PROJECT REPORT

DEVELOPMENT OF SOFTWARE ALGORITHM FOR COMPRESSION OF C-SCAN DATA FILES

(NON-DESTRUCTIVE TESTING APPLICATIONS)

Research By

AMITABH YADAV
B. Tech. (Electronics Engineering)

Under guidance of

S. K. LALWANI
Scientific Officer (H)
DA&PS Section, Electronics Division (E.D.)
Bhabha Atomic Research Centre (Mumbai)
Department of Atomic Energy (DAE), Government of India.

Department of Electronics, Instrumentation and Control Engineering
University of Petroleum & Energy Studies, Dehradun (INDIA).

June 3, 2016 to July 26, 2016

ACKNOWLEDGEMENTS

The satisfaction and euphoria that accompany the completion of any task would be incomplete without the mentioning of individuals whom we are greatly indebted to, who have either through guidance and/or encouragement served as a light and crowned our efforts with success.

I would like to acknowledge the contribution of my alma mater, University of Petroleum and Energy Studies (U.P.E.S.) which not only has given me a firm grounding in basics but also taught me how to build on it and widen the scope of my knowledge.

I would like to thank my Project Guide, Shri S. K. Lalwani from Bhabha Atomic Research Centre for his extensive support, in spite of his busy schedule. I am thankful to him for believing in me and providing me the opportunity to work on this project. I am grateful for his valuable suggestions and information that he provided, which helped in completing this project.

Last and most importantly, I am thankful and grateful to my loving parents and family who have always believed in me and have supported me in all walks of life and through all my decisions.

Without full support and encouragement from my family, professors and guide, the project would not have shaped the way it is. My sincere thanks and apologies to anyone who deserves credit but fail to appear in this list.

AMITABH YADAV

ABSTRACT

Ultrasonic imaging in medical, industrial and non-destructive testing (NDT) applications often requires a large amount of data collection. Consequently, it is desirable to use data compression techniques to reduce data and to facilitate the analysis and remote access of ultrasonic information. Ultrasonic Data Compression not only solves the problem of data storage but also facilitates the ease of data transmission. The precise data representation is paramount to the accurate analysis of the shape, size, and orientation of ultrasonic reflectors, as well as to the determination of the properties of the propagation path. In this project, a study of various Ultrasonic Data Compression Algorithms, primarily, lossless algorithms, is presented. The practical implementation of the Information Theory to achieve data compression is also performed and results are compiled.

KEYWORDS: DATA COMPRESSION, ASCII, NON-DESTRUCTIVE TESTING, INFORMATION THEORY AND CODING, WAVELET TRANSFORMS

LIST OF FIGURES AND TABLES

FIGURES

TABLES

TABLE OF CONTENTS

* * *

1

INTRODUCTION

The field of Ultrasonic has provided a vast scope for research in the field of Instrumentation, RADAR Technology, ***Non-Destructive Testing (NDT)*** etc. Ultrasonic test method is widely used for non-destructive examination of materials. Testing of thick metals using ultrasonic testing is often difficult due to large scattering and attenuation of the ultrasound 'echo' in the medium which results in poor signal-to-noise ratio (SNR) of the reflected signal amplitudes.

Understanding ***data compression*** requires us to understand the concept of 'information'. We all know what information is. We intuitively understand it but we consider it a qualitative concept. Information seems to be one of those entities that cannot be quantified and dealt with rigorously. There is, however, a mathematical field called ***information theory***, where information is handled quantitatively. Information theory defines how to precisely define ***redundancy***.

Compressing data is not done by stuffing or squeezing or zipping it, but by removing any redundancy that is present in the data. The concept of redundancy is central to the data compression. ***Data with redundancy can be compressed. Data without redundancy cannot be compressed, period.***

Now the natural question occurs, why redundant data is used in the first place?

There are two common types of computer data. The first type is text. These types of data are those that are non-numeric i.e. they deal with data whose elementary components are characters of text. However, the computer can process only binary information. So, each character is assigned a binary code. The present day computers use the ***ASCII code*** (short for, 'American Standard Code for Information Interchange'), although, the new computers use the UNICODE. In ASCII, each character is assigned an 8-bit code (code is of 7 bits, 8th bit is parity). A fixed size code is important because it makes it easy for software applications to handle characters of text. Also, a fixed size code is inherently redundant. In practice, files used are rarely random, and in general sense, we know that certain characters of text appear more frequently than others. ***ASCII code is redundant because it assigns to each character, common or rare, the same number of bits.*** Removing redundancy can be done by assigning variable size codes to the characters, with short codes assigned to the common characters, and long size codes assigned to the rare ones. This is precisely how Huffman Coding works.

The second type of common computer data is digital images. A digital image is a rectangular array of colored dots called pixels. Each pixel is represented by its color code. Due to the same reasons as for text, pixels are assigned codes of same size. The size of pixel depends on the number of colors in the image, and this number is normally a power of 2. If there are 2^k colors, then each pixel is a k-bit number. There are two types of redundancy in images. The first type of redundancy is similar to that of text. Other type of redundancy is more important and is the result of *pixel correlation.* If we move along pixel to pixel, we see that in most cases, adjacent pixels have similar colors. The individual pixels, therefore, are

not completely independent, and we say that neighboring pixels in an image tend to be ***correlated***.

Regardless of the method used to compress, the effectiveness of the compression depends on the amount of redundancy in the data. Most data files cannot be compressed no matter what the compression method is used. The point is that they are random and therefore there is no redundancy. This may seem strange first because we compress data files all the time. This is simply because, they have redundancy, are non-random and therefore useful and interesting.

In order to compress a data file, the compression algorithm has to examine the data, find redundancies in it, and try to remove them. Since the redundancies depends on the type of data (text, image, sound…) any compression method has to be developed for a specific type of data and works best on this type. ***There is no such thing as universal, efficient data compression algorithm.***

Some important terms in the field of data compression are:

- The *compressor* or *encoder* is the program that compresses the raw data in the input file and creates an output file with compressed (low-redundancy) data.
- The *decompressor* or *decoder* is the program that takes the compressed (low redundancy) file and regenerates the original file.
- *Non-Adaptive compression* method is rigid and does not modify its operations, its parameters, or its tables in response to a particular data being compressed. Such method is best to compress data that is of a single type. E.g. Facsimile compression.
- *Adaptive compression* method examines the raw data and modifies its operations and/or its parameters accordingly. E.g. Adaptive Huffman Coding.
- Some compression methods use a two-pass algorithm, where the first pass reads the input file to collect statistics on the data to be compressed and the second pass does the actual compressing using the parameters or codes set by the first pass. Such a method may be called *semi-adaptive compression*.
- A data compression method can also be *locally adaptive*, meaning, it adapts itself to local condtions in the input file and varies this adaptation as it moves from area to area in the input. An example is the move to front method.
- *Lossy/Lossless Compression*: Certain compression methods are lossy. They achieve better compression by losing some information. When the compressed file is decompressed, the result is not identical to the original data. Such a method makes sense, when compressing movies, images or sound. If the loss of data is small, we may not be able to tell the difference. In contrast, text files, specially files containing computer programs, may become worthless if even a single bit is missing and therefore, they must be compressed only by lossless algorithms.
- *Symmetric Compression* is the case where the compressor and decompressor use the same basic algorithm but work in the "opposite" direction.
- *Compression Performance*: There are several quantities to express compression performance:
  1. *The Compression Ratio*:

$$Compression\ Ratio = \frac{Size\ of\ Output\ File}{Size\ of\ Input\ File}$$

A value of 0.6 means the data occupies 60% of its original size after compression. Value greater than 1, indicates an output file greater than the size of input file! (negative compression).
The compression ratio is also called *bpb (bit-per-bit);* in image compression it is called *bpp (bit-per-pixel)*; and in modern efficient text compression methods, it is called *bpc (bits-per-character)*.

2. The inverse of compression ratio is called *Compression Factor*:
$$Compression\ Factor = \frac{Size\ of\ Input\ File}{Size\ of\ Output\ File}$$
In this case, value greater than 1 indicates compression and less than 1 indicates expansion.
3. The expression, *compression*: $100\times(1 - compression\ ratio)$ is also a reasonable measure of the compression performance. A value of 70 means the output file occupies 30% of its original size (or, compression has resulted in the savings of 70%).
In this report, we are using this expression to demonstrate the performance of the compression algorithm.

- *The probability model*: This concept is important in the statistical data compression methods. Sometimes, a compression algorithm consists of two parts, a probability model and the compressor itself. Before the next data item is compressed, the model is invoked and is asked to estimate the probability of the data item. The item and the probability of the data item are then sent to the compressor, which uses the estimated probability to compress the item. The better the probability, the better the item is compressed.
- The term *alphabet* refers to the set of symbols in the data being compressed. An alphabet may consist of two bits 0 and 1, of the 128 ASCII characters, of the 256 possible 8-bit bytes, or of any other symbols.

On a final note, we can conclude that the performance of any compression method is limited. No compression method can compress all data files efficiently. It seems reasonable to define efficient compression as compressing a file to at most half its original size. Thus, we may be satisfied (even happy) if we discover a sophisticated compression algorithm that compresses any data file to a file of size less than or equal to half the size of the original file. This sad conclusion should not however cause the reader to close the work with sigh and turn to more promising pursuits because the files that we actually want to compress are normally the ones that compress well. The ones that compress badly are normally random or close to random therefore, uninteresting, unimportant and not the candidates for compression, transmission or storage.

1.1 PROJECT DESCRIPTION

The project entitled, 'Development of Ultrasonic-Data Compression Algorithm for Non-Destructive Testing Applications' is a project done in the Electronics Division (ED) of Bhabha Atomic Research Centre (BARC).

The term 'data', from here on, refers to the ultrasonic data.

Testing on a single point on the surface of metal, generates a data file containing ultrasonic echo data on that point which contains number of samples identified by the quantity ***MSPS (millions of samples per second).*** A single run on the point generates an A-Scan.

