mp3
MPEG2
H.263
H.264
Wavelets
SPIHT
JPEG-LS
CELP
PPM
BWT
VQ
TCS
CALIC
MELP

KHALID SAYOOD

Introduction to DATA COMPRESSION

THIRD EDITION

MK
MORGAN KAUFMANN

THIRD EDITION

Introduction to Data Compression

The Morgan Kaufmann Series in Multimedia Information and Systems
Series Editor, Edward A. Fox, Virginia Polytechnic University

Introduction to Data Compression, Third Edition
Khalid Sayood

Understanding Digital Libraries, Second Edition
Michael Lesk

Bioinformatics: Managing Scientific Data
Zoe Lacroix and Terence Critchlow

How to Build a Digital Library
Ian H. Witten and David Bainbridge

Digital Watermarking
Ingemar J. Cox, Matthew L. Miller, and Jeffrey A. Bloom

Readings in Multimedia Computing and Networking
Edited by Kevin Jeffay and HongJiang Zhang

Introduction to Data Compression, Second Edition
Khalid Sayood

Multimedia Servers: Applications, Environments, and Design
Dinkar Sitaram and Asit Dan

Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition
Ian H. Witten, Alistair Moffat, and Timothy C. Bell

Digital Compression for Multimedia: Principles and Standards
Jerry D. Gibson, Toby Berger, Tom Lookabaugh, Dave Lindbergh, and Richard L. Baker

Readings in Information Retrieval
Edited by Karen Sparck Jones and Peter Willett

THIRD EDITION

Introduction to Data Compression

Khalid Sayood
University of Nebraska

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann is an imprint of Elsevier

MORGAN KAUFMANN®

Senior Acquisitions Editor	Rick Adams
Publishing Services Manager	Simon Crump
Assistant Editor	Rachel Roumeliotis
Cover Design	Cate Barr
Composition	Integra Software Services Pvt. Ltd.
Copyeditor	Jessika Bella Mura
Proofreader	Jacqui Brownstein
Indexer	Northwind Editorial Sevices
Interior printer	Maple Vail Book Manufacturing Group
Cover printer	Phoenix Color

Morgan Kaufmann Publishers is an imprint of Elsevier.
500 Sansome Street, Suite 400, San Francisco, CA 94111

This book is printed on acid-free paper.

Library of Congress Cataloging-in-Publication Data
Sayood, Khalid.
Introduction to data compression/Khalid Sayood.—3rd ed.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-12-620862-7
ISBN-10: 0-12-620862-X
1. Data compression (Telecommunication) 2. Coding theory. I. Title
TK5102.92.S39 2005
005.74′6—dc22
2005052759

ISBN 13: 978-0-12-620862-7
ISBN 10: 0-12-620862-X

For information on all Morgan Kaufmann publications,
visit our Web site at www.mkp.com or www.books.elsevier.com

Printed in the United States of America
05 06 07 08 09 5 4 3 2 1

To Füsun

Contents

Preface

Within the last decade the use of data compression has become ubiquitous. From *mp3* players whose headphones seem to adorn the ears of most young (and some not so young) people, to cell phones, to DVDs, to digital television, data compression is an integral part of almost all information technology. This incorporation of compression into more and more of our lives also points to a certain degree of maturation of the technology. This maturity is reflected in the fact that there are fewer differences between this and the previous edition of this book than there were between the second and first editions. In the second edition we had added new techniques that had been developed since the first edition of this book came out. In this edition our purpose is more to include some important topics, such as audio compression, that had not been adequately covered in the second edition. During this time the field has not entirely stood still and we have tried to include information about new developments. We have added a new chapter on audio compression (including a description of the *mp3* algorithm). We have added information on new standards such as the new video coding standard and the new facsimile standard. We have reorganized some of the material in the book, collecting together various lossless image compression techniques and standards into a single chapter, and we have updated a number of chapters, adding information that perhaps should have been there from the beginning.

All this has yet again enlarged the book. However, the intent remains the same: to provide an introduction to the art or science of data compression. There is a tutorial description of most of the popular compression techniques followed by a description of how these techniques are used for image, speech, text, audio, and video compression.

Given the pace of developments in this area, there are bound to be new ones that are not reflected in this book. In order to keep you informed of these developments, we will periodically provide updates at *http://www.mkp.com*.

Audience

If you are designing hardware or software implementations of compression algorithms, or need to interact with individuals engaged in such design, or are involved in development of multimedia applications and have some background in either electrical or computer engineering, or computer science, this book should be useful to you. We have included a large number of examples to aid in self-study. We have also included discussion of various multimedia standards. The intent here is not to provide all the details that may be required to implement a standard but to provide information that will help you follow and understand the standards documents.

Course Use

The impetus for writing this book came from the need for a self-contained book that could be used at the senior/graduate level for a course in data compression in either electrical engineering, computer engineering, or computer science departments. There are problems and project ideas after most of the chapters. A solutions manual is available from the publisher. Also at *http://sensin.unl.edu/idc/index.html* we provide links to various course homepages, which can be a valuable source of project ideas and support material.

The material in this book is too much for a one semester course. However, with judicious use of the starred sections, this book can be tailored to fit a number of compression courses that emphasize various aspects of compression. If the course emphasis is on lossless compression, the instructor could cover most of the sections in the first seven chapters. Then, to give a taste of lossy compression, the instructor could cover Sections 1–5 of Chapter 9, followed by Chapter 13 and its description of JPEG, and Chapter 18, which describes video compression approaches used in multimedia communications. If the class interest is more attuned to audio compression, then instead of Chapters 13 and 18, the instructor could cover Chapters 14 and 16. If the latter option is taken, depending on the background of the students in the class, Chapter 12 may be assigned as background reading. If the emphasis is to be on lossy compression, the instructor could cover Chapter 2, the first two sections of Chapter 3, Sections 4 and 6 of Chapter 4 (with a cursory overview of Sections 2 and 3), Chapter 8, selected parts of Chapter 9, and Chapter 10 through 15. At this point depending on the time available and the interests of the instructor and the students portions of the remaining three chapters can be covered. I have always found it useful to assign a term project in which the students can follow their own interests as a means of covering material that is not covered in class but is of interest to the student.

Approach

In this book, we cover both lossless and lossy compression techniques with applications to image, speech, text, audio, and video compression. The various lossless and lossy coding techniques are introduced with just enough theory to tie things together. The necessary theory is introduced just before we need it. Therefore, there are three *mathematical preliminaries* chapters. In each of these chapters, we present the mathematical material needed to understand and appreciate the techniques that follow.

Although this book is an introductory text, the word *introduction* may have a different meaning for different audiences. We have tried to accommodate the needs of different audiences by taking a dual-track approach. Wherever we felt there was material that could enhance the understanding of the subject being discussed but could still be skipped without seriously hindering your understanding of the technique, we marked those sections with a star ($\star$). If you are primarily interested in understanding how the various techniques function, especially if you are using this book for self-study, we recommend you skip the starred sections, at least in a first reading. Readers who require a slightly more theoretical approach should use the starred sections. Except for the starred sections, we have tried to keep the mathematics to a minimum.

Learning from This Book

I have found that it is easier for me to understand things if I can see examples. Therefore, I have relied heavily on examples to explain concepts. You may find it useful to spend more time with the examples if you have difficulty with some of the concepts.

Compression is still largely an art and to gain proficiency in an art we need to get a "feel" for the process. We have included software implementations for most of the techniques discussed in this book, along with a large number of data sets. The software and data sets can be obtained from *ftp://ftp.mkp.com/pub/Sayood/*. The programs are written in C and have been tested on a number of platforms. The programs should run under most flavors of UNIX machines and, with some slight modifications, under other operating systems as well. More detailed information is contained in the README file in the *pub/Sayood* directory.

You are strongly encouraged to use and modify these programs to work with your favorite data in order to understand some of the issues involved in compression. A useful and achievable goal should be the development of your own compression package by the time you have worked through this book. This would also be a good way to learn the trade-offs involved in different approaches. We have tried to give comparisons of techniques wherever possible; however, different types of data have their own idiosyncrasies. The best way to know which scheme to use in any given situation is to try them.

Content and Organization

The organization of the chapters is as follows: We introduce the mathematical preliminaries necessary for understanding lossless compression in Chapter 2; Chapters 3 and 4 are devoted to coding algorithms, including Huffman coding, arithmetic coding, Golomb-Rice codes, and Tunstall codes. Chapters 5 and 6 describe many of the popular lossless compression schemes along with their applications. The schemes include LZW, *ppm*, BWT, and DMC, among others. In Chapter 7 we describe a number of lossless image compression algorithms and their applications in a number of international standards. The standards include the JBIG standards and various facsimile standards.

Chapter 8 is devoted to providing the mathematical preliminaries for lossy compression. Quantization is at the heart of most lossy compression schemes. Chapters 9 and 10 are devoted to the study of quantization. Chapter 9 deals with scalar quantization, and Chapter 10 deals with vector quantization. Chapter 11 deals with differential encoding techniques, in particular differential pulse code modulation (DPCM) and delta modulation. Included in this chapter is a discussion of the CCITT G.726 standard.

Chapter 12 is our third mathematical preliminaries chapter. The goal of this chapter is to provide the mathematical foundation necessary to understand some aspects of the transform, subband, and wavelet-based techniques that are described in the next three chapters. As in the case of the previous mathematical preliminaries chapters, not all material covered is necessary for everyone. We describe the JPEG standard in Chapter 13, the CCITT G.722 international standard in Chapter 14, and EZW, SPIHT, and JPEG 2000 in Chapter 15.

Chapter 16 is devoted to audio compression. We describe the various MPEG audio compression schemes in this chapter including the scheme popularly known as *mp3*.

Chapter 17 covers techniques in which the data to be compressed are analyzed, and a model for the generation of the data is transmitted to the receiver. The receiver uses this model to synthesize the data. These analysis/synthesis and analysis by synthesis schemes include linear predictive schemes used for low-rate speech coding and the fractal compression technique. We describe the federal government LPC-10 standard. Code-excited linear prediction (CELP) is a popular example of an analysis by synthesis scheme. We also discuss three CELP-based standards, the federal standard 1016, the CCITT G.728 international standard, and the relatively new wideband speech compression standard G.722.2. We have also included a discussion of the mixed excitation linear prediction (MELP) technique, which is the new federal standard for speech coding at 2.4 kbps.

Chapter 18 deals with video coding. We describe popular video coding techniques via description of various international standards, including H.261, H.264, and the various MPEG standards.

A Personal View

For me, data compression is more than a manipulation of numbers; it is the process of discovering structures that exist in the data. In the 9th century, the poet Omar Khayyam wrote

> The moving finger writes, and having writ,
> moves on; not all thy piety nor wit,
> shall lure it back to cancel half a line,
> nor all thy tears wash out a word of it.
> *(The Rubaiyat of Omar Khayyam)*

To explain these few lines would take volumes. They tap into a common human experience so that in our mind's eye, we can reconstruct what the poet was trying to convey centuries ago. To understand the words we not only need to know the language, we also need to have a model of reality that is close to that of the poet. The genius of the poet lies in identifying a model of reality that is so much a part of our humanity that centuries later and in widely diverse cultures, these few words can evoke volumes.

Data compression is much more limited in its aspirations, and it may be presumptuous to mention it in the same breath as poetry. But there is much that is similar to both endeavors. Data compression involves identifying models for the many different types of structures that exist in different types of data and then using these models, perhaps along with the perceptual framework in which these data will be used, to obtain a compact representation of the data. These structures can be in the form of patterns that we can recognize simply by plotting the data, or they might be statistical structures that require a more mathematical approach to comprehend.

In *The Long Dark Teatime of the Soul* by Douglas Adams, the protagonist finds that he can enter Valhalla (a rather shoddy one) if he tilts his head in a certain way. Appreciating the structures that exist in data sometimes require us to tilt our heads in a certain way. There are an infinite number of ways we can tilt our head and, in order not to get a pain in the neck (carrying our analogy to absurd limits), it would be nice to know some of the ways that

will generally lead to a profitable result. One of the objectives of this book is to provide you with a frame of reference that can be used for further exploration. I hope this exploration will provide as much enjoyment for you as it has given to me.

Acknowledgments

It has been a lot of fun writing this book. My task has been made considerably easier and the end product considerably better because of the help I have received. Acknowledging that help is itself a pleasure.

The first edition benefitted from the careful and detailed criticism of Roy Hoffman from IBM, Glen Langdon from the University of California at Santa Cruz, Debra Lelewer from California Polytechnic State University, Eve Riskin from the University of Washington, Ibrahim Sezan from Kodak, and Peter Swaszek from the University of Rhode Island. They provided detailed comments on all or most of the first edition. Nasir Memon from Polytechnic University, Victor Ramamoorthy then at S3, Grant Davidson at Dolby Corporation, Hakan Caglar, who was then at TÜBITAK in Istanbul, and Allen Gersho from the University of California at Santa Barbara reviewed parts of the manuscript.

For the second edition Steve Tate at the University of North Texas, Sheila Horan at New Mexico State University, Edouard Lamboray at Oerlikon Contraves Group, Steven Pigeon at the University of Montreal, and Jesse Olvera at Raytheon Systems reviewed the entire manuscript. Emin Anarım of Boğaziçi University and Hakan Çağlar helped me with the development of the chapter on wavelets. Mark Fowler provided extensive comments on Chapters 12–15, correcting mistakes of both commission and omission. Tim James, Devajani Khataniar, and Lance Pérez also read and critiqued parts of the new material in the second edition. Chloeann Nelson, along with trying to stop me from splitting infinitives, also tried to make the first two editions of the book more user-friendly.

Since the appearance of the first edition, various readers have sent me their comments and critiques. I am grateful to all who sent me comments and suggestions. I am especially grateful to Roberto Lopez-Hernandez, Dirk vom Stein, Christopher A. Larrieu, Ren Yih Wu, Humberto D'Ochoa, Roderick Mills, Mark Elston, and Jeerasuda Keesorth for pointing out errors and suggesting improvements to the book. I am also grateful to the various instructors who have sent me their critiques. In particular I would like to thank Bruce Bomar from the University of Tennessee, Mark Fowler from SUNY Binghamton, Paul Amer from the University of Delaware, K.R. Rao from the University of Texas at Arlington, Ralph Wilkerson from the University of Missouri–Rolla, Adam Drozdek from Duquesne University, Ed Hong and Richard Ladner from the University of Washington, Lars Nyland from the Colorado School of Mines, Mario Kovac from the University of Zagreb, and Pierre Jouvelet from the Ecole Superieure des Mines de Paris.

Frazer Williams and Mike Hoffman, from my department at the University of Nebraska, provided reviews for the first edition of the book. Mike read the new chapters in the second and third edition in their raw form and provided me with critiques that led to major rewrites. His insights were always helpful and the book carries more of his imprint than he is perhaps aware of. It is nice to have friends of his intellectual caliber and generosity. Rob Maher at Montana State University provided me with an extensive critique of the new chapter on

audio compression pointing out errors in my thinking and gently suggesting corrections. I thank him for his expertise, his time, and his courtesy.

Rick Adams, Rachel Roumeliotis, and Simon Crump at Morgan Kaufmann had the task of actually getting the book out. This included the unenviable task of getting me to meet deadlines. Vytas Statulevicius helped me with LaTex problems that were driving me up the wall.

Most of the examples in this book were generated in a lab set up by Andy Hadenfeldt. James Nau helped me extricate myself out of numerous software puddles giving freely of his time. In my times of panic, he was always just an email or voice mail away.

I would like to thank the various "models" for the data sets that accompany this book and were used as examples. The individuals in the images are Sinan Sayood, Sena Sayood, and Elif Sevuktekin. The female voice belongs to Pat Masek.

This book reflects what I have learned over the years. I have been very fortunate in the teachers I have had. David Farden, now at North Dakota State University, introduced me to the area of digital communication. Norm Griswold at Texas A&M University introduced me to the area of data compression. Jerry Gibson, now at University of California at Santa Barbara was my Ph.D. advisor and helped me get started on my professional career. The world may not thank him for that, but I certainly do.

I have also learned a lot from my students at the University of Nebraska and Boğaziçi University. Their interest and curiosity forced me to learn and kept me in touch with the broad field that is data compression today. I learned at least as much from them as they learned from me.

Much of this learning would not have been possible but for the support I received from NASA. The late Warner Miller and Pen-Shu Yeh at the Goddard Space Flight Center and Wayne Whyte at the Lewis Research Center were a source of support and ideas. I am truly grateful for their helpful guidance, trust, and friendship.

Our two boys, Sena and Sinan, graciously forgave my evenings and weekends at work. They were tiny (witness the images) when I first started writing this book. Soon I will have to look up when talking to them. "The book" has been their (sometimes unwanted) companion through all these years. For their graciousness and for always being such perfect joys, I thank them.

Above all the person most responsible for the existence of this book is my partner and closest friend Füsun. Her support and her friendship gives me the freedom to do things I would not otherwise even consider. She centers my universe and, as with every significant endeavor that I have undertaken since I met her, this book is at least as much hers as it is mine.

1

Introduction

In the last decade we have been witnessing a transformation—some call it a revolution—in the way we communicate, and the process is still under way. This transformation includes the ever-present, ever-growing Internet; the explosive development of mobile communications; and the ever-increasing importance of video communication. Data compression is one of the enabling technologies for each of these aspects of the multimedia revolution. It would not be practical to put images, let alone audio and video, on websites if it were not for data compression algorithms. Cellular phones would not be able to provide communication with increasing clarity were it not for compression. The advent of digital TV would not be possible without compression. Data compression, which for a long time was the domain of a relatively small group of engineers and scientists, is now ubiquitous. Make a long-distance call and you are using compression. Use your modem, or your fax machine, and you will benefit from compression. Listen to music on your *mp3* player or watch a DVD and you are being entertained courtesy of compression.

So, what is data compression, and why do we need it? Most of you have heard of JPEG and MPEG, which are standards for representing images, video, and audio. Data compression algorithms are used in these standards to reduce the number of bits required to represent an image or a video sequence or music. In brief, data compression is the art or science of representing information in a compact form. We create these compact representations by identifying and using structures that exist in the data. Data can be characters in a text file, numbers that are samples of speech or image waveforms, or sequences of numbers that are generated by other processes. The reason we need data compression is that more and more of the information that we generate and use is in digital form—in the form of numbers represented by bytes of data. And the number of bytes required to represent multimedia data can be huge. For example, in order to digitally represent 1 second of video without compression (using the CCIR 601 format), we need more than 20 megabytes, or 160 megabits. If we consider the number of seconds in a movie, we can easily see why we would need compression. To represent 2 minutes of uncompressed CD-quality

music (44,100 samples per second, 16 bits per sample) requires more than 84 million bits. Downloading music from a website at these rates would take a long time.

As human activity has a greater and greater impact on our environment, there is an ever-increasing need for more information about our environment, how it functions, and what we are doing to it. Various space agencies from around the world, including the European Space Agency (ESA), the National Aeronautics and Space Agency (NASA), the Canadian Space Agency (CSA), and the Japanese Space Agency (STA), are collaborating on a program to monitor global change that will generate half a terabyte of data per *day* when they are fully operational. Compare this to the 130 terabytes of data currently stored at the EROS data center in South Dakota, that is the largest archive for land mass data in the world.

Given the explosive growth of data that needs to be transmitted and stored, why not focus on developing better transmission and storage technologies? This is happening, but it is not enough. There have been significant advances that permit larger and larger volumes of information to be stored and transmitted without using compression, including CD-ROMs, optical fibers, Asymmetric Digital Subscriber Lines (ADSL), and cable modems. However, while it is true that both storage and transmission capacities are steadily increasing with new technological innovations, as a corollary to Parkinson's First Law,[1] it seems that the need for mass storage and transmission increases at least twice as fast as storage and transmission capacities improve. Then there are situations in which capacity has not increased significantly. For example, the amount of information we can transmit over the airwaves will always be limited by the characteristics of the atmosphere.

An early example of data compression is Morse code, developed by Samuel Morse in the mid-19th century. Letters sent by telegraph are encoded with dots and dashes. Morse noticed that certain letters occurred more often than others. In order to reduce the average time required to send a message, he assigned shorter sequences to letters that occur more frequently, such as e ($\cdot$) and a ($\cdot -$), and longer sequences to letters that occur less frequently, such as q ($- - \cdot -$) and j ($\cdot - - -$). This idea of using shorter codes for more frequently occurring characters is used in Huffman coding, which we will describe in Chapter 3.

Where Morse code uses the frequency of occurrence of single characters, a widely used form of Braille code, which was also developed in the mid-19th century, uses the frequency of occurrence of words to provide compression [1]. In Braille coding, 2×3 arrays of dots are used to represent text. Different letters can be represented depending on whether the dots are raised or flat. In Grade 1 Braille, each array of six dots represents a single character. However, given six dots with two positions for each dot, we can obtain 2^6, or 64, different combinations. If we use 26 of these for the different letters, we have 38 combinations left. In Grade 2 Braille, some of these leftover combinations are used to represent words that occur frequently, such as "and" and "for." One of the combinations is used as a special symbol indicating that the symbol that follows is a word and not a character, thus allowing a large number of words to be represented by two arrays of dots. These modifications, along with contractions of some of the words, result in an average reduction in space, or compression, of about 20% [1].

[1] Parkinson's First Law: "Work expands so as to fill the time available," in *Parkinson's Law and Other Studies in Administration*, by Cyril Northcote Parkinson, Ballantine Books, New York, 1957.

Statistical structure is being used to provide compression in these examples, but that is not the only kind of structure that exists in the data. There are many other kinds of structures existing in data of different types that can be exploited for compression. Consider speech. When we speak, the physical construction of our voice box dictates the kinds of sounds that we can produce. That is, the mechanics of speech production impose a structure on speech. Therefore, instead of transmitting the speech itself, we could send information about the conformation of the voice box, which could be used by the receiver to synthesize the speech. An adequate amount of information about the conformation of the voice box can be represented much more compactly than the numbers that are the sampled values of speech. Therefore, we get compression. This compression approach is being used currently in a number of applications, including transmission of speech over mobile radios and the synthetic voice in toys that speak. An early version of this compression approach, called the *vocoder* (*vo*ice *coder*), was developed by Homer Dudley at Bell Laboratories in 1936. The vocoder was demonstrated at the New York World's Fair in 1939, where it was a major attraction. We will revisit the vocoder and this approach to compression of speech in Chapter 17.

These are only a few of the many different types of structures that can be used to obtain compression. The structure in the data is not the only thing that can be exploited to obtain compression. We can also make use of the characteristics of the user of the data. Many times, for example, when transmitting or storing speech and images, the data are intended to be perceived by a human, and humans have limited perceptual abilities. For example, we cannot hear the very high frequency sounds that dogs can hear. If something is represented in the data that cannot be perceived by the user, is there any point in preserving that information? The answer often is "no." Therefore, we can make use of the perceptual limitations of humans to obtain compression by discarding irrelevant information. This approach is used in a number of compression schemes that we will visit in Chapters 13, 14, and 16.

Before we embark on our study of data compression techniques, let's take a general look at the area and define some of the key terms and concepts we will be using in the rest of the book.

1.1 Compression Techniques

When we speak of a compression technique or compression algorithm,[2] we are actually referring to two algorithms. There is the compression algorithm that takes an input $\mathcal{X}$ and generates a representation $\mathcal{X}_c$ that requires fewer bits, and there is a reconstruction algorithm that operates on the compressed representation $\mathcal{X}_c$ to generate the reconstruction $\mathcal{Y}$. These operations are shown schematically in Figure 1.1. We will follow convention and refer to both the compression and reconstruction algorithms together to mean the compression algorithm.

[2] The word *algorithm* comes from the name of an early 9th-century Arab mathematician, Al-Khwarizmi, who wrote a treatise entitled *The Compendious Book on Calculation by* al-jabr *and* al-muqabala, in which he explored (among other things) the solution of various linear and quadratic equations via rules or an "algorithm." This approach became known as the method of Al-Khwarizmi. The name was changed to *algoritni* in Latin, from which we get the word *algorithm*. The name of the treatise also gave us the word *algebra* [2].

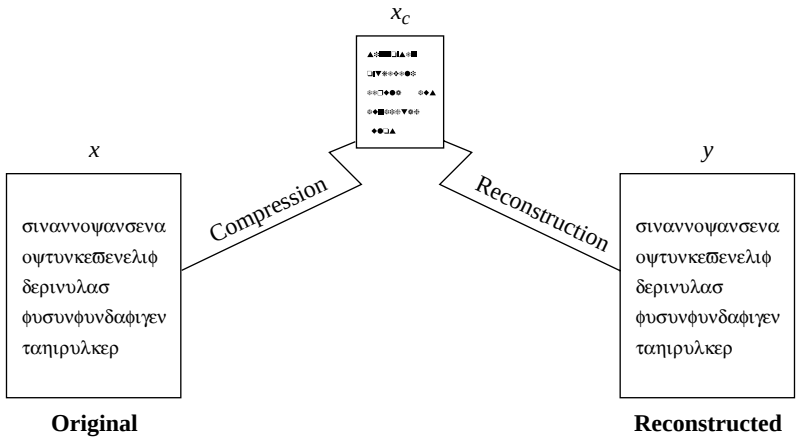

FIGURE 1.1 Compression and reconstruction.

Based on the requirements of reconstruction, data compression schemes can be divided into two broad classes: *lossless* compression schemes, in which $\mathcal{Y}$ is identical to $\mathcal{X}$, and *lossy* compression schemes, which generally provide much higher compression than lossless compression but allow $\mathcal{Y}$ to be different from $\mathcal{X}$.

1.1.1 Lossless Compression

Lossless compression techniques, as their name implies, involve no loss of information. If data have been losslessly compressed, the original data can be recovered exactly from the compressed data. Lossless compression is generally used for applications that cannot tolerate any difference between the original and reconstructed data.

Text compression is an important area for lossless compression. It is very important that the reconstruction is identical to the text original, as very small differences can result in statements with very different meanings. Consider the sentences "Do *not* send money" and "Do *now* send money." A similar argument holds for computer files and for certain types of data such as bank records.

If data of any kind are to be processed or "enhanced" later to yield more information, it is important that the integrity be preserved. For example, suppose we compressed a radiological image in a lossy fashion, and the difference between the reconstruction $\mathcal{Y}$ and the original $\mathcal{X}$ was visually undetectable. If this image was later enhanced, the previously undetectable differences may cause the appearance of artifacts that could seriously mislead the radiologist. Because the price for this kind of mishap may be a human life, it makes sense to be very careful about using a compression scheme that generates a reconstruction that is different from the original.

Data obtained from satellites often are processed later to obtain different numerical indicators of vegetation, deforestation, and so on. If the reconstructed data are not identical to the original data, processing may result in "enhancement" of the differences. It may not

be possible to go back and obtain the same data over again. Therefore, it is not advisable to allow for any differences to appear in the compression process.

There are many situations that require compression where we want the reconstruction to be identical to the original. There are also a number of situations in which it is possible to relax this requirement in order to get more compression. In these situations we look to lossy compression techniques.

1.1.2 Lossy Compression

Lossy compression techniques involve some loss of information, and data that have been compressed using lossy techniques generally cannot be recovered or reconstructed exactly. In return for accepting this distortion in the reconstruction, we can generally obtain much higher compression ratios than is possible with lossless compression.

In many applications, this lack of exact reconstruction is not a problem. For example, when storing or transmitting speech, the exact value of each sample of speech is not necessary. Depending on the quality required of the reconstructed speech, varying amounts of loss of information about the value of each sample can be tolerated. If the quality of the reconstructed speech is to be similar to that heard on the telephone, a significant loss of information can be tolerated. However, if the reconstructed speech needs to be of the quality heard on a compact disc, the amount of information loss that can be tolerated is much lower.

Similarly, when viewing a reconstruction of a video sequence, the fact that the reconstruction is different from the original is generally not important as long as the differences do not result in annoying artifacts. Thus, video is generally compressed using lossy compression.

Once we have developed a data compression scheme, we need to be able to measure its performance. Because of the number of different areas of application, different terms have been developed to describe and measure the performance.

1.1.3 Measures of Performance

A compression algorithm can be evaluated in a number of different ways. We could measure the relative complexity of the algorithm, the memory required to implement the algorithm, how fast the algorithm performs on a given machine, the amount of compression, and how closely the reconstruction resembles the original. In this book we will mainly be concerned with the last two criteria. Let us take each one in turn.

A very logical way of measuring how well a compression algorithm compresses a given set of data is to look at the ratio of the number of bits required to represent the data before compression to the number of bits required to represent the data after compression. This ratio is called the *compression ratio*. Suppose storing an image made up of a square array of 256×256 pixels requires 65,536 bytes. The image is compressed and the compressed version requires 16,384 bytes. We would say that the compression ratio is 4:1. We can also represent the compression ratio by expressing the reduction in the amount of data required as a percentage of the size of the original data. In this particular example the compression ratio calculated in this manner would be 75%.

Another way of reporting compression performance is to provide the average number of bits required to represent a single sample. This is generally referred to as the *rate*. For example, in the case of the compressed image described above, if we assume 8 bits per byte (or pixel), the average number of bits per pixel in the compressed representation is 2. Thus, we would say that the rate is 2 bits per pixel.

In lossy compression, the reconstruction differs from the original data. Therefore, in order to determine the efficiency of a compression algorithm, we have to have some way of quantifying the difference. The difference between the original and the reconstruction is often called the *distortion*. (We will describe several measures of distortion in Chapter 8.) Lossy techniques are generally used for the compression of data that originate as analog signals, such as speech and video. In compression of speech and video, the final arbiter of quality is human. Because human responses are difficult to model mathematically, many approximate measures of distortion are used to determine the quality of the reconstructed waveforms. We will discuss this topic in more detail in Chapter 8.

Other terms that are also used when talking about differences between the reconstruction and the original are *fidelity* and *quality*. When we say that the fidelity or quality of a reconstruction is high, we mean that the difference between the reconstruction and the original is small. Whether this difference is a mathematical difference or a perceptual difference should be evident from the context.

1.2 Modeling and Coding

While reconstruction requirements may force the decision of whether a compression scheme is to be lossy or lossless, the exact compression scheme we use will depend on a number of different factors. Some of the most important factors are the characteristics of the data that need to be compressed. A compression technique that will work well for the compression of text may not work well for compressing images. Each application presents a different set of challenges.

There is a saying attributed to Bobby Knight, the basketball coach at Texas Tech University: "If the only tool you have is a hammer, you approach every problem as if it were a nail." Our intention in this book is to provide you with a large number of tools that you can use to solve the particular data compression problem. It should be remembered that data compression, if it is a science at all, is an experimental science. The approach that works best for a particular application will depend to a large extent on the redundancies inherent in the data.

The development of data compression algorithms for a variety of data can be divided into two phases. The first phase is usually referred to as *modeling*. In this phase we try to extract information about any redundancy that exists in the data and describe the redundancy in the form of a model. The second phase is called *coding*. A description of the model and a "description" of how the data differ from the model are encoded, generally using a binary alphabet. The difference between the data and the model is often referred to as the *residual*. In the following three examples we will look at three different ways that data can be modeled. We will then use the model to obtain compression.

Example 1.2.1:

Consider the following sequence of numbers $\{x_1, x_2, x_3, \ldots\}$:

9	11	11	11	14	13	15	17	16	17	20	21

If we were to transmit or store the binary representations of these numbers, we would need to use 5 bits per sample. However, by exploiting the structure in the data, we can represent the sequence using fewer bits. If we plot these data as shown in Figure 1.2, we see that the data seem to fall on a straight line. A model for the data could therefore be a straight line given by the equation

$$\hat{x}_n = n + 8 \qquad n = 1, 2, \ldots$$

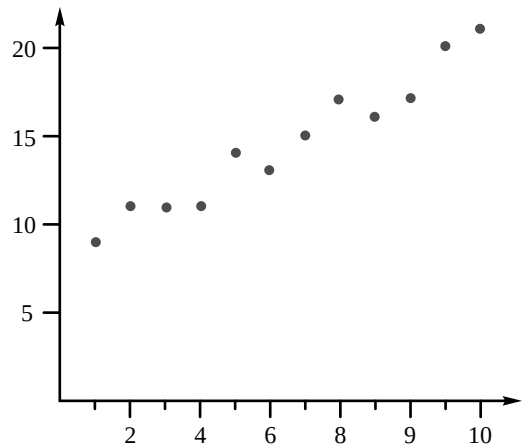

FIGURE 1.2 A sequence of data values.

Thus, the structure in the data can be characterized by an equation. To make use of this structure, let's examine the difference between the data and the model. The difference (or residual) is given by the sequence

$$e_n = x_n - \hat{x}_n \ : \ 0\ 1\ 0\ -1\ 1\ -1\ 0\ 1\ -1\ -1\ 1\ 1$$

The residual sequence consists of only three numbers $\{-1, 0, 1\}$. If we assign a code of 00 to -1, a code of 01 to 0, and a code of 10 to 1, we need to use 2 bits to represent each element of the residual sequence. Therefore, we can obtain compression by transmitting or storing the parameters of the model and the residual sequence. The encoding can be exact if the required compression is to be lossless, or approximate if the compression can be lossy. ♦

The type of structure or redundancy that existed in these data follows a simple law. Once we recognize this law, we can make use of the structure to *predict* the value of each element in the sequence and then encode the residual. Structure of this type is only one of many types of structure. Consider the following example.

Example 1.2.2:

Consider the following sequence of numbers:

27	28	29	28	26	27	29	28	30	32	34	36	38

The sequence is plotted in Figure 1.3.

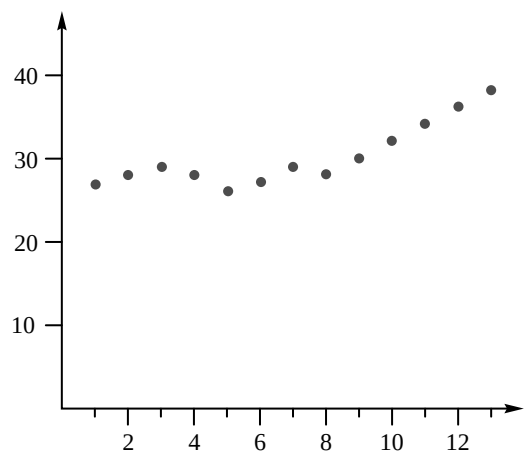

FIGURE 1.3 A sequence of data values.

The sequence does not seem to follow a simple law as in the previous case. However, each value is close to the previous value. Suppose we send the first value, then in place of subsequent values we send the difference between it and the previous value. The sequence of transmitted values would be

27	1	1	−1	−2	1	2	−1	2	2	2	2	2

Like the previous example, the number of distinct values has been reduced. Fewer bits are required to represent each number and compression is achieved. The decoder adds each received value to the previous decoded value to obtain the reconstruction corresponding

to the received value. Techniques that use the past values of a sequence to *predict* the current value and then encode the error in prediction, or residual, are called *predictive coding* schemes. We will discuss lossless predictive compression schemes in Chapter 7 and lossy predictive coding schemes in Chapter 11.

Assuming both encoder and decoder know the model being used, we would still have to send the value of the first element of the sequence. ♦

A very different type of redundancy is statistical in nature. Often we will encounter sources that generate some symbols more often than others. In these situations, it will be advantageous to assign binary codes of different lengths to different symbols.

Example 1.2.3:

Suppose we have the following sequence:

abarayaranbarraybranbfarbfaarbfaaarbaway

which is typical of all sequences generated by a source. Notice that the sequence is made up of eight different symbols. In order to represent eight symbols, we need to use 3 bits per symbol. Suppose instead we used the code shown in Table 1.1. Notice that we have assigned a codeword with only a single bit to the symbol that occurs most often, and correspondingly longer codewords to symbols that occur less often. If we substitute the codes for each symbol, we will use 106 bits to encode the entire sequence. As there are 41 symbols in the sequence, this works out to approximately 2.58 bits per symbol. This means we have obtained a compression ratio of 1.16:1. We will study how to use statistical redundancy of this sort in Chapters 3 and 4.

TABLE 1.1 A code with codewords of varying length.

a	1
n	001
b	01100
f	0100
n	0111
r	000
w	01101
y	0101

♦

When dealing with text, along with statistical redundancy, we also see redundancy in the form of words that repeat often. We can take advantage of this form of redundancy by constructing a list of these words and then represent them by their position in the list. This type of compression scheme is called a *dictionary* compression scheme. We will study these schemes in Chapter 5.

Often the structure or redundancy in the data becomes more evident when we look at groups of symbols. We will look at compression schemes that take advantage of this in Chapters 4 and 10.

Finally, there will be situations in which it is easier to take advantage of the structure if we decompose the data into a number of components. We can then study each component separately and use a model appropriate to that component. We will look at such schemes in Chapters 13, 14, and 15.

There are a number of different ways to characterize data. Different characterizations will lead to different compression schemes. We will study these compression schemes in the upcoming chapters, and use a number of examples that should help us understand the relationship between the characterization and the compression scheme.

With the increasing use of compression, there has also been an increasing need for standards. Standards allow products developed by different vendors to communicate. Thus, we can compress something with products from one vendor and reconstruct it using the products of a different vendor. The different international standards organizations have responded to this need, and a number of standards for various compression applications have been approved. We will discuss these standards as applications of the various compression techniques.

Finally, compression is still largely an art, and to gain proficiency in an art you need to get a feel for the process. To help, we have developed software implementations of most of the techniques discussed in this book, and also provided the data sets used for developing the examples in this book. Details on how to obtain these programs and data sets are provided in the Preface. You should use these programs on your favorite data or on the data sets provided in order to understand some of the issues involved in compression. We would also encourage you to write your own software implementations of some of these techniques, as very often the best way to understand how an algorithm works is to implement the algorithm.

1.3 Summary

In this chapter we have introduced the subject of data compression. We have provided some motivation for why we need data compression and defined some of the terminology we will need in this book. Additional terminology will be introduced as needed. We have briefly introduced the two major types of compression algorithms: lossless compression and lossy compression. Lossless compression is used for applications that require an exact reconstruction of the original data, while lossy compression is used when the user can tolerate some differences between the original and reconstructed representations of the data. An important element in the design of data compression algorithms is the modeling of the data. We have briefly looked at how modeling can help us in obtaining more compact representations of the data. We have described some of the different ways we can view the data in order to model it. The more ways we have of looking at the data, the more successful we will be in developing compression schemes that take full advantage of the structures in the data.

1.4 Projects and Problems

1. Use the compression utility on your computer to compress different files. Study the effect of the original file size and file type on the ratio of compressed file size to original file size.

2. Take a few paragraphs of text from a popular magazine and compress them by removing all words that are not essential for comprehension. For example, in the sentence "This is the dog that belongs to my friend," we can remove the words *is*, *the*, *that*, and *to* and still convey the same meaning. Let the ratio of the words removed to the total number of words in the original text be the measure of redundancy in the text. Repeat the experiment using paragraphs from a technical journal. Can you make any quantitative statements about the redundancy in the text obtained from different sources?

2

Mathematical Preliminaries for Lossless Compression

2.1 Overview

The treatment of data compression in this book is not very mathematical. (For a more mathematical treatment of some of the topics covered in this book, see [3, 4, 5, 6].) However, we do need some mathematical preliminaries to appreciate the compression techniques we will discuss. Compression schemes can be divided into two classes, lossy and lossless. Lossy compression schemes involve the loss of some information, and data that have been compressed using a lossy scheme generally cannot be recovered exactly. Lossless schemes compress the data without loss of information, and the original data can be recovered exactly from the compressed data. In this chapter, some of the ideas in information theory that provide the framework for the development of lossless data compression schemes are briefly reviewed. We will also look at some ways to model the data that lead to efficient coding schemes. We have assumed some knowledge of probability concepts (see Appendix A for a brief review of probability and random processes).

2.2 A Brief Introduction to Information Theory

Although the idea of a quantitative measure of information has been around for a while, the person who pulled everything together into what is now called information theory was Claude Elwood Shannon [7], an electrical engineer at Bell Labs. Shannon defined a quantity called *self-information*. Suppose we have an event A, which is a set of outcomes of some random

experiment. If $P(A)$ is the probability that the event A will occur, then the self-information associated with A is given by

$$i(A) = \log_b \frac{1}{P(A)} = -\log_b P(A). \tag{2.1}$$

Note that we have not specified the base of the log function. We will discuss this in more detail later in the chapter. The use of the logarithm to obtain a measure of information was not an arbitrary choice as we shall see later in this chapter. But first let's see if the use of a logarithm in this context makes sense from an intuitive point of view. Recall that $\log(1) = 0$, and $-\log(x)$ increases as x decreases from one to zero. Therefore, if the probability of an event is low, the amount of self-information associated with it is high; if the probability of an event is high, the information associated with it is low. Even if we ignore the mathematical definition of information and simply use the definition we use in everyday language, this makes some intuitive sense. The barking of a dog during a burglary is a high-probability event and, therefore, does not contain too much information. However, if the dog did not bark during a burglary, this is a low-probability event and contains a lot of information. (Obviously, Sherlock Holmes understood information theory!)[1] Although this equivalence of the mathematical and semantic definitions of information holds true most of the time, it does not hold all of the time. For example, a totally random string of letters will contain more information (in the mathematical sense) than a well-thought-out treatise on information theory.

Another property of this mathematical definition of information that makes intuitive sense is that the information obtained from the occurrence of two independent events is the sum of the information obtained from the occurrence of the individual events. Suppose A and B are two independent events. The self-information associated with the occurrence of both event A *and* event B is, by Equation (2.1),

$$i(AB) = \log_b \frac{1}{P(AB)}.$$

As A and B are independent,

$$P(AB) = P(A)P(B)$$

and

$$\begin{aligned} i(AB) &= \log_b \frac{1}{P(A)P(B)} \\ &= \log_b \frac{1}{P(A)} + \log_b \frac{1}{P(B)} \\ &= i(A) + i(B). \end{aligned}$$

The unit of information depends on the base of the log. If we use log base 2, the unit is *bits*; if we use log base e, the unit is *nats*; and if we use log base 10, the unit is *hartleys*.

[1] *Silver Blaze* by Arthur Conan Doyle.

Note that to calculate the information in bits, we need to take the logarithm base 2 of the probabilities. Because this probably does not appear on your calculator, let's review logarithms briefly. Recall that

$$\log_b x = a$$

means that

$$b^a = x.$$

Therefore, if we want to take the log base 2 of x

$$\log_2 x = a \Rightarrow 2^a = x,$$

we want to find the value of a. We can take the natural log (log base e) or log base 10 of both sides (which do appear on your calculator). Then

$$\ln(2^a) = \ln x \Rightarrow a \ln 2 = \ln x$$

and

$$a = \frac{\ln x}{\ln 2}$$

Example 2.2.1:

Let H and T be the outcomes of flipping a coin. If the coin is fair, then

$$P(H) = P(T) = \tfrac{1}{2}$$

and

$$i(H) = i(T) = 1 \text{ bit.}$$

If the coin is not fair, then we would expect the information associated with each event to be different. Suppose

$$P(H) = \tfrac{1}{8}, \quad P(T) = \tfrac{7}{8}.$$

Then

$$i(H) = 3 \text{ bits}, \quad i(T) = 0.193 \text{ bits.}$$

At least mathematically, the occurrence of a head conveys much more information than the occurrence of a tail. As we shall see later, this has certain consequences for how the information conveyed by these outcomes should be encoded. ♦

If we have a set of independent events A_i, which are sets of outcomes of some experiment $\mathcal{S}$, such that

$$\bigcup A_i = S$$

where S is the sample space, then the average self-information associated with the random experiment is given by

$$H = \sum P(A_i) i(A_i) = -\sum P(A_i) \log_b P(A_i).$$

This quantity is called the *entropy* associated with the experiment. One of the many contributions of Shannon was that he showed that if the experiment is a source that puts out symbols A_i from a set $\mathcal{A}$, then the entropy is a measure of the average number of binary symbols needed to code the output of the source. Shannon showed that the best that a lossless compression scheme can do is to encode the output of a source with an average number of bits equal to the entropy of the source.

The set of symbols $\mathcal{A}$ is often called the *alphabet* for the source, and the symbols are referred to as *letters*. For a general source $\mathcal{S}$ with alphabet $\mathcal{A} = \{1, 2, \ldots, m\}$ that generates a sequence $\{X_1, X_2, \ldots\}$, the entropy is given by

$$H(\mathcal{S}) = \lim_{n \to \infty} \frac{1}{n} G_n \tag{2.2}$$

where

$$G_n = -\sum_{i_1=1}^{i_1=m} \sum_{i_2=1}^{i_2=m} \cdots \sum_{i_n=1}^{i_n=m} P(X_1 = i_1, X_2 = i_2, \ldots, X_n = i_n) \log P(X_1 = i_1, X_2 = i_2, \ldots, X_n = i_n)$$

and $\{X_1, X_2, \ldots, X_n\}$ is a sequence of length n from the source. We will talk more about the reason for the limit in Equation (2.2) later in the chapter. If each element in the sequence is independent and identically distributed (*iid*), then we can show that

$$G_n = -n \sum_{i_1=1}^{i_1=m} P(X_1 = i_1) \log P(X_1 = i_1) \tag{2.3}$$

and the equation for the entropy becomes

$$H(S) = -\sum P(X_1) \log P(X_1). \tag{2.4}$$

For most sources Equations (2.2) and (2.4) are not identical. If we need to distinguish between the two, we will call the quantity computed in (2.4) the *first-order entropy* of the source, while the quantity in (2.2) will be referred to as the *entropy* of the source.

In general, it is not possible to know the entropy for a physical source, so we have to estimate the entropy. The estimate of the entropy depends on our assumptions about the structure of the source sequence.

Consider the following sequence:

1 2 3 2 3 4 5 4 5 6 7 8 9 8 9 10

Assuming the frequency of occurrence of each number is reflected accurately in the number of times it appears in the sequence, we can estimate the probability of occurrence of each symbol as follows:

$$P(1) = P(6) = P(7) = P(10) = \tfrac{1}{16}$$
$$P(2) = P(3) = P(4) = P(5) = P(8) = P(9) = \tfrac{2}{16}.$$

Assuming the sequence is *iid*, the entropy for this sequence is the same as the first-order entropy as defined in (2.4). The entropy can then be calculated as

$$H = -\sum_{i=1}^{10} P(i) \log_2 P(i).$$

With our stated assumptions, the entropy for this source is 3.25 bits. This means that the best scheme we could find for coding this sequence could only code it at 3.25 bits/sample.

However, if we assume that there was sample-to-sample correlation between the samples and we remove the correlation by taking differences of neighboring sample values, we arrive at the *residual* sequence

$$1\ 1\ 1 - 1\ 1\ 1\ 1\ -1\ 1\ 1\ 1\ 1\ 1\ -1\ 1\ 1$$

This sequence is constructed using only two values with probabilities $P(1) = \frac{13}{16}$ and $P(-1) = \frac{3}{16}$. The entropy in this case is 0.70 bits per symbol. Of course, knowing only this sequence would not be enough for the receiver to reconstruct the original sequence. The receiver must also know the process by which this sequence was generated from the original sequence. The process depends on our assumptions about the structure of the sequence. These assumptions are called the *model* for the sequence. In this case, the model for the sequence is

$$x_n = x_{n-1} + r_n$$

where x_n is the nth element of the original sequence and r_n is the nth element of the residual sequence. This model is called a *static* model because its parameters do not change with n. A model whose parameters change or adapt with n to the changing characteristics of the data is called an *adaptive* model.

Basically, we see that knowing something about the structure of the data can help to "reduce the entropy." We have put "reduce the entropy" in quotes because the entropy of the source is a measure of the amount of information generated by the source. As long as the information generated by the source is preserved (in whatever representation), the entropy remains the same. What we are reducing is our estimate of the entropy. The "actual" structure of the data in practice is generally unknowable, but anything we can learn about the data can help us to estimate the actual source entropy. Theoretically, as seen in Equation (2.2), we accomplish this in our definition of the entropy by picking larger and larger blocks of data to calculate the probability over, letting the size of the block go to infinity.

Consider the following contrived sequence:

$$1\ 2\ 1\ 2\ 3\ 3\ 3\ 3\ 1\ 2\ 3\ 3\ 3\ 3\ 1\ 2\ 3\ 3\ 1\ 2$$

Obviously, there is some structure to this data. However, if we look at it one symbol at a time, the structure is difficult to extract. Consider the probabilities: $P(1) = P(2) = \frac{1}{4}$, and $P(3) = \frac{1}{2}$. The entropy is 1.5 bits/symbol. This particular sequence consists of 20 symbols; therefore, the total number of bits required to represent this sequence is 30. Now let's take the same sequence and look at it in blocks of two. Obviously, there are only two symbols, 1 2, and 3 3. The probabilities are $P(1\ 2) = \frac{1}{2}$, $P(3\ 3) = \frac{1}{2}$, and the entropy is 1 bit/symbol.

As there are 10 such symbols in the sequence, we need a total of 10 bits to represent the entire sequence—a reduction of a factor of three. The theory says we can always extract the structure of the data by taking larger and larger block sizes; in practice, there are limitations to this approach. To avoid these limitations, we try to obtain an accurate model for the data and code the source with respect to the model. In Section 2.3, we describe some of the models commonly used in lossless compression algorithms. But before we do that, let's make a slight detour and see a more rigorous development of the expression for average information. While the explanation is interesting, it is not really necessary for understanding much of what we will study in this book and can be skipped.

2.2.1 Derivation of Average Information ★

We start with the properties we want in our measure of average information. We will then show that requiring these properties in the information measure leads inexorably to the particular definition of average information, or entropy, that we have provided earlier.

Given a set of independent events $A_1, A_2, \ldots, A_n$ with probability $p_i = P(A_i)$, we desire the following properties in the measure of average information H:

1. We want H to be a continuous function of the probabilities p_i. That is, a small change in p_i should only cause a small change in the average information.

2. If all events are equally likely, that is, $p_i = 1/n$ for all i, then H should be a monotonically increasing function of n. The more possible outcomes there are, the more information should be contained in the occurrence of any particular outcome.

3. Suppose we divide the possible outcomes into a number of groups. We indicate the occurrence of a particular event by first indicating the group it belongs to, then indicating which particular member of the group it is. Thus, we get some information first by knowing which group the event belongs to and then we get additional information by learning which particular event (from the events in the group) has occurred. The information associated with indicating the outcome in multiple stages should not be any different than the information associated with indicating the outcome in a single stage.

   For example, suppose we have an experiment with three outcomes A_1, A_2, and A_3, with corresponding probabilities p_1, p_2, and p_3. The average information associated with this experiment is simply a function of the probabilities:

   $$H = H(p_1, p_2, p_3).$$

   Let's group the three outcomes into two groups

   $$B_1 = \{A_1\}, \quad B_2 = \{A_2, A_3\}.$$

   The probabilities of the events B_i are given by

   $$q_1 = P(B_1) = p_1, \quad q_2 = P(B_2) = p_2 + p_3.$$

If we indicate the occurrence of an event A_i by first declaring which group the event belongs to and then declaring which event occurred, the total amount of average information would be given by

$$H = H(q_1, q_2) + q_1 H\left(\frac{p_1}{q_1}\right) + q_2 H\left(\frac{p_2}{q_2}, \frac{p_3}{q_2}\right).$$

We require that the average information computed either way be the same.

In his classic paper, Shannon showed that the only way all these conditions could be satisfied was if

$$H = -K \sum p_i \log p_i$$

where K is an arbitrary positive constant. Let's review his proof as it appears in the appendix of his paper [7].

Suppose we have an experiment with $n = k^m$ equally likely outcomes. The average information $H(\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n})$ associated with this experiment is a function of n. In other words,

$$H\left(\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n}\right) = A(n).$$

We can indicate the occurrence of an event from k^m events by a series of m choices from k equally likely possibilities. For example, consider the case of $k = 2$ and $m = 3$. There are eight equally likely events; therefore, $H(\frac{1}{8}, \frac{1}{8}, \ldots, \frac{1}{8}) = A(8)$.

We can indicate occurrence of any particular event as shown in Figure 2.1. In this case, we have a sequence of three selections. Each selection is between two equally likely possibilities. Therefore,

$$\begin{aligned} H\left(\tfrac{1}{8}, \tfrac{1}{8}, \ldots, \tfrac{1}{8}\right) &= A(8) \\ &= H(\tfrac{1}{2}, \tfrac{1}{2}) + \tfrac{1}{2}\left[H(\tfrac{1}{2}, \tfrac{1}{2}) + \tfrac{1}{2}H(\tfrac{1}{2}, \tfrac{1}{2}) \right. \\ &\qquad \left. + \tfrac{1}{2}H(\tfrac{1}{2}, \tfrac{1}{2})\right] \\ &\quad + \tfrac{1}{2}\left[H(\tfrac{1}{2}, \tfrac{1}{2}) + \tfrac{1}{2}H(\tfrac{1}{2}, \tfrac{1}{2}) \right. \\ &\qquad \left. + \tfrac{1}{2}H(\tfrac{1}{2}, \tfrac{1}{2})\right] \\ &= 3H(\tfrac{1}{2}, \tfrac{1}{2}) \\ &= 3A(2). \end{aligned} \tag{2.5}$$

In other words,

$$A(8) = 3A(2).$$

(The rather odd way of writing the left-hand side of Equation (2.5) is to show how the terms correspond to the branches of the tree shown in Figure 2.1.) We can generalize this for the case of $n = k^m$ as

$$A(n) = A(k^m) = mA(k).$$

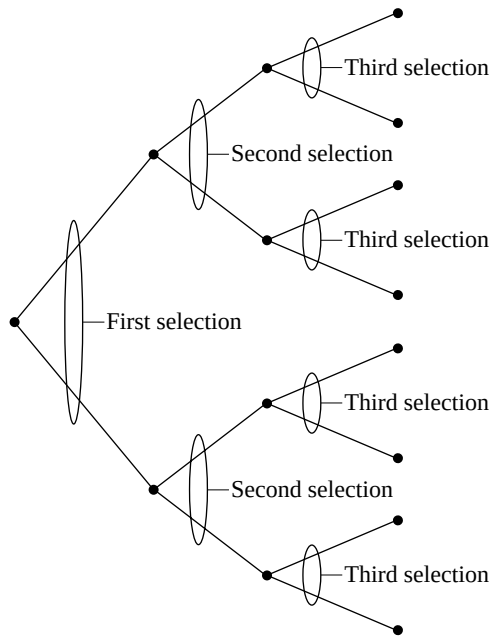

FIGURE 2. 1 A possible way of identifying the occurrence of an event.

Similarly, for j^l choices,

$$A(j^l) = lA(j).$$

We can pick l arbitrarily large (more on this later) and then choose m so that

$$k^m \leq j^l \leq k^{(m+1)}.$$

Taking logarithms of all terms, we get

$$m \log k \leq l \log j \leq (m+1) \log k.$$

Now divide through by $l \log k$ to get

$$\frac{m}{l} \leq \frac{\log j}{\log k} \leq \frac{m}{l} + \frac{1}{l}.$$

Recall that we picked l arbitrarily large. If l is arbitrarily large, then $\frac{1}{l}$ is arbitrarily small. This means that the upper and lower bounds of $\frac{\log j}{\log k}$ can be made arbitrarily close to $\frac{m}{l}$ by picking l arbitrarily large. Another way of saying this is

$$\left| \frac{m}{l} - \frac{\log j}{\log k} \right| < \epsilon$$

where ϵ can be made arbitrarily small. We will use this fact to find an expression for $A(n)$ and hence for $H(\frac{1}{n}, \ldots, \frac{1}{n})$.

To do this we use our second requirement that $H(\frac{1}{n}, \ldots, \frac{1}{n})$ be a monotonically increasing function of n. As

$$H\left(\frac{1}{n}, \ldots, \frac{1}{n}\right) = A(n),$$

this means that $A(n)$ is a monotonically increasing function of n. If

$$k^m \leq j^l \leq k^{m+1}$$

then in order to satisfy our second requirement

$$A(k^m) \leq A(j^l) \leq A(k^{m+1})$$

or

$$mA(k) \leq lA(j) \leq (m+1)A(k).$$

Dividing through by $lA(k)$, we get

$$\frac{m}{l} \leq \frac{A(j)}{A(k)} \leq \frac{m}{l} + \frac{1}{l}.$$

Using the same arguments as before, we get

$$\left|\frac{m}{l} - \frac{A(j)}{A(k)}\right| < \epsilon$$

where ϵ can be made arbitrarily small.

Now $\frac{A(j)}{A(k)}$ is at most a distance of ϵ away from $\frac{m}{l}$, and $\frac{\log j}{\log k}$ is at most a distance of ϵ away from $\frac{m}{l}$. Therefore, $\frac{A(j)}{A(k)}$ is at most a distance of 2ϵ away from $\frac{\log j}{\log k}$.

$$\left|\frac{A(j)}{A(k)} - \frac{\log j}{\log k}\right| < 2\epsilon$$

We can pick ϵ to be arbitrarily small, and j and k are arbitrary. The only way this inequality can be satisfied for arbitrarily small ϵ and arbitrary j and k is for $A(j) = K\log(j)$, where K is an arbitrary constant. In other words,

$$H = K\log(n).$$

Up to this point we have only looked at equally likely events. We now make the transition to the more general case of an experiment with outcomes that are not equally likely. We do that by considering an experiment with $\sum n_i$ equally likely outcomes that are grouped in n unequal groups of size n_i with rational probabilities (if the probabilities are not rational, we approximate them with rational probabilities and use the continuity requirement):

$$p_i = \frac{n_i}{\sum_{j=1}^{n} n_j}.$$

Given that we have $\sum n_i$ equally likely events, from the development above we have

$$H = K \log\left(\sum n_j\right). \tag{2.6}$$

If we indicate an outcome by first indicating which of the n groups it belongs to, and second indicating which member of the group it is, then by our earlier development the average information H is given by

$$\begin{aligned} H &= H(p_1, p_2, \ldots, p_n) + p_1 H\left(\frac{1}{n_1}, \ldots, \frac{1}{n_1}\right) + \cdots + p_n H\left(\frac{1}{n_n}, \ldots, \frac{1}{n_n}\right) && (2.7)\\ &= H(p_1, p_2, \ldots, p_n) + p_1 K \log n_1 + p_2 K \log n_2 + \cdots + p_n K \log n_n && (2.8)\\ &= H(p_1, p_2, \ldots, p_n) + K \sum_{i=1}^{n} p_i \log n_i. && (2.9) \end{aligned}$$

Equating the expressions in Equations (2.6) and (2.9), we obtain

$$K \log\left(\sum n_j\right) = H(p_1, p_2, \ldots, p_n) + K \sum_{i=1}^{n} p_i \log n_i$$

or

$$\begin{aligned} H(p_1, p_2, \ldots, p_n) &= K \log\left(\sum n_j\right) - K \sum_{i=1}^{n} p_i \log n_i \\ &= -K\left[\sum_{i=1}^{n} p_i \log n_i - \log\left(\sum_{j=1}^{n} n_j\right)\right] \\ &= -K\left[\sum_{i=1}^{n} p_i \log n_i - \log\left(\sum_{j=1}^{n} n_j\right) \sum_{i=1}^{n} p_i\right] && (2.10)\\ &= -K\left[\sum_{i=1}^{n} p_i \log n_i - \sum_{i=1}^{n} p_i \log\left(\sum_{j=1}^{n} n_j\right)\right] \\ &= -K \sum_{i=1}^{n} p_i \left[\log n_i - \log\left(\sum_{j=1}^{n} n_j\right)\right] \\ &= -K \sum_{i=1}^{n} p_i \log \frac{n_i}{\sum_{j=1}^{n} n_j} && (2.11)\\ &= -K \sum p_i \log p_i && (2.12) \end{aligned}$$

where, in Equation (2.10) we have used the fact that $\sum_{i=1}^{n} p_i = 1$. By convention we pick K to be 1, and we have the formula

$$H = -\sum p_i \log p_i.$$

Note that this formula is a natural outcome of the requirements we imposed in the beginning. It was not artificially forced in any way. Therein lies the beauty of information theory. Like the laws of physics, its laws are intrinsic in the nature of things. Mathematics is simply a tool to express these relationships.

2.3 Models

As we saw in Section 2.2, having a good model for the data can be useful in estimating the entropy of the source. As we will see in later chapters, good models for sources lead to more efficient compression algorithms. In general, in order to develop techniques that manipulate data using mathematical operations, we need to have a mathematical model for the data. Obviously, the better the model (i.e., the closer the model matches the aspects of reality that are of interest to us), the more likely it is that we will come up with a satisfactory technique. There are several approaches to building mathematical models.

2.3.1 Physical Models

If we know something about the physics of the data generation process, we can use that information to construct a model. For example, in speech-related applications, knowledge about the physics of speech production can be used to construct a mathematical model for the sampled speech process. Sampled speech can then be encoded using this model. We will discuss speech production models in more detail in Chapter 8.

Models for certain telemetry data can also be obtained through knowledge of the underlying process. For example, if residential electrical meter readings at hourly intervals were to be coded, knowledge about the living habits of the populace could be used to determine when electricity usage would be high and when the usage would be low. Then instead of the actual readings, the difference (residual) between the actual readings and those predicted by the model could be coded.

In general, however, the physics of data generation is simply too complicated to understand, let alone use to develop a model. Where the physics of the problem is too complicated, we can obtain a model based on empirical observation of the statistics of the data.

2.3.2 Probability Models

The simplest statistical model for the source is to assume that each letter that is generated by the source is independent of every other letter, and each occurs with the same probability. We could call this the *ignorance model*, as it would generally be useful only when we know nothing about the source. (Of course, that *really* might be true, in which case we have a rather unfortunate name for the model!) The next step up in complexity is to keep the independence assumption, but remove the equal probability assumption and assign a probability of occurrence to each letter in the alphabet. For a source that generates letters from an alphabet $\mathcal{A} = \{a_1, a_2, \ldots, a_M\}$, we can have a *probability model* $\mathcal{P} = \{P(a_1), P(a_2), \ldots, P(a_M)\}$.

Given a probability model (and the independence assumption), we can compute the entropy of the source using Equation (2.4). As we will see in the following chapters using the probability model, we can also construct some very efficient codes to represent the letters in $\mathcal{A}$. Of course, these codes are only efficient if our mathematical assumptions are in accord with reality.

If the assumption of independence does not fit with our observation of the data, we can generally find better compression schemes if we discard this assumption. When we discard

the independence assumption, we have to come up with a way to describe the dependence of elements of the data sequence on each other.

2.3.3 Markov Models

One of the most popular ways of representing dependence in the data is through the use of Markov models, named after the Russian mathematician Andrei Andrevich Markov (1856–1922). For models used in lossless compression, we use a specific type of Markov process called a *discrete time Markov chain*. Let $\{x_n\}$ be a sequence of observations. This sequence is said to follow a kth-order Markov model if

$$P(x_n|x_{n-1}, \ldots, x_{n-k}) = P(x_n|x_{n-1}, \ldots, x_{n-k}, \ldots). \tag{2.13}$$

In other words, knowledge of the past k symbols is equivalent to the knowledge of the entire past history of the process. The values taken on by the set $\{x_{n-1}, \ldots, x_{n-k}\}$ are called the *states* of the process. If the size of the source alphabet is l, then the number of states is l^k. The most commonly used Markov model is the first-order Markov model, for which

$$P(x_n|x_{n-1}) = P(x_n|x_{n-1}, x_{n-2}, x_{n-3}, \ldots). \tag{2.14}$$

Equations (2.13) and (2.14) indicate the existence of dependence between samples. However, they do not describe the form of the dependence. We can develop different first-order Markov models depending on our assumption about the form of the dependence between samples.

If we assumed that the dependence was introduced in a linear manner, we could view the data sequence as the output of a linear filter driven by white noise. The output of such a filter can be given by the difference equation

$$x_n = \rho x_{n-1} + \epsilon_n \tag{2.15}$$

where ϵ_n is a white noise process. This model is often used when developing coding algorithms for speech and images.

The use of the Markov model does not require the assumption of linearity. For example, consider a binary image. The image has only two types of pixels, white pixels and black pixels. We know that the appearance of a white pixel as the next observation depends, to some extent, on whether the current pixel is white or black. Therefore, we can model the pixel process as a discrete time Markov chain. Define two states S_w and S_b (S_w would correspond to the case where the current pixel is a white pixel, and S_b corresponds to the case where the current pixel is a black pixel). We define the transition probabilities $P(w/b)$ and $P(b/w)$, and the probability of being in each state $P(S_w)$ and $P(S_b)$. The Markov model can then be represented by the state diagram shown in Figure 2.2.

The entropy of a finite state process with states S_i is simply the average value of the entropy at each state:

$$H = \sum_{i=1}^{M} P(S_i)H(S_i). \tag{2.16}$$

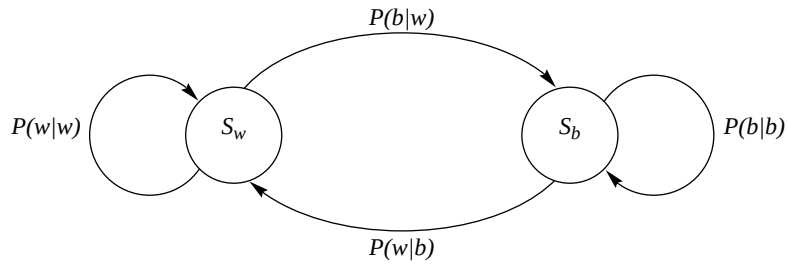

FIGURE 2. 2 A two-state Markov model for binary images.

For our particular example of a binary image

$$H(S_w) = -P(b/w)\log P(b/w) - P(w/w)\log P(w/w)$$

where $P(w/w) = 1 - P(b/w)$. $H(S_b)$ can be calculated in a similar manner.

Example 2.3.1: Markov model

To see the effect of modeling on the estimate of entropy, let us calculate the entropy for a binary image, first using a simple probability model and then using the finite state model described above. Let us assume the following values for the various probabilities:

$$P(S_w) = 30/31 \quad P(S_b) = 1/31$$

$$P(w|w) = 0.99 \quad P(b|w) = 0.01 \quad P(b|b) = 0.7 \quad P(w|b) = 0.3.$$

Then the entropy using a probability model and the *iid* assumption is

$$H = -0.8\log 0.8 - 0.2\log 0.2 = 0.206 \text{ bits.}$$

Now using the Markov model

$$H(S_b) = -0.3\log 0.3 - 0.7\log 0.7 = 0.881 \text{ bits}$$

and

$$H(S_w) = -0.01\log 0.01 - 0.99\log 0.99 = 0.081 \text{ bits}$$

which, using Equation (2.16), results in an entropy for the Markov model of 0.107 bits, about a half of the entropy obtained using the *iid* assumption. ♦

Markov Models in Text Compression

As expected, Markov models are particularly useful in text compression, where the probability of the next letter is heavily influenced by the preceding letters. In fact, the use of Markov models for written English appears in the original work of Shannon [7]. In current text compression literature, the kth-order Markov models are more widely known

as *finite context models*, with the word *context* being used for what we have earlier defined as state.

Consider the word *preceding*. Suppose we have already processed *precedin* and are going to encode the next letter. If we take no account of the context and treat each letter as a surprise, the probability of the letter *g* occurring is relatively low. If we use a first-order Markov model or single-letter context (that is, we look at the probability model given *n*), we can see that the probability of *g* would increase substantially. As we increase the context size (go from *n* to *in* to *din* and so on), the probability of the alphabet becomes more and more skewed, which results in lower entropy.

Shannon used a second-order model for English text consisting of the 26 letters and one space to obtain an entropy of 3.1 bits/letter [8]. Using a model where the output symbols were words rather than letters brought down the entropy to 2.4 bits/letter. Shannon then used predictions generated by people (rather than statistical models) to estimate the upper and lower bounds on the entropy of the second order model. For the case where the subjects knew the 100 previous letters, he estimated these bounds to be 1.3 and 0.6 bits/letter, respectively.

The longer the context, the better its predictive value. However, if we were to store the probability model with respect to all contexts of a given length, the number of contexts would grow exponentially with the length of context. Furthermore, given that the source imposes some structure on its output, many of these contexts may correspond to strings that would never occur in practice. Consider a context model of order four (the context is determined by the last four symbols). If we take an alphabet size of 95, the possible number of contexts is 95^4—more than 81 million!

This problem is further exacerbated by the fact that different realizations of the source output may vary considerably in terms of repeating patterns. Therefore, context modeling in text compression schemes tends to be an adaptive strategy in which the probabilities for different symbols in the different contexts are updated as they are encountered. However, this means that we will often encounter symbols that have not been encountered before for any of the given contexts (this is known as the *zero frequency problem*). The larger the context, the more often this will happen. This problem could be resolved by sending a code to indicate that the following symbol was being encountered for the first time, followed by a prearranged code for that symbol. This would significantly increase the length of the code for the symbol on its first occurrence (in the given context). However, if this situation did not occur too often, the overhead associated with such occurrences would be small compared to the total number of bits used to encode the output of the source. Unfortunately, in context-based encoding, the zero frequency problem is encountered often enough for overhead to be a problem, especially for longer contexts. Solutions to this problem are presented by the *ppm* (prediction with partial match) algorithm and its variants (described in detail in Chapter 6).

Briefly, the *ppm* algorithms first attempt to find if the symbol to be encoded has a nonzero probability with respect to the maximum context length. If this is so, the symbol is encoded and transmitted. If not, an escape symbol is transmitted, the context size is reduced by one, and the process is repeated. This procedure is repeated until a context is found with respect to which the symbol has a nonzero probability. To guarantee that this process converges, a null context is always included with respect to which all symbols have equal probability. Initially, only the shorter contexts are likely to be used. However, as more and more of the source output is processed, the longer contexts, which offer better prediction,

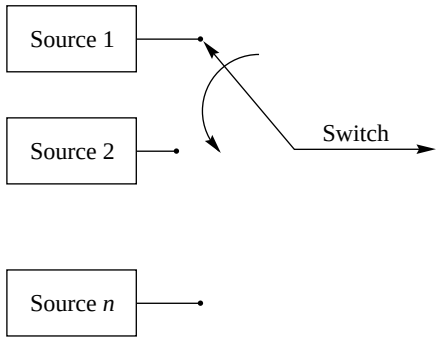

FIGURE 2.3 A composite source.

will be used more often. The probability of the escape symbol can be computed in a number of different ways leading to different implementations [1].

The use of Markov models in text compression is a rich and active area of research. We describe some of these approaches in Chapter 6 (for more details, see [1]).

2.3.4 Composite Source Model

In many applications, it is not easy to use a single model to describe the source. In such cases, we can define a *composite source*, which can be viewed as a combination or composition of several sources, with only one source being *active* at any given time. A composite source can be represented as a number of individual sources $\mathcal{S}_i$, each with its own model $\mathcal{M}_i$, and a switch that selects a source $\mathcal{S}_i$ with probability P_i (as shown in Figure 2.3). This is an exceptionally rich model and can be used to describe some very complicated processes. We will describe this model in more detail when we need it.

2.4 Coding

When we talk about *coding* in this chapter (and through most of this book), we mean the assignment of binary sequences to elements of an alphabet. The set of binary sequences is called a *code*, and the individual members of the set are called *codewords*. An *alphabet* is a collection of symbols called *letters*. For example, the alphabet used in writing most books consists of the 26 lowercase letters, 26 uppercase letters, and a variety of punctuation marks. In the terminology used in this book, a comma is a letter. The ASCII code for the letter *a* is 1000011, the letter *A* is coded as 1000001, and the letter "," is coded as 0011010. Notice that the ASCII code uses the same number of bits to represent each symbol. Such a code is called a *fixed-length code*. If we want to reduce the number of bits required to represent different messages, we need to use a different number of bits to represent different symbols. If we use fewer bits to represent symbols that occur more often, on the average we would use fewer bits per symbol. The average number of bits per symbol is often called the *rate* of the code. The idea of using fewer bits to represent symbols that occur more often is the

same idea that is used in Morse code: the codewords for letters that occur more frequently are shorter than for letters that occur less frequently. For example, the codeword for E is $\cdot$, while the codeword for Z is $- - \cdot \cdot$ [9].

2.4.1 Uniquely Decodable Codes

The average length of the code is not the only important point in designing a "good" code. Consider the following example adapted from [10]. Suppose our source alphabet consists of four letters a_1, a_2, a_3, and a_4, with probabilities $P(a_1) = \frac{1}{2}$, $P(a_2) = \frac{1}{4}$, and $P(a_3) = P(a_4) = \frac{1}{8}$. The entropy for this source is 1.75 bits/symbol. Consider the codes for this source in Table 2.1.

The average length l for for each code is given by

$$l = \sum_{i=1}^{4} P(a_i) n(a_i)$$

where $n(a_i)$ is the number of bits in the codeword for letter a_i and the average length is given in bits/symbol. Based on the average length, Code 1 appears to be the best code. However, to be useful, a code should have the ability to transfer information in an unambiguous manner. This is obviously not the case with Code 1. Both a_1 and a_2 have been assigned the codeword 0. When a 0 is received, there is no way to know whether an a_1 was transmitted or an a_2. We would like each symbol to be assigned a *unique* codeword.

At first glance Code 2 does not seem to have the problem of ambiguity; each symbol is assigned a distinct codeword. However, suppose we want to encode the sequence $a_2\ a_1\ a_1$. Using Code 2, we would encode this with the binary string 100. However, when the string 100 is received at the decoder, there are several ways in which the decoder can decode this string. The string 100 can be decoded as $a_2\ a_1\ a_1$, or as $a_2\ a_3$. This means that once a sequence is encoded with Code 2, the original sequence cannot be recovered with certainty. In general, this is not a desirable property for a code. We would like *unique decodability* from the code; that is, any given sequence of codewords can be decoded in one, and only one, way.

We have already seen that Code 1 and Code 2 are not uniquely decodable. How about Code 3? Notice that the first three codewords all end in a 0. In fact, a 0 always denotes the termination of a codeword. The final codeword contains no 0s and is 3 bits long. Because all other codewords have fewer than three 1s and terminate in a 0, the only way we can get three 1s in a row is as a code for a_4. The decoding rule is simple. Accumulate bits until you get a 0 or until you have three 1s. There is no ambiguity in this rule, and it is reasonably

TABLE 2.1 Four different codes for a four-letter alphabet.

Letters	Probability	Code 1	Code 2	Code 3	Code 4
a_1	0.5	0	0	0	0
a_2	0.25	0	1	10	01
a_3	0.125	1	00	110	011
a_4	0.125	10	11	111	0111
Average length		1.125	1.25	1.75	1.875

easy to see that this code is uniquely decodable. With Code 4 we have an even simpler condition. Each codeword starts with a 0, and the only time we see a 0 is in the beginning of a codeword. Therefore, the decoding rule is accumulate bits until you see a 0. The bit before the 0 is the last bit of the previous codeword.

There is a slight difference between Code 3 and Code 4. In the case of Code 3, the decoder knows the moment a code is complete. In Code 4, we have to wait till the beginning of the next codeword before we know that the current codeword is complete. Because of this property, Code 3 is called an *instantaneous* code. Although Code 4 is not an instantaneous code, it is almost that.

While this property of instantaneous or near-instantaneous decoding is a nice property to have, it is not a requirement for unique decodability. Consider the code shown in Table 2.2. Let's decode the string 011111111111111111. In this string, the first codeword is either 0 corresponding to a_1 or 01 corresponding to a_2. We cannot tell which one until we have decoded the whole string. Starting with the assumption that the first codeword corresponds to a_1, the next eight pairs of bits are decoded as a_3. However, after decoding eight a_3s, we are left with a single (dangling) 1 that does not correspond to any codeword. On the other hand, if we assume the first codeword corresponds to a_2, we can decode the next 16 bits as a sequence of eight a_3s, and we do not have any bits left over. The string can be uniquely decoded. In fact, Code 5, while it is certainly not instantaneous, is uniquely decodable.

We have been looking at small codes with four letters or less. Even with these, it is not immediately evident whether the code is uniquely decodable or not. In deciding whether larger codes are uniquely decodable, a systematic procedure would be useful. Actually, we should include a caveat with that last statement. Later in this chapter we will include a class of variable-length codes that are always uniquely decodable, so a test for unique decodability may not be that necessary. You might wish to skip the following discussion for now, and come back to it when you find it necessary.

Before we describe the procedure for deciding whether a code is uniquely decodable, let's take another look at our last example. We found that we had an incorrect decoding because we were left with a binary string (1) that was not a codeword. If this had not happened, we would have had two valid decodings. For example, consider the code shown in Table 2.3. Let's

TABLE 2.2 Code 5.

Letter	Codeword
a_1	0
a_2	01
a_3	11

TABLE 2.3 Code 6.

Letter	Codeword
a_1	0
a_2	01
a_3	10

encode the sequence a_1 followed by eight a_3s using this code. The coded sequence is 01010101010101010. The first bit is the codeword for a_1. However, we can also decode it as the first bit of the codeword for a_2. If we use this (incorrect) decoding, we decode the next seven pairs of bits as the codewords for a_2. After decoding seven a_2s, we are left with a single 0 that we decode as a_1. Thus, the incorrect decoding is also a valid decoding, and this code is not uniquely decodable.

A Test for Unique Decodability ★

In the previous examples, in the case of the uniquely decodable code, the binary string left over after we had gone through an incorrect decoding was not a codeword. In the case of the code that was not uniquely decodable, in the incorrect decoding what was left was a valid codeword. Based on whether the dangling suffix is a codeword or not, we get the following test [11, 12].

We start with some definitions. Suppose we have two binary codewords a and b, where a is k bits long, b is n bits long, and $k < n$. If the first k bits of b are identical to a, then a is called a *prefix* of b. The last $n-k$ bits of b are called the *dangling suffix* [11]. For example, if $a = 010$ and $b = 01011$, then a is a prefix of b and the dangling suffix is 11.

Construct a list of all the codewords. Examine all pairs of codewords to see if any codeword is a prefix of another codeword. Whenever you find such a pair, add the dangling suffix to the list unless you have added the same dangling suffix to the list in a previous iteration. Now repeat the procedure using this larger list. Continue in this fashion until one of the following two things happens:

1. You get a dangling suffix that is a codeword.
2. There are no more unique dangling suffixes.

If you get the first outcome, the code is not uniquely decodable. However, if you get the second outcome, the code is uniquely decodable.

Let's see how this procedure works with a couple of examples.

Example 2.4.1:

Consider Code 5. First list the codewords

$$\{0, 01, 11\}$$

The codeword 0 is a prefix for the codeword 01. The dangling suffix is 1. There are no other pairs for which one element of the pair is the prefix of the other. Let us augment the codeword list with the dangling suffix.

$$\{0, 01, 11, 1\}$$

Comparing the elements of this list, we find 0 is a prefix of 01 with a dangling suffix of 1. But we have already included 1 in our list. Also, 1 is a prefix of 11. This gives us a dangling suffix of 1, which is already in the list. There are no other pairs that would generate a dangling suffix, so we cannot augment the list any further. Therefore, Code 5 is uniquely decodable. ♦

Example 2.4.2:

Consider Code 6. First list the codewords

$$\{0, 01, 10\}$$

The codeword 0 is a prefix for the codeword 01. The dangling suffix is 1. There are no other pairs for which one element of the pair is the prefix of the other. Augmenting the codeword list with 1, we obtain the list

$$\{0, 01, 10, 1\}$$

In this list, 1 is a prefix for 10. The dangling suffix for this pair is 0, which is the codeword for a_1. Therefore, Code 6 is not uniquely decodable. ♦

2.4.2 Prefix Codes

The test for unique decodability requires examining the dangling suffixes initially generated by codeword pairs in which one codeword is the prefix of the other. If the dangling suffix is itself a codeword, then the code is not uniquely decodable. One type of code in which we will never face the possibility of a dangling suffix being a codeword is a code in which no codeword is a prefix of the other. In this case, the set of dangling suffixes is the null set, and we do not have to worry about finding a dangling suffix that is identical to a codeword. A code in which no codeword is a prefix to another codeword is called a *prefix code*. A simple way to check if a code is a prefix code is to draw the rooted binary tree corresponding to the code. Draw a tree that starts from a single node (the *root node*) and has a maximum of two possible branches at each node. One of these branches corresponds to a 1 and the other branch corresponds to a 0. In this book, we will adopt the convention that when we draw a tree with the root node at the top, the left branch corresponds to a 0 and the right branch corresponds to a 1. Using this convention, we can draw the binary tree for Code 2, Code 3, and Code 4 as shown in Figure 2.4.

Note that apart from the root node, the trees have two kinds of nodes—nodes that give rise to other nodes and nodes that do not. The first kind of nodes are called *internal nodes*, and the second kind are called *external nodes* or *leaves*. In a prefix code, the codewords are only associated with the external nodes. A code that is not a prefix code, such as Code 4, will have codewords associated with internal nodes. The code for any symbol can be obtained

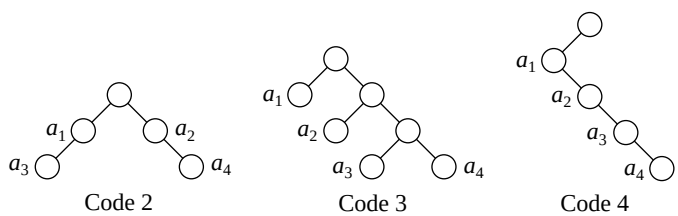

FIGURE 2.4 Binary trees for three different codes.

by traversing the tree from the root to the external node corresponding to that symbol. Each branch on the way contributes a bit to the codeword: a 0 for each left branch and a 1 for each right branch.

It is nice to have a class of codes, whose members are so clearly uniquely decodable. However, are we losing something if we restrict ourselves to prefix codes? Could it be that if we do not restrict ourselves to prefix codes, we can find shorter codes? Fortunately for us the answer is no. For any nonprefix uniquely decodable code, we can always find a prefix code with the same codeword lengths. We prove this in the next section.

2.4.3 The Kraft-McMillan Inequality ★

The particular result we look at in this section consists of two parts. The first part provides a necessary condition on the codeword lengths of uniquely decodable codes. The second part shows that we can always find a prefix code that satisfies this necessary condition. Therefore, if we have a uniquely decodable code that is not a prefix code, we can always find a prefix code with the same codeword lengths.

Theorem *Let $\mathcal{C}$ be a code with N codewords with lengths $l_1, l_2, \ldots, l_N$. If $\mathcal{C}$ is uniquely decodable, then*

$$K(\mathcal{C}) = \sum_{i=1}^{N} 2^{-l_i} \leq 1.$$

This inequality is known as the Kraft-McMillan inequality.

Proof The proof works by looking at the nth power of $K(\mathcal{C})$. If $K(\mathcal{C})$ is greater than one, then $K(\mathcal{C})^n$ should grow exponentially with n. If it does not grow exponentially with n, then this is proof that $\sum_{i=1}^{N} 2^{-l_i} \leq 1$.

Let n be an arbitrary integer. Then

$$\left[\sum_{i=1}^{N} 2^{-l_i}\right]^n = \left(\sum_{i_1=1}^{N} 2^{-l_{i_1}}\right)\left(\sum_{i_2=1}^{N} 2^{-l_{i_2}}\right)\cdots\left(\sum_{i_n=1}^{N} 2^{-l_{i_n}}\right) \tag{2.17}$$

$$= \sum_{i_1=1}^{N}\sum_{i_2=1}^{N}\cdots\sum_{i_n=1}^{N} 2^{-(l_{i_1}+l_{i_2}+\cdots+l_{i_n})}. \tag{2.18}$$

The exponent $l_{i_1} + l_{i_2} + \cdots + l_{i_n}$ is simply the length of n codewords from the code $\mathcal{C}$. The smallest value that this exponent can take is greater than or equal to n, which would be the case if all codewords were 1 bit long. If

$$l = \max\{l_1, l_2, \ldots, l_N\}$$

then the largest value that the exponent can take is less than or equal to nl. Therefore, we can write this summation as

$$K(\mathcal{C})^n = \sum_{k=n}^{nl} A_k 2^{-k}$$

where A_k is the number of combinations of n codewords that have a combined length of k. Let's take a look at the size of this coefficient. The number of possible distinct binary sequences of length k is 2^k. If this code is uniquely decodable, then each sequence can represent one and only one sequence of codewords. Therefore, the number of possible combinations of codewords whose combined length is k cannot be greater than 2^k. In other words,

$$A_k \leq 2^k.$$

This means that

$$K(\mathcal{C})^n = \sum_{k=n}^{nl} A_k 2^{-k} \leq \sum_{k=n}^{nl} 2^k 2^{-k} = nl - n + 1. \tag{2.19}$$

But if $K(\mathcal{C})$ is greater than one, it will grow exponentially with n, while $n(l-1)+1$ can only grow linearly. So if $K(\mathcal{C})$ is greater than one, we can always find an n large enough that the inequality (2.19) is violated. Therefore, for a uniquely decodable code $\mathcal{C}$, $K(\mathcal{C})$ is less than or equal to one. □

This part of the Kraft-McMillan inequality provides a necessary condition for uniquely decodable codes. That is, if a code is uniquely decodable, the codeword lengths have to satisfy the inequality. The second part of this result is that if we have a set of codeword lengths that satisfy the inequality, we can always find a prefix code with those codeword lengths. The proof of this assertion presented here is adapted from [6].

Theorem *Given a set of integers $l_1, l_2, \ldots, l_N$ that satisfy the inequality*

$$\sum_{i=1}^{N} 2^{-l_i} \leq 1$$

we can always find a prefix code with codeword lengths $l_1, l_2, \ldots, l_N$.

Proof We will prove this assertion by developing a procedure for constructing a prefix code with codeword lengths $l_1, l_2, \ldots, l_N$ that satisfy the given inequality.

Without loss of generality, we can assume that

$$l_1 \leq l_2 \leq \cdots \leq l_N.$$

Define a sequence of numbers $w_1, w_2, \ldots, w_N$ as follows:

$$\begin{aligned} w_1 &= 0 \\ w_j &= \sum_{i=1}^{j-1} 2^{l_j - l_i} \qquad j > 1. \end{aligned}$$

The binary representation of w_j for $j > 1$ would take up $\lceil \log_2(w_j + 1) \rceil$ bits. We will use this binary representation to construct a prefix code. We first note that the number of bits in the binary representation of w_j is less than or equal to l_j. This is obviously true for w_1. For $j > 1$,

$$\begin{aligned}
\log_2(w_j + 1) &= \log_2\left[\sum_{i=1}^{j-1} 2^{l_j - l_i} + 1\right] \\
&= \log_2\left[2^{l_j} \sum_{i=1}^{j-1} 2^{-l_i} + 2^{-l_j}\right] \\
&= l_j + \log_2\left[\sum_{i=1}^{j} 2^{-l_i}\right] \\
&\leq l_j.
\end{aligned}$$

The last inequality results from the hypothesis of the theorem that $\sum_{i=1}^{N} 2^{-l_i} \leq 1$, which implies that $\sum_{i=1}^{j} 2^{-l_i} \leq 1$. As the logarithm of a number less than one is negative, $l_j + \log_2\left[\sum_{i=1}^{j} 2^{-l_i}\right]$ has to be less than l_j.

Using the binary representation of w_j, we can devise a binary code in the following manner: If $\lceil \log_2(w_j + 1) \rceil = l_j$, then the jth codeword c_j is the binary representation of w_j. If $\lceil \log_2(w_j + 1) \rceil < l_j$, then c_j is the binary representation of w_j, with $l_j - \lceil \log_2(w_j + 1) \rceil$ zeros appended to the right. This is certainly a code, but is it a prefix code? If we can show that the code $\mathcal{C} = \{c_1, c_2, \ldots, c_N\}$ is a prefix code, then we will have proved the theorem by construction.

Suppose that our claim is not true. Then for some $j < k$, c_j is a prefix of c_k. This means that the l_j most significant bits of w_k form the binary representation of w_j. Therefore if we right-shift the binary representation of w_k by $l_k - l_j$ bits, we should get the binary representation for w_j. We can write this as

$$w_j = \left\lfloor \frac{w_k}{2^{l_k - l_j}} \right\rfloor.$$

However,

$$w_k = \sum_{i=1}^{k-1} 2^{l_k - l_i}.$$

Therefore,

$$\begin{aligned}
\frac{w_k}{2^{l_k - l_j}} &= \sum_{i=0}^{k-1} 2^{l_j - l_i} \\
&= w_j + \sum_{i=j}^{k-1} 2^{l_j - l_i} \\
&= w_j + 2^0 + \sum_{i=j+1}^{k-1} 2^{l_j - l_i} \\
&\geq w_j + 1. \qquad (2.20)
\end{aligned}$$

That is, the smallest value for $\frac{w_k}{2^{l_k - l_j}}$ is $w_j + 1$. This contradicts the requirement for c_j being the prefix of c_k. Therefore, c_j cannot be the prefix for c_k. As j and k were arbitrary, this means that no codeword is a prefix of another codeword, and the code $\mathcal{C}$ is a prefix code. □

Therefore, if we have a uniquely decodable code, the codeword lengths have to satisfy the Kraft-McMillan inequality. And, given codeword lengths that satisfy the Kraft-McMillan inequality, we can always find a prefix code with those codeword lengths. Thus, by restricting ourselves to prefix codes, we are not in danger of overlooking nonprefix uniquely decodable codes that have a shorter average length.

2.5 Algorithmic Information Theory

The theory of information described in the previous sections is intuitively satisfying and has useful applications. However, when dealing with real world data, it does have some theoretical difficulties. Suppose you were given the task of developing a compression scheme for use with a specific set of documentations. We can view the entire set as a single long string. You could develop models for the data. Based on these models you could calculate probabilities using the relative frequency approach. These probabilities could then be used to obtain an estimate of the entropy and thus an estimate of the amount of compression available. All is well except for a fly in the "ointment." The string you have been given is fixed. There is nothing probabilistic about it. There is no abstract source that will generate different sets of documentation at different times. So how can we talk about the entropies without pretending that reality is somehow different from what it actually is? Unfortunately, it is not clear that we can. Our definition of entropy requires the existence of an abstract source. Our estimate of the entropy is still useful. It will give us a very good idea of how much compression we can get. So, practically speaking, information theory comes through. However, theoretically it seems there is some pretending involved. Algorithmic information theory is a different way of looking at information that has not been as useful in practice (and therefore we will not be looking at it a whole lot) but it gets around this theoretical problem. At the heart of algorithmic information theory is a measure called *Kolmogorov complexity*. This measure, while it bears the name of one person, was actually discovered independently by three people: R. Solomonoff, who was exploring machine learning; the Russian mathematician A.N. Kolmogorov; and G. Chaitin, who was in high school when he came up with this idea.

The Kolmogorov complexity $K(x)$ of a sequence x is the size of the program needed to generate x. In this size we include all inputs that might be needed by the program. We do not specify the programming language because it is always possible to translate a program in one language to a program in another language at fixed cost. If x was a sequence of all ones, a highly compressible sequence, the program would simply be a print statement in a loop. On the other extreme, if x were a random sequence with no structure then the only program that could generate it would contain the sequence itself. The size of the program, would be slightly larger than the sequence itself. Thus, there is a clear correspondence between the size of the smallest program that can generate a sequence and the amount of compression that can be obtained. Kolmogorov complexity seems to be the

ideal measure to use in data compression. The problem is we do not know of any systematic way of computing or closely approximating Kolmogorov complexity. Clearly, any program that can generate a particular sequence is an upper bound for the Kolmogorov complexity of the sequence. However, we have no way of determining a lower bound. Thus, while the notion of Kolmogorov complexity is more satisfying theoretically than the notion of entropy when compressing sequences, in practice it is not yet as helpful. However, given the active interest in these ideas it is quite possible that they will result in more practical applications.

2.6 Minimum Description Length Principle

One of the more practical offshoots of Kolmogorov complexity is the minimum description length (MDL) principle. The first discoverer of Kolmogorov complexity, Ray Solomonoff, viewed the concept of a program that would generate a sequence as a way of modeling the data. Independent from Solomonoff but inspired nonetheless by the ideas of Kolmogorov complexity, Jorma Risannen in 1978 [13] developed the modeling approach commonly known as MDL.

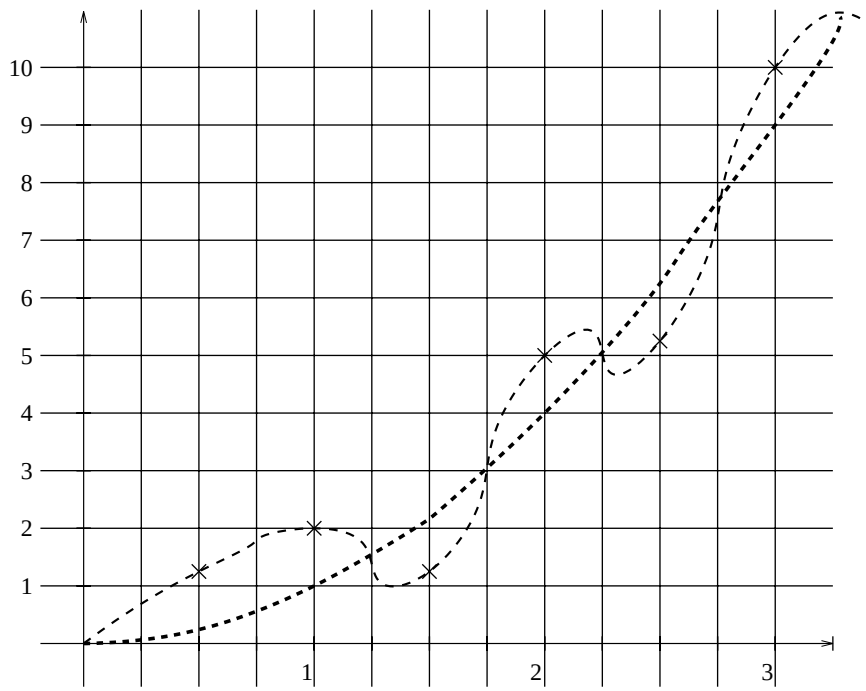

FIGURE 2. 5 **An example to illustrate the MDL principle.**

Let M_j be a model from a set of models $\mathcal{M}$ that attempt to characterize the structure in a sequence x. Let D_{M_j} be the number of bits required to describe the model M_j. For example, if the set of models $\mathcal{M}$ can be represented by a (possibly variable) number of coefficients, then the description of M_j would include the number of coefficients and the value of each coefficient. Let $R_{M_j}(x)$ be the number of bits required to represent x with respect to the model M_j. The minimum description length would be given by

$$\min_j (D_{M_j} + R_{M_j}(x))$$

Consider the example shown as Figure 2. 5, where the X's represent data values. Suppose the set of models $\mathcal{M}$ is the set of kth order polynomials. We have also sketched two polynomials that could be used to model the data. Clearly, the higher-order polynomial does a much "better" job of modeling the data in the sense that the model exactly describes the data. To describe the higher order polynomial, we need to specify the value of each coefficient. The coefficients have to be exact if the polynomial is to exactly model the data requiring a large number of bits. The quadratic model, on the other hand, does not fit any of the data values. However, its description is very simple and the data values are either $+1$ or -1 away from the quadratic. So we could exactly represent the data by sending the coefficients of the quadratic (1, 0) and 1 bit per data value to indicate whether each data value is $+1$ or -1 away from the quadratic. In this case, from a compression point of view, using the worse model actually gives better compression.

