5-2016

Towards Understanding the Compression of Sound Information

Kathleen Marie Fisher
Trinity University, kfisher2@trinity.edu

Towards Understanding the Compression of Sound Information

Kathleen Fisher

A departmental senior thesis submitted to the Department of Computer Science at Trinity University in partial fulfillment of the requirements for graduation with departmental honors.

April 15, 2016

Dr. Matthew Hibbs
Thesis Advisor

Dr. Paul Myers
Department Chair

Sheryl R. Tynes, AVPAA

Student Agreement

I grant Trinity University (“Institution”), my academic department (“Department”), and the Texas Digital Library ("TDL") the non-exclusive rights to copy, display, perform, distribute and publish the content I submit to this repository (hereafter called "Work") and to make the Work available in any format in perpetuity as part of a TDL, Institution or Department repository communication or distribution effort.

I understand that once the Work is submitted, a bibliographic citation to the Work can remain visible in perpetuity, even if the Work is updated or removed.

I understand that the Work's copyright owner(s) will continue to own copyright outside these non-exclusive granted rights.

I warrant that:

1) I am the copyright owner of the Work, or
2) I am one of the copyright owners and have permission from the other owners to submit the Work, or
3) My Institution or Department is the copyright owner and I have permission to submit the Work, or
4) Another party is the copyright owner and I have permission to submit the Work.

Based on this, I further warrant to my knowledge:

1) The Work does not infringe any copyright, patent, or trade secrets of any third party,
2) The Work does not contain any libelous matter, nor invade the privacy of any person or third party, and
3) That no right in the Work has been sold, mortgaged, or otherwise disposed of, and is free from all claims.

I agree to hold TDL, Institution, Department, and their agents harmless for any liability arising from any breach of the above warranties or any claim of intellectual property infringement arising from the exercise of these non-exclusive granted rights.”

I choose the following option for sharing my thesis (required):

[X] Open Access (full-text discoverable via search engines)
[ ] Restricted to campus viewing only (allow access only on the Trinity University campus via digitalcommons.trinity.edu)

I choose to append the following Creative Commons license (optional):

Contents

List of Figures

Towards Understanding the Compression of Sound Information

Kathleen Fisher

April 14, 2016

Abstract

The purpose of this thesis is to explore data compression, specifically as it relates to sound information. Data transfer is an important part of the current technology driven lifestyle and compressed data means faster transmission. This thesis will explore how compression can be applied to sound while considering often overlooked factors, such as the way the sense of human hearing works to interpret sounds. An example of Huffman compression follows the general discussion of the compression of sound information.

1 Introduction

Information exchange across digital mediums is an integral part of everyday life (at least for the majority of the human population). Present concerns

relate to making these exchanges faster, easier, and with minimum costs. Compressing information into a smaller size allows for faster transmission (because there is less data there to transfer). The smaller size also reduces the "cost" of space; a user can store more movies, images, or songs per specified chunk of memory. The issue of improving compression techniques has even surfaced on popular media, with a compression algorithm taking center focus on a recent HBO series, *Silicon Valley*.

An important part of compression is understanding the nature of the information that will be compressed. Many types of compression work only on information with certain characteristics. For example, a DNA sequence stored in an ASCII text file (where each character is stored with 8 bits of information) can be compressed to 25% of its original size by only using 2 bits to store each character since there are only four possible characters: A, G, C, or T. This, of course, would not work on an ASCII text file storing English sentences because there are more than four possible characters in English sentences. Many types of compression depend on the nature of the information, so it makes sense to consider compression techniques through the lens of a particular field–in this case sound information.

An important consideration for compressing sound information is streaming music. If music data can be decoded quickly enough, compression can help save bandwidth and loading time. If music data is compressed to a smaller size, less data has to be streamed to an end user. Compression runs into problems with streaming when it decodes compressed information too

slowly or data gets lost during transmission. There are far more detailed implications that will not be discussed in this thesis.

Let us take a moment to think about the absurdity of modern capabilities. Before addressing sound specifically, let us address another related media, the photo, which is also commonly transferred. Every picture taken is capturing a moment in time–in order to recreate that moment sometime in the future. If we consider the universe with four dimensions, where the fourth dimension is time, this moment exists in at a given coordinate in space and time, or spacetime. The light waves from this specific spacetime are carefully captured and cataloged. The camera attempts to mimic the eye in how it collects light waves. Every time you look at a picture, you are receiving the recreated light waves of a given spacetime–looking at a past spacetime. In that sense, looking at a picture is a simple version of time travel. Abstracting the specifics of photography away, this applies to sound as well.

1.1 What is sound?

Humans often think of sound in terms of what they can hear. A bird chirping is a sound. Water cascading into the sink from a faucet is a sound. Anything that is perceived from a humans sense of hearing, through the ears, is a sound. More scientifically, sound is a wave created from vibrations in the environment, requiring a medium through which to travel (particles in the air, liquids, etc.) [1, 2, 6, 14]. Sound is not necessarily one specific vibration, or frequency, but the culmination of frequencies associated with a specific

object or action. The brain distinguishes the car driving by outside your window as a separate sound from buzz of voices from a television in an adjacent room from the sound of a toilet flushing across the hall–even though they all happen simultaneously.

1.2 How sound is stored

Sound waves do not naturally lend themselves to the discrete nature of computation. *If you already understand the basics of computer memory, skip to section 1.2.1 on Data Entropy.* Everyday modern computers work with binary digits, or more succinctly *bits* [4]. Interesting historical side note: The term bit was first published and popularized by Shannon [4] in 1948 (though Shannon attributes the term to John W. Tukey) [4]. Each *bit* of information can exist in two possible states, which we consider as either a 1 or a 0. Any information stored on a computer is ultimately stored with a sequence of binary digits, which is a sequence of 1s and 0s. This means that a sound is stored as a sequence of 1s and 0s. Unfortunately for computers, sound, as a wave, is inherently continuous. Computers (the current standard computers, anyway) do not store continuous data, so sound information must be represented discretely in order to be stored digitally.

