Edgar H. Sibley
Panel Editor

The state of the art in data compression is arithmetic coding, not the better-known Huffman method. Arithmetic coding gives greater compression, is faster for adaptive models, and clearly separates the model from the channel encoding.

ARITHMETIC CODING FOR DATA COMPRESSION

IAN H. WITTEN, RADFORD M. NEAL, and JOHN G. CLEARY

Arithmetic coding is superior in most respects to the better-known Huffman [10] method. It represents information at least as compactly—sometimes considerably more so. Its performance is optimal without the need for blocking of input data. It encourages a clear separation between the model for representing data and the encoding of information with respect to that model. It accommodates adaptive models easily and is computationally efficient. Yet many authors and practitioners seem unaware of the technique. Indeed there is a widespread belief that Huffman coding cannot be improved upon.

We aim to rectify this situation by presenting an accessible implementation of arithmetic coding and by detailing its performance characteristics. We start by briefly reviewing basic concepts of data compression and introducing the model-based approach that underlies most modern techniques. We then outline the idea of arithmetic coding using a simple example, before presenting programs for both encoding and decoding. In these programs the model occupies a separate module so that different models can easily be used. Next we discuss the construction of fixed and adaptive models and detail the compression efficiency and execution time of the programs, including the effect of different arithmetic word lengths on compression efficiency. Finally, we outline a few applications where arithmetic coding is appropriate.

Financial support for this work has been provided by the Natural Sciences and Engineering Research Council of Canada.

UNIX is a registered trademark of AT&T Bell Laboratories.

DATA COMPRESSION

To many, data compression conjures up an assortment of ad hoc techniques such as conversion of spaces in text to tabs, creation of special codes for common words, or run-length coding of picture data (e.g., see [8]). This contrasts with the more modern model-based paradigm for coding, where, from an *input string* of symbols and a *model*, an *encoded string* is produced that is (usually) a compressed version of the input. The decoder, which must have access to the same model, regenerates the exact input string from the encoded string. Input symbols are drawn from some well-defined set such as the ASCII or binary alphabets; the encoded string is a plain sequence of bits. The model is a way of calculating, in any given context, the distribution of probabilities for the next input symbol. It must be possible for the decoder to produce exactly the same probability distribution in the same context. Compression is achieved by transmitting the more probable symbols in fewer bits than the less probable ones.

For example, the model may assign a predetermined probability to each symbol in the ASCII alphabet. No context is involved. These probabilities can be determined by counting frequencies in representative samples of text to be transmitted. Such a *fixed* model is communicated in advance to both encoder and decoder, after which it is used for many messages.

Alternatively, the probabilities that an *adaptive* model assigns may change as each symbol is transmitted, based on the symbol frequencies seen so far in the message. There is no need for a representative

sample of text, because each message is treated as an independent unit, starting from scratch. The encoder's model changes with each symbol transmitted, and the decoder's changes with each symbol received, in sympathy.

More complex models can provide more accurate probabilistic predictions and hence achieve greater compression. For example, several characters of previous context could condition the next-symbol probability. Such methods have enabled mixed-case English text to be encoded in around 2.2 bits/character with two quite different kinds of model [4, 6]. Techniques that do not separate modeling from coding so distinctly, like that of Ziv and Lempel [23], do not seem to show such great potential for compression, although they may be appropriate when the aim is raw speed rather than compression performance [22].

The effectiveness of any model can be measured by the entropy of the message with respect to it, usually expressed in bits/symbol. Shannon's fundamental theorem of coding states that, given messages randomly generated from a model, it is impossible to encode them into less bits (on average) than the entropy of that model [21].

A message can be coded with respect to a model using either Huffman or arithmetic coding. The former method is frequently advocated as the best possible technique for reducing the encoded data rate. It is not. Given that each symbol in the alphabet must translate into an integral number of bits in the encoding, Huffman coding indeed achieves "minimum redundancy." In other words, it performs optimally if all symbol probabilities are integral powers of ½. But this is not normally the case in practice; indeed, Huffman coding can take up to one extra bit per symbol. The worst case is realized by a source in which one symbol has probability approaching unity. Symbols emanating from such a source convey negligible information on average, but require at least one bit to transmit [7]. Arithmetic coding dispenses with the restriction that each symbol must translate into an integral number of bits, thereby coding more efficiently. It actually achieves the theoretical entropy bound to compression efficiency for any source, including the one just mentioned.