2.7 Summary

In this chapter we learned some of the basic definitions of information theory. This was a rather brief visit, and we will revisit the subject in Chapter 8. However, the coverage in this chapter will be sufficient to take us through the next four chapters. The concepts introduced in this chapter allow us to estimate the number of bits we need to represent the output of a source given the probability model for the source. The process of assigning a binary representation to the output of a source is called coding. We have introduced the concepts of unique decodability and prefix codes, which we will use in the next two chapters when we describe various coding algorithms. We also looked, rather briefly, at different approaches to modeling. If we need to understand a model in more depth later in the book, we will devote more attention to it at that time. However, for the most part, the coverage of modeling in this chapter will be sufficient to understand methods described in the next four chapters.

Further Reading

1. A very readable book on information theory and its applications in a number of fields is *Symbols, Signals, and Noise—The Nature and Process of Communications*, by J.R. Pierce [14].
2. Another good introductory source for the material in this chapter is Chapter 6 of *Coding and Information Theory*, by R.W. Hamming [9].

3. Various models for text compression are described very nicely and in more detail in *Text Compression*, by T.C. Bell, J.G. Cleary, and I.H. Witten [1].

4. For a more thorough and detailed account of information theory, the following books are especially recommended (the first two are my personal favorites): *Information Theory*, by R.B. Ash [15]; *Transmission of Information*, by R.M. Fano [16]; *Information Theory and Reliable Communication*, by R.G. Gallagher [11]; *Entropy and Information Theory*, by R.M. Gray [17]; *Elements of Information Theory*, by T.M. Cover and J.A. Thomas [3]; and *The Theory of Information and Coding*, by R.J. McEliece [6].

5. Kolmogorov complexity is addressed in detail in *An Introduction to Kolmogorov Complexity and Its Applications,* by M. Li and P. Vitanyi [18].

6. A very readable overview of Kolmogorov complexity in the context of lossless compression can be found in the chapter *Complexity Measures*, by S.R. Tate [19].

7. Various aspects of the minimum description length principle are discussed in *Advances in Minimum Description Length* edited by P. Grunwald, I.J. Myung, and M.A. Pitt [20]. Included in this book is a very nice introduction to the minimum description length principle by Peter Grunwald [21].

2.8 Projects and Problems

1. Suppose X is a random variable that takes on values from an M-letter alphabet. Show that $0 \le H(X) \le \log_2 M$.

2. Show that for the case where the elements of an observed sequence are *iid*, the entropy is equal to the first-order entropy.

3. Given an alphabet $\mathcal{A} = \{a_1, a_2, a_3, a_4\}$, find the first-order entropy in the following cases:

   (a) $P(a_1) = P(a_2) = P(a_3) = P(a_4) = \frac{1}{4}$.

   (b) $P(a_1) = \frac{1}{2}$, $P(a_2) = \frac{1}{4}$, $P(a_3) = P(a_4) = \frac{1}{8}$.

   (c) $P(a_1) = 0.505$, $P(a_2) = \frac{1}{4}$, $P(a_3) = \frac{1}{8}$, and $P(a_4) = 0.12$.

4. Suppose we have a source with a probability model $P = \{p_0, p_1, \ldots, p_m\}$ and entropy H_P. Suppose we have another source with probability model $Q = \{q_0, q_1, \ldots, q_m\}$ and entropy H_Q, where

$$q_i = p_i \qquad i = 0, 1, \ldots, j-2, j+1, \ldots, m$$

   and

$$q_j = q_{j-1} = \frac{p_j + p_{j-1}}{2}.$$

   How is H_Q related to H_P (greater, equal, or less)? Prove your answer.

5. There are several image and speech files among the accompanying data sets.

(a) Write a program to compute the first-order entropy of some of the image and speech files.

(b) Pick one of the image files and compute its second-order entropy. Comment on the difference between the first- and second-order entropies.

(c) Compute the entropy of the differences between neighboring pixels for the image you used in part (b). Comment on what you discover.

6. Conduct an experiment to see how well a model can describe a source.

(a) Write a program that randomly selects letters from the 26-letter alphabet $\{a, b, \ldots, z\}$ and forms four-letter words. Form 100 such words and see how many of these words make sense.

(b) Among the accompanying data sets is a file called `4letter.words`, which contains a list of four-letter words. Using this file, obtain a probability model for the alphabet. Now repeat part (a) generating the words using the probability model. To pick letters according to a probability model, construct the cumulative density function (*cdf*) $F_X(x)$ (see Appendix A for the definition of *cdf*). Using a uniform pseudorandom number generator to generate a value r, where $0 \leq r < 1$, pick the letter x_k if $F_X(x_k - 1) \leq r < F_X(x_k)$. Compare your results with those of part (a).

(c) Repeat (b) using a single-letter context.

(d) Repeat (b) using a two-letter context.

7. Determine whether the following codes are uniquely decodable:

(a) $\{0, 01, 11, 111\}$

(b) $\{0, 01, 110, 111\}$

(c) $\{0, 10, 110, 111\}$

(d) $\{1, 10, 110, 111\}$

8. Using a text file compute the probabilities of each letter p_i.

(a) Assume that we need a codeword of length $\lceil \log_2 \frac{1}{p_i} \rceil$ to encode the letter i. Determine the number of bits needed to encode the file.

(b) Compute the conditional probabilities $P(i/j)$ of a letter i given that the previous letter is j. Assume that we need $\lceil \log_2 \frac{1}{P(i/j)} \rceil$ to represent a letter i that follows a letter j. Determine the number of bits needed to encode the file.

3

Huffman Coding

3.1 Overview

In this chapter we describe a very popular coding algorithm called the Huffman coding algorithm. We first present a procedure for building Huffman codes when the probability model for the source is known, then a procedure for building codes when the source statistics are unknown. We also describe a few techniques for code design that are in some sense similar to the Huffman coding approach. Finally, we give some examples of using the Huffman code for image compression, audio compression, and text compression.

3.2 The Huffman Coding Algorithm

This technique was developed by David Huffman as part of a class assignment; the class was the first ever in the area of information theory and was taught by Robert Fano at MIT [22]. The codes generated using this technique or procedure are called *Huffman codes*. These codes are prefix codes and are optimum for a given model (set of probabilities).

The Huffman procedure is based on two observations regarding optimum prefix codes.

1. In an optimum code, symbols that occur more frequently (have a higher probability of occurrence) will have shorter codewords than symbols that occur less frequently.
2. In an optimum code, the two symbols that occur least frequently will have the same length.

It is easy to see that the first observation is correct. If symbols that occur more often had codewords that were longer than the codewords for symbols that occurred less often, the average number of bits per symbol would be larger than if the conditions were reversed. Therefore, a code that assigns longer codewords to symbols that occur more frequently cannot be optimum.

To see why the second observation holds true, consider the following situation. Suppose an optimum code $\mathcal{C}$ exists in which the two codewords corresponding to the two least probable symbols do not have the same length. Suppose the longer codeword is k bits longer than the shorter codeword. Because this is a prefix code, the shorter codeword cannot be a prefix of the longer codeword. This means that even if we drop the last k bits of the longer codeword, the two codewords would still be distinct. As these codewords correspond to the least probable symbols in the alphabet, no other codeword can be longer than these codewords; therefore, there is no danger that the shortened codeword would become the prefix of some other codeword. Furthermore, by dropping these k bits we obtain a new code that has a shorter average length than $\mathcal{C}$. But this violates our initial contention that $\mathcal{C}$ is an optimal code. Therefore, for an optimal code the second observation also holds true.

The Huffman procedure is obtained by adding a simple requirement to these two observations. This requirement is that the codewords corresponding to the two lowest probability symbols differ only in the last bit. That is, if γ and δ are the two least probable symbols in an alphabet, if the codeword for γ was $\mathbf{m} * 0$, the codeword for δ would be $\mathbf{m} * 1$. Here $\mathbf{m}$ is a string of 1s and 0s, and $*$ denotes concatenation.

This requirement does not violate our two observations and leads to a very simple encoding procedure. We describe this procedure with the help of the following example.

Example 3.2.1: Design of a Huffman code

Let us design a Huffman code for a source that puts out letters from an alphabet $\mathcal{A} = \{a_1, a_2, a_3, a_4, a_5\}$ with $P(a_1) = P(a_3) = 0.2$, $P(a_2) = 0.4$, and $P(a_4) = P(a_5) = 0.1$. The entropy for this source is 2.122 bits/symbol. To design the Huffman code, we first sort the letters in a descending probability order as shown in Table 3.1. Here $c(a_i)$ denotes the codeword for a_i.

TABLE 3.1 The initial five-letter alphabet.

Letter	Probability	Codeword
a_2	0.4	$c(a_2)$
a_1	0.2	$c(a_1)$
a_3	0.2	$c(a_3)$
a_4	0.1	$c(a_4)$
a_5	0.1	$c(a_5)$

The two symbols with the lowest probability are a_4 and a_5. Therefore, we can assign their codewords as

$$c(a_4) = \alpha_1 * 0$$
$$c(a_5) = \alpha_1 * 1$$

where α_1 is a binary string, and $*$ denotes concatenation.

We now define a new alphabet A' with a four-letter alphabet a_1, a_2, a_3, a'_4, where a'_4 is composed of a_4 and a_5 and has a probability $P(a'_4) = P(a_4) + P(a_5) = 0.2$. We sort this new alphabet in descending order to obtain Table 3.2.

TABLE 3.2 The reduced four-letter alphabet.

Letter	Probability	Codeword
a_2	0.4	$c(a_2)$
a_1	0.2	$c(a_1)$
a_3	0.2	$c(a_3)$
a'_4	0.2	α_1

In this alphabet, a_3 and a'_4 are the two letters at the bottom of the sorted list. We assign their codewords as

$$c(a_3) = \alpha_2 * 0$$
$$c(a'_4) = \alpha_2 * 1$$

but $c(a'_4) = \alpha_1$. Therefore,

$$\alpha_1 = \alpha_2 * 1$$

which means that

$$c(a_4) = \alpha_2 * 10$$
$$c(a_5) = \alpha_2 * 11.$$

At this stage, we again define a new alphabet A'' that consists of three letters a_1, a_2, a'_3, where a'_3 is composed of a_3 and a'_4 and has a probability $P(a'_3) = P(a_3) + P(a'_4) = 0.4$. We sort this new alphabet in descending order to obtain Table 3.3.

TABLE 3.3 The reduced three-letter alphabet.

Letter	Probability	Codeword
a_2	0.4	$c(a_2)$
a'_3	0.4	α_2
a_1	0.2	$c(a_1)$

In this case, the two least probable symbols are a_1 and a'_3. Therefore,

$$c(a'_3) = \alpha_3 * 0$$
$$c(a_1) = \alpha_3 * 1.$$

But $c(a_3') = \alpha_2$. Therefore,

$$\alpha_2 = \alpha_3 * 0$$

which means that

$$\begin{aligned} c(a_3) &= \alpha_3 * 00 \\ c(a_4) &= \alpha_3 * 010 \\ c(a_5) &= \alpha_3 * 011. \end{aligned}$$

Again we define a new alphabet, this time with only two letters a_3'', a_2. Here a_3'' is composed of the letters a_3' and a_1 and has probability $P(a_3'') = P(a_3') + P(a_1) = 0.6$. We now have Table 3.4.

TABLE 3.4 The reduced two-letter alphabet.

Letter	Probability	Codeword
a_3''	0.6	α_3
a_2	0.4	$c(a_2)$

As we have only two letters, the codeword assignment is straightforward:

$$\begin{aligned} c(a_3'') &= 0 \\ c(a_2) &= 1 \end{aligned}$$

which means that $\alpha_3 = 0$, which in turn means that

$$\begin{aligned} c(a_1) &= 01 \\ c(a_3) &= 000 \\ c(a_4) &= 0010 \\ c(a_5) &= 0011 \end{aligned}$$

TABLE 3.5 Huffman code for the original five-letter alphabet.

Letter	Probability	Codeword
a_2	0.4	1
a_1	0.2	01
a_3	0.2	000
a_4	0.1	0010
a_5	0.1	0011

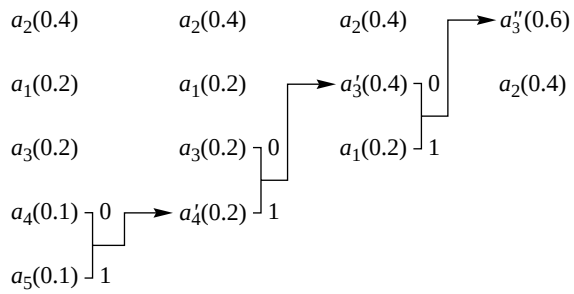

FIGURE 3. 1 **The Huffman encoding procedure. The symbol probabilities are listed in parentheses.**

and the Huffman code is given by Table 3.5. The procedure can be summarized as shown in Figure 3.1. ♦

The average length for this code is

$$l = .4 \times 1 + .2 \times 2 + .2 \times 3 + .1 \times 4 + .1 \times 4 = 2.2 \text{ bits/symbol.}$$

A measure of the efficiency of this code is its *redundancy*—the difference between the entropy and the average length. In this case, the redundancy is 0.078 bits/symbol. The redundancy is zero when the probabilities are negative powers of two.

An alternative way of building a Huffman code is to use the fact that the Huffman code, by virtue of being a prefix code, can be represented as a binary tree in which the external nodes or leaves correspond to the symbols. The Huffman code for any symbol can be obtained by traversing the tree from the root node to the leaf corresponding to the symbol, adding a 0 to the codeword every time the traversal takes us over an upper branch and a 1 every time the traversal takes us over a lower branch.

We build the binary tree starting at the leaf nodes. We know that the codewords for the two symbols with smallest probabilities are identical except for the last bit. This means that the traversal from the root to the leaves corresponding to these two symbols must be the same except for the last step. This in turn means that the leaves corresponding to the two symbols with the lowest probabilities are offspring of the same node. Once we have connected the leaves corresponding to the symbols with the lowest probabilities to a single node, we treat this node as a symbol of a reduced alphabet. The probability of this symbol is the sum of the probabilities of its offspring. We can now sort the nodes corresponding to the reduced alphabet and apply the same rule to generate a parent node for the nodes corresponding to the two symbols in the reduced alphabet with lowest probabilities. Continuing in this manner, we end up with a single node, which is the root node. To obtain the code for each symbol, we traverse the tree from the root to each leaf node, assigning a 0 to the upper branch and a 1 to the lower branch. This procedure as applied to the alphabet of Example 3.2.1 is shown in Figure 3.2. Notice the similarity between Figures 3.1 and 3.2. This is not surprising, as they are a result of viewing the same procedure in two different ways.

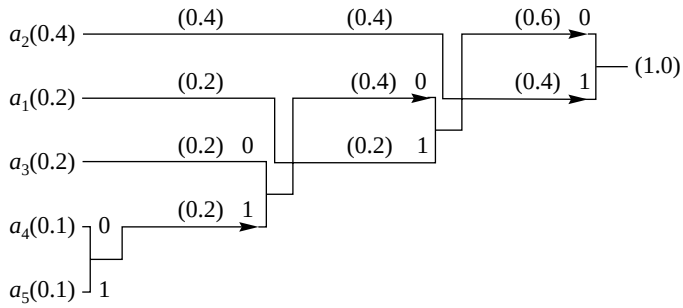

FIGURE 3.2 Building the binary Huffman tree.

3.2.1 Minimum Variance Huffman Codes

By performing the sorting procedure in a slightly different manner, we could have found a different Huffman code. In the first re-sort, we could place a_4' higher in the list, as shown in Table 3.6.

Now combine a_1 and a_3 into a_1', which has a probability of 0.4. Sorting the alphabet a_2, a_4', a_1' and putting a_1' as far up the list as possible, we get Table 3.7. Finally, by combining a_2 and a_4' and re-sorting, we get Table 3.8. If we go through the unbundling procedure, we get the codewords in Table 3.9. The procedure is summarized in Figure 3.3. The average length of the code is

$$l = .4 \times 2 + .2 \times 2 + .2 \times 2 + .1 \times 3 + .1 \times 3 = 2.2 \text{ bits/symbol.}$$

The two codes are identical in terms of their redundancy. However, the variance of the length of the codewords is significantly different. This can be clearly seen from Figure 3.4.

TABLE 3.6 Reduced four-letter alphabet.

Letter	Probability	Codeword
a_2	0.4	$c(a_2)$
a_4'	0.2	α_1
a_1	0.2	$c(a_1)$
a_3	0.2	$c(a_3)$

TABLE 3.7 Reduced three-letter alphabet.

Letter	Probability	Codeword
a_1'	0.4	α_2
a_2	0.4	$c(a_2)$
a_4'	0.2	α_1

TABLE 3.8 Reduced two-letter alphabet.

Letter	Probability	Codeword
a_2'	0.6	α_3
a_1'	0.4	α_2

TABLE 3.9 Minimum variance Huffman code.

Letter	Probability	Codeword
a_1	0.2	10
a_2	0.4	00
a_3	0.2	11
a_4	0.1	010
a_5	0.1	011

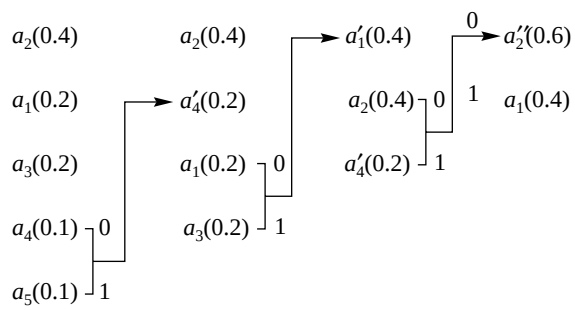

FIGURE 3.3 The minimum variance Huffman encoding procedure.

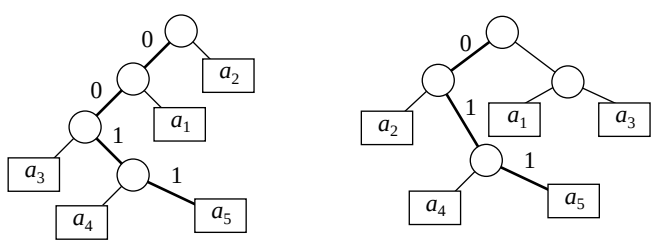

FIGURE 3.4 Two Huffman trees corresponding to the same probabilities.

Remember that in many applications, although you might be using a variable-length code, the available transmission rate is generally fixed. For example, if we were going to transmit symbols from the alphabet we have been using at 10,000 symbols per second, we might ask for transmission capacity of 22,000 bits per second. This means that during each second the channel expects to receive 22,000 bits, no more and no less. As the bit generation rate will

vary around 22,000 bits per second, the output of the source coder is generally fed into a buffer. The purpose of the buffer is to smooth out the variations in the bit generation rate. However, the buffer has to be of finite size, and the greater the variance in the codewords, the more difficult the buffer design problem becomes. Suppose that the source we are discussing generates a string of a_4s and a_5s for several seconds. If we are using the first code, this means that we will be generating bits at a rate of 40,000 bits per second. For each second, the buffer has to store 18,000 bits. On the other hand, if we use the second code, we would be generating 30,000 bits per second, and the buffer would have to store 8000 bits for every second this condition persisted. If we have a string of a_2s instead of a string of a_4s and a_5s, the first code would result in the generation of 10,000 bits per second. Remember that the channel will still be expecting 22,000 bits every second, so somehow we will have to make up a deficit of 12,000 bits per second. The same situation using the second code would lead to a deficit of 2000 bits per second. Thus, it seems reasonable to elect to use the second code instead of the first. To obtain the Huffman code with minimum variance, we always put the combined letter as high in the list as possible.

3.2.2 Optimality of Huffman Codes ★

The optimality of Huffman codes can be proven rather simply by first writing down the necessary conditions that an optimal code has to satisfy and then showing that satisfying these conditions necessarily leads to designing a Huffman code. The proof we present here is based on the proof shown in [16] and is obtained for the binary case (for a more general proof, see [16]).

The necessary conditions for an optimal variable-length binary code are as follows:

- **Condition 1:** Given any two letters a_j and a_k, if $P[a_j] \geq P[a_k]$, then $l_j \leq l_k$, where l_j is the number of bits in the codeword for a_j.
- **Condition 2:** The two least probable letters have codewords with the same maximum length l_m.

We have provided the justification for these two conditions in the opening sections of this chapter.

- **Condition 3:** In the tree corresponding to the optimum code, there must be two branches stemming from each intermediate node.

If there were any intermediate node with only one branch coming from that node, we could remove it without affecting the decipherability of the code while reducing its average length.

- **Condition 4:** Suppose we change an intermediate node into a leaf node by combining all the leaves descending from it into a composite word of a reduced alphabet. Then, if the original tree was optimal for the original alphabet, the reduced tree is optimal for the reduced alphabet.

If this condition were not satisfied, we could find a code with smaller average code length for the reduced alphabet and then simply expand the composite word again to get a new

code tree that would have a shorter average length than our original "optimum" tree. This would contradict our statement about the optimality of the original tree.

In order to satisfy conditions 1, 2, and 3, the two least probable letters would have to be assigned codewords of maximum length l_m. Furthermore, the leaves corresponding to these letters arise from the same intermediate node. This is the same as saying that the codewords for these letters are identical except for the last bit. Consider the common prefix as the codeword for the composite letter of a reduced alphabet. Since the code for the reduced alphabet needs to be optimum for the code of the original alphabet to be optimum, we follow the same procedure again. To satisfy the necessary conditions, the procedure needs to be iterated until we have a reduced alphabet of size one. But this is exactly the Huffman procedure. Therefore, the necessary conditions above, which are all satisfied by the Huffman procedure, are also sufficient conditions.

3.2.3 Length of Huffman Codes ★

We have said that the Huffman coding procedure generates an optimum code, but we have not said what the average length of an optimum code is. The length of any code will depend on a number of things, including the size of the alphabet and the probabilities of individual letters. In this section we will show that the optimal code for a source $\mathcal{S}$, hence the Huffman code for the source $\mathcal{S}$, has an average code length $\bar{l}$ bounded below by the entropy and bounded above by the entropy plus 1 bit. In other words,

$$H(\mathcal{S}) \leq \bar{l} < H(\mathcal{S}) + 1. \tag{3.1}$$

In order for us to do this, we will need to use the Kraft-McMillan inequality introduced in Chapter 2. Recall that the first part of this result, due to McMillan, states that if we have a uniquely decodable code $\mathcal{C}$ with K codewords of length $\{l_i\}_{i=1}^{K}$, then the following inequality holds:

$$\sum_{i=1}^{K} 2^{-l_i} \leq 1. \tag{3.2}$$

Example 3.2.2:

Examining the code generated in Example 3.2.1 (Table 3.5), the lengths of the codewords are $\{1, 2, 3, 4, 4\}$. Substituting these values into the left-hand side of Equation (3.2), we get

$$2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} + 2^{-4} = 1$$

which satisfies the Kraft-McMillan inequality.

If we use the minimum variance code (Table 3.9), the lengths of the codewords are $\{2, 2, 2, 3, 3\}$. Substituting these values into the left-hand side of Equation (3.2), we get

$$2^{-2} + 2^{-2} + 2^{-2} + 2^{-3} + 2^{-3} = 1$$

which again satisfies the inequality. ♦

The second part of this result, due to Kraft, states that if we have a sequence of positive integers $\{l_i\}_{i=1}^K$, which satisfies (3.2), then there exists a uniquely decodable code whose codeword lengths are given by the sequence $\{l_i\}_{i=1}^K$.

Using this result, we will now show the following:

1. The average codeword length $\bar{l}$ of an optimal code for a source $\mathcal{S}$ is greater than or equal to $H(\mathcal{S})$.
2. The average codeword length $\bar{l}$ of an optimal code for a source $\mathcal{S}$ is strictly less than $H(\mathcal{S})+1$.

For a source $\mathcal{S}$ with alphabet $\mathcal{A} = \{a_1, a_2, \ldots a_K\}$, and probability model $\{P(a_1), P(a_2), \ldots, P(a_K)\}$, the average codeword length is given by

$$\bar{l} = \sum_{i=1}^{K} P(a_i) l_i.$$

Therefore, we can write the difference between the entropy of the source $H(\mathcal{S})$ and the average length as

$$\begin{aligned}
H(\mathcal{S}) - \bar{l} &= -\sum_{i=1}^{K} P(a_i)\log_2 P(a_i) - \sum_{i=1}^{K} P(a_i) l_i \\
&= \sum_{i=1}^{K} P(a_i)\left(\log_2\left[\frac{1}{P(a_i)}\right] - l_i\right) \\
&= \sum_{i=1}^{K} P(a_i)\left(\log_2\left[\frac{1}{P(a_i)}\right] - \log_2[2^{l_i}]\right) \\
&= \sum_{i=1}^{K} P(a_i)\log_2\left[\frac{2^{-l_i}}{P(a_i)}\right] \\
&\le \log_2\left[\sum_{i=1}^{K} 2^{-l_i}\right].
\end{aligned}$$

The last inequality is obtained using Jensen's inequality, which states that if $f(x)$ is a concave (convex cap, convex $\cap$) function, then $E[f(X)] \le f(E[X])$. The log function is a concave function.

As the code is an optimal code $\sum_{i=1}^{K} 2^{-l_i} \le 1$, therefore

$$H(\mathcal{S}) - \bar{l} \le 0. \tag{3.3}$$

We will prove the upper bound by showing that there exists a uniquely decodable code with average codeword length $H(\mathcal{S})+1$. Therefore, if we have an optimal code, this code must have an average length that is less than or equal to $H(\mathcal{S})+1$.

Given a source, alphabet, and probability model as before, define

$$l_i = \left\lceil \log_2 \frac{1}{P(a_i)} \right\rceil$$

where $\lceil x \rceil$ is the smallest integer greater than or equal to x. For example, $\lceil 3.3 \rceil = 4$ and $\lceil 5 \rceil = 5$. Therefore,

$$\lceil x \rceil = x + \epsilon \qquad \text{where } 0 \leq \epsilon < 1.$$

Therefore,

$$\log_2 \frac{1}{P(a_i)} \leq l_i < \log_2 \frac{1}{P(a_i)} + 1. \tag{3.4}$$

From the left inequality of (3.4) we can see that

$$2^{-l_i} \leq P(a_i).$$

Therefore,

$$\sum_{i=1}^{K} 2^{-l_i} \leq \sum_{i=1}^{K} P(a_i) = 1$$

and by the Kraft-McMillan inequality there exists a uniquely decodable code with codeword lengths $\{l_i\}$. The average length of this code can be upper-bounded by using the right inequality of (3.4):

$$\bar{l} = \sum_{i=1}^{K} P(a_i) l_i < \sum_{i=1}^{K} P(a_i) \left[\log_2 \frac{1}{P(a_i)} + 1 \right]$$

or

$$\bar{l} < H(\mathcal{S}) + 1.$$

We can see from the way the upper bound was derived that this is a rather loose upper bound. In fact, it can be shown that if $p_{\max}$ is the largest probability in the probability model, then for $p_{\max} \geq 0.5$, the upper bound for the Huffman code is $H(\mathcal{S}) + p_{\max}$, while for $p_{\max} < 0.5$, the upper bound is $H(\mathcal{S}) + p_{\max} + 0.086$. Obviously, this is a much tighter bound than the one we derived above. The derivation of this bound takes some time (see [23] for details).

3.2.4 Extended Huffman Codes ★

In applications where the alphabet size is large, $p_{\max}$ is generally quite small, and the amount of deviation from the entropy, especially in terms of a percentage of the rate, is quite small. However, in cases where the alphabet is small and the probability of occurrence of the different letters is skewed, the value of $p_{\max}$ can be quite large and the Huffman code can become rather inefficient when compared to the entropy.

Example 3.2.3:

Consider a source that puts out *iid* letters from the alphabet $\mathcal{A} = \{a_1, a_2, a_3\}$ with the probability model $P(a_1) = 0.8$, $P(a_2) = 0.02$, and $P(a_3) = 0.18$. The entropy for this source is 0.816 bits/symbol. A Huffman code for this source is shown in Table 3.10.

TABLE 3.10 Huffman code for the alphabet $\mathcal{A}$.

Letter	Codeword
a_1	0
a_2	11
a_3	10

The average length for this code is 1.2 bits/symbol. The difference between the average code length and the entropy, or the redundancy, for this code is 0.384 bits/symbol, which is 47% of the entropy. This means that to code this sequence we would need 47% more bits than the minimum required. ♦

We can sometimes reduce the coding rate by blocking more than one symbol together. To see how this can happen, consider a source S that emits a sequence of letters from an alphabet $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$. Each element of the sequence is generated independently of the other elements in the sequence. The entropy for this source is given by

$$H(S) = -\sum_{i=1}^{m} P(a_i) \log_2 P(a_i).$$

We know that we can generate a Huffman code for this source with rate R such that

$$H(S) \leq R < H(S) + 1. \tag{3.5}$$

We have used the looser bound here; the same argument can be made with the tighter bound. Notice that we have used "rate R" to denote the number of bits per symbol. This is a standard convention in the data compression literature. However, in the communication literature, the word "rate" often refers to the number of bits per second.

Suppose we now encode the sequence by generating one codeword for every n symbols. As there are m^n combinations of n symbols, we will need m^n codewords in our Huffman code. We could generate this code by viewing the m^n symbols as letters of an *extended alphabet*

$$\mathcal{A}^{(n)} = \{\overbrace{a_1 a_1 \ldots a_1}^{n \text{ times}}, a_1 a_1 \ldots a_2, \ldots, a_1 a_1 \ldots a_m, a_1 a_1 \ldots a_2 a_1, \ldots, a_m a_m \ldots a_m\}$$

from a source $S^{(n)}$. Let us denote the rate for the new source as $R^{(n)}$. Then we know that

$$H(S^{(n)}) \leq R^{(n)} < H(S^{(n)}) + 1. \tag{3.6}$$

$R^{(n)}$ is the number of bits required to code n symbols. Therefore, the number of bits required per symbol, R, is given by

$$R = \frac{1}{n} R^{(n)}.$$

The number of bits per symbol can be bounded as

$$\frac{H(S^{(n)})}{n} \le R < \frac{H(S^{(n)})}{n} + \frac{1}{n}.$$

In order to compare this to (3.5), and see the advantage we get from encoding symbols in blocks instead of one at a time, we need to express $H(S^{(n)})$ in terms of $H(S)$. This turns out to be a relatively easy (although somewhat messy) thing to do.

$$\begin{aligned}
H(S^{(n)}) &= -\sum_{i_1=1}^{m}\sum_{i_2=1}^{m}\cdots\sum_{i_n=1}^{m} P(a_{i_1}, a_{i_2}, \ldots a_{i_n}) \log[P(a_{i_1}, a_{i_2}, \ldots a_{i_n})] \\
&= -\sum_{i_1=1}^{m}\sum_{i_2=1}^{m}\cdots\sum_{i_n=1}^{m} P(a_{i_1})P(a_{i_2})\ldots P(a_{i_n}) \log[P(a_{i_1})P(a_{i_2})\ldots P(a_{i_n})] \\
&= -\sum_{i_1=1}^{m}\sum_{i_2=1}^{m}\cdots\sum_{i_n=1}^{m} P(a_{i_1})P(a_{i_2})\ldots P(a_{i_n}) \sum_{j=1}^{n} \log[P(a_{i_j})] \\
&= -\sum_{i_1=1}^{m} P(a_{i_1}) \log[P(a_{i_1})] \left\{ \sum_{i_2=1}^{m}\cdots\sum_{i_n=1}^{m} P(a_{i_2})\ldots P(a_{i_n}) \right\} \\
&\quad -\sum_{i_2=1}^{m} P(a_{i_2}) \log[P(a_{i_2})] \left\{ \sum_{i_1=1}^{m}\sum_{i_3=1}^{m}\cdots\sum_{i_n=1}^{m} P(a_{i_1})P(a_{i_3})\ldots P(a_{i_n}) \right\} \\
&\quad \vdots \\
&\quad -\sum_{i_n=1}^{m} P(a_{i_n}) \log[P(a_{i_n})] \left\{ \sum_{i_1=1}^{m}\sum_{i_2=1}^{m}\cdots\sum_{i_{n-1}=1}^{m} P(a_{i_1})P(a_{i_2})\ldots P(a_{i_{n-1}}) \right\}
\end{aligned}$$

The $n-1$ summations in braces in each term sum to one. Therefore,

$$\begin{aligned}
H(S^{(n)}) &= -\sum_{i_1=1}^{m} P(a_{i_1}) \log[P(a_{i_1})] - \sum_{i_2=1}^{m} P(a_{i_2}) \log[P(a_{i_2})] - \cdots - \sum_{i_n=1}^{m} P(a_{i_n}) \log[P(a_{i_n})] \\
&= nH(S)
\end{aligned}$$

and we can write (3.6) as

$$H(S) \le R \le H(S) + \frac{1}{n}. \tag{3.7}$$

Comparing this to (3.5), we can see that by encoding the output of the source in longer blocks of symbols we are *guaranteed* a rate closer to the entropy. Note that all we are talking about here is a bound or guarantee about the rate. As we have seen in the previous chapter, there are a number of situations in which we can achieve a rate *equal* to the entropy with a block length of one!

Example 3.2.4:

For the source described in the previous example, instead of generating a codeword for every symbol, we will generate a codeword for every *two* symbols. If we look at the source sequence two at a time, the number of possible symbol pairs, or size of the extended alphabet, is $3^2 = 9$. The extended alphabet, probability model, and Huffman code for this example are shown in Table 3.11.

TABLE 3.11 The extended alphabet and corresponding Huffman code.

Letter	Probability	Code
a_1a_1	0.64	0
a_1a_2	0.016	10101
a_1a_3	0.144	11
a_2a_1	0.016	101000
a_2a_2	0.0004	10100101
a_2a_3	0.0036	1010011
a_3a_1	0.1440	100
a_3a_2	0.0036	10100100
a_3a_3	0.0324	1011

The average codeword length for this extended code is 1.7228 bits/symbol. However, each symbol in the extended alphabet corresponds to two symbols from the original alphabet. Therefore, in terms of the original alphabet, the average codeword length is $1.7228/2 = 0.8614$ bits/symbol. This redundancy is about 0.045 bits/symbol, which is only about 5.5% of the entropy. ♦

We see that by coding blocks of symbols together we can reduce the redundancy of Huffman codes. In the previous example, two symbols were blocked together to obtain a rate reasonably close to the entropy. Blocking two symbols together means the alphabet size goes from m to m^2, where m was the size of the initial alphabet. In this case, m was three, so the size of the extended alphabet was nine. This size is not an excessive burden for most applications. However, if the probabilities of the symbols were more unbalanced, then it would require blocking many more symbols together before the redundancy lowered to acceptable levels. As we block more and more symbols together, the size of the alphabet grows exponentially, and the Huffman coding scheme becomes impractical. Under these conditions, we need to look at techniques other than Huffman coding. One approach that is very useful in these conditions is *arithmetic coding*. We will discuss this technique in some detail in the next chapter.

3.3 Nonbinary Huffman Codes ★

The binary Huffman coding procedure can be easily extended to the nonbinary case where the code elements come from an m-ary alphabet, and m is not equal to two. Recall that we obtained the Huffman algorithm based on the observations that in an optimum binary prefix code

1. symbols that occur more frequently (have a higher probability of occurrence) will have shorter codewords than symbols that occur less frequently, and
2. the two symbols that occur least frequently will have the same length,

and the requirement that the two symbols with the lowest probability differ only in the last position.

We can obtain a nonbinary Huffman code in almost exactly the same way. The obvious thing to do would be to modify the second observation to read: "The m symbols that occur least frequently will have the same length," and also modify the additional requirement to read "The m symbols with the lowest probability differ only in the last position."

However, we run into a small problem with this approach. Consider the design of a ternary Huffman code for a source with a six-letter alphabet. Using the rules described above, we would first combine the three letters with the lowest probability into a composite letter. This would give us a reduced alphabet with four letters. However, combining the three letters with lowest probability from this alphabet would result in a further reduced alphabet consisting of only two letters. We have three values to assign and only two letters. Instead of combining three letters at the beginning, we could have combined two letters. This would result in a reduced alphabet of size five. If we combined three letters from this alphabet, we would end up with a final reduced alphabet size of three. Finally, we could combine two letters in the second step, which would again result in a final reduced alphabet of size three. Which alternative should we choose?

Recall that the symbols with lowest probability will have the longest codeword. Furthermore, all the symbols that we combine together into a composite symbol will have codewords of the same length. This means that all letters we combine together at the very first stage will have codewords that have the same length, and these codewords will be the longest of all the codewords. This being the case, if at some stage we are allowed to combine less than m symbols, the logical place to do this would be in the very first stage.

In the general case of an m-ary code and an M-letter alphabet, how many letters should we combine in the first phase? Let m' be the number of letters that are combined in the first phase. Then m' is the number between two and m, which is equal to M modulo $(m-1)$.

Example 3.3.1:

Generate a ternary Huffman code for a source with a six-letter alphabet and a probability model $P(a_1) = P(a_3) = P(a_4) = 0.2$, $P(a_5) = 0.25$, $P(a_6) = 0.1$, and $P(a_2) = 0.05$. In this case $m = 3$, therefore m' is either 2 or 3.

$$6 \pmod 2 = 0, \qquad 2 \pmod 2 = 0, \qquad 3 \pmod 2 = 1$$

Since $6 \pmod 2 = 2 \pmod 2$, $m' = 2$. Sorting the symbols in probability order results in Table 3.12.

TABLE 3.12 Sorted six-letter alphabet.

Letter	Probability	Codeword
a_5	0.25	$c(a_5)$
a_1	0.20	$c(a_1)$
a_3	0.20	$c(a_3)$
a_4	0.20	$c(a_4)$
a_6	0.10	$c(a_6)$
a_2	0.05	$c(a_2)$

As m' is 2, we can assign the codewords of the two symbols with lowest probability as

$$c(a_6) = \alpha_1 * 0$$
$$c(a_2) = \alpha_1 * 1$$

where α_1 is a ternary string and * denotes concatenation. The reduced alphabet is shown in Table 3.13.

TABLE 3.13 Reduced five-letter alphabet.

Letter	Probability	Codeword
a_5	0.25	$c(a_5)$
a_1	0.20	$c(a_1)$
a_3	0.20	$c(a_3)$
a_4	0.20	$c(a_4)$
a_6'	0.15	α_1

Now we combine the three letters with the lowest probability into a composite letter a_3' and assign their codewords as

$$c(a_3) = \alpha_2 * 0$$
$$c(a_4) = \alpha_2 * 1$$
$$c(a_6') = \alpha_2 * 2.$$

But $c(a_6') = \alpha_1$. Therefore,

$$\alpha_1 = \alpha_2 * 2$$

which means that

$$c(a_6) = \alpha_2 * 20$$
$$c(a_2) = \alpha_2 * 21.$$

Sorting the reduced alphabet, we have Table 3.14. Thus, $\alpha_2 = 0$, $c(a_5) = 1$, and $c(a_1) = 2$. Substituting for α_2, we get the codeword assignments in Table 3.15.

TABLE 3.14 Reduced three-letter alphabet.

Letter	Probability	Codeword
a_3'	0.45	α_2
a_5	0.25	$c(a_5)$
a_1	0.20	$c(a_1)$

TABLE 3.15 Ternary code for six-letter alphabet.

Letter	Probability	Codeword
a_1	0.20	2
a_2	0.05	021
a_3	0.20	00
a_4	0.20	01
a_5	0.25	1
a_6	0.10	020

The tree corresponding to this code is shown in Figure 3.5. Notice that at the lowest level of the tree we have only two codewords. If we had combined three letters at the first step, and combined two letters at a later step, the lowest level would have contained three codewords and a longer average code length would result (see Problem 7).

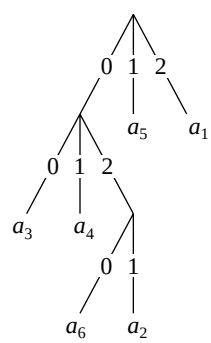

FIGURE 3.5 Code tree for the nonbinary Huffman code. ♦

3.4 Adaptive Huffman Coding

Huffman coding requires knowledge of the probabilities of the source sequence. If this knowledge is not available, Huffman coding becomes a two-pass procedure: the statistics are collected in the first pass, and the source is encoded in the second pass. In order to convert this algorithm into a one-pass procedure, Faller [24] and Gallagher [23] independently developed adaptive algorithms to construct the Huffman code based on the statistics of the symbols already encountered. These were later improved by Knuth [25] and Vitter [26].

Theoretically, if we wanted to encode the $(k+1)$-th symbol using the statistics of the first k symbols, we could recompute the code using the Huffman coding procedure each time a symbol is transmitted. However, this would not be a very practical approach due to the large amount of computation involved—hence, the adaptive Huffman coding procedures.

The Huffman code can be described in terms of a binary tree similar to the ones shown in Figure 3.4. The squares denote the external nodes or leaves and correspond to the symbols in the source alphabet. The codeword for a symbol can be obtained by traversing the tree from the root to the leaf corresponding to the symbol, where 0 corresponds to a left branch and 1 corresponds to a right branch. In order to describe how the adaptive Huffman code works, we add two other parameters to the binary tree: the *weight* of each leaf, which is written as a number inside the node, and a *node number*. The weight of each external node is simply the number of times the symbol corresponding to the leaf has been encountered. The weight of each internal node is the sum of the weights of its offspring. The node number y_i is a unique number assigned to each internal and external node. If we have an alphabet of size n, then the $2n-1$ internal and external nodes can be numbered as $y_1, \ldots, y_{2n-1}$ such that if x_j is the weight of node y_j, we have $x_1 \leq x_2 \leq \cdots \leq x_{2n-1}$. Furthermore, the nodes y_{2j-1} and y_{2j} are offspring of the same parent node, or siblings, for $1 \leq j < n$, and the node number for the parent node is greater than y_{2j-1} and y_{2j}. These last two characteristics are called the *sibling property*, and any tree that possesses this property is a Huffman tree [23].

In the adaptive Huffman coding procedure, neither transmitter nor receiver knows anything about the statistics of the source sequence at the start of transmission. The tree at both the transmitter and the receiver consists of a single node that corresponds to all symbols not yet transmitted (NYT) and has a weight of zero. As transmission progresses, nodes corresponding to symbols transmitted will be added to the tree, and the tree is reconfigured using an update procedure. Before the beginning of transmission, a fixed code for each symbol is agreed upon between transmitter and receiver. A simple (short) code is as follows:

If the source has an alphabet $(a_1, a_2, \ldots, a_m)$ of size m, then pick e and r such that $m = 2^e + r$ and $0 \leq r < 2^e$. The letter a_k is encoded as the $(e+1)$-bit binary representation of $k-1$, if $1 \leq k \leq 2r$; else, a_k is encoded as the e-bit binary representation of $k-r-1$. For example, suppose $m = 26$, then $e = 4$, and $r = 10$. The symbol a_1 is encoded as 00000, the symbol a_2 is encoded as 00001, and the symbol a_{22} is encoded as 1011.

When a symbol is encountered for the first time, the code for the NYT node is transmitted, followed by the fixed code for the symbol. A node for the symbol is then created, and the symbol is taken out of the NYT list.

Both transmitter and receiver start with the same tree structure. The updating procedure used by both transmitter and receiver is identical. Therefore, the encoding and decoding processes remain synchronized.

3.4.1 Update Procedure

The update procedure requires that the nodes be in a fixed order. This ordering is preserved by numbering the nodes. The largest node number is given to the root of the tree, and the smallest number is assigned to the NYT node. The numbers from the NYT node to the root of the tree are assigned in increasing order from left to right, and from lower level to upper level. The set of nodes with the same weight makes up a *block*. Figure 3.6 is a flowchart of the updating procedure.

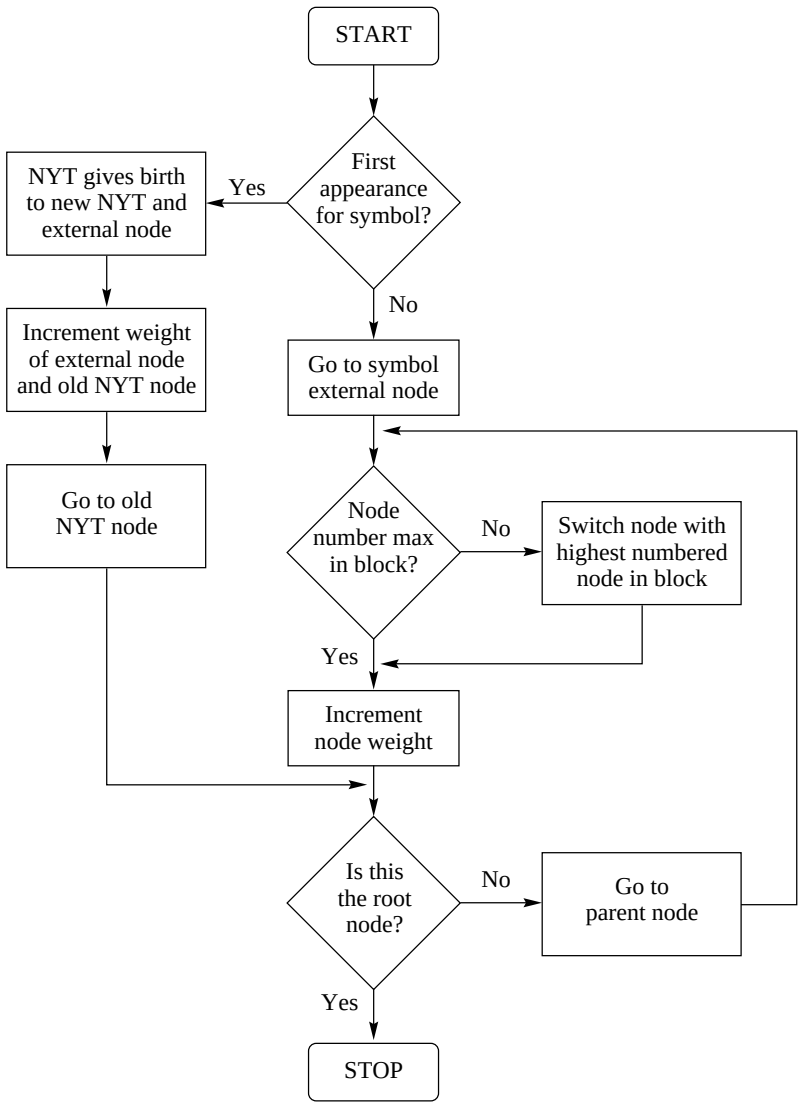

FIGURE 3.6 Update procedure for the adaptive Huffman coding algorithm.

The function of the update procedure is to preserve the sibling property. In order that the update procedures at the transmitter and receiver both operate with the same information, the tree at the transmitter is updated after each symbol is encoded, and the tree at the receiver is updated after each symbol is decoded. The procedure operates as follows:

After a symbol has been encoded or decoded, the external node corresponding to the symbol is examined to see if it has the largest node number in its block. If the external node does not have the largest node number, it is exchanged with the node that has the largest node number in the block, as long as the node with the higher number is not the parent of the node being updated. The weight of the external node is then incremented. If we did not exchange the nodes before the weight of the node is incremented, it is very likely that the ordering required by the sibling property would be destroyed. Once we have incremented the weight of the node, we have adapted the Huffman tree at that level. We then turn our attention to the next level by examining the parent node of the node whose weight was incremented to see if it has the largest number in its block. If it does not, it is exchanged with the node with the largest number in the block. Again, an exception to this is when the node with the higher node number is the parent of the node under consideration. Once an exchange has taken place (or it has been determined that there is no need for an exchange), the weight of the parent node is incremented. We then proceed to a new parent node and the process is repeated. This process continues until the root of the tree is reached.

If the symbol to be encoded or decoded has occurred for the first time, a new external node is assigned to the symbol and a new NYT node is appended to the tree. Both the new external node and the new NYT node are offsprings of the old NYT node. We increment the weight of the new external node by one. As the old NYT node is the parent of the new external node, we increment its weight by one and then go on to update all the other nodes until we reach the root of the tree.

Example 3.4.1: Update procedure

Assume we are encoding the message [a a r d v a r k], where our alphabet consists of the 26 lowercase letters of the English alphabet.

The updating process is shown in Figure 3.7. We begin with only the NYT node. The total number of nodes in this tree will be $2 \times 26 - 1 = 51$, so we start numbering backwards from 51 with the number of the root node being 51. The first letter to be transmitted is a. As a does not yet exist in the tree, we send a binary code 00000 for a and then add a to the tree. The NYT node gives birth to a new NYT node and a terminal node corresponding to a. The weight of the terminal node will be higher than the NYT node, so we assign the number 49 to the NYT node and 50 to the terminal node corresponding to the letter a. The second letter to be transmitted is also a. This time the transmitted code is 1. The node corresponding to a has the highest number (if we do not consider its parent), so we do not need to swap nodes. The next letter to be transmitted is r. This letter does not have a corresponding node on the tree, so we send the codeword for the NYT node, which is 0 followed by the index of r, which is 10001. The NYT node gives birth to a new NYT node and an external node corresponding to r. Again, no update is required. The next letter to be transmitted is d, which is also being sent for the first time. We again send the code for

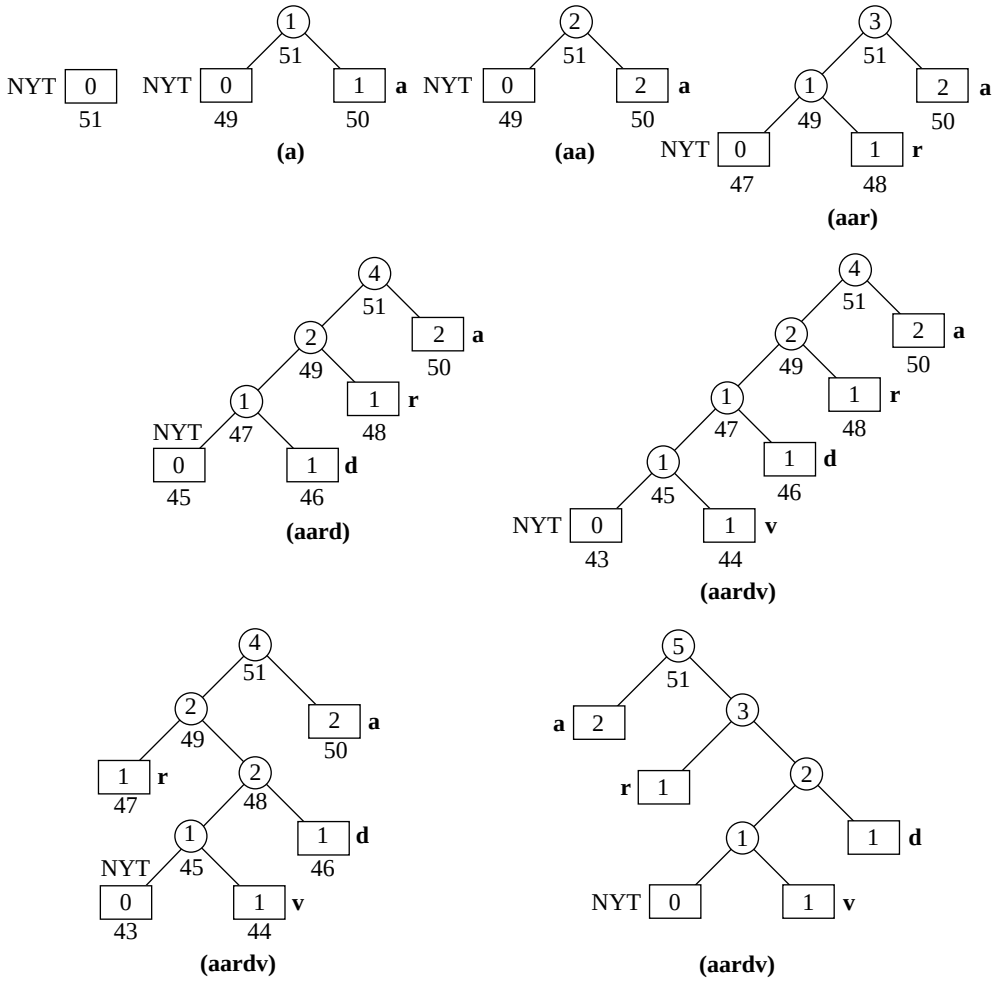

FIGURE 3.7 Adaptive Huffman tree after [a a r d v] is processed.

the NYT node, which is now 00 followed by the index for *d*, which is 00011. The NYT node again gives birth to two new nodes. However, an update is still not required. This changes with the transmission of the next letter, *v*, which has also not yet been encountered. Nodes 43 and 44 are added to the tree, with 44 as the terminal node corresponding to *v*. We examine the grandparent node of *v* (node 47) to see if it has the largest number in its block. As it does not, we swap it with node 48, which has the largest number in its block. We then increment node 48 and move to its parent, which is node 49. In the block containing node 49, the largest number belongs to node 50. Therefore, we swap nodes 49 and 50 and then increment node 50. We then move to the parent node of node 50, which is node 51. As this is the root node, all we do is increment node 51. ♦

3.4.2 Encoding Procedure

The flowchart for the encoding procedure is shown in Figure 3.8. Initially, the tree at both the encoder and decoder consists of a single node, the NYT node. Therefore, the codeword for the very first symbol that appears is a previously agreed-upon fixed code. After the very first symbol, whenever we have to encode a symbol that is being encountered for the first time, we send the code for the NYT node, followed by the previously agreed-upon fixed code for the symbol. The code for the NYT node is obtained by traversing the Huffman tree from the root to the NYT node. This alerts the receiver to the fact that the symbol whose code follows does not as yet have a node in the Huffman tree. If a symbol to be encoded has a corresponding node in the tree, then the code for the symbol is generated by traversing the tree from the root to the external node corresponding to the symbol.

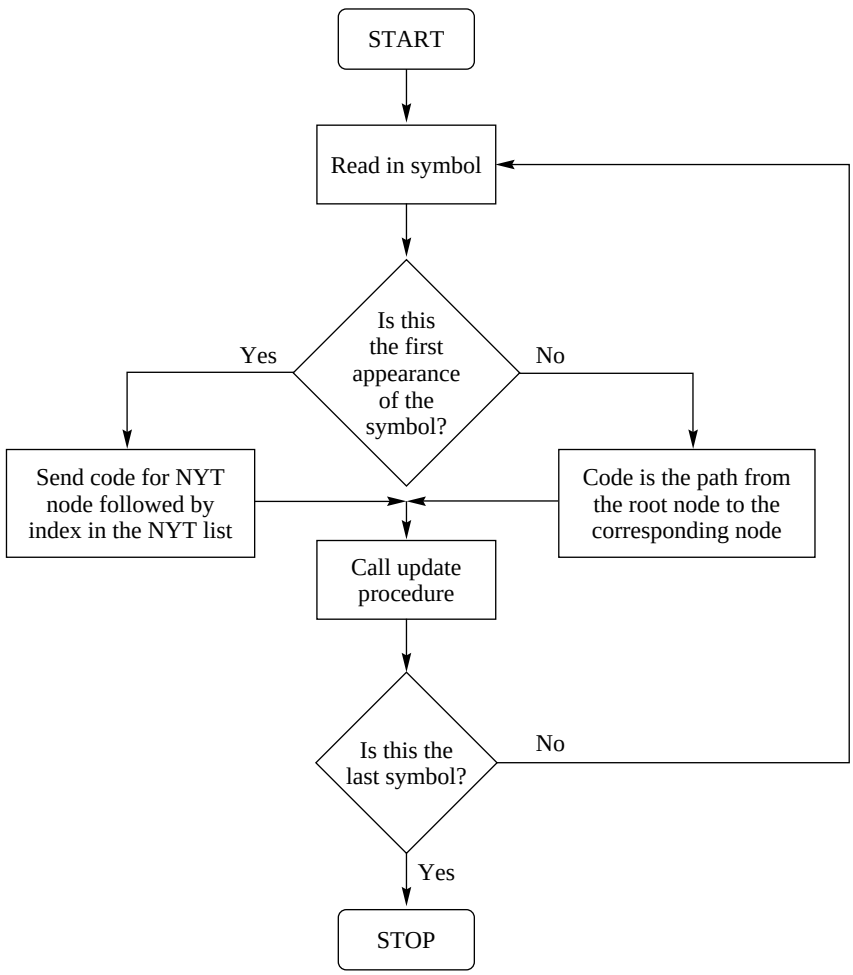

FIGURE 3.8 Flowchart of the encoding procedure.

To see how the coding operation functions, we use the same example that was used to demonstrate the update procedure.

Example 3.4.2: Encoding procedure

In Example 3.4.1 we used an alphabet consisting of 26 letters. In order to obtain our prearranged code, we have to find m and e such that $2^e + r = 26$, where $0 \leq r < 2^e$. It is easy to see that the values of $e = 4$ and $r = 10$ satisfy this requirement.

The first symbol encoded is the letter a. As a is the first letter of the alphabet, $k = 1$. As 1 is less than 20, a is encoded as the 5-bit binary representation of $k - 1$, or 0, which is 00000. The Huffman tree is then updated as shown in the figure. The NYT node gives birth to an external node corresponding to the element a and a new NYT node. As a has occurred once, the external node corresponding to a has a weight of one. The weight of the NYT node is zero. The internal node also has a weight of one, as its weight is the sum of the weights of its offspring. The next symbol is again a. As we have an external node corresponding to symbol a, we simply traverse the tree from the root node to the external node corresponding to a in order to find the codeword. This traversal consists of a single right branch. Therefore, the Huffman code for the symbol a is 1.

After the code for a has been transmitted, the weight of the external node corresponding to a is incremented, as is the weight of its parent. The third symbol to be transmitted is r. As this is the first appearance of this symbol, we send the code for the NYT node followed by the previously arranged binary representation for r. If we traverse the tree from the root to the NYT node, we get a code of 0 for the NYT node. The letter r is the 18th letter of the alphabet; therefore, the binary representation of r is 10001. The code for the symbol r becomes 010001. The tree is again updated as shown in the figure, and the coding process continues with symbol d. Using the same procedure for d, the code for the NYT node, which is now 00, is sent, followed by the index for d, resulting in the codeword 0000011. The next symbol v is the 22nd symbol in the alphabet. As this is greater than 20, we send the code for the NYT node followed by the 4-bit binary representation of $22 - 10 - 1 = 11$. The code for the NYT node at this stage is 000, and the 4-bit binary representation of 11 is 1011; therefore, v is encoded as 0001011. The next symbol is a, for which the code is 0, and the encoding proceeds. ♦

3.4.3 Decoding Procedure

The flowchart for the decoding procedure is shown in Figure 3.9. As we read in the received binary string, we traverse the tree in a manner identical to that used in the encoding procedure. Once a leaf is encountered, the symbol corresponding to that leaf is decoded. If the leaf is the NYT node, then we check the next e bits to see if the resulting number is less than r. If it is less than r, we read in another bit to complete the code for the symbol. The index for the symbol is obtained by adding one to the decimal number corresponding to the e- or $e + 1$-bit binary string. Once the symbol has been decoded, the tree is updated and the next received bit is used to start another traversal down the tree. To see how this procedure works, let us decode the binary string generated in the previous example.

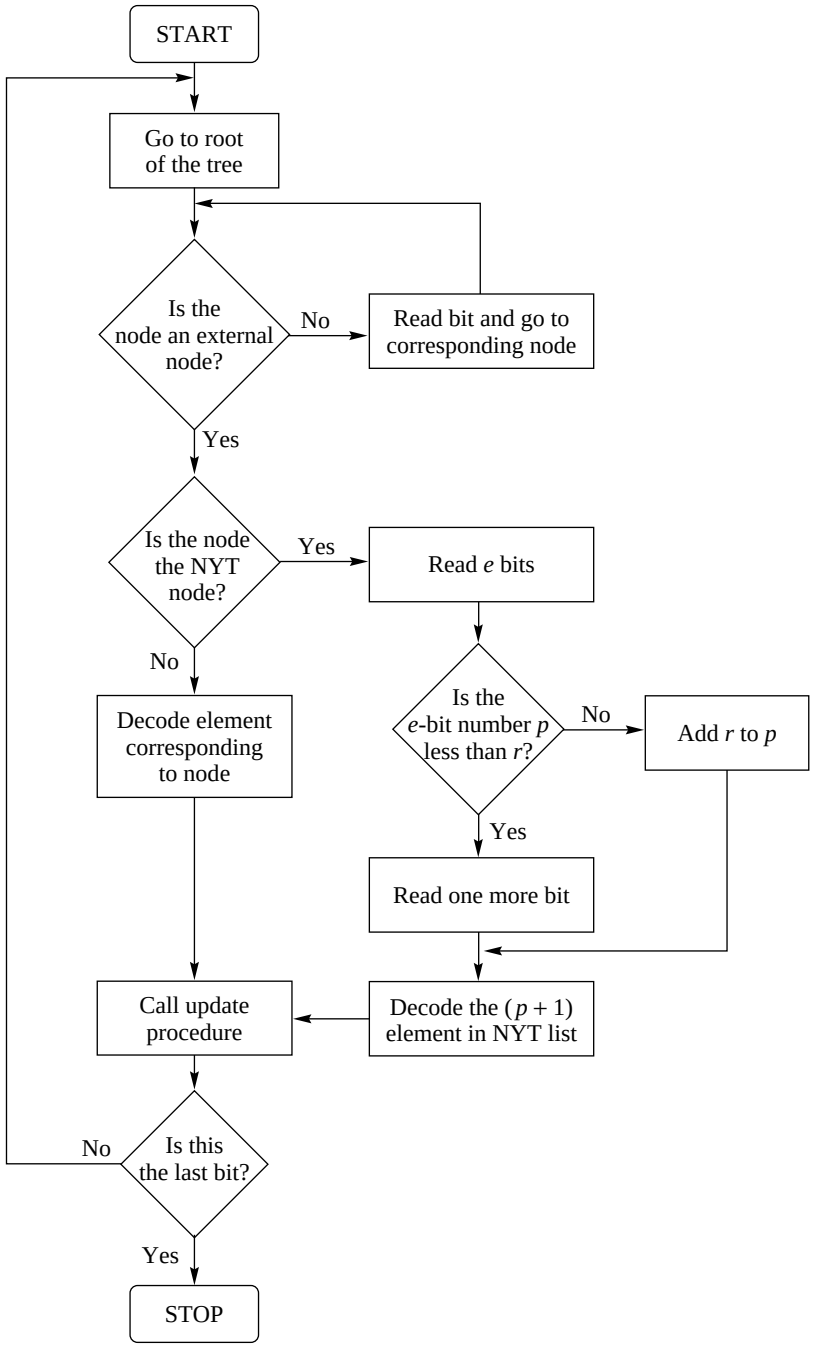

FIGURE 3.9 **Flowchart of the decoding procedure.**

Example 3.4.3: Decoding procedure

The binary string generated by the encoding procedure is

$$0000010100010000001100010110$$

Initially, the decoder tree consists only of the NYT node. Therefore, the first symbol to be decoded must be obtained from the NYT list. We read in the first 4 bits, 0000, as the value of e is four. The 4 bits 0000 correspond to the decimal value of 0. As this is less than the value of r, which is 10, we read in one more bit for the entire code of 00000. Adding one to the decimal value corresponding to this binary string, we get the index of the received symbol as 1. This is the index for a; therefore, the first letter is decoded as a. The tree is now updated as shown in Figure 3.7. The next bit in the string is 1. This traces a path from the root node to the external node corresponding to a. We decode the symbol a and update the tree. In this case, the update consists only of incrementing the weight of the external node corresponding to a. The next bit is a 0, which traces a path from the root to the NYT node. The next 4 bits, 1000, correspond to the decimal number 8, which is less than 10, so we read in one more bit to get the 5-bit word 10001. The decimal equivalent of this 5-bit word plus one is 18, which is the index for r. We decode the symbol r and then update the tree. The next 2 bits, 00, again trace a path to the NYT node. We read the next 4 bits, 0001. Since this corresponds to the decimal number 1, which is less than 10, we read another bit to get the 5-bit word 00011. To get the index of the received symbol in the NYT list, we add one to the decimal value of this 5-bit word. The value of the index is 4, which corresponds to the symbol d. Continuing in this fashion, we decode the sequence *aardva*. ♦

Although the Huffman coding algorithm is one of the best-known variable-length coding algorithms, there are some other lesser-known algorithms that can be very useful in certain situations. In particular, the Golomb-Rice codes and the Tunstall codes are becoming increasingly popular. We describe these codes in the following sections.

3.5 Golomb Codes

The Golomb-Rice codes belong to a family of codes designed to encode integers with the assumption that the larger an integer, the lower its probability of occurrence. The simplest code for this situation is the *unary* code. The unary code for a positive integer n is simply n 1s followed by a 0. Thus, the code for 4 is 11110, and the code for 7 is 11111110. The unary code is the same as the Huffman code for the semi-infinite alphabet $\{1, 2, 3, \ldots\}$ with probability model

$$P[k] = \frac{1}{2^k}.$$

Because the Huffman code is optimal, the unary code is also optimal for this probability model.

Although the unary code is optimal in very restricted conditions, we can see that it is certainly very simple to implement. One step higher in complexity are a number of coding schemes that split the integer into two parts, representing one part with a unary code and

the other part with a different code. An example of such a code is the Golomb code. Other examples can be found in [27].

The Golomb code is described in a succinct paper [28] by Solomon Golomb, which begins "Secret Agent 00111 is back at the Casino again, playing a game of chance, while the fate of mankind hangs in the balance." Agent 00111 requires a code to represent runs of success in a roulette game, and Golomb provides it! The Golomb code is actually a family of codes parameterized by an integer $m > 0$. In the Golomb code with parameter m, we represent an integer $n > 0$ using two numbers q and r, where

$$q = \left\lfloor \frac{n}{m} \right\rfloor$$

and

$$r = n - qm.$$

$\lfloor x \rfloor$ is the integer part of x. In other words, q is the quotient and r is the remainder when n is divided by m. The quotient q can take on values $0, 1, 2, \ldots$ and is represented by the unary code of q. The remainder r can take on the values $0, 1, 2, \ldots, m-1$. If m is a power of two, we use the $\log_2 m$-bit binary representation of r. If m is not a power of two, we could still use $\lceil \log_2 m \rceil$ bits, where $\lceil x \rceil$ is the smallest integer greater than or equal to x. We can reduce the number of bits required if we use the $\lfloor \log_2 m \rfloor$-bit binary representation of r for the first $2^{\lceil \log_2 m \rceil} - m$ values, and the $\lceil \log_2 m \rceil$-bit binary representation of $r + 2^{\lceil \log_2 m \rceil} - m$ for the rest of the values.

Example 3.5.1: Golomb code

Let's design a Golomb code for $m = 5$. As

$$\lceil \log_2 5 \rceil = 3, \quad \text{and} \quad \lfloor \log_2 5 \rfloor = 2$$

the first $8 - 5 = 3$ values of r (that is, $r = 0, 1, 2$) will be represented by the 2-bit binary representation of r, and the next two values (that is, $r = 3, 4$) will be represented by the 3-bit representation of $r + 3$. The quotient q is always represented by the unary code for q. Thus, the codeword for 3 is 0110, and the codeword for 21 is 1111001. The codewords for $n = 0, \ldots, 15$ are shown in Table 3.16.

TABLE 3.16 Golomb code for $m = 5$.

n	q	r	Codeword	n	q	r	Codeword
0	0	0	000	8	1	3	10110
1	0	1	001	9	1	4	10111
2	0	2	010	10	2	0	11000
3	0	3	0110	11	2	1	11001
4	0	4	0111	12	2	2	11010
5	1	0	1000	13	2	3	110110
6	1	1	1001	14	2	4	110111
7	1	2	1010	15	3	0	111000

♦

It can be shown that the Golomb code is optimal for the probability model

$$P(n) = p^{n-1}q, \qquad q = 1 - p$$

when

$$m = \left\lceil -\frac{1}{\log_2 p} \right\rceil.$$

3.6 Rice Codes

The Rice code was originally developed by Robert F. Rice (he called it the Rice machine) [29, 30] and later extended by Pen-Shu Yeh and Warner Miller [31]. The Rice code can be viewed as an adaptive Golomb code. In the Rice code, a sequence of nonnegative integers (which might have been obtained from the preprocessing of other data) is divided into blocks of J integers apiece. Each block is then coded using one of several options, most of which are a form of Golomb codes. Each block is encoded with each of these options, and the option resulting in the least number of coded bits is selected. The particular option used is indicated by an identifier attached to the code for each block.

The easiest way to understand the Rice code is to examine one of its implementations. We will study the implementation of the Rice code in the recommendation for lossless compression from the Consultative Committee on Space Data Standards (CCSDS).

3.6.1 CCSDS Recommendation for Lossless Compression

As an application of the Rice algorithm, let's briefly look at the algorithm for lossless data compression recommended by CCSDS. The algorithm consists of a preprocessor (the modeling step) and a binary coder (coding step). The preprocessor removes correlation from the input and generates a sequence of nonnegative integers. This sequence has the property that smaller values are more probable than larger values. The binary coder generates a bitstream to represent the integer sequence. The binary coder is our main focus at this point.

The preprocessor functions as follows: Given a sequence $\{y_i\}$, for each y_i we generate a prediction $\hat{y}_i$. A simple way to generate a prediction would be to take the previous value of the sequence to be a prediction of the current value of the sequence:

$$\hat{y}_i = y_{i-1}.$$

We will look at more sophisticated ways of generating a prediction in Chapter 7. We then generate a sequence whose elements are the difference between y_i and its predicted value $\hat{y}_i$:

$$d_i = y_i - \hat{y}_i.$$

The d_i value will have a small magnitude when our prediction is good and a large value when it is not. Assuming an accurate modeling of the data, the former situation is more likely than the latter. Let $y_{\max}$ and $y_{\min}$ be the largest and smallest values that the sequence

$\{y_i\}$ takes on. It is reasonable to assume that the value of $\hat{y}$ will be confined to the range $[y_{\min}, y_{\max}]$. Define

$$T_i = \min\{y_{\max} - \hat{y}, \hat{y} - y_{\min}\}. \tag{3.8}$$

The sequence $\{d_i\}$ can be converted into a sequence of nonnegative integers $\{x_i\}$ using the following mapping:

$$x_i = \begin{cases} 2d_i & 0 \le d_i \le T_i \\ 2|d_i| - 1 & -T_i \le d_i < 0 \\ T_i + |d_i| & \text{otherwise.} \end{cases} \tag{3.9}$$

The value of x_i will be small whenever the magnitude of d_i is small. Therefore, the value of x_i will be small with higher probability. The sequence $\{x_i\}$ is divided into segments with each segment being further divided into blocks of size J. It is recommended by CCSDS that J have a value of 16. Each block is then coded using one of the following options. The coded block is transmitted along with an identifier that indicates which particular option was used.

- **Fundamental sequence:** This is a unary code. A number n is represented by a sequence of n 0s followed by a 1 (or a sequence of n 1s followed by a 0).
- **Split sample options:** These options consist of a set of codes indexed by a parameter m. The code for a k-bit number n using the mth split sample option consists of the m least significant bits of k followed by a unary code representing the $k - m$ most significant bits. For example, suppose we wanted to encode the 8-bit number 23 using the third split sample option. The 8-bit representation of 23 is 00010111. The three least significant bits are 111. The remaining bits (00010) correspond to the number 2, which has a unary code 001. Therefore, the code for 23 using the third split sample option is 111011. Notice that different values of m will be preferable for different values of x_i, with higher values of m used for higher-entropy sequences.
- **Second extension option:** The second extension option is useful for sequences with low entropy—when, in general, many of the values of x_i will be zero. In the second extension option the sequence is divided into consecutive pairs of samples. Each pair is used to obtain an index γ using the following transformation:

$$\gamma = \frac{1}{2}(x_i + x_{i+1})(x_i + x_{i+1} + 1) + x_{i+1} \tag{3.10}$$

  and the value of γ is encoded using a unary code. The value of γ is an index to a lookup table with each value of γ corresponding to a pair of values x_i, x_{i+1}.
- **Zero block option:** The zero block option is used when one or more of the blocks of x_i are zero—generally when we have long sequences of y_i that have the same value. In this case the number of zero blocks are transmitted using the code shown in Table 3.17. The ROS code is used when the last five or more blocks in a segment are all zero.

The Rice code has been used in several space applications, and variations of the Rice code have been proposed for a number of different applications.

TABLE 3.17 Code used for zero block option.

Number of All-Zero Blocks	Codeword
1	1
2	01
3	001
4	0001
5	000001
6	0000001
$\vdots$	$\vdots$
63	$\overbrace{000\cdots0}^{63\ 0\text{s}}1$
ROS	00001

3.7 Tunstall Codes

Most of the variable-length codes that we look at in this book encode letters from the source alphabet using codewords with varying numbers of bits: codewords with fewer bits for letters that occur more frequently and codewords with more bits for letters that occur less frequently. The Tunstall code is an important exception. In the Tunstall code, all codewords are of equal length. However, each codeword represents a different number of letters. An example of a 2-bit Tunstall code for an alphabet $\mathcal{A} = \{A, B\}$ is shown in Table 3.18. The main advantage of a Tunstall code is that errors in codewords do not propagate, unlike other variable-length codes, such as Huffman codes, in which an error in one codeword will cause a series of errors to occur.

Example 3.7.1:

Let's encode the sequence $AAABAABAABAABAAA$ using the code in Table 3.18. Starting at the left, we can see that the string AAA occurs in our codebook and has a code of 00. We then code B as 11, AAB as 01, and so on. We finally end up with coded string 001101010100. ♦

TABLE 3.18 A 2-bit Tunstall code.

Sequence	Codeword
AAA	00
AAB	01
AB	10
B	11

TABLE 3.19 A 2-bit (non-Tunstall) code.

Sequence	Codeword
AAA	00
ABA	01
AB	10
B	11

The design of a code that has a fixed codeword length but a variable number of symbols per codeword should satisfy the following conditions:

1. We should be able to parse a source output sequence into sequences of symbols that appear in the codebook.
2. We should maximize the average number of source symbols represented by each codeword.

In order to understand what we mean by the first condition, consider the code shown in Table 3.19. Let's encode the same sequence $AAABAABAABAABAAA$ as in the previous example using the code in Table 3.19. We first encode AAA with the code 00. We then encode B with 11. The next three symbols are AAB. However, there are no codewords corresponding to this sequence of symbols. Thus, this sequence is unencodable using this particular code—not a desirable situation.

Tunstall [32] gives a simple algorithm that fulfills these conditions. The algorithm is as follows:

Suppose we want an n-bit Tunstall code for a source that generates *iid* letters from an alphabet of size N. The number of codewords is 2^n. We start with the N letters of the source alphabet in our codebook. Remove the entry in the codebook that has the highest probability and add the N strings obtained by concatenating this letter with every letter in the alphabet (including itself). This will increase the size of the codebook from N to $N+(N-1)$. The probabilities of the new entries will be the product of the probabilities of the letters concatenated to form the new entry. Now look through the $N+(N-1)$ entries in the codebook and find the entry that has the highest probability, keeping in mind that the entry with the highest probability may be a concatenation of symbols. Each time we perform this operation we increase the size of the codebook by $N-1$. Therefore, this operation can be performed K times, where

$$N+K(N-1) \leq 2^n.$$

Example 3.7.2: Tunstall codes

Let us design a 3-bit Tunstall code for a memoryless source with the following alphabet:

$$\mathcal{A} = \{A, B, C\}$$

$$P(A) = 0.6, \quad P(B) = 0.3, \quad P(C) = 0.1$$

TABLE 3.20 Source alphabet and associated probabilities.

Letter	Probability
A	0.60
B	0.30
C	0.10

TABLE 3.21 The codebook after one iteration.

Sequence	Probability
B	0.30
C	0.10
AA	0.36
AB	0.18
AC	0.06

TABLE 3.22 A 3-bit Tunstall code.

Sequence	Probability
B	000
C	001
AB	010
AC	011
AAA	100
AAB	101
AAC	110

We start out with the codebook and associated probabilities shown in Table 3.20. Since the letter A has the highest probability, we remove it from the list and add all two-letter strings beginning with A as shown in Table 3.21. After one iteration we have 5 entries in our codebook. Going through one more iteration will increase the size of the codebook by 2, and we will have 7 entries, which is still less than the final codebook size. Going through another iteration after that would bring the codebook size to 10, which is greater than the maximum size of 8. Therefore, we will go through just one more iteration. Looking through the entries in Table 3.22, the entry with the highest probability is AA. Therefore, at the next step we remove AA and add all extensions of AA as shown in Table 3.22. The final 3-bit Tunstall code is shown in Table 3.22. ♦

3.8 Applications of Huffman Coding

In this section we describe some applications of Huffman coding. As we progress through the book, we will describe more applications, since Huffman coding is often used in conjunction with other coding techniques.

3.8.1 Lossless Image Compression

A simple application of Huffman coding to image compression would be to generate a Huffman code for the set of values that any pixel may take. For monochrome images, this set usually consists of integers from 0 to 255. Examples of such images are contained in the accompanying data sets. The four that we will use in the examples in this book are shown in Figure 3.10.

FIGURE 3.10 **Test images.**

TABLE 3.23 Compression using Huffman codes on pixel values.

Image Name	Bits/Pixel	Total Size (bytes)	Compression Ratio
Sena	7.01	57,504	1.14
Sensin	7.49	61,430	1.07
Earth	4.94	40,534	1.62
Omaha	7.12	58,374	1.12

We will make use of one of the programs from the accompanying software (see Preface) to generate a Huffman code for each image, and then encode the image using the Huffman code. The results for the four images in Figure 3.10 are shown in Table 3.23. The Huffman code is stored along with the compressed image as the code will be required by the decoder to reconstruct the image.

The original (uncompressed) image representation uses 8 bits/pixel. The image consists of 256 rows of 256 pixels, so the uncompressed representation uses 65,536 bytes. The compression ratio is simply the ratio of the number of bytes in the uncompressed representation to the number of bytes in the compressed representation. The number of bytes in the compressed representation includes the number of bytes needed to store the Huffman code. Notice that the compression ratio is different for different images. This can cause some problems in certain applications where it is necessary to know in advance how many bytes will be needed to represent a particular data set.

The results in Table 3.23 are somewhat disappointing because we get a reduction of only about $\frac{1}{2}$ to 1 bit/pixel after compression. For some applications this reduction is acceptable. For example, if we were storing thousands of images in an archive, a reduction of 1 bit/pixel saves many megabytes in disk space. However, we can do better. Recall that when we first talked about compression, we said that the first step for any compression algorithm was to model the data so as to make use of the structure in the data. In this case, we have made absolutely no use of the structure in the data.

From a visual inspection of the test images, we can clearly see that the pixels in an image are heavily correlated with their neighbors. We could represent this structure with the crude model $\hat{x}_n = x_{n-1}$. The residual would be the difference between neighboring pixels. If we carry out this differencing operation and use the Huffman coder on the residuals, the results are as shown in Table 3.24. As we can see, using the structure in the data resulted in substantial improvement.

TABLE 3.24 Compression using Huffman codes on pixel difference values.

Image Name	Bits/Pixel	Total Size (bytes)	Compression Ratio
Sena	4.02	32,968	1.99
Sensin	4.70	38,541	1.70
Earth	4.13	33,880	1.93
Omaha	6.42	52,643	1.24

TABLE 3.25 Compression using adaptive Huffman codes on pixel difference values.

Image Name	Bits/Pixel	Total Size (bytes)	Compression Ratio
Sena	3.93	32,261	2.03
Sensin	4.63	37,896	1.73
Earth	4.82	39,504	1.66
Omaha	6.39	52,321	1.25

The results in Tables 3.23 and 3.24 were obtained using a two-pass system, in which the statistics were collected in the first pass and a Huffman table was generated. Instead of using a two-pass system, we could have used a one-pass adaptive Huffman coder. The results for this are given in Table 3.25.

Notice that there is little difference between the performance of the adaptive Huffman code and the two-pass Huffman coder. In addition, the fact that the adaptive Huffman coder can be used as an on-line or real-time coder makes the adaptive Huffman coder a more attractive option in many applications. However, the adaptive Huffman coder is more vulnerable to errors and may also be more difficult to implement. In the end, the particular application will determine which approach is more suitable.

3.8.2 Text Compression

Text compression seems natural for Huffman coding. In text, we have a discrete alphabet that, in a given class, has relatively stationary probabilities. For example, the probability model for a particular novel will not differ significantly from the probability model for another novel. Similarly, the probability model for a set of FORTRAN programs is not going to be much different than the probability model for a different set of FORTRAN programs. The probabilities in Table 3.26 are the probabilities of the 26 letters (upper- and lowercase) obtained for the U.S. Constitution and are representative of English text. The probabilities in Table 3.27 were obtained by counting the frequency of occurrences of letters in an earlier version of this chapter. While the two documents are substantially different, the two sets of probabilities are very much alike.

We encoded the earlier version of this chapter using Huffman codes that were created using the probabilities of occurrence obtained from the chapter. The file size dropped from about 70,000 bytes to about 43,000 bytes with Huffman coding.

While this reduction in file size is useful, we could have obtained better compression if we first removed the structure existing in the form of correlation between the symbols in the file. Obviously, there is a substantial amount of correlation in this text. For example, *Huf* is always followed by *fman*! Unfortunately, this correlation is not amenable to simple numerical models, as was the case for the image files. However, there are other somewhat more complex techniques that can be used to remove the correlation in text files. We will look more closely at these in Chapters 5 and 6.

TABLE 3.26 Probabilities of occurrence of the letters in the English alphabet in the U.S. Constitution.

Letter	Probability	Letter	Probability
A	0.057305	N	0.056035
B	0.014876	O	0.058215
C	0.025775	P	0.021034
D	0.026811	Q	0.000973
E	0.112578	R	0.048819
F	0.022875	S	0.060289
G	0.009523	T	0.078085
H	0.042915	U	0.018474
I	0.053475	V	0.009882
J	0.002031	W	0.007576
K	0.001016	X	0.002264
L	0.031403	Y	0.011702
M	0.015892	Z	0.001502

TABLE 3.27 Probabilities of occurrence of the letters in the English alphabet in this chapter.

Letter	Probability	Letter	Probability
A	0.049855	N	0.048039
B	0.016100	O	0.050642
C	0.025835	P	0.015007
D	0.030232	Q	0.001509
E	0.097434	R	0.040492
F	0.019754	S	0.042657
G	0.012053	T	0.061142
H	0.035723	U	0.015794
I	0.048783	V	0.004988
J	0.000394	W	0.012207
K	0.002450	X	0.003413
L	0.025835	Y	0.008466
M	0.016494	Z	0.001050

3.8.3 Audio Compression

Another class of data that is very suitable for compression is CD-quality audio data. The audio signal for each stereo channel is sampled at 44.1 kHz, and each sample is represented by 16 bits. This means that the amount of data stored on one CD is enormous. If we want to transmit this data, the amount of channel capacity required would be significant. Compression is definitely useful in this case. In Table 3.28 we show for a variety of audio material the file size, the entropy, the estimated compressed file size if a Huffman coder is used, and the resulting compression ratio.

TABLE 3.28 Huffman coding of 16-bit CD-quality audio.

File Name	Original File Size (bytes)	Entropy (bits)	Estimated Compressed File Size (bytes)	Compression Ratio
Mozart	939,862	12.8	725,420	1.30
Cohn	402,442	13.8	349,300	1.15
Mir	884,020	13.7	759,540	1.16

The three segments used in this example represent a wide variety of audio material, from a symphonic piece by Mozart to a folk rock piece by Cohn. Even though the material is varied, Huffman coding can lead to some reduction in the capacity required to transmit this material.

Note that we have only provided the *estimated* compressed file sizes. The estimated file size in bits was obtained by multiplying the entropy by the number of samples in the file. We used this approach because the samples of 16-bit audio can take on 65,536 distinct values, and therefore the Huffman coder would require 65,536 distinct (variable-length) codewords. In most applications, a codebook of this size would not be practical. There is a way of handling large alphabets, called recursive indexing, that we will describe in Chapter 9. There is also some recent work [14] on using a Huffman tree in which leaves represent sets of symbols with the same probability. The codeword consists of a prefix that specifies the set followed by a suffix that specifies the symbol within the set. This approach can accommodate relatively large alphabets.

As with the other applications, we can obtain an increase in compression if we first remove the structure from the data. Audio data can be modeled numerically. In later chapters we will examine more sophisticated modeling approaches. For now, let us use the very simple model that was used in the image-coding example; that is, each sample has the same value as the previous sample. Using this model we obtain the difference sequence. The entropy of the difference sequence is shown in Table 3.29.

Note that there is a further reduction in the file size: the compressed file sizes are about 60% of the original files. Further reductions can be obtained by using more sophisticated models.

Many of the lossless audio compression schemes, including FLAC (Free Lossless Audio Codec), Apple's ALAC or ALE, *Shorten* [33], *Monkey's Audio*, and the proposed (as of now) MPEG-4 ALS [34] algorithms, use a linear predictive model to remove some of

TABLE 3.29 Huffman coding of differences of 16-bit CD-quality audio.

File Name	Original File Size (bytes)	Entropy of Differences (bits)	Estimated Compressed File Size (bytes)	Compression Ratio
Mozart	939,862	9.7	569,792	1.65
Cohn	402,442	10.4	261,590	1.54
Mir	884,020	10.9	602,240	1.47

the structure from the audio sequence and then use Rice coding to encode the residuals. Most others, such as *AudioPak* [35] and *OggSquish*, use Huffman coding to encode the residuals.

3.9 Summary

In this chapter we began our exploration of data compression techniques with a description of the Huffman coding technique and several other related techniques. The Huffman coding technique and its variants are some of the most commonly used coding approaches. We will encounter modified versions of Huffman codes when we look at compression techniques for text, image, and video. In this chapter we described how to design Huffman codes and discussed some of the issues related to Huffman codes. We also described how adaptive Huffman codes work and looked briefly at some of the places where Huffman codes are used. We will see more of these in future chapters.

To explore further applications of Huffman coding, you can use the programs `huff_enc`, `huff_dec`, and `adap_huff` to generate your own Huffman codes for your favorite applications.

Further Reading

1. A detailed and very accessible overview of Huffman codes is provided in "Huffman Codes," by S. Pigeon [36], in *Lossless Compression Handbook*.
2. Details about nonbinary Huffman codes and a much more theoretical and rigorous description of variable-length codes can be found in *The Theory of Information and Coding*, volume 3 of *Encyclopedia of Mathematic and Its Application*, by R.J. McEliece [6].
3. The tutorial article "Data Compression" in the September 1987 issue of *ACM Computing Surveys*, by D.A. Lelewer and D.S. Hirschberg [37], along with other material, provides a very nice brief coverage of the material in this chapter.
4. A somewhat different approach to describing Huffman codes can be found in *Data Compression—Methods and Theory*, by J.A. Storer [38].
5. A more theoretical but very readable account of variable-length coding can be found in *Elements of Information Theory*, by T.M. Cover and J.A. Thomas [3].
6. Although the book *Coding and Information Theory*, by R.W. Hamming [9], is mostly about channel coding, Huffman codes are described in some detail in Chapter 4.

3.10 Projects and Problems

1. The probabilities in Tables 3.27 and 3.27 were obtained using the program `countalpha` from the accompanying software. Use this program to compare probabilities for different types of text, C programs, messages on Usenet, and so on.

Comment on any differences you might see and describe how you would tailor your compression strategy for each type of text.

2. Use the programs `huff_enc` and `huff_dec` to do the following (in each case use the codebook generated by the image being compressed):

(a) Code the Sena, Sinan, and Omaha images.

(b) Write a program to take the difference between adjoining pixels, and then use `huffman` to code the difference images.

(c) Repeat (a) and (b) using `adap_huff`.

Report the resulting file sizes for each of these experiments and comment on the differences.

3. Using the programs `huff_enc` and `huff_dec`, code the Bookshelf1 and Sena images using the codebook generated by the Sinan image. Compare the results with the case where the codebook was generated by the image being compressed.

4. A source emits letters from an alphabet $\mathcal{A} = \{a_1, a_2, a_3, a_4, a_5\}$ with probabilities $P(a_1) = 0.15$, $P(a_2) = 0.04$, $P(a_3) = 0.26$, $P(a_4) = 0.05$, and $P(a_5) = 0.50$.

(a) Calculate the entropy of this source.

(b) Find a Huffman code for this source.

(c) Find the average length of the code in (b) and its redundancy.

5. For an alphabet $\mathcal{A} = \{a_1, a_2, a_3, a_4\}$ with probabilities $P(a_1) = 0.1$, $P(a_2) = 0.3$, $P(a_3) = 0.25$, and $P(a_4) = 0.35$, find a Huffman code

(a) using the first procedure outlined in this chapter, and

(b) using the minimum variance procedure.

Comment on the difference in the Huffman codes.

6. In many communication applications, it is desirable that the number of 1s and 0s transmitted over the channel are about the same. However, if we look at Huffman codes, many of them seem to have many more 1s than 0s or vice versa. Does this mean that Huffman coding will lead to inefficient channel usage? For the Huffman code obtained in Problem 3, find the probability that a 0 will be transmitted over the channel. What does this probability say about the question posed above?

7. For the source in Example 3.3.1, generate a ternary code by combining three letters in the first and second steps and two letters in the third step. Compare with the ternary code obtained in the example.

8. In Example 3.4.1 we have shown how the tree develops when the sequence *a a r d v* is transmitted. Continue this example with the next letters in the sequence, *a r k*.

9. The Monte Carlo approach is often used for studying problems that are difficult to solve analytically. Let's use this approach to study the problem of buffering when

using variable-length codes. We will simulate the situation in Example 3.2.1, and study the time to overflow and underflow as a function of the buffer size. In our program, we will need a random number generator, a set of seeds to initialize the random number generator, a counter B to simulate the buffer occupancy, a counter T to keep track of the time, and a value N, which is the size of the buffer. Input to the buffer is simulated by using the random number generator to select a letter from our alphabet. The counter B is then incremented by the length of the codeword for the letter. The output to the buffer is simulated by decrementing B by 2 except when T is divisible by 5. For values of T divisible by 5, decrement B by 3 instead of 2 (why?). Keep incrementing T, each time simulating an input and an output, until either $B \geq N$, corresponding to a buffer overflow, or $B < 0$, corresponding to a buffer underflow. When either of these events happens, record what happened and when, and restart the simulation with a new seed. Do this with at least 100 seeds.

Perform this simulation for a number of buffer sizes ($N = 100, 1000, 10{,}000$), and the two Huffman codes obtained for the source in Example 3.2.1. Describe your results in a report.

10. While the variance of lengths is an important consideration when choosing between two Huffman codes that have the same average lengths, it is not the only consideration. Another consideration is the ability to recover from errors in the channel. In this problem we will explore the effect of error on two equivalent Huffman codes.

(a) For the source and Huffman code of Example 3.2.1 (Table 3.5), encode the sequence

$$a_2\ a_1\ a_3\ a_2\ a_1\ a_2$$

Suppose there was an error in the channel and the first bit was received as a 0 instead of a 1. Decode the received sequence of bits. How many characters are received in error before the first correctly decoded character?

(b) Repeat using the code in Table 3.9.

(c) Repeat parts (a) and (b) with the error in the third bit.

11. (This problem was suggested by P.F. Swaszek.)

(a) For a binary source with probabilities $P(0) = 0.9$, $P(1) = 0.1$, design a Huffman code for the source obtained by blocking m bits together, $m = 1, 2, \ldots, 8$. Plot the average lengths versus m. Comment on your result.

(b) Repeat for $P(0) = 0.99$, $P(1) = 0.01$.

You can use the program `huff_enc` to generate the Huffman codes.

12. Encode the following sequence of 16 values using the Rice code with $J = 8$ and one split sample option.

$$32, 33, 35, 39, 37, 38, 39, 40, 40, 40, 40, 39, 40, 40, 41, 40$$

For prediction use the previous value in the sequence

$$\hat{y}_i = y_{i-1}$$

and assume a prediction of zero for the first element of the sequence.

13. For an alphabet $\mathcal{A} = \{a_1, a_2, a_3\}$ with probabilities $P(a_1) = 0.7$, $P(a_2) = 0.2$, $P(a_3) = 0.1$, design a 3-bit Tunstall code.

14. Write a program for encoding images using the Rice algorithm. Use eight options, including the fundamental sequence, five split sample options, and the two low-entropy options. Use $J = 16$. For prediction use either the pixel to the left or the pixel above. Encode the Sena image using your program. Compare your results with the results obtained by Huffman coding the differences between pixels.

