$ for all $x \in \mathcal{A}$. Then the entropy of X is simply

$$H(X) = -\sum_{i=1}^{m} \frac{1}{m} \log_2 \left(\frac{1}{m}\right) = \log_2 m.$$

It turns out that for a given alphabet, a perfectly even frequency distribution in the file yields the worst case entropy, the largest entropy possible for that alphabet.

Files with uneven character distributions have lower entropies. For example, consider the alphabet $\mathcal{A} = \{A, B, C, D, E\}$ and some file X consisting of characters from this alphabet. If the probability of each letter occurring is equal, $p(x) = 1/5$ $\forall x \in \mathcal{A}$, then the entropy then file is

$$H(X) = \log_2 5 \approx 2.322.$$

However, if instead there is some variability in the frequencies, say $p(A) = 0.01$, $p(B) = 0.05$, $p(C) = 0.8$, $p(D) = 0.04$, and $p(E) = 0.1$, then entropy would be

$$H(X) \approx 1.058.$$

The entropy for this uneven distribution is less than half the entropy of the worst-case even distribution.

This observation leads us to another property of entropy: the overall entropy of a file may be higher than any of the individual entropies of partitions of the file. Consider the following two distributions of the alphabet $\{A, B\}$ found in two files X_1 and X_2:

$$\begin{aligned} \text{Distribution in } X_1\text{: } P(A) &= 0.25 \\ P(B) &= 0.75 \\ H(X_1) &\approx 0.36 \\ \text{Distribution in } X_2\text{: } P(A) &= 0.85 \\ P(B) &= 0.15 \\ H(X_2) &\approx 0.61 \end{aligned}$$

Now consider a file $X = X_1 + X_2$ (X_2 concatenated to the end of X_1). This new file has the following distributions:

$$\begin{aligned} \text{Distribution: } P(A) &= 0.55 \\ P(B) &= 0.45 \\ H(X) &\approx 0.99. \end{aligned}$$

Notice that putting the two files together causes our distribution to approach the worst-case even distribution (which would be $P(A) = P(B) = 0.5$). We will see in section 3.1.2 that there exists a source coding algorithm that can encode a file Y using almost as few as $|Y| \cdot H(Y)$ bits, that is, it encodes the file "at entropy." If we were to use such an algorithm to encode the file X we would achieve better compression by first encoding X_1, then encoding X_2 than if we encoded all of X together:

$$|X_1| \cdot H(X_1) + |X_2| \cdot H(X_2) = |X_1| \cdot 0.36 + |X_2| \cdot 0.61 < |X_1 + X_2| \cdot 0.99 = |X|H(X).$$

Thus, when using an algorithm that encodes at entropy, it will sometimes be effective to encode portions of the file separately rather than encoding the whole file at once. This insight will be exploited in Section 6.

Notice that in a raw sound file, since we use d bits to represent every character (pressure sample), so we might naively assume that the entropy would be d. However, the entropy of the file would typically be somewhat smaller than d because the frequencies of various characters in our alphabet are not distributed perfectly evenly.

3.1.2 Arithmetic Encoding

Arithmetic encoding is an optimal source coding algorithm; it encodes a file at, or at least very close to, entropy. That is, it encodes a source file X using almost exactly $|X| \cdot H(X)$ bits. In practice, arithmetic encoding adds an overhead of a few bits above entropy to the message. However, for a message of any length, this overhead is very low.

Arithmetic encoding represents the entire file X as a single real number in the interval $(0, 1)$. Consider the following example with alphabet $\mathcal{A} = \{A, B, C, D\}$. Suppose that in our message, these symbols occur with probabilities

$$\begin{aligned} P(A) &= 0.20 \\ P(B) &= 0.35 \\ P(C) &= 0.15 \\ P(D) &= 0.30. \end{aligned}$$

To encode arithmetically, we take the interval $[0, 1)$ and divide it into four sections, corresponding to the probabilities of each of the symbols. To encode the message BDC, we look first inside the interval that corresponds to the first symbol, B, which is $I_0 = [0.2, 0.55)$ (see Figure 1). Then, from there, we further subdivide I_0 into subintervals corresponding to our original four probabilities, that is, into the intervals

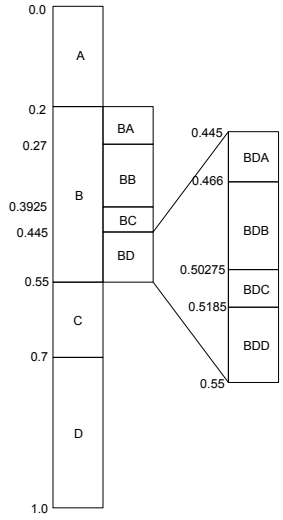

Figure 1: Intervals for Arithmetic Encoding

$I_1 = [0.2, 0.27)$, $I_2 = [0.27, 0.3925)$, $I_3 = [0.3925, 0.445)$ and $I_4 = [0.445, 0.55)$. Note that the interval I_1 is the first 20% of the interval I_0, the interval I_2 is the second 35%, etcetera. So, if we were to assign a number to our message so far, BD, it would be a number in the interval $I_4 = [0.445, 0.55)$. However, since our message is three symbols long, we have to further subdivide I_4 as we did earlier, and now we know that our message BDC should be in the interval $[0.50275, 0.5185)$. If we transmit any number in this interval, then we will be able to decode the message by simply reversing this process, so long as we transmit the probability distributions as well.

