Second Edition

Coding and Information Theory

RICHARD W. HAMMING

Naval Postgraduate School

PRENTICE-HALL / Englewood Cliffs NJ 07632

Library of Congress Cataloging-in-Publication Data

Hamming, R. W. (Richard Wesley) 1915
Coding and information theory.

Bibliography: p.
Includes index.
1. Coding theory. 2. Information theory.
I. Title.
QA268.H35 1986 519.4 85-17034
ISBN 0-13-139072-4

Editorial/production supervision and
interior design: Nicholas C. Romanelli
Cover design: Frederick Charles Ltd.
Manufacturing buyer: Rhett Conklin

Printed in the United States of America

10 9 8 7 6 5 4

ISBN 0-13-139072-4 01

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*
Whitehall Books Limited, *Wellington, New Zealand*

Contents

Preface to the Second Edition

The second edition is mainly an expansion of the material in the first edition. The changes were made to (1) increase the clarity of the presentation, (2) extend some of the basic ideas, and (3) indicate further practical details. I have resisted the urgings (both from others and myself) to include more difficult codes, such as Reed–Muller, Viterbi, Fire, and cyclic. They seem to be more suitable for a second course, and in any case require a firmer mathematical background than I have assumed, if they are to be "understood." I still believe that it is better, in this subject, to master the fundamentals than it is to rush through, with little real understanding, a large body of material. There are now many excellent texts devoted to algebraic coding theory.

Bostrom Management Corporation is thanked again for excellent cooperation.

R. W. HAMMING

Preface to the First Edition

This book combines the fields of coding and information theory in a natural way. They are both theories about the representation of abstract symbols. The two fields are now each so vast that only the elements can be presented in a short book.

Information theory is usually thought of as "sending information from here to there" (transmission of information), but this is exactly the same as "sending information from now to then" (storage of information). Both situations occur constantly when handling information. Clearly, the encoding of information for efficient storage as well as reliable recovery in the presence of "noise" is essential in computer science.

Since the representation, transmission, and transformation of information are fundamental to many other fields as well as to computer science, it is time to make the theories easily available. Whenever and wherever problems of generation, storage, or processing of information arise, there is a need to know both how to compress the textual material as well as how to protect it against possible mutilation. Of the many known encoding methods, we can indicate only a few of the more important ones, but hopefully the many examples in the text will alert the student to other possibilities.

The text covers the fundamentals of the two fields and gives examples of the use of the ideas in practice. The amount of background mathematics and electrical engineering is kept to a minimum. The book

uses, at most, simple calculus plus a little probability theory, and anything beyond that is developed as needed. Techniques that have recently arisen in computer science are used to simplify the presentation and the proofs of many results. These techniques are explained where they are used, so no special knowledge of computer science is required. Many other proofs have been greatly simplified, and when necessary new material has been developed to meet current technological needs. An effort has been made to arrange the material, especially the proof of Shannon's main result, so that it is evident *why* the theorems are true, not just that they have been proved mathematically.

Chapter 11, on algebraic codes, develops the needed mathematics of finite fields. Because of its mathematical difficulty, this chapter is placed last and out of logical order. It can follow Chapter 3, if desired. There is a deliberate repetition in the text; important ideas are usually presented at least twice to ensure that the reader understands them.

The text leaves out large areas of knowledge in the belief that it is better to master a little than to half know a lot. Thus more material may easily be added (at the discretion of the teacher) when it seems appropriate for the class.

I have followed custom in referring to *Hamming codes* and *Hamming distance;* to do otherwise would mislead the student and be false modesty.

Acknowledgments

It is difficult for the author to recall all his indebtedness to others, since he has known about the material from many years of working at the Bell Laboratories. Teaching a course at the Naval Postgraduate School, based on N. Abramson's elegant, small book, *Information Theory and Coding* (Ref. [A]), rearoused this author's interests in the two fields. Short courses on the topic at other places further developed many of the simplifications, elaborations, and examples, and the help of many students is gratefully acknowledged. The help of Albert Wong is especially appreciated.

In the actual production of the book, thanks are due to Bostrom Management Corporation, especially to Doug Thompson. However, as always, all faults are to be assigned to the author.

R. W. HAMMING

Chapter 1

Introduction

1.1 A Very Abstract Summary

Although the text uses the colorful words "information," "transmission," and "coding," a close examination will reveal that all that is actually assumed is an information source of symbols $s_1, s_2, \ldots, s_q$. At first nothing is said about the symbols themselves, nor of their possible meanings. All that is assumed is that they can be uniquely recognized.

We cannot define what we mean by a symbol. We communicate with each other in symbols, written or spoken (or even gestures), and we may or may not think in terms of symbols, but since we must use symbols to define what we mean by a symbol, we see the circularity of the process. Thus the meaning of a symbol must remain at the intuitive level.

Next we introduce (Sections 4.8 and 6.2) the probabilities $p_1, p_2, \ldots, p_q$ of these symbols occurring. How these p_i are determined is not part of the abstract theory. One way to get estimates of them is to examine past usage of the symbol system and hope that the future is not significantly different from the past. For any discrete probability distribution there is the value of the *entropy function:*

$$H = \sum_{i=1}^{q} p_i \log \frac{1}{p_i}$$

The function H of the probability distribution p_i measures the amount

of uncertainty, surprise, or information that the distribution contains. This function plays a leading role in the theory and provides a lower bound on the average code length. Later we examine more complex probability structures involving the symbols s_i.

The problem of representing the *source alphabet* symbols s_i in terms of another system of symbols (usually the binary system consisting of the two symbols 0 and 1) is the main topic of the book. The two main problems of representation are the following.

1. *Channel encoding:* How to represent the source symbols so that their representations are far apart in some suitable sense. As a result, in spite of small changes (noise) in their representations, the altered symbols can, at the receiving end, be discovered to be wrong and even possibly corrected. This is sometimes called "feed forward" error control.
2. *Source encoding:* How to represent the source symbols in a minimal form for purposes of efficiency. The average code length

$$L = \sum_{i=1}^{q} p_i l_i$$

is minimized where l_i is the length of the representation of the ith symbol s_i. The entropy function provides a lower bound on L.

Thus, in principle, the theory is simply an abstract mathematical theory of the representation of some undefined source symbols in terms of a fixed alphabet (usually the binary system) with the representation having various properties. In this abstract theory there is no transmission through the channel, no storage of information, and no "noise is added to the signal." These are merely colorful words used to motivate the theory. We shall continue to use them, but the reader should not be deceived; ultimately this is merely a theory of the representation of symbols. Clearly, this is a fundamental theory for much of human symbol manipulation.

1.2 History

The beginnings of both coding and information theory go far back in time. Many of the fundamental ideas were understood long before 1948, when information theory was first established on a firm basis. In 1948 Claude E. Shannon published two papers on "A Mathematical Theory of Communication" in the *Bell System Technical Journal* (re-

printed in Ref. [S]). The papers almost immediately popularized the field of information theory, and soon additional papers on information theory appeared in the journals, and courses were taught on the subject in electrical engineering and other departments of many universities.

As in most fields that suddenly open up, many of the early applications were ill advised; but how else are the limitations of a new field to be discovered? As a result of the overexpectations of what information theory could do, disenchantment gradually set in, and a consequent decrease in the number of courses taught. Now, perhaps, a more just evaluation can be made, somewhere between the wild enthusiasm of the first days and the sad disappointment that slowly followed.

Information theory sets bounds on what can be done, but does little to aid in the design of a particular system. The idea that it is therefore useless to know information theory is false, as the following analogy shows. Consider the theory of evolution as taught in biology. Few students will ever apply it directly in their lives, yet it is a valuable constellation of ideas. In spite of this lack of direct application, when the ideas behind

1. Small changes in the species (variations)
2. Survival of the fittest (selection)

are mastered, the ideas can profitably be used in many other situations, often far removed from biology. For example, when looking at a social institution, such as a computer science department, a university, a military organization, a banking system, a government, or even a family relationship, one asks: "How did the present situation arise?" and "What forces selected this particular realization for survival?"

A little more appreciation of the power of the theory suggests the questions: "Given the present forces on the social institution, what are its possible variations (its ability to respond to the forces)?" and "How will it evolve (what will survive)?" Thus the ideas in the theory of evolution can be used in many situations far removed from biology.

When asking "How will it evolve?" it is often worth the effort to consider three similar questions:

What can evolve?
What should evolve?
What will evolve?

With the answers in mind, you are in a position to consider how to bring the third answer more into line with the second—how to improve things,

rather than merely settling for the inevitable that will occur if you do nothing active to alter the world.

Similarly, information theory has ideas that are widely applicable to situations remote from its original inspiration. The applicability of the ideas is often not exact—they are often merely suggestive—but the ideas are still very useful.

At about the time that information theory was created, and in about the same place, coding theory was also created. The basic paper, however, was delayed by patent requirements until April 1950, when it appeared in the *Bell System Technical Journal* (reprinted in Refs. [B1], [B1a], and [T]). In the case of coding theory, the mathematical background was at first less elaborate than that for information theory, and for a long time it received less attention from the theorists. With the passing of time, however, various mathematical tools, such as group theory, the theory of finite fields (Galois theory), projective geometry, and even linear programming have been applied to coding theory. Thus coding theory has now become an active part of mathematical research (Refs. [B1], [B3], [B4], [C], [Gu], [L1], [L2], [MS], [Mc], [P], [W]).

Most bodies of knowledge give errors a secondary role, and recognize their existence only in the later stages of design. Both coding and information theory, however, give a central role to errors (noise) and are therefore of special interest, since in real life noise is everywhere.

Logically speaking, coding theory leads to information theory, and information theory provides bounds on what can be done by suitable encoding of the information. Thus the two theories are intimately related, although in the past they have been developed to a great extent quite separately. One of the main purposes of this book is to show their mutual relationships. For further details on the history of coding theory, see Refs. [B2] and [T]. By treating coding theory first, the meaning of the main results of information theory, and the proofs, are much clearer to the student.

1.3 Model of the Signaling System

The conventional signaling system is modeled by:

1. An information source
2. An encoding of this source
3. A channel over, or through, which the information is sent
4. A noise (error) source that is added to the signal in the channel
5. A decoding and hopefully a recovery of the original information from the contaminated received signal
6. A sink for the information

This model, shown in Figure 1.3-1, is the one that we will use for our signaling system. It has many features of signaling systems now in use.