4

Arithmetic Coding

4.1 Overview

In the previous chapter we saw one approach to generating variable-length codes. In this chapter we see another, increasingly popular, method of generating variable-length codes called *arithmetic coding*. Arithmetic coding is especially useful when dealing with sources with small alphabets, such as binary sources, and alphabets with highly skewed probabilities. It is also a very useful approach when, for various reasons, the modeling and coding aspects of lossless compression are to be kept separate. In this chapter, we look at the basic ideas behind arithmetic coding, study some of the properties of arithmetic codes, and describe an implementation.

4.2 Introduction

In the last chapter we studied the Huffman coding method, which guarantees a coding rate R within 1 bit of the entropy H. Recall that the coding rate is the average number of bits used to represent a symbol from a source and, for a given probability model, the entropy is the lowest rate at which the source can be coded. We can tighten this bound somewhat. It has been shown [23] that the Huffman algorithm will generate a code whose rate is within $p_{\max} + 0.086$ of the entropy, where $p_{\max}$ is the probability of the most frequently occurring symbol. We noted in the last chapter that, in applications where the alphabet size is large, $p_{\max}$ is generally quite small, and the amount of deviation from the entropy, especially in terms of a percentage of the rate, is quite small. However, in cases where the alphabet is small and the probability of occurrence of the different letters is skewed, the value of $p_{\max}$ can be quite large and the Huffman code can become rather inefficient when compared to the entropy. One way to avoid this problem is to block more than one symbol together and generate an extended Huffman code. Unfortunately, this approach does not always work.

Example 4.2.1:

Consider a source that puts out independent, identically distributed (*iid*) letters from the alphabet $\mathcal{A} = \{a_1, a_2, a_3\}$ with the probability model $P(a_1) = 0.95$, $P(a_2) = 0.02$, and $P(a_3) = 0.03$. The entropy for this source is 0.335 bits/symbol. A Huffman code for this source is given in Table 4.1.

TABLE 4.1 Huffman code for three-letter alphabet.

Letter	Codeword
a_1	0
a_2	11
a_3	10

The average length for this code is 1.05 bits/symbol. The difference between the average code length and the entropy, or the redundancy, for this code is 0.715 bits/symbol, which is 213% of the entropy. This means that to code this sequence we would need more than twice the number of bits promised by the entropy.

Recall Example 3.2.4. Here also we can group the symbols in blocks of two. The extended alphabet, probability model, and code can be obtained as shown in Table 4.2. The average rate for the extended alphabet is 1.222 bits/symbol, which in terms of the original alphabet is 0.611 bits/symbol. As the entropy of the source is 0.335 bits/symbol, the additional rate over the entropy is still about 72% of the entropy! By continuing to block symbols together, we find that the redundancy drops to acceptable values when we block eight symbols together. The corresponding alphabet size for this level of blocking is 6561! A code of this size is impractical for a number of reasons. Storage of a code like this requires memory that may not be available for many applications. While it may be possible to design reasonably efficient encoders, decoding a Huffman code of this size would be a highly inefficient and time-consuming procedure. Finally, if there were some perturbation in the statistics, and some of the assumed probabilities changed slightly, this would have a major impact on the efficiency of the code.

TABLE 4.2 Huffman code for extended alphabet.

Letter	Probability	Code
a_1a_1	0.9025	0
a_1a_2	0.0190	111
a_1a_3	0.0285	100
a_2a_1	0.0190	1101
a_2a_2	0.0004	110011
a_2a_3	0.0006	110001
a_3a_1	0.0285	101
a_3a_2	0.0006	110010
a_3a_3	0.0009	110000

♦

We can see that it is more efficient to generate codewords for groups or sequences of symbols rather than generating a separate codeword for each symbol in a sequence. However, this approach becomes impractical when we try to obtain Huffman codes for long sequences of symbols. In order to find the Huffman codeword for a particular sequence of length m, we need codewords for all possible sequences of length m. This fact causes an exponential growth in the size of the codebook. We need a way of assigning codewords to *particular* sequences without having to generate codes for all sequences of that length. The arithmetic coding technique fulfills this requirement.

In arithmetic coding a unique identifier or tag is generated for the sequence to be encoded. This tag corresponds to a binary fraction, which becomes the binary code for the sequence. In practice the generation of the tag and the binary code are the same process. However, the arithmetic coding approach is easier to understand if we conceptually divide the approach into two phases. In the first phase a unique identifier or tag is generated for a given sequence of symbols. This tag is then given a unique binary code. A unique arithmetic code can be generated for a sequence of length m without the need for generating codewords for all sequences of length m. This is unlike the situation for Huffman codes. In order to generate a Huffman code for a sequence of length m, where the code is not a concatenation of the codewords for the individual symbols, we need to obtain the Huffman codes for all sequences of length m.

4.3 Coding a Sequence

In order to distinguish a sequence of symbols from another sequence of symbols we need to tag it with a unique identifier. One possible set of tags for representing sequences of symbols are the numbers in the unit interval $[0, 1)$. Because the number of numbers in the unit interval is infinite, it should be possible to assign a unique tag to each distinct sequence of symbols. In order to do this we need a function that will map sequences of symbols into the unit interval. A function that maps random variables, and sequences of random variables, into the unit interval is the cumulative distribution function (*cdf*) of the random variable associated with the source. This is the function we will use in developing the arithmetic code. (If you are not familiar with random variables and cumulative distribution functions, or need to refresh your memory, you may wish to look at Appendix A.)

The use of the cumulative distribution function to generate a binary code for a sequence has a rather interesting history. Shannon, in his original 1948 paper [7], mentioned an approach using the cumulative distribution function when describing what is now known as the Shannon-Fano code. Peter Elias, another member of Fano's first information theory class at MIT (this class also included Huffman), came up with a recursive implementation for this idea. However, he never published it, and we only know about it through a mention in a 1963 book on information theory by Abramson [39]. Abramson described this coding approach in a note to a chapter. In another book on information theory by Jelinek [40] in 1968, the idea of arithmetic coding is further developed, this time in an appendix, as an example of variable-length coding. Modern arithmetic coding owes its birth to the independent discoveries in 1976 of Pasco [41] and Rissanen [42] that the problem of finite precision could be resolved.

Finally, several papers appeared that provided practical arithmetic coding algorithms, the most well known of which is the paper by Rissanen and Langdon [43].

Before we begin our development of the arithmetic code, we need to establish some notation. Recall that a random variable maps the outcomes, or sets of outcomes, of an experiment to values on the real number line. For example, in a coin-tossing experiment, the random variable could map a head to zero and a tail to one (or it could map a head to 2367.5 and a tail to -192). To use this technique, we need to map the source symbols or letters to numbers. For convenience, in the discussion in this chapter we will use the mapping

$$X(a_i) = i \qquad a_i \in \mathcal{A} \tag{4.1}$$

where $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$ is the alphabet for a discrete source and X is a random variable. This mapping means that given a probability model $\mathcal{P}$ for the source, we also have a probability density function for the random variable

$$P(X = i) = P(a_i)$$

and the cumulative density function can be defined as

$$F_X(i) = \sum_{k=1}^{i} P(X = k).$$

Notice that for each symbol a_i with a nonzero probability we have a distinct value of $F_X(i)$. We will use this fact in what follows to develop the arithmetic code. Our development may be more detailed than what you are looking for, at least on the first reading. If so, skip or skim Sections 4.3.1–4.4.1 and go directly to Section 4.4.2.

4.3.1 Generating a Tag

The procedure for generating the tag works by reducing the size of the interval in which the tag resides as more and more elements of the sequence are received.

We start out by first dividing the unit interval into subintervals of the form $[F_X(i-1), F_X(i))$, $i = 1, \ldots, m$. Because the minimum value of the *cdf* is zero and the maximum value is one, this exactly partitions the unit interval. We associate the subinterval $[F_X(i-1), F_X(i))$ with the symbol a_i. The appearance of the first symbol in the sequence restricts the interval containing the tag to one of these subintervals. Suppose the first symbol was a_k. Then the interval containing the tag value will be the subinterval $[F_X(k-1), F_X(k))$. This subinterval is now partitioned in exactly the same proportions as the original interval. That is, the jth interval corresponding to the symbol a_j is given by $[F_X(k-1) + F_X(j-1)/(F_X(k) - F_X(k-1)), F_X(k-1) + F_X(j)/(F_X(k) - F_X(k-1)))$. So if the second symbol in the sequence is a_j, then the interval containing the tag value becomes $[F_X(k-1) + F_X(j-1)/(F_X(k) - F_X(k-1)), F_X(k-1) + F_X(j)/(F_X(k) - F_X(k-1)))$. Each succeeding symbol causes the tag to be restricted to a subinterval that is further partitioned in the same proportions. This process can be more clearly understood through an example.

Example 4.3.1:

Consider a three-letter alphabet $\mathcal{A} = \{a_1, a_2, a_3\}$ with $P(a_1) = 0.7$, $P(a_2) = 0.1$, and $P(a_3) = 0.2$. Using the mapping of Equation (4.1), $F_X(1) = 0.7$, $F_X(2) = 0.8$, and $F_X(3) = 1$. This partitions the unit interval as shown in Figure 4.1.

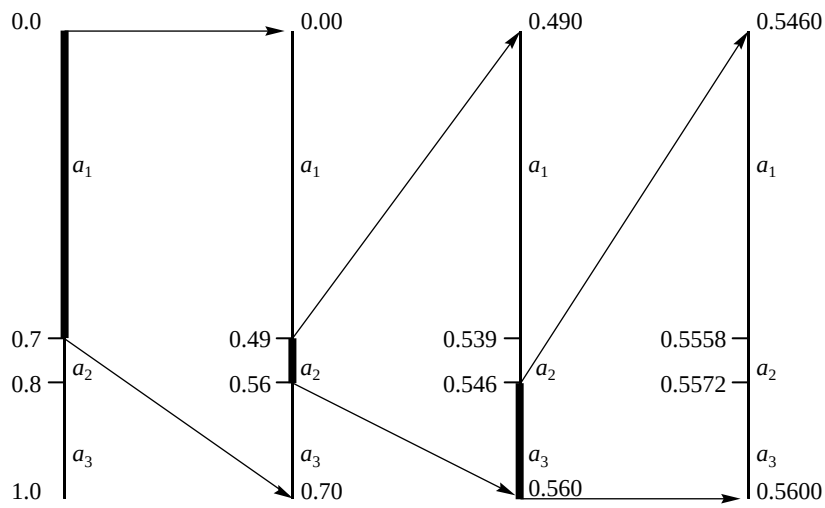

FIGURE 4.1 **Restricting the interval containing the tag for the input sequence $\{a_1, a_2, a_3, \ldots\}$.**

The partition in which the tag resides depends on the first symbol of the sequence being encoded. For example, if the first symbol is a_1, the tag lies in the interval $[0.0, 0.7)$; if the first symbol is a_2, the tag lies in the interval $[0.7, 0.8)$; and if the first symbol is a_3, the tag lies in the interval $[0.8, 1.0)$. Once the interval containing the tag has been determined, the rest of the unit interval is discarded, and this restricted interval is again divided in the same proportions as the original interval. Suppose the first symbol was a_1. The tag would be contained in the subinterval $[0.0, 0.7)$. This subinterval is then subdivided in exactly the same proportions as the original interval, yielding the subintervals $[0.0, 0.49)$, $[0.49, 0.56)$, and $[0.56, 0.7)$. The first partition as before corresponds to the symbol a_1, the second partition corresponds to the symbol a_2, and the third partition $[0.56, 0.7)$ corresponds to the symbol a_3. Suppose the second symbol in the sequence is a_2. The tag value is then restricted to lie in the interval $[0.49, 0.56)$. We now partition this interval in the same proportion as the original interval to obtain the subintervals $[0.49, 0.539)$ corresponding to the symbol a_1, $[0.539, 0.546)$ corresponding to the symbol a_2, and $[0.546, 0.56)$ corresponding to the symbol a_3. If the third symbol is a_3, the tag will be restricted to the interval $[0.546, 0.56)$, which can then be subdivided further. This process is described graphically in Figure 4.1.

Notice that the appearance of each new symbol restricts the tag to a subinterval that is disjoint from any other subinterval that may have been generated using this process. For

the sequence beginning with $\{a_1, a_2, a_3, \ldots\}$, by the time the third symbol a_3 is received, the tag has been restricted to the subinterval $[0.546, 0.56)$. If the third symbol had been a_1 instead of a_3, the tag would have resided in the subinterval $[0.49, 0.539)$, which is disjoint from the subinterval $[0.546, 0.56)$. Even if the two sequences are identical from this point on (one starting with a_1, a_2, a_3 and the other beginning with a_1, a_2, a_1), the tag interval for the two sequences will always be disjoint. ♦

As we can see, the interval in which the tag for a particular sequence resides is disjoint from all intervals in which the tag for any other sequence may reside. As such, any member of this interval can be used as a tag. One popular choice is the lower limit of the interval; another possibility is the midpoint of the interval. For the moment, let's use the midpoint of the interval as the tag.

In order to see how the tag generation procedure works mathematically, we start with sequences of length one. Suppose we have a source that puts out symbols from some alphabet $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$. We can map the symbols $\{a_i\}$ to real numbers $\{i\}$. Define $\bar{T}_X(a_i)$ as

$$\bar{T}_X(a_i) = \sum_{k=1}^{i-1} P(X = k) + \frac{1}{2}P(X = i) \tag{4.2}$$

$$= F_X(i-1) + \frac{1}{2}P(X = i). \tag{4.3}$$

For each a_i, $\bar{T}_X(a_i)$ will have a unique value. This value can be used as a unique tag for a_i.

Example 4.3.2:

Consider a simple dice-throwing experiment with a fair die. The outcomes of a roll of the die can be mapped into the numbers $\{1, 2, \ldots, 6\}$. For a fair die

$$P(X = k) = \frac{1}{6} \qquad \text{for } k = 1, 2, \ldots, 6.$$

Therefore, using (4.3) we can find the tag for $X = 2$ as

$$\bar{T}_X(2) = P(X = 1) + \frac{1}{2}P(X = 2) = \frac{1}{6} + \frac{1}{12} = 0.25$$

and the tag for $X = 5$ as

$$\bar{T}_X(5) = \sum_{k=1}^{4} P(X = k) + \frac{1}{2}P(X = 5) = 0.75.$$

The tags for all other outcomes are shown in Table 4.3.

TABLE 4.3 Toss for outcomes in a dice-throwing experment.

Outcome	Tag
1	$0.08\overline{33}$
3	$0.41\overline{66}$
4	$0.58\overline{33}$
6	$0.91\overline{66}$

♦

As we can see from the example above, giving a unique tag to a sequence of length one is an easy task. This approach can be extended to longer sequences by imposing an order on the sequences. We need an ordering on the sequences because we will assign a tag to a particular sequence $\mathbf{x}_i$ as

$$\bar{T}_{\mathbf{X}}^{(m)}(\mathbf{x}_i) = \sum_{\mathbf{y}<\mathbf{x}_i} P(\mathbf{y}) + \frac{1}{2}P(\mathbf{x}_i) \tag{4.4}$$

where $\mathbf{y} < \mathbf{x}$ means that $\mathbf{y}$ precedes $\mathbf{x}$ in the ordering, and the superscript denotes the length of the sequence.

An easy ordering to use is *lexicographic ordering*. In lexicographic ordering, the ordering of letters in an alphabet induces an ordering on the words constructed from this alphabet. The ordering of words in a dictionary is a good (maybe the original) example of lexicographic ordering. *Dictionary order* is sometimes used as a synonym for lexicographic order.

Example 4.3.3:

We can extend Example 4.3.1 so that the sequence consists of two rolls of a die. Using the ordering scheme described above, the outcomes (in order) would be 11 12 13 . . . 66. The tags can then be generated using Equation (4.4). For example, the tag for the sequence 13 would be

$$\bar{T}_X(13) = P(\mathbf{x}=11) + P(\mathbf{x}=12) + 1/2P(\mathbf{x}=13) \tag{4.5}$$

$$= 1/36 + 1/36 + 1/2(1/36) \tag{4.6}$$

$$= 5/72. \tag{4.7}$$

♦

Notice that to generate the tag for 13 we did not have to generate a tag for every other possible message. However, based on Equation (4.4) and Example 4.3.3, we need to know the probability of every sequence that is "less than" the sequence for which the tag is being generated. The requirement that the probability of all sequences of a given length be explicitly calculated can be as prohibitive as the requirement that we have codewords for all sequences of a given length. Fortunately, we shall see that to compute a tag for a given sequence of symbols, all we need is the probability of individual symbols, or the probability model.

Recall that, given our construction, the interval containing the tag value for a given sequence is disjoint from the intervals containing the tag values of all other sequences. This means that any value in this interval would be a unique identifier for x_i. Therefore, to fulfill our initial objective of uniquely identifying each sequence, it would be sufficient to compute the upper and lower limits of the interval containing the tag and select any value in that interval. The upper and lower limits can be computed recursively as shown in the following example.

Example 4.3.4:

We will use the alphabet of Example 4.3.2 and find the upper and lower limits of the interval containing the tag for the sequence 322. Assume that we are observing 3 2 2 in a sequential manner; that is, first we see 3, then 2, and then 2 again. After each observation we will compute the upper and lower limits of the interval containing the tag of the sequence observed to that point. We will denote the upper limit by $u^{(n)}$ and the lower limit by $l^{(n)}$, where n denotes the length of the sequence.

We first observe 3. Therefore,

$$u^{(1)} = F_X(3), \qquad l^{(1)} = F_X(2).$$

We then observe 2 and the sequence is $\mathbf{x} = 32$. Therefore,

$$u^{(2)} = F_X^{(2)}(32), \qquad l^{(2)} = F_X^{(2)}(31).$$

We can compute these values as follows:

$$\begin{aligned} F_X^{(2)}(32) &= P(\mathbf{x} = 11) + P(\mathbf{x} = 12) + \cdots + P(\mathbf{x} = 16) \\ &\quad + P(\mathbf{x} = 21) + P(\mathbf{x} = 22) + \cdots + P(\mathbf{x} = 26) \\ &\quad + P(\mathbf{x} = 31) + P(\mathbf{x} = 32). \end{aligned}$$

But,

$$\sum_{i=1}^{i=6} P(\mathbf{x} = ki) = \sum_{i=1}^{i=6} P(x_1 = k, x_2 = i) = P(x_1 = k)$$

where $\mathbf{x} = x_1 x_2$. Therefore,

$$\begin{aligned} F_X^{(2)}(32) &= P(x_1 = 1) + P(x_1 = 2) + P(\mathbf{x} = 31) + P(\mathbf{x} = 32) \\ &= F_X(2) + P(\mathbf{x} = 31) + P(\mathbf{x} = 32). \end{aligned}$$

However, assuming each roll of the dice is independent of the others,

$$P(\mathbf{x} = 31) = P(x_1 = 3)P(x_2 = 1)$$

and

$$P(\mathbf{x} = 32) = P(x_1 = 3)P(x_2 = 2).$$

Therefore,

$$\begin{aligned} P(\mathbf{x}=31)+P(\mathbf{x}=32) &= P(x_1=3)(P(x_2=1)+P(x_2=2)) \\ &= P(x_1=3)F_X(2). \end{aligned}$$

Noting that

$$P(x_1=3)=F_X(3)-F_X(2)$$

we can write

$$P(\mathbf{x}=31)+P(\mathbf{x}=32)=(F_X(3)-F_X(2))F_X(2)$$

and

$$F_X^{(2)}(32)=F_X(2)+(F_X(3)-F_X(2))F_X(2).$$

We can also write this as

$$u^{(2)}=l^{(1)}+(u^{(1)}-l^{(1)})F_X(2).$$

We can similarly show that

$$F_X^{(2)}(31)=F_X(2)+(F_X(3)-F_X(2))F_X(1)$$

or

$$l^{(2)}=l^{(1)}+(u^{(1)}-l^{(1)})F_X(1).$$

The third element of the observed sequence is 2, and the sequence is $\mathbf{x}=322$. The upper and lower limits of the interval containing the tag for this sequence are

$$u^{(3)}=F_X^{(3)}(322), \qquad l^{(3)}=F_X^{(3)}(321).$$

Using the same approach as above we find that

$$\begin{aligned} F_X^{(3)}(322) &= F_X^{(2)}(31)+(F_X^{(2)}(32)-F_X^{(2)}(31))F_X(2) \\ F_X^{(3)}(321) &= F_X^{(2)}(31)+(F_X^{(2)}(32)-F_X^{(2)}(31))F_X(1) \end{aligned} \tag{4.8}$$

or

$$\begin{aligned} u^{(3)} &= l^{(2)}+(u^{(2)}-l^{(2)})F_X(2) \\ l^{(3)} &= l^{(2)}+(u^{(2)}-l^{(2)})F_X(1). \end{aligned}$$

♦

In general, we can show that for any sequence $\mathbf{x}=(x_1x_2\ldots x_n)$

$$l^{(n)} = l^{(n-1)}+(u^{(n-1)}-l^{(n-1)})F_X(x_n-1) \tag{4.9}$$

$$u^{(n)} = l^{(n-1)}+(u^{(n-1)}-l^{(n-1)})F_X(x_n). \tag{4.10}$$

Notice that throughout this process we did not explicitly need to compute any joint probabilities.

If we are using the midpoint of the interval for the tag, then

$$\bar{T}_X(\mathbf{x}) = \frac{u^{(n)} + l^{(n)}}{2}.$$

Therefore, the tag for any sequence can be computed in a sequential fashion. The only information required by the tag generation procedure is the *cdf* of the source, which can be obtained directly from the probability model.

Example 4.3.5: Generating a tag

Consider the source in Example 3.2.4. Define the random variable $X(a_i) = i$. Suppose we wish to encode the sequence **1 3 2 1**. From the probability model we know that

$$F_X(k) = 0,\ k \le 0, \quad F_X(1) = 0.8, \quad F_X(2) = 0.82, \quad F_X(3) = 1, \quad F_X(k) = 1,\ k > 3.$$

We can use Equations (4.9) and (4.10) sequentially to determine the lower and upper limits of the interval containing the tag. Initializing $u^{(0)}$ to 1, and $l^{(0)}$ to 0, the first element of the sequence **1** results in the following update:

$$\begin{aligned} l^{(1)} &= 0 + (1-0)0 = 0 \\ u^{(1)} &= 0 + (1-0)(0.8) = 0.8. \end{aligned}$$

That is, the tag is contained in the interval $[0, 0.8)$. The second element of the sequence is **3**. Using the update equations we get

$$\begin{aligned} l^{(2)} &= 0 + (0.8-0)F_X(2) = 0.8 \times 0.82 = 0.656 \\ u^{(2)} &= 0 + (0.8-0)F_X(3) = 0.8 \times 1.0 = 0.8. \end{aligned}$$

Therefore, the interval containing the tag for the sequence **1 3** is $[0.656, 0.8)$. The third element, **2**, results in the following update equations:

$$\begin{aligned} l^{(3)} &= 0.656 + (0.8-0.656)F_X(1) = 0.656 + 0.144 \times 0.8 = 0.7712 \\ u^{(3)} &= 0.656 + (0.8-0.656)F_X(2) = 0.656 + 0.144 \times 0.82 = 0.77408 \end{aligned}$$

and the interval for the tag is $[0.7712, 0.77408)$. Continuing with the last element, the upper and lower limits of the interval containing the tag are

$$\begin{aligned} l^{(4)} &= 0.7712 + (0.77408-0.7712)F_X(0) = 0.7712 + 0.00288 \times 0.0 = 0.7712 \\ u^{(4)} &= 0.7712 + (0.77408-0.1152)F_X(1) = 0.7712 + 0.00288 \times 0.8 = 0.773504 \end{aligned}$$

and the tag for the sequence **1 3 2 1** can be generated as

$$\bar{T}_X(1321) = \frac{0.7712 + 0.773504}{2} = 0.772352.$$

♦

Notice that each succeeding interval is contained in the preceding interval. If we examine the equations used to generate the intervals, we see that this will always be the case. This property will be used to decipher the tag. An undesirable consequence of this process is that the intervals get smaller and smaller and require higher precision as the sequence gets longer. To combat this problem, a rescaling strategy needs to be adopted. In Section 4.4.2, we will describe a simple rescaling approach that takes care of this problem.

4.3.2 Deciphering the Tag

We have spent a considerable amount of time showing how a sequence can be assigned a unique tag, given a minimal amount of information. However, the tag is useless unless we can also decipher it with minimal computational cost. Fortunately, deciphering the tag is as simple as generating it. We can see this most easily through an example.

Example 4.3.6: Deciphering a tag

Given the tag obtained in Example 4.3.5, let's try to obtain the sequence represented by the tag. We will try to mimic the encoder in order to do the decoding. The tag value is 0.772352. The interval containing this tag value is a subset of every interval obtained in the encoding process. Our decoding strategy will be to decode the elements in the sequence in such a way that the upper and lower limits $u^{(k)}$ and $l^{(k)}$ will always contain the tag value for each k. We start with $l^{(0)} = 0$ and $u^{(0)} = 1$. After decoding the first element of the sequence x_1, the upper and lower limits become

$$\begin{aligned} l^{(1)} &= 0 + (1-0)F_X(x_1 - 1) = F_X(x_1 - 1) \\ u^{(1)} &= 0 + (1-0)F_X(x_1) = F_X(x_1). \end{aligned}$$

In other words, the interval containing the tag is $[F_X(x_1 - 1), F_X(x_1))$. We need to find the value of x_1 for which 0.772352 lies in the interval $[F_X(x_1 - 1), F_X(x_1))$. If we pick $x_1 = \mathbf{1}$, the interval is $[0, 0.8)$. If we pick $x_1 = \mathbf{2}$, the interval is $[0.8, 0.82)$, and if we pick $x_1 = \mathbf{3}$, the interval is $[0.82, 1.0)$. As 0.772352 lies in the interval $[0.0, 0.8]$, we choose $x_1 = \mathbf{1}$. We now repeat this procedure for the second element x_2, using the updated values of $l^{(1)}$ and $u^{(1)}$:

$$\begin{aligned} l^{(2)} &= 0 + (0.8-0)F_X(x_2 - 1) = 0.8F_X(x_2 - 1) \\ u^{(2)} &= 0 + (0.8-0)F_X(x_2) = 0.8F_X(x_2). \end{aligned}$$

If we pick $x_2 = \mathbf{1}$, the updated interval is $[0, 0.64)$, which does not contain the tag. Therefore, x_2 cannot be $\mathbf{1}$. If we pick $x_2 = \mathbf{2}$, the updated interval is $[0.64, 0.656)$, which also does not contain the tag. If we pick $x_2 = \mathbf{3}$, the updated interval is $[0.656, 0.8)$, which does contain the tag value of 0.772352. Therefore, the second element in the sequence is $\mathbf{3}$. Knowing the second element of the sequence, we can update the values of $l^{(2)}$ and $u^{(2)}$ and find the element x_3, which will give us an interval containing the tag:

$$\begin{aligned} l^{(3)} &= 0.656 + (0.8-0.656)F_X(x_3 - 1) = 0.656 + 0.144 \times F_X(x_3 - 1) \\ u^{(3)} &= 0.656 + (0.8-0.656)F_X(x_3) = 0.656 + 0.144 \times F_X(x_3). \end{aligned}$$

However, the expressions for $l^{(3)}$ and $u^{(3)}$ are cumbersome in this form. To make the comparisons more easily, we could subtract the value of $l^{(2)}$ from both the limits and the tag. That is, we find the value of x_3 for which the interval $[0.144 \times F_X(x_3 - 1), 0.144 \times F_X(x_3))$ contains $0.772352 - 0.656 = 0.116352$. Or, we could make this even simpler and divide the residual tag value of 0.116352 by 0.144 to get 0.808, and find the value of x_3 for which 0.808 falls in the interval $[F_X(x_3 - 1), F_X(x_3))$. We can see that the only value of x_3 for which this is possible is **2**. Substituting **2** for x_3 in the update equations, we can update the values of $l^{(3)}$ and $u^{(3)}$. We can now find the element x_4 by computing the upper and lower limits as

$$\begin{aligned} l^{(4)} &= 0.7712 + (0.77408 - 0.7712)F_X(x_4 - 1) = 0.7712 + 0.00288 \times F_X(x_4 - 1) \\ u^{(4)} &= 0.7712 + (0.77408 - 0.1152)F_X(x_4) = 0.7712 + 0.00288 \times F_X(x_4). \end{aligned}$$

Again we can subtract $l^{(3)}$ from the tag to get $0.772352 - 0.7712 = 0.001152$ and find the value of x_4 for which the interval $[0.00288 \times F_X(x_4 - 1),\ 0.00288 \times F_X(x_4))$ contains 0.001152. To make the comparisons simpler, we can divide the residual value of the tag by 0.00288 to get 0.4, and find the value of x_4 for which 0.4 is contained in $[F_X(x_4 - 1), F_X(x_4))$. We can see that the value is $x_4 = \mathbf{1}$, and we have decoded the entire sequence. Note that we knew the length of the sequence beforehand and, therefore, we knew when to stop. ♦

From the example above, we can deduce an algorithm that can decipher the tag.

1. Initialize $l^{(0)} = 0$ and $u^{(0)} = 1$.
2. For each k find $t^* = (tag - l^{(k-1)})/(u^{(k-1)} - l^{(k-1)})$.
3. Find the value of x_k for which $F_X(x_k - 1) \le t^* < F_X(x_k)$.
4. Update $u^{(k)}$ and $l^{(k)}$.
5. Continue until the entire sequence has been decoded.

There are two ways to know when the entire sequence has been decoded. The decoder may know the length of the sequence, in which case the deciphering process is stopped when that many symbols have been obtained. The second way to know if the entire sequence has been decoded is that a particular symbol is denoted as an end-of-transmission symbol. The decoding of this symbol would bring the decoding process to a close.

4.4 Generating a Binary Code

Using the algorithm described in the previous section, we can obtain a tag for a given sequence **x**. However, the *binary code* for the sequence is what we really want to know. We want to find a binary code that will represent the sequence **x** in a unique and efficient manner.

We have said that the tag forms a unique representation for the sequence. This means that the binary representation of the tag forms a unique binary code for the sequence. However, we have placed no restrictions on what values in the unit interval the tag can take. The binary

representation of some of these values would be infinitely long, in which case, although the code is unique, it may not be efficient. To make the code efficient, the binary representation has to be truncated. But if we truncate the representation, is the resulting code still unique? Finally, is the resulting code efficient? How far or how close is the average number of bits per symbol from the entropy? We will examine all these questions in the next section.

Even if we show the code to be unique and efficient, the method described to this point is highly impractical. In Section 4.4.2, we will describe a more practical algorithm for generating the arithmetic code for a sequence. We will give an integer implementation of this algorithm in Section 4.4.3.

4.4.1 Uniqueness and Efficiency of the Arithmetic Code

$\bar{T}_X(x)$ is a number in the interval $[0, 1)$. A binary code for $\bar{T}_X(x)$ can be obtained by taking the binary representation of this number and truncating it to $l(x) = \lceil \log \frac{1}{P(x)} \rceil + 1$ bits.

Example 4.4.1:

Consider a source $\mathcal{A}$ that generates letters from an alphabet of size four,

$$\mathcal{A} = \{a_1, a_2, a_3, a_4\}$$

with probabilities

$$P(a_1) = \frac{1}{2}, \quad P(a_2) = \frac{1}{4}, \quad P(a_3) = \frac{1}{8}, \quad P(a_4) = \frac{1}{8}.$$

A binary code for this source can be generated as shown in Table 4.4. The quantity $\bar{T}_x$ is obtained using Equation (4.3). The binary representation of $\bar{T}_x$ is truncated to $\lceil \log \frac{1}{P(x)} \rceil + 1$ bits to obtain the binary code.

TABLE 4.4 A binary code for a four-letter alphabet.

Symbol	F_X	$\bar{T}_X$	In Binary	$\lceil \log \frac{1}{P(x)} \rceil + 1$	Code
1	.5	.25	.010	2	01
2	.75	.625	.101	3	101
3	.875	.8125	.1101	4	1101
4	1.0	.9375	.1111	4	1111

♦

We will show that a code obtained in this fashion is a uniquely decodable code. We first show that this code is unique, and then we will show that it is uniquely decodable.

Recall that while we have been using $\bar{T}_X(x)$ as the tag for a sequence $\mathbf{x}$, any number in the interval $[F_X(\mathbf{x}-1), F_X(\mathbf{x}))$ would be a unique identifier. Therefore, to show that the code $\lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})}$ is unique, all we need to do is show that it is contained in the interval

$[F_X(\mathbf{x}-1), F_X(\mathbf{x}))$. Because we are truncating the binary representation of $\bar{T}_X(\mathbf{x})$ to obtain $\lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})}$, $\lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})}$ is less than or equal to $\bar{T}_X(\mathbf{x})$. More specifically,

$$0 \le \bar{T}_X(\mathbf{x}) - \lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})} < \frac{1}{2^{l(\mathbf{x})}}. \tag{4.11}$$

As $\bar{T}_X(\mathbf{x})$ is strictly less than $F_X(\mathbf{x}))$,

$$\lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})} < F_X(\mathbf{x}).$$

To show that $\lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})} \ge F_X(\mathbf{x}-1)$, note that

$$\begin{aligned}
\frac{1}{2^{l(\mathbf{x})}} &= \frac{1}{2^{\lceil \log \frac{1}{P(x)} \rceil + 1}} \\
&< \frac{1}{2^{\log \frac{1}{P(x)} + 1}} \\
&= \frac{1}{2 \frac{1}{P(x)}} \\
&= \frac{P(\mathbf{x})}{2}.
\end{aligned}$$

From (4.3) we have

$$\frac{P(\mathbf{x})}{2} = \bar{T}_X(\mathbf{x}) - F_X(\mathbf{x}-1).$$

Therefore,

$$\bar{T}_X(\mathbf{x}) - F_X(\mathbf{x}-1) > \frac{1}{2^{l(\mathbf{x})}}. \tag{4.12}$$

Combining (4.11) and (4.12), we have

$$\lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})} > F_X(\mathbf{x}-1). \tag{4.13}$$

Therefore, the code $\lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})}$ is a unique representation of $\bar{T}_X(\mathbf{x})$.

To show that this code is uniquely decodable, we will show that the code is a prefix code; that is, no codeword is a prefix of another codeword. Because a prefix code is always uniquely decodable, by showing that an arithmetic code is a prefix code, we automatically show that it is uniquely decodable. Given a number a in the interval $[0, 1)$ with an n-bit binary representation $[b_1 b_2 \ldots b_n]$, for any other number b to have a binary representation with $[b_1 b_2 \ldots b_n]$ as the prefix, b has to lie in the interval $[a, a + \frac{1}{2^n})$. (See Problem 1.)

If $\mathbf{x}$ and $\mathbf{y}$ are two distinct sequences, we know that $\lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})}$ and $\lfloor \bar{T}_X(\mathbf{y}) \rfloor_{l(\mathbf{y})}$ lie in two *disjoint* intervals, $[F_X(\mathbf{x}-1), F_X(\mathbf{x}))$ and $[F_X(\mathbf{y}-1), F_X(\mathbf{y}))$. Therefore, if we can show that for any sequence $\mathbf{x}$, the interval $[\lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})}, \lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})} + \frac{1}{2^{l(\mathbf{x})}})$ lies entirely within the interval $[F_X(\mathbf{x}-1), F_X(\mathbf{x}))$, this will mean that the code for one sequence cannot be the prefix for the code for another sequence.

We have already shown that $\lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})} > F_X(\mathbf{x}-1)$. Therefore, all we need to do is show that

$$F_X(\mathbf{x}) - \lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})} > \frac{1}{2^{l(\mathbf{x})}}.$$

This is true because

$$\begin{aligned} F_X(\mathbf{x}) - \lfloor \bar{T}_X(\mathbf{x}) \rfloor_{l(\mathbf{x})} &> F_X(\mathbf{x}) - \bar{T}_X(\mathbf{x}) \\ &= \frac{P(\mathbf{x})}{2} \\ &> \frac{1}{2^{l(\mathbf{x})}}. \end{aligned}$$

This code is prefix free, and by taking the binary representation of $\bar{T}_X(\mathbf{x})$ and truncating it to $l(x) = \lceil \log \frac{1}{P(x)} \rceil + 1$ bits, we obtain a uniquely decodable code.

Although the code is uniquely decodable, how efficient is it? We have shown that the number of bits $l(\mathbf{x})$ required to represent $F_X(\mathbf{x})$ with enough accuracy such that the code for different values of $\mathbf{x}$ are distinct is

$$l(\mathbf{x}) = \left\lceil \log \frac{1}{P(\mathbf{x})} \right\rceil + 1.$$

Remember that $l(\mathbf{x})$ is the number of bits required to encode the *entire* sequence $\mathbf{x}$. So, the average length of an arithmetic code for a sequence of length m is given by

$$\begin{aligned} l_{A^m} &= \sum P(\mathbf{x}) l(\mathbf{x}) && (4.14) \\ &= \sum P(\mathbf{x}) \left[\left\lceil \log \frac{1}{P(\mathbf{x})} \right\rceil + 1 \right] && (4.15) \\ &< \sum P(\mathbf{x}) \left[\log \frac{1}{P(\mathbf{x})} + 1 + 1 \right] && (4.16) \\ &= -\sum P(\mathbf{x}) \log P(\mathbf{x}) + 2 \sum P(\mathbf{x}) && (4.17) \\ &= H(X^m) + 2. && (4.18) \end{aligned}$$

Given that the average length is always greater than the entropy, the bounds on $l_{A^{(m)}}$ are

$$H(X^{(m)}) \le l_{A^{(m)}} < H(X^{(m)}) + 2.$$

The length per symbol, l_A, or rate of the arithmetic code is $\frac{l_{A^{(m)}}}{m}$. Therefore, the bounds on l_A are

$$\frac{H(X^{(m)})}{m} \le l_A < \frac{H(X^{(m)})}{m} + \frac{2}{m}. \tag{4.19}$$

We have shown in Chapter 3 that for *iid* sources

$$H(X^{(m)}) = mH(X). \tag{4.20}$$

Therefore,

$$H(X) \leq l_A < H(X) + \frac{2}{m}. \tag{4.21}$$

By increasing the length of the sequence, we can guarantee a rate as close to the entropy as we desire.

4.4.2 Algorithm Implementation

In Section 4.3.1 we developed a recursive algorithm for the boundaries of the interval containing the tag for the sequence being encoded as

$$l^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F_X(x_n - 1) \tag{4.22}$$

$$u^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F_X(x_n) \tag{4.23}$$

where x_n is the value of the random variable corresponding to the nth observed symbol, $l^{(n)}$ is the lower limit of the tag interval at the nth iteration, and $u^{(n)}$ is the upper limit of the tag interval at the nth iteration.

Before we can implement this algorithm, there is one major problem we have to resolve. Recall that the rationale for using numbers in the interval $[0, 1)$ as a tag was that there are an infinite number of numbers in this interval. However, in practice the number of numbers that can be uniquely represented on a machine is limited by the maximum number of digits (or bits) we can use for representing the number. Consider the values of $l^{(n)}$ and $u^{(n)}$ in Example 4.3.5. As n gets larger, these values come closer and closer together. This means that in order to represent all the subintervals uniquely we need increasing precision as the length of the sequence increases. In a system with finite precision, the two values are bound to converge, and we will lose all information about the sequence from the point at which the two values converged. To avoid this situation, we need to rescale the interval. However, we have to do it in a way that will preserve the information that is being transmitted. We would also like to perform the encoding *incrementally*—that is, to transmit portions of the code as the sequence is being observed, rather than wait until the entire sequence has been observed before transmitting the first bit. The algorithm we describe in this section takes care of the problems of synchronized rescaling and incremental encoding.

As the interval becomes narrower, we have three possibilities:

1. The interval is entirely confined to the lower half of the unit interval $[0, 0.5)$.
2. The interval is entirely confined to the upper half of the unit interval $[0.5, 1.0)$.
3. The interval straddles the midpoint of the unit interval.

We will look at the third case a little later in this section. First, let us examine the first two cases. Once the interval is confined to either the upper or lower half of the unit interval, it is forever confined to that half of the unit interval. The most significant bit of the binary representation of all numbers in the interval $[0, 0.5)$ is 0, and the most significant bit of the binary representation of all numbers in the interval $[0.5, 1]$ is 1. Therefore, once the interval gets restricted to either the upper or lower half of the unit interval, the most significant bit of

the tag is fully determined. Therefore, without waiting to see what the rest of the sequence looks like, we can indicate to the decoder whether the tag is confined to the upper or lower half of the unit interval by sending a 1 for the upper half and a 0 for the lower half. The bit that we send is also the first bit of the tag.

Once the encoder and decoder know which half contains the tag, we can ignore the half of the unit interval not containing the tag and concentrate on the half containing the tag. As our arithmetic is of finite precision, we can do this best by mapping the half interval containing the tag to the full $[0, 1)$ interval. The mappings required are

$$E_1 : [0, 0.5) \to [0, 1); \qquad E_1(x) = 2x \tag{4.24}$$

$$E_2 : [0.5, 1) \to [0, 1); \qquad E_2(x) = 2(x - 0.5). \tag{4.25}$$

As soon as we perform either of these mappings, we lose all information about the most significant bit. However, this should not matter because we have already sent that bit to the decoder. We can now continue with this process, generating another bit of the tag every time the tag interval is restricted to either half of the unit interval. This process of generating the bits of the tag without waiting to see the entire sequence is called incremental encoding.

Example 4.4.2: Tag generation with scaling

Let's revisit Example 4.3.5. Recall that we wish to encode the sequence **1 3 2 1**. The probability model for the source is $P(a_1) = 0.8$, $P(a_2) = 0.02$, $P(a_3) = 0.18$. Initializing $u^{(0)}$ to 1, and $l^{(0)}$ to 0, the first element of the sequence, **1**, results in the following update:

$$\begin{aligned} l^{(1)} &= 0 + (1-0)0 = 0 \\ u^{(1)} &= 0 + (1-0)(0.8) = 0.8. \end{aligned}$$

The interval $[0, 0.8)$ is not confined to either the upper or lower half of the unit interval, so we proceed.

The second element of the sequence is **3**. This results in the update

$$\begin{aligned} l^{(2)} &= 0 + (0.8-0)F_X(2) = 0.8 \times 0.82 = 0.656 \\ u^{(2)} &= 0 + (0.8-0)F_X(3) = 0.8 \times 1.0 = 0.8. \end{aligned}$$

The interval $[0.656, 0.8)$ is contained entirely in the upper half of the unit interval, so we send the binary code 1 and rescale:

$$\begin{aligned} l^{(2)} &= 2 \times (0.656 - 0.5) = 0.312 \\ u^{(2)} &= 2 \times (0.8 - 0.5) = 0.6. \end{aligned}$$

The third element, **2**, results in the following update equations:

$$\begin{aligned} l^{(3)} &= 0.312 + (0.6 - 0.312)F_X(1) = 0.312 + 0.288 \times 0.8 = 0.5424 \\ u^{(3)} &= 0.312 + (0.8 - 0.312)F_X(2) = 0.312 + 0.288 \times 0.82 = 0.54816. \end{aligned}$$

The interval for the tag is $[0.5424, 0.54816)$, which is contained entirely in the upper half of the unit interval. We transmit a 1 and go through another rescaling:

$$\begin{aligned} l^{(3)} &= 2 \times (0.5424 - 0.5) = 0.0848 \\ u^{(3)} &= 2 \times (0.54816 - 0.5) = 0.09632. \end{aligned}$$

This interval is contained entirely in the lower half of the unit interval, so we send a 0 and use the E_1 mapping to rescale:

$$\begin{aligned} l^{(3)} &= 2 \times (0.0848) = 0.1696 \\ u^{(3)} &= 2 \times (0.09632) = 0.19264. \end{aligned}$$

The interval is still contained entirely in the lower half of the unit interval, so we send another 0 and go through another rescaling:

$$\begin{aligned} l^{(3)} &= 2 \times (0.1696) = 0.3392 \\ u^{(3)} &= 2 \times (0.19264) = 0.38528. \end{aligned}$$

Because the interval containing the tag remains in the lower half of the unit interval, we send another 0 and rescale one more time:

$$\begin{aligned} l^{(3)} &= 2 \times 0.3392 = 0.6784 \\ u^{(3)} &= 2 \times 0.38528 = 0.77056. \end{aligned}$$

Now the interval containing the tag is contained entirely in the upper half of the unit interval. Therefore, we transmit a 1 and rescale using the E_2 mapping:

$$\begin{aligned} l^{(3)} &= 2 \times (0.6784 - 0.5) = 0.3568 \\ u^{(3)} &= 2 \times (0.77056 - 0.5) = 0.54112. \end{aligned}$$

At each stage we are transmitting the most significant bit that is the same in both the upper and lower limit of the tag interval. If the most significant bits in the upper and lower limit are the same, then the value of this bit will be identical to the most significant bit of the tag. Therefore, by sending the most significant bits of the upper and lower endpoint of the tag whenever they are identical, we are actually sending the binary representation of the tag. The rescaling operations can be viewed as left shifts, which make the second most significant bit the most significant bit.

Continuing with the last element, the upper and lower limits of the interval containing the tag are

$$\begin{aligned} l^{(4)} &= 0.3568 + (0.54112 - 0.3568)F_X(0) = 0.3568 + 0.18422 \times 0.0 = 0.3568 \\ u^{(4)} &= 0.3568 + (0.54112 - 0.3568)F_X(1) = 0.3568 + 0.18422 \times 0.8 = 0.504256. \end{aligned}$$

At this point, if we wished to stop encoding, all we need to do is inform the receiver of the final status of the tag value. We can do so by sending the binary representation of any value in the final tag interval. Generally, this value is taken to be $l^{(n)}$. In this particular example, it is convenient to use the value of 0.5. The binary representation of 0.5 is .10 . . . . Thus, we would transmit a 1 followed by as many 0s as required by the word length of the implementation being used. ♦

Notice that the tag interval size at this stage is approximately 64 times the size it was when we were using the unmodified algorithm. Therefore, this technique solves the finite precision problem. As we shall soon see, the bits that we have been sending with each mapping constitute the tag itself, which satisfies our desire for incremental encoding. The binary sequence generated during the encoding process in the previous example is 1100011. We could simply treat this as the binary expansion of the tag. A binary number .1100011 corresponds to the decimal number 0.7734375. Looking back to Example 4.3.5, notice that this number lies within the final tag interval. Therefore, we could use this to decode the sequence.

However, we would like to do incremental decoding as well as incremental encoding. This raises three questions:

1. How do we start decoding?
2. How do we continue decoding?
3. How do we stop decoding?

The second question is the easiest to answer. Once we have started decoding, all we have to do is mimic the encoder algorithm. That is, once we have started decoding, we know how to continue decoding. To begin the decoding process, we need to have enough information to decode the first symbol unambiguously. In order to guarantee unambiguous decoding, the number of bits received should point to an interval smaller than the smallest tag interval. Based on the smallest tag interval, we can determine how many bits we need before we start the decoding procedure. We will demonstrate this procedure in Example 4.4.4. First let's look at other aspects of decoding using the message from Example 4.4.2.

Example 4.4.3:

We will use a word length of 6 for this example. Note that because we are dealing with real numbers this word length may not be sufficient for a different sequence. As in the encoder, we start with initializing $u^{(0)}$ to 1 and $l^{(0)}$ to 0. The sequence of received bits is 110001100 . . . 0. The first 6 bits correspond to a tag value of 0.765625, which means that the first element of the sequence is **1**, resulting in the following update:

$$\begin{aligned} l^{(1)} &= 0+(1-0)0=0 \\ u^{(1)} &= 0+(1-0)(0.8)=0.8. \end{aligned}$$

The interval $[0, 0.8)$ is not confined to either the upper or lower half of the unit interval, so we proceed. The tag 0.765625 lies in the top 18% of the interval $[0, 0.8)$; therefore, the second element of the sequence is **3**. Updating the tag interval we get

$$\begin{aligned} l^{(2)} &= 0 + (0.8 - 0)F_X(2) = 0.8 \times 0.82 = 0.656 \\ u^{(2)} &= 0 + (0.8 - 0)F_X(3) = 0.8 \times 1.0 = 0.8. \end{aligned}$$

The interval $[0.656, 0.8)$ is contained entirely in the upper half of the unit interval. At the encoder, we sent the bit 1 and rescaled. At the decoder, we will shift 1 out of the receive buffer and move the next bit in to make up the 6 bits in the tag. We will also update the tag interval, resulting in

$$\begin{aligned} l^{(2)} &= 2 \times (0.656 - 0.5) = 0.312 \\ u^{(2)} &= 2 \times (0.8 - 0.5) = 0.6 \end{aligned}$$

while shifting a bit to give us a tag of 0.546875. When we compare this value with the tag interval, we can see that this value lies in the 80–82% range of the tag interval, so we decode the next element of the sequence as **2**. We can then update the equations for the tag interval as

$$\begin{aligned} l^{(3)} &= 0.312 + (0.6 - 0.312)F_X(1) = 0.312 + 0.288 \times 0.8 = 0.5424 \\ u^{(3)} &= 0.312 + (0.8 - 0.312)F_X(2) = 0.312 + 0.288 \times 0.82 = 0.54816. \end{aligned}$$

As the tag interval is now contained entirely in the upper half of the unit interval, we rescale using E_2 to obtain

$$\begin{aligned} l^{(3)} &= 2 \times (0.5424 - 0.5) = 0.0848 \\ u^{(3)} &= 2 \times (0.54816 - 0.5) = 0.09632. \end{aligned}$$

We also shift out a bit from the tag and shift in the next bit. The tag is now 000110. The interval is contained entirely in the lower half of the unit interval. Therefore, we apply E_1 and shift another bit. The lower and upper limits of the tag interval become

$$\begin{aligned} l^{(3)} &= 2 \times (0.0848) = 0.1696 \\ u^{(3)} &= 2 \times (0.09632) = 0.19264 \end{aligned}$$

and the tag becomes 001100. The interval is still contained entirely in the lower half of the unit interval, so we shift out another 0 to get a tag of 011000 and go through another rescaling:

$$\begin{aligned} l^{(3)} &= 2 \times (0.1696) = 0.3392 \\ u^{(3)} &= 2 \times (0.19264) = 0.38528. \end{aligned}$$

Because the interval containing the tag remains in the lower half of the unit interval, we shift out another 0 from the tag to get 110000 and rescale one more time:

$$\begin{aligned} l^{(3)} &= 2 \times 0.3392 = 0.6784 \\ u^{(3)} &= 2 \times 0.38528 = 0.77056. \end{aligned}$$

Now the interval containing the tag is contained entirely in the upper half of the unit interval. Therefore, we shift out a 1 from the tag and rescale using the E_2 mapping:

$$\begin{aligned} l^{(3)} &= 2 \times (0.6784 - 0.5) = 0.3568 \\ u^{(3)} &= 2 \times (0.77056 - 0.5) = 0.54112. \end{aligned}$$

Now we compare the tag value to the the tag interval to decode our final element. The tag is 100000, which corresponds to 0.5. This value lies in the first 80% of the interval, so we decode this element as **1**. ♦

If the tag interval is entirely contained in the upper or lower half of the unit interval, the scaling procedure described will prevent the interval from continually shrinking. Now we consider the case where the diminishing tag interval straddles the midpoint of the unit interval. As our trigger for rescaling, we check to see if the tag interval is contained in the interval $[0.25, 0.75)$. This will happen when $l^{(n)}$ is greater than 0.25 and $u^{(n)}$ is less than 0.75. When this happens, we double the tag interval using the following mapping:

$$E_3 : [0.25, 0.75) \to [0, 1); \qquad E_3(x) = 2(x - 0.25). \tag{4.26}$$

We have used a 1 to transmit information about an E_2 mapping, and a 0 to transmit information about an E_1 mapping. How do we transfer information about an E_3 mapping to the decoder? We use a somewhat different strategy in this case. At the time of the E_3 mapping, we do not send any information to the decoder; instead, we simply record the fact that we have used the E_3 mapping at the encoder. Suppose that after this, the tag interval gets confined to the upper half of the unit interval. At this point we would use an E_2 mapping and send a 1 to the receiver. Note that the tag interval at this stage is at least twice what it would have been if we had not used the E_3 mapping. Furthermore, the upper limit of the tag interval would have been less than 0.75. Therefore, if the E_3 mapping had not taken place right before the E_2 mapping, the tag interval would have been contained entirely in the lower half of the unit interval. At this point we would have used an E_1 mapping and transmitted a 0 to the receiver. In fact, the effect of the earlier E_3 mapping can be mimicked at the decoder by following the E_2 mapping with an E_1 mapping. At the encoder, right after we send a 1 to announce the E_2 mapping, we send a 0 to help the decoder track the changes in the tag interval at the decoder. If the first rescaling after the E_3 mapping happens to be an E_1 mapping, we do exactly the opposite. That is, we follow the 0 announcing an E_1 mapping with a 1 to mimic the effect of the E_3 mapping at the encoder.

What happens if we have to go through a series of E_3 mappings at the encoder? We simply keep track of the number of E_3 mappings and then send that many bits of the opposite variety after the first E_1 or E_2 mapping. If we went through three E_3 mappings at the encoder,

followed by an E_2 mapping, we would transmit a 1 followed by three 0s. On the other hand, if we went through an E_1 mapping after the E_3 mappings, we would transmit a 0 followed by three 1s. Since the decoder mimics the encoder, the E_3 mappings are also applied at the decoder when the tag interval is contained in the interval $[0.25, 0.75)$.

4.4.3 Integer Implementation

We have described a floating-point implementation of arithmetic coding. Let us now repeat the procedure using integer arithmetic and generate the binary code in the process.

Encoder Implementation

The first thing we have to do is decide on the word length to be used. Given a word length of m, we map the important values in the $[0, 1)$ interval to the range of 2^m binary words. The point 0 gets mapped to

$$\overbrace{00\ldots0}^{m \text{ times}},$$

1 gets mapped to

$$\overbrace{11\ldots1}^{m \text{ times}}.$$

The value of 0.5 gets mapped to

$$1\overbrace{00\ldots0}^{m-1 \text{ times}}.$$

The update equations remain almost the same as Equations (4.9) and (4.10). As we are going to do integer arithmetic, we need to replace $F_X(x)$ in these equations.

Define n_j as the number of times the symbol j occurs in a sequence of length *Total Count*. Then $F_X(k)$ can be estimated by

$$F_X(k) = \frac{\sum_{i=1}^{k} n_i}{Total\ Count}. \tag{4.27}$$

If we now define

$$Cum\_Count(k) = \sum_{i=1}^{k} n_i$$

we can write Equations (4.9) and (4.10) as

$$l^{(n)} = l^{(n-1)} + \left\lfloor \frac{(u^{(n-1)} - l^{(n-1)} + 1) \times Cum\_Count(x_n - 1)}{Total\ Count} \right\rfloor \tag{4.28}$$

$$u^{(n)} = l^{(n-1)} + \left\lfloor \frac{(u^{(n-1)} - l^{(n-1)} + 1) \times Cum\_Count(x_n)}{Total\ Count} \right\rfloor - 1 \tag{4.29}$$

where x_n is the nth symbol to be encoded, $\lfloor x \rfloor$ is the largest integer less than or equal to x, and where the addition and subtraction of one is to handle the effects of the integer arithmetic.

Because of the way we mapped the endpoints and the halfway points of the unit interval, when both $l^{(n)}$ and $u^{(n)}$ are in either the upper half or lower half of the interval, the leading bit of $u^{(n)}$ and $l^{(n)}$ will be the same. If the leading or most significant bit (MSB) is 1, then the tag interval is contained entirely in the upper half of the $[00\ldots0, 11\ldots1]$ interval. If the MSB is 0, then the tag interval is contained entirely in the lower half. Applying the E_1 and E_2 mappings is a simple matter. All we do is shift out the MSB and then shift in a 1 into the integer code for $u^{(n)}$ and a 0 into the code for $l^{(n)}$. For example, suppose m was 6, $u^{(n)}$ was 54, and $l^{(n)}$ was 33. The binary representations of $u^{(n)}$ and $l^{(n)}$ are 110110 and 100001, respectively. Notice that the MSB for both endpoints is 1. Following the procedure above, we would shift out (and transmit or store) the 1, and shift in 1 for $u^{(n)}$ and 0 for $l^{(n)}$, obtaining the new value for $u^{(n)}$ as 101101, or 45, and a new value for $l^{(n)}$ as 000010, or 2. This is equivalent to performing the E_2 mapping. We can see how the E_1 mapping would also be performed using the same operation.

To see if the E_3 mapping needs to be performed, we monitor the second most significant bit of $u^{(n)}$ and $l^{(n)}$. When the second most significant bit of $u^{(n)}$ is 0 and the second most significant bit of $l^{(n)}$ is 1, this means that the tag interval lies in the middle half of the $[00\ldots0, 11\ldots1]$ interval. To implement the E_3 mapping, we complement the second most significant bit in $u^{(n)}$ and $l^{(n)}$, and shift left, shifting in a 1 in $u^{(n)}$ and a 0 in $l^{(n)}$. We also keep track of the number of E_3 mappings in Scale3.

We can summarize the encoding algorithm using the following pseudocode:

```
Initialize l and u.
Get symbol.
```

$$l \longleftarrow l + \left\lfloor \frac{(u-l+1)\times Cum\_Count(x-1)}{TotalCount} \right\rfloor$$

$$u \longleftarrow l + \left\lfloor \frac{(u-l+1)\times Cum\_Count(x)}{TotalCount} \right\rfloor - 1$$

```
while(MSB of u and l are both equal to b or E3 condition holds)
if(MSB of u and l are both equal to b)

    {
    send b
    shift l to the left by 1 bit and shift 0 into LSB
    shift u to the left by 1 bit and shift 1 into LSB
    while(Scale3 > 0)
        {
        send complement of b
        decrement Scale3
        }
    }
```

```
if(E3 condition holds)

    {
    shift l to the left by 1 bit and shift 0 into LSB
    shift u to the left by 1 bit and shift 1 into LSB
    complement (new) MSB of l and u
    increment Scale3
    }
```

To see how all this functions together, let's look at an example.

Example 4.4.4:

We will encode the sequence **1 3 2 1** with parameters shown in Table 4.5. First we need to select the word length m. Note that $Cum\_Count(1)$ and $Cum\_Count(2)$ differ by only 1. Recall that the values of Cum_Count will get translated to the endpoints of the subintervals. We want to make sure that the value we select for the word length will allow enough range for it to be possible to represent the smallest difference between the endpoints of intervals. We always rescale whenever the interval gets small. In order to make sure that the endpoints of the intervals always remain distinct, we need to make sure that all values in the range from 0 to $Total\_Count$, which is the same as $Cum\_Count(3)$, are uniquely represented in the smallest range an interval under consideration can be without triggering a rescaling. The interval is smallest without triggering a rescaling when $l^{(n)}$ is just below the midpoint of the interval and $u^{(n)}$ is at three-quarters of the interval, or when $u^{(n)}$ is right at the midpoint of the interval and $l^{(n)}$ is just below a quarter of the interval. That is, the smallest the interval $[l^{(n)}, u^{(n)}]$ can be is one-quarter of the total available range of 2^m values. Thus, m should be large enough to accommodate uniquely the set of values between 0 and $Total\_Count$.

TABLE 4.5 Values of some of the parameters for arithmetic coding example.

$Count(1) = 40$	$Cum\_Count(0) = 0$	Scale3 = 0
$Count(2) = 1$	$Cum\_Count(1) = 40$	
$Count(3) = 9$	$Cum\_Count(2) = 41$	
$Total\_Count = 50$	$Cum\_Count(3) = 50$	

For this example, this means that the total interval range has to be greater than 200. A value of $m = 8$ satisfies this requirement.

With this value of m we have

$$l^{(0)} = 0 = (00000000)_2 \tag{4.30}$$

$$u^{(0)} = 255 = (11111111)_2 \tag{4.31}$$

where $(\cdots)_2$ is the binary representation of a number.

The first element of the sequence to be encoded is **1**. Using Equations (4.28) and (4.29),

$$l^{(1)} = 0 + \left\lfloor \frac{256 \times Cum\_Count(0)}{50} \right\rfloor = 0 = (00000000)_2 \tag{4.32}$$

$$u^{(1)} = 0 + \left\lfloor \frac{256 \times Cum\_Count(1)}{50} \right\rfloor - 1 = 203 = (11001011)_2. \tag{4.33}$$

The next element of the sequence is **3**.

$$l^{(2)} = 0 + \left\lfloor \frac{204 \times Cum\_Count(2)}{50} \right\rfloor = 167 = (10100111)_2 \tag{4.34}$$

$$u^{(2)} = 0 + \left\lfloor \frac{204 \times Cum\_Count(3)}{50} \right\rfloor - 1 = 203 = (11001011)_2 \tag{4.35}$$

The MSBs of $l^{(2)}$ and $u^{(2)}$ are both 1. Therefore, we shift this value out and send it to the decoder. All other bits are shifted left by 1 bit, giving

$$l^{(2)} = (01001110)_2 = 78 \tag{4.36}$$

$$u^{(2)} = (10010111)_2 = 151. \tag{4.37}$$

Notice that while the MSBs of the limits are different, the second MSB of the upper limit is 0, while the second MSB of the lower limit is 1. This is the condition for the E_3 mapping. We complement the second MSB of both limits and shift 1 bit to the left, shifting in a 0 as the LSB of $l^{(2)}$ and a 1 as the LSB of $u^{(2)}$. This gives us

$$l^{(2)} = (00011100)_2 = 28 \tag{4.38}$$

$$u^{(2)} = (10101111)_2 = 175. \tag{4.39}$$

We also increment Scale3 to a value of 1.

The next element in the sequence is **2**. Updating the limits, we have

$$l^{(3)} = 28 + \left\lfloor \frac{148 \times Cum\_Count(1)}{50} \right\rfloor = 146 = (10010010)_2 \tag{4.40}$$

$$u^{(3)} = 28 + \left\lfloor \frac{148 \times Cum\_Count(2)}{50} \right\rfloor - 1 = 148 = (10010100)_2. \tag{4.41}$$

The two MSBs are identical, so we shift out a 1 and shift left by 1 bit:

$$l^{(3)} = (00100100)_2 = 36 \tag{4.42}$$

$$u^{(3)} = (00101001)_2 = 41. \tag{4.43}$$

As Scale3 is 1, we transmit a 0 and decrement Scale3 to 0. The MSBs of the upper and lower limits are both 0, so we shift out and transmit 0:

$$l^{(3)} = (01001000)_2 = 72 \tag{4.44}$$

$$u^{(3)} = (01010011)_2 = 83. \tag{4.45}$$

Both MSBs are again 0, so we shift out and transmit 0:

$$l^{(3)} = (10010000)_2 = 144 \tag{4.46}$$

$$u^{(3)} = (10100111)_2 = 167. \tag{4.47}$$

Now both MSBs are 1, so we shift out and transmit a 1. The limits become

$$l^{(3)} = (00100000)_2 = 32 \tag{4.48}$$

$$u^{(3)} = (01001111)_2 = 79. \tag{4.49}$$

Once again the MSBs are the same. This time we shift out and transmit a 0.

$$l^{(3)} = (01000000)_2 = 64 \tag{4.50}$$

$$u^{(3)} = (10011111)_2 = 159. \tag{4.51}$$

Now the MSBs are different. However, the second MSB for the lower limit is 1 while the second MSB for the upper limit is 0. This is the condition for the E_3 mapping. Applying the E_3 mapping by complementing the second MSB and shifting 1 bit to the left, we get

$$l^{(3)} = (00000000)_2 = 0 \tag{4.52}$$

$$u^{(3)} = (10111111)_2 = 191. \tag{4.53}$$

We also increment Scale3 to 1.

The next element in the sequence to be encoded is **1**. Therefore,

$$l^{(4)} = 0 + \left\lfloor \frac{192 \times Cum\_Count(0)}{50} \right\rfloor = 0 = (00000000)_2 \tag{4.54}$$

$$u^{(4)} = 0 + \left\lfloor \frac{192 \times Cum\_Count(1)}{50} \right\rfloor - 1 = 152 = (10011000)_2. \tag{4.55}$$

The encoding continues in this fashion. To this point we have generated the binary sequence 1100010. If we wished to terminate the encoding at this point, we have to send the current status of the tag. This can be done by sending the value of the lower limit $l^{(4)}$. As $l^{(4)}$ is 0, we will end up sending eight 0s. However, Scale3 at this point is 1. Therefore, after we send the first 0 from the value of $l^{(4)}$, we need to send a 1 before sending the remaining seven 0s. The final transmitted sequence is 1100010010000000. ♦

Decoder Implementation

Once we have the encoder implementation, the decoder implementation is easy to describe. As mentioned earlier, once we have started decoding all we have to do is mimic the encoder algorithm. Let us first describe the decoder algorithm using pseudocode and then study its implementation using Example 4.4.5.

Decoder Algorithm

Initialize l and u.
Read the first m bits of the received bitstream into tag t.
$k = 0$
while$\left(\left\lfloor \frac{(t-l+1)\times Total\ Count - 1}{u-l+1} \right\rfloor \geq Cum_Count(k)\right)$
$k \longleftarrow k+1$
decode symbol x.
$l \longleftarrow l + \left\lfloor \frac{(u-l+1)\times Cum\_Count(x-1)}{Total\ Count} \right\rfloor$
$u \longleftarrow l + \left\lfloor \frac{(u-l+1)\times Cum\_Count(x)}{Total\ Count} \right\rfloor - 1$
while(MSB of u and l are both equal to b or E_3 condition holds)
if(MSB of u and l are both equal to b)

{
shift l to the left by 1 bit and shift 0 into LSB
shift u to the left by 1 bit and shift 1 into LSB
shift t to the left by 1 bit and read next bit from received bitstream into LSB
}

if(E_3 condition holds)

{
shift l to the left by 1 bit and shift 0 into LSB
shift u to the left by 1 bit and shift 1 into LSB
shift t to the left by 1 bit and read next bit from received bitstream into LSB
complement (new) MSB of l, u, and t
}

Example 4.4.5:

After encoding the sequence in Example 4.4.4, we ended up with the following binary sequence: 1100010010000000. Treating this as the received sequence and using the parameters from Table 4.5, let us decode this sequence. Using the same word length, eight, we read in the first 8 bits of the received sequence to form the tag t:

$$t = (11000100)_2 = 196.$$

We initialize the lower and upper limits as

$$l = (00000000)_2 = 0$$
$$u = (11111111)_2 = 255.$$

To begin decoding, we compute

$$\left\lfloor \frac{(t-l+1)\times Total\ Count-1}{u-l+1} \right\rfloor = \left\lfloor \frac{197\times 50-1}{255-0+1} \right\rfloor = 38$$

and compare this value to

$$Cum\_Count = \begin{bmatrix} 0 \\ 40 \\ 41 \\ 50 \end{bmatrix}$$

Since

$$0 \le 38 < 40,$$

we decode the first symbol as **1**. Once we have decoded a symbol, we update the lower and upper limits:

$$\begin{aligned} l &= 0+\left\lfloor \frac{256\times Cum\_Count[0]}{Total\ Count} \right\rfloor = 0+\left\lfloor 256\times \frac{0}{50} \right\rfloor = 0 \\ u &= 0+\left\lfloor \frac{256\times Cum\_Count[1]}{Total\ Count} \right\rfloor - 1 = 0+\left\lfloor 256\times \frac{40}{50} \right\rfloor - 1 = 203 \end{aligned}$$

or

$$\begin{aligned} l &= (00000000)_2 \\ u &= (11001011)_2. \end{aligned}$$

The MSB of the limits are different and the E_3 condition does not hold. Therefore, we continue decoding without modifying the tag value. To obtain the next symbol, we compare

$$\left\lfloor \frac{(t-l+1)\times Total\ Count-1}{u-l+1} \right\rfloor$$

which is 48, against the *Cum_Count* array:

$$Cum\_Count[2] \le 48 < Cum\_Count[3].$$

Therefore, we decode **3** and update the limits:

$$\begin{aligned} l &= 0+\left\lfloor \frac{204\times Cum\_Count[2]}{Total\ Count} \right\rfloor = 0+\left\lfloor 204\times \frac{41}{50} \right\rfloor = 167 = (1010011)_2 \\ u &= 0+\left\lfloor \frac{204\times Cum\_Count[3]}{Total\ Count} \right\rfloor - 1 = 0+\left\lfloor 204\times \frac{50}{50} \right\rfloor - 1 = 203 = (11001011)_2. \end{aligned}$$

As the MSB of u and l are the same, we shift the MSB out and read in a 0 for the LSB of l and a 1 for the LSB of u. We mimic this action for the tag as well, shifting the MSB out and reading in the next bit from the received bitstream as the LSB:

$$\begin{aligned} l &= (01001110)_2 \\ u &= (10010111)_2 \\ t &= (10001001)_2. \end{aligned}$$

Examining l and u we can see we have an E_3 condition. Therefore, for l, u, and t, we shift the MSB out, complement the new MSB, and read in a 0 as the LSB of l, a 1 as the LSB of u, and the next bit in the received bitstream as the LSB of t. We now have

$$\begin{aligned} l &= (00011100)_2 = 28 \\ u &= (10101111)_2 = 175 \\ t &= (10010010)_2 = 146. \end{aligned}$$

To decode the next symbol, we compute

$$\left\lfloor \frac{(t-l+1)\times Total\ Count-1}{u-l+1} \right\rfloor = 40.$$

Since $40 \leq 40 < 41$, we decode **2**.

Updating the limits using this decoded symbol, we get

$$\begin{aligned} l &= 28 + \left\lfloor \frac{(175-28+1)\times 40}{50} \right\rfloor = 146 = (10010010)_2 \\ u &= 28 + \left\lfloor \frac{(175-28+1)\times 41}{50} \right\rfloor - 1 = 148 = (10010100)_2. \end{aligned}$$

We can see that we have quite a few bits to shift out. However, notice that the lower limit l has the same value as the tag t. Furthermore, the remaining received sequence consists entirely of 0s. Therefore, we will be performing identical operations on numbers that are the same, resulting in identical numbers. This will result in the final decoded symbol being **1**. We knew this was the final symbol to be decoded because only four symbols had been encoded. In practice this information has to be conveyed to the decoder. ♦

4.5 Comparison of Huffman and Arithmetic Coding

We have described a new coding scheme that, although more complicated than Huffman coding, allows us to code *sequences* of symbols. How well this coding scheme works depends on how it is used. Let's first try to use this code for encoding sources for which we know the Huffman code.

Looking at Example 4.4.1, the average length for this code is

$$l = 2 \times 0.5 + 3 \times 0.25 + 4 \times 0.125 + 4 \times 0.125 \tag{4.56}$$

$$= 2.75 \text{ bits/symbol.} \tag{4.57}$$

Recall from Section 2.4 that the entropy of this source was 1.75 bits/symbol and the Huffman code achieved this entropy. Obviously, arithmetic coding is not a good idea if you are going to encode your message one symbol at a time. Let's repeat the example with messages consisting of two symbols. (Note that we are only doing this to demonstrate a point. In practice, we would not code sequences this short using an arithmetic code.)

Example 4.5.1:

If we encode two symbols at a time, the resulting code is shown in Table 4.6.

TABLE 4.6 Arithmetic code for two-symbol sequences.

Message	$P(x)$	$\bar{T}_X(x)$	$\bar{T}_X(x)$ in Binary	$\lceil \log \frac{1}{P(x)} \rceil + 1$	Code
11	.25	.125	.001	3	001
12	.125	.3125	.0101	4	0101
13	.0625	.40625	.01101	5	01101
14	.0625	.46875	.01111	5	01111
21	.125	.5625	.1001	4	1001
22	.0625	.65625	.10101	5	10101
23	.03125	.703125	.101101	6	101101
24	.03125	.734375	.101111	6	101111
31	.0625	.78125	.11001	5	11001
32	.03125	.828125	.110101	6	110101
33	.015625	.8515625	.1101101	7	1101101
34	.015625	.8671875	.1101111	7	1101111
41	.0625	.90625	.11101	5	11101
42	.03125	.953125	.111101	6	111101
43	.015625	.9765625	.1111101	7	1111101
44	.015625	.984375	.1111111	7	1111111

The average length per message is 4.5 bits. Therefore, using two symbols at a time we get a rate of 2.25 bits/symbol (certainly better than 2.75 bits/symbol, but still not as good as the best rate of 1.75 bits/symbol). However, we see that as we increase the number of symbols per message, our results get better and better. ♦

How many samples do we have to group together to make the arithmetic coding scheme perform better than the Huffman coding scheme? We can get some idea by looking at the bounds on the coding rate.

Recall that the bounds on the average length l_A of the arithmetic code are

$$H(X) \leq l_A \leq H(X) + \frac{2}{m}.$$

It does not take many symbols in a sequence before the coding rate for the arithmetic code becomes quite close to the entropy. However, recall that for Huffman codes, if we block m symbols together, the coding rate is

$$H(X) \leq l_H \leq H(X) + \frac{1}{m}.$$

The advantage seems to lie with the Huffman code, although the advantage decreases with increasing m. However, remember that to generate a codeword for a sequence of length m, using the Huffman procedure requires building the entire code for all possible sequences of length m. If the original alphabet size was k, then the size of the codebook would be k^m. Taking relatively reasonable values of $k = 16$ and $m = 20$ gives a codebook size of 16^{20}! This is obviously not a viable option. For the arithmetic coding procedure, we do not need to build the entire codebook. Instead, we simply obtain the code for the tag corresponding to a given sequence. Therefore, it is entirely feasible to code sequences of length 20 or much more. In practice, we can make m large for the arithmetic coder and not for the Huffman coder. This means that for most sources we can get rates closer to the entropy using arithmetic coding than by using Huffman coding. The exceptions are sources whose probabilities are powers of two. In these cases, the single-letter Huffman code achieves the entropy, and we cannot do any better with arithmetic coding, no matter how long a sequence we pick.

The amount of gain also depends on the source. Recall that for Huffman codes we are guaranteed to obtain rates within $0.086 + p_{\max}$ of the entropy, where $p_{\max}$ is the probability of the most probable letter in the alphabet. If the alphabet size is relatively large and the probabilities are not too skewed, the maximum probability $p_{\max}$ is generally small. In these cases, the advantage of arithmetic coding over Huffman coding is small, and it might not be worth the extra complexity to use arithmetic coding rather than Huffman coding. However, there are many sources, such as facsimile, in which the alphabet size is small, and the probabilities are highly unbalanced. In these cases, the use of arithmetic coding is generally worth the added complexity.

Another major advantage of arithmetic coding is that it is easy to implement a system with multiple arithmetic codes. This may seem contradictory, as we have claimed that arithmetic coding is more complex than Huffman coding. However, it is the computational machinery that causes the increase in complexity. Once we have the computational machinery to implement one arithmetic code, all we need to implement more than a single arithmetic code is the availability of more probability tables. If the alphabet size of the source is small, as in the case of a binary source, there is very little added complexity indeed. In fact, as we shall see in the next section, it is possible to develop multiplication-free arithmetic coders that are quite simple to implement (nonbinary multiplication-free arithmetic coders are described in [44]).

Finally, it is much easier to adapt arithmetic codes to changing input statistics. All we need to do is estimate the probabilities of the input alphabet. This can be done by keeping a count of the letters as they are coded. There is no need to preserve a tree, as with adaptive Huffman codes. Furthermore, there is no need to generate a code a priori, as in the case of

Huffman coding. This property allows us to separate the modeling and coding procedures in a manner that is not very feasible with Huffman coding. This separation permits greater flexibility in the design of compression systems, which can be used to great advantage.

4.6 Adaptive Arithmetic Coding

We have seen how to construct arithmetic coders when the distribution of the source, in the form of cumulative counts, is available. In many applications such counts are not available a priori. It is a relatively simple task to modify the algorithms discussed so that the coder learns the distribution as the coding progresses. A straightforward implementation is to start out with a count of 1 for each letter in the alphabet. We need a count of at least 1 for each symbol, because if we do not we will have no way of encoding the symbol when it is first encountered. This assumes that we know nothing about the distribution of the source. If we do know something about the distribution of the source, we can let the initial counts reflect our knowledge.

After coding is initiated, the count for each letter encountered is incremented *after* that letter has been encoded. The cumulative count table is updated accordingly. It is very important that the updating take place after the encoding; otherwise the decoder will not be using the same cumulative count table as the encoder to perform the decoding. At the decoder, the count and cumulative count tables are updated after each letter is decoded.

In the case of the static arithmetic code, we picked the size of the word based on Total Count, the total number of symbols to be encoded. In the adaptive case, we may not know ahead of time what the total number of symbols is going to be. In this case we have to pick the word length independent of the total count. However, given a word length m we know that we can only accomodate a total count of 2^{m-2} or less. Therefore, during the encoding and decoding processes when the total count approaches 2^{m-2}, we have to go through a rescaling, or renormalization, operation. A simple rescaling operation is to divide all counts by 2 and rounding up the result so that no count gets rescaled to zero. This periodic rescaling can have an added benefit in that the count table better reflects the local statisitcs of the source.

4.7 Applications

Arithmetic coding is used in a variety of lossless and lossy compression applications. It is a part of many international standards. In the area of multimedia there are a few principal organizations that develop standards. The International Standards Organization (ISO) and the International Electrotechnical Commission (IEC) are industry groups that work on multimedia standards, while the International Telecommunications Union (ITU), which is part of the United Nations, works on multimedia standards on behalf of the member states of the United Nations. Quite often these institutions work together to create international standards. In later chapters we will be looking at a number of these standards, and we will see how arithmetic coding is used in image compression, audio compression, and video compression standards.

For now let us look at the lossless compression example from the previous chapter.

TABLE 4.7 Compression using adaptive arithmetic coding of pixel values.

Image Name	Bits/Pixel	Total Size (bytes)	Compression Ratio (arithmetic)	Compression Ratio (Huffman)
Sena	6.52	53,431	1.23	1.16
Sensin	7.12	58,306	1.12	1.27
Earth	4.67	38,248	1.71	1.67
Omaha	6.84	56,061	1.17	1.14

TABLE 4.8 Compression using adaptive arithmetic coding of pixel differences.

Image Name	Bits/Pixel	Total Size (bytes)	Compression Ratio (arithmetic)	Compression Ratio (Huffman)
Sena	3.89	31,847	2.06	2.08
Sensin	4.56	37,387	1.75	1.73
Earth	3.92	32,137	2.04	2.04
Omaha	6.27	51,393	1.28	1.26

In Tables 4.7 and 4.8, we show the results of using adaptive arithmetic coding to encode the same test images that were previously encoded using Huffman coding. We have included the compression ratios obtained using Huffman code from the previous chapter for comparison. Comparing these values to those obtained in the previous chapter, we can see very little change. The reason is that beacuse the alphabet size for the images is quite large, the value of $p_{\max}$ is quite small, and in the Huffman coder performs very close to the entropy.

As we mentioned before, a major advantage of arithmetic coding over Huffman coding is the ability to separate the modeling and coding aspects of the compression approach. In terms of image coding, this allows us to use a number of different models that take advantage of local properties. For example, we could use different decorrelation strategies in regions of the image that are quasi-constant and will, therefore, have differences that are small, and in regions where there is a lot of activity, causing the presence of larger difference values.

4.8 Summary

In this chapter we introduced the basic ideas behind arithmetic coding. We have shown that the arithmetic code is a uniquely decodable code that provides a rate close to the entropy for long stationary sequences. This ability to encode sequences directly instead of as a concatenation of the codes for the elements of the sequence makes this approach more efficient than Huffman coding for alphabets with highly skewed probabilities. We have looked in some detail at the implementation of the arithmetic coding approach.

The arithmetic coding results in this chapter were obtained by using the program provided by Witten, Neal, and Cleary [45]. This code can be used (with some modifications) for exploring different aspects of arithmetic coding (see problems).

Further Reading

1. The book *Text Compression*, by T.C. Bell, J.G. Cleary, and I.H. Witten [1], contains a very readable section on arithmetic coding, complete with pseudocode and C code.
2. A thorough treatment of various aspects of arithmetic coding can be found in the excellent chapter *Arithmetic Coding*, by Amir Said [46] in the *Lossless Compression Handbook*.
3. There is an excellent tutorial article by G.G. Langdon, Jr. [47] in the March 1984 issue of the *IBM Journal of Research and Development*.
4. The separate model and code paradigm is explored in a precise manner in the context of arithmetic coding in a paper by J.J. Rissanen and G.G. Langdon [48].
5. The separation of modeling and coding is exploited in a very nice manner in an early paper by G.G. Langdon and J.J. Rissanen [49].
6. Various models for text compression that can be used effectively with arithmetic coding are described by T.G. Bell, I.H. Witten, and J.G. Cleary [50] in an article in the *ACM Computing Surveys*.
7. The coder used in the JBIG algorithm is a descendant of the Q coder, described in some detail in several papers [51, 52, 53] in the November 1988 issue of the *IBM Journal of Research and Development*.

4.9 Projects and Problems

1. Given a number a in the interval $[0, 1)$ with an n-bit binary representation $[b_1 b_2 \ldots b_n]$, show that for any other number b to have a binary representation with $[b_1 b_2 \ldots b_n]$ as the prefix, b has to lie in the interval $[a, a + \frac{1}{2^n})$.
2. The binary arithmetic coding approach specified in the JBIG standard can be used for coding gray-scale images via *bit plane encoding*. In bit plane encoding, we combine the most significant bits for each pixel into one bit plane, the next most significant bits into another bit plane, and so on. Use the function `extrctbp` to obtain eight bit planes for the `sena.img` and `omaha.img` test images, and encode them using arithmetic coding. Use the low-resolution contexts shown in Figure 7.11.
3. Bit plane encoding is more effective when the pixels are encoded using a *Gray code*. The Gray code assigns numerically adjacent values binary codes that differ by only 1 bit. To convert from the standard binary code $b_0 b_1 b_2 \ldots b_7$ to the Gray code $g_0 g_1 g_2 \ldots g_7$, we can use the equations

$$g_0 = b_0$$
$$g_k = b_k \oplus b_{k-1}.$$

Convert the test images `sena.img` and `omaha.img` to a Gray code representation, and bit plane encode. Compare with the results for the non-Gray-coded representation.

TABLE 4.9 Probability model for Problems 5 and 6.

Letter	Probability
a_1	.2
a_2	.3
a_3	.5

TABLE 4.10 Frequency counts for Problem 7.

Letter	Count
a	37
b	38
c	25

4. In Example 4.4.4, repeat the encoding using $m = 6$. Comment on your results.

5. Given the probability model in Table 4.9, find the real valued tag for the sequence $a_1\ a_1\ a_3\ a_2\ a_3\ a_1$.

6. For the probability model in Table 4.9, decode a sequence of length 10 with the tag 0.63215699.

7. Given the frequency counts shown in Table 4.10:

(a) What is the word length required for unambiguous encoding?

(b) Find the binary code for the sequence *abacabb*.

(c) Decode the code you obtained to verify that your encoding was correct.

8. Generate a binary sequence of length L with $P(0) = 0.8$, and use the arithmetic coding algorithm to encode it. Plot the difference of the rate in bits/symbol and the entropy as a function of L. Comment on the effect of L on the rate.

5

Dictionary Techniques

5.1 Overview

In the previous two chapters we looked at coding techniques that assume a source that generates a sequence of independent symbols. As most sources are correlated to start with, the coding step is generally preceded by a decorrelation step. In this chapter we will look at techniques that incorporate the structure in the data in order to increase the amount of compression. These techniques—both static and adaptive (or dynamic)—build a list of commonly occurring patterns and encode these patterns by transmitting their index in the list. They are most useful with sources that generate a relatively small number of patterns quite frequently, such as text sources and computer commands. We discuss applications to text compression, modem communications, and image compression.

5.2 Introduction

In many applications, the output of the source consists of recurring patterns. A classic example is a text source in which certain patterns or words recur constantly. Also, there are certain patterns that simply do not occur, or if they do, occur with great rarity. For example, we can be reasonably sure that the word *Limpopo*[1] occurs in a very small fraction of the text sources in existence.

A very reasonable approach to encoding such sources is to keep a list, or *dictionary*, of frequently occurring patterns. When these patterns appear in the source output, they are encoded with a reference to the dictionary. If the pattern does not appear in the dictionary, then it can be encoded using some other, less efficient, method. In effect we are splitting

[1] "How the Elephant Got Its Trunk" in *Just So Stories* by Rudyard Kipling.

the input into two classes, frequently occurring patterns and infrequently occurring patterns. For this technique to be effective, the class of frequently occurring patterns, and hence the size of the dictionary, must be much smaller than the number of all possible patterns.

Suppose we have a particular text that consists of four-character words, three characters from the 26 lowercase letters of the English alphabet followed by a punctuation mark. Suppose our source alphabet consists of the 26 lowercase letters of the English alphabet and the punctuation marks comma, period, exclamation mark, question mark, semicolon, and colon. In other words, the size of the input alphabet is 32. If we were to encode the text source one character at a time, treating each character as an equally likely event, we would need 5 bits per character. Treating all 32^4 ($= 2^{20} = 1{,}048{,}576$) four-character patterns as equally likely, we have a code that assigns 20 bits to each four-character pattern. Let us now put the 256 most likely four-character patterns into a dictionary. The transmission scheme works as follows: Whenever we want to send a pattern that exists in the dictionary, we will send a 1-bit flag, say, a 0, followed by an 8-bit index corresponding to the entry in the dictionary. If the pattern is not in the dictionary, we will send a 1 followed by the 20-bit encoding of the pattern. If the pattern we encounter is not in the dictionary, we will actually use more bits than in the original scheme, 21 instead of 20. But if it is in the dictionary, we will send only 9 bits. The utility of our scheme will depend on the percentage of the words we encounter that are in the dictionary. We can get an idea about the utility of our scheme by calculating the average number of bits per pattern. If the probability of encountering a pattern from the dictionary is p, then the average number of bits per pattern R is given by

$$R = 9p + 21(1-p) = 21 - 12p. \tag{5.1}$$

For our scheme to be useful, R should have a value less than 20. This happens when $p \geq 0.084$. This does not seem like a very large number. However, note that if all patterns were occurring in an equally likely manner, the probability of encountering a pattern from the dictionary would be less than 0.00025!

We do not simply want a coding scheme that performs slightly better than the simple-minded approach of coding each pattern as equally likely; we would like to improve the performance as much as possible. In order for this to happen, p should be as large as possible. This means that we should carefully select patterns that are most likely to occur as entries in the dictionary. To do this, we have to have a pretty good idea about the structure of the source output. If we do not have information of this sort available to us prior to the encoding of a particular source output, we need to acquire this information somehow when we are encoding. If we feel we have sufficient prior knowledge, we can use a *static* approach; if not, we can take an *adaptive* approach. We will look at both these approaches in this chapter.

5.3 Static Dictionary

Choosing a static dictionary technique is most appropriate when considerable prior knowledge about the source is available. This technique is especially suitable for use in specific applications. For example, if the task were to compress the student records at a university, a static dictionary approach may be the best. This is because we know ahead of time that certain words such as "Name" and "Student ID" are going to appear in almost all of the records.

Other words such as "Sophomore," "credits," and so on will occur quite often. Depending on the location of the university, certain digits in social security numbers are more likely to occur. For example, in Nebraska most student ID numbers begin with the digits 505. In fact, most entries will be of a recurring nature. In this situation, it is highly efficient to design a compression scheme based on a static dictionary containing the recurring patterns. Similarly, there could be a number of other situations in which an application-specific or data-specific static-dictionary-based coding scheme would be the most efficient. It should be noted that these schemes would work well only for the applications and data they were designed for. If these schemes were to be used with different applications, they may cause an expansion of the data instead of compression.

A static dictionary technique that is less specific to a single application is *digram coding*. We describe this in the next section.

5.3.1 Digram Coding

One of the more common forms of static dictionary coding is digram coding. In this form of coding, the dictionary consists of all letters of the source alphabet followed by as many pairs of letters, called *digrams*, as can be accommodated by the dictionary. For example, suppose we were to construct a dictionary of size 256 for digram coding of all printable ASCII characters. The first 95 entries of the dictionary would be the 95 printable ASCII characters. The remaining 161 entries would be the most frequently used pairs of characters.

The digram encoder reads a two-character input and searches the dictionary to see if this input exists in the dictionary. If it does, the corresponding index is encoded and transmitted. If it does not, the first character of the pair is encoded. The second character in the pair then becomes the first character of the next digram. The encoder reads another character to complete the digram, and the search procedure is repeated.

Example 5.3.1:

Suppose we have a source with a five-letter alphabet $\mathcal{A} = \{a, b, c, d, r\}$. Based on knowledge about the source, we build the dictionary shown in Table 5.1.

TABLE 5.1 A sample dictionary.

Code	Entry	Code	Entry
000	a	100	r
001	b	101	ab
010	c	110	ac
011	d	111	ad

Suppose we wish to encode the sequence

$$abracadabra$$

The encoder reads the first two characters ab and checks to see if this pair of letters exists in the dictionary. It does and is encoded using the codeword 101. The encoder then reads

the next two characters *ra* and checks to see if this pair occurs in the dictionary. It does not, so the encoder sends out the code for *r*, which is 100, then reads in one more character, *c*, to make the two-character pattern *ac*. This does exist in the dictionary and is encoded as 110. Continuing in this fashion, the remainder of the sequence is coded. The output string for the given input sequence is 10110011011110110000 0. ♦

TABLE 5.2 Thirty most frequently occurring pairs of characters in a 41,364-character-long LaTeX document.

Pair	Count	Pair	Count
e⊔	1128	*ar*	314
⊔t	838	*at*	313
⊔⊔	823	*⊔w*	309
th	817	*te*	296
he	712	*⊔s*	295
in	512	*d⊔*	272
s⊔	494	*⊔o*	266
er	433	*io*	257
⊔a	425	*co*	256
t⊔	401	*re*	247
en	392	*⊔$*	246
on	385	*r⊔*	239
n⊔	353	*di*	230
ti	322	*ic*	229
⊔i	317	*ct*	226

TABLE 5.3 Thirty most frequently occurring pairs of characters in a collection of C programs containing 64,983 characters.

Pair	Count	Pair	Count
⊔⊔	5728	*st*	442
nl⊔	1471	*le*	440
;*nl*	1133	*ut*	440
in	985	*f*(	416
nt	739	*ar*	381
=*⊔*	687	*or*	374
⊔i	662	*r⊔*	373
t⊔	615	*en*	371
⊔=	612	*er*	358
);	558	*ri*	357
,*⊔*	554	*at*	352
nlnl	506	*pr*	351
⊔f	505	*te*	349
e⊔	500	*an*	348
*⊔**	444	*lo*	347

A list of the 30 most frequently occurring pairs of characters in an earlier version of this chapter is shown in Table 5.2. For comparison, the 30 most frequently occurring pairs of characters in a set of C programs is shown in Table 5.3.

In these tables, *b̸* corresponds to a space and *nl* corresponds to a new line. Notice how different the two tables are. It is easy to see that a dictionary designed for compressing LaTeX documents would not work very well when compressing C programs. However, generally we want techniques that will be able to compress a variety of source outputs. If we wanted to compress computer files, we do not want to change techniques based on the content of the file. Rather, we would like the technique to *adapt* to the characteristics of the source output. We discuss adaptive-dictionary-based techniques in the next section.

5.4 Adaptive Dictionary

Most adaptive-dictionary-based techniques have their roots in two landmark papers by Jacob Ziv and Abraham Lempel in 1977 [54] and 1978 [55]. These papers provide two different approaches to adaptively building dictionaries, and each approach has given rise to a number of variations. The approaches based on the 1977 paper are said to belong to the LZ77 family (also known as LZ1), while the approaches based on the 1978 paper are said to belong to the LZ78, or LZ2, family. The transposition of the initials is a historical accident and is a convention we will observe in this book. In the following sections, we first describe an implementation of each approach followed by some of the more well-known variations.

5.4.1 The LZ77 Approach

In the LZ77 approach, the dictionary is simply a portion of the previously encoded sequence. The encoder examines the input sequence through a sliding window as shown in Figure 5.1. The window consists of two parts, a *search buffer* that contains a portion of the recently encoded sequence, and a *look-ahead buffer* that contains the next portion of the sequence to be encoded. In Figure 5.1, the search buffer contains eight symbols, while the look-ahead buffer contains seven symbols. In practice, the sizes of the buffers are significantly larger; however, for the purpose of explanation, we will keep the buffer sizes small.

To encode the sequence in the look-ahead buffer, the encoder moves a search pointer back through the search buffer until it encounters a match to the first symbol in the look-ahead

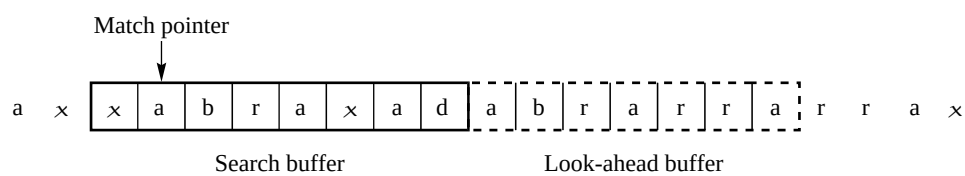

FIGURE 5.1 Encoding using the LZ77 approach.

buffer. The distance of the pointer from the look-ahead buffer is called the *offset*. The encoder then examines the symbols following the symbol at the pointer location to see if they match consecutive symbols in the look-ahead buffer. The number of consecutive symbols in the search buffer that match consecutive symbols in the look-ahead buffer, starting with the first symbol, is called the length of the match. The encoder searches the search buffer for the longest match. Once the longest match has been found, the encoder encodes it with a triple $\langle o, l, c \rangle$, where o is the offset, l is the length of the match, and c is the codeword corresponding to the symbol in the look-ahead buffer that follows the match. For example, in Figure 5.1 the pointer is pointing to the beginning of the longest match. The offset o in this case is 7, the length of the match l is 4, and the symbol in the look-ahead buffer following the match is ρ.