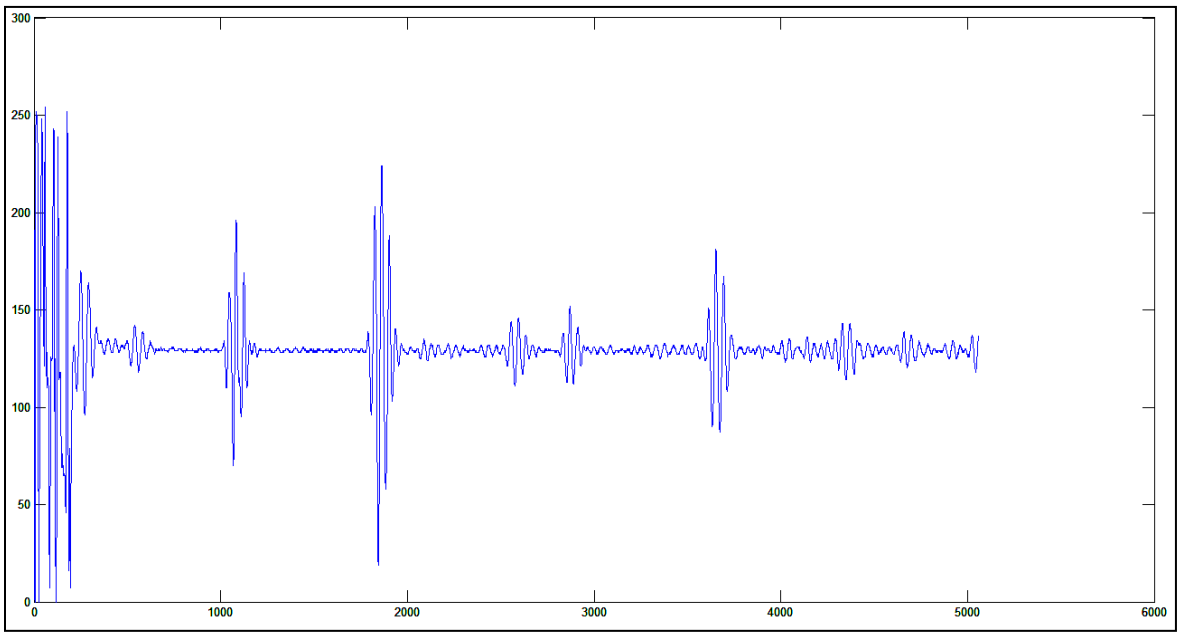

Figure 1: A-Scan Data (2MHz/50MSPS) (Source: ED, BARC)

An ***A-Scan*** data contains information of the received echo at the receiving transducer after the ultrasonic pulse is scattered from the internal defect. This signal is then passed on to amplifier and then digitizer for sampling. The digitized data are then stored in the ASCII format. The generated file is thus an ASCII file identified by the extension, ***.asc***.

An A-Scan is the basic form of data obtained from NDT. Practically, on a piece of metal, the NDT is performed linearly in one dimension or in two dimensions. The echo signal is obtained from single point, then the transducer head moves to the next point in that dimension up to the distance according to the set precision. In such a way, data is acquired for each point and output is generated. Such a data is called a ***B-Scan***. The B-Scan files are also ASCII files, identified by the extension, ***.bsn***.

In a similar way, as a B-Scan, data is generated and gathered for various points on the surface of the metal. A-Scan is performed in one-dimension, which gives the B-Scan data and then continuing in the second dimension to cover the whole surface of the metal. As a result, the generated file we obtain is called a ***C-Scan***. The C-Scan files are also ASCII files, identified by the extension, ***.csn***.

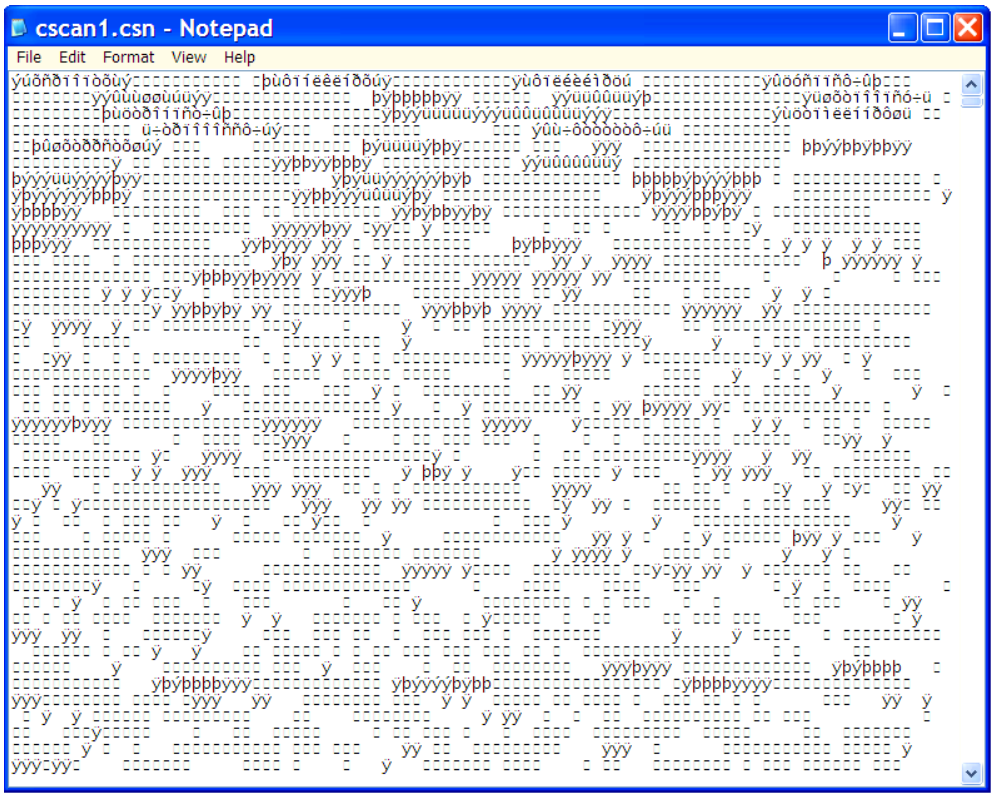

Figure 2: Sample C-Scan data file (Source: ED, BARC)

All three types of obtained results are of scientific importance. But, practically and mainly, C-Scans are performed.

The data file for a C-Scan can range from a few Kilobytes (KBs) to several Gigabytes (GBs), depending on the MSPS settings and area of the metal. Handling and transmitting such a huge amount of data is difficult and also creates the problem of storage. An efficient data compression method makes use of a compression algorithm for this type of file (ASCII file) to identify and reduce the redundancies in the data. This solves the problem of storage.

1.2 IMPORTANCE

- At BARC, the NDT performed for metals is carried out and saved for future reference and analysis. Thus, all information is required to be saved.
- A data compression algorithm for the 'specific' type of data (viz. Ultrasonic Data) is a promising way to achieve compression. This largely solves the problem of storage and transmission, especially, for those files that need to be kept as a record.
- An analysis of A-Scan and C-Scan signals showed that there are large amount of redundancies in the Ultrasonic Data - mostly constant signals with ultrasonic echo information at the points only where defects are detected. (The constant signal is due to the fact that when fewer defects are present, all ultrasonic signals pass through the metal and none is received back as there was no scattering.)

- Since ultrasonic data type is clearly specified (ASCII format) and we are aware that there are significant redundancies in the data, it shows the promise of developing an efficient data compression algorithm. (And as it will be later demonstrated that compression up to 90% can be obtained, in certain cases).
- The compression algorithm initially has the promise to solve the storage and handling problem of large data files. It can further be implemented on an FPGA or can be fabricated as an Application Specific Integrated Circuit (ASIC) which readily delivers ultrasonic data in the compressed form. For analysis, one can simply decompress the data first and then analyze.

1.3 OBJECTIVES

- Developing and efficient data compression algorithm for ultrasonic data.
- **Analyzing various lossless data compression algorithms and their performance in different probability models.**
- Study of Lossy Compression Algorithms and understand the effect and extent of losses to the data.
- Presenting conclusion and possibility of practical implementation and of future developments on the same.

2

LITERATURE REVIEW

Significant work has been done previously by esteemed academicians and researchers in the field of Data Compression. This is a topic well discussed and implemented in the field of data handling and data communication.

Data compression is defined for various kinds of data – text, image, sound, video etc. And it can be defined under various categories based on the type of data that needs to be compressed. Similarly, there are types of data that can bear 'certain' losses and still be useful. E.g. JPEG (Joint Photographic Experts Group) compression in case of Images used a lossy compression algorithm without significantly hampering the data, and still rendering it useful for use. On the other hand, Statistical data compression methods use statistical properties of data being compressed to assign variable size codes to the individual symbols to the data. This method is lossless and decompressing the same, gives back the original file.

Information theory is the creation, in 1948, of Claude Shannon of Bell Laboratories.

The first rule of assigning variable size codes is, Short codes should be assigned to the common symbols and Long codes should be assigned to the rare symbols.

Statistical Method are usually the Lossless Compression Methods, such as, Huffman Coding and Arithmetic Coding.

2.1 RELATED WORKS

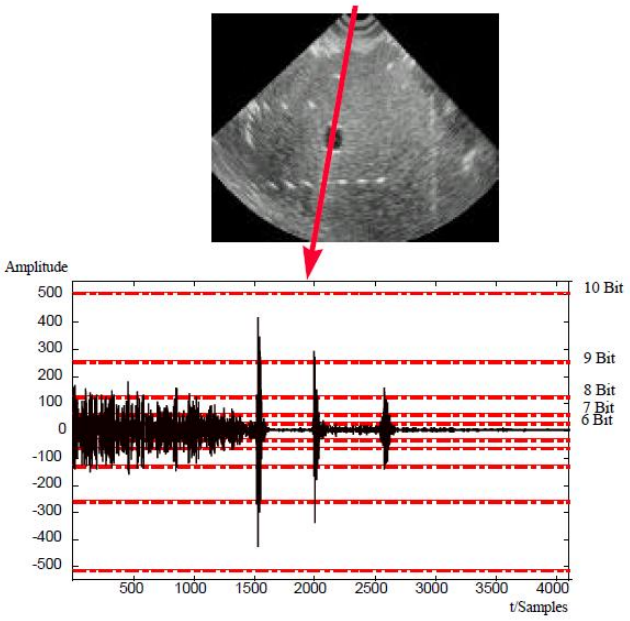

Figure 3: Sample A Scan of a Speckle Phantom [2]

Ultrasonic imaging in medical and industrial applications often requires a large amount of data collection. Consequently, it is desirable to use data compression techniques to reduce data and to facilitate the analysis and remote access of ultrasonic information. In case of Ultrasonic Signals, one can afford to bear certain losses. a successive parameter estimation algorithm based on a modified version of the continuous wavelet transform (CWT) to compress and de-noise ultrasonic signals is developed [1]. Here, a modified CWT (MCWT) based on the Gabor-Helstrom transform is introduced as a means to exactly estimate both time-of-arrival and center frequency of ultrasonic echoes.

A compression algorithm is developed, which utilizes the special properties of ultrasonic radio frequency (RF) data. The compression is done in two steps:

First, linear predictive coding (LPC) is applied, using an one-step-predictor. Further, the remaining error of the prediction is stored using only the necessary word length to store the signal. It is also further developed as a lossy compression method. [2]

Ultrasonic Non-Destructive Techniques often require capturing massive amounts of data and intensive signal processing for processes such as image formation, analysis, characterization, classification and diagnosis. One of the studies is focused around utilizing a Reconfigurable Ultrasonic System-on-Chip Hardware (RUSH) platform to process ultrasonic signals in real-time. OpenCL has been utilized in the RUSH platform to provide a means to accelerate computation using the FPGA fabric while providing consistent memory and execution models to enable portability. [3]

Two methods for ultrasonic signal compression which offer high signal fidelity are Discrete Wavelet Transform and signal decimation with the Nyquist rate limit. The compression algorithm can be implemented on a reconfigurable system-on-chip platform using programmable hardware logic as well as in software using an embedded processor. [4].

