



1996-03

# Analysis and performance comparison of adaptive differential pulse code modulation data compression systems

Cooperwood, Michael Vonshay.

Monterey, California. Naval Postgraduate School

---

<http://hdl.handle.net/10945/32143>



Calhoun is a project of the Dudley Knox Library at NPS, furthering the precepts and goals of open government and government transparency. All information contained herein has been approved for release by the NPS Public Affairs Officer.

**Dudley Knox Library / Naval Postgraduate School**  
**411 Dyer Road / 1 University Circle**  
**Monterey, California USA 93943**

<http://www.nps.edu/library>

# NAVAL POSTGRADUATE SCHOOL MONTEREY, CALIFORNIA



## THESIS

**ANALYSIS AND PERFORMANCE  
COMPARISON OF ADAPTIVE DIFFERENTIAL  
PULSE CODE MODULATION DATA  
COMPRESSION SYSTEMS**

by

Michael Vonshay Cooperwood, Sr.

March, 1996

Thesis Advisor:

Monique P. Fargues

Co-Advisor:

Ralph Hippenstiel

Approved for public release; distribution is unlimited.

19960503 081

INFO QUALITY ASSURED 1

# REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE MARCH 1996	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE ANALYSIS AND PERFORMANCE COMPARISON OF ADAPTIVE DIFFERENTIAL PULSE CODE MODULATION DATA COMPRESSION SYSTEMS		5. FUNDING NUMBERS	
6. AUTHOR(S) Michael Vonshay Cooperwood, Sr.			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000		8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.		12b. DISTRIBUTION CODE	
13. ABSTRACT (maximum 200 words) Advances in audio data compression are largely driven by the need to conserve transmission rate or bandwidth, while maintaining the ability to accurately reconstruct the signal at the receiver. This report examines data compression methods with an emphasis on techniques for the compression of audio data. An overview of data compression schemes is presented to provide the background for a performance comparison between selected versions of data compression systems featuring adaptive differential pulse code modulation (ADPCM) schemes. Two different types of data compression systems are investigated; IIR and FIR impulse implementations. A modification to the basic ADPCM system using a modular function is implemented. The modular operation results in a smaller size codebook and prevents data expansion when the source is not matched to the code. This modification is utilized for both types of ADPCM coders compared. To complete the compression system Huffman coding, is employed to encode and decode the compressed data to and from binary form.			
14. SUBJECT TERMS ADAPTIVE DIFFERENTIAL PULSE CODE MODULATION DATA COMPRESSION SYSTEMS PERFORMANCE COMPARISON.		15. NUMBER OF PAGES 94	16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)  
Prescribed by ANSI Std. Z39-18 298-102



Approved for public release; distribution is unlimited.

**ANALYSIS AND PERFORMANCE COMPARISON OF  
ADAPTIVE DIFFERENTIAL PULSE CODE MODULATION  
DATA COMPRESSION SYSTEMS**

Michael V. Cooperwood, Sr  
Lieutenant, United States Navy  
B.S., University of South Carolina, 1986

Submitted in partial fulfillment  
of the requirements for the degree of


**MASTER OF SCIENCE IN ELECTRICAL ENGINEERING**

from the

**NAVAL POSTGRADUATE SCHOOL**

March 1996


Author:

  
Michael V. Cooperwood, Sr.

Approved by:

  
Monique P. Fargues, Thesis Advisor

  
Ralph Hippenstiel, Co-Advisor

  
Herschel H. Loomis, Jr., Chairman  
Department of Electrical and Computer Engineering



## ABSTRACT

Advances in audio data compression are largely driven by the need to conserve transmission rate or bandwidth, while maintaining the ability to accurately reconstruct the signal at the receiver. This thesis examines data compression methods with an emphasis on techniques for the compression of audio data. An overview of data compression schemes is presented to provide the background for a performance comparison between selected versions of data compression systems featuring adaptive differential pulse code modulation (ADPCM) schemes. Two different types data compression systems are investigated; IIR and FIR impulse systems. A modification to the basic ADPCM system using a modular function is implemented. The modular operation results in a smaller size codebook and prevents data expansion when the source is not matched to the code. This modification is utilized for both types of ADPCM coders compared. To complete the compression system, Huffman coding is employed to encode and decode the compressed data to and from binary form.





## TABLE OF CONTENTS

I.	INTRODUCTION.....	1
	A. LITERATURE REVIEW.....	1
	B. DATA COMPRESSION BACKGROUND.....	2
	C. THESIS OVERVIEW.....	3
II.	LOSSLESS COMPRESSION TECHNIQUES.....	5
	A. THEORETICAL BACKGROUND.....	5
	B. SHANNON-FANO CODING.....	6
	C. HUFFMAN CODING.....	9
	D. ARITHMETIC CODING.....	16
III.	LOSSY DATA COMPRESSION TECHNIQUES.....	21
	A. LOSSY COMPRESSION THEORETICAL BACKGROUND.....	21
	B. SCALAR AND VECTOR QUANTIZATION.....	22
	C. MULTIPATH SEARCH CODING.....	24
	1. TREE/TRELLIS CODING.....	24
	2. CODEBOOK CODING.....	26
	D. PREDICTIVE CODING.....	28
	1. DELTA MODULATION.....	28
	2. DIFFERENTIAL PULSE CODE MODULATION.....	28
IV.	LOSSLESS ADAPTIVE DIFFERENTIAL PULSE CODE MODULATION.....	31
	A. INTRODUCTION.....	31
	B. FINITE IMPULSE RESPONSE FILTER.....	32
	C. INFINITE IMPULSE RESPONSE FILTER.....	36

D. MODULAR ARITHMETIC FUNCTION.....	39
V. PERFORMANCE COMPARISON.....	45
A. OVERVIEW.....	45
B. PRESENTATION OF DATA.....	46
VI. CONCLUSIONS.....	61
APPENDIX (MATLAB CODE).....	65
A. FIR ADPCM Implementation.....	66
B. IIR ADPCM Implementation.....	70
C. HUFFMAN Encoding/Decoding Scheme.....	77
LIST OF REFERENCES.....	81
INITIAL DISTRIBUTION LIST.....	83

## ACKNOWLEDGEMENT

The author would like to express his sincere gratitude and thanks for the extraordinary support of his wife, Linda, and sons Ashanta and Hampton, without whose understanding and forbearance this thesis could not have been completed.

The author wants to thank Prof. Fargues and Prof. Hippenstiel for their guidance and patience during the work in performing this investigation.

## I. INTRODUCTION

### A. LITERATURE REVIEW

The efficient digital representation of data and speech signals offers the possibility to increase the data rate transmitted over existing digital transmission networks by providing bit rate reductions as high as 16:1 in comparison with the use of logarithmic pulse code modulation schemes. Data compression is the application of methods to process information to obtain a more compact representation without suffering unacceptable loss of fidelity or accuracy (Davidson and Gray, 1976). Compression techniques are based on either lossy or lossless properties. Lossless methods generate an exact duplicate of the original signal upon decompression, whereas, lossy methods trade complete accuracy for increased compression. Some of the lossless algorithms developed for data compression include Shannon-Fano (Lynch, 1985), Huffman (Knuth, 1985), (Lu and Gough, 1993), and Arithmetic (Langdon, 1984) schemes. Lossy methods include various types of quantization coding, transform coding and predictive coding (Lynch, 1985), (Cappellini, 1985), (Sibul, 1987). Audio (speech, music, etc.) compression research has traditionally been separate from other areas. Since sound is often an integral part of other data types, this area of research is becoming increasingly more important with currently evolving applications. This thesis examines data compression techniques with a specific emphasis on compressing audio signals (Rabiner and Schafer, 1978). The basic theory of data compression, including a discussion of some of the most common techniques, is explored. A review of basic ADPCM coders, along with an improvement using modular arithmetic is presented (Sibul, 1987), (Einarsson, 1991). These latter techniques are the structures implemented for a comparison of the FIR and IIR versions of the modified ADPCM algorithms.

## B. DATA COMPRESSION BACKGROUND

Data Compression is the process used to reduce the physical space or bandwidth used to hold or transmit a particular set of data. The electromagnetic spectrum, time intervals and physical volumes are all compressible mediums. The following equation shows the interrelationship of all three.

$$Volume = f(time \times bandwidth) \quad (1.1)$$

In effect, reducing the volume which a set of data occupies, results in a reduction in transmission time or bandwidth. C. E. Shannon originated this concept in 1948 which initiated the branch called information theory. Shannon's work showed that the extent to which a message can be compressed and then accurately restored is limited by its entropy. Entropy is a measure of the message's information content: the more probable the message, the lower its entropy. Entropy can also be represented as a measure of surprise; the more unexpected are the contents of a message, the higher its entropy and vice-versa, which results in more bits being required to encode it. Shannon used source entropy and channel capacity as the basis for two basic theorems which set precise bounds to the accurate representation and errorless transmission of data (Lynch, 1985). With unlimited resources (time, computing power, etc.), then the code wordlength for optimal source codes is approximately equal to (but not less than) the source entropy. Hence, source coding, also known as entropy coding, is just another term for data compression.

Data compression methods take one of two approaches: lossless or lossy. Lossless data compression algorithms perfectly reconstruct the compressed data without error. Source message redundancy reduction is the method these algorithms employ to achieve

compression. Repetitive data in a message set or signal is eliminated by sending only the changes and number of repetitions. Conversely, entropy reduction is the principle used by lossy data compression techniques. The entropy reduction process results in an information loss. Using a threshold to monitor sample values is one example of a lossy compression process. In this case, compression is achieved by only transmitting the time at which a sample value exceeds the preassigned threshold. Data compression systems often combine both lossy and lossless techniques to achieve maximum compression.

### **C. THESIS OVERVIEW**

The current chapter introduces the basic mechanics of data compression including a review of the some of the relevant literature.

Chapter II describes the specifics of several lossless data compression methods. Huffman coding, perhaps the best-known method, used for encoding and decoding of the compressed signals is featured. Related compression techniques, Shannon-Fano and Arithmetic coding are also discussed.

Chapter III reviews lossy compression schemes. Some of the different types of quantization schemes are presented. These include vector quantization and the featured predictive coders.

Chapter IV introduces adaptive differential pulse code modulation. An improvement to the basic algorithm designed to decrease the size of the codebook used for channel coding is also explained.

In Chapter V, a comparative analysis of the FIR and IIR implementations of the ADPCM is performed. Specific comparison points include compression ratio, power reduction and speed of operation of each design.

The general conclusion reached from the comparative analysis of Chapter V are presented in Chapter VI. Topics for further study are also discussed.

The Appendix lists the Matlab code used to evaluate and simulate the data compression systems.

## II. LOSSLESS COMPRESSION TECHNIQUES

### A. THEORETICAL BACKGROUND

Lossless data compression algorithms preserve all the information in the data so that it can be reconstructed without error. Adhering to that constraint, their compression ratios are significantly less than their lossy counterparts; averaging 2:1 to 8:1, depending upon the redundancy of the information source and the efficiency of the algorithm. Despite that fact, some applications such as storing or transmitting financial documents, computer programs or numerical information, where a single bad bit could be catastrophic, demand lossless techniques. A number of different lossless compression methods with many variations exist. These techniques fall into the following general categories: optimum source coding, nonredundant sample coding and binary source coding. Again, note that coding techniques result in data compression, thus their discussion is germane to a data compression exploration. Nonredundant sample coding (NSC) makes use of threshold values and sends only time information. The dominant type of NSC is run-length coding and others include predictors and interpolators (Lynch, 1985). Since the timing information sent by this method is asynchronous, buffering is required. The compression systems studied in this thesis are synchronous ADPCM systems without buffering, and no further discussion of NSC occurs. The remainder of this chapter discusses optimum source and binary coding methods.

Optimum source encoding starts with statistically independent samples and codes them in such a way as to make the average word length equal to the sample entropy. This method is also called entropy coding (since the code approaches the entropy of the source). Entropy relates to randomness. If the contents of a message are unexpected,



then the entropy is high. If the contents of a message are as expected, then entropy is low. If the entropy of a data set is reduced, the lower entropy data set can then be encoded with fewer bits resulting in data compression. Two main methods occupy this category: Shannon-Fano and Huffman coding.

## B. SHANNON-FANO CODING

The Shannon-Fano coding procedure produces binary codes that are instantaneously decodable. An explanation of instantaneous decodability is provided later. One application of this technique is as a stage of the well-known PKZIP's (Apiki, 1991) "imploding" algorithm. Shannon-Fano code reaches an efficiency of 100% only when the source message probabilities are negative powers of two. The following coding procedure and example detail and illustrate the process (Lynch, 1985): 1) Arrange the source message probabilities in descending order, 2) Divide the message set into two subsets of equal, or almost equal, total probability and assign a zero as the first code digit in one subset, and a one as the first code digit in the second subset, 3) Continue this process until each subset contains only one message. The accompanying example shows how the efficiency is computed.

### EXAMPLE

The following set of messages with probabilities,  $P_i$ , is given:

$m(i)$	1	2	3	4	5	6	7
$P_i$	0.4	0.1	0.1	0.1	0.1	0.1	0.1

The average codeword length,  $L_{avg}$ , is computed by the formula:

$$L_{avg} = \sum_{i=1}^M l(m(i)) P_i \quad (2.1)$$

where  $M$  is the number of messages in the set and  $l(m(i))$  represents the number of bits in the code for symbol  $m(i)$ . Applying the procedure outlined earlier the following Shannon-Fano code would be generated as follows:

Split the seven messages into two groups with their probabilities being as equal as possible. Thus, group one would be message one and message two with a total  $P_i = 0.5$ , and group two would be the four remaining messages,  $m3-m7$ , with the same total  $P_i$ . The next step would be to arbitrarily assign a '1' to group one and '0' to group two. The process would be repeated; group one would be split into two groups and a second '1' would be arbitrarily assigned to message one,  $m1$ , and a '0' to  $m2$ . This process is arbitrary because it could have been done in the reverse, assigning a '0' to  $m1$  and a '1' to  $m2$ . Likewise, group two would be split into two subgroups,  $m3$  and  $m4$  in one group and  $m5-m7$  in another group. Again, each member of the first subgroup would be assigned a one and each member of the second subgroup would append a zero onto its current code. As per the procedure, the process would continue until each message had been assigned a unique, instantly decodeable codeword. The entire process is shown in Table 2.1. Using Equation 2.1,  $L_{avg}$  is computed to be 2.7 bits:

$$[2(0.4) + 2(0.1) + 3(0.1) + 3(0.1) + 3(0.1) + 4(0.1) + 4(0.1) = 2.7 \text{ bits}].$$

By comparison, the entropy is given by:

$$H = -\sum_{i=1}^M P_i \log_2 P_i \quad (2.2)$$

Original Group		New Grp		New Grp		Org. msg		
$m(i)$	Code		Code		Code	$m(i)$	$P_i$	Code
$m12$	1	$m1$	11	$m1$	11	$m1$	0.4	11
		$m2$	10	$m2$	10	$m2$	0.1	10
$m34567$	0	$m34$	01	$m3$	011	$m3$	0.1	011
				$m4$	010	$m4$	0.1	010
		$m567$	00	$m5$	001	$m5$	0.1	001
				$m67$	000	$m6$	0.1	0000
						$m7$	0.1	0001

TABLE 2.1 Shannon-Fano Codeword Construction

This value is computed to be 2.522 bits resulting in an efficiency of 93.4%, where efficiency,  $\eta$ , is given by:

$$\eta = \frac{H}{L_{avg}} \quad (2.3)$$

Shannon-Fano coding has a close relative, called Huffman coding, which is well known to have a significantly greater efficiency. Increased efficiency is one reason Huffman is a preferred technique.

