The Design and Analysis of Efficient Lossless Data Compression Systems

Paul Glor Howard

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-93-28
June 1993

The Design and Analysis of Efficient Lossless Data Compression Systems

Paul Glor Howard

Brown University
Department of Computer Science
Technical Report No. CS–93–28
June 1993

The Design and Analysis of Efficient Lossless Data Compression Systems[1]

Paul Glor Howard[2]
Department of Computer Science
Brown University
Providence, R.I. 02912–1910

Abstract

Our thesis is that high compression efficiency for text and images can be obtained by using sophisticated statistical compression techniques, and that greatly increased speed can be achieved at only a small cost in compression efficiency. Our emphasis is on elegant design and mathematical as well as empirical analysis.

We analyze arithmetic coding as it is commonly implemented and show rigorously that almost no compression is lost in the implementation. We show that high-efficiency lossless compression of both text and grayscale images can be obtained by using appropriate models in conjunction with arithmetic coding. We introduce a four-component paradigm for lossless image compression and present two methods that give state of the art compression efficiency. In the text compression area, we give a small improvement on the preferred method in the literature.

We show that we can often obtain significantly improved throughput at the cost of slightly reduced compression. The extra speed comes from simplified coding and modeling. Coding is simplified by using prefix codes when arithmetic coding is not necessary, and by using a new practical version of arithmetic coding, called *quasi-arithmetic coding*, when the precision of arithmetic coding is needed. We simplify image modeling by using small prediction contexts and making plausible assumptions about the distributions of pixel intensity values. For text modeling we use self-organizing-list heuristics and low-precision statistics.

Index terms: Data compression, analysis of algorithms, arithmetic coding, lossless image compression, text compression, adaptive modeling.

---

[1] A similar version of this report was submitted in fulfillment of the dissertation requirement for Paul Howard's Ph.D.

[2]Support was provided by NASA Graduate Student Researchers Program grant NGT–50420, by an NSF Presidential Young Investigators Award with matching funds from IBM, by a Universities Space Research Association/CESDIS appointment, and by National Science Foundation grant IRI–9116451.

THE DESIGN AND ANALYSIS OF EFFICIENT LOSSLESS DATA COMPRESSION SYSTEMS

by
Paul Glor Howard
B.S., Massachusetts Institute of Technology, 1977
Sc.M., Brown University, 1989

Thesis
Submitted in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy in the
Department of Computer Science at Brown University.

May 1993

© Copyright 1993
by
Paul Glor Howard

VITA

I was born in Buffalo, New York, on February 21, 1950. I received an excellent preparatory school education at the Nichols School of Buffalo, and an excellent undergraduate education at M.I.T., where, after a few delays, I graduated with a B.S. degree in 1977.

I was employed by Marine Midland Bank in Buffalo for several summers starting in 1968, and full time starting in 1974. Perhaps surprisingly, it was at Marine Midland that I first heard of data compression, in particular the remarkable notion of Huffman coding. In 1978 I moved to the Washington, D.C., area, became a Christian, and worked briefly for ADP Network Services. Then I worked for three years at Tracor, Inc. For a year starting in June 1982 I served as a short-term assistant with Wycliffe Bible Translators and the Summer Institute of Linguistics, based in Dallas, Texas. At the very beginning of that year I met Alice Mae Batcher, and in July 1983 we were married, much to my happiness.

From 1984 to 1987 I did compiler work at Microway, Inc., in Plymouth, Massachusetts. Our daughter Ruth Ellen was born in 1985. In 1987 I came to Brown – a good choice for me since the Computer Science Department at Brown is small enough that I didn't get lost. Working with Jeff Vitter, I received the Sc.M. degree in 1989; later in that year our son Peter was stillborn.

Now in 1993, after seven data compression papers (titles and publication information can be found in the References), I have finally finished my Ph.D. Just seven days after I handed my dissertation in to the Graduate School, our second daughter, Melinda Rose, was born!

PREFACE

This thesis deals with statistical data compression. Since good statistical data compression methods must attend both to the coding process itself (turning events into bits and back again) and to data modeling (deciding which events are possible and how likely each one is in a given situation), I address both the coding and the modeling components of statistical compression, considering both text compression and lossless image compression. My approach is to show by rigorous proof and by experiment that particular coding and modeling methods give optimal (or at least good) compression, and then to show in a similar fashion that sacrificing a small amount of compression allows much faster processing. Chapter 2 addresses coding issues using this approach, and Chapters 3 and 4 apply the same approach to image and text modeling respectively.

Acknowledgments

Writing a Ph.D. dissertation is a long process, and having a good relationship with one's advisor is essential. I have been extremely fortunate to have had the chance to work with Jeff Vitter. Aside from the excellent technical assistance that he readily provided, he gave me the pep talks I needed early in my career at Brown, the impetus to be precise in my thinking and writing, and the encouragement to continue doing research through the years. Jeff is a demanding advisor, but because of this I have grown both in research ability and in self-confidence. Thanks very much, Jeff, and all the best to you in your new position at Duke.

I am grateful to Jim Storer and Marty Cohn from Brandeis and John Reif from Duke for organizing the three Data Compression Conferences at Snowbird, Utah. Those conferences provided an ideal forum for my work, a forum that otherwise would not have existed. Jim also served on my committee, as did Franco Preparata from Brown.

Various people at Brown have given me encouragement and assistance – one can't go it alone! I thank Gail Mitchell, Misty and Mark Nodine, Scott Meyers, Jyh-Han Lin, Manojit Sarkar, Dave Durfee, P. Krishnan, Dzung Hoang, and Darren Vengroff among the students, and Mary Andrade and Dorinda Moulton from the staff. I also thank the people from the First Baptist Church of Plymouth, Massachusetts, for their interest and their prayers; I especially thank the ministers and interim ministers, Mark

Clinger, Meg Hess, Ray Rosa (deceased), and Priscilla Eppinger-Mendes. Gene Chase of Messiah College deserves mention for encouraging me to consider graduate school in the first place.

It's also necessary to pay the bills. Most of my financial support came from a NASA Graduate Student Researchers Program fellowship, which I am pleased to acknowledge here. Jim Tilton and Milt Halem at the Goddard Space Flight Center deserve special recognition.

By far my most important support came from my family. My father Lew gave me a great respect for education, prodding me to finish my undergraduate degree and paying for ten years of private school and most of my undergraduate study; unfortunately he died shortly before I started graduate work. My mother Florence has welcomed my studying (and even taking the programming section of the comprehensive exams!) in her home. My daughter Ruth Ellen can't remember her dad being anything but a graduate student; she could be a good researcher some day, if that's where God leads her. And Alice. We've been through some difficult times, but you've always stuck with me. I love you now, and I always will.

Providence
April 1993

For Alice and Ruth Ellen

CONTENTS

Vita iii

Preface iv

1 Introduction to Data Compression 1
- 1.1 Statistical coding . . . 1
- 1.2 Modeling . . . 2
- 1.3 Thesis . . . 3
- 1.4 Conventions . . . 3
  - 1.4.1 Notation . . . 4
  - 1.4.2 Compression gain . . . 4

2 Statistical Coding 7
- 2.1 Arithmetic coding . . . 8
  - 2.1.1 Basic algorithm for arithmetic coding . . . 8
  - 2.1.2 Implementation details . . . 9
  - 2.1.3 Use of integer arithmetic . . . 12
  - 2.1.4 Analysis of coding effects . . . 12
  - 2.1.5 Binary arithmetic coding . . . 15
- 2.2 Quasi-arithmetic coding . . . 16
  - 2.2.1 Development of binary quasi-arithmetic coding . . . 16
  - 2.2.2 Simplifications and applications of quasi-arithmetic coding . . . 19
  - 2.2.3 ϵ-partitions and ρ-partitions . . . 25
  - 2.2.4 Implementation of binary quasi-arithmetic coding . . . 27
  - 2.2.5 Analysis of binary quasi-arithmetic coding . . . 30
  - 2.2.6 Quasi-arithmetic coding for a multi-symbol alphabet . . . 33
- 2.3 Prefix codes . . . 34
  - 2.3.1 Huffman codes . . . 35
  - 2.3.2 Golomb and Rice codes . . . 36
  - 2.3.3 Selection of Golomb or Rice coding parameter . . . 37
- 2.4 Parallel coding . . . 40

2.4.1 Parallel Huffman coding 41
2.4.2 Parallel quasi-arithmetic coding 44

3 Lossless Image Compression 47
3.1 A paradigm for lossless image compression 47
3.2 Error modeling and coding 49
3.2.1 Precomputation of the distributions 50
3.2.2 Coding the error 51
3.3 PPPM: prediction by partial precision matching 52
3.3.1 Description of the PPPM algorithm 53
3.4 MLP: a multi-level progressive method 54
3.4.1 Description of the MLP algorithm 56
3.4.2 Error modeling by variability index 57
3.5 FELICS: a fast, efficient, lossless image compression system 61
3.5.1 Description of the FELICS algorithm 61
3.6 Experimental results 64
3.6.1 Compression performance 65
3.6.2 Speed performance of FELICS 65

4 Text Compression 71
4.1 Adaptive and semi-adaptive models for text compression 71
4.1.1 Semi-adaptive codes 71
4.1.2 Adaptive codes 74
4.1.3 Adaptive codes in practice 76
4.2 Scaling 76
4.2.1 Analysis of scaling 77
4.3 Prediction by partial matching 84
4.3.1 Application of scaling analysis to higher order models 85
4.3.2 PPMD: an improvement to PPMC 86
4.4 Fast PPM text compression 87
4.4.1 Implementation 88
4.4.2 Experimental Results 90

5 Contributions 91

References 93

LIST OF TABLES

1.1 Measures of compression efficiency 5

2.1 Arithmetic coding table for $N = 8$ 20
2.2 Probability arrays for quasi-arithmetic coding 27
2.3 Quasi-arithmetic coding table for $N = 8$ 28
2.4 Delta array for quasi-arithmetic coding 29
2.5 Right branch array for quasi-arithmetic coding 29
2.6 Quasi-arithmetic coding output example 30
2.7 Huffman coding example 35
2.8 Golomb and Rice code examples 36

3.1 Laplace distribution variances to guarantee coding loss of less than 0.005 bit per pixel. 51
3.2 Coefficients used in MLP for 16-point midpoint polynomial interpolation. 56
3.3 Image compression results, expressed in percent log ratio 66
3.4 Image compression results, expressed in compression ratio 67
3.5 Image compression results, expressed in bits per pixel 68
3.6 Compression ratios and encoding throughput for FELICS 69

4.1 PPM escape and symbol probabilities 84
4.2 Comparison of PPMC and PPMD 87
4.3 Compression and throughput results for Fast PPM 90

LIST OF FIGURES

2.1 Subdivision of current interval in arithmetic coding 8
2.2 Interval expansion process in arithmetic coding 10
2.3 Parallel reallocation coding 42

3.1 Comparison of Laplace and normal distributions 50
3.2 PPPM prediction and variance estimation contexts 52
3.3 MLP last level prediction neighborhood 55
3.4 Comparison of continuous Laplace distribution and actual distribution of prediction errors 57
3.5 Histograms showing distributions of prediction errors 59
3.6 Schematic probability distribution of intensity values in FELICS 62
3.7 Sample Landsat Thematic Mapper images 64

1

INTRODUCTION TO DATA COMPRESSION

DATA CAN BE COMPRESSED whenever some patterns of data symbols are more likely to occur than others. Shannon [67] showed that for the best possible compression code (in the sense of minimum average code length), the output length contains a contribution of $-\log_2 p$ bits from the encoding of each symbol whose probability of occurrence is p. It is possible to use various statistical coding techniques to encode any file using close-to-optimal code length, given a particular model of the data. It is desirable to use a good model of the source of the data so that the statistical coder can work with accurate probabilities. The separation of the compression process into coding and modeling is due to Rissanen and Langdon [59].

In this chapter we give a brief overview of both modeling and coding. We introduce arithmetic codes and distinguish them from prefix codes; we describe and mathematically analyze arithmetic codes in more detail in Chapter 2, where we also present *quasi-arithmetic coding*, our practical implementation of arithmetic coding. We then introduce the modeling problem for both grayscale images and text files; practical modeling methods are presented in Chapter 3 (images) and Chapter 4 (text). We summarize the contributions of this thesis in Chapter 5.

1.1 Statistical coding

The fundamental problem of lossless compression is to decompose a data set (for example, a text file or a grayscale image) into a sequence of events, then to encode the events using as few bits as possible. *Prefix codes* assign a single codeword to each possible event at each point in the coding process. The idea is to assign short codewords to more probable events and longer codewords to less probable events.

Arithmetic codes, in theory at least, assign one "codeword" to each possible data set. The codewords consist of half-open subintervals of the half-open unit interval $[0, 1)$, and are expressed by specifying enough bits to distinguish the final subinterval from all other possible final subintervals. As with prefix codes, shorter codes correspond to more probable input data sets. In practice, the subinterval is refined incrementally

using the probabilities of the individual events, with bits being output as soon as they are known. Arithmetic codes almost always give better compression than prefix codes, but they lack the direct correspondence between the events in the input data set and bits or groups of bits in the coded output file.

Shannon [67] showed that for a given set of probabilities $P = \{p_1, \ldots, p_n\}$, the optimal expected number of code bits is

$$\sum_{i=1}^{n} -p_i \log_2 p_i,$$

a quantity which he called the *entropy* of P, usually denoted by $H(P)$. The goal of statistical coding procedures is to assign codewords so that $-\log_2 p$ bits are output for an event with probability p; if we can do that, we have an optimal code.

1.2 Modeling

A statistical coder must work in conjunction with a modeler that estimates the probability of each possible event at each point in the coding. The model need not describe the process that generates the data; it merely has to deterministically provide probability information. The probabilities do not even have to be particularly accurate, but the more accurate they are, the better the compression will be. If the probabilities are wildly inaccurate, the file may even be expanded rather than compressed, but the original data can still be recovered.

Arithmetic coding allows us to compress a file as well as possible for a given model of the data. To obtain maximum compression of a file, we need both a good model and an efficient way of representing (or learning) the model. (This is related to Rissanen's *minimum description length* principle; he has investigated it thoroughly from a theoretical point of view [56,57,58].)

Most models for text compression involve estimating the probability p of a given symbol by

$$p = \frac{\text{weight of symbol}}{\text{total weight of all symbols}},$$

which we can then encode in $-\log_2 p$ bits using exact arithmetic coding. The weight of a symbol is usually based on the number of occurrences of the symbol, either in the entire file or in a particular context.

To ensure decodability, the encoder is limited to the use of model information that is available to the decoder. There are no other restrictions on the model; in particular, it can change as the file is being encoded. In Chapter 4 we describe several typical models for context-independent text compression. The models can be *adaptive* (dynamically estimating the probability of each symbol based on all symbols that precede it), *semi-adaptive* (using a preliminary pass of the input file to gather statistics), or *non-adaptive* (using fixed probabilities for all files). Non-adaptive models are not very interesting, since their effectiveness depends only on how well their probabilities happen to match

the statistics of the file being encoded; Bell, Cleary, and Witten show that the match can be arbitrarily bad [4]. In Section 4.1.2 we prove that one simple adaptive model gives exactly the same code length as a corresponding semi-adaptive decrementing model.

For both image and text compression, it is possible to use simple models, based solely on global event counts. To get good compression, however, we need models that take into account the structure of the data. For images, as we shall see in Chapter 3, this usually means using the numeric intensity values of nearly pixels to predict the intensity of each new pixel and using a suitable probability distribution to allow for noise and variation between regions within the image. For text, the previous letters form a context, in the manner of a Markov process, as we explain in Chapter 4.

1.3 Thesis

Our thesis is that high compression efficiency for text and images can be obtained by using sophisticated statistical compression techniques, and that greatly increased speed can be achieved at only a small cost in compression efficiency. Our emphasis is on elegant design and mathematical as well as empirical analysis.

We analyze arithmetic coding as it is commonly implemented and show rigorously that almost no compression is lost in the implementation. We show that high-efficiency lossless compression of both text and grayscale images can be obtained by using appropriate models in conjunction with arithmetic coding. We introduce a four-component paradigm for lossless image compression and present two methods that give state of the art compression efficiency. In the text compression area, we give a small improvement on the preferred method in the literature.

We show that we can often obtain significantly improved throughput at the cost of slightly reduced compression. The extra speed comes from simplified coding and modeling. Coding is simplified by using prefix codes when arithmetic coding is not necessary, and by using a new practical version of arithmetic coding, called *quasi-arithmetic coding*, when the precision of arithmetic coding is needed. We simplify image modeling, by using small prediction contexts and making plausible assumptions about the distributions of pixel intensity values. For text modeling we use self-organizing-list heuristics and low-precision statistics.

1.4 Conventions

We collect here most of the notation used in the remainder of this thesis. We also introduce a new measure of compression efficiency, called *compression gain*, useful for comparing compression methods and for discussing the effects of varying the details of a single technique.

1.4.1 Notation

We use the following notation throughout this thesis:

$$\begin{aligned}
t &= \text{length of the uncompressed file, in bytes;} \\
l &= \text{length of the compressed file, in bytes;} \\
L &= \text{length of the compressed file, in bits;} \\
n &= \text{number of symbols in the input alphabet;} \\
k &= \text{number of different alphabet symbols that occur in the file;} \\
a_i &= \text{the } i\text{th symbol in the alphabet;} \\
c_i &= \text{number of occurrences of symbol } a_i \text{ in the file;} \\
p_i &= c_i/t \quad \text{(the probability of symbol } a_i \text{ in the file);} \\
H(X) &= \textstyle\sum_i -x_i \log_2 x_i \quad \text{(the entropy implied by distribution } X\text{);} \\
A^{\overline{B}} &= A(A+1)\cdots(A+B-1) \quad \text{(the rising factorial function).}
\end{aligned}$$

In this thesis, results for a "typical" text file refer to a 100,000 byte uncompressed file that contains 100 different alphabet symbols out of a 256-symbol alphabet, i.e., $t = 100{,}000$, $n = 256$, and $k = 100$. We assume 8-bit bytes, and express code lengths in either bits or bytes depending on the context. In examples we generally show input symbols in italic type (*a*, *b*, *c*, *0*, *1*) or sans serif type (EOF, FOUND, NOT-FOUND, MPS, LPS, IGNORE), and we show output bits in bold face type (**0**, **1**, **f**, **follow**).

1.4.2 Compression gain

There is no single accepted method for reporting and comparing compression results. Several measures are used in the literature (see Table 1.1). All of the existing measures have two major theoretical shortcomings. First, they all express compression relative to the original file length. In practical applications this can be useful, but any such measure fails to discriminate between the effectiveness of the compression method and the inherent compressibility of the files. Second, none of the measures are cumulative: we cannot simply add together modeling and coding effects, or the effects of multiple cascaded compressors.

In [26] we introduce a new measure, which we call *compression gain*, that is additive and that can be used to measure compression with respect to any standard. In [29] we give an improved formulation that uses a more convenient scale; we define the compression gain by

$$\text{compression gain} = 100 \log_e \frac{\text{reference size}}{\text{compressed size}}, \tag{1.1}$$

where the reference size and compressed size are expressed in the same units, usually either total bytes in the file or average bits per pixel. Compression gain is expressed in

Measure	Formula	Example
Bits per input symbol (also called bit rate or average code length)	$\frac{L}{t}$	2.92 bits per pixel
Compression ratio	$\frac{t}{l}$	2.74 : 1
Relative compressed size	$\frac{l}{t}$	0.36 or 36%
Relative compression	$\frac{t-l}{t}$	0.64 or 64%
Compression gain, relative to original file	$100 \log_e \frac{t}{l}$	$100.9\frac{\circ}{\circ}$
Compression gain, relative to zero-order entropy	$100 \log_e \frac{tH_0}{L}$	$49.5\frac{\circ}{\circ}$
Compression gain, relative to JPEG lossless mode	$100 \log_e \frac{l_{\text{JPEG}}}{l}$	$6.1\frac{\circ}{\circ}$

Table 1.1: Measures of compression efficiency. Our new measure, *compression gain*, is illustrated on the last three lines. The original file is t bytes long, the compressed file is l bytes (L bits) long, and the zero-order entropy of the original file is H_0 bits per input symbol. The examples in the rightmost column are based on using the PPPM method on Band 3 of the Donaldsonville Landsat Thematic Mapper data set, a typical image described in Section 3.6. The image consists of t = 262,144 8-bit pixels, and the compressed file is l = 95,587 bytes long. This image has a zero-order entropy of H_0 = 4.784 bits per pixel (156,776 total bytes). The lossless mode of the JPEG standard, using two-point prediction, compresses the image to l_{JPEG} = 101,614 bytes. Note that the compression gain of JPEG lossless mode relative to the original file ($100 \log_e(262{,}144/101{,}614) = 94.8\frac{\circ}{\circ}$) can be obtained by simply subtracting the gain of PPPM relative to JPEG from the gain of PPPM relative to the original file.

percent log ratio, denoted by the $\frac{\circ}{\circ}$ sign. Since $\log_e(1+x) \approx x$ for small x, a difference of any given small percent log ratio means almost the same thing as the same difference expressed as an ordinary percentage; since $\log(A/B) = -\log(B/A)$ we can express the difference in either direction with no change in its absolute value. For example, if method A compresses a file to 99 bytes and method B compresses the same file to 101 bytes, we say that method A outcompresses method B by $\log_e(101/99) \approx 2.00007\frac{\circ}{\circ}$ or equivalently that method B is $2.00007\frac{\circ}{\circ}$ worse than method A. Using ordinary percentages we could say that method A is $2/101 \approx 1.980198\%$ better than method B, but we would have to say that method B is $2/99 \approx 2.020202\%$ worse than method A.

For the reference size we may choose the size of the original file, the size of the

compressed file produced by some standard lossless compression method, or the zero-order entropy of the file (that is, the entropy calculated from the occurrence counts of the alphabet symbols within the file). In the remainder of this thesis we express gains with respect to standard lossless methods: the lossless mode of the JPEG standard for images and the PPMC method for text files.

The logarithm in Formula 1.1 gives us additivity, so we can simply subtract compression gains to compare any two compression methods. The *gain/loss* terminology is natural: larger numbers mean better compression. In addition, coding effects, often expressed as percentages of code length, can now be given as losses that can simply be subtracted from the compression gain of the modeling method. In this form we can clearly separate coding and modeling, and we can see how tiny the coding effects usually are.

2

STATISTICAL CODING

THERE IS A SPECTRUM of statistical coding techniques. At one end of the spectrum, we have (pure) arithmetic coding, which can handle the probabilities produced by any model and produce an exactly optimal code; the cost of this flexibility is slow speed because of the exact arithmetic required. At the other end of the spectrum are Rice codes. They are extremely fast and straightforward to implement in software or hardware, but their compression performance is limited because they assume that the model has a probability distribution of a particular form. Rice codes are a special case of Golomb codes, which in turn are a special case of the widely-known Huffman codes. Our own *quasi-arithmetic* codes occupy an interesting place on the spectrum. Originally designed as a practical implementation of arithmetic coding, they can also be thought of as an extension of Huffman codes, adding a small amount of state information. Quasi-arithmetic codes bridge the gap between the slow, precise arithmetic codes, whose main feature is the use of essentially continuous probability information, and the much faster but less flexible prefix codes, which use discretized probability information.

In theory, arithmetic coding gives optimal compression: if we have a model that provides correct probabilities of the events comprising the input data, an arithmetic coder uses on average the minimum number of bytes to encode the data. In Section 2.1 we describe the arithmetic coding process in detail, and show how we can obtain optimal compression. However, to achieve optimal compression, we must use exact real arithmetic (or at least exact rational arithmetic if the probabilities are rational). Practical versions of arithmetic coding use fixed point approximations to exact arithmetic, and obtain very nearly optimal compression. Here we show rigorously just how close to optimal our compression must be; we defer analysis of modeling effects to Chapter 4.

Arithmetic coding as it is usually implemented is slow because of the multiplications required. In Section 2.2 we present a new method for approximate arithmetic coding called *quasi-arithmetic coding*. The idea is to use low enough precision that we can do all the arithmetic ahead of time, and store the results in lookup tables. We show how to construct the tables, and we also prove bounds on the loss of compression efficiency.

In an *instantaneous* code, the decoder knows immediately when a complete symbol has been received, and does not have to look further to correctly identify the symbol.

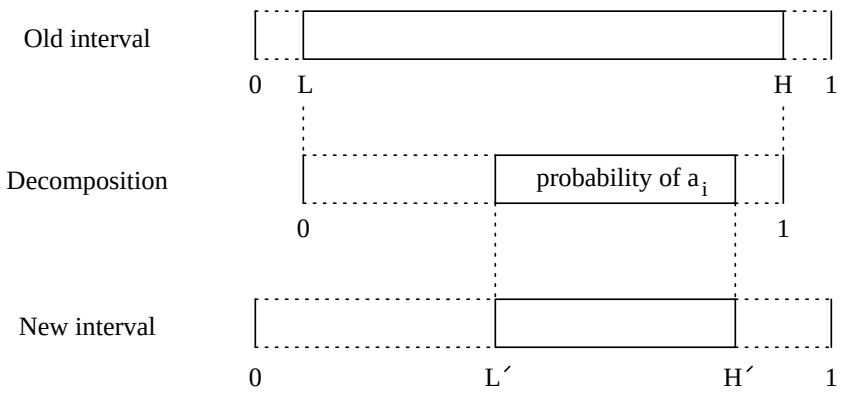


Figure 2.1: Subdivision of the current interval in arithmetic coding based on the probability of the input symbol a_i that occurs next.

Instantaneous codes are just those with the *prefix property*: no code word is a prefix of another code word [25, pages 53–55]. Coding is simple for an instantaneous code – the code bits for all events are disjoint and independent. In Section 2.3 we discuss three methods of coding that produce *instantaneous* codes, namely Huffman coding, Golomb coding, and Rice coding. We also present a fast new method for estimating the coding parameter in Golomb and Rice codes, and prove its effectiveness. Golomb and Rice codes are used in Chapters 3 and 4 in designing fast coders for both images and text.

In Section 2.4 we show that in some cases, particularly when compressing images, we can perform both encoding and decoding in parallel if many processors are available. Parallel compression and decompression is possible both with prefix codes and with quasi-arithmetic codes.

2.1 Arithmetic coding

In this section we explain how arithmetic coding works and give implementation details; our treatment is based on that of Witten, Neal, and Cleary [77]. We point out the usefulness of binary arithmetic coding, that is, coding with a two-symbol alphabet. Our focus is on encoding, but the decoding process is similar.

2.1.1 Basic algorithm for arithmetic coding

The algorithm for encoding a file using arithmetic coding works conceptually as follows:

1. We begin with a "current interval" $[L, H)$ initialized to $[0, 1)$.

2. For each symbol of the file, we perform two steps (see Figure 2.1):

   (a) We subdivide the current interval into subintervals, one for each possible alphabet symbol. The size of a symbol's subinterval is proportional to the

estimated probability that the symbol will be the next symbol in the file, according to the model of the input.

(b) We select the subinterval corresponding to the symbol that actually occurs next in the file, and make it the new current interval.

3. We output enough bits to distinguish the final current interval from all other possible final intervals.

The length of the final subinterval is clearly equal to the product of the probabilities of the individual symbols, which is the probability p of the particular sequence of symbols in the file. The final step uses almost exactly $-\log_2 p$ bits to distinguish the file from all other possible files. We need some mechanism to indicate the end of the file, either a special end-of-file symbol coded just once, or some external indication of the file's length.

In step 2, we need to compute only the subinterval corresponding to the symbol a_i that actually occurs. To do this we need two cumulative probabilities, $P_C = \sum_{k=1}^{i-1} p_k$ (the "cumulative" probability) and $P_N = \sum_{k=1}^{i} p_k$ (the "next cumulative" probability). The new subinterval is $[L + P_C(H - L),\ L + P_N(H - L))$. The need to maintain and supply cumulative probabilities requires the model to have a complicated data structure; Moffat [44] investigates this problem, and concludes for a multi-symbol alphabet that binary search trees are about twice as fast as move-to-front lists.

