Glen G. Langdon, Jr.

An Introduction to Arithmetic Coding

Arithmetic coding is a data compression technique that encodes data (the data string) by creating a code string which represents a fractional value on the number line between 0 and 1. The coding algorithm is symbolwise recursive; i.e., it operates upon and encodes (decodes) one data symbol per iteration or recursion. On each recursion, the algorithm successively partitions an interval of the number line between 0 and 1, and retains one of the partitions as the new interval. Thus, the algorithm successively deals with smaller intervals, and the code string, viewed as a magnitude, lies in each of the nested intervals. The data string is recovered by using magnitude comparisons on the code string to recreate how the encoder must have successively partitioned and retained each nested subinterval. Arithmetic coding differs considerably from the more familiar compression coding techniques, such as prefix (Huffman) codes. Also, it should not be confused with error control coding, whose object is to detect and correct errors in computer operations. This paper presents the key notions of arithmetic compression coding by means of simple examples.

1. Introduction

Arithmetic coding maps a string of data (source) symbols to a code string in such a way that the original data can be recovered from the code string. The encoding and decoding algorithms perform *arithmetic operations* on the code string. One recursion of the algorithm handles one data symbol. Arithmetic coding is actually a *family* of codes which share the property of treating the code string as a magnitude. For a brief history of the development of arithmetic coding, refer to Appendix 1.

Compression systems

The notion of compression systems captures the idea that data may be transformed into something which is encoded, then transmitted to a destination, then transformed back into the original data. Any data compression approach, whether employing arithmetic coding, Huffman codes, or any other coding technique, has a *model* which makes some assumptions about the data and the *events* encoded.

The code itself can be independent of the model. Some systems which compress waveforms (e.g., digitized speech) may predict the next value and encode the error. In this model the error and not the actual data is encoded. Typically, at the encoder side of a compression system, the data to be compressed feed a model unit. The model determines 1) the event(s) to be encoded, and 2) the estimate of the relative frequency (probability) of the events. The encoder accepts the event and some indication of its relative frequency and generates the code string.

A simple model is the memoryless model, where the data symbols themselves are encoded according to a single code. Another model is the first-order Markov model, which uses the previous symbol as the *context* for the current symbol. Consider, for example, compressing English sentences. If the data symbol (in this case, a letter) "q" is the previous letter, we would expect the next letter to be "u." The first-order Markov model is a *dependent* model; we have a different expectation for each symbol (or in the example, each letter), depending on the context. The context is, in a sense, a state governed by the past sequence of symbols. The purpose of a context is to provide a probability distribution, or *statistics*, for encoding (decoding) the next symbol.

Corresponding to the symbols are statistics. To simplify the discussion, consider a single-context model, i.e., the memoryless model. Data compression results from encoding the more-frequent symbols with short code-string length increases, and encoding the less-frequent events with long code length increases. Let c_i denote the occurrences of the ith symbol in a data string. For the memoryless model and a given code, let l_i denote the length (in bits) of the code-string increase associated

Table 1 Example Huffman code.

Symbol	*Codeword*	*Probability p* (in binary)	*Cumulative probability P*
a	0	.100	.000
b	10	.010	.100
c	110	.001	.110
d	111	.001	.111

with symbol i. The code-string length corresponding to the data string is obtained by replacing each data symbol with its associated length and summing the lengths:

$$\sum_i c_i \ell_i .$$

If ℓ_i is large for data symbols of high relative frequency (large values of c_i), the given code will almost surely fail to achieve compression. The wrong statistics (a popular symbol with a long length ℓ) lead to a code string which may have more bits than the original data. For compression it is imperative to closely approximate the relative frequency of the more-frequent events. Denote the relative frequency of symbol i as p_i where $p_i = c_i/N$, and N is the total number of symbols in the data string. If we use a fixed frequency for each data symbol value, the best we can compress (according to our given model) is to assign length ℓ_i as $-\log p_i$. Here, logarithms are taken to the base 2 and the unit of length is the bit. Knowing the ideal length for each symbol, we calculate the *ideal code length* for the given data string and memoryless model by replacing each instance of symbol i in the data string by length value $-\log p_i$, and summing the lengths.

Let us now review the components of a compression system: the model structure for contexts and events, the statistics unit for estimation of the event statistics, and the encoder.

Model structure

In practice, the model is a finite-state machine which operates successively on each data symbol and determines the current event to be encoded and its context (i.e., which relative frequency distribution applies to the current event). Often, each event is the data symbol itself, but the structure can define other events from which the data string could be reconstructed. For example, one could define an event such as the run length of a succession of repeated symbols, i.e., the number of times the current symbol repeats itself.

Statistics estimation

The estimation method computes the relative frequency distribution used for each context. The computation may be performed beforehand, or may be performed during the encoding process, typically by a counting technique. For Huffman codes, the event statistics are predetermined by the length of the event's codeword.

Encoder

The encoder accepts the events to be encoded and generates the code string.

The notions of model structure and statistics are important because they completely determine the available compression. Consider applications where the compression model is complex, i.e., has several contexts and a need to adapt to the data string statistics. Due to the flexibility of arithmetic coding, for such applications the "compression problem" is equivalent to the "modeling problem."

• *Desirable properties of a coding method*

We now list some properties for which arithmetic coding is amply suited.

For most applications, we desire the *first-in first-out* (FIFO) property: Events are decoded in the same order as they are encoded. FIFO coding allows for adapting to the statistics of the data string. With *last-in first-out* (LIFO) coding, the last event encoded is the first event decoded, so adapting is difficult.

We desire no more than a small storage buffer at the encoder. Once events are encoded, we do not want the encoding of subsequent events to alter what has already been generated.

The encoding algorithm should be capable of accepting successive events from different probability distributions. Arithmetic coding has this capability. Moreover, the code acts directly on the probabilities, and can adapt "on the fly" to changing statistics. Traditional Huffman codes require the design of a different codeword set for different statistics.

• *An initial view of Huffman and arithmetic codes*

We progress to a very simple arithmetic code by first using a prefix (Huffman) code as an example. Our purpose is to introduce the basic notions of arithmetic codes in a very simple setting.

Consider a four-symbol alphabet, for which the relative frequencies $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, and $\frac{1}{8}$ call for respective codeword lengths of 1, 2, 3, and 3. Let us order the alphabet $\{a, b, c, d\}$ according to relative frequency, and use the code of **Table 1.** The probability column has the binary fraction associated with the probability corresponding to the assigned length.

The encoding for the data string "$a\ a\ b\ c$" is 0.0.10.110, where "·" is used as a delimiter to show the substitution of the codeword for the symbol. The code also has the *prefix* property (no codeword is the prefix of another). Decoding is performed by a matching or comparison process starting with the first bit of the code string. For decoding code string

0010110, the first symbol is decoded as "a" (the only codeword beginning with 0). We remove codeword 0, and the remaining code is *010110*. The second symbol is similarly decoded as "a," leaving 10110. For string 10110, the only codeword beginning with 10 is "b," so we are left with 110 for c.

We have described Table 1 in terms of Huffman coding. We now present an arithmetic coding view, with the aid of **Figure 1.** We relate arithmetic coding to the process of subdividing the unit interval, and we make two points:

Point 1 Each codeword (code point) is the sum of the probabilities of the preceding symbols.

Point 2 The width or size of the subinterval to the right of each code point corresponds to the probability of the symbol.

We have purposely arranged Table 1 with the symbols ordered according to their probability p, and the codewords are assigned in numerical order. We now view the codewords as binary fractional values (*.000*, *.100*, *.110* and *.111*). We assume that the reader is familiar with binary fractions, i.e., that $\frac{1}{2}$, $\frac{1}{4}$, and $\frac{7}{8}$ are respectively represented as .1, .01, and .111 in the binary number system. Notice from the construction of Table 1, and referring to the previously stated Point 1, that each codeword is the sum of the probabilities of the preceding symbols. In other words, each codeword is a cumulative probability P.

Now we view the codewords as points (or code points) on the number line from 0 to 1, or the unit interval, as shown in Fig. 1. The four code points divide the unit interval into four subintervals. We identify each subinterval with the symbol corresponding to its leftmost point. For example, the interval for symbol "a" goes from 0 to .1, and for symbol "d" goes from .111 to 1.0. Note also from the construction of Table 1, and referring to the previous Point 2, that the width or size of the subinterval to the right of each code point corresponds to the probability of the symbol. The codeword for symbol "a" has $\frac{1}{2}$ the interval, the codeword for "b" (.100) has $\frac{1}{4}$ the interval, and "c" (.110) and "d" (.111) each have $\frac{1}{8}$ of the interval.