Numerous standards exist for recording and storing sound waves. There are varying standards for taking sound waves as input to a digital system (e.g. through a microphone) as well as varying standards for storing that input. Considering sound files, there is no universal standard; multiple standards

exist such as .wav, .mp3, .flac and many others.

1.2.1 Data Entropy

Essentially, there exists a mathematical concept of *entropy* associated with information. You may recall that the term entropy involves chaos or disorder. The entropy associated with information theory instead involves *uncertainty* [4]. This is sometimes referred to as Shannon entropy since it was first described by Shannon's A Mathematical Theory of Computation [4]. As Shannon describes more succinctly in a later paper, entropy is "how much information is produced on the average for each letter of a text in the language" [11]. Mathematically, H is the entropy for a character c where p_c is the probability of occurrence of c [4, 11, 15, 7].

$$H = -log_2(p_c)$$

1.2.2 PCM (Pulse Code Modulation)

One of the challenges of storing sound waves is turning continuous data into discrete data. Somehow we have to record all of the vibrations that make up a sound in 1s and 0s. A common method is to use pulse code modulation (PCM). This essentially takes the waves produced by sound and records their amplitudes to a given degree of specificity, the *bit depth*, and at a set frequency, the *sample rate*. So PCM records the *bit depth* number of bits of information every *sample rate* period of time.

1.2.3 Nyquist-Shannon Sampling Theorem

The Nyquist-Shannon theorem, otherwise known as the sampling theorem, deals with the sampling rate of methods such as PCM. Essentially, if you do not take samples often enough, PCM will not read enough data points per cycle to record the correct frequency [14, 16]. The sampling theorem essentially sets an upper bound on the necessary sampling rate. Called the Nyquist interval, "this upper bound is $\pi/\Delta w$ where Δw is the angular frequency bandwidth of the signal" [16].

1.2.4 Fourier Transform

Another way to transform the continuous data of sound waves into a discrete format is to use the Fourier transform. If you are familiar with the Taylor series, the Fourier transform works similarly. You can decompose a continuous data set into an infinite series of sine and cosine equations [14]. The sum of these sin and cosine equations represents the continuous waveform [14]. It is of course not practical or currently feasible to store an infinite series of sign and cosine equations, but the sine and cosine terms at the beginning of the Fourier series have the most weight over the resulting waveform. Instead of taking the infinite series, taking some terms from the beginning of the Fourier series will compose a decent approximation of the complete, exact waveform.

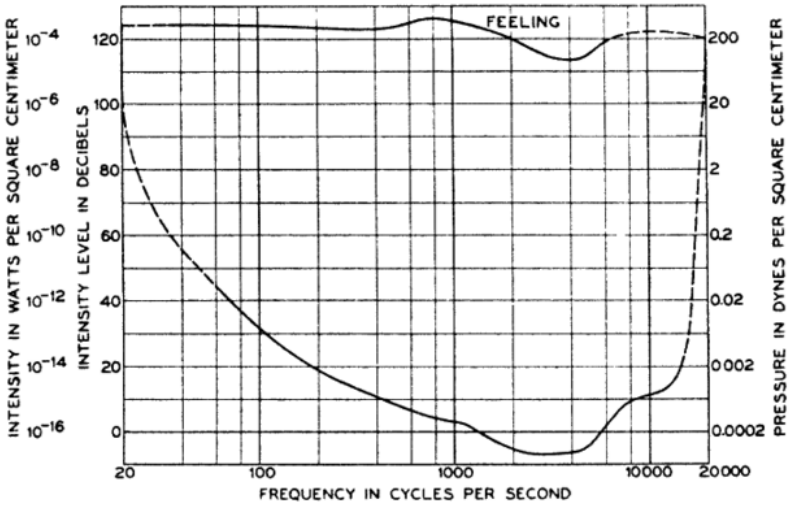


Figure 1: This figure shows that threshold of human hearing is 20 Hz to 20 kHz, depending on the loudness of a sound (pressure). This data was obtained from a study done by Bell Telephone Laboratories in collaboration with U.S. Public Health Service [2]. This figure is from [2]

1.3 How the ear works

This section will detail the workings of the human ear in how it affects the topic of storing sound data, drawing attention to the intricate nature of signal processing in the body. While considering the requirements and abilities of the digital transmission of sound, it is important not to forget our own enabling conditions, mainly the abilities and limitations of the human sensory systems ability to interpret vibrations as sound. Generally, the quoted range of discernible frequencies for the human ear is 20 Hz to 20 kHz [1, 2, 6, 14] This, however, is heavily dependent on the loudness of the sound (often measured in decibels) and individual characteristics, such as age [1, 6]. As