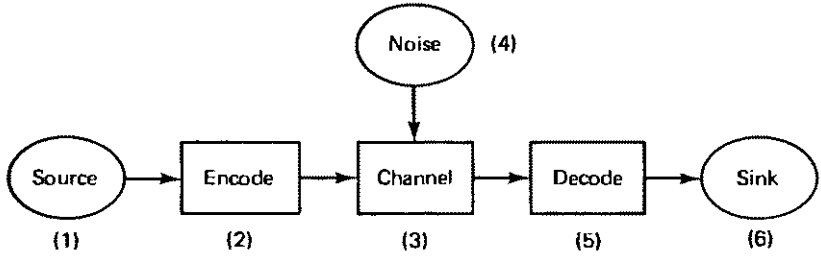

Figure 1.3-1 Standard signaling system

The boxes ENCODE and DECODE should both be thought of as being divided into two parts. In the ENCODE box, we first encode the *source*, taking advantage of any structure in the symbol stream. The more structure there is, the more, typically, we can compress the encoding. These encoded symbols are then further encoded to compensate for any known properties of the *channel*. Typically, this second stage of encoding expands the representation of the message. Thus the ENCODE box first does *source encoding*, followed by *channel encoding*. The DECODE box must, of course, reverse these encodings in the proper order to recover the original symbols.

This separation into two stages is very convenient in practice. The various types of sources are separately encoded into the standard interface between the source and the channel, and then are encoded for the particular channel that is to be used. Thus there is great flexibility in the entire system.

1.4 Information Source

We begin, therefore, with the *information source*. The power of both coding and information theory is to a great extent due to the fact that we *do not define* what information is—we *assume* a source of information, a sequence of symbols in a *source alphabet* $s_1, s_2, \ldots, s_q$, having q symbols. When we get to Chapter 6 we will find that information theory uses the entropy function H as a *measure of information*, and by implication this defines what is meant by "the amount of information." But this is an abstract definition that *agrees only partly* with the commonly accepted ideas concerning information. It is probably this ap-

pearance of treating what we mean by "information" that made information theory so popular in the early days; people did not notice the differences and thought that the theory supplied the proper meaning in all cases. The implied definition does give the proper measure to be used for information in many situations (such as the storage and transmission of data), and people hoped that their situation was one of these (they were often wrong!). Information theory *does not* handle the *meaning* of the information; it treats only the *amount* of information, and even the amount depends on how you view the source.

The source of information may be many things; for example, a book, a printed formal notice, and a company financial report are all information sources in the conventional alphabetic form. The dance, music, and other human activities have given rise to various forms (symbols) for representing their information, and therefore can also be information sources. Mathematical equations are still another information source. The various codes to which we turn next are merely particular ways of representing the information symbols of a source.

Information also exists in continuous forms; indeed, nature usually supplies information to us in that form. But modern practice is to sample the continuous signal at equally spaced intervals of time, and then to digitize (quantize; Ref. [J]) the amount observed. The information is then sent as a stream of digits. Much of the reason for this use of digital samples of the analog signal is that they can be stored, manipulated, and transmitted more reliably than can the analog signal. When the inevitable noise of the transmission system begins to degrade the signal, the digital pulses can be sensed (detected), reshaped, and amplified to standard form *before* relaying them down the system to their final destination. At the destination the digital pulses may, if necessary, be converted back to analog form. Analog signals cannot be so reshaped, and hence the farther the signal is sent and the more it is processed, the more degradation it suffers from small errors.

A second reason that modern systems use digital methods is that large-scale integrated circuits are now very cheap and provide a powerful method for flexibly and reliably processing and transforming digital signals.

Although information theory has a part devoted to analog (continuous) signals, we shall concentrate on digital signals both for simplicity of the theory and because, as noted above, analog signals are of decreasing importance in our technical society. Almost all of our large, powerful computers are now digital, having displaced the earlier analog machines almost completely in information-processing situations. There are almost no large, general-purpose hybrid systems left (1985). Most of our information transmission systems, including the common telephone and hi-fi music systems, are also rapidly going to digital form.

1.5 Encoding a Source Alphabet

It is conventional to represent information (digital signals or more simply symbols) as being in one of two possible states: a switch up or down, on or off, a hole punched or not, a relay closed or open, a transistor conducting or not conducting, a magnetic domain magnetized N-S or S-N, and so on. Currently, devices with two states, called *binary devices*, are much more reliable than are multistate devices. As a result, binary systems dominate all others. Even decimal information-processing systems, such as hand calculators, are usually made from binary parts.

It is customary to use the symbols "0" and "1" as the names of the two states, but any two distinct symbols (marks), such as a circle and a cross, will do. It is often useful to think of the 0 and I as only a pair of arbitrary symbols, not as numbers.

First we consider merely the problem of representing the various symbols of the source alphabet. Given two binary (two-state) devices (digits), we can represent four distinct states (things, symbols):

00
01
10
11

For three binary digits we get $2^3 = 8$ distinct states:

000	100
001	101
010	110
011	111

For a system having k binary digits—usually abbreviated as *bits*—the total number of distinct states is, by elementary combinatorial theory,

$$2^k$$

In general, if we have k different independent devices, the first having n_1 states, the second n_2 states, . . . , the kth having n_k states, then the total number of states is clearly the product

$$n_1 n_2 \cdots n_k$$

For example, if $n_1 = 4$, $n_2 = 2$, and $n_3 = 5$, we can represent $4 \times 2 \times 5 = 40$ distinct items (states). This provides an upper bound on the number of possible representations.

This is the maximum number of source symbols we can represent *if all we consider* is the number of distinct states. In Chapter 4 we discuss how, using variable-length codes, to take advantage of the *probabilities* of the source symbols occurring. For the present we merely consider the number of distinct states—and this is the same as assuming that all the source symbols are equally likely to occur.

We have emphasized two-state devices with two symbols; other multistate devices exist, such as a "dead-center switch," with three states. Some signaling systems also use more than two states, but the theory is most easily presented in terms of two states. We shall therefore generally use a binary system in the text and only occasionally mention systems with r states.

Human beings often attribute meanings to the sequences of 0's and 1's, for example in the ASCII code (Table 1.7-1). The computer (or signaling system), however, merely regards them as sequences of 0's and 1's. In particular, a digital computer is a processor of streams of the two symbols—it is the user (or possibly the input or output equipment) that assigns meanings; the computer merely combines 0's and 1's according to how it is built and how it is programmed. *The logic circuits of a computer are indifferent to the meanings we assign to the symbols; so is a signaling system.*

This is the reason that information theory ignores the meaning of the message, and by so doing it enables us to understand what the equipment does to messages. The theory provides an intellectual tool for understanding the *processing* of information without paying attention to meaning.

Typically, the channel encoding of the message increases the redundancy (to be defined accurately later), as shown in Chapters 2 and 3. The source encoding (mentioned at the end of Section 1.3) usually decreases the redundancy, as shown in Chapters 4 and 5.

We need to think of the source as a random, or stochastic, source of information, and ask how we may encode, transmit, and recover the original information. Specific messages are, of course, actually sent, but the designer of the system has no way of knowing which of the *ensemble* of possible messages will be chosen to be sent. The designer must view the particular message to be sent as a random sample from the population of all possible messages, and must design the system to handle any one of the possible messages. *Thus the theory is essentially statistical*, although we will use only elementary statistics in this book.

Exercises

1.5-1 Compute the number of license plates of the form number number number letter letter letter Ans. $(26)^3(10)^3$

1.5-2 How many instructions can there be in a computer with eight binary digits for each instruction? Ans. $2^8 = 256$

1.5-3 Discuss the biquinary system, which uses alternate two- and five-state devices.

1.6 Some Particular Codes

The binary code is awkward for human beings to use. Apparently, people prefer to make a single discrimination among many things. Evidence for this is the size of the usual alphabets, running from about 16 to 36 different letters (in both upper- and lowercase) as well as the decimal system, with 10 distinct symbols. Thus for human use it is often convenient to group the binary digits, called *bits*, into groups of three at a time, and call them the *octal code* (base 8). This code is given in Table 1.6-1.

TABLE **1.6-1** *Octal Code*

Binary	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

In the octal representation numbers are often enclosed in parentheses with a following subscript 8. For example, the decimal number 25 is written in octal as

$$(31)_8$$

Thus in America, Christmas is Halloween:

$$\text{Dec } 25 = \text{Oct } 31$$

$$(25)_{10} = (31)_8$$

As an example, in Table 1.7-1 of the ASCII code in the next section we have written the octal digits in the left-hand column rather than the

binary digits. The translation from octal to binary is so immediate that there is little trouble in going either way.

Occasionally, the binary digits are grouped in fours to make the *hexadecimal code* (Table 1.6-2). Since computers usually work in *bytes*, which are usually 8 bits each (there are now 9-bit bytes in some computers), the hexadecimal code fits into the machine architecture better than does the octal code, but the octal seems to fit better into human psychology. Thus neither code has a clear victory over the other in practice.

TABLE 1.6-2 *Hexadecimal Code*

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Exercises

1.6-1 Since $2^8 = 3^5$, compare base 2 and base 3 computers.

1.6-2 Make a multiplication table for octal numbers.

1.6-3 From Table 1.6-2, what is the binary representation of D6?

Ans. 11010110

1.7 The ASCII Code

Given an information source, we first consider an encoding of it. The standard ASCII code (Table 1.7-1), which represents alphabetic, numeric, and assorted other symbols, is an example of a code. Basically,

TABLE 1.7-1 *Seven-Bit ASCII Code*

Octal Code	Character	Octal Code	Character	Octal Code	Character	Octal Code	Character
000	NUL	040	SP	100	@	140	`
001	SOH	041	!	101	A	141	a
002	STX	042	"	102	B	142	b
003	ETX	043	#	103	C	143	c
004	EOT	044	$	104	D	144	d
005	ENQ	045	%	105	E	145	e
006	ACK	046	&	106	F	146	f
007	BEL	047	'	107	G	147	g
010	BS	050	(	110	H	150	h
011	HT	051	)	111	I	151	i
012	LF	052	*	112	J	152	j
013	VT	053	+	113	K	153	k
014	FF	054	´	114	L	154	l
015	CR	055	−	115	M	155	m
016	SO	056	·	116	N	156	n
017	SI	057	/	117	O	157	o
020	DLE	060	0	120	P	160	p
021	DC1	061	1	121	Q	161	q
022	DC2	062	2	122	R	162	r
023	DC3	063	3	123	S	163	s
024	DC4	064	4	124	T	164	t
025	NAK	065	5	125	U	165	u
026	SYN	066	6	126	V	166	v
027	ETB	067	7	127	W	167	w
030	CAN	070	8	130	X	170	x
031	EM	071	9	131	Y	171	y
032	SUB	072	:	132	Z	172	z
033	ESC	073	;	133	[	173	{
034	FS	074	<	134	\	174	|
035	GS	075	=	135	]	175	}
036	RS	076	>	136	^	176	~
037	US	077	?	137	—	177	DEL

this code uses 7 binary digits (bits). Since (as noted already) computers work in *bytes*, which are usually blocks of 8 bits, a single ASCII symbol often uses 8 bits. The eighth bit can be set in many ways. It is usually set so that the total number of 1's in the eight positions is an even number (or else an odd number—see Chapter 2). Sometimes it is always set as a 1 so that it can be used as a timing source. Finally, it may be left arbitrary and no use made of it. To convert to the modified ASCII code used by the LT33 8-bit teletype code, use 7-bit ASCII code + $(200)_8$.