The reason for sending the third element in the triple is to take care of the situation where no match for the symbol in the look-ahead buffer can be found in the search buffer. In this case, the offset and match-length values are set to 0, and the third element of the triple is the code for the symbol itself.

If the size of the search buffer is S, the size of the window (search and look-ahead buffers) is W, and the size of the source alphabet is A, then the number of bits needed to code the triple using fixed-length codes is $\lceil \log_2 S \rceil + \lceil \log_2 W \rceil + \lceil \log_2 A \rceil$. Notice that the second term is $\lceil \log_2 W \rceil$, not $\lceil \log_2 S \rceil$. The reason for this is that the length of the match can actually exceed the length of the search buffer. We will see how this happens in Example 5.4.1.

In the following example, we will look at three different possibilities that may be encountered during the coding process:

1. There is no match for the next character to be encoded in the window.
2. There is a match.
3. The matched string extends inside the look-ahead buffer.

Example 5.4.1: The LZ77 approach

Suppose the sequence to be encoded is

$$\ldots cabracadabrarrarrad \ldots$$

Suppose the length of the window is 13, the size of the look-ahead buffer is six, and the current condition is as follows:

cabraca	*dabrar*

with *dabrar* in the look-ahead buffer. We look back in the already encoded portion of the window to find a match for d. As we can see, there is no match, so we transmit the triple $\langle 0, 0, C(d) \rangle$. The first two elements of the triple show that there is no match to d in the search buffer, while $C(d)$ is the code for the character d. This seems like a wasteful way to encode a single character, and we will have more to say about this later.

For now, let's continue with the encoding process. As we have encoded a single character, we move the window by one character. Now the contents of the buffer are

abracad	*abrarr*

with *abrarr* in the look-ahead buffer. Looking back from the current location, we find a match to *a* at an offset of two. The length of this match is one. Looking further back, we have another match for *a* at an offset of four; again the length of the match is one. Looking back even further in the window, we have a third match for *a* at an offset of seven. However, this time the length of the match is four (see Figure 5.2). So we encode the string *abra* with the triple $\langle 7, 4, C(r)\rangle$, and move the window forward by five characters. The window now contains the following characters:

adabrar	*rarrad*

Now the look-ahead buffer contains the string *rarrad*. Looking back in the window, we find a match for *r* at an offset of one and a match length of one, and a second match at an offset of three with a match length of what at first appears to be three. It turns out we can use a match length of five instead of three.

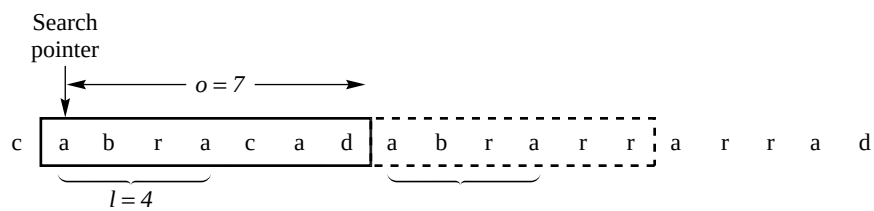

FIGURE 5. 2 The encoding process.

Why this is so will become clearer when we decode the sequence. To see how the decoding works, let us assume that we have decoded the sequence *cabraca* and we receive the triples $\langle 0, 0, C(d)\rangle$, $\langle 7, 4, C(r)\rangle$, and $\langle 3, 5, C(d)\rangle$. The first triple is easy to decode; there was no match within the previously decoded string, and the next symbol is *d*. The decoded string is now *cabracad*. The first element of the next triple tells the decoder to move the copy pointer back seven characters, and copy four characters from that point. The decoding process works as shown in Figure 5.3.

Finally, let's see how the triple $\langle 3, 5, C(d)\rangle$ gets decoded. We move back three characters and start copying. The first three characters we copy are *rar*. The copy pointer moves once again, as shown in Figure 5.4, to copy the recently copied character *r*. Similarly, we copy the next character *a*. Even though we started copying only three characters back, we end up decoding five characters. Notice that the match only has to *start* in the search buffer; it can extend into the look-ahead buffer. In fact, if the last character in the look-ahead buffer

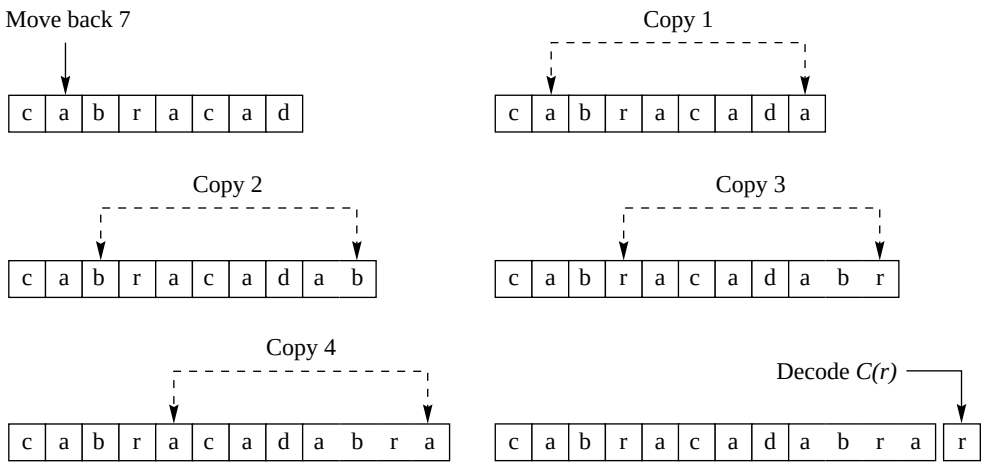

FIGURE 5. 3 Decoding of the triple ⟨7, 4, *C*(*r*)⟩.

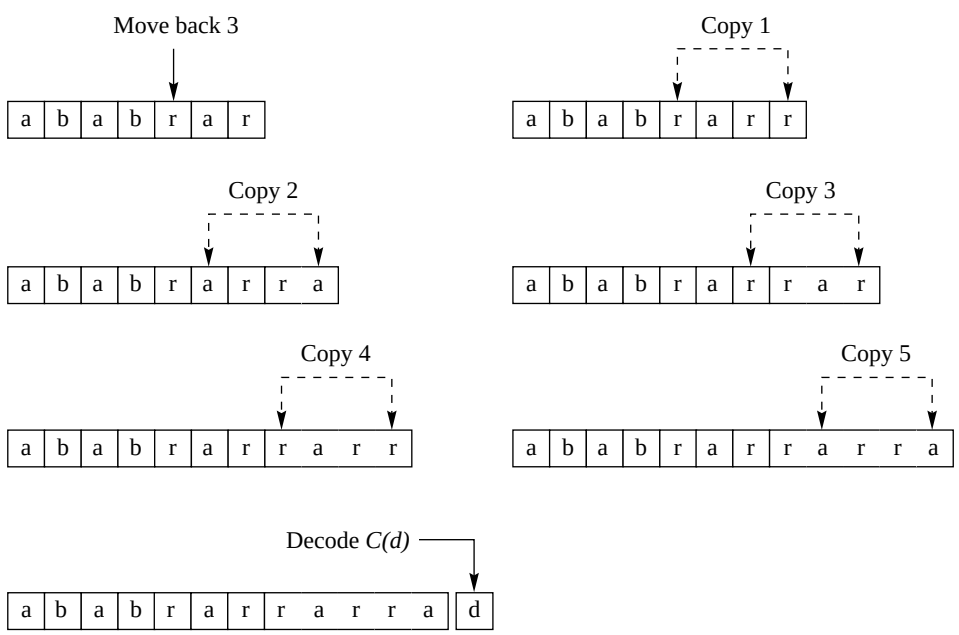

FIGURE 5. 4 Decoding the triple ⟨3, 5, *C*(*d*)⟩.

had been *r* instead of *d*, followed by several more repetitions of *rar*, the entire sequence of repeated *rar*s could have been encoded with a single triple. ♦

As we can see, the LZ77 scheme is a very simple adaptive scheme that requires no prior knowledge of the source and seems to require no assumptions about the characteristics of the source. The authors of this algorithm showed that asymptotically the performance of this algorithm approached the best that could be obtained by using a scheme that had full knowledge about the statistics of the source. While this may be true asymptotically, in practice there are a number of ways of improving the performance of the LZ77 algorithm as described here. Furthermore, by using the recent portions of the sequence, there is an assumption of sorts being used here—that is, that patterns recur "close" together. As we shall see, in LZ78 the authors removed this "assumption" and came up with an entirely different adaptive-dictionary-based scheme. Before we get to that, let us look at the different variations of the LZ77 algorithm.

Variations on the LZ77 Theme

There are a number of ways that the LZ77 scheme can be made more efficient, and most of these have appeared in the literature. Many of the improvements deal with the efficient encoding of the triples. In the description of the LZ77 algorithm, we assumed that the triples were encoded using a fixed-length code. However, if we were willing to accept more complexity, we could encode the triples using variable-length codes. As we saw in earlier chapters, these codes can be adaptive or, if we were willing to use a two-pass algorithm, they can be semiadaptive. Popular compression packages, such as PKZip, Zip, LHarc, PNG, gzip, and ARJ, all use an LZ77-based algorithm followed by a variable-length coder.

Other variations on the LZ77 algorithm include varying the size of the search and look-ahead buffers. To make the search buffer large requires the development of more effective search strategies. Such strategies can be implemented more effectively if the contents of the search buffer are stored in a manner conducive to fast searches.

The simplest modification to the LZ77 algorithm, and one that is used by most variations of the LZ77 algorithm, is to eliminate the situation where we use a triple to encode a single character. Use of a triple is highly inefficient, especially if a large number of characters occur infrequently. The modification to get rid of this inefficiency is simply the addition of a flag bit, to indicate whether what follows is the codeword for a single symbol. By using this flag bit we also get rid of the necessity for the third element of the triple. Now all we need to do is to send a pair of values corresponding to the offset and length of match. This modification to the LZ77 algorithm is referred to as LZSS [56, 57].

5.4.2 The LZ78 Approach

The LZ77 approach implicitly assumes that like patterns will occur close together. It makes use of this structure by using the recent past of the sequence as the dictionary for encoding.

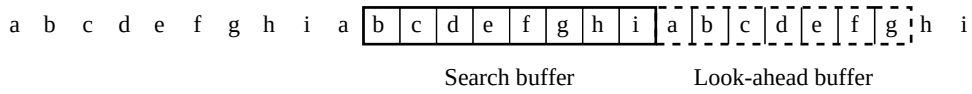

FIGURE 5.5 The Achilles' heel of LZ77.

However, this means that any pattern that recurs over a period longer than that covered by the coder window will not be captured. The worst-case situation would be where the sequence to be encoded was periodic with a period longer than the search buffer. Consider Figure 5.5.

This is a periodic sequence with a period of nine. If the search buffer had been just one symbol longer, this sequence could have been significantly compressed. As it stands, none of the new symbols will have a match in the search buffer and will have to be represented by separate codewords. As this involves sending along overhead (a 1-bit flag for LZSS and a triple for the original LZ77 algorithm), the net result will be an expansion rather than a compression.

Although this is an extreme situation, there are less drastic circumstances in which the finite view of the past would be a drawback. The LZ78 algorithm solves this problem by dropping the reliance on the search buffer and keeping an explicit dictionary. This dictionary has to be built at both the encoder and decoder, and care must be taken that the dictionaries are built in an identical manner. The inputs are coded as a double $\langle i, c\rangle$, with i being an index corresponding to the dictionary entry that was the longest match to the input, and c being the code for the character in the input following the matched portion of the input. As in the case of LZ77, the index value of 0 is used in the case of no match. This double then becomes the newest entry in the dictionary. Thus, each new entry into the dictionary is one new symbol concatenated with an existing dictionary entry. To see how the LZ78 algorithm works, consider the following example.

Example 5.4.2: The LZ78 approach

Let us encode the following sequence using the LZ78 approach:

$$wabba\not{b}wabba\not{b}wabba\not{b}wabba\not{b}wooo\not{b}wooo\not{b}woo$$[2]

where $\not{b}$ stands for space. Initially, the dictionary is empty, so the first few symbols encountered are encoded with the index value set to 0. The first three encoder outputs are $\langle 0, C(w)\rangle$, $\langle 0, C(a)\rangle$, $\langle 0, C(b)\rangle$, and the dictionary looks like Table 5.4.

The fourth symbol is a b, which is the third entry in the dictionary. If we append the next symbol, we would get the pattern ba, which is not in the dictionary, so we encode these two symbols as $\langle 3, C(a)\rangle$, and add the pattern ba as the fourth entry in the dictionary. Continuing in this fashion, the encoder output and the dictionary develop as in Table 5.5. Notice that the entries in the dictionary generally keep getting longer, and if this particular

[2] "The Monster Song" from *Sesame Street*.

TABLE 5.4 The initial dictionary.

Index	Entry
1	*w*
2	*a*
3	*b*

TABLE 5.5 Development of dictionary.

Encoder Output	Dictionary Index	Dictionary Entry
$\langle 0, C(w)\rangle$	1	*w*
$\langle 0, C(a)\rangle$	2	*a*
$\langle 0, C(b)\rangle$	3	*b*
$\langle 3, C(a)\rangle$	4	*ba*
$\langle 0, C(\not{b})\rangle$	5	$\not{b}$
$\langle 1, C(a)\rangle$	6	*wa*
$\langle 3, C(b)\rangle$	7	*bb*
$\langle 2, C(\not{b})\rangle$	8	$a\not{b}$
$\langle 6, C(b)\rangle$	9	*wab*
$\langle 4, C(\not{b})\rangle$	10	$ba\not{b}$
$\langle 9, C(b)\rangle$	11	*wabb*
$\langle 8, C(w)\rangle$	12	$a\not{b}w$
$\langle 0, C(o)\rangle$	13	*o*
$\langle 13, C(\not{b})\rangle$	14	$o\not{b}$
$\langle 1, C(o)\rangle$	15	*wo*
$\langle 14, C(w)\rangle$	16	$o\not{b}w$
$\langle 13, C(o)\rangle$	17	*oo*

sentence was repeated often, as it is in the song, after a while the entire sentence would be an entry in the dictionary. ♦

While the LZ78 algorithm has the ability to capture patterns and hold them indefinitely, it also has a rather serious drawback. As seen from the example, the dictionary keeps growing without bound. In a practical situation, we would have to stop the growth of the dictionary at some stage, and then either prune it back or treat the encoding as a fixed dictionary scheme. We will discuss some possible approaches when we study applications of dictionary coding.

Variations on the LZ78 Theme—The LZW Algorithm

There are a number of ways the LZ78 algorithm can be modified, and as is the case with the LZ77 algorithm, anything that can be modified probably has been. The most well-known modification, one that initially sparked much of the interest in the LZ algorithms, is a modification by Terry Welch known as LZW [58]. Welch proposed a technique for removing

the necessity of encoding the second element of the pair $\langle i, c \rangle$. That is, the encoder would only send the index to the dictionary. In order to do this, the dictionary has to be primed with all the letters of the source alphabet. The input to the encoder is accumulated in a pattern p as long as p is contained in the dictionary. If the addition of another letter a results in a pattern $p * a$ ($*$ denotes concatenation) that is not in the dictionary, then the index of p is transmitted to the receiver, the pattern $p * a$ is added to the dictionary, and we start another pattern with the letter a. The LZW algorithm is best understood with an example. In the following two examples, we will look at the encoder and decoder operations for the same sequence used to explain the LZ78 algorithm.

Example 5.4.3: The LZW algorithm—encoding

We will use the sequence previously used to demonstrate the LZ78 algorithm as our input sequence:

$$wabbab\not{b}wabbab\not{b}wabbab\not{b}wabbab\not{b}woo\not{b}woo\not{b}woo$$

Assuming that the alphabet for the source is $\{\not{b}, a, b, o, w\}$, the LZW dictionary initially looks like Table 5.6.

TABLE 5.6 Initial LZW dictionary.

Index	Entry
1	$\not{b}$
2	a
3	b
4	o
5	w

The encoder first encounters the letter w. This "pattern" is in the dictionary so we concatenate the next letter to it, forming the pattern wa. This pattern is not in the dictionary, so we encode w with its dictionary index 5, add the pattern wa to the dictionary as the sixth element of the dictionary, and begin a new pattern starting with the letter a. As a is in the dictionary, we concatenate the next element b to form the pattern ab. This pattern is not in the dictionary, so we encode a with its dictionary index value 2, add the pattern ab to the dictionary as the seventh element of the dictionary, and start constructing a new pattern with the letter b. We continue in this manner, constructing two-letter patterns, until we reach the letter w in the second $wabba$. At this point the output of the encoder consists entirely of indices from the initial dictionary: 5 2 3 3 2 1. The dictionary at this point looks like Table 5.7. (The 12th entry in the dictionary is still under construction.) The next symbol in the sequence is a. Concatenating this to w, we get the pattern wa. This pattern already exists in the dictionary (item 6), so we read the next symbol, which is b. Concatenating this to wa, we get the pattern wab. This pattern does not exist in the dictionary, so we include it as the 12th entry in the dictionary and start a new pattern with the symbol b. We also encode

wa with its index value of 6. Notice that after a series of two-letter entries, we now have a three-letter entry. As the encoding progresses, the length of the entries keeps increasing. The longer entries in the dictionary indicate that the dictionary is capturing more of the structure in the sequence. The dictionary at the end of the encoding process is shown in Table 5.8. Notice that the 12th through the 19th entries are all either three or four letters in length. Then we encounter the pattern *woo* for the first time and we drop back to two-letter patterns for three more entries, after which we go back to entries of increasing length.

TABLE 5.7 Constructing the 12th entry of the LZW dictionary.

Index	Entry
1	*b̸*
2	*a*
3	*b*
4	*o*
5	*w*
6	*wa*
7	*ab*
8	*bb*
9	*ba*
10	*ab̸*
11	*b̸*w
12	*w*…

TABLE 5.8 The LZW dictionary for encoding wabba*b̸*wabba*b̸*wabba*b̸*wabba*b̸*woo*b̸*woo*b̸*woo.

Index	Entry	Index	Entry
1	*b̸*	14	*ab̸w*
2	*a*	15	*wabb*
3	*b*	16	*bab̸*
4	*o*	17	*b̸wa*
5	*w*	18	*abb*
6	*wa*	19	*bab̸w*
7	*ab*	20	*wo*
8	*bb*	21	*oo*
9	*ba*	22	*ob̸*
10	*ab̸*	23	*b̸wo*
11	*b̸w*	24	*oob̸*
12	*wab*	25	*b̸woo*
13	*bba*		

The encoder output sequence is 5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4 4 11 21 23 4. ♦

Example 5.4.4: The LZW algorithm—decoding

In this example we will take the encoder output from the previous example and decode it using the LZW algorithm. The encoder output sequence in the previous example was

5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4 4 11 21 23 4

This becomes the decoder input sequence. The decoder starts with the same initial dictionary as the encoder (Table 5.6).

The index value 5 corresponds to the letter *w*, so we decode *w* as the first element of our sequence. At the same time, in order to mimic the dictionary construction procedure of the encoder, we begin construction of the next element of the dictionary. We start with the letter *w*. This pattern exists in the dictionary, so we do not add it to the dictionary and continue with the decoding process. The next decoder input is 2, which is the index corresponding to the letter *a*. We decode an *a* and concatenate it with our current pattern to form the pattern *wa*. As this does not exist in the dictionary, we add it as the sixth element of the dictionary and start a new pattern beginning with the letter *a*. The next four inputs 3 3 2 1 correspond to the letters *bbaƀ* and generate the dictionary entries *ab*, *bb*, *ba*, and *aƀ*. The dictionary now looks like Table 5.9, where the 11th entry is under construction.

TABLE 5.9 Constructing the 11th entry of the LZW dictionary while decoding.

Index	Entry
1	*ƀ*
2	*a*
3	*b*
4	*o*
5	*w*
6	*wa*
7	*ab*
8	*bb*
9	*ba*
10	*aƀ*
11	*ƀ*. . .

The next input is 6, which is the index of the pattern *wa*. Therefore, we decode a *w* and an *a*. We first concatenate *w* to the existing pattern, which is *ƀ*, and form the pattern *ƀw*. As *ƀw* does not exist in the dictionary, it becomes the 11th entry. The new pattern now starts with the letter *w*. We had previously decoded the letter *a*, which we now concatenate to *w* to obtain the pattern *wa*. This pattern is contained in the dictionary, so we decode the next input, which is 8. This corresponds to the entry *bb* in the dictionary. We decode the first *b* and concatenate it to the pattern *wa* to get the pattern *wab*. This pattern does not exist in the dictionary, so we add it as the 12th entry in the dictionary and start a new pattern with the letter *b*. Decoding the second *b* and concatenating it to the new pattern, we get the pattern *bb*. This pattern exists in the dictionary, so we decode the next element in the

sequence of encoder outputs. Continuing in this fashion, we can decode the entire sequence. Notice that the dictionary being constructed by the decoder is identical to that constructed by the encoder. ♦

There is one particular situation in which the method of decoding the LZW algorithm described above breaks down. Suppose we had a source with an alphabet $\mathcal{A} = \{a, b\}$, and we were to encode the sequence beginning with *abababab*. . . . The encoding process is still the same. We begin with the initial dictionary shown in Table 5.10 and end up with the final dictionary shown in Table 5.11.

The transmitted sequence is 1 2 3 5. . . . This looks like a relatively straightforward sequence to decode. However, when we try to do so, we run into a snag. Let us go through the decoding process and see what happens.

We begin with the same initial dictionary as the encoder (Table 5.10). The first two elements in the received sequence 1 2 3 5. . . are decoded as a and b, giving rise to the third dictionary entry ab, and the beginning of the next pattern to be entered in the dictionary, b. The dictionary at this point is shown in Table 5.12.

TABLE 5.10 Initial dictionary for *abababab*.

Index	Entry
1	a
2	b

TABLE 5.11 Final dictionary for *abababab*.

Index	Entry
1	a
2	b
3	ab
4	ba
5	aba
6	$abab$
7	b. . .

TABLE 5.12 Constructing the fourth entry of the dictionary while decoding.

Index	Entry
1	a
2	b
3	ab
4	b. . .

TABLE 5.13 Constructing the fifth entry (stage one).

Index	Entry
1	*a*
2	*b*
3	*ab*
4	*ba*
5	*a*. . .

TABLE 5.14 Constructing the fifth entry (stage two).

Index	Entry
1	*a*
2	*b*
3	*ab*
4	*ba*
5	*ab*. . .

The next input to the decoder is 3. This corresponds to the dictionary entry *ab*. Decoding each in turn, we first concatenate *a* to the pattern under construction to get *ba*. This pattern is not contained in the dictionary, so we add this to the dictionary (keep in mind, we have not used the *b* from *ab* yet), which now looks like Table 5.13.

The new entry starts with the letter *a*. We have only used the first letter from the pair *ab*. Therefore, we now concatenate *b* to *a* to obtain the pattern *ab*. This pattern is contained in the dictionary, so we continue with the decoding process. The dictionary at this stage looks like Table 5.14.

The first four entries in the dictionary are complete, while the fifth entry is still under construction. However, the very next input to the decoder is 5, which corresponds to the incomplete entry! How do we decode an index for which we do not as yet have a complete dictionary entry?

The situation is actually not as bad as it looks. (Of course, if it were, we would not now be studying LZW.) While we may not have a fifth entry for the dictionary, we do have the beginnings of the fifth entry, which is *ab*. . . . Let us, for the moment, pretend that we do indeed have the fifth entry and continue with the decoding process. If we had a fifth entry, the first two letters of the entry would be *a* and *b*. Concatenating *a* to the partial new entry we get the pattern *aba*. This pattern is not contained in the dictionary, so we add this to our dictionary, which now looks like Table 5.15. Notice that we now have the fifth entry in the dictionary, which is *aba*. We have already decoded the *ab* portion of *aba*. We can now decode the last letter *a* and continue on our merry way.

This means that the LZW decoder has to contain an exception handler to handle the special case of decoding an index that does not have a corresponding complete entry in the decoder dictionary.

TABLE 5.15 Completion of the fifth entry.

Index	Entry
1	*a*
2	*b*
3	*ab*
4	*ba*
5	*aba*
6	*a*. . .

5.5 Applications

Since the publication of Terry Welch's article [58], there has been a steadily increasing number of applications that use some variant of the LZ78 algorithm. Among the LZ78 variants, by far the most popular is the LZW algorithm. In this section we describe two of the best-known applications of LZW: GIF, and V.42 bis. While the LZW algorithm was initially the algorithm of choice patent concerns has lead to increasing use of the LZ77 algorithm. The most popular implementation of the LZ77 algorithm is the *deflate* algorithm initially designed by Phil Katz. It is part of the popular *zlib* library developed by Jean-loup Gailly and Mark Adler. Jean-loup Gailly also used deflate in the widely used *gzip* algorithm. The *deflate* algorithm is also used in PNG which we describe below.

5.5.1 File Compression—UNIX `compress`

The UNIX `compress` command is one of the earlier applications of LZW. The size of the dictionary is adaptive. We start with a dictionary of size 512. This means that the transmitted codewords are 9 bits long. Once the dictionary has filled up, the size of the dictionary is doubled to 1024 entries. The codewords transmitted at this point have 10 bits. The size of the dictionary is progressively doubled as it fills up. In this way, during the earlier part of the coding process when the strings in the dictionary are not very long, the codewords used to encode them also have fewer bits. The maximum size of the codeword, $b_{\max}$, can be set by the user to between 9 and 16, with 16 bits being the default. Once the dictionary contains $2^{b_{\max}}$ entries, `compress` becomes a static dictionary coding technique. At this point the algorithm monitors the compression ratio. If the compression ratio falls below a threshold, the dictionary is flushed, and the dictionary building process is restarted. This way, the dictionary always reflects the local characteristics of the source.

5.5.2 Image Compression—The Graphics Interchange Format (GIF)

The Graphics Interchange Format (GIF) was developed by Compuserve Information Service to encode graphical images. It is another implementation of the LZW algorithm and is very similar to the `compress` command. The compressed image is stored with the first byte

TABLE 5.16 Comparison of GIF with arithmetic coding.

Image	GIF	Arithmetic Coding of Pixel Values	Arithmetic Coding of Pixel Differences
Sena	51,085	53,431	31,847
Sensin	60,649	58,306	37,126
Earth	34,276	38,248	32,137
Omaha	61,580	56,061	51,393

being the minimum number of bits b per pixel in the original image. For the images we have been using as examples, this would be eight. The binary number 2^b is defined to be the *clear code*. This code is used to reset all compression and decompression parameters to a start-up state. The initial size of the dictionary is 2^{b+1}. When this fills up, the dictionary size is doubled, as was done in the `compress` algorithm, until the maximum dictionary size of 4096 is reached. At this point the compression algorithm behaves like a static dictionary algorithm. The codewords from the LZW algorithm are stored in blocks of characters. The characters are 8 bits long, and the maximum block size is 255. Each block is preceded by a header that contains the block size. The block is terminated by a block terminator consisting of eight 0s. The end of the compressed image is denoted by an end-of-information code with a value of $2^b + 1$. This codeword should appear before the block terminator.

GIF has become quite popular for encoding all kinds of images, both computer-generated and "natural" images. While GIF works well with computer-generated graphical images, and pseudocolor or color-mapped images, it is generally not the most efficient way to losslessly compress images of natural scenes, photographs, satellite images, and so on. In Table 5.16 we give the file sizes for the GIF-encoded test images. For comparison, we also include the file sizes for arithmetic coding the original images and arithmetic coding the differences.

Notice that even if we account for the extra overhead in the GIF files, for these images GIF barely holds its own even with simple arithmetic coding of the original pixels. While this might seem odd at first, if we examine the image on a pixel level, we see that there are very few repetitive patterns compared to a text source. Some images, like the Earth image, contain large regions of constant values. In the dictionary coding approach, these regions become single entries in the dictionary. Therefore, for images like these, the straight forward dictionary coding approach does hold its own. However, for most other images, it would probably be preferable to perform some preprocessing to obtain a sequence more amenable to dictionary coding. The PNG standard described next takes advantage of the fact that in natural images the pixel-to-pixel variation is generally small to develop an appropriate preprocessor. We will also revisit this subject in Chapter 7.

5.5.3 Image Compression—Portable Network Graphics (PNG)

The PNG standard is one of the first standards to be collaboratively developed over the Internet. The impetus for it was an announcement in December 1994 by Unisys (which had acquired the patent for LZW from Sperry) and CompuServe that they would start charging

royalties to authors of software that included support for GIF. The announcement resulted in an uproar in the segment of the compression community that formed the core of the Usenet group comp.compression. The community decided that a patent-free replacement for GIF should be developed, and within three months PNG was born. (For a more detailed history of PNG as well as software and much more, go to the PNG website maintained by Greg Roelof, http://www.libpng.org/pub/png/.)

Unlike GIF, the compression algorithm used in PNG is based on LZ77. In particular, it is based on the *deflate* [59] implementation of LZ77. This implementation allows for match lengths of between 3 and 258. At each step the encoder examines three bytes. If it cannot find a match of at least three bytes it puts out the first byte and examines the next three bytes. So, at each step it either puts out the value of a single byte, or literal, or the pair $< match\ length,\ offset >$. The alphabets of the *literal* and *match length* are combined to form an alphabet of size 286 (indexed by $0--285$). The indices $0--255$ represent literal bytes and the index 256 is an end-of-block symbol. The remaining 29 indices represent codes for ranges of lengths between 3 and 258, as shown in Table 5.17. The table shows the index, the number of selector bits to follow the index, and the lengths represented by the index and selector bits. For example, the index 277 represents the range of lengths from 67 to 82. To specify which of the sixteen values has actually occurred, the code is followed by four selector bits.

The index values are represented using a Huffman code. The Huffman code is specified in Table 5.18.

The *offset* can take on values between 1 and 32,768. These values are divided into 30 ranges. The thirty range values are encoded using a Huffman code (different from the Huffman code for the *literal* and *length* values) and the code is followed by a number of selector bits to specify the particular distance within the range.

We have mentioned earlier that in natural images there is not great deal of repetition of sequences of pixel values. However, pixel values that are spatially close also tend to have values that are similar. The PNG standard makes use of this structure by estimating the value of a pixel based on its causal neighbors and subtracting this estimate from the pixel. The difference modulo 256 is then encoded in place of the original pixel. There are four different ways of getting the estimate (five if you include no estimation), and PNG allows

TABLE 5.17 Codes for representations of *match length* [59].

Index	# of selector bits	Length	Index	# of selector bits	Length	Index	# of selector bits	Length
257	0	3	267	1	15,16	277	4	67–82
258	0	4	268	1	17,18	278	4	83–98
259	0	5	269	2	19–22	279	4	99–114
260	0	6	270	2	23–26	280	4	115–130
261	0	7	271	2	27–30	281	5	131–162
262	0	8	272	2	31–34	282	5	163–194
263	0	9	273	3	35–42	283	5	195–226
264	0	10	274	3	43–50	284	5	227–257
265	1	11, 12	275	3	51–58	285	0	258
266	1	13, 14	276	3	59–66			

TABLE 5.18 Huffman codes for the *match length* alphabet [59].

Index Ranges	# of bits	Binary Codes
0–143	8	00110000 through 10111111
144–255	9	110010000 through 111111111
256–279	7	0000000 through 0010111
280–287	8	11000000 through 11000111

TABLE 5.19 Comparison of PNG with GIF and arithmetic coding.

Image	PNG	GIF	Arithmetic Coding of Pixel Values	Arithmetic Coding of Pixel Differences
Sena	31,577	51,085	53,431	31,847
Sensin	34,488	60,649	58,306	37,126
Earth	26,995	34,276	38,248	32,137
Omaha	50,185	61,580	56,061	51,393

the use of a different method of estimation for each row. The first way is to use the pixel from the row above as the estimate. The second method is to use the pixel to the left as the estimate. The third method uses the average of the pixel above and the pixel to the left. The final method is a bit more complex. An initial estimate of the pixel is first made by adding the pixel to the left and the pixel above and subtracting the pixel to the upper left. Then the pixel that is closest to the initial esitmate (upper, left, or upper left) is taken as the estimate. A comparison of the performance of PNG and GIF on our standard image set is shown in Table 5.19. The PNG method clearly outperforms GIF.

5.5.4 Compression over Modems—V.42 bis

The ITU-T Recommendation V.42 bis is a compression standard devised for use over a telephone network along with error-correcting procedures described in CCITT Recommendation V.42. This algorithm is used in modems connecting computers to remote users. The algorithm described in this recommendation operates in two modes, a transparent mode and a compressed mode. In the transparent mode, the data are transmitted in uncompressed form, while in the compressed mode an LZW algorithm is used to provide compression.

The reason for the existence of two modes is that at times the data being transmitted do not have repetitive structure and therefore cannot be compressed using the LZW algorithm. In this case, the use of a compression algorithm may even result in expansion. In these situations, it is better to send the data in an uncompressed form. A random data stream would cause the dictionary to grow without any long patterns as elements of the dictionary. This means that most of the time the transmitted codeword would represent a single letter

from the source alphabet. As the dictionary size is much larger than the source alphabet size, the number of bits required to represent an element in the dictionary is much more than the number of bits required to represent a source letter. Therefore, if we tried to compress a sequence that does not contain repeating patterns, we would end up with more bits to transmit than if we had not performed any compression. Data without repetitive structure are often encountered when a previously compressed file is transferred over the telephone lines.

The V.42 bis recommendation suggests periodic testing of the output of the compression algorithm to see if data expansion is taking place. The exact nature of the test is not specified in the recommendation.

In the compressed mode, the system uses LZW compression with a variable-size dictionary. The initial dictionary size is negotiated at the time a link is established between the transmitter and receiver. The V.42 bis recommendation suggests a value of 2048 for the dictionary size. It specifies that the minimum size of the dictionary is to be 512. Suppose the initial negotiations result in a dictionary size of 512. This means that our codewords that are indices into the dictionary will be 9 bits long. Actually, the entire 512 indices do not correspond to input strings; three entries in the dictionary are reserved for control codewords. These codewords in the compressed mode are shown in Table 5.20.

When the numbers of entries in the dictionary exceed a prearranged threshold C_3, the encoder sends the STEPUP control code, and the codeword size is incremented by 1 bit. At the same time, the threshold C_3 is also doubled. When all available dictionary entries are filled, the algorithm initiates a reuse procedure. The location of the first string entry in the dictionary is maintained in a variable N_5. Starting from N_5, a counter C_1 is incremented until it finds a dictionary entry that is not a prefix to any other dictionary entry. The fact that this entry is not a prefix to another dictionary entry means that this pattern has not been encountered since it was created. Furthermore, because of the way it was located, among patterns of this kind this pattern has been around the longest. This reuse procedure enables the algorithm to prune the dictionary of strings that may have been encountered in the past but have not been encountered recently, on a continual basis. In this way the dictionary is always matched to the current source statistics.

To reduce the effect of errors, the CCITT recommends setting a maximum string length. This maximum length is negotiated at link setup. The CCITT recommends a range of 6–250, with a default value of 6.

The V.42 bis recommendation avoids the need for an exception handler for the case where the decoder receives a codeword corresponding to an incomplete entry by forbidding the use of the last entry in the dictionary. Instead of transmitting the codeword corresponding to the last entry, the recommendation requires the sending of the codewords corresponding

TABLE 5.20 Control codewords in compressed mode.

Codeword	Name	Description
0	ETM	Enter transparent mode
1	FLUSH	Flush data
2	STEPUP	Increment codeword size

to the constituents of the last entry. In the example used to demonstrate this quirk of the LZW algorithm, instead of transmitting the codeword 5, the V.42 bis recommendation would have forced us to send the codewords 3 and 1.

5.6 Summary

In this chapter we have introduced techniques that keep a dictionary of recurring patterns and transmit the index of those patterns instead of the patterns themselves in order to achieve compression. There are a number of ways the dictionary can be constructed.

- In applications where certain patterns consistently recur, we can build application-specific static dictionaries. Care should be taken not to use these dictionaries outside their area of intended application. Otherwise, we may end up with data expansion instead of data compression.
- The dictionary can be the source output itself. This is the approach used by the LZ77 algorithm. When using this algorithm, there is an implicit assumption that recurrence of a pattern is a local phenomenon.
- This assumption is removed in the LZ78 approach, which dynamically constructs a dictionary from patterns observed in the source output.

Dictionary-based algorithms are being used to compress all kinds of data; however, care should be taken with their use. This approach is most useful when structural constraints restrict the frequently occurring patterns to a small subset of all possible patterns. This is the case with text, as well as computer-to-computer communication.

Further Reading

1. *Text Compression*, by T.C. Bell, J.G. Cleary, and I.H. Witten [1], provides an excellent exposition of dictionary-based coding techniques.
2. *The Data Compression Book*, by M. Nelson and J.-L. Gailley [60], also does a good job of describing the Ziv-Lempel algorithms. There is also a very nice description of some of the software implementation aspects.
3. *Data Compression*, by G. Held and T.R. Marshall [61], contains a description of digram coding under the name "diatomic coding." The book also includes BASIC programs that help in the design of dictionaries.
4. The PNG algorithm is described in a very accessible manner in "PNG Lossless Compression," by G. Roelofs [62] in the *Lossless Compression Handbook*.
5. A more in-depth look at dictionary compression is provided in "Dictionary-Based Data Compression: An Algorithmic Perspective," by S.C. Şahinalp and N.M. Rajpoot [63] in the *Lossless Compression Handbook*.

5.7 Projects and Problems

1. To study the effect of dictionary size on the efficiency of a static dictionary technique, we can modify Equation (5.1) so that it gives the rate as a function of both p and the dictionary size M. Plot the rate as a function of p for different values of M, and discuss the trade-offs involved in selecting larger or smaller values of M.

2. Design and implement a digram coder for text files of interest to you.

(a) Study the effect of the dictionary size, and the size of the text file being encoded on the amount of compression.

(b) Use the digram coder on files that are not similar to the ones you used to design the digram coder. How much does this affect your compression?

3. Given an initial dictionary consisting of the letters *a b r y* ⊿, encode the following message using the LZW algorithm: *a*⊿*bar*⊿*barray*⊿*by*⊿*barrayar*⊿*bay*.

4. A sequence is encoded using the LZW algorithm and the initial dictionary shown in Table 5.21.

TABLE 5.21 Initial dictionary for Problem 4.

Index	Entry
1	*a*
2	⊿
3	*h*
4	*i*
5	*s*
6	*t*

(a) The output of the LZW encoder is the following sequence:

6	3	4	5	2	3	1	6	2	9	11	16	12	14	4	20	10	8	23	13

Decode this sequence.

(b) Encode the decoded sequence using the same initial dictionary. Does your answer match the sequence given above?

5. A sequence is encoded using the LZW algorithm and the initial dictionary shown in Table 5.22.

(a) The output of the LZW encoder is the following sequence:

3	1	4	6	8	4	2	1	2	5	10	6	11	13	6

Decode this sequence.

TABLE 5.22 Initial dictionary for Problem 5.

Index	Entry
1	a
2	$\not{b}$
3	r
4	t

(b) Encode the decoded sequence using the same initial dictionary. Does your answer match the sequence given above?

6. Encode the following sequence using the LZ77 algorithm:

$$barrayar\not{b}bar\not{b}by\not{b}barrayar\not{b}bay$$

Assume you have a window size of 30 with a look-ahead buffer of size 15. Furthermore, assume that $C(a) = 1$, $C(b) = 2$, $C(\not{b}) = 3$, $C(r) = 4$, and $C(y) = 5$.

7. A sequence is encoded using the LZ77 algorithm. Given that $C(a) = 1$, $C(\not{b}) = 2$, $C(r) = 3$, and $C(t) = 4$, decode the following sequence of triples:

$$\langle 0, 0, 3\rangle\ \langle 0, 0, 1\rangle\ \langle 0, 0, 4\rangle\ \langle 2, 8, 2\rangle\ \langle 3, 1, 2\rangle\ \langle 0, 0, 3\rangle\ \langle 6, 4, 4\rangle\ \langle 9, 5, 4\rangle$$

Assume that the size of the window is 20 and the size of the look-ahead buffer is 10. Encode the decoded sequence and make sure you get the same sequence of triples.

8. Given the following primed dictionary and the received sequence below, build an LZW dictionary *and* decode the transmitted sequence.

Received Sequence: 4, 5, 3, 1, 2, 8, 2, 7, 9, 7, 4

Decoded Sequence:_____

Initial dictionary:

(a) S

(b) $\not{b}$

(c) I

(d) T

(e) H

6

Context-Based Compression

6.1 Overview

In this chapter we present a number of techniques that use minimal prior assumptions about the statistics of the data. Instead they use the context of the data being encoded and the past history of the data to provide more efficient compression. We will look at a number of schemes that are principally used for the compression of text. These schemes use the context in which the data occurs in different ways.

6.2 Introduction

In Chapters 3 and 4 we learned that we get more compression when the message that is being coded has a more skewed set of probabilities. By "skewed" we mean that certain symbols occur with much higher probability than others in the sequence to be encoded. So it makes sense to look for ways to represent the message that would result in greater skew. One very effective way to do so is to look at the probability of occurrence of a letter in the context in which it occurs. That is, we do not look at each symbol in a sequence as if it had just happened out of the blue. Instead, we examine the history of the sequence before determining the likely probabilities of different values that the symbol can take.

In the case of English text, Shannon [8] showed the role of context in two very interesting experiments. In the first, a portion of text was selected and a subject (possibly his wife, Mary Shannon) was asked to guess each letter. If she guessed correctly, she was told that she was correct and moved on to the next letter. If she guessed incorrectly, she was told the correct answer and again moved on to the next letter. Here is a result from one of these experiments. Here the dashes represent the letters that were correctly guessed.

Actual Text	THE ROOM WAS NOT VERY LIGHT A SMALL OBLONG
Subject Performance	_ _ _ _ROO _ _ _ _ _ _NOT_V_ _ _ _ _I_ _ _ _ _ _SM_ _ _ _OBL_ _ _

Notice that there is a good chance that the subject will guess the letter, especially if the letter is at the end of a word or if the word is clear from the context. If we now represent the original sequence by the subject performance, we would get a very different set of probabilities for the values that each element of the sequence takes on. The probabilities are definitely much more skewed in the second row: the "letter" _ occurs with high probability. If a mathematical twin of the subject were available at the other end, we could send the "reduced" sentence in the second row and have the twin go through the same guessing process to come up with the original sequence.

In the second experiment, the subject was allowed to continue guessing until she had guessed the correct letter and the number of guesses required to correctly predict the letter was noted. Again, most of the time the subject guessed correctly, resulting in 1 being the most probable number. The existence of a mathematical twin at the receiving end would allow this skewed sequence to represent the original sequence to the receiver. Shannon used his experiments to come up with upper and lower bounds for the English alphabet (1.3 bits per letter and 0.6 bits per letter, respectively).

The difficulty with using these experiments is that the human subject was much better at predicting the next letter in a sequence than any mathematical predictor we can develop. Grammar is hypothesized to be innate to humans [64], in which case development of a predictor as efficient as a human for language is not possible in the near future. However, the experiments do provide an approach to compression that is useful for compression of all types of sequences, not simply language representations.

If a sequence of symbols being encoded does not consist of independent occurrences of the symbols, then the knowledge of which symbols have occurred in the neighborhood of the symbol being encoded will give us a much better idea of the value of the symbol being encoded. If we know the context in which a symbol occurs we can guess with a much greater likelihood of success what the value of the symbol is. This is just another way of saying that, given the context, some symbols will occur with much higher probability than others. That is, the probability distribution given the context is more skewed. If the context is known to both encoder and decoder, we can use this skewed distribution to perform the encoding, thus increasing the level of compression. The decoder can use its knowledge of the context to determine the distribution to be used for decoding. If we can somehow group like contexts together, it is quite likely that the symbols following these contexts will be the same, allowing for the use of some very simple and efficient compression strategies. We can see that the context can play an important role in enhancing compression, and in this chapter we will look at several different ways of using the context.

Consider the encoding of the word *probability*. Suppose we have already encoded the first four letters, and we want to code the fifth letter, a. If we ignore the first four letters, the probability of the letter a is about 0.06. If we use the information that the previous letter is b, this reduces the probability of several letters such as q and z occurring and boosts the probability of an a occurring. In this example, b would be the first-order context for a, ob would be the second-order context for a, and so on. Using more letters to define the context in which a occurs, or higher-order contexts, will generally increase the probability

of the occurrence of a in this example, and hence reduce the number of bits required to encode its occurrence. Therefore, what we would like to do is to encode each letter using the probability of its occurrence with respect to a context of high order.

If we want to have probabilities with respect to all possible high-order contexts, this might be an overwhelming amount of information. Consider an alphabet of size M. The number of first-order contexts is M, the number of second-order contexts is M^2, and so on. Therefore, if we wanted to encode a sequence from an alphabet of size 256 using contexts of order 5, we would need 256^5, or about 1.09951×10^{12} probability distributions! This is not a practical alternative. A set of algorithms that resolve this problem in a very simple and elegant way is based on the *prediction with partial match (ppm)* approach. We will describe this in the next section.

6.3 Prediction with Partial Match (*ppm*)

The best-known context-based algorithm is the *ppm* algorithm, first proposed by Cleary and Witten [65] in 1984. It has not been as popular as the various Ziv-Lempel-based algorithms mainly because of the faster execution speeds of the latter algorithms. Lately, with the development of more efficient variants, *ppm*-based algorithms are becoming increasingly more popular.

The idea of the *ppm* algorithm is elegantly simple. We would like to use large contexts to determine the probability of the symbol being encoded. However, the use of large contexts would require us to estimate and store an extremely large number of conditional probabilities, which might not be feasible. Instead of estimating these probabilities ahead of time, we can reduce the burden by estimating the probabilities as the coding proceeds. This way we only need to store those contexts that have occurred in the sequence being encoded. This is a much smaller number than the number of all possible contexts. While this mitigates the problem of storage, it also means that, especially at the beginning of an encoding, we will need to code letters that have not occurred previously in this context. In order to handle this situation, the source coder alphabet always contains an escape symbol, which is used to signal that the letter to be encoded has not been seen in this context.

6.3.1 The Basic Algorithm

The basic algorithm initially attempts to use the largest context. The size of the largest context is predetermined. If the symbol to be encoded has not previously been encountered in this context, an escape symbol is encoded and the algorithm attempts to use the next smaller context. If the symbol has not occurred in this context either, the size of the context is further reduced. This process continues until either we obtain a context that has previously been encountered with this symbol, or we arrive at the conclusion that the symbol has not been encountered previously in *any* context. In this case, we use a probability of $1/M$ to encode the symbol, where M is the size of the source alphabet. For example, when coding the a of *probability*, we would first attempt to see if the string *proba* has previously occurred—that is, if a had previously occurred in the context of *prob*. If not, we would encode an

escape and see if *a* had occurred in the context of *rob*. If the string *roba* had not occurred previously, we would again send an escape symbol and try the context *ob*. Continuing in this manner, we would try the context *b*, and failing that, we would see if the letter *a* (with a zero-order context) had occurred previously. If *a* was being encountered for the first time, we would use a model in which all letters occur with equal probability to encode *a*. This equiprobable model is sometimes referred to as the context of order -1.

As the development of the probabilities with respect to each context is an adaptive process, each time a symbol is encountered, the count corresponding to that symbol is updated. The number of counts to be assigned to the escape symbol is not obvious, and a number of different approaches have been used. One approach used by Cleary and Witten is to give the escape symbol a count of one, thus inflating the total count by one. Cleary and Witten call this method of assigning counts Method A, and the resulting algorithm *ppma*. We will describe some of the other ways of assigning counts to the escape symbol later in this section.

Before we delve into some of the details, let's work through an example to see how all this works together. As we will be using arithmetic coding to encode the symbols, you might wish to refresh your memory of the arithmetic coding algorithms.

Example 6.3.1:

Let's encode the sequence

*this*ƀ*is*ƀ*the*ƀ*tithe*

Assuming we have already encoded the initial seven characters *this*ƀ*is*, the various counts and *Cum_Count* arrays to be used in the arithmetic coding of the symbols are shown in Tables 6.1–6.4. In this example, we are assuming that the longest context length is two. This is a rather small value and is used here to keep the size of the example reasonably small. A more common value for the longest context length is five.

We will assume that the word length for arithmetic coding is six. Thus, $l = 000000$ and $u = 111111$. As *this*ƀ*is* has already been encoded, the next letter to be encoded is ƀ. The second-order context for this letter is *is*. Looking at Table 6.4, we can see that the letter ƀ

TABLE 6.1 Count array for −1 order context.

Letter	Count	*Cum_Count*
t	1	1
h	1	2
i	1	3
s	1	4
e	1	5
ƀ	1	6
Total Count		6

TABLE 6.2 Count array for zero-order context.

Letter	Count	*Cum_Count*
t	1	1
h	1	2
i	2	4
s	2	6
b̸	1	7
⟨*Esc*⟩	1	8
Total Count		8

TABLE 6.3 Count array for first-order contexts.

Context	Letter	Count	*Cum_Count*
t	*h*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
h	*i*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
i	*s*	2	2
	⟨*Esc*⟩	1	3
	Total Count		3
b̸	*i*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
s	*b̸*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2

is the first letter in this context with a *Cum_Count* value of 1. As the *Total_Count* in this case is 2, the update equations for the lower and upper limits are

$$l = 0 + \left\lfloor (63 - 0 + 1) \times \frac{0}{2} \right\rfloor = 0 = 000000$$

$$u = 0 + \left\lfloor (63 - 0 + 1) \times \frac{1}{2} \right\rfloor - 1 = 31 = 011111.$$

TABLE 6.4 Count array for second-order contexts.

Context	Letter	Count	*Cum_Count*
th	*i*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
hi	*s*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
is	*b̸*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
sb̸	*i*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
b̸i	*s*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2

As the MSBs of both l and u are the same, we shift that bit out, shift a 0 into the LSB of l, and a 1 into the LSB of u. The transmitted sequence, lower limit, and upper limit after the update are

$$\begin{aligned} \text{Transmitted sequence}: &\ 0 \\ l: &\ 000000 \\ u: &\ 111111 \end{aligned}$$

We also update the counts in Tables 6.2–6.4.

The next letter to be encoded in the sequence is t. The second-order context is *sb̸*. Looking at Table 6.4, we can see that t has not appeared before in this context. We therefore encode an escape symbol. Using the counts listed in Table 6.4, we update the lower and upper limits:

$$\begin{aligned} l &= 0 + \left\lfloor (63 - 0 + 1) \times \frac{1}{2} \right\rfloor = 32 = 100000 \\ u &= 0 + \left\lfloor (63 - 0 + 1) \times \frac{2}{2} \right\rfloor - 1 = 63 = 111111. \end{aligned}$$

Again, the MSBs of l and u are the same, so we shift the bit out and shift 0 into the LSB of l, and 1 into u, restoring l to a value of 0 and u to a value of 63. The transmitted sequence is now 01. After transmitting the escape, we look at the first-order context of t, which is $\not b$. Looking at Table 6.3, we can see that t has not previously occurred in this context. To let the decoder know this, we transmit another escape. Updating the limits, we get

$$l = 0 + \left\lfloor (63 - 0 + 1) \times \frac{1}{2} \right\rfloor = 32 = 100000$$

$$u = 0 + \left\lfloor (63 - 0 + 1) \times \frac{2}{2} \right\rfloor - 1 = 63 = 111111.$$

As the MSBs of l and u are the same, we shift the MSB out and shift 0 into the LSB of l and 1 into the LSB of u. The transmitted sequence is now 011. Having escaped out of the first-order contexts, we examine Table 6.5, the updated version of Table 6.2, to see if we can encode t using a zero-order context. Indeed we can, and using the *Cum_Count* array, we can update l and u:

$$l = 0 + \left\lfloor (63 - 0 + 1) \times \frac{0}{9} \right\rfloor = 0 = 000000$$

$$u = 0 + \left\lfloor (63 - 0 + 1) \times \frac{1}{9} \right\rfloor - 1 = 6 = 000110.$$

TABLE 6.5 Updated count array for zero-order context.

Letter	Count	*Cum_Count*
t	1	1
h	1	2
i	2	4
s	2	6
$\not b$	2	8
$\langle Esc \rangle$	1	9
Total Count		9

The three most significant bits of both l and u are the same, so we shift them out. After the update we get

$$\begin{aligned} \text{Transmitted sequence}: &\ 011000 \\ l: &\ 000000 \\ u: &\ 110111 \end{aligned}$$

The next letter to be encoded is h. The second-order context $\not{b}t$ has not occurred previously, so we move directly to the first-order context t. The letter h has occurred previously in this context, so we update l and u and obtain

$$\begin{aligned} \text{Transmitted sequence}: &\ 0110000 \\ l: &\ 000000 \\ u: &\ 110101 \end{aligned}$$

TABLE 6.6 Count array for zero-order context.

Letter	Count	*Cum_Count*
t	2	2
h	2	4
i	2	6
s	2	8
$\not{b}$	2	10
$\langle Esc \rangle$	1	11
Total Count		11

TABLE 6.7 Count array for first-order contexts.

Context	Letter	Count	*Cum_Count*
t	h	2	2
	$\langle Esc \rangle$	1	3
	Total Count		3
h	i	1	1
	$\langle Esc \rangle$	1	2
	Total Count		2
i	s	2	2
	$\langle Esc \rangle$	1	3
	Total Count		3
$\not{b}$	i	1	1
	t	1	2
	$\langle Esc \rangle$	1	3
	Total Count		3
s	$\not{b}$	2	2
	$\langle Esc \rangle$	1	3
	Total Count		3

TABLE 6.8 Count array for second-order contexts.

Context	Letter	Count	*Cum_Count*
th	*i*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
hi	*s*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
is	*b̸*	2	2
	⟨*Esc*⟩	1	3
	Total Count		3
sb̸	*i*	1	1
	t	1	2
	⟨*Esc*⟩	1	3
	Total Count		3
b̸i	*s*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
b̸t	*h*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2

The method of encoding should now be clear. At this point the various counts are as shown in Tables 6.6–6.8. ♦

Now that we have an idea of how the *ppm* algorithm works, let's examine some of the variations.

6.3.2 The Escape Symbol

In our example we used a count of one for the escape symbol, thus inflating the total count in each context by one. Cleary and Witten call this Method A, and the corresponding algorithm is referred to as *ppma*. There is really no obvious justification for assigning a count of one to the escape symbol. For that matter, there is no obvious method of assigning counts to the escape symbol. There have been various methods reported in the literature.

Another method described by Cleary and Witten is to reduce the counts of each symbol by one and assign these counts to the escape symbol. For example, suppose in a given

TABLE 6.9 Counts using Method A.

Context	Symbol	Count
prob	*a*	10
	l	9
	o	3
	⟨*Esc*⟩	1
Total Count		23

TABLE 6.10 Counts using Method B.

Context	Symbol	Count
prob	*a*	9
	l	8
	o	2
	⟨*Esc*⟩	3
Total Count		22

sequence *a* occurs 10 times in the context of *prob*, *l* occurs 9 times, and *o* occurs 3 times in the same context (e.g., *problem, proboscis*, etc.). In Method A we assign a count of one to the escape symbol, resulting in a total count of 23, which is one more than the number of times *prob* has occurred. The situation is shown in Table 6.9.

In this second method, known as Method B, we reduce the count of each of the symbols *a*, *l*, and *o* by one and give the escape symbol a count of three, resulting in the counts shown in Table 6.10.

The reasoning behind this approach is that if in a particular context more symbols can occur, there is a greater likelihood that there is a symbol in this context that has not occurred before. This increases the likelihood that the escape symbol will be used. Therefore, we should assign a higher probability to the escape symbol.

A variant of Method B, appropriately named Method C, was proposed by Moffat [66]. In Method C, the count assigned to the escape symbol is the number of symbols that have occurred in that context. In this respect, Method C is similar to Method B. The difference comes in the fact that, instead of "robbing" this from the counts of individual symbols, the total count is inflated by this amount. This situation is shown in Table 6.11.

While there is some variation in the performance depending on the characteristics of the data being encoded, of the three methods for assigning counts to the escape symbol, on the average, Method C seems to provide the best performance.

6.3.3 Length of Context

It would seem that as far as the maximum length of the contexts is concerned, more is better. However, this is not necessarily true. A longer maximum length will usually result

TABLE 6.11 Counts using Method C.

Context	Symbol	Count
prob	*a*	10
	l	9
	o	3
	⟨*Esc*⟩	3
Total Count		25

in a higher probability if the symbol to be encoded has a nonzero count with respect to that context. However, a long maximum length also means a higher probability of long sequences of escapes, which in turn can increase the number of bits used to encode the sequence. If we plot the compression performance versus maximum context length, we see an initial sharp increase in performance until some value of the maximum length, followed by a steady drop as the maximum length is further increased. The value at which we see a downturn in performance changes depending on the characteristics of the source sequence.

An alternative to the policy of a fixed maximum length is used in the algorithm *ppm** [67]. This algorithm uses the fact that long contexts that give only a single prediction are seldom followed by a new symbol. If *mike* has always been followed by *y* in the past, it will probably not be followed by *b̸* the next time it is encountered. Contexts that are always followed by the same symbol are called *deterministic* contexts. The *ppm** algorithm first looks for the longest deterministic context. If the symbol to be encoded does not occur in that context, an escape symbol is encoded and the algorithm defaults to the maximum context length. This approach seems to provide a small but significant amount of improvement over the basic algorithm. Currently, the best variant of the *ppm** algorithm is the *ppmz* algorithm by Charles Bloom. Details of the *ppmz* algorithm as well as implementations of the algorithm can be found at *http://www.cbloom.com/src/ppmz.html.*

6.3.4 The Exclusion Principle

The basic idea behind arithmetic coding is the division of the unit interval into subintervals, each of which represents a particular letter. The smaller the subinterval, the more bits are required to distinguish it from other subintervals. If we can reduce the number of symbols to be represented, the number of subintervals goes down as well. This in turn means that the sizes of the subintervals increase, leading to a reduction in the number of bits required for encoding. The exclusion principle used in *ppm* provides this kind of reduction in rate. Suppose we have been compressing a text sequence and come upon the sequence *proba*, and suppose we are trying to encode the letter *a*. Suppose also that the state of the two-letter context *ob* and the one-letter context *b* are as shown in Table 6.12.

First we attempt to encode *a* with the two-letter context. As *a* does not occur in this context, we issue an escape symbol and reduce the size of the context. Looking at the table for the one-letter context *b*, we see that *a* does occur in this context with a count of 4 out of a total possible count of 21. Notice that other letters in this context include *l* and *o*. However,

TABLE 6.12 Counts for exclusion example.

Context	Symbol	Count
ob	*l*	10
	o	3
	⟨*Esc*⟩	2
Total Count		15
b	*l*	5
	o	3
	a	4
	r	2
	e	2
	⟨*Esc*⟩	5
Total Count		21

TABLE 6.13 Modified table used for exclusion example.

Context	Symbol	Count
b	*a*	4
	r	2
	e	2
	⟨*Esc*⟩	3
Total Count		11

by sending the escape symbol in the context of *ob*, we have already signalled to the decoder that the symbol being encoded is not any of the letters that have previously been encountered in the context of *ob*. Therefore, we can increase the size of the subinterval corresponding to *a* by temporarily removing *l* and *o* from the table. Instead of using Table 6.12, we use Table 6.13 to encode *a*. This exclusion of symbols from contexts on a temporary basis can result in cumulatively significant savings in terms of rate.

You may have noticed that we keep talking about small but significant savings. In lossless compression schemes, there is usually a basic principle, such as the idea of prediction with partial match, followed by a host of relatively small modifications. The importance of these modifications should not be underestimated because often together they provide the margin of compression that makes a particular scheme competitive.

6.4 The Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) algorithm also uses the context of the symbol being encoded, but in a very different way, for lossless compression. The transform that

is a major part of this algorithm was developed by Wheeler in 1983. However, the BWT compression algorithm, which uses this transform, saw the light of day in 1994 [68]. Unlike most of the previous algorithms we have looked at, the BWT algorithm requires that the entire sequence to be coded be available to the encoder before the coding takes place. Also, unlike most of the previous algorithms, the decoding procedure is not immediately evident once we know the encoding procedure. We will first describe the encoding procedure. If it is not clear how this particular encoding can be reversed, bear with us and we will get to it.

The algorithm can be summarized as follows. Given a sequence of length N, we create $N-1$ other sequences where each of these $N-1$ sequences is a cyclic shift of the original sequence. These N sequences are arranged in lexicographic order. The encoder then transmits the sequence of length N created by taking the last letter of each sorted, cyclically shifted, sequence. This sequence of last letters L, and the position of the original sequence in the sorted list, are coded and sent to the decoder. As we shall see, this information is sufficient to recover the original sequence.

We start with a sequence of length N and end with a representation that contains $N+1$ elements. However, this sequence has a structure that makes it highly amenable to compression. In particular we will use a method of coding called move-to-front (*mtf*), which is particularly effective on the type of structure exhibited by the sequence L.

Before we describe the *mtf* approach, let us work through an example to generate the L sequence.

Example 6.4.1:

Let's encode the sequence

*this*b̸*is*b̸*the*

We start with all the cyclic permutations of this sequence. As there are a total of 11 characters, there are 11 permutations, shown in Table 6.14.

TABLE 6.14 Permutations of *this*b̸*is*b̸*the*.

0	*t*	*h*	*i*	*s*	b̸	*i*	*s*	b̸	*t*	*h*	*e*
1	*h*	*i*	*s*	b̸	*i*	*s*	b̸	*t*	*h*	*e*	*t*
2	*i*	*s*	b̸	*i*	*s*	b̸	*t*	*h*	*e*	*t*	*h*
3	*s*	b̸	*i*	*s*	b̸	*t*	*h*	*e*	*t*	*h*	*i*
4	b̸	*i*	*s*	b̸	*t*	*h*	*e*	*t*	*h*	*i*	*s*
5	*i*	*s*	b̸	*t*	*h*	*e*	*t*	*h*	*i*	*s*	b̸
6	*s*	b̸	*t*	*h*	*e*	*t*	*h*	*i*	*s*	b̸	*i*
7	b̸	*t*	*h*	*e*	*t*	*h*	*i*	*s*	b̸	*i*	*s*
8	*t*	*h*	*e*	*t*	*h*	*i*	*s*	b̸	*i*	*s*	b̸
9	*h*	*e*	*t*	*h*	*i*	*s*	b̸	*i*	*s*	b̸	*t*
10	*e*	*t*	*h*	*i*	*s*	b̸	*i*	*s*	b̸	*t*	*h*

TABLE 6.15 Sequences sorted into lexicographic order.

0	ƀ	*i*	*s*	ƀ	*t*	*h*	*e*	*t*	*h*	*i*	*s*
1	ƀ	*t*	*h*	*e*	*t*	*h*	*i*	*s*	ƀ	*i*	*s*
2	*e*	*t*	*h*	*i*	*s*	ƀ	*i*	*s*	ƀ	*t*	*h*
3	*h*	*e*	*t*	*h*	*i*	*s*	ƀ	*i*	*s*	ƀ	*t*
4	*h*	*i*	*s*	ƀ	*i*	*s*	ƀ	*t*	*h*	*e*	*t*
5	*i*	*s*	ƀ	*i*	*s*	ƀ	*t*	*h*	*e*	*t*	*h*
6	*i*	*s*	ƀ	*t*	*h*	*e*	*t*	*h*	*i*	*s*	ƀ
7	*s*	ƀ	*i*	*s*	ƀ	*t*	*h*	*e*	*t*	*h*	*i*
8	*s*	ƀ	*t*	*h*	*e*	*t*	*h*	*i*	*s*	ƀ	*i*
9	*t*	*h*	*e*	*t*	*h*	*i*	*s*	ƀ	*i*	*s*	ƀ
10	*t*	*h*	*i*	*s*	ƀ	*i*	*s*	ƀ	*t*	*h*	*e*

Now let's sort these sequences in lexicographic (dictionary) order (Table 6.15). The sequence of last letters L in this case is

$$L : sshtthƀiiƀe$$

Notice how like letters have come together. If we had a longer sequence of letters, the *runs* of like letters would have been even longer. The *mtf* algorithm, which we will describe later, takes advantage of these runs.

The original sequence appears as sequence number 10 in the sorted list, so the encoding of the sequence consists of the sequence L and the index value 10. ♦

Now that we have an encoding of the sequence, let's see how we can decode the original sequence by using the sequence L and the index to the original sequence in the sorted list. The important thing to note is that all the elements of the initial sequence are contained in L. We just need to figure out the permutation that will let us recover the original sequence.

The first step in obtaining the permutation is to generate the sequence F consisting of the first element of each row. That is simple to do because we lexicographically ordered the sequences. Therefore, the sequence F is simply the sequence L in lexicographic order. In our example this means that F is given as

$$F : ƀƀehhiisstt$$

We can use L and F to generate the original sequence. Look at Table 6.15 containing the cyclically shifted sequences sorted in lexicographic order. Because each row is a cyclical shift, the letter in the first column of any row is the letter appearing after the last column in the row in the original sequence. If we know that the original sequence is in the kth row, then we can begin unraveling the original sequence starting with the kth element of F.

Example 6.4.2:

In our example

$$F = \begin{bmatrix} \not{b} \\ \not{b} \\ e \\ h \\ h \\ i \\ i \\ s \\ s \\ t \\ t \end{bmatrix} \quad L = \begin{bmatrix} s \\ s \\ h \\ t \\ t \\ h \\ \not{b} \\ i \\ i \\ \not{b} \\ e \end{bmatrix}$$

the original sequence is sequence number 10, so the first letter in of the original sequence is $F[10] = t$. To find the letter following t we look for t in the array L. There are two t's in L. Which should we use? The t in F that we are working with is the lower of two t's, so we pick the lower of two t's in L. This is $L[4]$. Therefore, the next letter in our reconstructed sequence is $F[4] = h$. The reconstructed sequence to this point is th. To find the next letter, we look for h in the L array. Again there are two h's. The h at $F[4]$is the lower of two h's in F, so we pick the lower of the two h's in L. This is the fifth element of L, so the next element in our decoded sequence is $F[5] = i$. The decoded sequence to this point is thi. The process continues as depicted in Figure 6.1 to generate the original sequence.

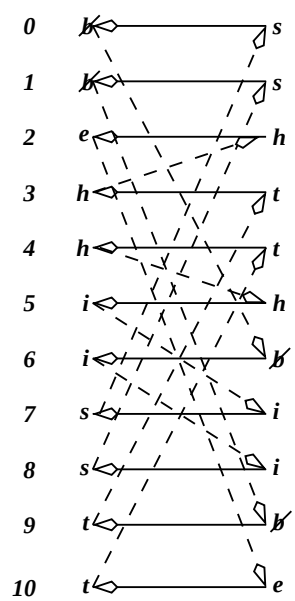

FIGURE 6.1 **Decoding process.** ♦

Why go through all this trouble? After all, we are going from a sequence of length N to another sequence of length N plus an index value. It appears that we are actually causing expansion instead of compression. The answer is that the sequence L can be compressed much more efficiently than the original sequence. Even in our small example we have runs of like symbols. This will happen a lot more when N is large. Consider a large sample of text that has been cyclically shifted and sorted. Consider all the rows of A beginning with *he*$\not b$. With high probability *he*$\not b$ would be preceded by t. Therefore, in L we would get a long run of ts.

6.4.1 Move-to-Front Coding

A coding scheme that takes advantage of long runs of identical symbols is the move-to-front (mtf) coding. In this coding scheme, we start with some initial listing of the source alphabet. The symbol at the top of the list is assigned the number 0, the next one is assigned the number 1, and so on. The first time a particular symbol occurs, the number corresponding to its place in the list is transmitted. Then it is moved to the top of the list. If we have a run of this symbol, we transmit a sequence of 0s. This way, long runs of different symbols get transformed to a large number of 0s. Applying this technique to our example does not produce very impressive results due to the small size of the sequence, but we can see how the technique functions.

Example 6.4.3:

Let's encode $L = ssht\,t\,h\not b\,i\,i\not b\,e$. Let's assume that the source alphabet is given by

$$\mathcal{A} = \{\not b, e, h, i, s, t\}.$$

We start out with the assignment

0	1	2	3	4	5
$\not b$	e	h	i	s	t

The first element of L is s, which gets encoded as a 4. We then move s to the top of the list, which gives us

0	1	2	3	4	5
s	$\not b$	e	h	i	t

The next s is encoded as 0. Because s is already at the top of the list, we do not need to make any changes. The next letter is h, which we encode as 3. We then move h to the top of the list:

0	1	2	3	4	5
h	*s*	*b̸*	*e*	*i*	*t*

The next letter is *t*, which gets encoded as 5. Moving *t* to the top of the list, we get

0	1	2	3	4	5
t	*h*	*s*	*b̸*	*e*	*i*

The next letter is also a *t*, so that gets encoded as a 0.

Continuing in this fashion, we get the sequence

4 0 3 5 0 1 3 5 0 1 5

As we warned, the results are not too impressive with this small sequence, but we can see how we would get large numbers of 0s and small values if the sequence to be encoded was longer. ♦

6.5 Associative Coder of Buyanovsky (ACB)

A different approach to using contexts for compression is employed by the eponymous compression utility developed by George Buyanovsky. The details of this very efficient coder are not well known; however, the way the context is used is interesting and we will briefly describe this aspect of ACB. More detailed descriptions are available in [69] and [70]. The ACB coder develops a sorted dictionary of all encountered contexts. In this it is similar to other context based encoders. However, it also keeps track of the *contents* of these contexts. The content of a context is what appears after the context. In a traditional left-to-right reading of text, the contexts are unbounded to the left and the contents to the right (to the limits of text that has already been encoded). When encoding the coder searches for the longest match to the current context reading right to left. This again is not an unusual thing to do. What is interesting is what the coder does after the best match is found. Instead of simply examining the *content* corresponding to the best matched context, the coder also examines the *contents* of the coders in the neighborhood of the best matched contexts. Fenwick [69] describes this process as first finding an anchor point then searching the *contents* of the neighboring contexts for the best match. The location of the anchor point is known to both the encoder and the decoder. The location of the best *content* match is signalled to the decoder by encoding the offset δ of the context of this *content* from the anchor point. We have not specified what we mean by "best" match. The coder takes the utilitarian approach that the best match is the one that ends up providing the most compression. Thus, a longer match farther away from the anchor may not be as advantageous as a shorter match closer to the anchor because of the number of bits required to encode δ. The length of the match λ is also sent to the decoder.

The interesting aspect of this scheme is that it moves away from the idea of exactly matching the past. It provides a much richer environment and flexibility to enhance the compression and will, hopefully, provide a fruitful avenue for further research.

6.6 Dynamic Markov Compression

Quite often the probabilities of the value that the next symbol in a sequence takes on depend not only on the current value but on the past values as well. The *ppm* scheme relies on this longer-range correlation. The *ppm* scheme, in some sense, reflects the application, that is, text compression, for which it is most used. Dynamic Markov compression (DMC), introduced by Cormack and Horspool [71], uses a more general framework to take advantage of relationships and correlations, or contexts, that extend beyond a single symbol.

Consider the sequence of pixels in a scanned document. The sequence consists of runs of black and white pixels. If we represent black by 0 and white by 1, we have runs of 0s and 1s. If the current value is 0, the probability that the next value is 0 is higher than if the current value was 1. The fact that we have two different sets of probabilities is reflected in the two-state model shown in Figure 6.2. Consider state A. The probability of the next value being 1 changes depending on whether we reached state A from state B or from state A itself. We can have the model reflect this by *cloning* state A, as shown in Figure 6.3, to create state A'. Now if we see a white pixel after a run of black pixels, we go to state A'. The probability that the next value will be 1 is very high in this state. This way, when we estimate probabilities for the next pixel value, we take into account not only the value of the current pixel but also the value of the previous pixel.

This process can be continued as long as we wish to take into account longer and longer histories. "As long as we wish" is a rather vague statement when it comes to implementing the algorithm. In fact, we have been rather vague about a number of implementation issues. We will attempt to rectify the situation.

There are a number of issues that need to be addressed in order to implement this algorithm:

1. What is the initial number of states?
2. How do we estimate probabilities?

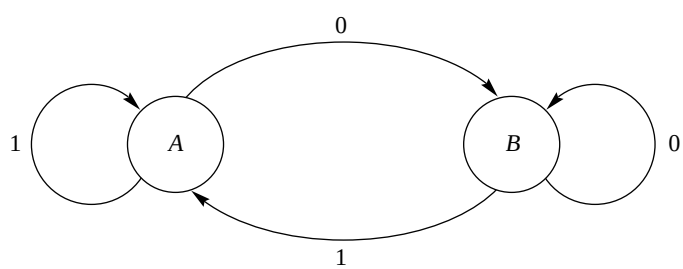

FIGURE 6.2 **A two-state model for binary sequences.**

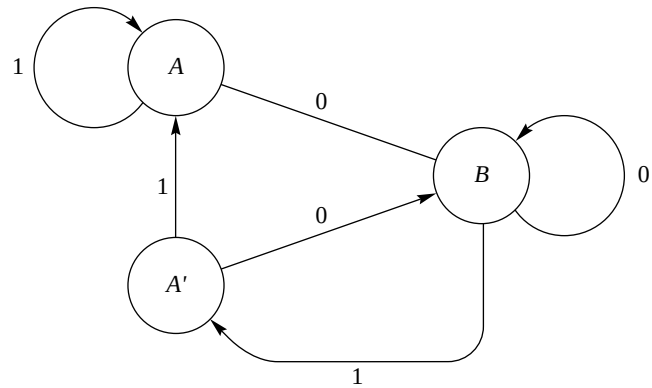

FIGURE 6. 3 A three-state model obtained by cloning.

3. How do we decide when a state needs to be cloned?

4. What do we do when the number of states becomes too large?

Let's answer each question in turn.

We can start the encoding process with a single state with two self-loops for 0 and 1. This state can be cloned to two and then a higher number of states. In practice it has been found that, depending on the particular application, it is more efficient to start with a larger number of states than one.

The probabilities from a given state can be estimated by simply counting the number of times a 0 or a 1 occurs in that state divided by the number of times the particular state is occupied. For example, if in state V the number of times a 0 occurs is denoted by n_0^V and the number of times a 1 occurs is denoted by n_1^V, then

$$P(0|V) = \frac{n_0^V}{n_0^V + n_1^V}$$

$$P(1|V) = \frac{n_1^V}{n_0^V + n_1^V}.$$

What if a 1 has never previously occurred in this state? This approach would assign a probability of zero to the occurrence of a 1. This means that there will be no subinterval assigned to the possibility of a 1 occurring, and when it does occur, we will not be able to represent it. In order to avoid this, instead of counting from zero, we start the count of 1s and 0s with a small number c and estimate the probabilities as

$$P(0|V) = \frac{n_0^V + c}{n_0^V + n_1^V + 2c}$$

$$P(1|V) = \frac{n_1^V + c}{n_0^V + n_1^V + 2c}.$$

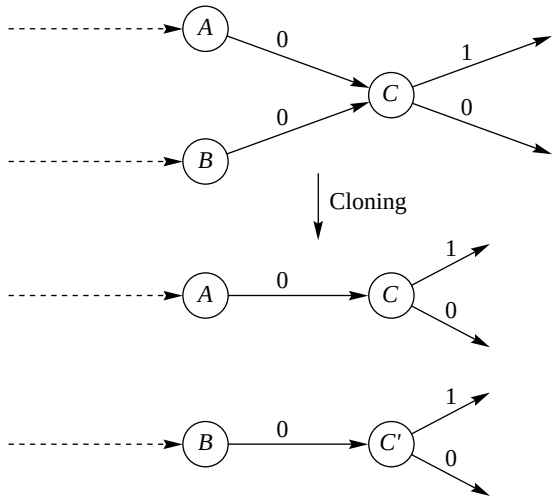

FIGURE 6.4 The cloning process.

Whenever we have two branches leading to a state, it can be cloned. And, theoretically, cloning is never harmful. By cloning we are providing additional information to the encoder. This might not reduce the rate, but it should never result in an increase in the rate. However, cloning does increase the complexity of the coding, and hence the decoding, process. In order to control the increase in the number of states, we should only perform cloning when there is a reasonable expectation of reduction in rate. We can do this by making sure that both paths leading to the state being considered for cloning are used often enough. Consider the situation shown in Figure 6.4. Suppose the current state is A and the next state is C. As there are two paths entering C, C is a candidate for cloning. Cormack and Horspool suggest that C be cloned if $n_0^A > T_1$ and $n_0^B > T_2$, where T_1 and T_2 are threshold values set by the user. If there are more than three paths leading to a candidate for cloning, then we check that both the number of transitions from the current state is greater than T_1 and the number of transitions from all other states to the candidate state is greater than T_2.

Finally, what do we do when, for practical reasons, we cannot accommodate any more states? A simple solution is to restart the algorithm. In order to make sure that we do not start from ground zero every time, we can train the initial state configuration using a certain number of past inputs.

6.7 Summary

The context in which a symbol occurs can be very informative about the value that the symbol takes on. If this context is known to the decoder then this information need not be encoded: it can be inferred by the decoder. In this chapter we have looked at several creative ways in which the knowledge of the context can be used to provide compression.

Further Reading

1. The basic *ppm* algorithm is described in detail in *Text Compression*, by T.C. Bell, J.G. Cleary, and I.H. Witten [1].
2. For an excellent description of Burrows-Wheeler Coding, including methods of implementation and improvements to the basic algorithm, see "Burrows-Wheeler Compression," by P. Fenwick [72] in *Lossless Compression Handbook*.
3. The ACB algorithm is described in "Symbol Ranking and ACB Compression," by P. Fenwick [69] in the *Lossless Compression Handbook*, and in *Data Compression: The Complete Reference* by D. Salomon [70]. The chapter by Fenwick also explores compression schemes based on Shannon's experiments.

6.8 Projects and Problems

1. Decode the bitstream generated in Example 6.3.1. Assume you have already decoded *this␣is* and Tables 6.1–6.4 are available to you.
2. Given the sequence *the␣beta␣cat␣ate␣the␣ceta␣hat*:
   - **(a)** Encode the sequence using the *ppma* algorithm and an adaptive arithmetic coder. Assume a six-letter alphabet $\{h, e, t, a, c, ␣\}$.
   - **(b)** Decode the encoded sequence.
3. Given the sequence *eta␣ceta␣and␣beta␣ceta*:
   - **(a)** Encode using the Burrows-Wheeler transform and move-to-front coding.
   - **(b)** Decode the encoded sequence.
4. A sequence is encoded using the Burrows-Wheeler transform. Given $L = elbkkee$, and index = 5 (we start counting from 1, not 0), find the original sequence.

7

Lossless Image Compression

7.1 Overview

In this chapter we examine a number of schemes used for lossless compression of images. We will look at schemes for compression of grayscale and color images as well as schemes for compression of binary images. Among these schemes are several that are a part of international standards.

7.2 Introduction

In the previous chapters we have focused on compression techniques. Although some of them may apply to some preferred applications, the focus has been on the technique rather than on the application. However, there are certain techniques for which it is impossible to separate the technique from the application. This is because the techniques rely upon the properties or characteristics of the application. Therefore, we have several chapters in this book that focus on particular applications. In this chapter we will examine techniques specifically geared toward lossless image compression. Later chapters will examine speech, audio, and video compression.

In the previous chapters we have seen that a more skewed set of probabilities for the message being encoded results in better compression. In Chapter 6 we saw how the use of context to obtain a skewed set of probabilities can be especially effective when encoding text. We can also transform the sequence (in an invertible fashion) into another sequence that has the desired property in other ways. For example, consider the following sequence:

1	2	5	7	2	−2	0	−5	−3	−1	1	−2	−7	−4	−2	1	3	4

If we consider this sample to be fairly typical of the sequence, we can see that the probability of any given number being in the range from −7 to 7 is about the same. If we were to encode this sequence using a Huffman or arithmetic code, we would use almost 4 bits per symbol.

Instead of encoding this sequence directly, we could do the following: add two to the previous number in the sequence and send the difference between the current element in the sequence and this *predicted* value. The transmitted sequence would be

1	−1	1	0	−7	−4	0	−7	0	0	0	−5	−7	1	0	1	0	−1

This method uses a rule (add two) and the history (value of the previous symbol) to generate the new sequence. If the rule by which this *residual sequence* was generated is known to the decoder, it can recover the original sequence from the residual sequence. The length of the residual sequence is the same as the original sequence. However, notice that the residual sequence is much more likely to contain 0s, 1s, and −1s than other values. That is, the probability of 0, 1, and −1 will be significantly higher than the probabilities of other numbers. This, in turn, means that the entropy of the residual sequence will be low and, therefore, provide more compression.

We used a particular method of prediction in this example (add two to the previous element of the sequence) that was specific to this sequence. In order to get the best possible performance, we need to find the prediction approach that is best suited to the particular data we are dealing with. We will look at several prediction schemes used for lossless image compression in the following sections.

7.2.1 The Old JPEG Standard

The Joint Photographic Experts Group (JPEG) is a joint ISO/ITU committee responsible for developing standards for continuous-tone still-picture coding. The more famous standard produced by this group is the lossy image compression standard. However, at the time of the creation of the famous JPEG standard, the committee also created a lossless standard [73]. At this time the standard is more or less obsolete, having been overtaken by the much more efficient JPEG-LS standard described later in this chapter. However, the old JPEG standard is still useful as a first step into examining predictive coding in images.

The old JPEG lossless still compression standard [73] provides eight different predictive schemes from which the user can select. The first scheme makes no prediction. The next seven are listed below. Three of the seven are one-dimensional predictors, and four are two-dimensional prediction schemes. Here, $I(i, j)$ is the (i, j)th pixel of the original image, and $\hat{I}(i, j)$ is the predicted value for the (i, j)th pixel.

1 $$\hat{I}(i, j) = I(i-1, j) \tag{7.1}$$

2 $$\hat{I}(i, j) = I(i, j-1) \tag{7.2}$$

3 $$\hat{I}(i, j) = I(i-1, j-1) \tag{7.3}$$

4 $$\hat{I}(i, j) = I(i, j-1) + I(i-1, j) - I(i-1, j-1) \tag{7.4}$$

$$\mathbf{5} \quad \hat{I}(i,j) = I(i,j-1) + (I(i-1,j) - I(i-1,j-1))/2 \tag{7.5}$$

$$\mathbf{6} \quad \hat{I}(i,j) = I(i-1,j) + (I(i,j-1) - I(i-1,j-1))/2 \tag{7.6}$$

$$\mathbf{7} \quad \hat{I}(i,j) = (I(i,j-1) + I(i-1,j))/2 \tag{7.7}$$

Different images can have different structures that can be best exploited by one of these eight modes of prediction. If compression is performed in a nonreal-time environment—for example, for the purposes of archiving—all eight modes of prediction can be tried and the one that gives the most compression is used. The mode used to perform the prediction can be stored in a 3-bit header along with the compressed file. We encoded our four test images using the various JPEG modes. The residual images were encoded using adaptive arithmetic coding. The results are shown in Table 7.1.

The best results—that is, the smallest compressed file sizes—are indicated in bold in the table. From these results we can see that a different JPEG predictor is the best for the different images. In Table 7.2, we compare the best JPEG results with the file sizes obtained using GIF and PNG. Note that PNG also uses predictive coding with four possible predictors, where each row of the image can be encoded using a different predictor. The PNG approach is described in Chapter 5.

Even if we take into account the overhead associated with GIF, from this comparison we can see that the predictive approaches are generally better suited to lossless image compression than the dictionary-based approach when the images are "natural" gray-scale images. The situation is different when the images are graphic images or pseudocolor images. A possible exception could be the Earth image. The best compressed file size using the second JPEG mode and adaptive arithmetic coding is 32,137 bytes, compared to 34,276 bytes using GIF. The difference between the file sizes is not significant. We can see the reason by looking at the Earth image. Note that a significant portion of the image is the

TABLE 7.1 Compressed file size in bytes of the residual images obtained using the various JPEG prediction modes.

Image	JPEG 0	JPEG 1	JPEG 2	JPEG 3	JPEG 4	JPEG 5	JPEG 6	JPEG 7
Sena	53,431	37,220	31,559	38,261	31,055	**29,742**	33,063	32,179
Sensin	58,306	41,298	37,126	43,445	**32,429**	33,463	35,965	36,428
Earth	38,248	32,295	**32,137**	34,089	33,570	33,057	33,072	32,672
Omaha	56,061	**48,818**	51,283	53,909	53,771	53,520	52,542	52,189

TABLE 7.2 Comparison of the file sizes obtained using JPEG lossless compression, GIF, and PNG.

Image	Best JPEG	GIF	PNG
Sena	31,055	51,085	31,577
Sensin	32,429	60,649	34,488
Earth	32,137	34,276	26,995
Omaha	48,818	61,341	50,185

background, which is of a constant value. In dictionary coding, this would result in some very long entries that would provide significant compression. We can see that if the ratio of background to foreground were just a little different in this image, the dictionary method in GIF might have outperformed the JPEG approach. The PNG approach which allows the use of a different predictor (or no predictor) on each row, prior to dictionary coding significantly outperforms both GIF and JPEG on this image.

7.3 CALIC

The Context Adaptive Lossless Image Compression (CALIC) scheme, which came into being in response to a call for proposal for a new lossless image compression scheme in 1994 [74, 75], uses both context and prediction of the pixel values. The CALIC scheme actually functions in two modes, one for gray-scale images and another for bi-level images. In this section, we will concentrate on the compression of gray-scale images.

In an image, a given pixel generally has a value close to one of its neighbors. Which neighbor has the closest value depends on the local structure of the image. Depending on whether there is a horizontal or vertical edge in the neighborhood of the pixel being encoded, the pixel above, or the pixel to the left, or some weighted average of neighboring pixels may give the best prediction. How close the prediction is to the pixel being encoded depends on the surrounding texture. In a region of the image with a great deal of variability, the prediction is likely to be further from the pixel being encoded than in the regions with less variability.

In order to take into account all these factors, the algorithm has to make a determination of the environment of the pixel to be encoded. The only information that can be used to make this determination has to be available to both encoder and decoder.

Let's take up the question of the presence of vertical or horizontal edges in the neighborhood of the pixel being encoded. To help our discussion, we will refer to Figure 7.1. In this figure, the pixel to be encoded has been marked with an X. The pixel above is called the north pixel, the pixel to the left is the west pixel, and so on. Note that when pixel X is being encoded, all other marked pixels (N, W, NW, NE, WW, NN, NE, and NNE) are available to both encoder and decoder.

		NN	*NNE*
	NW	*N*	*NE*
WW	*W*	*X*	

FIGURE 7.1 Labeling the neighbors of pixel *X*.

We can get an idea of what kinds of boundaries may or may not be in the neighborhood of X by computing

$$d_h = |W - WW| + |N - NW| + |NE - N|$$
$$d_v = |W - NW| + |N - NN| + |NE - NNE|.$$

The relative values of d_h and d_v are used to obtain the initial prediction of the pixel X. This initial prediction is then refined by taking other factors into account. If the value of d_h is much higher than the value of d_v, this will mean there is a large amount of horizontal variation, and it would be better to pick N to be the initial prediction. If, on the other hand, d_v is much larger than d_h, this would mean that there is a large amount of vertical variation, and the initial prediction is taken to be W. If the differences are more moderate or smaller, the predicted value is a weighted average of the neighboring pixels.

The exact algorithm used by CALIC to form the initial prediction is given by the following pseudocode:

```
if d_h − d_v > 80
   X̂ ← N
else if d_v − d_h > 80
   X̂ ← W
else
{
        X̂ ← (N + W)/2 + (NE − NW)/4
        if d_h − d_v > 32
           X̂ ← (X̂ + N)/2
        else if d_v − d_h > 32
           X̂ ← (X̂ + W)/2
        else if d_h − d_v > 8
           X̂ ← (3X̂ + N)/4
        else if d_v − d_h > 8
           X̂ ← (3X̂ + W)/4
}
```

Using the information about whether the pixel values are changing by large or small amounts in the vertical or horizontal direction in the neighborhood of the pixel being encoded provides a good initial prediction. In order to refine this prediction, we need some information about the interrelationships of the pixels in the neighborhood. Using this information, we can generate an offset or refinement to our initial prediction. We quantify the information about the neighborhood by first forming the vector

$$[N, W, NW, NE, NN, WW, 2N - NN, 2W - WW]$$

We then compare each component of this vector with our initial prediction $\hat{X}$. If the value of the component is less than the prediction, we replace the value with a 1; otherwise

we replace it with a 0. Thus, we end up with an eight-component binary vector. If each component of the binary vector was independent, we would end up with 256 possible vectors. However, because of the dependence of various components, we actually have 144 possible configurations. We also compute a quantity that incorporates the vertical and horizontal variations and the previous error in prediction by

$$\delta = d_h + d_v + 2\,|N - \hat{N}| \tag{7.8}$$

where $\hat{N}$ is the predicted value of N. This range of values of δ is divided into four intervals, each being represented by 2 bits. These four possibilities, along with the 144 texture descriptors, create $144 \times 4 = 576$ contexts for X. As the encoding proceeds, we keep track of how much prediction error is generated in each context and offset our initial prediction by that amount. This results in the final predicted value.

Once the prediction is obtained, the difference between the pixel value and the prediction (the prediction error, or residual) has to be encoded. While the prediction process outlined above removes a lot of the structure that was in the original sequence, there is still some structure left in the residual sequence. We can take advantage of some of this structure by coding the residual in terms of its context. The context of the residual is taken to be the value of δ defined in Equation (7.8). In order to reduce the complexity of the encoding, rather than using the actual value as the context, CALIC uses the range of values in which δ lies as the context. Thus:

$$\begin{aligned}
0 \le \delta < q_1 &\Rightarrow \text{Context 1}\\
q_1 \le \delta < q_2 &\Rightarrow \text{Context 2}\\
q_2 \le \delta < q_3 &\Rightarrow \text{Context 3}\\
q_3 \le \delta < q_4 &\Rightarrow \text{Context 4}\\
q_4 \le \delta < q_5 &\Rightarrow \text{Context 5}\\
q_5 \le \delta < q_6 &\Rightarrow \text{Context 6}\\
q_6 \le \delta < q_7 &\Rightarrow \text{Context 7}\\
q_7 \le \delta < q_8 &\Rightarrow \text{Context 8}
\end{aligned}$$

The values of q_1–q_8 can be prescribed by the user.

If the original pixel values lie between 0 and $M-1$, the differences or prediction residuals will lie between $-(M-1)$ and $M-1$. Even though most of the differences will have a magnitude close to zero, for arithmetic coding we still have to assign a count to all possible symbols. This means a reduction in the size of the intervals assigned to values that do occur, which in turn means using a larger number of bits to represent these values. The CALIC algorithm attempts to resolve this problem in a number of ways. Let's describe these using an example.

Consider the sequence

$$x_n : 0,\ 7,\ 4,\ 3,\ 5,\ 2,\ 1,\ 7$$

We can see that all the numbers lie between 0 and 7, a range of values that would require 3 bits to represent. Now suppose we predict a sequence element by the previous element in the sequence. The sequence of differences

$$r_n = x_n - x_{n-1}$$

is given by

$$r_n : 0,\ 7,\ -3,\ -1,\ 2,\ -3,\ -1,\ 6$$

If we were given this sequence, we could easily recover the original sequence by using

$$x_n = x_{n-1} + r_n.$$

However, the prediction residual values r_n lie in the $[-7, 7]$ range. That is, the alphabet required to represent these values is almost twice the size of the original alphabet. However, if we look closely we can see that the value of r_n actually lies between $-x_{n-1}$ and $7 - x_{n-1}$. The smallest value that r_n can take on occurs when x_n has a value of 0, in which case r_n will have a value of $-x_{n-1}$. The largest value that r_n can take on occurs when x_n is 7, in which case r_n has a value of $7 - x_{n-1}$. In other words, given a particular value for x_{n-1}, the number of different values that r_n can take on is the same as the number of values that x_n can take on. Generalizing from this, we can see that if a pixel takes on values between 0 and $M-1$, then given a predicted value $\hat{X}$, the difference $X - \hat{X}$ will take on values in the range $-\hat{X}$ to $M - 1 - \hat{X}$. We can use this fact to map the difference values into the range $[0, M-1]$, using the following mapping:

$$\begin{aligned}
0 &\rightarrow 0 \\
1 &\rightarrow 1 \\
-1 &\rightarrow 2 \\
2 &\rightarrow 3 \\
\vdots \quad & \quad \vdots \\
-\hat{X} &\rightarrow 2\hat{X} \\
\hat{X}+1 &\rightarrow 2\hat{X}+1 \\
\hat{X}+2 &\rightarrow 2\hat{X}+2 \\
\vdots \quad & \quad \vdots \\
M-1-\hat{X} &\rightarrow M-1
\end{aligned}$$

where we have assumed that $\hat{X} \leq (M-1)/2$.

Another approach used by CALIC to reduce the size of its alphabet is to use a modification of a technique called *recursive indexing* [76]. Recursive indexing is a technique for representing a large range of numbers using only a small set. It is easiest to explain using an example. Suppose we want to represent positive integers using only the integers between 0 and 7—that is, a representation alphabet of size 8. Recursive indexing works as follows: If the number to be represented lies between 0 and 6, we simply represent it by that number. If the number to be represented is greater than or equal to 7, we first send the number 7, subtract 7 from the original number, and repeat the process. We keep repeating the process until the remainder is a number between 0 and 6. Thus, for example, 9 would be represented by 7 followed by a 2, and 17 would be represented by two 7s followed by a 3. The decoder, when it sees a number between 0 and 6, would decode it at its face value, and when it saw 7, would keep accumulating the values until a value between 0 and 6 was received. This method of representation followed by entropy coding has been shown to be optimal for sequences that follow a geometric distribution [77].