Example 1: We illustrate a non-adaptive code, encoding the file containing the symbols *bbb* using arbitrary fixed probability estimates $p_a = 0.4$, $p_b = 0.5$, and $p_{\mathsf{EOF}} = 0.1$. Encoding proceeds as follows:

Current Interval	Action	Subintervals			Input
		a	*b*	EOF	
[0.0000, 1.0000)	Subdivide	[0.0000, 0.4000)	[0.4000, 0.9000)	[0.9000, 1.0000)	*b*
[0.4000, 0.9000)	Subdivide	[0.4000, 0.6000)	[0.6000, 0.8500)	[0.8500, 0.9000)	*b*
[0.6000, 0.8500)	Subdivide	[0.6000, 0.7000)	[0.7000, 0.8250)	[0.8250, 0.8500)	*b*
[0.7000, 0.8250)	Subdivide	[0.7000, 0.7500)	[0.7500, 0.8125)	[0.8125, 0.8250)	EOF
[0.8125, 0.8250)					

The final interval is [0.8125, 0.8250), which in binary is approximately [0.11010 00000, 0.11010 01100). We can uniquely identify this interval by outputting **11010 00**. According to the fixed model, the probability p of this particular file is $(0.5)^3(0.1) = 0.0125$ (exactly the size of the final interval) and the code length (in bits) should be $-\log_2 p \approx 6.322$. In practice we have to output 7 bits. □

2.1.2 Implementation details

The basic implementation of arithmetic coding described above has two major difficulties: the shrinking current interval requires the use of high precision arithmetic, and no output is produced until the entire file has been read. The most straightforward solution to both of these problems is to output each leading bit as soon as it is known, and then to double the length of the current interval so that it reflects only the unknown

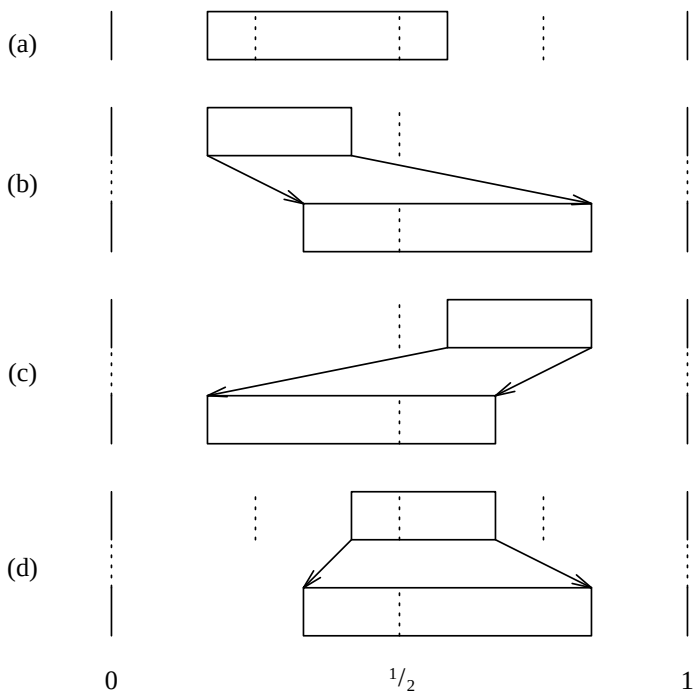


Figure 2.2: Interval expansion process in arithmetic coding. (a) No expansion. (b) Interval in $[0, 1/2)$. (c) Interval in $[1/2, 1)$. (d) Interval in $[1/4, 3/4)$ (follow-on case).

part of the final interval. Witten, Neal, and Cleary [77] add a clever mechanism for preventing the current interval from shrinking too much when the endpoints are close to $1/2$ but straddle $1/2$. In that case we do not yet know the next output bit, but we do know that whatever it is, the *following* bit will have the opposite value; we merely keep track of that fact, and expand the current interval symmetrically about $1/2$. This follow-on procedure may be repeated any number of times, so the current interval size is always longer than $1/4$.

Mechanisms for incremental transmission and fixed precision arithmetic have been developed through the years by Pasco [49], Rissanen [60], Rubin [65], Rissanen and Langdon [61], Guazzo [23], and Witten, Neal, and Cleary [77]. The bit-stuffing idea of Langdon and others at IBM that limits the propagation of carries in the additions is roughly equivalent to the follow-on procedure described above.

We now describe in detail how the coding and interval expansion work. This process takes place immediately after the selection of the subinterval corresponding to an input symbol. We repeat the following steps (illustrated schematically in Figure 2.2) as many times as possible:

a. If the new subinterval is not entirely within one of the intervals $[0, 1/2)$, $[1/4, 3/4)$, or $[1/2, 1)$, we stop iterating and return.

b. If the new subinterval lies entirely within $[0, 1/2)$, we output **0** and any following **1**s left over from previous symbols; then we double the size of the interval $[0, 1/2)$,

expanding toward the right.

c. If the new subinterval lies entirely within $[1/2, 1)$, we output **1** and any following **0**s left over from previous symbols; then we double the size of the interval $[1/2, 1)$, expanding toward the left.

d. If the new subinterval lies entirely within $[1/4, 3/4)$, we keep track of this fact for future output by incrementing the follow count; then we double the size of the interval $[1/4, 3/4)$, expanding in both directions away from the midpoint.

Example 2: We show the details of encoding the same file as in Example 1.

Current Interval	Action	Subintervals a	Subintervals b	Subintervals EOF	Input
$[0.00, 1.00)$	Subdivide	$[0.00, 0.40)$	$[0.40, 0.90)$	$[0.90, 1.00)$	b
$[0.40, 0.90)$	Subdivide	$[0.40, 0.60)$	$[0.60, 0.85)$	$[0.85, 0.90)$	b
$[0.60, 0.85)$	Output **1**				
	Expand $[1/2, 1)$				
$[0.20, 0.70)$	Subdivide	$[0.20, 0.40)$	$[0.40, 0.65)$	$[0.65, 0.70)$	b
$[0.40, 0.65)$	**follow**				
	Expand $[1/4, 3/4)$				
$[0.30, 0.80)$	Subdivide	$[0.30, 0.50)$	$[0.50, 0.75)$	$[0.75, 0.80)$	EOF
$[0.75, 0.80)$	Output **10**				
	Expand $[1/2, 1)$				
$[0.50, 0.60)$	Output **1**				
	Expand $[1/2, 1)$				
$[0.00, 0.20)$	Output **0**				
	Expand $[0, 1/2)$				
$[0.00, 0.40)$	Output **0**				
	Expand $[0, 1/2)$				
$[0.00, 0.80)$	Output **0**				

The "**follow**" output in the sixth line indicates application of the follow-on procedure: we keep track of our knowledge that the next output bit will be followed by its opposite; this "opposite" bit is the **0** output in the ninth line. The encoded file is **11010 00**, as before. The final **0** eliminates the subinterval $[0.800, 1.000)$ from consideration. □

Clearly the current interval contains some information about the preceding inputs; this information has not yet been output, so we can think of it as the coder's state. If a is the length of the current interval, the state holds $-\log_2 a$ bits not yet output. In the basic method (illustrated by Example 1) the state contains *all* the information about the output, since nothing is output until the end. In the implementation illustrated by Example 2, the state always contains fewer than two bits of output information, since the length of the current interval is always more than $1/4$. The final state in Example 2 is $[0, 0.8)$, which contains $-\log_2 0.8 \approx 0.322$ bits of information; the final output bit transmits this information.

2.1.3 Use of integer arithmetic

In practice, the arithmetic can be done by storing the current interval in sufficiently long integers rather than in floating point or exact rational numbers. (We can think of Example 2 as using the integer interval $[0, 100)$ by omitting all the decimal points.) We also use integers for the frequency counts used to estimate symbol probabilities. The subdivision process involves selecting non-overlapping intervals (of length at least 1) with lengths approximately proportional to the counts. To encode symbol a_i we need two cumulative counts, $C = \sum_{k=1}^{i-1} c_k$ and $N = \sum_{k=1}^{i} c_k$, and the sum T of all counts, $T = \sum_{k=1}^{n} c_k$. The new subinterval is $[L + \lfloor \frac{C(H-L)}{T} \rfloor, L + \lfloor \frac{N(H-L)}{T} \rfloor)$. (In this discussion we continue to use half-open intervals as in the real arithmetic case. In implementations [77] it is more convenient to subtract 1 from the right endpoints and use closed intervals.)

Example 3: Suppose that at a certain point in the encoding we have symbol counts $c_a = 4$, $c_b = 5$, and $c_{\mathsf{EOF}} = 1$ and current interval $[25, 89)$ from the full interval $[0, 128)$. Let the next input symbol be b. The total count T is 10, and the cumulative counts for b are $C = 4$ and $N = 9$. The new interval is $[25 + \lfloor \frac{4(89-25)}{10} \rfloor, 25 + \lfloor \frac{9(89-25)}{10} \rfloor) = [50, 82)$; we then increment the follow-on count and expand the interval once about the midpoint 64, giving $[36, 100)$. No further expansion or output is possible. □

2.1.4 Analysis of coding effects

In this section we prove analytically that the coding effects of arithmetic coding (as distinguished from modeling effects) are insignificant; hence for all practical purposes arithmetic coding realizes the minimum code length permitted by the model. Empirical evidence that the coding effects are negligible appears in [4,77].

Periodic scaling

The symbol counts used to compute the probability information used by an arithmetic coder can become so large that time-consuming extended precision arithmetic becomes necessary; even worse, the counts may overflow the registers used to store them. To avoid the problem of large counts, Witten, Neal, and Cleary periodically *scale* the symbol counts by halving all of them when their sum reaches a threshold $2B$; this effectively divides the file into blocks of length B. The relative proportions of the counts remain the same, so the implied probabilities are the same.

Since the counts are maintained as integers, it is necessary to round their values during the scaling process. To prevent any count from falling to 0, fractional counts are rounded to the next higher integer. This gives slightly more weight to symbols whose count happens to be odd when scaling occurs. (There are other methods of preventing counts from becoming 0. We can truncate all fractional counts except those that result in a scaled count of 0. If we are maintaining only cumulative symbol counts we can truncate fractional counts, then add the index of each symbol to its cumulative

count; this is useful in a hardware-based solution. Moffat [45] rounds fractional counts upwards, but retains more precision by using fixed-point counts with a few fractional bits; this allows counts to become almost 0. With three fractional bits, the counts can become as small as $^1/_8$.)

The scaling process introduces a modeling effect. Less recent events are effectively given a lower weight, so the most recently encountered part of the file has more influence on the model statistics. Since many files exhibit locality of reference, this recency effect often improves compression. We fully analyze the modeling effect of scaling in Section 4.2.1.

Rounding counts to integers

Here we analyze the coding effect of periodic scaling. The rounding mechanism used by Witten, Neal, and Cleary leads to slightly inaccurate probabilities, which in turn can increase the code length. We bound the amount of the increase in the following theorem.

Theorem 1 *Rounding counts up to the next higher integer increases the code length for the file by no more than $n/2B$ bits per input symbol.*

Proof: Each symbol whose weight is rounded up causes the denominators of all probabilities in the next block to be too large, by $^1/_2$. If r is the number of symbols subject to roundup, $r/2$ of the denominators in computing the coding interval will be approximately T instead of $T/2$, each adding one bit to the code length of the block, so the block's code length will be $r/2$ bits longer. The effect for the entire file (t/B blocks) is $rt/2B$ bits, or, since $r \leq n$, at most $n/2B$ bits per input symbol. □

This effect is typically 0.02 bit or less per input symbol.

Using integer arithmetic

Witten, Neal, and Cleary use integers from a large fixed range, typically $[0, 8B]$, instead of using exact rational arithmetic, and they transmit encoded bits as soon as they know them instead of waiting until the entire string has been encoded. Once a bit is known, half of the range is known *not* to be relevant, so we can expand the other half to fill the entire range. The result is nearly the same compression efficiency as with exact rational arithmetic.

As is apparent from the following description of the coding section of the Witten-Neal-Cleary algorithm, expanding the range is only approximate:

1. We select a subrange of the current interval $[L, H)$ whose length within $[L, H)$ is at least 1 and approximately proportional to p.

2. We repeat the following steps as many times as possible:

(a) If the new subrange is not entirely in the lower, middle, or upper half of the full range of values, we return.

(b) If the new subrange is in the lower or upper half, we output **0** or **1** respectively, plus any **follow** bits left over from previous symbols. If the subrange is entirely in the middle half, we add 1 to the follow-on count to keep track of this fact for future output.

(c) We expand the subrange:

   i. We shift the subrange to ignore the part of the full range in which the subrange is known not to lie.

   ii. We double both L and H.

In this algorithm, any roundoff error in selecting the first subrange will propagate through the entire interval expansion process. In the worst case, a symbol with a count of 1 could result in a subrange of length 1, even though the unrounded subrange size might be just below 2. In effect, this would assign a symbol probability of only half the correct value, resulting in a code length one bit too long. In practice, however, the code length error in one symbol is seldom anywhere near that large, and because the errors can be of either sign and have an approximately symmetrical distribution with mean 0, the average error is usually very small. Witten, Neal, and Cleary empirically estimate it at 10^{-4} bit per input symbol.

In order to get a rigorous bound on the compression loss, we analyze a new algorithm that maintains full precision when expanding the range. At each step in the expansion process, we adjust the range by adding either 0 or 1 to L and independently adding either 0 or 1 to H according to the exact result of the initial subrange selection. To prevent the overlap of subranges from different alphabet symbols, we add 1 to L and subtract 1 from H when no more expansion is possible. This last step always makes the code length longer than that of exact arithmetic coding, but by a tiny amount, as shown in the following theorem:

Theorem 2 *When we use the high precision algorithm for expanding the subrange, the average code length is greater than the optimal average code length by an amount bounded by* $2/(B \ln 2)$ *bits per input symbol.*

Proof: After the expansion process, the subinterval size is at least $2B$. As a result of rounding while expanding, *low* could be too high by as much as 1, and *high* could be too low by as much as 1, so the subinterval could be too short by as much as 2. (It could also be too long by as much as 2.) The final overlap-prevention step always shortens the subinterval by 2, so the subinterval is always too short, possibly by as much as 4. An interval too short by 4 could be too short by a factor of as much as $(2B-4)/2B$, which corresponds to a code length increase of $-\log_2((B-2)/B) \sim 2/(B \ln 2)$. □

A loss of $2/(B \ln 2)$ bits per input symbol is negligible, about 4×10^{-4} bit per symbol for typical parameters. In practice the high precision algorithm gives exactly the same

compression as the algorithm of Witten, Neal, and Cleary, but its compression loss is *provably* small. The high precision algorithm is only of theoretical interest; in practice it complicates the program without improving compression.

Encoding end-of-file

The end of the file must be explicitly indicated when we use arithmetic coding. The extra code length required is typically very small and is often ignored; for completeness, we provide a brief analysis of the end-of-file effect.

Witten, Neal, and Cleary introduce a special low-weight symbol in addition to the normal alphabet symbols; it is encoded once, at the end of the file. In the following theorem we bound the cost of identifying end-of-file by this method:

Theorem 3 *The use of a special end-of-file symbol results in additional code length of less than* $t/(B \ln 2) + \log_2 B + 10$ *bits.*

Proof: The cost has four components:

- at most $\log_2 B + 1$ bits to encode the end-of-file symbol (since its probability must be at least as large as the smallest possible probability, $1/2B$)
- fewer than $t/(B \ln 2)$ bits in wasted code space to allow end-of-file at any point (each probability can be reduced by a factor of between $(2B-1)/2B$ and $(B-1)/B$, resulting in a loss of between $-\log_2((2B-1)/2B) \approx 1/(2B \ln 2)$ and $-\log_2((B-1)/B) \approx 1/(B \ln 2)$ bits per symbol)
- two disambiguating bits after the end-of-file symbol
- up to seven bits to fill the last byte.

□

An alternative, transmitting the length of the original file before its encoding, reduces the cost to between $\log_2 t$ and $2 \log_2 t$ bits by using an appropriate encoding of integers [13,69,75], but requires the file length to be known before encoding can begin.

The end-of-file cost using either of these methods is negligible for a typical file, less than 0.01 bit per input symbol.

2.1.5 Binary arithmetic coding

The preceding discussion has focused on coding with a multi-symbol alphabet, although in principle it applies to a binary alphabet as well. It is useful to distinguish the two cases since both the coder and the interface to the model are simpler for a binary alphabet. The coding of bilevel images, an important problem with a natural two-symbol alphabet, often produces probabilities close to 1, suggesting the use of arithmetic coding to obtain good compression. Historically, much of the arithmetic coding research by Rissanen, Langdon, and others at IBM has focused on bilevel images [39]. The

Q-Coder [1,37,41,50,51,52] is a binary arithmetic coder; work by Rissanen and Mohiuddin [62,63], Chevion *et al.* [6] , and Feygin *et al.* [16] extends some of the Q-Coder ideas to multi-symbol alphabets. The quasi-arithmetic coder discussed in Section 2.2 is formulated as a binary coder, though in Section 2.2.6 we show that it can be extended to a multi-symbol alphabet.

In most text and image compression applications, a multi-symbol alphabet is more natural, but even then we can map the possible symbols to the leaves of a binary tree, and encode an event by traversing the tree and encoding a decision at each internal node. (As an important special case, we can even use a list.) When we use a binary tree, the model no longer has to maintain and produce cumulative probabilities; a single probability at each node suffices to encode each decision. Calculating the new current interval is also simplified, since just one endpoint changes after each decision. On the other hand, we now usually have to encode more than one event for each input symbol, and we have a new data structure problem, maintaining the coding trees efficiently without using excessive space. The smallest average number of events coded per input symbol occurs when the tree is a Huffman tree, since such trees have minimum average weighted path length; however, maintaining such trees dynamically is complicated [10, 36,73,74].

2.2 Quasi-arithmetic coding

The primary disadvantage of arithmetic coding is its slowness. Since small errors in probability estimates cause very small increases in code length, we expect that by introducing approximations into the arithmetic coding process in a controlled way we can improve coding speed without significantly degrading compression performance. In the Q-Coder work at IBM, the time-consuming multiplications are replaced by additions and shifts, and low-order bits are ignored.

In [27] we describe a different approach to approximate arithmetic coding. Recalling that the fractional bits characteristic of arithmetic coding are stored as state information in the coder, we reduce the number of possible states, and replace arithmetic operations by table lookups. Here we present this reduced precision binary arithmetic coder, which we call a *quasi-arithmetic coder*, and develop it through a series of examples. It should be noted that the compression is still completely reversible; using reduced precision merely affects the average code length.

2.2.1 Development of binary quasi-arithmetic coding

The number of possible states (after applying the interval expansion procedure) of an arithmetic coder using the integer interval $[0, N)$ is $3N^2/16$. If we can reduce the number of states to a more manageable level, we can precompute all state transitions and outputs and substitute table lookups for arithmetic in the coder. The obvious way to reduce the number of states is to reduce N. The value of N should be a power of 2; its value must be at least 4.

Example 4: The simplest non-trivial coders have $N = 4$, and have only three states. By applying the arithmetic coding algorithm in a straightforward way, we obtain the following coding table. A "**follow**" output indicates application of the follow-on procedure described in Section 2.1.2.

		0 input		*1* input	
State	Prob{*0*}	Output	Next state	Output	Next state
$[0,4)$	$0 < p < 1-\alpha$	**00**	$[0,4)$	–	$[1,4)$
	$1-\alpha \le p \le \alpha$	**0**	$[0,4)$	**1**	$[0,4)$
	$\alpha < p < 1$	–	$[0,3)$	**11**	$[0,4)$
$[0,3)$	$0 < p < 1/2$	**00**	$[0,4)$	**follow**	$[0,4)$
	$1/2 \le p < 1$	**0**	$[0,4)$	**10**	$[0,4)$
$[1,4)$	$0 < p < 1/2$	**01**	$[0,4)$	**1**	$[0,4)$
	$1/2 \le p < 1$	**follow**	$[0,4)$	**11**	$[0,4)$

The value of the cutoff probability α in state $[0,4)$ depends on assumptions about the input data and the goal of the coder. Under reasonable assumptions it can be shown using the method of Section 2.2.5 to be $1/\log_2 3 \approx 0.631$. The amount of excess code length is not very sensitive to the value of α; any value from about 0.55 to 0.73 gives less than $1\frac{0}{0}$ coding inefficiency. □

Half-integer subdivisions

Additional transitions are available when subdivision points fall at half-integer values if both resulting subintervals can be expanded. This happens for half-integer subdivision points in the second quarter of the full range when the high end of the current subrange is in the third quarter, and for half-integer subdivision points in the third quarter of the full range when the low end of the current subrange is in the second quarter.

Example 5: If we are currently in state $[0,3)$ of a coder with $N = 4$, and the inputs are equally likely, we can subdivide the interval into $[0, 3/2)$ for a *0* input and $[3/2, 3)$ for a *1* input. In the first case we output **0** and expand to $[0,3)$; in the second we keep track of a **follow** bit and expand to $[1,4)$. Similarly, we can subdivide state $[1,4)$ at $5/2$. We can thus expand the coding table in Example 4 to obtain the following table.

State	Prob{*0*}	*0* input		*1* input	
		Output	Next state	Output	Next state
$[0,4)$	$0 < p < 1-\alpha$	**00**	$[0,4)$	–	$[1,4)$
	$1-\alpha \le p \le \alpha$	**0**	$[0,4)$	**1**	$[0,4)$
	$\alpha < p < 1$	–	$[0,3)$	**11**	$[0,4)$
$[0,3)$	$0 < p < 1-\beta$	**00**	$[0,4)$	**follow**	$[0,4)$
	$1-\beta \le p \le \beta$	**0**	$[0,3)$	**follow**	$[1,4)$
	$\beta < p < 1$	**0**	$[0,4)$	**10**	$[0,4)$
$[1,4)$	$0 < p < 1-\beta$	**01**	$[0,4)$	**1**	$[0,4)$
	$1-\beta \le p \le \beta$	**follow**	$[0,3)$	**1**	$[1,4)$
	$\beta < p < 1$	**follow**	$[0,4)$	**11**	$[0,4)$

Again we must choose the cutoff probabilities α and β. We should use the same value of α as in Example 4; using the method of Section 2.2.5 we find that $\beta = \log_2(3/2) \approx 0.585$. □

The use of half-integer subdivisions improves the theoretical compression efficiency of quasi-arithmetic coding. In small coders like the $N = 4$ coder of Examples 4 and 5, the improvement can be noticeable; such coders may be practical when hard coded. Otherwise, the improvement is small and the construction of the coder becomes less regular and more complex, so we shall ignore this refinement for the remainder of this discussion.

Reassignment of input symbols

Arithmetic coding does not mandate any particular assignment of subintervals to input symbols; all that is required is that subinterval lengths be approximately proportional to symbol probabilities and that the decoder make the same assignment as the encoder. In Example 4 we uniformly assigned the left subinterval to input symbol *0*. By preventing the longer subinterval from straddling the midpoint whenever possible, we can sometimes obtain a simpler coder that never requires the follow-on procedure; it may also use fewer states.

Example 6: This coder assigns the *right* subinterval to *0* in lines 4 and 7 of Example 4, eliminating the need for using the follow-on procedure; otherwise it is the same as Example 4.

State	Prob{*0*}	*0* input Output	*0* input Next state	*1* input Output	*1* input Next state
$[0,4)$	$0 < p < 1-\alpha$	**00**	$[0,4)$	–	$[1,4)$
	$1-\alpha \le p \le \alpha$	**0**	$[0,4)$	**1**	$[0,4)$
	$\alpha < p < 1$	–	$[0,3)$	**11**	$[0,4)$
$[0,3)$	$0 < p < 1/2$	**10**	$[0,4)$	**0**	$[0,4)$
	$1/2 \le p < 1$	**0**	$[0,4)$	**10**	$[0,4)$
$[1,4)$	$0 < p < 1/2$	**01**	$[0,4)$	**1**	$[0,4)$
	$1/2 \le p < 1$	**1**	$[0,4)$	**01**	$[0,4)$

□

Using more states

If we design a coder with more states, we obtain a more fine-grained set of probabilities.

Example 7: In Table 2.1 on the following two pages we show a coder obtained by letting $N = 8$ and allowing all possible subdivisions. The probability ranges are obtained by following the method described in Section 2.2.4. Output symbol **f** indicates application of the follow-on procedure.

This coder is easily programmed and extremely fast. Its only shortcoming is that on average high-probability symbols require $1/4$ bit (corresponding to a more-probable symbol probability of $2^{-1/4} \approx 0.841$) no matter how high the actual probability is. In Section 2.2.4 we give a practical implementation of the same coder. □

2.2.2 Simplifications and applications of quasi-arithmetic coding

Langdon and Rissanen [39] suggest identifying the input symbols as the less probable symbol (LPS) and more probable symbol (MPS) rather than as *0* and *1*. By doing this we can often combine transitions and eliminate states.

Example 8: We modify Example 6 to use the MPS/LPS idea. We are able to reduce the coder to just two states.

State	Prob{MPS}	LPS input Output	LPS input Next state	MPS input Output	MPS input Next state
$[0,4)$	$1/2 \le p \le \alpha$	**0**	$[0,4)$	**1**	$[0,4)$
	$\alpha < p < 1$	**00**	$[0,4)$	–	$[1,4)$
$[1,4)$	$1/2 \le p < 1$	**01**	$[0,4)$	**1**	$[0,4)$

□

Start state	Probability of *0* input	*0* input		*1* input	
		Output	Next state	Output	Next state
$[0,8)$	0.000 – 0.182	**000**	$[0,8)$	–	$[1,8)$
	0.182 – 0.310	**00**	$[0,8)$	–	$[2,8)$
	0.310 – 0.437	**0**	$[0,6)$	–	$[3,8)$
	0.437 – 0.563	**0**	$[0,8)$	**1**	$[0,8)$
	0.563 – 0.690	–	$[0,5)$	**1**	$[2,8)$
	0.690 – 0.818	–	$[0,6)$	**11**	$[0,8)$
	0.818 – 1.000	–	$[0,7)$	**111**	$[0,8)$
$[0,7)$	0.000 – 0.208	**000**	$[0,8)$	–	$[1,7)$
	0.208 – 0.355	**00**	$[0,8)$	–	$[2,7)$
	0.355 – 0.500	**0**	$[0,6)$	–	$[3,7)$
	0.500 – 0.645	**0**	$[0,8)$	**1**	$[0,6)$
	0.645 – 0.792	–	$[0,5)$	**1f**	$[0,8)$
	0.792 – 1.000	–	$[0,6)$	**110**	$[0,8)$
$[0,6)$	0.000 – 0.244	**000**	$[0,8)$	–	$[1,6)$
	0.244 – 0.415	**00**	$[0,8)$	**f**	$[0,8)$
	0.415 – 0.585	**0**	$[0,6)$	**f**	$[2,8)$
	0.585 – 0.756	**0**	$[0,8)$	**10**	$[0,8)$
	0.756 – 1.000	–	$[0,5)$	**101**	$[0,8)$
$[0,5)$	0.000 – 0.293	**000**	$[0,8)$	–	$[1,5)$
	0.293 – 0.500	**00**	$[0,8)$	**f**	$[0,6)$
	0.500 – 0.707	**0**	$[0,6)$	**ff**	$[0,8)$
	0.707 – 1.000	**0**	$[0,8)$	**100**	$[0,8)$
$[1,8)$	0.000 – 0.208	**001**	$[0,8)$	–	$[2,8)$
	0.208 – 0.355	**0f**	$[0,8)$	–	$[3,8)$
	0.355 – 0.500	**0**	$[2,8)$	**1**	$[0,8)$
	0.500 – 0.645	–	$[1,5)$	**1**	$[2,8)$
	0.645 – 0.792	–	$[1,6)$	**11**	$[0,8)$
	0.792 – 1.000	–	$[1,7)$	**111**	$[0,8)$

Table 2.1: Complete arithmetic coding code table for $N = 8$.

Start state	Probability of *0* input	*0* input		*1* input	
		Output	Next state	Output	Next state
$[1,7)$	0.000 – 0.244	**001**	$[0,8)$	–	$[2,7)$
	0.244 – 0.415	**0f**	$[0,8)$	–	$[3,7)$
	0.415 – 0.585	**0**	$[2,8)$	**1**	$[0,6)$
	0.585 – 0.756	–	$[1,5)$	**1f**	$[0,8)$
	0.756 – 1.000	–	$[1,6)$	**110**	$[0,8)$
$[1,6)$	0.000 – 0.293	**001**	$[0,8)$	**f**	$[0,8)$
	0.293 – 0.500	**0f**	$[0,8)$	**f**	$[2,8)$
	0.500 – 0.707	**0**	$[2,8)$	**10**	$[0,8)$
	0.707 – 1.000	–	$[1,5)$	**101**	$[0,8)$
$[1,5)$	0.000 – 0.369	**001**	$[0,8)$	**f**	$[0,6)$
	0.369 – 0.631	**0f**	$[0,8)$	**ff**	$[0,8)$
	0.631 – 1.000	**0**	$[2,8)$	**100**	$[0,8)$
$[2,8)$	0.000 – 0.244	**010**	$[0,8)$	–	$[3,8)$
	0.244 – 0.415	**01**	$[0,8)$	**1**	$[0,8)$
	0.415 – 0.585	**f**	$[0,6)$	**1**	$[2,8)$
	0.585 – 0.756	**f**	$[0,8)$	**11**	$[0,8)$
	0.756 – 1.000	–	$[2,7)$	**111**	$[0,8)$
$[2,7)$	0.000 – 0.293	**010**	$[0,8)$	–	$[3,7)$
	0.293 – 0.500	**01**	$[0,8)$	**1**	$[0,6)$
	0.500 – 0.707	**f**	$[0,6)$	**1f**	$[0,8)$
	0.707 – 1.000	**f**	$[0,8)$	**110**	$[0,8)$
$[3,8)$	0.000 – 0.293	**011**	$[0,8)$	**1**	$[0,8)$
	0.293 – 0.500	**ff**	$[0,8)$	**1**	$[2,8)$
	0.500 – 0.707	**f**	$[2,8)$	**11**	$[0,8)$
	0.707 – 1.000	–	$[3,7)$	**111**	$[0,8)$
$[3,7)$	0.000 – 0.369	**011**	$[0,8)$	**1**	$[0,6)$
	0.369 – 0.631	**ff**	$[0,8)$	**1f**	$[0,8)$
	0.631 – 1.000	**f**	$[2,8)$	**110**	$[0,8)$

Table 2.1 (continued)

Restricting allowable subdivisions

Another way of simplifying an arithmetic coder is to allow only a subset of the possible interval subdivisions. Using integer arithmetic has the effect of making the symbol probabilities approximate, especially as the integer range is made smaller; limiting the number of subdivisions simply makes them even less precise. Since the main benefit of arithmetic coding is its ability to code efficiently when probabilities are close to 1, we usually want to allow at least some pairs of unequal probabilities.

Example 9: If we know that one symbol occurs considerably more often than the other, we can eliminate the transitions in Example 8 for approximately equal probabilities. This makes it unnecessary for the coder to decide which transition pair to use in the $[0, 4)$ state, and gives a very simple reduced-precision arithmetic coder.