The purpose of the even number of 1's in the eight positions is that then any single error, a 0 changed into a 1 or a 1 changed into a 0, will be detected, since after the change there will be, in total, an odd number of 1's in the whole eight positions. Thus we have an error-detecting code that gives some protection against errors. *Perhaps more important*, the code enables the maintenance to be done much more easily and reliably since the presence of errors is determined by the machine itself, and to some extent the source of the errors can actually be located by the code.

We shall frequently use the check of an even (or odd) number of 1's. It is called a *parity check*, since what is checked is only the parity (the evenness or oddness) of the number of 1's in the message. Many computers have a very useful instruction that computes the parity of the contents of the accumulator.

We are now in a better position to understand the ASCII code (Table 1.7-1). The ASCII has a source alphabet of

$$2^7 = 128$$

possible characters (symbols). These characters are represented (encoded) inside a computer in the binary code. An even-parity check can be used to set the eighth position of the 8-bit bytes of the ASCII code. The three printed symbols of Table 1.7-1 are in the octal code. As an example

$$127 = 1 \quad 010 \quad 111$$

(where we have dropped the first 2 bits of the first octal symbol). For an even-parity check this would be 127 = 11 010 111.

Exercises

1.7-1 Write the letters P and p in binary.

1.7-2 To what does 01 010 010 correspond?

1.7-3 Using base 4 code, write out the lowercase ASCII alphabet.

1.7-4 Write out the uppercase ASCII alphabet in base 16.

1.8 Some Other Codes

Another familiar code is the *Morse code*, which was once widely used. Part of the code is given in Table 1.8-1. The dash is supposed to be three times the length of the dot. Although the Morse code may appear

TABLE 1.8-1 *Morse Code*

A	.–
B	–. . .
C	–.–.
D	–. .
E	.
F	. .–.
G	––.
H	. . . .
I	. .
J	.–––
K	–.–
L	.–. .
M	––
N	–.
O	–––
P	.––.
Q	––.–
R	.–.
S	. . .
T	–
U	. .–
V	. . .–
W	.––
X	–. .–
Y	–.––
Z	––. .

to be encoded in a binary code, it is, in fact, a ternary (radix 3, $r = 3$) code, having the symbols dot, dash, and space. The space between dots and dashes in a single letter is 1 unit of time, between letters it is 3 time units, and between words it is 6 time units.

We now make a brief digression to introduce a notation. We will continually need the binomial coefficients

$$C(n, k) = \frac{n!}{k!\,(n - k)!}$$

They count the number of ways a set of k items can be selected from a set of n items. We have adopted the old-fashioned notation $C(n, k)$ because it is easily produced on a typewriter, it is easily set in print, and it can be gracefully handled by most computers. The currently popular notation

$$\binom{n}{k}$$

is difficult for most equipment to handle and is awkward when it appears in running text.

The Morse code is clearly a *variable-length code* that takes advantage of the high frequency of occurrence of some letters, such as "E," by making them short, and the very infrequent letters, such as "J," relatively longer. However, the problems that arise when trying to recognize the words of a variable-length code are great enough in this case to cause almost complete replacement of the Morse code by the van Duuren code, which uses three out of seven positions filled with 1's and the other four with 0's. There are $C(7, 3) = 35$ possible words in this code, and as with the 8-bit ASCII code, the van Duuren code permits the detection of many types of errors since the receiver knows exactly the number of 1's that should be in the received message unit of seven time slots.

Another widely used simple code is the *2-out-of-5 code*. As the name implies, two out of five positions are filled with 1's. In this code there are, very conveniently, exactly $C(5, 2) = 10$ possible symbols. One of the many ways of associating the code symbols with the numerical values of the decimal digits is the *01247 code*, which is a *weighted code* where we attach *weights* 0, 1, 2, 4, and 7 to the successive columns of the code. The corresponding decimal digit is the sum of the weights where the 1's occur, with the sole exception that the combination 4, 7 is taken as 0. The code is given in Table 1.8-2. Again, any single error in a message can be recognized because it will have an odd number of 1's in it.

Exercises

1.8-1 Table 1.8-2 is one assignment of number values to the 10 possible symbols of the 2-out-of-5 code; how many possible 2-out-of-5 codes can there be? Ans. 10!

TABLE 1.8-2 *The 2-Out-Of-5 Code*

0	1	2	4	7	Decimal	Corresponds to:
1	1	0	0	0	1	0 + 1
1	0	1	0	0	2	0 + 2
0	1	1	0	0	3	1 + 2
1	0	0	1	0	4	0 + 4
0	1	0	1	0	5	1 + 4
0	0	1	1	0	6	2 + 4
1	0	0	0	1	7	0 + 7
0	1	0	0	1	8	1 + 7
0	0	1	0	1	9	2 + 7
0	0	0	1	1	0	4 + 7

1.8-2 Write 125 in the 2-out-of-5 code.

Ans. 11000 10100 01010

1.8-3 Of all possible 7-bit odd-parity symbols, how many are not used in the van Duuren code? Ans. 29

1.9 Radix r Codes

As noted earlier, most systems of representing information in computers (and other machines) use two states, although the Morse code is an example of a signaling system with three symbols in the underlying alphabet. The reason for the dominance of two-symbol signaling systems is that two-state devices tend to be more reliable than multistate devices. On the other hand, human beings clearly work better with multistate systems—witness the letters of the alphabet, together with the various punctuation symbols and the decimal digits. Thus it is necessary at times to consider codes with r symbols in their alphabets. For the Morse code, $r = 3$.

The English language alphabet uses 26 letters, both upper- and lowercase, plus miscellaneous punctuation marks. Occasionally, you see a system in which there are exactly the 26 letters, the 10 decimal digits, and a space, a total of 37 symbols. We examine this important code in Section 2.7.

In more abstract notation we assume a source alphabet S consisting of q symbols $s_1, s_2, \ldots, s_q$. These are, in turn, represented in some other symbols, say the binary code. We may, therefore, either think of the ASCII code as having $r = q = 2^7 = 128$ symbols $s_1, s_2, \ldots, s_{128}$, or we may think of each symbol as being a block of 8 binary digits with 1 extra bit appended to the 7 necessary bits.

1.10 Escape Characters

When sending information it is usually necessary to control, from the source end, the remote equipment that is being used. For example, we have to tell the equipment what to do with the information that is being sent. Therefore, it is necessary to have a number of *reserved symbols* to which the remote equipment responds. "End Of Message" is such a symbol; another might be "Carriage Return"; another a "Shift Type Font" from lower- to uppercase or back again; and still another might be "Repeat Last Message." In the ASCII code the entire first column of the table is devoted to special characters.

How can this control be done if we do not wish to restrict the information that can be sent through the system? If, for example, we

were using binary digits as numbers in our message, how do we avoid those combinations of bits that are control symbols for the remote equipment? If by chance our stream of binary numbers happened to contain such a reserved symbol, then the equipment would respond to the reserved symbol when we did not want it to do so. Evidently, some kind of restriction on what can be sent is necessary, but the question is: How can this be done with the least pain to the user of the system?

One way this could be done is to add an extra binary digit to each block of digits, using, say, a 0 if it is message and a 1 if it is a control instruction. As can be seen, this will waste a good deal of channel capacity *if* the control words are rarely used. Furthermore, it does *not* solve the problem of relaying control words *through* one piece of equipment to another piece of equipment.

In one form of FORTRAN, the aspostrophe (') is used to delimit a string of characters. For example, the string of symbols ABC is written

'ABC'

In the FORTRAN processing, the apostrophes are stripped off and the string ABC emerges.

If the string happens to contain an apostrophe, such as the string FINNEGAN'S WAKE, you need to write

'FINNEGAN''S WAKE'

The processing strips off the first of the pair but lets through the one *immediately* following.

To see how to write the underlying program of the FORTRAN compiler, we use a (slightly extended) *state diagram*, which is widely used in finite automata theory. Each state consists of READ NEXT SYMBOL followed by WHAT TO DO. We have three states to consider: (1) the normal state of processing FORTRAN text, (2) the usual inside-the-string state, and (3) the inside-the-string state that looks for an immediately succeeding apostrophe. The state diagram is shown in Figure 1.10-1. If the system has more than one reserved symbol, slight variations on this diagram will handle them.

In the field of logic this problem is known as the *metalanguage problem*: How does one talk about the language itself, especially when one has to use essentially the same language to do the talking? In practice we use voice accents (quote marks) to indicate that we are talking about the language itself, and we leave them out when we are merely talking. There are other formal devices for solving this common problem that arises in computer science and in other fields of information

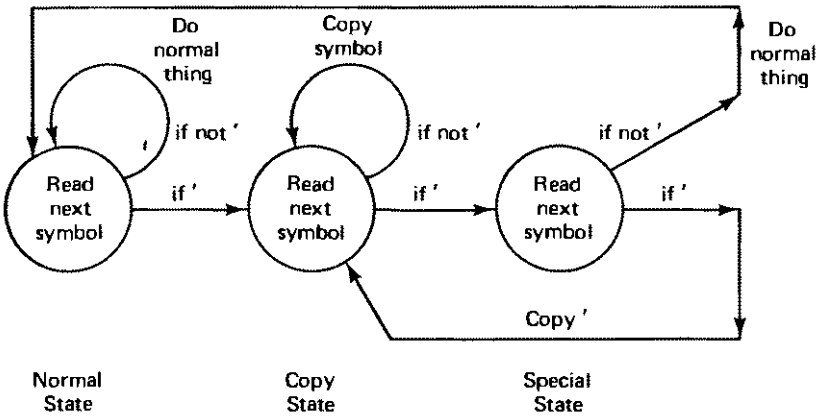

Figure 1.10-1 Reserved symbol automata

handling and processing, and these may also be adapted to this common, vexatious problem.