## C. HUFFMAN CODING

### 1. Static Huffman Coding

Huffman coding is probably the best-known method of data compression and has many practical applications. These include the last stage of JPEG compression and the MNP-5 modem data compression standard (Apiki, 1991). The basic premise of Huffman coding is the creation of a binary tree with internal nodes, called branches, that represent the path to external nodes, known as leaves. There is one leaf on the tree for each symbol in the data set. These leaves are combined by connecting them to branches. The tree begins at a branch called the root, which has no number assigned to it and has a probability of one. The binary numbers encountered along the path to the leaves comprise the variable-length codes for each symbol of a given sequence, with each code being represented by an integral number of bits. Symbols with higher probabilities are given shorter bit codes while symbols with lower probabilities are assigned longer bit codes and thus, longer branches on the tree. The Huffman tree is constructed, after determining the frequency of occurrence or probability for each symbol in a source, by repeatedly combining the two least probable symbols at each stage. This process continues until the original source is reduced to only two symbols. These two symbols are respectively assigned the bit values of '0' and '1'. The codes for the previous reduced stage are then determined by appending a '0' or a '1' to the right of the code corresponding to the two least probable symbols, and so on. Once each symbol in the original source is assigned a binary code, the Huffman coding is complete. Table 2.2 shows an example source reduction and Table 2.3 performs the resulting codeword construction for generating the Huffman code (Apiki, 1991), (Langdon, 1984).

Table 2.2 displays a source reduction process for a source with five symbols;  $m_1$  through  $m_5$ . Symbol probability is determined by dividing the total number of occurrences, No. occ, of each symbol by the total number of symbols. The two lowest probability symbols are combined to make a new symbol with a probability equal to that of the combined symbols. The next two lowest probabilities are then combined. When symbols of equal probability are encountered, they can be combined in any order; with an optimal code still resulting. The Huffman code is optimal in that it produces a code with the minimal average word length. Thus, in Table 2.2,  $m_5$  could have been combined with  $m_3$ , or  $m_2$ , not just  $m_4$ . No ambiguity results from the arbitrary combinations since each symbol is given a unique binary code. The construction of this code is depicted in Table 2.3. The combination process continues until only two symbols remain. These symbols are assigned the codes of '0' and '1'; shown in Table 2.3. The reconstruction occurs from the right side of the table, starting with the final two reduced source symbols from Table 2.2.

Original source			New symbol	Symbol Prob.	New symbol	Symbol Prob.	New symbol	Symbol Prob.
$m(i)$	No occ	Prob						
$m_1$	20	0.4	$m_1$	0.4	$m_1$	0.4	$m_{2345}$	0.6
$m_2$	10	0.2	$m_2$	0.2	$m_{345}$	0.4	$m_1$	0.4
$m_3$	10	0.2	$m_3$	0.2	$m_2$	0.2		
$m_4$	6	0.2	$m_{45}$	0.2				
$m_5$	4	0.08						

TABLE 2.2 Huffman Source Reduction Procedure

The higher probability symbol is arbitrarily assigned the value of '0' in Table 2.3. It could have been assigned a '1' and the lower probability symbol could have received a '0'. The symbol which is a combination of symbols ( $m_{2345}$ ) is broken into two symbols meaning there are now three. A '0' or a '1' is appended to the right of the original code of the two least probable symbols, following whichever convention was established with the first two symbols,  $m_1$  and  $m_{2345}$ . This process continues until a codeword is generated for each of the five original symbols. Table 2.3, which shows how to construct a codeword, must also be transmitted to the receiver so that the receiver may correctly decode received messages.

From Table 2.3, it is easily seen that the final codewords have the unique prefix property, thus no single code is a prefix (or subset) for another code. This is what is meant by instantaneously decodable. Each symbol can be transmitted and immediately decoded without confusion with any other symbol as the symbols arrive because the decompressor must already have a copy of the probability table. In communication channel applications, Huffman compression is further limited since a copy of the probability table must be transmitted with the compressed data to allow decompression at the receiver (Cappellini, 1985). As the decompressor receives code, it processes it in reverse. The decompressor starts at the tree's root and follows the sequence of incoming '1's and '0's through the tree until it reaches a leaf. The symbol attached to that leaf is the decoded character and the next bit received obviously starts a new character. Therefore, there is no need for the receiver to explicitly know the length of each symbol (Apiki, 1991). The requirement that the number of bits for each code must be an integer is a restriction on the Huffman codes efficiency. The ideal binary code length for a symbol  $m(i)$ , given by:

$$l(m(i)) = -\log_2 P(i), \quad (2.4)$$

where  $l$  is the wordlength and  $P(i)$  is the probability of symbol  $m(i)$ , is achieved only when its probability is a negative power of two, such as  $1/2$ ,  $1/4$ ,  $1/8$ , and so on, which results in integer values for the wordlengths. In that case, no bits would be wasted while representing decimal values like 2.67 bits, for example, since that word length would require three bits wasting part of the third bit.

Original source		Reduced source		Reduced source		Reduced source	
$m(i)$	Codeword						
$m1$	1	$m1$	1	$m1$	1	$m2345$	0
$m2$	01	$m2$	01	$m345$	00	$m1$	1
$m3$	000	$m3$	000	$m2$	01		
$m4$	0010	$m45$	001				
$m5$	0011						

TABLE 2.3 Huffman Codeword Construction Procedure

Thus, the chance of the Huffman code being set to ideal lengths is unlikely. The example described in Tables 2.2 and 2.3 accomplishes bit compression by reducing the average symbol length ( $L_{avg}$ ), Equation 2.5, from three to 2.52.

$$L_{avg} = \sum l(m(i))P_r \quad (2.5)$$

The original average symbol length is three because three binary bits are required to

differentiate between five symbols. The calculation of the compressed  $L_{avg}$ , using probability values from Table 2.2 and message lengths from Table 2.3 is:

$$[L_{avg} = 1(0.4) + 2(0.2) + 3(0.2) + 4(0.2) + 4(0.08) = 2.52 \text{ bits}].$$

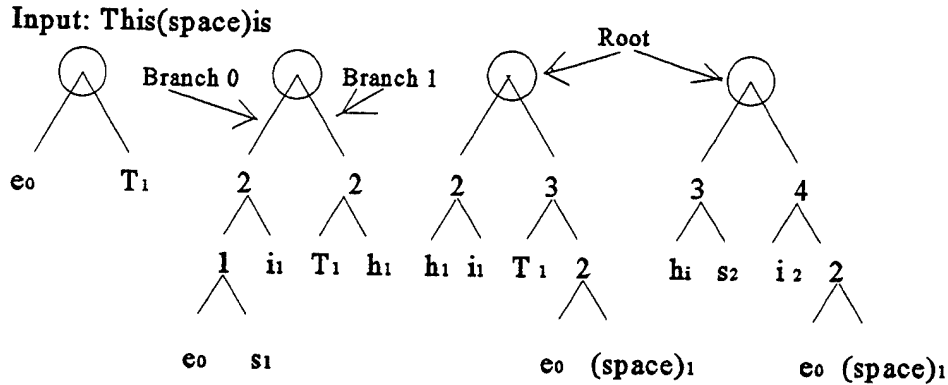
## 2. Dynamic Huffman Coding

A dynamic version of Huffman compression can construct the Huffman encoding tree on the fly while reading and actively compressing, effectively eliminating the above efficiency limitations. The encoding tree is continually corrected to reflect the changing probabilities of the input data (Knuth, 1985). Therefore, instead of first determining probabilities and then encoding as in static Huffman, the adaptive model initially assumes all symbol weights are zero and counts the symbol frequencies as it encodes them. After reading each symbol, the Huffman code is modified to account for the new character. Similarly, the decoder learns the symbol frequencies and updates the Huffman code in the same manner. The encoder and decoder remain synchronized because any changes to symbol probabilities in the encoder also occur in the decoder. The number of different symbols must be sent to the receiver to allow decoding of the compressed data. The adaptive Huffman tree is generated starting with an uninitialized tree and an empty leaf, which represents a node with no symbol attached to it, of zero probability. The following example, (Apiki, 1991), demonstrates the process. The input symbols are: "This is", including the space.

The example shows that the tree starts with the empty leaf,  $e_0$  and then sends the actual ASCII character the first time it encounters a symbol, generating a code for each symbol in the process. Therefore, when the T is received the tree looks like the left-most structure in Figure 2.1. The circle represents the root and if only a 'T' were being transmitted, then the ASCII character for a 'T' would be actually sent. By the time the



entire word 'This' has been seen by the encoder the tree has been changed to match the structure second from the left in Figure 2.1. The second time a symbol appears, its code is transmitted instead of its literal symbol. As the frequency of a symbol increases, in this case the symbols 'i' and 's', it move higher up the tree towards the root.



Tree after T      After s      After (space)      Final Tree

Output: T0h00i100s000(space)01111

Figure 2.1 Dynamic Huffman Tree Construction (Apiki, 1991)

The initial tree, held by both the compressor and decompressor, has only the root and a single empty leaf,  $e_0$ . The compressor starts the process by reading in a character. It attaches this character to the 1-branch of the root, leaving the empty leaf on branch '0'. It then sends this character to the decompressor as a literal ASCII code, and the decompressor make the same adjustment to its tree.

For each character read thereafter, the compressor performs the following steps. First, it check to see if the code is in the encoding tree. If the code is there, the compressor sends it in the same fashion as in the static case. If not, it sends the code for the empty leaf. Then it sends the new character as a literal ASCII code. Finally, the compressor adds two codes, one for a new empty leaf on branch 0 and one for the new code on branch 1. When the tree is full, the compressor just changes the last empty leaf node into the last character (Apiki, 1991).

In the example above, there are four trees shown. Each tree is labeled underneath, showing what the last character processed was. The intermediate trees which show the current tree structures after the 'h', and the first and second 'i's are not shown. The compressor starts out with just the empty leaf and then reads in character 'T'. The 'T' is placed on branch '1' and the empty leaf is placed on branch '0'. As the next character, 'h' is read in, the procedure in the quote above is followed. Since the 'h' is not already in the table, the code is sent for the empty leaf and the ASCII code for an 'h' is transmitted. Then the compressor builds a new tree, with the empty leaf attached to the root by branch '0' and the other two symbols 'T' and 'h' attached to the root by branch '1' which will have a weight of two; one for the number of occurrences of each symbol. The process continues with each new symbol. The numerals in Figure 2.1 indicate the weights of the nodes below it. This value must be updated so that when a node has a weight higher than a node above it, the two nodes are swapped, with the child nodes remaining in their same places. This is observed in the final structure where the 's' with a weight of two has been swapped to a higher node with the 'T' of weight one. These swaps are necessary to make the tree adapt to the changing probabilities of the data symbols. Again, note that each character is transmitted as it is read in and then a new tree structure defined which allows the decompressor to simultaneously make the same changes to its tree. The final output is shown on Figure 2.1 and the codes are deciphered as follows: The literal ASCII code for a 'T' is transmitted for the 'T', then the code for the empty leaf, '0' and the literal code for 'h' are sent for the 'h', next the new code for the empty leaf '00' is sent along with the literal code for the first 'i', 100s, where 100 is the new code for the empty leaf is sent for the first 's', 000(space) represents the space, then the Huffman code '01' is

transmitted for the second 'i' and the final '111' is the code for the second 's'.

The other major type of lossless data compression explored is binary source coding. Though binary source codes are typically grouped as redundancy-reduction methods, and thus usually require buffering, the next technique under consideration is very similar to Huffman coding. Its advantages over Huffman are presented next.

#### **D. ARITHMETIC CODING**

Though Huffman coding is largely considered the most efficient fixed-length lossless coding method, it has one major disadvantage. That is the requirement that symbol codes be an integral number of bits. As earlier stated, this only occurs for probabilities which are a negative power of two. If the symbol probability,  $P_i$ , is 1/5, for example, the optimum code length is given by:

$$-\log_2(P_i) = 2.32 \text{ bits.} \quad (2.6)$$

Huffman code would use two or three bits to encode the symbol. Clearly the compression is neither completely efficient nor maximum. Arithmetic coding provides a viable solution to this limitation. This technique represents the entire message as a number stream (Langdon, 1984). The entire symbol domain is encompassed on the interval of real numbers between zero inclusive and one exclusive,  $[0, 1)$ . Each symbol is assigned a range within the interval, corresponding to its probability.

### EXAMPLE 1

Table 2.4 demonstrates a sample interval range assignment. The initial range is  $[0, 1)$ . The initial range is divided up amongst the symbols starting with the highest probability symbol 'T'. The 'T' is assigned the interval  $[0.0, 0.4)$ , next the second symbol, 'I', is assigned twenty five percent of the total range  $[0, 1)$ , starting where the last symbol's interval ended, 0.4, since the symbol intervals cannot overlap. The procedure is followed until all symbols have been assigned a portion of the initial range,  $[0, 1)$ . Table 2.5 demonstrates the encoding of the word 'timing' based on the range assignments of Table 2.4. The first symbol to be coded assumes the same range on the initial interval,  $[0, 1)$ , as it is assigned in Table 2.4. Therefore, the current range is now  $[0.4, 0.65)$ .

As each additional symbol is processed, the range is narrowed to that interval within the current range which is allocated to the symbol. So when the 'I' is to be processed its probability,  $P_i$ , is multiplied by the current range producing the new range of 0.1.  $[0.4 * 0.25 = 0.1]$ . The greater a symbol's probability, the less it will reduce the current range; resulting in fewer bits being added to the code. This can be seen from the bracketed Equation above where the  $P_i$  for an 'I' is 0.25, which means the previous range is reduced to 25% of the initial range of 0.4 or 0.1. The choice of where to place the new interval within the range of  $[0.4, 0.65)$  is arbitrary. Thus, the new interval could be  $[0.49, 0.59)$  or any other 0.01 interval within  $[0.4, 0.65)$ . In Table 2.5, the new range was chosen to be  $[0.4, 0.5)$ . When the 'M' is processed, its probability is multiplied by the current range resulting in a new range of 0.01  $[0.1 * 0.1 = 0.01]$ . Again, this 0.01 interval is arbitrarily chosen from the entire range of  $[0.4, 0.5)$ . No confusion results from the arbitrary range selection because, just like in Huffman coding, the coding table must be sent to the receiver to decode messages. The process continues until one decimal number

is determined which represents the entire message, 'timing'. The decoding process is then fairly straightforward.

The first symbol is determined from the sub-interval, of the initial range, [0, 1) in which the encoded message falls. Since the received value is on the interval of [0.48314, 0.48320), it can be seen that the first symbol came from the sub-interval [0.4, 0.65) and must be a 'T'. The next symbol is determined by subtracting from the received encoded value, 0.48314 in Table 2.6, the low value from Table 2.4, of the first symbol and dividing by the probability,  $P_i$ , of the first symbol's range, 0.25. The symbol is then found via the interval in which the new encoded value falls. In Table 2.5, the following values result:  $[(0.48314 - 0.4)/0.25 = 0.33256]$ . This new value falls within the range [0.0, 0.4) indicating the second symbol is an 'I'. Results for the remaining decoding of symbols are illustrated in Table 2.6. Note that in actual coding, the values of the encoded numbers will be represented in binary. Decimal values were utilized in the above example to assist in concept understanding. Since the decoder interprets the encoded number 0.0 as a symbol ('I' in Table 2. 4) in the domain interval, an end of message symbol must be transmitted with the code.

Symbol	Probability	Range
I	0.40	[0.00, 0.40)
T	0.25	[0.40, 0.65)
N	0.15	[0.65, 0.8)
M	0.10	[0.80, 0.9)
G	0.10	[0.90, 1.0)

TABLE 2.4 Arithmetic Coding Range Assignment for Example 1

Symbol Number	Symbol	Low Value	High Value
2	T	0.40	0.65
1	I	0.40	0.50
4	M	0.48	0.49
5	I	0.480	0.484
1	N	0.4826	0.4832
3	G	0.48314	0.48320

**TABLE 2.5** Arithmetic Encoding Process for Example 1

**EXAMPLE 2**

Assume a stream of VVVVVVV's ' is to be compressed. The probability of V is known to be 0.9, while the end-of-message character has a probability of 0.1. The range [0, 0.9) is assigned to the letter V and [0.9, 1.0) is assigned to the end-of-message character. Table 2.7 displays the results (Langdon, 1984).

