

High Efficiency, Multiplication Free Approximation of Arithmetic Coding

Dan Chevion, Ehud D. Karnin, Eugene Walach

*IBM Science and Technology, Technion City, Haifa 32000, Israel
email: karnin@israearn.bitnet*

Introduction and Background

Arithmetic coding is a compression method that maps a sequence of symbols to an interval of real numbers [1], [2]. If the mapping is so constructed that some initial interval is divided among all possible messages (sequences), allocating to each message a subinterval that is proportional to its probability, compression down to the source entropy is reached. Obviously this can be achieved only if we know the correct probabilities. It was shown in [1] that the issues of determining the probabilities and the coding itself can be separated, and here we deal with the second issue only.

The coding process starts with an initial semi-open interval $I = [0, 1)$, which is recursively subdivided according to the (conditional) probabilities of the incoming symbols. That is, as each symbol is processed the interval is narrowed down by a factor which is the symbol probability (given the current, already processed prefix of the sequence). The coding process keeps track of the size of the interval at stage n , denoted by A_n , and the left point of the interval denoted C_n , where initially $C_0 = 0$ and $A_0 = 1$. When the process is done, it suffices to specify the value of the last C_n to the precision of the last A_n , i.e., with $-\log A_n$ bits (the log is taken to the base 2). (This reiterates the claim that if the subdivision of A is carried out according to the correct symbol probabilities, the source entropy is achieved).

The encoding process calls for the computation of A and C for every new symbol in the sequence. Let m be the number of symbols in the alphabet, with probabilities (or actually the conditional probabilities at stage n) p_i , $i = 1, \dots, m$. Say the next symbol to be encoded is number k , then:

$$A_{n+1} = A_n p_k$$
$$C_{n+1} = C_n + A_n S_k$$

where we define $S_1 = 0$ and $S_k = \sum_{j=1}^{k-1} p_j$ for $2 \leq k \leq m$.

Note, that this encoding process can be reversed if the values of C_n and p_k are known. That is, the specific value of k can be found, and the original input sequence can be iteratively reconstructed, provided the decoder keeps track of A_n and C_n .

The arithmetic operations do not reveal how to carry out the computation in a practical manner. In particular, it seems that the precision of the computation increases with decreasing A_n . Rissanen and Langdon proposed an elegant solution (see [1], [2]), which enables to perform the necessary computations on a practical, finite precision machine. The basic idea is to have, at each stage, an effective normalization process such that the value of the parameter A_n will always stay within the interval $[0.5, 1)$. Thus A and C are kept in finite length registers (say N bits long), and whenever A falls below 0.5, it is left shifted J positions, the number of leading zeroes in A . At the same time C is left shifted J positions too, meaning that J bits of the code are generated.

Also, for practical coders it is important to devise the most effective implementation from the computational point of view. Reviewing the aforementioned algorithm, it is easy to see that the computational bottleneck lies in the fact that at each step (i.e., for every input symbol) at least two multiplications are required. These multiplications will prolong both encoding and decoding procedures on many existing machine architectures, which do not have a multiplier. Also, special purpose hardware is much simpler when multiplication is avoided.

Accordingly, a number of methods have been proposed to reduce the computational complexity at the expense of adopting certain approximations, which in turn, cause some deterioration in the overall performance. For the binary case Langdon and Rissanen suggested to approximate the probability of the least probable symbol by an exact negative power of two [3]. Multiplication is thus substituted by a single shift operation.

Another binary arithmetic coder is the Q-Coder [4], which was adopted in the proposed standards for image compression [5]. The basic idea there is to approximate the value of A (which is kept normalized) by a constant intermediate value. By changing the range of A to $[0.75, 1.5)$ this value is set to 1, so multiplication is avoided.

A multi-level, multiplication free arithmetic coding was devised by Rissanen and Mohuiddin [6]. The idea is to set the normalized interval length A to either 0.5 or 1 (or any other two constants with ratio 1:2) as a case may be. As a result simple shift operations replace the multiplications, after the condition for setting A to either 0 or 1 has been checked.

The purpose of this work is to propose an alternative solution, which is applicable to both the binary and the multi-alphabet case. In the next section we describe the procedure, and follow with an analysis of its performance, with respect to the (small) increase in the code length. It turns out that the method is both computationally simpler and more efficient (closer to the theoretical optimum) than the previously discussed methods.

Description of the Coding Procedure

Since the parameter A_n is always in the range $[0.5, 1)$, we can represent A_n by 0.5 plus a certain increment x . In order to reduce the computational complexity we will approximate x by an exponent of 2. In such a manner appropriate shifts can be substituted for every multiplication. Detailed compression and decompression procedures will be as follows:

Denote by p'_i ($1 \leq i \leq m$) the probabilities of each of the possible m symbols divided by 2, and let A and C remain as defined above. In the computer A and C are kept in some registers, denoted by the A-register and C-register, respectively. Let S' be defined similarly to S above, but in terms of p'_i , namely $S'_0 = 0$ and $S'_i = \sum_{j=0}^{i-1} p'_j$, $i = 2, \dots, m$.