We can model ultrasonic backscattered echoes in terms of superimposed Gaussian echoes corrupted by noise. Each Gaussian echo in the model is a nonlinear function of a set of parameters: echo bandwidth, arrival time, center frequency, amplitude, and phase. These parameters are sensitive to the echo shape and can be linked to the physical properties of reflectors and frequency characteristics of the propagation path. We address the estimation of these parameters using the maximum likelihood estimation (MLE) principle, assuming that all of the parameters describing the shape of the echo are unknown but deterministic. [5].

Compression can also be carried out systematically. A systematic design flow consisting of two main stages for ultrasonic signal compression. The first stage of the algorithm is concerned with finding the maximum energy compaction. The second stage of the algorithm is concerned with the coefficient reduction using thresholding techniques. [6]

3

THEORY

4.1 DATA COMPRESSION

Data compression is often referred to as coding, where coding is a very general term encompassing any special representation of data which satisfies a given need. Information theory is defined to be the study of efficient coding and its consequences, in the form of speed of transmission and probability of error. Data compression may be viewed as a branch of information theory in which the primary objective is to minimize the amount of data to be transmitted.

Information theory is the creation, in 1948, of Claude Shannon of Bell Laboratories. This theory came upon the world as a surprise because we normally consider information as a qualitative concept. The only information-theoretical concept needed to understand data compression is ***entropy***.

Entropy of a symbol *a* with probability P is interpreted as the amount of information included in a, and is defined as $-P log_2 P$.

4.2 LOSSLESS COMPRESSION ALGORITHMS

Lossless data compression algorithms usually exploit statistical redundancy to represent data without losing any information, so that the process is reversible. Lossless compression is possible because most real-world data exhibits statistical redundancy. For example, an image may have areas of color that do not change over several pixels; instead of coding "red pixel, red pixel, ..." the data may be encoded as "279 red pixels". This is a basic example of ***run-length encoding***; there are many schemes to reduce file size by eliminating redundancy.

The ***Lempel–Ziv (LZ) compression*** methods are among the most popular algorithms for lossless storage. ***DEFLATE*** is a variation on LZ optimized for decompression speed and compression ratio, but compression can be slow. DEFLATE is used in PKZIP, Gzip, and PNG. LZW (Lempel–Ziv–Welch) is used in GIF images. Also noteworthy is the LZR (Lempel-Ziv–Renau) algorithm, which serves as the basis for the Zip method. LZ methods use a table-based compression model where table entries are substituted for repeated strings of data. For most LZ methods, this table is generated dynamically from earlier data in the input. The table itself is often ***Huffman encoded*** (e.g. SHRI, LZX). Current LZ-based coding schemes that perform well are Brotli and LZX. LZX is used in Microsoft's CAB format.

The best modern lossless compressors use probabilistic models, such as prediction by partial matching. The ***Burrows–Wheeler transform*** can also be viewed as an indirect form of statistical modeling.

The class of ***grammar-based codes*** is gaining popularity because they can compress highly repetitive input extremely effectively, for instance, a biological data collection of the same or closely related species, a huge versioned document collection, internet archival, etc. The basic task of grammar-based codes is constructing a context-free grammar deriving a single string. Sequitur and Re-Pair are practical grammar compression algorithms for which software is publicly available.

In a further refinement of the direct use of probabilistic modeling, statistical estimates can be coupled to an algorithm called arithmetic coding. ***Arithmetic coding*** is a more modern coding technique that uses the mathematical calculations of a finite-state machine to produce a string of encoded bits from a series of input data symbols. It can achieve superior compression to other techniques such as the better-known ***Huffman algorithm***. It uses an internal memory state to avoid the need to perform a one-to-one mapping of individual input symbols to distinct representations that use an integer number of bits, and it clears out the internal memory only after encoding the entire string of data symbols. ***Arithmetic coding applies especially well to adaptive data compression tasks where the statistics vary and are context-dependent, as it can be easily coupled with an adaptive model of the probability distribution of the input data.*** An early example of the use of arithmetic coding was its use as an optional (but not widely used) feature of the JPEG image coding standard. It has since been applied in various other designs including H.264/MPEG-4 AVC and HEVC for video coding.

4.3 LOSSY COMPRESSION ALGORITHMS

Lossy data compression is the converse of lossless data compression. In these schemes, some loss of information is acceptable. Dropping nonessential detail from the data source can save storage space. Lossy data compression schemes are designed by research on how people perceive the data in question. For example, the human eye is more sensitive to subtle variations in luminance than it is to the variations in color. JPEG image compression works in part by rounding off nonessential bits of information. There is a corresponding trade-off between preserving information and reducing size. A number of popular compression formats exploit these perceptual differences, including those used in music files, images, and video.

Lossy image compression can be used in digital cameras, to increase storage capacities with minimal degradation of picture quality. Similarly, DVDs use the lossy MPEG-2 video coding format for video compression.

In lossy audio compression, methods of psychoacoustics are used to remove non-audible (or less audible) components of the audio signal. Compression of human speech is often performed with even more specialized techniques; speech coding, or voice coding, is sometimes distinguished as a separate discipline from audio compression. Different audio and speech compression standards are listed under audio coding formats. Voice

compression is used in internet telephony, for example, audio compression is used for CD ripping and is decoded by the audio players.

4.4 SHANNON-FANO CODING

The Shannon-Fano technique has as an advantage its simplicity. The code is constructed as follows: the source messages *a(i)* and their probabilities *p(a(i))* are listed in order of non-increasing probability. This list is then divided in such a way as to form two groups of as nearly equal total probabilities as possible. Each message in the first group receives 0 as the first digit of its codeword; the messages in the second half have code words beginning with 1. Each of these groups is then divided according to the same criterion and additional code digits are appended. The process is continued until each subset contains only one message. Clearly the Shannon-Fano algorithm yields a minimal prefix code.

a	1/2	0
b	1/4	10
c	1/8	110
d	1/16	1110
e	1/32	11110
f	1/32	11111

Table 1: A Shannon-Fano Code.

Table 1 shows the application of the method to a particularly simple probability distribution. The length of each codeword x is equal to -lg p(x). This is true as long as it is possible to divide the list into subgroups of exactly equal probability. When this is not possible, some code words may be of length -lg p(x)+1. The Shannon-Fano algorithm yields an average codeword length S which satisfies H <= S <= H + 1. Note that, the Shannon-Fano algorithm is not guaranteed to produce an optimal code.

4.5 STATIC HUFFMAN CODING

Huffman's algorithm, expressed graphically, takes as input a list of nonnegative weights {w(1), ... ,w(n) } and constructs a full binary tree [a binary tree is full if every node has either zero or two children] whose leaves are labeled with the weights. When the Huffman algorithm is used to construct a code, the weights represent the probabilities associated with the source letters. Initially there is a set of singleton trees, one for each weight in the list. At each step in the algorithm the trees corresponding to the two smallest weights, w(i) and w(j), are merged into a new tree whose weight is w(i)+w(j) and whose root has two

children which are the sub-trees represented by w(i) and w(j). The weights w(i) and w(j) are removed from the list and w(i)+w(j) is inserted into the list. This process continues until the weight list contains a single value. If, at any time, there is more than one way to choose a smallest pair of weights, any such pair may be chosen. In Huffman's paper, the process begins with a non-increasing list of weights. This detail is not important to the correctness of the algorithm, but it does provide a more efficient implementation [Huffman 1952]. The Huffman algorithm is demonstrated in Figure 4.

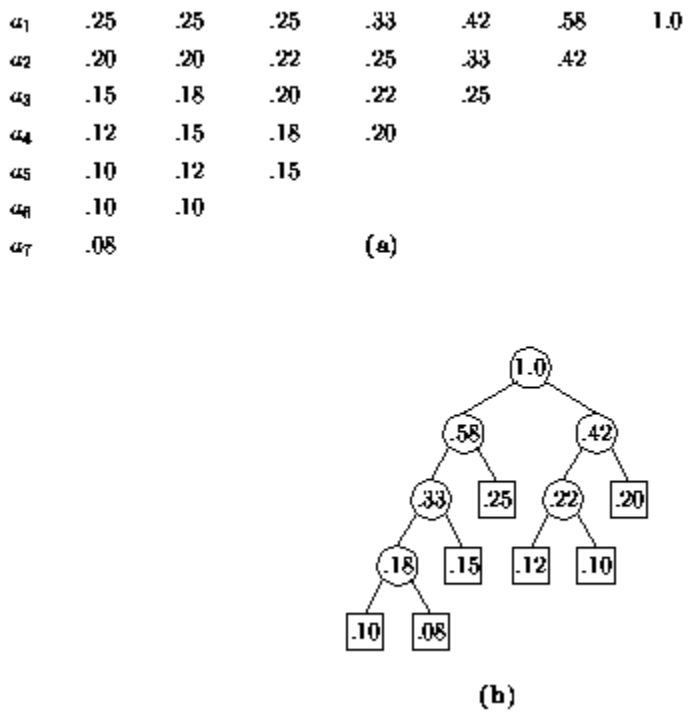

Figure 4 The Huffman process: (a) The list (b) the tree.

The Huffman algorithm determines the lengths of the codewords to be mapped to each of the source letters a(i). There are many alternatives for specifying the actual digits; it is necessary only that the code have the prefix property. The usual assignment entails labeling the edge from each parent to its left child with the digit 0 and the edge to the right child with 1. The codeword for each source letter is the sequence of labels along the path from the root to the leaf node representing that letter. The codewords for the source of Figure 4, in order of decreasing probability, are {01,11,001,100,101,000,0001}. Clearly, this process yields a minimal prefix code. ***Further, the algorithm is guaranteed to produce an optimal (minimum redundancy) code [Huffman 1952]***. Gallager has proved an upper bound on the redundancy of a Huffman code of p(n) + lg [(2 lg e)/e] which is approximately p(n) + 0.086, where p(n) is the probability of the least likely source message [Gallager 1978]. In a recent paper, Capocelli et al. provide new bounds which are tighter than those

of Gallagher for some probability distributions [Capocelli et al. 1986]. Figure 5 shows a distribution for which the Huffman code is optimal while the Shannon-Fano code is not.

In addition to the fact that there are many ways of forming code words of appropriate lengths, there are cases in which the Huffman algorithm does not uniquely determine these lengths due to the arbitrary choice among equal minimum weights. As an example, codes with codeword lengths of {1,2,3,4,4} and of {2,2,2,3,3} both yield the same average codeword length for a source with probabilities {.4,.2,.2,.1,.1}. Schwartz defines a variation of the Huffman algorithm which performs "bottom merging"; that is, orders a new parent node above existing nodes of the same weight and always merges the last two weights in the list. The code constructed is the Huffman code with minimum values of maximum codeword length (MAX{ l(i) }) and total codeword length (SUM{ l(i) }) [Schwartz 1964]. Schwartz and Kallick describe an implementation of Huffman's algorithm with bottom merging [Schwartz and Kallick 1964]. The ***Schwartz-Kallick algorithm*** and a later algorithm by Connell [Connell 1973] use Huffman's procedure to determine the lengths of the code words, and actual digits are assigned so that the code has the numerical sequence property. That is, code words of equal length form a consecutive sequence of binary numbers. Shannon-Fano codes also have the numerical sequence property. This property can be exploited to achieve a compact representation of the code and rapid encoding and decoding.