The version of arithmetic encoding presented above was developed by Claude Shannon and Robert Fano in 1948. It can be considered the "ideal" version of arithmetic encoding in that assumes that we have the use of infinite precision arithmetic. However, computers actually use finite precision arithmetic and represent numbers in binary not decimal. This gives rise to a few technical challenges to implementing the arithmetic encoding algorithm. In the 1970's, J.J. Rissanen had a breakthrough concerning the use of finite precision arithmetic in arithmetic encoding. This breakthrough allowed IBM to develop the first effective implementation of the arithmetic encoding algorithm shortly thereafter. MATLAB's `arithenco.m` file implements such an arithmetic encoding algorithm similar to the one developed to by IBM, and it is this algorithm that we will implicit use throughout this project[4].

The use of finite precision arithmetic allows for the streaming of a message in real time. That is, it is not necessary to know the entire original message before

the encoder can encode it and send the first binary digits of the encoded message. We can see this in our example above as well. Let us assume that the message we considered before, BDC, is actually the beginning of a larger message, say $BDCADACDBDDB$. The first symbol is B, which corresponds to the interval $I_0 = [0.2, 0.55)$, so if the encoder wants to transmit the first digit of the encoded message, it needs to know if that first digit is 2, 3, 4, or 5. Therefore, it narrows these options by looking at the next symbol, which is D, and since the message BD is encoded in the interval $I_4 = [0.445, 0.55)$, the encoder knows that the first digit sent is going to be either a 4 or a 5. But which is it? To find this out, the encoder must narrow the interval down more by looking at the third symbol, C, which tells it that, so far, the message is encoded in the interval $[0.50275, 0.5185)$, which then allows the encoder to send the first digit: 5. Thus, the encoder can start encoding and transmitting the message without having seen the entire string of symbols. Notice that this is not possible in infinite precision since due to the lack of unique representation of numbers in decimal; for example, $0.1\overline{9} = 0.2$.

Because arithmetic encoding is such an efficient source coding algorithm, we will use it as the last step in all compression algorithms presented. Keep in mind that we will need to create histograms of the alphabet of our sound files in order to perform arithmetic encoding, and it will be necessary to transmit these histograms along with the encoded data so that we may subsequently arithmetically decode. These histograms are part of the *metadata* of the file. Metadata is any information that must be transmitted to allow the file to be decompressed accurately. When we calculate the size of our compressed files, we will have to include the size of the necessary metadata.

3.2 Statistical Modeling

The first component of compression is statistical modeling. Since we have an algorithm that does source coding at entropy, the purpose of statistical modeling will be to reduce the entropy of our file so that we achieve as much compression as possible from arithmetic encoding. We will accomplish this using the discrete cosine transform (Section 4), uniform quantization (Section 5), and K-means clustering (Section 6).

3.3 SNR

All algorithms in this paper will be lossy compression algorithms. That is, we expect that upon decompressing our compressed signal, we will have something different

from our original signal. The question is how different and how do measure it? How do we measure the quality of the sound signal we transmit?

We will use *signal-to-noise ratio*, or *SNR*, as a measure of how close the transmitted signal is to the original signal. SNR is defined as

$$\text{SNR} = 20 \cdot \log_{10} \frac{\|\text{signal}\|_2}{\|\text{error}\|_2}, \tag{2}$$

where the signal is the original input signal (not the transmitted signal) and has units of decibels. The higher the SNR of a file, the smaller the error is, relative to the size of the source file. A file that was losslessly compressed would have zero error, and thus its SNR would be infinite. Lossy compression algorithms will have some finite SNR. We will assume that an SNR of 40 dB or higher is good enough for most applications, and an SNR under 20 dB indicates that there is too much error for most uses.

Another, similar measure of sound quality is the *peak SNR*, or *PSNR*. This is defined as

$$\text{PSNR} = 20 \log_{10} \left(\frac{\max}{\text{MSE}} \right), \tag{3}$$

where

$$\text{MSE} = \frac{1}{n} \|\text{error}\|^2$$

is the mean-square error, and max is the maximum possible value of the signal input. If the sound was recorded in d bits, then $\max = 2^k - 1$.

4 Cosine Transforms

As we mentioned before, the discrete cosine transform drives the compression schemes that we built. We will now develop an understanding of why this transform is so useful.

4.1 Fourier Cosine Series

Noticing that the set of functions $\cos\left(\frac{n\pi x}{l}\right)$ forms an orthonormal basis for the space of continuous functions on a finite interval $(0, l)$, allows us to write any function in this space in the basis of cosines. This decomposition is called the *Fourier cosine series* and is given by

$$f(x) = \frac{1}{2} A_0 + \sum_{m=1}^{\infty} A_m \cos\left(\frac{m\pi x}{l}\right),$$

where

$$A_m = \frac{2}{l}\int_0^l f(x)\cos\left(\frac{m\pi x}{l}\right)\,dx.[5]$$

The coefficients A_m indicate how much of the function f is captured by $\cos\left(\frac{m\pi x}{l}\right)$.