Encoding:

1. Initialize register C to 0 and A all 1's. (That is, A holds the binary fraction 0.111...111).
2. Let L be 1 plus the number of zeroes between the first two '1'-s on the left of A (e.g., for A = 1011, $L = 2$).
In case there is only one '1', $L = N$, the length of the register A.
To encode the symbol i , do either step 3 or 4, as the case may be.
3. If $1 \leq i < m$, add the number $S'_i + SHIFT_RIGHT_L(S'_i)$ to the contents of register C, load register A with the number $p'_i + SHIFT_RIGHT_L(p'_i)$. Proceed to step 5.
4. If $i = m$, add the number $S'_m + SHIFT_RIGHT_L(S'_m)$ to the contents of register C, and subtract the same number from A.
5. Let J be the number of leading 0's in A. Shift both C and A J positions to the left and fill the vacated positions with 0's.
6. Read the next symbol and go to Step 2.

Decoding:

1. Fill C with the first N symbols of the code string, and A with 1's, which means that $L = 1$.
2. Find the largest i such that $C \geq S'_i + \text{SHIFT_RIGHT_L}(S'_i)$, where C is interpreted as a binary fraction with the binary point (radix point) placed to the left of the register.
The symbol number i is the output of the decoder at the current stage.

3. If $1 \leq i < m$ then

$$C \leftarrow C - S'_i - \text{SHIFT_RIGHT_L}(S'_i)$$

$$A \leftarrow p'_i + \text{SHIFT_RIGHT_L}(p'_i)$$

Go to step 5.

4. If $i = m$ then

$$C \leftarrow C - S'_m - \text{SHIFT_RIGHT_L}(S'_m)$$

$$A \leftarrow A - S'_m - \text{SHIFT_RIGHT_L}(S'_m)$$

5. Let J be the number of leading 0's in A . Shift A J positions to the left, fill the vacated positions with 0's. Shift C J positions to the left, fill vacated positions by the next J bits of the code string.
6. Set L to be the index of the second '1' from left in A , and go to step 2.

The essence of the proposed algorithm is in the novel way of approximating the result of finite precision multiplication of two numbers. The approximation is computed by means of a *single shift* and a single addition. As a result the computational process is very efficient, and at the same time the performance (the degree of compression) is extremely close to that of the theoretical optimum (as we show in the next section).

Shifts and summations are standard operations in every machine architecture, so our scheme is readily applicable implementation both in software and hardware. In addition, one has to compute the value of the parameter L . This can be done very efficiently by means of the appropriate look-up tables, or special purpose hardware (which already exists as an integral part of arithmetic coder, for the purpose of interval normalization).

As mentioned above the scope of this work is limited to the coding phase of the arithmetic coder, and does not deal with modelling and statistical estimation (see

[1]). Nevertheless we comment that this algorithm can be used in conjunction with any of the known statistical adaptation techniques (e.g., with the efficient finite state machines of the Q-Coder [4]). The only modification (which costs nothing) is to keep track of half the symbol probabilities, rather than the probabilities. (This is done to avoid an extra single left shift operation. Alternatively, we could have used an A value which is kept in the range $[1, 2)$.)

Performance Analysis

In this section we derive an upper-bound for the increase in the code length, due to the approximation of the multiplication operation.

If the exact probabilities for the m symbols, p_1, p_2, \dots, p_m , were multiplied by the true value of the A register, an ideal code would result. At each iteration, the code would be lengthened, in average, by the entropy of the source, i.e., by the value of H given by:

$$H = - \sum_{i=1}^m p_i \log p_i \quad (1)$$

(Note: The log is taken to the base 2, to measure the code length in bits). The effect of simplifying the multiplication by doing add/shift operation instead, is equivalent, as we will shortly show, to some deviation from the true probabilities $p_i, i = 1, \dots, m$. When a different set of "probabilities", say $q_i, i = 1, \dots, m$ is assigned, the i -th symbol will add $-\log q_i$ bits to the code. Therefore, in our code

$$\hat{H} = - \sum_{i=1}^m p_i \log q_i \quad (2)$$

The difference between \hat{H} and H , denoted D , is obtained by subtracting (1) from (2), to yield:

$$D = \sum_{i=1}^m p_i \log \frac{p_i}{q_i} \quad (3)$$

Our goal is to upper-bound D , the excess code length that was introduced by our procedure.