Encoded Number	Symbol	Low	High	Range
0.48314	T	0.40	0.65	0.25
0.33256	I	0.00	0.40	0.40
0.8314	M	0.80	0.90	0.10
0.314	I	0.00	0.40	0.40
0.785	N	0.65	0.80	0.15
0.90	G	0.90	1.00	0.10

**TABLE 2.6** Arithmetic Decoding Process for Example 1

New Character	Low Value	High Value
V	0.0	0.9
V	0.0	0.81
V	0.0	0.729
V	0.0	0.6561
V	0.0	0.59049
V	0.0	0.531441
V	0.0	0.4782969
END OF FILE	0.43046721	0.4782979

**TABLE 2.7** Arithmetic Encoding for Example 2

The value 0.4782979 would then be transmitted to represent a string of nine V's rather than sending nine, eight bit, ASCII characters. Like Huffman codes, arithmetic codes have a dynamic version and often use a zero-order Markov model; though higher-order models can be implemented. The major problem with arithmetic coding is that most computers cannot process numbers of the length needed to encode certain data types, i.e. images. This is can be seen in the example of Table VII in that the precision required to represent only a few more 'V's would result in a precision greater than the normal computer could achieve. This problem is overcome by only sending a portion of a message or data set which can be represented within the precision of the computer system sending it (Weiss and Schremp, 1993).

Another drawback is that of loss of precision between the high and low values as the ranges gets very small. This often results in the low value being higher than the high value and consequently, causing overflow. Inserting checks into the process prevent this problem at the expense of greater complexity (Langdon, 1984).

### III. LOSSY DATA COMPRESSION TECHNIQUES

#### A. LOSSY COMPRESSION THEORETICAL BACKGROUND

Lossy compression describes processes where information is irretrievably lost. It is typically used for applications where there is a notion of fidelity associated with the data. Such applications often involve digitally sampled analog data (still images, video, etc.) where it is only necessary that the decompressed data be acceptably close in quality to the original. Thus, it is very useful for audio compression applications where the human ear is not discerning enough to detect the loss. Lossy compression techniques are a subset of the entropy reduction class of data compression. The major types of lossy compression methods use some form of quantization. Efficient quantizer design requires an accurate statistical model of the data source. For this thesis, linear prediction is used to model the data.

There are three general categories of quantization compression: 1) Zero-memory, 2) Block and 3) Sequential. The distinction between these categories is not strong, as techniques which fall under one category also fit into another. Zero-memory quantization is the process where samples are quantized individually, while in block quantization, a block of input samples are represented by a block of output values chosen from a finite set of possible output blocks. Finally, sequential quantization quantizes input samples using information from its surrounding samples on a block or non-block basis (Knuth, 1985). Thus, the sequential techniques can all also be block quantization techniques. Under the zero-memory category are vector and scalar quantization. Multi-path search coders and predictive coders are the dominant form of sequential quantization techniques, which as



stated earlier can also be used in a block quantization form. The block quantization category encompasses various types of transform coding. Transform coding operates on a block of  $n$  samples generating an output sequence by matching the input block with its closest approximation from a codebook of sequences. The compression scheme is optimized by minimizing the difference between the two sequences. See Lynch (1985), and Jayant and Noll (1984) for additional information on transform coding. No further discussion of transform coding is presented here.

## **B. SCALAR AND VECTOR QUANTIZATION**

For the zero-memory method, each data sample is quantized independently of all other samples using the same quantizer, therefore, the system has no memory of previous data. The compression algorithm is optimized to minimize the quantizing noise using fixed quantizing levels. Another name for zero memory quantization is scalar quantization. Scalar quantization is just a special case of a very powerful data compression technique called vector quantization. Vector quantization is a process based on a codebook. A codebook is a collection of vectors or lists of "typical" data sequences. The codebook vectors are very similar to the branches in a tree structure used for tree or trellis coding which will be discussed later. The codebook vectors contain the parameters used to reproduce the original source sequences. The index associated with the parameter vector, or codeword, that most reduces the distortion between the source sequence and a reproduction of that sequence is transmitted as side information to a decoder. The decoder matches that index to the index of a codebook of optimal coefficient vectors. The selected optimal coefficient vector is then used to supply the parameters to reproduce the original source at the decoder. The closeness of reproduced data to original source

sequence is made in a mean squared error sense. The advantage of this procedure is the elimination of quantizing of the information sent that's not original data, the side information. Extensive searching can be required, when the codebook is large, to find the best parameter codewords.

The huge computational load incurred during the searching of the codebook is the main disadvantage of this approach. In addition to various audio applications, vector quantization is commonly employed in the compression of images. In this latter case, the source sequences are subarrays of pixels and the codebooks are vectors of pixel values which can reproduce the source pixels (Xue and Crissey, 1991). The process achieves compression in that the index of the reproduction parameters can be specified in fewer bits than the original pixels themselves. The following relationship for the number of bits required per pixel, or compression rate ( $r$ ), illustrates that principle:

$$r = (R/k) \text{ bits/pixel}, \quad (3.1)$$

where  $R = \log_2(m)$  is the quantizer rate in bits/vector,  $m$  is the number of codebook vectors, and  $k$  is the number of pixels. The input source vector can take many forms and a vector of pixels is just one example.

Another example involves character recognition from half-tone or fax data, where vectors are arrays of bits that are positioned over the character positions and the table (or codebook) is the alphabet of characters that are being recognized. Larger vectors and tables result in higher fidelity for a given amount of compression at the expense of increased computational resources. In scalar quantization, each vector consists of a single data element. In fact, the initial process of analog to digital data conversion is an example of scalar quantization (Bookstein and Storer, 1992).

## C. MULTIPATH SEARCH CODING

Sequential quantization, like block quantization, takes advantage of dependency between samples, and it has been shown that they can provide better performance than the scalar version of quantization. Two categories of sequential quantization exist: Multipath search coding and predictive coding.

Multipath search coding (MSC) is a relatively new form of quantization being studied. The motivation for its study stems from the performance bounds predicted by the rate distortion function for sources with memory developed by Shannon. Generally, this type of quantization, which is of the block type, has the potential for approaching the rate distortion bound with the least amount of design complexity. Trellis, and tree coding comprise the dominant types of multi-path search coding, though codebook implementations are increasingly used (Lynch, 1985).

### 1. Tree /Trellis Coding

MSC's use future as well as previous sample values in order to select a quantized version of a given input sample. They are often called look-ahead coders, or tree or trellis encoders due to this property. Compression is obtained by virtue of the fact that the selected quantized version of the input sample sequence is coded into a binary channel sequence wherein each sample is represented by one binary digit. The techniques of trellis and tree coding make use of geometric structures and are very similar to the vectors of vector quantization and the lists of list coding. Each typical sample is stored as a sequence of branches in a tree structure. When a sequence is selected, its corresponding tree path is transmitted as a binary sequence, with each bit corresponding to a direction at each sequential node of the tree, much like the procedure used in Huffman coding. A similar procedure is used in trellis coding since the trellis structure is really a truncated tree

structure (Jayant and Noll, 1984). The following example, (Jayant and Noll, 1984), illustrates the process.

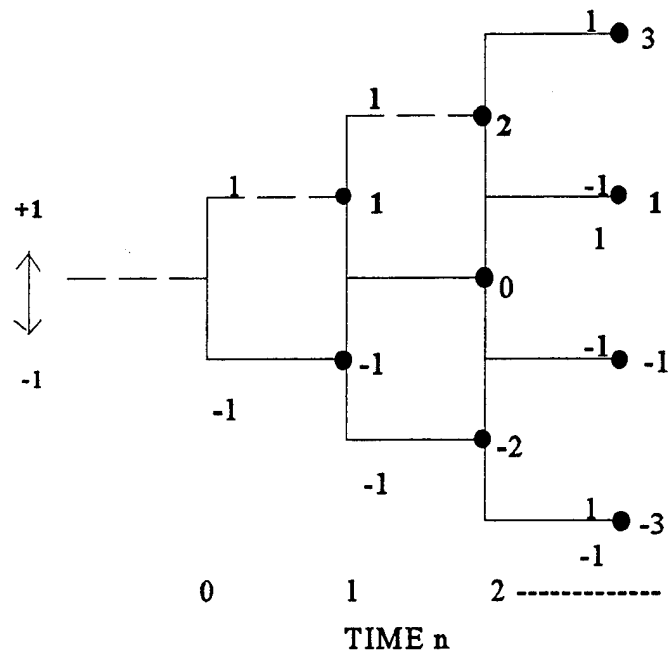


Figure 3.1 Delta Modulation Code Tree (Jayant and Noll, 1984)

The code tree of Figure 3.1 has a set of nodes for each time index  $n$ . The nodes are indicated by the black dots and the branches by the horizontal lines. Each node has  $2^R = 2$  branches, where  $R = 1$  bit/sample. In Delta Modulation, the decision outcome of a waveform encoder is input to the tree coder and each previous value is updated with a positive or negative step of fixed size. Figure 3.1 shows the possible DM outputs, which can be input to the tree coder, at either end of the two-headed arrow on the left side of the figure. A branch letter or reconstruction value, selected from the alphabet of

reconstruction values (-1, +1), is labeled on each horizontal branch. The tree is traced by following the branches (dashed in this case), which show the DM output at each time instant, to the nodes, where the sum of all the previous output values is labeled. The dashed-line branches are labeled with sample input values, and the path is located by a binary sequence called a path map (+1, +1, -1 for this example). The nodes with the bold numbers show the result at each time instant (Jayant and Noll, 1984). The bits corresponding to the reconstruction values are transmitted and determine the output sequences generated at the decoder. These bits lay out a path to follow through the tree. The decoder has the same tree structure and can follow the branches to each node, in a process that basically integrates the map sequence, to determine the correct value which was input to the tree coder. Unlike codebook coding, tree and trellis coding can be done on a sample-by-sample basis instead of on a block basis. This is very similar to maximum-likelihood or Viterbi decoding of convolutional channel codes. This latter process is used in the newest standard for cellular phone communications: Code Division Multiple Access (CDMA), which has wide ranging audio compression employment. A full treatment of trellis and tree coding is available in (Lynch, 1985).

## **2. Codebook Coding**

Codebook coding (also known as list coding) involves the use of a codebook of  $2^n$  highly probable  $n$ -sample sequences. The codebook vectors are highly probable since, in image coding, an image is scanned and used to train the program used to develop the codebook to produce reproduction vectors which are "highly probable". One sequence is selected from the codebook, that minimizes the distortion between itself and the actual input sequence (Lynch, 1985) and (Cappelini, 1985). The index of the selected sequence is coded as an  $n$ -bit word and sent to the receiver, where the same codebook is stored.

If each sample has  $M$  levels, then there are  $M^n$  possible sequences but only  $2^n$  "typical" sequences, giving a compression ratio of:

$$R = \frac{\log_2(M^n)}{\log_2(2^n)} = \log_2(M). \quad (3.2)$$

For example (Jayant and Noll, 1984), given a coder sequence of length  $N$  and rate  $R$  bits/sample, the total number of unique codewords is given by:

$$J = 2^{NR}. \quad (3.3)$$

A sequence of output samples,  $y_i : i = 1, 2, \dots, 2^{NR}$  is assigned to each codeword and the codebook is comprised of the set of all possible output sequences. The output sample values are selected from an "alphabet" of reproducing values. The codebook coder is also called a vector quantizer. Jayant and Noll (1984) describe the process, which is the same for vector quantization, as follows:

A codebook coder accepts a block  $x$  of  $N$  input samples, searches through the codebook with  $J = 2^{NR}$  entries, finds the output sequence best matching the input block, and transmits the corresponding codeword index  $i_{opt}$  to the decoder in the form of  $NR$  bits. The decoder looks up the corresponding codeword in its codebook and releases the  $N$  samples of  $y_{i, opt}$  as the output sequence.

Since a codeword is transmitted once for every  $N$  input sample block, the codebook coder is also a block coder; further evidence of its equivalence as a vector quantizer.

## **D. PREDICTIVE CODING**

Predictive coders include the well-known techniques of delta modulation and differential pulse code modulation (DPCM). Both of these techniques predict the next sample value and then quantize the difference between the predicted value and the actual value. The prediction is based on a weighted combination of previously predicted values. Delta modulation uses a 1-bit quantizer, whereas DPCM uses a k-bit quantizer.

### **1. Delta Modulation**

Delta modulation is a predictive coding technique in which the difference between a given sample and its predicted value is quantized into one of two levels ( $-\delta$ ,  $+\delta$ ). If the difference is positive,  $+\delta$  is coded, and if the difference is negative,  $-\delta$  is coded. The important feature of delta modulation is that it allows only two possible levels to be coded and transmitted. Thus, it is known as a "1-bit" system. Delta modulation can be done in two ways: conventional and adaptive.

### **2. Differential Pulse Code Modulation**

In Pulse code modulation (PCM), the original analog signal is time-sampled and each sample is quantized and transmitted as a digital signal. Instead of quantizing each sample, in DPCM, the next sample is predicted and the difference between the actual and predicted values is quantized. This is also the basis of delta modulation, thus many similarities exist between DM and DPCM. In DPCM, the predicted value, which is obtained from previous predicted values and differences, is also available at the receiver, since the identical predictor is used there. In many applications, a more accurate prediction can be obtained if more than just the previous sample is used. This is to be expected since many data sources produce sequential samples that are not independent. The number of previous samples to use for prediction and the predictor function itself

depend upon the statistical properties of the data source. Adaptive DPCM (ADPCM) is frequently employed to allow non-stationary signals to be tracked by the compression algorithm. In the next chapter, a more detailed examination of the ADPCM scheme is conducted.





## IV. LOSSLESS ADAPTIVE DIFFERENTIAL PULSE CODE MODULATION

### A. INTRODUCTION

Normally, when designing a DPCM system, it is assumed that the input data is stationary. Thus, a predictor and quantizer are designed with fixed parameters. But when the input data is non-stationary, these fixed-parameter designs show inconsistent and generally poor performance with respect to signal-to-quantizing-noise ratio. Adaptive designs have been used effectively in these cases and the approach boils down to one of three choices: an adaptive predictor with a fixed quantizer, a fixed predictor with an adaptive quantizer or an adaptive predictor and quantizer. In the ADPCM comparison to follow, an adaptive predictor with a fixed quantizer will be simulated.

Acoustical signal digitization uses 64 kb/s PCM (8 bits per 8 kHz sample), in communication networks. For efficiency, the transmission rate is reduced to 32 kb/s (4 bits per 8 kHz sample) with ADPCM coding. Redundancy removal is accomplished by subtracting a predicted value from each input sample, and entropy reduction is achieved by quantizing the difference between the input data sample and the predicted value to a limited number of amplitude levels. Speech contains a relatively high short-term correlation, therefore the power of the prediction error is less than that of the original signal and fewer bits are required to transmit the error signal. This is what makes ADPCM especially useful for audio data compression applications. Adaptive predictors are now a major component of 32 kb/s DPCM systems, and instrumental in obtaining the best performance in differential encoding systems below 32 kb/s.

The principal components of the ADPCM system are the quantizer, the binary encoder/decoder pair, and the predictor, shown in Figure 4.1. For the data compression

system simulation herein, the quantizer is a symmetric,  $2L$ -level one with fixed step size. The quantizer is implemented in Matlab code using the round function, which takes the input values and rounds them towards the nearest integer. The difference between the original prediction error and the quantized prediction error from the quantizer output is known as the quantization noise and represents the lossy part of the system which can not be recovered. However, in the adaptive systems implemented in this thesis input values were also rounded so that only integer data values were used. This form of lossless compression, using integer values, was used since data stored on a computer can only be represented to a limited finite precision. In addition, receiver outputs were sent through a round function. As a result, these ADPCM systems are lossless and quantization noise is eliminated.

