CHAPTER 3:

Arithmetic Coding Applied to Quantization

The previous chapter discussed the performance of scalar quantization with entropy coding, under the assumption that the entropy coding could be performed very efficiently. This chapter justifies that assumption by describing an extremely efficient coding technique. In particular, this chapter presents an easily realizable *arithmetic* code that is well-suited to the task of encoding quantized samples.

Sections 3.1–3.5 provide detailed theoretical and implementation-level discussions of a particular arithmetic code for an arbitrary source having an alphabet of K letters. This code is referred to as a *K-ary arithmetic code,* to distinguish it from the more widely used (and perhaps more widely understood) arithmetic codes designed for binary source alphabets [32][47][48]. Although ultimately the code is to be applied to the scalar quantization problem for a memoryless source, the discussion in these first sections does not assume independence of source letters; this is to make the material useful to those interested in high-performance entropy coding in the general case. The K-ary arithmetic code presented in these sections can be used, without modification, in a variety of entropy-coding problems. A computer program (in the C programming language) to implement the code is provided in Appendix C.

Section 3.6 discusses the application of the code to quantization. Results of computer simulations are presented in that section; the simulations demonstrate that the performance of the code is very close to that of an optimal code when applied to quantization of Gaussian, Laplacian, and gamma memoryless sources over a broad range of transmission rates.

The material presented in this chapter is intended to be self-contained and precise, but not tutorial. Sections 3.1-3, in particular, should be read slowly. For a simplified treatment of arithmetic coding, along with a description of its origins, the reader is referred to Langdon [16]. More theoretically oriented treatments are given by Gallager

in [49] and Rissanen and Langdon in [50] and [51].

The particular arithmetic code described in this chapter is based on a strategy outlined in [51], although the derivation and analysis of the code presented here are original. The main contribution of the present work lies in the application of arithmetic coding to scalar quantization in a way that essentially solves the source coding problem in that application. The suitability of arithmetic coding in the quantization application was apparently first recognized by Brusewitz [52] in 1983, then independently by the author [53] in 1985, and most recently by Chou *et al.* in 1989.[1] It should be noted that Brusewitz' and Chou's discussions of arithmetic coding applied to quantization consist of a single paragraph and a single sentence, respectively; neither explores implementation or performance issues in any detail.

Secondary contributions of this work are

- in supplying various important details not given in [51] or elsewhere in the literature (these details were worked out by the present author.)
- in providing an algorithm for rounding-off source letter probabilities to admissible values (finite-precision values that allow unique decodability) in a way that minimizes the attendant increase in the average number of code bits.

3.1 Notation and Basic Entropy-Coding Concepts

This section presents preliminary material that will simplify discussion of K-ary arithmetic coding.

Let $\mathbf{u}^N = [u_1, \ldots, u_N]$ denote a sequence of N discrete random variables or *letters* emitted by a source, where each letter u_i is a random selection from the alphabet $A = \{a_1, \ldots, a_K\}$. The sequence $\mathbf{u}^N$ can be regarded as a random selection from the *superalphabet* $A^N = \{\mathbf{a}_1^N, \ldots, \mathbf{a}_{K^N}^N\}$, where each $\mathbf{a}_l^N$ represents one of the K^N possible

[1] Recently LeGall and Tabatabai [54] have suggested applying a hybrid PCM and self-adapting binary arithmetic code to the high-frequency subbands in an image coding system. Their scheme is similar to a run length code proposed by Gallager [55], and has similar limitations (i.e., it works well at extremely low bit-rates, but becomes either prohibitively complicated or else inefficient at medium bit-rates — see Section 3.7). The approach presented here is considerably more general.

sequences of N letters (in this notation, the order of indexing of the sequences in A^N is arbitrary).

Let $U_i \in A$ be an experimental outcome of the letter u_i; let $\mathbf{U}^N = [U_1, \ldots, U_N]$ be an experimental outcome of the random sequence $\mathbf{u}^N$; and let $\mathbf{U}^0$ be defined as the null or empty sequence. Here, the experimental outcomes $\mathbf{U}^i$ and $\mathbf{U}^j$, $0 < i, j \leq N$, always refer to the *same* experiment, so that, for example, $\mathbf{U}^i$ is obtained simply by appending U_i to $\mathbf{U}^{i-1}$.

Define $P_i(U_i)$ to be the nonzero[2] probability that, in a particular experiment, the i^{th} letter in $\mathbf{u}^N$ assumes the value U_i, conditioned on the values assumed by all previous letters. That is, define

$$P_i(U_i) = \Pr\{u_i = U_i \,|\, \mathbf{U}^{i-1}\}. \tag{3.1}$$

By the chain rule of probability [7, p. 22], the probability $P_{\mathbf{u}^N}(\mathbf{U}^N)$ that $\mathbf{u}^N$ assumes the value $\mathbf{U}^N$ can be written

$$P_{\mathbf{u}^N}(\mathbf{U}^N) = \Pr\{\mathbf{u}^N = \mathbf{U}^N\} = P_1(U_1)P_2(U_2)\cdots P_N(U_N). \tag{3.2}$$

In the special case where $P_i(a_k) = P_j(a_k)$ for all possible sequences $\mathbf{U}^N$, for $i, j \in \{1, \ldots, N\}$, and for $k = 1, \ldots, K$, the sequence is said to be *stationary* (letter probabilities are independent of shifts of the time axis) and *memoryless* (letter probabilities are independent of all previous letters).

The *self-information* $I(\omega)$ associated with an event ω, when expressed in bits, is given by $I(\omega) = -\log_2 \Pr\{\omega\}$. Thus, the self-information associated with the particular sequence $\mathbf{U}^N$ is

$$\begin{aligned} I(\mathbf{U}^N) &= -\log_2 P_{\mathbf{u}^N}(\mathbf{U}^N) \\ &= -\sum_{i=1}^{N} \log_2 P_i(U_i). \end{aligned} \tag{3.3}$$

[2] The requirement that none of the probabilities be zero is for simplicity and is not essential. It can be eliminated at the expense of a more cumbersome notation. In particular, it can be eliminated if the source alphabet is allowed to change with i to exclude the letters with zero probability.

Each term in the sum in (3.3) can be regarded as the incremental amount of uncertainty removed by learning, in sequence, that $u_i = U_i$.

The expected value of the self-information $I(\mathbf{U}^N)$ is the *entropy* of $\mathbf{u}^N$, denoted $H(\mathbf{u}^N)$, and has the important interpretation as being the theoretical minimum number of bits needed, on the average, to represent $\mathbf{u}^N$ exactly.[3] Thus, the entropy serves as a performance limit/target for the average number of bits produced in coding a sequence of a given length N.

Besides the entropy of the entire sequence $\mathbf{u}^N$, of interest in the present application is the *per-letter entropy*, defined here as $H_N(\mathbf{u}^N) = H(\mathbf{u}^N)/N$. This quantity has a special significance in the case of stationary memoryless sequences — it is simply the entropy of the individual letters as discussed in the previous chapter. In the case of such sequences, the performance of the code can be assessed by comparing the average number of code bits produced per source letter to $H_N(\mathbf{u}^N)$. In particular, if the number of bits generated in encoding $\mathbf{a}_i^N$ is $L(\mathbf{a}_i^N)$, then the inefficiency γ_N of the code for $\mathbf{u}^N$ can be defined as

$$\gamma_N = E\{L(\mathbf{U}^N)\}/N - H_N(\mathbf{u}^N), \tag{3.4}$$

where $E\{\cdot\}$ denotes expectation.