	LPS input		MPS input	
State	Output	Next state	Output	Next state
$[0, 4)$	**00**	$[0, 4)$	–	$[1, 4)$
$[1, 4)$	**01**	$[0, 4)$	**1**	$[0, 4)$

This simple code is quite useful, providing almost a 50 percent improvement on the unary code for representing non-negative integers. To encode n in unary, we output n **1**s and a **0**. Using the code just derived, we re-encode the unary coding, treating **1** as the MPS. The resulting code consists of $\lfloor n/2 \rfloor$ **1**s, followed by **00** if n is even and **01** if n is odd. This code is in fact equivalent to the the Rice code with $k = 1$ and the Golomb code with $m = 2$, as shown in Section 2.3.2. We can do even better with slightly more complex codes, as we shall see in examples that follow. □

Maximally unbalanced subdivisions

We now introduce the *maximally unbalanced subdivision* and show how it can be used to obtain excellent compression when Prob{MPS} ≈ 1. Suppose the current interval is $[L, H)$. If Prob{MPS} is very high we can subdivide the interval at $L + 1$ or $H - 1$, indicating Prob{LPS} $= 1/(H - L)$ and Prob{MPS} $= 1 - 1/(H - L)$. Since the length of the current interval $H - L$ is always more than $N/4$, such a subdivision always indicates a Prob{MPS} of more than $1 - 4/N$. By choosing a large value of N and always including the maximally unbalanced subdivision in our coder, we ensure that very likely symbols can always be given an appropriately high probability.

Example 10: Let $N = 8$ and let the MPS always be *1*. We obtain the following four-state code if we allow only the maximally unbalanced subdivision in each state.

State	*0* (LPS) input		*1* (MPS) input	
	Output	Next state	Output	Next state
$[0,8)$	**000**	$[0,4)$	–	$[1,8)$
$[1,8)$	**001**	$[0,4)$	–	$[2,8)$
$[2,8)$	**010**	$[0,4)$	–	$[3,8)$
$[3,8)$	**011**	$[0,4)$	**1**	$[0,8)$

We can use this code to re-encode unary-coded non-negative integers with $\lfloor n/4 \rfloor + 3$ bits. In effect, we represent n in the form $4a + b$; we encode a in unary, then use two bits to encode b in binary. (This is the Rice code with $k = 2$ and the Golomb code with $m = 4$.) □

Changing codes

Whenever the current interval coincides with the full interval, we can switch to a different code.

Example 11: We can derive the Elias code for the positive integers [13] by using the maximally unbalanced subdivision technique of Example 10 and by doubling the full integer range whenever we see enough **1**s to output a bit and expand the current interval so that it coincides with the full range. This coder has an infinite number of states; no state is visited more than once. We use the notation $[L,H)/M$ to indicate the subinterval $[L,H)$ selected from the range $[0,M)$.

State	*0* (LPS) input		*1* (MPS) input	
	Output	Next state	Output	Next state
$[0,2)/2$	**0**	STOP	**1**	$[0,4)/4$
$[0,4)/4$	**00**	STOP	–	$[1,4)/4$
$[1,4)/4$	**01**	STOP	**1**	$[0,8)/8$
$[0,8)/8$	**000**	STOP	–	$[1,8)/8$
$[1,8)/8$	**001**	STOP	–	$[2,8)/8$
⋮	⋮	⋮	⋮	⋮

This code corresponds to encoding positive integers as follows:

n	Code
1	**0**
2	**100**
3	**101**
4	**11000**
5	**11001**
⋮	⋮

In effect we represent n in the form $2^a + b$; we encode a in unary, then use a bits to encode b in binary. This is essentially the Elias code; it requires $\lfloor 2 \log_2 n \rfloor + 1$ bits to encode n. □

Design of a class of reduced precision coders

We now present a very flexible yet simple coder design incorporating most of the features just discussed. We choose N to be any power of 2, but at least 4. All states in the coder are of the form $[k, N)$, so the number of states is only $N/2$. (Intervals with $k \geq N/2$ will produce output, and the interval will be expanded.) In every state $[k, N)$ we include the maximally unbalanced subdivision (at $k+1$), which corresponds to values of Prob{MPS} between $(N-2)/N$ and $(N-1)/N$. We include a nearly balanced subdivision so that we will not lose efficiency when Prob{MPS} $\approx 1/2$. In addition, we locate other subdivision points such that the subinterval expansion that follows each input symbol leaves the coder in a state of the form $[k, N)$, and we choose one or more of them to correspond to intermediate values of Prob{MPS}. For simplicity we denote state $[k, N)$ by k.

We always allow the interval $[k, N)$ to be divided at $k+1$; if the LPS occurs we output the $\log_2 N$ bits of k and move to state 0, while if the MPS occurs we simply move to state $k+1$, then if the new state is $N/2$ we output a **1** and move to state 0. The other permitted subdivisions are given in the following table; it may not be necessary to include all of them in the coder.

Range of states k	Subdivision LPS	Subdivision MPS	LPS input Output	LPS input Next State	MPS input Output	MPS input Next State
$[0, \frac{N}{2})$	$[k, \frac{N}{2})$	$[\frac{N}{2}, N)$	**0**	$2k$	**1**	0
$[0, \frac{N}{4})$	$[k, \frac{N}{4})$	$[\frac{N}{4}, N)$	**00**	$4k$	–	$\frac{N}{4}$
$[\frac{N}{8}, \frac{3N}{8})$	$[k, \frac{3N}{8})$	$[\frac{3N}{8}, N)$	**0f**	$4k - \frac{N}{2}$	–	$\frac{3N}{8}$
$[\frac{3N}{8}, \frac{N}{2})$	$[k, \frac{5N}{8})$	$[\frac{5N}{8}, N)$	**ff**	$4k - \frac{3N}{2}$	**1**	$\frac{N}{4}$
$[\frac{7N}{16}, \frac{N}{2})$	$[k, \frac{9N}{16})$	$[\frac{9N}{16}, N)$	**fff**	$8k - \frac{7N}{2}$	**1**	$\frac{N}{8}$
$[\frac{N}{4}, \frac{N}{2})$	$[\frac{3N}{4}, N)$	$[k, \frac{3N}{4})$	**11**	0	**f**	$2k - \frac{N}{2}$

For example, the fourth line indicates that for all states k for which $3N/8 \leq k < N/2$, we may subdivide the interval at $5N/8$. If the LPS occurs, we perform the follow-on procedure twice, which leaves us with the interval $[4k - 3N/2, N)$; otherwise we output a **1** and expand the interval to $[N/4, N)$.

A coder constructed using this procedure will have a small number of states, but in every state it will allow us to use estimates of Prob{MPS} near 1, near $1/2$, and in between. Thus we can choose a large N so that highly probable events require negligible code length, while keeping the number of states small enough to allow table lookups rather than arithmetic. In practice, however, use of this design procedure is

usually not necessary; the coders that result from systematically allowing *all* states and transitions are compact and give good compression efficiency.

2.2.3 ϵ-partitions and ρ-partitions

We have shown that is is possible to design a binary arithmetic coder that admits only a small number of possible probabilities. In this section we give a theoretical basis for selecting the probabilities. Often practical considerations limit our choices, but we can show that it is reasonable to expect that choosing only a few probabilities will give close to optimal compression. In Section 2.2.5 we give a method for computing transition cutoff probabilities for a complete coder.

For a binary alphabet, we can compute $E(p, q)$, the extra code length resulting from using estimates q and $1 - q$ for actual probabilities p and $1 - p$, respectively. We let $d_i = q_i - p_i$ and expand asymptotically in d to obtain

$$\begin{aligned} E &= \sum_{i=1}^{n} -p_i \log_2 q_i - \sum_{i=1}^{n} -p_i \log_2 p_i \\ &= \sum_{i=1}^{n} \left(\frac{1}{2 \ln 2} \frac{d_i^2}{p_i} + O\left(\frac{d_i^3}{p_i^2} \right) \right). \end{aligned}$$

For any desired maximum excess code length ϵ, we can partition the space of possible probabilities to guarantee that the use of approximate probabilities will never add more than ϵ to the code length of any event. We select partitioning probabilities $P_0, P_1, \ldots$ and estimated probabilities $Q_0, Q_1, \ldots$. Each probability Q_i is used to encode all events whose probability p is in the range $P_i < p \leq P_{i+1}$. We compute the partition, which we call an *ϵ-partition*, as follows:

1. Set $i := 0$ and $Q_0 := 1/2$.
2. Find the value of P_{i+1} (greater than Q_i) such that $E(P_{i+1}, Q_i) = \epsilon$. We will use Q_i as the estimated probability for all probabilities p such that $Q_i < p \leq P_{i+1}$.
3. Find the value of Q_{i+1} (greater than P_{i+1}) such that $E(P_{i+1}, Q_{i+1}) = \epsilon$. After we compute P_{i+2} in step 2 of the next iteration, we will use Q_{i+1} as the estimate for all probabilities p such that $P_{i+1} < p \leq P_{i+2}$.

We increment i and repeat steps 2 and 3 until P_{i+1} or Q_{i+1} reaches 1. The values for $p < 1/2$ are symmetrical with those for $p > 1/2$.

Example: We show the ϵ-partition for $\epsilon = 0.05$ bit per binary input symbol.

Range of actual probabilities	Probability to use
$[0.0000, 0.0130)$	0.0003
$[0.0130, 0.1427)$	0.0676
$[0.1427, 0.3691)$	0.2501
$[0.3691, 0.6309)$	0.5000
$[0.6309, 0.8579)$	0.7499
$[0.8579, 0.9870)$	0.9324
$[0.9870, 1.0000)$	0.9997

Thus by using only 7 probabilities we can guarantee that the excess code length does not exceed 0.05 bit for each binary decision coded. □

We might wish to limit the *relative* error so that the code length can never exceed the optimal by more than a multiplicative factor of $1 + \rho$. We can begin to compute these *ρ-partitions* using a procedure similar to that for ϵ-partitions, but unfortunately the process does not terminate, since ρ-partitions are not finite. As P approaches 1, the optimal average code length grows very small, so to obtain a small relative loss Q must be very close to P. Nevertheless, we can obtain a partial ρ-partition.

It is also possible to express the maximum allowable relative error in terms of percent log ratio compression loss, as described in Section 1.4.2. We call the resulting partitions λ-partitions; they are essentially the same as ρ-partitions.

Example: We show part of the ρ-partition for $\rho = 0.05$; the maximum relative error is 5 percent. (This is the same as the λ-partition for $\lambda = \log_e 1.05 \approx 4.88\frac{0}{0}$.)

Range of actual probabilities	Probability to use
⋮	⋮
$[0.0033, 0.0154)$	0.0069
$[0.0154, 0.0573)$	0.0291
$[0.0573, 0.1670)$	0.0982
$[0.1670, 0.3722)$	0.2555
$[0.3722, 0.6278)$	0.5000
$[0.6278, 0.8330)$	0.7445
$[0.8330, 0.9427)$	0.9018
$[0.9427, 0.9846)$	0.9709
$[0.9846, 0.9967)$	0.9931
⋮	⋮

□

In practice we will use an approximation to an ϵ-partition or a ρ-partition for values of Prob{MPS} up to the maximum probability representable by our coder.

Index	Counts		Probability	Transitions	
	0	*1*	of *0*	after *0*	after *1*
$P = 0$	1	4	0.200	$P = 3$	$P = 0$
$P = 1$	1	3	0.250	$P = 4$	$P = 0$
$P = 2$	1	2	0.333	$P = 7$	$P = 1$
$P = 3$	2	4	0.333	$P = 5$	$P = 1$
$P = 4$	2	3	0.400	$P = 8$	$P = 3$
$P = 5$	3	4	0.429	$P = 9$	$P = 4$
$P = 6$	1	1	0.500	$P = 13$	$P = 2$
$P = 7$	2	2	0.500	$P = 11$	$P = 4$
$P = 8$	3	3	0.500	$P = 10$	$P = 5$
$P = 9$	4	4	0.500	$P = 10$	$P = 5$
$P = 10$	4	3	0.571	$P = 11$	$P = 9$
③ $P = 11$	3	2	0.600	$P = 12$	$P = 8$
$P = 12$	4	2	0.667	$P = 14$	$P = 10$
$P = 13$	2	1	0.667	$P = 14$	$P = 7$
① $P = 14$	3	1	0.750	$P = 15$	② $P = 11$
$P = 15$	4	1	0.800	$P = 15$	$P = 12$

Table 2.2: Probability arrays for quasi-arithmetic coding.

2.2.4 Implementation of binary quasi-arithmetic coding

We now give a full example showing a practical probability estimation mechanism and the systematic construction of a small but complete binary quasi-arithmetic coder. This coder makes efficient use of space, so that it is possible to use medium-sized values of N (up to 128), and makes heavy use of pointers to make use of the lookup tables efficient. In the example we use $N = 8$, i.e., the full interval is $[0, 8)$. The size of the tables used in practice remains manageable; their construction and use follow exactly the same principles. In practice, using $N = 32$ improves compression by about $3.5\frac{0}{0}$, and using $N = 128$ gives only another $0.2\frac{0}{0}$ improvement.

Probability estimation for quasi-arithmetic coding

We use a modification of the scaled-count technique to estimate the probabilities used by the quasi-arithmetic coder. In effect we use small counts c_0 and c_1 for the *0* and *1* events at each decision point; i.e., we keep a count pair $c_0 : c_1$. Only a few bits are used for each count. When either count overflows, we scale both counts downward; the new scaled count pair is the closest to the (unavailable) new count pair, closeness being measured by average excess code length. The use of scaling does not hurt compression by much, as shown in Section 4.2.1. In [27] we present an alternative deterministic probability estimation method.

In the implementation we denote each possible pair of counts by an index number, and we precompute all the transitions to new count states, including those requiring scaling. In Table 2.2 we show the correspondence among counts, probabilities, and

Terminal states			Nonterminal states					More nonterminal states				
	W	H		L	R	F	Q		L	R	F	Q
㉔Q_{08}	W_8	H_8	Q_{04}	**0**	–	0	Q_{08}	Q_{48}	**1**	–	0	Q_{08}
Q_{07}	W_7	H_7	Q_{03}	**0**	–	0	Q_{06}	Q_{47}	**1**	–	0	Q_{08}
Q_{06}	W_6	H_6	Q_{02}	**0**	**0**	0	Q_{08}	Q_{46}	**1**	**0**	0	Q_{08}
Q_{05}	W_5	H_5	Q_{01}	**0**	**00**	0	Q_{08}	Q_{45}	**1**	**00**	0	Q_{08}
Q_{18}	W_7	H_8	Q_{14}	**0**	–	0	Q_{28}	Q_{58}	**1**	–	0	Q_{08}
④Q_{17}	$W_6$⑤	H_7⑩	Q_{13}	**0**	–	1	Q_{08}	⑭Q_{57}	**1**⑯	–⑳	1㉑	Q_{08}㉓
Q_{16}	W_5	H_6	Q_{12}	**0**	**01**	0	Q_{08}	Q_{56}	**1**	**01**	0	Q_{08}
⑨Q_{15}	W_4	H_5										
Q_{28}	W_6	H_8	Q_{26}	–	–	1	Q_{08}	Q_{68}	**1**	**1**	0	Q_{08}
Q_{27}	W_5	H_7	Q_{25}	–	–	1	Q_{06}	Q_{67}	**1**	**10**	0	Q_{08}
			Q_{24}	**0**	**1**	0	Q_{08}					
			Q_{23}	**0**	**10**	0	Q_{08}	Q_{78}	**1**	**11**	0	Q_{08}
Q_{38}	W_5	H_8	Q_{36}	–	–	1	Q_{28}					
Q_{37}	W_4	H_7	Q_{35}	–	–	2	Q_{08}					
			Q_{34}	**0**	**11**	0	Q_{08}					

Table 2.3: Complete quasi-arithmetic coding table for $N = 8$.

probability index numbers for a small example coder, as well as all the transitions. For example[1], ① index $P = 14$ corresponds to $c_0 : c_1 = 3 : 1$, and ② we find that $P = 11$ is the index of the new count state after a *1* event, where ③ index $P = 11$ corresponds to $c_0 : c_1 = 3 : 2$. In the example we allow counts to reach 4; in practice we allow somewhat larger counts (up to 10 or so), and allow some of the unbalanced counts to be larger than the balanced ones. It is quite feasible to store each probability index number in one byte. Note that the probability index array (containing counts c_0 and c_1 and the probability values) is used only during table construction; only the transition array remains during the actual coding.

Use of quasi-arithmetic coding

We use quasi-arithmetic coding to encode binary events, with probabilities (indicated by probability index numbers) supplied by the model. We use a pointer into a code table to indicate the state of the coder, corresponding to the current interval in a true reduced-precision arithmetic coder. Table 2.3 shows a complete code table for $N = 8$ (full interval $[0, 8)$); the initial state is Q_{08}. This table contains the same information as Table 2.1, but in a more compact form. We use left subintervals for *0* events and right subintervals for *1* events.

We illustrate the use of the coder. ④ Suppose we are in state $Q_{17} = [1, 7)$, the

[1]The small circled numbers in this section key the text to the tables.

	W_4	W_5	W_6 ⑥	W_7	W_8
$P = 0$	3	4	5	6	6
$P = 1$	3	4	4	5	6
$P = 2$	3	3	4	5	5
$P = 3$	3	3	4	5	5
$P = 4$	2	3	4	4	5
$P = 5$	2	3	3	4	5
$P = 6$	2	2	3	3	4
$P = 7$	2	2	3	3	4
$P = 8$	2	2	3	3	4
$P = 9$	2	2	3	3	4
$P = 10$	2	2	3	3	3
$P = 11$	2	2	2	3	3
$P = 12$	1	2	2	2	3
$P = 13$	1	2	2	2	3
⑦ $P = 14$	1	1	2 ⑧	2	2
$P = 15$	1	1	1	1	2

Table 2.4: Delta array for quasi-arithmetic coding. The five vectors, one for each possible terminal state width, are indexed by probability index numbers to find Δ, the size of the right subinterval.

	H_5	H_6	H_7 ⑪	H_8
$\Delta = 1$	Q_{45}	Q_{56}	Q_{67}	Q_{78}
⑫ $\Delta = 2$	Q_{35}	Q_{46}	Q_{57} ⑬	Q_{68}
$\Delta = 3$	Q_{25}	Q_{36}	Q_{47}	Q_{58}
$\Delta = 4$	Q_{15}	Q_{26}	Q_{37}	Q_{48}
$\Delta = 5$		Q_{16}	Q_{27}	Q_{38}
$\Delta = 6$			Q_{17}	Q_{28}
$\Delta = 7$				Q_{18}

Table 2.5: Right branch array for quasi-arithmetic coding. The four vectors, one for each possible value of the high end of a terminal state, are indexed by Δ, the size of the right subinterval, to find a pointer to the next state.

$c_0 : c_1$ counts are $3:1$, indicated by index $P = 14$ ①, and the next event is *1*. ⑤ The W entry for state Q_{17} is W_6 since the width of the interval is 6; ⑥ W_6 is a pointer to one of the five vectors in the delta array (Table 2.4), the interface between the probability estimator and the coder. (In Section 2.2.5 we show how to find the cutoff probabilities between successive values of Δ, which can then be used with Table 2.2 to compute the delta array.) ⑦ We use $P = 14$ to index into the W_6 vector, and ⑧ find $\Delta = 2$; this is the size of the right subinterval of $[1, 7)$. ⑨ If the event were *0*, we would move down

	Output buffer		Bits left	Pending count
⑮	10**010110**	00000000	2	2
⑰	10**010110**	**100**00000	2	3
⑱	**01011010**	**0**0000000	8	1
⑲	1011010**0**	00000000	7	0
㉒	1011010**0**	00000000	7	1

Table 2.6: Quasi-arithmetic coding output example. Useful bits not yet output are shown in bold face type.

$\Delta = 2$ rows in the code table to Q_{15}, a "terminal state" (one for which no output or interval expansion is possible). But in fact the event is *1*, so ⑩ we use the H entry for state Q_{17}, namely H_7, which indicates that 7 is the high end of the interval $[1, 7)$. ⑪ H_7 is a pointer to one of the four vectors in the right-branch array (Table 2.5). ⑫ We use $\Delta = 2$ as an index into the H_7 vector, and ⑬ find a pointer to the next state, Q_{57}. ⑭ We go to state Q_{57} in the code table. It is a nonterminal state, so we perform the output indicated by the L, R, F, and Q entries, which were computed by applying the Witten-Neal-Cleary algorithm to the interval $[5, 7)$. This is the same information found in Table 2.1, in the $[1, 7)$ state, on the line for probabilities between 0.585 and 0.756, with a *1* input.

To do the output, we use a two-byte buffer and two counts (see Table 2.6). We insert new bits into the upper end of the low-order byte, then shift the useful bits into the high-order byte; when the high-order byte is full of useful bits, we output them. Continuing the example, ⑮ suppose that the output buffer contains 6 useful bits, so there is room for 2 more, and that the *pending* count is 2, meaning that the next output bit will be followed by two opposite bits, as in the follow-on mechanism described in Section 2.1.2. ⑯ The leading output bit L is **1**, so ⑰ we put **10000000** into the low byte of the buffer (if L had been **0**, we would have put **01111111** into the low byte of the buffer). We then shift left by three bits altogether, one for the leading bit and two for the *follow* bits. Since there was only room for two bits, ⑱ we shift left by two bits, output **01011010**, indicate that space remains for 8 bits, and ⑲ shift left by one more bit. ⑳ The R entry shows that there are no remaining bits. (If there had been, we would have put them into the upper end of the low-order byte of the buffer, then shifted them into the high-order byte.) ㉑ The F entry shows that the *pending* count should be increased by 1. The resulting buffer state is shown at ㉒. Finally, ㉓ the Q entry points to the next coder state, Q_{08}, indicated at ㉔.

Decoding uses essentially the same tables, and in fact is easier than encoding.

2.2.5 Analysis of binary quasi-arithmetic coding

We now show that using binary quasi-arithmetic coding causes an insignificant increase in the code length compared with pure arithmetic coding. We mathematically analyze

several cases.

In this analysis we exclude very large and very small probabilities, namely those that are less than $1/W$ or greater than $(W-1)/W$, where W is the width of the current interval. For these probabilities the relative excess code length becomes infinite.

First we assume that we know the probability p of the left (0) branch of each event, and we show both how to minimize the average excess code length and how small the excess is. In arithmetic coding we divide the current interval (of width W) into subintervals of length L and R; this gives an effective coding probability $q = L/W$ since the resulting code length is $-\log_2 q$ for the left branch and $-\log_2(1-q)$ for the right. When we encode a sequence of binary events with probabilities p and $1-p$ using effective coding probabilities q and $1-q$, the average code length $L(p,q)$ is given by

$$L(p,q) = -p\log_2 q - (1-p)\log_2(1-q).$$

If we use exact arithmetic coding, we can subdivide the interval into lengths pW and $(1-p)W$, thus making $q = p$ and giving an average code length equal to the entropy, $H(p) = -p\log_2 p - (1-p)\log_2(1-p)$; this is optimal.

Consider two probabilities p_1 and p_2 that are adjacent based on the subdivision of an interval of width W; in other words, $p_1 = (W-\Delta_1)/W$, $p_2 = (W-\Delta_2)/W$, and $\Delta_2 = \Delta_1 - 1$. For any probability p between p_1 and p_2, either p_1 or p_2 should be chosen, whichever gives a shorter average code length. There is a cutoff probability p^* for which p_1 and p_2 give the same average code length. We can compute p^* by solving the equation $L(p^*,p_1) = L(p^*,p_2)$, giving

$$\begin{aligned} p^* &= \frac{1}{1+\dfrac{\log\dfrac{p_2}{p_1}}{\log\dfrac{1-p_1}{1-p_2}}} \\ &= \frac{\log\dfrac{\Delta_1}{\Delta_2}}{\log\dfrac{W-\Delta_2}{W-\Delta_1}\dfrac{\Delta_1}{\Delta_2}}. \end{aligned} \tag{2.1}$$

We can construct the delta table by computing cutoff probabilities for every pair of adjacent coding probabilities and every possible interval size and then applying them to the count state probabilities. As an example, we compute the value of Δ, the size of the right subinterval, to be used for $c_0 : c_1 = 3 : 1$ (i.e., for $p = 3/4$) and $W = 6$. Clearly $\Delta = 1$ or 2, so $p_1 = 4/6$ ($\Delta_1 = 2$) and $p_2 = 5/6$ ($\Delta_2 = 1$). We compute $p^* = \log 2/\log(5/2) \approx 0.756$, and choose $\Delta = \Delta_1 = 2$ since $p_1 < p < p^* < p_2$. This is the entry at ⑧ in Table 2.4.

Probability p^* is the probability between p_1 and p_2 with the worst average quasi-arithmetic coding performance, both in excess bits per event and in excess bits relative to optimal compression. This can be shown by monotonicity arguments.

Theorem 4 *If we construct a quasi-arithmetic coder based on full interval* $[0, N)$*, and use correct probability estimates, the number of bits per input symbol by which the average code length obtained by the quasi-arithmetic coder exceeds that of an exact arithmetic coder is at most*

$$\frac{4}{\ln 2} \log_2 \frac{2}{e \ln 2} \frac{1}{N} + O\left(\frac{1}{N^2}\right) \approx \frac{0.497}{N} + O\left(\frac{1}{N^2}\right),$$

and the fraction by which the average code length obtained by the quasi-arithmetic coder exceeds that of an exact arithmetic coder is at most

$$\log_2 \frac{2}{e \ln 2} \frac{1}{\log_2 N} + O\left(\frac{1}{(\log N)^2}\right) \approx \frac{0.0861}{\log_2 N} + O\left(\frac{1}{(\log N)^2}\right).$$

Proof: For a quasi-arithmetic coder with full interval $[0, N)$, the shortest terminal state intervals have size $W = N/4 + 2$. The worst average error occurs for the smallest W and the most extreme probabilities, that is, for $W = N/4 + 2$, $p_1 = (W-2)/W$, and $p_2 = (W-1)/W$ (or symmetrically, $p_1 = 1/W$ and $p_2 = 2/W$). For these probabilities, we find the cutoff probability P^*. Then for the first part of the theorem we take the asymptotic expansion of $L(p_2, p^*) - H(p_2)$, and for the second part we take the asymptotic expansion of $(L(p_2, p^*) - H(p_2))/H(p_2)$. □

If we let $\overline{p} = (p_1 + p_2)/2$ and note that the maximum value of p in our analysis is $1 - 1/W$, we can expand Equation (2.2) asymptotically in W to express p^* as

$$p^* = \overline{p} + \frac{1}{6W^2} \frac{\overline{p} - 1/2}{\overline{p}(1-\overline{p})} + O(1/W).$$

The $O(\cdot)$ term is $1/W$ because of the effect of the maximum possible value of p. The constant in the $O(1/W)$ term is very small, less than 0.002. By computing

$$\widetilde{p^*} = \overline{p} + \frac{1}{6W^2} \frac{\overline{p} - 1/2}{\overline{p}(1-\overline{p})}$$

we can find the cutoff probabilities using rational arithmetic; the compression loss introduced by using $\widetilde{p^*}$ instead of p^* is completely negligible, never more than $0.06\frac{0}{0}$. In the example above with $p_1 = 2/3$ and $p_2 = 5/6$, we find that $p^* = \log 2 / \log(5/2) \approx 0.75647$ and $\widetilde{p^*} = 245/324 \approx 0.75617$.

Next we consider a more general case, in which we compare quasi-arithmetic coding with arithmetic coding for a single worst-case event. We assume that both coders use the same estimated probability, but that the estimate may be arbitrarily bad. In this case we find the cutoff probability for $1/2 \le p_1 < p_2$ by equating the excess code length from using probability p_1 for the more probable event and the excess from using probability p_2 for the less probable event, that is, by solving the equation $-\log_2 p_1 + \log_2 p^* = -\log_2(1-p_2) + \log_2(1-p^*)$; this yields

$$p^* = \frac{1}{1 + \dfrac{1-p_2}{p_1}} = \frac{W - \Delta_1}{W - 1}.$$

The excess code length in this case is just $\log_2(W/(W-1)) \sim 1/W \ln 2$ regardless of the value of Δ_1. As in the previous theorem, the smallest value of W is $N/4+2$. By substituting this value of W into the expression for the excess code length and expanding asymptotically, we prove the following theorem which bounds the worst-case excess code length.

Theorem 5 *If we construct a quasi-arithmetic coder based on full interval* $[0, N)$*, and use arbitrary probability estimates, the number of bits per input symbol by which the code length obtained by the quasi-arithmetic coder exceeds that of an exact arithmetic coder in the worst case is at most*

$$\log_2 \frac{N+8}{N+4} \sim \frac{4}{N \ln 2} \approx \frac{5.771}{N}.$$

2.2.6 Quasi-arithmetic coding for a multi-symbol alphabet

Constructing an optimal quasi-arithmetic code for a multi-symbol alphabet is a non-trivial matter, but we can easily construct a suboptimal code by the following three-step procedure.

1. We design a multi-state code table for a two-symbol alphabet with various possible symbol probabilities.

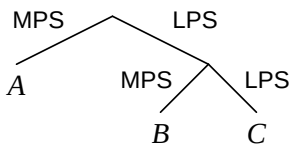
2. We decompose the multi-symbol alphabet into a binary tree with the alphabet symbols at the leaves, such that the product of the edge probabilities from the root to each leaf approximately equals the corresponding symbol probability.

3. We follow all paths through the tree of Step 2, using the table from Step 1 to compute an output codeword and next state for each alphabet symbol from each possible starting state.

Example 12: We construct a two-state code for the three-symbol alphabet $\{A, B, C\}$, with $p_A = 0.7$, $p_B = 0.2$, and $p_C = 0.1$. First we construct a two-state binary code. In this example we use a very simple binary code, in which we merely distinguish the input symbols as more or less probable; a more complete two-state code would also allow the two probabilities to be approximately equal. We indicate the more probable symbol by MPS and the less probable symbol by LPS; the probability of MPS can be taken to be about 0.71.

From state	MPS input		LPS input	
	Output	Next state	Output	Next state
E_0	-	E_1	**00**	E_0
E_1	**1**	E_0	**01**	E_0

Next we construct a binary tree with each branch labeled either MPS or LPS, the probability of MPS being 0.71.