In this chapter the ADPCM predictor systems used for the performance comparison are examined and the various system components presented. The finite impulse response and infinite impulse response predictors are developed and the process of their underlying algorithm, the least mean square algorithm is explained. Next, the modular function, added to the basic ADPCM system to reduce the size of the codebook, is introduced.

## **B. FINITE IMPULSE RESPONSE FILTER**

Figure 4.1 depicts the block diagram of an ADPCM system. These systems fall into one of two types, those that have a finite-duration impulse response (FIR) and those that have an infinite-duration impulse response (IIR), (Proakis and Manolakis, 1992). These two types of systems will be the basis of this thesis' comparative analysis. The binary coder block in Figure 4.1 performs lossless source coding of the residual

sequence,  $eq(k)$ . The coder assigns a binary word of length  $\log_2(2L)$  to each quantization level,  $L$ , on a sample-by-sample basis. The Huffman coding algorithm will be utilized to implement the binary coder. The box in Figure 4.1 labeled  $P(z)$  represents the adaptive predictor. The predictor structure is chosen to emulate an assumed model of the input signal process. In this thesis,  $P(z)$  is chosen to have a FIR structure defined as:

$$P(z) = \sum_{i=1}^M a_i z^{-i}, \quad (4.1)$$

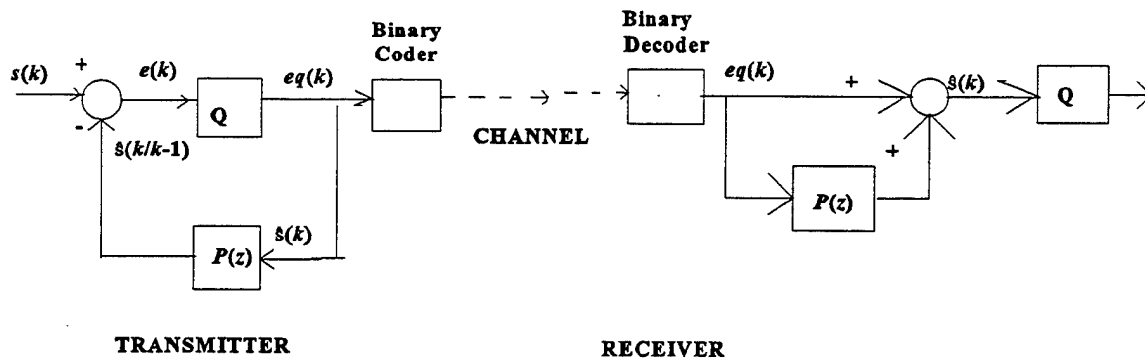
where  $a_i$  represents the predictor weights, based on the assumption that speech can be modeled appropriately by a linear prediction model. The transmitter portion of this structure is an all-zero model. It utilizes a finite impulse response filter (FIR), or a moving average (MA) filter, to predict output values.

In Figure 4.1, the transfer function  $H(z)$ , from  $eq(k)$  to  $\hat{s}(k/k-1)$ , is derived as shown below.

$$Z[\hat{s}(k/k-1)] = P(z)EQ(z),$$

Which leads to:

$$H(z) = \frac{Z[\hat{s}(k/(k-1))]}{EQ(z)} = P(z). \quad (4.2)$$



**Figure 4.1** FIR ADPCM system

$P(z)$  represents the transfer function of the MA predictor and  $H(z)$  is the transfer function of the transmitter portion between  $eq(k)$  and  $\hat{s}(k/k-1)$  which has an FIR structure.

The adaptive predictors of both the transmitter and receiver of Figure 4.1 use the least mean square (LMS) algorithm which determines the predictor weights by minimizing the instantaneous square of the error. This algorithm is used to allow the gradient of the error vector to be estimated from available data since no prior knowledge of the input signal correlation matrix,  $R$ , and the cross-correlation vector between the input and the desired response, is available. As a result, the weight coefficients of the predictors are not initially optimal weights like those derived using Wiener-Hopf equations, though the predictor weights do eventually converge to the optimal weights in stationary environments. However, the LMS algorithm performs far fewer calculations as a result as it does not require computation of matrix inverses. For the FIR ADPCM implementation, the standard LMS algorithm will be used. Haykin (1991) provides a detailed explanation of this process which results in three basic relations:

1. Filter Output:

$$v(k) = \underline{c}^H(k) \underline{eq}(k), \quad (4.3)$$

where  $\underline{eq}(k) = [eq(k-1), eq(k-2), \dots, eq(k-M)]^T$  a vector of length  $M$ ,  $v(k)$  is the output of the predictor, and  $\underline{c}(k)$  is the vector of filter coefficients which are updated after each input sample.

2. Estimation error:

$$e(k) = s(k) - v(k), \quad (4.4)$$

3. Predictor weight update:

$$\underline{c}(k+1) = \underline{c}(k) + \mu \underline{eq}(k) e(k). \quad (4.5)$$

The step size  $\mu$  is used to control the updates to future weight coefficients. The bounds on this variable are derived from the bounds for updating the steepest-descent algorithm. In that case, the steepest descent algorithm converges when:

$$0 < \mu < \frac{1}{M R_{eq}(0)}, \quad (4.6)$$

where  $M$  is the length of the predictor and  $R_{eq}(0)$  is the autocorrelation of the input sequence. However, for the LMS algorithm an initial value for  $\mu$  is usually chosen as 10% of the maximum value,  $\mu$ , obtained in Equation 4.6.

### C. INFINITE-IMPULSE RESPONSE FILTER

In contrast to the standard, FIR ADPCM implementation, the IIR filter system output is weighted by the linear combination of the past input and output samples instead of just a finite number of past input samples. This fact gives rise to the notion of infinite memory or infinite impulse response. To accurately model the vocal tract, the model should contain zeros as well as poles, and an ADPCM system transmitter based on a pole-zero model is shown in Figure 4.2. Its transfer function,  $H(z)$ , from  $eq(k)$  to  $v(k) = \hat{s}(k/k-1)$  is derived from the following equations:

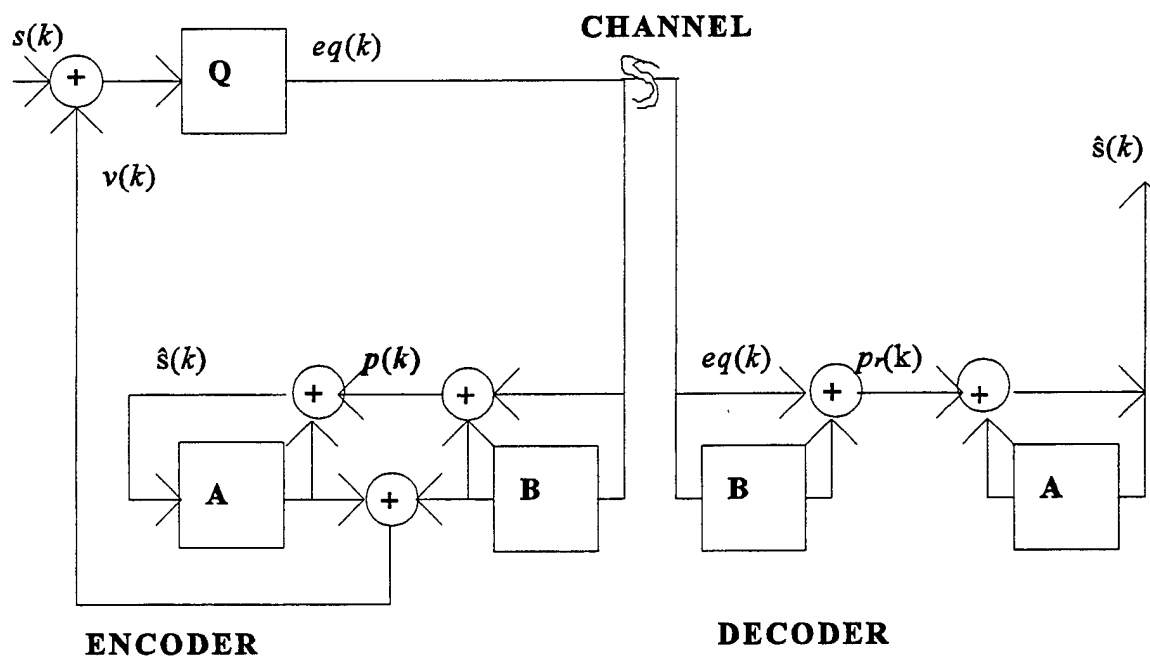
$$\begin{aligned} Z[v(k)] = v(z) &= B(z)EQ(z) \\ &+ [B(z)EQ(z) + EQ(z)]A(z) + A(z)[v(z) - B(z)EQ(z)], \end{aligned}$$

Which leads to:

$$v(z)[1 - A(z)] = [A(z) + B(z)]EQ(z).$$

Therefore,

$$H(z) = \frac{v(z)}{EQ(z)} = \frac{A(z) + B(z)}{1 - A(z)}. \quad (4.7)$$



**Figure 4.2** IIR ADPCM System

Thus, it is a standard IIR predictor where zeros and poles can be individually specified.

For the IIR filter implementation, a version of the International Telephone and Telegraph Consultative Committee (CCITT) G.721 recommendation is used (CCITT, 1984). The basic LMS algorithm has been modified to provide additional system stability restraints, ensure synchronization between the transmitter and receiver, and prevent drift of the AR portion of the receiver (Bonnet et al, 1990). The block labeled 'A' in Figure 4.2 represents the AR portion of the system and the 'B' block represents the MA portion. The modification to the basic LMS algorithm made to the G.721 algorithm lies in the



addition of leakage factors,  $\delta_1$  and  $\delta_2$ , the stability constraints on the AR predictors weight coefficients, and the definition of the signal  $p(k)$ .

The effect of the modifications are to allow the updating of the AR portion in the same fashion as the MA portion is updated, i.e., like a transversal filter vice a recursive one (Bonnet, 1990). The stability constraints are explained below. Last, the signal  $p(k)$  improves adjustment of the receiver onto the transmitter and allows resynchronization of the receiver to the transmitter in the presence of transmission errors. Bonnet (1990) defines the signal  $p(k)$  in the z-domain as:

$$P(z)=[1-A(z)]\hat{S}(z), \quad (4.8)$$

where  $\hat{s}(k)$  represents the input to the AR predictor. As a result of these modifications to the basic LMS algorithm, the modified CCITT G.721 update equations become:

$$\begin{aligned} a_1(n+1) &= (1-\delta_1)a_1(n) + \alpha_1 \text{sgn}(p_n) \text{sgn}(p_{n-1}) \\ a_2(n+1) &= (1-\delta_2)a_2(n) + \alpha_2 \text{sgn}(p_n) [\text{sgn}(p_{n-2}) - f(a_1(n)) \text{sgn}(p_{n-1})] \\ b_j(n+1) &= (1-\delta_j)b_j(n) + \alpha_j \text{sgn}(eq_n) \text{sgn}(eq_{n-j}) \quad j=1, \dots, 6; \end{aligned} \quad (4.9)$$

where  $a_1(n+1)$  and  $a_2(n+1)$  represent the recursive weights for a length two AR predictor,  $b_j(n+1)$  represents the  $j$ th weight coefficient for a length six MA predictor, and  $\text{sgn}(\ast)$  stands for the sign function. Note that the parameters  $\alpha_1$  and  $\alpha_2$  perform the same function as  $\mu$  does in Equation 4.5. Similar equations hold at the receiver. The function  $f(a)$  is used as a stability constraint to control the boundedness of the AR parameter  $a_2$  and its range of values are given by:

$$f(a) = \begin{cases} 4a & \text{if } \text{abs}(a) \leq 0.5 \\ 2\text{sgn}(a) & \text{otherwise.} \end{cases} \quad (4.10)$$

Note that, the value of 0.5, Equation 4.10, was selected by Bonnet et al. to correspond to a limit on  $\alpha_1$  for frequencies in the middle of the telephone bandwidth (Bonnet, 1990). This algorithm was modified from a standard LMS algorithm by the addition of the leakage factors  $\delta_1$  and  $\delta_2$ . These terms are used to control the drift in the AR parameters at the decoder. Nominal values for  $\delta_1$  and  $\delta_2$ , and the updating factors,  $\alpha_1$  and  $\alpha_2$  are:

$$\begin{aligned} \delta_1 &= 2 \times 10^{-8}, & \delta_2 &= 2 \times 10^{-7} \\ \alpha_1 &= 3.2 \times 10^{-8}, & \alpha_2 &= 2 \times 10^{-7}. \end{aligned} \quad (4.11)$$

Actual values used for the IIR varied by a factor of 100 or 1000 depending on the signal and are given in the following chapter when the signals are introduced. Additionally, the actual values of  $\alpha_1$  and  $\alpha_2$  are limited to the following range to ensure stability.

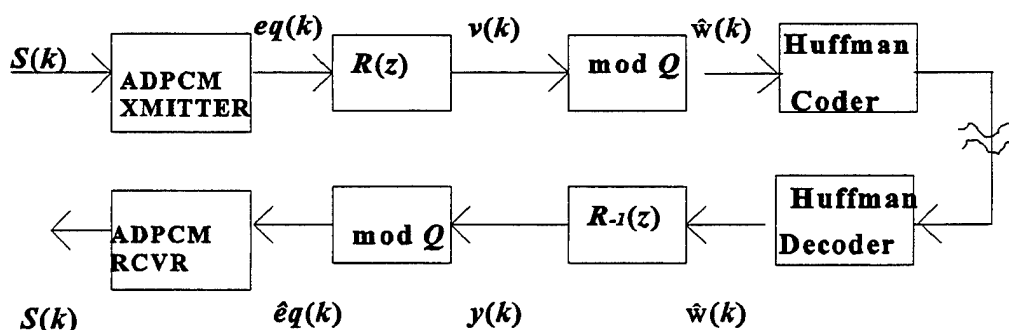
$$|\alpha_2| \leq 0.75, \quad |\alpha_1| \leq 1 - 2^{-4} - \alpha_2. \quad (4.12)$$

#### D. MODULAR ARITHMETIC FUNCTION

The standard ADPCM compression system transmits the quantized error signal. The amount of data compression achievable is dependent upon the entropy of the error vector. The first-order entropy, corresponding to a coder which encodes each error,  $eq(n)$ , separately, is given by:

$$H_{eq} = - \sum_{j=0}^{Q-1} P_{eq}(j) \log P_{eq}(j), \quad (4.13)$$

where  $P_{eq}(j)$  is the relative frequency of the symbol  $eq(k)=j$  and  $Q$  is the number of discrete integer values assumed by the input  $eq(k)$  (Einarsson, 1991). The entropy of the error sequence can usually be reduced using a modular function such as the one depicted in Figure 4.3. This figure shows the output of the ADPCM predictor,  $eq(k)$ , as the input to a residual function which produces the integer-valued sequence,  $v(k)$ . This sequence is then input to the modular function, which uses modular arithmetic to generate integer-valued data,  $w(k)$ . The lossless procedure presented by Einarsson to reduce the size of the codebook is a two-step procedure.