In the example data, the first symbol is "a," and the corresponding interval on the unit interval is [0,.1). The notation "[0,.1)" means that 0 is included in the interval, and that fractions equal to or greater than 0 but less than .1 are in the interval. The interval for symbol "b" is [.1,.110). Note that .1 belongs to the interval for "b" and not for "a." Thus, code strings generated by Table 1 which correspond to continuations of data strings beginning with "a," when viewed as a fraction, *never* equal or exceed the value 0.1. Data string "$a\ d\,d\,d\,d\,d\cdots$" is encoded as "0111111$\cdots$," which when viewed as a fraction approaches but never reaches value

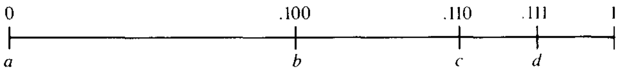


Figure 1 Codewords of Table 1 as points on unit interval.

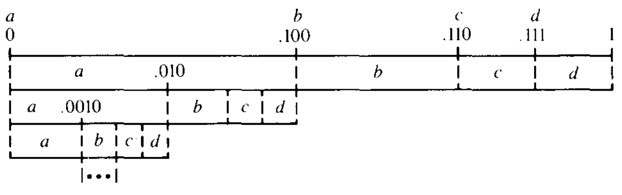


Figure 2 Successive subdivision of unit interval for code of Table 1 and data string "$a\ a\ b\cdots$."

.100000$\cdots$. We can decode by magnitude comparison; if the code string is less than 0.1, the first data symbol must have been "a."

Figure 2 shows how the encoding process continues. Once "a" has been encoded to [0,.1), we next subdivide this interval into the same proportions as the original unit interval. Thus, the subinterval assigned to the second "a" is [0,.01). For the third symbol, we subdivide [0,.01), and the subinterval belonging to third symbol "b" is [.001,.0011). Note that each of the two leading 0s in the binary representation of this interval comes from the codewords (Table 1) of the two symbols "a" which precede the "b." For the fourth symbol "c," which follows "$a\ \ a\ \ b$," the corresponding subinterval is [.0010110,.0010111).

In arithmetic coding we treat the code points, which delimit the interval partitions, as magnitudes. To define the interval, we specify 1) the leftmost point C, and 2) the interval width A. (Alternatively one can define the interval by the leftmost point and rightmost point, or by defining the rightmost point and the available width.) Width A is available for further partitioning.

We now present some mathematics to describe what is happening pictorially in Fig. 2. From Fig. 2 and its description, we see that there are two recursive operations needed to define the current interval. For encoding, the recursion begins with the "current" values of code point C and available width A, and uses the value of the symbol encoded to determine "new" values of code point C and width A. At the end of the current recursion, and before the next recursion, the "new" values of code point and width become the "current" values.

New code point

The new leftmost point of the new interval is the sum of the current code point C, and the product of the interval width

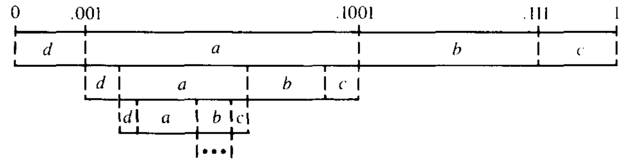


Figure 3 Subdivision of unit interval for arithmetic code of Table 2 and data string "*a a b*· · · ."

Table 2 Arithmetic code example.

Symbol	*Cumulative probability P*	*Symbol probability p*	*Length*
d	.000	.001	3
b	.001	.010	2
a	.011	.100	1
c	.111	.001	3

W of the current interval and the cumulative probability P_i for the symbol i being encoded:

New C = Current $C + (A \times P_i)$.

For example, after encoding "*a a*," the current code point C is 0 and the interval width A is .01. For "*a a b*," the new code point is .001, determined as 0 (current code point C), plus the product (.01) × (.100). The factor on the left is the width A of the current interval, and the factor on the right is the cumulative probability P for symbol "*b*"; see the "Cumulative probability" column of Table 1.

New interval width A
The width A of the current interval is the product of the probabilities of the data symbols encoded so far. Thus, the new interval width is

New A = Current $A \times P_i$,

where the current symbol is i. For example, after encoding "*a a b*," the interval width is (.1) × (.1) × (.01), which is .0001.

In summary, we can systematically calculate the next interval from the leftmost point C and width A of the current interval, given the probability p and cumulative probability P values in Table 1 for the symbol to be encoded. The two operations (new code point and new width) thus form a *double recursion.* This double recursion is central to arithmetic coding, and this particular version is characteristic of the class of FIFO arithmetic codes which use the symbol probabilities directly.

The Huffman code of Table 1 corresponds to a special integer-length arithmetic code. With arithmetic codes we can rearrange the symbols and forsake the notion of a k-bit codeword for a symbol corresponding to a probability of 2^{-k}. We retain the important technique of the double recursion. Consider the arrangement of **Table 2.** The "codeword" corresponds to the cumulative probability P of the preceding symbols in the ordering.

The subdivision of the unit interval for Table 2, and for the data string "*a a b*," is shown in **Figure 3.** In this example, we retain Points 1 and 2 of the previous example, but no longer have the prefix property of Huffman codes. Compare Figs. 2 and 3 to see that the interval widths are the same but the locations of the intervals have been changed in Fig. 3 to conform with the new ordering in Table 2.

Let us again code the string "*a a b c*." This example reinforces the double recursion operations, where the new values become the current values for the next recursion. It is helpful to understand the arithmetic provided here, using the "picture" of Fig. 3 for motivation.

The first "*a*" symbol yields the code point .011 and interval [.011,.111), as follows:

First symbol (*a*)
  C: New code point $C = 0 + 1 \times (.011) = .011$.
    (Current code point plus current width A times P.)
  A: New interval width $A = 1 \times (.1) = .1$.
    (Current width A times probability p.)
In the arithmetic coding literature, we have called the value $A \times P$ added to the old code point C, the *augend.*

The second "*a*" identifies subinterval [.1001,.1101).

Second symbol (*a*)
  C: New code point = .011 + .1 × (.011) =
    .011 (current code point)
    .0011 (current width A times P, or augend)
    ―――
    .1001. (new code point)
  A: New interval width $A = .1 \times (.1) = .01$.
    (Current width A times probability p.)
Now the remaining interval is one-fourth the width of the unit interval.

For the third symbol, "*b*," we repeat the recursion.

Third symbol (*b*)
  C: New code point = .1001 + .01 × (.001) = .10011.
    .1001 (current code point C)
    .00001 (current width A times P, or augend)
    ―――
    .10011 (new code point)
  A: New interval width $A = .01 \times (.01) = .0001$.
    (Current width A times probability p.)
Thus, following the coding of "*a a b*," the interval is [.10011,.10101).

To handle the fourth letter, "*c*," we continue as follows.

Fourth symbol (*c*)

C: New code point = .10011 + .0001 × (.111) = .1010011.

```
.10011     (current code point)
.0000111   (current width A times P, or augend)
_______
.1010011   (new code point)
```

A: New interval width *A* = .0001 × (.001) = .0000001. (Current width *A* times probability *p*.)

Carry-over problem

The encoding of the fourth symbol exemplifies a small problem, called the carry-over problem. After encoding symbols "*a*," "*a*," and "*b*," each having codewords of lengths 1, 1, and 2, respectively, in Table 1, the first four bits of an encoded string using Huffman coding would not change. However, in this arithmetic code, the encoding of symbol "*c*" changed the value of the third code-string bit. (The first three bits changed from .100 to .101.) The change was prompted by a *carry-over*, because we are basically adding quantities to the code string. We discuss carry-over control later on in the paper.

Code-string termination

Following encoding of "*a a b c*," the current interval is [.1010011,.1010100). If we were to terminate the code string at this point (no more data symbols to handle), any value equal to or greater than .1010011, but less than .1010100, would serve to identify the interval.

Let us overview the example. In our creation of code string .1010011, we in effect added properly scaled cumulative probabilities *P*, called augends, to the code string. For the width recursion on *A*, the interval widths are, fortuitously, negative integral powers of two, which can be represented as floating point numbers with one bit of precision. Multiplication by a negative integral power of two may be performed by a shift right. The code string for "*a a b c*" is the result of the following sum of augends, which displays the scaling by a right shift:

```
.011
  011
   001
     111
________
.1010011
```

Decoding

Let us retain code string .1010011 and decode it. Basically, the code string tells the decoder what the encoder did. In a sense, the decoder recursively "undoes" the encoder's recursion. If, for the first data symbol, the encoder had encoded a "*b*," then (referring to the cumulative probability *P* column of Table 2), the code-string value would be at least .001 but less than .011. For encoding an "*a*," the code-string value would be at least .011 but less than .111. Therefore, the first symbol of the data string must be "*a*" because code-string .1010011 lies in [.011,.110), which is *a*'s subinterval. We can summarize this step as follows.

Step 1: Decoder C comparison Examine the code string and determine the interval in which it lies. Decode the symbol corresponding to that interval.

Since the second subinterval code point was obtained at the encoder by adding something to .011, we can prepare to decode the second symbol by subtracting .011 from the code string: .1010011 − .011 = .0100011. We then have Step 2.

Step 2: Decoder C readjust Subtract from the code string the augend value of the code point for the decoded symbol.

Also, since the values for the second subinterval were adjusted by multiplying by .1 in the encoder A recursion, we can "undo" that multiplication by multiplying the remaining value of the code string by 2. Our code string is now .1000011. In summary, we have Step 3.