Let A be the value of register A at a given step, say $A = 0.10\dots 01xxx$, where x is either 0 or 1. As in the previous section, we define L to be 1 plus the number of zeroes between the first two '1'-s on the left of A . (If there is no second '1', L equals to the length of register A). The effect of replacing the multiplication by

the add/shift operation is equivalent to approximating A by $\hat{A} = 0.10\dots 01000\dots 0000$ (i.e., the x 's in A were replaced by zeroes). The proposed encoding procedure computes the quantities $\hat{A}p_i$, $i = 1, \dots, m-1$, instead of the (exact) products Ap_i . Since $\hat{A}p_i = A(\hat{A}/A)p_i$, we may describe the procedure as the multiplication of the exact contents of A by a set of "probabilities"

$$q_i = \frac{\hat{A}}{A} p_i, \quad i = 1, 2, \dots, m-1 \quad (4.1)$$

The portion of A allocated to the m -th symbol is $A - \hat{A} \sum_{i=1}^{m-1} p_i$. This, along with the definitions of (4.1), means that $q_m = 1 - \sum_{i=1}^{m-1} q_i$ is the correct "probability" for the last symbol. Hence,

$$q_m = 1 - \frac{\hat{A}}{A} \sum_{i=1}^{m-1} p_i = 1 - \frac{\hat{A}}{A} (1 - p_m) \quad (4.2)$$

Substituting (4.1) and (4.2) in (3), we obtain the following expression for D (the increase over the ideal code length):

$$D = \sum_{i=1}^{m-1} p_i \log\left(\frac{p_i}{q_i}\right) + p_m \log\left(\frac{p_m}{q_m}\right) = \sum_{i=1}^{m-1} p_i \log\left(\frac{A}{\hat{A}}\right) + p_m \log \frac{p_m}{1 - \frac{\hat{A}}{A}(1 - p_m)}$$

By adding and subtracting the quantity $p_m \log \frac{A}{\hat{A}}$ we further obtain:

$$D = \log \frac{A}{\hat{A}} - p_m \log \frac{A}{\hat{A}} + p_m \log \frac{p_m}{1 - \frac{\hat{A}}{A}(1 - p_m)} = \log \frac{A}{\hat{A}} - p_m \log \left[\frac{A}{\hat{A}} \frac{1 - \frac{\hat{A}}{A}(1 - p_m)}{p_m} \right]$$

or more compactly

$$D = \log \frac{A}{\hat{A}} - p_m \log \left(1 + \frac{\frac{A}{\hat{A}} - 1}{p_m} \right) \quad (5)$$

By standard calculus techniques it can be shown that D is a monotonically decreasing function of p_m . Since the m -th symbol can be taken as the most probable one, $\frac{1}{m} \leq p_m \leq 1$, and D is upper bounded by

$$D \leq \log \frac{A}{\lambda} - \frac{1}{m} \log \left[1 + m \left(\frac{A}{\lambda} - 1 \right) \right] \quad (6)$$

Note that when the probability of the most probable symbol equals to $\frac{1}{m}$, then all symbols must have the same probability, namely $\frac{1}{m}$ (since they sum up to 1). Hence, the upper bound for D is achieved in the case of a uniform distribution on the alphabet.

Anticipating future needs, we define the following:

$$Y = \frac{A}{\lambda} - 1$$

$$g_m(Y) = \frac{1}{\ln 2} \left[\ln(1 + Y) - \frac{1}{m} \ln(1 + mY) \right]$$

Equation (6) can be now rewritten as

$$D \leq g_m(Y) \quad (7)$$

We now proceed to calculate $E(D)$, the expected value of D with respect to A .

$$E(D) \leq \int_{-\infty}^{\infty} g_m(y) f_Y(y) dy, \quad (8)$$

where $f_Y(y)$ is the probability distribution function of the random variable Y .

Recalling the definition of L we condition $f_Y(y)$ on the *mutually disjoint* events $L = 1, 2, \dots, N$, where N is the number of bits of the A-register

$$f_Y(y) = \sum_{l=1}^N f_Y(y | L=l) P(L=l) \quad (9)$$

The event $L = l$, $l = 1, \dots, N-1$, means that the second '1' in A appears at the $l+1$ position, hence

$$P(L=l) = P\left(\frac{1}{2} + 2^{-(l+1)} \leq A < \frac{1}{2} + 2^{-l}\right) = 2^{-l} \quad (10.1)$$