In general, sophisticated models expose the deficiencies of Huffman coding more starkly than simple ones. This is because they more often predict symbols with probabilities close to one, the worst case for Huffman coding. For example, the techniques mentioned above that code English text in 2.2 bits/character both use arithmetic coding as the final step, and performance would be impacted severely if Huffman coding were substituted. Nevertheless, since our topic is coding and not modeling, the illustrations in this article all employ simple models. Even so, as we shall see, Huffman coding is inferior to arithmetic coding.

The basic concept of arithmetic coding can be traced back to Elias in the early 1960s (see [1, pp. 61–62]). Practical techniques were first introduced by Rissanen [16] and Pasco [15], and developed further by Rissanen [17]. Details of the implementation presented here have not appeared in the literature before; Rubin [20] is closest to our approach. The reader interested in the broader class of arithmetic codes is referred to [18]; a tutorial is available in [13]. Despite these publications, the method is not widely known. A number of recent books and papers on data compression mention it only in passing, or not at all.

THE IDEA OF ARITHMETIC CODING

In arithmetic coding, a message is represented by an interval of real numbers between 0 and 1. As the message becomes longer, the interval needed to represent it becomes smaller, and the number of bits needed to specify that interval grows. Successive symbols of the message reduce the size of the interval in accordance with the symbol probabilities generated by the model. The more likely symbols reduce the range by less than the unlikely symbols and hence add fewer bits to the message.

Before anything is transmitted, the range for the message is the entire interval [0, 1), denoting the half-open interval $0 \leq x < 1$. As each symbol is processed, the range is narrowed to that portion of it allocated to the symbol. For example, suppose the alphabet is {*a*, *e*, *i*, *o*, *u*, *!*}, and a fixed model is used with probabilities shown in Table I. Imagine transmitting the message *eaii!*. Initially, both encoder and decoder know that the range is [0, 1). After seeing the first symbol, *e*, the encoder narrows it to [0.2, 0.5), the range the model allocates to this symbol. The second symbol, *a*, will narrow this new range to the first one-fifth of it, since *a* has been

TABLE I. Example Fixed Model for Alphabet {*a*, *e*, *i*, *o*, *u*, *!*}

Symbol	Probability	Range
a	.2	[0, 0.2)
e	.3	[0.2, 0.5)
i	.1	[0.5, 0.6)
o	.2	[0.6, 0.8)
u	.1	[0.8, 0.9)
!	.1	[0.9, 1.0)

allocated [0, 0.2). This produces [0.2, 0.26), since the previous range was 0.3 units long and one-fifth of that is 0.06. The next symbol, *i*, is allocated [0.5, 0.6), which when applied to [0.2, 0.26) gives the smaller range [0.23, 0.236). Proceeding in this way, the encoded message builds up as follows:

Initially	[0,	1)
After seeing *e*	[0.2,	0.5)
a	[0.2,	0.26)
i	[0.23,	0.236)
i	[0.233,	0.2336)
!	[0.23354,	0.2336)

Figure 1 shows another representation of the encoding process. The vertical bars with ticks represent the symbol probabilities stipulated by the model. After the first symbol has been processed, the model is scaled into the range [0.2, 0.5), as shown in Figure 1a. The second symbol scales it again into the range [0.2, 0.26). But the picture cannot be continued in this way without a magnifying glass! Consequently, Figure 1b shows the ranges expanded to full height at every stage and marked with a scale that gives the endpoints as numbers.

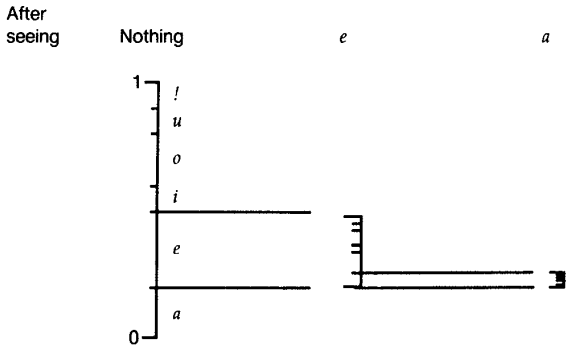

FIGURE 1a. Representation of the Arithmetic Coding Process

Suppose all the decoder knows about the message is the final range, [0.23354, 0.2336). It can immediately deduce that the first character was *e*, since the range lies entirely within the space the model of Table I allocates for *e*. Now it can simulate the operation of the *en*coder:

Initially	[0, 1)
After seeing *e*	[0.2, 0.5)

This makes it clear that the second character is *a*, since this will produce the range

After seeing *a* [0.2, 0.26),

which entirely encloses the given range [0.23354, 0.2336). Proceeding like this, the decoder can identify the whole message.