The Riemann-Lebesgue lemma states that if f is a continuous function, then its Fourier coefficients A_m have the property

$$\lim_{m\to\infty} A_m = 0.[1]$$

Therefore it is typically possibly to approximate continuous functions on finite intervals with high accuracy only using the first few terms of the Fourier series:

$$f(x) \approx \frac{1}{2}A_0 + \sum_{m=1}^{N} A_m \cos\left(\frac{n\pi x}{l}\right).$$

4.2 Discrete Cosine Transform

The Fourier cosine series is an effective method of decomposing a function if we have an analytic expression for that function. However, in many engineering applications, the function is not known explicitly, but instead we have some samples of values of this function, as is the case for this project. We would like to consider the Fourier series of the sound file, but we do not have an expression for sound pressure as a function of time. Instead, we have samples of the pressure at many different times. Thus, instead of writing a Fourier cosine series for the sound "function," we will perform a *discrete cosine transform* (DCT) on the sampled values.

The DCT for an input vector $\mathbf{x} = \{x_n\}_{n=1}^N$ of sample points is given by

$$y_1 = \frac{1}{\sqrt{N}}\sum_{n=1}^{N} x_n \tag{4}$$

$$y_m = \sqrt{\frac{2}{N}}\sum_{n=1}^{N} x_n \cos\left(\frac{\pi(2n-1)(m-1)}{2N}\right) \text{ if } 2 \le m \le N. \tag{5}$$

The vector $\mathbf{y} = \{y_m\}_{k=1}^N$ computed in the DCT is analogous to the coefficients A_m calculated in Fourier cosine series. Given $\mathbf{y}$, $\mathbf{x}$ can be reconstituted through the formula

$$x_n = \frac{1}{\sqrt{N}}y_1 + \sqrt{\frac{2}{N}}\sum_{m=2}^{N} y_m \cos\left(\frac{\pi(2n-1)(m-1)}{2N}\right), \text{ where } 1 \le n \le N. \tag{6}$$

This is called the *inverse discrete cosine transform* (IDCT).[4]

The DCT and IDCT are implemented in MATLAB as `dct` and `idct` respectively and are based of the fast fourier transform (FFT). Whereas the naive implementation of a discrete Fourier transform has a running time that is $O(N^2)$, where N is the length of the input vector, the FFT has a running time that is $O(N \log N)$. This dramatic improvement in algorithm running time allows discrete Fourier transforms to be useful in practice. However, since the FFT is a divide and conquer algorithm, the DCT and IDCT both run much faster on input vectors of length 2^k, with k an integer, than on vectors of other lengths.

4.3 The DCT of a Single Note

As a toy example, let us consider the case where the sound file we wish to compress represents a single note of frequency f played for some short time. Recall from equation (1) that the amplitude of this note as function of time is given by

$$S(t) = A\cos(2\pi f t + \phi)$$

Assume that in recording this note was sampled with frequency f_s. Then the sound file contains a vector of amplitudes $\{x_n\}$ defined by

$$x_n = A\cos\left(2\pi\frac{f}{f_s}n + \phi\right).$$

The DCT writes $\{x_n\}$ in the basis of functions

$$g_m(n) = \cos\left(\frac{\pi(2n-1)(m-1)}{2N}\right) = \cos\left(\frac{\pi(m-1)n}{N} + \frac{\pi(1-m)}{2N}\right).$$

If

$$2\pi\frac{f}{f_s} = \frac{\pi(m-1)}{N} \qquad \text{and} \qquad \phi = \frac{\pi(1-m)}{2N}$$

for some m, then $\{x_n\}$ is composed of exactly one basis function and $\{y_m\}$, the DCT of $\{x_n\}$, is given by

$$y_{m'} = \begin{cases} 0 & m' \neq m \\ A & m' = m. \end{cases}$$

[4]There are actually a few different forms of the DCT and IDCT that are commonly used. For convenience, we have chosen to discuss the form used in the MATLAB implementation of the DCT and IDCT.

This also means that for fixed f_s and N, each basis function g_m corresponds to a single note of frequency

$$f_m = \frac{f_s(m-1)}{2N}.$$

When we take the DCT of a sound vector, we are writing the sound as a superposition of single notes of frequency f_m where the amplitudes of each of these notes is given by y_m. Table 1 gives numerical values of f_m for various m as well as the approximate corresponding note in the twelve-tone equal-tempered scale with $N = 2^8$ and $f_s = 44,100$ Hz. Notice that only the first 32 basis functions correspond to notes that can be played on a piano, and that basis functions beyond about 175 are so high-pitch that they are either beyond the range of human hearing or close to it.

4.4 The DCT of Compound Sounds

In the previous section we discussed the special case where we explored the DCT of a single note. Building on this understanding, we will now consider more complicated forms of sound and specifically examine the DCT of two examples: the first twenty seconds of Pachelbel's Canon and a seven-second clip of an interview recorded by Arizona Public Media. Before we begin to analyze these samples, we should first notice that our input data has a dynamic range of $[-2^{d-1}, 2^{d-1})$, since each pressure sample is represented as a d-bit integer. We break the input sound vector into *packets* containing $N = 256$ samples; each of these packets captures about 0.005 seconds of sound. We then calculate the DCT of each packet.