Step 3: Decoder C scaling Rescale the code *C* for direct comparison with *P* by undoing the multiplication for the value *A*.

Now we can decode the second symbol from the adjusted code string .100011 by dealing directly with the values in Table 2 and repeating Decoder Steps 1, 2, and 3.

Decoder Step 1 Table 2 identifies "*a*" as the second data symbol, because the adjusted code string is greater than .011 (codeword for "*a*") but less than .111 (codeword for "*c*").

Decoder Step 2 Repeating the operation of subtracting .011, we obtain

.100011 − .011 = .001011.

Decoder Step 3 Symbol "*a*" causes multiplication by .1 at the encoder, so the rescaled code string is obtained by doubling the result of Decoder Step 2:

.01011.

The third symbol is decoded as follows.

Decoder Step 1 Referring to Table 2 to decode the third symbol, we see that .01011 is equal to or greater than .001 (codeword for "*b*") but less than .011 (codeword for "*a*"), and symbol "*b*" is decoded.

Decoder Step 2 Subtracting out .001 we have .00111:

.01011 − .001 = .00111.

Decoder Step 3 Symbol "*b*" caused the encoder to multiply by .01, which is undone by rescaling with a 2-bit shift:

.00111 becomes .111.

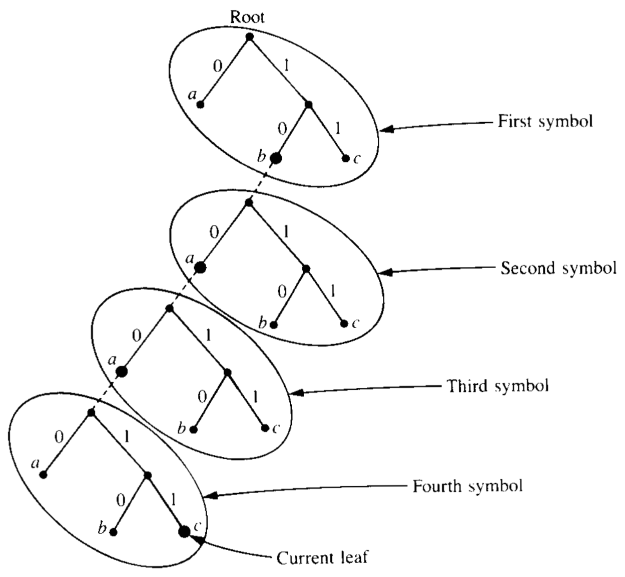
To decode the fourth and last symbol, Decoder Step 1 is sufficient. The fourth symbol is decoded as "*c*," whose code point corresponds to the remaining code string.

Symbol	Codeword
a	0
b	10
c	11

(a)

$$b\ a\ a\ c \longrightarrow \underline{1}\,\underline{0}\,\underline{0}\,\underline{0}\,1\,1$$

(b)



(c)

Figure 4 The code-string tree of a prefix code. Example code, image of data string is a single leaf: (a) code table, (b) data-string-to-code-string transformation, (c) code-string tree.

• *A general view of the coding process*
We have related a special prefix code (the symbols were ordered with respect to their probability) and a special arithmetic code (the probabilities were all negative integral powers of two) by picturing the unit interval. The multiplication inherent in the encoder width recursion for A, in the general case, yields a new A which has a greater number of significant digits than the factors. In our simple example, however, this multiplication did not cause the required precision to increase with the length of the code string because the probabilities p were integral negative powers of two. Arithmetic coding is capable of using arbitrary probabilities by keeping the product to a fixed number of bits of precision.

A key advance of arithmetic coding was to contain the required precision so that the significant digits of the multiplications do not grow with the code string. We can describe the use of a fixed number of significant bits in the setting of a code-string tree. Moreover, constrained channel codes [1] as well as compression codes benefit from the code tree view. Again, the code alphabet is the binary alphabet $\{0,1\}$. The code-string tree represents the set of all finite binary code strings, denoted $\{0,1\}^*$.

We illustrate a conventional Huffman code in **Figure 4,** which shows the code-string tree for mapping data string s = "$b\ a\ a\ c$" to a binary code string. Figure 4(a) shows a codeword table for data symbols "a," "b," and "c." The encoding proceeds recursively. The first symbol of string s, "b," is mapped to code string 10, "$b\ a$" is mapped to code string 100, and so on, as indicated in Fig. 4(b). Thus the depth of the code tree increases with each recursion. In Fig. 4(c) we highlight the branches traveled to demonstrate that the coding process successively identifies the underlined nodes in the code-string tree. The root of the codeword tree is attached to the leaf at the current depth of the tree, as per the previous data symbol.

At initialization, the available *code space* (A) is a *set* $\{0,1\}^*$, which corresponds to the unit interval. Following the encoding of the first data symbol b, we identify node 10 by the path from the root. The depth is 2. The *current* code space is now all continuations of code string 10. We recursively *subdivide*, or subset, the current code space. A property of prefix codes is that a *single node* in the code space is identified as the result of the *subdivision operation.* In the unit interval analogy, prefix codes identify single points on the interval. For arithmetic codes, we can view the code as mapping a data string to an interval of the unit interval, as shown in Fig. 3, or we can view the result of the mapping as a set of finite strings, as shown in Fig. 4.

2. A Binary Arithmetic Code (BAC)

We have presented a view of prefix codes as the successive application of a subdivision operation on the code space in order to show that arithmetic coding successively subdivides the unit interval. We conceptually associate the unit interval with the code-string tree by a correspondence between the set of leaves of the code-string tree at tree depth D on one hand, and the rational fractions of denominator 2^D on the other hand.

We teach the *binary arithmetic coding* (BAC) algorithm by means of an example. We have already laid the groundwork, since we follow the encoder and decoder operations and general strategy of the previous example. See [2] for a more formal description.

The BAC algorithm may be used for encoding any set of events, whatever the original form, by breaking the events down for encoding into a succession of binary events. The BAC accepts this succession of events and delivers successive bits of the code string.

The encoder operates on variable *MIN*, whose values are T (true) and F (false). The name *MIN* denotes "more probable in." If *MIN* is true (T), the event to be encoded is the more probable, and if *MIN* is false (F), the event to be encoded is the less probable. The decoder result is binary variable *MOUT*, of values T and F, where *MOUT* means "more probable out." Similarly, at the decoder side, output value *MOUT* is true (T) only when the decoded event is the more probable.

In practice, data to be encoded are not conveniently provided to us as the "more" or "less" probable values. Binary data usually represent bits from the real world. Here, we leave to a statistics unit the determination of event values T or F.

Consider, for example, a black and white image of two-valued pels (picture elements) which has a primary white background. For these data we associate the instances of a white pel value to the "more probable" value (T) and a black pel value into the "less probable" value (F). The statistics unit would thus have an internal variable, *MVAL*, indicating that white maps to T. On the other hand, if we had an image with a black background, the mapping of values black and white would be respectively to values T and F (*MVAL* is black). In a more complex model, if the same black and white image had areas of white background interspersed with neighborhoods of black, the mapping of pel values black/white to event values F and T could change dynamically in accordance with the context (neighborhood) of the pel location. In a black context, the black pel would be value T, whereas in the context of a white neighborhood the black pel would be value F.

The statistics unit must determine the additional information as to *by how much* value T is more popular than value F. The BAC coder requires us to estimate the relative ratio of F to the nearest power of 2; does F appear 1 in 2, or 1 in 4, or 1 in 8, etc., or 1 in 4096? We therefore have 12 distinct coding parameters *SK*, called *skew*, of respective index 1 through 12, to indicate the relative frequency of value F. In a crude sense, we select one of 12 "codes" for each event to be encoded or decoded. By approximating to 12 skew values, instead of using a continuum of values, the maximum loss in coding efficiency is less than 4 percent of the original file size at probabilities falling between skew numbers 1 and 2. The loss at higher skew numbers is even less; see [2].

In what follows, our concern is how to code binary events after the relative frequencies have been estimated.

• *The basic encoding process*
The double recursion introduced in conjunction with Table 2 appears in the BAC algorithm as a recursion on variables C (for code point) and A (for available space). The BAC algorithm is initialized with the code space as the *unit interval* [0,1) from value 0.0 to value 1.0, with $C = 0.0$ and $A = 1.0$. The BAC coder successively splits the width or size of the available code space A, or current interval, into two subintervals. The left subinterval is associated with F and the right subinterval with T. Variables C and A jointly describe the current interval as, respectively, the leftmost point and the width. As with the initial code space, the current interval is closed on the left and open on the right: $[C, C + A)$.

In the BAC, not all interval widths are integral negative powers of two. For example, where p of event F is $\frac{1}{4}$, the other probability for T must be $\frac{3}{4}$. For the width associated with T of $\frac{3}{4}$, we have more than one bit of precision. The product of probabilities greater than $\frac{1}{2}$ can lead to a growing precision problem. We solve the problem by representing space A with a floating point number to a fixed precision. We introduce variable E for the exponent, which controls the "data handling" and "shifting" aspect of the algorithm. We represent variable A in floating point with the most significant bit of A in position E from the left. Thus the leading 1-bit of the binary representation of A has value 2^{-E}. For example, if A = 0.001011, E has value 3, 2^{-E} is 0.001, and A is 1.011.