Finally, note that if a code produces a number of bits that is always close to the self-information of the particular sequence encoded, then the average number of bits produced will be close to the entropy; i.e., the performance of the code will be nearly optimal. The next section describes a coding technique that always produces less than $I(\mathbf{U}^N) + 1$ bits in encoding the sequence $\mathbf{U}^N$, and hence has nearly optimal performance. That technique, though impractical for a variety of reasons, will serve to illustrate the basic concepts that underlie a practical K-ary arithmetic code, which will be discussed subsequently.

[3] This is the same as the definition of entropy used in the previous chapters, except now the probability space is the joint space for a sequence of letters instead of the space for a single letter.

3.2 The Elias-Gallager-Shannon Sequential Arithmetic Code

Consider a one-to-one mapping of A^N (the set of all possible sequences of length N) onto a set of adjacent and disjoint (non-overlapping) subintervals of the unit interval,[4] where the mapping is done in such a way that the width of each subinterval is equal to the probability of the corresponding sequence. Since the subintervals do not overlap, a particular subinterval can be uniquely specified by any number contained within it. Thus, a sequence can be encoded simply by first selecting a suitable number in the corresponding subinterval, then representing that number by expanding it as a binary fraction in the usual way (with the first bit representing the 1/2's place, the second representing the 1/4's place, and so on).

The efficiency of such a code derives from the fact that if a subinterval of the unit interval has width W, then it contains at least one number θ of the form[5]

$$\theta = \frac{j}{2^{\lceil -\log_2 W \rceil}}, \tag{3.5}$$

where j is a nonnegative integer less than $2^{\lceil -\log_2 W \rceil}$. This is easily seen by noting that the spacing between two adjacent such numbers is $2^{\lfloor \log_2 W \rfloor} \leq W$. Note that any number of the form indicated in (3.5) can be represented by $\lceil -\log_2 W \rceil$ bits. Thus, with the representative number for each subinterval appropriately chosen, the number of bits $L(\mathbf{U}^N)$ needed to represent $\mathbf{U}^N$ is given by

$$L(\mathbf{U}^N) = \lceil I(\mathbf{U}^N) \rceil, \tag{3.6}$$

which implies that

$$I(\mathbf{U}^N) \leq L(\mathbf{U}^N) < I(\mathbf{U}^N) + 1. \tag{3.7}$$

This becomes, upon taking expectations,

$$H(\mathbf{u}^N) \leq E\{L(\mathbf{u}^N)\} < H(\mathbf{u}^N) + 1. \tag{3.8}$$

[4] Throughout, the term 'interval' refers to a right-semi-open interval — that is, intervals are of the form $[a, b)$. Moreover, the width of $[a, b)$ is defined as $b - a$.

[5] The ceiling of x, denoted $\lceil x \rceil$, is defined to be the least integer not less than x. Similarly, the floor of x, denoted $\lfloor x \rfloor$, is defined to be the greatest integer not greater than x.

For stationary and memoryless sequences, dividing by N and subtracting the per-letter entropy from the three parts of this inequality bounds the inefficiency of the code as

$$0 \leq \gamma_N < 1/N, \tag{3.9}$$

so that in the limit as N gets large, γ_N approaches 0. That is, the code becomes perfectly efficient as the sequence to be encoded becomes very long.

It is worth noting that such a code does not necessarily satisfy the prefix condition.[6] However, this fact is inconsequential when the objective is to encode a *single* sequence of length N, where N is large. This objective will be assumed for the remainder of the chapter.

Consider now the task of *constructing* a suitable mapping. Recall that the objective is to associate with each sequence $\mathbf{a}_l^N$ an interval $\mathcal{J}(\mathbf{a}_l^N)$ of width $P_{\mathbf{u}^N}(\mathbf{a}_l^N)$, such that $\mathcal{J}(\mathbf{a}_l^N) \cap \mathcal{J}(\mathbf{a}_m^N) = \emptyset$ for all $m \neq l$ (the subintervals are disjoint), and $\bigcup_{l=1}^{K^N} \mathcal{J}(\mathbf{a}_l^N) = [0,1)$ (the union of the subintervals is the unit interval). The most straightforward mapping that has these properties associates each sequence $\mathbf{a}_l^N$ with the subinterval $\mathcal{J}(\mathbf{a}_l^N) = \left[\alpha_l, \alpha_l + P_{\mathbf{u}^N}(\mathbf{a}_l^N)\right)$, where $\alpha_1 = 0$, $\alpha_l = \sum_{m=1}^{l-1} P_{\mathbf{u}^N}(\mathbf{a}_m^N)$ for $l = 2, \ldots, K^N$.[7] Although conceptually very simple, this mapping suffers from three major limitations. First, it requires working directly with the entire superalphabet A^N, which has K^N superletters (sequences) in it. This is impractical for typical values of K and N (e.g., $K = 50$ and $N = 1{,}000$). Second, since the entire sequence must be encoded at once, the first letter cannot be decoded until the N^{th} letter is encoded, so that the coding delay is large. Third, the arithmetic required in the mapping must be carried out with arbitrarily high precision (as determined by the precision required to represent the probabilities $\{P_{\mathbf{u}^N}(\mathbf{a}_1^N), \ldots, P_{\mathbf{u}^N}(\mathbf{a}_{K^N}^N)\}$).

An alternative approach, devised independently by Elias, Gallager, and Shannon, all in unpublished work[56][57, pp. 61-62][58], dispenses with the first limitation cited

[6] A prefix-condition code has the property that no one codeword is a prefix of another codeword.

[7] This mapping is described by Gallager in an exercise [7, pp. 513-514]. Shannon describes a similar mapping in proving a source coding theorem [1, pp. 60-61].

above by accomplishing the mapping and inverse-mapping in a letter-by-letter fashion. The technique avoids working with the entire superalphabet A^N, and instead maps each subsequence $\mathbf{U}^i$, $0 < i \leq N$, to a subinterval $\mathcal{J}(\mathbf{U}^i) = \big[X_i, X_i + W_i\big)$, where X_i and W_i are determined recursively in terms of $\mathcal{J}(\mathbf{U}^{i-1})$, U_i, and the conditional letter probabilities $\{P_i(a_k)\}$. In particular, $\mathcal{J}(\mathbf{U}^i)$ is determined by the joint recursion formulas

$$X_i = X_{i-1} + W_{i-1} \sum_{j<k:\, a_k = U_i} P_i(a_j) \tag{3.10 a}$$

and

$$W_i = P_i(U_i) W_{i-1}, \tag{3.10 b}$$

for $i = 1, \ldots, N$, where $X_0 \stackrel{\text{def}}{=} 0$ and $W_0 \stackrel{\text{def}}{=} 1$. The notation beneath the summation operation in (3.10 a) should be read, "all $j < k$, where k satisfies $a_k = U_i$."