Figure 2 shows a profile of the DCT coefficients for these two sample files. The x-axis gives the coefficient number m, and the y-axis shows the mean of the absolute value of the DCT coefficient y_m across all packets. Although the shapes and maximum heights of the graphs for the two examples are somewhat different, they share some qualitative similarities that are characteristic of the DCT coefficients for any sound file. In both, for small coefficient numbers, the mean coefficient value is relatively large, and the coefficient values rapidly decrease as m increases. Most of the energy of the sound file is captured in the first 16 to 32 DCT coefficients. This corresponds nicely to ideas discussed in section 4.3. Recall that we showed that performing a DCT is equivalent to writing the sound data as a superposition of single notes. These graphs show that the notes that are most heavily weighted in this superposition are of low frequency and correspond to notes in the middle range of a piano. Notes of very high frequency barely contribute at all. These observations will motivate our approach to sound compression for the rest of this paper; we will work hard to preserve the information about the first few DCT coefficients, and readily

m	f_m	note (Freq)
1	0.0	
2	86.1	F_2 (87.3)
3	172.3	F_3 (174.6)
4	258.4	C_4 (261.6)
5	344.5	F_4 (349.2)
6	430.7	A_4 (440.0)
7	516.8	C_5 (523.2)
8	602.9	
9	689.1	
10	775.2	G_5 (784.0)
11	861.3	
12	947.5	
13	1,033.6	C_6 (1,046.5)
14	1,119.7	
15	1,205.9	
16	1,292.0	E_6 (1,318.5)
⋮	⋮	⋮
32	2,670.1	E_7 (2,637.0)
55	4,651.2	D_8 (4.698.6)
100	8,527.1	
175	14,987.1†	
256	21,963.9	

Table 1: DCT Frequencies and their corresponding notes for $f_s = 44,100$ Hz, $N = 2^8$. The first column indicates the DCT number, the second the frequency of the DCT basis function, and the third the corresponding note in the twelve-tone equal-tempered scale. C_4 is middle C.

†This is the highest frequency that a personal computer would play and could be heard with the human ear.

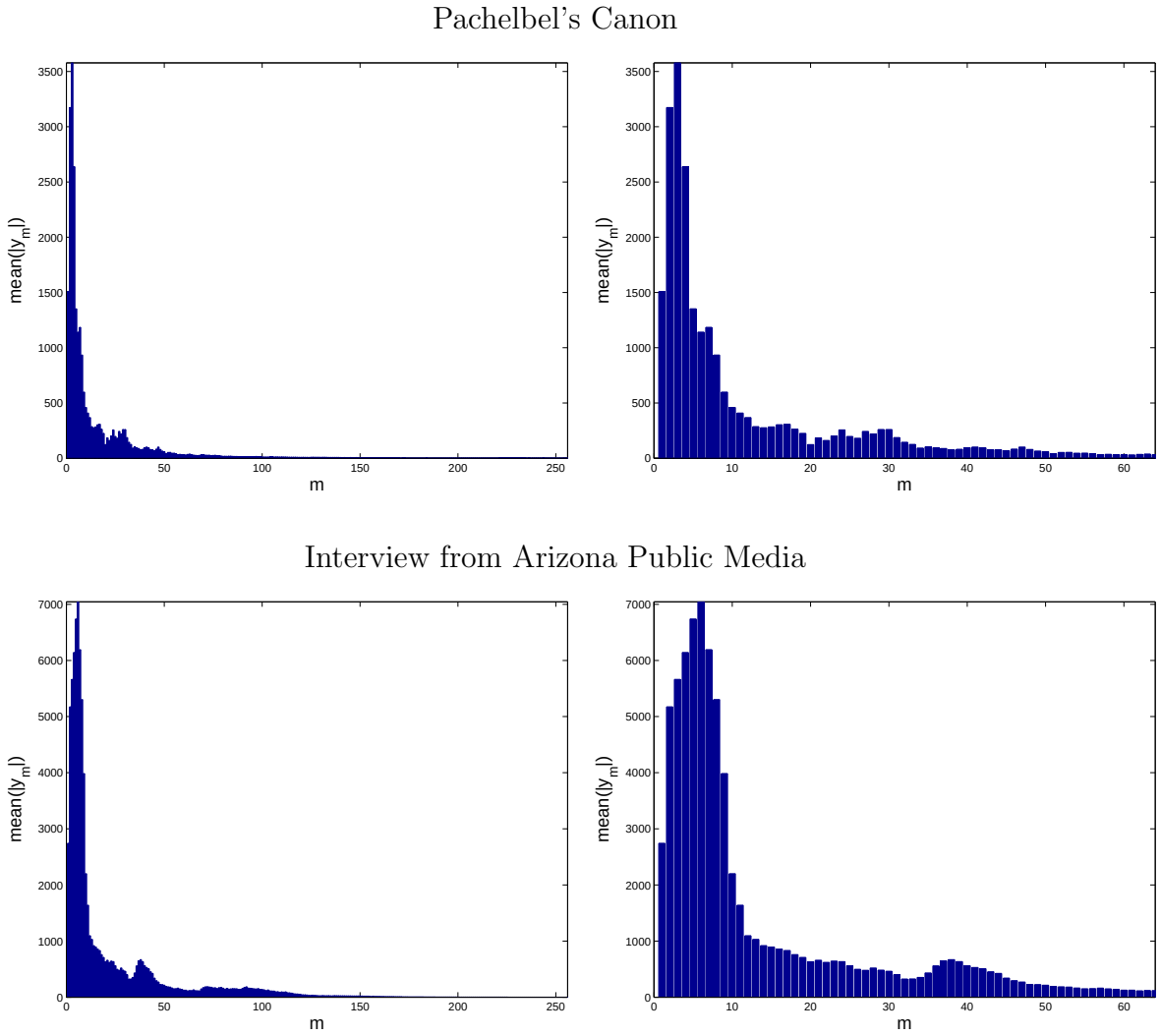

Figure 2: These figures show a profile of the DCT Coefficients of two example sound files: 20 seconds of Pachelbel's Canon in the top graphs, and a 7-second sample of an interview from Arizona Public Media in the bottom graphs. The graphs show the mean value across all packets of the absolute value of each DCT coefficient y_m as a function of coefficient number m. The left hand figures show the means for all 256 DCT coefficients, while the right hand figures show the means for only the first 64.