In CALIC, the representation alphabet is different for different coding contexts. For each coding context k, we use an alphabet $A_k = \{0, 1, \ldots, N_k\}$. Furthermore, if the residual occurs in context k, then the first number that is transmitted is coded with respect to context k; if further recursion is needed, we use the $k+1$ context.

We can summarize the CALIC algorithm as follows:

1. Find initial prediction $\hat{X}$.
2. Compute prediction context.
3. Refine prediction by removing the estimate of the bias in that context.
4. Update bias estimate.
5. Obtain the residual and remap it so the residual values lie between 0 and $M-1$, where M is the size of the initial alphabet.
6. Find the coding context k.
7. Code the residual using the coding context.

All these components working together have kept CALIC as the state of the art in lossless image compression. However, we can get almost as good a performance if we simplify some of the more involved aspects of CALIC. We study such a scheme in the next section.

7.4 JPEG-LS

The JPEG-LS standard looks more like CALIC than the old JPEG standard. When the initial proposals for the new lossless compression standard were compared, CALIC was rated first in six of the seven categories of images tested. Motivated by some aspects of CALIC, a team from Hewlett-Packard proposed a much simpler predictive coder, under the name LOCO-I (for low complexity), that still performed close to CALIC [78].

As in CALIC, the standard has both a lossless and a lossy mode. We will not describe the lossy coding procedures.

The initial prediction is obtained using the following algorithm:

```
if NW ≥ max(W, N)
X̂ = max(W, N)
else
{
    if NW ≤ min(W, N)
    X̂ = min(W, N)
    else
    X̂ = W + N − NW
}
```

This prediction approach is a variation of Median Adaptive Prediction [79], in which the predicted value is the median of the N, W, and NW pixels. The initial prediction is then refined using the average value of the prediction error in that particular context.

The contexts in JPEG-LS also reflect the local variations in pixel values. However, they are computed differently from CALIC. First, measures of differences D_1, D_2, and D_3 are computed as follows:

$$\begin{aligned} D_1 &= NE - N \\ D_2 &= N - NW \\ D_3 &= NW - W. \end{aligned}$$

The values of these differences define a three-component context vector $\mathbf{Q}$. The components of $\mathbf{Q}$ (Q_1, Q_2, and Q_3) are defined by the following mappings:

$$\begin{aligned} D_i \le -T_3 &\Rightarrow Q_i = -4 \\ -T_3 < D_i \le -T_2 &\Rightarrow Q_i = -3 \\ -T_2 < D_i \le -T_1 &\Rightarrow Q_i = -2 \\ -T_1 < D_i \le 0 &\Rightarrow Q_i = -1 \\ D_i = 0 &\Rightarrow Q_i = 0 \\ 0 < D_i \le T_1 &\Rightarrow Q_i = 1 \\ T_1 < D_i \le T_2 &\Rightarrow Q_i = 2 \\ T_2 < D_i \le T_3 &\Rightarrow Q_i = 3 \\ T_3 < D_i &\Rightarrow Q_i = 4 \end{aligned} \tag{7.9}$$

where T_1, T_2, and T_3 are positive coefficients that can be defined by the user. Given nine possible values for each component of the context vector, this results in $9 \times 9 \times 9 = 729$ possible contexts. In order to simplify the coding process, the number of contexts is reduced by replacing any context vector $\mathbf{Q}$ whose first nonzero element is negative by $-\mathbf{Q}$. Whenever

TABLE 7.3 Comparison of the file sizes obtained using new and old JPEG lossless compression standard and CALIC.

Image	Old JPEG	New JPEG	CALIC
Sena	31,055	27,339	26,433
Sensin	32,429	30,344	29,213
Earth	32,137	26,088	25,280
Omaha	48,818	50,765	48,249

this happens, a variable $SIGN$ is also set to -1; otherwise, it is set to $+1$. This reduces the number of contexts to 365. The vector $\mathbf{Q}$ is then mapped into a number between 0 and 364. (The standard does not specify the particular mapping to use.)

The variable $SIGN$ is used in the prediction refinement step. The correction is first multiplied by $SIGN$ and then added to the initial prediction.

The prediction error r_n is mapped into an interval that is the same size as the range occupied by the original pixel values. The mapping used in JPEG-LS is as follows:

$$r_n < -\frac{M}{2} \Rightarrow r_n \leftarrow r_n + M$$

$$r_n > \frac{M}{2} \Rightarrow r_n \leftarrow r_n - M$$

Finally, the prediction errors are encoded using adaptively selected codes based on Golomb codes, which have also been shown to be optimal for sequences with a geometric distribution. In Table 7.3 we compare the performance of the old and new JPEG standards and CALIC. The results for the new JPEG scheme were obtained using a software implementation courtesy of HP.

We can see that for most of the images the new JPEG standard performs very close to CALIC and outperforms the old standard by 6% to 18%. The only case where the performance is not as good is for the Omaha image. While the performance improvement in these examples may not be very impressive, we should keep in mind that for the old JPEG we are picking the best result out of eight. In practice, this would mean trying all eight JPEG predictors and picking the best. On the other hand, both CALIC and the new JPEG standard are single-pass algorithms. Furthermore, because of the ability of both CALIC and the new standard to function in multiple modes, both perform very well on compound documents, which may contain images along with text.

7.5 Multiresolution Approaches

Our final predictive image compression scheme is perhaps not as competitive as the other schemes. However, it is an interesting algorithm because it approaches the problem from a slightly different point of view.

Δ	$\bullet$	X	$\bullet$	Δ	$\bullet$	X	$\bullet$	Δ
$\bullet$	$*$	$\bullet$	$*$	$\bullet$	$*$	$\bullet$	$*$	$\bullet$
X	$\bullet$	$\circ$	$\bullet$	X	$\bullet$	$\circ$	$\bullet$	X
$\bullet$	$*$	$\bullet$	$*$	$\bullet$	$*$	$\bullet$	$*$	$\bullet$
Δ	$\bullet$	X	$\bullet$	Δ	$\bullet$	X	$\bullet$	Δ
$\bullet$	$*$	$\bullet$	$*$	$\bullet$	$*$	$\bullet$	$*$	$\bullet$
X	$\bullet$	$\circ$	$\bullet$	X	$\bullet$	$\circ$	$\bullet$	X
$\bullet$	$*$	$\bullet$	$*$	$\bullet$	$*$	$\bullet$	$*$	$\bullet$
Δ	$\bullet$	X	$\bullet$	Δ	$\bullet$	X	$\bullet$	Δ

FIGURE 7.2 The HINT scheme for hierarchical prediction.

Multiresolution models generate representations of an image with varying spatial resolution. This usually results in a pyramidlike representation of the image, with each layer of the pyramid serving as a prediction model for the layer immediately below.

One of the more popular of these techniques is known as HINT (Hierarchical INTerpolation) [80]. The specific steps involved in HINT are as follows. First, residuals corresponding to the pixels labeled Δ in Figure 7.2 are obtained using linear prediction and transmitted. Then, the intermediate pixels ($\circ$) are estimated by linear interpolation, and the error in estimation is then transmitted. Then, the pixels X are estimated from Δ and $\circ$, and the estimation error is transmitted. Finally, the pixels labeled $*$ and then $\bullet$ are estimated from known neighbors, and the errors are transmitted. The reconstruction process proceeds in a similar manner.

One use of a multiresolution approach is in progressive image transmission. We describe this application in the next section.

7.5.1 Progressive Image Transmission

The last few years have seen a very rapid increase in the amount of information stored as images, especially remotely sensed images (such as images from weather and other satellites) and medical images (such as CAT scans, magnetic resonance images, and mammograms). It is not enough to have information. We also need to make these images accessible to individuals who can make use of them. There are many issues involved with making large amounts of information accessible to a large number of people. In this section we will look at one particular issue—transmitting these images to remote users. (For a more general look at the problem of managing large amounts of information, see [81].)

Suppose a user wants to browse through a number of images in a remote database. The user is connected to the database via a 56 kbits per second (kbps) modem. Suppose the

images are of size 1024×1024, and on the average users have to look through 30 images before finding the image they are looking for. If these images were monochrome with 8 bits per pixel, this process would take close to an hour and 15 minutes, which is not very practical. Even if we compressed these images before transmission, lossless compression on average gives us about a two-to-one compression. This would only cut the transmission in half, which still makes the approach cumbersome. A better alternative is to send an approximation of each image first, which does not require too many bits but still is sufficiently accurate to give users an idea of what the image looks like. If users find the image to be of interest, they can request a further refinement of the approximation, or the complete image. This approach is called *progressive image transmission*.

Example 7.5.1:

A simple progressive transmission scheme is to divide the image into blocks and then send a representative pixel for the block. The receiver replaces each pixel in the block with the representative value. In this example, the representative value is the value of the pixel in the top-left corner. Depending on the size of the block, the amount of data that would need to be transmitted could be substantially reduced. For example, to transmit a 1024×1024 image at 8 bits per pixel over a 56 kbps line takes about two and a half minutes. Using a block size of 8×8, and using the top-left pixel in each block as the representative value, means we approximate the 1024×1024 image with a 128×128 subsampled image. Using 8 bits per pixel and a 56 kbps line, the time required to transmit this approximation to the image takes less than two and a half seconds. Assuming that this approximation was sufficient to let the user decide whether a particular image was the desired image, the time required now to look through 30 images becomes a minute and a half instead of the hour and a half mentioned earlier. If the approximation using a block size of 8×8 does not provide enough resolution to make a decision, the user can ask for a refinement. The transmitter can then divide the 8×8 block into four 4×4 blocks. The pixel at the upper-left corner of the upper-left block was already transmitted as the representative pixel for the 8×8 block, so we need to send three more pixels for the other three 4×4 blocks. This takes about seven seconds, so even if the user had to request a finer approximation every third image, this would only increase the total search time by a little more than a minute. To see what these approximations look like, we have taken the Sena image and encoded it using different block sizes. The results are shown in Figure 7.3. The lowest-resolution image, shown in the top left, is a 32×32 image. The top-left image is a 64×64 image. The bottom-left image is a 128×128 image, and the bottom-right image is the 256×256 original.

Notice that even with a block size of 8 the image is clearly recognizable as a person. Therefore, if the user was looking for a house, they would probably skip over this image after seeing the first approximation. If the user was looking for a picture of a person, they could still make decisions based on the second approximation.

Finally, when an image is built line by line, the eye tends to follow the scan line. With the progressive transmission approach, the user gets a more global view of the image very early in the image formation process. Consider the images in Figure 7.4. The images on the left are the 8×8, 4×4, and 2×2 approximations of the Sena image. On the right, we show

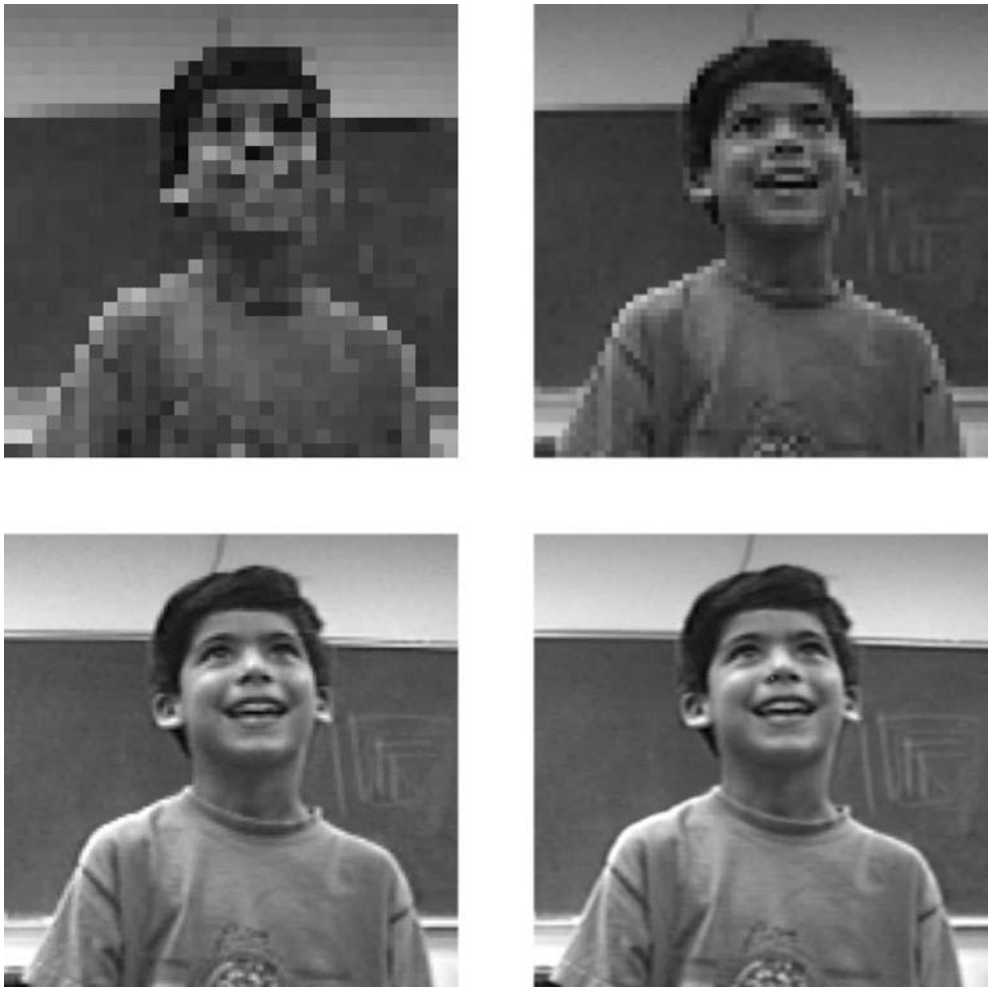

FIGURE 7. 3 **Sena image coded using different block sizes for progressive transmission. Top row: block size 8 × 8 and block size 4 × 4. Bottom row: block size 2 × 2 and original image.**

how much of the image we would see in the same amount of time if we used the standard line-by-line raster scan order. ♦

We would like the first approximations that we transmit to use as few bits as possible yet be accurate enough to allow the user to make a decision to accept or reject the image with a certain degree of confidence. As these approximations are lossy, many progressive transmission schemes use well-known lossy compression schemes in the first pass.

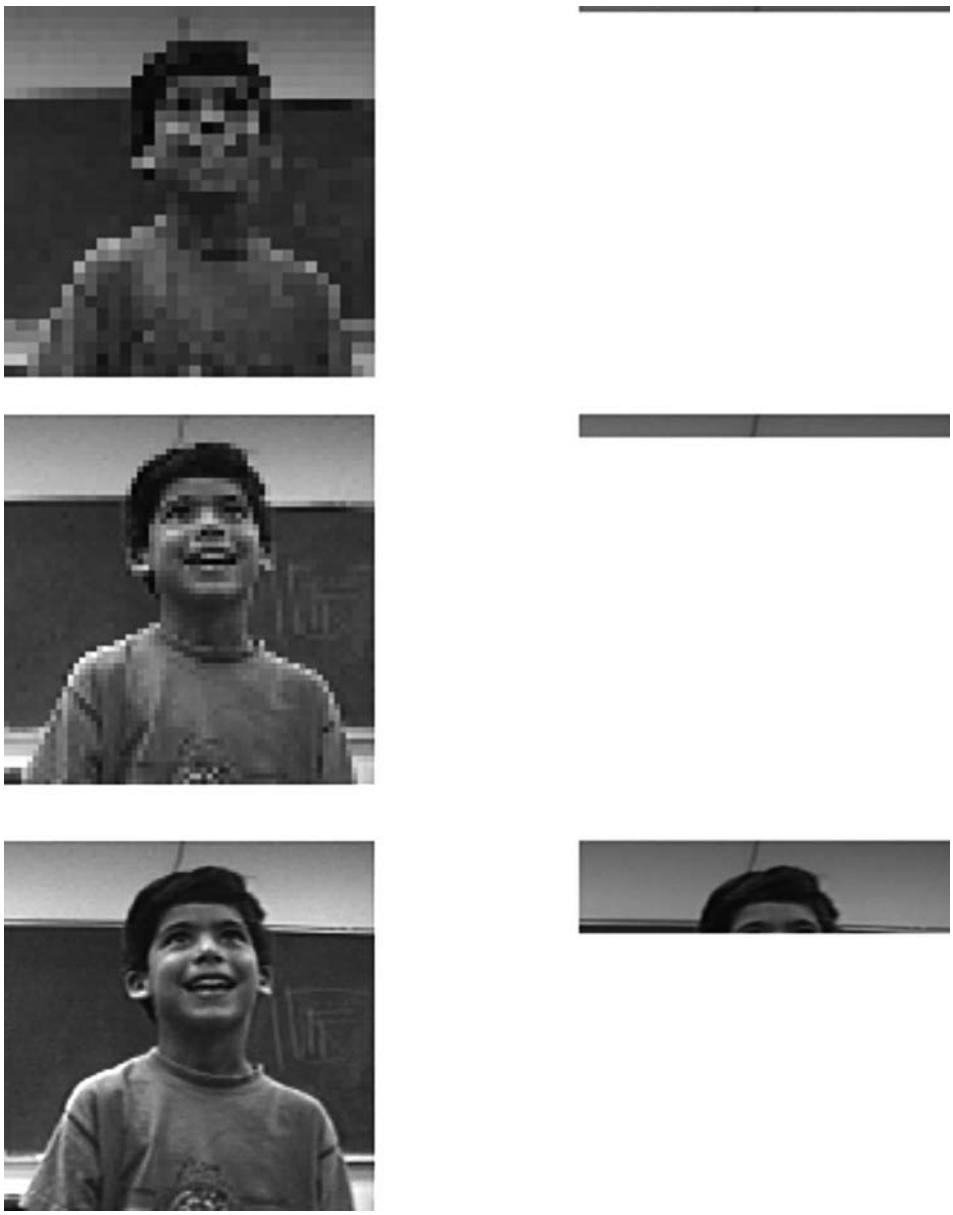

FIGURE 7. 4 **Comparison between the received image using progressive transmission and using the standard raster scan order.**

The more popular lossy compression schemes, such as transform coding, tend to require a significant amount of computation. As the decoders for most progressive transmission schemes have to function on a wide variety of platforms, they are generally implemented in software and need to be simple and fast. This requirement has led to the development of a number of progressive transmission schemes that do not use lossy compression schemes for their initial approximations. Most of these schemes have a form similar to the one described in Example 7.5.1, and they are generally referred to as *pyramid schemes* because of the manner in which the approximations are generated and the image is reconstructed.

When we use the pyramid form, we still have a number of ways to generate the approximations. One of the problems with the simple approach described in Example 7.5.1 is that if the pixel values vary a lot within a block, the "representative" value may not be very representative. To prevent this from happening, we could represent the block by some sort of an average or composite value. For example, suppose we start out with a 512×512 image. We first divide the image into 2×2 blocks and compute the integer value of the average of each block [82, 83]. The integer values of the averages would constitute the penultimate approximation. The approximation to be transmitted prior to that can be obtained by taking the average of 2×2 averages and so on, as shown in Figure 7.5.

Using the simple technique in Example 7.5.1, we ended up transmitting the same number of values as the original number of pixels. However, when we use the mean of the pixels as our approximation, after we have transmitted the mean values at each level, we still have to transmit the actual pixel values. The reason is that when we take the integer part of the average we end up throwing away information that cannot be retrieved. To avoid this problem of data expansion, we can transmit the sum of the values in the 2×2 block. Then we only need to transmit three more values to recover the original four values. With this approach, although we would be transmitting the same number of values as the number of pixels in the image, we might still end up sending more bits because representing all possible

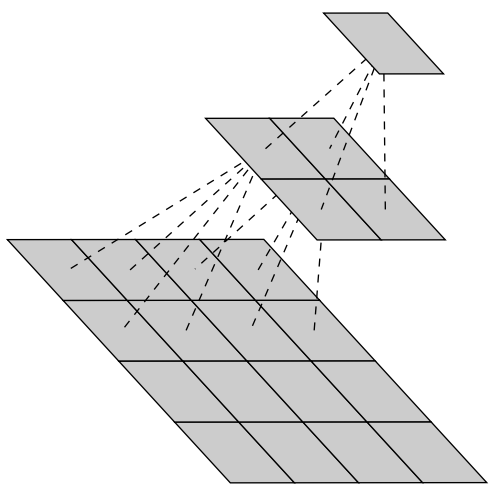

FIGURE 7.5 The pyramid structure for progressive transmission.

values of the sum would require transmitting 2 more bits than was required for the original value. For example, if the pixels in the image can take on values between 0 and 255, which can be represented by 8 bits, their sum will take on values between 0 and 1024, which would require 10 bits. If we are allowed to use entropy coding, we can remove the problem of data expansion by using the fact that the neighboring values in each approximation are heavily correlated, as are values in different levels of the pyramid. This means that differences between these values can be efficiently encoded using entropy coding. By doing so, we end up getting compression instead of expansion.

Instead of taking the arithmetic average, we could also form some sort of weighted average. The general procedure would be similar to that described above. (For one of the more well-known weighted average techniques, see [84].)

The representative value does not have to be an average. We could use the pixel values in the approximation at the lower levels of the pyramid as indices into a lookup table. The lookup table can be designed to preserve important information such as edges. The problem with this approach would be the size of the lookup table. If we were using 2×2 blocks of 8-bit values, the lookup table would have 2^{32} values, which is too large for most applications. The size of the table could be reduced if the number of bits per pixel was lower or if, instead of taking 2×2 blocks, we used rectangular blocks of size 2×1 and 1×2 [85].

Finally, we do not have to build the pyramid one layer at a time. After sending the lowest-resolution approximations, we can use some measure of information contained in a block to decide whether it should be transmitted [86]. One possible measure could be the difference between the largest and smallest intensity values in the block. Another might be to look at the maximum number of similar pixels in a block. Using an information measure to guide the progressive transmission of images allows the user to see portions of the image first that are visually more significant.

7.6 Facsimile Encoding

One of the earliest applications of lossless compression in the modern era has been the compression of facsimile, or fax. In facsimile transmission, a page is scanned and converted into a sequence of black or white pixels. The requirements of how fast the facsimile of an A4 document (210×297 mm) must be transmitted have changed over the last two decades. The CCITT (now ITU-T) has issued a number of recommendations based on the speed requirements at a given time. The CCITT classifies the apparatus for facsimile transmission into four groups. Although several considerations are used in this classification, if we only consider the time to transmit an A4-size document over phone lines, the four groups can be described as follows:

- **Group 1:** This apparatus is capable of transmitting an A4-size document in about six minutes over phone lines using an analog scheme. The apparatus is standardized in recommendation T.2.
- **Group 2:** This apparatus is capable of transmitting an A4-size document over phone lines in about three minutes. A Group 2 apparatus also uses an analog scheme and,

therefore, does not use data compression. The apparatus is standardized in recommendation T.3.

- **Group 3:** This apparatus uses a digitized binary representation of the facsimile. Because it is a digital scheme, it can and does use data compression and is capable of transmitting an A4-size document in about a minute. The apparatus is standardized in recommendation T.4.
- **Group 4:** This apparatus has the same speed requirement as Group 3. The apparatus is standardized in recommendations T.6, T.503, T.521, and T.563.

With the arrival of the Internet, facsimile transmission has changed as well. Given the wide range of rates and "apparatus" used for digital communication, it makes sense to focus more on protocols than on apparatus. The newer recommendations from the ITU provide standards for compression that are more or less independent of apparatus.

Later in this chapter, we will look at the compression schemes described in the ITU-T recommendations T.4, T.6, T.82 (JBIG) T.88 (JBIG2), and T.42 (MRC). We begin with a look at an earlier technique for facsimile called *run-length coding*, which still survives as part of the T.4 recommendation.

7.6.1 Run-Length Coding

The model that gives rise to run-length coding is the Capon model [87], a two-state Markov model with states S_w and S_b (S_w corresponds to the case where the pixel that has just been encoded is a white pixel, and S_b corresponds to the case where the pixel that has just been encoded is a black pixel). The transition probabilities $P(w|b)$ and $P(b|w)$, and the probability of being in each state $P(S_w)$ and $P(S_b)$, completely specify this model. For facsimile images, $P(w|w)$ and $P(w|b)$ are generally significantly higher than $P(b|w)$ and $P(b|b)$. The Markov model is represented by the state diagram shown in Figure 7.6.

The entropy of a finite state process with states S_i is given by Equation (2.16). Recall that in Example 2.3.1, the entropy using a probability model and the *iid* assumption was significantly more than the entropy using the Markov model.

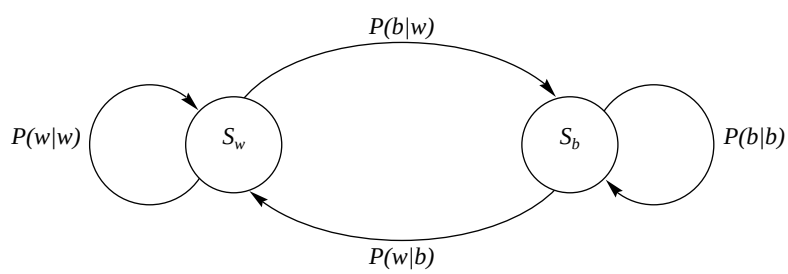

FIGURE 7.6 The Capon model for binary images.

Let us try to interpret what the model says about the structure of the data. The highly skewed nature of the probabilities $P(b|w)$ and $P(w|w)$, and to a lesser extent $P(w|b)$ and $P(b|b)$, says that once a pixel takes on a particular color (black or white), it is highly likely that the following pixels will also be of the same color. So, rather than code the color of each pixel separately, we can simply code the length of the runs of each color. For example, if we had 190 white pixels followed by 30 black pixels, followed by another 210 white pixels, instead of coding the 430 pixels individually, we would code the sequence 190, 30, 210, along with an indication of the color of the first string of pixels. Coding the lengths of runs instead of coding individual values is called run-length coding.

7.6.2 CCITT Group 3 and 4—Recommendations T.4 and T.6

The recommendations for Group 3 facsimile include two coding schemes. One is a one-dimensional scheme in which the coding on each line is performed independently of any other line. The other is two-dimensional; the coding of one line is performed using the line-to-line correlations.

The one-dimensional coding scheme is a run-length coding scheme in which each line is represented as a series of alternating white runs and black runs. The first run is always a white run. If the first pixel is a black pixel, then we assume that we have a white run of length zero.

Runs of different lengths occur with different probabilities; therefore, they are coded using a variable-length code. The approach taken in the CCITT standards T.4 and T.6 is to use a Huffman code to encode the run lengths. However, the number of possible lengths of runs is extremely large, and it is simply not feasible to build a codebook that large. Therefore, instead of generating a Huffman code for each run length r_l, the run length is expressed in the form

$$r_l = 64 \times m + t \qquad \text{for } t = 0, 1, \ldots, 63, \text{ and } m = 1, 2, \ldots, 27. \tag{7.10}$$

When we have to represent a run length r_l, instead of finding a code for r_l, we use the corresponding codes for m and t. The codes for t are called the *terminating codes*, and the codes for m are called the *make-up codes*. If $r_l < 63$, we only need to use a terminating code. Otherwise, both a make-up code and a terminating code are used. For the range of m and t given here, we can represent lengths of 1728, which is the number of pixels per line in an A4-size document. However, if the document is wider, the recommendations provide for those with an optional set of 13 codes. Except for the optional codes, there are separate codes for black and white run lengths. This coding scheme is generally referred to as a *modified Huffman (MH)* scheme.

In the two-dimensional scheme, instead of reporting the run lengths, which in terms of our Markov model is the length of time we remain in one state, we report the transition times when we move from one state to another state. Look at Figure 7.7. We can encode this in two ways. We can say that the first row consists of a sequence of runs 0, 2, 3, 3, 8, and the second row consists of runs of length 0, 1, 8, 3, 4 (notice the first runs of length zero). Or, we can encode the location of the pixel values that occur at a transition from white to

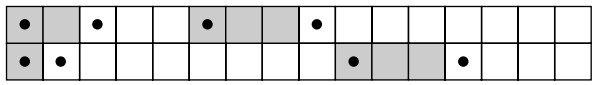

FIGURE 7.7 Two rows of an image. The transition pixels are marked with a dot.

black or black to white. The first pixel is an imaginary white pixel assumed to be to the left of the first actual pixel. Therefore, if we were to code transition locations, we would encode the first row as 1, 3, 6, 9 and the second row as 1, 2, 10, 13.

Generally, rows of a facsimile image are heavily correlated. Therefore, it would be easier to code the transition points with reference to the previous line than to code each one in terms of its absolute location, or even its distance from the previous transition point. This is the basic idea behind the recommended two-dimensional coding scheme. This scheme is a modification of a two-dimensional coding scheme called the *Relative Element Address Designate* (READ) code [88, 89] and is often referred to as *Modified READ* (MR). The READ code was the Japanese proposal to the CCITT for the Group 3 standard.

To understand the two-dimensional coding scheme, we need some definitions.

a_0: This is the last pixel whose value is known to both encoder and decoder. At the beginning of encoding each line, a_0 refers to an imaginary white pixel to the left of the first actual pixel. While it is often a transition pixel, it does not have to be.

a_1: This is the first transition pixel to the right of a_0. By definition its color should be the opposite of a_0. The location of this pixel is known only to the encoder.

a_2: This is the second transition pixel to the right of a_0. Its color should be the opposite of a_1, which means it has the same color as a_0. The location of this pixel is also known only to the encoder.

b_1: This is the first transition pixel on the line above the line currently being encoded to the right of a_0 whose color is the opposite of a_0. As the line above is known to both encoder and decoder, as is the value of a_0, the location of b_1 is also known to both encoder and decoder.

b_2: This is the first transition pixel to the right of b_1 in the line above the line currently being encoded.

For the pixels in Figure 7.7, if the second row is the one being currently encoded, and if we have encoded the pixels up to the second pixel, the assignment of the different pixels is shown in Figure 7.8. The pixel assignments for a slightly different arrangement of black and white pixels are shown in Figure 7.9.

If b_1 and b_2 lie between a_0 and a_1, we call the coding mode used the *pass mode*. The transmitter informs the receiver about the situation by sending the code 0001. Upon receipt of this code, the receiver knows that from the location of a_0 to the pixel right below b_2, all pixels are of the same color. If this had not been true, we would have encountered a transition pixel. As the first transition pixel to the right of a_0 is a_1, and as b_2 occurs before a_1, no transitions have occurred and all pixels from a_0 to right below b_2 are the same color. At this time, the last pixel known to both the transmitter and receiver is the pixel below b_2.

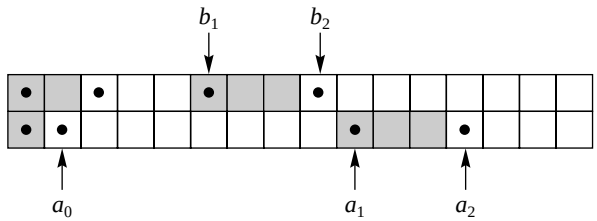

FIGURE 7. 8 **Two rows of an image. The transition pixels are marked with a dot.**

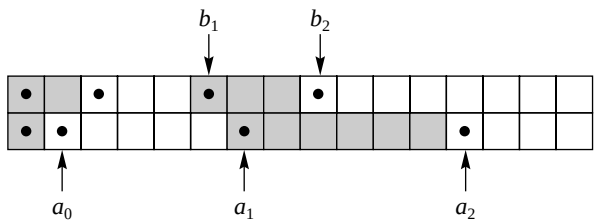

FIGURE 7. 9 **Two rows of an image. The transition pixels are marked with a dot.**

Therefore, this now becomes the new a_0, and we find the new positions of b_1 and b_2 by examining the row above the one being encoded and continue with the encoding process.

If a_1 is detected before b_2 by the encoder, we do one of two things. If the distance between a_1 and b_1 (the number of pixels from a_1 to right under b_1) is less than or equal to three, then we send the location of a_1 with respect to b_1, move a_0 to a_1, and continue with the coding process. This coding mode is called the *vertical mode*. If the distance between a_1 and b_1 is large, we essentially revert to the one-dimensional technique and send the distances between a_0 and a_1, and a_1 and a_2, using the modified Huffman code. Let us look at exactly how this is accomplished.

In the vertical mode, if the distance between a_1 and b_1 is zero (that is, a_1 is exactly under b_1), we send the code 1. If the a_1 is to the right of b_1 by one pixel (as in Figure 7.9), we send the code 011. If a_1 is to the right of b_1 by two or three pixels, we send the codes 000011 or 0000011, respectively. If a_1 is to the left of b_1 by one, two, or three pixels, we send the codes 010, 000010, or 0000010, respectively.

In the horizontal mode, we first send the code 001 to inform the receiver about the mode, and then send the modified Huffman codewords corresponding to the run length from a_0 to a_1, and a_1 to a_2.

As the encoding of a line in the two-dimensional algorithm is based on the previous line, an error in one line could conceivably propagate to all other lines in the transmission. To prevent this from happening, the T.4 recommendations contain the requirement that after each line is coded with the one-dimensional algorithm, at most $K - 1$ lines will be coded using the two-dimensional algorithm. For standard vertical resolution, $K = 2$, and for high resolution, $K = 4$.

The Group 4 encoding algorithm, as standardized in CCITT recommendation T.6, is identical to the two-dimensional encoding algorithm in recommendation T.4. The main difference between T.6 and T.4 from the compression point of view is that T.6 does not have a one-dimensional coding algorithm, which means that the restriction described in the previous paragraph is also not present. This slight modification of the modified READ algorithm has earned the name *modified modified READ* (MMR)!

7.6.3 JBIG

Many bi-level images have a lot of local structure. Consider a digitized page of text. In large portions of the image we will encounter white pixels with a probability approaching 1. In other parts of the image there will be a high probability of encountering a black pixel. We can make a reasonable guess of the situation for a particular pixel by looking at values of the pixels in the neighborhood of the pixel being encoded. For example, if the pixels in the neighborhood of the pixel being encoded are mostly white, then there is a high probability that the pixel to be encoded is also white. On the other hand, if most of the pixels in the neighborhood are black, there is a high probability that the pixel being encoded is also black. Each case gives us a skewed probability—a situation ideally suited for arithmetic coding. If we treat each case separately, using a different arithmetic coder for each of the two situations, we should be able to obtain improvement over the case where we use the same arithmetic coder for all pixels. Consider the following example.

Suppose the probability of encountering a black pixel is 0.2 and the probability of encountering a white pixel is 0.8. The entropy for this source is given by

$$H = -0.2\log_2 0.2 - 0.8\log_2 0.8 = 0.722. \tag{7.11}$$

If we use a single arithmetic coder to encode this source, we will get an average bit rate close to 0.722 bits per pixel. Now suppose, based on the neighborhood of the pixels, that we can divide the pixels into two sets, one comprising 80% of the pixels and the other 20%. In the first set, the probability of encountering a white pixel is 0.95, and in the second set the probability of encountering a black pixel is 0.7. The entropy of these sets is 0.286 and 0.881, respectively. If we used two different arithmetic coders for the two sets with frequency tables matched to the probabilities, we would get rates close to 0.286 bits per pixel about 80% of the time and close to 0.881 bits per pixel about 20% of the time. The average rate would be about 0.405 bits per pixel, which is almost half the rate required if we used a single arithmetic coder. If we use only those pixels in the neighborhood that had already been transmitted to the receiver to make our decision about which arithmetic coder to use, the decoder can keep track of which encoder was used to encode a particular pixel.

As we have mentioned before, the arithmetic coding approach is particularly amenable to the use of multiple coders. All coders use the same computational machinery, with each coder using a different set of probabilities. The JBIG algorithm makes full use of this feature of arithmetic coding. Instead of checking to see if most of the pixels in the neighborhood are white or black, the JBIG encoder uses the pattern of pixels in the neighborhood, or *context*, to decide which set of probabilities to use in encoding a particular pixel. If the neighborhood consists of 10 pixels, with each pixel capable of taking on two different values, the number of

	O	O	O	
O	O	O	O	A
O	O	X		

(a)

	O	O	O	O	O	A
O	O	O	O	X		

(b)

FIGURE 7.10 (a) Three-line and (b) two-line neighborhoods.

	0	0	0	
1	0	0	0	1
1	0	0		

(a)

	0	0	1	1	1	0
0	0	0	1	1		

(b)

FIGURE 7.11 (a) Three-line and (b) two-line contexts.

possible patterns is 1024. The JBIG coder uses 1024 to 4096 coders, depending on whether a low- or high-resolution layer is being encoded.

For the low-resolution layer, the JBIG encoder uses one of the two different neighborhoods shown in Figure 7.10. The pixel to be coded is marked **X**, while the pixels to be used for templates are marked **O** or **A**. The **A** and **O** pixels are previously encoded pixels and are available to both encoder and decoder. The **A** pixel can be thought of as a floating member of the neighborhood. Its placement is dependent on the input being encoded. Suppose the image has vertical lines 30 pixels apart. The **A** pixel would be placed 30 pixels to the left of the pixel being encoded. The **A** pixel can be moved around to capture any structure that might exist in the image. This is especially useful in halftone images in which the **A** pixels are used to capture the periodic structure. The location and movement of the **A** pixel are transmitted to the decoder as side information.

In Figure 7.11, the symbols in the neighborhoods have been replaced by 0s and 1s. We take 0 to correspond to white pixels, while 1 corresponds to black pixels. The pixel to be encoded is enclosed by the heavy box. The pattern of 0s and 1s is interpreted as a binary number, which is used as an index to the set of probabilities. The context in the case of the three-line neighborhood (reading left to right, top to bottom) is 0001000110, which corresponds to an index of 70. For the two-line neighborhood, the context is 0011100001, or 225. Since there are 10 bits in these templates, we will have 1024 different arithmetic coders.

In the JBIG standard, the 1024 arithmetic coders are a variation of the arithmetic coder known as the QM coder. The QM coder is a modification of an adaptive binary arithmetic coder called the Q coder [51, 52, 53], which in turn is an extension of another binary adaptive arithmetic coder called the skew coder [90].

In our description of arithmetic coding, we updated the tag interval by updating the endpoints of the interval, $u^{(n)}$ and $l^{(n)}$. We could just as well have kept track of one endpoint

and the size of the interval. This is the approach adopted in the QM coder, which tracks the lower end of the tag interval $l^{(n)}$ and the size of the interval $A^{(n)}$, where

$$A^{(n)} = u^{(n)} - l^{(n)}. \tag{7.12}$$

The tag for a sequence is the binary representation of $l^{(n)}$.

We can obtain the update equation for $A^{(n)}$ by subtracting Equation (4.9) from Equation (4.10) and making this substitution

$$A^{(n)} = A^{(n-1)}(F_X(x_n) - F_X(x_n - 1) \tag{7.13}$$

$$= A^{(n-1)}P(x_n). \tag{7.14}$$

Substituting $A^{(n)}$ for $u^{(n)} - l^{(n)}$ in Equation (4.9), we get the update equation for $l^{(n)}$:

$$l^{(n)} = l^{(n-1)} + A^{(n-1)}F_X(x_n - 1). \tag{7.15}$$

Instead of dealing directly with the 0s and 1s put out by the source, the QM coder maps them into a More Probable Symbol (MPS) and Less Probable Symbol (LPS). If 0 represents black pixels and 1 represents white pixels, then in a mostly black image 0 will be the MPS, whereas in an image with mostly white regions 1 will be the MPS. Denoting the probability of occurrence of the LPS for the context C by q_c and mapping the MPS to the lower subinterval, the occurrence of an MPS symbol results in the following update equations:

$$l^{(n)} = l^{(n-1)} \tag{7.16}$$

$$A^{(n)} = A^{(n-1)}(1 - q_c) \tag{7.17}$$

while the occurrence of an LPS symbol results in the following update equations:

$$l^{(n)} = l^{(n-1)} + A^{(n-1)}(1 - q_c) \tag{7.18}$$

$$A^{(n)} = A^{(n-1)}q_c. \tag{7.19}$$

Until this point, the QM coder looks very much like the arithmetic coder described earlier in this chapter. To make the implementation simpler, the JBIG committee recommended several deviations from the standard arithmetic coding algorithm. The update equations involve multiplications, which are expensive in both hardware and software. In the QM coder, the multiplications are avoided by assuming that $A^{(n)}$ has a value close to 1, and multiplication with $A^{(n)}$ can be approximated by multiplication with 1. Therefore, the update equations become

For MPS:

$$l^{(n)} = l^{(n-1)} \tag{7.20}$$

$$A^{(n)} = 1 - q_c \tag{7.21}$$

For LPS:

$$l^{(n)} = l^{(n-1)} + (1 - q_c) \tag{7.22}$$

$$A^{(n)} = q_c \tag{7.23}$$

In order not to violate the assumption on $A^{(n)}$ whenever the value of $A^{(n)}$ drops below 0.75, the QM coder goes through a series of rescalings until the value of $A^{(n)}$ is greater than or equal to 0.75. The rescalings take the form of repeated doubling, which corresponds to a left shift in the binary representation of $A^{(n)}$. To keep all parameters in sync, the same scaling is also applied to $l^{(n)}$. The bits shifted out of the buffer containing the value of $l^{(n)}$ make up the encoder output. Looking at the update equations for the QM coder, we can see that a rescaling will occur every time an LPS occurs. Occurrence of an MPS may or may not result in a rescale, depending on the value of $A^{(n)}$.

The probability q_c of the LPS for context C is updated each time a rescaling takes place and the context C is active. An ordered list of values for q_c is listed in a table. Every time a rescaling occurs, the value of q_c is changed to the next lower or next higher value in the table, depending on whether the rescaling was caused by the occurrence of an LPS or an MPS.

In a nonstationary situation, the symbol assigned to LPS may actually occurs more often than the symbol assigned to MPS. This condition is detected when $q_c > (A^{(n)} - q_c)$. In this situation, the assignments are reversed; the symbol assigned the LPS label is assigned the MPS label and vice versa. The test is conducted every time a rescaling takes place.

The decoder for the QM coder operates in much the same way as the decoder described in this chapter, mimicking the encoder operation.

Progressive Transmission

In some applications we may not always need to view an image at full resolution. For example, if we are looking at the layout of a page, we may not need to know what each word or letter on the page is. The JBIG standard allows for the generation of progressively lower-resolution images. If the user is interested in some gross patterns in the image (for example, if they were interested in seeing if there were any figures on a particular page) they could request a lower-resolution image, which could be transmitted using fewer bits. Once the lower-resolution image was available, the user could decide whether a higher-resolution image was necessary. The JBIG specification recommends generating one lower-resolution pixel for each 2×2 block in the higher-resolution image. The number of lower-resolution images (called layers) is not specified by JBIG.

A straightforward method for generating lower-resolution images is to replace every 2×2 block of pixels with the average value of the four pixels, thus reducing the resolution by two in both the horizontal and vertical directions. This approach works well as long as three of the four pixels are either black or white. However, when we have two pixels of each kind, we run into trouble; consistently replacing the four pixels with either a white or black pixel causes a severe loss of detail, and randomly replacing with a black or white pixel introduces a considerable amount of noise into the image [81].

Instead of simply taking the average of every 2×2 block, the JBIG specification provides a table-based method for resolution reduction. The table is indexed by the neighboring pixels shown in Figure 7.12, in which the circles represent the lower-resolution layer pixels and the squares represent the higher-resolution layer pixels.

Each pixel contributes a bit to the index. The table is formed by computing the expression

$$4e + 2(b + d + f + h) + (a + c + g + i) - 3(B + C) - A.$$

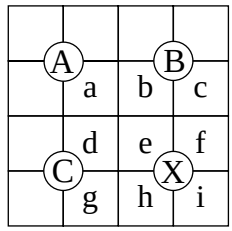

FIGURE 7.12 Pixels used to determine the value of a lower-level pixel.

If the value of this expression is greater than 4.5, the pixel X is tentatively declared to be 1. The table has certain exceptions to this rule to reduce the amount of edge smearing, generally encountered in a filtering operation. There are also exceptions that preserve periodic patterns and dither patterns.

As the lower-resolution layers are obtained from the higher-resolution images, we can use them when encoding the higher-resolution images. The JBIG specification makes use of the lower-resolution images when encoding the higher-resolution images by using the pixels of the lower-resolution images as part of the context for encoding the higher-resolution images. The contexts used for coding the lowest-resolution layer are those shown in Figure 7.10. The contexts used in coding the higher-resolution layer are shown in Figure 7.13.

Ten pixels are used in each context. If we include the 2 bits required to indicate which context template is being used, 12 bits will be used to indicate the context. This means that we can have 4096 different contexts.

Comparison of MH, MR, MMR, and JBIG

In this section we have seen three old facsimile coding algorithms: modified Huffman, modified READ, and modified modified READ. Before we proceed to the more modern techniques found in T.88 and T.42, we compare the performance of these algorithms with the earliest of the modern techniques, namely JBIG. We described the JBIG algorithm as an application of arithmetic coding in Chapter 4. This algorithm has been standardized in ITU-T recommendation T.82. As we might expect, the JBIG algorithm performs better than the MMR algorithm, which performs better than the MR algorithm, which in turn performs better than the MH algorithm. The level of complexity also follows the same trend, although we could argue that MMR is actually less complex than MR.

A comparison of the schemes for some facsimile sources is shown in Table 7.4. The modified READ algorithm was used with $K = 4$, while the JBIG algorithm was used with an adaptive three-line template and adaptive arithmetic coder to obtain the results in this table. As we go from the one-dimensional MH coder to the two-dimensional MMR coder, we get a factor of two reduction in file size for the sparse text sources. We get even more reduction when we use an adaptive coder and an adaptive model, as is true for the JBIG coder. When we come to the dense text, the advantage of the two-dimensional MMR over the one-dimensional MH is not as significant, as the amount of two-dimensional correlation becomes substantially less.

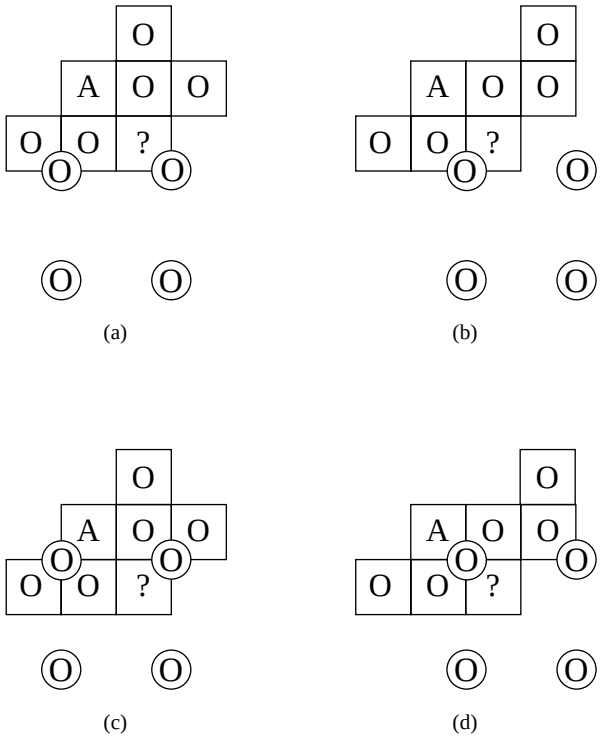

FIGURE 7.13 Contexts used in the coding of higher-resolution layers.

TABLE 7.4 Comparison of binary image coding schemes. Data from [91].

Source Description	Original Size (pixels)	MH (bytes)	MR (bytes)	MMR (bytes)	JBIG (bytes)
Letter	4352×3072	20,605	14,290	8,531	6,682
Sparse text	4352×3072	26,155	16,676	9,956	7,696
Dense text	4352×3072	135,705	105,684	92,100	70,703

The compression schemes specified in T.4 and T.6 break down when we try to use them to encode halftone images. In halftone images, gray levels are represented using binary pixel patterns. A gray level closer to black would be represented by a pattern that contains more black pixels, while a gray level closer to white would be represented by a pattern with fewer black pixels. Thus, the model that was used to develop the compression schemes specified in T.4 and T.6 is not valid for halftone images. The JBIG algorithm, with its adaptive model and coder, suffers from no such drawbacks and performs well for halftone images also [91].

7.6.4 JBIG2—T.88

The JBIG2 standard was approved in February of 2000. Besides facsimile transmission, the standard is also intended for document storage, archiving, wireless transmission, print spooling, and coding of images on the Web. The standard provides specifications only for the decoder, leaving the encoder design open. This means that the encoder design can be constantly refined, subject only to compatibility with the decoder specifications. This situation also allows for lossy compression, beacuse the encoder can incorporate lossy transformations to the data that enhance the level of compression.

The compression algorithm in JBIG provides excellent compression of a generic bi-level image. The compression algorithm proposed for JBIG2 uses the same arithmetic coding scheme as JBIG. However, it takes advantage of the fact that a significant number of bi-level images contain structure that can be used to enhance the compression performance. A large percentage of bi-level images consist of text on some background, while another significant percentage of bi-level images are or contain halftone images. The JBIG2 approach allows the encoder to select the compression technique that would provide the best performance for the type of data. To do so, the encoder divides the page to be compressed into three types of regions called *symbol regions*, *halftone regions*, and *generic regions*. The symbol regions are those containing text data, the halftone regions are those containing halftone images, and the generic regions are all the regions that do not fit into either category.

The partitioning information has to be supplied to the decoder. The decoder requires that all information provided to it be organized in *segments* that are made up of a segment header, a data header, and segment data. The page information segment contains information about the page including the size and resolution. The decoder uses this information to set up the page buffer. It then decodes the various regions using the appropriate decoding procedure and places the different regions in the appropriate location.

Generic Decoding Procedures

There are two procedures used for decoding the generic regions: the generic region decoding procedure and the generic refinement region decoding procedure. The generic region decoding procedure uses either the MMR technique used in the Group 3 and Group 4 fax standards or a variation of the technique used to encode the lowest-resolution layer in the JBIG recommendation. We describe the operation of the MMR algorithm in Chapter 6. The latter procedure is described as follows.

The second generic region decoding procedure is a procedure called *typical prediction*. In a bi-level image, a line of pixels is often identical to the line above. In typical prediction, if the current line is the same as the line above, a bit flag called $LNTP_n$ is set to 0, and the line is not transmitted. If the line is not the same, the flag is set to 1, and the line is coded using the contexts currently used for the low-resolution layer in JBIG. The value of $LNTP_n$ is encoded by generating another bit, $SLNTP_n$, according to the rule

$$SLNTP_n = !(LNTP_n \oplus LNTP_{n-1})$$

which is treated as a virtual pixel to the left of each row. If the decoder decodes an *LNTP* value of 0, it copies the line above. If it decodes an *LNTP* value of 1, the following bits

in the segment data are decoded using an arithmetic decoder and the contexts described previously.

The generic refinement decoding procedure assumes the existence of a *reference* layer and decodes the segment data with reference to this layer. The standard leaves open the specification of the reference layer.

Symbol Region Decoding

The symbol region decoding procedure is a dictionary-based decoding procedure. The symbol region segment is decoded with the help of a symbol dictionary contained in the symbol dictionary segment. The data in the symbol region segment contains the location where a symbol is to be placed, as well as the index to an entry in the symbol dictionary. The symbol dictionary consists of a set of bitmaps and is decoded using the generic decoding procedures. Note that because JBIG2 allows for lossy compression, the symbols do not have to exactly match the symbols in the original document. This feature can significantly increase the compression performance when the original document contains noise that may preclude exact matches with the symbols in the dictionary.

Halftone Region Decoding

The halftone region decoding procedure is also a dictionary-based decoding procedure. The halftone region segment is decoded with the help of a halftone dictionary contained in the halftone dictionary segment. The halftone dictionary segment is decoded using the generic decoding procedures. The data in the halftone region segment consists of the location of the halftone region and indices to the halftone dictionary. The dictionary is a set of fixed-size halftone patterns. As in the case of the symbol region, if lossy compression is allowed, the halftone patterns do not have to exactly match the patterns in the original document. By allowing for nonexact matches, the dictionary can be kept small, resulting in higher compression.

7.7 MRC—T.44

With the rapid advance of technology for document production, documents have changed in appearance. Where a document used to be a set of black and white printed pages, now documents contain multicolored text as well as color images. To deal with this new type of document, the ITU-T developed the recommendation T.44 for Mixed Raster Content (MRC). This recommendation takes the approach of separating the document into elements that can be compressed using available techniques. Thus, it is more an approach of partitioning a document image than a compression technique. The compression strategies employed here are borrowed from previous standards such as JPEG (T.81), JBIG (T.82), and even T.6.

The T.44 recommendation divides a page into slices where the width of the slice is equal to the width of the entire page. The height of the slice is variable. In the base mode, each

FIGURE 7. 14 **Ruby's birthday invitation.**

FIGURE 7. 15 **The background layer.**

slice is represented by three layers: a background layer, a foreground layer, and a mask layer. These layers are used to effectively represent three basic data types: color images (which may be continuous tone or color mapped), bi-level data, and multilevel (multicolor) data. The multilevel image data is put in the background layer, and the mask and foreground layers are used to represent the bi-level and multilevel nonimage data. To work through the various definitions, let us use the document shown in Figure 7.14 as an example. We have divided the document into two slices. The top slice contains the picture of the cake and two lines of writing in two "colors." Notice that the heights of the two slices are not the same and the complexity of the information contained in the two slices is not the same. The top slice contains multicolored text and a continuous tone image whereas the bottom slice contains only bi-level text. Let us take the upper slice first and see how to divide it into the three layers. We will discuss how to code these layers later. The background layer consists of the cake and nothing else. The default color for the background layer is white (though this can be changed). Therefore, we do not need to send the left half of this layer, which contains only white pixels.

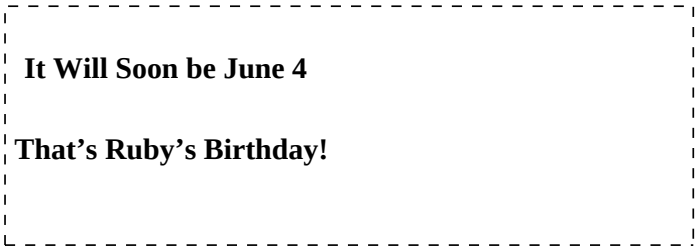

FIGURE 7. 16 The mask layer.

FIGURE 7. 17 The foreground layer.

The mask layer (Figure 7.16) consists of a bi-level representation of the textual information, while the foreground layer contains the colors used in the text. To reassemble the slice we begin with the background layer. We then add to it pixels from the foreground layer using the mask layer as the guide. Wherever the mask layer pixel is black (1) we pick the corresponding pixel from the foreground layer. Wherever the mask pixel is white (0) we use the pixel from the background layer. Because of its role in selecting pixels, the mask layer is also known as the selector layer. During transmission the mask layer is transmitted first, followed by the background and the foreground layers. During the rendering process the background layer is rendered first.

When we look at the lower slice we notice that it contains only bi-level information. In this case we only need the mask layer because the other two layers would be superfluous. In order to deal with this kind of situation, the standard defines three different kinds of stripes. Three-layer stripes (3LS) contain all three layers and is useful when there is both image and textual data in the strip. Two-layer stripes (2LS) only contain two layers, with the third set to a constant value. This kind of stripe would be useful when encoding a stripe with multicolored text and no images, or a stripe with images and bi-level text or line drawings. The third kind of stripe is a one-layer stripe (1LS) which would be used when a stripe contains only bi-level text or line art, or only continuous tone images.

Once the document has been partitioned it can be compressed. Notice that the types of data we have after partitioning are continuous tone images, bi-level information, and multilevel regions. We already have efficient standards for compressing these types of data. For the mask layer containing bi-level information, the recommendation suggests that one of several approaches can be used, including modified Huffman or modified READ

(as described in recomendation T.4), MMR (as described in recommendation T.6) or JBIG (recommendation T.82). The encoder includes information in the datastream about which algorithm has been used. For the continuous tone images and the multilevel regions contained in the foreground and background layers, the recommendation suggests the use of the JPEG standard (recommendation T.81) or the JBIG standard. The header for each slice contains information about which algorithm is used for compression.

7.8 Summary

In this section we have examined a number of ways to compress images. All these approaches exploit the fact that pixels in an image are generally highly correlated with their neighbors. This correlation can be used to predict the actual value of the current pixel. The prediction error can then be encoded and transmitted. Where the correlation is especially high, as in the case of bi-level images, long stretches of pixels can be encoded together using their similarity with previous rows. Finally, by identifying different components of an image that have common characteristics, an image can be partitioned and each partition encoded using the algorithm best suited to it.

Further Reading

1. A detailed survey of lossless image compression techniques can be found in "Lossless Image Compression" by K.P. Subbalakshmi. This chapter appears in the *Lossless Compression Handbook, Academic Press*, 2003.
2. For a detailed description of the LOCO-I and JPEG-LS compression algorithm, see "The LOCO-I Lossless Image Compression Algorithm: Principles and Standardization into JPEG-LS," Hewlett-Packard Laboratories Technical Report HPL-98-193, November 1998 [92].
3. The JBIG and JBIG2 standards are described in a very accessible manner in "Lossless Bilevel Image Compression," by M.W. Hoffman. This chapter appears in the *Lossless Compression Handbook, Academic Press*, 2003.
4. The area of lossless image compression is a very active one, and new schemes are being published all the time. These articles appear in a number of journals, including *Journal of Electronic Imaging*, *Optical Engineering*, *IEEE Transactions on Image Processing*, *IEEE Transactions on Communications*, *Communications of the ACM*, *IEEE Transactions on Computers*, and *Image Communication*, among others.

7.9 Projects and Problems

1. Encode the binary image shown in Figure 7.18 using the modified Huffman scheme.
2. Encode the binary image shown in Figure 7.18 using the modified READ scheme.
3. Encode the binary image shown in Figure 7.18 using the modified modified READ scheme.

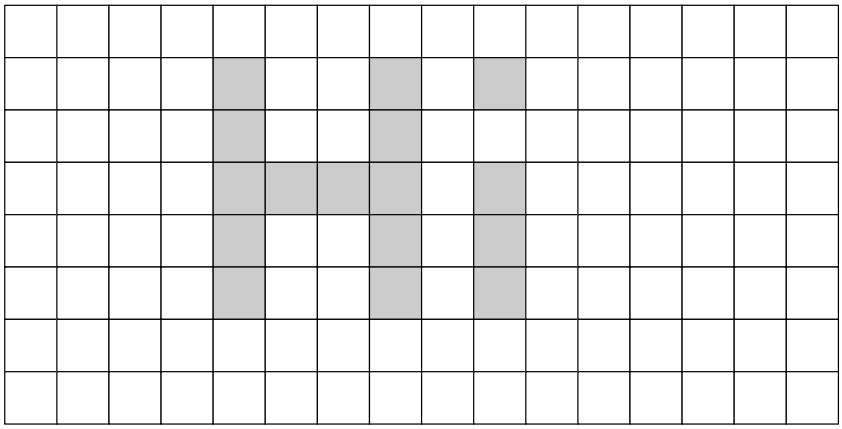

FIGURE 7. 18 **An 8 × 16 binary image.**

4. Suppose we want to transmit a 512 × 512, 8-bits-per-pixel image over a 9600 bits per second line.

(a) If we were to transmit this image using raster scan order, after 15 seconds how many rows of the image will the user have received? To what fraction of the image does this correspond?

(b) If we were to transmit the image using the method of Example 7.5.1, how long would it take the user to receive the first approximation? How long would it take to receive the first two approximations?

5. An implementation of the progressive transmission example (Example 7.5.1) is included in the programs accompanying this book. The program is called `prog_tran1.c`. Using this program as a template, experiment with different ways of generating approximations (you could use various types of weighted averages) and comment on the qualitative differences (or lack thereof) with using various schemes. Try different block sizes and comment on the practical effects in terms of quality and rate.

6. The program `jpegll_enc.c` generates the residual image for the different JPEG prediction modes, while the program `jpegll_dec.c` reconstructs the original image from the residual image. The output of the encoder program can be used as the input to the public domain arithmetic coding program mentioned in Chapter 4 and the Huffman coding programs mentioned in Chapter 3. Study the performance of different combinations of prediction mode and entropy coder using three images of your choice. Account for any differences you see.

7. Extend `jpegll_enc.c` and `jpegll_dec.c` with an additional prediction mode—be creative! Compare the performance of your predictor with the JPEG predictors.

8. Implement the portions of the CALIC algorithm described in this chapter. Encode the Sena image using your implementation.

8

Mathematical Preliminaries for Lossy Coding

8.1 Overview

Before we discussed lossless compression, we presented some of the mathematical background necessary for understanding and appreciating the compression schemes that followed. We will try to do the same here for lossy compression schemes. In lossless compression schemes, rate is the general concern. With lossy compression schemes, the loss of information associated with such schemes is also a concern. We will look at different ways of assessing the impact of the loss of information. We will also briefly revisit the subject of information theory, mainly to get an understanding of the part of the theory that deals with the trade-offs involved in reducing the rate, or number of bits per sample, at the expense of the introduction of distortion in the decoded information. This aspect of information theory is also known as rate distortion theory. We will also look at some of the models used in the development of lossy compression schemes.

8.2 Introduction

This chapter will provide some mathematical background that is necessary for discussing lossy compression techniques. Most of the material covered in this chapter is common to many of the compression techniques described in the later chapters. Material that is specific to a particular technique is described in the chapter in which the technique is presented. Some of the material presented in this chapter is not essential for understanding the techniques described in this book. However, to follow some of the literature in this area, familiarity with these topics is necessary. We have marked these sections with a $\star$. If you are primarily interested in the techniques, you may wish to skip these sections, at least on first reading.

On the other hand, if you wish to delve more deeply into these topics, we have included a list of resources at the end of this chapter that provide a more mathematically rigorous treatment of this material.

When we were looking at lossless compression, one thing we never had to worry about was how the reconstructed sequence would differ from the original sequence. By definition, the reconstruction of a losslessly constructed sequence is identical to the original sequence. However, there is only a limited amount of compression that can be obtained with lossless compression. There is a floor (a hard one) defined by the entropy of the source, below which we cannot drive the size of the compressed sequence. As long as we wish to preserve all of the information in the source, the entropy, like the speed of light, is a fundamental limit.

The limited amount of compression available from using lossless compression schemes may be acceptable in several circumstances. The storage or transmission resources available to us may be sufficient to handle our data requirements after lossless compression. Or the possible consequences of a loss of information may be much more expensive than the cost of additional storage and/or transmission resources. This would be the case with the storage and archiving of bank records; an error in the records could turn out to be much more expensive than the cost of buying additional storage media.

If neither of these conditions hold—that is, resources are limited and we do not require absolute integrity—we can improve the amount of compression by accepting a certain degree of loss during the compression process. Performance measures are necessary to determine the efficiency of our *lossy* compression schemes. For the lossless compression schemes we essentially used only the rate as the performance measure. That would not be feasible for lossy compression. If rate were the only criterion for lossy compression schemes, where loss of information is permitted, the best lossy compression scheme would be simply to throw away all the data! Therefore, we need some additional performance measure, such as some measure of the difference between the original and reconstructed data, which we will refer to as the *distortion* in the reconstructed data. In the next section, we will look at some of the more well-known measures of difference and discuss their advantages and shortcomings.

In the best of all possible worlds we would like to incur the minimum amount of distortion while compressing to the lowest rate possible. Obviously, there is a trade-off between minimizing the rate and keeping the distortion small. The extreme cases are when we transmit no information, in which case the rate is zero, or keep all the information, in which case the distortion is zero. The rate for a discrete source is simply the entropy. The study of the situations between these two extremes is called *rate distortion theory*. In this chapter we will take a brief look at some important concepts related to this theory.

Finally, we need to expand the dictionary of models available for our use, for several reasons. First, because we are now able to introduce distortion, we need to determine how to add distortion intelligently. For this, we often need to look at the sources somewhat differently than we have done previously. Another reason is that we will be looking at compression schemes for sources that are analog in nature, even though we have treated them as discrete sources in the past. We need models that more precisely describe the true nature of these sources. We will describe several different models that are widely used in the development of lossy compression algorithms.

We will use the block diagram and notation used in Figure 8.1 throughout our discussions. The output of the source is modeled as a random variable X. The *source coder*

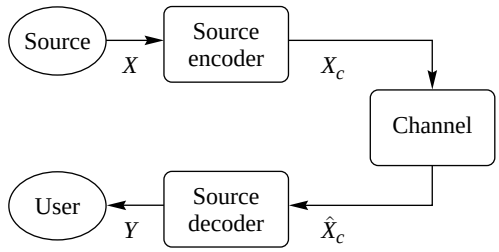

FIGURE 8. 1 Block diagram of a generic compression scheme.

takes the source output and produces the compressed representation X_c. The channel block represents all transformations the compressed representation undergoes before the source is reconstructed. Usually, we will take the channel to be the identity mapping, which means $X_c = \hat{X}_c$. The source decoder takes the compressed representation and produces a reconstruction of the source output for the user.

8.3 Distortion Criteria

How do we measure the closeness or fidelity of a reconstructed source sequence to the original? The answer frequently depends on what is being compressed and who is doing the answering. Suppose we were to compress and then reconstruct an image. If the image is a work of art and the resulting reconstruction is to be part of a book on art, the best way to find out how much distortion was introduced and in what manner is to ask a person familiar with the work to look at the image and provide an opinion. If the image is that of a house and is to be used in an advertisement, the best way to evaluate the quality of the reconstruction is probably to ask a real estate agent. However, if the image is from a satellite and is to be processed by a machine to obtain information about the objects in the image, the best measure of fidelity is to see how the introduced distortion affects the functioning of the machine. Similarly, if we were to compress and then reconstruct an audio segment, the judgment of how close the reconstructed sequence is to the original depends on the type of material being examined as well as the manner in which the judging is done. An audiophile is much more likely to perceive distortion in the reconstructed sequence, and distortion is much more likely to be noticed in a musical piece than in a politician's speech.

In the best of all worlds we would always use the end user of a particular source output to assess quality and provide the feedback required for the design. In practice this is not often possible, especially when the end user is a human, because it is difficult to incorporate the human response into mathematical design procedures. Also, there is difficulty in objectively reporting the results. The people asked to assess one person's design may be more easygoing than the people who were asked to assess another person's design. Even though the reconstructed output using one person's design is rated "excellent" and the reconstructed output using the other person's design is only rated "acceptable," switching observers may change the ratings. We could reduce this kind of bias by recruiting a large

number of observers in the hope that the various biases will cancel each other out. This is often the option used, especially in the final stages of the design of compression systems. However, the rather cumbersome nature of this process is limiting. We generally need a more practical method for looking at how close the reconstructed signal is to the original.

A natural thing to do when looking at the fidelity of a reconstructed sequence is to look at the differences between the original and reconstructed values—in other words, the distortion introduced in the compression process. Two popular measures of distortion or difference between the original and reconstructed sequences are the squared error measure and the absolute difference measure. These are called *difference distortion measures*. If $\{x_n\}$ is the source output and $\{y_n\}$ is the reconstructed sequence, then the squared error measure is given by

$$d(x, y) = (x - y)^2 \tag{8.1}$$

and the absolute difference measure is given by

$$d(x, y) = |x - y| . \tag{8.2}$$

In general, it is difficult to examine the difference on a term-by-term basis. Therefore, a number of average measures are used to summarize the information in the difference sequence. The most often used average measure is the average of the squared error measure. This is called the *mean squared error* (mse) and is often represented by the symbol σ^2 or σ_d^2:

$$\sigma^2 = \frac{1}{N}\sum_{n=1}^{N}(x_n - y_n)^2. \tag{8.3}$$

If we are interested in the size of the error relative to the signal, we can find the ratio of the average squared value of the source output and the mse. This is called the *signal-to-noise ratio* (SNR).

$$\text{SNR} = \frac{\sigma_x^2}{\sigma_d^2} \tag{8.4}$$

where σ_x^2 is the average squared value of the source output, or signal, and σ_d^2 is the mse. The SNR is often measured on a logarithmic scale and the units of measurement are *decibels* (abbreviated to dB).

$$\text{SNR(dB)} = 10\log_{10}\frac{\sigma_x^2}{\sigma_d^2} \tag{8.5}$$

Sometimes we are more interested in the size of the error relative to the peak value of the signal x_{peak} than with the size of the error relative to the average squared value of the signal. This ratio is called the *peak-signal-to-noise-ratio* (PSNR) and is given by

$$\text{PSNR(dB)} = 10\log_{10}\frac{x_{\text{peak}}^2}{\sigma_d^2}. \tag{8.6}$$

Another difference distortion measure that is used quite often, although not as often as the mse, is the average of the absolute difference, or

$$d_1 = \frac{1}{N}\sum_{n=1}^{N} |x_n - y_n| \,. \tag{8.7}$$

This measure seems especially useful for evaluating image compression algorithms.

In some applications, the distortion is not perceptible as long as it is below some threshold. In these situations we might be interested in the maximum value of the error magnitude,

$$d_\infty = \max_n |x_n - y_n| \,. \tag{8.8}$$

We have looked at two approaches to measuring the fidelity of a reconstruction. The first method involving humans may provide a very accurate measure of perceptible fidelity, but it is not practical and not useful in mathematical design approaches. The second is mathematically tractable, but it usually does not provide a very accurate indication of the perceptible fidelity of the reconstruction. A middle ground is to find a mathematical model for human perception, transform both the source output and the reconstruction to this perceptual space, and then measure the difference in the perceptual space. For example, suppose we could find a transformation $\mathcal{V}$ that represented the actions performed by the human visual system (HVS) on the light intensity impinging on the retina before it is "perceived" by the cortex. We could then find $\mathcal{V}(x)$ and $\mathcal{V}(y)$ and examine the difference between them. There are two problems with this approach. First, the process of human perception is very difficult to model, and accurate models of perception are yet to be discovered. Second, even if we could find a mathematical model for perception, the odds are that it would be so complex that it would be mathematically intractable.

In spite of these disheartening prospects, the study of perception mechanisms is still important from the perspective of design and analysis of compression systems. Even if we cannot obtain a transformation that accurately models perception, we can learn something about the properties of perception that may come in handy in the design of compression systems. In the following, we will look at some of the properties of the human visual system and the perception of sound. Our review will be far from thorough, but the intent here is to present some properties that will be useful in later chapters when we talk about compression of images, video, speech, and audio.

8.3.1 The Human Visual System

The eye is a globe-shaped object with a lens in the front that focuses objects onto the retina in the back of the eye. The retina contains two kinds of receptors, called *rods* and *cones*. The rods are more sensitive to light than cones, and in low light most of our vision is due to the operation of rods. There are three kinds of cones, each of which are most sensitive at different wavelengths of the visible spectrum. The peak sensitivities of the cones are in the red, blue, and green regions of the visible spectrum [93]. The cones are mostly concentrated in a very small area of the retina called the *fovea*. Although the rods are more numerous than the cones, the cones provide better resolution because they are more closely packed in the fovea. The muscles of the eye move the eyeball, positioning the image of the object on

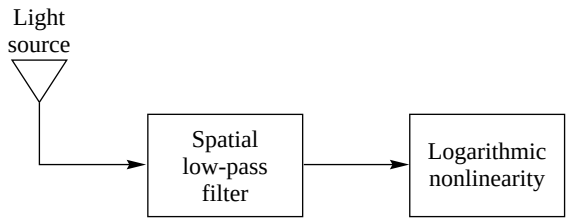

FIGURE 8.2 A model of monochromatic vision.

the fovea. This becomes a drawback in low light. One way to improve what you see in low light is to focus to one side of the object. This way the object is imaged on the rods, which are more sensitive to light.

The eye is sensitive to light over an enormously large range of intensities; the upper end of the range is about 10^{10} times the lower end of the range. However, at a given instant we cannot perceive the entire range of brightness. Instead, the eye adapts to an average brightness level. The range of brightness levels that the eye can perceive at any given instant is much smaller than the total range it is capable of perceiving.

If we illuminate a screen with a certain intensity I and shine a spot on it with different intensity, the spot becomes visible when the difference in intensity is ΔI. This is called the *just noticeable difference* (jnd). The ratio $\frac{\Delta I}{I}$ is known as the *Weber fraction* or *Weber ratio*. This ratio is known to be constant at about 0.02 over a wide range of intensities in the absence of background illumination. However, if the background illumination is changed, the range over which the Weber ratio remains constant becomes relatively small. The constant range is centered around the intensity level to which the eye adapts.

If $\frac{\Delta I}{I}$ is constant, then we can infer that the sensitivity of the eye to intensity is a logarithmic function ($d(\log I) = dI/I$). Thus, we can model the eye as a receptor whose output goes to a logarithmic nonlinearity. We also know that the eye acts as a spatial low-pass filter [94, 95]. Putting all of this information together, we can develop a model for monochromatic vision, shown in Figure 8.2.

How does this description of the human visual system relate to coding schemes? Notice that the mind does not perceive everything the eye sees. We can use this knowledge to design compression systems such that the distortion introduced by our lossy compression scheme is not noticeable.

8.3.2 Auditory Perception

The ear is divided into three parts, creatively named the outer ear, the middle ear, and the inner ear. The outer ear consists of the structure that directs the sound waves, or pressure waves, to the *tympanic membrane*, or eardrum. This membrane separates the outer ear from the middle ear. The middle ear is an air-filled cavity containing three small bones that provide coupling between the tympanic membrane and the *oval window*, which leads into the inner ear. The tympanic membrane and the bones convert the pressure waves in the air to acoustical vibrations. The inner ear contains, among other things, a snail-shaped passage called the *cochlea* that contains the transducers that convert the acoustical vibrations to nerve impulses.

The human ear can hear sounds from approximately 20 Hz to 20 kHz, a 1000:1 range of frequencies. The range decreases with age; older people are usually unable to hear the higher frequencies. As in vision, auditory perception has several nonlinear components. One is that loudness is a function not only of the sound level, but also of the frequency. Thus, for example, a pure 1 kHz tone presented at a 20 dB intensity level will have the same apparent loudness as a 50 Hz tone presented at a 50 dB intensity level. By plotting the amplitude of tones at different frequencies that sound equally loud, we get a series of curves called the *Fletcher-Munson curves* [96].

Another very interesting audio phenomenon is that of *masking*, where one sound blocks out or masks the perception of another sound. The fact that one sound can drown out another seems reasonable. What is not so intuitive about masking is that if we were to try to mask a pure tone with noise, only the noise in a small frequency range around the tone being masked contributes to the masking. This range of frequencies is called the *critical band*. For most frequencies, when the noise just masks the tone, the ratio of the power of the tone divided by the power of the noise in the critical band is a constant [97]. The width of the critical band varies with frequency. This fact has led to the modeling of auditory perception as a bank of band-pass filters. There are a number of other, more complicated masking phenomena that also lend support to this theory (see [97, 98] for more information). The limitations of auditory perception play a major role in the design of audio compression algorithms. We will delve further into these limitations when we discuss audio compression in Chapter 16.

8.4 Information Theory Revisited ★

In order to study the trade-offs between rate and the distortion of lossy compression schemes, we would like to have rate defined explicitly as a function of the distortion for a given distortion measure. Unfortunately, this is generally not possible, and we have to go about it in a more roundabout way. Before we head down this path, we need a few more concepts from information theory.

In Chapter 2, when we talked about information, we were referring to letters from a single alphabet. In the case of lossy compression, we have to deal with two alphabets, the source alphabet and the reconstruction alphabet. These two alphabets are generally different from each other.

Example 8.4.1:

A simple lossy compression approach is to drop a certain number of the least significant bits from the source output. We might use such a scheme between a source that generates monochrome images at 8 bits per pixel and a user whose display facility can display only 64 different shades of gray. We could drop the two least significant bits from each pixel before transmitting the image to the user. There are other methods we can use in this situation that are much more effective, but this is certainly simple.

Suppose our source output consists of 4-bit words $\{0, 1, 2, \ldots, 15\}$. The source encoder encodes each value by shifting out the least significant bit. The output alphabet for the source coder is $\{0, 1, 2, \ldots, 7\}$. At the receiver we cannot recover the original value exactly. However,

we can get an approximation by shifting in a 0 as the least significant bit, or in other words, multiplying the source encoder output by two. Thus, the reconstruction alphabet is $\{0, 2, 4, \ldots, 14\}$, and the source and reconstruction do not take values from the same alphabet. ♦

As the source and reconstruction alphabets can be distinct, we need to be able to talk about the information relationships between two random variables that take on values from two different alphabets.

8.4.1 Conditional Entropy

Let X be a random variable that takes values from the source alphabet $\mathcal{X} = \{x_0, x_1, \ldots, x_{N-1}\}$. Let Y be a random variable that takes on values from the reconstruction alphabet $\mathcal{Y} = \{y_0, y_1, \ldots, y_{M-1}\}$. From Chapter 2 we know that the entropy of the source and the reconstruction are given by

$$H(X) = -\sum_{i=0}^{N-1} P(x_i) \log_2 P(x_i)$$

and

$$H(Y) = -\sum_{j=0}^{M-1} P(y_j) \log_2 P(y_j).$$

A measure of the relationship between two random variables is the *conditional entropy* (the average value of the conditional self-information). Recall that the self-information for an event A was defined as

$$i(A) = \log \frac{1}{P(A)} = -\log P(A).$$

In a similar manner, the conditional self-information of an event A, given that another event B has occurred, can be defined as

$$i(A|B) = \log \frac{1}{P(A|B)} = -\log P(A|B).$$

Suppose B is the event "Frazer has not drunk anything in two days," and A is the event "Frazer is thirsty." Then $P(A|B)$ should be close to one, which means that the conditional self-information $i(A|B)$ would be close to zero. This makes sense from an intuitive point of view as well. If we know that Frazer has not drunk anything in two days, then the statement that Frazer is thirsty would not be at all surprising to us and would contain very little information.

As in the case of self-information, we are generally interested in the average value of the conditional self-information. This average value is called the conditional entropy. The conditional entropies of the source and reconstruction alphabets are given as

$$H(X|Y) = -\sum_{i=0}^{N-1} \sum_{j=0}^{M-1} P(x_i|y_j) P(y_j) \log_2 P(x_i|y_j) \tag{8.9}$$

and

$$H(Y|X) = -\sum_{i=0}^{N-1}\sum_{j=0}^{M-1} P(x_i|y_j)P(y_j)\log_2 P(y_j|x_i). \tag{8.10}$$

The conditional entropy $H(X|Y)$ can be interpreted as the amount of uncertainty remaining about the random variable X, or the source output, given that we know what value the reconstruction Y took. The additional knowledge of Y should reduce the uncertainty about X, and we can show that

$$H(X|Y) \leq H(X) \tag{8.11}$$

(see Problem 5).

Example 8.4.2:

Suppose we have the 4-bits-per-symbol source and compression scheme described in Example 8.4.1. Assume that the source is equally likely to select any letter from its alphabet. Let us calculate the various entropies for this source and compression scheme.

As the source outputs are all equally likely, $P(X = i) = \frac{1}{16}$ for all $i \in \{0, 1, 2, \ldots, 15\}$, and therefore

$$H(X) = -\sum_i \frac{1}{16}\log\frac{1}{16} = \log 16 = 4 \text{ bits}. \tag{8.12}$$

We can calculate the probabilities of the reconstruction alphabet:

$$P(Y = j) = P(X = j) + P(X = j+1) = \frac{1}{16} + \frac{1}{16} = \frac{1}{8}. \tag{8.13}$$

Therefore, $H(Y) = 3$ bits. To calculate the conditional entropy $H(X|Y)$, we need the conditional probabilities $\{P(x_i|y_j)\}$. From our construction of the source encoder, we see that

$$P(X = i|Y = j) = \begin{cases} \frac{1}{2} & \text{if } i = j \text{ or } i = j+1, \text{ for } j = 0, 2, 4, \ldots, 14 \\ 0 & \text{otherwise.} \end{cases} \tag{8.14}$$

Substituting this in the expression for $H(X|Y)$ in Equation (8.9), we get

$$\begin{aligned} H(X|Y) &= -\sum_i\sum_j P(X = i|Y = j)P(Y = j)\log P(X = i|Y = j) \\ &= -\sum_j [P(X = j|Y = j)P(Y = j)\log P(X = j|Y = j) \\ &\quad + P(X = j+1|Y = j)P(Y = j)\log P(X = j+1|Y = j)] \\ &= -8\left[\frac{1}{2}\cdot\frac{1}{8}\log\frac{1}{2} + \frac{1}{2}\cdot\frac{1}{8}\log\frac{1}{2}\right] \qquad (8.15) \\ &= 1. \qquad (8.16) \end{aligned}$$

Let us compare this answer to what we would have intuitively expected the uncertainty to be, based on our knowledge of the compression scheme. With the coding scheme described

here, knowledge of Y means that we know the first 3 bits of the input X. The only thing about the input that we are uncertain about is the value of the last bit. In other words, if we know the value of the reconstruction, our uncertainty about the source output is 1 bit. Therefore, at least in this case, our intuition matches the mathematical definition.

To obtain $H(Y|X)$, we need the conditional probabilities $\{P(y_j|x_i)\}$. From our knowledge of the compression scheme, we see that

$$P(Y=j|X=i) = \begin{cases} 1 & \text{if } i=j \text{ or } i=j+1, \text{ for } j=0,2,4,\ldots,14 \\ 0 & \text{otherwise.} \end{cases} \tag{8.17}$$

If we substitute these values into Equation (8.10), we get $H(Y|X)=0$ bits (note that $0\log 0 = 0$). This also makes sense. For the compression scheme described here, if we know the source output, we know 4 bits, the first 3 of which are the reconstruction. Therefore, in this example, knowledge of the source output at a specific time completely specifies the corresponding reconstruction. ♦

8.4.2 Average Mutual Information

We make use of one more quantity that relates the uncertainty or entropy of two random variables. This quantity is called the *mutual information* and is defined as

$$i(x_k; y_j) = \log\left[\frac{P(x_k|y_j)}{P(x_k)}\right]. \tag{8.18}$$

We will use the average value of this quantity, appropriately called the *average mutual information*, which is given by

$$I(X;Y) = \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} P(x_i, y_j)\log\left[\frac{P(x_i|y_j)}{P(x_i)}\right] \tag{8.19}$$

$$= \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} P(x_i|y_j)P(y_j)\log\left[\frac{P(x_i|y_j)}{P(x_i)}\right]. \tag{8.20}$$

We can write the average mutual information in terms of the entropy and the conditional entropy by expanding the argument of the logarithm in Equation (8.20).

$$I(X;Y) = \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} P(x_i, y_j)\log\left[\frac{P(x_i|y_j)}{P(x_i)}\right] \tag{8.21}$$

$$= \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} P(x_i, y_j)\log P(x_i|y_j) - \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} P(x_i, y_j)\log P(x_i) \tag{8.22}$$

$$= H(X) - H(X|Y) \tag{8.23}$$

where the second term in Equation (8.22) is $H(X)$, and the first term is $-H(X|Y)$. Thus, the average mutual information is the entropy of the source minus the uncertainty that remains

about the source output after the reconstructed value has been received. The average mutual information can also be written as

$$I(X; Y) = H(Y) - H(Y|X) = I(Y; X). \tag{8.24}$$

Example 8.4.3:

For the source coder of Example 8.4.2, $H(X) = 4$ bits, and $H(X|Y) = 1$ bit. Therefore, using Equation (8.23), the average mutual information $I(X; Y)$ is 3 bits. If we wish to use Equation (8.24) to compute $I(X; Y)$, we would need $H(Y)$ and $H(Y|X)$, which from Example 8.4.2 are 3 and 0, respectively. Thus, the value of $I(X; Y)$ still works out to be 3 bits. ♦

8.4.3 Differential Entropy

Up to this point we have assumed that the source picks its outputs from a discrete alphabet. When we study lossy compression techniques, we will see that for many sources of interest to us this assumption is not true. In this section, we will extend some of the information theoretic concepts defined for discrete random variables to the case of random variables with continuous distributions.

Unfortunately, we run into trouble from the very beginning. Recall that the first quantity we defined was self-information, which was given by $\log \frac{1}{P(x_i)}$, where $P(x_i)$ is the probability that the random variable will take on the value x_i. For a random variable with a continuous distribution, this probability is zero. Therefore, if the random variable has a continuous distribution, the "self-information" associated with any value is infinity.

If we do not have the concept of self-information, how do we go about defining entropy, which is the average value of the self-information? We know that many continuous functions can be written as limiting cases of their discretized version. We will try to take this route in order to define the entropy of a continuous random variable X with probability density function *(pdf)* $f_X(x)$.

While the random variable X cannot generally take on a particular value with nonzero probability, it can take on a value in an *interval* with nonzero probability. Therefore, let us divide the range of the random variable into intervals of size Δ. Then, by the mean value theorem, in each interval $[(i-1)\Delta, i\Delta)$, there exists a number x_i, such that

$$f_X(x_i)\Delta = \int_{(i-1)\Delta}^{i\Delta} f_X(x)\, dx. \tag{8.25}$$

Let us define a discrete random variable X_d with *pdf*

$$P(X_d = x_i) = f_X(x_i)\Delta. \tag{8.26}$$

Then we can obtain the entropy of this random variable as

$$H(X_d) = -\sum_{i=-\infty}^{\infty} P(x_i) \log P(x_i) \tag{8.27}$$

$$= -\sum_{i=-\infty}^{\infty} f_X(x_i)\Delta \log f_X(x_i)\Delta \tag{8.28}$$

$$= -\sum_{i=-\infty}^{\infty} f_X(x_i)\Delta \log f_X(x_i) - \sum_{i=-\infty}^{\infty} f_X(x_i)\Delta \log \Delta \tag{8.29}$$

$$= -\sum_{i=-\infty}^{\infty} [f_X(x_i) \log f_X(x_i)]\,\Delta - \log \Delta. \tag{8.30}$$

Taking the limit as $\Delta \to 0$ of Equation (8.30), the first term goes to $-\int_{-\infty}^{\infty} f_X(x) \log f_X(x)\, dx$, which looks like the analog to our definition of entropy for discrete sources. However, the second term is $-\log \Delta$, which goes to plus infinity when Δ goes to zero. It seems there is not an analog to entropy as defined for discrete sources. However, the first term in the limit serves some functions similar to that served by entropy in the discrete case and is a useful function in its own right. We call this term the *differential entropy* of a continuous source and denote it by $h(X)$.

Example 8.4.4:

Suppose we have a random variable X that is uniformly distributed in the interval $[a, b)$. The differential entropy of this random variable is given by

$$h(X) = -\int_{-\infty}^{\infty} f_X(x) \log f_X(x)\, dx \tag{8.31}$$

$$= -\int_{a}^{b} \frac{1}{b-a} \log \frac{1}{b-a}\, dx \tag{8.32}$$

$$= \log(b-a). \tag{8.33}$$

Notice that when $b-a$ is less than one, the differential entropy will become negative—in contrast to the entropy, which never takes on negative values. ♦

Later in this chapter, we will find particular use for the differential entropy of the Gaussian source.

Example 8.4.5:

Suppose we have a random variable X that has a Gaussian *pdf*,

$$f_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp -\frac{(x-\mu)^2}{2\sigma^2}. \tag{8.34}$$

The differential entropy is given by

$$h(X) = -\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}} \exp -\frac{(x-\mu)^2}{2\sigma^2} \log \left[\frac{1}{\sqrt{2\pi\sigma^2}} \exp -\frac{(x-\mu)^2}{2\sigma^2} \right] dx \tag{8.35}$$

$$= -\log \frac{1}{\sqrt{2\pi\sigma^2}} \int_{-\infty}^{\infty} f_X(x) dx + \int_{-\infty}^{\infty} \frac{(x-\mu)^2}{2\sigma^2} \log e f_X(x) dx \tag{8.36}$$

$$= \frac{1}{2}\log 2\pi\sigma^2 + \frac{1}{2}\log e \tag{8.37}$$

$$= \frac{1}{2}\log 2\pi e\sigma^2. \tag{8.38}$$

Thus, the differential entropy of a Gaussian random variable is directly proportional to its variance. ♦

The differential entropy for the Gaussian distribution has the added distinction that it is larger than the differential entropy for any other continuously distributed random variable with the same variance. That is, for any random variable X, with variance σ^2

$$h(X) \leq \frac{1}{2}\log 2\pi e\sigma^2. \tag{8.39}$$

The proof of this statement depends on the fact that for any two continuous distributions $f_X(X)$ and $g_X(X)$

$$-\int_{-\infty}^{\infty} f_X(x)\log f_X(x)dx \leq -\int_{-\infty}^{\infty} f_X(x)\log g_X(x)dx. \tag{8.40}$$

We will not prove Equation (8.40) here, but you may refer to [99] for a simple proof. To obtain Equation (8.39), we substitute the expression for the Gaussian distribution for $g_X(x)$. Noting that the left-hand side of Equation (8.40) is simply the differential entropy of the random variable X, we have

$$\begin{aligned} h(X) &\leq -\int_{-\infty}^{\infty} f_X(x)\log\frac{1}{\sqrt{2\pi\sigma^2}}\exp-\frac{(x-\mu)^2}{2\sigma^2}dx \\ &= \frac{1}{2}\log(2\pi\sigma^2) + \log e\int_{-\infty}^{\infty} f_X(x)\frac{(x-\mu)^2}{2\sigma^2}dx \\ &= \frac{1}{2}\log(2\pi\sigma^2) + \frac{\log e}{2\sigma^2}\int_{-\infty}^{\infty} f_X(x)(x-\mu)^2dx \\ &= \frac{1}{2}\log(2\pi e\sigma^2). \end{aligned} \tag{8.41}$$

We seem to be striking out with continuous random variables. There is no analog for self-information and really none for entropy either. However, the situation improves when we look for an analog for the average mutual information. Let us define the random variable Y_d in a manner similar to the random variable X_d, as the discretized version of a continuous valued random variable Y. Then we can show (see Problem 4)

$$H(X_d|Y_d) = -\sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty}\left[f_{X|Y}(x_i|y_j)f_Y(y_j)\log f_{X|Y}(x_i|y_j)\right]\Delta\Delta - \log\Delta. \tag{8.42}$$

Therefore, the average mutual information for the discretized random variables is given by

$$I(X_d; Y_d) = H(X_d) - H(X_d|Y_d) \tag{8.43}$$

$$= -\sum_{i=-\infty}^{\infty} f_X(x_i)\Delta \log f_X(x_i) \tag{8.44}$$

$$-\sum_{i=-\infty}^{\infty}\left[\sum_{j=-\infty}^{\infty} f_{X|Y}(x_i|y_j)f_Y(y_j)\log f_{X|Y}(x_i|y_j)\Delta\right]\Delta. \tag{8.45}$$

Notice that the two $\log\Delta$s in the expression for $H(X_d)$ and $H(X_d|Y_d)$ cancel each other out, and as long as $h(X)$ and $h(X|Y)$ are not equal to infinity, when we take the limit as $\Delta \to 0$ of $I(X_d; Y_d)$ we get

$$I(X; Y) = h(X) - h(X|Y). \tag{8.46}$$

The average mutual information in the continuous case can be obtained as a limiting case of the average mutual information for the discrete case and has the same physical significance.

We have gone through a lot of mathematics in this section. But the information will be used immediately to define the rate distortion function for a random source.

8.5 Rate Distortion Theory ★

Rate distortion theory is concerned with the trade-offs between distortion and rate in lossy compression schemes. Rate is defined as the average number of bits used to represent each sample value. One way of representing the trade-offs is via a *rate distortion function R(D)*. The rate distortion function $R(D)$ specifies the lowest rate at which the output of a source can be encoded while keeping the distortion less than or equal to D. On our way to mathematically defining the rate distortion function, let us look at the rate and distortion for some different lossy compression schemes.

In Example 8.4.2, knowledge of the value of the input at time k completely specifies the reconstructed value at time k. In this situation,

$$P(y_j|x_i) = \begin{cases} 1 & \text{for some } j = j_i \\ 0 & \text{otherwise.} \end{cases} \tag{8.47}$$

Therefore,

$$D = \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} P(y_j|x_i)P(x_i)d(x_i, y_j) \tag{8.48}$$

$$= \sum_{i=0}^{N-1} P(x_i)d(x_i, y_{j_i}) \tag{8.49}$$

where we used the fact that $P(x_i, y_j) = P(y_j|x_i)P(x_i)$ in Equation (8.48). The rate for this source coder is the output entropy $H(Y)$ of the source decoder. If this were always the case, the task of obtaining a rate distortion function would be relatively simple. Given a

distortion constraint D^*, we could look at all encoders with distortion less than D^* and pick the one with the lowest output entropy. This entropy would be the rate corresponding to the distortion D^*. However, the requirement that knowledge of the input at time k completely specifies the reconstruction at time k is very restrictive, and there are many efficient compression techniques that would have to be excluded under this requirement. Consider the following example.

Example 8.5.1:

With a data sequence that consists of height and weight measurements, obviously height and weight are quite heavily correlated. In fact, after studying a long sequence of data, we find that if we plot the height along the x axis and the weight along the y axis, the data points cluster along the line $y = 2.5x$. In order to take advantage of this correlation, we devise the following compression scheme. For a given pair of height and weight measurements, we find the orthogonal projection on the $y = 2.5x$ line as shown in Figure 8.3. The point on this line can be represented as the distance to the nearest integer from the origin. Thus, we encode a pair of values into a single value. At the time of reconstruction, we simply map this value back into a pair of height and weight measurements.