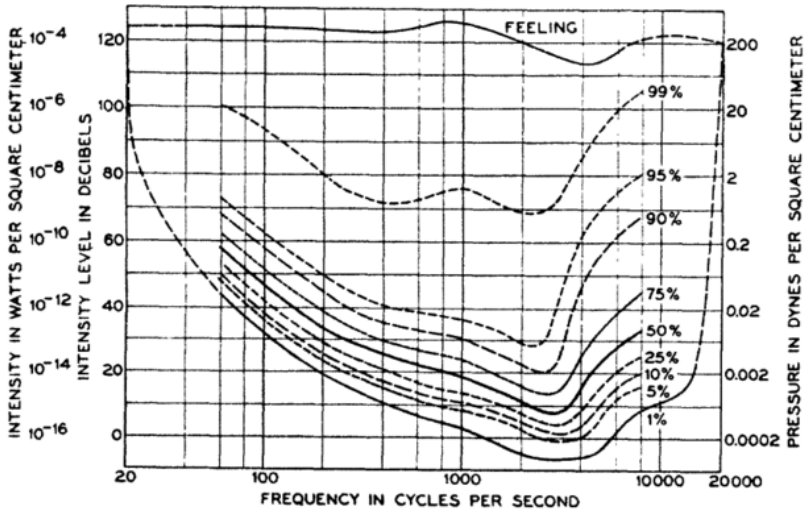


Figure 2: This figure shows that threshold of human hearing (generally 20 Hz to 20 kHz) has a large amount of variation at the individual level. This data was obtained from a study done by Bell Telephone Laboratories in collaboration with U.S. Public Health Service [2]. This figure is from [2]

is evident in Figure 1, the ability to hear a certain frequency depends on the intensity of the sound [1, 6]. And this threshold is not the same for each individual human, as evident in Figure 2 [1]. This is due to natural variations and degradation in the complicated process of the sense of hearing. Note also that increasing intensity relates to increasing pressure (the units along the rightmost y-axis).

The ear has thousands of tiny pieces working together to help the body sense various frequencies and translate them into meaningful sounds (among other functions that will not be discussed here, such as balance). First consider the three standard parts of the ear: outer, middle, and inner. Sounds (vibrations) travel through the ear from the outer ear to the inner ear. The

outer ear includes the visible portion of the ear (the pinna) and the ear canal [1, 2]. After sound waves pass through the ear canal, they hit the tympanic membrane (eardrum) which passes the vibrations through the ossicular system [1]. The ossicular system consists of a few small bones (ossicles) in the middle ear, called the malleus, incus, and stapes (more commonly referred to as hammer, anvil, and stirrup respectively due to their shape [1, 2]. When a sound wave hits the tympanic membrane (again, eardrum), the vibrations must then pass through the ossicles (malleus, incus, and stapes) of varying shapes and sizes [1, 2]. Consider how vibrations/waves (of equal amplitude, speed, etc.) transfer through objects of varying shapes and sizes. Once the vibrations/sound waves hit the tympanic membrane, they begin to pass through mediums other than the standard air. (This is, of course, assuming that the waves have not passed through something other than air on the way to your ear–consider instead hearing while underwater). The ossicles (tiny bones) essentially pass the vibrations that are sound information from one to the other into the cochlea [1, 2].

The cochlea has many intricate pieces, but we primarily care about the basilar membrane because it is where sound waves are converted into electrical signals the brain can process. Before discussing the basilar membrane however, it is important to mention that the cochlea is filled with liquid [1, 2]. This means that sound waves must change mediums from air to liquid as they pass from the middle to the inner ear. Fluid requires a greater pressure to achieve the same vibration patterns, so the ear must adjust somehow to ac-

count for this difference. To do this, the ear utilizes the tympanic membrane and ossicular system to increase the pressure of a sound wave against the fluid in the cochlea to attain proper impedance matching. The pressure against the fluid is 22 times greater than the pressure of a sound wave against the tympanic membrane [1]. The basilar membrane is essentially a long chamber with nerve endings along its surfaces [2]. These nerve endings are what send electrical signals to the brain identifying sounds [2]. The brain differentiates the frequencies of sound waves by how far they travel along the basilar membrane [1]. Higher frequencies will not travel as far along the basilar membrane as lower frequencies [1]. There are over 20,000 basilar fibers along the basilar membrane whose vibrations determine how your brain interprets sounds [1, 2]. If even a small change occurs in the frequency of a vibration on its way to these fibers, the brain will interpret the frequency differently.

Another relevant feature of the ear is what is called *attenuation reflex*. The attenuation reflex is a natural muscle response triggered when the ear is exposed to very loud (high pressure) sounds [1]. This response causes a rigidity in the ossicular system that dampens sounds below 1000 Hz by reducing their intensity [1].

The function of the ear is important because after being recorded, compressed, uncompressed, and pumped out of a speaker to your ear, sound waves have to go through all of the parts of the ear to finally be interpreted as sound by the brain. The way the brain receives information about sound though the sense of hearing is intricate and complicated and varies on an

individual basis (as seen in Figure 2). Since the ultimate goal of storing and compressing sound information is for it to be consumed by humans, the humans sense of sound should be kept in mind when dealing with sound information.

1.4 Process of sound travel

This section will address the complexity of hearing and interpreting a sound in terms of the physical path of the vibrations that make up sound. This will be addressed first from the perspective of listening to an *original* sound, then from the perspective of listening to a *recorded* sound. For the purposes of this paper, an *original* sound will refer to a frequency generated in physical space that has not been processed through a digital medium, while a *recorded* sound will refer to a sound that has been processed digitally.

First consider the path of the *original* sound. We can consider this path to have three states: the creation The sound waves must first traverse from their point of origin, bouncing off of and interacting with parts of the environment (causing, for example, resonance), to the physical position of the ear. Once the vibrations reach the outer ear, they travel down the auditory canal (ear canal) to the tympanic membrane (eardrum) [1, 2]. Vibrations must then pass through the ossicular system (small bones) to the cochlea (more complicated inner ear parts) [1, 2]. And so on and so forth until your brain can decipher the sound, for the more detailed explanation, see section 1.3 How the ear works. Aside from the complexities of the ear, the sound

waves only have to pass from the position of origin to the position of the ear, only being changed or altered by the environment between those two points.