The procedure specified by the recursion formulas (3.10 a) and (3.10 b) can be visualized as follows. Initially, the unit interval is partitioned into K adjacent, disjoint subintervals, one for each possible value of the first letter in the sequence. The partitioning is done in such a way that the width of each subinterval is equal to the probability of the first letter assuming the value corresponding to that subinterval. The representative subinterval $\mathcal{J}(\mathbf{U}^1)$ for the initial subsequence $\mathbf{U}^1 = [U_1]$ is taken to be the subinterval that corresponds to U_1. Next, $\mathcal{J}(\mathbf{U}^1)$ is itself partitioned into K adjacent, disjoint subintervals, one for each possible value of the *second* letter in the sequence. The partitioning is done in such a way that the width of each new subinterval is proportional to the conditional probability of the second letter assuming the value corresponding to that subinterval, given that the first letter was U_1. That is, the widths are $W_1 P_2(a_1), \ldots, W_1 P_2(a_K)$. The representative subinterval $\mathcal{J}(\mathbf{U}^2)$ for the subsequence $\mathbf{U}^2 = [U_1, U_2]$ is then simply taken to be the newly-formed subinterval that corresponds to U_2. This process is carried out for each successive letter in the

source sequence, so that, in general:

> The i^{th} source letter is processed by partitioning $\mathcal{J}(\mathbf{U}^{i-1})$ into K new subintervals, each corresponding to a different value for u_i. The partitioning is done in such a way that the width of each newly-formed subinterval is proportional to the probability of u_i assuming the value corresponding to that subinterval. Of these newly-formed subintervals, the one corresponding to U_i is taken to be $\mathcal{J}(\mathbf{U}^i)$.

From the foregoing description, it is evident that the subintervals formed in this way for all distinct sequences of N letters meet the requirements of being adjacent and disjoint, and have as their union the unit interval. Moreover, from (3.2) and the recursion formula (3.10 b), it is clear that the subinterval corresponding to a particular sequence $\mathbf{U}^N$ has the desired width $P_{\mathbf{u}^N}(\mathbf{U}^N)$.

Once the mapping has been performed — that is, once a representative subinterval $\mathcal{J}(\mathbf{U}^N)$ has been found for the sequence $\mathbf{U}^N$ — all that remains to complete the encoding process is to select some number $\theta(\mathbf{U}^N)$ from $\mathcal{J}(\mathbf{U}^N)$ that has a sufficiently compact binary representation. Recall from the argument presented earlier in this section that $\mathcal{J}(\mathbf{U}^N)$ contains at least one number which, when expressed as a binary fraction in the usual way, consists of no more than $\lceil -\log_2 W_N \rceil$ bits. One of these numbers (the smallest one) can be determined in a systematic way by applying the following rule:

> If X_N can be represented exactly using $\lceil -\log_2 W_N \rceil$ bits, then set $\theta(\mathbf{U}^N) = X_N$. Otherwise, obtain $\theta(\mathbf{U}^N)$ by first truncating X_N to $\lceil -\log_2 W_N \rceil$ bits, then adding $2^{\lfloor \log_2 W_N \rfloor}$ to the result.

Alternatively, if N is very large (i.e., if the source sequence is very long), then it is not critical that $\theta(\mathbf{U}^N)$ be chosen to have the *most* compact binary representation, since a few extra bits required to represent a suboptimally chosen $\theta(\mathbf{U}^N)$ will be "amortized" over the entire sequence. The suboptimal choice will result in a negligible increase in the average number of bits per letter. Note that $X_N \in \mathcal{J}(\mathbf{U}^N)$, so that if X_N happens

to have a reasonably compact binary representation, it can serve as a particularly convenient choice for $\theta(\mathbf{U}^N)$.

Consider now the problem of decoding — recovering $\mathbf{U}^N$ from $\theta(\mathbf{U}^N)$. Again, it is desired that individual letters be processed sequentially to avoid working with the superalphabet A^N. It turns out that sequential decoding is indeed possible, because the following inequality holds for any $i \leq N$:

$$X_{i-1} + W_{i-1} \sum_{j<k:\ a_k = U_i} P_i(a_j) \quad \leq \quad \theta(\mathbf{U}^N) \quad < \quad X_{i-1} + W_{i-1} \sum_{j \leq k:\ a_k = U_i} P_i(a_j). \qquad (3.11)$$

Note that the leftmost quantity in (3.11) is, from (3.10 a), simply X_i. The use of the above inequality in sequential decoding will be discussed shortly; first, it is shown that the inequality is in fact valid.

It is immediately apparent that for (3.11) to be plausible, W_i must be greater than zero for all i and all $\mathbf{U}^N$, which, by (3.10 b), requires that

$$P_i(a_k) \quad > \quad 0 \qquad \text{for } i = 1, \ldots, N \text{ and } k = 1, \ldots, K. \qquad (3.12)$$

This requirement was previously stated in the beginning of Section 3.1.

The first inequality in (3.11) holds because $\theta(\mathbf{U}^N) \geq X_N$ (since $\theta(\mathbf{U}^N) \in \mathcal{J}(\mathbf{U}^N)$) and $X_N \geq X_i$ (since, according to (3.10 a), X_N is obtained by adding nonnegative quantities to X_i). The second inequality in (3.11) is derived by noting that for a given $\mathbf{U}^i$, the maximum value for X_N results when $U_j = a_K$ for $j = i+1, \ldots, N$. The maximum value for X_N can therefore be obtained by applying (3.10 a) and (3.10 b) recursively, substituting a_K for all U_j, $i < j \leq N$. This substitution, along with the observation that

$$\sum_{k=1}^{K-1} P_i(a_k) = 1 - P_i(a_K) \qquad (3.13)$$

for every i yields the inequality

$$\begin{aligned} X_N \;\leq\; & X_{i-1} + W_{i-1} \sum_{j<k:\ a_k = U_i} P_i(a_j) \\ & + W_{i-1} P_i(U_i)[1 - P_{i+1}(a_K)] \\ & + W_{i-1} P_i(U_i) P_{i+1}(a_K)[1 - P_{i+2}(a_K)] \\ & + \ldots + W_{n-1} P_i(U_i) P_{i+1}(a_K) \cdots P_{N-1}(a_K)[1 - P_N(a_K)], \end{aligned}$$

which simplifies to

$$X_N + W_N \leq X_{i-1} + W_{i-1} \sum_{j \leq k:\ a_k = U_i} P_i(a_j). \tag{3.14}$$

Now since the subinterval $\mathcal{J}(\mathbf{U}^N)$ is right-semi-open and its right endpoint is $X_N + W_N$, where $W_N > 0$,

$$\theta(\mathbf{U}^N) < X_N + W_N. \tag{3.15}$$

Combining (3.14) and (3.15) yields the second inequality in (3.11).