The effective probabilities of the leaves are $p_A^* \approx 0.710$, $p_B^* \approx 0.206$, and $p_c^* \approx 0.084$. Next we use the tree and the table to derive the following code.

From state	Input A		Input B		Input C	
	Output	Next state	Output	Next state	Output	Next state
E_0	–	E_1	**00**	E_1	**0000**	E_0
E_1	**1**	E_0	**01**	E_1	**0100**	E_0

Even though it does not have the prefix property, this code is uniquely decodable with bounded delay. For example, from state E_0, the code for B (**00**) is a prefix of the code for C (**0000**); but a B input leads to state E_1, and the first two bits output from state E_1 are never **00**; hence a B will not be decoded as a C. The coding delay is at most one symbol, and the following four-state table can perform the decoding.

From state	Input **0**		Input **1**	
	Output	Next state	Output	Next state
D_0	–	D_1	AA	D_0
D_1	–	D_2	A	D_2
D_2	–	D_3	BA	D_0
D_3	C	D_0	B	D_2

The asymptotic average code length for this code is 1.171 bits per symbol. The entropy of the source is 1.157 bits per symbol, so the compression loss is only $1.2\frac{0}{0}$. The Huffman code for this source has an average code length of 1.3 bits per symbol, giving a compression loss of $10.5\frac{0}{0}$, over 8 times as large. In this example we used a two-state code; using more states usually gives even more efficiency. □

Note that the two-symbol code is used only for constructing the multi-symbol code. The multi-symbol code tables and coding algorithm are similar to the Huffman tables and algorithm, but with added state information.

2.3 Prefix codes

The simplest statistical coders have a direct correspondence between input events and code words. To ensure decodability, no codeword may be a prefix of any other; codes

Symbol	Probability	Codeword
a_1	$^{13}/_{41}$	**00**
a_2	$^{11}/_{41}$	**01**
a_3	$^{7}/_{41}$	**11**
a_4	$^{5}/_{41}$	**100**
a_5	$^{3}/_{41}$	**1010**
a_6	$^{2}/_{41}$	**1011**

Table 2.7: Example of a Huffman code for a small alphabet. The symbol weights were arbitrarily selected to be the first six prime numbers.

with this property are called *prefix codes*. A number of prefix codes are possible for any set of events.

The simplest prefix codes are fixed length binary codes, of which the ASCII code is an example. Fixed length binary codes have little independent interest, but they are used as components of other codes. For instance, they are sometimes used to encode symbols occurring for the first time in more complicated methods, and they are used as one of the two parts of each codeword in Rice coding. If the number of symbols to be encoded is a power of two, all symbols are assigned codewords of the exactly same length. Otherwise, it is possible to construct an "adjusted binary code," in which some symbols are assigned a codeword one bit shorter. Specifically, if the alphabet contains $n = 2^k + b$ symbols $0, 1, 2, \ldots, n-1$, then the codewords for symbols $0, 1, 2, \ldots, n-2b-1$ are just the k-bit binary representations of the symbols, while the codewords for the remaining symbols are $(k+1)$-bit binary numbers, symbol j in the range $n-2b \leq j < n$ being represented by $j + n - 2b$. Adjusted binary codes are used as one of the two parts of each codeword in Golomb coding, and we use them in the image compression algorithm of Section 3.5.

2.3.1 Huffman codes

If we are given the probabilities of n events, we can use Huffman's procedure [33] to construct an optimal prefix coding tree for those events. We briefly describe the procedure. We begin with a forest of n weighted trees, each consisting of a leaf node corresponding to an event; the weights of the leaves are proportional to the probabilities of the corresponding events. We repeatedly select the two trees with smallest weights (ties being broken in any systematic way), and make them the left and right subtrees of a new tree, whose weight is the sum of the weights of the two subtrees. The procedure is complete when exactly one tree remains. The codeword for any event is obtained by traversing the tree from the root to the leaf corresponding to the event, outputting **0** at every left branch and **1** at every right branch. Table 2.7 illustrates a Huffman code for a six-symbol alphabet.

For a fixed set of probabilities, this procedure is straightforward. In a dynamic setting, however, changing the codes can be very time-consuming since small changes

Golomb Rice	$m=1$ $k=0$	$m=2$ $k=1$	$m=3$	$m=4$ $k=2$	$m=5$	$m=6$	$m=7$	$m=8$ $k=3$	$m=9$
$n=0$	**0**	**00**	**00**	**000**	**000**	**000**	**000**	**0000**	**0000**
1	**10**	**01**	**010**	**001**	**001**	**001**	**0010**	**0001**	**0001**
2	**110**	**100**	**011**	**010**	**010**	**0100**	**0011**	**0010**	**0010**
3	**1110**	**101**	**100**	**011**	**0110**	**0101**	**0100**	**0011**	**0011**
4	**11110**	**1100**	**1010**	**1000**	**0111**	**0110**	**0101**	**0100**	**0100**
5	$\mathbf{1^5 0}$	**1101**	**1011**	**1001**	**1000**	**0111**	**0110**	**0101**	**0101**
6	$\mathbf{1^6 0}$	**11100**	**1100**	**1010**	**1001**	**1000**	**0111**	**0110**	**0110**
7	$\mathbf{1^7 0}$	**11101**	**11010**	**1011**	**1010**	**1001**	**1000**	**0111**	**01110**
8	$\mathbf{1^8 0}$	**111100**	**11011**	**11000**	**10110**	**10100**	**10010**	**10000**	**01111**
9	$\mathbf{1^9 0}$	**111101**	**11100**	**11001**	**10111**	**10101**	**10011**	**10001**	**10000**
10	$\mathbf{1^{10} 0}$	$\mathbf{1^5 00}$	**111010**	**11010**	**11000**	**10110**	**10100**	**10010**	**10001**
11	$\mathbf{1^{11} 0}$	$\mathbf{1^5 01}$	**111011**	**11011**	**11001**	**10111**	**10101**	**10011**	**10010**
12	$\mathbf{1^{12} 0}$	$\mathbf{1^6 00}$	**111100**	**111000**	**11010**	**11000**	**10110**	**10100**	**10011**
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 2.8: Examples of Golomb and Rice codes. For typographical convenience, strings of $j \geq 5$ consecutive **1**s are denoted by $\mathbf{1}^j$. The codes can be extended to any non-negative values of n, and codes can be constructed for all $m > 0$ and all $k \geq 0$.

to the probabilities can cause global changes to the optimal tree [15,19,36,73,74]. Although Huffman codes are optimal among prefix codes, the expected code length of a Huffman code is the same as the entropy only if all the probabilities are powers of $1/2$.

2.3.2 Golomb and Rice codes

A simpler procedure, due to Golomb [22], gives prefix codes that are suboptimal but very easy to implement. Golomb codes are distinguished from each other by a single parameter, so dynamic updating is accomplished by estimating the value of the parameter. In Section 2.3.3 we show how this can be done effectively.

Golomb codes are used to encode symbols from a countable alphabet. The symbols are arranged in descending probability order, and non-negative integers are assigned to the symbols, beginning with 0 for the most probable event. To encode integer n using the Golomb code with parameter m, we first compute $\lfloor n/m \rfloor$ and output this integer using a unary code. Then we compute n mod m and output this value using a binary code, adjusted as described above so that we sometimes use $\lfloor \log_2 m \rfloor$ bits and sometimes $\lceil \log_2 m \rceil$ bits.

Rice coding, developed independently by Rice [53,54,55], is the same as Golomb coding except that only a subset of the parameter values may be used, namely the powers of 2. The Rice code with parameter k is exactly the same as the Golomb code with parameter $m = 2^k$. Hence to encode integer n using the Rice code with parameter k, we first compute $\lfloor n/2^k \rfloor$ and output this integer using a unary code.

Then we compute $n \bmod 2^k$ and output this value using a k bit binary code. The resulting codes give less compression efficiency than Golomb codes, but they are even easier to implement, especially in hardware [72], since we can compute $\lfloor n/2^k \rfloor$ by shifting and $n \bmod 2^k$ by masking out all but the k low order bits. Our parameter estimation method for Golomb codes applies to Rice codes too. Table 2.8 shows the beginning of Golomb and Rice codes with a number of different parameter values.

When the distribution of values to be encoded is exponential, it can be shown that a Golomb code with the correct choice of the parameter m produces an optimal prefix code for the distribution [20]. Rice coding has been used as the basis for a lossless hardware compressor [72]. Its compression effectiveness is analyzed in [78].

2.3.3 Selection of Golomb or Rice coding parameter

We now describe an on-line algorithm for estimating the coding parameter for Golomb and Rice codes, and prove a bound on its effectiveness. For simplicity, in this discussion we assume the use of Rice coding because it is easier to implement and faster than Golomb coding, and because it gives fewer parameter values to choose from; the extra code length introduced is very small.

A common method of obtaining and transmitting coding parameters in algorithms that use Rice coding [55] is to divide the data into blocks, and for each block to compute the code lengths that would be obtained using each of a set of reasonable parameter values and output the best parameter value as side information. The problems with this approach are the delay at the start of each block and the need to send the block parameter value as side information.

Rather than assuming that the parameter is constant over an arbitrary block of data, we make the more reasonable assumption that it is the same for data in similar contexts. It might be possible to adaptively estimate the mean and variance of the distribution of encoded values for each context and then to select the best-fitting exponential distribution, but we have found a more straightforward method.

For each context we maintain a cumulative total, for each reasonable Rice parameter value k, of the code length we would have if we had used parameter k to encode all values encountered so far in the context. Then we simply use the parameter with the smallest cumulative code length to encode the next value encountered in the context. Both in theory and in practice we quickly converge to good parameter estimates.

We now analyze the code lengths obtained by using this parameter estimation method. We assume that the source probabilities follow a geometric distribution, given by $p_n = p_0(1-p_0)^n$, where p_n is the probability that symbol n occurs. We also assume that p_0 has a fixed but unknown value. For each non-negative integer value k of the coding parameter, we define

$$\beta_k = (1-p_0)^{2^k}.$$

We define random variable l_k to be the code length for one symbol encoded using parameter value k, and we define random variable d_k to be $l_{k+1} - l_k$. When we are

using parameter k, the 2^k smallest values (0 to 2^k-1) require $k+1$ bits each to encode, the next 2^k values require $k+2$ bits, and so on, the number of bits increasing by 1 every 2^k values. From this we can find the values of $d_k(n)$, the code length difference for encoding symbol n. The value of $d_k(n)$ is 1 for $0 \le n < 2^k$; it is 0 for the next 2^{k+1} values of n; then it decreases by 1 every 2^{k+1} values. The aggregate probability of the symbols for which $d_k = 1$ can be shown to be $1-\beta_k$; for $d_k = \Delta \le 0$ the probability is $\beta_k^{-2\Delta+1}(1-\beta_k^2)$. Using these probabilities, we find that the expected value of d_k, denoted by $\overline{d_k}$, is given by

$$\overline{d_k} = \mathrm{E}(d_k) = 1 - \frac{\beta_k}{1-\beta_k^2},$$

and that the variance of d_k, denoted by σ_k^2, is given by

$$\sigma_k^2 = \mathrm{Var}(d_k) = \frac{\beta_k(1-\beta_k+\beta_k^2)}{(1-\beta_k^2)^2}.$$

For a given value of p_0, we find that $\mathrm{E}(l_k) = \mathrm{E}(l_{k+1})$ (that is, $\mathrm{E}(d_k) = 0$) when $\beta_k = (\sqrt{5}-1)/2 \approx 0.618$. (This value, $(\sqrt{5}-1)/2$, denoted by $\hat{\varphi}$, is $\varphi - 1 = 1/\varphi$, where φ is the golden ratio.) The values β_k determine the critical values of p_0 where the best parameter choice changes from k to $k+1$. It is remarkable that this analysis holds for all values of k.

For now we restrict our choice to two parameter values, k and $k+1$, presumed to be the two best choices for the current value of p_0. We assume that k is the best choice; similar reasoning applies if $k+1$ is optimal. At any point our algorithm may choose the worse of the parameter values, but we now show that to code a sequence of N symbols, we use on average at most $O(\sqrt{N})$ bits more than the average number used by the best parameter.

We note that since $\beta_{k+1} = \beta_k^2$ and $\beta_{k-1} = \sqrt{\beta_k}$, the value of β_k satisfies the relation $\hat{\varphi}^2 \le \beta_k \le \sqrt{\hat{\varphi}}$ when k is the optimal parameter value. Throughout this range, the standard deviation of d_k is bounded by a small constant $s \approx 2.117$, the value when $\beta_k = \sqrt{\hat{\varphi}}$.

As we proceed in the coding, the average value of the cumulative difference D_t between the code lengths for the two candidate parameters will increase linearly. At first, when its expected value is not large (up to $O(\sqrt{N})$), the actual value will sometimes be negative, causing our algorithm to select the wrong parameter; but in this range the total expected excess code length is only $O(\sqrt{N})$. Eventually the average difference becomes large enough that choosing the wrong parameter becomes very unlikely, the low probability canceling the potentially larger number of incorrect choices. The net effect is an excess of $O(\sqrt{N})$ bits. We formalize this reasoning in the following theorem.

Theorem 6 *For a stationary source whose probability distribution is given by $p_n = p_0(1-p_0)^n$, using our parameter selection algorithm in conjunction with Rice coding gives an expected code length that exceeds the expected code length given by the optimal parameter by at most $O(\sqrt{N})$ bits.*

Proof: We define D_t to be the cumulative sum of random variable d_k up to time t. We let $\delta = s\sqrt{N}$, where s is the maximum standard deviation of d_k in the range of interest, and we let T be the number of symbols needed until $\mathrm{E}(D_T) = T\overline{d_k}$ becomes δ. Up to symbol T, even if we always choose the wrong parameter the total expected excess code length is only $s\sqrt{N}$, by definition.

We divide the remaining symbols into intervals of length T. In the interval from rT to $(r+1)T$, we will choose the wrong parameter only when $D_t < 0$. Since the expected difference is $r\delta$ at the beginning of the interval and increases within the interval, and the standard deviation of D_t throughout the interval is less than $s\sqrt{(r+1)T} \leq s\sqrt{N} = \delta$, we see that D_t becomes negative only if its value is more than $r\delta/\delta = r$ standard deviations away from its mean. By Chebyshev's inequality, the probability of this happening is at most $1/r^2$. Hence the expected number of times we choose the wrong parameter in the interval is at most T/r^2. The average excess code length when we choose the wrong parameter is $\overline{d_k} = \delta/T$, so the expected excess for the interval is at most δ/r^2. We sum this excess over all intervals, and find the total to be

$$\sum_{r=1}^{N/T} \frac{\delta}{r^2} < \delta \sum_{r=1}^{\infty} \frac{1}{r^2} = \delta \frac{\pi^2}{6} = \frac{\pi^2}{6} s\sqrt{N}.$$

Including the excess for the first interval, we see that the total expected number of excess bits over all N input symbols is less than

$$s\sqrt{N}\left(1 + \frac{\pi^2}{6}\right) = O(\sqrt{N}).$$

□

The constant factor on the bound can be improved by better tail estimates. We omit the proof that parameters other than the two closest to optimal contribute a negligible amount to the excess code length.

Our theorem bounds the average excess code length used by our algorithm. We expect that it can be extended by an introduction of randomness to arbitrary individual sequences, showing that with high probability our method gives a code length for an arbitrary sequence within $O(\sqrt{N})$ bits of that produced by using the best value of the parameter on that sequence.

Applications of the parameter estimation technique

The parameter estimation technique described here makes it possible to use Golomb or Rice coding as an alternative to arithmetic or Huffman coding in almost any setting requiring adaptive modeling. All that is required is that the events to be encoded be arranged in approximately descending order of probability. In the image compression system described in Section 3.5, the ordering is the natural one due to the exponential distribution of prediction errors, while the fast PPM method described in Section 4.4 maintains event lists in frequency count order. In other applications it is possible to maintain approximate ordering by using heuristics such as move-to-front (move an event to the head of the list whenever it occurs) or transpose (move an event up one

place in the list whenever it occurs). In all cases the only extra storage required is that needed for the cumulative counts for the possible parameter values in each context.

Golomb and Rice coding give the fast, flexible modeling obtained with arithmetic coding without the time-consuming arithmetic. They give faster coding even than Huffman coding because of the especially simple prefix codes involved, and adaptive modeling is possible without the complicated data structure manipulations required in dynamic Huffman coding. The main drawback to Golomb and Rice coding is the limited class of distributions that can be modeled exactly, but even this is not a serious problem (unless one event's probability is close to 1) because the probabilities of the more probable events will be estimated fairly well.

The parameter estimation technique described here can be used to adaptively estimate any modeling parameter. Most compression techniques can be improved by allowing tweaking of modeling parameters, but end users can be confused by a multiplicity of choices. Our technique permits hiding the tweaking within the program, and gives rapid convergence to good parameter values, at least for sources that can be modeled by stationary distributions.

2.4 Parallel coding

The increasing availability of massively parallel computing architectures and the enormous volume of scientific data being produced make parallel data compression a natural subject of research. In [30] we present general-purpose algorithms for encoding and decoding using both Huffman and quasi-arithmetic coding, and apply them to the problem of lossless compression of high-resolution grayscale images.

Our system for lossless image compression described in Chapter 3 has four components: pixel sequence, prediction, error modeling, and coding. In Section 3.4 we describe MLP, a multi-level progressive method for lossless image compression. We shall see that the prediction and error modeling steps of MLP can be easily parallelized. In this section we show that the coding step can also be done in parallel.

A distinguishing characteristic of compression algorithms is the need to maintain decodability: the encoder may use only information that will be available to the decoder. Since we are considering the case where we wish to do both encoding and decoding in parallel, the location of bits in the encoded file must be computable by the decoding processors before they begin decoding.

The main requirement for our parallel coder is that the model for each event is known ahead of time. It is not necessary for each event to have the same model. In the MLP algorithm, the prediction error at each pixel is modeled as a random variate from an appropriate distribution with zero mean; the variance of the distribution is estimated by the error modeler. We then use a statistical coder to encode the error with respect to the distribution with the given variance. The distribution family is chosen so that the variance is the only parameter of the distribution, and the code length is not sensitive to small differences in the estimated variance, so we can select a small number of variances (fewer than 50) and precompute a distribution (and perhaps a code) for each of them.

Thus each event is encoded using a fixed known code, satisfying our requirement. In Section 2.4.1 we develop a parallel encoder and decoder for the simpler Huffman codes; in Section 2.4.2 we extend our coders to quasi-arithmetic coding. The parallel Huffman algorithm can be applied directly to any prefix codes, including Golomb and Rice codes.

We use the PRAM model of parallel computation; the number of processors is p. We allow concurrent reading of the parallel memory, and limited concurrent writing, to a single one-bit register, which we call the *completion register*. It is initialized to **0** at the beginning of each time step; during the time step any number of processors (or none at all) may write **1** to it. We assume that n events are to be coded; in MLP these would be the n pixels of a level.

The basic problem in parallel coding is to encode the values of n events in parallel, in such a way that they can be decoded in parallel. The values of the events become available simultaneously at the beginning of encoding. The values may come from different probability distributions, but all the probability distributions become available at the same time as the events' values. The main issue is in dealing with the differences in code lengths of different events. For simplicity we do not consider input data routing.

2.4.1 Parallel Huffman coding

We now develop the basic parallel coding algorithms using Huffman coding. In practice Huffman coding is often the best choice for statistical coding, since the resulting codes are usually close to optimal and they can be very fast if implemented by table lookups. Because Huffman codes are instantaneous, coding is simplified: the code bits for all events are disjoint and independent.

Codeword-length coding

One simple approach involves assigning one event to each processor and noting that each encoding processor can easily compute the code length of its event. By a single prefix operation each processor can compute the location of its code bits in the output stream; it can then write them directly. When all processors have finished, a new event is assigned to each processor. This method works if the decoding is to be sequential; in fact it produces the same coded output as a sequential encoder. Parallel decoding is difficult, since the decoding processors cannot easily determine the lengths or the starting locations of the output codewords, although in fact, De Agostino and Storer [11] show that this can be done by assigning processors to encoded bits.

Bit-transpose coding

We can achieve decodability by rearranging (transposing) the output bits. Instead of outputting all bits for the first event, then all bits for the second, and so on, we output the first bit for each event in the first time step, then the second bit for each event in the second time step, and so on. After each time step, we can determine whether any codes are complete by having all completing processors concurrently write a **1** to

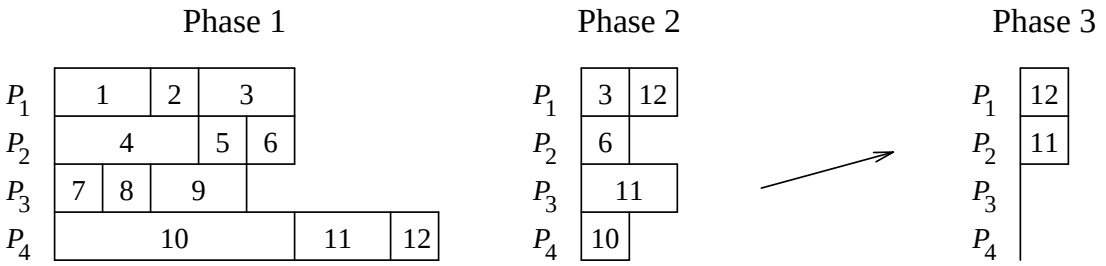


Figure 2.3: Parallel reallocation coding. There are four processors, $P_1, \ldots, P_4$ and 12 events, 1,...,12. Initially three events are assigned to each processor. The first phase ends when processor P_3 finishes event 9 after 4 time steps; each processor has output 4 bits by this time. Events 3 and 10 have been partially encoded; they remain assigned to processors P_1 and P_4; the untouched events are reassigned to balance the remaining load. The second phase ends one time unit later, when P_2 finishes event 6 and P_4 finishes event 10. The in-progress event (11) temporarily remains assigned to P_3; the untouched event (12) is "reassigned" to P_1; then since P_2 is inactive, P_3's processing is shifted to P_2. Events 11 and 12 are finished in the last time step.

the completion register. If any codes are complete, the corresponding processors no longer produce output; output from the remaining processors is shifted accordingly by a prefix operation. This code can be decoded in parallel: each decoding processor can determine when the code for its event is complete, and the same concurrent write and prefix operation can determine where the remaining processors are to obtain their next bits. When all processors have completed their output, a new event is assigned to each processor. If there is one processor for each event, this method makes good use of available processors. The difficulty lies in the time-consuming prefix operation that must be performed after every time step in which a processor finishes.

Reallocation coding

If the number of processors is much less than the number of events, we can reduce the number of prefix operations by allowing each processor to begin working on another event as soon as it has completed an event. First we distribute the n events equally among the p processors. For analysis we assume that the allocation is random.

During processing, we follow the bit-transpose protocol of Section 2.4.1: at each time step each processor writes a single bit to a location that will be known to both encoder and decoder. When a processor finishes outputting the code for one event, it goes on to the next event allocated to it, even though other processors are still working on earlier events. Code lookup is assumed to be fast: the codes are small enough that the full code can be stored in each processor's memory, or perhaps distributed among small groups of nearby processors. When one processor finishes all its allocated events, it indicates this fact by writing **1** to the completion register. When the completion register is **1**, the output process is interrupted. At this time the events allocated

to other processors fall into three categories: completed, in-progress, and untouched. In-progress events remain assigned to their current processors; untouched events are distributed evenly among the processors. Processing of the reallocated events then continues until the next time that a processor finishes all its allocated events. Toward the end of processing, the number of remaining events will become less than the number of available processors. At this point we deactivate processors, making the number of active processors equal to the number of remaining events. We then revert to the bit-transpose method of Section 2.4.1. The entire process is illustrated in Figure 2.3.

Analysis of parallel Huffman coding

We analyze the time required by the parallel algorithm. We assume that in one time unit each processor can output one bit, and that a prefix operation, as required for reallocation, takes $2\lceil \log_2 p_t \rceil$ time units, where p_t is the number of active processors at time t. We denote the number of bits in the longest single code word by L, and the average number of bits per event by H.

We can easily show that the time required for bit output is between $\lceil nH/p \rceil$ and $\lceil nH/p \rceil + L$. The processors are operating at full efficiency until there are fewer events than processors, using at most $\lceil nH/p \rceil$ time units. At that time no processor has more than L bits remaining, so no more than L additional time units are needed. It is clear that even if the processors can operate at full efficiency, they require at least $\lceil nH/p \rceil$ time units.

The more interesting analysis concerns the number of reallocations required. We define a *phase* to be the period between reallocations. *Early phases* are those which take place while the number of events is greater than the number of processors; *late phases* are those needed to code the final p or fewer events. We bound the number of reallocations needed in the following theorem.

Theorem 7 *When coding n events using p processors, the number of reallocations needed by the reallocation coding algorithm is at most $L \log_2(2n/p)$ in the worst case.*

Proof: We define a *superphase* to be the time needed to halve the number of remaining events. Consider the first superphase. The number of events assigned to each processor ranges from n/p in the first phase down to $n/2p$ in the last. At least one processor completes all its events in each phase; such a processor must output at least one bit per event, since all code words in a Huffman code have at least one bit. This processor (and hence all processors) thus output at least $n/2p$ bits in each phase, making a total of at least $n/2$ bits output in each phase. The total number of bits that must be output in the first superphase is at most $nL/2$, so the number of phases in the first superphase is at most L.

The same reasoning holds for all superphases. The number of superphases needed to reduce the number of remaining events from n to p is $\log_2(n/p)$, so the number of phases needed is just $L \log_2(n/p)$. Once p or fewer events remain, we fall back to bit-transpose coding, which may require L late phases, so the total number of phases needed is at most $L \log_2(2n/p)$. □

Parallel Huffman coding in practice

We have simulated parallel Huffman compression for a set of 21 Landsat Thematic Mapper 8-bit grayscale images; the images are described in Section 3.6. We simulate only the last level of coding (n = 131,072 pixels) for each image, using p = 4,096 processors. Theorem 7 shows that at most $6L$ reallocations are needed for these parameters.

For our test images, the number of early phases is at most 7, the average being 5.6. It usually happens, however, that the remaining code lengths take on many of the possible possible values, so the number of late phases is large. The average value of L in the simulations is 23.7, and the average number of late phases is 13.7.

To improve compression time, we must reduce the number of phases. Using the variability index technique described in Section 3.4.2, we can more evenly balance the output bits among the processors. The result is a reduction in the number of early phases for most of the images; a few stayed the same. The average falls to 4.6. The number of late phases is essentially unaffected.

By reducing L, the length of the longest code, we can reduce the number of late phases. We can do this by substituting a special IGNORE code for the codes of events longer than a certain threshold θ. These *long events* must then be transmitted separately. There are not very many of them and their code lengths are long, so we lose very little compression efficiency by sending them unencoded at the end of encoding; this is easy to do in parallel after a prefix operation to assign long events to processors. The IGNORE code itself can be of length θ. In simulations with $\theta = 10$ using the variability index technique, the average number of late phases falls to 9.1. The average loss in compression is only 196 bytes.

We can further reduce the number of early phases by performing local reallocations. Instead of using time $2\lceil \log_2 p_t \rceil$ to reallocate all untouched events when one processor completes its events, we can arrange local exchanges between neighboring processors, thus lengthening the time between full reallocations.

2.4.2 Parallel quasi-arithmetic coding

Huffman coding is nearly optimal in that it produces an average code length close to the entropy of the source model used for coding. Its suboptimality can be appreciable, however, whenever one input event has a probability near 1. In image compression, this happens when the variance of the model's distribution is small, in which case the zero-error event has high probability. For example, when the variance of a Laplace distribution is less than 1.04, the probability of a zero error is more than 0.5; when the variance is less than 0.26, the probability of a zero error is more than 0.75.

When Huffman coding is inadequate, we can turn to arithmetic coding. In particular, we use the multi-symbol-alphabet version of quasi-arithmetic coding. Since quasi-arithmetic coding can be viewed as a generalization of Huffman coding, the extension of the parallel Huffman algorithm is natural.

Parallel algorithm

We can apply the reallocation coding protocol of Section 2.4.1 directly to quasi-arithmetic coding. The only complication arises when the last event of a processor's allocation leaves the processor in a state other than the starting state. We deal with this by providing each processor with one additional "event," to be encoded after the last allocated event has been encoded. This event is designed to force the processor back to the starting state by the output of a small number of bits. (Often just one bit is required.)

Example 13: If we were using the three-symbol code in Example 12, and if the last symbol to be output by a processor (starting in state S_0) were B, we would output **00**; then we would have to force the processor back to state S_0. This can be done by outputting **1** or **01**; of course we would choose the shorter string. Without the extra bit, the decoder would not know whether the last symbol was B or C, since the codes for both of them begin with **00**. After reading the **1** and decoding B, the decoder would know that the processor had no more data to encode, so it would not attempt any further decoding. □

In effect, we have a number of arithmetically coded output streams; we have to solve the end-of-file problem for each of them. It is not difficult, merely a nuisance. (A simpler solution, reverting to a Huffman code for the last event of a processor's allocation, does not work: toward the end of processing, an event with a long code may *become* a processor's last event through reallocation, even though it was not the last when the processor began working on it. This happened to event 10 in the example of Figure 2.3.)

The algorithm for parallel quasi-arithmetic coding is as follows:

1. We assign each event to a processor, assigning the same number of events to each processor. The events may be assigned randomly or, even better, we may attempt to give each processor approximately the same number of bits to output, as discussed in Section 3.4.2.

2. Each processor proceeds sequentially through its assigned events, writing one bit to a preassigned location at each time step. If a processor completes all its assigned events, and it is in the starting state, it writes **1** to the completion register. If a completing processor is not in the starting state, it begins the finishing-up procedure described above. After finishing up, it writes **1** to the completion register (unless a reallocation has taken place, giving the processor more events to work on).