		S-F	Huffman
a(1)	0.35	00	1
a(2)	0.17	01	011
a(3)	0.17	10	010
a(4)	0.16	110	001
a(5)	0.15	111	000
Average codeword length		2.31	2.30

Table 2 Comparison of Shannon-Fano and Huffman Codes

Both the Huffman and the Shannon-Fano mappings can be generated in O(n) time, where n is the number of messages in the source ensemble (assuming that the weights have been presorted). Each of these algorithms maps a source message a(i) with probability p to a codeword of length l (-lg p <= l <= - lg p + 1). Encoding and decoding times depend upon the representation of the mapping. If the mapping is stored as a binary tree, then decoding the codeword for a(i) involves following a path of length l in the tree. A table indexed by the source messages could be used for encoding; the code for a(i) would be stored in position i of the table and encoding time would be O(l). Connell's algorithm makes use of the index of the Huffman code, a representation of the distribution of

codeword lengths, to encode and decode in O(c) time where c is the number of different codeword lengths. Tanaka presents an implementation of Huffman coding based on finite-state machines which can be realized efficiently in either hardware or software [Tanaka 1987].

As noted earlier, the redundancy bound for Shannon-Fano codes is 1 and the bound for the Huffman method is p(n + 0.086 where p(n) is the probability of the least likely source message (so p(n) is less than or equal to .5, and generally much less). It is important to note that in defining redundancy to be average codeword length minus entropy, the cost of transmitting the code mapping computed by these algorithms is ignored. The overhead cost for any method where the source alphabet has not been established prior to transmission includes n lg n bits for sending the n source letters. For a Shannon-Fano code, a list of code words ordered so as to correspond to the source letters could be transmitted. The additional time required is then SUM l(i), where the l(i) are the lengths of the code words. For Huffman coding, an encoding of the shape of the code tree might be transmitted. Since any full binary tree may be a legal Huffman code tree, encoding tree shape may require as many as lg 4^n = 2n bits. In most cases the message ensemble is very large, so that the number of bits of overhead is minute by comparison to the total length of the encoded transmission. However, it is imprudent to ignore this cost.

If a less-than-optimal code is acceptable, the overhead costs can be avoided through a prior agreement by sender and receiver as to the code mapping. Rather than using a Huffman code based upon the characteristics of the current message ensemble, the code used could be based on statistics for a class of transmissions to which the current ensemble is assumed to belong. That is, both sender and receiver could have access to a codebook with k mappings in it; one for Pascal source, one for English text, etc. The sender would then simply alert the receiver as to which of the common codes he is using. This requires only lg k bits of overhead. Assuming that classes of transmission with relatively stable characteristics could be identified, this hybrid approach would greatly reduce the redundancy due to overhead without significantly increasing expected codeword length. In addition, the cost of computing the mapping would be amortized over all files of a given class. That is, the mapping would be computed once on a statistically significant sample and then used on a great number of files for which the sample is representative. There is clearly a substantial risk associated with assumptions about file characteristics and great care would be necessary in choosing both the sample from which the mapping is to be derived and the categories into which to partition transmissions. An extreme example of the risk associated with the codebook approach is provided by author Ernest V. Wright who wrote a novel Gadsby (1939) containing no occurrences of the letter E. Since E is the most commonly used letter in the English language, an encoding based upon a sample from Gadsby would be disastrous if used with "normal" examples of English text. Similarly, the "normal" encoding would provide poor compression of Gadsby.

McIntyre and Pechura describe an experiment in which the codebook approach is compared to static Huffman coding [McIntyre and Pechura 1985]. The sample used for comparison is a collection of 530 source programs in four languages. The codebook contains a Pascal code tree, a FORTRAN code tree, a COBOL code tree, a PL/1 code tree, and an ALL code tree. The Pascal code tree is the result of applying the static Huffman algorithm to the combined character frequencies of all of the Pascal programs in the sample. The ALL code tree is based upon the combined character frequencies for all of the programs. The experiment involves encoding each of the programs using the five codes in the codebook and the static Huffman algorithm. The data reported for each of the 530 programs consists of the size of the coded program for each of the five predetermined codes, and the size of the coded program plus the size of the mapping (in table form) for the static Huffman method. In every case, the code tree for the language class to which the program belongs generates the most compact encoding. Although using the Huffman algorithm on the program itself yields an optimal mapping, the overhead cost is greater than the added redundancy incurred by the less-than-optimal code. In many cases, the ALL code tree also generates a more compact encoding than the static Huffman algorithm. In the worst case, an encoding constructed from the codebook is only 6.6% larger than that constructed by the Huffman algorithm. These results suggest that, for files of source code, the codebook approach may be appropriate.

Gilbert discusses the construction of Huffman codes based on inaccurate source probabilities [Gilbert 1971]. A simple solution to the problem of incomplete knowledge of the source is to avoid long code words, thereby minimizing the error of underestimating badly the probability of a message. The problem becomes one of constructing the optimal binary tree subject to a height restriction (see [Knuth 1971; Hu and Tan 1972; Garey 1974]). Another approach involves collecting statistics for several sources and then constructing a code based upon some combined criterion. This approach could be applied to the problem of designing a single code for use with English, French, German, etc., sources. To accomplish this, Huffman's algorithm could be used to minimize either the average codeword length for the combined source probabilities; or the average codeword length for English, subject to constraints on average codeword lengths for the other sources.

4.6 ARITHMETIC CODING

The method of arithmetic coding was suggested by Elias, and presented by Abramson in his text on Information Theory [Abramson 1963]. Implementations of Elias' technique were developed by Rissanen, Pasco, Rubin, and, most recently, Witten et al. [Rissanen 1976; Pasco 1976; Rubin 1979; Witten et al. 1987]. We present the concept of arithmetic coding first and follow with a discussion of implementation details and performance.

In arithmetic coding a source ensemble is represented by an interval between 0 and 1 on the real number line. Each symbol of the ensemble narrows this interval. As the interval becomes smaller, the number of bits needed to specify it grows. Arithmetic coding assumes an explicit probabilistic model of the source. It is a defined-word scheme which uses the probabilities of the source messages to successively narrow the interval used to represent the ensemble. A high probability message narrows the interval less than a low probability message, so that high probability messages contribute fewer bits to the coded ensemble. The method begins with an unordered list of source messages and their probabilities. The number line is partitioned into subintervals based on cumulative probabilities.

A small example will be used to illustrate the idea of arithmetic coding. Given source messages {A,B,C,D,#} with probabilities {.2, .4, .1, .2, .1}, Table 3 demonstrates the initial partitioning of the number line. The symbol A corresponds to the first 1/5 of the interval [0,1); B the next 2/5; D the subinterval of size 1/5 which begins 70% of the way from the left endpoint to the right. When encoding begins, the source ensemble is represented by the entire interval [0,1). For the ensemble AADB#, the first A reduces the interval to [0,.2) and the second A to [0,.04) (the first 1/5 of the previous interval). The D further narrows the interval to [.028,.036) (1/5 of the previous size, beginning 70% of the distance from left to right). The B narrows the interval to [.0296,.0328), and the # yields a final interval of [.03248,.0328). The interval, or alternatively any number i within the interval, may now be used to represent the source ensemble.

Source message	Probability	Cumulative probability	Range
A	.2	.2	[0,.2)
B	.4	.6	[.2,.6)
C	.1	.7	[.6,.7)
D	.2	.9	[.7,.9)
#	.1	1.0	[.9,1.0)

Table 3 The Arithmetic coding model

Two equations may be used to define the narrowing process described above:

newleft = prevleft + msgleft*prevsize (1)
newsize = prevsize * msgsize (2)

The first equation states that the left endpoint of the new interval is calculated from the previous interval and the current source message. The left endpoint of the range associated with the current message specifies what percent of the previous interval to remove from the left in order to form the new interval. For *D* in the above example, the new left endpoint is moved over by .7 * .04 (70% of the size of the previous interval). The second equation computes the size of the new interval from the previous interval size and

the probability of the current message (which is equivalent to the size of its associated range). Thus, the size of the interval determined by *D* is .04*.2, and the right endpoint is .028+.008=.036 (left endpoint + size).

The size of the final subinterval determines the number of bits needed to specify a number in that range. The number of bits needed to specify a subinterval of [0,1) of size *s* is -lg *s*. Since the size of the final subinterval is the product of the probabilities of the source messages in the ensemble (that is, *s*=PROD{*i*=1 to *N*} *p*(source message *i*) where *N* is the length of the ensemble), we have -lg *s* = -SUM{*i*=1 to *N* lg*p*(source message *i*) = - SUM{*i*=1 to *n*} *p*(*a*(*i*)) lg *p*(*a*(*i*)), where *n* is the number of unique source messages *a*(1), *a*(2), ... *a*(*n*). Thus, the number of bits generated by the arithmetic coding technique is exactly equal to entropy, *H*. This demonstrates the fact that arithmetic coding achieves compression which is almost exactly that predicted by the entropy of the source.

In order to recover the original ensemble, the decoder must know the model of the source used by the encoder (eg., the source messages and associated ranges) and a single number within the interval determined by the encoder. Decoding consists of a series of comparisons of the number *i* to the ranges representing the source messages. For this example, *i* might be .0325 (.03248, .0326, or .0327 would all do just as well). The decoder uses *i* to simulate the actions of the encoder. Since *i* lies between 0 and .2, he deduces that the first letter was *A* (since the range [0,.2) corresponds to source message *A*). This narrows the interval to [0,.2). The decoder can now deduce that the next message will further narrow the interval in one of the following ways: to [0,.04) for *A*, to [.04,.12) for *B*, to [.12,.14) for *C*, to [.14,.18) for *D*, and to [.18,.2) for *#*. Since *i* falls into the interval [0,.04), he knows that the second message is again *A*. This process continues until the entire ensemble has been recovered.

Several difficulties become evident when implementation of arithmetic coding is attempted. The first is that the decoder needs some way of knowing when to stop. As evidence of this, the number 0 could represent any of the source ensembles *A, AA, AAA,* etc. Two solutions to this problem have been suggested. One is that the encoder transmit the size of the ensemble as part of the description of the model. Another is that a special symbol be included in the model for the purpose of signaling end of message. The # in the above example serves this purpose. The second alternative is preferable for several reasons. First, sending the size of the ensemble requires a two-pass process and precludes the use of arithmetic coding as part of a hybrid codebook scheme. Secondly, adaptive methods of arithmetic coding are easily developed and a first pass to determine ensemble size is inappropriate in an on-line adaptive scheme.