In controlling a digital computer the failure to distinguish between the various levels of metalanguages is a source of constant confusion. A FORTRAN program and the instruction to COMPILE it are in different languages. A program requires two end marks, one for the metalanguage level of COMPILE to indicate the end of the source program and to stop the compilation process, and one for the FORTRAN program to be used at RUN TIME to end the actual computation.

Exercise

1.10-1 Describe in detail how you would encode 'Help!' to get through the FORTRAN-type system described in the text.

Ans. "Help!"

1.11 Outline of the Course

We have introduced the main topics as well as a number of codes with varying properties, and now have a basis for discussing the material the course is to cover.

In Chapters 2, 3, and 11 we look at ways of channel-encoding information (messages, sources of symbols) so that any errors made in going through a channel, up to a given level, may be detected and/or corrected at the terminal end without recourse to the source. For a

detected but uncorrectable error, we might call for a repeat of the message, hoping to get it correct the next time. In general, the error-detecting and error-correcting capability will be accomplished by adding some digits to the message, thus making the message slightly longer. The main problem is to achieve the required protection against the inevitable errors (noise) in the channel without paying too high a price in adding extra digits. This is encoding for the channel.

In Chapters 4 and 5 we focus on source encoding. The compression of messages is important for efficiency. In transmission through space the shorter message will use the signaling equipment for a shorter time; for storage problems less storage will be needed for the compressed code. This is source encoding. In Chapters 4 and 5 we look at this side of the problem and examine ways to reduce the amount of information being sent. We do this by examining the structure of the messages being sent. When there is a great deal of structure in the information being sent, a good deal of message compression can be achieved, and hence greater efficiency results. Since there are so many properties that messages can have which allow message compression, we can look at only a few of the most common ones. We restrict our attention to general methods and must neglect the many special, trick methods, which are often fairly easy to invent.

In Chapter 6 we introduce the central concept of the *entropy* of a source of information and show how it is connected with the concept of the maximum amount of information that can be sent through a given channel. Thus we get a bit closer to the concept of exactly what information is. The first of Shannon's encoding theorems is for a noiseless channel, and is fairly easy to prove. The second theorem, presented in Chapter 10, discusses sending information through a noisy channel and is much more difficult to prove. Fortunately, the simple *binary symmetric channel* is the realistic case, and for this case the proof is easy to understand. The proof in the more general case is more difficult, and is only sketched.

Shannon's two theorems set bounds on what encoding can accomplish. Unfortunately, the second of his theorems is somewhat nonconstructive and does not tell us how in practice to achieve the bounds indicated. The result is not useless since it indicates where we can expect to achieve large improvements in a signaling system and where we can expect at best only small improvements.

The definition of a channel in information theory is often not a practical one and we need other ways of assessing this central concept of channel capacity for transmitting information. Therefore, Appendix A looks briefly at how, in practice, we measure the channel capacity of a signaling system, especially for continuous signals.

We remind the reader that while we often use the language of "signaling from here to there," all of it is applicable to "signaling from now to then" through some storage medium. In this case it is often the first of Shannon's theorems (the noise-free one) that is important.

Traditionally, the development of both coding and information theory is done in as elegant and general a way as possible—usually highly abstract, with many fancy mathematical symbols, and devoid of practical aspects. This approach is not necessary, and we regularly pause to show how what we are talking about is reflected in common sense and in actual practice, and how it suggests going about the design of future systems. We also try to make the proofs of the results as intuitively obvious as possible rather than merely mathematically elegant. Along the way we introduce many small, practical details that are important when designing a whole system.

But let us be clear about the approach we are adopting. When there are so many different highly evolved and highly involved signaling systems now in use, and so many new ones over the immediate horizon, it is impractical to take them up one at a time. It is necessary to take an overview approach and concentrate on the fundamentals that seem most likely to be relevant for understanding both past and future signaling systems. Indeed, this is the only reasonable approach to any rapidly changing field of knowledge. The approach through special cases simply leaves the student surprised by tomorrow's system. Thus, of necessity, the treatment is, at times, abstract and avoids many of the messy details of current systems.

Chapter 2

Error-Detecting Codes

2.1 Why Error-Detecting Codes?

We first examine channel encoding. The problem is to design codes that will compensate for the inevitable noise that is in all realistic channels. More and more often we require very reliable transmission through the channel, whether it be through space when signaling from here to there (transmission), or through time when signaling from now to then (storage). Experience shows that it is not easy to build equipment that is highly reliable. By "highly reliable" consider how much computing a modern computer does in 1 hour. At one operation per microsecond it does 3.6 billion (10^9) operations in 1 hour (and each operation involves many individual components). By comparison there are less than 3.16 billion seconds in 100 years (more than your probable lifetime). Similarly, reliable transmission systems require very high reliability of their individual components. Reliability in the transmission of words of human languages is one thing; for transmission of computer programs it is something else—hence the importance of detecting errors. Furthermore, as already noted, error detection is a great aid in high-quality maintenance. Without error detection, a large digital system soon becomes unmaintainable.

If repetition is possible, then it is frequently sufficient merely to detect the presence of an error. When an error is detected we simply repeat the message, the operation, or whatever was being done, and with reasonable luck it will be right the second (or even possibly the third) time (see Section 2.5).

It is obviously not possible to detect an error if every possible symbol, or set of symbols, that can be received is a legitimate message. *It is possible to catch errors only if there are some restrictions on what is a proper message.* The problem is to keep these restrictions on the possible messages down to ones that are simple. In practice, "simple" has in the past tended to mean "easily computable," but it could in the future mean "easily looked up." In this chapter we investigate the problem of designing codes such that *at the receiving end*, any single, isolated error can be detected. In Chapter 3 we consider correcting at the receiving end the errors that occur in the message.

2.2 *Simple Parity Checks*

The simplest way of encoding a binary message to make it error detecting is to count the number of 1's in the message, and then append a final binary digit chosen so that the entire message has an even number of 1's in it. The entire message is therefore of even parity. Thus to $(n-1)$ message positions we append an nth parity-check position. At the receiving end the count of the number of 1's is made, and an odd number of 1's in the entire n positions indicates that at least one error has occurred.

Evidently, in this code a double error cannot be detected. Nor can any even number of errors be detected. But any odd number of errors can be detected. If (1) the probability of an error in any one binary position is assumed to be a definite number p, and (2) errors in different positions are assumed to be independent, then for n much less than $1/p$, the probable number of single errors is approximately np. The probability of a double error is approximately $n(n-1)p^2/2$, which is approximately one-half the square of single error. From this it follows that the optimal length of message to be checked depends on both the reliability desired (the chance of a double error going undetected) and the probability of a single error in any one position, p. For more details, see Section 2.4.

Counting the number of 1's and selecting an even number is equivalent to working in a *modulo 2 arithmetic*. "Modulo 2" means that every number is divided by 2 (the modulus) and only the remainder is kept. In this arithmetic (which counts 0, 1, 0, 1, . . .) we count modulo 2 the number of 1's in the $(n-1)$ message positions and then put this sum into the nth position. Thus there are an even number of 1's in the n positions sent. This *even-parity check* is generally used in the theory. In practice, it is occasionally convenient to adopt an *odd-parity check* so that the message of all 0's is not a legitimate message. The changes

necessary in the theory are sufficiently easy to make that we will continue to use even-parity checks throughout the theory.

To actually find the parity of a string of 0's and 1's we can use a two-state finite automaton (Figure 2.2-1). We start in the state 0, and each 1 in the message causes a change in the state. The final state at the end of the message gives the parity count.

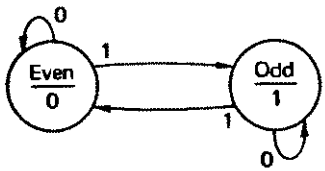

Figure 2.2-1 Parity count circuit

Many computers have an instruction that directly counts the number of 1's in the accumulator register. The right-hand digit of this sum is the required count modulo 2. If this or an equivalent instruction is not available, then *logically adding* (exclusive OR, XOR; see Section 2.8) one half of the message to the other half preserves the parity count in the sum. Repeated use of this observation halves at each step the length of the message and will produce the required sum modulo 2 of all the 1's in the accumulator. Thus about "the first integer greater than or equal to $\log_2 n$" logical additions will be needed.

2.3 *Error-Detecting Codes*

It is common practice to break up a long message in the binary alphabet into runs (blocks) of $(n - 1)$ digits each, and to append one binary digit to each run, making the block that is sent n digits long. The final block may need to be padded out with 0's. This produces the *redundancy* of

$$\frac{n}{n-1} = 1 + \frac{1}{n-1}$$

where the redundancy is defined as the number of binary digits actually used divided by the minimum necessary. The *excess redundancy* is $1/(n - 1)$.

Clearly, for low redundancy we want to use long messages. But for high reliability short messages are better. Thus the choice of the

length n for the blocks of message to send for a given probability of error p is an engineering compromise between two opposing forces.

The 2-out-of-5 code mentioned in Section 1.8 is an example of an error-detecting code with $n = 5$ that uses an even-parity check. The van Duuren 3-out-of-7 code is another such code, but uses odd parity. Neither uses all the symbols it could. A variant is the word count occasionally used in sending telegrams.

Many other examples of this simple kind of encoding for error detection occur in practice. For example, when making a memory transfer to tape, drum, or disk, the *logical sum* of all the words being transferred can be appended as a check word. See Ref. [W] for hardware realizations.

2.4 Independent Errors: White Noise

We have already mentioned (Section 2.2) the model usually assumed for errors in a message: (1) an equal probability p of an error in each position, and (2) an independence of errors in different positions. This is called "white noise" in (a poor) analogy with white light, which is supposed to contain uniformly all the frequencies that are detected by the human eye. The theory for this case is very easy to understand. But in practice there are often reasons for errors to be more common in some positions in the message than in others, and it is often true that errors tend to occur in bursts and not be independent (a common power supply, for example, tends to produce a correlation among errors; so does a nearby lightning strike; see Section 2.6).

It is only fair to note that due to the rapid pace of technological change, it is not possible at the design stage to have much of an idea about the noise patterns of errors that will actually occur, so that white noise is often a reasonable assumption.