It is evident from the above derivation that (3.14) would still hold if (3.13) were replaced by

$$\sum_{k=1}^{K-1} P_i(a_k) \;\leq\; 1 - P_i(a_K), \tag{3.16}$$

and if (3.10 b) were replaced by

$$W_i = P_i(U_i) W_{i-1} - \epsilon, \tag{3.17}$$

where ϵ can be any nonnegative number for which W_i is strictly positive.[8] The first inequality in (3.11) is unaffected by these substitutions since the left-hand side is always equal to X_i, and $\theta(\mathbf{U}^N)$ is never less than X_i, as noted above. These properties will be important in the next section, where the problem of fixed-precision implementation is addressed.

[8] Since both of these modifications potentially reduce the width of the final subinterval, they potentially reduce the performance of the code. Efficiency is discussed in subsequent sections; here, the concern is exclusively with ensuring sequential decodability.

Because of (3.11), the letters can be decoded sequentially as follows. Since it is known initially that $X_0 = 0$ and $W_0 = 1$, the first letter U_1 can be decoded immediately from $\theta(\mathbf{U}^N)$ by finding the unique k for which (3.11) is satisfied [9] (with $i = 1$). Now for any $i > 1$, suppose that the first $i-1$ letters, $U_1, \ldots, U_{i-1}$, have been successfully decoded. From (3.10 a) and (3.10 b), X_{i-1} and W_{i-1} can be determined, and U_i can be decoded from $\theta(\mathbf{U}^N)$ by again finding the unique k for which (3.11) is satisfied. Thus, by beginning with U_1 and applying the above procedure for $i = 2, \ldots, N$, all letters can be decoded in sequence from $\theta(\mathbf{U}^N)$.

It was mentioned previously that one of the limitations of the Elias-Gallager-Shannon code as described above is that the entire sequence of N letters must be encoded before the first letter is decoded. It is possible to modify the encoding process to generate a $\theta(\mathbf{U}^i)$ for every i, with $\theta(\mathbf{U}^1) \leq \theta(\mathbf{U}^2) \leq \ldots \theta(\mathbf{U}^N)$. [10] In this case, the limitation can be replaced by the slightly less severe limitation that it *might* be necessary to wait until *many* letters have been encoded before the first can be decoded [49]. Some difficulty of this type is inherent to the class of codes in which variable-length source strings are mapped to either variable- or fixed-length code strings, (this class includes arithmetic codes). In the present application, the difficulty is ultimately overcome by employing a buffer-underflow/overflow prevention mechanism — an arrangement which guarantees constant-delay coding at the expense of slightly reduced quantization fidelity.

A more serious problem with the code described above is that arithmetic operations must be carried out with arbitrarily high precision. This problem is addressed in the next section.

3.3 A Fixed-Precision K-ary Arithmetic Code

This section discusses modification of the Elias-Gallager-Shannon sequential code so that it can be realized using fixed-precision digital hardware.

[9] Since the second inequality in (3.11) is strict, the value of k is unique.

[10] The use of finite-precision arithmetic, discussed in the next section, indirectly achieves this modification.

Suppose that the code is to be implemented using hardware that is capable of manipulating, storing, and performing arithmetic operations on ρ-bit data. For any nonnegative real x, let $[\![x]\!]_\rho$ denote the result of truncating the binary representation of the fractional part of x to the most significant ρ bits, where the i^{th} most significant bit corresponds to 2^{-i}. That is, define

$$[\![x]\!]_\rho \stackrel{\text{def}}{=} \frac{\lfloor 2^\rho x \rfloor}{2^\rho}.$$

By hypothesis, the hardware is capable of representing $[\![x]\!]_\rho$ exactly.

It is easy to see that simply truncating all the quantities in the recursion formulas (3.10 a) and (3.10 b) and the decoding inequality (3.11) to ρ bits is *not* sufficient to allow fixed-precision implementation of the code. Observe, for example, that simple truncation would almost certainly give rise to zero values for W_i as i increases, which in turn would invalidate (3.11).

As it turns out, fixed-precision implementation of the code is considerably more involved than simple truncation. The key lies in the following three properties of the Elias-Gallager-Shannon code:

(1) For sequential decodability, any values that meet requirements (3.12) and (3.16) can be used in place of $P_i(a_k)$; the actual conditional probabilities need not be used.

(2) If X_i and W_i were to be represented in binary (with unlimited precision), then, except for occasional carry-overs, the addition in (3.10 a) would not affect bits in X_i that are more significant than the most significant nonzero bit of W_{i-1}.

(3) The recursion formula for W_i, (3.10 b), can be replaced by (3.17) without affecting sequential decodability (as was observed in the previous section).

Property (1) allows the conditional source letter probabilities to be rounded-off to any convenient fixed-precision values without affecting decodability,[11] as long as

[11] The price paid for the rounding-off is some reduction in coding efficiency. This is discussed in detail in Section 3.5.

each of the rounded-off values is greater than zero and the sum of the rounded-off values does not exceed unity. A rounded-off value for $P_i(a_k)$ will be denoted $Q_i(a_k)$. Accordingly, requirements (3.16) and (3.12) can be rewritten

$$\sum_{k=1}^{K} Q_i(a_k) \;\leq\; 1; \tag{3.18}$$

$$Q_i(a_k) \;>\; 0 \qquad \text{for } i = 1, \ldots, N \text{ and } k = 1, \ldots, K. \tag{3.19}$$

Also, the encoding recursions (3.10 a) and (3.17) can be rewritten

$$X_i = X_{i-1} + W_{i-1} \sum_{j<k:\, a_k = U_i} Q_i(a_j) \tag{3.20 a}$$

and

$$W_i = Q_i(U_i) W_{i-1} - \epsilon, \tag{3.20 b}$$

for $i = 1, \ldots, N$, where $X_0 \stackrel{\text{def}}{=} 0$ and $W_0 \stackrel{\text{def}}{=} 1$, and ϵ can be any nonnegative number for which W_i is strictly positive.

Property (2) allows some of the bits in the binary representation of X_i — namely the most significant bits that are unchanging — to be removed from the arithmetic unit of a hypothetical unlimited-precision processor and set aside in a buffer.[12] Equivalently, this property allows the binary representations of X_i and W_i to be jointly shifted to the left (each multiplied by two) after each recursion repeatedly until the unchanging most significant bits of X_i have been shifted to lie to the left of the radix point (that is, until the most significant nonzero bit of W_i occupies the 1/2's position), whereupon the integer portion of X_i may be safely set aside. As long as X_i and W_i are shifted or *renormalized* in this way jointly, X_N will be computed correctly to within a power-of-two scaling factor, and this factor can be determined automatically during decoding by using the same renormalizing procedure.