A second issue left unresolved by the fundamental concept of arithmetic coding is that of incremental transmission and reception. It appears from the above discussion that the encoding algorithm transmits nothing until the final interval is determined. However, this delay is not necessary. As the interval narrows, the leading bits of the left and right endpoints become the same. Any leading bits that are the same may be transmitted immediately, as they will not be affected by further narrowing. A third issue is that of precision. From the description of arithmetic coding it appears that the precision required grows without bound as the length of the ensemble grows. Witten et al. and Rubin address

this issue [Witten et al. 1987; Rubin 1979]. Fixed precision registers may be used as long as underflow and overflow are detected and managed. The degree of compression achieved by an implementation of arithmetic coding is not exactly *H*, as implied by the concept of arithmetic coding. Both the use of a message terminator and the use of fixed-length arithmetic reduce coding effectiveness. However, it is clear that an end-of-message symbol will not have a significant effect on a large source ensemble. Witten et al. approximate the overhead due to the use of fixed precision at 10^-4 bits per source message, which is also negligible.

4.7 WAVELET COMPRESSION

Wavelet compression is a form of data compression well suited for image compression (sometimes also video compression and audio compression). Notable implementations are JPEG 2000, DjVu and ECW for still images, CineForm, and the BBC's Dirac. The goal is to store image data in as little space as possible in a file. Wavelet compression can be either lossless or lossy.

Using a wavelet transform, the wavelet compression methods are adequate for representing transients, such as percussion sounds in audio, or high-frequency components in two-dimensional images, for example an image of stars on a night sky. This means that the transient elements of a data signal can be represented by a smaller amount of information than would be the case if some other transform, such as the more widespread ***discrete cosine transform***, had been used.

Discrete wavelet transform has been successfully applied for the compression of electrocardiograph (ECG) signals. In this work, the high correlation between the corresponding wavelet coefficients of signals of successive cardiac cycles is utilized employing linear prediction.

Wavelet compression is not good for all kinds of data: transient signal characteristics mean good wavelet compression, while smooth, periodic signals are better compressed by other methods, particularly traditional harmonic compression (frequency domain, as by Fourier transforms and related).

4.7.1 METHOD OF WAVELET COMPRESSION

First a wavelet transform is applied. This produces as many coefficients as there are pixels in the image (i.e. there is no compression yet since it is only a transform). These coefficients can then be compressed more easily because the information is statistically concentrated in just a few coefficients. This principle is called transform coding. After that, the coefficients are quantized and the quantized values are entropy encoded and/or run length encoded. A few 1D and 2D applications of wavelet compression use a technique called "wavelet footprints".

4.8 (GNU) ZIP ALGORITHM (LEMPEL-ZIV ALGORITHM): A CASE STUDY [7]

The deflation algorithm used by zip and gzip is a variation of LZ77 (Lempel-Ziv 1977 [7]). It finds duplicated strings in the input data.

The second occurrence of a string is replaced by a pointer to the previous string, in the form of a pair (distance, length).

Distances are limited to 32K bytes, and lengths are limited to 258 bytes. When a string does not occur anywhere in the previous32K bytes, it is emitted as a sequence of literal bytes.

(In this description, 'string' must be taken as an arbitrary sequence of bytes, and is not restricted to printable characters.)

Literals or match lengths are compressed with one Huffman tree, and match distances are compressed with another tree. The trees are stored in a compact form at the start of each block. The blocks can have any size (except that the compressed data for one block must fit in available memory). A block is terminated when zip determines that it would be useful to start another block with fresh trees. (This is somewhat similar to compress.)

Duplicated strings are found using a hash table. All input strings of length 3 are inserted in the hash table. A hash index is computed for the next 3 bytes. If the hash chain for this index is not empty, all strings in the chain are compared with the current input string, and the longest match is selected.

The hash chains are searched starting with the most recent strings, to favor small distances and thus take advantage of the Huffman encoding. The hash chains are singly linked. There are no deletions from the hash chains, the algorithm simply discards matches that are too old.

To avoid a worst-case situation, very long hash chains are arbitrarily truncated at a certain length, determined by a runtime option (zip -1to -9). So zip does not always find the longest possible match but generally finds a match which is long enough.

Zip also defers the selection of matches with a lazy evaluation mechanism. After a match of length N has been found, zip searches for a longer match at the next input byte. If a longer match is found, the previous match is truncated to a length of one (thus producing a single literal byte) and the longer match is emitted afterwards.

Otherwise, the original match is kept, and the next match search is attempted only N steps later.

The lazy match evaluation is also subject to a runtime parameter. If the current match is long enough, zip reduces the search for a longer match, thus speeding up the whole

process. If compression ratio is more important than speed, zip attempts a complete second search even if the first match is already long enough.

The lazy match evaluation is no performed for the fastest compression modes (speed options -1 to -3). For these fast modes, new strings are inserted in the hash table only when no match was found, or when the match is not too long. This degrades the compression ratio but saves time since there are both fewer insertions and fewer searches.

2. gzip file format

The pkzip format imposes a lot of overhead in various headers, which are useful for an archiver but not necessary when only one file is compressed. gzip uses a much simpler structure. Numbers are in littleendian format, and bit 0 is the least significant bit. A gzip file is a sequence of compressed members. Each member has the following structure:

2 bytes magic header 0x1f, 0x8b (\037 \213)

1 byte compression method (0..7 reserved, 8 = deflate)1 byte flags

bit 0 set: file probably ASCII text

bit 1 set: continuation of multi-part gzip file

bit 2 set: extra field present

bit 3 set: original file name present

bit 4 set: file comment present

bit 5 set: file is encrypted

bit 6,7: reserved

4 bytes file modification time in Unix format

1 byte extra flags (depend on compression method)

1 byte operating system on which compression took place

2 bytes optional part number (second part=1)

2 bytes optional extra field length

? bytes optional extra field

? bytes optional original file name, zero terminated

? bytes optional file comment, 0 terminated12 bytes optional encryption header

? bytes compressed data

4 bytes crc32

4 bytes uncompressed input size modulo 2^32

The format was designed to allow single pass compression without any backwards seek, and without a priori knowledge of the uncompressed input size or the available size on the output media. If input does not come from a regular disk file, the file modification time is set to the time at which compression started.

The time stamp is useful mainly when one gzip file is transferred over a network. In this case it would not help to keep ownership attributes. In the local case, the ownership attributes are preserved by gzip when compressing/decompressing the file. A time stamp of zero is ignored.

Bit 0 in the flags is only an optional indication, which can be set by a small look ahead in the input data. In case of doubt, the flag is cleared indicating binary data. For systems which have different file formats for ASCII text and binary data, the decompressor can use the flag to choose the appropriate format.

The extra field, if present, must consist of one or more subfields, each with the following format:

subfield id : 2 bytes

subfield size : 2 bytes(little-endian format)

subfield data

The subfield id can consist of two letters with some mnemonic value. Please send any such id to jloup@chorus.fr. Ids with a zero second byte are reserved for future use. The following ids are defined:

Ap (0x41, 0x70) : Apollo file type information

The subfield size is the size of the subfield data and does not include the id and the size itself. The field 'extra field length' is the total size of the extra field, including subfield ids and sizes.

It must be possible to detect the end of the compressed data with any compression format, regardless of the actual size of the compressed data. If the compressed data cannot fit in one file (in particular for diskettes), each part starts with a header as described above, but

only the last part has the crc32 and uncompressed size. A decompressor may prompt for additional data for multipart compressed files. It is desirable but not mandatory that multiple parts be extractable independently so that partial data can be recovered if one of the parts is damaged. This is possible only if no compression state is kept from one part to the other. The compression-type dependent flag scan indicates this.

If the file being compressed is on a file system with case insensitive names, the original name field must be forced to lower case. There is no original file name if the data was compressed from standard input.

Compression is always performed, even if the compressed file is slightly larger than the original. The worst case expansion is a few bytes for the gzip file header, plus 5 bytes every 32K block, or an expansion ratio of 0.015% for large files. Note that the actual number of used disk blocks almost never increases.

The encryption is that of zip 1.9. For the encryption check, the last byte of the decoded encryption header must be zero. The timestamp of an encrypted file might be set to zero to avoid giving a clue about the construction of the random header.

4

METHODOLOGY

(DATA COMPRESSION/DECOMPRESSION)

Data Compression involves two main tasks, namely, Data Compression and Data Decompression. The process of data compression is only feasible if one can compress the data and also retrieve the original file by decompressing/uncompressing the compressed file.

Thus, the complete methodology consists of these two tasks – Compression and Decompression.

Since, C-Scan data files are very large, these files need to be read, analyzed, compressed and decompressed in 'segments'.

5.1 COMPRESSION ALGORITHM

C-Scan: ASCII file [figure 2] Extension: *.csn
Statistical Method employed: Huffman Coding

- ✓ Huffman Encoding: Assigns variable size codes to the characters. Frequent characters get binary codes of less number of bits and rare codes get binary codes of large number of bits.
- ✓ How frequent a character is, is decided by probability of occurrence of each character. Thus, it should be clearly defined that we have *n* number of characters that could possibly occur.
- ✓ In C-Scan, ASCII (American Standard Code for Information Interchange), the number of characters or their corresponding ASCII code is fixed i.e. 0 to 255.
- ✓ Probability of occurrence is calculated from the sample data i.e. for a 'large' C-Scan file the probability is calculated once and based on that 'large' C-Scan we define a standard set of Huffman Codes for all the compressions.
- ✓ Thus, a drawback is that the algorithm is specific to C-Scan file only and no other ASCII data type. It would work on other data types too but would not render best compression results, because the probability distribution is defined strictly for the C-Scan type data.

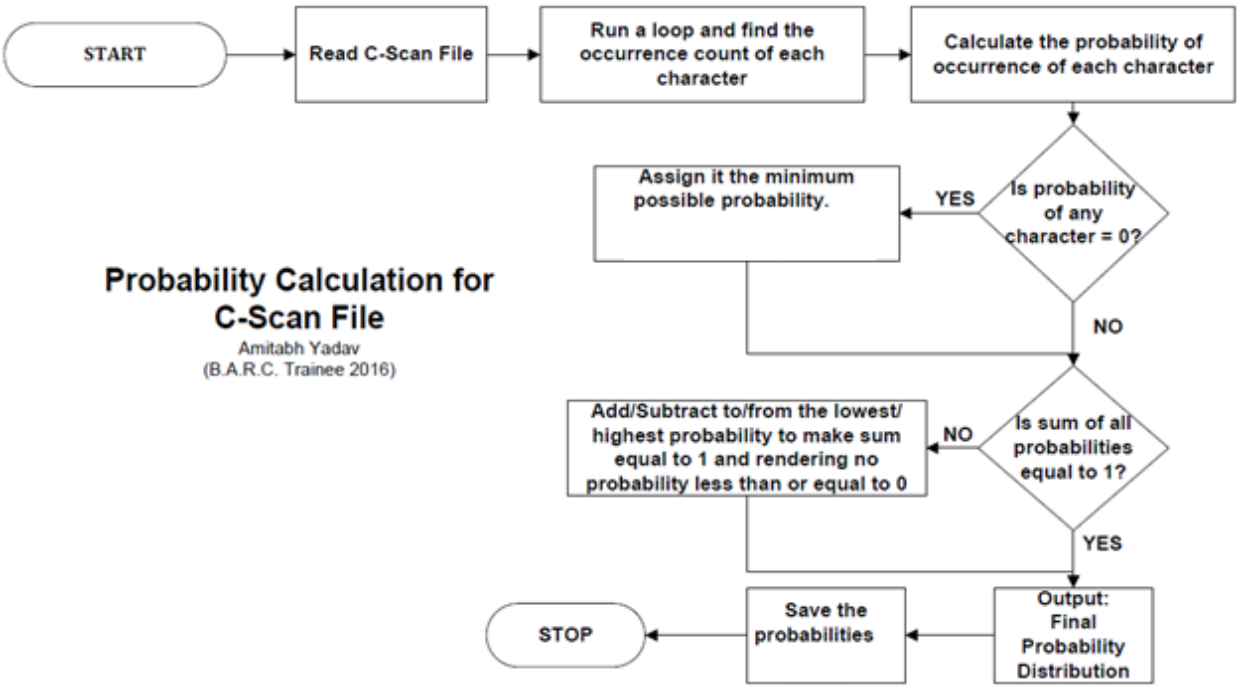

Figure 5 Probability Calculation Algorithm for C-Scan data (Flowchart)

- ✓ We have the probability distribution, using these we will generate variable size codes to the individual characters sequentially. The set of these variable size codes is called a 'Dictionary'. To generate a dictionary we need an array of possible characters (or symbols) and another array with corresponding value of probability.