In the encoder A recursion of the example of Table 2, the width is determined by a multiplication. In the simple BAC algorithm, the smaller width is determined by the value *SK*, as in Eq. (1), which follows. The other width is the difference between the current width and the smaller width, as in Eq. (2), which follows. No multiplication is needed.

The current interval is split according to the skew value *SK* as follows. If *SK* is 1, the interval is split nearly in half, and if *SK* is 12, the interval is split with a very small subinterval assigned to F. Note that we roughly subdivide the current interval in a way which corresponds to the relative frequency of each event. Let $W(\mathrm{F})$ and $W(\mathrm{T})$ be respectively the subinterval widths assigned to F and to T. Specifically,

$$W(\mathrm{F}) = 2^{-(E+SK)}, \tag{1}$$

with the remainder of the interval width A assigned to T:

$$W(\mathrm{T}) = A - 2^{-(E+SK)}. \tag{2}$$

We can summarize the handling of an event (value T or F) in the BAC algorithm in three steps. The first and second steps correspond to the A and C recursions described earlier. The third step is a "data handling" or scaling step which we have ignored in the earlier examples. Let s denote the string of data symbols already encoded, and let notation $C(s)$, $A(s)$, and $E(s)$ respectively denote the values of variables C, A, and E following the encoding of the data string. Now, after handling the next event T, let the new values of variables C, A, and E be respectively denoted $C(s,\mathrm{T})$, $A(s,\mathrm{T})$, and $E(s,\mathrm{T})$.

Steps for encoding of next event

Table 3 Example encoding—refining the interval.

Event	*MIN* (value)	*SK* (skew)	*E* (A's lead 0s)	*W(F)* (F width)	*C* (least pt)	*A* (interval A)
Initial	—	—	0	—	0.000000	1.000000
1	T	3	0	.001	0.001000	0.111000
2	T	1	1	.01	0.011000	0.101000
3	F	1	1	.01	0.011000	0.010000
4	T	1	2	.001	0.100000	0.001000

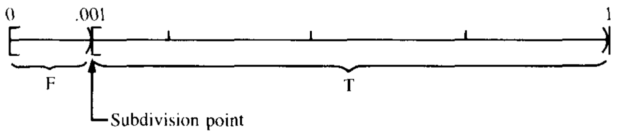


Figure 5 Interval splitting—subdivision for Event 1, Table 3.

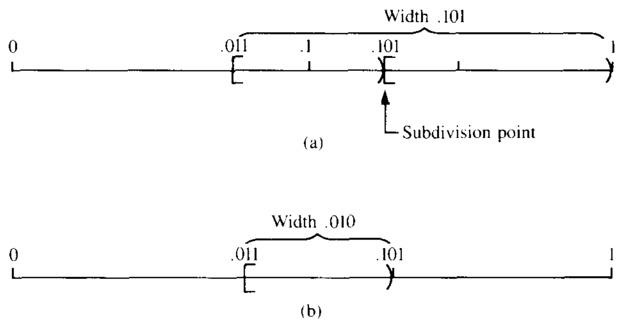


Figure 6 Interval splitting—subdivision for Event 3, Table 3: (a) Current interval at end of Event 2 and subdivision point. (b) Current interval following encoding of Event 3.

Step 1 Given skew *SK* and *E* (the leading 0s of *A*), subdivide the current width as in Eqs. (1) and (2).

Step 2 Given the event values (T or F), *C*, *W*(T), and *W*(F), describe the new interval:

If T: $C(s,\mathrm{T}) = C(s) + W(\mathrm{F})$ and $A(s,\mathrm{T}) = W(\mathrm{T})$. (3a)

If F: $C(s,\mathrm{F}) = C(s)$ and $A(s,\mathrm{F}) = W(\mathrm{F})$. (3b)

Step 3 Given the new value of *A*, determine the new value of *E*:

If T: If $A(s,\mathrm{T}) < 2^{-E(s)}$, then $E(s,\mathrm{T}) = E(s) + 1$;

otherwise $E(s,\mathrm{T}) = E(s)$.

If F: $E(s,\mathrm{F}) = E(s) + SK$.

We continue the discussion by an example, where we encode the four-event string T, T, F, T under respective skews 3, 1, 1, 1. The encoding is described by **Table 3,** and the following description accompanies this table.

For Event 1, *SK* is 3 and *E* is 0. For Step 1, the width associated with the value F, *W*(F), is 2^{-3} or 0.001. *W*(T) is what is left over, or 1.000 − 0.001 = 0.111. See **Figure 5.** Relative to Step 2, Eq. (3), the subdivision point is *C* + *W*(T) or 0 + .001 = .001. Since the binary value is T and the relative frequency of the T event is equal to or greater than $\frac{1}{2}$, we keep the larger (rightmost) subinterval. Referring to Fig. 5, we see that the new values of *C* and *A* which describe the interval are now *C*(T) = 0.001 and *A*(T) = *W*(T) = 0.111. For Step 3, we note that *A* has developed a leading 0, so *E* = 1.

For Event 2 of Table 3, the skew *SK* is 1 and *E* is now 1, so *W*(F) is $2^{-(1+1)}$ or 0.01. *W*(T) is thus 0.111 − 0.010 = 0.101. The subdivision point of the current interval is *C* + *W*(F), or 0.011. Again, the event value is T, so we keep the rightmost part. The new value of *C* is the subdivision point 0.011, and the new value of *A* is *W*(T) or 0.101. The leading 1-bit position of *A* has not changed, so *E* is still 1.

For Event 3 of Table 3, see **Figure 6,** which displays current interval [.011,1) of width .101. The smaller width *W*(F) is $2^{-(1+1)}$ or .01. We add this value to *C* to obtain *C* + *W*(F), or subdivision point .101. See Fig. 6(a). Referring to Event 3 of Table 3, the value to encode is F, so we must now keep the left side of the subdivision. By keeping the F subinterval, the value of *C* remains at .011 and *A* becomes *W*(F) or 0.01. Available width *A* has a new leading 0, so *E* becomes 2. The resulting interval is shown in Fig. 6(b).

• *Separation of data handling from the arithmetic*

Arithmetic codes generate the code string by adding a summand (called *augend* in the arithmetic coding literature) to the current code string and possibly shifting the result. The summation operation creates a problem called the *carry-over problem.* We can, in the course of code generation, shift out a long string of 1s from the coding process. An addition could propagate a carry into the long string of 1s, changing the values of higher-order bits until a 0 is converted to a 1 and stopping the *carry chain.*

In this section we show how the arithmetic using *A* and *C* can be separated from the carry-over handling and databuffering functions. The scheme of Table 3 assumes that the A and C registers are quite long, i.e., that they increase in length with the number of events encoded. As the value of variable E increases, so does the length of A and C. In reality, the A and C registers can be made of fixed precision, and the results shifted left out of the C register. Also, the leading 0s of the A register are not needed, which justifies a fixed-length A register. Let **A, C,** and **W** denote fixed-length registers (perhaps no more than 16 bits, for example).

In making the adjustment to fixed-length registers **C** and **A**, Eq. (1) and Eq. (2) are normalized to operate on **A**:

$$\mathbf{W}(\mathrm{F}) = 2^{-SK},$$

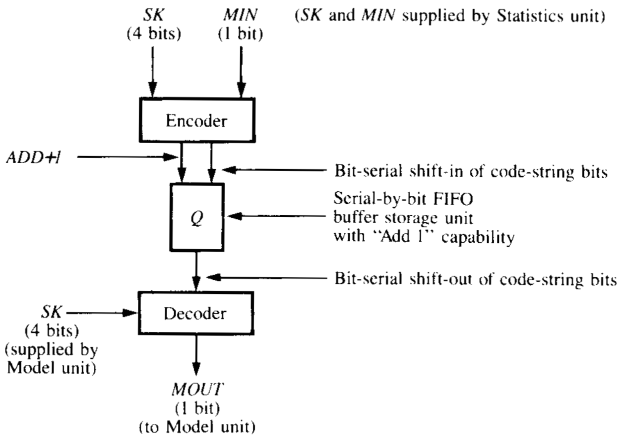
$$\mathbf{W}(\mathrm{T}) = \mathbf{A} - 2^{-SK}.$$

The widths **W** may have leading 0s, and when becoming the new **A** we must renormalize. Thus, as the **A** register develops leading 0s, a *normalization* left shift restores value 1 to the most significant bit of the **A** register. However, **A** and **C** must maintain their relative alignment, so we have a rule: *Register C undergoes a normalization shift whenever register A does.*

The basic conceptual input/output view of the algorithm, both for the compression and decompression process, is shown in **Figure 7.** In this figure we have decomposed the encoding task into the encoder itself which handles the A and C registers, and the special arbitrarily long FIFO buffer Q which handles the code string and the carry-over. Note that the encoder and decoder in practice are interfaced to the original data via a statistics unit which is not shown. The statistics unit provides skew numbers SK.