The probability of an error in any one position is p; hence the probability of no error is $1 - p$. From the assumed independence of the errors, that is, for white noise, the probability of no error in the n positions is

$$(1 - p)^n$$

The probability of a single error in the n positions is

$$np(1 - p)^{n-1}$$

The probability of k errors is given by the kth term in the binomial expansion

$$1 = [(1 - p) + p]^n = (1 - p)^n + np(1 - p)^{n-1} + \frac{n(n-1)}{2} p^2(1 - p)^{n-2} + \ldots + p^n$$

For example, the probability of exactly two errors is

$$\frac{n(n-1)}{2} p^2(1 - p)^{n-2}$$

We can get a formula for the probability of an even number of errors (0, 2, 4, . . .) by adding the following two binomial expansions and dividing by 2:

$$1 = [(1 - p) + p]^n = \sum_{k=0}^{n} C(n, k)p^k(1 - p)^{n-k}$$

$$[(1 - p) - p]^n = \sum_{k=0}^{n} (-1)^k C(n, k)p^k(1 - p)^{n-k}$$

(The [ ] in the next equation means "greatest integer in.")

$$\frac{1 + (1 - 2p)^n}{2} = \sum_{m=0}^{[n/2]} C(n, 2m)p^{2m}(1 - p)^{n-2m} \qquad (2.4\text{-}1)$$

The probability of an odd number of errors (which will always be detected) is 1 minus this number.

The probability of no errors is the first term ($m = 0$) of the series (2.4-1). Therefore, to get the probability of an undetected error, we drop the first term of (2.4-1), to get

$$\sum_{m=1}^{[n/2]} C(n, 2m)p^{2m}(1 - p)^{n-2m} \qquad (2.4\text{-}2)$$

Usually, only the first few terms are of any importance in evaluating this formula.

Exercises

2.4-1 If $p = 0.001$ and $n = 100$, what is the probability of no error?
Ans. $\exp(-\frac{1}{10})$

2.4-2 If $p = 0.001$ and $n = 100$, what is the probability of an undetected error?

2.4-3 If $p = 0.01$ and you want the probability of an undetected error to be 0.005, what is the maximum length n that you can use?
Ans. $n = 10$ but almost $n = 11$

2.4-4 For small p and arbitrary n, do Exercise 2.4-1. Ans. $\exp(-np)$

2.5 Retransmission of Message

When an error is detected, it is often possible to ask for a retransmission, a recomputation, or repeat of the process being done. For example, in the Bell Relay Computers the code used for representing the decimal digits was a 2-out-of-5 code, and whenever an error was detected, a second, and even a third trial was made. On the Model 6 Relay Computer, if after the third trial the message was still not acceptable, then the whole problem was dropped and the next one taken up, in the hope that the defective part would not be used in the new problem.

When reading magnetic tapes it is the custom to use at least an error-detecting code, and to call for repeated readings of the tape if the parity checks are violated. How many repetitions to use depends on your model for the error. If you think that the error is due to a slight loss of magnetization, then with luck a subsequent trial will read the tape correctly; but if you think it is due to a more permanent failure, then repetitions will probably occur until another, independent error occurs, so that then the parity check is met. You will therefore get a message with two errors in it, rather than the right message! Thus the strategy of "detect an error, and if one is found, call for one or more repetitions" is sound *only if* the type of error you expect is transient.

Parity checks have long been used in computers, in both hardware and software (Ref. [W]). For example, in the early days of unreliable drum storage, every WRITE on the drum was followed by the logical sum of all the registers that were used for the message being stored on the drum. This check sum was then stored in a final register of the block on the drum. The drum was next read back and checked to see if the message was written properly. Only then was the information being stored released from the machine's storage registers. When later the drum was read, the parity check (the logical sum of *all* the registers being stored) was again computed to see if the sum was identically zero. If not, retrials were made to read it again. This method depends on the fact that $x + x = 0$ for logical addition.

2.6 Simple Burst Error-Detecting Codes

Noise (errors) often occur in bursts rather than in isolated positions in the received message. Lightning strikes, power-supply fluctuations, and loose flakes on a magnetic surface are all typical causes of a burst of noise.

Suppose that from measurements in the field we agree on the maximum length L of any burst we are to detect. For ease of discussion (since the changes necessary to handle other cases are easy to invent) assume that the burst length L is the accumulator word length of the computer. We have only to select the appropriate error-detecting code (or error-correcting code of Chapter 3) and instead of computing parity checks over the bit positions, we compute parity word checks over the corresponding word positions. In effect, we work in words, not bits, and have L independent (interleaved) codes, one over each bit position in the word (see Section 3.8). If a burst covers the end of one word and the beginning of another, still no two errors will be in the same code since we *assumed* that any burst length k was $(0 \leq k \leq L)$.

Thus we can send messages through noise that is "bursty" provided that we recognize the noise pattern and design for it. For reliable hardware, see Ref. [W].

Example. If the message is

Fall 1980

this can be encoded in ASCII in a burst code as follows (no parity check is used here):

F = 106 =	01	000	110	
a = 141 =	01	100	001	
1 = 154 =	01	101	100	
1 = 154 =	01	101	100	
sp = 040 =	00	100	000	
1 = 061 =	00	110	001	
9 = 071 =	00	111	001	
8 = 070 =	00	111	000	
0 = 060 =	00	110	000	
Check sum =	00	000	111	= BEL

The encoded message is therefore

Fall 1980BEL

where BEL is the single ASCII symbol 00 000 111.

Exercise

2.6-1 Given the message

101 01010
011 00110
000 11110
110 00110
111 10101

add the appropriate check to form a burst error-detecting code of word length 1 byte. Ans. 111 00001

2.7 *Alphabet Plus Number Codes: Weighted Codes*

The codes we have discussed so far have generally assumed a simple form of white noise. This is very suitable for many types of machines, although in serial transmission the loss of a symbol (or the insertion of an extra one) is a common error in some systems, and is not caught by such codes, hence causes a loss of synchronization.

When dealing with human beings, another type of noise is more appropriate. People have a tendency to interchange adjacent digits of numbers; for example, 67 becomes 76. A second common error is to double the wrong one of a triple of digits, two adjacent ones of which are the same; for example, 667 becomes 677, merely a change of one digit. These are the two most common human errors in arithmetic. In a combined alphabet/number system, the confusion of "oh" and "zero" is very common.

A rather frequent situation is to have an alphabet, plus space, plus the 10 decimal digits as the complete set of symbols to be used. This amounts to $26 + 1 + 10 = 37$ symbols in the sending alphabet. Fortunately, 37 is a prime number and we can use the following method for error checking. We *weight* the symbols with weights 1, 2, 3, . . . beginning with the check digit of the message. We reduce the sum modulo 37 (divide by 37 and take the remainder) so that a check symbol can be selected that will make the sum 0 modulo 37. Note that a "blank" at the end, as a check symbol, is *not* the same as nothing.

It is easy to compute that an interchange of adjacent digits will be detected, and that doubling the wrong digit (which is the changing of a single symbol) will also be detected. If the interchanged digits are the kth and $(k + 1)$st, their original sum,

$$ks_k + (k + 1)s_{k+1}$$

becomes

$$(k + 1)s_k + ks_{k+1}$$

and the change

$$ks_k + (k + 1)s_{k+1} - (k + 1)s_k - ks_{k+1} = s_{k+1} - s_k$$

cannot be divided by 37 with a remainder 0 unless $s_k = s_{k+1}$. A similar argument applies to the interchange of any two digits, adjacent or not (assuming a message length less than 37). Many other interchanges can also be caught. The change of "oh" to "zero" is the change of a single symbol. Such a code is very useful when human beings are involved in the process. Notice that the length of the message being encoded is not fixed, and can exceed 37 symbols if necessary (although special precautions are then required). This type of encoding can be used on credit cards, the names of some kinds of items in inventory, and so on. The probability that a random encoding will get through the input check is $\frac{1}{37}$. Thus by using this simple parity check, forgery attempts will be caught about 97% of the time.

When the check symbol is the "space," this can sometimes be awkward. If it is awkward, then frequently the whole label, say an inventory name, can be dropped and a new label chosen in its place (a loss of about $\frac{1}{37}$ of the possible inventory names results).

To find the weighted check sum easily, note that if you compute the running sum of a set of n numbers, and then sum these again, you will have the first number entered into the final sum n times, the next number $n - 1$ times, the next $n - 2$ times, and so on, down to the last number, which will get into the final total only once. Thus you have the required weighted sum of the numbers that correspond to the symbols of the alphabet being used.

If this "summing the sum" seems mysterious, consider the message $w\ x\ y\ z$:

Message	Sum	Sum of sum
w	w	w
x	$w + x$	$2w + x$
y	$w + x + y$	$3w + 2x + y$
z	$w + x + y + z$	$4w + 3x + 2y + z$

where we see the weighted sum emerging. This is sometimes known as "progressive digiting."

Example. If 0 = 0, 1 = 1, 2 = 2, . . . , 9 = 9, A = 10, B = 11, . . . , Z = 35, b1 = 36, then encode

A6 7

We proceed as follows:

	Sum	Sum of Sum
A = 10	10	10
6 = 6	16	26
b1 = 36	52	78
7 = 7	59	137
$x = x$	$59 + x$	$196 + x$

$$37\overline{)196 + x}\ \text{with quotient } 5,\ \text{remainder } 11 + x$$

Since $11 + x$ must be divisible by 37, it follows that

$$x = 37 - 11 = 26 = \text{Q}$$

The encoded message is therefore

A6 7Q

To check at the receiver that this is a legitimate encoded message, we proceed as follows:

A	$10 \times 5 =$	50
6	$6 \times 4 =$	24
b1	$36 \times 3 =$	108
7	$7 \times 2 =$	14
Q	$26 \times 1 =$	26
	Sum	$= 222 = 37 \times 6 \equiv 0$ modulo 37

The method of encoding used in this section is an example of encoding for *human beings* rather than for the conventional *white noise*. We see how dependent the encoding method chosen is on the assumed noise. It is necessary to emphasize this point because the theory so often tacitly assumes the presence of white noise that the reader is apt to forget this basic assumption when involved with an elegant, complex code.

Exercises

2.7-1 If 0 = 0, 1 = 1, . . . , 9 = 9, A = 10, B = 11, . . . , Z = 35, b1 = 36, encode

B23F mod 37

Ans. B23F9

2.7-2 In the same code as that of Example 2.7-1, is

K9K9

a correct message?