For instance, suppose somebody is 72 inches tall and weighs 200 pounds (point A in Figure 8.3). This corresponds to a point at a distance of 212 along the $y = 2.5x$ line. The reconstructed values of the height and weight corresponding to this value are 79 and 197. Notice that the reconstructed values differ from the original values. Suppose we now have

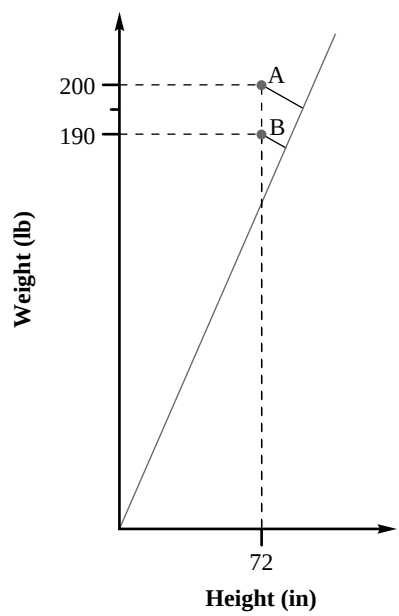

FIGURE 8.3 **Compression scheme for encoding height-weight pairs.**

another individual who is also 72 inches tall but weighs 190 pounds (point B in Figure 8.3). The source coder output for this pair would be 203, and the reconstructed values for height and weight are 75 and 188, respectively. Notice that while the height value in both cases was the same, the reconstructed value is different. The reason for this is that the reconstructed value for the height depends on the weight. Thus, for this particular source coder, we do not have a conditional probability density function $\{P(y_j|x_i)\}$ of the form shown in Equation (8.47). ♦

Let us examine the distortion for this scheme a little more closely. As the conditional probability for this scheme is not of the form of Equation (8.47), we can no longer write the distortion in the form of Equation (8.49). Recall that the general form of the distortion is

$$D = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} d(x_i, y_j) P(x_i) P(y_j|x_i). \tag{8.50}$$

Each term in the summation consists of three factors: the distortion measure $d(x_i, y_j)$, the source density $P(x_i)$, and the conditional probability $P(y_j|x_i)$. The distortion measure is a measure of closeness of the original and reconstructed versions of the signal and is generally determined by the particular application. The source probabilities are solely determined by the source. The third factor, the set of conditional probabilities, can be seen as a description of the compression scheme.

Therefore, for a given source with some *pdf* $\{P(x_i)\}$ and a specified distortion measure $d(\cdot, \cdot)$, the distortion is a function only of the conditional probabilities $\{P(y_j|x_i)\}$; that is,

$$D = D(\{P(y_j|x_i)\}). \tag{8.51}$$

Therefore, we can write the constraint that the distortion D be less than some value D^* as a requirement that the conditional probabilities for the compression scheme belong to a set of conditional probabilities Γ that have the property that

$$\Gamma = \{\{P(y_j|x_i\} \text{ such that } D(\{P(y_j|x_i)\}) \leq D^*\}. \tag{8.52}$$

Once we know the set of compression schemes to which we have to confine ourselves, we can start to look at the rate of these schemes. In Example 8.4.2, the rate was the entropy of Y. However, that was a result of the fact that the conditional probability describing that particular source coder took on only the values 0 and 1. Consider the following trivial situation.

Example 8.5.2:

Suppose we have the same source as in Example 8.4.2 and the same reconstruction alphabet. Suppose the distortion measure is

$$d(x_i, y_j) = (x_i - y_j)^2$$

and $D^* = 225$. One compression scheme that satisfies the distortion constraint randomly maps the input to any one of the outputs; that is,

$$P(y_j|x_i) = \frac{1}{8} \qquad \text{for } i = 0, 1, \ldots, 15 \text{ and } j = 0, 2, \ldots, 14.$$

We can see that this conditional probability assignment satisfies the distortion constraint. As each of the eight reconstruction values is equally likely, $H(Y)$ is 3 bits. However, we are not transmitting *any* information. We could get exactly the same results by transmitting 0 bits and randomly picking Y at the receiver. ♦

Therefore, the entropy of the reconstruction $H(Y)$ cannot be a measure of the rate. In his 1959 paper on source coding [100], Shannon showed that the minimum rate for a given distortion is given by

$$R(D) = \min_{\{P(y_j|x_i)\}\in\Gamma} I(X; Y). \tag{8.53}$$

To prove this is beyond the scope of this book. (Further information can be found in [3] and [4].) However, we can at least convince ourselves that defining the rate as an average mutual information gives sensible answers when used for the examples shown here. Consider Example 8.4.2. The average mutual information in this case is 3 bits, which is what we said the rate was. In fact, notice that whenever the conditional probabilities are constrained to be of the form of Equation (8.47),

$$H(Y|X) = 0,$$

then

$$I(X; Y) = H(Y),$$

which had been our measure of rate.

In Example 8.5.2, the average mutual information is 0 bits, which accords with our intuitive feeling of what the rate should be. Again, whenever

$$H(Y|X) = H(Y),$$

that is, knowledge of the source gives us no knowledge of the reconstruction,

$$I(X; Y) = 0,$$

which seems entirely reasonable. We should not have to transmit any bits when we are not sending any information.

At least for the examples here, it seems that the average mutual information does represent the rate. However, earlier we had said that the average mutual information between the source output and the reconstruction is a measure of the information conveyed by the reconstruction about the source output. Why are we then looking for compression schemes that *minimize* this value? To understand this, we have to remember that the process of finding the performance of the optimum compression scheme had two parts. In the first part we

specified the desired distortion. The entire set of conditional probabilities over which the average mutual information is minimized satisfies the distortion constraint. Therefore, we can leave the question of distortion, or fidelity, aside and concentrate on minimizing the rate.

Finally, how do we find the rate distortion function? There are two ways: one is a computational approach developed by Arimoto [101] and Blahut [102]. While the derivation of the algorithm is beyond the scope of this book, the algorithm itself is relatively simple. The other approach is to find a lower bound for the average mutual information and then show that we can achieve this bound. We use this approach to find the rate distortion functions for two important sources.

Example 8.5.3: Rate distortion function for the binary source

Suppose we have a source alphabet $\{0, 1\}$, with $P(0) = p$. The reconstruction alphabet is also binary. Given the distortion measure

$$d(x_i, y_j) = x_i \oplus y_j, \tag{8.54}$$

where $\oplus$ is modulo 2 addition, let us find the rate distortion function. Assume for the moment that $p < \frac{1}{2}$. For $D > p$ an encoding scheme that would satisfy the distortion criterion would be not to transmit anything and fix $Y = 1$. So for $D \geq p$

$$R(D) = 0. \tag{8.55}$$

We will find the rate distortion function for the distortion range $0 \leq D < p$.

Find a lower bound for the average mutual information:

$$\begin{aligned} I(X; Y) &= H(X) - H(X|Y) && (8.56)\\ &= H(X) - H(X \oplus Y|Y) && (8.57)\\ &\geq H(X) - H(X \oplus Y) \qquad \text{from Equation (8.11).} && (8.58) \end{aligned}$$

In the second step we have used the fact that if we know Y, then knowing X we can obtain $X \oplus Y$ and vice versa as $X \oplus Y \oplus Y = X$.

Let us look at the terms on the right-hand side of (8.11):

$$H(X) = -p \log_2 p - (1-p) \log_2 (1-p) = H_b(p), \tag{8.59}$$

where $H_b(p)$ is called the *binary entropy function* and is plotted in Figure 8.4. Note that $H_b(p) = H_b(1-p)$.

Given that $H(X)$ is completely specified by the source probabilities, our task now is to find the conditional probabilities $\{P(x_i|y_j)\}$ such that $H(X \oplus Y)$ is maximized while the average distortion $E[d(x_i, y_j)] \leq D$. $H(X \oplus Y)$ is simply the binary entropy function $H_b(P(X \oplus Y = 1))$, where

$$P(X \oplus Y = 1) = P(X = 0, Y = 1) + P(X = 1, Y = 0). \tag{8.60}$$

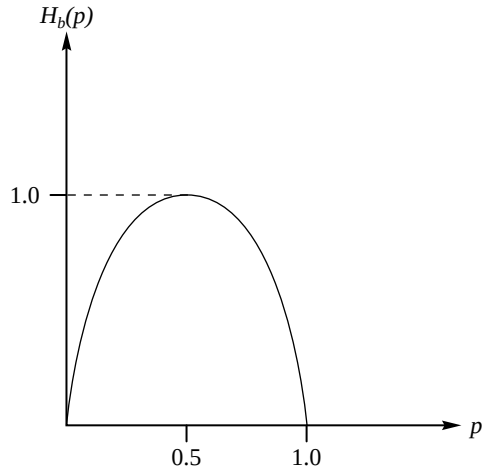

FIGURE 8.4 The binary entropy function.

Therefore, to maximize $H(X \oplus Y)$, we would want $P(X \oplus Y = 1)$ to be as close as possible to one-half. However, the selection of $P(X \oplus Y)$ also has to satisfy the distortion constraint. The distortion is given by

$$\begin{aligned} E[d(x_i, y_j)] &= 0 \times P(X = 0, Y = 0) + 1 \times P(X = 0, Y = 1) \\ &\quad + 1 \times P(X = 1, Y = 0) + 0 \times P(X = 1, Y = 1) \\ &= P(X = 0, Y = 1) + P(X = 1, Y = 0) \\ &= P(Y = 1|X = 0)p + P(Y = 0|X = 1)(1 - p). \end{aligned} \tag{8.61}$$

But this is simply the probability that $X \oplus Y = 1$. Therefore, the maximum value that $P(X \oplus Y = 1)$ can have is D. Our assumptions were that $D < p$ and $p \leq \frac{1}{2}$, which means that $D < \frac{1}{2}$. Therefore, $P(X \oplus Y = 1)$ is closest to $\frac{1}{2}$ while being less than or equal to D when $P(X \oplus Y = 1) = D$. Therefore,

$$I(X; Y) \geq H_b(p) - H_b(D). \tag{8.62}$$

We can show that for $P(X = 0|Y = 1) = P(X = 1|Y = 0) = D$, this bound is achieved. That is, if $P(X = 0|Y = 1) = P(X = 1|Y = 0) = D$, then

$$I(X; Y) = H_b(p) - H_b(D). \tag{8.63}$$

Therefore, for $D < p$ and $p \leq \frac{1}{2}$,

$$R(D) = H_b(p) - H_b(D). \tag{8.64}$$

Finally, if $p > \frac{1}{2}$, then we simply switch the roles of p and $1-p$. Putting all this together, the rate distortion function for a binary source is

$$R(D) = \begin{cases} H_b(p) - H_b(D) & \text{for } D < \min\{p, 1-p\} \\ 0 & \text{otherwise.} \end{cases} \tag{8.65}$$

♦

Example 8.5.4: Rate distortion function for the Gaussian source

Suppose we have a continuous amplitude source that has a zero mean Gaussian *pdf* with variance σ^2. If our distortion measure is given by

$$d(x, y) = (x-y)^2, \tag{8.66}$$

our distortion constraint is given by

$$E\left[(X-Y)^2\right] \leq D. \tag{8.67}$$

Our approach to finding the rate distortion function will be the same as in the previous example; that is, find a lower bound for $I(X; Y)$ given a distortion constraint, and then show that this lower bound can be achieved.

First we find the rate distortion function for $D < \sigma^2$.

$$I(X; Y) = h(X) - h(X|Y) \tag{8.68}$$

$$= h(X) - h(X-Y|Y) \tag{8.69}$$

$$\geq h(X) - h(X-Y) \tag{8.70}$$

In order to minimize the right-hand side of Equation (8.70), we have to maximize the second term subject to the constraint given by Equation (8.67). This term is maximized if $X-Y$ is Gaussian, and the constraint can be satisfied if $E\left[(X-Y)^2\right] = D$. Therefore, $h(X-Y)$ is the differential entropy of a Gaussian random variable with variance D, and the lower bound becomes

$$I(X; Y) \geq \frac{1}{2}\log(2\pi e\sigma^2) - \frac{1}{2}\log(2\pi e D) \tag{8.71}$$

$$= \frac{1}{2}\log\frac{\sigma^2}{D}. \tag{8.72}$$

This average mutual information can be achieved if Y is zero mean Gaussian with variance $\sigma^2 - D$, and

$$f_{X|Y}(x|y) = \frac{1}{\sqrt{2\pi D}}\exp\frac{-x^2}{2D}. \tag{8.73}$$

For $D > \sigma^2$, if we set $Y = 0$, then

$$I(X; Y) = 0 \tag{8.74}$$

and

$$E\left[(X-Y)^2\right] = \sigma^2 < D. \tag{8.75}$$

Therefore, the rate distortion function for the Gaussian source can be written as

$$R(D) = \begin{cases} \frac{1}{2}\log\frac{\sigma^2}{D} & \text{for } D < \sigma^2 \\ 0 & \text{for } D > \sigma^2. \end{cases} \tag{8.76}$$

♦

Like the differential entropy for the Gaussian source, the rate distortion function for the Gaussian source also has the distinction of being larger than the rate distortion function for any other source with a continuous distribution and the same variance. This is especially valuable because for many sources it can be very difficult to calculate the rate distortion function. In these situations, it is helpful to have an upper bound for the rate distortion function. It would be very nice if we also had a lower bound for the rate distortion function of a continuous random variable. Shannon described such a bound in his 1948 paper [7], and it is appropriately called the *Shannon lower bound*. We will simply state the bound here without derivation (for more information, see [4]).

The Shannon lower bound for a random variable X and the magnitude error criterion

$$d(x, y) = |x - y| \tag{8.77}$$

is given by

$$R_{SLB}(D) = h(X) - \log(2eD). \tag{8.78}$$

If we used the squared error criterion, the Shannon lower bound is given by

$$R_{SLB}(D) = h(X) - \frac{1}{2}\log(2\pi eD). \tag{8.79}$$

In this section we have defined the rate distortion function and obtained the rate distortion function for two important sources. We have also obtained upper and lower bounds on the rate distortion function for an arbitrary *iid* source. These functions and bounds are especially useful when we want to know if it is possible to design compression schemes to provide a specified rate and distortion given a particular source. They are also useful in determining the amount of performance improvement that we could obtain by designing a better compression scheme. In these ways the rate distortion function plays the same role for lossy compression that entropy plays for lossless compression.

8.6 Models

As in the case of lossless compression, models play an important role in the design of lossy compression algorithms; there are a variety of approaches available. The set of models we can draw on for lossy compression is much wider than the set of models we studied for

lossless compression. We will look at some of these models in this section. What is presented here is by no means an exhaustive list of models. Our only intent is to describe those models that will be useful in the following chapters.

8.6.1 Probability Models

An important method for characterizing a particular source is through the use of probability models. As we shall see later, knowledge of the probability model is important for the design of a number of compression schemes.

Probability models used for the design and analysis of lossy compression schemes differ from those used in the design and analysis of lossless compression schemes. When developing models in the lossless case, we tried for an exact match. The probability of each symbol was estimated as part of the modeling process. When modeling sources in order to design or analyze lossy compression schemes, we look more to the general rather than exact correspondence. The reasons are more pragmatic than theoretical. Certain probability distribution functions are more analytically tractable than others, and we try to match the distribution of the source with one of these "nice" distributions.

Uniform, Gaussian, Laplacian, and Gamma distribution are four probability models commonly used in the design and analysis of lossy compression systems:

- **Uniform Distribution:** As for lossless compression, this is again our ignorance model. If we do not know anything about the distribution of the source output, except possibly the range of values, we can use the uniform distribution to model the source. The probability density function for a random variable uniformly distributed between a and b is

$$f_X(x) = \begin{cases} \frac{1}{b-a} & \text{for } a \le x \le b \\ 0 & \text{otherwise.} \end{cases} \tag{8.80}$$

- **Gaussian Distribution:** The Gaussian distribution is one of the most commonly used probability models for two reasons: it is mathematically tractable and, by virtue of the central limit theorem, it can be argued that in the limit the distribution of interest goes to a Gaussian distribution. The probability density function for a random variable with a Gaussian distribution and mean μ and variance σ^2 is

$$f_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp -\frac{(x-\mu)^2}{2\sigma^2}. \tag{8.81}$$

- **Laplacian Distribution:** Many sources that we deal with have distributions that are quite peaked at zero. For example, speech consists mainly of silence. Therefore, samples of speech will be zero or close to zero with high probability. Image pixels themselves do not have any attraction to small values. However, there is a high degree of correlation among pixels. Therefore, a large number of the pixel-to-pixel differences will have values close to zero. In these situations, a Gaussian distribution is not a very close match to the data. A closer match is the Laplacian distribution, which is peaked

at zero. The distribution function for a zero mean random variable with Laplacian distribution and variance σ^2 is

$$f_X(x) = \frac{1}{\sqrt{2\sigma^2}} \exp \frac{-\sqrt{2}\,|x|}{\sigma}. \tag{8.82}$$

- **Gamma Distribution:** A distribution that is even more peaked, though considerably less tractable, than the Laplacian distribution is the Gamma distribution. The distribution function for a Gamma distributed random variable with zero mean and variance σ^2 is given by

$$f_X(x) = \frac{\sqrt[4]{3}}{\sqrt{8\pi\sigma\,|x|}} \exp \frac{-\sqrt{3}\,|x|}{2\sigma}. \tag{8.83}$$

The shapes of these four distributions, assuming a mean of zero and a variance of one, are shown in Figure 8.5.

One way of obtaining the estimate of the distribution of a particular source is to divide the range of outputs into "bins" or intervals I_k. We can then find the number of values n_k that fall into each interval. A plot of $\frac{n_k}{n_T}$, where n_T is the total number of source outputs being considered, should give us some idea of what the input distribution looks like. Be aware that this is a rather crude method and can at times be misleading. For example, if we were not careful in our selection of the source output, we might end up modeling some local peculiarities of the source. If the bins are too large, we might effectively filter out some important properties of the source. If the bin sizes are too small, we may miss out on some of the gross behavior of the source.

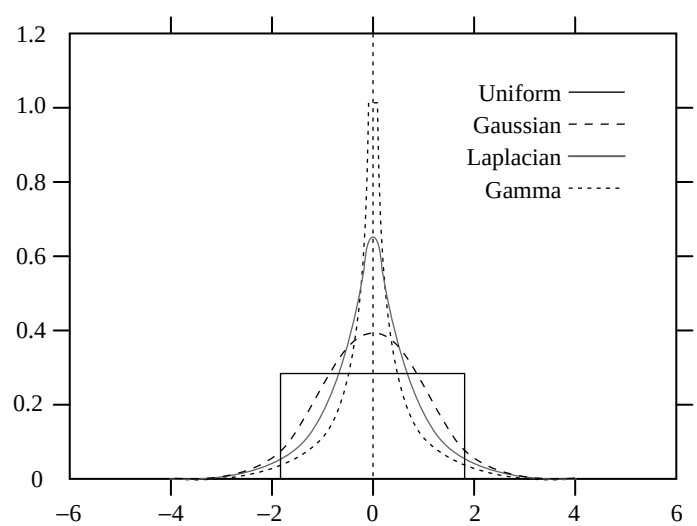

FIGURE 8.5 Uniform, Gaussian, Laplacian, and Gamma distributions.

Once we have decided on some candidate distributions, we can select between them using a number of sophisticated tests. These tests are beyond the scope of this book but are described in [103].

Many of the sources that we deal with when we design lossy compression schemes have a great deal of structure in the form of sample-to-sample dependencies. The probability models described here capture none of these dependencies. Fortunately, we have a lot of models that can capture most of this structure. We describe some of these models in the next section.

8.6.2 Linear System Models

A large class of processes can be modeled in the form of the following difference equation:

$$x_n = \sum_{i=1}^{N} a_i x_{n-i} + \sum_{j=1}^{M} b_j \epsilon_{n-j} + \epsilon_n, \tag{8.84}$$

where $\{x_n\}$ are samples of the process we wish to model, and $\{\epsilon_n\}$ is a white noise sequence. We will assume throughout this book that we are dealing with real valued samples. Recall that a zero-mean wide-sense-stationary noise sequence $\{\epsilon_n\}$ is a sequence with autocorrelation function

$$R_{\epsilon\epsilon}(k) = \begin{cases} \sigma_\epsilon^2 & \text{for } k = 0 \\ 0 & \text{otherwise.} \end{cases} \tag{8.85}$$

In digital signal-processing terminology, Equation (8.84) represents the output of a linear discrete time invariant filter with N poles and M zeros. In the statistical literature, this model is called an autoregressive moving average model of order (N,M), or an ARMA (N,M) model. The autoregressive label is because of the first summation in Equation (8.84), while the second summation gives us the moving average portion of the name.

If all the b_j were zero in Equation (8.84), only the autoregressive part of the ARMA model would remain:

$$x_n = \sum_{i=1}^{N} a_i x_{n-i} + \epsilon_n. \tag{8.86}$$

This model is called an Nth-order autoregressive model and is denoted by AR(N). In digital signal-processing terminology, this is an *all pole filter*. The AR(N) model is the most popular of all the linear models, especially in speech compression, where it arises as a natural consequence of the speech production model. We will look at it a bit more closely.

First notice that for the AR(N) process, knowing all the past history of the process gives no more information than knowing the last N samples of the process; that is,

$$P(x_n | x_{n-1}, x_{n-2}, \ldots) = P(x_n | x_{n-1}, x_{n-2}, \ldots, x_{n-N}), \tag{8.87}$$

which means that the AR(N) process is a Markov model of order N.

The autocorrelation function of a process can tell us a lot about the sample-to-sample behavior of a sequence. A slowly decaying autocorrelation function indicates a high sample-to-sample correlation, while a fast decaying autocorrelation denotes low sample-to-sample

correlation. In the case of *no* sample-to-sample correlation, such as white noise, the autocorrelation function is zero for lags greater than zero, as seen in Equation (8.85). The autocorrelation function for the AR(N) process can be obtained as follows:

$$R_{xx}(k) = E[x_n x_{n-k}] \tag{8.88}$$

$$= E\left[\left(\sum_{i=1}^{N} a_i x_{n-i} + \epsilon_n\right)(x_{n-k})\right] \tag{8.89}$$

$$= E\left[\sum_{i=1}^{N} a_i x_{n-i} x_{n-k}\right] + E[\epsilon_n x_{n-k}] \tag{8.90}$$

$$= \begin{cases} \sum_{i=1}^{N} a_i R_{xx}(k-i) & \text{for } k > 0 \\ \sum_{i=1}^{N} a_i R_{xx}(i) + \sigma_\epsilon^2 & \text{for } k = 0. \end{cases} \tag{8.91}$$

Example 8.6.1:

Suppose we have an AR(3) process. Let us write out the equations for the autocorrelation coefficient for lags 1, 2, 3:

$$\begin{aligned} R_{xx}(1) &= a_1 R_{xx}(0) + a_2 R_{xx}(1) + a_3 R_{xx}(2) \\ R_{xx}(2) &= a_1 R_{xx}(1) + a_2 R_{xx}(0) + a_3 R_{xx}(1) \\ R_{xx}(3) &= a_1 R_{xx}(2) + a_2 R_{xx}(1) + a_3 R_{xx}(0). \end{aligned}$$

If we know the values of the autocorrelation function $R_{xx}(k)$, for $k = 0, 1, 2, 3$, we can use this set of equations to find the AR(3) coefficients $\{a_1, a_2, a_3\}$. On the other hand, if we know the model coefficients and σ_ϵ^2, we can use the above equations along with the equation for $R_{xx}(0)$ to find the first four autocorrelation coefficients. All the other autocorrelation values can be obtained by using Equation (8.91). ♦

To see how the autocorrelation function is related to the temporal behavior of the sequence, let us look at the behavior of a simple AR(1) source.

Example 8.6.2:

An AR(1) source is defined by the equation

$$x_n = a_1 x_{n-1} + \epsilon_n. \tag{8.92}$$

The autocorrelation function for this source (see Problem 8) is given by

$$R_{xx}(k) = \frac{1}{1 - a_1^2} a_1^k \sigma_\epsilon^2. \tag{8.93}$$

From this we can see that the autocorrelation will decay more slowly for larger values of a_1. Remember that the value of a_1 in this case is an indicator of how closely the current

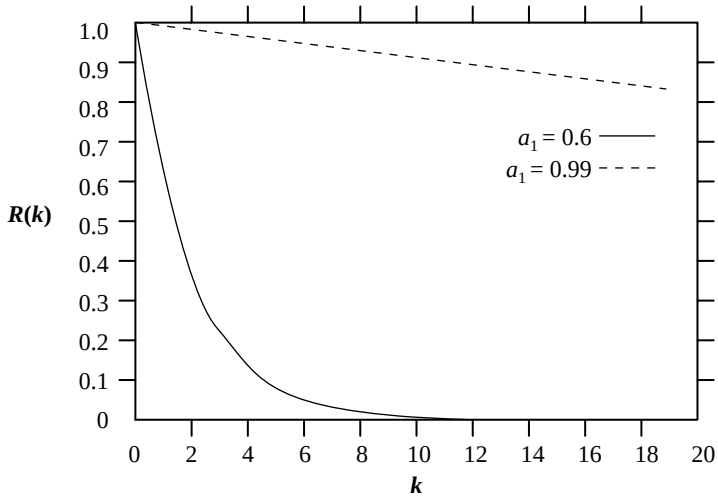

FIGURE 8.6 Autocorrelation function of an AR(1) process with two values of a_1.

sample is related to the previous sample. The autocorrelation function is plotted for two values of a_1 in Figure 8.6. Notice that for a_1 close to 1, the autocorrelation function decays extremely slowly. As the value of a_1 moves farther away from 1, the autocorrelation function decays much faster.

Sample waveforms for $a_1 = 0.99$ and $a_1 = 0.6$ are shown in Figures 8.7 and 8.8. Notice the slower variations in the waveform for the process with a higher value of a_1. Because

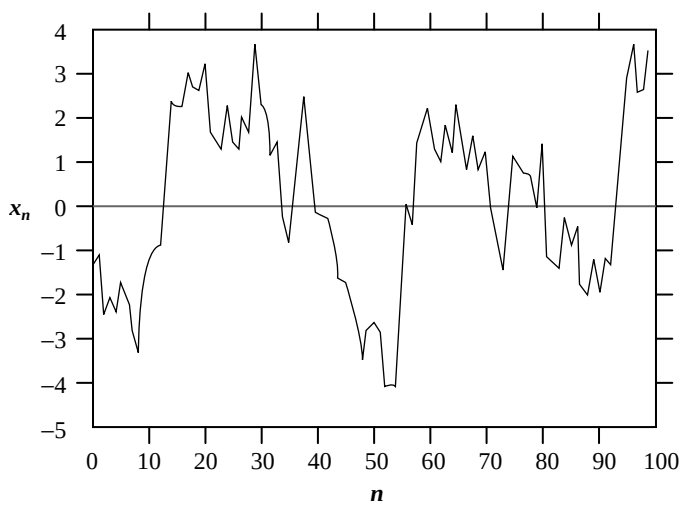

FIGURE 8.7 Sample function of an AR(1) process with $a_1 = 0.99$.

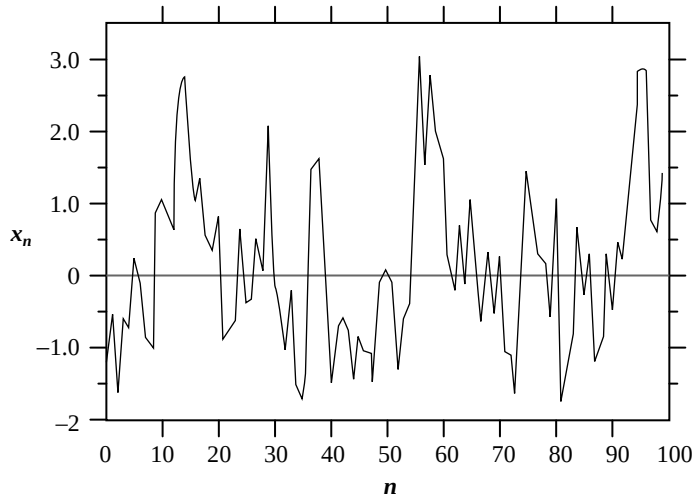

FIGURE 8. 8 **Sample function of an AR(1) process with $a_1 = 0.6$.**

the waveform in Figure 8.7 varies more slowly than the waveform in Figure 8.8, samples of this waveform are much more likely to be close in value than the samples of the waveform of Figure 8.8.

Let's look at what happens when the AR(1) coefficient is negative. The sample waveforms are plotted in Figures 8.9 and 8.10. The sample-to-sample variation in these waveforms

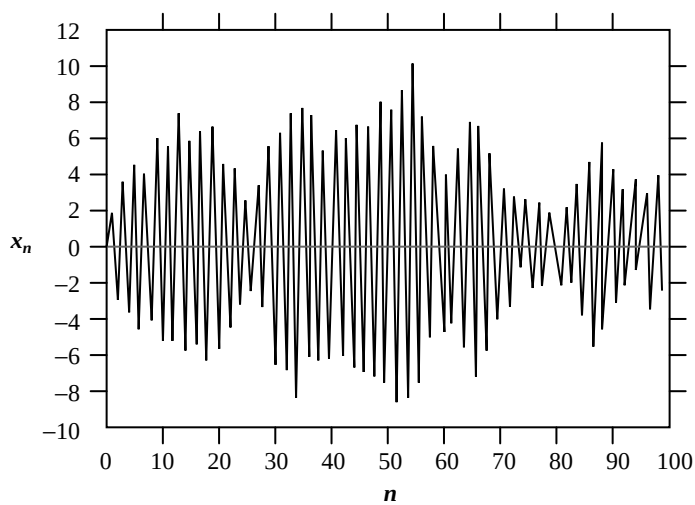

FIGURE 8. 9 **Sample function of an AR(1) process with $a_1 = -0.99$.**

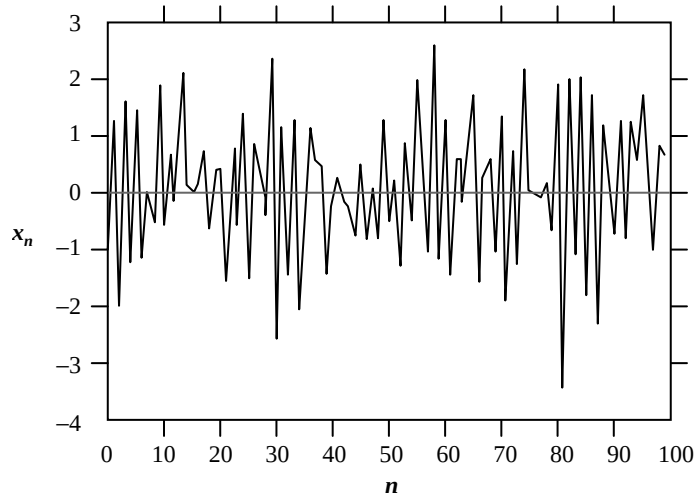

FIGURE 8.10 **Sample function of an AR(1) process with $a_1 = -0.6$.**

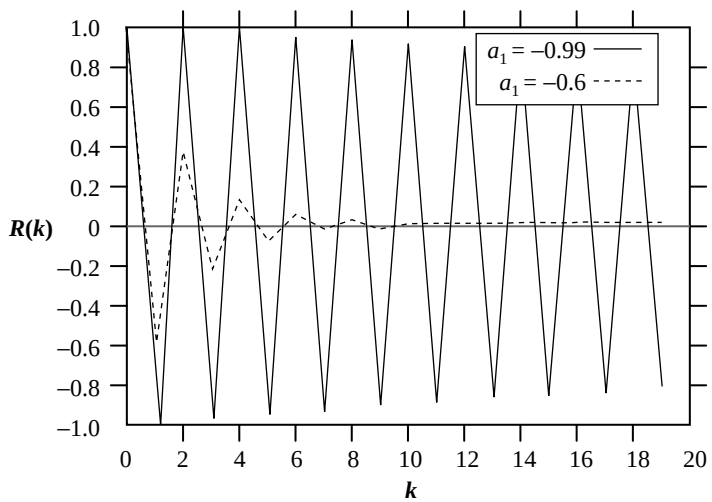

FIGURE 8.11 **Autocorrelation function of an AR(1) process with two negative values of a_1.**

is much higher than in the waveforms shown in Figures 8.7 and 8.8. However, if we were to look at the variation in magnitude, we can see that the higher value of a_1 results in magnitude values that are closer together.

This behavior is also reflected in the autocorrelation function, shown in Figure 8.11, as we might expect from looking at Equation (8.93). ♦

In Equation (8.84), instead of setting all the $\{b_j\}$ coefficients to zero, if we set all the $\{a_i\}$ coefficients to zero, we are left with the moving average part of the ARMA process:

$$x_n = \sum_{j=1}^{M} b_j \epsilon_{n-j} + \epsilon_n. \tag{8.94}$$

This process is called an Mth-order moving average process. This is a weighted average of the current and M past samples. Because of the form of this process, it is most useful when modeling slowly varying processes.

8.6.3 Physical Models

Physical models are based on the physics of the source output production. The physics are generally complicated and not amenable to a reasonable mathematical approximation. An exception to this rule is speech generation.

Speech Production

There has been a significant amount of research conducted in the area of speech production [104], and volumes have been written about it. We will try to summarize some of the pertinent aspects in this section.

Speech is produced by forcing air first through an elastic opening, the vocal cords, and then through cylindrical tubes with nonuniform diameter (the laryngeal, oral, nasal, and pharynx passages), and finally through cavities with changing boundaries such as the mouth and the nasal cavity. Everything past the vocal cords is generally referred to as the *vocal tract*. The first action generates the sound, which is then modulated into speech as it traverses through the vocal tract.

We will often be talking about filters in the coming chapters. We will try to describe filters more precisely at that time. For our purposes at present, a filter is a system that has an input and an output, and a rule for converting the input to the output, which we will call the *transfer function*. If we think of speech as the output of a filter, the sound generated by the air rushing past the vocal cords can be viewed as the input, while the rule for converting the input to the output is governed by the shape and physics of the vocal tract.

The output depends on the input and the transfer function. Let's look at each in turn. There are several different forms of input that can be generated by different conformations of the vocal cords and the associated cartilages. If the vocal cords are stretched shut and we force air through, the vocal cords vibrate, providing a periodic input. If a small aperture is left open, the input resembles white noise. By opening an aperture at different locations along the vocal cords, we can produce a white-noise–like input with certain dominant frequencies that depend on the location of the opening. The vocal tract can be modeled as a series of tubes of unequal diameter. If we now examine how an acoustic wave travels through this series of tubes, we find that the mathematical model that best describes this process is an autoregressive model. We will often encounter the autoregressive model when we discuss speech compression algorithms.

8.7 Summary

In this chapter we have looked at a variety of topics that will be useful to us when we study various lossy compression techniques, including distortion and its measurement, some new concepts from information theory, average mutual information and its connection to the rate of a compression scheme, and the rate distortion function. We have also briefly looked at some of the properties of the human visual system and the auditory system—most importantly, visual and auditory masking. The masking phenomena allow us to incur distortion in such a way that the distortion is not perceptible to the human observer. We also presented a model for speech production.

Further Reading

There are a number of excellent books available that delve more deeply in the area of information theory:

1. *Information Theory*, by R.B. Ash [15].
2. *Information Transmission*, by R.M. Fano [16].
3. *Information Theory and Reliable Communication*, by R.G. Gallagher [11].
4. *Entropy and Information Theory*, by R.M. Gray [17].
5. *Elements of Information Theory*, by T.M. Cover and J.A. Thomas [3].
6. *The Theory of Information and Coding*, by R.J. McEliece [6].

The subject of rate distortion theory is discussed in very clear terms in *Rate Distortion Theory*, by T. Berger [4].

For an introduction to the concepts behind speech perception, see *Voice and Speech Processing*, by T. Parsons [105].

8.8 Projects and Problems

1. Although SNR is a widely used measure of distortion, it often does not correlate with perceptual quality. In order to see this we conduct the following experiment. Using one of the images provided, generate two "reconstructed" images. For one of the reconstructions add a value of 10 to each pixel. For the other reconstruction, randomly add either $+10$ or -10 to each pixel.

   (a) What is the SNR for each of the reconstructions? Do the relative values reflect the difference in the perceptual quality?

   (b) Devise a mathematical measure that will better reflect the difference in perceptual quality for this particular case.

2. Consider the following lossy compression scheme for binary sequences. We divide the binary sequence into blocks of size M. For each block we count the number

of 0s. If this number is greater than or equal to $M/2$, we send a 0; otherwise, we send a 1.

(a) If the sequence is random with $P(0) = 0.8$, compute the rate and distortion (use Equation (8.54)) for $M = 1, 2, 4, 8, 16$. Compare your results with the rate distortion function for binary sources.

(b) Repeat assuming that the output of the encoder is encoded at a rate equal to the entropy of the output.

3. Write a program to implement the compression scheme described in the previous problem.

(a) Generate a random binary sequence with $P(0) = 0.8$, and compare your simulation results with the analytical results.

(b) Generate a binary first-order Markov sequence with $P(0|0) = 0.9$, and $P(1|1) = 0.9$. Encode it using your program. Discuss and comment on your results.

4. Show that

$$H(X_d|Y_d) = -\sum_{j=-\infty}^{\infty}\sum_{i=-\infty}^{\infty} f_{X|Y}(x_i|y_j)f_Y(y_j)\Delta\Delta \log f_{X|Y}(x_i|y_j) - \log\Delta. \tag{8.95}$$

5. For two random variables X and Y, show that

$$H(X|Y) \leq H(X)$$

with equality if X is independent of Y.
Hint: $E[\log(f(x))] \leq \log\{E[f(x)]\}$ (Jensen's inequality).

6. Given two random variables X and Y, show that $I(X; Y) = I(Y; X)$.

7. For a binary source with $P(0) = p$, $P(X = 0|Y = 1) = P(X = 1|Y = 0) = D$, and distortion measure

$$d(x_i, y_j) = x_i \oplus y_j,$$

show that

$$I(X; Y) = H_b(p) - H_b(D). \tag{8.96}$$

8. Find the autocorrelation function in terms of the model coefficients and σ_ϵ^2 for

(a) an AR(1) process,

(b) an MA(1) process, and

(c) an AR(2) process.

9

Scalar Quantization

9.1 Overview

In this chapter we begin our study of quantization, one of the simplest and most general ideas in lossy compression. We will look at scalar quantization in this chapter and continue with vector quantization in the next chapter. First, the general quantization problem is stated, then various solutions are examined, starting with the simpler solutions, which require the most assumptions, and proceeding to more complex solutions that require fewer assumptions. We describe uniform quantization with fixed-length codewords, first assuming a uniform source, then a source with a known probability density function (*pdf*) that is not necessarily uniform, and finally a source with unknown or changing statistics. We then look at *pdf*-optimized nonuniform quantization, followed by companded quantization. Finally, we return to the more general statement of the quantizer design problem and study entropy-coded quantization.

9.2 Introduction

In many lossy compression applications we are required to represent each source output using one of a small number of codewords. The number of possible distinct source output values is generally much larger than the number of codewords available to represent them. The process of representing a large—possibly infinite—set of values with a much smaller set is called *quantization*.

Consider a source that generates numbers between -10.0 and 10.0. A simple quantization scheme would be to represent each output of the source with the integer value closest to it. (If the source output is equally close to two integers, we will randomly pick one of them.) For example, if the source output is 2.47, we would represent it as 2, and if the source output is 3.1415926, we would represent it as 3.

This approach reduces the size of the alphabet required to represent the source output; the infinite number of values between -10.0 and 10.0 are represented with a set that contains only 21 values ($\{-10, \ldots, 0, \ldots, 10\}$). At the same time we have also forever lost the original value of the source output. If we are told that the reconstruction value is 3, we cannot tell whether the source output was 2.95, 3.16, 3.057932, or any other of an infinite set of values. In other words, we have lost some information. This loss of information is the reason for the use of the word "lossy" in many lossy compression schemes.

The set of inputs and outputs of a quantizer can be scalars or vectors. If they are scalars, we call the quantizers *scalar quantizers*. If they are vectors, we call the quantizers *vector quantizers*. We will study scalar quantizers in this chapter and vector quantizers in Chapter 10.

9.3 The Quantization Problem

Quantization is a very simple process. However, the design of the quantizer has a significant impact on the amount of compression obtained and loss incurred in a lossy compression scheme. Therefore, we will devote a lot of attention to issues related to the design of quantizers.

In practice, the quantizer consists of two mappings: an encoder mapping and a decoder mapping. The encoder divides the range of values that the source generates into a number of intervals. Each interval is represented by a distinct codeword. The encoder represents all the source outputs that fall into a particular interval by the codeword representing that interval. As there could be many—possibly infinitely many—distinct sample values that can fall in any given interval, the encoder mapping is irreversible. Knowing the code only tells us the interval to which the sample value belongs. It does not tell us which of the many values in the interval is the actual sample value. When the sample value comes from an analog source, the encoder is called an analog-to-digital (A/D) converter.

The encoder mapping for a quantizer with eight reconstruction values is shown in Figure 9.1. For this encoder, all samples with values between -1 and 0 would be assigned the code 011. All values between 0 and 1.0 would be assigned the codeword 100, and so on. On the two boundaries, all inputs with values greater than 3 would be assigned the code 111, and all inputs with values less than -3.0 would be assigned the code 000. Thus, any input

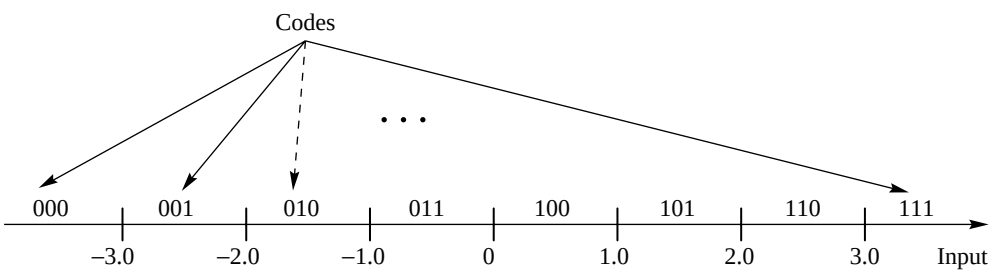

FIGURE 9.1 Mapping for a 3-bit encoder.

Input Codes	Output
000	−3.5
001	−2.5
010	−1.5
011	−0.5
100	0.5
101	1.5
110	2.5
111	3.5

FIGURE 9.2 Mapping for a 3-bit D/A converter.

that we receive will be assigned a codeword depending on the interval in which it falls. As we are using 3 bits to represent each value, we refer to this quantizer as a 3-bit quantizer.

For every codeword generated by the encoder, the decoder generates a reconstruction value. Because a codeword represents an entire interval, and there is no way of knowing which value in the interval was actually generated by the source, the decoder puts out a value that, in some sense, best represents all the values in the interval. Later, we will see how to use information we may have about the distribution of the input in the interval to obtain a representative value. For now, we simply use the midpoint of the interval as the representative value generated by the decoder. If the reconstruction is analog, the decoder is often referred to as a digital-to-analog (D/A) converter. A decoder mapping corresponding to the 3-bit encoder shown in Figure 9.1 is shown in Figure 9.2.

Example 9.3.1:

Suppose a sinusoid $4\cos(2\pi t)$ was sampled every 0.05 second. The sample was digitized using the A/D mapping shown in Figure 9.1 and reconstructed using the D/A mapping shown in Figure 9.2. The first few inputs, codewords, and reconstruction values are given in Table 9.1. Notice the first two samples in Table 9.1. Although the two input values are distinct, they both fall into the same interval in the quantizer. The encoder, therefore, represents both inputs with the same codeword, which in turn leads to identical reconstruction values.

TABLE 9.1 Digitizing a sine wave.

t	$4\cos(2\pi t)$	A/D Output	D/A Output	Error
0.05	3.804	111	3.5	0.304
0.10	3.236	111	3.5	−0.264
0.15	2.351	110	2.5	−0.149
0.20	1.236	101	1.5	−0.264

♦

Construction of the intervals (their location, etc.) can be viewed as part of the design of the encoder. Selection of reconstruction values is part of the design of the decoder. However, the fidelity of the reconstruction depends on both the intervals and the reconstruction values. Therefore, when designing or analyzing encoders and decoders, it is reasonable to view them as a pair. We call this encoder-decoder pair a *quantizer*. The quantizer mapping for the 3-bit encoder-decoder pair shown in Figures 9.1 and 9.2 can be represented by the input-output map shown in Figure 9.3. The quantizer accepts sample values, and depending on the interval in which the sample values fall, it provides an output codeword and a representation value. Using the map of Figure 9.3, we can see that an input to the quantizer of 1.7 will result in an output of 1.5, and an input of -0.3 will result in an output of -0.5.

From Figures 9.1–9.3 we can see that we need to know how to divide the input range into intervals, assign binary codes to these intervals, and find representation or output values for these intervals in order to specify a quantizer. We need to do all of this while satisfying distortion and rate criteria. In this chapter we will define distortion to be the average squared difference between the quantizer input and output. We call this the mean squared quantization error (msqe) and denote it by σ_q^2. The rate of the quantizer is the average number of bits

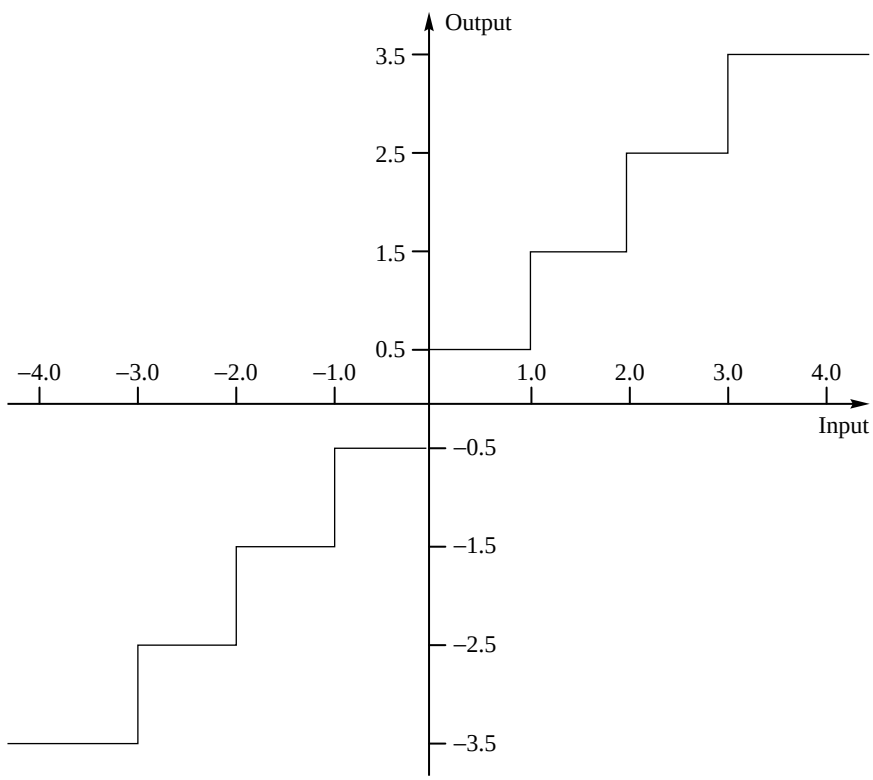

FIGURE 9. 3 Quantizer input-output map.

required to represent a single quantizer output. We would like to get the lowest distortion for a given rate, or the lowest rate for a given distortion.

Let us pose the design problem in precise terms. Suppose we have an input modeled by a random variable X with *pdf* $f_X(x)$. If we wished to quantize this source using a quantizer with M intervals, we would have to specify $M+1$ endpoints for the intervals, and a representative value for each of the M intervals. The endpoints of the intervals are known as *decision boundaries*, while the representative values are called *reconstruction levels*. We will often model discrete sources with continuous distributions. For example, the difference between neighboring pixels is often modeled using a Laplacian distribution even though the differences can only take on a limited number of discrete values. Discrete processes are modeled with continuous distributions because it can simplify the design process considerably, and the resulting designs perform well in spite of the incorrect assumption. Several of the continuous distributions used to model source outputs are unbounded—that is, the range of values is infinite. In these cases, the first and last endpoints are generally chosen to be $\pm\infty$.

Let us denote the decision boundaries by $\{b_i\}_{i=0}^{M}$, the reconstruction levels by $\{y_i\}_{i=1}^{M}$, and the quantization operation by $Q(\cdot)$. Then

$$Q(x) = y_i \quad \text{iff} \quad b_{i-1} < x \leq b_i. \tag{9.1}$$

The mean squared quantization error is then given by

$$\sigma_q^2 = \int_{-\infty}^{\infty} (x - Q(x))^2 f_X(x)dx \tag{9.2}$$

$$= \sum_{i=1}^{M} \int_{b_{i-1}}^{b_i} (x - y_i)^2 f_X(x)dx. \tag{9.3}$$

The difference between the quantizer input x and output $y = Q(x)$, besides being referred to as the quantization error, is also called the *quantizer distortion* or *quantization noise*. But the word "noise" is somewhat of a misnomer. Generally, when we talk about noise we mean a process external to the source process. Because of the manner in which the quantization error is generated, it is dependent on the source process and, therefore, cannot be regarded as external to the source process. One reason for the use of the word "noise" in this context is that from time to time we will find it useful to model the quantization process as an additive noise process as shown in Figure 9.4.

If we use fixed-length codewords to represent the quantizer output, then the size of the output alphabet immediately specifies the rate. If the number of quantizer outputs is M, then the rate is given by

$$R = \lceil \log_2 M \rceil. \tag{9.4}$$

For example, if $M = 8$, then $R = 3$. In this case, we can pose the quantizer design problem as follows:

> Given an input *pdf* $f_x(x)$ and the number of levels M in the quantizer, find the decision boundaries $\{b_i\}$ and the reconstruction levels $\{y_i\}$ so as to minimize the mean squared quantization error given by Equation (9.3).

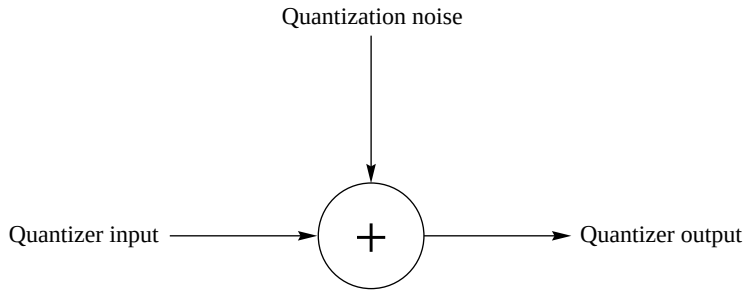

FIGURE 9.4 Additive noise model of a quantizer.

TABLE 9.2 Codeword assignment for an eight-level quantizer.

y_1	1110
y_2	1100
y_3	100
y_4	00
y_5	01
y_6	101
y_7	1101
y_8	1111

However, if we are allowed to use variable-length codes, such as Huffman codes or arithmetic codes, along with the size of the alphabet, the selection of the decision boundaries will also affect the rate of the quantizer. Consider the codeword assignment for the output of an eight-level quantizer shown in Table 9.2.

According to this codeword assignment, if the output y_4 occurs, we use 2 bits to encode it, while if the output y_1 occurs, we need 4 bits to encode it. Obviously, the rate will depend on how often we have to encode y_4 versus how often we have to encode y_1. In other words, the rate will depend on the probability of occurrence of the outputs. If l_i is the length of the codeword corresponding to the output y_i, and $P(y_i)$ is the probability of occurrence of y_i, then the rate is given by

$$R = \sum_{i=1}^{M} l_i P(y_i). \tag{9.5}$$

However, the probabilities $\{P(y_i)\}$ depend on the decision boundaries $\{b_i\}$. For example, the probability of y_i occurring is given by

$$P(y_i) = \int_{b_{i-1}}^{b_i} f_X(x)\,dx.$$

Therefore, the rate R is a function of the decision boundaries and is given by the expression

$$R = \sum_{i=1}^{M} l_i \int_{b_{i-1}}^{b_i} f_X(x)dx. \tag{9.6}$$

From this discussion and Equations (9.3) and (9.6), we see that for a given source input, the partitions we select and the representation for those partitions will determine the distortion incurred during the quantization process. The partitions we select and the binary codes for the partitions will determine the rate for the quantizer. Thus, the problem of finding the optimum partitions, codes, and representation levels are all linked. In light of this information, we can restate our problem statement:

Given a distortion constraint

$$\sigma_q^2 \leq D^* \tag{9.7}$$

find the decision boundaries, reconstruction levels, and binary codes that minimize the rate given by Equation (9.6), while satisfying Equation (9.7).

Or, given a rate constraint

$$R \leq R^* \tag{9.8}$$

find the decision boundaries, reconstruction levels, and binary codes that minimize the distortion given by Equation (9.3), while satisfying Equation (9.8).

This problem statement of quantizer design, while more general than our initial statement, is substantially more complex. Fortunately, in practice there are situations in which we can simplify the problem. We often use fixed-length codewords to encode the quantizer output. In this case, the rate is simply the number of bits used to encode each output, and we can use our initial statement of the quantizer design problem. We start our study of quantizer design by looking at this simpler version of the problem, and later use what we have learned in this process to attack the more complex version.

9.4 Uniform Quantizer

The simplest type of quantizer is the uniform quantizer. All intervals are the same size in the uniform quantizer, except possibly for the two outer intervals. In other words, the decision boundaries are spaced evenly. The reconstruction values are also spaced evenly, with the same spacing as the decision boundaries; in the inner intervals, they are the midpoints of the intervals. This constant spacing is usually referred to as the step size and is denoted by Δ. The quantizer shown in Figure 9.3 is a uniform quantizer with $\Delta = 1$. It does not have zero as one of its representation levels. Such a quantizer is called a *midrise quantizer*. An alternative uniform quantizer could be the one shown in Figure 9.5. This is called a *midtread quantizer*. As the midtread quantizer has zero as one of its output levels, it is especially useful in situations where it is important that the zero value be represented—for example,

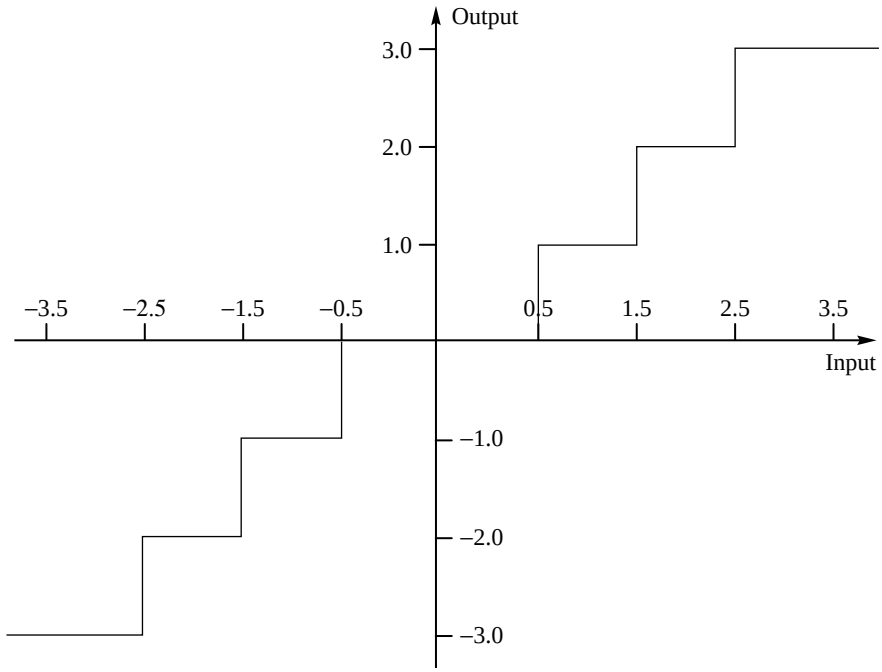

FIGURE 9. 5 A midtread quantizer.

control systems in which it is important to represent a zero value accurately, and audio coding schemes in which we need to represent silence periods. Notice that the midtread quantizer has only seven intervals or levels. That means that if we were using a fixed-length 3-bit code, we would have one codeword left over.

Usually, we use a midrise quantizer if the number of levels is even and a midtread quantizer if the number of levels is odd. For the remainder of this chapter, unless we specifically mention otherwise, we will assume that we are dealing with midrise quantizers. We will also generally assume that the input distribution is symmetric around the origin and the quantizer is also symmetric. (The optimal minimum mean squared error quantizer for a symmetric distribution need not be symmetric [106].) Given all these assumptions, the design of a uniform quantizer consists of finding the step size Δ that minimizes the distortion for a given input process and number of decision levels.

Uniform Quantization of a Uniformly Distributed Source

We start our study of quantizer design with the simplest of all cases: design of a uniform quantizer for a uniformly distributed source. Suppose we want to design an M-level uniform quantizer for an input that is uniformly distributed in the interval $[-X_{\max}, X_{\max}]$. This means

we need to divide the $[-X_{\max}, X_{\max}]$ interval into M equally sized intervals. In this case, the step size Δ is given by

$$\Delta = \frac{2X_{\max}}{M}. \tag{9.9}$$

The distortion in this case becomes

$$\sigma_q^2 = 2\sum_{i=1}^{\frac{M}{2}} \int_{(i-1)\Delta}^{i\Delta} \left(x - \frac{2i-1}{2}\Delta\right)^2 \frac{1}{2X_{\max}} dx. \tag{9.10}$$

If we evaluate this integral (after some suffering), we find that the msqe is $\Delta^2/12$.

The same result can be more easily obtained if we examine the behavior of the quantization error q given by

$$q = x - Q(x). \tag{9.11}$$

In Figure 9.6 we plot the quantization error versus the input signal for an eight-level uniform quantizer, with an input that lies in the interval $[-X_{\max}, X_{\max}]$. Notice that the quantization error lies in the interval $[-\frac{\Delta}{2}, \frac{\Delta}{2}]$. As the input is uniform, it is not difficult to establish that the quantization error is also uniform over this interval. Thus, the mean squared quantization error is the second moment of a random variable uniformly distributed in the interval $[-\frac{\Delta}{2}, \frac{\Delta}{2}]$:

$$\sigma_q^2 = \frac{1}{\Delta}\int_{-\frac{\Delta}{2}}^{\frac{\Delta}{2}} q^2 dq \tag{9.12}$$

$$= \frac{\Delta^2}{12}. \tag{9.13}$$

Let us also calculate the signal-to-noise ratio for this case. The signal variance σ_s^2 for a uniform random variable, which takes on values in the interval $[-X_{\max}, X_{\max}]$, is $\frac{(2X_{\max})^2}{12}$.

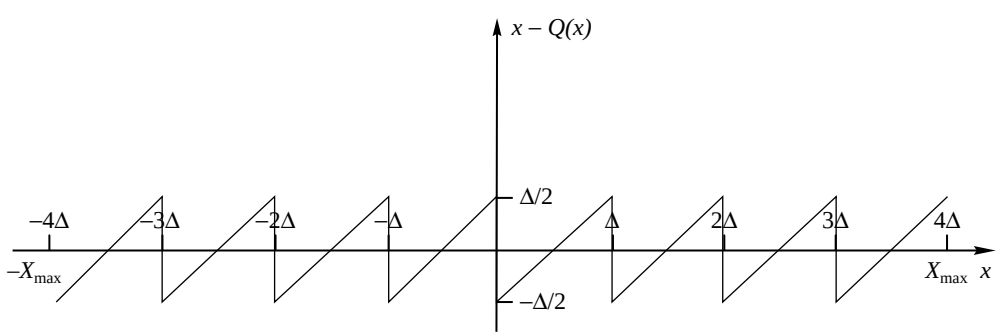

FIGURE 9. 6 **Quantization error for a uniform midrise quantizer with a uniformly distributed input.**

The value of the step size Δ is related to $X_{\max}$ and the number of levels M by

$$\Delta = \frac{2X_{\max}}{M}.$$

For the case where we use a fixed-length code, with each codeword being made up of n bits, the number of codewords or the number of reconstruction levels M is 2^n. Combining all this, we have

$$\begin{aligned}
\mathrm{SNR(dB)} &= 10\log_{10}\left(\frac{\sigma_s^2}{\sigma_q^2}\right) && (9.14)\\
&= 10\log_{10}\left(\frac{(2X_{\max})^2}{12}\cdot\frac{12}{\Delta^2}\right) && (9.15)\\
&= 10\log_{10}\left(\frac{(2X_{\max})^2}{12}\frac{12}{(\frac{2X_{\max}}{M})^2}\right) && (9.16)\\
&= 10\log_{10}(M^2)\\
&= 20\log_{10}(2^n)\\
&= 6.02n\,\mathrm{dB}. && (9.17)
\end{aligned}$$

This equation says that for every additional bit in the quantizer, we get an increase in the signal-to-noise ratio of 6.02 dB. This is a well-known result and is often used to get an indication of the maximum gain available if we increase the rate. However, remember that we obtained this result under some assumptions about the input. If the assumptions are not true, this result will not hold true either.

Example 9.4.1: Image compression

A probability model for the variations of pixels in an image is almost impossible to obtain because of the great variety of images available. A common approach is to declare the pixel values to be uniformly distributed between 0 and $2^b - 1$, where b is the number of bits per pixel. For most of the images we deal with, the number of bits per pixel is 8; therefore, the pixel values would be assumed to vary uniformly between 0 and 255. Let us quantize our test image Sena using a uniform quantizer.

If we wanted to use only 1 bit per pixel, we would divide the range [0, 255] into two intervals, [0, 127] and [128, 255]. The first interval would be represented by the value 64, the midpoint of the first interval; the pixels in the second interval would be represented by the pixel value 196, the midpoint of the second interval. In other words, the boundary values are $\{0, 128, 255\}$, while the reconstruction values are $\{64, 196\}$. The quantized image is shown in Figure 9.7. As expected, almost all the details in the image have disappeared. If we were to use a 2-bit quantizer, with boundary values $\{0, 64, 128, 196, 255\}$ and reconstruction levels $\{32, 96, 160, 224\}$, we get considerably more detail. The level of detail increases as the use of bits increases until at 6 bits per pixel, the reconstructed image is indistinguishable from the original, at least to a casual observer. The 1-, 2-, and 3-bit images are shown in Figure 9.7.

FIGURE 9.7 **Top left: original Sena image; top right: 1 bit/pixel image; bottom left: 2 bits/pixel; bottorm right: 3 bits/pixel.**

Looking at the lower-rate images, we notice a couple of things. First, the lower-rate images are darker than the original, and the lowest-rate reconstructions are the darkest. The reason for this is that the quantization process usually results in scaling down of the dynamic range of the input. For example, in the 1-bit-per-pixel reproduction, the highest pixel value is 196, as opposed to 255 for the original image. As higher gray values represent lighter shades, there is a corresponding darkening of the reconstruction. The other thing to notice in the low-rate reconstruction is that wherever there were smooth changes in gray values there are now abrupt transitions. This is especially evident in the face and neck area, where gradual shading has been transformed to blotchy regions of constant values. This is because a range of values is being mapped to the same value, as was the case for the first two samples of the sinusoid in Example 9.3.1. For obvious reasons, this effect is called *contouring*. The perceptual effect of contouring can be reduced by a procedure called *dithering* [107]. ♦

Uniform Quantization of Nonuniform Sources

Quite often the sources we deal with do not have a uniform distribution; however, we still want the simplicity of a uniform quantizer. In these cases, even if the sources are bounded, simply dividing the range of the input by the number of quantization levels does not produce a very good design.

Example 9.4.2:

Suppose our input fell within the interval $[-1, 1]$ with probability 0.95, and fell in the intervals $[-100, 1)$, $(1, 100]$ with probability 0.05. Suppose we wanted to design an eight-level uniform quantizer. If we followed the procedure of the previous section, the step size would be 25. This means that inputs in the $[-1, 0)$ interval would be represented by the value -12.5, and inputs in the interval $[0, 1)$ would be represented by the value 12.5. The maximum quantization error that can be incurred is 12.5. However, at least 95% of the time, the *minimum* error that will be incurred is 11.5. Obviously, this is not a very good design. A much better approach would be to use a smaller step size, which would result in better representation of the values in the $[-1, 1]$ interval, even if it meant a larger maximum error. Suppose we pick a step size of 0.3. In this case, the maximum quantization error goes from 12.5 to 98.95. However, 95% of the time the quantization error will be less than 0.15. Therefore, the average distortion, or msqe, for this quantizer would be substantially less than the msqe for the first quantizer. ♦

We can see that when the distribution is no longer uniform, it is not a good idea to obtain the step size by simply dividing the range of the input by the number of levels. This approach becomes totally impractical when we model our sources with distributions that are unbounded, such as the Gaussian distribution. Therefore, we include the *pdf* of the source in the design process.

Our objective is to find the step size that, for a given value of M, will minimize the distortion. The simplest way to do this is to write the distortion as a function of the step size, and then minimize this function. An expression for the distortion, or msqe, for an M-level uniform quantizer as a function of the step size can be found by replacing the b_is and y_is in Equation (9.3) with functions of Δ. As we are dealing with a symmetric condition, we need only compute the distortion for positive values of x; the distortion for negative values of x will be the same.

From Figure 9.8, we see that the decision boundaries are integral multiples of Δ, and the representation level for the interval $[(k-1)\Delta, k\Delta)$ is simply $\frac{2k-1}{2}\Delta$. Therefore, the expression for msqe becomes

$$\sigma_q^2 = 2\sum_{i=1}^{\frac{M}{2}-1}\int_{(i-1)\Delta}^{i\Delta}\left(x-\frac{2i-1}{2}\Delta\right)^2 f_X(x)dx$$
$$+2\int_{\left(\frac{M}{2}-1\right)\Delta}^{\infty}\left(x-\frac{M-1}{2}\Delta\right)^2 f_X(x)dx. \tag{9.18}$$

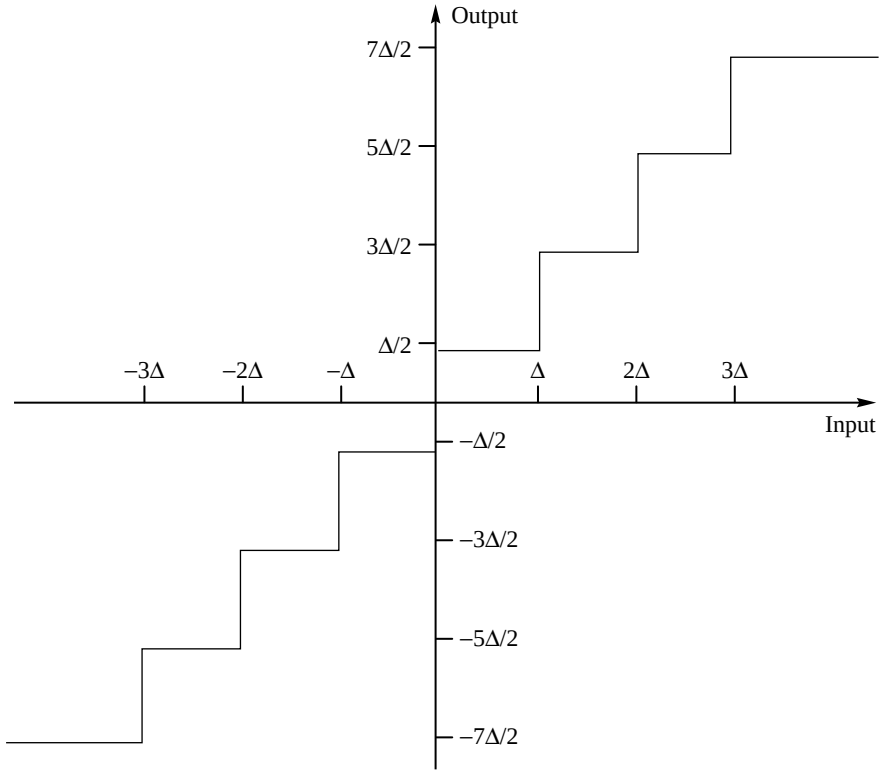

FIGURE 9.8 A uniform midrise quantizer.

To find the optimal value of Δ, we simply take a derivative of this equation and set it equal to zero [108] (see Problem 1).

$$\frac{\delta\sigma_q^2}{\delta\Delta} = -\sum_{i=1}^{\frac{M}{2}-1}(2i-1)\int_{(i-1)\Delta}^{i\Delta}(x-\frac{2i-1}{2}\Delta)f_X(x)dx$$
$$-(M-1)\int_{(\frac{M}{2}-1)\Delta}^{\infty}\left(x-\frac{M-1}{2}\Delta\right)f_X(x)dx = 0. \qquad (9.19)$$

This is a rather messy-looking expression, but given the *pdf* $f_X(x)$, it is easy to solve using any one of a number of numerical techniques (see Problem 2). In Table 9.3, we list step sizes found by solving (9.19) for nine different alphabet sizes and three different distributions.

Before we discuss the results in Table 9.3, let's take a look at the quantization noise for the case of nonuniform sources. Nonuniform sources are often modeled by *pdf*s with unbounded support. That is, there is a nonzero probability of getting an unbounded input. In practical situations, we are not going to get inputs that are unbounded, but often it is very convenient to model the source process with an unbounded distribution. The classic example of this is measurement error, which is often modeled as having a Gaussian distribution,

TABLE 9.3 Optimum step size and SNR for uniform quantizers for different distributions and alphabet sizes [108, 109].

Alphabet	Uniform		Gaussian		Laplacian	
Size	Step Size	SNR	Step Size	SNR	Step Size	SNR
2	1.732	6.02	1.596	4.40	1.414	3.00
4	0.866	12.04	0.9957	9.24	1.0873	7.05
6	0.577	15.58	0.7334	12.18	0.8707	9.56
8	0.433	18.06	0.5860	14.27	0.7309	11.39
10	0.346	20.02	0.4908	15.90	0.6334	12.81
12	0.289	21.60	0.4238	17.25	0.5613	13.98
14	0.247	22.94	0.3739	18.37	0.5055	14.98
16	0.217	24.08	0.3352	19.36	0.4609	15.84
32	0.108	30.10	0.1881	24.56	0.2799	20.46

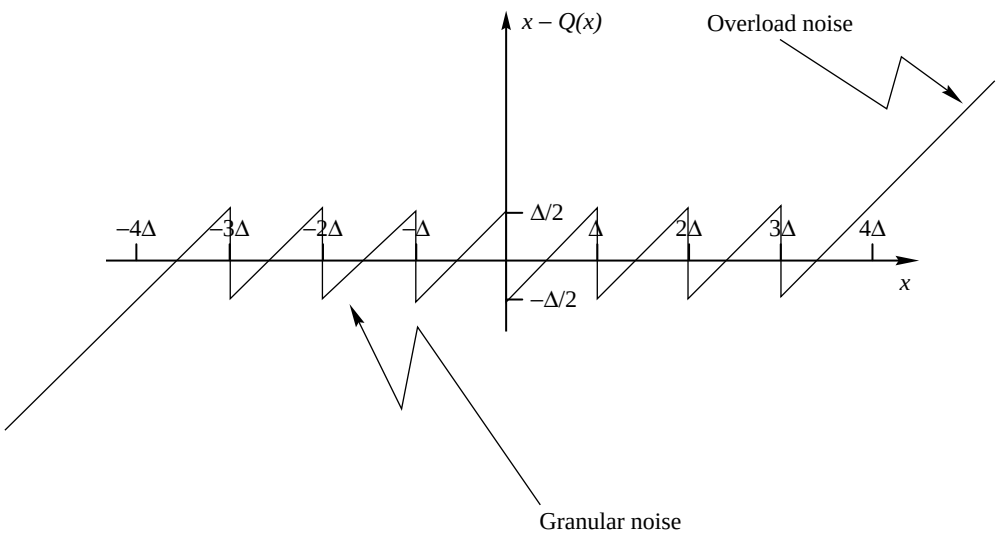

FIGURE 9.9 Quantization error for a uniform midrise quantizer.

even when the measurement error is known to be bounded. If the input is unbounded, the quantization error is no longer bounded either. The quantization error as a function of input is shown in Figure 9.9. We can see that in the inner intervals the error is still bounded by $\frac{\Delta}{2}$; however, the quantization error in the outer intervals is unbounded. These two types of quantization errors are given different names. The bounded error is called *granular error* or *granular noise*, while the unbounded error is called *overload error* or *overload noise*. In the expression for the msqe in Equation (9.18), the first term represents the granular noise, while the second term represents the overload noise. The probability that the input will fall into the overload region is called the *overload probability* (Figure 9.10).

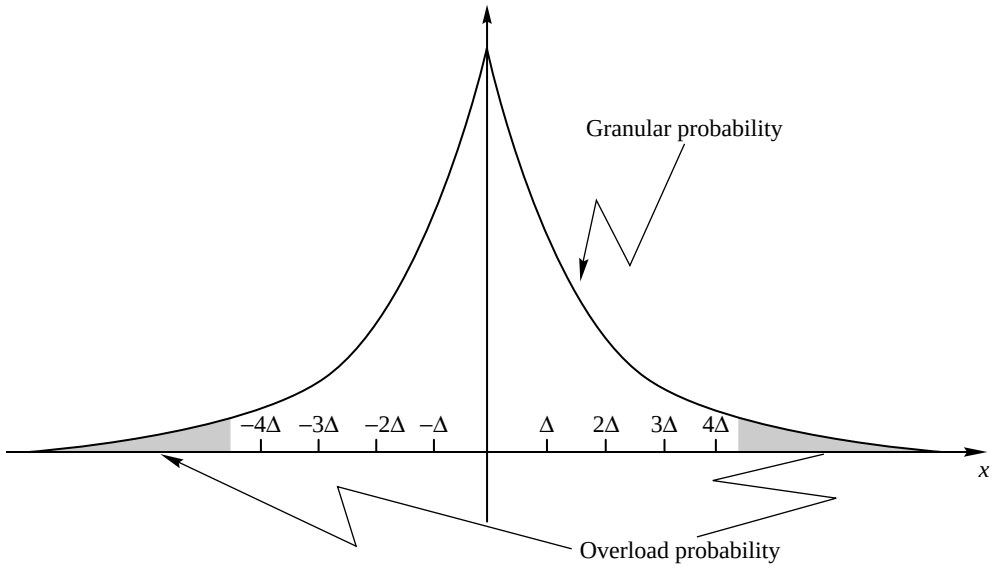

FIGURE 9. 10 Overload and granular regions for a 3-bit uniform quantizer.

The nonuniform sources we deal with have probability density functions that are generally peaked at zero and decay as we move away from the origin. Therefore, the overload probability is generally much smaller than the probability of the input falling in the granular region. As we see from Equation (9.19), an increase in the size of the step size Δ will result in an increase in the value of $\left(\frac{M}{2}-1\right)\Delta$, which in turn will result in a decrease in the overload probability and the second term in Equation (9.19). However, an increase in the step size Δ will also increase the granular noise, which is the first term in Equation (9.19). The design process for the uniform quantizer is a balancing of these two effects. An important parameter that describes this trade-off is the loading factor f_l, defined as the ratio of the maximum value the input can take in the granular region to the standard deviation. A common value of the loading factor is 4. This is also referred to as 4σ *loading*.

Recall that when quantizing an input with a uniform distribution, the SNR and bit rate are related by Equation (9.17), which says that for each bit increase in the rate there is an increase of 6.02 dB in the SNR. In Table 9.3, along with the step sizes, we have also listed the SNR obtained when a million input values with the appropriate *pdf* are quantized using the indicated quantizer.

From this table, we can see that, although the SNR for the uniform distribution follows the rule of a 6.02 dB increase in the signal-to-noise ratio for each additional bit, this is not true for the other distributions. Remember that we made some assumptions when we obtained the $6.02n$ rule that are only valid for the uniform distribution. Notice that the more peaked a distribution is (that is, the further away from uniform it is), the more it seems to vary from the 6.02 dB rule.

We also said that the selection of Δ is a balance between the overload and granular errors. The Laplacian distribution has more of its probability mass away from the origin in

its tails than the Gaussian distribution. This means that for the same step size and number of levels there is a higher probability of being in the overload region if the input has a Laplacian distribution than if the input has a Gaussian distribution. The uniform distribution is the extreme case, where the overload probability is zero. For the same number of levels, if we increase the step size, the size of the overload region (and hence the overload probability) is reduced at the expense of granular noise. Therefore, for a given number of levels, if we were picking the step size to balance the effects of the granular and overload noise, distributions that have heavier tails will tend to have larger step sizes. This effect can be seen in Table 9.3. For example, for eight levels the step size for the uniform quantizer is 0.433. The step size for the Gaussian quantizer is larger (0.586), while the step size for the Laplacian quantizer is larger still (0.7309).

Mismatch Effects

We have seen that for a result to hold, the assumptions we used to obtain the result have to hold. When we obtain the optimum step size for a particular uniform quantizer using Equation (9.19), we make some assumptions about the statistics of the source. We assume a certain distribution and certain parameters of the distribution. What happens when our assumptions do not hold? Let's try to answer this question empirically.

We will look at two types of mismatches. The first is when the assumed distribution type matches the actual distribution type, but the variance of the input is different from the assumed variance. The second mismatch is when the actual distribution type is different from the distribution type assumed when obtaining the value of the step size. Throughout our discussion, we will assume that the mean of the input distribution is zero.

In Figure 9.11, we have plotted the signal-to-noise ratio as a function of the ratio of the actual to assumed variance of a 4-bit Gaussian uniform quantizer, with a Gaussian

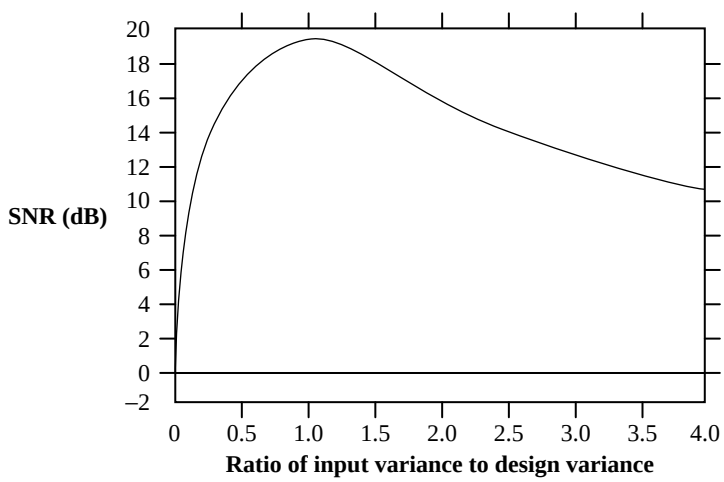

FIGURE 9.11 **Effect of variance mismatch on the performance of a 4-bit uniform quantizer.**

input. (To see the effect under different conditions, see Problem 5.) Remember that for a distribution with zero mean, the variance is given by $\sigma_x^2 = E[X^2]$, which is also a measure of the power in the signal X. As we can see from the figure, the signal-to-noise ratio is maximum when the input signal variance matches the variance assumed when designing the quantizer. From the plot we also see that there is an asymmetry; the SNR is considerably worse when the input variance is lower than the assumed variance. This is because the SNR is a ratio of the input variance and the mean squared quantization error. When the input variance is smaller than the assumed variance, the mean squared quantization error actually drops because there is less overload noise. However, because the input variance is low, the ratio is small. When the input variance is higher than the assumed variance, the msqe increases substantially, but because the input power is also increasing, the ratio does not decrease as dramatically. To see this more clearly, we have plotted the mean squared error versus the signal variance separately in Figure 9.12. We can see from these figures that the decrease in signal-to-noise ratio does not always correlate directly with an increase in msqe.

The second kind of mismatch is where the input distribution does not match the distribution assumed when designing the quantizer. In Table 9.4 we have listed the SNR when inputs with different distributions are quantized using several different eight-level quantizers. The quantizers were designed assuming a particular input distribution.

Notice that as we go from left to right in the table, the designed step size becomes progressively larger than the "correct" step size. This is similar to the situation where the input variance is smaller than the assumed variance. As we can see when we have a mismatch that results in a smaller step size relative to the optimum step size, there is a greater drop in performance than when the quantizer step size is larger than its optimum value.

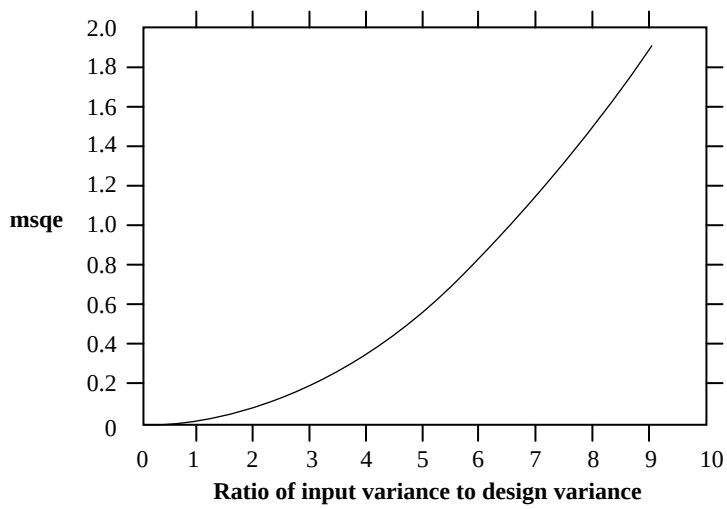

FIGURE 9.12 **The msqe as a function of variance mismatch with a 4-bit uniform quantizer.**

TABLE 9.4 Demonstration of the effect of mismatch using eight- level quantizers (dB).

Input Distribution	Uniform Quantizer	Gaussian Quantizer	Laplacian Quantizer	Gamma Quantizer
Uniform	18.06	15.56	13.29	12.41
Gaussian	12.40	14.27	13.37	12.73
Laplacian	8.80	10.79	11.39	11.28
Gamma	6.98	8.06	8.64	8.76

9.5 Adaptive Quantization

One way to deal with the mismatch problem is to adapt the quantizer to the statistics of the input. Several things might change in the input relative to the assumed statistics, including the mean, the variance, and the *pdf*. The strategy for handling each of these variations can be different, though certainly not exclusive. If more than one aspect of the input statistics changes, it is possible to combine the strategies for handling each case separately. If the mean of the input is changing with time, the best strategy is to use some form of differential encoding (discussed in some detail in Chapter 11). For changes in the other statistics, the common approach is to adapt the quantizer parameters to the input statistics.

There are two main approaches to adapting the quantizer parameters: an *off-line* or *forward adaptive* approach, and an *on-line* or *backward adaptive* approach. In forward adaptive quantization, the source output is divided into blocks of data. Each block is analyzed before quantization, and the quantizer parameters are set accordingly. The settings of the quantizer are then transmitted to the receiver as *side information*. In backward adaptive quantization, the adaptation is performed based on the quantizer output. As this is available to both transmitter and receiver, there is no need for side information.

9.5.1 Forward Adaptive Quantization

Let us first look at approaches for adapting to changes in input variance using the forward adaptive approach. This approach necessitates a delay of at least the amount of time required to process a block of data. The insertion of side information in the transmitted data stream may also require the resolution of some synchronization problems. The size of the block of data processed also affects a number of other things. If the size of the block is too large, then the adaptation process may not capture the changes taking place in the input statistics. Furthermore, large block sizes mean more delay, which may not be tolerable in certain applications. On the other hand, small block sizes mean that the side information has to be transmitted more often, which in turn means the amount of overhead per sample increases. The selection of the block size is a trade-off between the increase in side information necessitated by small block sizes and the loss of fidelity due to large block sizes (see Problem 7).

The variance estimation procedure is rather simple. At time n we use a block of N future samples to compute an estimate of the variance

$$\hat{\sigma}_q^2 = \frac{1}{N}\sum_{i=0}^{N-1} x_{n+i}^2. \tag{9.20}$$

Note that we are assuming that our input has a mean of zero. The variance information also needs to be quantized so that it can be transmitted to the receiver. Usually, the number of bits used to quantize the value of the variance is significantly larger than the number of bits used to quantize the sample values.

Example 9.5.1:

In Figure 9.13 we show a segment of speech quantized using a fixed 3-bit quantizer. The step size of the quantizer was adjusted based on the statistics of the entire sequence. The sequence was the `testm.raw` sequence from the sample data sets, consisting of about 4000 samples of a male speaker saying the word "test." The speech signal was sampled at 8000 samples per second and digitized using a 16-bit A/D.

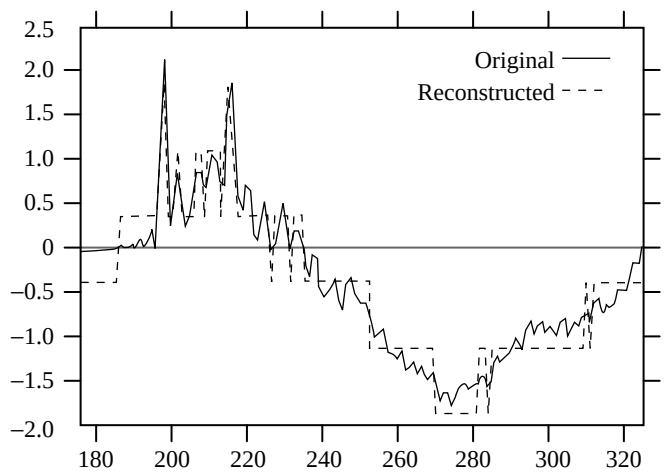

FIGURE 9. 13 Original 16-bit speech and compressed 3-bit speech sequences.

We can see from the figure that, as in the case of the example of the sinusoid earlier in this chapter, there is a considerable loss in amplitude resolution. Sample values that are close together have been quantized to the same value.

The same sequence quantized with a forward adaptive quantizer is shown in Figure 9.14. For this example, we divided the input into blocks of 128 samples. Before quantizing the samples in a block, the standard deviation for the samples in the block was obtained. This value was quantized using an 8-bit quantizer and sent to both the transmitter and receiver.

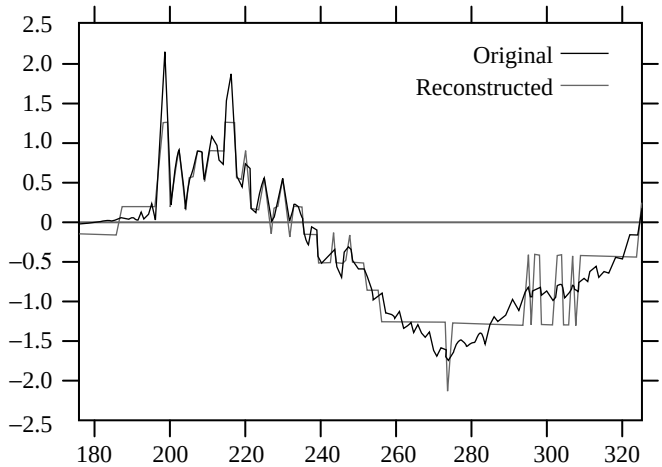

FIGURE 9. 14 **Original 16-bit speech sequence and sequence obtained using an eight-level forward adaptive quantizer.**

The samples in the block were then normalized using this value of the standard deviation. Notice that the reconstruction follows the input much more closely, though there seems to be room for improvement, especially in the latter half of the displayed samples. ♦

Example 9.5.2:

In Example 9.4.1, we used a uniform quantizer with the assumption that the input is uniformly distributed. Let us refine this source model a bit and say that while the source is uniformly distributed over different regions, the range of the input changes. In a forward adaptive quantization scheme, we would obtain the minimum and maximum values for each block of data, which would be transmitted as side information. In Figure 9.15, we see the Sena image quantized with a block size of 8×8 using 3-bit forward adaptive uniform quantization. The side information consists of the minimum and maximum values in each block, which require 8 bits each. Therefore, the overhead in this case is $\frac{16}{8\times 8}$ or 0.25 bits per pixel, which is quite small compared to the number of bits per sample used by the quantizer.

The resulting image is hardly distinguishable from the original. Certainly at higher rates, forward adaptive quantization seems to be a very good alternative. ♦

9.5.2 Backward Adaptive Quantization

In backward adaptive quantization, only the past quantized samples are available for use in adapting the quantizer. The values of the input are only known to the encoder; therefore, this information cannot be used to adapt the quantizer. How can we get information about mismatch simply by examining the output of the quantizer without knowing what the input was? If we studied the output of the quantizer for a long period of time, we could get some idea about mismatch from the distribution of output values. If the quantizer step size Δ is

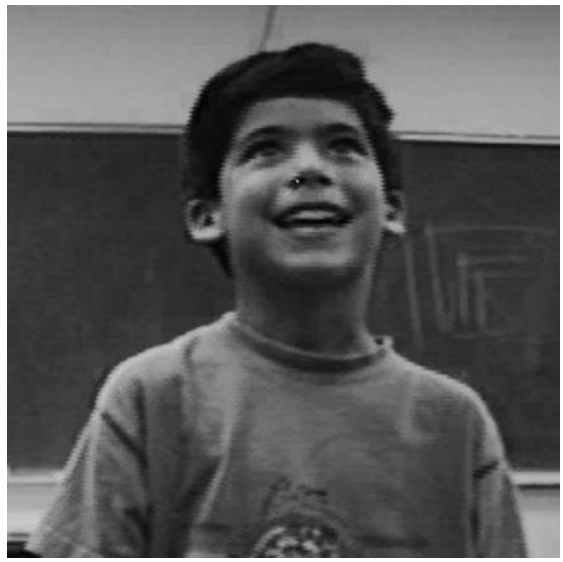

FIGURE 9.15 **Sena image quantized to 3.25 bits per pixel using forward adaptive quantization.**

well matched to the input, the probability that an input to the quantizer would land in a particular interval would be consistent with the *pdf* assumed for the input. However, if the actual *pdf* differs from the assumed *pdf*, the number of times the input falls in the different quantization intervals will be inconsistent with the assumed *pdf*. If Δ is smaller than what it should be, the input will fall in the outer levels of the quantizer an excessive number of times. On the other hand, if Δ is larger than it should be for a particular source, the input will fall in the inner levels an excessive number of times. Therefore, it seems that we should observe the output of the quantizer for a long period of time, then expand the quantizer step size if the input falls in the outer levels an excessive number of times, and contract the step size if the input falls in the inner levels an excessive number of times.

Nuggehally S. Jayant at Bell Labs showed that we did not need to observe the quantizer output over a long period of time [110]. In fact, we could adjust the quantizer step size after observing a single output. Jayant named this quantization approach "quantization with one word memory." The quantizer is better known as the *Jayant quantizer*. The idea behind the Jayant quantizer is very simple. If the input falls in the outer levels, the step size needs to be expanded, and if the input falls in the inner quantizer levels, the step size needs to be reduced. The expansions and contractions should be done in such a way that once the quantizer is matched to the input, the product of the expansions and contractions is unity.

The expansion and contraction of the step size is accomplished in the Jayant quantizer by assigning a *multiplier* M_k to each interval. If the $(n-1)$th input falls in the kth interval, the step size to be used for the nth input is obtained by multiplying the step size used for the $(n-1)$th input with M_k. The multiplier values for the inner levels in the quantizer are less than one, and the multiplier values for the outer levels of the quantizer are greater than one.

Therefore, if an input falls into the inner levels, the quantizer used to quantize the next input will have a smaller step size. Similarly, if an input falls into the outer levels, the step size will be multiplied with a value greater than one, and the next input will be quantized using a larger step size. Notice that the step size for the current input is modified based on the previous quantizer output. The previous quantizer output is available to both the transmitter and receiver, so there is no need to send any additional information to inform the receiver about the adaptation. Mathematically, the adaptation process can be represented as

$$\Delta_n = M_{l(n-1)}\Delta_{n-1} \tag{9.21}$$

where $l(n-1)$ is the quantization interval at time $n-1$.

In Figure 9.16 we show a 3-bit uniform quantizer. We have eight intervals represented by the different quantizer outputs. However, the multipliers for symmetric intervals are identical because of symmetry:

$$M_0 = M_4 \quad M_1 = M_5 \quad M_2 = M_6 \quad M_3 = M_7$$

Therefore, we only need four multipliers. To see how the adaptation proceeds, let us work through a simple example using this quantizer.

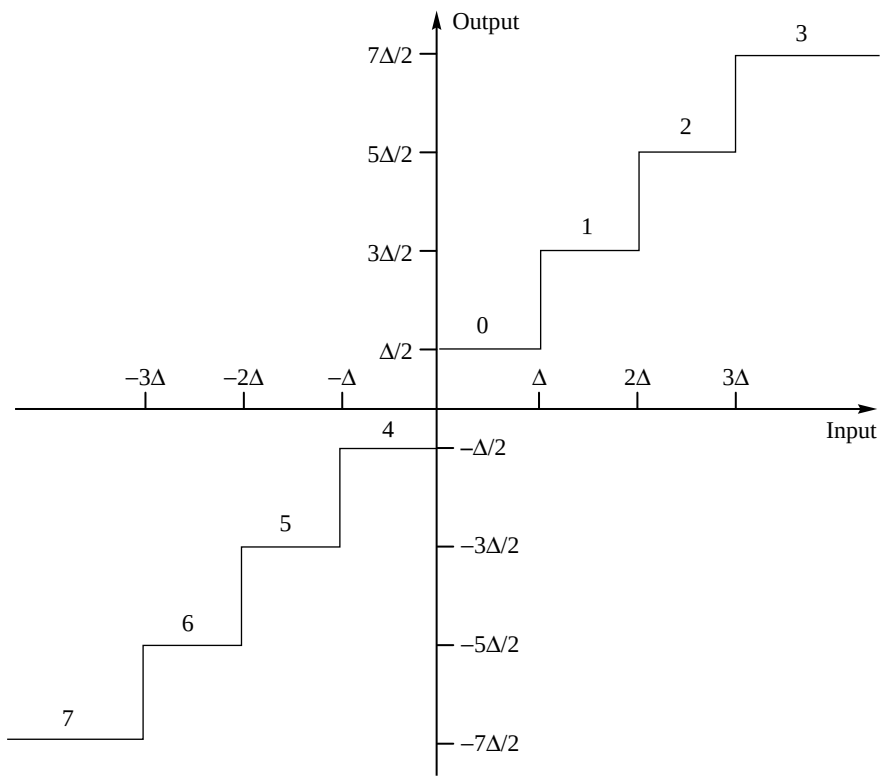

FIGURE 9.16 Output levels for the Jayant quantizer.