Fraction Transmitted	Pachelbel's Canon			Interview		
	SNR	PSNR	Compression	SNR	PSNR	Compression
1	65.8	101.2	43.7 %	76.9	101.1	49.9 %
$\frac{3}{4}$	46.8	82.2	35.7 %	60.9	85.1	42.4 %
$\frac{1}{2}$	41.4	76.8	26.7 %	40.7	64.9	31.7 %
$\frac{1}{4}$	32.2	67.7	15.6 %	22.9	47.1	17.4 %

Table 2: This table shows the compression rate and signal to noise ratio achieved by by only transmitting some fraction of the DCT coefficients.

abandon the information about later coefficients.

4.5 A First Attempt at Sound Compression

Given what we have discussed so far, we are ready to make a first attempt at sound compression. In this algorithm, we arrange the input sound data into packets of size 2^8 and compute the DCT of each packet, as discussed in section 4.4. Next, we round all DCT coefficients to integer values and arithmetically encode the first N DCT coefficients and write an output file. Engineers typically refer to this as applying a "low pass filter." To decompress this file, we arithmetically decode, pad each packet with zeros so that they have length 2^8, take the IDCT of each packet, then reassemble the packets into a single output vector. The purpose of rounding is to limit the allowed values of the DCT coefficients and thus increase the efficiency of arithmetic encoding. This idea is explored further in Section 5.

Table 2 shows the results of this compression technique on our two example files, with different fractions of the DCT coefficients being transmitted. Notice that significant compression is achieved in transmitting all DCT coefficients due to the rounding of the coefficients and subsequent arithmetic encoding of data. This is a relatively effective algorithm; by transmitting just half of the DCT coefficients, we are able to compress the sound file to between $\frac{1}{4}$ and $\frac{1}{3}$ of its original size while still maintaining decent sound quality. However, we can do better both in terms of compression and sound quality.

5 Quantization

In the compression algorithm presented in Section 4.5, we rounded the DCT coefficients before arithmetically encoding the data. This is a form of *uniform quantization*

which has the effect of limiting the alphabet from which the DCT coefficients are chosen, and thus decreasing the entropy of our data. Generally, if we seek to encode data chosen from a very large alphabet, we can lower the entropy of our data and thus improve compression,if we first restrict our alphabet to some smaller set. This is called *quantization*, and the use of quantization will lead us to superior compression algorithms. We will start by using uniform quantization to restrict the allowed values of the DCT coefficients.

The process of uniform quantization is simple; we need only pick some number ε, divide each entry in each packet by ε, and round to the nearest integer. This will not only restrict the alphabet from which DCT coefficients are chosen, but will cause the frequency distributions of the various characters in this new alphabet to be skewed away from the even distribution. Most notably, most of the high frequency DCT coefficients will be mapped to 0 in the new alphabet thus making the frequency of zeros in our data quite high. This phenomena will also contribute to a decreased entropy for our data.

5.1 Compression with Quantization

Using this idea of quantization, we may now modify the algorithm presented in Section 4.5 to create a somewhat better compression algorithm. Once again, we arrange the data into packets of size 2^8, and compute the DCT of each packet. Next, we divide all DCT coefficients by some fixed ε, and round to the nearest integer. We then arithmetically encode this signal, and transmit it. To decompress, we first arithmetically decode the transmitted signal, multiply by ε, and take the IDCT of each packet, then rearrange the packets into a single output signal. Notice that this compression algorithm with $\varepsilon = 1$ is equivalent to using the algorithm discussed in section 4.5 and keeping all DCT coefficients.

	Pachelbel's Canon			Interview		
ε	SNR	PSNR	Compression	SNR	PSNR	Compression
1	65.8	101.2	43.7 %	76.9	101.1	49.9 %
10	45.8	81.3	23.5 %	57.7	81.9	30.1%
18	41.2	76.7	18.4%	53.1	77.3	25.7 %
25	39.0	74.5	15.9 %	50.5	74.7	23.3%

Table 3: This table shows the compression rate and signal to noise ratio achieved by quantizing the DCT coefficients using some fixed ε. Note: $\varepsilon = 1$ is equivalent to keeping all DCT coefficients in the previous algorithm.