The encoder accepts successive events of binary information from the input variable MIN, whose values T or F are encoded under a coding parameter SK. The code string is shifted out of the encoder into a FIFO storage unit Q. At the other end of Q, the decoder receives the code string from FIFO Q. The decoder is also supplied the same value of the coding parameter SK under which the successive input values MIN were encoded. The decoder, with a knowledge of the code string C coming from Q at one input port and the successive coding parameter values at the other input port SK coming from the decoder's statistics unit, produces successive values at output port $MOUT$.

For the description of the algorithm given here, we assume that the FIFO store Q has sufficient capacity for the entire code string, and that Q has the capability of propagating an incoming carry. Thus, in Fig. 7 the encoder has an output $ADD{+}1$ which signifies a carry-over from the encoder to the FIFO buffer Q. When $ADD{+}1$ is active, the FIFO buffer first propagates the carry (adds 1 to the lowest-order bit position of Q) and then performs the shift.



SK (4 bits)
MIN (1 bit)
(SK and MIN supplied by Statistics unit)
Encoder
ADD+l
Bit-serial shift-in of code-string bits
Q
Serial-by-bit FIFO buffer storage unit with "Add 1" capability
Bit-serial shift-out of code-string bits
SK (4 bits) (supplied by Model unit)
Decoder
MOUT (1 bit) (to Model unit)

Figure 7 Block diagram of a binary event encoder/decoder system.

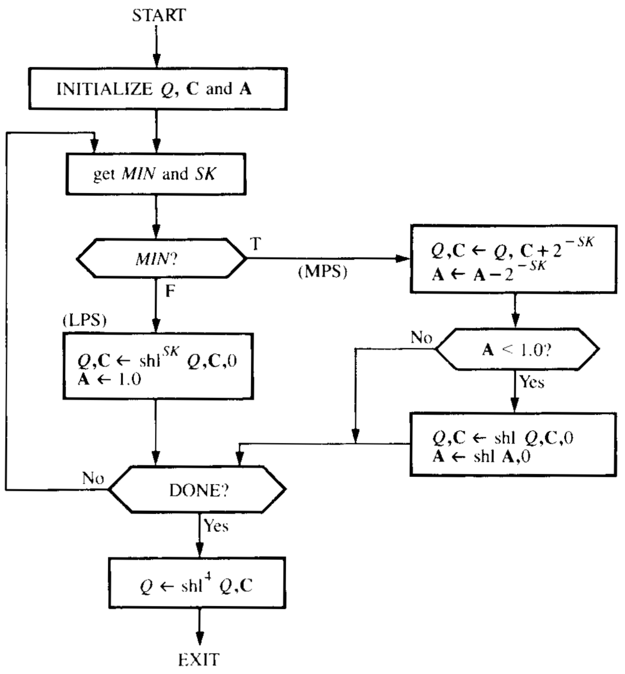


Figure 8 Flowchart for BAC algorithm encoder.

The BAC encoder algorithm with normalization is described in flowchart form in **Figure 8.** The figure uses a simple register transfer notation we define in the obvious way by example. We assume the existence of three registers: Q (the FIFO buffer), **C**, and **A**. Let us assume that values of SK are limited to 1, 2, and 3. Registers **C** and **A** are one bit longer than the maximum SK, and so are 4 bits. Register **C** is initialized to 0.000, and Register **A** is initialized to 1.000. The basic idea of the algorithm using normalization is very simple. To encode a T, add 2^{-SK} to **C** and subtract 2^{-SK} from **A**. Let "," denote concatenation. In the register transfer notation,

Table 4 Example encoding with normalization.

Event	*MIN*	*SK*	*Q*	*C*	*A*	*Normalization*
Initial	—	—	—	0.000	1.000	—
1	T	3	0	0.010	1.110	Yes
2	T	1	0	0.110	1.010	No
3	F	1	00	1.100	1.000	F-shift of *SK*
4	T	1	010	0.000	1.000	Yes

$$Q,\mathbf{C} \leftarrow Q,\mathbf{C} + 2^{-SK}, \tag{4a}$$

$$\mathbf{A} \leftarrow \mathbf{A} - 2^{-SK}. \tag{4b}$$

If the result in **A** is equal to or greater than 1.000, we are done with the double recursion. If the result in **A** is less than 1.000, a *normalization* shift is needed. We shift Q and **C** left as a pair, and shift left **A**. Let "shl" denote shift left one bit, and "shl^2" denote a shift left of two bits, etc. If **A** is less than 1.000, then

$$Q,\mathbf{C} \leftarrow \text{shl } Q,\mathbf{C},0, \tag{5a}$$

$$\mathbf{A} \leftarrow \text{shl } \mathbf{A},0. \tag{5b}$$

In the above, ",0" denotes "0-fill" (the vacated positions are filled with 0s).

If the symbol to be encoded is F, Fig. 8 shows that the action to perform is relatively simple:

$$Q,\mathbf{C} \leftarrow \text{shl}^{SK}\ Q,\mathbf{C},0, \tag{6a}$$

$$\mathbf{A} \leftarrow 1.0. \tag{6b}$$

We use the same example as in Table 3 redone as shown in **Table 4.** Columns Q, **C**, and **A** show the *result* of applying the *MIN* and *SK* values of that step. The first row is the initialization.

Event 1, with **C** and **A** as initialized, encodes value T with an *SK* of 3. The arithmetic result for Eq. (4a) is **C** = 0.000 + 0.001, and for Eq. (4b) is **A** = 1.000 − .001 = 0.111. Since 0.111 is less than 1.0, we must apply Eq. (5) to normalize. Following the normalization shift, **Q** is now 0, **C** is 0.010, and **A** is 1.110.

Event 2 encodes value T with a skew of 1. We perform the operations of Eq. (4a), resulting in **C** of 0.110 as follows:

$$\begin{array}{rl} 0.010 & (\text{old } \mathbf{C}) \\ +\underline{\ .1\ \ \ } & (2^{-1}) \\ 0.110 & (\text{new } \mathbf{C}) \end{array}$$

Equation (4b) gives 1.010 for **A**:

$$\begin{array}{rl} 1.110 & (\text{old } \mathbf{A}) \\ -\underline{\ .1\ \ \ } & (-2^{-1}) \\ 1.010 & (\text{new } \mathbf{A}) \end{array}$$

Since the register **A** result is greater than 1.0, the normalization shift of Eq. (5) is not needed.

Event 3 encodes value F at skew 1. The algorithm for encoding an F is Eq. (6). The value F is encoded by shifting the Q,**C** pair left *SK* bits and reinitializing **A** to 1.000. Summarizing Event 3, an F of skew 1 is a one-bit shift left for Q,**C**, so Q is 00 and **C** is 1.100. Equation (6b) reinitializes **A** to 1.000.

Event 4 illustrates a carry-over. Event 4 encodes value T with a skew of 1. Following Eq. (4a), adding 2^{-1} (0.100) to **C** (1.100) results in ^{1}0.000, where ^{1} indicates the carry-out from the **C** register. This carry propagates to Q by activating encoder output signal *ADD+1*, and this carry-over operation converts Q from 00 to 01. Meanwhile, for Eq. (4b), 2^{-1} subtracts from register **A** leaving 0.100, so the normalization shift of Eq. (5) is needed. Q now becomes 010. The value of code string is 0100000, which is the same result of Table 3, as expected.

- *Carry-over control*

Arithmetic coding ensures that no future value of C can exceed the current value of $C + A$. Consequently, once a carry-over has propagated into a given code-string position in Q, no other carry-over will reach the same code-string position. In the above sample, the second bit of the code string received a carry-over. The algorithm ensures that this same bit position (second from the beginning) cannot receive another carry-over during the succeeding encoding operations. This observation leads to a method for controlling the carry called *bit-stuffing* [3]. At the encoder side, if 16 1s in a row are emitted, the buffer can *insert* (stuff) a 0. This 0 converts to a 1 and blocks the carry from reaching the string of 16 1s. Therefore a bit-stuff permits the block with the 16 1s to be transmitted. At the decoder side, if the decoder encounters 16 1s in a row, the decoder buffer removes and examines the stuffed bit. If the stuffed bit value is 1, the carry is propagated inside the decoder.

- *Code-string termination*

When the encoding is done, the **C** register may be shifted into the Q FIFO store. However, after the last event has been coded, we remain with interval $[C,C + A)$. If we know the length of the data string, then we know when we have decoded the last event. Therefore, any code string whose magnitude lies in $[C,C + A)$ decodes to the original data string. In the present case, we can simply truncate the trailing 0s. The truncation process leaves "01" as the code string, with the convention that the decoder insert as many trailing 0s as it takes to decode four data bits.

In the general case, any code string greater than 0100000 (smallest value in current interval) and strictly less than $C + A$ = 010000 + 0001000 = 010100 suffices. Our shortest selection remains 01.