On the other hand, the path of the *recorded* sound has additional intermediate steps. The recorded sound path differs from the original sound path when it is intercepted by an input device such as a microphone. To be stored in a computer's memory, vibrations in the air must be transformed into binary information. This transformation is likely to cause in a loss of information since the sound waves can only be recorded to a certain degree of precision. After this the binary data can go through multiple intervening steps, ultimately ending up as the input to a speaker or similar device. The speaker then converts the binary data back into vibrations that can be interpreted by the ear. Then those vibrations must undergo the entire process of being interpreted as sound by the human sense of hearing.

1.5 Data compression basics

1.5.1 Lossless vs. lossy compression

A defining characteristic of a compression technique is whether it is lossless or lossy. A compression technique is considered lossless when the compressed can be uncompressed exactly back to its original state; no information is lost during the compression process for a lossless compression [6, 15]. On the other hand, lossy compression techniques do not retain all of the information of the original data [6, 15]. A common example of this is MP3 compression;

MP3 compression discards *unnecessary* information in order to significantly reduce data sizes [6]. Whether or not the lost information is completely *unnecessary* remains debated. MP3 removes information about the sound that humans should not be able to hear [6], but as previously discussed in section 1.3 How the ear works, the capabilities of human hearing are not identical across the board. It is entirely possible that some humans can hear differences in an original and MP3 compressed file (and many humans state that they can).

1.5.2 Quantitatively comparing compression

One of the simplest, most common metrics for comparing compression techniques is the resulting *compression ratio*. The compression ratio is simply the ratio of uncompressed size to compressed size, $\frac{uncompressed}{compressed}$. This metric does not take speed into consideration. Calculating and comparing the compression ratios for some given file or set of files is a way to compare multiple compression techniques against each other. However, depending on the type of information stored in the files, certain compression techniques could have a natural advantage if they are optimized for that file type. This sort of competition could also cause the creators of compression techniques to optimize for the set of test files as opposed to the general use case.

Shannon's data entropy can also be used as a metric for rating compression. The entropy of a character x is theoretically the number of bits required to store the amount of information inherent to that character x in

its respective alphabet [4]. Knowing the entropy value for each character in an alphabet would allow you to calculate the minimum amount of space theoretically required to hold the information inherent to a certain set of characters. Though this only applies to lossless techniques, the closer you can get to the theoretical minimum, the better the compression technique.

Interestingly, a brand new metric called the Weissman score, created by a Stanford professor Tsachy Weissman, emerged from the compression-centered HBO series *Silicon Valley* [10]. The Weissman score, W, is as follows:

$$W = \alpha \frac{r}{\bar{r}} \frac{log\overline{T}}{\log T}$$

Variables r and $\bar{r}$ refer to the compression ratios of the target compression technique and a standard compression technique respectively[10]. While T and $\overline{T}$ refer to the time it takes to compress with the target and standard respectively [10]. It has only been used in at least one academic paper [5], but that paper patronized the metric by referring to it as a "fictional Weissman score".

1.6 Huffman coding

Huffman coding is a lossless compression method [6]. Huffman coding runs on a set alphabet of characters, utilizing probability information on how often a given character will occur. Characters with higher occurrence frequency are represented with fewer bits while characters with a lower frequency are repre-

sented with more bits [3, 6]. Consider running Huffman coding on "*lossless*". The word contains the character set $\{l, o, s, e\}$, so the alphabet considered, A, is $\{l, o, s, e\}$. The probability of encountering the character l for this example is $\frac{2}{8}$, or 25%. The probability of encountering s is 50% while o and e only occur once with a probability of 12.5%. Instead of storing each character as the standard 8-bits of ASCII or a fixed bit length, characters are stored with a variable bit length corresponding to their probability. So for this example, the most likely character, s, could be represented simply by one bit, 0, with the others represented by more bits: 10 for l, 110 for o, and 111 for e. The Huffman coding for "*lossless*" would then be 10110001011100, 14 bits long. The smallest fixed bit length that could represent each character is two bits, which for eight characters would result in using 16 bits to store "*lossless*". This is only two bits longer than the Huffman coded "*lossless*", and the Huffman coded version must also keep track of a "key" (a tree, which will be discussed later) for decoding purposes. While Huffman coding will result in a larger file size for worst case scenarios, it will result in a smaller file size with certain types of character probability distributions.

1.7 MP3 Basics

MP3 compression is a lossy method of compressing sound information [6]. There are two large steps of MP3 coding: throwing out 'useless' information based on psychoacoustics and Huffman coding [6]. The MP3 dictates that information can be thrown out that is seen to be not audible to the human

hear or not processed by the human brain [6]. To understand the specifics of which data MP3 loses, more detailed aspects of psychoacoustics would need to be understood such as simultaneous masking and temporal masking [6]. The space savings of MP3 coding are primarily gained by ignoring much of the sound information in the first part of the process; the Huffman coding after only allows a smaller space saving. This is due to the respective natures of lossy and lossless compression. Because lossless compression must perfectly store all information, the reduction of space is inhibited. The analysis required to remove 'unnecessary' information also takes up a lot of time and computer resources [6]. This means that the encoding process for MP3 is lengthy, while the decoding is much quicker.

2 Methods

2.1 Language Choice

The chosen programming language for this project was C++. This was primarily chosen due to requirements for utilizing GPU processing on the finished compression technique. The next steps for this project are to speed up the compression technique with GPU processing using the NVIDIA CUDA language. CUDA is currently only compatible with C++. CUDA was decided on due to the limitations of the author–in possession of a NVIDIA graphics card and lacking funds to purchase another type.

2.2 Exploring the WAV format

The WAV format was used as the starting point for the compression algorithm. It was decided that the WAV format would be the appropriate starting point for compression because WAV is the starting point for MP3 compression, one of the most widespread compression techniques [6]. The WAV format is essentially the PCM of the sound waves with header information to detail the specifications of the PCM such as the sampling rate and bit depth.