For concreteness, let s_i denote the number of places that X_i and W_i are jointly shifted during renormalization in processing the i^{th} letter, and let $S_i = \sum_{j=1}^{i} s_i$ denote

[12] The problem of carry-overs is temporarily ignored — it will be addressed shortly.

the total number of places shifted for all letters up to and including the i^{th} one. Then (3.20 a) and (3.20 b) become

$$X_i = 2^{s_i}\left[X_{i-1} + W_{i-1}\sum_{j<k:\ a_k=U_i} Q_i(a_j)\right] \tag{3.21 a}$$

and

$$W_i = 2^{s_i}\left[Q_i(U_i)W_{i-1} - \epsilon\right], \tag{3.21 b}$$

for $i = 1, \ldots, N$, where $X_0 \stackrel{\text{def}}{=} 0$ and $W_0 \stackrel{\text{def}}{=} 1$, and ϵ can be any nonnegative number for which W_i is strictly positive. The decoding inequality (3.11) becomes

$$X_{i-1}+W_{i-1}\sum_{j<k:\ a_k=U_i} Q_i(a_j) \ \leq\ 2^{S_{i-1}}\theta(\mathbf{U}^N) \ <\ X_{i-1}+W_{i-1}\sum_{j\leq k:\ a_k=U_i} Q_i(a_j). \tag{3.22}$$

It is emphasized that s_i depends on the bracketed quantity in (3.21 b); the exact relationship can be expressed as

$$s_i = \left\lceil 1 - \log_2[Q_i(U_i)W_{i-1} - \epsilon]\right\rceil, \tag{3.23}$$

although in practice, s_i is determined more readily by the renormalization procedure described above.

Although it is not immediately apparent, renormalization and the freedom to round-off the source letter probabilities combine to make it possible to truncate the term

$$W_{i-1}\sum_{j<k:\ a_k=U_i} Q_i(a_j) \tag{3.24}$$

in (3.21 a) to ρ bits of precision without affecting decodability, provided that for every i, each of the rounded-off values $Q_i(a_k)$, $k = 1\ldots, K$ be greater than $2^{-\rho+1}$. To see this, observe that if $[\![W_{i-1}Q_i(a_k)]\!]_\rho > 0$ for every i and k, then (3.24) is distinct when truncated to ρ bits for every possible value of U^i, and consequently there exists some *other* set of modified probabilities $Q_i'(a_k)$ that satisfy (3.19) and are such that

$$W_{i-1}\sum_{j<k:\ a_k=U_i} Q_i'(a_j) = [\![W_{i-1}\sum_{j<k:\ a_k=U_i} Q_i(a_j)]\!]_\rho \quad \text{for } i = 1, \ldots, N \text{ and } U^i = a_1, \ldots, a_K.$$

Moreover, since it is always true that

$$\lfloor W_{i-1} \sum_{j<k:\, a_k=U_i} Q_i(a_j) \rfloor_\rho \;\; \leq \;\; W_{i-1} \sum_{j<k:\, a_k=U_i} Q_i(a_j),$$

$Q'_i(a_k) \leq Q_i(a_k)$ for every i and k, so that $Q'_i(a_k)$ also satisfies (3.18). In words, the truncation of the term (3.24) in (3.21 a) is mathematically equivalent to rounding-off the probabilities to slightly different — but still admissible — values, so that decodability is preserved. Note that the requirement $\lfloor W_i Q_i(a_k) \rfloor_\rho > 0$ that led to this result is equivalent to the requirement

$$W_i Q_i(a_k) \geq 2^{-\rho} \tag{3.25}$$

for every i and k. After renormalization, $W_i \geq 1/2$, so that (3.25) can be satisfied by ensuring that

$$Q_i(a_k) \;\; \geq \;\; 2^{-\rho+1} \tag{3.26}$$

for every i and k, as was claimed. By substituting $Q'_i(a_j)$ for $Q_i(a_j)$, (3.21 a) becomes

$$X_i = 2^{s_i} \left[X_{i-1} + \lfloor W_{i-1} \sum_{j<k:\, a_k=U_i} Q_i(a_j) \rfloor_\rho \right], \tag{3.27}$$

where $X_0 \stackrel{\text{def}}{=} \lfloor 0 \rfloor_\rho = 0$. Similarly, (3.22) becomes

$$X_{i-1} + \lfloor W_{i-1} \sum_{j<k:\, a_k=U_i} Q_i(a_j) \rfloor_\rho \;\; \leq \;\; 2^{S_{i-1}} \theta(\mathbf{U}^N) \;\; < \;\; X_{i-1} + \lfloor W_{i-1} \sum_{j\leq k:\, a_k=U_i} Q_i(a_j) \rfloor_\rho . \tag{3.28}$$

Observe that since the term (3.24) in (3.27) is truncated to ρ bits at every recursion, $X_i = \lfloor X_i \rfloor_\rho$ for every i (i.e., neither the adding of X_{i-1} nor the renormalization increases precision), so that the addition within the bracketed expression in (3.27) can be carried out using ρ bits of precision.

A similar argument can be used to establish a fixed-precision version of the recursion formula for W_i, (3.21 b). In particular, renormalization and the condition (3.26) imply that $\lfloor Q_i(a_k) W_{i-1} \rfloor_\rho > 0$ for every i and k. But it is always true that

$Q_i(a_k)W_{i-1} \geq \lfloor\!\lfloor Q_i(a_k)W_{i-1} \rfloor\!\rfloor_\rho$, so that an admissible value for ϵ in (3.21 b) is given by

$$\epsilon = Q_i(U_i)W_{i-1} - \lfloor\!\lfloor Q_i(U_i)W_{i-1} \rfloor\!\rfloor_\rho,$$

which allows (3.21 b) to be replaced by

$$W_i = 2^{s_i} \lfloor\!\lfloor Q_i(U_i)W_{i-1} \rfloor\!\rfloor_\rho, \tag{3.29}$$

for $i = 1, \ldots, N$, where $W_0 \stackrel{\text{def}}{=} \lfloor\!\lfloor 1 \rfloor\!\rfloor_\rho = 1 - 2^{-\rho}$.

That (3.27) and (3.29) involve only truncated arithmetic on truncated quantities, allows these formulas to to be computed using fixed-precision hardware.[13]

Attention is now turned to the problem of fixed-precision decoding. Consider first the problem of selecting $\theta(\mathbf{U}^N)$, the representative number for the final subinterval $\mathcal{J}(\mathbf{U}^N)$. Recall that in the previous section it was argued that if X_N has a reasonably compact binary representation, then it can serve as a convenient choice for $\theta(\mathbf{U}^N)$. With the renormalization introduced in the present section, this changes slightly — a scaling factor of 2^{-S_N} must be applied to X_N before it can be a suitable choice for $\theta(\mathbf{U}^N)$. This scaling can be accomplished without *a priori* knowledge of S_N by positioning the radix point to the immediate left of the leftmost bit in the binary representation of X_N.[14] Note that the number of bits that comprise X_N is simply S_N, the total number of shift-left operations performed during the renormalization process in the recursive computation of X_N, plus ρ, the number of bits in the fractional part of X_N. Thus, the total number of bits $L(\mathbf{U}^N)$ needed to represent X_N is

$$L(\mathbf{U}^N) = \rho + \lceil -\log_2[\eta Q_1(U_1)Q_2(U_2)\cdots Q_N(U_N)] \rceil \tag{3.30}$$

[13] The reader may have noticed that if the truncation in (3.29) were postponed to happen *after* the renormalization, i.e., if the product term in (3.21 b) were initially computed as a double-precision quantity and truncated only after shifting to eliminate all leading zeroes, then the necessary conditions for sequential decodability would still be satisfied, and in general a smaller value of ϵ would result, leading to a more efficient code. As will become apparent later in the present section, such an approach would require that comparison operations be carried out with double-precision arithmetic during decoding, so that properly speaking, a 2ρ-bit processing unit would be required.