Example 9.5.3: Jayant quantizer

For the quantizer in Figure 9.16, suppose the multiplier values are $M_0 = M_4 = 0.8$, $M_1 = M_5 = 0.9$, $M_2 = M_6 = 1$, $M_3 = M_7 = 1.2$; the initial value of the step size, Δ_0, is 0.5; and the sequence to be quantized is $0.1, -0.2, 0.2, 0.1, -0.3, 0.1, 0.2, 0.5, 0.9, 1.5, \ldots$. When the first input is received, the quantizer step size is 0.5. Therefore, the input falls into level 0, and the output value is 0.25, resulting in an error of 0.15. As this input fell into the quantizer level 0, the new step size Δ_1 is $M_0 \times \Delta_0 = 0.8 \times 0.5 = 0.4$. The next input is -0.2, which falls into level 4. As the step size at this time is 0.4, the output is -0.2. To update, we multiply the current step size with M_4. Continuing in this fashion, we get the sequence of step sizes and outputs shown in Table 9.5.

TABLE 9.5 Operation of a Jayant quantizer.

n	Δ_n	Input	Output Level	Output	Error	Update Equation
0	0.5	0.1	0	0.25	0.15	$\Delta_1 = M_0 \times \Delta_0$
1	0.4	−0.2	4	−0.2	0.0	$\Delta_2 = M_4 \times \Delta_1$
2	0.32	0.2	0	0.16	0.04	$\Delta_3 = M_0 \times \Delta_2$
3	0.256	0.1	0	0.128	0.028	$\Delta_4 = M_0 \times \Delta_3$
4	0.2048	−0.3	5	−0.3072	−0.0072	$\Delta_5 = M_5 \times \Delta_4$
5	0.1843	0.1	0	0.0922	−0.0078	$\Delta_6 = M_0 \times \Delta_5$
6	0.1475	0.2	1	0.2212	0.0212	$\Delta_7 = M_1 \times \Delta_6$
7	0.1328	0.5	3	0.4646	−0.0354	$\Delta_8 = M_3 \times \Delta_7$
8	0.1594	0.9	3	0.5578	−0.3422	$\Delta_9 = M_3 \times \Delta_8$
9	0.1913	1.5	3	0.6696	−0.8304	$\Delta_{10} = M_3 \times \Delta_9$
10	0.2296	1.0	3	0.8036	0.1964	$\Delta_{11} = M_3 \times \Delta_{10}$
11	0.2755	0.9	3	0.9643	0.0643	$\Delta_{12} = M_3 \times \Delta_{11}$

Notice how the quantizer adapts to the input. In the beginning of the sequence, the input values are mostly small, and the quantizer step size becomes progressively smaller, providing better and better estimates of the input. At the end of the sample sequence, the input values are large and the step size becomes progressively bigger. However, the size of the error is quite large during the transition. This means that if the input was changing rapidly, which would happen if we had a high-frequency input, such transition situations would be much more likely to occur, and the quantizer would not function very well. However, in cases where the statistics of the input change slowly, the quantizer could adapt to the input. As most natural sources such as speech and images tend to be correlated, their values do not change drastically from sample to sample. Even when some of this structure is removed through some transformation, the residual structure is generally enough for the Jayant quantizer (or some variation of it) to function quite effectively. ♦

The step size in the initial part of the sequence in this example is progressively getting smaller. We can easily conceive of situations where the input values would be small for a long period. Such a situation could occur during a silence period in speech-encoding systems,

or while encoding a dark background in image-encoding systems. If the step size continues to shrink for an extended period of time, in a finite precision system it would result in a value of zero. This would be catastrophic, effectively replacing the quantizer with a zero output device. Usually, a minimum value $\Delta_{\min}$ is defined, and the step size is not allowed to go below this value to prevent this from happening. Similarly, if we get a sequence of large values, the step size could increase to a point that, when we started getting smaller values, the quantizer would not be able to adapt fast enough. To prevent this from happening, a maximum value $\Delta_{\max}$ is defined, and the step size is not allowed to increase beyond this value.

The adaptivity of the Jayant quantizer depends on the values of the multipliers. The further the multiplier values are from unity, the more adaptive the quantizer. However, if the adaptation algorithm reacts too fast, this could lead to instability. So how do we go about selecting the multipliers?

First of all, we know that the multipliers correponding to the inner levels are less than one, and the multipliers for the outer levels are greater than one. If the input process is stationary and P_k represents the probability of being in quantizer interval k (generally estimated by using a fixed quantizer for the input data), then we can impose a stability criterion for the Jayant quantizer based on our requirement that once the quantizer is matched to the input, the product of the expansions and contractions are equal to unity. That is, if n_k is the number of times the input falls in the kth interval,

$$\prod_{k=0}^{M} M_k^{n_k} = 1. \tag{9.22}$$

Taking the Nth root of both sides (where N is the total number of inputs) we obtain

$$\prod_{k=0}^{M} M_k^{\frac{n_k}{N}} = 1,$$

or

$$\prod_{k=0}^{M} M_k^{P_k} = 1 \tag{9.23}$$

where we have assumed that $P_k = n_k/N$.

There are an infinite number of multiplier values that would satisfy Equation (9.23). One way to restrict this number is to impose some structure on the multipliers by requiring them to be of the form

$$M_k = \gamma^{l_k} \tag{9.24}$$

where γ is a number greater than one and l_k takes on only integer values [111, 112]. If we substitute this expression for M_k into Equation (9.23), we get

$$\prod_{k=0}^{M} \gamma^{l_k P_k} = 1, \tag{9.25}$$

which implies that

$$\sum_{k=0}^{M} l_k P_k = 0. \tag{9.26}$$

The final step is the selection of γ, which involves a significant amount of creativity. The value we pick for γ determines how fast the quantizer will respond to changing statistics. A large value of γ will result in faster adaptation, while a smaller value of γ will result in greater stability.

Example 9.5.4:

Suppose we have to obtain the multiplier functions for a 2-bit quantizer with input probabilities $P_0 = 0.8$, $P_1 = 0.2$. First, note that the multiplier value for the inner level has to be less than 1. Therefore, l_0 is less than 0. If we pick $l_0 = -1$ and $l_1 = 4$, this would satisfy Equation (9.26), while making M_0 less than 1 and M_1 greater than 1. Finally, we need to pick a value for γ.

In Figure 9.17 we see the effect of using different values of γ in a rather extreme example. The input is a square wave that switches between 0 and 1 every 30 samples. The input is quantized using a 2-bit Jayant quantizer. We have used $l_0 = -1$ and $l_1 = 2$. Notice what happens when the input switches from 0 to 1. At first the input falls in the outer level of the quantizer, and the step size increases. This process continues until Δ is just greater than 1. If γ is close to 1, Δ has been increasing quite slowly and should have a value close to 1 right before its value increases to greater than 1. Therefore, the output at this point is close to 1.5. When Δ becomes greater than 1, the input falls in the inner level, and if γ is close to 1, the output suddenly drops to about 0.5. The step size now decreases until it is just

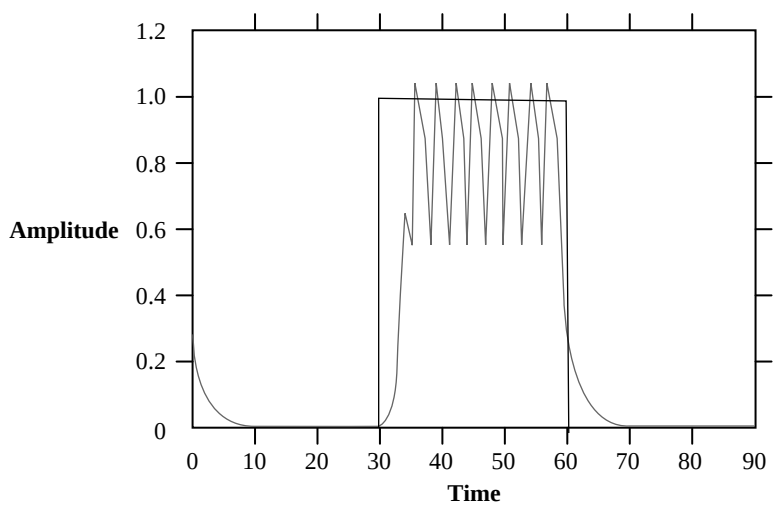

FIGURE 9.17 **Effect of γ on the performance of the Jayant quantizer.**

below 1, and the process repeats, causing the "ringing" seen in Figure 9.17. As γ increases, the quantizer adapts more rapidly, and the magnitude of the ringing effect decreases. The reason for the decrease is that right before the value of Δ increases above 1, its value is much smaller than 1, and subsequently the output value is much smaller than 1.5. When Δ increases beyond 1, it may increase by a significant amount, so the inner level may be much greater than 0.5. These two effects together compress the ringing phenomenon. Looking at this phenomenon, we can see that it may have been better to have two adaptive strategies, one for when the input is changing rapidly, as in the case of the transitions between 0 and 1, and one for when the input is constant, or nearly so. We will explore this approach further when we describe the quantizer used in the CCITT standard G.726. ♦

When selecting multipliers for a Jayant quantizer, the best quantizers expand more rapidly than they contract. This makes sense when we consider that, when the input falls into the outer levels of the quantizer, it is incurring overload error, which is essentially unbounded. This situation needs to be mitigated with dispatch. On the other hand, when the input falls in the inner levels, the noise incurred is granular noise, which is bounded and therefore may be more tolerable. Finally, the discussion of the Jayant quantizer was motivated by the need for robustness in the face of changing input statistics. Let us repeat the earlier experiment with changing input variance and distributions and see the performance of the Jayant quantizer compared to the *pdf*-optimized quantizer. The results for these experiments are presented in Figure 9.18.

Notice how flat the performance curve is. While the performance of the Jayant quantizer is much better than the nonadaptive uniform quantizer over a wide range of input variances, at the point where the input variance and design variance agree, the performance of the nonadaptive quantizer is significantly better than the performance of the Jayant quantizer.

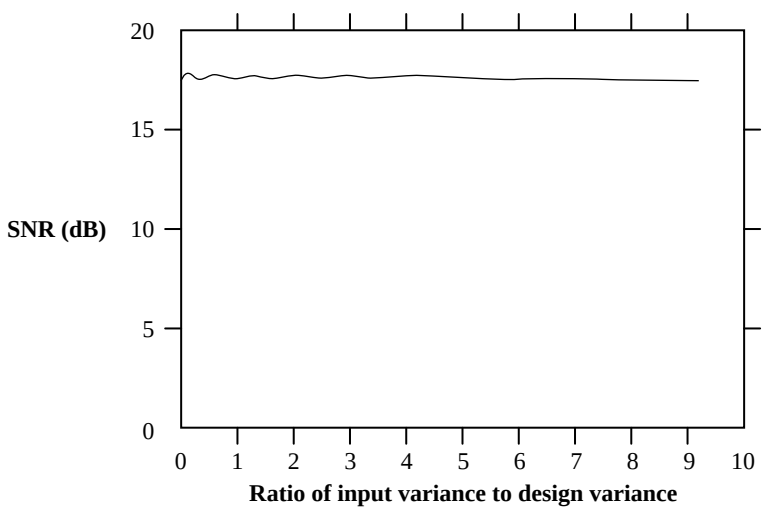

FIGURE 9.18 **Performance of the Jayant quantizer for different input variances.**

This means that if we know the input statistics and we are reasonably certain that the input statistics will not change over time, it is better to design for those statistics than to design an adaptive system.

9.6 Nonuniform Quantization

As we can see from Figure 9.10, if the input distribution has more mass near the origin, the input is more likely to fall in the inner levels of the quantizer. Recall that in lossless compression, in order to minimize the *average* number of bits per input symbol, we assigned shorter codewords to symbols that occurred with higher probability and longer codewords to symbols that occurred with lower probability. In an analogous fashion, in order to decrease the average distortion, we can try to approximate the input better in regions of high probability, perhaps at the cost of worse approximations in regions of lower probability. We can do this by making the quantization intervals smaller in those regions that have more probability mass. If the source distribution is like the distribution shown in Figure 9.10, we would have smaller intervals near the origin. If we wanted to keep the number of intervals constant, this would mean we would have larger intervals away from the origin. A quantizer that has nonuniform intervals is called a *nonuniform quantizer*. An example of a nonuniform quantizer is shown in Figure 9.19.

Notice that the intervals closer to zero are smaller. Hence the maximum value that the quantizer error can take on is also smaller, resulting in a better approximation. We pay for this improvement in accuracy at lower input levels by incurring larger errors when the input falls in the outer intervals. However, as the probability of getting smaller input values is much higher than getting larger signal values, on the average the distortion will be lower than if we had a uniform quantizer. While a nonuniform quantizer provides lower average distortion, the design of nonuniform quantizers is also somewhat more complex. However, the basic idea is quite straightforward: find the decision boundaries and reconstruction levels that minimize the mean squared quantization error. We look at the design of nonuniform quantizers in more detail in the following sections.

9.6.1 *pdf*-Optimized Quantization

A direct approach for locating the best nonuniform quantizer, if we have a probability model for the source, is to find the $\{b_i\}$ and $\{y_i\}$ that minimize Equation (9.3). Setting the derivative of Equation (9.3) with respect to y_j to zero, and solving for y_j, we get

$$y_j = \frac{\int_{b_{j-1}}^{b_j} x f_X(x)dx}{\int_{b_{j-1}}^{b_j} f_X(x)dx}. \tag{9.27}$$

The output point for each quantization interval is the centroid of the probability mass in that interval. Taking the derivative with respect to b_j and setting it equal to zero, we get an expression for b_j as

$$b_j = \frac{y_{j+1} + y_j}{2}. \tag{9.28}$$

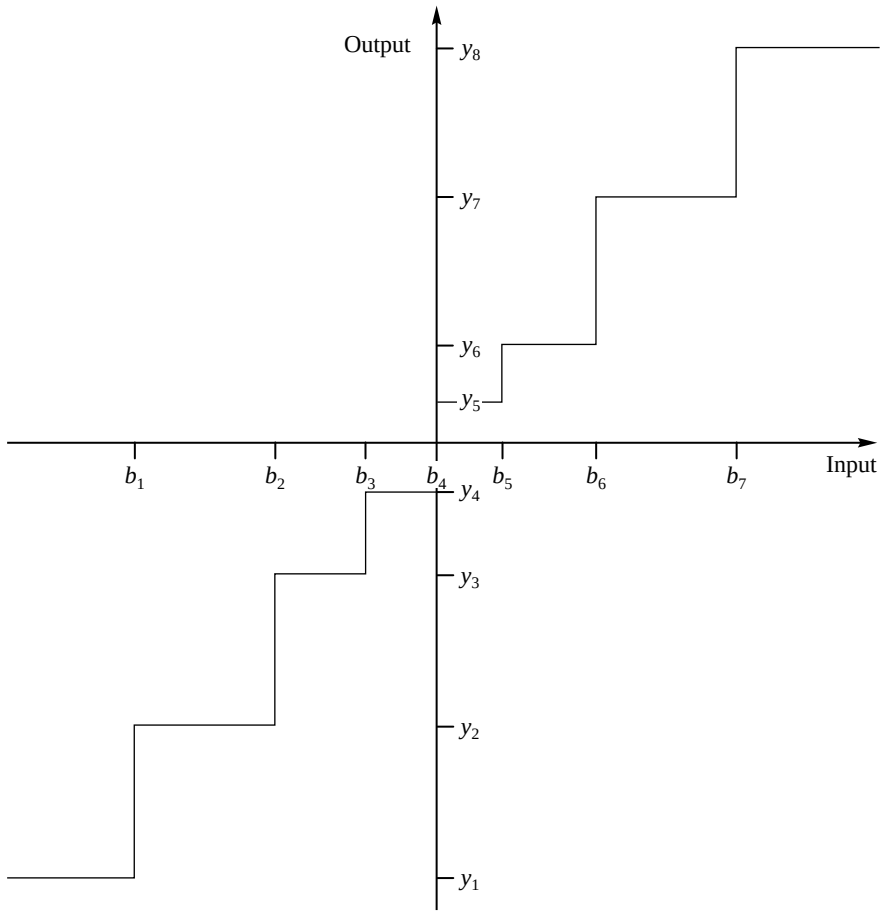

FIGURE 9. 19 A nonuniform midrise quantizer.

The decision boundary is simply the midpoint of the two neighboring reconstruction levels. Solving these two equations will give us the values for the reconstruction levels and decision boundaries that minimize the mean squared quantization error. Unfortunately, to solve for y_j, we need the values of b_j and b_{j-1}, and to solve for b_j, we need the values of y_{j+1} and y_j. In a 1960 paper, Joel Max [108] showed how to solve the two equations iteratively. The same approach was described by Stuart P. Lloyd in a 1957 internal Bell Labs memorandum. Generally, credit goes to whomever publishes first, but in this case, because much of the early work in quantization was done at Bell Labs, Lloyd's work was given due credit and the algorithm became known as the Lloyd-Max algorithm. However, the story does not end (begin?) there. Allen Gersho [113] points out that the same algorithm was published by Lukaszewicz and Steinhaus in a Polish journal in 1955 [114]! Lloyd's paper remained unpublished until 1982, when it was finally published in a special issue of the *IEEE Transactions on Information Theory* devoted to quantization [115].

To see how this algorithm works, let us apply it to a specific situation. Suppose we want to design an M-level symmetric midrise quantizer. To define our symbols, we will use Figure 9.20. From the figure, we see that in order to design this quantizer, we need to obtain the reconstruction levels $\{y_1, y_2, \ldots, y_{\frac{M}{2}}\}$ and the decision boundaries $\{b_1, b_2, \ldots, b_{\frac{M}{2}-1}\}$. The reconstruction levels $\{y_{-1}, y_{-2}, \ldots, y_{-\frac{M}{2}}\}$ and the decision boundaries $\{b_{-1}, b_{-2}, \ldots, b_{-(\frac{M}{2}-1)}\}$ can be obtained through symmetry, the decision boundary b_0 is zero, and the decision boundary $b_{\frac{M}{2}}$ is simply the largest value the input can take on (for unbounded inputs this would be ∞).

Let us set j equal to 1 in Equation (9.27):

$$y_1 = \frac{\int_{b_0}^{b_1} x f_X(x)dx}{\int_{b_0}^{b_1} f_X(x)dx}. \tag{9.29}$$

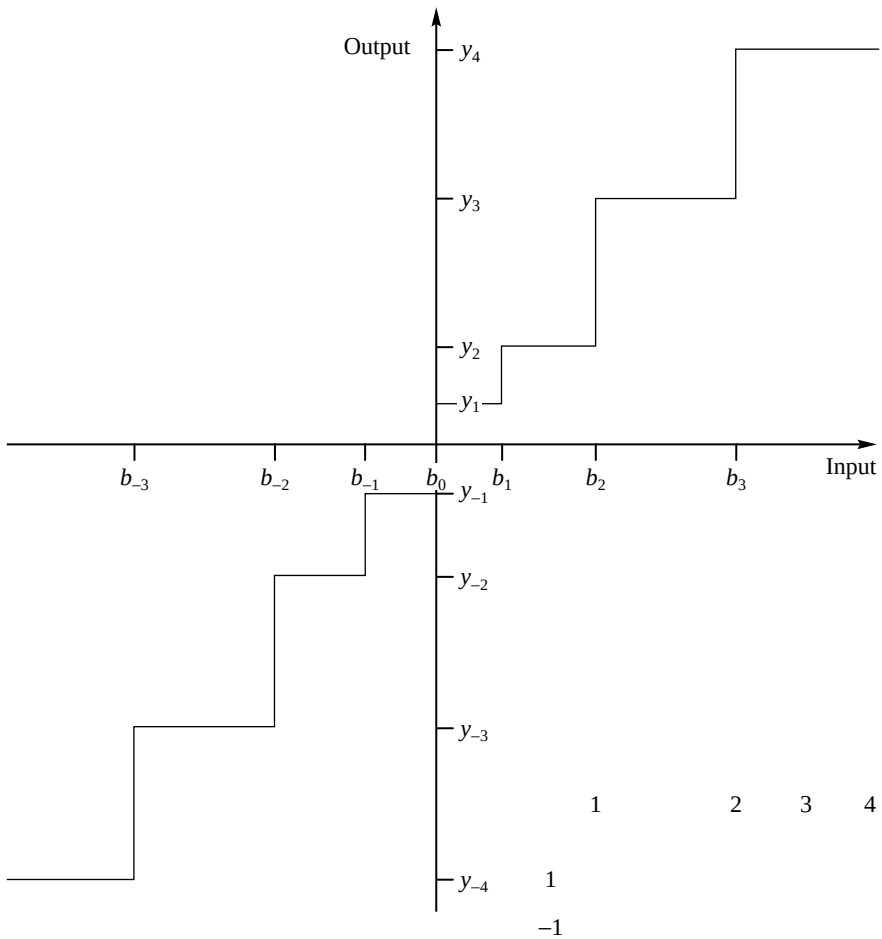

FIGURE 9. 20 A nonuniform midrise quantizer.

As b_0 is known to be 0, we have two unknowns in this equation, b_1 and y_1. We make a guess at y_1 and later we will try to refine this guess. Using this guess in Equation (9.29), we numerically find the value of b_1 that satisfies Equation (9.29). Setting j equal to 1 in Equation (9.28), and rearranging things slightly, we get

$$y_2 = 2b_1 + y_1 \tag{9.30}$$

from which we can compute y_2. This value of y_2 can then be used in Equation (9.27) with $j = 2$ to find b_2, which in turn can be used to find y_3. We continue this process, until we obtain a value for $\{y_1, y_2, \ldots, y_{\frac{M}{2}}\}$ and $\{b_1, b_2, \ldots, b_{\frac{M}{2}-1}\}$. Note that the accuracy of all the values obtained to this point depends on the quality of our initial estimate of y_1. We can check this by noting that $y_{\frac{M}{2}}$ is the centroid of the probability mass of the interval $[b_{\frac{M}{2}-1}, b_{\frac{M}{2}}]$. We know $b_{\frac{M}{2}}$ from our knowledge of the data. Therefore, we can compute the integral

$$y_{\frac{M}{2}} = \frac{\int_{b_{\frac{M}{2}-1}}^{b_{\frac{M}{2}}} x f_X(x)dx}{\int_{b_{\frac{M}{2}-1}}^{b_{\frac{M}{2}}} f_X(x)dx} \tag{9.31}$$

and compare it with the previously computed value of $y_{\frac{M}{2}}$. If the difference is less than some tolerance threshold, we can stop. Otherwise, we adjust the estimate of y_1 in the direction indicated by the sign of the difference and repeat the procedure.

Decision boundaries and reconstruction levels for various distributions and number of levels generated using this procedure are shown in Table 9.6. Notice that the distributions that have heavier tails also have larger outer step sizes. However, these same quantizers have smaller inner step sizes because they are more heavily peaked. The SNR for these quantizers is also listed in the table. Comparing these values with those for the *pdf*-optimized uniform quantizers, we can see a significant improvement, especially for distributions further away from the uniform distribution. Both uniform and nonuniform *pdf*-optimized, or Lloyd-Max,

TABLE 9.6 Quantizer boundary and reconstruction levels for nonuniform Gaussian and Laplacian quantizers.

	Gaussian			Laplacian		
Levels	b_i	y_i	SNR	b_i	y_i	SNR
4	0.0	0.4528		0.0	0.4196	
	0.9816	1.510	9.3 dB	1.1269	1.8340	7.54 dB
6	0.0	0.3177		0.0	0.2998	
	0.6589	1.0		0.7195	1.1393	
	1.447	1.894	12.41 dB	1.8464	2.5535	10.51 dB
8	0.0	0.2451		0.0	0.2334	
	0.7560	0.6812		0.5332	0.8330	
	1.050	1.3440		1.2527	1.6725	
	1.748	2.1520	14.62 dB	2.3796	3.0867	12.64 dB

quantizers have a number of interesting properties. We list these properties here (their proofs can be found in [116, 117, 118]):

- **Property 1:** The mean values of the input and output of a Lloyd-Max quantizer are equal.
- **Property 2:** For a given Lloyd-Max quantizer, the variance of the output is always less than or equal to the variance of the input.
- **Property 3:** The mean squared quantization error for a Lloyd-Max quantizer is given by

$$\sigma_q^2 = \sigma_x^2 - \sum_{j=1}^{M} y_j^2 P[b_{j-1} \le X < b_j] \tag{9.32}$$

  where σ_x^2 is the variance of the quantizer input, and the second term on the right-hand side is the second moment of the output (or variance if the input is zero mean).
- **Property 4:** Let N be the random variable corresponding to the quantization error. Then for a given Lloyd-Max quantizer,

$$E[XN] = -\sigma_q^2. \tag{9.33}$$

- **Property 5:** For a given Lloyd-Max quantizer, the quantizer output and the quantization noise are orthogonal:

$$E[Q(X)N \mid b_0, b_1, \ldots, b_M] = 0. \tag{9.34}$$

Mismatch Effects

As in the case of uniform quantizers, the *pdf*-optimized nonuniform quantizers also have problems when the assumptions underlying their design are violated. In Figure 9.21 we show the effects of variance mismatch on a 4-bit Laplacian nonuniform quantizer.

This mismatch effect is a serious problem because in most communication systems the input variance can change considerably over time. A common example of this is the telephone system. Different people speak with differing amounts of loudness into the telephone. The quantizer used in the telephone system needs to be quite robust to the wide range of input variances in order to provide satisfactory service.

One solution to this problem is the use of adaptive quantization to match the quantizer to the changing input characteristics. We have already looked at adaptive quantization for the uniform quantizer. Generalizing the uniform adaptive quantizer to the nonuniform case is relatively straightforward, and we leave that as a practice exercise (see Problem 8). A somewhat different approach is to use a nonlinear mapping to flatten the performance curve shown in Figure 9.21. In order to study this approach, we need to view the nonuniform quantizer in a slightly different manner.

9.6.2 Companded Quantization

Instead of making the step size small, we could make the interval in which the input lies with high probability large—that is, expand the region in which the input lands with high

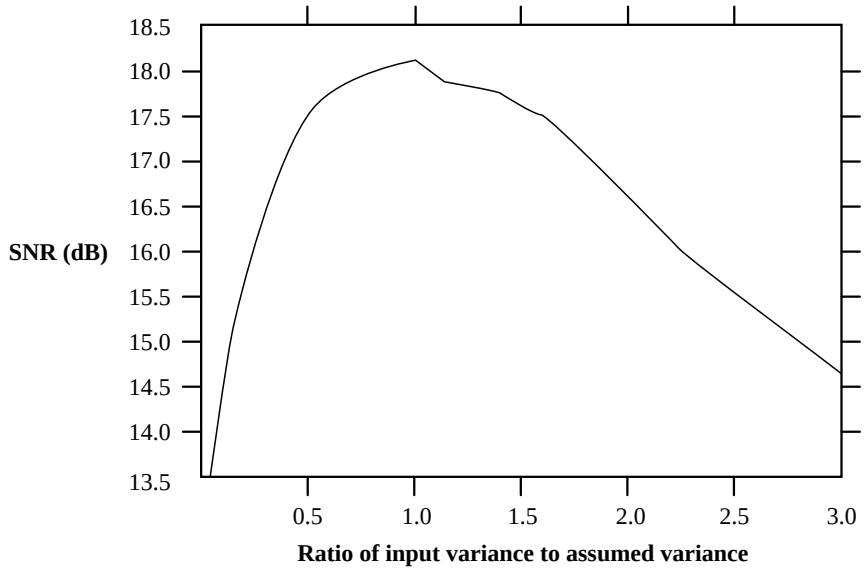

FIGURE 9. 21 Effect of mismatch on nonuniform quantization.

probability in proportion to the probability with which the input lands in this region. This is the idea behind companded quantization. This quantization approach can be represented by the block diagram shown in Figure 9.22. The input is first mapped through a *compressor* function. This function "stretches" the high-probability regions close to the origin, and correspondingly "compresses" the low-probability regions away from the origin. Thus, regions close to the origin in the input to the compressor occupy a greater fraction of the total region covered by the compressor. If the output of the compressor function is quantized using a uniform quantizer, and the quantized value transformed via an *expander* function,

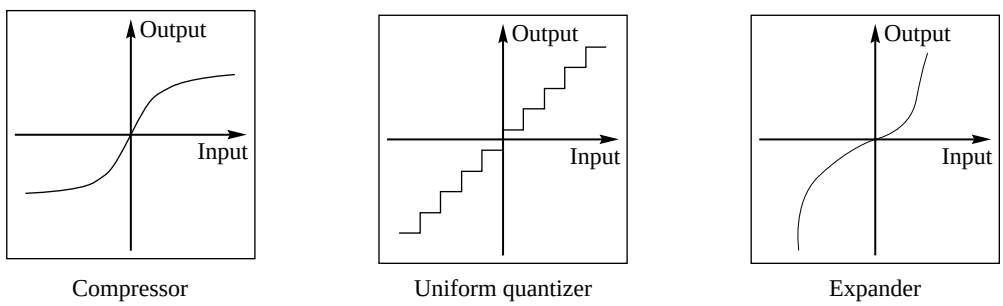

FIGURE 9. 22 Block diagram for log companded quantization.

the overall effect is the same as using a nonuniform quantizer. To see this, we devise a simple compander and see how the process functions.

Example 9.6.1:

Suppose we have a source that can be modeled as a random variable taking values in the interval $[-4, 4]$ with more probability mass near the origin than away from it. We want to quantize this using the quantizer of Figure 9.3. Let us try to flatten out this distribution using the following compander, and then compare the companded quantization with straightforward uniform quantization. The compressor characteristic we will use is given by the following equation:

$$c(x) = \begin{cases} 2x & \text{if } -1 \le x \le 1 \\ \frac{2x}{3} + \frac{4}{3} & x > 1 \\ \frac{2x}{3} - \frac{4}{3} & x < -1. \end{cases} \tag{9.35}$$

The mapping is shown graphically in Figure 9.23. The inverse mapping is given by

$$c^{-1}(x) = \begin{cases} \frac{x}{2} & \text{if } -2 \le x \le 2 \\ \frac{3x}{2} - 2 & x > 2 \\ \frac{3x}{2} + 2 & x < -2. \end{cases} \tag{9.36}$$

The inverse mapping is shown graphically in Figure 9.24.

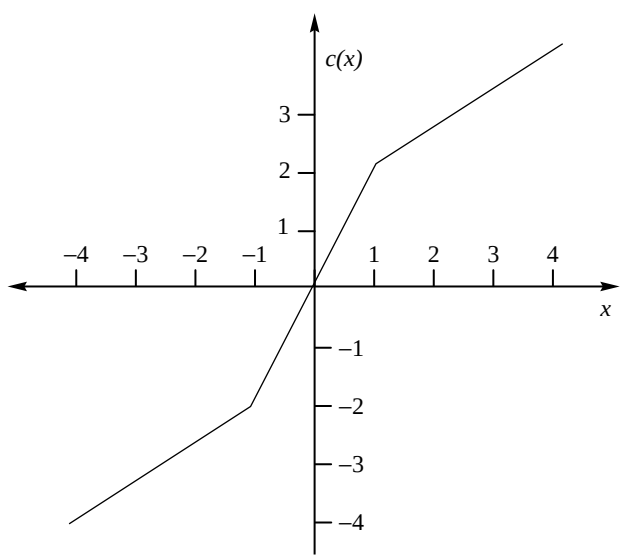

FIGURE 9. 23 **Compressor mapping.**

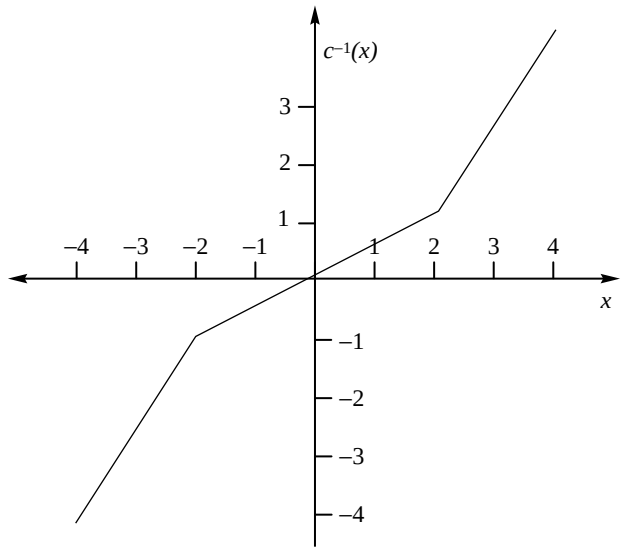

FIGURE 9. 24 Expander mapping.

Let's see how using these mappings affects the quantization error both near and far from the origin. Suppose we had an input of 0.9. If we quantize directly with the uniform quantizer, we get an output of 0.5, resulting in a quantization error of 0.4. If we use the companded quantizer, we first use the compressor mapping, mapping the input value of 0.9 to 1.8. Quantizing this with the same uniform quantizer results in an output of 1.5, with an apparent error of 0.3. The expander then maps this to the final reconstruction value of 0.75, which is 0.15 away from the input. Comparing 0.15 with 0.4, we can see that relative to the input we get a substantial reduction in the quantization error. In fact, for all values in the interval $[-1, 1]$, we will not get any increase in the quantization error, and for most values we will get a decrease in the quantization error (see Problem 6 at the end of this chapter). Of course, this will not be true for the values outside the $[-1, 1]$ interval. Suppose we have an input of 2.7. If we quantized this directly with the uniform quantizer, we would get an output of 2.5, with a corresponding error of 0.2. Applying the compressor mapping, the value of 2.7 would be mapped to 3.13, resulting in a quantized value of 3.5. Mapping this back through the expander, we get a reconstructed value of 3.25, which differs from the input by 0.55.

As we can see, the companded quantizer effectively works like a nonuniform quantizer with smaller quantization intervals in the interval $[-1, 1]$ and larger quantization intervals outside this interval. What is the effective input-output map of this quantizer? Notice that all inputs in the interval [0, 0.5] get mapped into the interval [0, 1], for which the quantizer output is 0.5, which in turn corresponds to the reconstruction value of 0.25. Essentially, all values in the interval [0, 0.5] are represented by the value 0.25. Similarly, all values in

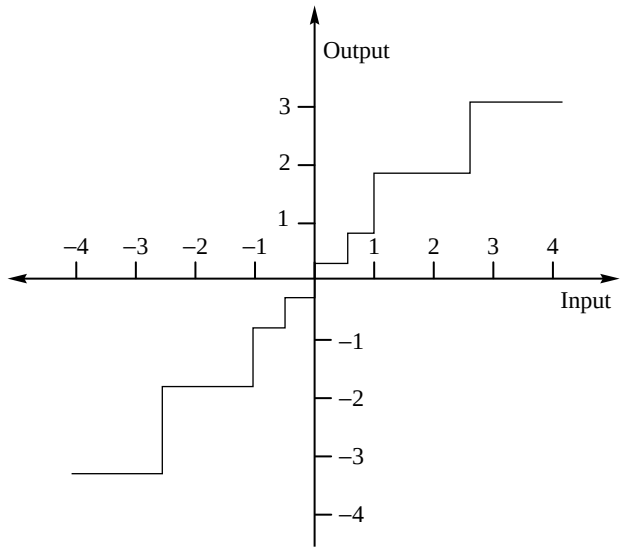

FIGURE 9. 25 Nonuniform companded quantizer.

the interval [0.5, 1] are represented by the value 0.75, and so on. The effective quantizer input-output map is shown in Figure 9.25. ♦

If we bound the source output by some value $x_{\max}$, any nonuniform quantizer can always be represented as a companding quantizer. Let us see how we can use this fact to come up with quantizers that are robust to mismatch. First we need to look at some of the properties of high-rate quantizers, or quantizers with a large number of levels.

Define

$$\Delta_k = b_k - b_{k-1}. \tag{9.37}$$

If the number of levels is high, then the size of each quantization interval will be small, and we can assume that the *pdf* of the input $f_X(x)$ is essentially constant in each quantization interval. Then

$$f_X(x) = f_X(y_k) \qquad \text{if } b_{k-1} \le x < b_k. \tag{9.38}$$

Using this we can rewrite Equation (9.3) as

$$\sigma_q^2 = \sum_{i=1}^{M} f_X(y_i) \int_{b_{i-1}}^{b_i} (x - y_i)^2 dx \tag{9.39}$$

$$= \frac{1}{12} \sum_{i=1}^{M} f_X(y_i) \Delta_i^3. \tag{9.40}$$

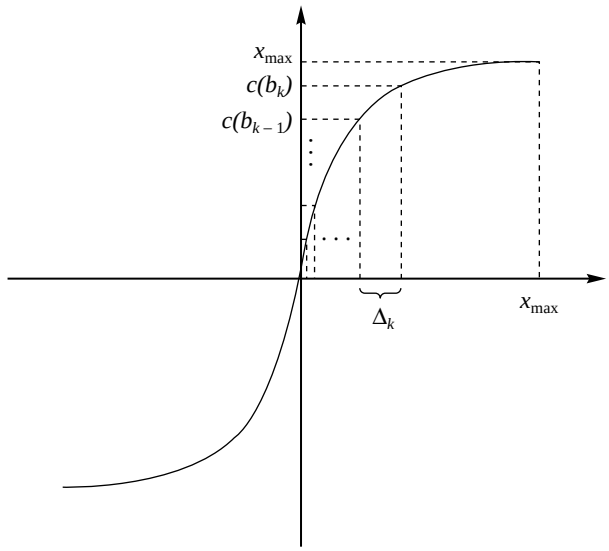

FIGURE 9.26 A compressor function.

Armed with this result, let us return to companded quantization. Let $c(x)$ be a companding characteristic for a symmetric quantizer, and let $c'(x)$ be the derivative of the compressor characteristic with respect to x. If the rate of the quantizer is high, that is, if there are a large number of levels, then within the kth interval, the compressor characteristic can be approximated by a straight line segment (see Figure 9.26), and we can write

$$c'(y_k) = \frac{c(b_k) - c(b_{k-1})}{\Delta_k}. \tag{9.41}$$

From Figure 9.26 we can also see that $c(b_k) - c(b_{k-1})$ is the step size of a uniform M-level quantizer. Therefore,

$$c(b_k) - c(b_{k-1}) = \frac{2x_{max}}{M}. \tag{9.42}$$

Substituting this into Equation (9.41) and solving for Δ_k, we get

$$\Delta_k = \frac{2x_{max}}{Mc'(y_k)}. \tag{9.43}$$

Finally, substituting this expression for Δ_k into Equation (9.40), we get the following relationship between the quantizer distortion, the *pdf* of the input, and the compressor characteristic:

$$\begin{aligned}\sigma_q^2 &= \frac{1}{12}\sum_{i=1}^{M} f_X(y_i)\left(\frac{2x_{\max}}{Mc'(y_i)}\right)^3 \\ &= \frac{x_{\max}^2}{3M^2}\sum_{i=1}^{M}\frac{f_X(y_i)}{c'^2(y_i)}\cdot\frac{2x_{\max}}{Mc'(y_i)} \\ &= \frac{x_{\max}^2}{3M^2}\sum_{i=1}^{M}\frac{f_X(y_i)}{c'^2(y_i)}\Delta_i \qquad (9.44)\end{aligned}$$

which for small Δ_i can be written as

$$\sigma_q^2 = \frac{x_{\max}^2}{3M^2}\int_{-x_{\max}}^{x_{\max}}\frac{f_X(x)}{(c'(x))^2}dx. \qquad (9.45)$$

This is a famous result, known as the Bennett integral after its discoverer, W.R. Bennett [119], and it has been widely used to analyze quantizers. We can see from this integral that the quantizer distortion is dependent on the *pdf* of the source sequence. However, it also tells us how to get rid of this dependence. Define

$$c'(x) = \frac{x_{\max}}{\alpha\,|x|}, \qquad (9.46)$$

where α is a constant. From the Bennett integral we get

$$\sigma_q^2 = \frac{x_{\max}^2}{3M^2}\frac{\alpha^2}{x_{\max}^2}\int_{-x_{\max}}^{x_{\max}} x^2 f_X(x)dx \qquad (9.47)$$

$$= \frac{\alpha^2}{3M^2}\sigma_x^2 \qquad (9.48)$$

where

$$\sigma_x^2 = \int_{-x_{\max}}^{x_{\max}} x^2 f_X(x)dx. \qquad (9.49)$$

Substituting the expression for σ_q^2 into the expression for SNR, we get

$$\text{SNR} = 10\log_{10}\frac{\sigma_x^2}{\sigma_q^2} \qquad (9.50)$$

$$= 10\log_{10}(3M^2) - 20\log_{10}\alpha \qquad (9.51)$$

which is independent of the input *pdf*. This means that if we use a compressor characteristic whose derivative satisfies Equation (9.46), then regardless of the input variance, the signal-to-noise ratio will remain constant. This is an impressive result. However, we do need some caveats.

Notice that we are not saying that the mean squared quantization error is independent of the quantizer input. It is not, as is clear from Equation (9.48). Remember also that this

result is valid as long as the underlying assumptions are valid. When the input variance is very small, our assumption about the *pdf* being constant over the quantization interval is no longer valid, and when the variance of the input is very large, our assumption about the input being bounded by $x_{\max}$ may no longer hold.

With fair warning, let us look at the resulting compressor characteristic. We can obtain the compressor characteristic by integrating Equation (9.46):

$$c(x) = x_{\max} + \beta \log \frac{|x|}{x_{\max}} \tag{9.52}$$

where β is a constant. The only problem with this compressor characteristic is that it becomes very large for small x. Therefore, in practice we approximate this characteristic with a function that is linear around the origin and logarithmic away from it.

Two companding characteristics that are widely used today are μ-law companding and A-law companding. The μ-law compressor function is given by

$$c(x) = x_{\max} \frac{\ln\left(1 + \mu \frac{|x|}{x_{\max}}\right)}{\ln(1+\mu)} \operatorname{sgn}(x). \tag{9.53}$$

The expander function is given by

$$c^{-1}(x) = \frac{x_{\max}}{\mu}[(1+\mu)^{\frac{|x|}{x_{\max}}} - 1]\operatorname{sgn}(x). \tag{9.54}$$

This companding characteristic with $\mu = 255$ is used in the telephone systems in North America and Japan. The rest of the world uses the A-law characteristic, which is given by

$$c(x) = \begin{cases} \frac{A|x|}{1+\ln A}\operatorname{sgn}(x) & 0 \le \frac{|x|}{x_{\max}} \le \frac{1}{A} \\ x_{\max}\frac{1+\ln \frac{A|x|}{x_{\max}}}{1+\ln A}\operatorname{sgn}(x) & \frac{1}{A} \le \frac{|x|}{x_{\max}} \le 1 \end{cases} \tag{9.55}$$

and

$$c^{-1}(x) = \begin{cases} \frac{|x|}{A}(1+\ln A) & 0 \le \frac{|x|}{x_{\max}} \le \frac{1}{1+\ln A} \\ \frac{x_{\max}}{A}\exp\left[\frac{|x|}{x_{\max}}(1+\ln A) - 1\right] & \frac{1}{1+\ln A} \le \frac{|x|}{x_{\max}} \le 1. \end{cases} \tag{9.56}$$

9.7 Entropy-Coded Quantization

In Section 9.3 we mentioned three tasks: selection of boundary values, selection of reconstruction levels, and selection of codewords. Up to this point we have talked about accomplishment of the first two tasks, with the performance measure being the mean squared quantization error. In this section we will look at accomplishing the third task, assigning codewords to the quantization interval. Recall that this becomes an issue when we use variable-length codes. In this section we will be looking at the latter situation, with the rate being the performance measure.

We can take two approaches to the variable-length coding of quantizer outputs. We can redesign the quantizer by taking into account the fact that the selection of the decision boundaries will affect the rate, or we can keep the design of the quantizer the same

(i.e., Lloyd-Max quantization) and simply entropy-code the quantizer output. Since the latter approach is by far the simpler one, let's look at it first.

9.7.1 Entropy Coding of Lloyd-Max Quantizer Outputs

The process of trying to find the optimum quantizer for a given number of levels and rate is a rather difficult task. An easier approach to incorporating entropy coding is to design a quantizer that minimizes the msqe, that is, a Lloyd-Max quantizer, then entropy-code its output.

In Table 9.7 we list the output entropies of uniform and nonuniform Lloyd-Max quantizers. Notice that while the difference in rate for lower levels is relatively small, for a larger number of levels, there can be a substantial difference between the fixed-rate and entropy-coded cases. For example, for 32 levels a fixed-rate quantizer would require 5 bits per sample. However, the entropy of a 32-level uniform quantizer for the Laplacian case is 3.779 bits per sample, which is more than 1 bit less. Notice that the difference between the fixed rate and the uniform quantizer entropy is generally greater than the difference between the fixed rate and the entropy of the output of the nonuniform quantizer. This is because the nonuniform quantizers have smaller step sizes in high-probability regions and larger step sizes in low-probability regions. This brings the probability of an input falling into a low-probability region and the probability of an input falling in a high-probability region closer together. This, in turn, raises the output entropy of the nonuniform quantizer with respect to the uniform quantizer. Finally, the closer the distribution is to being uniform, the less difference in the rates. Thus, the difference in rates is much less for the quantizer for the Gaussian source than the quantizer for the Laplacian source.

9.7.2 Entropy-Constrained Quantization ★

Although entropy coding the Lloyd-Max quantizer output is certainly simple, it is easy to see that we could probably do better if we take a fresh look at the problem of quantizer

TABLE 9.7 Output entropies in bits per sample for minimum mean squared error quantizers.

Number of	Gaussian		Laplacian	
Levels	Uniform	Nonuniform	Uniform	Nonuniform
4	1.904	1.911	1.751	1.728
6	2.409	2.442	2.127	2.207
8	2.759	2.824	2.394	2.479
16	3.602	3.765	3.063	3.473
32	4.449	4.730	3.779	4.427

design, this time with the entropy as a measure of rate rather than the alphabet size. The entropy of the quantizer output is given by

$$H(Q) = -\sum_{i=1}^{M} P_i \log_2 P_i \tag{9.57}$$

where P_i is the probability of the input to the quantizer falling in the ith quantization interval and is given by

$$P_i = \int_{b_{i-1}}^{b_i} f_X(x)dx. \tag{9.58}$$

Notice that the selection of the representation values $\{y_j\}$ has no effect on the rate. This means that we can select the representation values solely to minimize the distortion. However, the selection of the boundary values affects both the rate and the distortion. Initially, we found the reconstruction levels and decision boundaries that minimized the distortion, while keeping the rate fixed by fixing the quantizer alphabet size and assuming fixed-rate coding. In an analogous fashion, we can now keep the entropy fixed and try to minimize the distortion. Or, more formally:

> For a given R_o, find the decision boundaries $\{b_j\}$ that minimize σ_q^2 given by Equation (9.3), subject to $H(Q) \leq R_o$.

The solution to this problem involves the solution of the following $M-1$ nonlinear equations [120]:

$$\ln \frac{P_{l+1}}{P_l} = \lambda(y_{k+1} - y_k)(y_{k+1} + y_k - 2b_k) \tag{9.59}$$

where λ is adjusted to obtain the desired rate, and the reconstruction levels are obtained using Equation (9.27). A generalization of the method used to obtain the minimum mean squared error quantizers can be used to obtain solutions for this equation [121]. The process of finding optimum entropy-constrained quantizers looks complex. Fortunately, at higher rates we can show that the optimal quantizer is a uniform quantizer, simplifying the problem. Furthermore, while these results are derived for the high-rate case, it has been shown that the results also hold for lower rates [121].

9.7.3 High-Rate Optimum Quantization ★

At high rates, the design of optimum quantizers becomes simple, at least in theory. Gish and Pierce's work [122] says that at high rates the optimum entropy-coded quantizer is a uniform quantizer. Recall that any nonuniform quantizer can be represented by a compander and a uniform quantizer. Let us try to find the optimum compressor function at high rates that minimizes the entropy for a given distortion. Using the calculus of variations approach, we will construct the functional

$$J = H(Q) + \lambda\sigma_q^2, \tag{9.60}$$

then find the compressor characteristic to minimize it.

For the distortion σ_q^2, we will use the Bennett integral shown in Equation (9.45). The quantizer entropy is given by Equation (9.57). For high rates, we can assume (as we did before) that the *pdf* $f_X(x)$ is constant over each quantization interval Δ_i, and we can replace Equation (9.58) by

$$P_i = f_X(y_i)\Delta_i. \tag{9.61}$$

Substituting this into Equation (9.57), we get

$$H(Q) = -\sum f_X(y_i)\Delta_i \log[f_X(y_i)\Delta_i] \tag{9.62}$$

$$= -\sum f_X(y_i)\log[f_X(y_i)]\Delta_i - \sum f_X(y_i)\log[\Delta_i]\Delta_i \tag{9.63}$$

$$= -\sum f_X(y_i)\log[f_X(y_i)]\Delta_i - \sum f_X(y_i)\log\frac{2x_{\max}/M}{c'(y_i)}\Delta_i \tag{9.64}$$

where we have used Equation (9.43) for Δ_i. For small Δ_i we can write this as

$$H(Q) = -\int f_X(x)\log f_X(x)dx - \int f_X(x)\log\frac{2x_{\max}/M}{c'(x)}dx \tag{9.65}$$

$$= -\int f_X(x)\log f_X(x)dx - \log\frac{2x_{\max}}{M} + \int f_X(x)\log c'(x)dx \tag{9.66}$$

where the first term is the differential entropy of the source $h(X)$. Let's define $g = c'(x)$. Then substituting the value of $H(Q)$ into Equation (9.60) and differentiating with respect to g, we get

$$\int f_X(x)[g^{-1} - 2\lambda\frac{x_{\max}^2}{3M^2}g^{-3}]dx = 0. \tag{9.67}$$

This equation is satisfied if the integrand is zero, which gives us

$$g = \sqrt{\frac{2\lambda}{3}}\frac{x_{\max}}{M} = K(constant). \tag{9.68}$$

Therefore,

$$c'(x) = K \tag{9.69}$$

and

$$c(x) = Kx + \alpha. \tag{9.70}$$

If we now use the boundary conditions $c(0) = 0$ and $c(x_{\max}) = x_{\max}$, we get $c(x) = x$, which is the compressor characteristic for a uniform quantizer. Thus, at high rates the optimum quantizer is a uniform quantizer.

Substituting this expression for the optimum compressor function in the Bennett integral, we get an expression for the distortion for the optimum quantizer:

$$\sigma_q^2 = \frac{x_{\max}^2}{3M^2}. \tag{9.71}$$

Substituting the expression for $c(x)$ in Equation (9.66), we get the expression for the entropy of the optimum quantizer:

$$H(Q) = h(X) - \log \frac{2x_{\max}}{M}. \tag{9.72}$$

Note that while this result provides us with an easy method for designing optimum quantizers, our derivation is only valid if the source *pdf* is entirely contained in the interval $[-x_{\max}, x_{\max}]$, and if the step size is small enough that we can reasonably assume the *pdf* to be constant over a quantization interval. Generally, these conditions can only be satisfied if we have an extremely large number of quantization intervals. While theoretically this is not much of a problem, most of these reconstruction levels will be rarely used. In practice, as mentioned in Chapter 3, entropy coding a source with a large output alphabet is very problematic. One way we can get around this is through the use of a technique called *recursive indexing*.

Recursive indexing is a mapping of a countable set to a collection of sequences of symbols from another set with finite size [76]. Given a countable set $A = \{a_0, a_1, \ldots\}$ and a finite set $B = \{b_0, b_1, \ldots, b_M\}$ of size $M+1$, we can represent any element in A by a sequence of elements in B in the following manner:

1. Take the index i of element a_i of A.
2. Find the quotient m and remainder r of the index i such that

$$i = mM + r.$$

3. Generate the sequence: $\underbrace{b_M b_M \cdots b_M}_{m \text{ times}} b_r$.

B is called the representation set. We can see that given any element in A we will have a unique sequence from B representing it. Furthermore, no representative sequence is a prefix of any other sequence. Therefore, recursive indexing can be viewed as a trivial, uniquely decodable prefix code. The inverse mapping is given by

$$\underbrace{b_M b_M \cdots b_M}_{m \text{ times}} b_r \mapsto a_{mM+r}.$$

Since it is one-to-one, if it is used at the output of the quantizer to convert the index sequence of the quantizer output into the sequence of the recursive indices, the former can be recovered without error from the latter. Furthermore, when the size $M+1$ of the representation set B is chosen appropriately, in effect we can achieve the reduction in the size of the output alphabets that are used for entropy coding.

Example 9.7.1:

Suppose we want to represent the set of nonnegative integers $A = \{0, 1, 2, \ldots\}$ with the representation set $B = \{0, 1, 2, 3, 4, 5\}$. Then the value 12 would be represented by the sequence 5, 5, 2, and the value 16 would be represented by the sequence 5, 5, 5, 1. Whenever the

decoder sees the value 5, it simply adds on the next value until the next value is smaller than 5. For example, the sequence 3, 5, 1, 2, 5, 5, 1, 5, 0 would be decoded as 3, 6, 2, 11, 5. ♦

Recursive indexing is applicable to any representation of a large set by a small set. One way of applying recursive indexing to the problem of quantization is as follows: For a given step size $\Delta > 0$ and a positive integer K, define x_l and x_h as follows:

$$x_l = -\left\lfloor \frac{K-1}{2} \right\rfloor \Delta$$
$$x_h = x_l + (K-1)\Delta$$

where $\lfloor x \rfloor$ is the largest integer not exceeding x. We define a recursively indexed quantizer of size K to be a uniform quantizer with step size Δ and with x_l and x_h being its smallest and largest output levels. (Q defined this way also has 0 as its output level.) The quantization rule Q, for a given input value x, is as follows:

1. If x falls in the interval $(x_l + \frac{\Delta}{2}, x_h - \frac{\Delta}{2})$, then $Q(x)$ is the nearest output level.
2. If x is greater than $x_h - \frac{\Delta}{2}$, see if $x_1 \stackrel{\Delta}{=} x - x_h \in (x_l + \frac{\Delta}{2}, x_h - \frac{\Delta}{2})$. If so, $Q(x) = (x_h, Q(x_1))$. If not, form $x_2 = x - 2x_h$ and do the same as for x_1. This process continues until for some m, $x_m = x - mx_h$ falls in $(x_l + \frac{\Delta}{2}, x_h - \frac{\Delta}{2})$, which will be quantized into
$$Q(x) = (\underbrace{x_h, x_h, \ldots, x_h}_{m \text{ times}}, Q(x_m)). \tag{9.73}$$
3. If x is smaller than $x_l + \frac{\Delta}{2}$, a similar procedure to the above is used; that is, form $x_m = x + mx_l$ so that it falls in $(x_l + \frac{\Delta}{2}, x_h - \frac{\Delta}{2})$, and quantize it to $(x_l, x_l, \ldots, x_l, Q(x_m))$.

In summary, the quantizer operates in two modes: one when the input falls in the range (x_l, x_h), the other when it falls outside of the specified range. The recursive nature in the second mode gives it the name.

We pay for the advantage of encoding a larger set by a smaller set in several ways. If we get a large input to our quantizer, the representation sequence may end up being intolerably large. We also get an increase in the rate. If $H(Q)$ is the entropy of the quantizer output, and γ is the average number of representation symbols per input symbol, then the minimum rate for the recursively indexed quantizer is $\gamma H(Q)$.

In practice, neither cost is too large. We can avoid the problem of intolerably large sequences by adopting some simple strategies for representing these sequences, and the value of γ is quite close to one for reasonable values of M. For Laplacian and Gaussian quantizers, a typical value for M would be 15 [76].

9.8 Summary

The area of quantization is a well-researched area and much is known about the subject. In this chapter, we looked at the design and performance of uniform and nonuniform quantizers for a variety of sources, and how the performance is affected when the assumptions used

in the design process are not correct. When the source statistics are not well known or change with time, we can use an adaptive strategy. One of the more popular approaches to adaptive quantization is the Jayant quantizer. We also looked at the issues involved with entropy-coded quantization.

Further Reading

With an area as broad as quantization, we had to keep some of the coverage rather cursory. However, there is a wealth of information on quantization available in the published literature. The following sources are especially useful for a general understanding of the area:

1. A very thorough coverage of quantization can be found in *Digital Coding of Waveforms*, by N.S. Jayant and P. Noll [123].
2. The paper "Quantization," by A. Gersho, in *IEEE Communication Magazine*, September 1977 [113], provides an excellent tutorial coverage of many of the topics listed here.
3. The original paper by J. Max, "Quantization for Minimum Distortion," *IRE Transactions on Information Theory* [108], contains a very accessible description of the design of *pdf*-optimized quantizers.
4. A thorough study of the effects of mismatch is provided by W. Mauersberger in [124].

9.9 Projects and Problems

1. Show that the derivative of the distortion expression in Equation (9.18) results in the expression in Equation (9.19). You will have to use a result called Leibnitz's rule and the idea of a telescoping series. Leibnitz's rule states that if $a(t)$ and $b(t)$ are monotonic, then

$$\frac{\delta}{\delta t}\int_{a(t)}^{b(t)} f(x,t)dx = \int_{a(t)}^{b(t)} \frac{\delta f(x,t)}{\delta t}dx + f(b(t),t)\frac{\delta b(t)}{\delta t} - f(a(t),t)\frac{\delta a(t)}{\delta t}. \quad (9.74)$$

2. Use the program `falspos` to solve Equation (9.19) numerically for the Gaussian and Laplacian distributions. You may have to modify the function `func` in order to do this.
3. Design a 3-bit uniform quantizer (specify the decision boundaries and representation levels) for a source with a Laplacian *pdf*, with a mean of 3 and a variance of 4.
4. The pixel values in the Sena image are not really distributed uniformly. Obtain a histogram of the image (you can use the `hist_image` routine), and using the fact that the quantized image should be as good an approximation as possible for the original, design 1-, 2-, and 3-bit quantizers for this image. Compare these with the results displayed in Figure 9.7. (For better comparison, you can reproduce the results in the book using the program `uquan_img`.)

5. Use the program `misuquan` to study the effect of mismatch between the input and assumed variances. How do these effects change with the quantizer alphabet size and the distribution type?

6. For the companding quantizer of Example 9.6.1, what are the outputs for the following inputs: $-0.8, 1.2, 0.5, 0.6, 3.2, -0.3$? Compare your results with the case when the input is directly quantized with a uniform quantizer with the same number of levels. Comment on your results.

7. Use the test images Sena and Bookshelf1 to study the trade-offs involved in the selection of block sizes in the forward adaptive quantization scheme described in Example 9.5.2. Compare this with a more traditional forward adaptive scheme in which the variance is estimated and transmitted. The variance information should be transmitted using a uniform quantizer with differing number of bits.

8. Generalize the Jayant quantizer to the nonuniform case. Assume that the input is from a known distribution with unknown variance. Simulate the performance of this quantizer over the same range of ratio of variances as we have done for the uniform case. Compare your results to the fixed nonuniform quantizer and the adaptive uniform quantizer. To get a start on your program, you may wish to use `misnuq.c` and `juquan.c`.

9. Let's look at the rate distortion performance of the various quanitzers.

(a) Plot the rate-distortion function $R(D)$ for a Gaussian source with mean zero and variance $\sigma_X^2 = 2$.

(b) Assuming fixed length codewords, compute the rate and distortion for 1, 2, and 3 bit pdf-optimized nonuniform quantizers. Also, assume that X is a Gaussian random variable with mean zero and $\sigma_X^2 = 2$. Plot these values on the same graph with **x**'s.

(c) For the 2 and 3 bit quantizers, compute the rate and distortion assuming that the quantizer outputs are entropy coded. Plot these on the graph with **o**'s.

10

Vector Quantization

10.1 Overview

By grouping source outputs together and encoding them as a single block, we can obtain efficient lossy as well as lossless compression algorithms. Many of the lossless compression algorithms that we looked at took advantage of this fact. We can do the same with quantization. In this chapter, several quantization techniques that operate on blocks of data are described. We can view these blocks as vectors, hence the name "vector quantization." We will describe several different approaches to vector quantization. We will explore how to design vector quantizers and how these quantizers can be used for compression.

10.2 Introduction

In the last chapter, we looked at different ways of quantizing the output of a source. In all cases the quantizer inputs were scalar values, and each quantizer codeword represented a single sample of the source output. In Chapter 2 we saw that, by taking longer and longer sequences of input samples, it is possible to extract the structure in the source coder output. In Chapter 4 we saw that, even when the input is random, encoding sequences of samples instead of encoding individual samples separately provides a more efficient code. Encoding sequences of samples is more advantageous in the lossy compression framework as well. By "advantageous" we mean a lower distortion for a given rate, or a lower rate for a given distortion. As in the previous chapter, by "rate" we mean the average number of bits per input sample, and the measures of distortion will generally be the mean squared error and the signal-to-noise ratio.

The idea that encoding sequences of outputs can provide an advantage over the encoding of individual samples was first put forward by Shannon, and the basic results in information

theory were all proved by taking longer and longer sequences of inputs. This indicates that a quantization strategy that works with sequences or blocks of output would provide some improvement in performance over scalar quantization. In other words, we wish to generate a representative set of sequences. Given a source output sequence, we would represent it with one of the elements of the representative set.

In vector quantization we group the source output into blocks or vectors. For example, we can treat L consecutive samples of speech as the components of an L-dimensional vector. Or, we can take a block of L pixels from an image and treat each pixel value as a component of a vector of size or dimension L. This vector of source outputs forms the input to the vector quantizer. At both the encoder and decoder of the vector quantizer, we have a set of L-dimensional vectors called the *codebook* of the vector quantizer. The vectors in this codebook, known as *code-vectors*, are selected to be representative of the vectors we generate from the source output. Each code-vector is assigned a binary index. At the encoder, the input vector is compared to each code-vector in order to find the code-vector closest to the input vector. The elements of this code-vector are the quantized values of the source output. In order to inform the decoder about which code-vector was found to be the closest to the input vector, we transmit or store the binary index of the code-vector. Because the decoder has exactly the same codebook, it can retrieve the code-vector given its binary index. A pictorial representation of this process is shown in Figure 10.1.

Although the encoder may have to perform a considerable amount of computations in order to find the closest reproduction vector to the vector of source outputs, the decoding consists of a table lookup. This makes vector quantization a very attractive encoding scheme for applications in which the resources available for decoding are considerably less than the resources available for encoding. For example, in multimedia applications, considerable

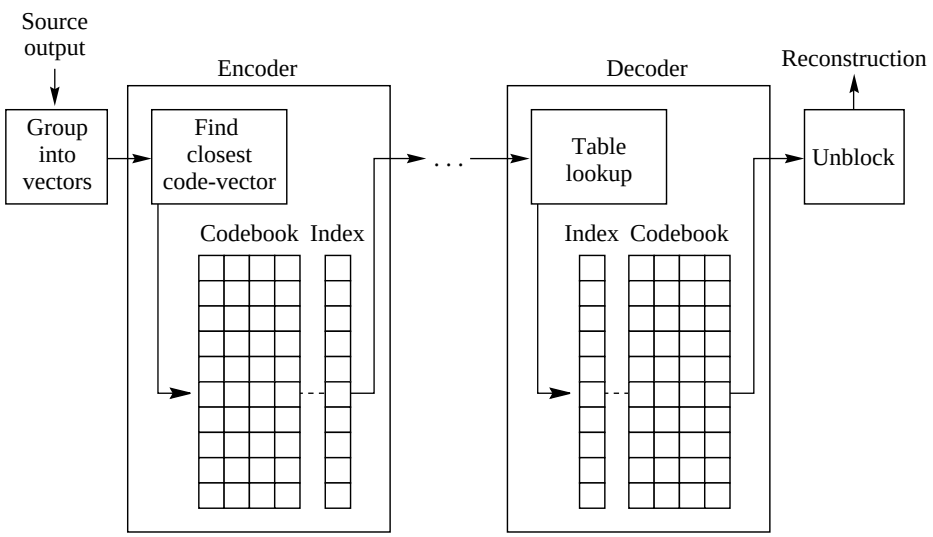

FIGURE 10. 1 **The vector quantization procedure.**

computational resources may be available for the encoding operation. However, if the decoding is to be done in software, the amount of computational resources available to the decoder may be quite limited.

Even though vector quantization is a relatively new area, it has developed very rapidly, and now even some of the subspecialties are broad areas of research. If this chapter we will try to introduce you to as much of this fascinating area as we can. If your appetite is whetted by what is available here and you wish to explore further, there is an excellent book by Gersho and Gray [5] devoted to the subject of vector quantization.

Our approach in this chapter is as follows: First, we try to answer the question of why we would want to use vector quantization over scalar quantization. There are several answers to this question, each illustrated through examples. In our discussion, we assume that you are familiar with the material in Chapter 9. We will then turn to one of the most important elements in the design of a vector quantizer, the generation of the codebook. While there are a number of ways of obtaining the vector quantizer codebook, most of them are based on one particular approach, popularly known as the Linde-Buzo-Gray (LBG) algorithm. We devote a considerable amount of time in describing some of the details of this algorithm. Our intent here is to provide you with enough information so that you can write your own programs for design of vector quantizer codebooks. In the software accompanying this book, we have also included programs for designing codebooks that are based on the descriptions in this chapter. If you are not currently thinking of implementing vector quantization routines, you may wish to skip these sections (Sections 10.4.1 and 10.4.2). We follow our discussion of the LBG algorithm with some examples of image compression using codebooks designed with this algorithm, and then with a brief sampling of the many different kinds of vector quantizers. Finally, we describe another quantization strategy, called trellis-coded quantization (TCQ), which, though different in implementation from the vector quantizers, also makes use of the advantage to be gained from operating on sequences.

Before we begin our discussion of vector quantization, let us define some of the terminology we will be using. The amount of compression will be described in terms of the rate, which will be measured in bits per sample. Suppose we have a codebook of size K, and the input vector is of dimension L. In order to inform the decoder of which code-vector was selected, we need to use $\lceil \log_2 K \rceil$ bits. For example, if the codebook contained 256 code-vectors, we would need 8 bits to specify which of the 256 code-vectors had been selected at the encoder. Thus, the number of bits *per vector* is $\lceil \log_2 K \rceil$ bits. As each code-vector contains the reconstruction values for L source output samples, the number of bits *per sample* would be $\frac{\lceil \log_2 K \rceil}{L}$. Thus, the rate for an L-dimensional vector quantizer with a codebook of size K is $\frac{\lceil \log_2 K \rceil}{L}$. As our measure of distortion we will use the mean squared error. When we say that in a codebook $\mathcal{C}$, containing the K code-vectors $\{Y_i\}$, the input vector X is closest to Y_j, we will mean that

$$\left\| X - Y_j \right\|^2 \leq \| X - Y_i \|^2 \qquad \text{for all} Y_i \in \mathcal{C} \tag{10.1}$$

where $X = (x_1 x_2 \cdots x_L)$ and

$$\| X \|^2 = \sum_{i=1}^{L} x_i^2. \tag{10.2}$$

The term *sample* will always refer to a scalar value. Thus, when we are discussing compression of images, a sample refers to a single pixel. Finally, the output points of the quantizer are often referred to as *levels*. Thus, when we wish to refer to a quantizer with K output points or code-vectors, we may refer to it as a K-level quantizer.

10.3 Advantages of Vector Quantization over Scalar Quantization

For a given rate (in bits per sample), use of vector quantization results in a lower distortion than when scalar quantization is used at the same rate, for several reasons. In this section we will explore these reasons with examples (for a more theoretical explanation, see [3, 4, 17]).

If the source output is correlated, vectors of source output values will tend to fall in clusters. By selecting the quantizer output points to lie in these clusters, we have a more accurate representation of the source output. Consider the following example.

Example 10.3.1:

In Example 8.5.1, we introduced a source that generates the height and weight of individuals. Suppose the height of these individuals varied uniformly between 40 and 80 inches, and the weight varied uniformly between 40 and 240 pounds. Suppose we were allowed a total of 6 bits to represent each pair of values. We could use 3 bits to quantize the height and 3 bits to quantize the weight. Thus, the weight range between 40 and 240 pounds would be divided into eight intervals of equal width of 25 and with reconstruction values $\{52, 77, \ldots, 227\}$. Similarly, the height range between 40 and 80 inches can be divided into eight intervals of width five, with reconstruction levels $\{42, 47, \ldots, 77\}$. When we look at the representation of height and weight separately, this approach seems reasonable. But let's look at this quantization scheme in two dimensions. We will plot the height values along the x-axis and the weight values along the y-axis. Note that we are not changing anything in the quantization process. The height values are still being quantized to the same eight different values, as are the weight values. The two-dimensional representation of these two quantizers is shown in Figure 10.2.

From the figure we can see that we effectively have a quantizer output for a person who is 80 inches (6 feet 8 inches) tall and weighs 40 pounds, as well as a quantizer output for an individual whose height is 42 inches but weighs more than 200 pounds. Obviously, these outputs will never be used, as is the case for many of the other outputs. A more sensible approach would be to use a quantizer like the one shown in Figure 10.3, where we take account of the fact that the height and weight are correlated. This quantizer has exactly the same number of output points as the quantizer in Figure 10.2; however, the output points are clustered in the area occupied by the input. Using this quantizer, we can no longer quantize the height and weight separately. We have to consider them as the coordinates of a point in two dimensions in order to find the closest quantizer output point. However, this method provides a much finer quantization of the input.

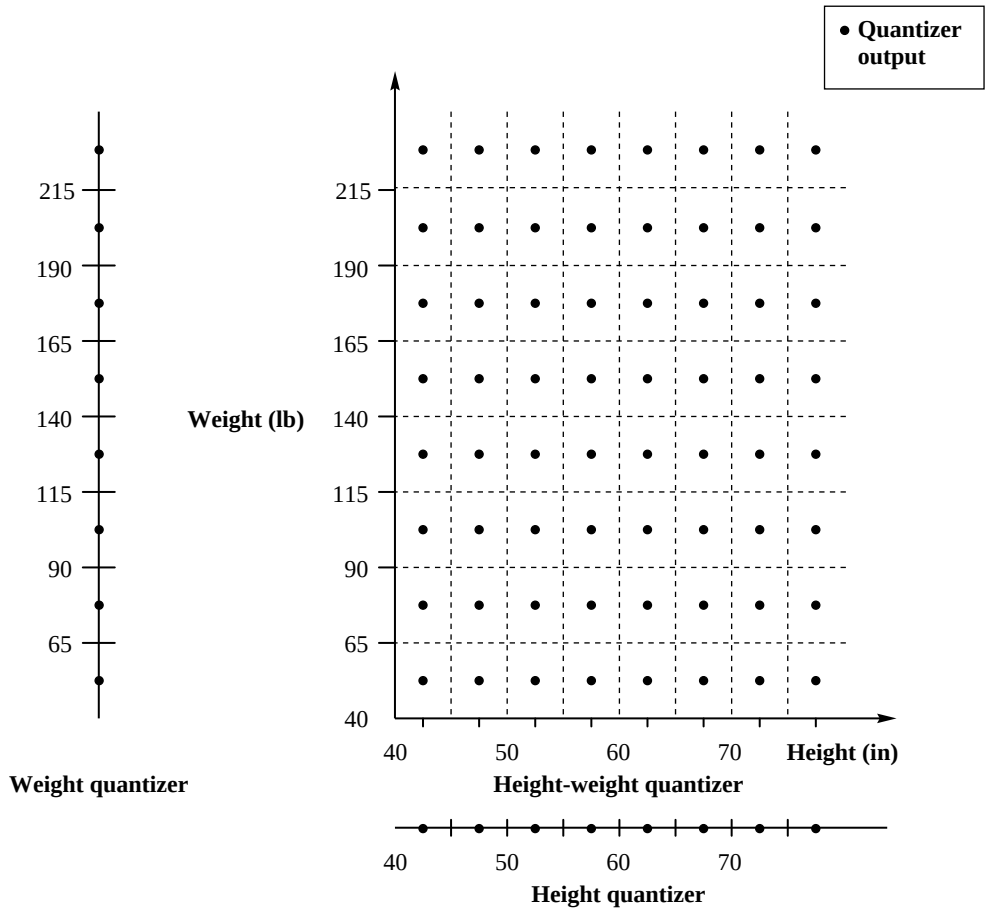

FIGURE 10. 2 **The height/weight scalar quantizers when viewed in two dimensions.**

Note that we have not said how we would obtain the locations of the quantizer outputs shown in Figure 10.3. These output points make up the codebook of the vector quantizer, and we will be looking at codebook design in some detail later in this chapter. ♦

We can see from this example that, as in lossless compression, looking at longer sequences of inputs brings out the structure in the source output. This structure can then be used to provide more efficient representations.

We can easily see how structure in the form of correlation between source outputs can make it more efficient to look at sequences of source outputs rather than looking at each sample separately. However, the vector quantizer is also more efficient than the scalar quantizer when the source output values are not correlated. The reason for this is actually

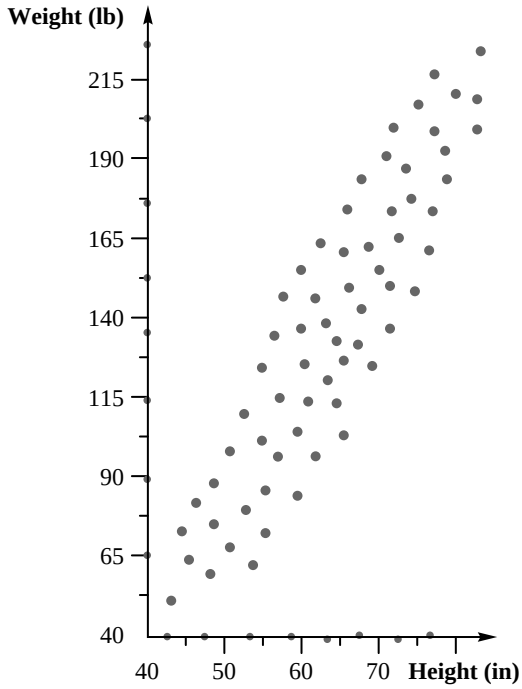

FIGURE 10.3 The height-weight vector quantizer.

quite simple. As we look at longer and longer sequences of source outputs, we are afforded more flexibility in terms of our design. This flexibility in turn allows us to match the design of the quantizer to the source characteristics. Consider the following example.

Example 10.3.2:

Suppose we have to design a uniform quantizer with eight output values for a Laplacian input. Using the information from Table 9.3 in Chapter 9, we would obtain the quantizer shown in Figure 10.4, where Δ is equal to 0.7309. As the input has a Laplacian distribution, the probability of the source output falling in the different quantization intervals is not the same. For example, the probability that the input will fall in the interval $[0, \Delta)$ is 0.3242, while the probability that a source output will fall in the interval $[3\Delta, \infty)$ is 0.0225. Let's look at how this quantizer will quantize two consecutive source outputs. As we did in the previous example, let's plot the first sample along the x-axis and the second sample along the y-axis. We can represent this two-dimensional view of the quantization process as shown in Figure 10.5. Note that, as in the previous example, we have not changed the quantization process; we are simply representing it differently. The first quantizer input, which we have represented in the figure as x_1, is quantized to the same eight possible output values as before. The same is true for the second quantizer input, which we have represented in the

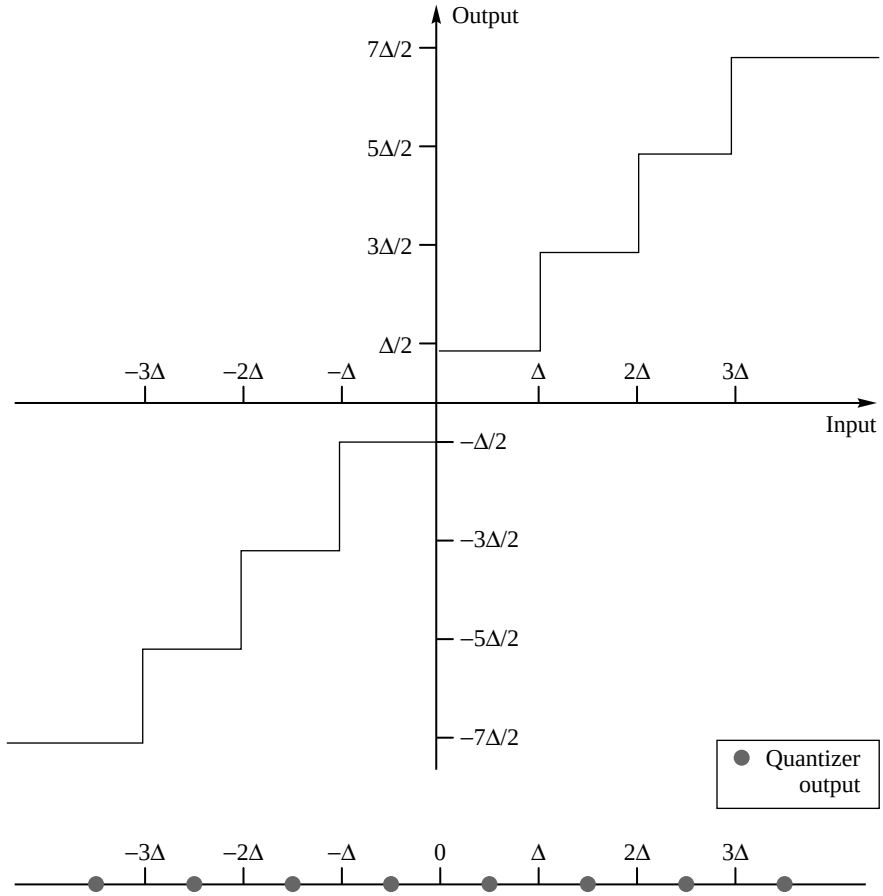

FIGURE 10. 4 **Two representations of an eight-level scalar quantizer.**

figure as x_2. This two-dimensional representation allows us to examine the quantization process in a slightly different manner. Each filled-in circle in the figure represents a sequence of two quantizer outputs. For example, the top rightmost circle represents the two quantizer outputs that would be obtained if we had two consecutive source outputs with a value greater than 3Δ. We computed the probability of a single source output greater than 3Δ to be 0.0225. The probability of two consecutive source outputs greater than 2.193 is simply $0.0225 \times 0.0225 = 0.0005$, which is quite small. Given that we do not use this output point very often, we could simply place it somewhere else where it would be of more use. Let us move this output point to the origin, as shown in Figure 10.6. We have now modified the quantization process. Now if we get two consecutive source outputs with values greater than 3Δ, the quantizer output corresponding to the second source output may not be the same as the first source output.

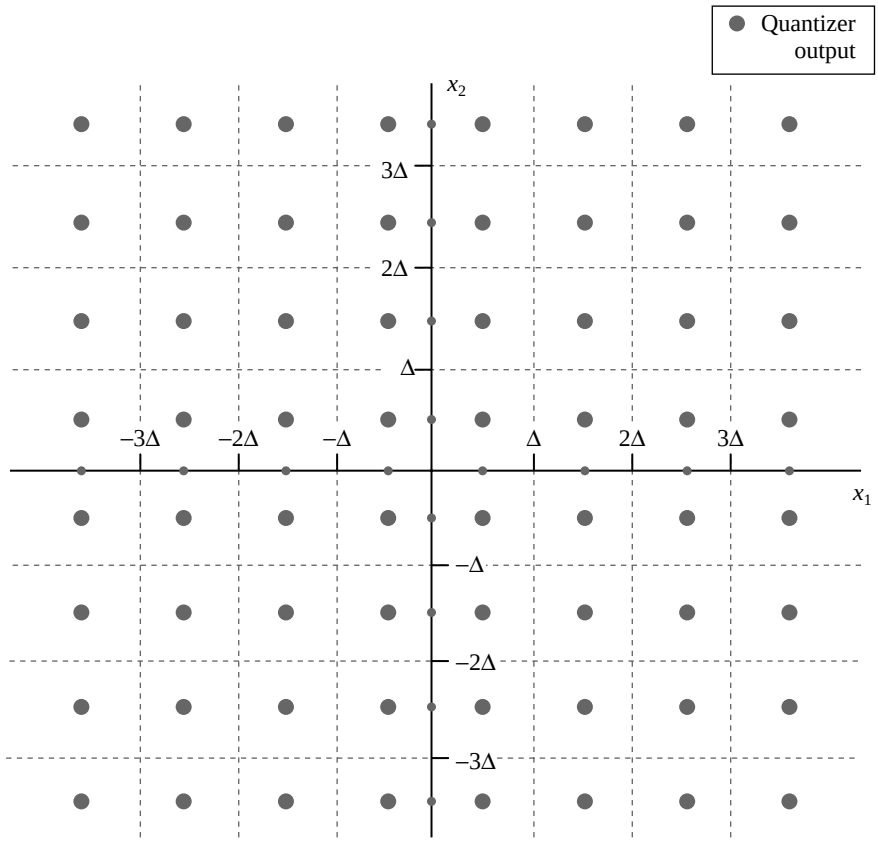

FIGURE 10.5 **Input-output map for consecutive quantization of two inputs using an eight-level scalar quantizer.**

If we compare the rate distortion performance of the two vector quantizers, the SNR for the first vector quantizer is 11.44 dB, which agrees with the result in Chapter 9 for the uniform quantizer with a Laplacian input. The SNR for the modified vector quantizer, however, is 11.73 dB, an increase of about 0.3 dB. Recall that the SNR is a measure of the average squared value of the source output samples and the mean squared error. As the average squared value of the source output is the same in both cases, an increase in SNR means a decrease in the mean squared error. Whether this increase in SNR is significant will depend on the particular application. What is important here is that by treating the source output in groups of two we could effect a positive change with only a minor modification. We could argue that this modification is really not that minor since the uniform characteristic of the original quantizer has been destroyed. However, if we begin with a nonuniform quantizer and modify it in a similar way, we get similar results.

Could we do something similar with the scalar quantizer? If we move the output point at $\frac{7\Delta}{2}$ to the origin, the SNR *drops* from 11.44 dB to 10.8 dB. What is it that permits us to make

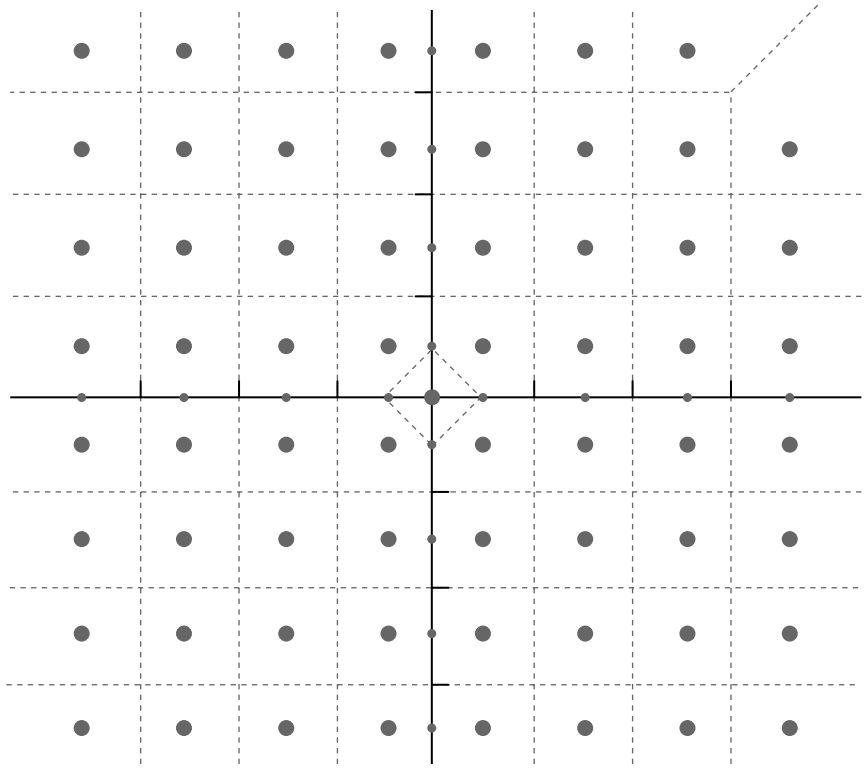

FIGURE 10. 6 Modified two-dimensional vector quantizer.

modifications in the vector case, but not in the scalar case? This advantage is caused by the added flexibility we get by viewing the quantization process in higher dimensions. Consider the effect of moving the output point from $\frac{7\Delta}{2}$ to the origin in terms of two consecutive inputs. This one change in one dimension corresponds to moving 15 output points in two dimensions. Thus, modifications at the scalar quantizer level are gross modifications when viewed from the point of view of the vector quantizer. Remember that in this example we have only looked at two-dimensional vector quantizers. As we block the input into larger and larger blocks or vectors, these higher dimensions provide even greater flexibility and the promise of further gains to be made. ♦

In Figure 10.6, notice how the quantization regions have changed for the outputs around the origin, as well as for the two neighbors of the output point that were moved. The decision boundaries between the reconstruction levels can no longer be described as easily as in the case for the scalar quantizer. However, if we know the distortion measure, simply knowing the output points gives us sufficent information to implement the quantization

process. Instead of defining the quantization rule in terms of the decision boundary, we can define the quantization rule as follows:

$$Q(X) = Y_j \quad \text{iff} \quad d(X, Y_j) < d(X, Y_i) \ \forall i \neq j. \tag{10.3}$$

For the case where the input X is equidistant from two output points, we can use a simple tie-breaking rule such as "use the output point with the smaller index." The quantization regions V_j can then be defined as

$$V_j = \{X : d(X, Y_j) < d(X, Y_i) \ \forall i \neq j\}. \tag{10.4}$$

Thus, the quantizer is completely defined by the output points and a distortion measure.

From a multidimensional point of view, using a scalar quantizer for each input restricts the output points to a rectangular grid. Observing several source output values at once allows us to move the output points around. Another way of looking at this is that in one dimension the quantization intervals are restricted to be intervals, and the only parameter that we can manipulate is the size of these intervals. When we divide the input into vectors of some length n, the quantization regions are no longer restricted to be rectangles or squares. We have the freedom to divide the range of the inputs in an infinite number of ways.

These examples have shown two ways in which the vector quantizer can be used to improve performance. In the first case, we exploited the sample-to-sample dependence of the input. In the second case, there was no sample-to-sample dependence; the samples were independent. However, looking at two samples together still improved performance.

These two examples can be used to motivate two somewhat different approaches toward vector quantization. One approach is a pattern-matching approach, similar to the process used in Example 10.3.1, while the other approach deals with the quantization of random inputs. We will look at both of these approaches in this chapter.

10.4 The Linde-Buzo-Gray Algorithm

In Example 10.3.1 we saw that one way of exploiting the structure in the source output is to place the quantizer output points where the source output (blocked into vectors) are most likely to congregate. The set of quantizer output points is called the *codebook* of the quantizer, and the process of placing these output points is often referred to as *codebook design*. When we group the source output in two-dimensional vectors, as in the case of Example 10.3.1, we might be able to obtain a good codebook design by plotting a representative set of source output points and then visually locate where the quantizer output points should be. However, this approach to codebook design breaks down when we design higher-dimensional vector quantizers. Consider designing the codebook for a 16-dimensional quantizer. Obviously, a visual placement approach will not work in this case. We need an automatic procedure for locating where the source outputs are clustered.

This is a familiar problem in the field of pattern recognition. It is no surprise, therefore, that the most popular approach to designing vector quantizers is a clustering procedure known as the k-means algorithm, which was developed for pattern recognition applications.

The k-means algorithm functions as follows: Given a large set of output vectors from the source, known as the *training set*, and an initial set of k representative patterns, assign each element of the training set to the closest representative pattern. After an element is assigned, the representative pattern is updated by computing the centroid of the training set vectors assigned to it. When the assignment process is complete, we will have k groups of vectors clustered around each of the output points.

Stuart Lloyd [115] used this approach to generate the *pdf*-optimized scalar quantizer, except that instead of using a training set, he assumed that the distribution was known. The Lloyd algorithm functions as follows:

1. Start with an initial set of reconstruction values $\left\{y_i^{(0)}\right\}_{i=1}^{M}$. Set $k = 0$, $D^{(0)} = 0$. Select threshold ϵ.
2. Find decision boundaries
$$b_j^{(k)} = \frac{y_{j+1}^{(k)} + y_j^{(k)}}{2} \qquad j = 1, 2, \ldots, M-1.$$
3. Compute the distortion
$$D^{(k)} = \sum_{i=1}^{M} \int_{b_{i-1}^{(k)}}^{b_i^{(k)}} (x - y_i)^2 f_X(x)\,dx.$$
4. If $D^{(k)} - D^{(k-1)} < \epsilon$, stop; otherwise, continue.
5. $k = k + 1$. Compute new reconstruction values
$$y_j^{(k)} = \frac{\int_{b_{j-1}^{(k-1)}}^{b_j^{(k-1)}} x f_X(x)\,dx}{\int_{b_{j-1}^{(k-1)}}^{b_j^{(k-1)}} f_X(x)\,dx}.$$
Go to Step 2.

Linde, Buzo, and Gray generalized this algorithm to the case where the inputs are no longer scalars [125]. For the case where the distribution is known, the algorithm looks very much like the Lloyd algorithm described above.

1. Start with an initial set of reconstruction values $\left\{Y_i^{(0)}\right\}_{i=1}^{M}$. Set $k = 0$, $D^{(0)} = 0$. Select threshold ϵ.
2. Find quantization regions
$$V_i^{(k)} = \{X : d(X, Y_i) < d(X, Y_j)\ \forall j \neq i\} \qquad j = 1, 2, \ldots, M.$$
3. Compute the distortion
$$D^{(k)} = \sum_{i=1}^{M} \int_{V_i^{(k)}} \left\| X - Y_i^{(k)} \right\|^2 f_X(X)\,dX.$$

4. If $\frac{(D^{(k)}-D^{(k-1)})}{D^{(k)}} < \epsilon$, stop; otherwise, continue.
5. $k = k+1$. Find new reconstruction values $\left\{Y_i^{(k)}\right\}_{i=1}^{M}$ that are the centroids of $\left\{V_i^{(k-1)}\right\}$. Go to Step 2.

This algorithm is not very practical because the integrals required to compute the distortions and centroids are over odd-shaped regions in n dimensions, where n is the dimension of the input vectors. Generally, these integrals are extremely difficult to compute, making this particular algorithm more of an academic interest.

Of more practical interest is the algorithm for the case where we have a training set available. In this case, the algorithm looks very much like the k-means algorithm.

1. Start with an initial set of reconstruction values $\left\{Y_i^{(0)}\right\}_{i=1}^{M}$ and a set of training vectors $\{X_n\}_{n=1}^{N}$. Set $k = 0$, $D^{(0)} = 0$. Select threshold ϵ.
2. The quantization regions $\left\{V_i^{(k)}\right\}_{i=1}^{M}$ are given by

$$V_i^{(k)} = \{X_n : d(X_n, Y_i) < d(X_n, Y_j)\ \forall j \neq i\} \qquad i = 1, 2, \ldots, M.$$

   We assume that none of the quantization regions are empty. (Later we will deal with the case where $V_i^{(k)}$ is empty for some i and k.)
3. Compute the average distortion $D^{(k)}$ between the training vectors and the representative reconstruction value.
4. If $\frac{(D^{(k)}-D^{(k-1)})}{D^{(k)}} < \epsilon$, stop; otherwise, continue.
5. $k = k+1$. Find new reconstruction values $\left\{Y_i^{(k)}\right\}_{i=1}^{M}$ that are the average value of the elements of each of the quantization regions $V_i^{(k-1)}$. Go to Step 2.

This algorithm forms the basis of most vector quantizer designs. It is popularly known as the Linde-Buzo-Gray or LBG algorithm, or the generalized Lloyd algorithm (GLA) [125]. Although the paper of Linde, Buzo, and Gray [125] is a starting point for most of the work on vector quantization, the latter algorithm had been used several years prior by Edward E. Hilbert at the NASA Jet Propulsion Laboratories in Pasadena, California. Hilbert's starting point was the idea of clustering, and although he arrived at the same algorithm as described above, he called it the *cluster compression algorithm* [126].

In order to see how this algorithm functions, consider the following example of a two-dimensional vector quantizer codebook design.

Example 10.4.1:

Suppose our training set consists of the height and weight values shown in Table 10.1. The initial set of output points is shown in Table 10.2. (For ease of presentation, we will always round the coordinates of the output points to the nearest integer.) The inputs, outputs, and quantization regions are shown in Figure 10.7.

TABLE 10.1 Training set for designing vector quantizer codebook.

Height	Weight
72	180
65	120
59	119
64	150
65	162
57	88
72	175
44	41
62	114
60	110
56	91
70	172

TABLE 10.2 Initial set of output points for codebook design.