• *Decoding process*

The decoding part of the BAC algorithm is shown in **Figure 9.** Consider decoding the example code string of Table 4. We demonstrate the decoding process with the aid of **Table 5.** Register **A** is initialized to 1.000 and **C** is initialized to the first four bits out of the FIFO buffer **Q.** Since **Q** only has 2 bits (01), we insert 0s as needed. **C** is initialized to 0.100. The description of each event that follows accompanies Fig. 9 and one line of Table 5.

Event 1 To decode the first event we need the value of *SK*, which is 3. We subtract 2^{-3} from **C** as an *intermediate result* called *CBUF*. *CBUF* is 0.011, which is greater than 0, so the resulting bit is T. So 2^{-3} is subtracted from **A,** giving 0.111 and the contents of *CBUF* are transferred into **C.** Register **A** is less than 1.0, so a normalization shift is needed. **C** and **A** are now 0.110 and 1.110.

Event 2 Now we obtain the second value of *SK*, which is 1. Subtracting 2^{-1}, or 0.100, from **C** gives a *CBUF* value of 0.010, which is positive. Therefore the result is T, and 0.010 is the new value of **C.** Subtracting 0.100 from register **A** gives 1.010, so no normalization is needed.

Event 3 For the third event, *SK* is again 1, so we again subtract 0.100 from **C** (which is now 0.010). The result *CBUF* is negative, so the event is F. We do not load **C** from *CBUF*, but simply shift **C** one position left, and reinitialize **A. C** is now 0.100 and **A** is 1.000.

Event 4 The fourth *SK* is 1, and subtracting 0.100 from **C** leaves 0.000 for *CBUF*. The result is not negative, so the event is T. To continue the algorithm, we subtract 2^{-1} from **A,** discover that the result 0.100 is less than 1, and do a normalization shift of **C** and **A. A** is now 1.000 and decoding is complete.

Note that column **A** and the Normalization columns of Table 4 (encoder) and Table 5 (decoder) are identical. The **A** register contents always follow the same sequence of values for the decode phase as for the encode phase.

• *Framework for prefix codes and arithmetic codes*

We can apply the code-string-tree representation of the coding operations to arithmetic codes. However, unlike prefix codes, in arithmetic coding the encoding of a given symbol may result in a code space described by continuations of more than one leaf of the tree. We illustrate the point by showing the example of Event 1 of Table 4 in the form of a code-string tree of **Figure 10.**

The smallest subinterval at the current depth is a single leaf. With a maximum skew *SK* of 2^{-3}, we identify value .001 with a single leaf at the current depth. With a maximum *SK* of 3, the value of **A** can range from 1.000 to 1.110. At the same current depth where 2^{-3} is one leaf, the subset of code-string

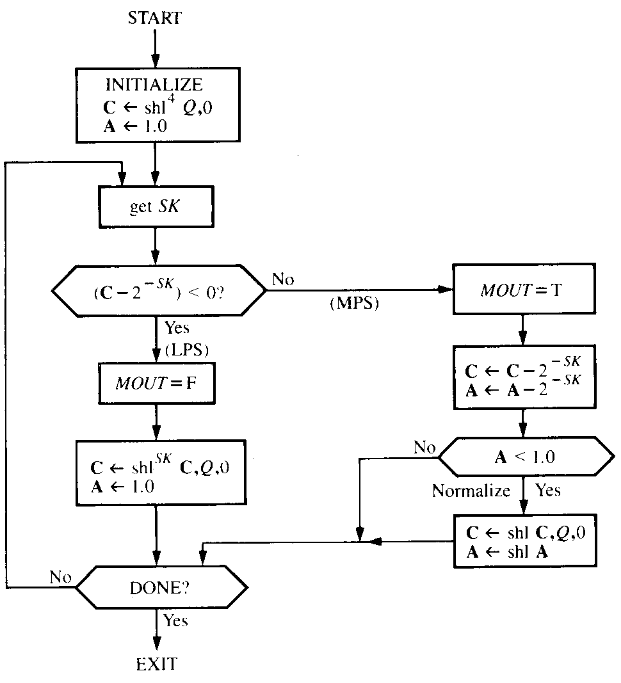


Figure 9 Flowchart for BAC algorithm decoder.

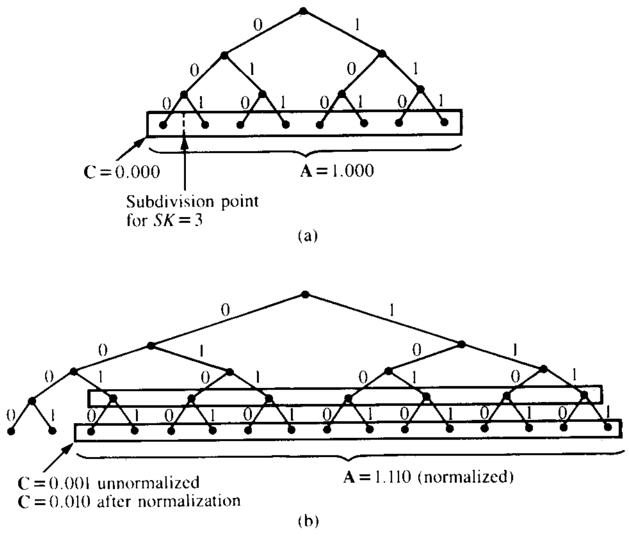
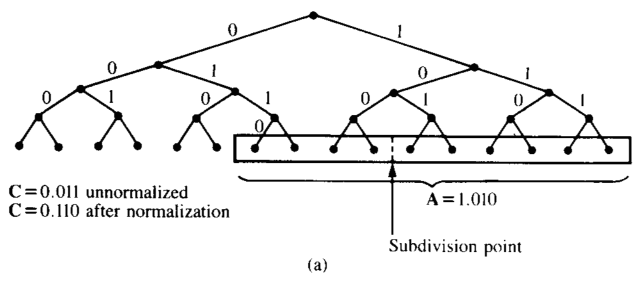


Figure 10 Code-string tree for Event 1, Table 4: (a) Initial tree. (b) Following encoding of Event 1.

Table 5 Example decoding.

Event	*SK*	*C (after)*	*A (after)*	*CBUF*	*MOUT*	*Normalization*
Initial	—	0.100	1.000	—	—	—
1	3	0.110	1.110	0.011	T	Yes
2	1	0.010	1.010	0.010	T	No
3	1	0.100	1.000	−1.110	F	F-shift of *SK*
4	1	0.000	1.000	0.000	T	Yes



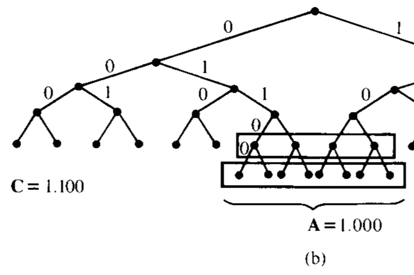


Figure 11 Code-string tree for Event 3, Table 4: (a) Following Event 2. (b) Following Event 3.

tree leaves which describe the code space is from eight leaves to 14 leaves. Also at this depth, an *SK* of 2 corresponds to two leaves and an *SK* of 1 corresponds to four leaves. The initial code-string tree has eight leaves (**A** = 1.000) and a depth of three. See Fig. 10(a) for the subdivision point for Event 1 of Table 4. Event 1 is T, so we keep the right subset of seven leaves. For subsequent encoding, the code string will be a continuation of 001, 01, or 1, and we clearly are not dealing with a prefix code.

With only seven leaves at depth three, we increase the depth by one to four, so that we now have 14 leaves at the new depth. This process of increasing the depth corresponds to the normalization shifting done in Table 4. The result is shown in Fig. 10(b).

Figure 11 shows Event 3 of the example. Figure 11(a) shows the subdivision point for the skew of 1 for Event 3. An *SK* of 1 corresponds to a subinterval of four leaves, which are the leftmost leaves of the current interval. The event value is F, so in this case we retain the left subinterval with the four leaves. Figure 11(b) shows the result after encoding Event 3. Since only four leaves were taken, we increase the tree depth one level, again analogous to the normalization operation in the example of Table 4. Now the current code space which we subdivide for Event 4 consists of the continuations of code strings: 011 and 100.

We close this section with a suggestion. The path by which we discovered the particular binary arithmetic code described here [2] was a simplification of the code in [3] and was not a generalization of Golomb's code [4]. However, an interesting relationship exists, which the readers are invited to discover for themselves. We suggest that a Golomb code be selected, that a short run be encoded, and that the reader construct the corresponding code-string tree.

3. The general case: Multisymbol data alphabets

As noted from the example of Table 2, arithmetic coding applies to n-symbol alphabets as well. Here we use notation A and C in the arbitrarily long register sense of Table 3, and $A(s)$ and $C(s)$ are the respective values following the encoding of data string s. We describe some encoding equations for multisymbol alphabets.