3. When the completion register becomes **1**, processing is interrupted for event reallocation. Events currently being processed remain with their current processor. The untouched events are divided among all the processors. Processors that have begun but not completed the finishing-up procedure must complete the procedure.

4. If a reallocation leaves any processors with no events and no finishing up to do, those processors are deactivated. We perform a prefix operation on the remaining events to determine the location of each processor's next output bit.

After reallocation, we return to Step 2, and repeat until no events remain.

Parallel coding summary

We have shown that efficient parallel lossless compression and decompression is feasible, using either Huffman coding or, more surprisingly, quasi-arithmetic coding; we have presented algorithms and analysis. Our algorithm uses randomization to limit the number of reallocations. For images, the variability index technique of Section 3.4.2 provides sufficient balancing of output bits to obviate randomization. The ideas of this section apply to any compression problem in which the model needed to encode each event is fixed ahead of time.

3

LOSSLESS IMAGE COMPRESSION

Lossless compression of images, in which the compression is completely reversible and the original data can be reconstructed exactly, is required in several fields, including space science, medical imaging, and nondestructive testing of materials. In each of these fields the features of interest in the images have characteristics similar to noise, so any lossy method that removes noise may also remove meaningful data. The image compression techniques discussed in this chapter are all lossless. It should be noted that the data being compressed is usually quantized analog data. The quantization process makes the data inherently lossy, but our methods introduce no additional loss.

One of our lossless techniques is *progressive*, meaning that we can use a small part of the losslessly encoded data to recover a lossy version of the original data; thus a user can browse the lossy data prior to obtaining the full lossless data.

In Section 3.1 we present a new four-component paradigm for lossless image compression; in Section 3.2 we describe a general method for coding prediction errors using arithmetic coding. In Sections 3.3 through 3.5 we give details of three lossless image compression systems; two of them are designed for high compression efficiency, while the third is designed for speed. In Section 3.6 we detail experiments showing that our methods improve on other methods in the literature.

3.1 A paradigm for lossless image compression

Our approach to lossless image compression is to develop practical, efficient compression algorithms with a sound theoretical basis and to prove rigorous guarantees of compression performance. We concentrate equally on encoding and decoding; in fact, most parts of our programs are common to both processes.

Lossless compression of high resolution images presents a unique set of challenges. Although considerable research has been done on lossless compression of text, we find that the usual techniques for text compression are not effective for lossless image compression. All good methods for text compression found to date [7,9,45,79,80] involve

some form of moderately high-order exact string matching. Images, however, are two-dimensional, so the contexts are more complicated than for one-dimensional strings. In addition, images are typically quantized analog data, and the exact matches needed for high-order modeling are only rarely found in the data. Lempel and Ziv [40] have presented a method for two-dimensional dictionary based coding that uses a space-filling curve for its pixel sequence, and Sheinwald, Lempel, and Ziv [68] give another dictionary method that covers the image with repeated rectangles of various sizes. These methods can be proven asymptotically optimal for images generated by finite state sources, but their practical usefulness appears to be limited.

Our work on lossless data compression revolves around a paradigm we have developed [26] that separates the process into the following four components:

- Pixel sequence. Careful selection of the pixel processing order can permit progressive compression, parallel computation, and improved compression efficiency.
- Prediction. Accurate prediction of pixel values based on the values of previously coded pixels allows us to encode only the prediction errors, leading to a great saving of code length. Predictive coding is the basis for almost all lossless image compression techniques.
- Error modeling. Precise characterization of the errors inherent in the prediction process gives us a model that can be used effectively by a statistical coder.
- Statistical coding. Arithmetic coding allows us to obtain optimal average code length; Huffman coding and quasi-arithmetic coding give almost optimal compression with considerably faster running times. Golomb and Rice coding run even faster while sacrificing only small amounts of compression efficiency.

In our methods, we obtain lossless image compression by iterating the four-step coding process. We choose a pixel p (with intensity A_p); we make a prediction P_p of the pixel's intensity and compute the prediction error $\Delta_p = A_p - P_p$; then we model the prediction error by estimating the parameters of the distribution from which it came; finally we encode Δ_p using the estimated distribution. We can then use the actual intensity for predicting other pixels, since the decoder can reconstruct the actual value from the prediction and the coded difference.

In our two high-compression methods the prediction is a linear combination of the intensities of pixels whose intensities are already known. Our methods differ in the sequence in which pixels are coded, the specific prediction function, and the method of estimating variances. In Section 3.3, we describe our *prediction by partial precision matching* method (PPPM), a context-based method with approximate matches. PPPM encodes an image in raster-scan order, predicting with a two-point average; variance estimation of each pixel is based on a context of nearby pixels, allowing for inexact matches in a novel way. In Section 3.4 we describe our *multi-level progressive* method (MLP). It uses a many-point interpolation function for prediction; a number of different methods are available for variance estimation. PPPM gives compression similar to that

of MLP, but it is not progressive. In Section 3.5 we take a different approach, combining the prediction and error modeling steps in a fast raster-scan method, named FELICS, for *fast, efficient lossless image compression system.*

Most of the output code length comes from the error encoding. We form a probabilistic model for the errors and use arithmetic coding to encode the errors efficiently with respect to the model. For good compression we want the model to assign as high a probability as possible to the prediction error that actually occurs at each pixel. To obtain this, first we need a good predictor, one whose errors have a small variance so that only a few error values are likely and the probability of each is high. In addition, we need a good model for the errors, so that the arithmetic coder is using realistic probabilities. Prediction errors for images have traditionally been considered to be distributed according to a Laplace distribution with zero mean, although we show in Section 3.4.2 that this assumption is not always valid. To obtain a realistic model based on Laplace distributions we need a good estimate of the variance of the errors. Too high a variance allocates too much probability (and code space) to large, unlikely errors, while too low an estimate assigns too little probability to errors that actually occur. For each pixel we estimate the variance and choose one of a number of precomputed Laplace distributions.

To keep the code length short, we prefer that the error modeling be done implicitly, with no need to transmit side information about the models used. Of course, any computations used in modeling the error for encoding must be computable by the decoder as well.

3.2 Error modeling and coding

It has long been accepted that, for most images, prediction errors can be closely approximated by a Laplace (or symmetric exponential) distribution [24,34,47,48]. In particular, the distribution of prediction errors is sharply peaked at zero, which is characteristic of the Laplace distribution but not of the normal distribution (see Figure 3.1).

For our prediction methods, we similarly find that prediction errors closely follow the Laplace distribution, at least on an image-wide basis. The probability density function of the Laplace distribution with mean μ and variance σ^2 is given by

$$f_{\mu,\sigma^2}(x) = \frac{1}{\sqrt{2\sigma^2}} \exp\left(-\sqrt{\frac{2}{\sigma^2}}\,|x-\mu|\right).$$

We assume that $\mu = 0$, and we define the discrete probability mass function by

$$p_{\sigma^2}(k) = \int_{k-1/2}^{k+1/2} \frac{1}{\sqrt{2\sigma^2}} \exp\left(-\sqrt{\frac{2}{\sigma^2}}\,|x|\right) dx. \tag{3.1}$$

We can use the usual entropy formula to compute the average code length H (in bits per pixel) required to optimally encode a sequence of random variates taken from a

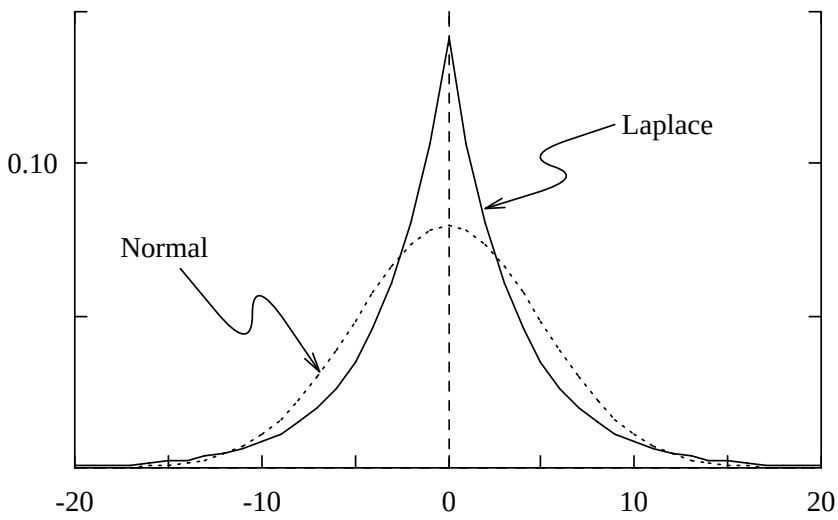


Figure 3.1: Comparison of Laplace and normal distributions, $\sigma = 5$.

Laplace distribution with mean 0 and variance σ^2:

$$H = \sum_k -p_{\sigma^2}(k) \log_2 p_{\sigma^2}(k). \tag{3.2}$$

If we use probabilities computed from Equation (3.1) to encode random variates from a Laplace distribution with variance $\sigma_1^2 \neq \sigma^2$, the average code length will be longer than the ideal code length, but if the variances are close, the extra code length will be small. The actual average code length $\overline{L}$ obtained by using a discrete Laplace distribution with mean 0 and variance σ^2 to encode random variates from a Laplace distribution with mean 0 and variance σ_1^2 is given by

$$\overline{L} = \sum_k -p_{\sigma_1^2}(k) \log_2 p_{\sigma^2}(k). \tag{3.3}$$

By appropriate coder design using Equations (3.2) and (3.3), we can place a limit on the average extra code length.

3.2.1 Precomputation of the distributions

For maximum compression we will use arithmetic coding. We wish to obtain probabilities to pass to the arithmetic coder without excessive computation. To this end, we select a closely-spaced set of variances, with the idea of encoding a prediction error by using the closest variance from the set. We use Equation (3.1) to compute the Laplace distribution probabilities corresponding to each variance, and build these variances and distributions into both the encoder and the decoder. By careful choice of the set of variances, we can guarantee that the average extra code length due to using an approximate variance will always be less than any given amount. Only 37 variances and distributions are needed to ensure that the coding loss caused by the approximation is

Variance Range	Variance Used	Variance Range	Variance Used	Variance Range	Variance Used
0.005 −0.023	0.016	2.882− 4.053	3.422	165.814− 232.441	195.569
0.023 −0.043	0.033	4.053− 5.693	4.809	232.441− 326.578	273.929
0.043 −0.070	0.056	5.693− 7.973	6.747	326.578− 459.143	384.722
0.070 −0.108	0.088	7.973− 11.170	9.443	459.143− 645.989	540.225
0.108 −0.162	0.133	11.170− 15.627	13.219	645.989− 910.442	759.147
0.162 −0.239	0.198	15.627− 21.874	18.488	910.442− 1285.348	1068.752
0.239 −0.348	0.290	21.874− 30.635	25.875	1285.348− 1816.634	1506.524
0.348 −0.502	0.419	30.635− 42.911	36.235	1816.634− 2574.021	2125.419
0.502 −0.718	0.602	42.911− 60.123	50.715	2574.021− 3663.589	3007.133
0.718 −1.023	0.859	60.123− 84.237	71.021	3663.589− 5224.801	4267.734
1.023 −1.450	1.221	84.237−118.157	99.506	5224.801− 7247.452	6070.918
1.450−2.046	**1.726**	118.157−165.814	139.489	7247.452−10195.990	8550.934
2.046 −2.882	2.433				

Table 3.1: Laplace distribution variances to guarantee coding loss of less than 0.005 bit per pixel.

less than 0.005 bit per pixel, a barely measurable quantity. The variances and variance ranges are listed in Table 3.1. Note that exactly the same procedure can be used for normal distributions as well as other distributions in the generalized symmetric exponential family described in Section 3.4.2. The idea of a limited number of variances is similar in spirit to the ϵ-partition idea of Section 2.2.

Example: To encode a prediction error when we estimate that $\sigma^2 = 2$, we use the 12th line of Table 3.1 (highlighted in bold face), since $1.450 < 2 < 2.046$; thus we use the precomputed Laplace distribution with $\sigma^2 = 1.726$. The optimal average code length for Laplace random variates with $\sigma^2 = 2$ is 2.484 bits per sample; by using the discrete Laplace distribution with $\sigma^2 = 1.726$ we achieve an average code length of 2.488 bits per sample, within 0.004 bits per sample of the optimal length. □

3.2.2 Coding the error

Once we have prepared the probability distributions, arithmetic coding allows us to do the coding without difficulty. Given any discrete probability distribution, we can use arithmetic coding to encode random variates from that distribution with an average code length almost exactly equal to the entropy of the distribution, $\sum_k -p_k \log_2 p_k$; the extra code length introduced in the coding process is provably small, as shown in Section 2.1.4. Unlike Huffman coding, arithmetic coding performs well even when one of the probabilities is close to 1.

To encode a prediction error Δ_p (presumed to be a random variate from a Laplace distribution with zero mean and a given estimated variance), we select the "nearest" distribution according to Table 3.1, that is, the one that minimizes the average extra

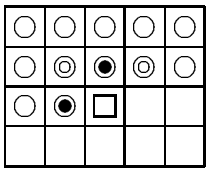


Figure 3.2: PPPM prediction and variance estimation contexts. When PPPM, proceeding in raster scan order, reaches the pixel marked □, the pixel values at all the circled points (marked ○, ◉, and ◎) are known. The intensity values of two pixels indicated by ◉ are used to predict the value of the □ pixel, and the prediction errors at the four pixels marked ◉ and ◎ are used to estimate the variance to be used for encoding the prediction error.

code length. We then simply use arithmetic coding to encode Δ_p according to the distribution selected.

3.3 PPPM: prediction by partial precision matching

The success of high-order, context-based methods for text compression naturally leads us to consider similar ideas for lossless image compression. We present a method similar in spirit to the prediction by partial matching method (PPM) of Cleary and Witten [7], but with a significant difference. PPM encodes each input symbol using the longest context in which the symbol has already occurred, up to a specified maximum length. For image compression this method as presented is not completely satisfactory: the main problem is that the exactly-repeated strings that make text compression possible are not present in images. Our solution is a new way of detecting approximate two-dimensional matches, eminently suited for image compression. We call this method *prediction by partial precision matching,* or *PPPM.*

PPPM encodes an image in raster-scan order. We separate the coding process into prediction, variance estimation, and error coding. For prediction we use a simple linear combination of a few nearby pixels. For variance estimation we attempt to make use of previous occurrences of the current context; however, instead of trying to maximize the *size* of the matching context as in PPM, we use a small, fixed-size context and focus on the *closeness* of the match. If we fail to find enough previous occurrences of the exact current context, we relax the exactness constraint, ignoring the least significant bit of each of the context pixels. If we still cannot find a match, we ignore the next significant bits until we finally find an approximate "context" that has occurred enough times; we use the statistics of that context to estimate the variance.

The prediction context and the variance estimation contexts need not be the same. In our implementation (see Figure 3.2) we use two pixels for prediction, the one immediately above and the one immediately to the left of the pixel being coded; the context consists of the intensity values of the predicting pixels. For variance estimation we

use a four-pixel ⊞-shaped neighborhood consisting of the three pixels above and the one immediately to the left of the pixel being coded; the context consists of the errors previously made in predicting the values of the pixels.

It is possible to use four or even eight *surrounding* pixels for prediction. If this is done for each pixel, we can reconstruct the image exactly if we are given the raw intensity value of just one pixel by solve a large set of sparse linear equations. However in practice this modification is difficult to implement because it requires careful attention to numerical issues. In addition it requires significantly more computation and achieves only marginally better compression.

3.3.1 Description of the PPPM algorithm

The encoding algorithm proceeds in raster-scan order. We encode each pixel p as follows:

1. We make a prediction P_p using the prediction context, and compute the prediction error $\Delta_p = A_p - P_p$.
2. We repeat the following steps as many times as necessary to estimate the variance:
   (a) We construct a key, based on the prediction errors of the pixels in the variance estimation context.
   (b) We search (using a hash table) for previous occurrences of the key.
   (c) If we find enough previous occurrences, we use the error statistics of those occurrences to obtain an estimated mean μ_p and variance σ_p^2 for the prediction error of the pixel; we leave the variance-estimation loop.
   (d) If we do not find enough previous occurrences, we drop one low-order bit from the prediction errors of the pixels in the variance estimation context.
3. We encode $\Delta_p - \mu_p$ as described in Section 3.2.
4. We update the statistics for one or more of the contexts just examined.

Decoding is very similar. We now examine the steps in more detail.

Prediction

In our current implementation, we predict a pixel's intensity by the rounded arithmetic mean of the intensities of the two pixels in the prediction context. At the top and left edges we simply use a single pixel's intensity. Our prediction in this method is admittedly very simple; a more sophisticated extrapolation function would give reduced error variances and improved compression. It does not suffice, however, simply to average more nearby pixels: using the average of the four pixels in the ⊞-shaped neighborhood gives worse compression than the two-pixel neighborhood that we use.

Variance estimation

As noted above, we construct "contexts" of decreasing precision based on the previously computed prediction errors of the four pixels in the variance estimation context. (We could use intensity values, but in practice using error values works slightly better.) Since we are using errors rather than intensities, we have to deal with negative values; we use a signed-magnitude representation, the sign being associated with the most significant nonzero bit.
Example: Using 8 "bits," we represent -5 and $+10$ as follows:

-5	0	0	0	0	0	$^-1$	0	1
$+10$	0	0	0	0	$^+1$	0	1	0

□

We follow the procedure described above to find a context that has occurred enough times to allow a satisfactory estimate of the variance of prediction errors made in that context. In practice, a context's statistics are unreliable until the context has occurred 10 or 15 times; we set the default threshold at 12. For the selected context we compute the mean μ_p and the variance σ_p^2 of its previous occurrences for use in the coding step.

Encoding the error

We adjust Δ_p by subtracting μ_p to reflect the context's mean prediction error, and we encode the result as described in Section 3.2.

Updating the statistics

Each pixel occurs in a number of variance estimation "contexts" of different precisions. Instead of updating each of them, we adopt a "lazy update" rule that saves time and gives better variance estimates. After coding each pixel we update the statistics for at most two contexts, the one used for coding and, if possible, the one with one additional bit of precision.

3.4 MLP: a multi-level progressive method

Our second image coding method, the *multi-level progressive* method (MLP), has the advantages that it is *progressive* and can be parallelized. By "progressive," we mean that the image is encoded in *levels*; the first level corresponds to a very highly compressed and very lossy encoding, and each successive level provides more detail, until ultimately the encoding is completely lossless. Truncating the encoded file will result in lossy compression of the entire image, rather than exact compression of part of the image. A progressive method allows an end user to browse an image without having to decode much of the encoded data; if more detail is desired, the image can be successively refined as more of the encoded data is decoded. An excellent survey of progressive techniques appears in [71].

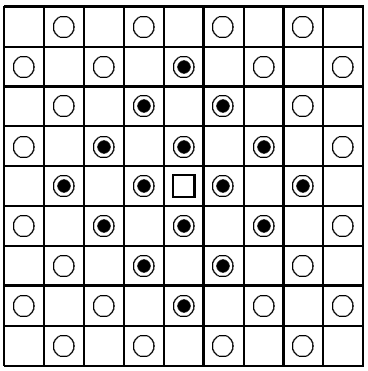


Figure 3.3: MLP last level prediction neighborhood. Before MLP processes its last level, the pixel values at the circled points (marked ○ and ◉) are known. The pixels indicated by ◉ are used to predict the value of the pixel at the center, marked □. All unmarked pixels will also be predicted during the last level.

The prediction step of MLP can be done completely in parallel by the encoder, and with a high degree of parallelism by the decoder; the coding step can be done in parallel as described in Section 2.4.

We encode the file in a series of *levels*, each level including twice as many pixels as the previous one. As we begin to encode each level, the pixels with known values are on the lattice points of a square grid. Using only the values of pixels on the lattice points, we predict the value of the pixel at the midpoint of each grid square, and encode the prediction error. (The decoder can make the same prediction, and hence it can use the decoded error to reconstruct the original value.) After all pixels in the level have been encoded, the set of known pixels forms a checkerboard pattern; by rotating and scaling the coordinate system we obtain a new square grid of pixels with known values, with the distance between adjacent pixels only $1/\sqrt{2}$ as much as before. Figure 3.3 shows the pixels involved in coding the last level.

Clearly the method is progressive: each level gives us the value of pixels selected uniformly from the entire image. For example, just before encoding the last level, we know the values of half the pixels of the image, in a checkerboard pattern over the whole image. If we stopped the encoding at that point, the decoder could compute the known pixels, and use the prediction function to interpolate the remaining pixels. In practice, even interpolating after skipping the last two levels ($^3/_4$ of the pixels in the image) leaves us with an image virtually indistinguishable from the original.

The time-consuming step in this method is prediction (not arithmetic coding), and the predictions are easy to parallelize. The set of predicting pixels for a given pixel is fixed at the beginning of coding, so for encoding we could even use one processor per pixel to predict in constant parallel time. Since the predictions made during decoding depend on values obtained at earlier levels, only the pixels at a given level can be predicted in parallel; the parallel prediction time for decoding an $n \times n$ image is still small, proportional to the number of levels, $2 \log_2 n$. The coding can be done in parallel as described in Section 2.4.

$\frac{1}{256}$	$-\frac{9}{256}$	$-\frac{9}{256}$	$\frac{1}{256}$
$-\frac{9}{256}$	$\frac{81}{256}$	$\frac{81}{256}$	$-\frac{9}{256}$
$-\frac{9}{256}$	$\frac{81}{256}$	$\frac{81}{256}$	$-\frac{9}{256}$
$\frac{1}{256}$	$-\frac{9}{256}$	$-\frac{9}{256}$	$\frac{1}{256}$

Table 3.2: Coefficients used in MLP for 16-point midpoint polynomial interpolation.

Precursors of MLP, which use much simpler predictors and less sophisticated variance estimation, are developed in [21,35,70]. A rotating coordinate system appears in [14,64].

3.4.1 Description of the MLP algorithm

In the MLP algorithm, the pixels in an image are divided into levels, each level having twice as many pixels as the preceding one. The pixels in a level are arranged in a (possibly rotated) checkerboard pattern. Within a level we apply three operations to each pixel:

1. We predict the pixel's intensity based on the intensities of other nearby pixels known from earlier levels, and compute the prediction error (the difference between the actual and predicted values).

2. We find an appropriate model for the prediction error, consisting of a probability distribution and an estimated variance.

3. We encode the prediction error using the estimated model in conjunction with a statistical coder, such as arithmetic coding.

Decoding is very similar.

For MLP we have chosen a pixel sequence based on the rotating coordinate system. That is, after all the pixels in a level have been coded, we rotate the coordinate system by 45 degrees and scale it by a factor of $1/\sqrt{2}$. In practice we do this by alternating between diagonal and axis-parallel coordinate systems, reducing the step size at each level.

It is necessary to deal with startup and edge effects, when the points that should be used as predictors are not present in the image. Because of the small number of pixels involved, these effects have little practical significance. In our implementation we simply take the prediction coefficients of missing points to be zero, and then renormalize the remaining coefficients.

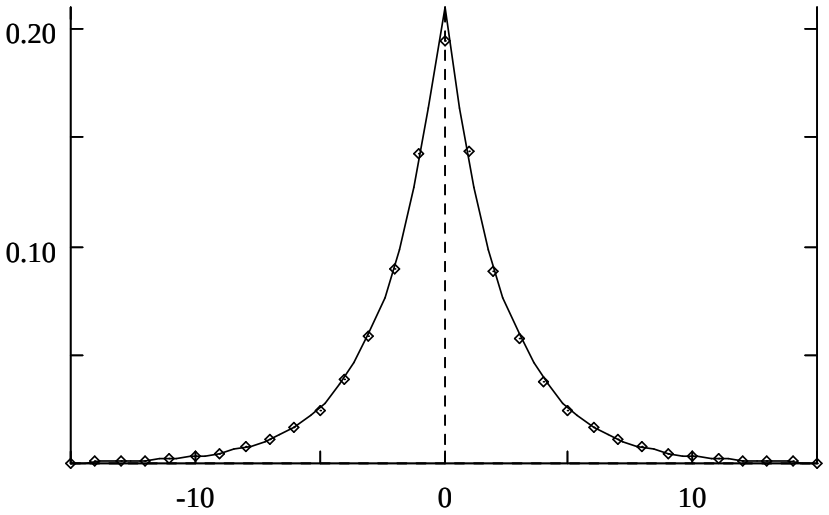


Figure 3.4: Comparison of continuous Laplace distribution ($\sigma = 3.370$) and the actual distribution of prediction errors (shown by small diamonds) found during the last level of application of method MLP to Band 4 of the Donaldsonville data set. The actual values are virtually indistinguishable from the theoretical values, except when the prediction error is 0; most of the difference in this case is because the curve is the continuous Laplace distribution, not the discrete distribution.

Parallel prediction

The prediction step for a pixel involves computing a linear combination of the values of a fixed constellation of nearby pixels whose values are already known. Clearly the encoder, having access to all pixel intensities at the beginning of the computation, can predict all values simultaneously; the decoder can make the same predictions, but only level-by-level, since the predictions depend on values from preceding levels.

3.4.2 Error modeling by variability index

Simple variance estimation

There is considerable choice in the method of variance estimation. In theory, for each pixel we can find the distribution that encodes the pixel's error as compactly as possible. For the Laplace distribution with mean 0, we find (by solving the equation $\partial p_{\sigma^2}(x)/\partial(\sigma^2) = 0$) that the optimal variance is $\sigma^2 = 2x^2$. Of course, this method is not practical because of the large overhead of encoding the optimal variance for each pixel. However, by optimally encoding the errors and *ignoring* the cost of encoding the variances, we can obtain a reasonable lower bound on the attainable code length. This lower bound applies to MLP for a given prediction function and for a particular image file, and is based on the assumption that the errors follow the Laplace distribution, but it does give a good indication of the compressibility of a particular image.

A more practical method of encoding the variances is to compute and encode the

variance of all the errors at a given level; the overhead is minimal, only $\log_2 37 \approx 5.21$ bits per level. A refinement of this approach is to divide each level into blocks of $k \times k$ pixels and compute and encode the variance of the errors of the pixels in each block. The overhead increases with the number of blocks, but for medium-sized values of k (say $k = 16$) the net code length is often reduced because of the more realistic error model.

Modeling by variability index

In [29] we introduce the notion of error modeling by *variability index*, presenting and motivating the algorithm. In this section we use the variability index both for error modeling and as an experimental tool to show that prediction errors are not always Laplace distributed, as previously assumed. We also indicate a family of distributions that can provide more accurate error models. In our experiments we find that application of the variability index algorithm to Laplace distributions leads to an improvement in compression efficiency of about 4 percent compared with implementations with explicit transmission of variances; further refinement by allowing a wider family of error distributions gives another half percent improvement.

Our goal is to estimate the local variance of the pixel prediction errors in a given level of the MLP encoding. This will allow us to select the appropriate distribution (the one with the estimated variance) to use in encoding each prediction error by arithmetic coding. Using a single variance for the entire level disregards local variations in the image; estimating and explicitly encoding variances for small sections of the image incurs considerable overhead. It would seem that previously encountered prediction errors of nearby pixels would be related to a given pixel's error, but there is no obvious way to use them directly.

We estimate the variance by an indirect method: for each pixel we compute a *variability index*, a quantity with a strong correlation to the local variance. On the supposition that pixels with similar variability index will have similar error models, we adaptively estimate the local variance based on the variability index. The algorithm is as follows:

1. We initialize the variance estimate V.
2. For each pixel, we compute the variability index.
3. We sort the pixels in variability index order.
4. For each pixel (in order of decreasing variability index):
   (a) We code the prediction error d, using V as the variance.
   (b) We use the prediction error d to update V. The variance is just the mean squared error; our variance estimate is an exponentially weighted mean squared error, computed by exponential smoothing:

$$V := f \cdot V + (1 - f) \cdot d^2.$$

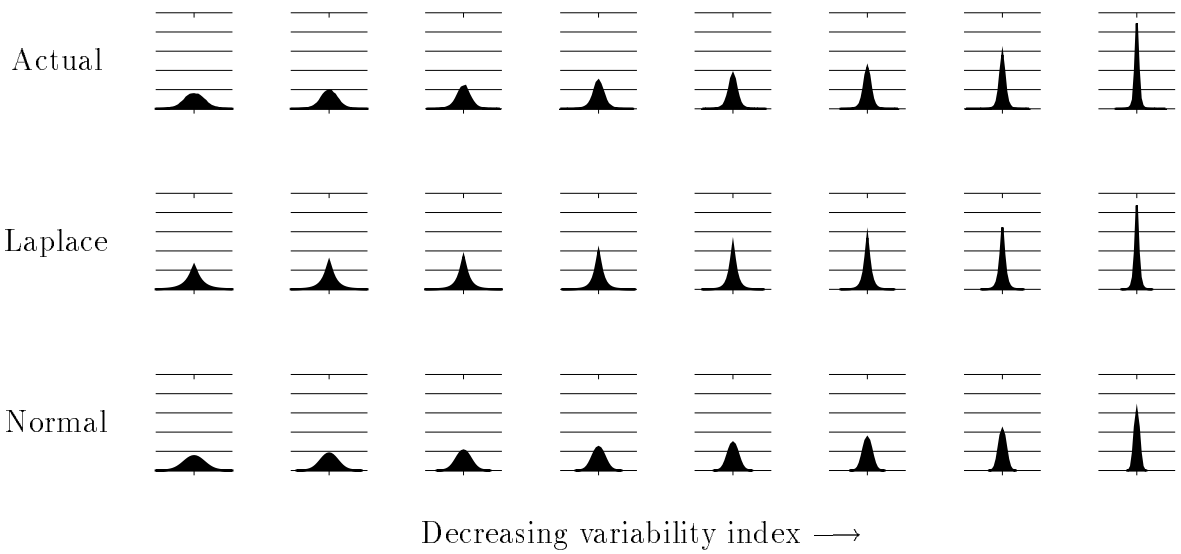


Figure 3.5: Histograms showing distributions of prediction errors. The top row shows the actual distributions of prediction errors for eight batches of 16,384 points each taken from the last level of the coding of Band 4 of the Donaldsonville Thematic Mapper data set. The variability index decreases from left to right. The second and third rows show the discretized Laplace distributions and normal distributions with the same variances as the actual data in the corresponding column. The horizontal lines are at intervals of 10 percent probability; prediction errors from -20 to $+20$ are plotted.