2.8 Review of Modular Arithmetic

Because we are often going to use parity checks, we need to get a firm grasp on the arithmetic of the corresponding manipulations. We have seen that mod 2 addition ("mod" is an abbreviation for "modulo") is the arithmetic that the simple binary parity checks use and is the same as logical addition (exclusive OR; XOR). The rules for this form of addition are

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 0$$

There are no numbers other than 0 and 1 in the system. If we choose to work in normal arithmetic, then we merely divide the result by 2 and take the remainder. When we later come to the algebra of linear equations and polynomials, where the coefficients are from the mod 2 number system, we will have the same table for addition. For multiplication we have the rules

$$0 \times 0 = 0$$
$$0 \times 1 = 0$$
$$1 \times 0 = 0$$
$$1 \times 1 = 1$$

Thus multiplication is the logical AND of computing.

Occasionally, we will work modulo some other number than 2. For example, in the preceding section we used numbers modulo 37. Generally, the theory in any prime base p (such as 37) is very much like that of base 2, and we need not go into the details here. You have only to read the preceding paragraph and make simple changes to understand the corresponding arithmetic and algebra. For addition and subtraction mod p, we divide every number by p and take the positive remainder.

For multiplication mod m (not a prime) we must be more careful. Suppose that we have the numbers a and b congruent to a' and b' modulo the modulus m. This means that

$$a \equiv a' \mod m$$

$$b \equiv b' \mod m$$

or

$$a = a' + k_1 m$$

$$b = b' + k_2 m$$

for some integers k_1 and k_2. For the product ab we have

$$ab = a'b' + a'k_2 m + b'k_1 m + k_1 k_2 m^2$$

$$ab \equiv a'b' \mod m$$

Now consider the particular case

$$a = 15, \qquad b = 12, \qquad m = 10$$

We have

$$a' = 5, \qquad b' = 2$$

and

$$ab \equiv a'b' \equiv 0 \mod 10$$

But neither a nor b is congruent to zero! *Only for a prime modulus do we have the important property that if a product is zero, then at least one factor is zero.* Hence the importance of a prime modulus. Now we see why 37 was so convenient a number in Section 2.7.

Modular arithmetic should be clearly understood, especially the need for a prime modulus, because in Chapter 11 we will face the problem of constructing a corresponding modular algebra. In the next section, we therefore give yet another example of this type of encoding.

2.9 ISBN Book Numbers

The International Standard Book Number (ISBN) now appears on most textbooks. It is (usually) a 10-digit code that publishers assign to their books. A typical book will have

0-1321-2571-4

although the hyphens may appear in different positions. (The hyphens are of no importance.) The 0 is for United States and some other English-speaking countries. The 13 is for Prentice-Hall, the publisher. The next six digits, 21-2571, are the book number assigned by the publisher, and the final digit is the weighted check sum, as in Section 2.6. Modulo 10 will not work, since 10 is a composite number. Thus they use the check sum modulo 11, and are forced to allow an X if the required check digit is 10. As before, this could be avoided by giving up $\frac{1}{11}$ of the possible book names.

Searching around, I found the ISBN

0-1315-2447-X

To check that this number is a proper ISBN, we proceed as follows:

$$
\begin{array}{rrrl}
 & 0 & & \\
 & 1 & 1 & 1 \\
 & 3 & 4 & 5 \\
 & 1 & 5 & 10 \\
 & 5 & 10 & 20 \\
 & 2 & 12 & 32 \\
 & 4 & 16 & 48 \\
 & 4 & 20 & 68 \\
 & 7 & 27 & 95 \\
X = & 10 & 37 & 132 = (11) \times (12) = 0 \mod 11
\end{array}
$$

It checks!

Here again we see a simple error-detecting code designed for human use rather than for computer use. Evidently, such codes have wide applicability.

Exercises

2.9-1 Check the ISBN 0-13165332-6.

2.9-2 Check the ISBN 0-1391-4101-4.

2.9-3 Consider the ISBN 07-028761-4. Does it make sense?

2.9-4 Check the ISBNs in your current textbooks.

Chapter 3

Error-Correcting Codes

3.1 Need for Error Correction

For channel encoding, simple error detection with repetition is often not enough. The immediate question is: If a computer can find out that there is an error, why can it not find out where it is? The answer is that by proper encoding this can, indeed, be done. For example, we can write out, or do, each thing three times, and then take a vote. But as we shall see, much more efficient methods are available. As an example of the use of an error-correcting code, when information is sent from space probes to Earth, the signaling time is so long that by the time an error can be discovered, the source may long since have been erased. The codes currently used are very sophisticated and can correct many simultaneous errors in a block.

Error correction is also very useful in the typical storage (memory) system. Many organizations, for example, store much of their information on magnetic tapes. Over the years the quality of the recording gradually deteriorates, and the time comes when some of it cannot be read reliably. When this happens, the original source of the information is often no longer available and therefore the information stored on the tape is forever lost. If some error-correction ability is incorporated into the encoding of the information, the corresponding types of errors can be corrected.

Error correction has long been used and can be incorporated into the hardware. For example, the old NORC computer had a simple type of error correction for isolated errors that occurred on its cathode-

ray-tube storage device. Often, it is only the storage device that uses the error correction. The STRETCH computer that IBM built "to *stretch* the state of the art" had single-error correction and double-error detection (Section 3.7) throughout much of the computer. On the STRETCH acceptance test it has been claimed that an error in a circuit developed in the first few minutes and that the error-correcting circuits fixed up the errors for the rest of the hour. Currently, most high-density magnetic recording uses some form of error correction, as does hi-fi digital recording of music. As the components get smaller, solid-state memories generally go to error-correcting codes. See Ref. [W] for hardware checking.

The error correction capability can be in the software as well as in hardware. We mentioned earlier that error-detection programs for drum storage were in the software. Error correction could also be used there. In many other places, correcting codes are built into the software. The software approach has the advantage that the error correction can more easily be put on the important parts of the information and omitted from the less important parts. Furthermore, only as experience reveals the weaknesses of the computer system is it known where error protection is most needed as well as how much is needed. Only after simulation of error protection in software has proved that it handles the problem should it be added to the system in the form of hardware. However, error correction in software will clearly use a lot of computer time.

3.2 Rectangular Codes

The first, and simplest, error-correcting codes are the *triplication codes*, where every message is repeated three times and at the receiving end a majority vote is taken. It is obvious that this system will correct isolated single errors. However, it is also obvious that this encoding system is very inefficient. Three computers in parallel with the associated "compare and interrupt if necessary circuits" is expensive. Therefore, we look for better methods of encoding information so that a single error can be corrected.

The next simplest error-correcting codes are the *rectangular codes*, where the information is arranged, in principle, although not necessarily in fact, in the form of an $m - 1$ by $n - 1$ rectangle (Figure 3.2-1). A parity-check bit is then added to each row of $m - 1$ bits to make a total of m bits. Similarly, a check bit is added for each column. Whether or not the final corner bit is used has little importance; for even-parity checking, the parity sum of its row and its column are the same. Thus

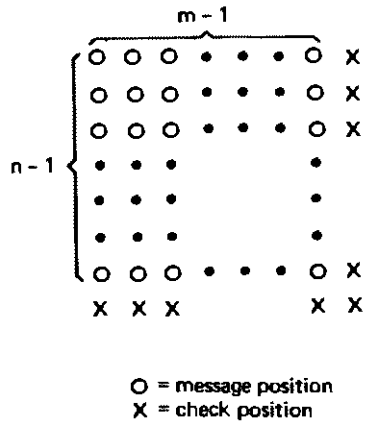

Figure 3-2.1 Rectangular codes

the original code of $(m - 1)(n - 1)$ bits of the rectangle becomes an array of mn bits. The redundancy is therefore $1 + 1/(m - 1) + 1/(n - 1) + 1/(m - 1)(n - 1)$. For a given size mn, the redundancy will be smaller the more the rectangle approaches a square. For square codes of side n, we have $(n - 1)^2$ bits of information and $2n - 1$ bits of checking along the sides.

A rectangular code, for example, was the code that the NORC computer used for its cathode-ray storage tubes. Rectangular codes often are used on tapes (Refs. [B2], [W]). The longitudinal parity checks are on the individual lines, and an extra line is put at the end of the block to form the vertical parity check. Unfortunately, the machine designer rarely lets the user get at the parity-check information so that by using suitable software programs the isolated error could be corrected.

Exercises

3.2-1 Discuss various possible rectangular codes for 24 message bits.

3.2-2 Discuss the use of the corner check bit. Prove that for even-parity checks it checks *both* row and column. Discuss the use of odd-parity checks in this case.

3.2-3 Compare an n^2 code with an $(n - 1)(n + 1)$ code.

3.3 Triangular, Cubic, and n-Dimensional Codes

Thinking about rectangular codes soon suggests a *triangular code*, where each element on the diagonal (and there are only n of them if the entire triangle has n bits on a side) is set by a parity check covering *both* its own row and its own column (Figure 3.3-1). Thus for a given size we

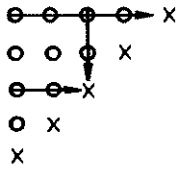

Figure 3-3.1 Triangular codes

lower the redundancy we use. The triangular array of bits of information has $n(n-1)/2$ bits with n bits added for the checks. Thus the redundancy is $1 + 2/(n-1)$.

A triangular code of side n has

$$\frac{n(n-1)}{2} \text{ message and } n \text{ check bits}$$

When we compare a triangular code with a square code for a given number of parity check-bits we find that the triangular code has more message bits. We have included the cubic code in Table 3.3-1.

TABLE 3.3-1 *Comparison of Three Codes*

	Square		Triangular		Cubic	
n	Message $(n-1)^2$	Check $2n-1$	Message $n(n-1)/2$	Check n	Message n^3-3n+2	Check $3n-2$
2	1	3	1	2	4	4
3	4	5	3	3	20	7
4	9	7	6	4	54	10
5	16	9	10	5	112	13
6	25	11	15	6	200	16
7	36	13	21	7	324	19
8	49	15	28	8	490	22
9	64	17	36	9	704	25
10	81	19	45	10	972	28

Once we have found a better code than the simple rectangular ones, we are led to ask the question: What is the best that we can do with respect to keeping down the redundancy? For a cube we can check each plane in three dimensions (Figure 3.3-2) by a single bit on the

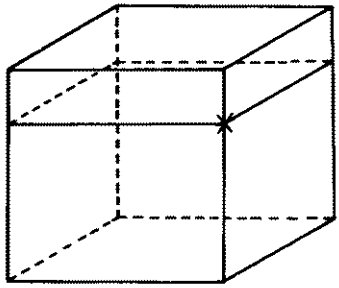

Figure 3-3.2 Cubic code, showing only one plane and its check position

corresponding edge. The three intersecting edges of check bits use $3n - 2$ positions out of the n^3 total positions sent. The excess redundancy is roughly $3/n^2$ extra bits.