It is not really necessary for the decoder to know both ends of the range produced by the encoder. Instead, a single number within the range—for example, 0.23355—will suffice. (Other numbers, like 0.23354, 0.23357, or even 0.23354321, would do just as well.) However, the decoder will face the problem of detecting the end of the message, to determine when to stop decoding. After all, the single number 0.0 could represent any of *a*, *aa*, *aaa*, *aaaa*, ... To resolve the ambiguity, we ensure that each message ends with a special EOF symbol known to both encoder and decoder. For the alphabet of Table I, *!* will be used to terminate messages, and only to termi-

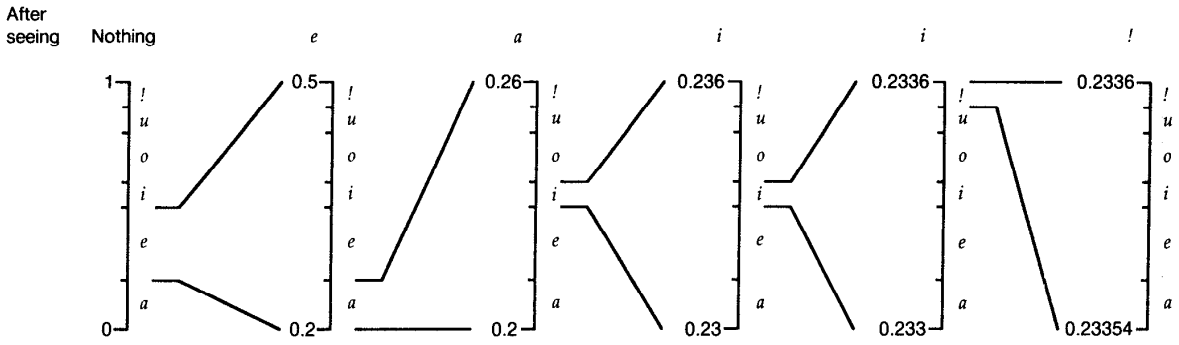

FIGURE 1b. Representation of the Arithmetic Coding Process with the Interval Scaled Up at Each Stage

```
/* ARITHMETIC ENCODING ALGORITHM. */

/* Call encode_symbol repeatedly for each symbol in the message.          */
/* Ensure that a distinguished "terminator" symbol is encoded last, then  */
/* transmit any value in the range [low, high).                           */

encode_symbol(symbol, cum_freq)
    range = high - low
    high  = low + range*cum_freq[symbol-1]
    low   = low + range*cum_freq[symbol]


/* ARITHMETIC DECODING ALGORITHM. */

/* "Value" is the number that has been received.                          */
/* Continue calling decode_symbol until the terminator symbol is returned. */

decode_symbol(cum_freq)
    find symbol such that
        cum_freq[symbol] <= (value-low)/(high-low) < cum_freq[symbol-1]
                          /* This ensures that value lies within the new  */
                          /* [low, high) range that will be calculated by */
                          /* the following lines of code.                 */

    range = high - low
    high  = low + range*cum_freq[symbol-1]
    low   = low + range*cum_freq[symbol]
    return symbol
```

FIGURE 2. Pseudocode for the Encoding and Decoding Procedures

nate messages. When the decoder sees this symbol, it stops decoding.

Relative to the fixed model of Table I, the entropy of the five-symbol message *eaii!* is

$$-\log 0.3 - \log 0.2 - \log 0.1 - \log 0.1 - \log 0.1$$

$$= -\log 0.00006 \approx 4.22$$

(using base 10, since the above encoding was performed in decimal). This explains why it takes five decimal digits to encode the message. In fact, the size of the final range is $0.2336 - 0.23354 = 0.00006$, and the entropy is the negative logarithm of this figure. Of course, we normally work in binary, transmitting binary digits and measuring entropy in bits.