Note that the variability index is used only for sorting the pixels, then discarded. We use decreasing variability index order because compression efficiency is less dependent on accurate variance estimates for large variances; hence we encode pixels with large variances first in each level, before our variance estimate has stabilized. The sorting step removes any natural ordering of the pixels, so both the encoder and the decoder must maintain the pixel coordinates.

We have tried several different quantities for the variability index, including both intensity values and prediction errors of pixel neighborhoods of various sizes. Empirically, the most effective is simply the variance of the four nearest pixels. The smoothing parameter f must also be selected. Experiments show that the best results come from a large value like 0.992; smaller values make the estimated variance too sensitive to random fluctuations in the variability index.

Distribution Selection

When we code the pixels of a level in variability index order, we can plot histograms of the error distributions for different values of the variability index. A typical set of plots is shown in Figure 3.5. The error distributions are not always Laplacian; in fact, for large variability index they appear closer to normal, but they become closer to the Laplace distribution for smaller values of variability index.

This leads us to consider a family of generalized symmetric exponential distribution functions which includes both the Laplace distribution and the normal distribution. Distributions in this family have the form

$$f_{n,\sigma}(x) = \frac{\alpha_n}{\sigma} \exp\left(-\beta_n \left|\frac{x}{\sigma}\right|^n\right). \tag{3.4}$$

From the constraints on a probability distribution with variance σ^2, we can find α_n and β_n:

$$\alpha_n = \frac{n}{2}\left(\frac{\Gamma(3/n)}{\Gamma(1/n)^3}\right)^{1/2}, \qquad \beta_n = \left(\frac{\Gamma(3/n)}{\Gamma(1/n)}\right)^{n/2}.$$

For the Laplace distribution ($n = 1$), we have $\alpha_1 = 1/\sqrt{2}$ and $\beta_1 = \sqrt{2}$; for the normal distribution ($n = 2$), we have $\alpha_2 = 1/\sqrt{2\pi}$ and $\beta_2 = 1/2$.

We can use these distributions to optimize the exponent n in Equation (3.4), either image by image or for a class of images. The best values of n for our individual test images range from 1.0 to 1.9, the best overall value being 1.25. For many of our test images we obtain a further slight improvement when we vary the exponent within each level; in our experiments we use a given starting exponent and reduce it linearly toward 1 as we progress through each level. The improvement obtained by using a non-Laplace distribution is about a half percent; the extra improvement obtained by varying the exponent within a level is small, as is the improvement from selecting the optimal exponent image by image.

It appears that the distributions we see are actually mixtures of normal or near-normal distributions. High variability indices arise only from regions with high local variance, so the mixture distribution contains contributions only from a small range of distributions. Low variability indices, on the other hand, can come either from regions with low local variance or from the (not unusual) occurrence of small deviates from regions with high local variance. The resulting mixture has contributions from many different distributions, and tends to be more peaked near zero; this gives the characteristic Laplace distribution shape.

Parallel error modeling using the variability index technique

Error modeling is most effective when done implicitly. The variability index technique is an implicit method for estimating local image variances that leads to better compression than any other published lossless image compression method. Like the intensity prediction, the variability index computation depends only on the values of a few nearby pixels, known from a previous level, so it can be done in parallel by both encoder and decoder with no loss of efficiency. The assignment of variances to values of the variability index can be done implicitly in a sequential environment, adaptively estimating the variance while working through the pixels in order of variability index. For parallel coding the use of side information is more appropriate: we can group the pixels after sorting (sorting can be done efficiently in parallel), then compute the variance of each group and transmit it in coded form. Each variance requires only a few

bits to transmit, typically four or fewer; if we divide each level of n pixels into $\sqrt{n}$ groups, only about $(\sqrt{2}+1)m$ variances must be transmitted for an $m \times m$ image. For a 512×512 image this amounts to 1236 variances, or only 618 bytes of side information.

The variability index technique has the effect of classifying pixels by local variance. This translates roughly into a classification by code length, since distributions with larger variances usually have larger average code length. Using this classification we can assign pixels to processors in a way that divides the *code length* (not just the number of pixels) approximately evenly among the processors, thus reducing the number of pixel reallocations needed. The simulation results described in Section 2.4.1 confirm the usefulness of this technique.

3.5 FELICS: a fast, efficient, lossless image compression system

In [32] we present a simple system for lossless image compression that runs very fast with only minimal loss of compression efficiency. We call this technique *FELICS*, for *Fast, Efficient, Lossless Image Compression System*. We use raster-scan order, and we use a pixel's two nearest neighbors to directly obtain an approximate probability distribution for its intensity, in effect combining the prediction and error modeling steps. We use the parameter estimation technique of Section 2.3.3 to select the closest of a set of error models, each corresponding to a Rice code. Finally we encode the intensity using the Rice code. The resulting compressor runs about five times as fast as an implementation of the lossless mode of the JPEG proposed standard while obtaining slightly better compression on many images.

3.5.1 Description of the FELICS algorithm

Proceeding in raster-scan order, we code each new pixel P[1] using the intensities of the two nearest neighbors of P that have already been coded; except along the top and left edges, these are the pixel above and the pixel to the left of the new pixel. We call the smaller neighboring value L and the larger value H, and we define Δ to be the difference $H - L$. We treat Δ as the prediction context of P, used for code parameter selection.

The idea of the coding algorithm is to use one bit to indicate whether P is in the range from L to H, an additional bit if necessary to indicate whether it is above or below the range, and a few bits, using a simple prefix code, to specify the exact value. This method leads to good compression for two reasons: the two nearest neighbors provide a good context for prediction, and the image model implied by the algorithm closely matches the distributions found in real images. In addition, the method is very fast, since the coding uses only single bits and simple prefix codes.

[1] With a slight abuse of notation, we use symbols P, H, L, N_1, and N_2 to refer both to pixels and to their intensity values.

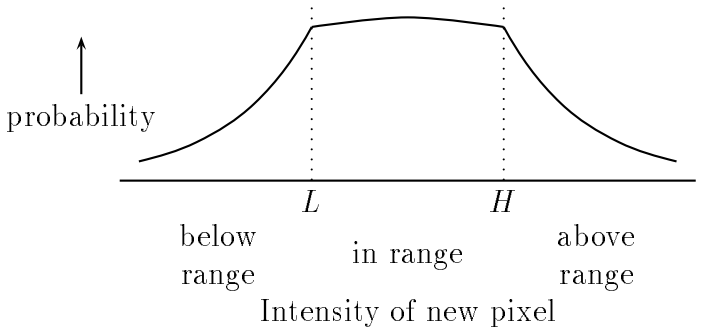


Figure 3.6: Schematic probability distribution of intensity values for a given context $\Delta = H - L$ in FELICS.

Intensity distributions

When we examine the distribution of an image's intensity values for each context Δ, we find that the intensities are generally distributed as shown in Figure 3.6. Typically P lies in the range $[L, H]$ about half the time, requiring one bit to encode, and when P is out of range, the above-range/below-range decision is symmetrical, so another one-bit code is appropriate. In-range values of P are almost uniformly distributed, with a slight crest near the middle of the range, so an adjusted binary encoding (as described in Section 2.3) gives nearly optimal compression when P is in range. The probability of out-of-range values falls off sharply, so when P is out of range it is reasonable to use exponential prefix codes, i.e., Golomb codes or the simpler Rice codes. This distribution clearly differs from the Laplace distribution commonly assumed in predictive image coding. Fränti [18] has applied the FELICS technique to the problem of block truncation coding for lossy image compression; he finds that the distributions of quantization levels of the sub-sample images are similar to the distributions described here, and that the FELICS method is generally effective in encoding the quantization levels.

Formal description of FELICS algorithm

To encode an image, we output the first two pixels without coding, then repeat the following steps:

1. We select the next pixel P and find its two nearest neighbors N_1 and N_2.
2. We compute $L = \min(N_1, N_2)$, $H = \max(N_1, N_2)$, and $\Delta = H - L$.
3. (a) If $L \leq P \leq H$, we use one bit to encode IN-RANGE; then we use an adjusted binary code to encode $P - L$ in $[0, \Delta]$.
   (b) if $P < L$, we use one bit to encode OUT-OF-RANGE, and one bit to encode BELOW-RANGE. Then we use a Rice code to encode the non-negative integer $L - P - 1$.

(c) if $P > H$, we use one bit to encode OUT-OF-RANGE, and one bit to encode ABOVE-RANGE. Then we use a Rice code to encode the non-negative integer $P - H - 1$. The coding parameter k is chosen according to the method of Section 2.3.3.

(d) We update the statistics used in selecting the Rice coding parameter.

The decoding algorithm involves simply reversing step 3, decoding the in-range/out-of-range and above-range/below-range decisions, branching accordingly, and adjusting the decoded numbers to obtain the value of P.

Implementation and refinements

The basic algorithm of Section 3.5.1 is very easy to implement. As described it encodes and decodes very quickly and gives good compression. The implementation requires very little memory, only enough to store one scan line of input data and a few counts for each of the few hundred possible contexts.

In this section we describe several enhancements that can be made to improve speed and compression. One possibility is to freeze the parameter choice for a context after a number of symbols have been encoded. This saves the time needed to maintain the cumulative counts, and does not hurt compression much since in practice the parameter selection algorithm converges quickly to the best value.

A second enhancement is to adjust the range $[L, H]$ when $L = H$. In this special case, the in-range probability tends to be somewhat smaller than $1/2$, and it makes sense to use the range $[L-1, H+1]$. We can do this either unconditionally or based on the number of times that the value to be encoded is equal to L (and H) when $\Delta = 0$. We choose the second possibility, adjusting the interval when the values encoded have fallen out of the one-value "range" more than $2/3$ of the time. We might also consider *balance coding*, adjusting the range for each context to adaptively balance the in-range and out-of-range probabilities. We can use the parameter estimation algorithm of Section 2.3.3 to choose the amount of adjustment.

One final useful refinement is to assign a small initial penalty to the cumulative code lengths for small values of the parameter k in each context, to prevent their accidental use in contexts where the probability distribution is flat; such use can greatly increase the code length for a single pixel.

We have considered periodic count scaling to exploit locality of reference within an image. We tried applying it to the cumulative code lengths in each context. Halving all code lengths when the smallest one reaches 1024 can improve compression by up to 0.3 percent, but encoding time increases by about 10 percent, too much of a time penalty to pay for such a small increase in compression.

Finally, we note that Golomb coding gives only a marginal improvement over Rice coding despite the wider range of model parameters available. The extra compression is typically less than one percent, and the time required almost doubles because of the need to maintain cumulative bit counts for more possible parameter values. Therefore

Washington, D.C. Donaldsonville, Louisiana Ridgely, Maryland

Figure 3.7: A 256×256 section of Band 4 of each Landsat Thematic Mapper data set.

we use Rice coding in our implementation. In practice, it is seldom necessary to allow any value of the Rice coding parameter k other than 0, 1, 2, or 3.

Extension to progressive coding

FELICS is based on a raster-scan pixel sequence. We expect in the future to extend the FELICS system to a hierarchical pixel sequence as in the MLP method, to allow progressive coding. Before committing to a particular prediction method, it is necessary to study the actual distributions of intensity values. One possibility is to use the smallest and largest of the current pixel's four nearest neighbors, but the resulting distributions must be tested.

3.6 Experimental results

The 28 images comprising our test data include 21 Landsat Thematic Mapper images and seven other images (the "USC images") widely used in compression studies. There are three Landsat data sets, each consisting of seven images (spectral bands); the locations are Washington, D.C., Donaldsonville, Louisiana (90 kilometers west of New Orleans), and Ridgely, Maryland (70 kilometers southeast of Baltimore). Bands 1 through 7 of the Washington data are identified as W1 through W7; the Donaldsonville and Ridgely images are identified as D1 through D7 and R1 through R7 respectively. Each band is coded independently of the other bands; we do not attempt to make use of the correlation between bands in a data set. All images consist of 512×512 8-bit grayscale pixels except the Ridgely images, which are 368×468 pixels. Each of the Landsat data sets contains one highly compressible image (compressible to better than $8:1$) and six "normal" images. Portions of three of the Landsat images are shown in Figure 3.7.

3.6.1 Compression performance

We compare our results with the lossless mode of the standard for image compression recently developed by the Joint Photographic Experts Group (JPEG) of the CCITT/ISO. The JPEG lossless mode is based on prediction by one, two, or three points, followed by Huffman coding or arithmetic coding; encoding proceeds in raster scan order. The standard gives some latitude to implementors; we report results from an implementation based on two-point prediction with arithmetic coding[2]. For all except the three highly compressible images, two-point prediction provides $4\frac{0}{0}$ to $10\frac{0}{0}$ more compression gain than three-point prediction.

We include results for the Minimax coder, AT&T's original lossless compression submission to the JPEG; it seems to give somewhat better compression performance than JPEG lossless mode. As a matter of interest we include the UNIX *compress* program, although it generally does poorly on images; most of its compression is attributable to its ability to capture the context-independent entropy of any file.

As for our methods, we give results for the PPPM method, using two-point prediction. We report results for just one version of the MLP algorithm, the one that uses variability index modeling, using Equation (3.4) with n starting at 1.5 in each level, falling to $n = 1$ (Laplace) by the end of each level. As noted in Section 3.4.2, there is only a small improvement when we optimize n for each image or vary the exponent within each level. Optimizing for each image requires compressing each image many times, so we omit that step in our reported results; varying the exponent within each level is a simple procedure that does improve compression slightly, so we include that step. For the FELICS method we include freezing when the smallest code length for a context reaches 1024, we adjust the range when $\Delta = 0$, and we assign small initial penalties to the cumulative counts for small parameter values. The refinements do not have much effect on compression, but they increase throughput by roughly 5 to 20 percent.

We present our results in three tables. Table 3.3 gives the compression gains (in percent log ratio) with respect to the lossless mode of the JPEG proposed standard. We also give the same results in terms of the more familiar compression ratio in Table 3.4, and in bits per pixel in Table 3.5. Results are unavailable for the Minimax coder and PPPM for the USC images.

3.6.2 Speed performance of FELICS

We give more detailed results for the FELICS algorithm to show that it increases throughput without degrading compression by much. All runs were made on a Sun SPARCstation 1GX.

In Table 3.6 we see that except for the three highly compressible images, FELICS compresses as well as JPEG lossless mode, with about five times the throughput.

[2] Assistance in the evaluation of JPEG lossless mode was provided by Allan R. Moser of E. I. duPont de Nemours and Company (Inc.).

Image	JPEG	Minimax	*compress*	PPPM	MLP	FELICS
W1	–	3.6	−19.9	5.3	6.3	1.2
W2	–	3.4	−18.9	4.4	5.8	−1.4
W3	–	3.5	−17.2	5.8	6.6	1.2
W4	–	4.0	−21.6	6.4	6.7	1.1
W5	–	3.6	−22.6	6.4	6.7	0.9
W6	–	2.0	−39.1	−8.0	20.7	−47.6
W7	–	3.5	−21.2	5.2	6.3	0.9
D1	–	3.2	−23.5	5.8	6.4	1.4
D2	–	2.8	−26.6	3.6	5.2	−7.9
D3	–	2.7	−26.1	6.1	5.4	−4.2
D4	–	3.2	−32.1	8.4	7.1	0.5
D5	–	2.8	−30.6	8.0	6.8	1.0
D6	–	1.1	−41.0	−7.5	19.6	−55.4
D7	–	2.7	−27.3	6.5	6.5	0.4
R1	–	4.5	−24.3	5.5	7.6	2.8
R2	–	4.4	−26.0	4.3	7.8	−3.6
R3	–	4.1	−27.7	5.6	8.1	0.5
R4	–	4.1	−23.8	6.5	8.0	1.8
R5	–	3.7	−28.3	7.5	8.3	3.1
R6	–	3.7	−27.3	6.4	8.0	2.4
R7	–	5.5	−30.1	−5.7	16.5	−39.4
couple	–	–	−27.4	–	6.3	4.0
crowd	–	–	−35.4	–	8.3	−4.0
lax	–	–	−23.5	–	5.3	2.9
lena	–	–	−41.8	–	9.4	−0.4
man	–	–	−35.2	–	6.7	1.9
woman1	–	–	−19.2	–	5.5	1.9
woman2	–	–	−48.8	–	9.2	−2.4

Table 3.3: Compression gains for images, expressed in percent log ratio and compared with the JPEG lossless mode.

Image	JPEG	Minimax	*compress*	PPPM	MLP	FELICS
W1	2.07	2.15	1.70	2.18	2.21	2.10
W2	2.67	2.76	2.21	2.79	2.83	2.63
W3	2.28	2.36	1.92	2.42	2.44	2.31
W4	1.81	1.88	1.46	1.93	1.93	1.83
W5	1.68	1.74	1.34	1.79	1.80	1.70
W6	7.92	8.08	5.36	7.31	9.74	4.92
W7	2.10	2.17	1.70	2.21	2.23	2.12
D1	2.26	2.34	1.79	2.40	2.41	2.29
D2	3.07	3.16	2.36	3.19	3.24	2.84
D3	2.58	2.65	1.99	2.74	2.72	2.47
D4	1.85	1.91	1.34	2.01	1.98	1.86
D5	1.82	1.87	1.34	1.97	1.95	1.84
D6	9.25	9.35	6.14	8.58	11.25	5.32
D7	2.17	2.23	1.65	2.31	2.31	2.18
R1	2.28	2.38	1.79	2.41	2.46	2.34
R2	2.94	3.07	2.26	3.06	3.17	2.83
R3	2.45	2.56	1.86	2.60	2.66	2.47
R4	2.24	2.33	1.76	2.39	2.42	2.28
R5	1.78	1.85	1.34	1.92	1.94	1.84
R6	2.08	2.15	1.58	2.21	2.25	2.13
R7	7.43	7.85	5.50	7.02	8.76	5.01
couple	1.54	–	1.17	–	1.64	1.61
crowd	1.87	–	1.31	–	2.03	1.79
lax	1.31	–	1.04	–	1.38	1.35
lena	1.72	–	1.14	–	1.89	1.72
man	1.64	–	1.15	–	1.75	1.67
woman1	1.58	–	1.30	–	1.67	1.61
woman2	2.28	–	1.40	–	2.50	2.23

Table 3.4: Compression ratios for images, expressed as original image size divided by compressed file size.

Image	JPEG	Minimax	*compress*	PPPM	MLP	FELICS
W1	3.86	3.73	4.71	3.66	3.63	3.82
W2	2.99	2.89	3.62	2.86	2.83	3.04
W3	3.51	3.39	4.17	3.31	3.28	3.47
W4	4.42	4.25	5.48	4.15	4.13	4.37
W5	4.76	4.59	5.97	4.47	4.46	4.72
W6	1.01	0.99	1.49	1.09	0.82	1.63
W7	3.82	3.68	4.72	3.62	3.58	3.78
D1	3.54	3.43	4.48	3.34	3.32	3.49
D2	2.60	2.53	3.39	2.51	2.47	2.82
D3	3.10	3.02	4.02	2.92	2.94	3.23
D4	4.33	4.20	5.98	3.98	4.04	4.31
D5	4.40	4.28	5.97	4.06	4.11	4.36
D6	0.87	0.86	1.30	0.93	0.71	1.50
D7	3.69	3.59	4.85	3.46	3.46	3.67
R1	3.51	3.36	4.48	3.32	3.25	3.41
R2	2.73	2.61	3.54	2.61	2.52	2.82
R3	3.26	3.13	4.30	3.08	3.01	3.24
R4	3.58	3.43	4.54	3.35	3.30	3.51
R5	4.48	4.32	5.95	4.16	4.13	4.35
R6	3.86	3.71	5.07	3.62	3.56	3.76
R7	1.08	1.02	1.46	1.14	0.91	1.60
couple	5.19	–	6.82	–	4.87	4.98
crowd	4.28	–	6.11	–	3.94	4.46
lax	6.11	–	7.73	–	5.80	5.94
lena	4.64	–	7.05	–	4.23	4.66
man	4.88	–	6.95	–	4.57	4.80
woman1	5.07	–	6.14	–	4.80	4.97
woman2	3.50	–	5.71	–	3.19	3.59

Table 3.5: Average code lengths for images, expressed in bits per pixel.

	Compressed size				Encoding throughput			
	FELICS		JPEG	*compress*	FELICS		JPEG	*compress*
	Plain	Adjusted			Plain	Adjusted		
W1	2.10	2.10	2.07	1.70	78.5	87.7	16.3	93.6
W2	2.63	2.63	2.67	2.21	87.1	92.6	21.0	93.6
W3	2.31	2.31	2.28	1.92	82.2	90.1	17.8	77.1
W4	1.83	1.83	1.81	1.46	75.8	85.4	14.2	70.8
W5	1.70	1.70	1.68	1.34	73.6	83.5	13.2	70.8
W6	3.78	4.92	7.92	5.36	107.9	98.6	48.5	163.8
W7	2.11	2.12	2.10	1.70	79.0	87.7	16.5	63.9
D1	2.30	2.29	2.26	1.79	81.7	90.7	17.7	84.6
D2	2.84	2.84	3.07	2.36	92.6	93.0	23.6	100.8
D3	2.47	2.47	2.58	1.99	87.7	90.1	20.0	79.4
D4	1.86	1.86	1.85	1.34	76.9	84.6	14.6	81.9
D5	1.84	1.84	1.82	1.34	77.6	85.1	14.4	84.6
D6	3.84	5.32	9.25	6.14	107.0	98.2	53.5	154.2
D7	2.18	2.18	2.17	1.65	81.2	88.9	17.1	69.0
R1	2.34	2.34	2.28	1.79	80.9	89.2	17.9	101.3
R2	2.83	2.83	2.94	2.26	90.6	93.6	22.7	107.6
R3	2.47	2.47	2.45	1.86	86.1	90.6	19.4	95.7
R4	2.28	2.28	2.24	1.76	82.8	87.4	17.8	90.6
R5	1.84	1.84	1.78	1.34	76.2	80.9	14.0	71.8
R6	2.13	2.13	2.08	1.58	79.7	85.7	16.4	74.9
R7	3.82	5.01	7.43	5.50	105.0	97.9	46.5	143.5
couple	1.61	1.61	1.54	1.17	74.9	84.6	12.2	84.6
crowd	1.80	1.79	1.87	1.31	79.4	86.2	14.8	87.4
lax	1.35	1.35	1.31	1.04	68.4	79.9	10.4	61.0
lena	1.75	1.72	1.72	1.14	73.4	83.8	13.9	81.9
man	1.68	1.67	1.64	1.15	75.1	84.6	13.2	77.1
woman1	1.62	1.61	1.58	1.30	74.3	83.8	12.7	72.8
woman2	2.23	2.23	2.28	1.40	82.2	89.8	18.1	72.8

Table 3.6: Image compression ratios and encoding throughput. The compression ratios are expressed as original size divided by compressed size. The encoding throughput is expressed as thousands of pixels encoded per second on a SPARCstation 1GX.

In fact, FELICS is about as fast as *compress*, and gives much better compression, not surprising since *compress* is designed for text, not images. The "plain" FELICS figures refer to a version with none of the refinements mentioned in Section 3.5.1; the "adjusted" FELICS figures include freezing when the smallest code length for a context reaches 1024, adjusting the range when $\Delta = 0$, and assigning small initial penalties to the cumulative counts for small parameter values. Note that the refinements seldom have much effect on compression, and they increase throughput by roughly 5 to 20 percent.

4

TEXT COMPRESSION

In text compression, we usually would like to compress files using just a single pass of the data, gathering statistics as we go. Such a modeling system is called *adaptive*. In Section 4.1 we discuss adaptive and semi-adaptive (two-pass) models, and show that in a certain sense they are equivalent. In Section 4.2 we discuss *scaling*, a method for improving compression by utilizing locality of reference. In Section 4.2.1 we give a detailed analysis showing that scaling can improve compression considerably, and can never hurt compression by very much. In Section 4.3 we discuss the Prediction by Partial Matching (PPM) method of Cleary and Witten [7], and in Section 4.3.2 we present a practical improvement to PPMC, the preferred text compression program in the literature. In Section 4.4 we present a fast PPM-like text compression system based on the use of prefix codes and quasi-arithmetic coding.

4.1 Adaptive and semi-adaptive models for text compression

In this section we show the relationship between semi-adaptive and adaptive models based on a zero-order Markov model, that is, context-independent symbol counts.

4.1.1 Semi-adaptive codes

Semi-adaptive codes are conceptually simple, and useful when real-time operation is not required; their main drawback is that they require knowledge of the file statistics prior to encoding. Statistics collected during the first pass over the file are normally used in a *static code*; that is, the probabilities used to encode the file remain the same during the entire encoding. It is possible to obtain better compression by using a *decrementing code*, dynamically adjusting the probabilities to reflect the statistics of just that part of the file not yet coded. Assuming that encoding uses exact arithmetic, and that there are no computational artifacts, we can use the file statistics to form a

static probabilistic model. Not including the cost of transmitting the model, the code length L_{SS} for a static semi-adaptive code is

$$\begin{aligned} L_{SS} &= -\log_2 \prod_{i=1}^{n} (c_i/t)^{c_i} \\ &= t \log_2 t - \sum_{i=1}^{n} c_i \log_2 c_i, \end{aligned}$$

the information content of the file. Dividing by the file length gives the self-entropy of the file.

Example 14: We illustrate a static code, encoding the eight-symbol file *abacbcba*. Based on exact symbol counts within the file, we have probabilities $p_a = 3/8$, $p_b = 3/8$, and $p_c = 1/4$, which we use during the entire encoding. (We ignore the problem of indicating the end of the file.) In the next few examples we show symbol probabilities rather than subintervals, because the changing probabilities are the distinctive feature of adaptive models.

Symbol	Current Interval		p_a	p_b	p_c
Start	0.000000000	1.000000000	3/8	3/8	2/8
a	0.000000000	0.375000000	3/8	3/8	2/8
b	0.140625000	0.281250000	3/8	3/8	2/8
a	0.140625000	0.193359375	3/8	3/8	2/8
c	0.180175781	0.193359375	3/8	3/8	2/8
b	0.185119629	0.190063477	3/8	3/8	2/8
c	0.188827515	0.190063477	3/8	3/8	2/8
b	0.189291000	0.189754486	3/8	3/8	2/8
a	0.189291000	0.189464808	3/8	3/8	2/8

The final interval is [0.00110 00001 11011, 0.00110 00010 00000] in binary. The theoretical code length is $-\log_2 0.000173808 \approx 12.490$ bits. The actual code length is 13 bits, since the final subinterval can be determined from the output **00110 00001 111**. □

If we know a file's exact statistics ahead of time, we can get improved compression by using a *decrementing code*. We modify the symbol weights dynamically (by decrementing each symbol's weight each time it is encoded) to reflect symbol frequencies for just that part of the file not yet encoded; hence for each symbol we always use the best available estimates of the next-symbol probabilities. In a sense, it is a misnomer to call this a "semi-adaptive" model since the model adapts throughout the second pass, but we apply the term here since the symbol probability estimates are based primarily on the first pass. The decrementing count idea appears in the analysis of enumerative codes by Cleary and Witten [8] and also in [42]. The code length for a semi-adaptive

decrementing code is

$$L_{SD} = -\log_2 \left(\left(\prod_{i=1}^{k} c_i! \right) \Big/ t! \right). \tag{4.1}$$

Example 15: We illustrate a decrementing code with the same file as in Example 14. Again we start with probabilities $p_a = 3/8$, $p_b = 3/8$, and $p_c = 1/4$, but during encoding the probabilities change.

Symbol	Current Interval		p_a	p_b	p_c
Start	0.000000000	1.000000000	$3/8$	$3/8$	$2/8$
a	0.000000000	0.375000000	$2/7$	$3/7$	$2/7$
b	0.107142857	0.267857143	$2/6$	$2/6$	$2/6$
a	0.107142857	0.160714286	$1/5$	$2/5$	$2/5$
c	0.139285714	0.160714286	$1/4$	$2/4$	$1/4$
b	0.144642857	0.155357143	$1/3$	$1/3$	$1/3$
c	0.151785714	0.155357143	$1/2$	$1/2$	$0/2$
b	0.153571429	0.155357143	$1/1$	$0/1$	$0/1$
a	0.153571429	0.155357143	$0/0$	$0/0$	$0/0$

The final interval is $[0.00100\ 11101\ 01000, 0.00100\ 11111\ 00011]$ in binary. The theoretical code length is $-\log_2 0.001785714 \approx 9.129$ bits, considerably shorter than in Example 14, because the probabilities used at each point reflect the actual frequencies of the symbols in just the remaining portion of the file, not the entire file. The actual code length is 10 bits, since the final subinterval can be determined from the output **00100 11110**. □

In the following simple theorem we show that a decrementing code improves on the corresponding semi-adaptive static code:

Theorem 8 *For all input files, the code length of a semi-adaptive decrementing code is at most that of a semi-adaptive static code. Equality holds only when the file consists of repeated occurrences of a single letter.*

Proof: Neither L_{SS} nor L_{SD} depends on the order in which the symbols appear in the file, so we can inductively build up a file with the same code length as any given file by starting with one occurrence of each symbol, then adding the remaining occurrences of each symbol. We show that the inequality holds for files with one occurrence of each symbol, and that adding occurrences of any symbol maintains the inequality.