If the use of three dimensions is better than two dimensions, then would not going to still higher dimensions be better? Of course, we do not mean to arrange the bits in three-dimensional or higher space; we only imagine them so arranged to calculate the proper parity checks. A little thought about checking over the three-dimensional planes in four-dimensional space will lead to (approximately) an excess redundancy of $4/n^3$. We soon come to the conclusion that the highest possible dimensional space might be the best, and we want a total of 2^n bits of which $(n + 1)$ are checks (one check in each direction plus the common corner bit). There would be, therefore, $(n + 1)$ parity checks. If they were suitably arranged in some order, then they would give a number called the *syndrome*, which we would get by writing a 1 for each failure of a parity check, or a 0 when it is satisfied. These $(n + 1)$ bits form an $(n + 1)$-bit number, and this number (the syndrome) can specify any of 2^{n+1} things—more than the 2^n locations of the corresponding single error *plus* the fact that no error occurred. Notice that the check bits, as in the simple error-detecting codes, are *equally protected* with the message bits. All the bits sent enter equally—none is treated specially.

However, we see that there are almost twice as many states in the syndrome as are needed to indicate when the entire message is correct plus the position of the error if one occurs. Thus there is almost a

factor of 2 in the loss of efficiency. Nevertheless, this suggests a new approach to the problem of designing an error-correcting code, which we take up in the next section.

Exercises

3.3-1 Make a triangular code of 10 message positions.

3.3-2 Discuss in detail the four-dimensional $2 \times 2 \times 2 \times 2$ code.

3.3-3 Compare the three codes of Table 3.3-1 that use approximately 10 check bits.

3.4 Hamming Error-Correcting Codes

In this section we adopt an algebraic approach to the problem that was started in the preceding section—finding the *best* encoding scheme for single-error correction *for white noise*. Suppose that we have m independent parity checks. By "independent" we mean, of course, that no sum of any combination of the checks is any other check—remember that they are to be added in our modulo 2 number system! Thus the three parity checks over positions

Check 1:	1,	2,	5,	7			
2:			5,	7,	8,	9	
3:	1,	2,			8,	9	

are dependent checks since the sum of any two rows is the third. The third parity check provides no new information over that of the first two, and is simply wasted effort.

The *syndrome* which results from writing a 0 for each of the m parity checks that is correct and a 1 for each failure can be viewed as an m-bit number and can represent at most 2^m things. We need to represent the state of all the message positions being correct, plus the location of any single error in the n bits of the message. Thus we must have the inequality

$$2^m \geq n + 1 \tag{3.4-1}$$

Can such a code be found? We will show how to build ones that exactly meet the equality condition. They represent a particular so-

lution to the problem of designing a suitable code and are known as *Hamming codes.*

The simple, underlying idea of the Hamming code is that the syndrome shall give the actual position (location) of the error, with all 0's in the syndrome meaning no error. What positions should the parity checks use? Let us look at the binary representations of the position numbers (Table 3.4-1). Evidently, if the syndrome is to indicate the position of an error when it occurs, every position that has a 1 in the last position of its binary representation must be in the first parity check (see the extreme right-hand column of the table). *Think carefully why this must be true.* Similarly, the second parity check must be tripped by those positions that have a 1 in the second-lowest position in its binary representation; and so on.

TABLE 3.4-1 *Check Positions*

Position	Binary Representation
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
.	
.	
.	

Thus we see that the first parity check covers positions 1, 3, 5, 7, 9, 11, 13, 15, . . . ; the second, 2, 3, 6, 7, 10, 11, 14, 15, . . . ; the third, 4, 5, 6, 7, 12, 13, 14, 15, . . . ; the next, 8, 9, 10, 11, 12, 13, 14, 15, 24, 25, . . . ; and so on.

To illustrate what we have just said (see Table 3.4-2), let us design a simple error-correcting code for four binary digits. We must have $2^m \geq n + 1$ and we easily see that $2^3 \geq (4 + 3) + 1 = 8$. Therefore, we will need to have $m = 3$ parity checks, and this gives $7 = n =$ total of message plus parity-check positions. The positions to use to set the parity checks will for convenience be picked as 1, 2, and 4. Thus the information message positions are 3, 5, 6, and 7.

TABLE 3.4-2 *Encoding a 4-Bit Message and Locating Error*

Encode

1	2	3	4	5	6	7	Position
–	–	1	–	0	1	1	Message
0	1	1	0	0	1	1	Encode
		×					Error
0	1	0	0	0	1	1	Receive

Locate error

Check 1	1	3	5	7	
	0	0	0	1	Fails ⟶ 1
Check 2	2	3	6	7	
	1	0	1	1	Fails ⟶ 1
Check 3	4	5	6	7	
	0	0	1	1	Correct ⟶ 0
Syndrome =	0	1	1	=	3 ⟶ position of error

Correct (add 1 into the error position)

		1					Correct error
0	1	1	0	0	1	1	Corrected message

To encode the information message, I write the message in the positions 3, 5, 6, 7 and compute the parity checks in positions, 1, 2, 4. Let this message be, say, __1_011; the spaces are where the parity checks are to go. The first parity check, which goes in position 1, is computed over positions 1, 3, 5, 7, and looking at the message I see that position 1 gets a 0. I now have 0_1_011. The second parity check, over positions 2, 3, 6, 7, sets position 2 as a 1. I then have 011_011 as the partially encoded message. The third parity check is over positions 4, 5, 6, 7, and looking at what I have, I see that position 4 must get a 0. Thus the final encoded message is 0110011.

To see how the code corrects an isolated error, suppose that when the message 0110011 is sent, the channel removes the 1 in the third position from the left. The corrupted message is then 0100011. For the receiving end, you apply the parity checks in order. The first, over positions 1, 3, 5, 7, evidently fails, so the lowest-order digit of the syndrome is a 1. The second parity check, over 2, 3, 6, 7, fails, so the second digit of the syndrome is 1. The third parity check, over 4, 5, 6, 7, succeeds, so the highest-order digit is a 0. Looking at the syndrome as a binary number, you have the decimal number 3. Thus you change (logically add 1) the symbol in position 3 from the received 0 to a 1. The resulting sequence of 0's and 1's is now correct, and when you strip

off the checking bits 1, 2, and 4, you have the *original information message 1011 in positions 3, 5, 6, and 7.*

Note that the check positions are equally corrected with the message positions. The code is *uniform in its protection; once encoded there is no difference between the message and the check digits.* The cute part of the Hamming code is the numerical ease of both the encoding and the code correction based on the syndrome at the received end. Also, note that the syndrome indicates the position of the error *regardless* of the message being sent; logically adding a 1 to the bit in the position given by the syndrome corrects the received message, where, of course, the syndrome "all 0's" means that the entire message has no errors.

The redundancy in the example of four message bits with three check bits seems high. But if we took 10 check bits, we have, from equation (3.4-1), $2^{10} \leq 10 +$ message positions $+ 1$, or $1024 - 11 = 1013 \geq$ message positions. This shows that the excess redundancy rises like $\log_2$ of the number of message positions.

Exercises

3.4-1 Discuss the Hamming code using four checks.

3.4-2 Discuss the Hamming code for two checks. Ans. Triplicate code

3.4-3 Show that 1111111 is a correct message.

3.4-4 Correct 1001111 and decode.

3.4-5 Locate the error in 011100010111110, where we have used four checks as in Exercise 3.4-1. What is the correct message?

3.4-6 What is the probability that a random sequence of $2^m - 1$ 0's and 1's will be a code word? Ans. $1/2^m$

3.5 Equivalent Codes

The example above is one way of encoding a message. There are many other *equivalent codes*. It should be obvious that any interchange of the positions of the code will leave us with a code that is only trivially different. Similarly, if we were to *complement* (change 0's to 1's and 1's to 0's) all the digits appearing in a certain position, then again we would have a trivially different code.

We can, if we wish, use this observation about the interchange of positions to move all the check bits to the end of the message (see Figure 3.5-1). For the arrangement we have the permutation

$$
\begin{aligned}
1 &\longrightarrow 5\\
2 &\longrightarrow 6\\
3 &\longrightarrow 1\\
4 &\longrightarrow 7\\
5 &\longrightarrow 2\\
6 &\longrightarrow 3\\
7 &\longrightarrow 4
\end{aligned}
$$

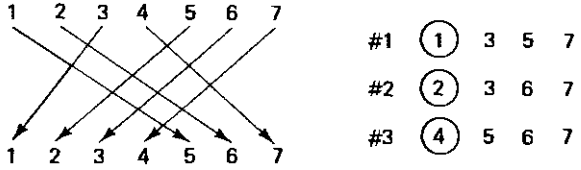

Figure 3-5.1 Hamming code positions

Of course, the syndrome that emerges must also be changed by the same table. This permutation makes the "masking" of the message to get the parity checks, and to make the correction, less simple in appearance, but it is exactly the same amount of work by the computer, since for each check bit we still mask the received message to get the check bit by the "parity sum." Where we place the check bit is also changed. Although the meaning of the syndrome has its position changed, the syndrome is still essentially the same and refers to a unique place. The position to logically add the correcting bit is therefore changed, but that can be found either by a formula or by a table lookup, which enters the table with the syndrome, and the entry in the table is the word with a 1 in the position to be corrected. Of course, for a long code a message might be two or more words long. The table size is proportional to n. The original Hamming code used computing power rather than table lookup, and, in general, for very long codes this is necessary.

One advantage of putting the check bits at the trailing end rather than distributed throughout the message is that sorting on the encoded message will give the same result as on the original messages.

As an alternative to decoding a message, you could, in these days of cheap, large memories, simply use the message as the address and look up the encoded message. Similarly, the encoding process could be a table lookup using addresses as long as the message of the encoded

block. Such methods, being rapid, tend to emphasize short blocks with higher redundancy than the more elaborate process of encoding (as given above), which works on arbitrarily long messages (although, of course, the longer the message, the more parity checks must be computed—in parallel if you have the available equipment).

Exercise

3.5-1 In the $n = 15$ code, discuss in detail the encoding and decoding if the four parity checks are moved to positions 12, 13, 14, and 15.