[14] Henceforth, when the meaning is clear, the words "binary representation of" will be omitted.

where η is some positive factor less than or equal to unity and reflects the reduction in the width of the final subinterval $\mathcal{J}(\mathbf{U}^N)$ due to the use of fixed-precision arithmetic.[15] For very long sequences, the fixed overhead ρ contributes little to the average number of bits per source letter, and can be ignored. Consequently, $2^{-S_N}X_N$ can be used as the representative number $\theta(\mathbf{U}^N)$ without much loss in efficiency beyond that incurred by the rounding-off of probabilities[16] and the truncations in the modified recursion formulas.[17] Henceforth it will be assumed that $2^{-S_N}X_N$ is the number used as $\theta(\mathbf{U}^N)$. Accordingly, (3.28) can be rewritten

$$X_{i-1} + \lfloor W_{i-1} \sum_{j<k:\ a_k=U_i} Q_i(a_j) \rfloor_\rho \ \leq \ 2^{S_{i-1}-S_N}X_N \ < \ X_{i-1} + \lfloor W_{i-1} \sum_{j\leq k:\ a_k=U_i} Q_i(a_j) \rfloor_\rho. \tag{3.31}$$

It is convenient to define a new quantity X_i' as

$$X_i' = 2^{S_{i-1}-S_N}X_N - X_{i-1} \tag{3.32}$$

so that (3.31) becomes

$$\lfloor W_{i-1} \sum_{j<k:\ a_k=U_i} Q_i(a_j) \rfloor_\rho \ \leq \ X_i' \ < \ \lfloor W_{i-1} \sum_{j\leq k:\ a_k=U_i} Q_i(a_j) \rfloor_\rho. \tag{3.33}$$

Combining (3.32) and (3.27) yields, after some algebraic manipulation, the following recursive formula for X_i':

$$X_i' = 2^{s_{i-1}}\left[X_{i-1}' - \lfloor W_{i-2} \sum_{j<k:\ a_k=U_{i-1}} Q_{i-1}(a_j) \rfloor_\rho\right] \tag{3.34}$$

for $i \geq 2$, where $X_1' \stackrel{\text{def}}{=} 2^{-S_N}X_N$. It is instructive to interpret X_i' as being the amount contributed to $2^{S_{i-1}-S_N}X_N$ by the (as yet undecoded) letters $u_i, \ldots, u_N$.

[15] Note that if $Q_i(a_k) = P_i(a_k)$ for all i and k, then $L(\mathbf{U}^N) = \rho + \lceil I(\mathbf{U}^N) - \log_2 \eta \rceil$. By comparing this with (3.4), it can be seen that in the case of very long stationary memoryless sequences, if the probabilities have been rounded off with extremely high precision, then the per-letter inefficiency γ_N becomes the expected value of $-\log_2 \eta$ divided by N.

[16] While the rounding-off of probabilities can either increase or decrease efficiency in coding a particular sequence, it can only decrease efficiency on average. This is shown in a subsequent section, where the problem of rounding-off the probabilities is considered in detail.

[17] Recall from Section 3.1 that if $L(\mathbf{U}^N) = I(\mathbf{U}^N)$ for every $\mathbf{U}^N$ in A^N, then the code would be perfectly efficient.

This interpretation follows readily from (3.32) and (3.27). Note that although X_i' is of arbitrarily high precision, the subtraction in (3.34) involves only the most significant ρ bits, and can therefore be realized with fixed-precision arithmetic in a manner analogous to the computation of X_i in (3.27).

Since $[\![X_i']\!]_\rho \leq X_i'$ for any X_i', (3.33) can be rewritten

$$[\![W_{i-1} \sum_{j<k:\, a_k=U_i} Q_i(a_j)]\!]_\rho \;\leq\; [\![X_i']\!]_\rho \;<\; [\![W_{i-1} \sum_{j\leq k:\, a_k=U_i} Q_i(a_j)]\!]_\rho, \tag{3.35}$$

that is, the decoding inequality remains valid when X_i' is truncated to ρ bits of precision. Thus sequential decoding can in fact be carried out using finite-precision arithmetic.

The main results of the present section can now be summarized. If the source letter probabilities are rounded-off to fixed precision values that satisfy (3.18) and (3.26),[18] then the recursion formulas (3.10 a) and (3.10 b) can be replaced by their fixed-precision counterparts (3.27) and (3.29). Also, the decoding inequality (3.11) can be replaced by its fixed precision counterpart (3.35). This new decoding inequality involves a new fixed-precision quantity X_i' which can be computed by the decoder in place of X_i via (3.34).

Before concluding this section, it is appropriate to make some observations about the modified code.

First, note that the use of rounded-off values for the letter probabilities instead of the actual probabilities affects the number of bits emitted during the renormalization process, and hence affects the efficiency of the code. For this reason, the manner in which the probabilities are rounded-off is is potentially very critical. In the next section, a rounding-off procedure that results in greatest possible code efficiency is presented.

[18] Little has been said in this section about the manner in which the source letter probabilities should be rounded off. This topic is treated at length in the next section, where it will be seen that the rounding-off determines, in large part, the efficiency of the code.

Next, observe that the analysis in the present section was based in part on the assumption that the bits that comprise the integer part of X_{i-1} are unaffected by the addition within the bracketed quantity in (3.27), and that consequently these bits can be removed from the arithmetic unit of the processor and placed in a buffer as X_N is recursively computed. A complication arises when the addition within the bracketed quantity in (3.27) results in a carry-over from the 1/2's place into the 1's place. In fact, this carry-over may "propagate" through several bits in the "unused" part of X_{i-1} if those bits happen to be set. A method for controlling the propagation of carry-overs is described in a later section.

Further observe that the ability to remove the bits that are shifted out of the fractional part of X_i from the arithmetic unit (once carry-overs have been properly handled) implies that those bits can be transmitted to the receiver — the entire source sequence need not be processed before bits are sent. Similarly, computation of X_i' (and hence decoding) can begin as soon as the most significant $\rho + \xi$ bits have been received. This allows some of the letters to be decoded before all of the letters have been encoded.[19]

Finally, observe that requirement (3.26) results in the desirable property that the maximum number of bits emitted in encoding U_i is limited. In particular, the limit is $\rho - 1$, the maximum number of shift operations that might be necessary for renormalization. This property simplifies the design of a buffering scheme to allow fixed-rate, fixed-delay transmission.

3.4 Optimal Rounding-Off of the Source Letter Probabilities

As mentioned in the previous section, the source letter probabilities must be converted to fixed-precision values before they can be used in the K-ary arithmetic code. This section presents a method of making the conversion that results in the best possible coding performance.

[19] In effect, the transmitted bits determine a valid $\theta(\mathbf{U}^i)$ for every i as was mentioned at the end of the previous section.