Using a superscript "1" to denote a file consisting of exactly one occurrence of each of k symbols, we see that $L^1_{SS} = -\log_2(1/k^k)$ and $L^1_{SD} = -\log_2(1/k!)$. Since $k^k > k!$ for all $k \geq 2$, we have $L^1_{SS} > L^1_{SD}$ for $k \geq 2$. If we have a file with t symbols in it and we add another occurrence of a symbol that already occurs c times, we increase L_{SS} by $\log_2(((t+1)/t)^t) - \log_2(((c+1)/c)^c) + \log_2((t+1)/(c+1))$, and we increase L_{SD} by $\log_2((t+1)/(c+1))$. Since $t > c$ and $\log_2(((x+1)/x)^x)$ is an increasing function for

$x > 0$, we have $\log_2(((t+1)/t)^t) > \log_2(((c+1)/c)^c)$, so L_{SS} increases by more than L_{SD}.

Finally, if $k = 1$ (the file consists of repeated occurrences of a single letter), $L_{SS} = L_{SD} = 0$. □

Static semi-adaptive codes have been widely used in conjunction with Huffman coding. They are appropriate in this case: in Huffman coding we wish to avoid changing weights since changing weights often requires changing the structure of the coding tree. Our theorem does not contradict Shannon's theorem [67]; he discusses only the best static code.

Encoding the model

If we assume that all symbol distributions are equally likely for a given file length t, the cost of transmitting the exact model statistics is

$$\begin{aligned} L_M &= \log_2(\text{number of possible distributions}) \\ &= \log_2 \binom{t+n-1}{n-1} \\ &\sim n \log_2(et/n). \end{aligned} \tag{4.2}$$

Example: For our short sample file, $t = 8$ and $n = 3$, so $L_M = \log_2 \binom{10}{2} = \log_2 45 \approx 5.492$. □

A similar result appears in [4] and [8]. For a typical file L_M is only about 2560 bits, or 320 bytes. The assumption of equally-likely distributions is not very good for text files; in practice we can reduce the cost of encoding the model by 50 percent or more by encoding each of the counts using a suitable encoding of the integers, such as Fibonacci coding [17].

Strictly speaking we must also encode the file length t before encoding the model; the cost is insignificant, between $\log_2 t$ and $2 \log_2 t$ bits using an appropriate encoding of integers [13,69,75].

4.1.2 Adaptive codes

Adaptive codes use a continually changing model, in which we use the frequency of each symbol up to a given point in the file as an estimate of its probability at that point.

We can encode a symbol using arithmetic coding only if its probability is nonzero, but in an adaptive code we have no way of estimating the probability of a symbol before it has occurred for the first time. This is the *zero-frequency problem*, discussed in detail in [4] and [76]. For large files with small alphabets and simple models, all solutions to this problem give roughly the same compression. In this section we adopt

the solution used in [77], simply assigning an initial weight of 1 to all alphabet symbols. The code length using this adaptive code is

$$L_A = -\log_2 \left(\left(\prod_{i=1}^{k} c_i! \right) \Big/ n^{\overline{t}} \right). \tag{4.3}$$

Example 16: Once again we encode the same file, starting with counts $c_a = 1$, $c_b = 1$, and $c_c = 1$.

Symbol	Current Interval		p_a	p_b	p_c
Start	0.000000000	1.000000000	$1/3$	$1/3$	$1/3$
a	0.000000000	0.333333333	$2/4$	$1/4$	$1/4$
b	0.166666667	0.250000000	$2/5$	$2/5$	$1/5$
a	0.166666667	0.200000000	$3/6$	$2/6$	$1/6$
c	0.194444444	0.200000000	$3/7$	$2/7$	$2/7$
b	0.196825397	0.198412698	$3/8$	$3/8$	$2/8$
c	0.198015873	0.198412698	$3/9$	$3/9$	$3/9$
b	0.198148148	0.198280423	$3/10$	$4/10$	$3/10$
a	0.198148148	0.198187831	$4/11$	$4/11$	$3/11$

The final interval is [0.00110 01010 11100 11101, 0.00110 01010 11110 00111] in binary. The theoretical code length is $-\log_2 0.000039683 \approx 14.621$ bits, 5.492 bits longer than in Example 15; the difference is exactly the cost of sending the model that we found in Example 4.1.1. The actual code length is 15 bits, since the final subinterval can be determined from the output **00110 01010 11101**. □

In the following theorem we compare context-free coding using a two-pass method and a one-pass adaptive method. In the two-pass method, the exact symbol counts are encoded after the first pass; during the second pass each symbol's count is decremented whenever it occurs, so at each point the relative counts reflect the correct symbol probabilities for the remainder of the file. In the one-pass adaptive method, all symbols are given initial counts of 1; we add 1 to a symbol's count whenever it occurs.

Theorem 9 *For all input files, the adaptive code with initial 1-weights gives the same code length as the semi-adaptive decrementing code in which the input model is encoded based on the assumption that all symbol distributions are equally likely. In other words,* $L_A = L_{SD} + L_M$.

Proof: Combine Equations (4.1), (4.2), and (4.3), noting that $\binom{t+n-1}{n-1} = n^{\overline{t}}/t!$. □

Cleary and Witten [8] and Bell, Cleary, and Witten [4] present a similar result in a more general setting, showing approximate equality between enumerative codes (which are similar to arithmetic codes) and adaptive codes. Our result applies to a specific method of dealing with the first occurrence of each symbol; we show exact equality in that important special case. Intuitively, the reason for the equality is that in the adaptive code, the cost of "learning" the model is not avoided, but merely spread over the entire file.

4.1.3 Adaptive codes in practice

The simplest adaptive models do not rely on contexts for conditioning probabilities; a symbol's probability is just its relative frequency in the part of the file already coded. The average code length per input symbol of a file encoded using such a zero-order adaptive model is very close to the zero-order entropy of the file. We shall see that adaptive compression can be improved by taking advantage of locality of reference and especially by using higher order models.

4.2 Scaling

One problem with maintaining symbol counts is that the counts can become arbitrarily large, requiring increased precision arithmetic in the coder and more memory to store the counts themselves. By periodically reducing all symbol's counts by the same factor, we can keep the relative frequencies approximately the same while using only a fixed amount of storage for each count. This process is called *scaling*. It allows us to use lower precision arithmetic, possibly hurting compression because of the reduced accuracy of the model. On the other hand, it introduces a *locality of reference* (or *recency*) effect, which often improves compression when the distribution of symbols is variable. We now discuss and quantify the locality effect.

In most text files we find that most of the occurrences of at least some words are clustered in one part of the file. We can take advantage of this locality by assigning more weight to recent occurrences of a symbol in an adaptive model. In practice there are several ways to do this:

- Periodically restarting the model. This often discards too much information to be effective, although Cormack and Horspool find that it gives good results when growing large dynamic Markov models [9].
- Using a sliding window on the text [36]. This requires excessive computational resources.
- Recency rank coding [5,12,66]. This is simple but corresponds to a rather coarse model of recency.
- Exponential aging (giving exponentially increasing weights to successive symbols) [10,46]. This is moderately difficult to implement because of the changing weight increments.
- Periodic scaling [77]. This is simple to implement, fast and effective in operation, and amenable to analysis. It also has the computationally desirable property of keeping the symbol weights small. In effect, scaling is a practical version of exponential aging.

In our discussion of scaling, we assume that a file is divided into *blocks* of length B. Our model assumes that at the end of each block, the weights for all symbols are

multiplied by a scaling factor f, usually $1/2$. Within a block we update the symbol weights by adding 1 for each occurrence.

Example 17: We encode the same file as in Examples 14 through 16. We set the scaling threshold at 10, so we scale only once, just before the last symbol in the file. To retain integer weights without allowing any weight to fall to 0, we round all fractional weights obtained during scaling to the next higher integer.

Symbol	Current Interval		p_a	p_b	p_c
Start	0.000000000	1.000000000	$1/3$	$1/3$	$1/3$
a	0.000000000	0.333333333	$2/4$	$1/4$	$1/4$
b	0.166666667	0.250000000	$2/5$	$2/5$	$1/5$
a	0.166666667	0.200000000	$3/6$	$2/6$	$1/6$
c	0.194444444	0.200000000	$3/7$	$2/7$	$2/7$
b	0.196825397	0.198412698	$3/8$	$3/8$	$2/8$
c	0.198015873	0.198412698	$3/9$	$3/9$	$3/9$
b	0.198148148	0.198280423	$3/10$	$4/10$	$3/10$
Scaling			$2/6$	$2/6$	$2/6$
a	0.198148148	0.198192240	$3/7$	$2/7$	$2/7$

The final interval is $[0.00110\ 01010\ 11100\ 11101, 0.00110\ 01010\ 11110\ 01100]$ in binary. The theoretical code length is $-\log_2 0.000044092 = 14.469$ bits, slightly shorter than in Example 16. The actual code length is 15 bits, since the final subinterval can be determined from the output **00110 01010 11101**. □

4.2.1 Analysis of scaling

In [28] we rigorously analyze arithmetic coding from both the modeling and coding points of view. Coding effects are negligible, as shown in Section 2.1.4. Here we give a mathematical analysis of the modeling effects of periodic scaling, and show that scaling often improves compression and never degrades it by very much.

Notation. We introduce the following additional notation for the analysis of scaling:

$$
\begin{aligned}
s_i &= \text{weight of symbol } a_i \text{ at the start of the block;} \\
c_i &= \text{number of occurrences of symbol } a_i \text{ in the block;} \\
m &= \text{the minimum weight allowed for any symbol, usually 1;} \\
A &= \sum_{i=1}^{n} s_i \quad \text{(the total weight at the start of the block);} \\
B &= \sum_{i=1}^{n} c_i \quad \text{(the size of the block);}
\end{aligned}
$$

$$
\begin{aligned}
C &= A + B \quad \text{(the total weight at the end of the block);} \\
f &= A/C \quad \text{(the scaling factor);} \\
q_i &= s_i/A \quad \text{(probability of symbol } a_i \text{ at the start of the block);} \\
p_i &= c_i/B \quad \text{(probability of symbol } a_i \text{ in the block);} \\
r_i &= (s_i + c_i)/C \quad \text{(probability of symbol } a_i \text{ at the end of the block);} \\
Q, R &= \text{probability distributions } \{q_i\} \text{ and } \{r_i\}; \\
b &= \text{number of blocks resulting from scaling.}
\end{aligned}
$$

When $f = 1/2$, we scale by halving all weights every B symbols, so $A = B$ and $b \approx t/B$. In a typical implementation, $A = B = 8192$.

We assume an adaptive model and exact arithmetic. In our scaling model each symbol's counts are multiplied by f whenever scaling takes place. Thus scaling has the effect of giving more weight to more recent symbols. If we denote the number of occurrences of symbol a_i in block m by $c_{i,m}$, the weight $w_{i,m}$ of symbol a_i after m blocks is given by

$$
\begin{aligned}
w_{i,m} &= c_{i,m} + f w_{i,m-1} \\
&= c_{i,m} + f c_{i,m-1} + f^2 c_{i,m-2} + \ldots .
\end{aligned} \tag{4.4}
$$

The weighted probability $p_{i,m}$ of symbol a_i at the end of block m is then

$$
p_{i,m} = \frac{w_{i,m}}{\sum_{i=1}^{n} w_{i,m}}. \tag{4.5}
$$

We now define a weighted entropy in terms of the weighted probabilities:

Definition 1 The *weighted entropy* of a file at the end of the mth block, denoted by H_m, is the entropy implied by the probability distribution at that time, computed according to the scaling model. That is,

$$
H_m = \sum_{i=1}^{n} -p_{i,m} \log_2 p_{i,m},
$$

where the $p_{i,m}$ are given by Equation (4.5).

We find that l_m, the average code length per symbol for block m, is related to the starting weighted entropy H_{m-1} and the ending weighted entropy H_m in a particularly simple way:

$$
\begin{aligned}
l_m &\approx \frac{1}{1-f} H_m - \frac{f}{1-f} H_{m-1} \\
&= H_m + \frac{f}{1-f}(H_m - H_{m-1}).
\end{aligned}
$$

Letting $f = 1/2$, we obtain

$$
l_m \approx 2H_m - H_{m-1}.
$$

When we multiply by the block length and sum over all blocks, we obtain the following precise and elegant characterization of the code length in terms of weighted entropies:

Theorem 10 *Let L be the code length of a file compressed with arithmetic coding using an adaptive model with scaling. Then*

$$B\left(\left(\sum_{m=1}^{b} H_m\right) + H_b - H_0\right) - t\,\frac{k}{B}$$
$$< L < B\left(\left(\sum_{m=1}^{b} H_m\right) + H_b - H_0\right) + t\left(\frac{k}{B}\log_2\left(\frac{B}{k_{\min}}\right) + O\left(\frac{k^2}{B^2}\right)\right),$$

where $H_0 = \log_2 n$ is the entropy of the initial model, H_m is the (weighted) entropy implied by the scaling model's probability distribution at the end of block m, and $k_{\min}$ is the smallest number of different symbols that occur in any block.

This theorem enables us to compute the code length of a file coded with periodic scaling. To do the calculation we need to know only the symbol counts within each scaling block; we can then use Equations (4.4) and (4.5) and Definition 1. The occurrence of entropy-like terms in the code length is to be expected; the main contribution of Theorem 10 is to show the precise and simple form that the entropy expressions take.

Proof of the upper bound

We prove the upper bound of Theorem 10 by showing first that the code length of a block depends only on the beginning weights and block counts of the symbols, and not on the order of symbols within the block. Then we show that there is an order such that for each symbol certain equalities and inequalities hold, which we use to compute a value and an upper bound for the code length of all occurrences of a single symbol in one block. Finally we sum over all symbols to obtain the worst case code length of a block, and over all blocks to obtain the code length of the file. The proof of the lower bound is similar.

The first lemma enables us to choose any convenient symbol order within a block.

Lemma 1 *The code length of a block depends only on the beginning weights and block counts of the symbols, and not on the order of symbols within the block.*

Proof: The exact code length L of a block,

$$L = -\log_2\left(\left(\prod_{i=1}^{k} {s_i}^{\overline{c_i}}\right) \Big/ A^{\overline{B}}\right),$$

has no dependence on the order of symbols. For each symbol a_i, the adaptive model ensures that the numerators in the set of probabilities used for coding the symbol in the block always form the sequence $\langle s_i, s_i + 1, \ldots, s_i + c_i - 1\rangle$, and that for the block as a whole the denominators always form the sequence $\langle A, A + 1, \ldots, A + B - 1\rangle$. □

In the next lemma we prove the almost obvious fact that there is some order of symbols such that the occurrences of each symbol are roughly evenly distributed through the block. This order will enable us to use a smoothly varying function to estimate the probabilities used for coding the occurrences.

Definition 2 A block of length B containing $c_1, c_2, \ldots, c_k$ occurrences of symbols $a_1, a_2, \ldots, a_k$, respectively, has the *evenly-distributed* (or *ED*) property if for each symbol a_i and for all m, $1 \leq m \leq c_i$, the symbol occurs for the mth time not after position mB/c_i.

Lemma 2 *Every distribution of symbol counts has an order with the ED property.*

Proof: Let $r_i(j)$ be the number of occurrences of symbol a_i required up through the jth position in any ED order. Such an order exists if and only if $\sum_{i=1}^{k} r_i(j) \leq j$ for $1 \leq j \leq B$. We find that $r_i(j) = \lceil \frac{c_i(j+1)}{B} \rceil - 1 < c_i(j+1)/B$. Since $\sum_{i=1}^{k} c_i = B$, $\sum_{i=1}^{k} r_i(j) < j+1$, or $\sum_{i=1}^{k} r_i(j) \leq j$ since $\sum_{i=1}^{k} r_i(j)$ is an integer. This holds for any j, so an ED order exists. □

Next we define a number of related sums and integrals approximating l, the code length of one symbol in one block. For a given symbol with probability $p = c/B$ within the block, we can divide the block into c subblocks each of length $B/c = 1/p$. Then $p_B(k)$ and $p_E(k)$ give the probability that the dynamic model would give to the kth occurrence of the symbol if it occurred precisely at the beginning and end of the kth subblock respectively.

$$p_B(k) = \frac{qA + k}{A + k/p}; \qquad p_E(k) = \frac{qA + k}{A + (k+1)/p}.$$

The expressions S_L and S_U are the lower and upper bounds of the symbol's code length, based on its occurrences being as early or as late as possible in the subblocks.

$$S_L = \sum_{j=0}^{c-1} -\log_2 p_B(j); \qquad S_U = \sum_{j=0}^{c-1} -\log_2 p_E(j).$$

The integrals I_L and I_U approximate S_L and S_U.

$$I_L = \int_0^c -\log_2 p_B(x)\, dx; \qquad I_U = \int_0^c -\log_2 p_E(x)\, dx.$$

We define Δ_I to be $I_U - I_L$. After a considerable amount of algebra, we get

$$\begin{aligned} \Delta_I &= \frac{1}{1-f}\left((c+1-f)\log_2(c+1-f) - c\log_2 c \right. \\ &\qquad \left. -(fc+1-f)\log_2(fc+1-f) + fc\log_2(fc)\right). \end{aligned}$$

The expressions Δ_U and Δ_L are used to bound the error in approximating S_U by I_U and S_L by I_L respectively.

$$\Delta_U = \begin{cases} \log_2(1+\frac{c}{qA}) - \log_2(1+\frac{c}{pA+1}) & \text{if } p > q - 1/A; \\ 0 & \text{if } p \le q - 1/A; \end{cases}$$

$$\Delta_L = \begin{cases} \log_2(1+\frac{c}{pA}) - \log_2(1+\frac{c}{qA}) & \text{if } p < q; \\ 0 & \text{if } p \ge q. \end{cases}$$

We need a simple lemma from integral calculus.

Lemma 3 *If $g(x)$ is monotone increasing, then*

$$\sum_{j=0}^{c-1} g(j) < \int_0^c g(x)\,dx.$$

If $g(x)$ is monotone decreasing, then

$$\sum_{j=0}^{c-1} g(j) < \int_0^c g(x)\,dx - g(c) + g(0).$$

Proof: The increasing case is obvious. In the decreasing case, the integral for any unit interval is greater than the value at the right end of the interval: $\int_j^{j+1} g(x)\,dx > g(j+1)$. We obtain the lemma by adding $g(j) - g(j+1)$ to both sides and summing over j. □

Lemma 4 *The code length l for all occurrences of a single symbol in a block in ED order is less than $I_L + \Delta_I + \Delta_U$.*

Proof: We show that $l < S_U \le I_U + \Delta_U = I_L + \Delta_I + \Delta_U$. Since S_U represents the code length for the symbol if all occurrences of the symbol come as late as possible in an ED order, we have $l < S_U$. If $p < q - 1/A$, then $-\log_2 p_E(x)$ is monotone increasing, so by Lemma 3 we have $S_U < I_U$. If $p > q - 1/A$, then $-\log_2 p_E(x)$ is monotone decreasing, so by Lemma 3 we have $S_U < I_U + \log_2 p_E(c) - \log_2 p_E(0) = I_U + \Delta_U$. If $p = q - 1/A$, then $S_U = I_U$. In all three cases, $S_U \le I_U + \Delta_U$. From the definition of Δ_I, $I_U = I_L + \Delta_I$. □

We now relate I_L to the entropy of the beginning and ending probability distributions. We write $I_L(i)$ to differentiate the values of I_L evaluated for different symbols a_i.

Lemma 5 *Let $L_H = B\left(\frac{1}{1-f}H(R) - \frac{f}{1-f}H(Q)\right)$. Then $L_H = \sum_{i=1}^k I_L(i)$.*

Proof: By appropriate substitutions, we have

$$\sum_{i=1}^k I_L(i) = \sum_{i=1}^k B\left(\frac{1}{1-f}(-r_i \log_2 r_i) - \frac{f}{1-f}(-q_i \log_2 q_i)\right) + \sum_{i=1}^k B\left(\frac{f}{(1-f)^2}(q_i - r_i)\right).$$

The last sum is 0 because Q and R are probability distributions, so $\sum_{i=1}^k q_i = \sum_{i=1}^k r_i = 1$. The lemma follows from the definition of $H(\cdot)$. □

Finally we bound the per-block error.

Lemma 6 *Let L_j be the compressed length of block j. Then*

$$L_j \le B\left(\frac{1}{1-f}H(R) - \frac{f}{1-f}H(Q) + k\log_2\frac{B}{km}\right).$$

Proof: From Lemmas 4 and 5, we have $L_j \le L_H + \sum_{i=1}^{k}(\Delta_I + \Delta_U)$. A scaling factor f of $1/2$ implies that $A = B$. Asymptotics under this condition give

$$\Delta_I + \Delta_U = \begin{cases} 1 - O(1/c) & \text{if } p \le q - 1/A; \\ \log_2(c/s) + O((s+1)/c) & \text{if } p > q - 1/A. \end{cases}$$

The sum $\sum_{i=1}^{k}(\Delta_I + \Delta_U)$ is maximized when as many symbols as possible have large c and small s; the sum's largest possible value cannot exceed its value when $s = m$ and $c = B/k$ for all k symbols, in which case $\Delta_I + \Delta_U = \log_2(B/km) + O((m+1)/c)$. We obtain the result by setting $m = 1$ and summing over the symbols. □

The proof of the upper bound in Theorem 10 follows from Lemma 6 by summing over all blocks. (We are neglecting any special effects of a longer first block or shorter last block.) There is much cancellation because $H(R)$ of one block is $H(Q)$ of the next.

Proof of the lower bound

In the following proof of the lower bound of Theorem 10 we append a prime to the label of each definition and lemma to show the correspondence with the definition and lemmas used in the proof of the upper bound.

Definition 2′ A block of length B containing $c_1, c_2, \ldots, c_k$ occurrences of symbols $a_1, a_2, \ldots, a_k$, respectively, has the *reverse ED* property if for each symbol a_i and for all m, $1 \le m \le c_i$, the symbol occurs for the mth time not before position $(m-1)B/c_i$.

Lemma 2′ *Every distribution of symbol counts has an order with the reverse ED property.*

Proof: By Lemma 2 there is always an order with the ED property. Such an order, when reversed, has the reverse ED property. □

Lemma 3′ *If $g(x)$ is monotone decreasing, then*

$$\sum_{j=0}^{c-1} g(j) > \int_0^c g(x)\,dx.$$

If $g(x)$ is monotone increasing, then

$$\sum_{j=0}^{c-1} g(j) > \int_0^c g(x)\,dx - g(c) + g(0).$$

Proof: Similar to that of Lemma 3. □

Lemma 4′ *The code length l for all occurrences of a single symbol in a block in reverse ED order is greater than $I_L - \Delta_L$.*

Proof: We show that $l > S_L \geq I_L - \Delta_L$. Since I_L represents the code length for the symbol if all occurrences of the symbol come as early as possible in a reverse ED order, we have $l > S_L$. If $p > q$, then $-\log_2 p_B(x)$ is monotone decreasing, so by Lemma 3′ we have $S_L > I_L$. If $p < q$, then $-\log_2 p_B(x)$ is monotone increasing, so by Lemma 3′ we have $S_L > I_L + \log_2 p_B(c) - \log_2 p_B(0) = I_L - \Delta_L$. If $p = q$, then $S_L = I_L$. In all three cases, we have $S_L \geq I_L - \Delta_L$. □

Lemma 6′ *Let L_j be the compressed length of block j. Then*

$$L_j \geq B\left(\frac{1}{1-f}H(R) - \frac{f}{1-f}H(Q)\right) - k.$$

Proof: From Lemmas 4′ and 5, $L_j \geq L_H - \sum_{i=1}^{k} \Delta_L(i)$. Asymptotics when $f = 1/2$ give $\Delta_L = 1 - O(c/s)$, which has maximum value 1, so the sum is at most k. □

The proof of the lower bound in Theorem 10 follows from Lemma 6′ by summing over all blocks, noting that $b = t/B$ and $m = 1$.

Non-scaling corollary

By letting $B = t$, $f = n/(t+n)$, and $m = 1$ in Lemmas 6 and 6′, we obtain the code length without scaling:

Corollary 1 *When we do not scale at all, the code length L_{NS} satisfies:*

$$tH_{\text{final}} + n(H_{\text{final}} - H_0) - k < L_{NS} < tH_{\text{final}} + n(H_{\text{final}} - H_0) + k\log_2(t/k).$$

We can get important insights by contrasting upper bounds in this corollary and Theorem 10. Scaling will bring about a shorter encoding by tracking the block-by-block entropies rather than matching a single entropy for the entire file, but when we forgo scaling the overhead is less, proportional to $\log_2 t$ instead of to t. Scaling will do worst on a homogeneously distributed file, but even then the overhead will increase the code length by only about $(k/B)\log_2(B/k)$ bits per input symbol, less than 0.1 bit per symbol for a typical file. We conclude that the benefits of scaling usually outweigh the minor inefficiencies it sometimes introduces.

	PPMA	PPMB	PPMC	PPMD	PPMP	PPMX
p_{esc}	$\frac{1}{t+1}$	$\frac{k}{t}$	$\frac{k}{t+k}$	$\frac{k/2}{t}$	$\frac{n_1}{t} - \frac{n_2}{t^2} + \cdots$	$\frac{n_1}{t}$
p_i	$\frac{c_i}{t+1}$	$\frac{c_i - 1}{t}$	$\frac{c_i}{t+k}$	$\frac{c_i - 1/2}{t}$		

Table 4.1: PPM escape probabilities (p_{esc}) and symbol probabilities (p_i). The number of symbols that have occurred j times is denoted by n_j.

4.3 Prediction by partial matching

The only way to obtain substantial improvements in compression is to use more sophisticated models. For text files, the increased sophistication invariably takes the form of conditioning the symbol probabilities on contexts consisting of one or more symbols of preceding text. (Langdon [38] and Bell, Witten, Cleary, and Moffat [2,3, 4] have proven that both Ziv-Lempel coding [79,80] and the dynamic Markov coding method of Cormack and Horspool [9] can be reduced to finite context models, despite superficial indications to the contrary.)

One significant difficulty with using high-order models is that many contexts do not occur often enough to provide reliable symbol probability estimates. Cleary and Witten deal with this problem with a technique called *Prediction by Partial Matching* (*PPM*). The PPM idea is to maintain contexts of different lengths up to a fixed maximum order o. To encode a symbol, we check whether the current order-o context has occurred, and if so, whether the new symbol has occurred in that context. If it has, we use arithmetic coding to encode the symbol based on the current symbol counts in the context. Otherwise, we encode a special *escape* symbol (whose probability must be estimated) and repeat the process with progressively shorter contexts until we succeed in encoding the symbol. (In the shorter contexts we may exclude from consideration symbols that have already been rejected at longer contexts.) If a symbol has never occurred in any context, we escape to a special context containing the entire alphabet (including an end-of-file marker), ensuring that every symbol can be encoded.

The symbols are coded using a multi-symbol arithmetic coder. The probabilities passed to the coder are based on symbol frequency counts, periodically scaled down to exploit locality of reference. Cleary and Witten specify two *ad hoc* methods, called PPMA and PPMB, for computing the probability of the escape symbol. Moffat [45] implements the algorithm and proposes a third method, PPMC, for computing the escape probability: he treats the escape event as a separate symbol; when a symbol occurs for the first time he adds 1 to both the escape count and the new symbol's count. In practice, PPMC compresses better than PPMA and PPMB. PPMP and PPMX appear in [76]; they are based on the assumption that the appearance of symbols for the first time in a file is approximately a Poisson process. In Section 4.3.2 we indicate a method, called *PPMD*, that provides improved estimation of the escape probability. See Table 4.1 for formulas for the probabilities used by the different methods.

4.3.1 Application of scaling analysis to higher order models

We now extend our scaling analysis to the PPM method. All of the methods for estimating the probability of the escape symbol give approximately the same compression; PPMB is the most readily analyzed.

In PPMB, in each context the escape event is treated as a separate symbol with its own weight and probability; the first occurrence of an ordinary symbol is not counted and the first two occurrences are coded as escapes. Treating the escape event as a normal symbol, we can apply the above results if we make adjustments for the first two occurrences of each symbol, since in PPMB the code length is independent of the order of the symbols.

In the block in which a given symbol occurs for the first time, we can take the occurrences to be evenly distributed in the sense of Definition 2, with symbol weights (numerators) running from 1 to c and occurrence positions (denominators) running from $A+B/c$ to $A+B$. If this were coded in the normal way, the code length would be bounded above by Lemmas 4 and 5. Since the mechanism of PPMB excludes the last two numerators, $c-1$ and c, and the first two denominators, $A+B/c$ and $A+2B/c$, the code length from the approximation must be adjusted by adding $L_{\text{adjustment}}$:

$$\begin{aligned} L_{\text{adjustment}} &= L_{\text{actual}} - L_{\text{estimated}} \\ &= \log_2 \frac{(c-1)c}{(A+B/c)(A+2B/c)} \\ &= 2\log_2 c - 2\log_2 A + \log_2\left(1-\frac{1}{c}\right) - \log_2\left(1+\frac{1/f-1}{c}\right) \\ &\qquad - \log_2\left(1+2\frac{1/f-1}{c}\right). \end{aligned}$$

We must also adjust the entropy: the actual value of the initial probability q is 0 instead of $1/A$, and the actual value of the final probability r is $(c-1)/(A+B)$ instead of $(c+1)/(A+B)$. For convenience we use the symbol h to denote the entropy term $B\left(\frac{1}{1-f}H(R) - \frac{f}{1-f}H(Q)\right)$. We define and compute the adjustment:

$$\begin{aligned} h_{\text{adjustment}} &= h_{\text{actual}} - h_{\text{estimated}} \\ &= c\log_2\left(1+\frac{2}{c-1}\right) + 2\log_2 c - \log_2 A + 2\log_2 f + \log_2\left(1-\frac{1}{c^2}\right). \end{aligned}$$

The code length is then given by

$$\begin{aligned} L_{\text{actual}} &= h_{\text{actual}} + (L_{\text{adjustment}} - h_{\text{adjustment}}) + \text{small terms} \\ &= h_{\text{actual}} + c\log_2\left(1+\frac{2}{c-1}\right) + \log_2 A + 2\log_2 f + \log_2\left(1+\frac{1}{c}\right) \\ &\qquad + \log_2\left(1+\frac{1/f-1}{c}\right) + \log_2\left(1+2\frac{1/f-1}{c}\right) \\ &\qquad + \text{small terms}. \end{aligned}$$

The net adjustment is negative when $c = 3$, the smallest possible value of c, and increases with c. The largest value occurs when $c \gg A$, that is, when the number of occurrences of a new symbol exceeds the initial weight of all the symbols; in this unlikely event the net adjustment is positive but less than $2 \log_2 c - 3 \log_2 A$.