Five decimal digits seems a lot to encode a message comprising four vowels! It is perhaps unfortunate that our example ended up by expanding rather than compressing. Needless to say, however, different models will give different entropies. The best single-character model of the message *eaii!* is the set of symbol frequencies {*e*(0.2), *a*(0.2), *i*(0.4), *!*(0.2)}, which gives an entropy of 2.89 decimal digits. Using this model the encoding would be only three digits long. Moreover, as noted earlier, more sophisticated models give much better performance in general.

A PROGRAM FOR ARITHMETIC CODING

Figure 2 shows a pseudocode fragment that summarizes the encoding and decoding procedures developed in the last section. Symbols are numbered, 1, 2, 3, ... The frequency range for the ith symbol is from $cum\_freq[i]$ to $cum\_freq[i-1]$. As i decreases, $cum\_freq[i]$ increases, and $cum\_freq[0] = 1$. (The reason for this "backwards" convention is that $cum\_freq[0]$ will later contain a normalizing factor, and it will be convenient to have it begin the array.) The "current interval" is [*low*, *high*), and for both encoding and decoding, this should be initialized to [0, 1).

Unfortunately, Figure 2 is overly simplistic. In practice, there are several factors that complicate both encoding and decoding:

Incremental transmission and reception. The encode algorithm as described does not transmit anything until the entire message has been encoded; neither does the decode algorithm begin decoding until it has received the complete transmission. In most applications an incremental mode of operation is necessary.

The desire to use integer arithmetic. The precision required to represent the [*low*, *high*) interval grows with the length of the message. Incremental operation will help overcome this, but the potential for

overflow and underflow must still be examined carefully.

Representing the model so that it can be consulted efficiently. The representation used for the model should minimize the time required for the decode algorithm to identify the next symbol. Moreover, an adaptive model should be organized to minimize the time-consuming task of maintaining cumulative frequencies.

Figure 3 shows working code, in C, for arithmetic encoding and decoding. It is considerably more detailed than the bare-bones sketch of Figure 2! Implementations of two different models are given in Figure 4; the Figure 3 code can use either one.

The remainder of this section examines the code of Figure 3 more closely, and includes a proof that decoding is still correct in the integer implementation and a review of constraints on word lengths in the program.

arithmetic_coding.h

```
1   /* DECLARATIONS USED FOR ARITHMETIC ENCODING AND DECODING */
2
3
4   /* SIZE OF ARITHMETIC CODE VALUES. */
5
6   #define Code_value_bits 16              /* Number of bits in a code value   */
7   typedef long code_value;                /* Type of an arithmetic code value */
8
9   #define Top_value (((long)1<<Code_value_bits)-1)      /* Largest code value */
10
11
12  /* HALF AND QUARTER POINTS IN THE CODE VALUE RANGE. */
13
14  #define First_qtr (Top_value/4+1)       /* Point after first quarter        */
15  #define Half      (2*First_qtr)         /* Point after first half           */
16  #define Third_qtr (3*First_qtr)         /* Point after third quarter        */
```

model.h

```
17  /* INTERFACE TO THE MODEL. */
18
19
20  /* THE SET OF SYMBOLS THAT MAY BE ENCODED. */
21
22  #define No_of_chars 256                 /* Number of character symbols      */
23  #define EOF_symbol (No_of_chars+1)      /* Index of EOF symbol              */
24
25  #define No_of_symbols (No_of_chars+1)   /* Total number of symbols          */
26
27
28  /* TRANSLATION TABLES BETWEEN CHARACTERS AND SYMBOL INDEXES. */
29
30  int char_to_index[No_of_chars];         /* To index from character          */
31  unsigned char index_to_char[No_of_symbols+1]; /* To character from index    */
32
33
34  /* CUMULATIVE FREQUENCY TABLE. */
35
36  #define Max_frequency 16383             /* Maximum allowed frequency count  */
37                                          /*   2^14 - 1                       */
38  int cum_freq[No_of_symbols+1];          /* Cumulative symbol frequencies    */
```

FIGURE 3. C Implementation of Arithmetic Encoding and Decoding

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.