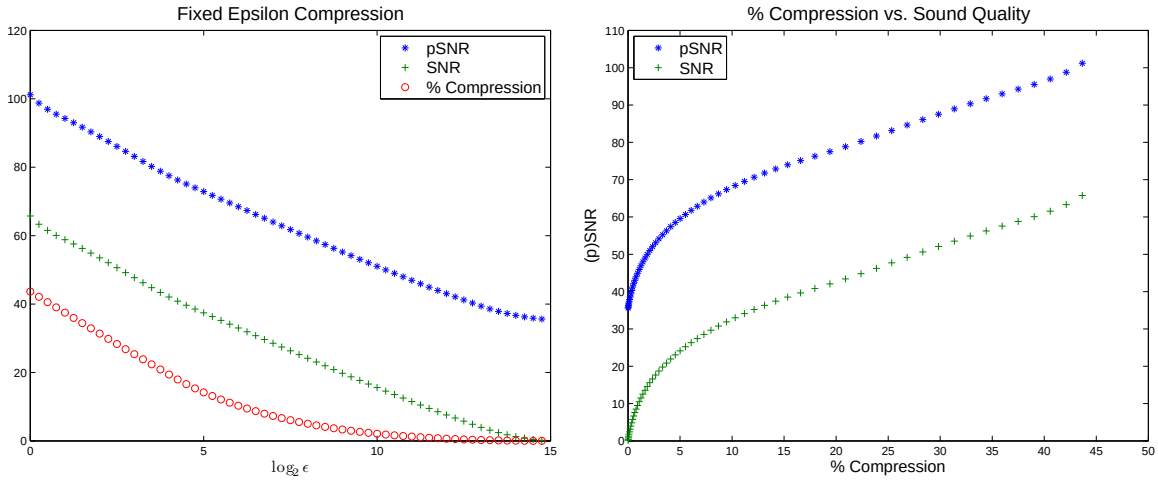

Figure 3: Fixed Epsilon Compression on Pachelbel's Cannon

Table 3 shows the results of running this compression scheme for various values of ε. Comparing Table 2 and Table 3, we notice that this scheme achieves higher SNR for a given compression rate. From here on, we will consider this algorithm as our point of comparison and try to improve upon it. We will no longer consider the algorithm of Section 4.5. Figure 3 shows the compression rate and sound quality for various values of epsilon. Notice that as we increase ε, the compressed file size decreases as does the sound quality.

Note that in order to decompress a file, we must also transmit the ε that we have chosen. This is another piece of metadata, but it is very cheap: sending a single ε does not cost us more than a handful of bits and makes very little difference in the size of the file.

6 K-means Clustering

In Section 3.1.1 we saw how a message that has an apparently high entropy may sometimes be decomposed into two shorter messages each with significantly smaller entropies. In this section, we will utilize this idea to achieve further compression building on the algorithm discussed in Section 5.1. To do so we will use K-means clustering to identify groups of DCT coefficients and then groups of packets with similar properties, and transmit these groups separately.

6.1 K-means Clustering

K-means clustering is a standard algorithm used to group data [2]. Given a set of data points, and n parameters quantifying each data point, the K-means clustering algorithm identifies K centroids in the parameter space and partitions the data points into K disjoint sets by identifying each data point with the closest centroid. Although imperfect in some regards, K-means clustering is usually an effective method of partitioning data.

6.2 Clustering the DCT coefficients

Our second algorithm is a modification of the algorithm presented in section 5.1. Again, we arrange the data into packets of size 2^8, and compute the DCT of each packet. We then divide all DCT coefficients by some fixed ε, and round to the nearest integer. Next, we partition the DCT coefficients into K clusters based on average size of the coefficients, where $K = 3$ in this algorithm. We use this partitioning to form three groups of data, each set containing information about each packet in the sound sample for each of the DCT coefficients in the corresponding cluster. We then arithmetically encode each of these groups separately. We then transmit the arithmetic encoding of all groups and the metadata generated by this algorithm, which consists of a histogram for each group (which allows us to arithmetically decode) and a mapping which indicates which DCT coefficients are in which groups.

To cluster the DCT coefficients, we calculate the mean value of each DCT coefficient across all of the packets. We then perform a one-dimensional K-means clustering on this parameter such that coefficients that are relatively large are clustered together, as are coefficients that are relatively small. Figure 4 illustrates the clustering found for both of our example pieces.

6.3 Clustering the Packets

Our third and final algorithm further extends the idea of K-means clustering. We follow the same lines as the algorithm presented in section 6.2, except after quantization we first cluster the packets into 4 groups, then cluster the DCT coefficients in each packet cluster into 3 DCT clusters (in the same manner as described in section 6.2). This gives a total 12 groups of data, and we arithmetically encode and transmit the data and a histogram for each of these groups separately.

We cluster the packets using a two-dimensional K-means clustering. To do this, we must first create two parameters that quantify the packet. The first parameter we use is the smallest index i such that at least 95% of the energy of the packet is