The first step of this compression technique is to read in the information stored in a WAV file. The WAV format is composed of a header and data section. The header contains important information such as the size of the data section in bytes, the bit depth in bits, and the sample rate in Hertz (Hz, cycles per second). The compression technique for this experiment records the header information (as well as the data, of course) for use during compression. At this point a compression technique had not been decided because a closer look at the nature of WAV formatted data was warranted.

For a 16 bit depth file, there are 2^{16}, or 65,536, possible bit arrangements for each sample. We were curious if all possible combinations are used and if we could find any trends in WAV formatted data. By plotting a histogram of possible bit arrangements (Figure 3), we discovered that data values in the middle were significantly less likely to occur than those at the ends, like 0 and 65,535. There were also many middle values that did not occur at all. The probability distributions demonstrated in Figure 3 lend themselves Huffman

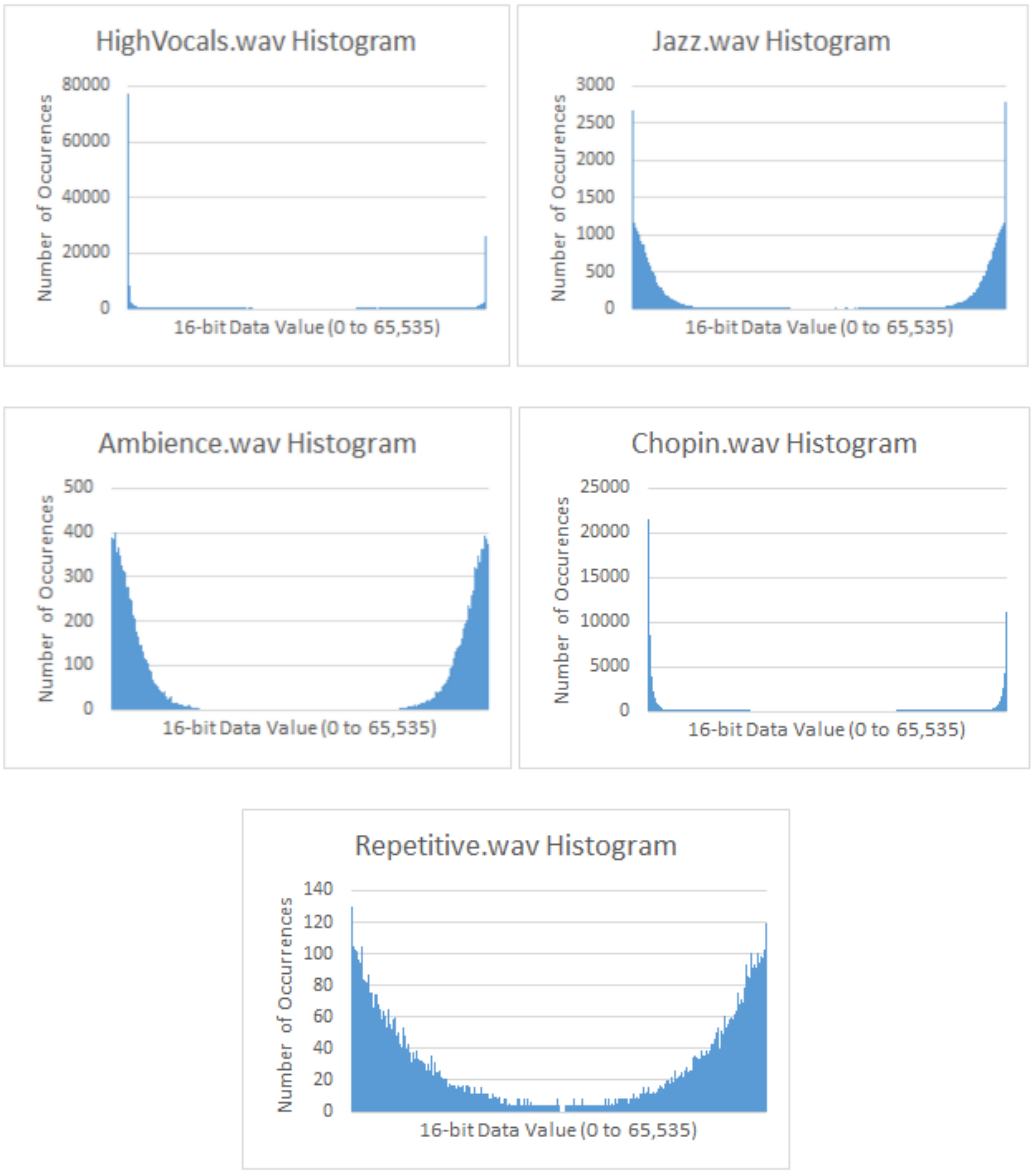


Figure 3: The graphs above show the histograms of five 16-bit depth .wav files. The lowest and highest integer values (e.g. 0 and 65,535) occur the most frequently in .wav files, with many of the middle data (e.g. 32,767) values not occurring at all. (The values that do not occur differ across different files.)

In the pseudocode that follows, we assume that C is a set of n characters and that each character $c \in C$ is an object with an attribute $c.freq$ giving its frequency. The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with a set of $|C|$ leaves and performs a sequence of $|C| - 1$ "merging" operations to create the final tree. The algorithm uses a min-priority queue Q, keyed on the *freq* attribute, to identify the two least-frequent objects to merge together. When we merge two objects, the result is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4      allocate a new node z
5      z.left = x = EXTRACT-MIN(Q)
6      z.right = y = EXTRACT-MIN(Q)
7      z.freq = x.freq + y.freq
8      INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```

Figure 4: This is the algorithm for Huffman coding from the CLRS Introduction to Algorithms textbook [3]

coding due to the steep difference between likely and unlikely characters of the 65,536 possible in the 16 bit depth alphabet.