• *Encoding*

Let the n-symbol alphabet, whatever it may be, have an ordering. Our interest is in the symbol position of the ordering: $1, 2, \cdots, k, \cdots, n$. Let relative frequencies $p_1, p_2, \cdots, p_n$, and interval width $A(s)$ be given. The interval is subdivided according to relative frequencies, so the respective widths are $p_1 \times A(s), p_2 \times A(s), \cdots, p_n \times A(s)$. Recall the calculation of cumulative probabilities P:

$$P_k = \sum_{j<k} p_j \text{ for } k \geq 2, \text{ and } P_1 = 0.$$

For encoding the event whose order is k, following the encoding of prior string s, the new subinterval $[C(s,k), C(s,k) + A(s,k))$ is described with the following two equations of the familiar arithmetic coding double recursion:

$$C(s,k) = C(s) + D(s,k), \text{ where } D(s,k) = A(s) \times P_k, \qquad (7)$$

$$A(s,k) = A(s) \times p_k. \qquad (8)$$

• *Decoding*

Let $|C(s)|$ denote the result of the termination operation on $C(s)$, where it is understood that the decoder inserts trailing 0s as needed. The decoding operation, also performed recursively per symbol, may be accomplished by subtracting out of $|C(s)|$ the previously decoded summands. For example, let s^0 be the prefix of data string s which has already been decoded. We are now left with quantity $|C(s)| - C(s^0)$. The decoder must find symbol i such that

$$D(s^0,k) \leq |C(s)| - C(s^0) < D(s^0,k+1),$$

where $k < n$. We decode symbol n if $D(s^0,n) \leq |C(s)| - C(s^0)$.

• *Precision problem*

Equations (7) and (8) describe Elias' code, which has a precision problem. If $A(s)$ has 8 bits of precision, and p_k has 8 bits of precision, then $A(s,k)$ requires 16 bits of precision. If we keep the 16-bit result, A needs 24 bits of precision for the next recursion, and so on. In practice, we wish $A(s,k)$ to have the same precision as $A(s)$, and use fixed-precision registers **A** and **C** for the arithmetic as we do in the BAC examples of Tables

4 and 5. Pasco [5], who is responsible for the first FIFO arithmetic code, solved the precision problem by *truncating* the product of Eq. (8) to the same precision as $A(s)$. In terms of the leaves at the current depth of the code-string tree, truncation in Eq. (8) allows *gaps* to form in the code space. A gap is a leaf of the current interval which is not assigned to any of the symbols. For such unassigned leaves, no data string maps to a continuation of such leaves, and there is some wasted code space. Martin [6] and Jones [7], independently of Pasco and each other, discovered practical versions of Eqs. (7) and (8) which partition the code space and have no gaps. They do this by ensuring that

$$C(s,k+1) = C(s,k) + A(s,k).$$

• *Decodability*

Codes are not uniquely decodable if two data strings map into the same code space. In arithmetic codes, if the subdivision of the interval yields an *overlap*, more than one data string can map to the overlapped subinterval. Overlap occurs when, in subdividing interval $A(s)$, the sum of the subinterval widths exceeds $A(s)$ itself. Thus, some continuations of data string s may map to the subinterval whose least point is $C(s+1)$. We define string $s+1$ to be the data string of the same length as s which is next in the lexical ordering. If s is already the last string in the ordering, i.e., string "$n\ n \cdots n$," then $C(n\ n \cdots n)$ is 1.000. The general decodability criterion is thus

$$C(s,n) + A(s,n) < C(s+1), \tag{9}$$

where s,n denotes symbol n concatenated to string s. For arithmetic codes which do not explicitly use value $A(s,n)$, it may be calculated as $C(s\ n\ n\ n\ n, \cdots) - C(s,n)$. In cases where Eq. (9) is violated, the interval of overlap is

$$[C(s+1), C(s,n) + A(s,n)).$$

Similarly, a gap is formed between the mappings of data string s and $s+1$ if

$$C(s+1) - C(s) > A(s,1) + \cdots + A(s,n). \tag{10}$$

A gap does not affect decodability; it simply means that the code space is not fully utilized.

• *P-based arithmetic codes*

For P-based arithmetic codes, the code space is represented as a number A which is subdivided by a multiplication in proportion to the relative frequencies of the symbols.

The *decodability criterion* for P-based codes is given by

$$A(s) \geq A(s,1) + A(s,2) + \cdots + A(s,n). \tag{11}$$

If this equation is met with equality for all s, then the algorithm leaves no gaps.

• *L-based arithmetic codes*

The L-based arithmetic codes represent the width of the code space $A(s)$ as a value $2^{-[Y(s)+X(s)]}$, where $Y(s)$ is an integer and $X(s)$ is a rational fraction of an integer denominator M. In fact, the length of the code string for s, denoted $L(s)$, is $Y(s) + X(s)$. Here, $Y(s)$ corresponds to $E(s)$ of the example of Table 3. [When the code string is terminated following the encoding of the last symbol, the code-string length of $C(s)$ is within a few bits of $L(s)$.] For L-based arithmetic codes, $D(s,i)$ is determined as the product

$$D(s,k) = \mathbf{D}(X(s),k) \times 2^{-Y(s)}, \tag{12}$$

so that

$$C(s,k) = C(s) + D(s,k).$$

Since there are M distinct values of $X(s)$, i.e., 0, $1/M$, $2/M, \cdots, (M-1)/M$, and n symbols, we precalculate an M-by-n table of values $\mathbf{D}(X,k)$. [Actually, for symbol $k = 1$, the value of $\mathbf{D}$ is 0, so only M-by-$(n-1)$ values need be stored.] In Eq. (12), multiplication by $2^{-Y(s)}$ is simply a shift.

Corresponding to the relative frequency estimates, p_k, are instead length approximations, $\ell_k \simeq -\log p_k$. These lengths must satisfy the generalized Kraft inequality [8]. Following encoding of symbol k, the internal variables are updated:

$$L(s,k) = Y(s) + X(s) + \ell_k,$$

where again $L(s)$ is broken into integer part $Y(s,k)$ and fraction part $X(s,k)$. The decodability criterion of Eq. (11) is also the decodability criterion for L-based codes if $A(s,k)$ is as defined in Eq. (13).

For $k < n$ define

$$A(s,k) = B(s,k+1) - B(s,k),$$

and for n,

$$A(s,n) = B(s,n,n) + B(s,n,n,n) + \cdots. \tag{13}$$

• *Applications*

Langdon and Rissanen [3] applied a FIFO P-based binary arithmetic code to the encoding of black and white images, using as a context the value of neighboring pels already encoded. This work introduced the notion of the binary coding parameter called skew. A binary code is particularly useful for black and white images because the contexts employed for successive pels may have different relative frequencies. Traditional run-length codes such as Golomb's [4] are only designed for a particular relative frequency for the repeating symbol. In [3], a method to dynamically adapt to the pel statistics is described. Particularly simple adaptation techniques exist for determining the skew number.

Arithmetic coding can also be applied to file compression [5–7, 9, 10]. In [9], the first-order contexts are also determined adaptively. A linearized binary tree can be used to store the skew numbers required to encode the decomposed 8-bit byte as eight encodings, each of a binary event.

Arithmetic codes have been employed to design codes for a *constrained channel.* A constrained channel has an alphabet, e.g. {0,1}, where not all strings in {0,1}* are allowed. The Manchester code, where clocking and data information are recorded together, is an example of a constrained channel code used in magnetic recording channels. Shannon [11] studied the constrained channel, defined the constraints in terms of allowed transitions in a state table, and determined the probabilities for the allowed transitions which are needed in order to map the maximum amount of information to a set of allowed strings. Guazzo [12] showed that mapping data strings to channel strings is analogous to the decompression operation, and recovering the data string from the channel string corresponds to the compression operation. Martin et al. [1] showed that the dual of *decodability* for compression coding is *representability* for constrained channel coding. To be representable, the encoding operation can have overlaps but cannot have a gap. Some interesting L-based codes for constrained channels are derived in [13].

Guazzo [12], in addition to suggesting the application to constrained-channel codes, contributed a code which subdivides the code space according to the given probabilities. However, the subdivision method is quite crude compared to Pasco's [5].

4. Comments

Arithmetic codes can achieve compression as close to the ideal as desired, for given statistics. In addition, the P-based FIFO arithmetic codes which accept statistics directly facilitate dynamic adaptation to the statistics in one pass of the data [3]. A good binary code is important, as Shannon and others have noted, because other alphabets can be converted to binary form by decomposition.

Of overriding importance to compression now is the modeling of the data. Rissanen and Langdon [14] have studied a framework for the encoding of data strings and have assigned a cost to a model based on the coding parameters required. Different modeling approaches may be compared. They showed that blocking to form larger alphabets results in the same model entropy at the same cost in coding parameters as a symbolwise approach. In general, the compression system designer seeks ways to give up a small percentage of the ideal compression in order to simplify the implementation. The existence of an efficient coding technique now places the emphasis on efficient context selection and parameter-reduction techniques [14].