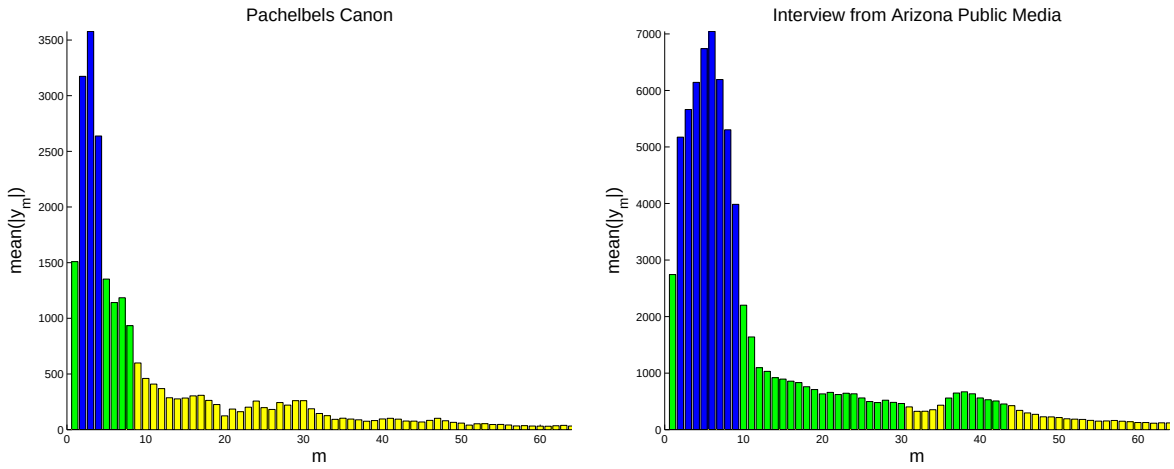

Figure 4: Clustering of the DCT coefficients. The x-axis shows the DCT coefficient number, and the y-axis shows the mean value of each DCT coefficient across all packets in the sound sample. Note that only the first 64 coefficients are shown; the remaining coefficients are assigned to the yellow cluster.

contained in the first i DCT coefficients. That is, if y is the vector of DCT coefficients for some packet, then we define y_n to be the first n coefficients of y. We seek the smallest i such that

$$\frac{\|y_i\|_2}{\|y\|_2} > .95.$$

The second parameter used is the index j of the DCT coefficient in the packet of maximal size. There are many other ways to quantify packets, some of which may be better than the method presented here. However, this parameterization did yield improvements in compression, as shown in figure 7.

7 Comparisons and Conclusions

The three algorithms we developed in sections 5.1, 6.2 and 6.3 are illustrated in the block diagram in Figure 6. The algorithms are all quite similar; the first steps in each are to partition the data into packets, take the DCT of each packet, and quantize the DCT coefficients, and the last step in each is arithmetic encoding. However, in algorithms two and three, we cluster the data to increase the efficiency of arithmetic

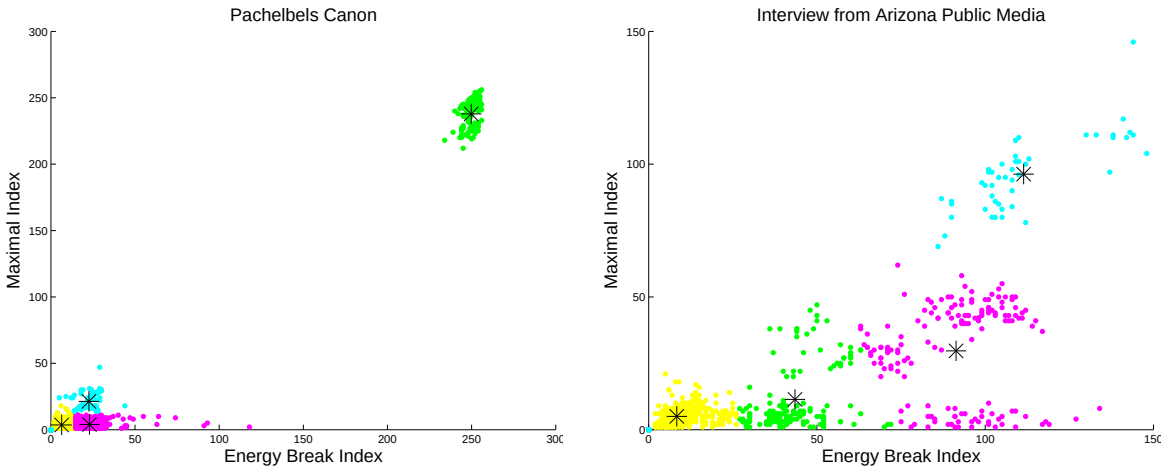

Figure 5: Clustering of the Packets. Each point of the graph corresponds to a single packet. The x-axis shows the index in each packet before which 95% of the energy of the packet occurs. The y-axis shows the index with the maximally sized DCT coefficient.

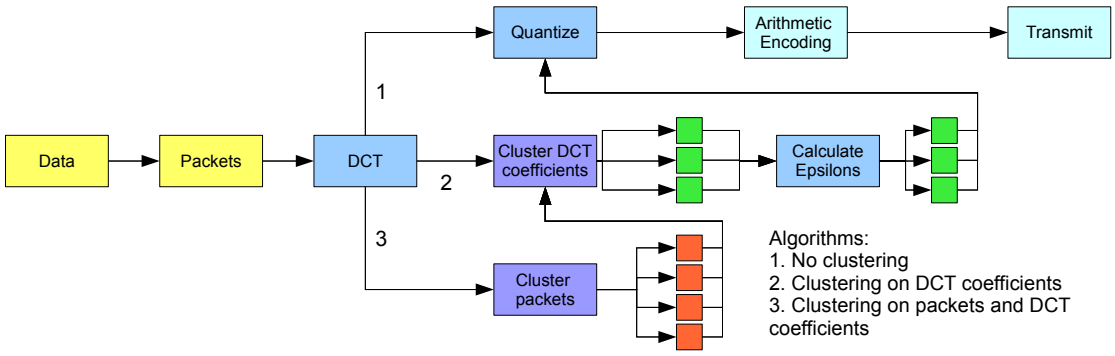

Figure 6: Block Diagram of Three Algorithms. The first algorithm outlines the basic procedure used in DCT compression schemes. The second and third algorithms describe improvements on the first algorithm.