Recall from Equation (3.30) that the total number of bits $L(\mathbf{U}^N)$ required by the arithmetic code to represent the sequence $\mathbf{U}^N$ is given by

$$L(\mathbf{U}^N) = \rho + \lceil -\log_2[\eta Q_1(U_1)Q_2(U_2)\cdots Q_N(U_N)]\rceil,$$

where η reflects the accumulated inefficiency due to the use of finite-precision arithmetic in the computation of W_N. Let $L_N(\mathbf{U}^N)$ denote $L(\mathbf{U}^N)/N$, the average number of bits per letter required by the code to represent the sequence $\mathbf{U}^N$. If η is temporarily assumed to be unity (i.e., if the accumulated reduction in W_N due to finite-precision arithmetic is temporarily ignored), then in the case of large N, the average number of bits per letter is given approximately by

$$L_N(\mathbf{U}^N) = -\frac{1}{N}\log_2[Q_1(U_1)Q_2(U_2)\cdots Q_N(U_N)],$$

which has expected value

$$E\{L_N(\mathbf{U}^N)\} = -\frac{1}{N}\sum_{\mathbf{U}^N\in A^N} P_{\mathbf{u}^N}(\mathbf{U}^N)\log_2[Q_1(U_1)Q_2(U_2)\cdots Q_N(U_N)].$$

The problem is to select values for the $Q_i(a_k)$'s that minimize $E\{L_N(\mathbf{U}^N)\}$ while meeting the decodability requirements (3.18) and (3.26).

For simplicity of notation, it is assumed henceforth that the source is both stationary and memoryless. With $P_i(a_k)$ and $Q_i(a_k)$ no longer dependent on i, the following shorthand is introduced. Let

$$p_k \stackrel{\text{def}}{=} P_i(a_k);$$
$$q_k \stackrel{\text{def}}{=} Q_i(a_k);$$

for $k = 1,\ldots,K$. Under the assumptions of stationarity and memorylessness and using the new notation, the expected value of $L_N(\mathbf{U}^N)$ becomes

$$E\{L_N(\mathbf{U}^N)\} = -\sum_{k=1}^{K} p_k \log_2 q_k. \tag{3.36}$$

The decodability requirements (3.18) and (3.26) can be rewritten as

$$\sum_{k=1}^{K} q_k \leq 1; \tag{3.37}$$

and

$$q_k \geq 2^{-\rho+1}, \tag{3.38}$$

respectively. The last requirement is that of finite-precision — i.e., that q_k be expressible in the form

$$q_k = \tilde{q}_k 2^{-\rho} \tag{3.39}$$

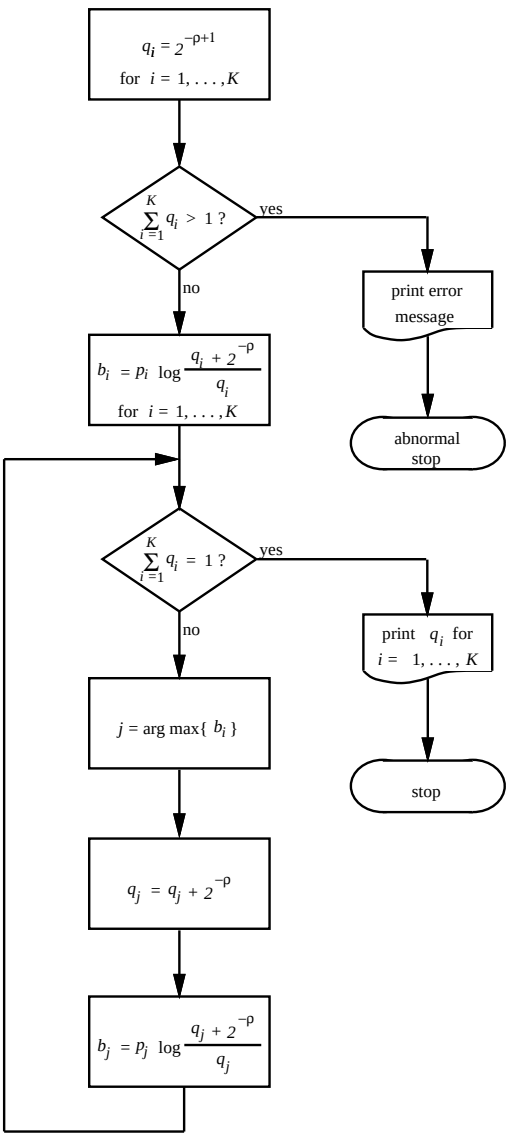
where $\tilde{q}_k$ is an integer, for $k = 1, \ldots, K$.

If q_k were simply required to be a probability measure with no further restrictions, then it is easily shown (via the method of Lagrange multipliers) that $E\{L_N(\mathbf{U}^N)\}$ is minimized by selecting $q_k = p_k$ for $k = 1, \ldots, K$. That is, no improvement in efficiency can be had by using any values other than the actual probabilities for the q_k's. With this in mind, a reasonable approach to selecting values for the q_k's is to find the values "closest" to the p_k's that satisfy the additional requirements. This idea leads to the naive approach of obtaining the q_k's by first simply rounding the p_k's to ρ bits of precision, then adjusting the resulting q_k's to satisfy (3.38) and (3.37) in some *ad hoc* manner. An algorithm of this type is depicted in flowchart form in Figure 3.15 (b). In general, this approach yields good but suboptimal coding efficiency.

Alternatively, the problem can be regarded as a discrete resource allocation problem with a separable and convex objective function. An optimal solution to this type of problem is obtainable by an incremental or greedy algorithm [59, p. 54]. Such an algorithm is shown in Figure 3.15 (a).

Uniform quantization of a Gaussian source provides a simple example to illustrate the difference between the best-case efficiencies of the K-ary arithmetic code when the naive and optimal algorithms are used. In Figure 3.16, it can be seen that when

(a) Incremental (optimal) algorithm



(b) Naive algorithm

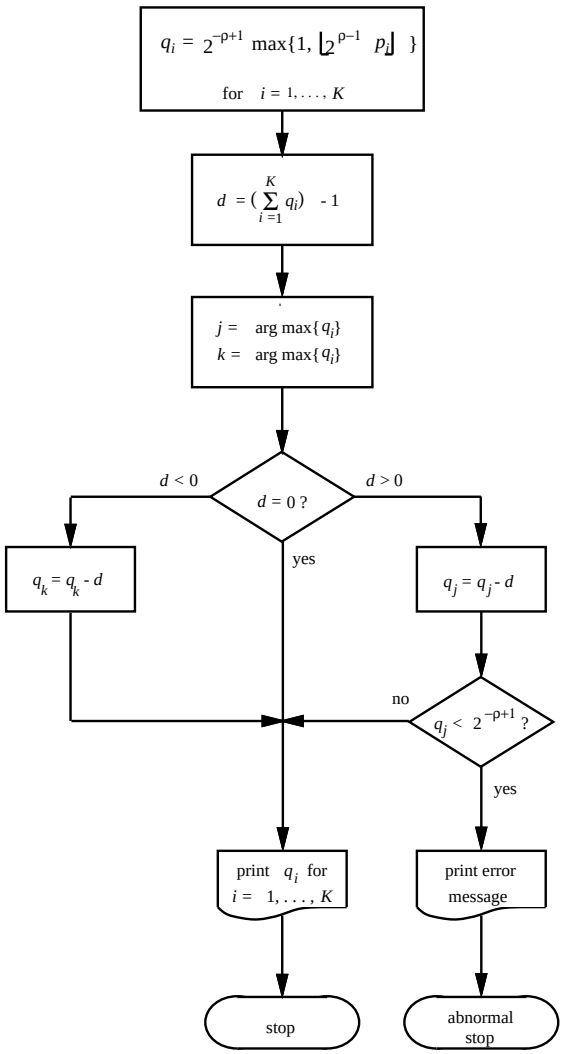


Figure 3.15: Two algorithms for rounding-off source letter probabilities $p_1, \ldots, p_K$ to admissible values $q_1, \ldots, q_K$: (a) is an incremental or greedy algorithm that always results in an optimal solution, and (b) is a naive algorithm that in general results in a suboptimal solution.