The case $L = N$ means that there is no second '1' in the A-register, hence A has to be in the range $2^{-1} \leq A < 2^{-1} + 2^{-N}$, and therefore

$$P(L=N) = 2^{-(N-1)}. \quad (10.2)$$

The contents of the A-register is a random variable that is uniformly distributed in $[0.5, 1)$. Given the condition that $L = l$, i.e., $\hat{A} = 2^{-l} + 2^{-(l+1)}$, A is uniformly distributed in $[2^{-l} + 2^{-(l+1)}, 2^{-l} + 2^{-l})$, $l = 1, 2, \dots, N-1$, and in $[2^{-N}, 2^{-N} + 2^{-N})$ for $l = N$. It follows that $Y = A/\hat{A} - 1$ has a uniform distribution too, but in the range $[0, \frac{1}{2^l + 1})$ for $l = 1, 2, \dots, N-1$, and $[0, 2^{-(N-1)})$ for $l = N$. It means that the conditional probability distribution, for a given L , is:

$$f_Y(y|L=l) = \begin{cases} 2^l + 1 & 0 \leq y < \frac{1}{2^l + 1} \\ 0 & \text{otherwise} \end{cases} \quad (11.1)$$

for $l = 1, 2, \dots, N-1$, and

$$f_Y(y|L=N) = \begin{cases} 2^{N-1} & 0 \leq y < \frac{1}{2^{N-1}} \\ 0 & \text{otherwise} \end{cases} \quad (11.2)$$

Plugging equations (10) and (11) into (9) yields

$$E(D) \leq \sum_{l=1}^{N-1} \int_0^{\frac{1}{2^l+1}} g_m(y) (2^l + 1) 2^{-l} dy + \int_0^{\frac{1}{2^{N-1}}} g_m(y) 2^{N-1} \frac{1}{2^{N-1}} dy \quad (12)$$

Define $G_m(x) = \int_0^x g_m(y) dy$. By standard calculus techniques one can show that

$$G_m(x) = \frac{1}{\ln 2} [(1+x) \ln(1+x) - \frac{1}{m^2} (1+mx) \ln(1+mx) - (1 - \frac{1}{m})x]. \quad (13)$$

Using this definition (12) can be rewritten as:

$$E(D) \leq \sum_{l=1}^{N-1} (1 + 2^{-l}) G_m(\frac{1}{2^l + 1}) + G_m(\frac{1}{2^{N-1}}). \quad (14)$$

The expression above can be easily computed, so we used it to evaluate the efficiency of our procedure for various alphabet sizes. The values tabulated below are the upper-bounds for $E(D)/\log m$, i.e., they are normalized to the entropy of the source (the ideal code length).

m	2	12	22	32	52	72	256
(E(D)/log m)%	1.101	1.747	1.866	1.903	1.911	1.896	1.735

It is interesting to see that the worst-case in binary alphabet needs only 1.1% more bits than the ideal code length (i.e., worst case efficiency is 98.9%) For a general m , the worst-case, which appears in a case of a uniform distribution, is only 1.91%. For the multi-alphabet, multiplication free method described in [6] the authors (in private communication) reported a worst case of 12%.

We have also looked at the important case of binary alphabet, with non-equal probabilities. Tabulated below is the performance, i.e. $E(D)/H$, for various values of p , the probability of the least probable symbol. (We used equation (5), and the expectation with respect to A was computed using the same techniques by which (14) was derived, i.e., just put $m = \frac{1}{1-p}$ in (14)).

P	0.025	0.075	0.125	0.225	0.325	0.425	0.475
$\frac{\hat{H} - H}{H}$ %	0.187	0.257	0.318	0.449	0.616	0.851	1.010

These results can be compared to the algorithm reported in [3], which is analyzed in [7]. The worst case there is about 3.5%, and occurs near $p = 0.4$.

Last remark is about N, the length of the A-register. Our calculations were carried out for N=12, but the changes due to increasing N were negligible (smaller than 10^{-10}).

References

- [1] J. Rissanen and G. G. Langdon.
Universal Modeling and Coding.
IEEE Transactions on Information Theory, IT-27(1):12-23, 1981.
- [2] G. G. Langdon.
An Introduction to Arithmetic Coding.
IBM Journal of Research and Development, 28(2):135-149, 1984.
- [3] G. G. Langdon and J. Rissanen.
A Simple General Binary Source Code.
IEEE Transactions on Information Theory, IT-28(5):800-803, 1982.
- [4] W. B. Pennebaker, J. L. Mitchel, G. G. Langdon, and R. B. Arps.
An Overview of the Basic Principles of the Q-Coder.
IBM Journal of Research and Development, 32(6):717-726, 1988.
- [5] Joint Photographic Experts Group of the ISO.
JPEG Technical Specification, Revision 8.
ISO/IEC, August 14 1990.
- [6] J. Rissanen and K. Mohuiddin.
U.S. Patent 4,652,856.
IBM, 1987.
- [7] G. G. Langdon and J. Rissanen.
Compression of Black-White Images with Arithmetic Coding.
IEEE Transactions on Communications, COM-29(6):858-867, 1981.