**Figure 4.3** ADPCM system with Modular function

The modular coding process described by Einarsson (1991) starts with a positive data sequence,  $eq(k)$   $k = \dots -1, 0, 1, 2, \dots$ , constrained in a range of  $Q$  discrete values. The next step of the procedure is to generate a sequence of integer-valued residuals  $v(k)$ ,  $k =$

... -1,0,1,2,...., constrained in a range of  $Q$  discrete values. The next step of the procedure is to generate a sequence of integer-valued residuals  $v(k)$ ,  $k = \dots -1, 0, 1, 2, \dots$ , referred to as the shifted sequence. This step is accomplished through the application of a linear filtering operation between the input data,  $eq(k)$  and a linear filter,  $R$ , with coefficients  $f_0=1$  and  $f_1=-1$ . The filter  $R(z)$  forms the first difference of  $eq(k)$  and is given by:

$$v(k) = eq(k) - eq(k-1). \quad (4.14)$$

Then the modular arithmetic function,  $\text{mod } Q$ , is applied to the sequence  $v(k)$  to reduce it to the appropriate range of 0 to  $Q-1$  before it is entropy coded. Table 4.1 depicts the process. Therein, sample data values for the input,  $eq(k)$ , are given, shifted to make them all positive, filtered through the residual filter  $R$ , and processed by the modular function to generate the resulting "modular sequence"  $\hat{w}(k)$ . The binary coding process is external to this procedure and is not shown. At the receiver, the linear filter operation is reversed and the modular operation is applied exactly as before resulting in the same original data values.

Note however that the shift, (+4) and (-4), illustrated in Table 4.1, is not required and this step could have been skipped. This shift was done to match the procedure given by Einarsson exactly, by starting with all positive data. The algorithm to reconstruct the original data at the receiver is derived from:

$$\hat{eq}(k) = [\hat{w}(k) - (y(k) - y(k-1))] \text{mod } Q, \quad (4.15)$$

where  $\hat{eq}(k)$  is the reconstructed  $eq(k)$ . The entropy,  $H$ , of  $\hat{w}(k)$  is given by:

$$H_{\hat{w}} = - \sum_{j=0}^{Q-1} P_{\hat{w}}(j) \log P_{\hat{w}}(j), \quad (4.16)$$

TRANSMITTER				RECEIVER	
$eq(k)$	shift (+4)	$v(k)=$ $eq(k)-$ $eq(k-1)$	$\hat{w}(k)$ mod $Q$ $Q=12$	$y(k) =$ $\hat{w}(k)+$ $y(k-1)$ mod $Q$ $Q=12$	Shift (-4) $\hat{e}q(k)$
7	11	11	11	11	7
5	9	-2	10	9	5
2	6	-3	9	6	2
-3	1	-5	7	1	-3
-4	0	-1	11	0	-4
6	10	10	10	10	6

TABLE 4.1 Modular Coding Process

where  $P_*(j)$  is the frequency of occurrence of the symbol  $\hat{w}(k)=j$ . Einarsson showed that the entropy for the modular function residual,  $\hat{w}(k)$ , is lower than that for  $v(k)$ . For  $Q=256$ , the residual  $v(k)=eq(k)-eq(k-1)$  will take on integer values in the range (-255, 255). Thus,  $eq(k) = 239$ , and  $eq(k) = -17$  reduced modulo 256 both result in  $v(k) = 239$ . Therefore the entropy of  $v(k)$  is the sum of two terms as shown in Equation 4.18,

$$H_v = P_v(j_1) \log P_v(j_1) + P_v(j_2) \log P_v(j_2), \quad (4.17)$$

where  $v_1(k)=j_1$  and  $v_2(k)=j_2$  both result in  $\hat{w}(k)=j$ . Writing Equation 4.17 in terms of the entropy for  $\hat{w}(k)$  gives the single term Equation:

$$H_{\psi} = P_{\psi} \log P_{\psi}(j), \quad (4.18)$$

where  $P_{\psi}(j) = P_{\psi}(j_1) + P_{\psi}(j_2)$ . Thus, since for  $a, b > 0$

$$(a + b) \log(a + b) > a \log(a) + b \log(b), \quad (4.19)$$

then  $H_{\psi} < H_w$ . Despite this theoretical advantage of reduced entropy, Einarsson states that the difference is usually negligible. The real advantage of using the modular coding function is a decrease in the size of the Huffman codebook necessary to encode the transmitted error vector, which, in turn, implies that fewer bits will be required. This modular function is incorporated into both the FIR and IIR versions of the ADPCM data compression systems whose performances are compared in Chapter 5.



## V. PERFORMANCE COMPARISON

### A. OVERVIEW

Generally, performance comparisons of ADPCM compression systems have been made using PCM systems as a baseline; often using the signal to quantizer noise ratio as a basic parameter of comparison. When directly comparing ADPCM coders, the mean square error (MSE) is usually used since these coders are designed to minimize this parameter. This measurement would be valid for the FIR coder as it is designed to minimize the MSE. However, due to the modifications of the LMS algorithm incorporated into the G.721 IIR coder reviewed herein, the IIR coder doesn't minimize the MSE but the quantity given by the following relation:

$$E(eq_n)^2 + \frac{\delta}{\alpha} E(\underline{A}_n^t \underline{A}_n), \quad (5.1)$$

where  $E(\bullet)$  is the expectation operator,  $\alpha$  is a positive adaption parameter,  $\delta$  is a positive leakage factor and  $\underline{A}_n$  is the vector of the AR parameters at time  $n$  (Bonnett, 1990). The first term of Equation 5.1 represents the MSE of the system while the second term is a weighted version of the AR predictor parameters. Thus, comparing MSE's between the FIR and IIR coders would not be completely accurate. Of course, MSE's are just one parameter that is frequently compared in ADPCM comparisons. Others include compression ratios, sound fidelity, variances, compression/decompression time and SNR's. In fact, no single measurement is sufficient to completely, reliably and easily classify the coders performance. This is one reason for the importance of a variety of studies investigating all the relevant properties of ADPCM coders. In this report, the variance of the original signal is compared to the variance of the quantized error signal. Additionally, the compression ratios of each type of coder is also compared. The compression ratio has



many definitions; it is defined in this thesis as the number of bits needed to encode the original signal in a PCM format divided by the number of bits used to encode the transmitted error sequence. The Huffman coder is utilized to encode the quantized error and to determine the compression ratio achieved.

## B. PRESENTATION OF DATA

Five data signals were examined. Four of them were audio signals and the fifth is a sinusoid, which was included since it is nonpersistently-exciting, and is of the type most difficult for the IIR ADPCM compressor to handle (Bonnet, 1990). The original signals, shown as the middle image in Figures 5.1-5.5, are:

- 1.) PFREE - a fast-paced pop song with a male voice and music,
- 2.) GUITAR - a musical selection played on a guitar,
- 3.) VOICE - a male voice speaking in a normal tone
- 4.) TRANSIENT - a filtered version of a male voice speaking,
- 5.) SINE - a sinusoid generated from  $s(k) = \sin(2\pi 4/100 k)$ ,  $k=1, \dots, 7025$ ,

where  $k$  is the number of points in the signal. All the signals had a duration of 7025 points. Each signal had a zero mean. The upper plot in each Figure is the IIR receiver output signal and the lower plot represents the FIR receiver estimate of the original signal shown in the middle plot. In every case the received signal is an exact integer representation of the input data after about three filter lengths. Note that the IIR predictor was implemented with different values than the nominal values proposed by Bonnet for the leakage factors,  $\delta_1$  and  $\delta_2$ , and update control variables,  $\alpha_1$  and  $\alpha_2$ , given in Chapter 4, Equation (4.11) and repeated here as Equation 5.2:

$$\begin{aligned} \delta_1 &= 2 \times 10^{-8}, & \delta_2 &= 2 \times 10^{-7} \\ \alpha_1 &= 3.2 \times 10^{-8}, & \alpha_2 &= 2 \times 10^{-7}. \end{aligned} \tag{5.2}$$

When using these nominal values for leakage and update factors, the compression achieved in the ADPCM coder was significantly lower for the IIR version than for the FIR version. As a result, these values were experimentally determined to produce the maximum compression in the quantized error power. Each actual value used was increased by a factor of 100 for the PFREE, GUITAR, and VOICE signals, a factor of 1000 for the TRANSIENT signal and were unchanged for the SINE signal. The FIR predictor used an update control variable,  $\mu$ , experimentally picked to provide the best compression for each signal and a filter order of six in each case. The actual values were  $4 \times 10^{-4}$ ,  $6 \times 10^{-4}$ ,  $10^{-5}$ ,  $5 \times 10^{-6}$ , and  $6 \times 10^{-5}$ , listed in the same order as the corresponding signal. The filter order of six was used for two reasons. First, to make it equal to the order of the AR portion of the IIR coder which also uses a predictor of order six. Naturally, this was not done in expectation of producing similar results but rather as a convenient starting point to experiment with. Secondly, experiments with predictors of order 2, 4, 5, 7, 8, and 10 all provided inferior results in terms of reducing the power of the error sequence. These experimental results were expected for the filter orders less than six but were inexplicable for filter orders greater than six. Nonetheless, the results were repeatable and six was determined to be the best filter order for this implementation. Time for compression and decompression was significantly lower for the FIR model as well.

The power of the quantized error given for each coder was one of the parameters measured to determine the amount of compression achieved by each ADPCM coder. All five signals were compressed at two different SNR decibel levels. The SNR was increased by increasing the power of the input signal while maintaining the noise level at a constant. This was done by multiplying the original signal by a randomly selected scaling factor of 2 to produce a six dB increase. Each signal power was determined and Table 5.1 lists the results. In this table, the original input signal power,  $\sigma_{IP}^2$ , and the quantized error

power, shown as  $\sigma_{EQ}^2$ , are presented. The reduction in power is listed as  $P/R$ . Greater compression is represented by a higher percentage of power reduction, in the ADPCM coder itself. The power reduction (in percentage) is determined by:

$$P/R = 1 - \frac{\sigma_{EQ}^2}{\sigma_{IP}^2} \quad (5.3)$$

The fifth and eleventh columns, respectively, show the power reduction for the IIR and FIR coders. A direct comparison between these columns shows that the FIR coder performed better, higher  $P/R$ , in this category than the IIR coder, though both coders significantly reduced the signal power as compared to the original signal variances. Of particular note is the large difference in compression between the two systems for a sinusoidal input. Bonnet (1990) prefaced this result by noting that the IIR is less effective on nonpersistently excited signals. In addition to measuring the power of each signal, the transmitted quantized error signal was input to the modular coding function and its output,  $\hat{w}$ , coded using Huffman coding to determine the compression ratios ( $CR$ ) achieved on each signal. Again, a higher compression ratio implies better compression than a lower ratio, where compression ratio,  $CR$ , is defined by:

$$CR = \frac{\text{total no. of sample points} \cdot 8}{\text{total no. of bits used}} \quad (5.4)$$

For the first four signals the compression ratio achieved in the IIR is superior, as shown in Table 5.1- column three versus column nine, with the only exception being the sinusoid at both dB levels. The reduced signal power of the error sequence output from the FIR coder, allowing fewer code words to be generated by the Huffman coder, would tend to

explain this exception. In the middle section of Table 5.1, columns six and seven, the compression ratios achieved by sending the quantized error signal through the Huffman coder without using the modular function are listed. This section is included to allow a tabular comparison of the compression achieved with and without the modular function. Thus column three compares directly to column six, for the FIR, and column seven to column nine, for the IIR coder. In every instance of comparison between the IIR systems and all but two cases of comparison between the FIR systems (Transient and Sine signals at lower db level) it can be seen that the  $CR$  achieved going through the modular function is greater. This result agrees with Einarsson's (1991) paper wherein he stated that the  $CR$  might theoretically increase. However, in most of the cases displayed in Table 5.1, the modular function  $CR$  was significantly better, exceeding Einarsson's projections of only modest  $CR$  increases. As stated above, the compression ratio,  $CR$ , was determined by totaling the number of bits used to encode the residuals and dividing that value into the number of bits required to encode an 8-bit PCM signal.

Table 5.2 (IIR coder) and 5.3 (FIR coder) show the results of the comparison of a number of other parameters, including average wordlength,  $L_{avg}$ , entropy, ( $H$ ), relative redundancy ratio, ( $RR$ ), and maximum compression, ( $R_{max}$ ). All of the values used in both Table 5.2 and Table 5.3 come from the higher SNR data set of Table 5.1. Specifically,  $CR$ 's obtained for the IIR implementation at the higher SNR level in Table 5.1, column three, coincide with the modular sequence values shown in Table 5.2,  $CR$  row. Likewise, compression ratio values obtained for the higher SNR levels in the FIR implementation of Table 5.1, column seven, correspond to the values in the  $CR$  row of Table 5.2. The tables are provided solely to assess the effectiveness of Einarsson's modular function on the operation of each coder and not to form a basis of comparison between the two coders. Thus, each table displays an intra-coder only comparison between  $\hat{w}(k)$  and  $eq(k)$ . The reasoning for this limitation is provided below. In both Table 5.2 and 5.3, the results show

a smaller average wordlength for the modular sequence,  $\hat{w}(k)$ , as compared to the error sequence,  $eq(k)$ , with the only exception being in the case of the IIR coder and the guitar signal where the opposite is true.  $L_{avg}$  is derived from the relation of Equation 5.4:

$$L_{avg} = \sum_{i=1}^M l_i p_i \quad (5.5)$$

where  $l_i$  is the length of the codeword 'i',  $P_i$  is the probability of the codeword 'i', and  $M$  is the total number of codewords. This result is expected from Einarsson's (1990) conclusions. In addition to  $L_{avg}$  being smaller for  $\hat{w}(k)$ , Table 5.3 (FIR coder) shows, that the entropy,  $H$ , is higher, in each case for the FIR error sequence,  $eq(k)$ , than for the FIR modular sequence,  $w(k)$ . The entropy, defined in Equation 2.2 and repeated here as Equation 5.6 is:

$$H = - \sum_{i=1}^M P_i \log_2 P_i \quad (5.6)$$

This development, in turn leads to the higher  $CR$  achieved on the FIR modular sequence as compared to the FIR error sequence. In Table 5.2 (IIR coder), this same relation, where a lower  $H$  leads to a higher  $CR$ , also holds true, again with the exception of the guitar signal. The guitar signal error sequence has a lower  $H$  than does the residual sequence. Despite this contradiction, the  $CR$  is still higher for the IIR modular sequence than it is for the IIR error sequence. The data is insufficient to explain this anomaly, however, since nine of ten such comparisons (across both Tables 5.2 and 5.3) follow the above relationship, it still appears valid. Relative redundancy gives a theoretical measure of how much a signal can be compressed and is higher for the modular sequence (than for

the error sequences) indicating a greater compressibility. The relative redundancy,  $RR$ , is defined by:

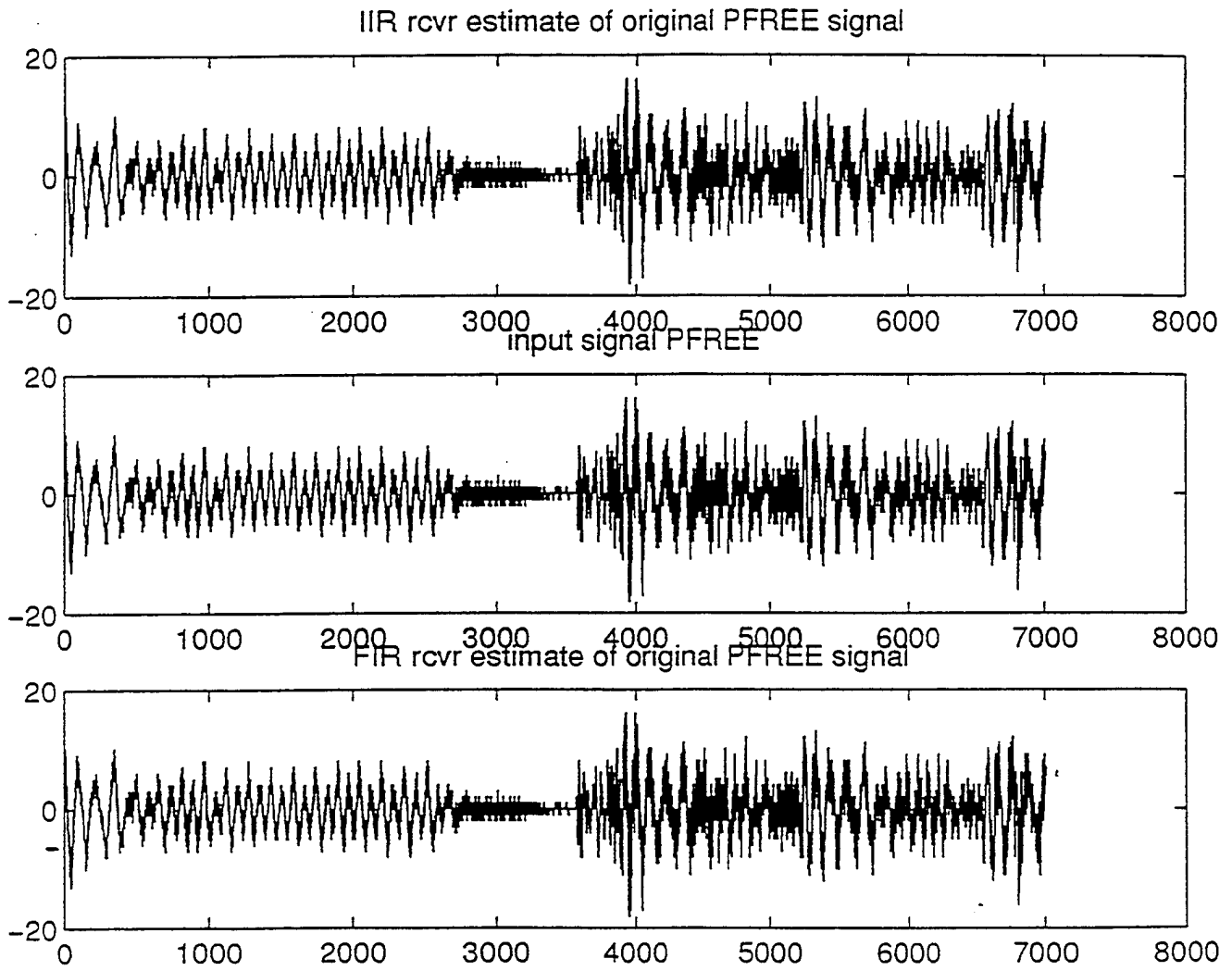
$$RR = 1 - \frac{H}{\log_2 M} \quad (5.7)$$

and is related to the maximum compression ratio by the following relation:

$$R_{max} = \frac{1}{1 - RR} \quad (5.8)$$

where  $H$  is the entropy of the signal,  $M$  represents the number of amplitude levels in the compressed signal, and  $R_{max}$  is the maximum compression that can theoretically be achieved on the given signal. Since there are different numbers of amplitude levels in the FIR and IIR signals, the  $M$  parameter, the utility of comparisons between Tables 5.2 and 5.3, is reduced. Each ADPCM coder and associated modular function reduces its error sequence to  $Q$  integer levels. However, the  $Q$  value is signal and coder system dependent. Therefore, just as comparisons cannot be made between the same parameters of different signals, no direct comparison can be made between the  $RR$ 's or  $R_{max}$ 's of each coder because they depend on  $M$  which is  $Q$  dependent. Conversely, comparisons can be made between the  $CR$  of each coder system due to the way  $CR$  is defined which is simply as the ratio of the total number of bits used to encode a given compressed signal to the number required to encode the same signal in PCM format. Thus, comparing  $CR$ 's is just a simple comparison of total bits used, irrespective of other parameters. In a similar manner, although better compression is achieved when  $RR$  increases and likewise for the  $CR$ , these two parameters are not related. The  $RR$  is an imprecise theoretical value and is only a guideline used to determine a theoretical maximum compression which is usually never reached (Lynch, 1984). In any case,  $RR$  is dependent on  $M$  and  $H$  and thus is a direct

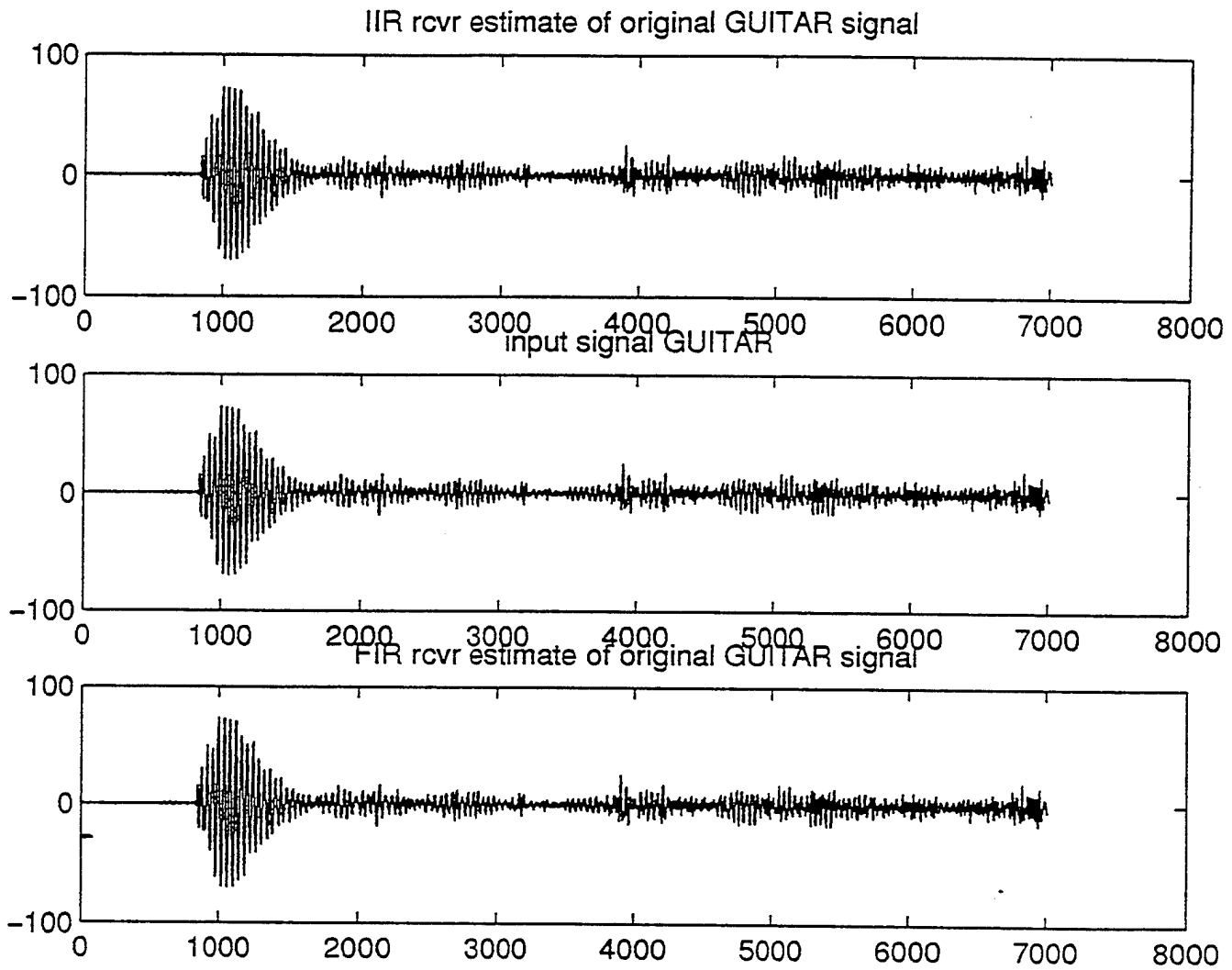
function of compression achieved in the ADPCM coder itself. Alternately, the  $CR$  is based on a comparison between the actual number of bits used in the Huffman coder and the number which would have been required for PCM and is only indirectly related to coder compression. Therefore,  $RR$  and  $CR$  can move independently of one another. In effect then,  $RR$ 's only purpose is to calculate  $R_{max}$  which was determined for both the modular sequence,  $\hat{w}(k)$ , output from the modular function and the original error sequence,  $eq(k)$ , output from the ADPCM compressors. This term is included in Tables 5.2 and 5.3 only to show that the actual  $CR$  achieved in the Huffman coders did not achieve the theoretical maximum value.



---

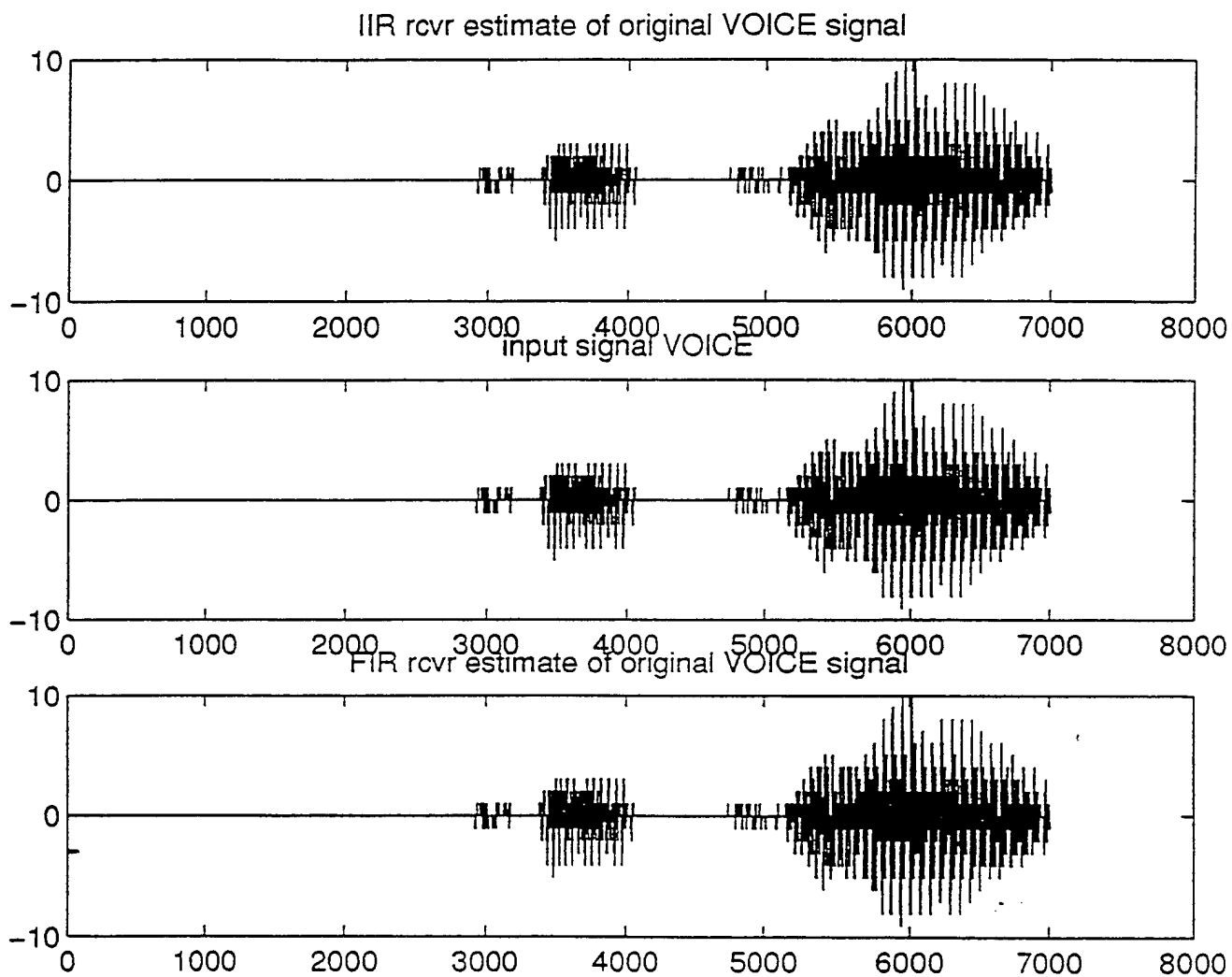
**Figure 5.1** PFREE SIGNAL, Sampling Frequency  $f_s = 8$  kHz, time shown in no. of samples





---

Figure 5.2 GUITAR SIGNAL, Sampling Frequency  $f_s = 8$  kHz, time shown in no. of samples