3.6 Geometric Approach

We have just given an algebraic approach to error-correcting codes. A different, equivalent approach to the topic is from n-dimensional geometry. In this model we consider the string of n 0's and 1's as a point in n-dimensional space. Each digit gives the value of the corresponding coordinate in the n-dimensional space (where we are assuming that the encoded message is exactly n bits long). Thus we have a cube in n-dimensional space; each vertex is a string of n 0's and 1's. The space consists *only* of the 2^n vertices; there is nothing else in the space of all possible messages except the 2^n vertices. This is sometimes called a "vector space."

Each vertex is a possible received message, but only selected vertices are to be original messages. A *single* error in a message moves the message point along one edge of the imagined cube to an immediately adjacent point. If we require that every possible originating message be at least a *distance* of two sides away from any other message point, then it is clear that any single error will move a message only one side away and leave the received message as an illegitimate message. If the minimum distance between message points is three sides of the cube, then any single error will leave the received message *closer* to the original message than to any other message and thus we can have single error correction.

Effectively, we have introduced a *distance function*, which is the minimum number of sides of the cube we have to traverse to get from one point to another. This is the same as the number of bits in the representations of the two points that differ. Thus the distance can be looked on as being the number of 1's in the *logical* difference, or sum, of the two points. It is a legitimate distance function since it satisfies the following three conditions:

1. The distance from a point to itself is 0.
2. The distance from point x to point y is the same as the distance from y to x and is a positive number.
3. The triangle inequality holds—the sum of two sides (distance from a to c plus the distance from c to b) is at least as great as the length of the third side (the distance from a to b).

This distance function is usually called the *Hamming distance*. It is the distance function for *binary white noise*.

Using this distance function, we can define various things in the space. In particular, the *surface of a sphere* about a point is the set of points a given distance away. The surface of sphere of radius 1 about the point (0, 0, 0, . . . , 0) is the set of all vertices in the space which are one unit away, that is, all vertices that have only one 1 in their coordinate representation (see Figure 3.6-1). There are $C(n, 1)$ such points.

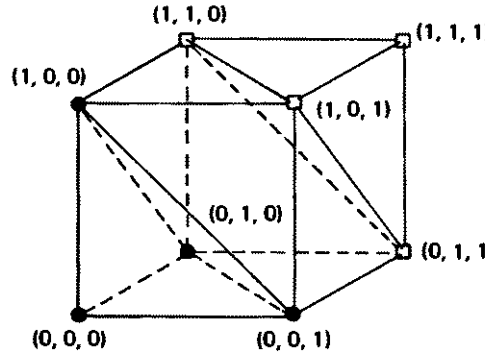

Figure 3-6.1 Three-dimensional spheres about (0, 0, 0) and (1, 1, 1)

We can express the minimum distance between vertices of a set of message points in terms of the error correctability possible. The minimum distance must be at least 1 for uniqueness of the code (Table 3.6-1). A minimum distance of 2 gives single-error detectability. A minimum distance of 3 gives single-error correctability; any single error leaves the point closer to where it was than to any other possible message. Of course, this minimum-distance code could be used instead for double-error detection. A minimum distance of 4 will give both single-error correction plus double-error detection. A minimum distance of 5 would allow double-error correction. *Conversely*, if the required de-

TABLE 3.6-1 *Meaning of Minimum Distance*

Minimum Distance	Meaning
1	Uniqueness
2	Single-error detection
3	Single-error correction (*or* double-error detection)
4	Single-error correction plus double-error detection (*or* triple-error detection)
5	Double-error correction
etc.	

gree of detection or correction is to be achieved, the corresponding minimum distance between message points must be observed.

In the case of single-error correction with the minimum distance of 3, we can surround each message point by a unit sphere and not have the spheres overlap. The *volume* of a sphere of radius 1 is the center plus the n points with just one coordinate changed, a volume of $1 + n$. The total volume of the n-dimensional space is clearly 2^n, the number of possible points. Since the spheres do not overlap, the maximum number of message positions k must satisfy

$$\frac{\text{Total volume}}{\text{Volume of a sphere}} \geq \text{maximum number of spheres}$$

or

$$\frac{2^n}{n+1} \geq 2^k \tag{3.6-1}$$

Since

$$n = m + k$$

$$2^{m+k} \geq 2^k(n+1)$$

or

$$2^m \geq (n+1)$$

This is the same inequality as we derived from the algebraic approach, equation (3.4-1).

We can now see the restrictions for higher error correction. Thus for double-error correction we must have a minimum distance of 5, and we can put nonoverlapping spheres of radius 2 about each message position. The volume of a sphere of radius 2 is the center position plus the n positions a distance 1 away, plus those with two coordinates out of the n changed, which is the binomial coefficient $C(n, 2) = n(n - 1)/2$. Dividing the total volume of the space, 2^n, by the volume of these spheres gives an upper bound on the number k of possible code message positions in the space

$$\frac{2^n}{1 + n + n(n - 1)/2} \geq 2^k \tag{3.6-2}$$

It does not mean that this number can be achieved, only that this is an upper bound. Similar inequalities can be written for larger spheres.

When the spheres about the message points completely exhaust the space of 2^n points, leaving no points outside some sphere, the code is called a *perfect code*. Such codes have a high degree of symmetry (in the geometric model each point is equivalent to any other point) and a particularly simple theory. In only a comparatively few cases do perfect codes exist. They require that the inequalities be equalities.

Exercises

3.6-1 Extend the bounds of (3.6-1) and (3.6-2) to higher error-correcting codes.

3.6-2 Make a table for the bound for double-error correction of equation (3.6-2) ($n = 3, 4, 5, \ldots, 11$).

3.7 Single-Error-Correction Plus Double-Error-Detection Codes

It is seldom wise to use only single-error correction, because a double error would then fool the system in its attempts to correct, and the system would use the syndrome to correct the wrong place; thus there would be three errors in the decoded message. Instead, a single-error-correction plus a double-detection code makes a reasonably balanced system for many (but not all) situations. The condition for the double detection is that the minimum distance must be increased by 1; it must be 4.

To make a double-error-detecting code from a single-error-correcting code, we add one more parity check (and one more position): this check is over the whole message. Thus any single error will still produce the right syndrome and the added parity check will give a 1. A double error will now cause a nonzero syndrome, but leave this added parity check satisfied. This situation can then be recognized as a double error; namely, some syndrome appears, but the extra parity check is still correct (Table 3.7-1). It is easy to see that two points which were

TABLE 3.7-1 *Double-Error Detection*

Original Syndrome	New Parity Check	Meaning
0	0	Correct
0	1	Error in added position
Something	1	Original meaning
Something	0	Double error

at the minimum distance of 3 from each other in the original code had a different number of 1's in them, modulo 2. Thus their corresponding extra parity checks would be set differently, increasing the distance between them to 4.

The argument we have just given applies to both the algebraic and geometric approaches, and shows how the two tend to complement each other. We have not yet given any constructive method for finding the higher error-correcting codes; we have only given bounds on them. Chapter 11 is devoted to the elements of their construction. The full theory for error-correcting codes has been developed over the years and is very complex, so we will only indicate the general approach.

Notice that the theory *assumes* that the correcting equipment is working properly; it is only the errors in the received (computed) message that are being handled.

Exercise

3.7-1 Show that the argument to get extra error detection can be applied to any odd minimum distance to get the next-higher (even) minimum distance.

3.8 Hamming Codes Using Words

On many computers it is difficult to get at the individual bits of information, but the computer words are readily available. We can therefore think of the parity checks as being whole words; we logically add, not bits, but words. The logical sum over all the selected words plus the check should give a word with all 0's, and any failure marks the failure of the parity check. If we have, for example, a single-error-correcting code, then we get the location of the wrong word and can reconstruct it easily by, if nothing else, looking at the bits of the parity checks. It should be clear that we could simply add the parity-check failure word to the word in the location of the erroneous word to get the correct word. In this fashion we can correct the errors in any one word, whether it is one, two, . . . , or all the bits of the word that are wrong.

A mixed system results when you use the parity check on the ASCII code bits. The ASCII parity check locates the word with the error and the single-error-detecting parity word over all the words gives the *position in the word* that is wrong. Again, you simply add the parity check to the word that is wrong to do the correction. This amounts to a rectangular code. It will correct any odd number of errors in a single word.

3.9 Applications of the Ideas

In Section 1.2 we claimed that not only were the ideas in the text directly useful but they also had wider general application—we used evolution as an example.

The central idea of error detection and correction is that the meaningful messages must be kept far apart (in the space of probable errors) if we are to handle errors successfully. If two of the possible messages are not far enough apart, one message can be carried by an error (or errors) into the other, or carried at least so close that at the receiving end we will make a mistake in identifying the source.

In assigning names to variables and labels in a FORTRAN, COBOL, ALGOL, Pascal, Ada, and other high-level language, the names should be kept far apart; otherwise, an intellectual "slip of the pen" or a typical error in keying in the name can transfer one name to another meaningful name. If the names are made to differ in at least two positions, single typos will be caught by the assembler. Thus the use of short mnemonic names should be tempered by the prudent need to protect oneself against small slips.

If you estimate the time to locate an error in miswriting a name against the time to add one or more characters to all the names while writing the original program, you will see that using longer-than-minimum names is probably a very wise idea—but it is unlikely that this computation will convince many people to do so! Minimal-length names are the source of much needless confusion and waste of time, yours and the machine's.

The distance function in the Hamming codes is based completely on white noise. Sections 2.7 and 2.9 are included to emphasize that the proper distance function to use depends in some cases on the *psychological distance* between the names as well as more uniformly random keystroke errors.

3.10 Summary

We have given the fundamental nature of error detection and error correction for white noise, namely the minimum distance between message points that must be observed. We have given methods for constructing codes for:

Single-error detecting	min. dist. = 2
Single-error correction	min. dist. = 3
Single-error correction + double-error detection	min. dist. = 4

Their design is easy, and they are practical to construct in software or in hardware chips. They can compensate for weak spots in a system, or can be used throughout an entire system, to get reliable performance out of unreliable parts. One need not put up with poor equipment performance, but the price is both in storage (or time of transmission) and equipment (or time) to encode and possible correct. You do not get something for nothing! The codes also make valuable contributions to maintenance since they pinpoint the error, and the repairmen will not try fixing the wrong things (meaning "fix" what is working right and ignore what is causing the error!). A more widespread use of the idea of distance between messages was sketched in Section 3.9.

The use of such codes, and more highly developed codes, is rapidly spreading as we get faster and smaller integrated-circuit components. Increasingly, in VLSI (very large system integration) chips the code is part of the hardware.