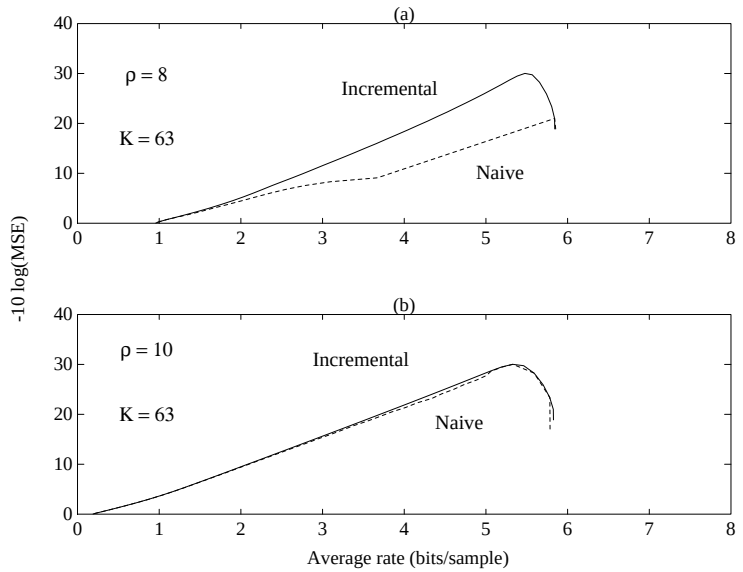


Figure 3.16: The performance of entropy-coded uniform quantization of a Gaussian memoryless source, when each of the two algorithms diagrammed in Figure 3.15 is used to round-off the probabilities of the quantization levels. Graph (a) corresponds to 8 bits of precision; graph (b) corresponds to 10. In both graphs the solid line represents the optimal rounding obtained via the incremental algorithm, and the dotted line represents the suboptimal rounding obtained by using the naive algorithm. The number of quantization regions in both cases is 63.

only 8 bits of precision are used, there is considerable difference between the naive and optimal algorithms. The difference is reduced considerably when 10 bits are used.

Figure 3.17 shows the interdependence between the number of levels K and the precision ρ. Part (d) of that figure shows that if the number of levels is allowed to vary with rate, then 8 bits of precision can provide good quantization performance.

3.5 A Simple Implementation

This section considers a straightforward microprocessor-based implementation of the fixed-precision arithmetic code developed in the previous section. The purpose of this section is twofold: (1) to clarify the material presented thus far, and (2) to discuss the practical problem of controlling carry-overs. The implementation described in this section is not necessarily the most computationally efficient, but is simple in that it closely follows the mathematical description of the code presented earlier. Moreover, the software to perform arithmetic coding presented in Appendix C is based directly

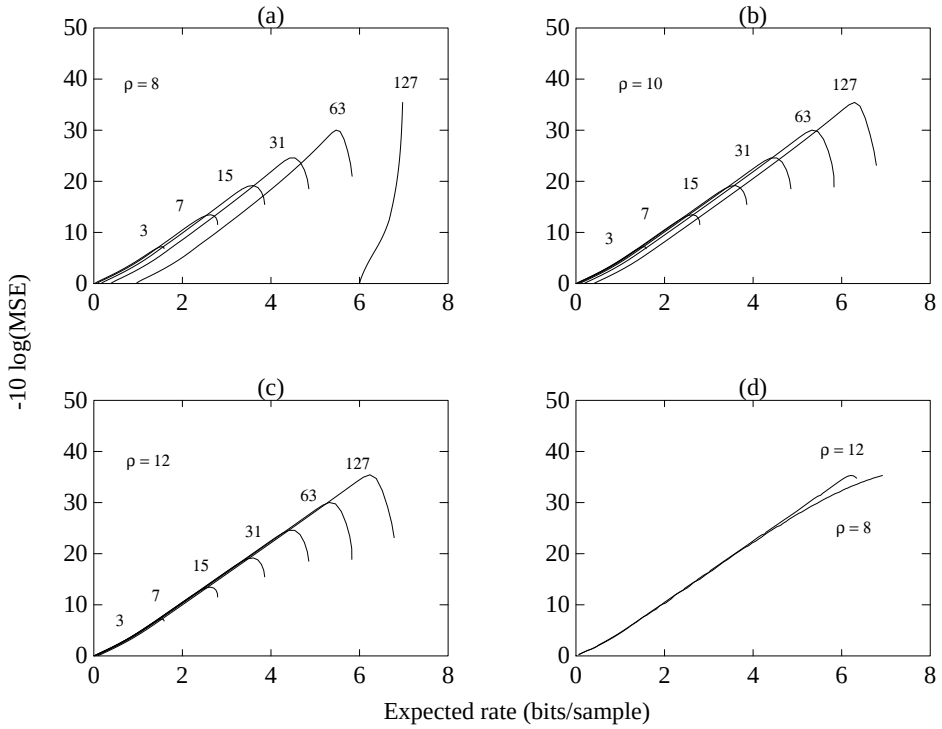


Figure 3.17: The effect on performance (rate versus mean-square error) of optimally rounding-off the probabilities of the quantization levels, in the case of a Gaussian memoryless source and a quantizer having uniform quantization regions. Graphs (a), (b), and (c) correspond to arithmetic precisions (values of ρ) of 8, 10, and 12 bits, respectively. In these three graphs, each of the solid curves corresponds to a different number of quantization regions K (indicated). Graph (d) shows the best performance attainable for each of the three precisions when the number of quantization regions is allowed to vary with bit-rate.

on the implementation described here.

Availability is assumed of a microprocessor that is capable of performing bit-manipulation, addition, subtraction, multiplication, and comparison on $\rho - bit$ fixed-point numbers. A suitable arrangement of registers is shown in Figure 3.18.

The encoder is considered first. The basic approach is to develop the representative number X_N using the large register denoted EX in Figure 3.18(a). The subinterval width W_i is carried in register EW, and is jointly shifted with EX as needed after each letter is encoded for renormalization of EW. As the precision of X_i grows through successive renormalizations, bits are shifted (emitted) into the channel. An additional register, ED, is used to hold intermediate results of computation.