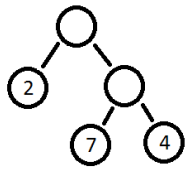
2.3 Basic technique

The overview of the implemented technique is as follows. First, the WAV format file is read into the program using the $fread$ function in the C++ standard library. The header information and sound data are stored for later use in the program. The sound data is passed through once, using a for loop, to acquire probability information for each character in the alphabet. This is done by counting the number of occurrences of each character, the probability is gained by dividing the number of occurrences for a character

by the data size as defined by the WAV header. A priority queue is then initialized, where values with a lower probability have higher priority. Each character that occurs more than once is added to the priority queue. Then the Huffman tree is built with this priority queue, Q. The Huffman tree is built according to the standard algorithm outlined in the CLRS Introduction to Algorithms textbook [3]. After creating the tree, it runs through the tree in-order to assign the value of each leaf to its Huffman encoded value in a map structure to provide constant look-up times. Then it runs through the data of the WAV file again, outputting the encoded version of the value to an encoded data file. For convenience, the header information and Huffman tree information are output to separate files, so in total the result of the compression is three files: a header file, a tree file, and a data file.

2.4 Tree storage

An important consideration is how to store the tree information with the encoded sound data so that it may be decoded later. A natural implementation is to add this tree information to the header, but storing a tree structure does not take a trivial amount of memory. To reduce the size of the stored tree, the following method was used. In an in-order traversal of the tree, a 0 is recorded for every move to a left child. Whenever a leaf is reached (where the relevant data is in a Huffman tree), a 1 is recorded followed by the data value at that leaf. Consider the example tree in Figure 5. This tree would be stored as 01(2)01(7)1(4) where the values in parentheses would instead



a

Figure 5: This tree would be stored as 01(2)01(7)1(4) where the values in parentheses would be the actual 8 or 16 bit binary representation of the value. So if it was the tree for an 8 bit depth WAV file, the stored tree would be 01 00000010 01 00000111 1 00000100.

be the 8 or 16 bit binary representation of the value.

2.5 Exploring lossy methods

After implementing the lossless Huffman technique, we were curious to see how we could implement some lossy versions to further reduce file sizes. In addition to regular bit depth reduction and sampling rate reduction, we tested different methods of *corrupting* WAV files to see if we could notice a difference in the auditory experience. A few corruption methods were tested: flip the least significant bit of every 16 bit value (Appendix 5.1.1), flip the most significant bit of every 16 bit value (Appendix 5.1.2, *very* bad), shift every 16 bit value left by 2 (Appendix 5.1.3, bad), shift every 16 bit value right by 2(Appendix 5.1.4, *very* bad), round data in middle values of 5000 to 60000 to the closest high (60000) or low (5000) value (bad), and add an offset to each 16 bit value (Appendix 5.1.5, introduced high pitch tone at large offsets) See Audio Appendix for audio samples. As expected, the only corruption method that did not appear to have an auditory impact was

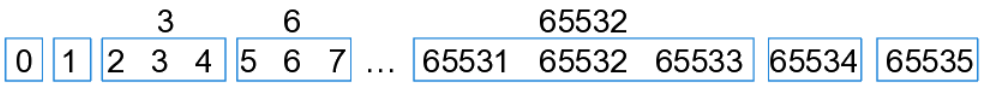


Figure 6: This shows how adjacent values are combined to form windows of 3.

flipping the least significant bit for each data point (because data points with similar values have similar frequencies). The lossy technique implemented involves combining adjacent alphabet characters into the same encoded value to reduce the size of the Huffman tree, thus decreasing the resulting file size. When combining every 3 adjacent characters into the middle value (see Figure 6), the compressed Repetitive.wav file was reduced to 1,434,817 bytes with the Huffman tree reduced to 33,908 bytes. This is in comparison to the lossless compression of 1,589,940 bytes with 86,297 bytes of tree data. The tree data is essentially 3 times smaller with not very apparent audio quality reduction. Listen to the resulting file in the Audio Appendix 5.1.6.

2.6 Generic Huffman Tree

Using a generic Huffman tree based on standard WAV file probabilities would decrease encoding time and total compressed file size (improving compression ratios). The idea would be to use the same tree for every compression. This runs into a problem because each WAV file has different holes in their middle values (different values occur zero times in different WAV files). If every possible value for 16-bits was included in this generic Huffman tree, it

is unlikely more space could be saved than individually calculating a Huffman tree for each compression. There would need to be some way to cut out unnecessary middle values in a generic Huffman tree while maintaining the most important data. Since combining adjacent values did not seem to have a large affect on audio quality (See Audio Appendix 5.1.6), we decided to implement a generic Huffman tree that combines adjacent middle values to decrease size. Unfortunately, this makes any resulting compression lossy because not all values would be returned to their original value upon decoding. We used windows of 7 to combine values above 1,000 and below 65,435. This resulted in a Huffman tree of 17,418 bytes, significantly smaller than all of the individually calculated Huffman trees (See table in Figure 7). However, it is important to remember that this catastrophic decrease is due to the transition from a lossless Huffman tree to a lossy Huffman tree. Not only would the size of the Huffman tree be reduced, the generic Huffman tree would not need to be stored with the data of each file since it would be standard across files. This would mean only the sound data and header data would need to be stored for each file, reducing the total size. As an example with the use of a generic tree, the Jazz.wav file would have an improved compression ratio of 1.36 as opposed to the regular 1.16 compression ratio.