---

**Figure 5.3** VOICE SIGNAL, Sampling Frequency  $f_s = 8$  kHz, time shown in no. of samples

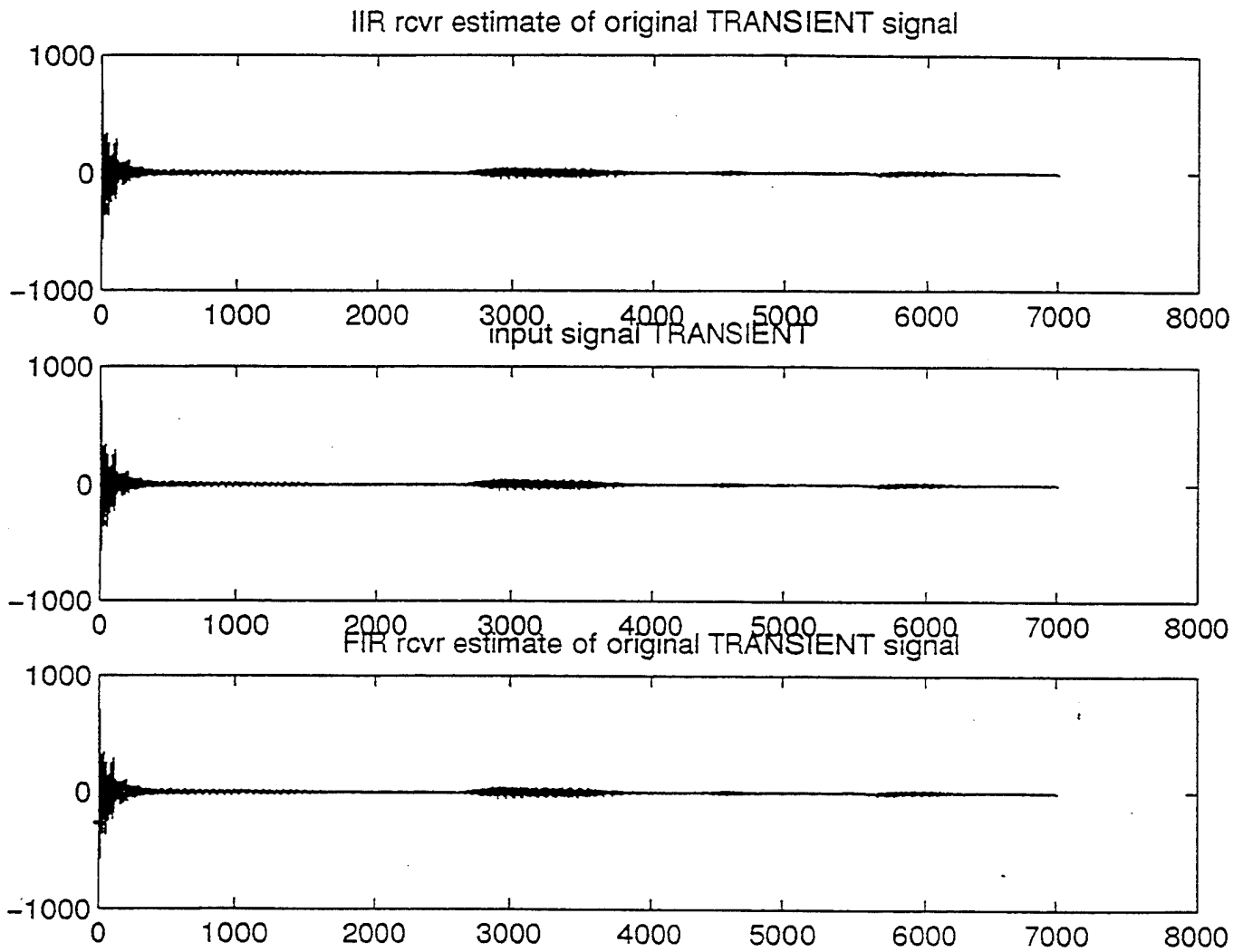
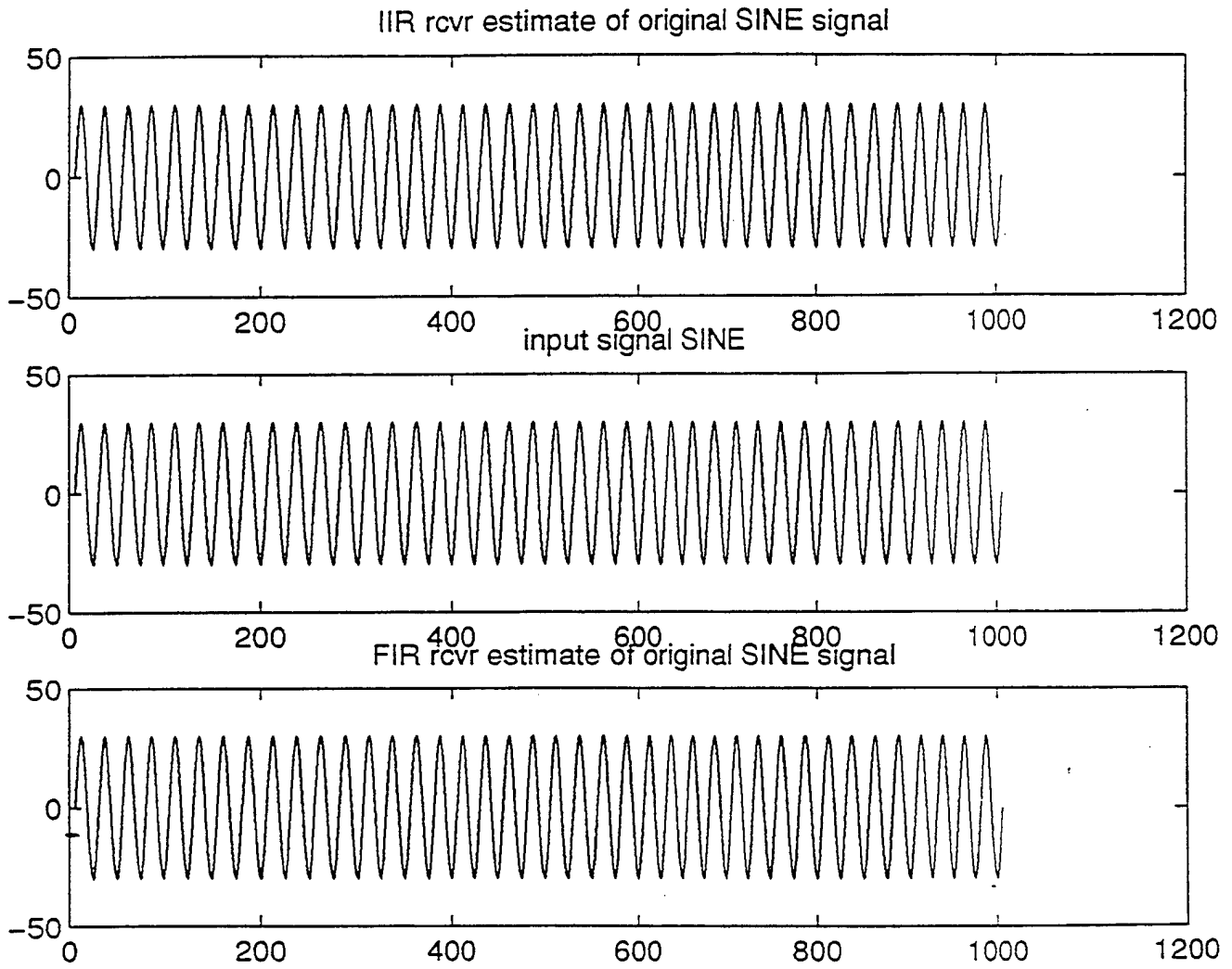


Figure 5.4 TRANSIENT SIGNAL, Sampling Frequency  $f_s = 8$  kHz, time shown in no. of samples



---

**Figure 5.5** SINUSOID SIGNAL, Sampling Frequency  $f_s = 8$  kHz, time shown in no. of samples

		MOD FUNCTION			W/O MOD		MOD FUNCTION			
S/N		IIR	IIR	IIR	IIR	FIR		FIR	FIR	FIR
dB	Signal	<i>CR</i>	$\sigma^2_{EQ}$	<i>P/R</i> %	<i>CR</i>	<i>CR</i>	$\sigma^2_{IP}$	<i>CR</i>	$\sigma^2_{EQ}$	<i>P/R</i> %
20	Pfree	2.00	0.60	14.0	0.81	0.62	0.70	1.36	0.49	30.0
20	Guitar	1.40	1.20	54.0	0.69	0.61	2.60	1.20	0.60	77.0
26	Voice	6.60	5.20	48.0	1.74	1.88	9.90	6.20	4.63	53.0
28	Trans	3.20	11.7	23.0	2.95	3.17	15.1	3.10	6.01	60.0
25	Sine	0.80	11.8	01.0	0.52	0.92	12.5	0.91	0.89	93.0
dB										
26	Pfree	2.10	9.30	37.0	0.85	0.68	14.8	1.54	3.79	74.0
26	Guitar	1.60	50.0	59.0	0.72	0.65	122	1.40	7.52	94.0
32	Voice	7.01	419	42.0	2.01	2.03	723	6.06	314	57.0
34	Trans	3.10	792	52.0	2.66	2.08	1662	2.70	612	63.0
31	Sine	1.10	19.4	09.0	0.51	0.87	21.2	2.00	4.04	81.0

TABLE 5.1 FIR and IIR Compression Ratios (*CR*) and Power Reductions (*P/R*)

SIGNAL	pfree		guitar		voice		transient		sine	
SYSTEM	$\hat{w}$	$eq$	$\hat{w}$	$eq$	$\hat{w}$	$eq$	$\hat{w}$	$eq$	$\hat{w}$	$eq$
$R_{max}$	2.78	4.17	2.78	2.86	14.3	7.14	3.13	3.57	3.28	2.13
$RR$	0.64	0.76	0.64	0.65	0.93	0.86	0.68	0.72	0.69	0.53
$L_{avg}$	1.87	2.26	2.41	2.37	1.47	1.91	1.61	2.13	3.20	3.85
$H$	1.76	1.78	2.36	2.31	0.19	0.29	2.60	2.74	2.07	3.15
$CR$	2.10	0.85	1.60	0.72	7.01	2.01	3.10	2.66	1.10	0.51
Total Bits ( $\times 10^3$ )	11.6	28.4	30.6	65.1	2.20	19.9	13.4	19.6	22.3	28.4

TABLE 5.2 (IIR) Comparison of Coding Parameters for Residual,  $\hat{w}(k)$ , vs.  $eq(k)$

SIGNAL	pfree		guitar		voice		transient		sine	
Parameter	$\hat{w}$	$eq$	$\hat{w}$	$eq$	$\hat{w}$	$eq$	$\hat{w}$	$eq$	$\hat{w}$	$eq$
$R_{max}$	3.57	3.13	4.55	2.04	7.69	3.57	3.13	3.57	2.27	3.23
$RR$	0.72	0.68	0.78	0.51	0.87	0.72	0.68	0.72	0.56	0.69
$L_{avg}$	1.56	2.17	1.53	1.61	1.05	1.21	2.31	3.17	1.89	2.07
$H$	1.46	2.13	2.69	2.96	0.37	0.73	2.14	2.91	1.75	3.01
$CR$	1.54	0.68	1.40	0.65	6.06	2.03	2.70	2.08	2.00	0.87
Total Bits ( $\times 10^3$ )	20.8	23.3	33.2	61.6	5.68	19.7	13.4	17.1	15.3	18.4

TABLE 5.3 (FIR) Comparison of Coding Parameters for Residual,  $\hat{w}(k)$ , vs.  $eq(k)$



## VI. CONCLUSIONS

This thesis set out to compare the performance of two versions of ADPCM data compression systems. The primary objective was to use a set of evaluation criteria to determine, for a given implementation and a given set of algorithms, whether the FIR ADPCM compressor performed better than the IIR ADPCM compressor or vice versa. A Huffman binary coder/decoder was added to the basic ADPCM compressors thus completing an entire data compression system. Therefore, a comparison was also made of each associated binary coder's performance. The specific results of the testing are detailed in the previous chapter. It must be noted that these results are valid only for the specific implementation used in this thesis, which included a FIR filter of order 6 only, and do not necessarily hold for different implementations, filters of different orders, or other changes. As stated in the previous chapter, a FIR filter of order six was experimentally picked on the basis of providing better compression in the ADPCM coder than a sampling of filters using different orders.

Table 5.1 displays the results of the main comparison between IIR and FIR ADPCM compressors. The power reduction, ( $P/R$ ), percentage was used to determine the percentage by which each compressor reduced the original signal power. Clearly, the FIR performed far better than the IIR in this category. This fact was particularly highlighted by the greater power reduction the FIR compressor achieved using the sinusoidal signal; a non-persistently excited signal known to be problematic for the IIR compressor, (Table 5.1, column five -rows eight and fourteen vs. column eleven- and the same rows). Not visible in any of the tables was the fact that the FIR coder implementation compressed each signal approximately 40% faster than did the IIR coder. As dramatic as this difference was, since the IIR coder was not optimized for the fastest possible speed, this result cannot be assumed to always hold for other implementations.



The second major analysis focused on how effectively the modular function performed its job. The purpose of the modular function was to reduce the size of the codepage required to transmit the compressed signal. In order to accomplish this task, the average wordlength,  $L_{avg}$ , was determined; where a smaller  $L_{avg}$  indicated a smaller codepage. The compression ratio, ( $CR$ ), of the binary coder was also calculated to help facilitate the comparison of each modular functions' effectiveness. Thus, in Table 5.2 and Table 5.3, both of which refer only to data taken from the higher SNR set of Table 5.1, both the error sequence,  $eq$ , and the modular sequence,  $\hat{w}$ , were run through the Huffman coder. Table 5.2 shows the results from the IIR coder, where the modular sequence had a smaller  $L_{avg}$  than the error sequence for all signals except the GUITAR signal. The FIR modular function produced a smaller modular sequence  $L_{avg}$  for all five signals (Table 5.3). Furthermore, the  $CR$  of each signal was greater for the modular sequence than for the error sequence. This increase in  $CR$  was a theoretical possibility foretold in Einarsson's work. This result held equally well for the FIR coder as well as the IIR coder. Thus, the addition of the modular function did improve the performance of both ADPCM compression systems as postulated by Einarsson. The compression ratios which are less than one show that when the sequence was coded with the Huffman coder, the codepage required to transmit the sequence actually expanded rather than become compressed. This expansion phenomenon occurs when the input sequence statistics do not match the statistical expectations of the binary coder, resulting in a mismatch which causes the data to expand rather than compress. It is also possible that this statistical mismatch is what sometime caused the  $CR$  to move in the opposite direction from what the  $RR$  would cause one to expect (Tables 5.2 and 5.3).

The limited data set and restricted applicability of the data still make the results somewhat inconclusive and further testing is needed to completely specify each system

capabilities. This inconclusivity is also partly due to the lack of a standard comparison criteria upon which to definitively judge different types of ADPCM systems. Additional testing might concentrate on different types of compression ratios, see Lynch (1985) for more discussion on types of compression ratios, other SNR's and certainly different types of data signals and filter orders, as starting points for further research.



**APPENDIX**

**(MATLAB CODE)**

## A. FIR ADPCM Implementation

```
%-----  
% FINAL OUTPUT IS SEST1  
% LMS of order M=6 using round for Q wrt error, mu values signal dependent  
% This program simulates an ADPCM compression scheme using an fir predictor  
%-----  
clear  
clg  
k=1:2000; %no. of data points  
%load input signal  
mu=X; % step size mu  
M=6; % predictor length  
A=20; % scaling factor used to change SNR  
u=A*[zeros([1,M]),s]; % i/p scaled by A  
u1=round(u); % round input  
N=length(u); % data length  
shat=0; % predictor output  
w=zeros(M); % initial weights  
e=0; % initial error  
eq=zeros(1,M); % predictor input  
  
cw=0;  
minV1=min(u1); % get minimum value of input for H calc.  
maxV1=max(u1); % get max value of input for H calc.  
mod1=maxV1-minV1; % determine mod function value  
c1=zeros(1,mod1);  
  
% implement LMS algorithm  
for n = M+1:N,  
    eq1=eq(n-1:-1:n-M);  
    shat = w(:,n-1)' *eq1; % predictor output  
    e = u1(n) - shat; % error: voice signal  
    eq(n) =round(e); % quantizer output  
    [j,i] = min(abs(Q-eq(n))); % get index of quantizer value  
    c1(i)=c1(i)+1; % update counter  
    w(:,n)=w(:,n-1)+mu*eq1*eq(n); % recalculate weights  
end
```

```

W=max(eq); % get max value of eq to det. mod value
V1=min(eq); % get min value of eq to det. mod value
mod=W-V1+1 % get mod value for error sequence
minV=V1; % store value of V1 in minV variable
V1=-V1; % negate min value of eq to make it positive
%%%%%%%%%%
%
% This section computes the probability vector for the error vector eq
%
%%%%%%%%%%
C1=sum(c1); % total number of data values
Pe=c1./C1; % vector of probabilities
Pe=sort(Pe); % sort prob vector from low to high
pe=fliplr(Pe); % arrange prob vector from high to low
% Initialize Modulus Function
c=zeros(1,mod);
wk=zeros(1,M);
Q1=[0:1:mod-1];
R=zeros(1,mod);
% Modulus Function which takes values from -mod to +mod and reduces the range
% to 0 to +mod to reduce the number of codewords necessary to transmit the
% quantized error signal
for n = M+1:N,
    eqa(n)=V1+eq(n);
    wk(n)=eqa(n)-eqa(n-1);
    [j,i] = min(abs(Q1-wk(n))); % get index of quantizer value
    R(i)=R(i)+1;
    if wk(n) >= 0
        wk1(n)=rem(wk(n),mod); % mod function
        [j,i] = min(abs(Q1-wk1(n))); % get index of quantizer value
        c(i)=c(i)+1;
    else
        wk1(n)=mod+rem(wk(n),mod);
        [j,i] = min(abs(Q1-wk1(n))); % get index of quantizer value
        c(i)=c(i)+1;
    end
end
end

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% This section computes the probability vector for the mod function o/p w'
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

C=sum(c); % total number of data values
P=c./C; % vector of probabilities
P=sort(P); % sort prob vector from low to high
p=fliplr(P); % sort prob vector from high to low

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% This section computes the probability vector for the residuals
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

wc=sum(R); % total number of data values
Pw=R./wc; % vector of probabilities
Pw=sort(Pw); % sort prob vector from low to high
pw=fliplr(Pw);

```

```

% Implement Huffman Encoder/Decoder using function huffman4.m
% which returns wkmod- rcvd residuals, CR-compression ratio,
% Eb -bits in error, and
% bits-total no. of bits used

```

```

Eb1=0;
CR1=0;
Bits=0;
%minV=0; % set to zero when calculating values for w'

```

```

[wkmod,H,L_avg,Bits,E,CR,Lvar,Rmax,eff]=huffman4(eq,mod,pe,minV);

```

### **%Implement FIR Receiver**

**%Initializations**

```
L=length(eq); % data length
wr=zeros(M); % assumed predictor coefficients
shatr=0; % assumed predictor output
sest=zeros(1,M); % assumed rcvr output
wkmod1=zeros(1,M);
```