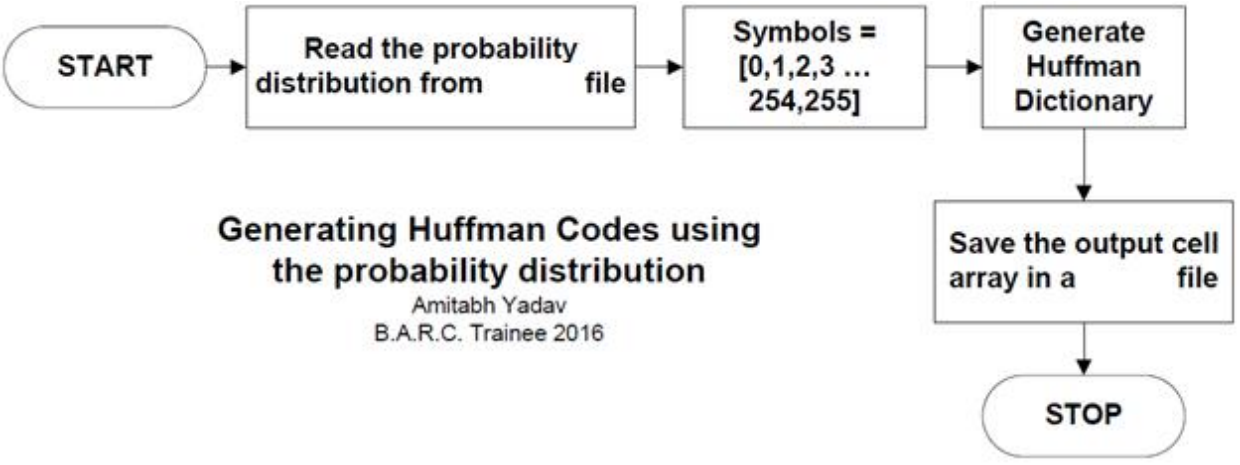

Figure 6 Generating Huffman Codes using Probability Distribution (Flowchart)

- ✓ NOTE: Even with the same set of values for the symbols and probability, the Huffman Dictionary generating function may generate a different set of codes for every different execution.
  So, probability calculation and dictionary generation should be done only once and for all. ***Any single change in the values of the dictionary, and the***

compressed data can <u>never</u> be recovered. Besides, ***once the dictionary has been generated and codes assigned to each character, we can use these codes to implement data compression in any programming language such as C/C++ which will render faster execution time.*** There are 256 symbols (0-255) and corresponding 256 variable-size binary codes.

✓ Compression:
INPUT FILE: *.csn OUTPUT FILE: *.bin

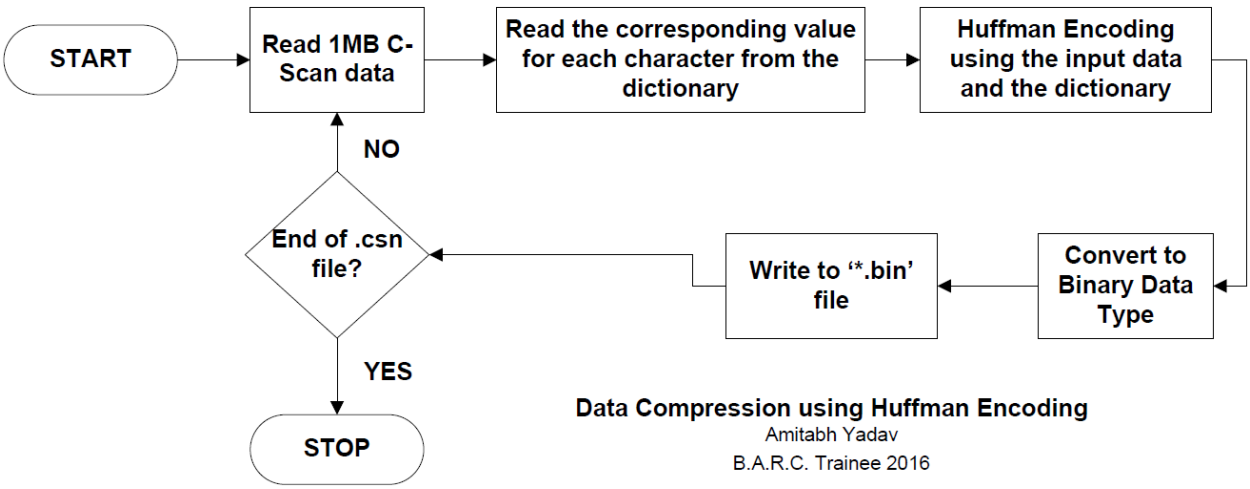

Figure 7 Data Compression Using Huffman Encoding (Flowchart)

5.2 DECOMPRESSION ALGORITHM

- From the dictionary created during the compression, read all the variable size codes assigned to the symbols and save it in a file sequentially.
- In this file, maximum number of bits is assigned to the rarest character. In our project, we had maximum number of bits in a code = 13.
- Create an array, say a[ ], of size 2^{13} and initialize each element to -1 (because -1 is not a possible ASCII value).
- One by one, read all the codes in the file created in step 1. Convert them into decimal (base10) and assign the possible character of that binary code to the position in array a[ ] which is obtained by converting to decimal (base10) value.
- Decompression:
  INPUT FILE: '*.bin' OUTPUT FILE: '*.csn

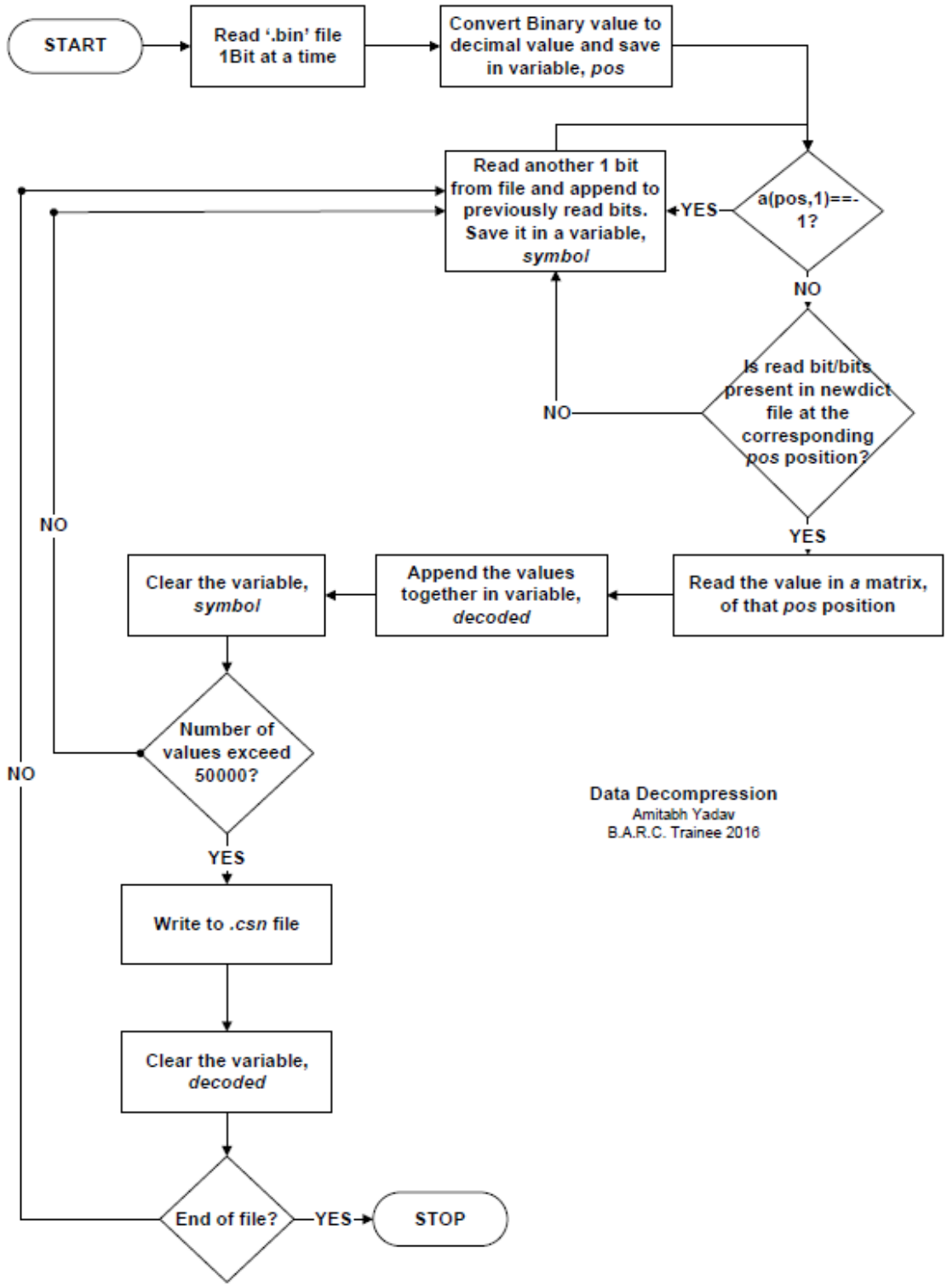

Figure 8 Data Decompression (Flowchart)

- OPTIMISATION: To save time, in this decompressing algorithm, read 1MB data from .bin file at a time. But, since variable size codes has been assigned, one will require to 'take care' of the end bits in that segment and if no assigned value detected in table, append the last few bits to the next segment and continue decoding.

5

RESULTS

Initial study was performed for lossy compression algorithms such as **Discrete Cosine Transform (DCT)** and **Discrete Wavelet Transform (DWT)** [1]. The JPEG compression uses DCT to compress RAW images and is often implemented in Digital Cameras to acquire images. DWT is less commonly used and is under R&D by the JPEG group for image compression application.