2.7 Limitations

Only music files in the WAV format were used, and only a handful of sample files were used all of which with 16 bit depth. This is not sufficient to confirm

the trends seen in this data apply on a broader scope. Further work would need to be conducted on a larger collection of sample WAV files with a larger variety (varying music type, clip length, sample rate, bit depth, etc.). The Huffman algorithm does not incorporate any of the more recent improvements such as would allow the algorithm to only make one pass through the sound data [5].

3 Data

3.1 Resulting files

The resulting files from our compression technique all remained slightly smaller than the original (as can be seen when looking at the next section on compression ratios). The size of the resulting files can be broken down into three categories: the compressed sound data, the Huffman tree information, and the WAV header information. The compressed sound data is the result of taking the data section of the WAV file and running our Huffman compression technique on it. Of course, in order to get any useful information out of the encoded compression, the Huffman tree that will be used to decode the data must also be included. Finally, the WAV header data, though extremely tiny, must also be recorded. No compression was conducted on the WAV header information due to its already small size, but for actual applications this data would likely be compressed with another technique. The discussed breakdown is enumerated for each WAV file in the table from

	Compressed Data	Tree Data	Header Data	Total Size (bytes)
Repetitive.wav	1,598,940	86,297	58	1,685,295
Chopin.wav	13,883,890	56,511	44	13,940,445
High Vocals.wav	10,817,633	59,794	44	10,877,471
Jazz.wav	11,648,081	72,635	44	11,720,760
Ambience.wav	4,181,644	52,144	58	4,233,846

Figure 7: All values are listed in bytes. The result of this compression technique has three parts: the actual data, the WAV header data, and the Huffman tree header data. This table denotes the size of each section.

Figure 7.

3.2 Compression ratios

As described earlier, the compression ratio is defined as uncompressed size over compressed size. So a compression ratio of 1 would mean the compressed file was exactly the same size. Any compression ratio greater than 1 means that the compressed file was smaller than the original. From the table in Figure 8, we see that the resulting compression ratios for our compression technique are all slightly higher than 1 but less than 2. This is fairly standard for lossless compression techniques. Lossy techniques, however, can achieve much higher compression ratios because they are storing less information. The MP3 compression technique can achieve compression ratios of 10 times or more [6].

	Original File	Compressed Data	Total w/ headers	Compression Ratio
Repetitive.wav	1.74E+06	1.60E+06	1.69E+06	1.03427
Chopin.wav	1.83E+07	1.39E+07	1.39E+07	1.31285
High Vocals.wav	1.37E+07	1.08E+07	1.09E+07	1.25678
Jazz.wav	1.36E+07	1.16E+07	1.17E+07	1.15743
Ambience.wav	4.83E+06	4.18E+06	4.23E+06	1.14197

Figure 8: All values are listed in bytes. The Original File refers to the size of the complete original WAV file in bytes. The Compressed Data column lists the size of the compressed Huffman encoded sound data in bytes; this does not include the WAV header or tree data, these are included in the next column, Total w/ headers. Finally, the compression ratios are listed in the last column, calculated with the total file size including headers.

3.3 Entropy comparison

The entropy was calculated on a character by character basis using the formula described in section 1.2.1 on Data Entropy. As described previously, the data entropy can be as the theoretical minimum possible bits necessary to store the information losslessly. (Note, however, that the entropy is simply a mathematical model and was developed specifically with the intention of describing the number of bits necessary to store English character information.) Our compression technique achieved results close to the entropy values for each WAV file, with less than a 0.3% increase in file size across the board. This, however, does not include the size of the header information (the WAV header and Huffman tree data). As seen in Figure 7, this header information can add about 50-90 kilobytes of data. When this is added, the percent

	Original File	Compressed Data	Entropy	Difference	% Increase
Repetitive.wav	1.743E+06	1.599E+06	1.596E+06	3144.275	0.1970%
Chopin.wav	1.830E+07	1.388E+07	1.385E+07	31904.840	0.2303%
High Vocals.wav	1.367E+07	1.082E+07	1.079E+07	24490.262	0.2269%
Jazz.wav	1.357E+07	1.165E+07	1.163E+07	20169.303	0.1735%
Ambience.wav	4.835E+06	4.182E+06	4.174E+06	7679.296	0.1841%

Figure 9: All values are listed in bytes. The Original File refers to the size of the complete original WAV file in bytes. The Compressed Data column lists the size of the compressed Huffman encoded sound data in bytes; this does not include the WAV header or tree data. The Difference is, in bytes, how much larger our compression technique was than the entropy estimate. The % Increase is how much larger our compressed data was than the entropy of the data.

increase for each file changes significantly for some cases. While most of the files remain below a 1% increase, the Repetitive.wav and Ambience.wav files rise to 5.61% and 1.43% respectively. This large increase when you add the header information stems from the large Huffman trees required to encode and decode these files. If you recall from the histograms in Figure 3, Repetitive.wav and Ambience.wav had a greater distribution of probabilities than the other WAV files. This causes the Huffman tree to be larger, increasing the amount of space necessary to store it.