Now we can extend Theorem 10 to the PPMB model with scaling:

Theorem 11 *When we use PPMB with scaling, the code length L is bounded by*

$$L < \sum_{\text{contexts } X} \left(B \left(\sum_{m=1}^{b_X} H_{X,m} + H_{X,b} - H_{X,0} \right) + O(k_X t_X) \right).$$

Proof: The proof follows from the discussion above, noting that the maximum adjustment for one symbol a_i in one context X is $O(\log_2 c_{X,i})$ and that $\log_2 c_{X,i} < c_{X,i} < t_X$. □

This theorem does not readily estimate the code length of a file in a direct way. However, it does show that the code length of a file coded using a high-order Markov model with scaling can be expressed using the weighted entropy formulation. In particular, the code length for each context is expressed directly in terms of the weighted entropies for that context.

4.3.2 PPMD: an improvement to PPMC

The PPM idea gives excellent compression on text files. In this section we present a slightly improved method for estimating the escape probability, which we call PPMD.

Moffat's PPMC method [45] is widely considered to be the best method of estimating escape probabilities. In PPMC, each symbol's weight in a context is taken to be number of times it has occurred so far in the context. The escape "event," that is, the occurrence of a symbol for the first time in the context, is also treated as a "symbol," with its own count. When a letter occurs for the first time, its weight becomes 1; the escape count is incremented by 1, so the total weight increases by 2. At all other times the total weight increases by 1.

Our new method, called PPMD, is similar to PPMC except that it makes the treatment of new symbols more consistent by adding $\frac{1}{2}$ instead of 1 to both the escape count and the new symbol's count when a new symbol occurs; hence the total weight always increases by 1. We have compared PPMC and PPMD on the Bell-Cleary-Witten corpus [4]. Table 4.2 shows that for text files PPMD compresses consistently about $1\frac{\circ}{\circ}$ better than PPMC. The compression results for PPMC differ from those reported in [4] because of implementation differences; we used versions of PPMC and PPMD that were identical except for the escape probability calculations. Moffat [43] has applied the PPMD modification to his implementation and reports similar results. PPMD has the added advantage of making analysis more tractable by making the code length independent of the appearance order of symbols in the context.

File	PPMC	PPMD	Improvement using PPMD
bib	2.11	2.09	0.02
book1	2.65	2.63	0.02
book2	2.37	2.35	0.02
news	2.91	2.90	0.01
paper1	2.48	2.46	0.02
paper2	2.45	2.42	0.03
progc	2.48	2.47	0.01
progl	1.87	1.85	0.02
progp	1.82	1.80	0.02
trans	1.74	1.72	0.02

Table 4.2: Comparison of PPMC and PPMD on the text files in the Calgary corpus. Compression figures are in bits per input symbol.

4.4 Fast PPM text compression

In [31] we show that we can obtain significantly faster compression with only a small loss of compression efficiency by modifying both the modeling and coding aspects of PPM. The important idea is to concentrate computer resources where they are needed for good compression while using simplifying approximations where they cause only slight degradation of compression performance.

On the modeling side, we eliminate the explicit use of *escape* symbols, we use approximate probability estimation, and we simplify the repeated-symbol-exclusion mechanism. For the coder, we replace the time-consuming arithmetic coding step with various combinations of quasi-arithmetic coding and simple Rice prefix codes.

We observe that the use of arithmetic coding guarantees good compression but runs slowly: the multi-symbol version used in PPMC requires two multiplications and two divisions for each symbol coded, including *escape*s. We also note that for many text files the PPM method predicts very well: the most frequent symbol in the longest available context is usually the one that occurs; this implies that the *escape* mechanism is not needed very often. (This is one reason for the observation by Cleary and Witten that the choice of *escape* probability makes little difference in the amount of compression obtained.) Finally, we recall that arithmetic coding significantly outperforms prefix codes like Huffman coding only when the symbol probabilities are highly skewed.

In the method presented here, we eliminate the *escape* mechanism altogether. First we concatenate the symbol lists of the current contexts of various orders, beginning with the longest. (Of course the concatenation is only conceptual. In practice we simply search through the context's lists, moving to the next list when one is exhausted and stopping when we find the current symbol.) To avoid wasting code space, we exclude all but the first occurrence of repeated symbols using the fast exclusion mechanism

described in Section 4.4.1.

We must identify the current symbol's position within the concatenated list. We choose one of a number of related methods, our choice depending on the speed and compression required. The idea is to use binary quasi-arithmetic coding to encode NOT-FOUND/FOUND decisions for the symbols with highest probability, then if necessary to use a simple prefix code (a Rice code) to encode the symbol's position in the remainder of the list. For maximum speed, we can eliminate the quasi-arithmetic coding step altogether, while for maximum compression we can eliminate the prefix code, using only a series of binary decisions to identify each symbol. Using quasi-arithmetic coding for just the first symbol in the longest context is a good practical choice, as is using quasi-arithmetic coding until the FOUND probability falls below a specified threshold.

Rice codes

Because a quasi-arithmetic coder must encode a number of binary decisions, a text coder that uses quasi-arithmetic coding alone can take about as long as PPMC. By encoding a number of decisions at once, however, we can speed up the coder. Rice codes, described in Section 2.3.2, are eminently suitable for encoding a number of NOT-FOUND decisions followed by a single FOUND decision.

Strictly speaking, Rice codes apply to exponential distributions, but in fact they will give good compression for almost any decaying probability distribution. If we keep our symbol lists ordered by frequency count within each context, the concatenated list used to find a symbol will be in decreasing probability order, except possibly for bumps where the context lists are joined, so we can use Rice coding to encode symbol positions within the concatenated lists. We use the method of Section 2.3.3 for estimating the value of the Rice coding parameter.

4.4.1 Implementation

In this section we describe an implementation of the Fast PPM text compression system, explaining the differences in modeling between our method and the PPMC method.

Data structure for high order models

We use a multiply-linked list structure similar to the vine pointers of Bell *et al.* [4]. In the versions of the Fast PPM system that use Rice coding, we keep the context lists sorted according to frequency count, while in the version that uses only quasi-arithmetic coding we do not reorganize the lists at all.

We delay creating new nodes in order to save time and control the number of nodes present. Every symbol instance appears simultaneously in contexts of all orders from 0 to o, but we do not create nodes for all possible orders. Instead, we create at most one new node for any symbol instance, just one order higher than the one at which

the symbol was found. (If it was found at the highest order, we do not create any new nodes.) This procedure runs somewhat counter to a recommendation of Bell *et al.* [4, pages 149–150], but compression does not appear to suffer greatly. We also use a lazy update rule as in [4], updating statistics only for contexts actually searched. In our implementation we allow the model to grow without bound, never deleting nodes or restarting the model.

Exclusion mechanism

Our exclusion method is faster than the standard approach of maintaining a bit map of alphabet symbols, together with a list of currently excluded symbols to reset the bit map after every symbol. We use a time stamp mechanism, the "clock" being the position of the current symbol in the file. As we pass each symbol for the first time while searching through the concatenated list, we write the current "time" in the symbol's position in a time stamp array. Then if we encounter the symbol again, we check whether its time stamp matches the current time; if so, we can exclude it. We must clear the time stamp array only when the symbol position counter overflows, typically after about $2^{32} \approx 4 \times 10^9$ bytes. When we are using quasi-arithmetic coding for all coding, this mechanism introduces a small inaccuracy in the FOUND/NOT-FOUND probabilities: the NOT-FOUND probabilities will be higher than they should be since they include symbols further down the list that should be excluded. Fortunately the effect is minor.

Coding new symbols

At any point in the coding, the concatenated, duplicate-free context list contains exactly k symbols, where k is the number of distinct alphabet symbols seen so far in the file. If the next symbol has not yet been seen, we encode k NOT-FOUNDs, using quasi-arithmetic coding k times, a Rice code to encode k, or some combination of the two. Then instead of using exclusions and an arithmetic or prefix code, we simply send the bits of the new symbol directly. Doing so greatly simplifies the coding while expanding the compressed file by only $k \log_2 n - (\log_2 n! - \log_2(n-k)!)$ bits, where n is the alphabet size. If $n = 256$ and $k = 100$, the excess code length is about 4 bytes, which is insignificant.

We use a small trick to encode the end of the file without expanding the alphabet: at end-of-file we indicate "new symbol" as just described; then we output the bits of the *first* symbol that occurred in the file. This symbol is always known (unless the input file is empty, in which case we simply make the coded file empty), and will never again be encoded as a new symbol.

Use of Rice coding

The use of Rice codes to encode the symbol positions is straightforward. The only complication is the difficulty of interleaving the quasi-arithmetic code output and the

	Compressed size				Encoding throughput			
	Fast PPM QA	Fast PPM QA/Rice	PPMC	*compress*	Fast PPM QA	Fast PPM QA/Rice	PPMC	*compress*
bib	2.19	2.32	2.12	3.35	23.2	29.0	16.4	111.3
book1	2.51	2.58	2.52	3.46	23.2	30.1	18.5	108.3
book2	2.29	2.41	2.28	3.28	23.5	30.6	18.1	111.1
news	2.78	2.94	2.77	3.86	16.9	23.5	12.6	99.2
paper1	2.62	2.83	2.48	3.77	17.8	24.7	13.6	106.3
paper2	2.51	2.67	2.46	3.52	21.1	26.5	15.2	102.7
progc	2.68	2.92	2.49	3.87	16.9	23.6	12.4	99.0
progl	1.99	2.16	1.87	3.03	24.8	31.7	18.4	119.4
progp	1.96	2.17	1.82	3.11	22.0	31.1	16.5	98.8
trans	1.88	2.09	1.75	3.27	23.7	32.1	18.0	117.1

Table 4.3: Compression and encoding throughput on the ten text files in the Calgary corpus. Compressed sizes are expressed in bits per input symbol. Encoding throughput is expressed in thousands of uncompressed bytes per second.

prefix code output. The bits (or bytes) must be output by the encoder in the order that the decoder will read them. We sidestep the resulting buffering problem by simply using two separate output files.

4.4.2 Experimental Results

We compare the Fast PPM method with PPMC and with the UNIX *compress* program; the results appear in Table 3.6. We show results for two versions of Fast PPM: one that uses quasi-arithmetic coding for all binary decisions (QA) and one that uses quasi-arithmetic coding for one decision in each context, then uses Rice coding if necessary to encode the symbol's position in the remainder of the concatenated context list (QA/Rice). For quasi-arithmetic coding use we $N = 32$ and an order-3 coder; the time needed to precompute the tables is not included, since the tables can be compiled into the coder. The PPMC implementation also uses exclusions and an order 3 model. The test data consists of the 10 text files of the Calgary corpus.

We see that Fast PPM outcompresses the *compress* program on all text files. Fast PPM with quasi-arithmetic coding gives compression performance comparable to that of PPMC, especially for larger files.

We show timing results for encoding on a Sun SPARCstation 1GX; decoding times are similar for the PPM methods. We see that Fast PPM, even using quasi-arithmetic coding alone, is always faster than PPMC; the version that uses some Rice coding is nearly twice as fast as PPMC.

5

CONTRIBUTIONS

THE CONTRIBUTIONS OF THIS WORK are in three areas: statistical coding, modeling for lossless image compression, and modeling for text compression. The modeling and coding ideas in this thesis can be combined to build compression systems either with state-of-the-art compression efficiency or with very high throughput; compression efficiency is only slightly reduced in the latter case. We prove bounds on compression efficiency for a number of important coding methods, as well as presenting practical algorithms that we have implemented.

In the coding area, we have given the first detailed analysis of arithmetic coding. We prove that the compression loss due to coding effects is negligible. We also precisely characterize the code length in a dynamic modeling scenario, incorporating periodic frequency count scaling by way of an elegant new concept called *weighted entropy*. Our analysis extends to high-order models for text compression.

Since arithmetic coding provides nearly optimal compression but tends to run slowly, we are motivated to improve the speed. We describe *quasi-arithmetic coding*, a successful variant of arithmetic coding that retains most of the high compression efficiency of arithmetic coding while running significantly faster. Quasi-arithmetic coding forms a bridge between the mostly continuous world of arithmetic coding and the discrete world of Huffman coding and other prefix codes. We motivate and develop the design of quasi-arithmetic coding, proving bounds on its loss of compression efficiency.

Often the high compression efficiency of arithmetic coding is not required; in such cases we can use the simpler prefix codes for more speed. Golomb and Rice codes form two families of especially simple prefix codes. Both are easy to implement and run very fast. Each family is characterized by a single parameter. We provide a new algorithm for dynamically estimating the value of the parameter; our algorithm gives provably negligible loss of compression efficiency for a stationary source.

We also investigate parallel coding, and give an algorithm for parallel Huffman coding; we give a bound on its running time. We extend this algorithm to parallel quasi-arithmetic coding.

In the area of lossless image compression, we develop a new four-component paradigm, involving pixel sequence, prediction, error modeling, and coding. The explicit

recognition of an error modeling component, distinct from coding, appears to be new. We investigate experimentally the long-held assumption that prediction errors are Laplace-distributed, and we find that often it does not hold. We model the prediction errors more accurately by generalizing the Laplace distribution, and by indirectly estimating the parameters of the generalized distribution by a new technique called the *variability index* technique.

We present and implement two methods for high-efficiency lossless image compression, a raster-scan method called *Prediction by Partial Precision Matching* (*PPPM*) and the *Multi-Level Progressive* (*MLP*) method, a progressive hierarchical method. Both methods give state-of-the-art compression efficiency by using careful prediction and error modeling techniques in conjunction with arithmetic coding.

We also present a new faster lossless image compression method called *FELICS*, for *Fast, Efficient, Lossless Image Compression System*. In FELICS we combine the prediction and error modeling steps to directly provide a simple model to the coder; the coder uses simple prefix codes matched to the model, giving exceptional speed. We lose surprisingly little compression efficiency by using such simple models.

Finally, in the area of text compression, we make some significant enhancements to the Prediction by Partial Matching (PPM) method of Cleary and Witten, recognized as giving the best compression of text compression methods in the literature. We give a small improvement on Moffat's PPMC implementation, consistently obtaining about one percent more compression. We also present a different implementation strategy, designed for maximum speed. By eliminating some unimportant details, using Rice codes (with our parameter estimation technique) instead of arithmetic codes when the precision of arithmetic coding is not needed, and replacing arithmetic coding by quasi-arithmetic coding when we do need the precision, we obtain approximately the same compression as PPMC with nearly twice the speed.

REFERENCES

[1] R. B. Arps, T. K. Truong, D. J. Lu, R. C. Pasco, and T. D. Friedman, "A Multi-Purpose VLSI Chip for Adaptive Data Compression of Bilevel Images," *IBM J. Res. Develop.* 32 (Nov. 1988), 775–795.

[2] T. Bell, "A Unifying Theory and Improvements for Existing Approaches to Text Compression," Univ. of Canterbury, Ph.D. Thesis, 1986.

[3] T. Bell and A. M. Moffat, "A Note on the DMC Data Compression Scheme," *Computer Journal* 32 (1989), 16–20.

[4] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text Compression*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[5] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei, "A Locally Adaptive Data Compression Scheme," *Comm. ACM* 29 (Apr. 1986), 320–330.

[6] D. Chevion, E. D. Karnin, and E. Walach, "High Efficiency, Multiplication Free Approximation of Arithmetic Coding," in *Proc. Data Compression Conference*, J. A. Storer and J. H. Reif, eds., Snowbird, Utah, Apr. 8–11, 1991, 43–52.

[7] J. G. Cleary and I. H. Witten, "Data Compression Using Adaptive Coding and Partial String Matching," *IEEE Trans. Comm.* COM–32 (Apr. 1984), 396–402.

[8] J. G. Cleary and I. H. Witten, "A Comparison of Enumerative and Adaptive Codes," *IEEE Trans. Inform. Theory* IT–30 (Mar. 1984), 306–315.

[9] G. V. Cormack and R. N. Horspool, "Data Compression Using Dynamic Markov Modelling," *Computer Journal* 30 (Dec. 1987), 541–550.

[10] G. V. Cormack and R. N. Horspool, "Algorithms for Adaptive Huffman Codes," *Inform. Process. Lett.* 18 (Mar. 1984), 159–165.

[11] S. De Agostino and J. A. Storer, "Parallel Algorithms for Optimal Compression using Dictionaries with the Prefix Property," in *Proc. Data Compression Conference*, J. A. Storer and M. Cohn, eds., Snowbird, Utah, Mar. 24-26, 1992, 52–61.

[12] P. Elias, "Interval and Recency Rank Source Coding: Two On-line Adaptive Variable Length Schemes," *IEEE Trans. Inform. Theory* IT–33 (Jan. 1987), 3–10.

[13] P. Elias, "Universal Codeword Sets and Representations of Integers," *IEEE Trans. Inform. Theory* IT–21 (Mar. 1975), 194–203.

[14] T. Endoh and Y. Yamakazi, "Progressive Coding Scheme for Multilevel Images," *Picture Coding Symp.* (1986), 21–22.

[15] N. Faller, "An Adaptive System for Data Compression," Record of the 7th Asilomar Conference on Circuits, Systems, and Computers, 1973.

[16] G. Feygin, P. G. Gulak, and P. Chow, "Minimizing Error and VLSI Complexity in the Multiplication Free Approximation of Arithmetic Coding," in *Proc. Data Compression Conference*, J. A. Storer and M. Cohn, eds., Snowbird, Utah, Mar. 30-Apr. 1, 1993, 118–127.

[17] A. S. Fraenkel and S. T. Klein, "Robust Universal Complete Codes as Alternatives to Huffman Codes," Dept. of Applied Mathematics, The Weizmann Institute of Science, Technical Report, Rehovot, Israel, 1985.

[18] P. Fränti, Personal communication, 1993.

[19] R. G. Gallager, "Variations on a Theme by Huffman," *IEEE Trans. Inform. Theory* IT–24 (Nov. 1978), 668–674.

[20] R. G. Gallager and D. C. Van Voorhis, "Optimal Source Codes for Geometrically Distributed Integer Alphabets," *IEEE Trans. Inform. Theory* IT–21 (Mar. 1975), 228–230.

[21] N. Garcia, C. Munoz, and A. Sanz, "Image Compression Based on Hierarchical Coding," *SPIE Image Coding* 594 (1985), 150–157.

[22] S. W. Golomb, "Run-Length Encodings," *IEEE Trans. Inform. Theory* IT–12 (July 1966), 399–401.

[23] M. Guazzo, "A General Minimum-Redundancy Source-Coding Algorithm," *IEEE Trans. Inform. Theory* IT–26 (Jan. 1980), 15–25.

[24] A. Habibi, "Comparison of nth-Order DPCM Encoder With Linear Transformations and Block Quantization Techniques," *IEEE Trans. Comm. Tech.* COM–19 (Dec. 1971), 948–956.

[25] R. W. Hamming, *Coding and Information Theory*, Prentice-Hall, Englewood Cliffs, NJ, 1980.

[26] P. G. Howard and J. S. Vitter, "New Methods for Lossless Image Compression Using Arithmetic Coding," *Information Processing and Management* 28 (1992), 765–779.

[27] P. G. Howard and J. S. Vitter, "Practical Implementations of Arithmetic Coding," in *Image and Text Compression*, J. A. Storer, ed., Kluwer Academic Publishers, Norwell, MA, 1992, 85–112.

[28] P. G. Howard and J. S. Vitter, "Analysis of Arithmetic Coding for Data Compression," *Information Processing and Management* 28 (1992), 749–763.

[29] P. G. Howard and J. S. Vitter, "Error Modeling for Hierarchical Lossless Image Compression," in *Proc. Data Compression Conference*, J. A. Storer and M. Cohn, eds., Snowbird, Utah, Mar. 24-26, 1992, 269–278.

[30] P. G. Howard and J. S. Vitter, "Parallel Lossless Image Compression Using Huffman and Arithmetic Coding," in *Proc. Data Compression Conference*, J. A. Storer and M. Cohn, eds., Snowbird, Utah, Mar. 24-26, 1992, 299–308.

[31] P. G. Howard and J. S. Vitter, "Design and Analysis of Fast Text Compression Based on Quasi-Arithmetic Coding," in *Proc. Data Compression Conference*, J. A. Storer and M. Cohn, eds., Snowbird, Utah, Mar. 30-Apr. 1, 1993, 98–107.

[32] P. G. Howard and J. S. Vitter, "Fast and Efficient Lossless Image Compression," in *Proc. Data Compression Conference*, J. A. Storer and M. Cohn, eds., Snowbird, Utah, Mar. 30-Apr. 1, 1993, 351–360.

[33] D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes," *Proceedings of the Institute of Radio Engineers* 40 (1952), 1098–1101.

[34] A. K. Jain, "Image Data Compression: A Review," *Proc. of the IEEE* 69 (Mar. 1981), 349–389.

[35] K. Knowlton, "Progressive Transmission of Gray-Scale and Binary Pictures by Simple, Efficient, and Lossless Encoding Schemes," *Proc. of the IEEE* 68 (July 1980), 885–896.

[36] D. E. Knuth, "Dynamic Huffman Coding," *J. Algorithms* 6 (June 1985), 163–180.

[37] G. G. Langdon, "Probabilistic and Q-Coder Algorithms for Binary Source Adaptation," in *Proc. Data Compression Conference*, J. A. Storer and J. H. Reif, eds., Snowbird, Utah, Apr. 8–11, 1991, 13–22.

[38] G. G. Langdon, "A Note on the Ziv-Lempel Model for Compressing Individual Sequences," *IEEE Trans. Inform. Theory* IT–29 (Mar. 1983), 284–287.

[39] G. G. Langdon and J. Rissanen, "Compression of Black-White Images with Arithmetic Coding," *IEEE Trans. Comm.* COM–29 (1981), 858–867.

[40] A. Lempel and J. Ziv, "Compression of Two-Dimensional Images," in *Combinatorial Algorithms on Words*, A. Apostolico and Z. Galil, eds., NATO ASI Series #F12, Springer-Verlag, Berlin, 1984, 141–154.

[41] J. L. Mitchell and W. B. Pennebaker, "Optimal Hardware and Software Arithmetic Coding Procedures for the Q-Coder," *IBM J. Res. Develop.* 32 (Nov. 1988), 727–736.

[42] A. M. Moffat, "Predictive Text Compression Based upon the Future Rather than the Past," *Australian Computer Science Communications* 9 (1987), 254–261.

[43] A. M. Moffat, Personal communication, 1992.

[44] A. M. Moffat, "Word-Based Text Compression," *Software–Practice and Experience* 19 (Feb. 1989), 185–198.

[45] A. M. Moffat, "Implementing the PPM Data Compression Scheme," *IEEE Trans. Comm.* COM–38 (Nov. 1990), 1917–1921.

[46] K. Mohiuddin, J. J. Rissanen, and M. Wax, "Adaptive Model for Nonstationary Sources," *IBM Technical Disclosure Bulletin* 28 (Apr. 1986), 4798–4800.

[47] A. N. Netravali and J. O. Limb, "Picture Coding: A Review," *Proc. of the IEEE* 68 (Mar. 1980), 366–406.

[48] J. B. O'Neal, "Predictive Quantizing Differential Pulse Code Modulation for the Transmission of Television Signals," *Bell Syst. Tech. J.* 45 (May–June 1966), 689–721.

[49] R. Pasco, "Source Coding Algorithms for Fast Data Compression," Stanford Univ., Ph.D. Thesis, 1976.

[50] W. B. Pennebaker and J. L. Mitchell, "Probability Estimation for the Q-Coder," *IBM J. Res. Develop.* 32 (Nov. 1988), 737–752.

[51] W. B. Pennebaker and J. L. Mitchell, "Software Implementations of the Q-Coder," *IBM J. Res. Develop.* 32 (Nov. 1988), 753–774.

[52] W. B. Pennebaker, J. L. Mitchell, G. G. Langdon, and R. B. Arps, "An Overview of the Basic Principles of the Q-Coder Adaptive Binary Arithmetic Coder," *IBM J. Res. Develop.* 32 (Nov. 1988), 717–726.

[53] R. F. Rice, "Some Practical Universal Noiseless Coding Techniques," Jet Propulsion Laboratory, JPL Publication 79–22, Pasadena, California, Mar. 1979.

[54] R. F. Rice, "Some Practical Universal Noiseless Coding Techniques—Part II," Jet Propulsion Laboratory, JPL Publication 83–17, Pasadena, California, Mar. 1983.

[55] R. F. Rice, "Some Practical Universal Noiseless Coding Techniques—Part III, Module PSI14,K+," Jet Propulsion Laboratory, JPL Publication 91–3, Pasadena, California, Nov. 1991.

[56] J. Rissanen, "Modeling by Shortest Data Description," *Automatica* 14 (1978), 465–571.

[57] J. Rissanen, "A Universal Prior for Integers and Estimation by Minimum Description Length," *Ann. Statist.* 11 (1983), 416–432.

[58] J. Rissanen, "Universal Coding, Information, Prediction, and Estimation," *IEEE Trans. Inform. Theory* IT–30 (July 1984), 629–636.

[59] J. Rissanen and G. G. Langdon, "Universal Modeling and Coding," *IEEE Trans. Inform. Theory* IT–27 (Jan. 1981), 12–23.

[60] J. J. Rissanen, "Generalized Kraft Inequality and Arithmetic Coding," *IBM J. Res. Develop.* 20 (May 1976), 198–203.

[61] J. J. Rissanen and G. G. Langdon, "Arithmetic Coding," *IBM J. Res. Develop.* 23 (Mar. 1979), 146–162.

[62] J. J. Rissanen and K. M. Mohiuddin, 1987, U. S. Patent 4,652,856, IBM.

[63] J. J. Rissanen and K. M. Mohiuddin, "A Multiplication-Free Multialphabet Arithmetic Code," *IEEE Trans. Comm.* 37 (Feb. 1989), 93–98.

[64] P. Roos, M. A. Viergever, M. C. A. van Dijke, and J. H. Peters, "Reversible Intraframe Compression of Medical Images," *IEEE Trans. Medical Imaging* 7 (Dec. 1988), 328–336.

[65] F. Rubin, "Arithmetic Stream Coding Using Fixed Precision Registers," *IEEE Trans. Inform. Theory* IT–25 (Nov. 1979), 672–675.

[66] B. Y. Ryabko, "Data Compression by Means of a Book Stack," *Problemy Peredachi Informatsii* 16 (1980).

[67] C. E. Shannon, "A Mathematical Theory of Communication," *Bell Syst. Tech. J.* 27 (July 1948), 398–403.

[68] D. Sheinwald, A. Lempel, and J. Ziv, "Two-Dimensional Encoding by Finite State Encoders," *IEEE Transactions on Communications* COM-38 (1990), 341–347.

[69] R. G. Stone, "On Encoding of Commas Between Strings," *Comm. ACM* 22 (May 1979), 310–311.

[70] H. H. Torbey and H. E. Meadows, "System for Lossless Digital Image Compression," *Proc. of SPIE Visual Communication and Image Processing IV* 1199 (Nov. 8-10, 1989), 989–1002.

[71] K. H. Tzou, "Progressive Image Transmission: a Review and Comparsion of Techniques," *Optical Engineering* 26 (July 1987), 581–589.

[72] J. Venbrux, N. Liu, K. Liu, P. Vincent, and R. Merrell, "A Very High Speed Lossless Compression/Decompression Chip Set," Jet Propulsion Laboratory, JPL Publication 91–13, Pasadena, California, July 1991.

[73] J. S. Vitter, "Dynamic Huffman Coding," *ACM Trans. Math. Software* 15 (June 1989), 158–167, also appears as Algorithm 673, Collected Algorithms of ACM, 1989.

[74] J. S. Vitter, "Design and Analysis of Dynamic Huffman Codes," *Journal of the ACM* 34 (Oct. 1987), 825–845.

[75] M. Wang, "Almost Asymptotically Optimal Flag Encoding of the Integers," *IEEE Trans. Inform. Theory* IT–34 (Mar. 1988), 324–326.

[76] I. H. Witten and T. C. Bell, "The Zero Frequency Problem: Estimating the Probabilities of Novel Events in Adaptive Text Compression," *IEEE Trans. Inform. Theory* IT–37 (July 1991), 1085–1094.

[77] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic Coding for Data Compression," *Comm. ACM* 30 (June 1987), 520–540.

[78] P.-S. Yeh, R. F. Rice, and W. Miller, "On the Optimality of Code Options for a Universal Noiseless Coder," Jet Propulsion Laboratory, JPL Publication 91–2, Pasadena, California, Feb. 1991.

[79] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Trans. Inform. Theory* IT–23 (May 1977), 337–343.

[80] J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable Rate Coding," *IEEE Trans. Inform. Theory* IT–24 (Sept. 1978), 530–536.

The text of this report is set in Computer Modern 12/14. The chapter titles are 14-point Computer Modern Dunhill, and the chapter numbers are 20-point Computer Modern Dunhill. The versals at the start of each chapter are 36-point Computer Duerer Roman.