However, due to complicated instructions and difficultly in implementation, the preferred method for compression was chosen to be a lossless statistical compression such as Huffman Coding, Arithmetic Coding etc.

Arithmetic Coding requires the frequency count of symbol thus, is unsuitable for practical application as the program has to go through the data twice, once to calculate the frequency of each symbol and the second time to code it.

So, preferred method for Coding was chosen to be **Huffman Coding**.

I. A-Scan Data

- A GUI was developed test Huffman Coding compression performance for different types of probability distributions, namely, Equal Probability (where all symbols were assigned nearly equal probabilities except those that clearly seem to occur frequently); Binomial Probability (looking at the probability distribution graph of the A-Scan Signal it seems to resemble the Binomial Probability distribution, so this distribution was also tested) ;and, Calculated Probability (where probability was initially calculated for that particular data set and Huffman codes generated accordingly).

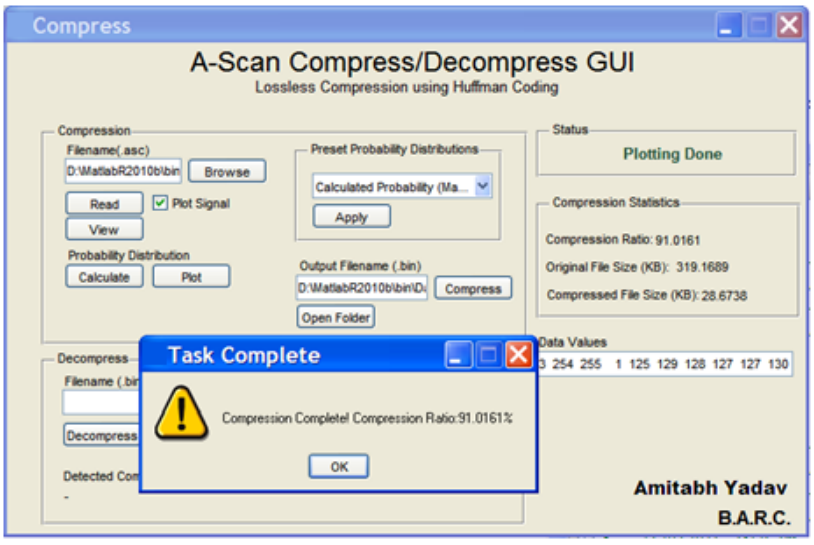

Figure 9 GUI for A-Scan Data Compression

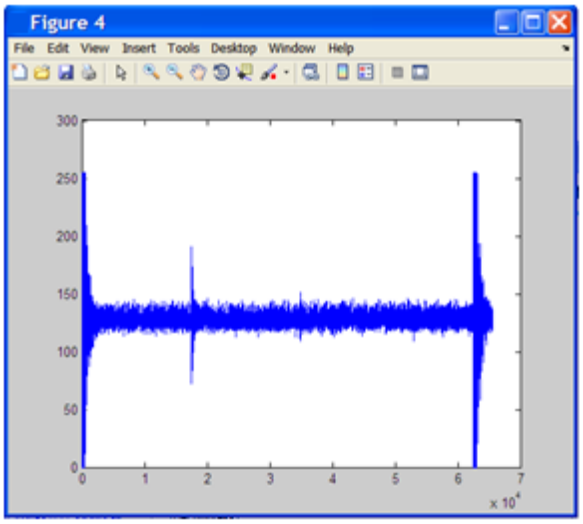

Figure 10 A noisy (short) A-Scan Signal

- **Equal Probabilities:**

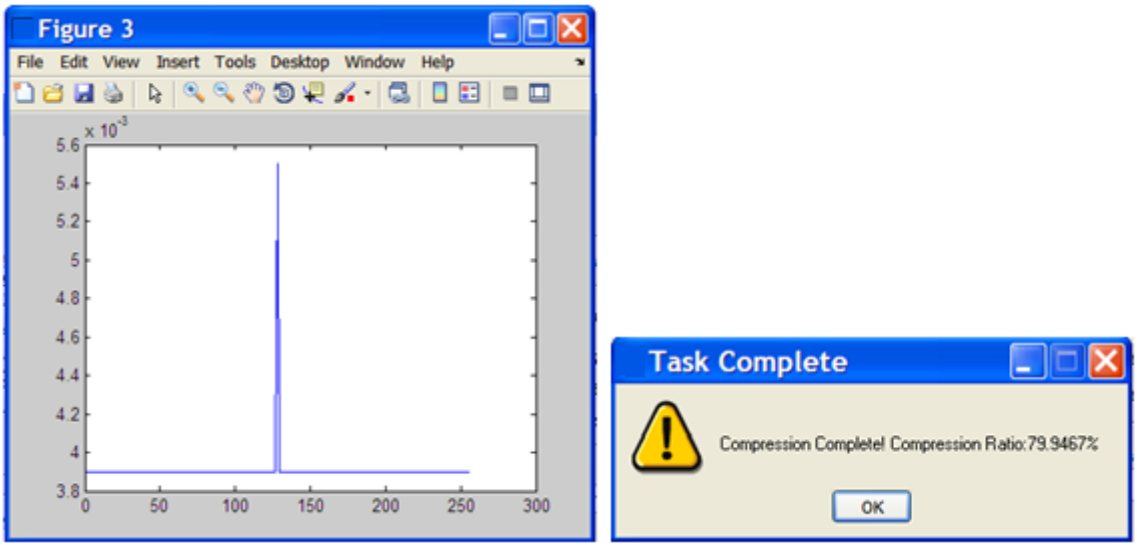

Figure 11 (a)Equal Probabilities assigned to all values in A-Scan Signal. (b) Compression Achieved: 79.94%

- **Binomial Probabilities:**

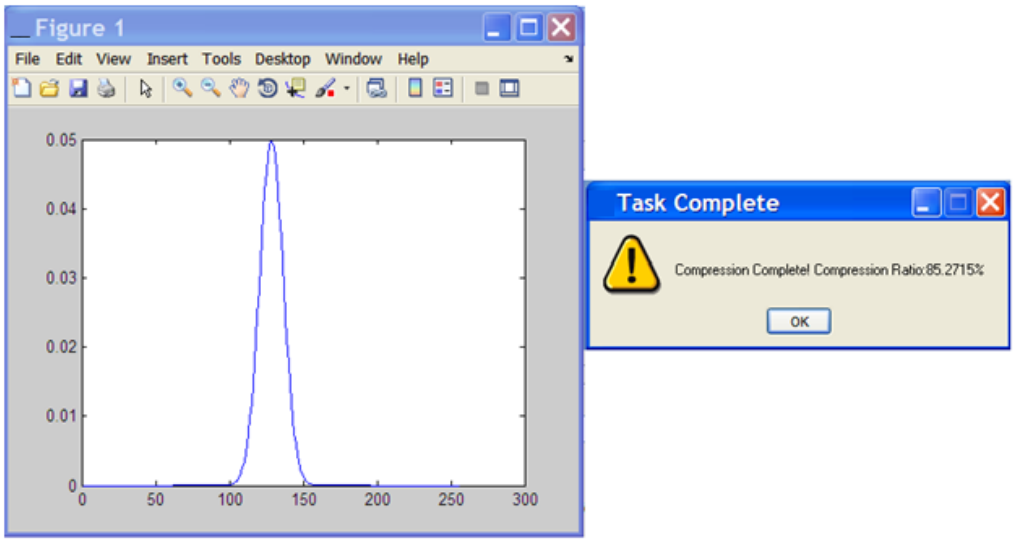

Figure 12 (a)Binomial Probability Distribution corresponding to all values in A-Scan Signal. (b) Compression Achieved: 85.27%

- **Calculated Probabilities:** (in certain cases, the probabilities needed to be adjusted to make the total sum of probabilities equal to 1)

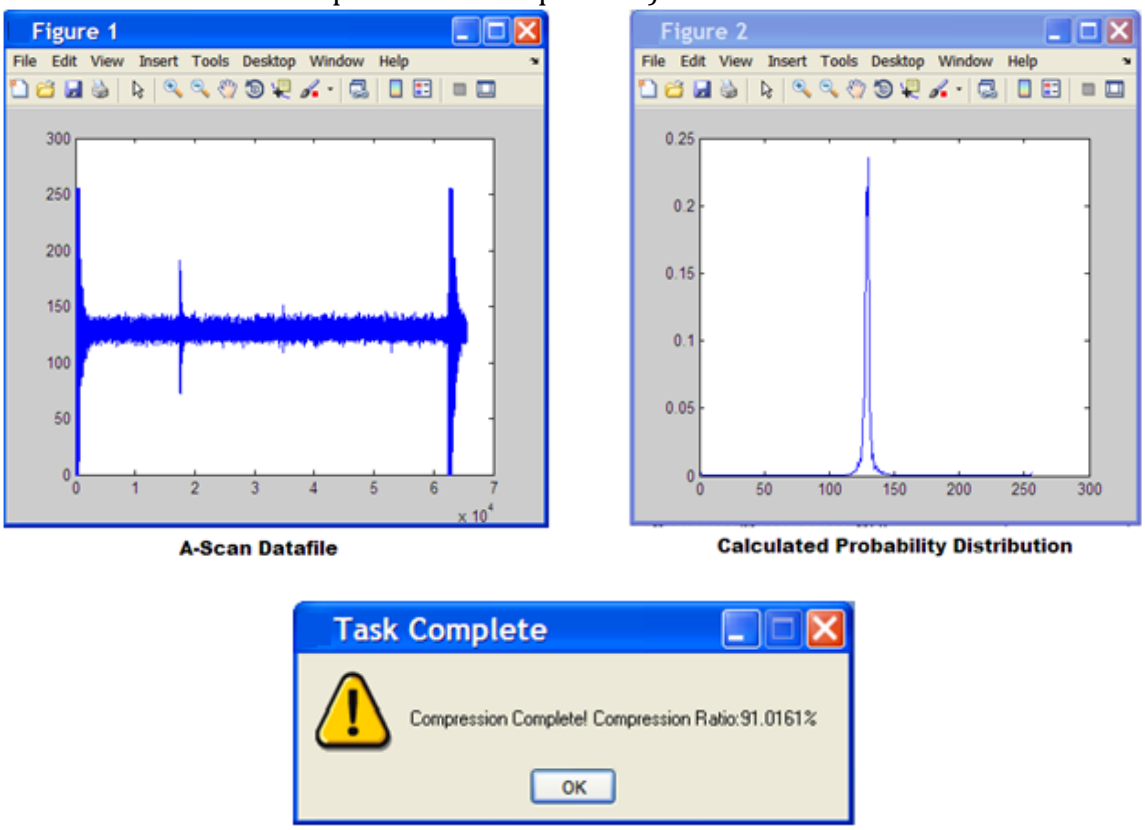

Figure 13 (a) A-Scan Signal Datafile (b)Calculated Probabilities assigned to all values in A-Scan Signal. (c) Compression Achieved: 91.01%

- A-Scan Signals were tested for various probability distributions. The best results were obtained from the distribution which was calculated from the calculated probability, which was about 91%.
- Data Decompression for A-Scan was successfully achieved and both the uncompressed file and the original file was found to be identical.