3.4 Lossy method results

The table in Figure 10 shows the result of various lossy windowed methods on the Repetitive.wav file. As described in Section 2.5 Exploring lossy methods,

Window Size	Sound Data	Tree Data	Range Affected
1 (Lossless)	1,598,940	86,297	0 to 65535
3	1,434,817	33,908	1 to 65534
5	1,357,319	21,677	3 to 65532
7	1,304,182	16,052	4 to 65531
9	1,266,327	12,846	5 to 65530
21	1,135,062	5,983	10 to 65525

Figure 10: This table shows the result of combining adjacent values in increasing quantities for the Repetitive.wav file. Combining adjacent values dramatically reduces tree data size while also reducing total data size.

the windowed method involves rounding data values to the closest multiple of the window size (See Figure 6). The range affected refers to the 16-bit values that were not rounded to the nearest multiple of the window size. The lowest and highest 16-bit values are the most important in a WAV file; they contain the most inherent information by having the largest Shannon entropy per character. For this reason, some values at the head and tail end of this spectrum were left out of the rounding process and considered in exactly the same manner as the lossless method. To counter the adverse affects of increasing the window size, the range affected was reduced for the larger window sizes. With the window sizes and ranges listed in this table, the files sound almost exactly like the original Repetitive.wav.

4 Conclusion

Data compression, even just the field of sound data compression, is a complex topic that spans decades of academic research. From the beginning of telecommunications to the current streaming of music wirelessly over the internet, compression of sound information has been of interest. Morse Code was an early method of compression that keyed in on the need to convey information with a higher probability of occurrence with less data; this is why an E in Morse code is merely a dot while Q and Z are four "bits" long [11]. Huffman coding is built off this concept. All new scientific achievements are built off of achievements of the past. Consider that any achievement which involves mathematics is reliant on the existence of the standard mathematical models we use today that were developed a long, long time ago. None of the information discussed in this thesis is new, it builds off of concepts built up by previous humans from their attempts to understand the world. This thesis serves as an exercise of working to understand a topic as it exists in the current day and as its components have been interpreted by humans who have worked to understand it in the past.

4.1 Future Work

There are many opportunities to continue with future work on this topic. Ideally, the Huffman compression (or any compression method) would be sped up using GPU processing. For any compression method that breaks

the sound data up by time, time chunks of a certain size could be sent to the GPU for concurrent processing. For Huffman encoding, all the GPU needs to do for each data value is look up the replacement encoded value in a table. Decoding would be only slightly more complicated, due to the need to traverse the Huffman tree. This would then need to be timed in comparison to non-GPU methods in terms of both encoding and decoding time. Also more recent and elaborate versions of Huffman compression should be analyzed such as adaptive Huffman and arithmetic coding [5].

5 Audio Appendix

5.1 Accessing the Audio Appendix

Since the PDF format will only allow .mp3 files to be embedded in the document, the audio files must be accessed externally. A Google Drive folder has been set up with the following files associated to their corresponding numbers, 1 to 11. Here is the link to access the Audio Appendix Drive folder: https://drive.google.com/folderview?id=0B0kTcNNQp9hCeFBsUk8zckZCSms&usp=sharing. The permissions are set so that anyone with the link should have access. Email kfisher2@trinity.edu if you have trouble accessing these files.

5.1.1 Jazz.wav flip least significant bit

5.1.2 Jazz.wav flip most significant bit

5.1.3 Jazz.wav shift bits left by 2

5.1.4 Jazz.wav shift bits right by 2

5.1.5 Jazz.wav add offset of 2000

5.1.6 Repetitive.wav combine adjacent 3

5.1.7 Repetitive.wav

5.1.8 Chopin.wav

5.1.9 HighVocals.wav

5.1.10 Jazz.wav

5.1.11 Ambience.wav

References

[1] Guyton, A. C., & Hall, J. E. (2000). Textbook of Medical Physiology. *Philadelphia: WB Saunders Company*, 602-612.

[2] Fletcher, H. (1940). Auditory Patterns. *Reviews of Modern Physics*, 12(1), 47-66.

[3] Thomas H.. Cormen, Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to algorithms* (Vol. 6). Cambridge: MIT press.

[4] Shannon, C. E. (2001). A Mathematical Theory of Communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1), 3-55.

[5] Zoph, B., Ghazvininejad, M., & Knight, K. (2013). How Much Information Does a Human Translator Add to the Original?. Information Sciences Institute, University of Southern California.

[6] Hacker, S. (2000). *MP3: The Definitive Guide*. Sebastopol, CA: O'Reilly. Chicago

[7] Weaver, W. (1953). Recent Contributions to the Mathematical Theory of Communication. *ETC: A Review of General Semantics*, 261-281.

[8] Quantblog. (2011). Adding Circles - Javascript Animation of Fourier Series, https://quantblog.wordpress.com/2011/10/05/adding-circles-javascript-animation-of-fourier-series/

[9] Ingber, A., & Weissman, T. (2013). The Minimal Compression Rate for Similarity Identification. *arXiv preprint arXiv:1312.2063*.

[10] Perry, T. (2014, July 28). A Fictional Compression Metric Moves Into the Real World. Retrieved March 27, 2016, from http://spectrum.ieee.org/view-from-the-valley/computing/software/a-madefortv-compression-metric-moves-to-the-real-worlds

[11] Shannon, C. E. (1951). Prediction and Entropy of Printed English. *The Bell System Technical Journal*, (1). http://doi.org/10.1002/j.1538-7305.1951.tb01366.x

[12] Shannon, C. E. (1949). Communication in the presence of noise. *Proceedings of the IRE*, 37(1), 10-21.

[13] Serra, X. (1997). Musical Sound Modeling with Sinusoids Plus Noise. *Musical signal processing*, 91-122.

[14] Milar, K. (2011). *Fast Fourier Transform Analysis of Oboes, Oboe Reeds and Oboist: Which matters most to Timbre?* (Doctoral dissertation).

[15] Nelson, M., & Gailly, J. L. (1996). *The Data Compression Book* (Vol. 2). New York: M&t Books.

[16] Farrow, C. L., Shaw, M., Kim, H., Juhs, P., & Billinge, S. J. (2011). Nyquist-Shannon sampling theorem applied to refinements of the atomic pair distribution function. *Physical Review B*, 84(13), 134105.