### **%FIR Receiver LMS algorithm**

```
for n=M+1:N
    wkmod1(n)=rem(wkmod(n)+wkmod1(n-1),mod);
    eqhat(n)=wkmod1(n)-V1;
    eq2=eqhat(n-1:-1:n-M);
    wr(:,n)=wr(:,n-1)+mu*eq2'*eqhat(n);
    shatr=wr(:,n-1)*eq2'; %filter output
    sest(n) = shatr + eqhat(n); % est of noisy signal y
end
```

```
end
sest1=round(sest);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

%

% This section computes signal to noise ratios and input and  
% output power ratios

%

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Pn=1

yo=xcorr(sest,'biased');

yi=xcorr(u,'biased');

ra=length(yi)/2;

ri=round(ra);

rb=length(yo)/2;

ro=round(rb);

Pi=10\*log(yi(ri))

Po=10\*log(yo(ro))

SNRo=Po/Pn

SNRi=Pi/Pn



## B. IIR ADPCM Implementation

```

%-----
% FINAL OUTPUT IS SRHAT1
% transmitter uses backward adaption IIR. Rcvr uses error signal.
% ARMA MODEL MA part is B, AR part is A.
% This program simulates an ADPCM compression scheme using an iir predictor
%-----

clear
clg
%%%%%%%%%%
%%%%%%%%%%
%
% input data section
%
%load guitar.mat;
%d=g_synth(1:7025);
%
%%%%%%%%%%
%%%%%%%%%%

P=length(d);
p=2;           % ar predictor length
q=6;           % ma predictor length
A=10;         % scaling factor

%Initializations
u=A*[zeros([1,q],d)];
N=length(u);  % data length
k=length(u);
uhatA=0;
uhatB=0;
uhat=zeros([1,q+1]);
e=0;          % initial error
u1=round(u);
eq1(1:q)=zeros([1,q]); % predictor input
u2(1:q)=zeros(1,q);

```

```

eq=zeros(1,q);
P=zeros(1,q);
shat=zeros(1,q);
delta1=1-1e-6;
delta2=1-2e-6;
a1=zeros(1,p);           % predictor coefficients
a2=zeros(1,p);
b=zeros(1,q);
alpha1=3.2e-6;
alpha2=2e-6;
f=0;
minV1=min(u1);           % get minimum value of input for H calc.
maxV1=max(u1);           % get maximum value of input for H calc.
mod1=maxV1-minV1;       % get value to use for modular function
Q=(0:mod1-1);
c1=zeros(1,mod1+1);

```

**%IIR Transmitter LMS algorithm**

```

for n = q+1:N,
    e = u1(n) - uhat(n);           % error: voice signal
    eq(n)= round(e);
    [j,i] = min(abs(Q-eq(n)));     % get index of quantizer value
    c1(i)=c1(i)+1;
    eq1 = eq(n-1:-1:n-q);
    uhatB = b(:,n-1)* eq1';
    P(n) = uhatB+eq(n);
    shat(n) = P(n) + uhatA;
    shat1=shat(n:-1:n-1);
    uhatA = [a1(n-1) a2(n-1)]*(shat1)'; %predictor output
    uhat(n+1) = uhatA + uhatB;
    a1(n)=delta1*a1(n-1) + alpha1*sgn(P(n-1))*sgn(P(n-2));
    if abs(a1(n)) <= 0.5,
        f=4*a1(n);
    else
        f=2*sgn(a1(n));
    end
    a2(n)=delta2*a2(n-1)+alpha2*sgn(P(n))*[sgn(P(n-2))-f*sgn(P(n-1))];
    if a2(n) > 0.75,
        a2(n)=0.75;

```

```

elseif a2(n) < -0.75,
    a2(n)=-0.75;
else
    a2(n)=a2(n);
end
if a1(n) > 1-2^(-4)-a2(n),
    a1(n)=1-2^(-4)-a2(n);
elseif a1(n) < -(1-2^(-4)-a2(n));
    a1(n)= -(1-2^(-4)-a2(n));
else
    a1(n)=a1(n);
end
b(:,n)=delta1*b(:,n-1)+alpha2*sgn(eq1)*sgn(eq(n-1));
end
clear a1 a2 b uhatA uhatB P

V=min(eq); % get minimum value of eq to shift with
minV=V;
V1=-1*V; % change min value to positive
W=max(eq); % get max value of eq to det. mod value
mod=W+V1+1 % get mod value
wk=zeros(1,q);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% This section computes the probability vector for the error vector eq
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

C1=sum(c1); % total number of data values
Pe=c1./C1; % vector of probabilities
Pe=sort(Pe); % sort prob vector from low to high
pe=fliplr(Pe);
Q1=[0:1:mod-1];

% Initialize Modulus Function
c=zeros(1,mod);

```

```

w=zeros(1,mod);

% Modulus Function which reduces range toto 0 to +mod to reduce the number of
% codewords necessary to transmit the quantized error signal

for n = q+1:N,
    eqa(n)=V1+eq(n);           % adjust for min of eq
    wk(n)=eqa(n)-eqa(n-1);
    [j,i] = min(abs(Q1-wk(n))); % get index of quantizer value
    w(i)=w(i)+1;
    if wk(n) >= 0
        wk1(n)=rem(wk(n),mod); % mod function
        [j,i] = min(abs(Q1-wk1(n))); % get index of quantizer value
        c(i)=c(i)+1;
    else
        wk1(n)=mod+rem(wk(n),mod);
        [j,i] = min(abs(Q1-wk1(n))); % get index of quantizer value
        c(i)=c(i)+1;
    end
end
clear eqa

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% This section computes the probability vector for the mod function o/p w'
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

C=sum(c);           % total number of data values
P=c./C;            % vector of probabilities
P=sort(P);         % sort prob vector from low to high
p=fliplr(P);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% This section computes the probability vector for the residuals w

```

```
%
%%%%%%%%%%
%%%%%%%%%%
```

```
wc=sum(w); % total number of data values
Pw=w./wc; % vector of probabilities
Pw=sort(Pw); % sort prob vector from low to high
pw=fliplr(Pw);
```

**%Implement IIR Receiver**

**%Initializations**

```
L=length(eq); % data length
wkmod1=zeros(1,q);
x(1:q)=[zeros([1,q])]'; % predictor input
br=zeros(q);
shatr(1:q)=zeros([1,q]); % rcvr pred output
shatrA=0;
shatrB=0;
srhat=zeros([1,q]); % rcvr output
Pr=zeros(1,q);
ar1=zeros(1,q);
ar2=zeros(1,q);
CR1=0;
Bits=0;
Eb1=0;
minV=0;
```

**% Implement Huffman Encoder/Decoder using function huffman4.m**

% which returns wkmod- rcvd residuals, CR-compression ratio,  
 % Eb -bits in error, and  
 % bits-total no. of bits used

```
[wkmod,H,L_avg,Bits,E,CR,Lvar,Rmax,eff]=huffman4(wk1,mod,p,minV);
```

```
for n=q+1:N
```

```
    wkmod1(n)=rem(wkmod(n)+wkmod1(n-1),mod);
    eqhat(n)=wkmod1(n)-V1; % adjust for min of eq
    eq2=eqhat(n-1:-1:n-q);
```

**%IIR Receiver LMS algorithm**

```
Pr(n) = shatrB + eqhat(n);
srhat(n) = Pr(n) + shatrA;
shatrB = br(:,n-1)*(eq2');
shatrA = [ar1(n-1) ar2(n-1)]*(srhat(n-1:-1:n-2));
ar1(n)=delta1*ar1(n-1) + alpha1*sgn(Pr(n-1))*sgn(Pr(n-2));
if abs(ar1(n)) <= 0.5,
    f=4*ar1(n);
else
    f=2*sgn(ar1(n));
end
ar2(n)=delta2*ar2(n-1)+alpha2*sgn(Pr(n))*[sgn(Pr(n-2))-f*sgn(Pr(n-1))];
if ar2(n) > 0.75,
    ar2(n)=0.75;
elseif ar2(n) < -0.75,
    ar2(n)=-0.75;
else
    ar2(n)=ar2(n);
end
if ar1(n) > 1-2^(-4)-ar2(n),
    ar1(n)=1-2^(-4)-ar2(n);
elseif ar1(n) < -(1-2^(-4)-ar2(n));
    ar1(n)= -(1-2^(-4)-ar2(n));
else
    ar1(n)=ar1(n);
end
br(:,n)=delta1*br(:,n-1)+alpha2*sgn(eq2)*sgn(eqhat(n));
end

clear shatrB shatrA Pr ar1 ar2 br
srhat1=round(srhat); % round output to match rounded input
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% This section computes signal to noise ratios and input and
% output power ratios
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
Pn=1
yo=xcorr(srhat,'biased');
yi=xcorr(u,'biased');
ra=length(yi)/2;
ri=round(ra);
rb=length(yo)/2;
ro=round(rb);
Pi=10*log(yi(ri))
Po=10*log(yo(ro))
SNRo=Po/Pn
SNRi=Pi/Pn
```

### C. HUFFMAN Encoding/Decoding Scheme

```
% HUFFMAN4 finds the minimum variance Huffman code for the symbol
% probabilities entered by the user. The algorithm makes use of
% permutation matrices for the combination and sorting of probabilities.
% Permutation matrices are used because they provide a convenient record
% of operations, so that the codewords can then be constructed fairly easily
% once the combination and sorting of probabilities yields just two
% probabilities. At this point a zero is assigned to one of the
% probabilities and a one assigned to the other. The permutation matrices
% are used to append additional zeros and ones as appropriate to obtain
% the final codeword for each symbol. This program both encodes and decodes
% Program revised to accept a vector of data (wk1) and with (mod) different
% values and apply the generated Huffman code (generated as CW and converted
% to decimal- Cwdr) to the data and decode it at the receiver.
% Written by K.L. Frack for EC4580 Course Project% Revised by M. V. Cooperwood
% for data compression thesis
% Last Update: 20 June 1995
% clear Bits out H eff RR Lavg Rmax

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%INPUT THE SYMBOLS TO BE CODED %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%function [wkmod,H,L_avg,Bits,E,CR,Lvar,Rmax,eff]=huffman4(wk1,mod,p,minV);
disp('Commence symbol probability determination process')

total=length(wk1);
% determines the number of characters in the input file.
% This section determines the frequency of occurrence of each unique character
%(As)
% in the input file and associates the frequency with the respective character
%(X).

%nb=input('Input the number of bins (128 or 256) ');
Z=[1:mod];
[As,X]=hist(wk1,Z);

% This section determines calculates the integer length (l) of the respective codeword.
```



```

l=zeros(1,mod); % initialize length vector
for
% INPUT THE NUMBER OF SYMBOLS TO BE CODED. NO TRIVIAL
% SOLUTION ALLOWED.
q=0; % q = number of symbols. Set to 0 to ensure that
% the loop
% will be executed at least once
while q<3 % Need at least 3 symbols for a non-trivial solution
q=mod
%q=input('Enter the number of symbols: '); %allows for keyboard input
if q<3,
beep,
disp('Trivial solution. Use a larger number of symbols. '),
end
end

% ENTER THE SYMBOL PROBABILITIES. For keyboard input only
% Note: The probabilities must sum to 1.00 and must be in entered in
% descending order for the algorithm to work properly. Since the
% algorithm
% will give erroneous results if these errors are overlooked, error
% checking
% routines are included in later steps.
%disp(' ')
%disp('Enter the symbol probabilities ( in descending order).')
%for =1:q,
%p(i)=input([' Enter the probability of s',int2str(i),': ']);
%end
% ENSURE THERE ARE ENOUGH PROBABILITIES ENTERED
% If <RETURN> is inadvertently struck before a probability is entered
% the
% input command could yield a probability vector which is too small.
% This
% causes the program to crash. This procedure prevents this from
% happening
% by setting all of the missing probabilities to zero. In this event the
% user can correct the wrong probabilities in a later step.
%if length(p)<q,
% p=[p;zeros(q-length(p),1)];

```

```

%end
% ERROR CHECK THE SYMBOL PROBABILITIES
correct='n'; %correct = 'n' ensures at least once through the error
%checking
% loop.
count=0;
% count = 0 makes the loop a little simpler. It
%prevents      % the
% program from prompting for a correction until the loop
% has          % been executed at least once.
while correct ~= 'y' % Keep looping until correct.
if count>0; % This procedure will be executed only if there are
            % errors to be corrected.
s=input('Enter the index of the incorrect probability: ');
p(s)=input(['Enter the correct probability for s',int2str(s),': ']);
end
count=1;

```



## LIST OF REFERENCES

- Apiki, S., "Lossless Data Compression," *Byte*, March 1991.
- Bonnet, M., Macchi, O., and Jaidane-Saidane, M., "Theoretical Analysis of the ADPCM CCITT Algorithm," *IEEE Transactions on Communications*, Vol. 38, No. 6, June 1990.
- Bookstein, A. and Storer, J. A., "Data Compression," *Information Processing & Management*, Vol. 28, No. 6, 1992.
- Cappellini, V., *Data Compression and Error Control Techniques with Applications*, New York, NY: Harcourt Brace Janovich, 1985.
- CCITT, Recommendation G.721, *Red Book*, Tome III-3, Malaga-Torremolinos, October 1984.
- Davission L. D., and Gray R. M., *Data Compression*, Stroudsburg, PA: Halsted-Press, 1976.
- Einarsson, G., "An Improved Implementation of Predictive Coding Compression," *IEEE Transactions on Communications*, Vol. 39, No. 2, February 1991.
- Gibson, J. D., "Adaptive Prediction for Speech Encoding," *Acoustic Speech Signal Processing Magazine*, Vol. 21, No. 7, July 1984.
- Haykin, S., *Adaptive Filter Theory*, Englewood Cliffs, NJ, Prentice-Hall, 1991.
- Jayant N. S., and Noll, P., *Digital Coding of Waveforms*, Englewood Cliffs, NJ: Prentice-Hall, 1984.
- Knuth, D. E., "Dynamic Huffman Coding," *Journal of Algorithms*, No. 6, March 1985.
- Langdon, Jr., G. G., "An Introduction to Arithmetic Coding," *IBM J. Res. Develop.*, March 1984.
- Lu, W. W., and Gough, M. P., "A Fast-Adaptive Huffman Coding Algorithm," *IEEE Transactions on Communications*, Vol. 41, No. 4, April 1993.
- Lynch, J., *Data Compression Techniques*, Belmont, CA: Wadsworth, Inc., 1985.

Proakis, J. G., and Manolakis, D. G., *Digital Signal Processing*, New York, NY: Macmillan Publishing Co., 1992.

Rabiner, L. R., and Schafer, R. W., *Digital Processing of Speech Signals*, Englewood Cliffs, NJ: Prentice-Hall, 1978.

Sibul, L. H., *Adaptive Signal Processing*, New York, NY: IEEE, 1987.

Weiss, J., and Schremp, D., "Putting Data On a Diet," *IEEE Spectrum*, Vol. 9, No. 3, August 1993.

Xue, K., and Crissey, J. M., "An Iteratively Interpolative Vector Quantization Algorithm for Image Data Compression," *IEEE Proceedings Data Compression Conference*, Snowbird, UT, 1991.

## INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center 8725 John J. Kingman Rd. STE 0944 Ft. Belvoir, VA 22060-6218	2
2. Dudley Knox Library, Code 13 Naval Postgraduate School Monterey, California 93943-5101	2
3. Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5121	1
4. Professor Monique P. Fargues, Code EC/Fa Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5121	2
5. Professor Ralph Hippenstiel, Code EC/Hi Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5121	1
6. Lt. Michael Vonshay Cooperwood, Sr. 6 Rustic Place Buffalo, New York 14215	1