Height	Weight
45	50
75	117
45	117
80	180

The input (44, 41) has been assigned to the first output point; the inputs (56, 91), (57, 88), (59, 119), and (60, 110) have been assigned to the second output point; the inputs (62, 114), and (65, 120) have been assigned to the third output; and the five remaining vectors from the training set have been assigned to the fourth output. The distortion for this assignment is 387.25. We now find the new output points. There is only one vector in the first quantization region, so the first output point is (44, 41). The average of the four vectors in the second quantization region (rounded up) is the vector (58, 102), which is the new second output point. In a similar manner, we can compute the third and fourth output points as (64, 117) and (69, 168). The new output points and the corresponding quantization regions are shown in Figure 10.8. From Figure 10.8, we can see that, while the training vectors that were initially part of the first and fourth quantization regions are still in the same quantization regions, the training vectors (59,115) and (60,120), which were in quantization region 2, are now in quantization region 3. The distortion corresponding to this assignment of training vectors to quantization regions is 89, considerably less than the original 387.25. Given the new assignments, we can obtain a new set of output points. The first and fourth output points do not change because the training vectors in the corresponding regions have not changed. However, the training vectors in regions 2 and 3 have changed. Recomputing the output points for these regions, we get (57, 90) and (62, 116). The final form of the

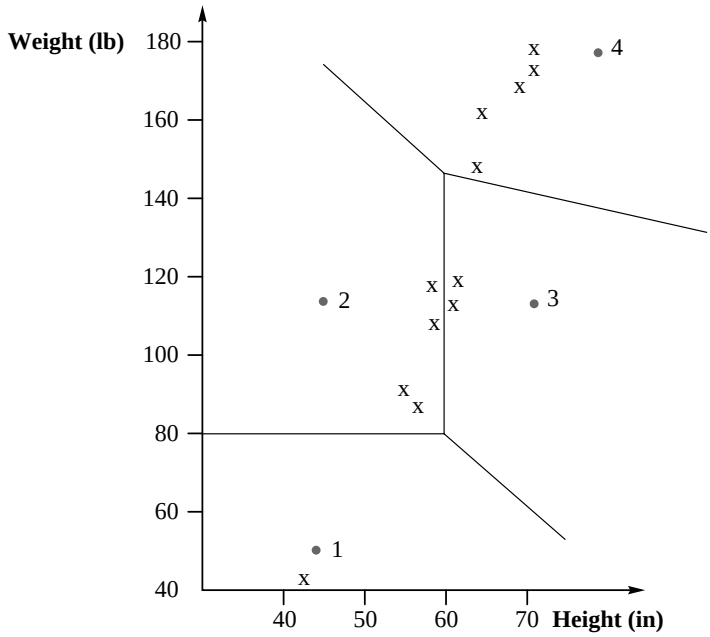

FIGURE 10.7 **Initial state of the vector quantizer.**

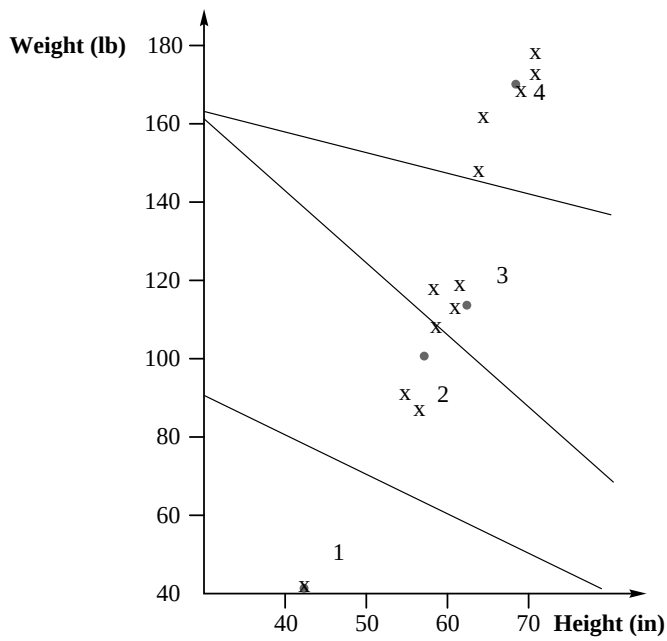

FIGURE 10.8 **The vector quantizer after one iteration.**

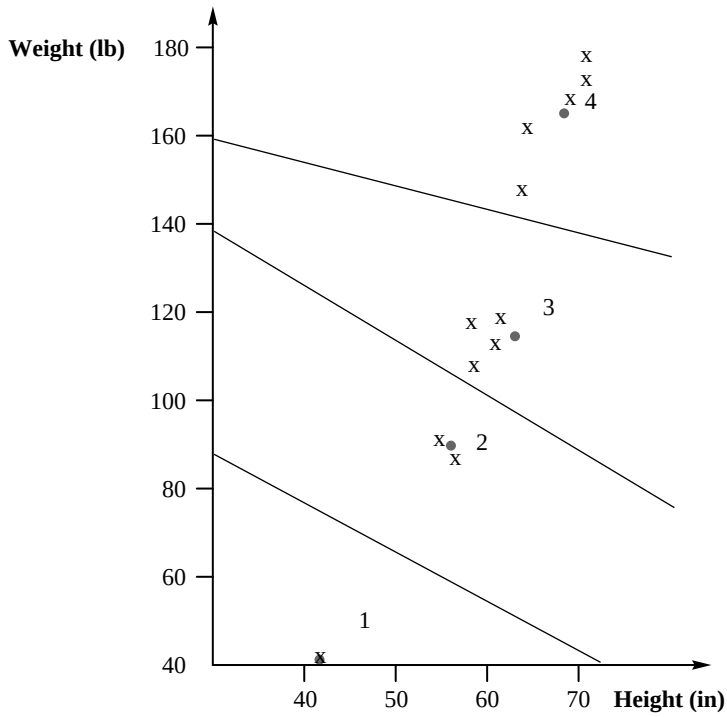

FIGURE 10.9 Final state of the vector quantizer.

quantizer is shown in Figure 10.9. The distortion corresponding to the final assignments is 60.17. ♦

The LBG algorithm is conceptually simple, and as we shall see later, the resulting vector quantizer is remarkably effective in the compression of a wide variety of inputs, both by itself and in conjunction with other schemes. In the next two sections we will look at some of the details of the codebook design process. While these details are important to consider when designing codebooks, they are not necessary for the understanding of the quantization process. If you are not currently interested in these details, you may wish to proceed directly to Section 10.4.3.

10.4.1 Initializing the LBG Algorithm

The LBG algorithm guarantees that the distortion from one iteration to the next will not increase. However, there is no guarantee that the procedure will converge to the optimal solution. The solution to which the algorithm converges is heavily dependent on the initial conditions. For example, if our initial set of output points in Example 10.4 had been those

TABLE 10.3 An alternate initial set of output points.

Height	Weight
75	50
75	117
75	127
80	180

TABLE 10.4 Final codebook obtained using the alternative initial codebook.

Height	Weight
44	41
60	107
64	150
70	172

shown in Table 10.3 instead of the set in Table 10.2, by using the LBG algorithm we would get the final codebook shown in Table 10.4.

The resulting quantization regions and their membership are shown in Figure 10.10. This is a very different quantizer than the one we had previously obtained. Given this heavy dependence on initial conditions, the selection of the initial codebook is a matter of some importance. We will look at some of the better-known methods of initialization in the following section.

Linde, Buzo, and Gray described a technique in their original paper [125] called the *splitting technique* for initializing the design algorithm. In this technique, we begin by designing a vector quantizer with a single output point; in other words, a codebook of size one, or a one-level vector quantizer. With a one-element codebook, the quantization region is the entire input space, and the output point is the average value of the entire training set. From this output point, the initial codebook for a two-level vector quantizer can be obtained by including the output point for the one-level quantizer and a second output point obtained by adding a fixed perturbation vector ϵ. We then use the LBG algorithm to obtain the two-level vector quantizer. Once the algorithm has converged, the two codebook vectors are used to obtain the initial codebook of a four-level vector quantizer. This initial four-level codebook consists of the two codebook vectors from the final codebook of the two-level vector quantizer and another two vectors obtained by adding ϵ to the two codebook vectors. The LBG algorithm can then be used until this four-level quantizer converges. In this manner we keep doubling the number of levels until we reach the desired number of levels. By including the final codebook of the previous stage at each "splitting," we guarantee that the codebook after splitting will be at least as good as the codebook prior to splitting.

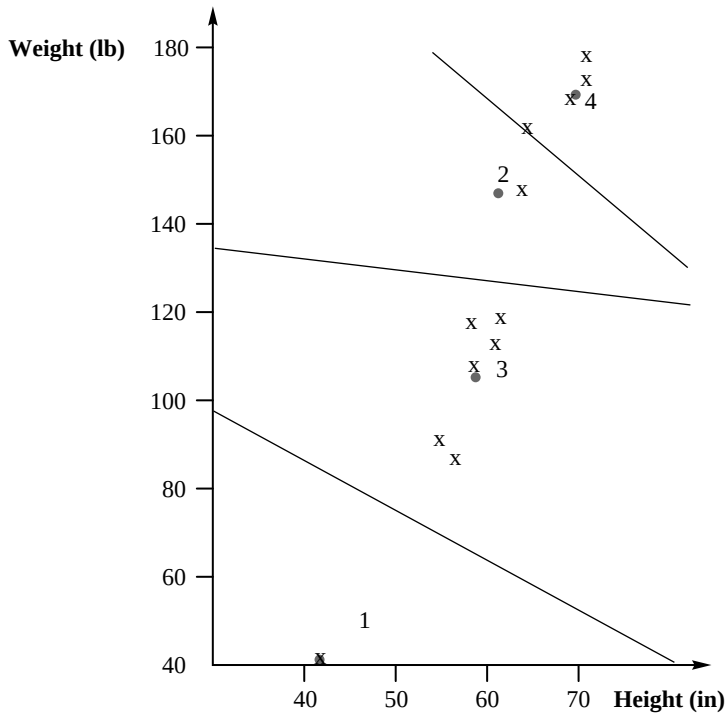

FIGURE 10. 10 **Final state of the vector quantizer.**

Example 10.4.2:

Let's revisit Example 10.4.1. This time, instead of using the initial codewords used in Example 10.4.1, we will use the splitting technique. For the perturbations, we will use a fixed vector $\epsilon = (10, 10)$. The perturbation vector is usually selected randomly; however, for purposes of explanation it is more useful to use a fixed perturbation vector.

We begin with a single-level codebook. The codeword is simply the average value of the training set. The progression of codebooks is shown in Table 10.5.

The perturbed vectors are used to initialize the LBG design of a two-level vector quantizer. The resulting two-level vector quantizer is shown in Figure 10.11. The resulting distortion is 468.58. These two vectors are perturbed to get the initial output points for the four-level design. Using the LBG algorithm, the final quantizer obtained is shown in Figure 10.12. The distortion is 156.17. The average distortion for the training set for this quantizer using the splitting algorithm is higher than the average distortion obtained previously. However, because the sample size used in this example is rather small, this is no indication of relative merit. ♦

TABLE 10.5 Progression of codebooks using splitting.

Codebook	Height	Weight
One-level	62	127
Initial two-level	62	127
	72	137
Final two-level	58	98
	69	168
Initial four-level	58	98
	68	108
	69	168
	79	178
Final four-level	52	73
	62	116
	65	156
	71	176

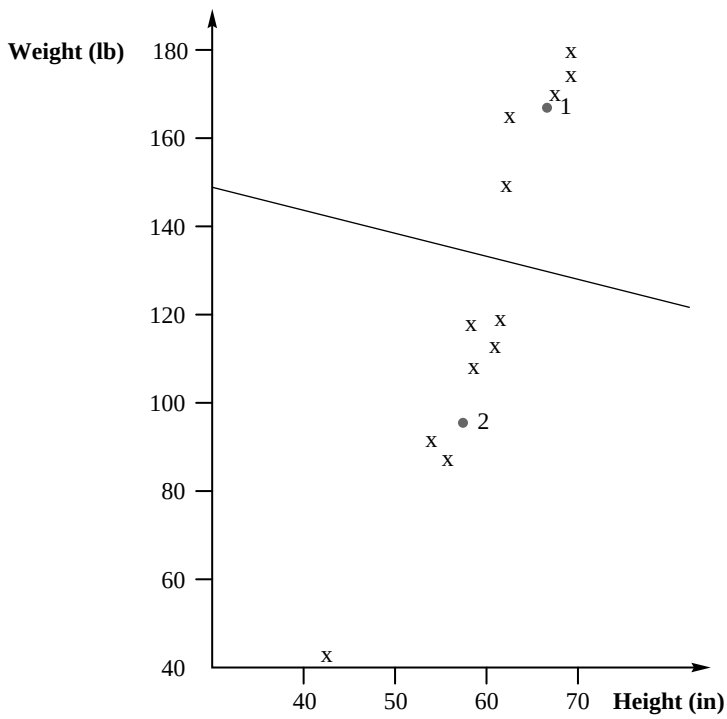

FIGURE 10.11 Two-level vector quantizer using splitting approach.

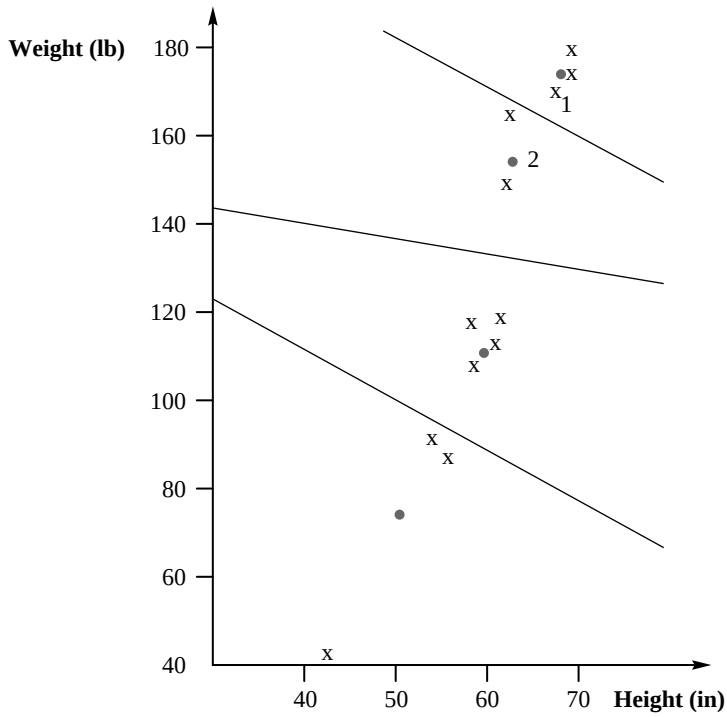

FIGURE 10. 12 Final design using the splitting approach.

If the desired number of levels is not a power of two, then in the last step, instead of generating two initial points from each of the output points of the vector quantizer designed previously, we can perturb as many vectors as necessary to obtain the desired number of vectors. For example, if we needed an eleven-level vector quantizer, we would generate a one-level vector quantizer first, then a two-level, then a four-level, and then an eight-level vector quantizer. At this stage, we would perturb only three of the eight vectors to get the eleven initial output points of the eleven-level vector quantizer. The three points should be those with the largest number of training set vectors, or the largest distortion.

The approach used by Hilbert [126] to obtain the initial output points of the vector quantizer was to pick the output points randomly from the training set. This approach guarantees that, in the initial stages, there will always be at least one vector from the training set in each quantization region. However, we can still get different codebooks if we use different subsets of the training set as our initial codebook.

Example 10.4.3:

Using the training set of Example 10.4.1, we selected different vectors of the training set as the initial codebook. The results are summarized in Table 10.6. If we pick the codebook labeled "Initial Codebook 1," we obtain the codebook labeled "Final Codebook 1." This

TABLE 10.6 Effect of using different subsets of the training sequence as the initial codebook.

Codebook	Height	Weight
Initial Codebook 1	72	180
	72	175
	65	120
	59	119
Final Codebook 1	71	176
	65	156
	62	116
	52	73
Initial Codebook 2	65	120
	44	41
	59	119
	57	88
Final Codebook 2	69	168
	44	41
	62	116
	57	90

codebook is identical to the one obtained using the split algorithm. The set labeled "Initial Codebook 2" results in the codebook labeled "Final Codebook 2." This codebook is identical to the quantizer we obtained in Example 10.4.1. In fact, most of the other selections result in one of these two quantizers. ♦

Notice that by picking different subsets of the input as our initial codebook, we can generate different vector quantizers. A good approach to codebook design is to initialize the codebook randomly several times, and pick the one that generates the least distortion in the training set from the resulting quantizers.

In 1989, Equitz [127] introduced a method for generating the initial codebook called the *pairwise nearest neighbor* (PNN) algorithm. In the PNN algorithm, we start with as many clusters as there are training vectors and end with the initial codebook. At each stage, we combine the two closest vectors into a single cluster and replace the two vectors by their mean. The idea is to merge those clusters that would result in the smallest increase in distortion. Equitz showed that when we combine two clusters C_i and C_j, the increase in distortion is

$$\frac{n_i n_j}{n_i + n_j} \left\| Y_i - Y_j \right\|^2, \tag{10.5}$$

where n_i is the number of elements in the cluster C_i, and Y_i is the corresponding output point. In the PNN algorithm, we combine clusters that cause the smallest increase in the distortion.

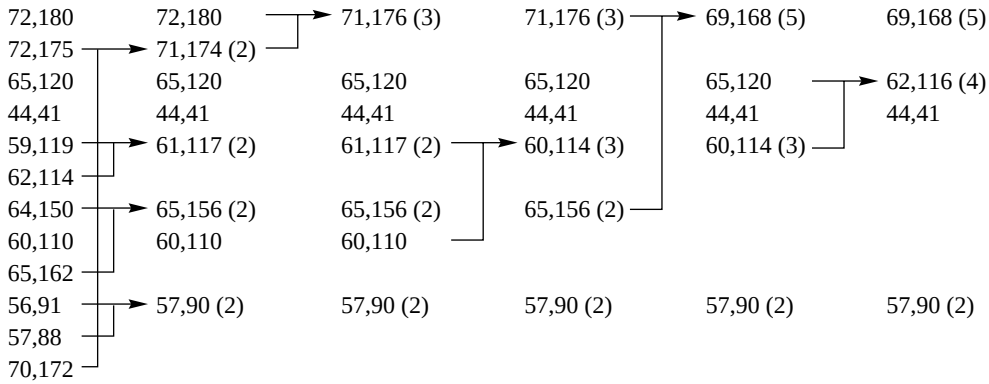

FIGURE 10. 13 Obtaining initial output points using the PNN approach.

Example 10.4.4:

Using the PNN algorithm, we combine the elements in the training set as shown in Figure 10.13. At each step we combine the two clusters that are closest in the sense of Equation (10.5). If we use these values to initialize the LBG algorithm, we get a vector quantizer shown with output points (70, 172), (60, 107), (44, 41), (64, 150), and a distortion of 104.08. ♦

Although it was a relatively easy task to generate the initial codebook using the PNN algorithm in Example 10.4.4, we can see that, as the size of the training set increases, this procedure becomes progressively more time-consuming. In order to avoid this cost, we can use a fast PNN algorithm that does not attempt to find the absolute smallest cost at each step (see [127] for details).

Finally, a simple initial codebook is the set of output points from the corresponding scalar quantizers. In the beginning of this chapter we saw how scalar quantization of a sequence of inputs can be viewed as vector quantization using a rectangular vector quantizer. We can use this rectangular vector quantizer as the initial set of outputs.

Example 10.4.5:

Return once again to the quantization of the height-weight data set. If we assume that the heights are uniformly distributed between 40 and 180, then a two-level scalar quantizer would have reconstruction values 75 and 145. Similarly, if we assume that the weights are uniformly distributed between 40 and 80, the reconstruction values would be 50 and 70. The initial reconstruction values for the vector quantizer are (50, 75), (50, 145), (70, 75), and (70, 145). The final design for this initial set is the same as the one obtained in Example 10.4.1 with a distortion of 60.17. ♦

We have looked at four different ways of initializing the LBG algorithm. Each has its own advantages and drawbacks. The PNN initialization has been shown to result in better designs, producing a lower distortion for a given rate than the splitting approach [127]. However, the procedure for obtaining the initial codebook is much more involved and complex. We cannot make any general claims regarding the superiority of any one of these initialization techniques. Even the PNN approach cannot be proven to be optimal. In practice, if we are dealing with a wide variety of inputs, the effect of using different initialization techniques appears to be insignificant.

10.4.2 The Empty Cell Problem

Let's take a closer look at the progression of the design in Example 10.4.5. When we assign the inputs to the initial output points, no input point gets assigned to the output point at (70, 75). This is a problem because in order to update an output point, we need to take the average value of the input vectors. Obviously, some strategy is needed. The strategy that we actually used in Example 10.4.5 was not to update the output point if there were no inputs in the quantization region associated with it. This strategy seems to have worked in this particular example; however, there is a danger that we will end up with an output point that is never used. A common approach to avoid this is to remove an output point that has no inputs associated with it, and replace it with a point from the quantization region with the most output points. This can be done by selecting a point at random from the region with the highest population of training vectors, or the highest associated distortion. A more systematic approach is to design a two-level quantizer for the training vectors in the most heavily populated quantization region. This approach is computationally expensive and provides no significant improvement over the simpler approach. In the program accompanying this book, we have used the first approach. (To compare the two approaches, see Problem 3.)

10.4.3 Use of LBG for Image Compression

One application for which the vector quantizer described in this section has been extremely popular is image compression. For image compression, the vector is formed by taking blocks of pixels of size $N \times M$ and treating them as an $L = NM$ dimensional vector. Generally, we take $N = M$. Instead of forming vectors in this manner, we could form the vector by taking L pixels in a row of the image. However, this does not allow us to take advantage of the two-dimensional correlations in the image. Recall that correlation between the samples provides the clustering of the input, and the LBG algorithm takes advantage of this clustering.

Example 10.4.6:

Let us quantize the Sinan image shown in Figure 10.14 using a 16-dimensional quantizer. The input vectors are constructed using 4×4 blocks of pixels. The codebook was trained on the Sinan image.

The results of the quantization using codebooks of size 16, 64, 256, and 1024 are shown in Figure 10.15. The rates and compression ratios are summarized in Table 10.7. To see how these quantities were calculated, recall that if we have K vectors in a codebook, we need

FIGURE 10. 14 Original Sinan image.

$\lceil \log_2 K \rceil$ bits to inform the receiver which of the K vectors is the quantizer output. This quantity is listed in the second column of Table 10.7 for the different values of K. If the vectors are of dimension L, this means that we have used $\lceil \log_2 K \rceil$ bits to send the quantized value of L pixels. Therefore, the rate in bits per pixel is $\frac{\lceil \log_2 K \rceil}{L}$. (We have assumed that the codebook is available to both transmitter and receiver, and therefore we do not have to use any bits to transmit the codebook from the transmitter to the receiver.) This quantity is listed in the third column of Table 10.7. Finally, the compression ratio, given in the last column of Table 10.7, is the ratio of the number of bits per pixel in the original image to the number of bits per pixel in the compressed image. The Sinan image was digitized using 8 bits per pixel. Using this information and the rate after compression, we can obtain the compression ratios.

Looking at the images, we see that reconstruction using a codebook of size 1024 is very close to the original. At the other end, the image obtained using a codebook with 16 reconstruction vectors contains a lot of visible artifacts. The utility of each reconstruction depends on the demands of the particular application. ♦

In this example, we used codebooks trained on the image itself. Generally, this is not the preferred approach because the receiver has to have the same codebook in order to reconstruct the image. Either the codebook must be transmitted along with the image, or the receiver has the same training image so that it can generate an identical codebook. This is impractical because, if the receiver already has the image in question, much better compression can be obtained by simply sending the name of the image to the receiver. Sending the codebook with the image is not unreasonable. However, the transmission of

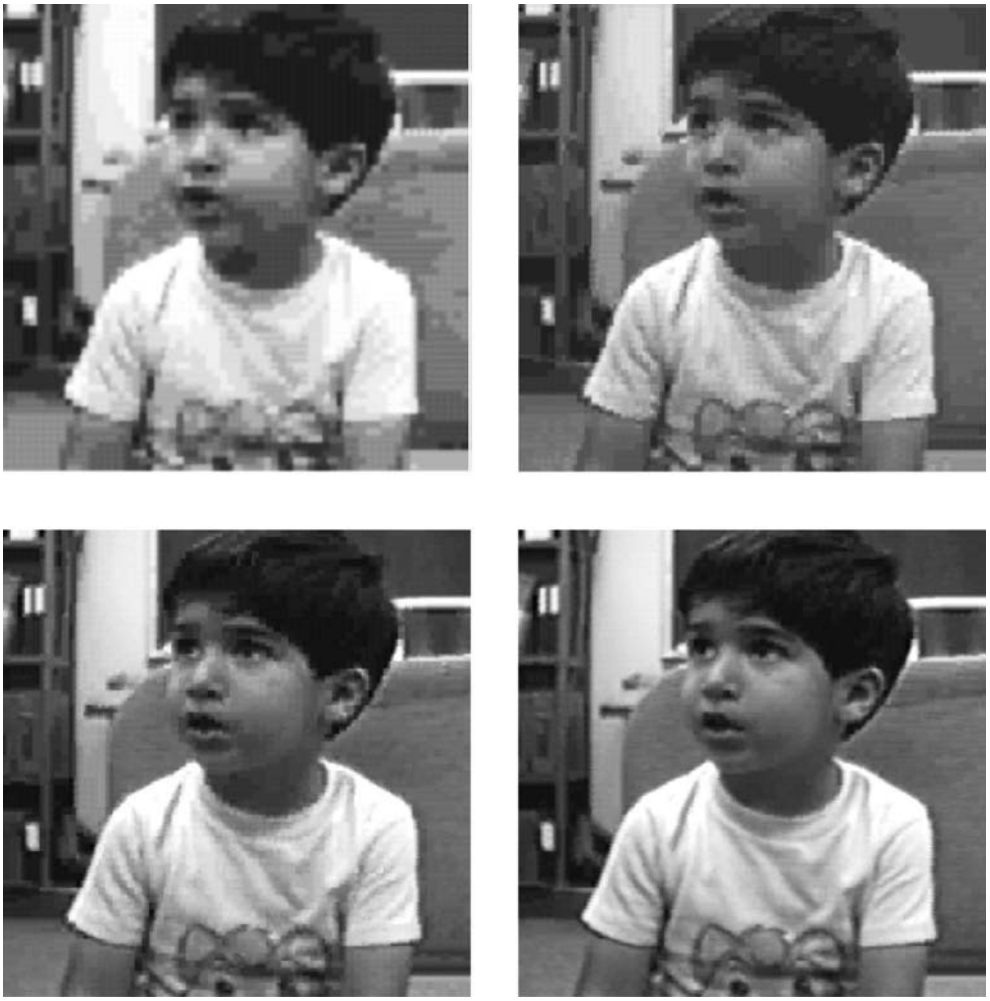

FIGURE 10.15 **Top left: codebook size 16; top right: codebook size 64; bottom left: codebook size 256; bottom right: codebook size 1024.**

TABLE 10.7 **Summary of compression measures for image compression example.**

Codebook Size (# of codewords)	Bits Needed to Select a Codeword	Bits per Pixel	Compression Ratio
16	4	0.25	32:1
64	6	0.375	21.33:1
256	8	0.50	16:1
1024	10	0.625	12.8:1

TABLE 10.8 Overhead in bits per pixel for codebooks of different sizes.

Codebook Size K	Overhead in Bits per Pixel
16	0.03125
64	0.125
256	0.50
1024	2.0

the codebook is overhead that could be avoided if a more generic codebook, one that is available to both transmitter and receiver, were to be used.

In order to compute the overhead, we need to calculate the number of bits required to transmit the codebook to the receiver. If each codeword in the codebook is a vector with L elements and if we use B bits to represent each element, then in order to transmit the codebook of a K-level quantizer we need $B \times L \times K$ bits. In our example, $B = 8$ and $L = 16$. Therefore, we need $K \times 128$ bits to transmit the codebook. As our image consists of 256×256 pixels, the overhead in bits per pixel is $128K/65,536$. The overhead for different values of K is summarized in Table 10.8. We can see that while the overhead for a codebook of size 16 seems reasonable, the overhead for a codebook of size 1024 is over three times the rate required for quantization.

Given the excessive amount of overhead required for sending the codebook along with the vector quantized image, there has been substantial interest in the design of codebooks that are more generic in nature and, therefore, can be used to quantize a number of images. To investigate the issues that might arise, we quantized the Sinan image using four different codebooks generated by the Sena, Sensin, Earth, and Omaha images. The results are shown in Figure 10.16.

As expected, the reconstructed images from this approach are not of the same quality as when the codebook is generated from the image to be quantized. However, this is only true as long as the overhead required for storage or transmission of the codebook is ignored. If we include the extra rate required to encode and transmit the codebook of output points, using the codebook generated by the image to be quantized seems unrealistic. Although using the codebook generated by another image to perform the quantization may be realistic, the quality of the reconstructions is quite poor. Later in this chapter we will take a closer look at the subject of vector quantization of images and consider a variety of ways to improve this performance.

You may have noticed that the bit rates for the vector quantizers used in the examples are quite low. The reason is that the size of the codebook increases exponentially with the rate. Suppose we want to encode a source using R bits per sample; that is, the average number of bits per sample in the compressed source output is R. By "sample" we mean a scalar element of the source output sequence. If we wanted to use an L-dimensional quantizer, we would group L samples together into vectors. This means that we would have RL bits available to represent each vector. With RL bits, we can represent 2^{RL} different output vectors. In other words, the size of the codebook for an L-dimensional R-bits-per-sample quantizer is 2^{RL}. From Table 10.7, we can see that when we quantize an image using 0.25 bits per pixel and 16-dimensional quantizers, we have $16 \times 0.25 = 4$ bits available to represent each

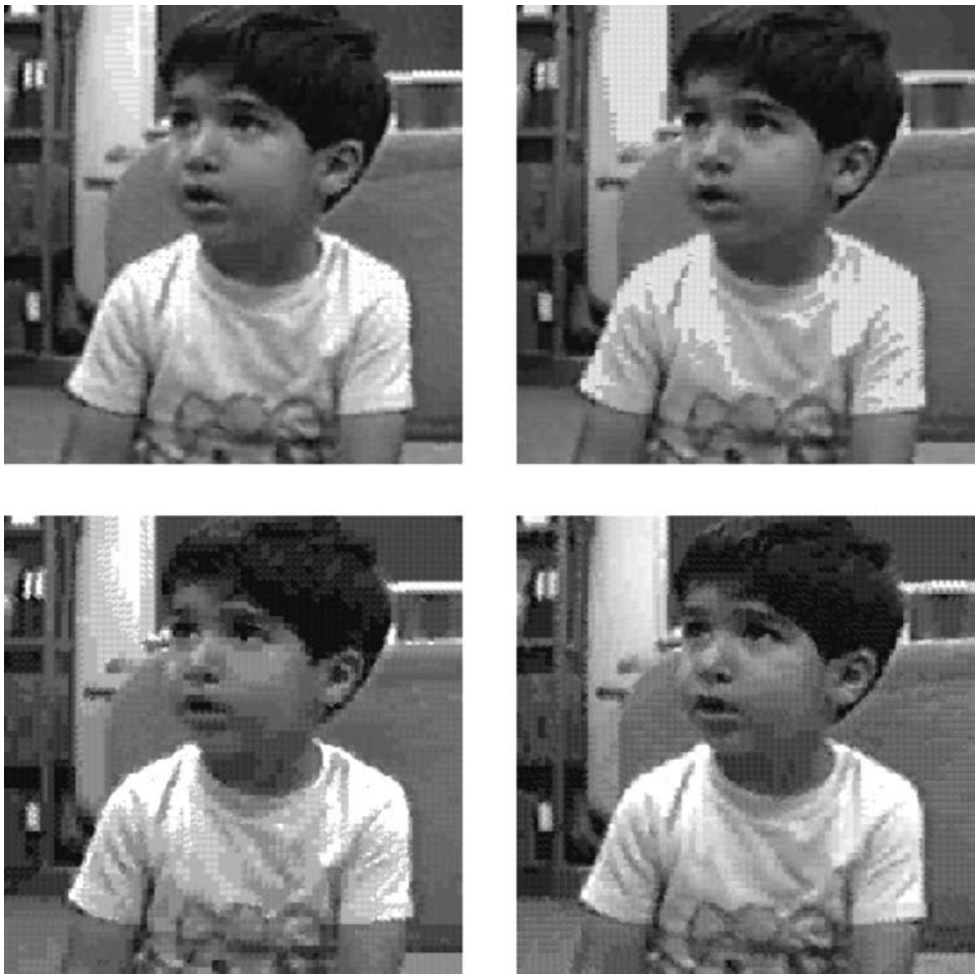

FIGURE 10. 16 **Sinan image quantized at the rate of 0.5 bits per pixel. The images used to obtain the codebook were (clockwise from top left) Sensin, Sena, Earth, Omaha.**

vector. Hence, the size of the codebook is $2^4 = 16$. The quantity RL is often called the *rate dimension product*. Note that the size of the codebook grows exponentially with this product.

Consider the problems. The codebook size for a 16-dimensional, 2-bits-per-sample vector quantizer would be $2^{16\times 2}$! (If the source output was originally represented using 8 bits per sample, a rate of 2 bits per sample for the compressed source corresponds to a compression ratio of 4:1.) This large size causes problems both with storage and with the quantization process. To store 2^{32} sixteen-dimensional vectors, assuming that we can store each component of the vector in a single byte, requires $2^{32} \times 16$ bytes—approximately 64 gigabytes of storage. Furthermore, to quantize a single input vector would require over four billion vector

comparisons to find the closest output point. Obviously, neither the storage requirements nor the computational requirements are realistic. Because of this problem, most vector quantization applications operate at low bit rates. In many applications, such as low-rate speech coding, we want to operate at very low rates; therefore, this is not a drawback. However, for applications such as high-quality video coding, which requires higher rates, this is definitely a problem.

There are several approaches to solving these problems. Each entails the introduction of some structure in the codebook and/or the quantization process. While the introduction of structure mitigates some of the storage and computational problems, there is generally a trade-off in terms of the distortion performance. We will look at some of these approaches in the following sections.

10.5 Tree-Structured Vector Quantizers

One way we can introduce structure is to organize our codebook in such a way that it is easy to pick which part contains the desired output vector. Consider the two-dimensional vector quantizer shown in Figure 10.17. Note that the output points in each quadrant are the mirror image of the output points in neighboring quadrants. Given an input to this vector quantizer, we can reduce the number of comparisons necessary for finding the closest output point by using the sign on the components of the input. The sign on the components of the input vector will tell us in which quadrant the input lies. Because all the quadrants are mirror images of the neighboring quadrants, the closest output point to a given input will lie in the same quadrant as the input itself. Therefore, we only need to compare the input to the output points that lie in the same quadrant, thus reducing the number of required comparisons by a factor of four. This approach can be extended to L dimensions, where the signs on the L components of the input vector can tell us in which of the 2^L hyperquadrants the input lies, which in turn would reduce the number of comparisons by 2^L.

This approach works well when the output points are distributed in a symmetrical manner. However, it breaks down as the distribution of the output points becomes less symmetrical.

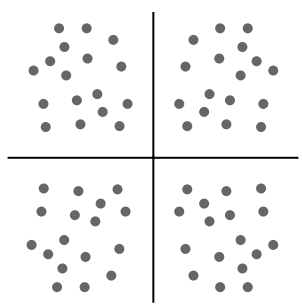

FIGURE 10. 17 **A symmetrical vector quantizer in two dimensions.**

Example 10.5.1:

Consider the vector quantizer shown in Figure 10.18. This is different from the output points in Figure 10.17; we have dropped the mirror image requirement of the previous example. The output points are shown as filled circles, and the input point is the X. It is obvious from the figure that while the input is in the first quadrant, the closest output point is in the fourth quadrant. However, the quantization approach described above will force the input to be represented by an output in the first quadrant.

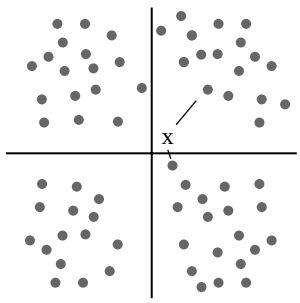

FIGURE 10.18 Breakdown of the method using the quadrant approach.

The situation gets worse as we lose more and more of the symmetry. Consider the situation in Figure 10.19. In this quantizer, not only will we get an incorrect output point when the input is close to the boundaries of the first quadrant, but also there is no significant reduction in the amount of computation required.

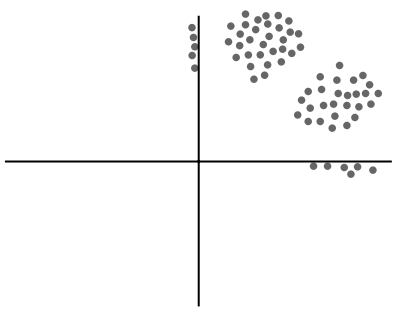

FIGURE 10.19 Breakdown of the method using the quadrant approach.

Most of the output points are in the first quadrant. Therefore, whenever the input falls in the first quadrant, which it will do quite often if the quantizer design is reflective of the distribution of the input, knowing that it is in the first quadrant does not lead to a great reduction in the number of comparisons. ♦

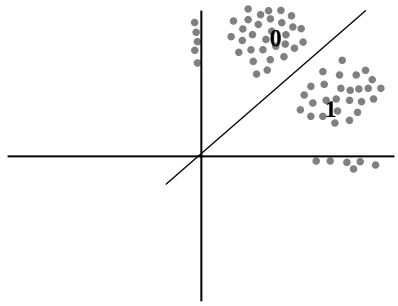

FIGURE 10. 20 Division of output points into two groups.

The idea of using the L-dimensional equivalents of quadrants to partition the output points in order to reduce the computational load can be extended to nonsymmetrical situations, like those shown in Figure 10.19, in the following manner. Divide the set of output points into two groups, *group0* and *group1*, and assign to each group a test vector such that output points in each group are closer to the test vector assigned to that group than to the test vector assigned to the other group (Figure 10.20). Label the two test vectors 0 and 1. When we get an input vector, we compare it against the test vectors. Depending on the outcome, the input is compared to the output points associated with the test vector closest to the input. After these two comparisons, we can discard half of the output points. Comparison with the test vectors takes the place of looking at the signs of the components to decide which set of output points to discard from contention. If the total number of output points is K, with this approach we have to make $\frac{K}{2}+2$ comparisons instead of K comparisons.

This process can be continued by splitting the output points in each group into two groups and assigning a test vector to the subgroups. So *group0* would be split into *group00* and *group01*, with associated test vectors labeled 00 and 01, and *group1* would be split into *group10* and *group11*, with associated test vectors labeled 10 and 11. Suppose the result of the first set of comparisons was that the output point would be searched for in *group1*. The input would be compared to the test vectors 10 and 11. If the input was closer to the test vector 10, then the output points in *group11* would be discarded, and the input would be compared to the output points in *group10*. We can continue the procedure by successively dividing each group of output points into two, until finally, if the number of output points is a power of two, the last set of groups would consist of single points. The number of comparisons required to obtain the final output point would be $2\log K$ instead of K. Thus, for a codebook of size 4096 we would need 24 vector comparisons instead of 4096 vector comparisons.

This is a remarkable decrease in computational complexity. However, we pay for this decrease in two ways. The first penalty is a possible increase in distortion. It is possible at some stage that the input is closer to one test vector while at the same time being closest to an output belonging to the rejected group. This is similar to the situation shown in Figure 10.18. The other penalty is an increase in storage requirements. Now we not only have to store the output points from the vector quantizer codebook, we also must store the test vectors. This means almost a doubling of the storage requirement.

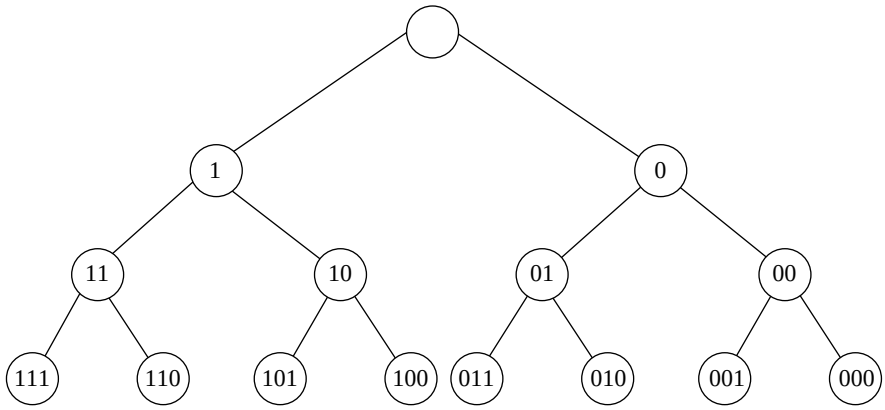

FIGURE 10.21 Decision tree for quantization.

The comparisons that must be made at each step are shown in Figure 10.21. The label inside each node is the label of the test vector that we compare the input against. This tree of decisions is what gives tree-structured vector quantizers (TSVQ) their name. Notice also that, as we are progressing down a tree, we are also building a binary string. As the leaves of the tree are the output points, by the time we reach a particular leaf or, in other words, select a particular output point, we have obtained the binary codeword corresponding to that output point.

This process of building the binary codeword as we progress through the series of decisions required to find the final output can result in some other interesting properties of tree-structured vector quantizers. For instance, even if a partial codeword is transmitted, we can still get an approximation of the input vector. In Figure 10.21, if the quantized value was the codebook vector 5, the binary codeword would be 011. However, if only the first two bits 01 were received by the decoder, the input can be approximated by the test vector labeled 01.

10.5.1 Design of Tree-Structured Vector Quantizers

In the last section we saw how we could reduce the computational complexity of the design process by imposing a tree structure on the vector quantizer. Rather than imposing this structure after the vector quantizer has been designed, it makes sense to design the vector quantizer within the framework of the tree structure. We can do this by a slight modification of the splitting design approach proposed by Linde et al. [125].

We start the design process in a manner identical to the splitting technique. First, obtain the average of all the training vectors, perturb it to obtain a second vector, and use these vectors to form a two-level vector quantizer. Let us label these two vectors 0 and 1, and the groups of training set vectors that would be quantized to each of these two vectors *group0* and *group1*. We will later use these vectors as test vectors. We perturb these output points to get the initial vectors for a four-level vector quantizer. At this point, the design procedure

for the tree-structured vector quantizer deviates from the splitting technique. Instead of using the entire training set to design a four-level vector quantizer, we use the training set vectors in *group0* to design a two-level vector quantizer with output points labeled 00 and 01. We use the training set vectors in *group1* to design a two-level vector quantizer with output points labeled 10 and 11. We also split the training set vectors in *group0* and *group1* into two groups each. The vectors in *group0* are split, based on their proximity to the vectors labeled 00 and 01, into *group00* and *group01*, and the vectors in *group1* are divided in a like manner into the groups *group10* and *group11*. The vectors labeled 00, 01, 10, and 11 will act as test vectors at this level. To get an eight-level quantizer, we use the training set vectors in each of the four groups to obtain four two-level vector quantizers. We continue in this manner until we have the required number of output points. Notice that in the process of obtaining the output points, we have also obtained the test vectors required for the quantization process.

10.5.2 Pruned Tree-Structured Vector Quantizers

Once we have built a tree-structured codebook, we can sometimes improve its rate distortion performance by removing carefully selected subgroups. Removal of a subgroup, referred to as *pruning*, will reduce the size of the codebook and hence the rate. It may also result in an increase in distortion. Therefore, the objective of the pruning is to remove those subgroups that will result in the best trade-off of rate and distortion. Chou, Lookabaugh, and Gray [128] have developed an optimal pruning algorithm called the *generalized BFOS algorithm*. The name of the algorithm derives from the fact that it is an extension of an algorithm originally developed by Brieman, Freidman, Olshen, and Stone [129] for classification applications. (See [128] and [5] for description and discussion of the algorithm.)

Pruning output points from the codebook has the unfortunate effect of removing the structure that was previously used to generate the binary codeword corresponding to the output points. If we used the structure to generate the binary codewords, the pruning would cause the codewords to be of variable length. As the variable-length codes would correspond to the leaves of a binary tree, this code would be a prefix code and, therefore, certainly usable. However, it would not require a large increase in complexity to assign fixed-length codewords to the output points using another method. This increase in complexity is generally offset by the improvement in performance that results from the pruning [130].

10.6 Structured Vector Quantizers

The tree-structured vector quantizer solves the complexity problem, but acerbates the storage problem. We now take an entirely different tack and develop vector quantizers that do not have these storage problems; however, we pay for this relief in other ways.

Example 10.3.1 was our motivation for the quantizer obtained by the LBG algorithm. This example showed that the correlation between samples of the output of a source leads to clustering. This clustering is exploited by the LBG algorithm by placing output points at the location of these clusters. However, in Example 10.3.2, we saw that even when there

is no correlation between samples, there is a kind of probabilistic structure that becomes more evident as we group the random inputs of a source into larger and larger blocks or vectors.

In Example 10.3.2, we changed the position of the output point in the top-right corner. All four corner points have the same probability, so we could have chosen any of these points. In the case of the two-dimensional Laplacian distribution in Example 10.3.2, all points that lie on the contour described by $|x| + |y| = constant$ have equal probability. These are called *contours of constant probability*. For spherically symmetrical distributions like the Gaussian distribution, the contours of constant probability are circles in two dimensions, spheres in three dimensions, and hyperspheres in higher dimensions.

We mentioned in Example 10.3.2 that the points away from the origin have very little probability mass associated with them. Based on what we have said about the contours of constant probability, we can be a little more specific and say that the points on constant probability contours farther away from the origin have very little probability mass associated with them. Therefore, we can get rid of all of the points outside some contour of constant probability without incurring much of a distortion penalty. In addition as the number of reconstruction points is reduced, there is a decrease in rate, thus improving the rate distortion performance.

Example 10.6.1:

Let us design a two-dimensional uniform quantizer by keeping only the output points in the quantizer of Example 10.3.2 that lie on or within the contour of constant probability given by $|x_1| + |x_2| = 5\Delta$. If we count all the points that are retained, we get 60 points. This is close enough to 64 that we can compare it with the eight-level uniform scalar quantizer. If we simulate this quantization scheme with a Laplacian input, and the same step size as the scalar quantizer, that is, $\Delta = 0.7309$, we get an SNR of 12.22 dB. Comparing this to the 11.44 dB obtained with the scalar quantizer, we see that there is a definite improvement. We can get slightly more improvement in performance if we modify the step size. ♦

Notice that the improvement in the previous example is obtained only by restricting the outer boundary of the quantizer. Unlike Example 10.3.2, we did not change the shape of any of the inner quantization regions. This gain is referred to in the quantization literature as *boundary gain*. In terms of the description of quantization noise in Chapter 8, we reduced the overload error by reducing the overload probability, without a commensurate increase in the granular noise. In Figure 10.22, we have marked the 12 output points that belonged to the original 64-level quantizer, but do not belong to the 60-level quantizer, by drawing circles around them. Removal of these points results in an increase in overload probability. We also marked the eight output points that belong to the 60-level quantizer, but were not part of the original 64-level quantizer, by drawing squares around them. Adding these points results in a decrease in the overload probability. If we calculate the increases and decreases (Problem 5), we find that the net result is a decrease in overload probability. This overload probability is further reduced as the dimension of the vector is increased.

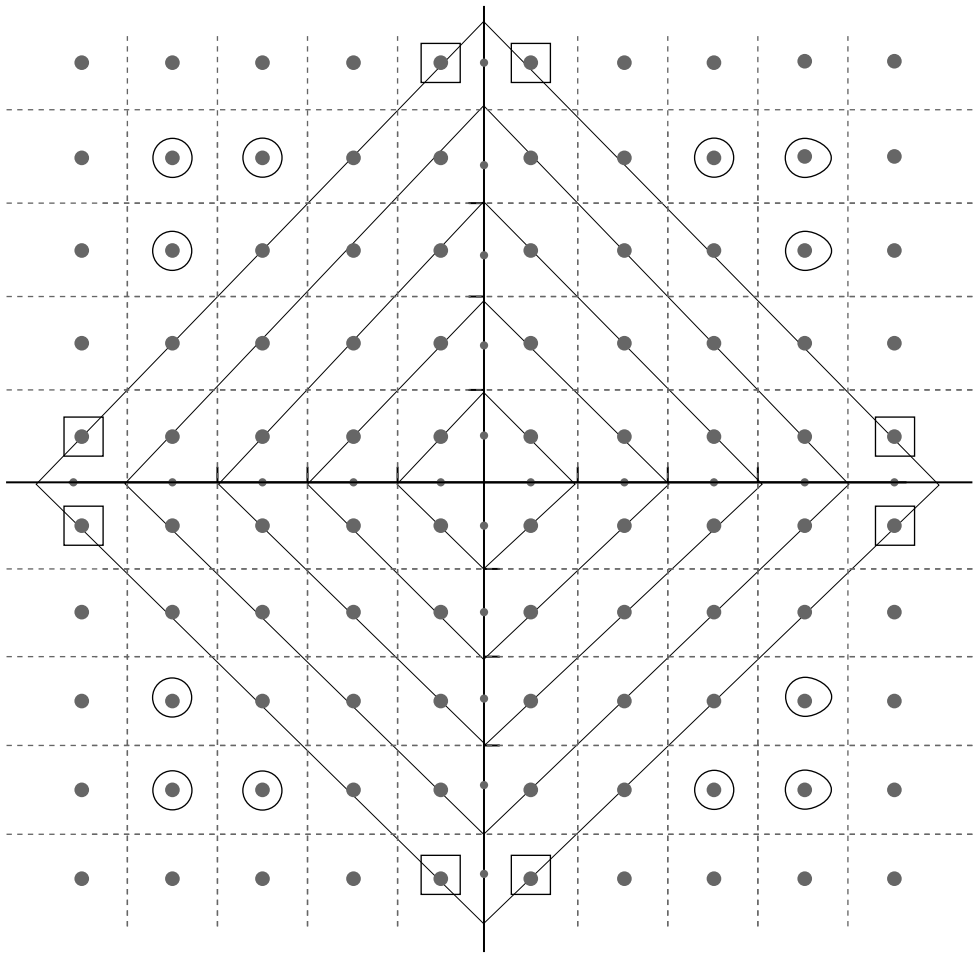

FIGURE 10. 22 Contours of constant probability.

10.6.1 Pyramid Vector Quantization

As the dimension of the input vector increases, something interesting happens. Suppose we are quantizing a random variable X with *pdf* $f_X(X)$ and differential entropy $h(X)$. Suppose we block samples of this random variable into a vector $\mathbf{X}$. A result of Shannon's, called the *asymptotic equipartition property* (AEP), states that for sufficiently large L and arbitrarily small ϵ

$$\left| \frac{\log f_{\mathbf{X}}(\mathbf{X})}{L} + h(X) \right| < \epsilon \tag{10.6}$$

for all but a set of vectors with a vanishingly small probability [7]. This means that almost all the L-dimensional vectors will lie on a contour of constant probability given by

$$\left|\frac{\log f_{\mathbf{X}}(\mathbf{X})}{L}\right| = -h(X). \tag{10.7}$$

Given that this is the case, Sakrison [131] suggested that an optimum manner to encode the source would be to distribute 2^{RL} points uniformly in this region. Fischer [132] used this insight to design a vector quantizer called the *pyramid vector quantizer* for the Laplacian source that looks quite similar to the quantizer described in Example 10.6.1. The vector quantizer consists of points of the rectangular quantizer that fall on the hyperpyramid given by

$$\sum_{i=1}^{L} |x_i| = C$$

where C is a constant depending on the variance of the input. Shannon's result is asymptotic, and for realistic values of L, the input vector is generally not localized to a single hyperpyramid.

For this case, Fischer first finds the distance

$$r = \sum_{i=1}^{L} |x_i|.$$

This value is quantized and transmitted to the receiver. The input is normalized by this gain term and quantized using a single hyperpyramid. The quantization process for the shape term consists of two stages: finding the output point on the hyperpyramid closest to the scaled input, and finding a binary codeword for this output point. (See [132] for details about the quantization and coding process.) This approach is quite successful, and for a rate of 3 bits per sample and a vector dimension of 16, we get an SNR value of 16.32 dB. If we increase the vector dimension to 64, we get an SNR value of 17.03. Compared to the SNR obtained from using a nonuniform scalar quantizer, this is an improvement of more than 4 dB.

Notice that in this approach we separated the input vector into a *gain* term and a pattern or *shape* term. Quantizers of this form are called *gain-shape vector quantizers*, or *product code vector quantizers* [133].

10.6.2 Polar and Spherical Vector Quantizers

For the Gaussian distribution, the contours of constant probability are circles in two dimensions and spheres and hyperspheres in three and higher dimensions. In two dimensions, we can quantize the input vector by first transforming it into polar coordinates r and θ:

$$r = \sqrt{x_1^2 + x_2^2} \tag{10.8}$$

and

$$\theta = \tan^{-1}\frac{x_2}{x_1}. \tag{10.9}$$

r and θ can then be either quantized independently [134], or we can use the quantized value of r as an index to a quantizer for θ [135]. The former is known as a polar quantizer; the latter, an unrestricted polar quantizer. The advantage to quantizing r and θ independently is one of simplicity. The quantizers for r and θ are independent scalar quantizers. However, the performance of the polar quantizers is not significantly higher than that of scalar quantization of the components of the two-dimensional vector. The unrestricted polar quantizer has a more complex implementation, as the quantization of θ depends on the quantization of r. However, the performance is also somewhat better than the polar quantizer. The polar quantizer can be extended to three or more dimensions [136].

10.6.3 Lattice Vector Quantizers

Recall that quantization error is composed of two kinds of error, overload error and granular error. The overload error is determined by the location of the quantization regions furthest from the origin, or the boundary. We have seen how we can design vector quantizers to reduce the overload probability and thus the overload error. We called this the boundary gain of vector quantization. In scalar quantization, the granular error was determined by the size of the quantization interval. In vector quantization, the granular error is affected by the size and shape of the quantization interval.

Consider the square and circular quantization regions shown in Figure 10.23. We show only the quantization region at the origin. These quantization regions need to be distributed in a regular manner over the space of source outputs. However, for now, let us simply consider the quantization region at the origin. Let's assume they both have the same area so that we can compare them. This way it would require the same number of quantization regions to cover a given area. That is, we will be comparing two quantization regions of the same "size." To have an area of one, the square has to have sides of length one. As the area of a circle is given by πr^2, the radius of the circle is $\frac{1}{\sqrt{\pi}}$. The maximum quantization error possible with the square quantization region is when the input is at one of the four corners of the square. In this case, the error is $\frac{1}{\sqrt{2}}$, or about 0.707. For the circular quantization region, the maximum error occurs when the input falls on the boundary of the circle. In this case, the error is $\frac{1}{\sqrt{\pi}}$, or about 0.56. Thus, the maximum granular error is larger for the square region than the circular region.

In general, we are more concerned with the average squared error than the maximum error. If we compute the average squared error for the square region, we obtain

$$\int_{\text{Square}} \|X\|^2 \, dX = 0.166\bar{6}.$$

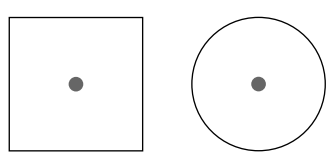

FIGURE 10. 23 **Possible quantization regions.**

For the circle, we obtain

$$\int_{\text{Circle}} \|X\|^2 \, dX = 0.159.$$

Thus, the circular region would introduce less granular error than the square region.

Our choice seems to be clear; we will use the circle as the quantization region. Unfortunately, a basic requirement for the quantizer is that for every possible input vector there should be a unique output vector. In order to satisfy this requirement and have a quantizer with sufficient structure that can be used to reduce the storage space, a union of translates of the quantization region should cover the output space of the source. In other words, the quantization region should *tile* space. A two-dimensional region can be tiled by squares, but it cannot be tiled by circles. If we tried to tile the space with circles, we would either get overlaps or holes.

Apart from squares, other shapes that tile space include rectangles and hexagons. It turns out that the best shape to pick for a quantization region in two dimensions is a hexagon [137].

In two dimensions, it is relatively easy to find the shapes that tile space, then select the one that gives the smallest amount of granular error. However, when we start looking at higher dimensions, it is difficult, if not impossible, to visualize different shapes, let alone find which ones tile space. An easy way out of this dilemma is to remember that a quantizer can be completely defined by its output points. In order for this quantizer to possess structure, these points should be spaced in some regular manner.

Regular arrangements of output points in space are called *lattices*. Mathematically, we can define a lattice as follows:

Let $\{\mathbf{a_1}, \mathbf{a_2}, \ldots, \mathbf{a_L}\}$ be L independent L-dimensional vectors. Then the set

$$\mathcal{L} = \left\{ \mathbf{x} : \mathbf{x} = \sum_{i=1}^{L} u_i \mathbf{a}_i \right\} \tag{10.10}$$

is a lattice if $\{u_i\}$ are all integers.

When a subset of lattice points is used as the output points of a vector quantizer, the quantizer is known as a *lattice vector quantizer*. From this definition, the pyramid vector quantizer described earlier can be viewed as a lattice vector quantizer. Basing a quantizer on a lattice solves the storage problem. As any lattice point can be regenerated if we know the basis set, there is no need to store the output points. Further, the highly structured nature of lattices makes finding the closest output point to an input relatively simple. Note that what we give up when we use lattice vector quantizers is the clustering property of LBG quantizers.

Let's take a look at a few examples of lattices in two dimensions. If we pick $a_1 = (1, 0)$ and $a_2 = (0, 1)$, we obtain the integer lattice—the lattice that contains all points in two dimensions whose coordinates are integers.

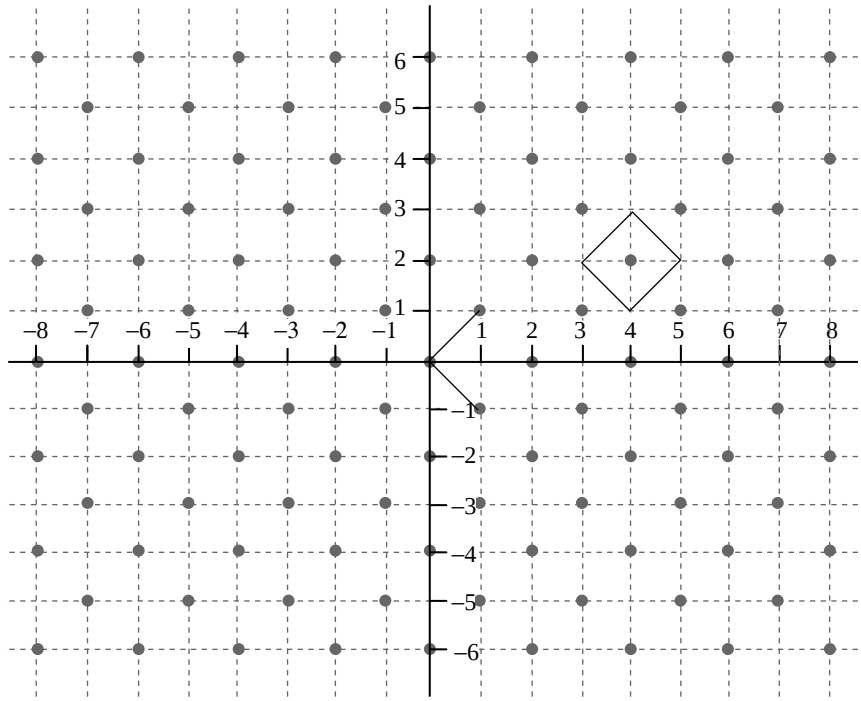

FIGURE 10. 24 The D_2 lattice.

If we pick $a_1 = (1, 1)$ and $a_2 = (1, -1)$, we get the lattice shown in Figure 10.24. This lattice has a rather interesting property. Any point in the lattice is given by $na_1 + ma_2$, where n and m are integers. But

$$na_1 + ma_2 = \begin{bmatrix} n+m \\ n-m \end{bmatrix}$$

and the sum of the coefficients is $n+m+n-m = 2n$, which is even for all n. Therefore, all points in this lattice have an even coordinate sum. Lattices with these properties are called *D lattices*.

Finally, if $a_1 = (1, 0)$ and $a_2 = \left(-\frac{1}{2}, \frac{\sqrt{3}}{2}\right)$, we get the hexagonal lattice shown in Figure 10.25. This is an example of an *A lattice*.

There are a large number of lattices that can be used to obtain lattice vector quantizers. In fact, given a dimension L, there are an infinite number of possible sets of L independent vectors. Among these, we would like to pick the lattice that produces the greatest reduction in granular noise. When comparing the square and circle as candidates for quantization regions, we used the integral over the shape of $\|X\|^2$. This is simply the second moment of the shape. The shape with the smallest second moment for a given volume is known to be the circle in two dimensions and the sphere and hypersphere in higher dimensions [138]. Unfortunately, circles and spheres cannot tile space; either there will be overlap or there will

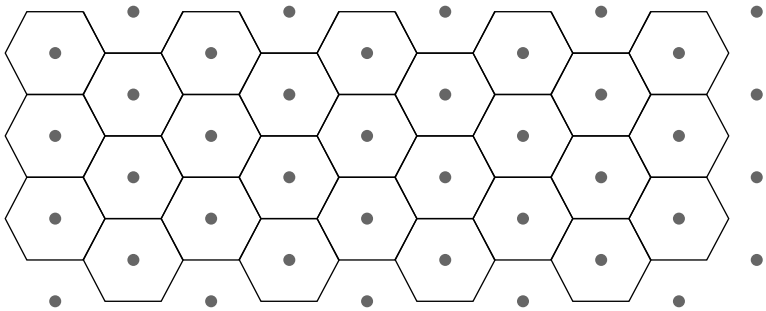

FIGURE 10.25 The A_2 lattice.

be holes. As the ideal case is unattainable, we can try to approximate it. We can look for ways of arranging spheres so that they cover space with minimal overlap [139], or look for ways of packing spheres with the least amount of space left over [138]. The centers of these spheres can then be used as the output points. The quantization regions will not be spheres, but they may be close approximations to spheres.

The problems of sphere covering and sphere packing are widely studied in a number of different areas. Lattices discovered in these studies have also been useful as vector quantizers [138]. Some of these lattices, such as the A_2 and D_2 lattices described earlier, are based on the root systems of Lie algebras [140]. The study of Lie algebras is beyond the scope of this book; however, we have included a brief discussion of the root systems and how to obtain the corresponding lattices in Appendix C.

One of the nice things about root lattices is that we can use their structural properties to obtain fast quantization algorithms. For example, consider building a quantizer based on the D_2 lattice. Because of the way in which we described the D_2 lattice, the size of the lattice is fixed. We can change the size by picking the basis vectors as (Δ, Δ) and $(\Delta, -\Delta)$, instead of $(1, 1)$ and $(1, -1)$. We can have exactly the same effect by dividing each input by Δ before quantization, and then multiplying the reconstruction values by Δ. Suppose we pick the latter approach and divide the components of the input vector by Δ. If we wanted to find the closest lattice point to the input, all we need to do is find the closest integer to each coordinate of the scaled input. If the sum of these integers is even, we have a lattice point. If not, find the coordinate that incurred the largest distortion during conversion to an integer and then find the next closest integer. The sum of coordinates of this new vector differs from the sum of coordinates of the previous vector by one. Therefore, if the sum of coordinates of the previous vector was odd, the sum of the coordinates of the current vector will be even, and we have the closest lattice point to the input.

Example 10.6.2:

Suppose the input vector is given by (2.3, 1.9). Rounding each coefficient to the nearest integer, we get the vector (2, 2). The sum of the coordinates is even; therefore, this is the closest lattice point to the input.

Suppose the input was (3.4, 1.8). Rounding the components to the nearest integer, we get (3, 2). The sum of the components is 5, which is odd. The differences between the components of the input vector and the nearest integer are 0.4 and 0.2. The largest difference was incurred by the first component, so we round it up to the next closest integer, and the resulting vector is (4, 2). The sum of the coordinates is 6, which is even; therefore, this is the closest lattice point. ♦

Many of the lattices have similar properties that can be used to develop fast algorithms for finding the closest output point to a given input [141, 140].

To review our coverage of lattice vector quantization, overload error can be reduced by careful selection of the boundary, and we can reduce the granular noise by selection of the lattice. The lattice also provides us with a way to avoid storage problems. Finally, we can use the structural properties of the lattice to find the closest lattice point to a given input.

Now we need two things: to know how to find the closest *output* point (remember, not all lattice points are output points), and to find a way of assigning a binary codeword to the output point and recovering the output point from the binary codeword. This can be done by again making use of the specific structures of the lattices. While the procedures necessary are simple, explanations of the procedures are lengthy and involved (see [142] and [140] for details).

10.7 Variations on the Theme

Because of its capability to provide high compression with relatively low distortion, vector quantization has been one of the more popular lossy compression techniques over the last decade in such diverse areas as video compression and low-rate speech compression. During this period, several people have come up with variations on the basic vector quantization approach. We briefly look at a few of the more well-known variations here, but this is by no means an exhaustive list. For more information, see [5] and [143].

10.7.1 Gain-Shape Vector Quantization

In some applications such as speech, the dynamic range of the input is quite large. One effect of this is that, in order to be able to represent the various vectors from the source, we need a very large codebook. This requirement can be reduced by normalizing the source output vectors, then quantizing the normalized vector and the normalization factor separately [144, 133]. In this way, the variation due to the dynamic range is represented by the normalization factor or *gain*, while the vector quantizer is free to do what it does best, which is to capture the structure in the source output. Vector quantizers that function in this manner are called *gain-shape vector quantizers*. The pyramid quantizer discussed earlier is an example of a gain-shape vector quantizer.

10.7.2 Mean-Removed Vector Quantization

If we were to generate a codebook from an image, differing amounts of background illumination would result in vastly different codebooks. This effect can be significantly reduced if we remove the mean from each vector before quantization. The mean and the mean-removed vector can then be quantized separately. The mean can be quantized using a scalar quantization scheme, while the mean-removed vector can be quantized using a vector quantizer. Of course, if this strategy is used, the vector quantizer should be designed using mean-removed vectors as well.

Example 10.7.1:

Let us encode the Sinan image using a codebook generated by the Sena image, as we did in Figure 10.16. However, this time we will use a mean-removed vector quantizer. The result is shown in Figure 10.26. For comparison we have also included the reconstructed image from Figure 10.16. Notice the annoying blotches on the shoulder have disappeared. However, the reconstructed image also suffers from more blockiness. The blockiness increases because adding the mean back into each block accentuates the discontinuity at the block boundaries.

FIGURE 10. 26 **Left: Reconstructed image using mean-removed vector quantization and the Sena image as the training set. Right: LBG vector quantization with the Sena image as the training set.**

Each approach has its advantages and disadvantages. Which approach we use in a particular application depends very much on the application. ♦

10.7.3 Classified Vector Quantization

We can sometimes divide the source output into separate classes with different spatial properties. In these cases, it can be very beneficial to design separate vector quantizers for the different classes. This approach, referred to as *classified vector quantization*, is especially useful in image compression, where edges and nonedge regions form two distinct classes. We can separate the training set into vectors that contain edges and vectors that do not. A separate vector quantizer can be developed for each class. During the encoding process, the vector is first tested to see if it contains an edge. A simple way to do this is to check the variance of the pixels in the vector. A large variance will indicate the presence of an edge. More sophisticated techniques for edge detection can also be used. Once the vector is classified, the corresponding codebook can be used to quantize the vector. The encoder transmits both the label for the codebook used and the label for the vector in the codebook [145].

A slight variation of this strategy is to use different kinds of quantizers for the different classes of vectors. For example, if certain classes of source outputs require quantization at a higher rate than is possible using LBG vector quantizers, we can use lattice vector quantizers. An example of this approach can be found in [146].

10.7.4 Multistage Vector Quantization

Multistage vector quantization [147] is an approach that reduces both the encoding complexity and the memory requirements for vector quantization, especially at high rates. In this approach, the input is quantized in several stages. In the first stage, a low-rate vector quantizer is used to generate a coarse approximation of the input. This coarse approximation, in the form of the label of the output point of the vector quantizer, is transmitted to the receiver. The error between the original input and the coarse representation is quantized by the second-stage quantizer, and the label of the output point is transmitted to the receiver. In this manner, the input to the nth-stage vector quantizer is the difference between the original input and the reconstruction obtained from the outputs of the preceding $n-1$ stages. The difference between the input to a quantizer and the reconstruction value is often called the *residual*, and the multistage vector quantizers are also known as *residual vector quantizers* [148]. The reconstructed vector is the sum of the output points of each of the stages. Suppose we have a three-stage vector quantizer, with the three quantizers represented by $\mathbf{Q_1}$, $\mathbf{Q_2}$, and $\mathbf{Q_3}$. Then for a given input $\mathbf{X}$, we find

$$\begin{aligned}\mathbf{Y_1} &= \mathbf{Q_1}(\mathbf{X})\\ \mathbf{Y_2} &= \mathbf{Q_2}(\mathbf{X}-\mathbf{Q_1}(\mathbf{X}))\\ \mathbf{Y_3} &= \mathbf{Q_3}(\mathbf{X}-\mathbf{Q_1}(\mathbf{X})-\mathbf{Q_2}(\mathbf{X}-\mathbf{Q_1}(\mathbf{X}))).\end{aligned} \tag{10.11}$$

The reconstruction $\hat{\mathbf{X}}$ is given by

$$\hat{\mathbf{X}} = \mathbf{Y_1}+\mathbf{Y_2}+\mathbf{Y_3}. \tag{10.12}$$

This process is shown in Figure 10.27.

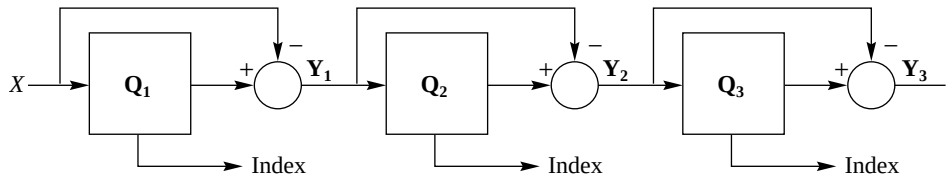

FIGURE 10. 27 A three-stage vector quantizer.

If we have K stages, and the codebook size of the nth-stage vector quantizer is L_n, then the effective size of the overall codebook is $L_1 \times L_2 \times \cdots \times L_K$. However, we need to store only $L_1 + L_2 + \cdots + L_K$ vectors, which is also the number of comparisons required. Suppose we have a five-stage vector quantizer, each with a codebook size of 32, meaning that we would have to store 160 codewords. This would provide an effective codebook size of $32^5 = 33,554,432$. The computational savings are also of the same order.

This approach allows us to use vector quantization at much higher rates than we could otherwise. However, at rates at which it is feasible to use LBG vector quantizers, the performance of the multistage vector quantizers is generally lower than the LBG vector quantizers [5]. The reason for this is that after the first few stages, much of the structure used by the vector quantizer has been removed, and the vector quantization advantage that depends on this structure is not available. Details on the design of residual vector quantizers can be found in [148, 149].

There may be some vector inputs that can be well represented by fewer stages than others. A multistage vector quantizer with a variable number of stages can be implemented by extending the idea of recursively indexed scalar quantization to vectors. It is not possible to do this directly because there are some fundamental differences between scalar and vector quantizers. The input to a scalar quantizer is assumed to be *iid*. On the other hand, the vector quantizer can be viewed as a pattern-matching algorithm [150]. The input is assumed to be one of a number of different patterns. The scalar quantizer is used after the redundancy has been removed from the source sequence, while the vector quantizer takes advantage of the redundancy in the data.

With these differences in mind, the recursively indexed vector quantizer (RIVQ) can be described as a two-stage process. The first stage performs the normal pattern-matching function, while the second stage recursively quantizes the residual if the magnitude of the residual is greater than some prespecified threshold. The codebook of the second stage is ordered so that the magnitude of the codebook entries is a nondecreasing function of its index. We then choose an index I that will determine the mode in which the RIVQ operates.

The quantization rule Q, for a given input value $\mathbf{X}$, is as follows:

- Quantize $\mathbf{X}$ with the first-stage quantizer Q_1.
- If the residual $\|\mathbf{X} - \mathbf{Q_1}(\mathbf{X})\|$ is below a specified threshold, then $\mathbf{Q_1}(\mathbf{X})$ is the nearest output level.

- Otherwise, generate $\mathbf{X_1} = \mathbf{X} - \mathbf{Q_1}(\mathbf{X})$ and quantize using the second-stage quantizer $\mathbf{Q_2}$. Check if the index J_1 of the output is below the index I. If so,

$$\mathbf{Q}(\mathbf{X}) = \mathbf{Q_1}(\mathbf{X}) + \mathbf{Q_2}(\mathbf{X_1}).$$

  If not, form

$$\mathbf{X_2} = \mathbf{X_1} - \mathbf{Q}(\mathbf{X_1})$$

  and do the same for $\mathbf{X_2}$ as we did for $\mathbf{X_1}$.

This process is repeated until for some m, the index J_m falls below the index I, in which case $\mathbf{X}$ will be quantized to

$$\mathbf{Q}(\mathbf{X}) = \mathbf{Q_1}(\mathbf{X}) + \mathbf{Q_2}(\mathbf{X_1}) + \cdots + \mathbf{Q_2}(\mathbf{X_M}).$$

Thus, the RIVQ operates in two modes: when the index J of the quantized input falls below a given index I and when the index J falls above the index I.

Details on the design and performance of the recursively indexed vector quantizer can be found in [151, 152].

10.7.5 Adaptive Vector Quantization

While LBG vector quantizers function by using the structure in the source output, this reliance on the use of the structure can also be a drawback when the characteristics of the source change over time. For situations like these, we would like to have the quantizer adapt to the changes in the source output.

For mean-removed and gain-shape vector quantizers, we can adapt the scalar aspect of the quantizer, that is, the quantization of the mean or the gain using the techniques discussed in the previous chapter. In this section, we look at a few approaches to adapting the codebook of the vector quantizer to changes in the characteristics of the input.

One way of adapting the codebook to changing input characteristics is to start with a very large codebook designed to accommodate a wide range of source characteristics [153]. This large codebook can be ordered in some manner known to both transmitter and receiver. Given a sequence of input vectors to be quantized, the encoder can select a subset of the larger codebook to be used. Information about which vectors from the large codebook were used can be transmitted as a binary string. For example, if the large codebook contained 10 vectors, and the encoder was to use the second, third, fifth, and ninth vectors, we would send the binary string 0110100010, with a 1 representing the position of the codeword used in the large codebook. This approach permits the use of a small codebook that is matched to the local behavior of the source.

This approach can be used with particular effectiveness with the recursively indexed vector quantizer [151]. Recall that in the recursively indexed vector quantizer, the quantized output is always within a prescribed distance of the inputs, determined by the index I. This means that the set of output values of the RIVQ can be viewed as an accurate representation of the inputs and their statistics. Therefore, we can treat a subset of the output set of the previous intervals as our large codebook. We can then use the method described in [153] to

inform the receiver of which elements of the previous outputs form the codebook for the next interval. This method (while not the most efficient) is quite simple. Suppose an output set, in order of first appearance, is $\{p, a, q, s, l, t, r\}$, and the desired codebook for the interval to be encoded is $\{a, q, l, r\}$. Then we would transmit the binary string 0110101 to the receiver. The 1s correspond to the letters in the output set, which would be elements of the desired codebook. We select the subset for the current interval by finding the closest vectors from our collection of past outputs to the input vectors of the current set. This means that there is an inherent delay of one interval imposed by this approach. The overhead required to send the codebook selection is M/N, where M is the number of vectors in the output set and N is the interval size.

Another approach to updating the codebook is to check the distortion incurred while quantizing each input vector. Whenever this distortion is above some specified threshold, a different higher-rate mechanism is used to encode the input. The higher-rate mechanism might be the scalar quantization of each component, or the use of a high-rate lattice vector quantizer. This quantized representation of the input is transmitted to the receiver and, at the same time, added to both the encoder and decoder codebooks. In order to keep the size of the codebook the same, an entry must be discarded when a new vector is added to the codebook. Selecting an entry to discard is handled in a number of different ways. Variations of this approach have been used for speech coding, image coding, and video coding (see [154, 155, 156, 157, 158] for more details).

10.8 Trellis-Coded Quantization

Finally, we look at a quantization scheme that appears to be somewhat different from other vector quantization schemes. In fact, some may argue that it is not a vector quantizer at all. However, the trellis-coded quantization (TCQ) algorithm gets its performance advantage by exploiting the statistical structure exploited by the lattice vector quantizer. Therefore, we can argue that it should be classified as a vector quantizer.

The trellis-coded quantization algorithm was inspired by the appearance of a revolutionary concept in modulation called trellis-coded modulation (TCM). The TCQ algorithm and its entropy-constrained variants provide some of the best performance when encoding random sources. This quantizer can be viewed as a vector quantizer with very large dimension, but a restricted set of values for the components of the vectors.

Like a vector quantizer, the TCQ quantizes sequences of source outputs. Each element of a sequence is quantized using 2^R reconstruction levels selected from a set of 2^{R+1} reconstruction levels, where R is the number of bits per sample used by a trellis-coded quantizer. The 2^R element subsets are predefined; which particular subset is used is based on the reconstruction level used to quantize the previous quantizer input. However, the TCQ algorithm allows us to postpone a decision on which reconstruction level to use until we can look at a sequence of decisions. This way we can select the sequence of decisions that gives us the lowest amount of average distortion.

Let's take the case of a 2-bit quantizer. As described above, this means that we will need 2^3, or 8, reconstruction levels. Let's label these reconstruction levels as shown in Figure 10.28. The set of reconstruction levels is partitioned into two subsets: one consisting

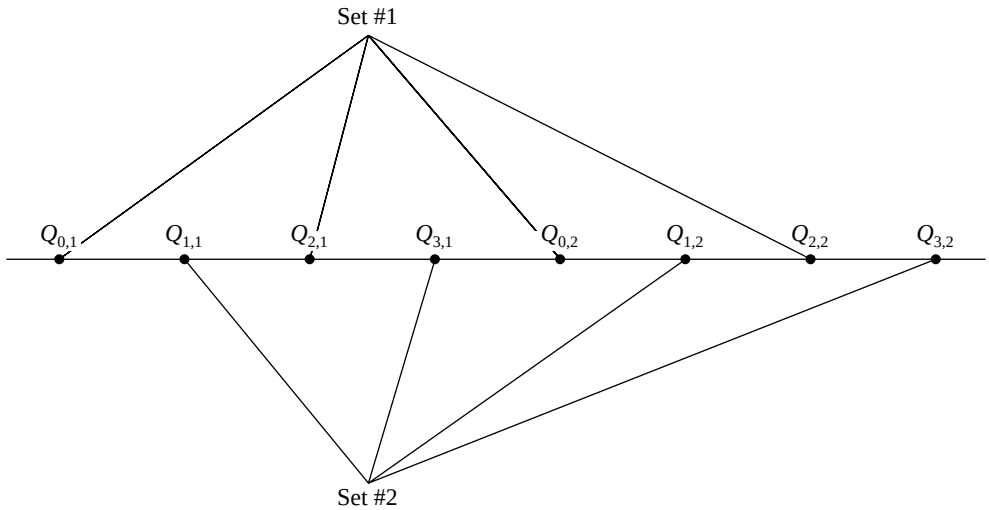

FIGURE 10. 28 Reconstruction levels for a 2-bit trellis-coded quantizer.

of the reconstruction values labeled $Q_{0,i}$ and $Q_{2,i}$, and the remainder comprising the second set. We use the first set to perform the quantization if the previous quantization level was one labeled $Q_{0,i}$ or $Q_{1,i}$; otherwise, we use the second set. Because the current reconstructed value defines the subset that can be used to perform the quantization on the next input, sometimes it may be advantageous to actually accept more distortion than necessary for the current sample in order to have less distortion in the next quantization step. In fact, at times it may be advantageous to accept poor quantization for several samples so that several samples down the line the quantization can result in less distortion. If you have followed this reasoning, you can see how we might be able to get lower overall distortion by looking at the quantization of an entire sequence of source outputs. The problem with delaying a decision is that the number of choices increases exponentially with each sample. In the 2-bit example, for the first sample we have four choices; for each of these four choices we have four choices for the second sample. For each of these 16 choices we have four choices for the third sample, and so on. Luckily, there is a technique that can be used to keep this explosive growth of choices under control. The technique, called the *Viterbi algorithm* [159], is widely used in error control coding.

In order to explain how the Viterbi algorithm works, we will need to formalize some of what we have been discussing. The sequence of choices can be viewed in terms of a state diagram. Let's suppose we have four states: S_0, S_1, S_2, and S_3. We will say we are in state S_k if we use the reconstruction levels $Q_{k,1}$ or $Q_{k,2}$. Thus, if we use the reconstruction levels $Q_{0,i}$, we are in state S_0. We have said that we use the elements of Set #1 if the previous quantization levels were $Q_{0,i}$ or $Q_{1,i}$. As Set #1 consists of the quantization levels $Q_{0,i}$ and $Q_{2,i}$, this means that we can go from state S_0 and S_1 to states S_0 and S_2. Similarly, from states S_2 and S_3 we can only go to states S_1 and S_3. The state diagram can be drawn as shown in Figure 10.29.

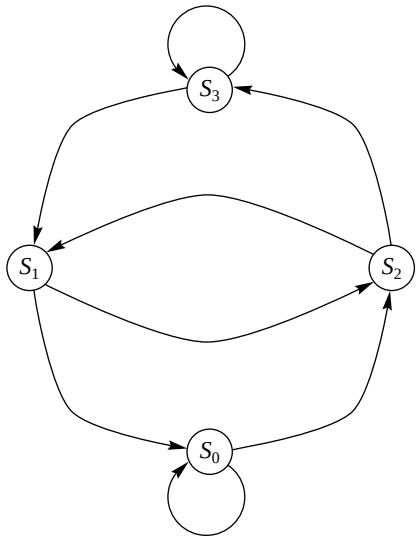

FIGURE 10. 29 State diagram for the selection process.

Let's suppose we go through two sequences of choices that converge to the same state, after which both sequences are identical. This means that the sequence of choices that had incurred a higher distortion at the time the two sequences converged will have a higher distortion from then on. In the end we will select the sequence of choices that results in the lowest distortion; therefore, there is no point in continuing to keep track of a sequence that we will discard anyway. This means that whenever two sequences of choices converge, we can discard one of them. How often does this happen? In order to see this, let's introduce time into our state diagram. The state diagram with the element of time introduced into it is called a *trellis diagram*. The trellis for this particular example is shown in Figure 10.30. At each time instant, we can go from one state to two other states. And, at each step we

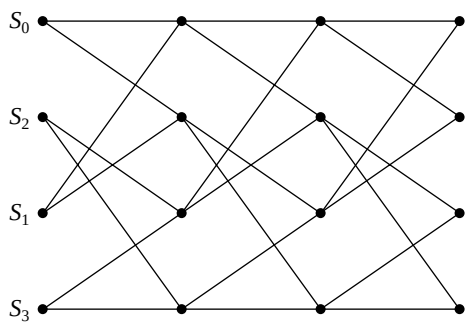

FIGURE 10. 30 Trellis diagram for the selection process.

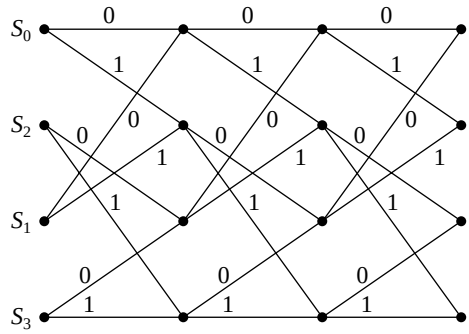

FIGURE 10. 31 **Trellis diagram for the selection process with binary labels for the state transitions.**

have two sequences that converge to each state. If we discard one of the two sequences that converge to each state, we can see that, no matter how long a sequence of decisions we use, we will always end up with four sequences.

Notice that, assuming the initial state is known to the decoder, any path through this particular trellis can be described to the decoder using 1 bit per sample. From each state we can only go to two other states. In Figure 10.31, we have marked the branches with the bits used to signal that transition. Given that each state corresponds to two quantization levels, specifying the quantization level for each sample would require an additional bit, resulting in a total of 2 bits per sample. Let's see how all this works together in an example.

Example 10.8.1:

Using the quantizer whose quantization levels are shown in Figure 10.32, we will quantize the sequence of values 0.2, 1.6, 2.3. For the distortion measure we will use the sum of absolute differences. If we simply used the quantization levels marked as Set #1 in Figure 10.28, we would quantize 0.2 to the reconstruction value 0.5, for a distortion of 0.3. The second sample value of 1.6 would be quantized to 2.5, and the third sample value of 2.3 would also be quantized to 2.5, resulting in a total distortion of 1.4. If we used Set #2 to quantize these values, we would end up with a total distortion of 1.6. Let's see how much distortion results when using the TCQ algorithm.

We start by quantizing the first sample using the two quantization levels $Q_{0,1}$ and $Q_{0,2}$. The reconstruction level $Q_{0,2}$, or 0.5, is closer and results in an absolute difference of 0.3. We mark this on the first node corresponding to S_0. We then quantize the first sample using

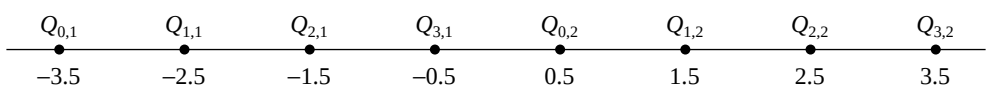

FIGURE 10. 32 **Reconstruction levels for a 2-bit trellis-coded quantizer.**

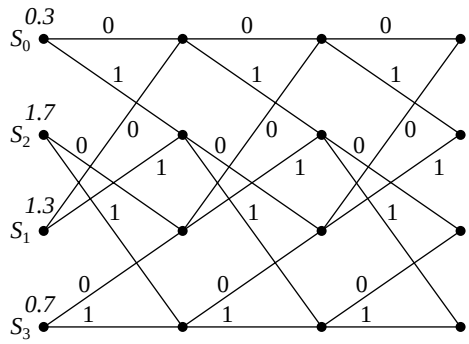

FIGURE 10. 33 Quantizing the first sample.

$Q_{1,1}$ and $Q_{1,2}$. The closest reconstruction value is $Q_{1,2}$, or 1.5, which results in a distortion value of 1.3. We mark the first node corresponding to S_1. Continuing in this manner, we get a distortion value of 1.7 when we use the reconstruction levels corresponding to state S_2 and a distortion value of 0.7 when we use the reconstruction levels corresponding to state S_3. At this point the trellis looks like Figure 10.33. Now we move on to the second sample. Let's first quantize the second sample value of 1.6 using the quantization levels associated with state S_0. The reconstruction levels associated with state S_0 are -3.5 and 0.5. The closest value to 1.6 is 0.5. This results in an absolute difference for the second sample of 1.1. We can reach S_0 from S_0 and from S_1. If we accept the first sample reconstruction corresponding to S_0, we will end up with an accumulated distortion of 1.4. If we accept the reconstruction corresponding to state S_1, we get an accumulated distortion of 2.4. Since the accumulated distortion is less if we accept the transition from state S_0, we do so and discard the transition from state S_1. Continuing in this fashion for the remaining states, we end up with the situation depicted in Figure 10.34. The sequence of decisions that have been terminated are shown by an X on the branch corresponding to the particular transition. The accumulated distortion is listed at each node. Repeating this procedure for the third sample value of 2.3, we obtain the

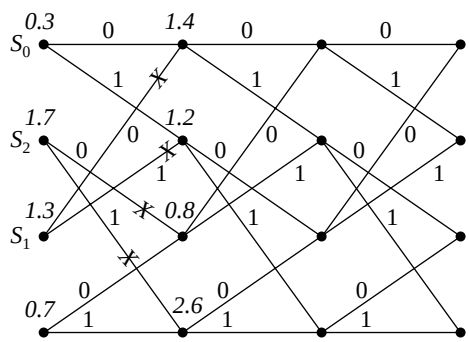

FIGURE 10. 34 Quantizing the second sample.

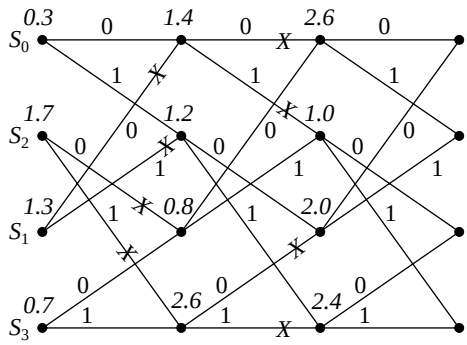

FIGURE 10. 35 Quantizing the third sample.

trellis shown in Figure 10.35. If we wanted to terminate the algorithm at this time, we could pick the sequence of decisions with the smallest accumulated distortion. In this particular example, the sequence would be S_3, S_1, S_2. The accumulated distortion is 1.0, which is less than what we would have obtained using either Set #1 or Set #2. ♦

10.9 Summary

In this chapter we introduced the technique of vector quantization. We have seen how we can make use of the structure exhibited by groups, or vectors, of values to obtain compression. Because there are different kinds of structure in different kinds of data, there are a number of different ways to design vector quantizers. Because data from many sources, when viewed as vectors, tend to form clusters, we can design quantizers that essentially consist of representations of these clusters. We also described aspects of the design of vector quantizers and looked at some applications. Recent literature in this area is substantial, and we have barely skimmed the surface of the large number of interesting variations of this technique.

Further Reading

The subject of vector quantization is dealt with extensively in the book *Vector Quantization and Signal Compression*, by A. Gersho and R.M. Gray [5]. There is also an excellent collection of papers called *Vector Quantization*, edited by H. Abut and published by IEEE Press [143].

There are a number of excellent tutorial articles on this subject:

1. "Vector Quantization," by R.M. Gray, in the April 1984 issue of *IEEE Acoustics, Speech, and Signal Processing Magazine* [160].
2. "Vector Quantization: A Pattern Matching Technique for Speech Coding," by A. Gersho and V. Cuperman, in the December 1983 issue of *IEEE Communications Magazine* [150].

3. "Vector Quantization in Speech Coding," by J. Makhoul, S. Roucos, and H. Gish, in the November 1985 issue of the *Proceedings of the IEEE* [161].

4. "Vector Quantization," by P.F. Swaszek, in *Communications and Networks*, edited by I.F. Blake and H.V. Poor [162].

5. A survey of various image-coding applications of vector quantization can be found in "Image Coding Using Vector Quantization: A Review," by N.M. Nasrabadi and R.A. King, in the August 1988 issue of the *IEEE Transactions on Communications* [163].

6. A thorough review of lattice vector quantization can be found in "Lattice Quantization," by J.D. Gibson and K. Sayood, in *Advances in Electronics and Electron Physics* [140].

The area of vector quantization is an active one, and new techniques that use vector quantization are continually being developed. The journals that report work in this area include *IEEE Transactions on Information Theory*, *IEEE Transactions on Communications*, *IEEE Transactions on Signal Processing*, and *IEEE Transactions on Image Processing*, among others.

10.10 Projects and Problems

1. In Example 10.3.2 we increased the SNR by about 0.3 dB by moving the top-left output point to the origin. What would happen if we moved the output points at the four corners to the positions $(\pm\Delta, 0)$, $(0, \pm\Delta)$. As in the example, assume the input has a Laplacian distribution with mean zero and variance one, and $\Delta = 0.7309$. You can obtain the answer analytically or through simulation.

2. For the quantizer of the previous problem, rather than moving the output points to $(\pm\Delta, 0)$ and $(0, \pm\Delta)$, we could have moved them to other positions that might have provided a larger increase in SNR. Write a program to test different (reasonable) possibilities and report on the best and worst cases.

3. In the program `trainvq.c` the empty cell problem is resolved by replacing the vector with no associated training set vectors with a training set vector from the quantization region with the largest number of vectors. In this problem, we will investigate some possible alternatives.

   Generate a sequence of pseudorandom numbers with a triangular distribution between 0 and 2. (You can obtain a random number with a triangular distribution by adding two uniformly distributed random numbers.) Design an eight-level, two-dimensional vector quantizer with the initial codebook shown in Table 10.9.

   (a) Use the `trainvq` program to generate a codebook with 10,000 random numbers as the training set. Comment on the final codebook you obtain. Plot the elements of the codebook and discuss why they ended up where they did.

   (b) Modify the program so that the empty cell vector is replaced with a vector from the quantization region with the largest distortion. Comment on any changes in

TABLE 10.9 Initial codebook for Problem 3.

1	1
1	2
1	0.5
0.5	1
0.5	0.5
1.5	1
2	5
3	3

the distortion (or lack of change). Is the final codebook different from the one you obtained earlier?

(c) Modify the program so that whenever an empty cell problem arises, a two-level quantizer is designed for the quantization region with the largest number of output points. Comment on any differences in the codebook and distortion from the previous two cases.

4. Generate a 16-dimensional codebook of size 64 for the Sena image. Construct the vector as a 4×4 block of pixels, an 8×2 block of pixels, and a 16×1 block of pixels. Comment on the differences in the mean squared errors and the quality of the reconstructed images. You can use the program `trvqsp_img` to obtain the codebooks.

5. In Example 10.6.1 we designed a 60-level two-dimensional quantizer by taking the two-dimensional representation of an 8-level scalar quantizer, removing 12 output points from the 64 output points, and adding 8 points in other locations. Assume the input is Laplacian with zero mean and unit variance, and $\Delta = 0.7309$.

(a) Calculate the increase in the probability of overload by the removal of the 12 points from the original 64.

(b) Calculate the decrease in overload probability when we added the 8 new points to the remaining 52 points.

6. In this problem we will compare the performance of a 16-dimensional pyramid vector quantizer and a 16-dimensional LBG vector quantizer for two different sources. In each case the codebook for the pyramid vector quantizer consists of 272 elements:

- 32 vectors with 1 element equal to $\pm\Delta$, and the other 15 equal to zero, and
- 240 vectors with 2 elements equal to $\pm\Delta$ and the other 14 equal to zero.

The value of Δ should be adjusted to give the best performance. The codebook for the LBG vector quantizer will be obtained by using the program `trvqsp_img` on the source output. You will have to modify `trvqsp_img` slightly to give you a codebook that is not a power of two.

(a) Use the two quantizers to quantize a sequence of 10,000 zero mean unit variance Laplacian random numbers. Using either the mean squared error or the SNR as a measure of performance, compare the performance of the two quantizers.

(b) Use the two quantizers to quantize the Sinan image. Compare the two quantizers using either the mean squared error or the SNR and the reconstructed image. Compare the difference between the performance of the two quantizers with the difference when the input was random.