Acknowledgments

Most of the author's work in this field was done jointly with J. Rissanen, and this debt is obvious. I also owe a debt to Joan Mitchell, who has made several contributions to arithmetic coding [15]. If this paper is more accessible to the general reader, it is due to Joan's gracious and patient encouragement. I thank Gerry Goertzel for his insights in explaining the operation of arithmetic coding. I have also benefited from the encouragement of Janet Kelly, Nigel Martin, Stephen Todd, Ron Arps, and Murali Varanasi.

Appendix 1: Origins of arithmetic coding

The first step toward arithmetic coding was taken by Shannon [11], who observed in a 1948 paper that messages N symbols long could be encoded by first sorting the messages in order of their probabilities and then sending the cumulative probability of the preceding messages in the ordering. The code string was a binary fraction and was decoded by magnitude comparison. The next step was taken by Peter Elias in an unpublished result; Abramson [16] described Elias' improvement in 1963 in a note at the end of a chapter. Elias observed that Shannon's scheme worked without sorting the messages, and that the cumulative probability of a message of N symbols could be recursively calculated from individual symbol probabilities and the cumulative probability of the message of $N - 1$ symbols. Elias' code was studied by Jelinek [17]. The codes of Shannon and Elias suffered from a serious problem: As the message increased in length the arithmetic involved required *increasing precision.* By using fixed-width arithmetic units for these codes, the time to encode each symbol is increased linearly with the length of the code string.

Meanwhile, another approach to coding was having a similar problem with precision. In 1972, Schalkwijk [18] studied coding from the standpoint of providing an index to the encoded string within a set of possible strings. As symbols were added to the string, the index increased in size. This is a *last-in-first-out* (LIFO) code, because the last symbol encoded was the first symbol decoded. Cover [19] made improvements to this scheme, which is now called *enumerative coding.* These codes suffered from the same precision problem.

Both Shannon's code and the Schalkwijk-Cover code can be viewed as a mapping of strings to a number, forming two branches of pre-arithmetic codes, called FIFO (*first-in-first-out*) and LIFO. Both branches use a double recursion, and both have a precision problem. Rissanen [8] alleviated the precision problem by suitable approximations in designing a LIFO arithmetic code. Code strings of any length could be generated with a fixed calculation time per data symbol using fixed-precision arithmetic.

Pasco [5] discovered a FIFO arithmetic code, discussed earlier, which controlled the precision problem by essentially the same idea proposed by Rissanen. In Pasco's work, the code string was kept in computer memory until the last symbol was encoded. This strategy allowed a carry-over to be propagated over a long carry chain. Pasco [5] also conjectured on the family of arithmetic codes based on their mechanization.

In Rissanen [8] and Pasco [5], the original (given, or presumed) symbol probabilities were used. (In practice, we use estimates of the relative frequencies. However, the notion of an imaginary "source" emitting symbols according to given probabilities is commonly found in the coding literature.) In [20] and [3], Rissanen and Langdon introduced the notion of *coding parameters* "based" on the symbol probabilities. The uncoupling of the coding parameters from the symbol probabilities simplifies the implementation of the code at very little compression loss, and gives the code designer some tradeoff possibilities. In [20] it was stated that there were ways to block the carry-over, and in [3] bit-stuffing was presented. In [10] F. Rubin also improved Pasco's code by preventing carry-overs. The result was called a "stream" code. Jones [7] and Martin [6] have independently discovered P-based FIFO arithmetic codes.

Rissanen and Langdon [20] successfully generalized and characterized the family of arithmetic codes through the notion of the decodability criterion which applies to all such codes, be they LIFO or FIFO, L-based or P-based. The arithmetic coding family is seen to be a practical generalization of many pre-arithmetic coding algorithms, including Elias' code, Schalkwijk [18], and Cover [19]. Gilbert and Moore [21] devised the prefix coding approach used in Table 1. In [22], Rissanen presents an interesting view of an arithmetic code as a number-representation system, and shows that Elias' code and enumerative codes are duals.

References

1. G. Nigel N. Martin, Glen G. Langdon, Jr., and Stephen J. P. Todd, "Arithmetic Codes for Constrained Channels," *IBM J. Res. Develop.* **27,** 94–106 (March 1983).
2. G. G. Langdon, Jr. and J. Rissanen, "A Simple General Binary Source Code," *IEEE Trans. Info. Theory* **IT-28,** 800–803 (September 1982).
3. Glen G. Langdon, Jr. and Jorma Rissanen, "Compression of Black-White Images with Arithmetic Coding," *IEEE Trans. Commun.* **COM-29,** 858–867 (June 1981).
4. S. W. Golomb, "Run-Length Encoding," *IEEE Trans. Info. Theory* **IT-12,** 399–401 (July 1966).
5. R. Pasco, "Source Coding Algorithms for Fast Data Compression," Ph.D. Thesis, Department of Electrical Engineering, Stanford University, CA, 1976.
6. G. N. N. Martin, "Range Encoding: an Algorithm for Removing Redundancy from a Digitized Message," presented at the Video and Data Recording Conference, Southampton, England, July 1979.
7. C. B. Jones, "An Efficient Coding System for Long Source Sequences," *IEEE Trans. Info. Theory* **IT-27,** 280–291 (May 1981).
8. J. J. Rissanen, "Generalized Kraft Inequality and Arithmetic Coding," *IBM J. Res. Develop.* **20,** 198–203 (1976).
9. Glen G. Langdon, Jr. and Jorma J. Rissanen, "A Double Adaptive File Compression Algorithm," *IEEE Trans. Commun.* **COM-31,** 1253–1255 (1983).
10. Frank Rubin, "Arithmetic Stream Coding Using Fixed Precision Registers," *IEEE Trans. Info. Theory* **IT-25,** 672–675 (November 1979).
11. C. Shannon, "A Mathematical Theory of Communication," *Bell Syst. Tech. J.* **27,** 379–423 (July 1948).
12. Mauro Guazzo, "A General Minimum-Redundancy Source-Coding Algorithm," *IEEE Trans. Info. Theory* **IT-26,** 15–25 (January 1980).
13. Stephen J. P. Todd, Glen G. Langdon, Jr., and G. Nigel N. Martin, "A General Fixed Rate Arithmetic Coding Method for Constrained Channels," *IBM J. Res. Develop.* **27,** 107–115 (1983).
14. J. J. Rissanen and G. Langdon, "Universal Modeling and Coding," *IEEE Trans. Info. Theory* **IT-27,** 12–23 (January 1981).
15. D. Anastassiou, M. K. Brown, H. C. Jones, J. L. Mitchell, W. B. Pennebaker, and K. S. Pennington, "Series/1-Based Videoconferencing System," *IBM Syst. J.* **22,** 97–110 (1983).
16. N. Abramson, *Information Theory and Coding*, McGraw-Hill Book Co., Inc., New York, 1963.
17. F. Jelinek, *Probabilistic Information Theory*, McGraw-Hill Book Co., Inc., New York, 1968.
18. J. Schalkwijk, "An Algorithm for Source Coding," *IEEE Trans. Info. Theory* **IT-18,** 395 (1972).
19. T. M. Cover, "Enumerative Source Coding," *IEEE Trans. Info. Theory* **IT-19,** 73 (1973).
20. J. J. Rissanen and G. G. Langdon, Jr., "Arithmetic Coding," *IBM J. Res. Develop.* **23,** 149–162 (1979).
21. E. N. Gilbert and E. F. Moore, "Variable-Length Binary Encodings," *Bell Syst. Tech. J.* **38,** 933 (1959).
22. J. Rissanen, "Arithmetic Coding as Number Representations," *Acta Polyt. Scandinavica Math.* **34,** 44–51 (December 1979).

Received June 29, 1983; revised October 8, 1983

Glen G. Langdon, Jr. *IBM Research Division, 5600 Cottle Road, San Jose, California 95193.* Dr. Langdon received the B.S. from Washington State University, Pullman, in 1957, the M.S. from the University of Pittsburgh, Pennsylvania, in 1963, and the Ph.D. from Syracuse University, New York, in 1968, all in electrical engineering. He worked for Westinghouse on instrumentation and data logging from 1961 to 1962 and was an application programmer for the PRODAC computer for process control for most of 1963. In 1963 he joined IBM at the Endicott, New York, development laboratory, where he did logic design on small computers. In 1965 he received an IBM Resident Study Fellowship. On his return from Syracuse University, he was involved in future system architectures and storage subsystem design. During 1971, 1972, and part of 1973, he was a Visiting Professor at the University of Sao Paulo, Brazil, where he developed graduate courses on computer design, design automation, microprogramming, operating systems, and MOS technology. The first Brazilian computer, called Patinho Feio (Ugly Duckling), was developed by the students at the University of Sao Paulo during his stay. He is author of *Logic Design: A Review of Theory and Practice*, an ACM monograph, and coauthor of the Brazilian text *Projecto de Sistemas Digitais*; he has recently published *Computer Design.* He joined the IBM Research laboratory in 1974 to work on distributed systems and later on stand-alone color graphic systems. He has taught graduate courses on logic and computer design at the University of Santa Clara, California. He is currently working in data compression. Dr. Langdon received an IBM Outstanding Innovation Award for his contributions to arithmetic coding compression techniques. He holds eight patents.