II. C-Scan Data

- C-Scan Data was analyzed for compression through software program development.
- C-Scan file size: 16MB
- Compressed File Size: 4.06MB
- This algorithm was tested for various probability distributions including equal probabilities (dictionary 1), rough approximation (dictionary 2) and calculated probabilities and approximation (dictionary 3).#
- For dictionary 1, compression was ~1%. For Dictionary2, the compression achieved is ~11.2% and finally, for Dictionary3, the compression achieved is ~75.37%.
- Case Study: For a C-Scan file of size 16928KB (16.5MB), the compressed file size achieved was 4168KB. **The time elapsed was ~456sec (7.5mins)**.
- **Data Decompression was achieved correctly but with drastic <u>performance reduction</u>.** Execution took long to actually decompress the data! Attempts to reduce time taken were performed to improve the performance.

III. Wavelet Analysis

- A very basic wavelet analysis was performed on the A-Scan signals to understand the concept of wavelets and how to construct the desired waveform (in this case, ultrasonic echo signals) using the basic wavelets, such as, Haar, Daubechies etc.

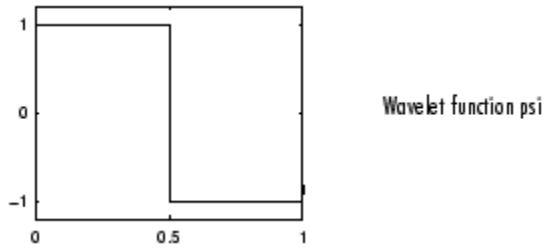

Figure 14 Haar Wavelet

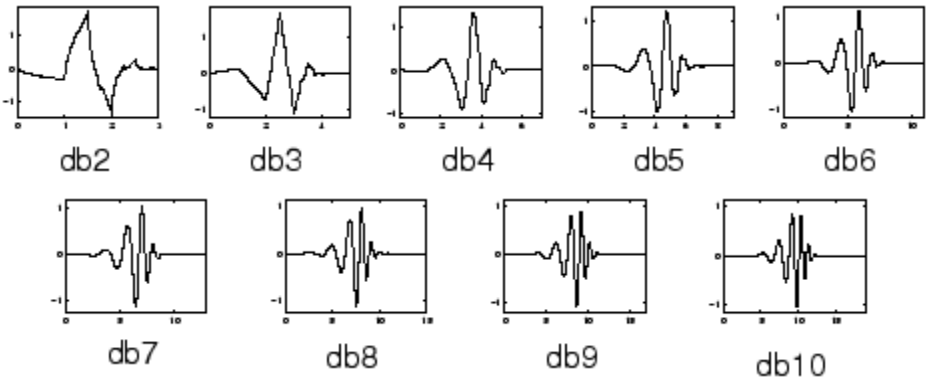

Figure 15 Daubechies (db2-db10) Wavelets

- According to the study [1], the lossy compression algorithms can provide very high compression (up to 98%) while rendering negligible loss to data. The DWT in [1] works on parameter estimation and hence 'constructs' the waveform using the basic wavelet. The basic wavelet chosen was a modified version of Morlet Wavelet.

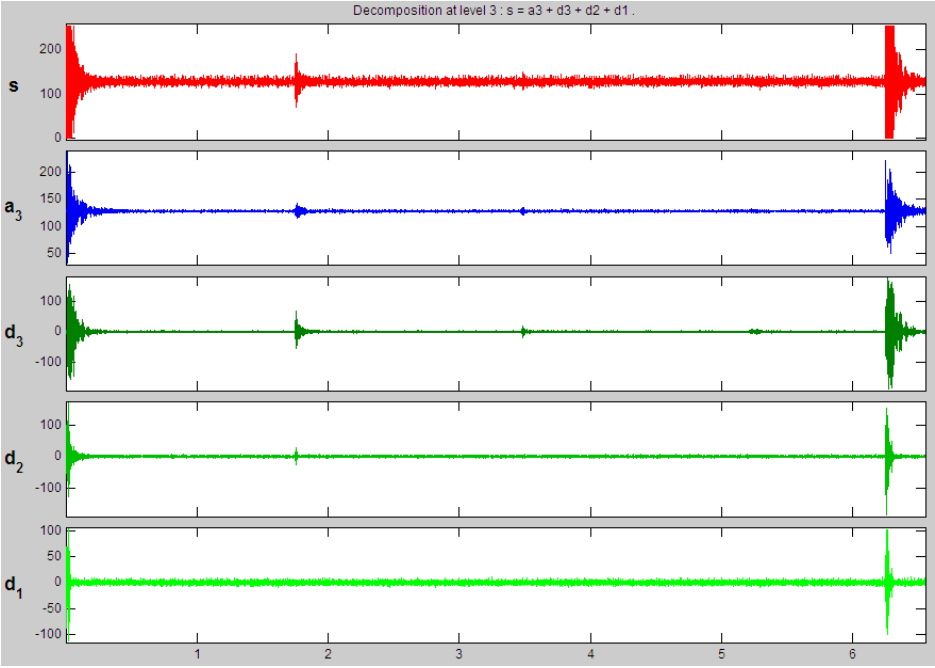

Figure 16 A-Scan Signal Decomposition using DB10 at level 3 decomposition and coefficient calculation

- Wavelet Transform also performs significant noise reduction.

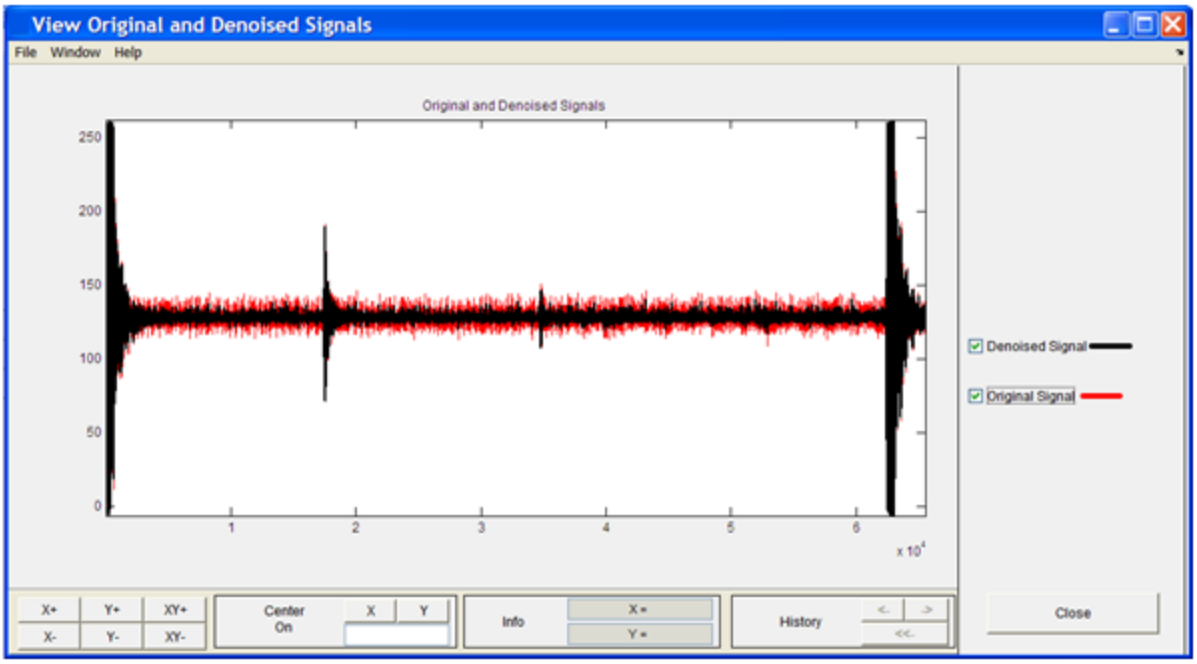

Figure 17 Denoising of the A-Scan Signal using Wavelet Transform

- Wavelet Analysis is a complex method and requires a significantly deep knowledge of Digital Signal Processing and Wavelet Transform to implement in data compression application.

6

CONCLUSION & FURTHER DEVELOPMENTS

Conclusions:

- ✓ Data Compression can be achieved by eliminating the redundancies in the data.
- ✓ Signal Analysis of A-Scan/B-Scan/C-Scan data shows that there are large amount of redundancies present, as, mostly the ultrasonic signals are constant at the center value.
- ✓ Ultrasonic Signal Data are in the form of ASCII value files.
- ✓ Huffman Coding is the best coding algorithm for assigning variable size codes to the data/ASCII values (0to255)

Further Developments:

- ✓ The algorithm can be optimized for faster execution.
- ✓ Since time taken is too much, it is not feasible to implement directly the Huffman Coding Algorithm to the C-Scan Signal Compression. So, a variant of the same can be tested for further development viz. Lempel-Ziv Algorithm (refer to Case Study, Section 7.1)
- ✓ A detailed study on Wavelet Transform can prove to give promising results in the data compression, by reducing the number of coefficients. The preferred wavelets for the same for A-Scan signals are: DB-20, Gaussian and Morlet Wavelet.

7

REFERENCES

1. “Ultrasonic Data Compression via Parameter Estimation”, Guilherme Cardoso and Jafar Saniie*;* IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control, Vol. 52, No. 2, February 2005 (313-325)
2. “Compression of Ultrasonic RF Data” A. Pesavento, V. Burow, H. Ermert; IEEE 1997 Ultrasonics Symposium Proceedings (1996) pp. 1471-1474
3. “Multidimensional Representation of Ultrasonic Data Processed by Reconfigurable Ultrasonic System-on-Chip Using OpenCL High-Level Synthesis”, Spenser Gilliland, Clementine Boulet, Thomas Gonnot and Jafar Saniie; 2014 IEEE International Ultrasonics Symposium Proceedings pp.1936-1939
4. “3D Ultrasonic Signal Compression Algorithms for High Signal Fidelity”, Pramod Govindan, Thomas Gonnot, Spenser Gilliland and Jafar Saniie, IEEE 2013 pp.1263-1266
5. “Model-based estimation of ultrasonic echoes”, Ramazan Demirli, Student Member, IEEE, and Jafar Saniie, Senior Member, IEEE; IEEE Transactions on Ultrasonics Ferroelectrics and Frequency Control · June 2001
6. “Ultrasonic Signal Compression Using Wavelet Packet Decomposition and Adaptive Thresholding”, Erdal Oruklu, Namitha Jayakumar and Jafar Saniie; 2008 IEEE International Ultrasonics Symposium Proceedings pp.171-175
7. [LZ77] Ziv J., Lempel A., "A Universal Algorithm for Sequential DataCompression", IEEE Transactions on Information Theory", Vol. 23, No. 3,pp. 337-343
8. “A guide to data compression methods” by David Salomon. Springer.
9. “Fractal and Wavelet Image compression Techniques” by Stephen Welstead.