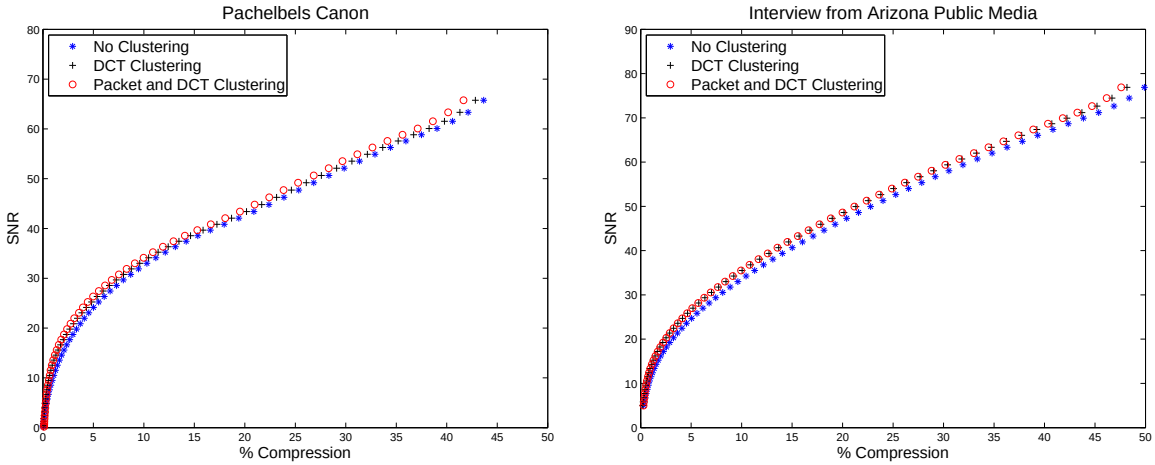

Figure 7: Comparison of Algorithms. By varying ε, the algorithm compresses a file with a given SNR. The percent compression is graphed on the horizontal axis. By picking a specific SNR, we can compare compression across the three algorithms.

encoding.

Figure 7 shows a comparison of the three algorithms on our two sample pieces. In these plots, we see the SNR as a function of the compression achieved. Notice that the left-most curve corresponds to algorithm 3 (clustering both packets and DCT coefficients), the middle curve corresponds to algorithm 2 (clustering only DCT coefficients), and the right most curve corresponds to algorithm 1 (no clustering). This indicates that algorithm 3 achieves the smallest file size for a given sound quality, followed by algorithm 2, and then algorithm 1. We ran similar tests on thirteen other samples of classical music and radio interviews and found the same trend; algorithm 3 consistently outperforms algorithm 2, which in turn outperforms algorithm 1. When studying Figure 7, the improvement in compression may appear small, but it constitutes a 4% to 30% reduction in compressed file size between the first and third algorithms. The improved performance of algorithm 3 justifies the increased complexity of this algorithm.

As a final test of our algorithms, we compared their performance to the commercially popular MP3 compression codec. Table 4 shows the compression and SNR of a variety of files which have been compressed with MP3 compression. AZPM Sample 5 is the same interview from Arizona Public Media referenced in Figure 7 and elsewhere in this paper. The MP3 codec compresses this file to about 18% of its

File	Compression	SNR
Arizona Public Media Sample 1 (30 seconds)	18.17%	25.5
AZPM Sample 2 (5 seconds)	18.33%	24.8
AZPM Sample 3 (10 seconds)	18.23%	25.5
AZPM Sample 4 (4.5 seconds)	18.33%	25.9
AZPM Sample 5 (7 seconds)	18.15%	25.5
AZPM Sample 6 (7 seconds)	18.32%	25.7
AZPM Sample 7 (5 seconds)	18.30%	25.7
Moonlight Sonata (30 seconds)	18.17%	26.0
Clarinet Sample (31 seconds)	18.23%	25.0
Rachmaninoff Sample (11 seconds)	18.21%	25.8

Table 4: Compression with the MP3 codec. This table shows the compression rate and signal to noise ratio achieved by the mp3 compression scheme. Our algorithms achieve an SNR of 44-45 at 18% compression, and a compression of 5-6% for an SNR of 26.

original size with an SNR of 25.5. At an SNR of 25.8, all three of our algorithms do much better, with algorithm 1 compressing the file to 6.2% of the original file size, algorithm 2 to 5.2% and algorithm 3 to 5.1%. At comparable compression to the MP3 codec, our algorithms have the following SNRs: 44.6 for algorithm 1 (at 18.2% compression), 45.9 for algorithm 2 (at 17.8% compression) and 45.9 for algorithm 1 (at 17.7% compression). In short, our algorithms outperform the MP3 compression scheme in every measure considered.

8 Acknowledgements

The authors would like to thank Dr. Marek Rychlik for advising us and guiding us through a fascinating subject. We would also like to acknowledge Anna Latta and Arizona Public Media for loaning us uncompressed audio files on which we could test our algorithms.

References

[1] Gerald B. Folland. *Real Analysis: Modern Techniques and Their Applications.* John Wiley & Sons, Inc, 1999.

[2] David J. C. Mackay. *Information Theory, Inference & Learning Algorithms.* Cambridge University Press, June 2002.

[3] Mark Nelson. Arithmetic coding + statistical modeling = data compression. *Dr. Dobbs Journal*, 1991.

[4] K. Sayood. *Introduction to Data Compression.* Morgan Kaufmann, 2000.

[5] Walter A Strauss. *Partial Differential Equations: An Introduction.* John Wiley & Sons, Inc, 1992.