G. Nigel N. Martin
Glen G. Langdon, Jr.
Stephen J. P. Todd

Arithmetic Codes for Constrained Channels

Arithmetic codes have been studied in the context of compression coding, *i.e., transformations to code strings which take up less storage space or require less transmission time over a communications link. Another application of coding theory is that of noiseless* channel coding, *where constraints on strings in the channel symbol alphabet prevent an obvious mapping of data strings to channel strings. An interesting duality exists between compression coding and channel coding. The* source *alphabet and* code *alphabet of a compression system correspond, respectively, to the* channel *alphabet and* data *alphabet of a constrained channel system. The decodability criterion of compression codes corresponds to the representability criterion of constrained channel codes, as the generalized Kraft Inequality has a dual inequality due to the senior author.*

1. Introduction

The first paper dealing with a systematic approach to construct fixed rate codes for constrained channels is [1]. A code construction approach is developed by Franaszek in [2, 3]. A different approach, called sliding block codes, is described in [4]. Reference [4] provides an introduction to constrained channel coding, complete with an extensive bibliography. It is noted in [4] that the approach in [2, 3] and the sliding block approach have a connection via the Perron-Frobenius theory.

In this paper we provide another general viewpoint and approach to constrained channel coding, which has a very different flavor. The basic ideas appeared in an unpublished work by the senior author [5]. This paper includes the work in [5], plus extensions due to the coauthors. The present approach is based on arithmetic coding, a technique originally developed for data compression. We extend arithmetic compression coding to the constrained channel. In effect, we replace the decodability criterion (with its roots in the Kraft Inequality) with a *representability criterion.* The techniques apply to a general class of constraints and provide the designer with powerful new tools. With appropriate attention to precision, the present approach can achieve very close to channel capacity.

The task of a channel coding scheme is shown in Fig. 1. A data string is mapped to a channel string which conforms to the constraints. The channel string undergoes an inverse mapping to recover the original data string. Transformation X^1 is an expansion operation. Transformation X^2 is a compression operation.

A constrained channel accepts strings of symbols from a channel alphabet. However the "constraint" is viewed as a mechanism for disallowing certain substring combinations. A popular constraint, useful in magnetic recording channels, is called a (d,k) channel. Information is transmitted as magnetic flux changes within bit times. Due to intersymbol interference (ISI), flux is not permitted to change in successive bit times. Flux changes too close together tend to cancel each other, reducing the amplitude of the signal peak at the detector. The ISI constraint requires that d bit times must pass before the next flux change is permitted to occur. As a consequence, adjacent peaks occur at least $d + 1$ bit times apart.

We also require the information recorded on the media to have a *self-clocking* property. Although the information is recorded using a clock with a tightly controlled period (e.g.,

30 ns), the clock period (bit time) may skew slightly due to variations in temperature or delays in the clocking circuits, etc. To make any necessary small adjustments to the clock phase, a flux change is required within k bit times of the last flux change of constrained channels called (d,k) channels.

In terms of **1**'s and **0**'s, (d,k) channel strings must have at least d **0**'s following every **1**, but no more than k **0**'s following each **1**. For example, to be allowable by a (2,4) channel, the channel string is a concatenation of the following substrings of Table 1. Except possibly for a beginning or ending substring, any allowable channel string can be parsed into this new set of symbols, i.e., the three symbols of Table 1 could be considered an alternative alphabet to symbols {**0**,**1**}.

We now examine sets of allowable channel strings to see how the constraints come into play. To describe a set of strings, we introduce a tree. Thus, for strings which share a common prefix, the prefix is shown only once.

Consider the set of channel strings allowed by the (2,4) channel. The tree shown in Fig. 2 is a representation of Table 1 and can serve as a generator for a channel string tree. The symbol Λ is the "null" string.

In Fig. 2, node Λ is the root, the other interior nodes are $\langle\,\rangle$, and the leaves are []. The path segments are labeled by either symbol **0** or **1**. The concatenation of path segments gives a *path*. A node or leaf is identified by its path from the root. The root and interior nodes are further identified by the numbers 1 through 5 in parentheses. These nodes are subsequently associated with states of a finite state machine.

The constrained channel has been studied by Shannon [6]. The channel is described in terms of a channel finite state machine (CFSM). The CFSM differs from an ordinary FSM. The CFSM "output" is specifically the identity of its internal state, which places the CFSM in the Moore machine class. However, a CFSM differs from a Moore machine in that a value ℓ is associated with each input symbol. In magnetic recording, the value ℓ is the symbol duration measured in units of time.

The description of the channel constraint is actually of the form of a finite state transition diagram of "allowed" transitions. The value ℓ enters into the rate calculation. More precisely, the CFSM is defined as: (1) a set of states (i,j), (2) channel symbols $\alpha, \beta, \cdots, \xi, \cdots, \omega$ of respective lengths (durations) $\ell(\alpha), \ell(\beta), \cdots, \ell(\xi), \cdots, \ell(\omega)$, and (3) a state transition function $T(i,\xi)$, where $T(i,\xi)$ is the state reached when symbol ξ is received in state i. The channel symbol length is measured in integral channel time units.

In practice, the channel coding problem is to map arbitrary data strings into strings of allowed channel symbols.

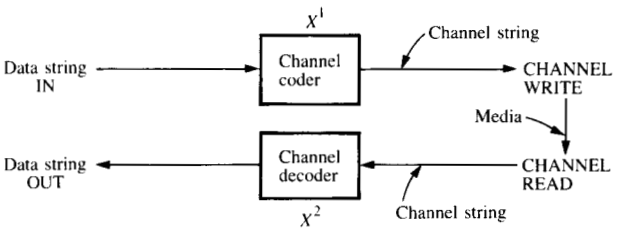

Figure 1 A typical channel coding system.

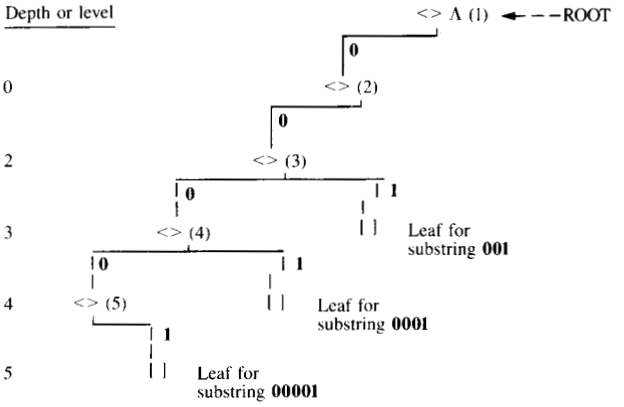

Figure 2 A code string tree generator for (2,4) channel.

Table 1 Channel alphabet of substrings for (2,4) constraints.

001
0001
00001

The channel capacity is log W bits per channel time unit, where W is a growth rate of allowable channel strings per unit increase in length [6]. To achieve the channel capacity rate, Shannon probabilities $p(i,\xi)$ are associated with each CFSM transition from channel state i under channel symbol ξ.

Many channel coding approaches [7, 8] approximate the Shannon transition probabilities with integer length code word mappings. Guazzo [9] uses the probabilities directly in an approach more closely related to this paper. Following Guazzo, we use an "inverted encoder-decoder pair."

The application of the inverted encoder-decoder pair gives rise to a dual of decodability encountered in compression

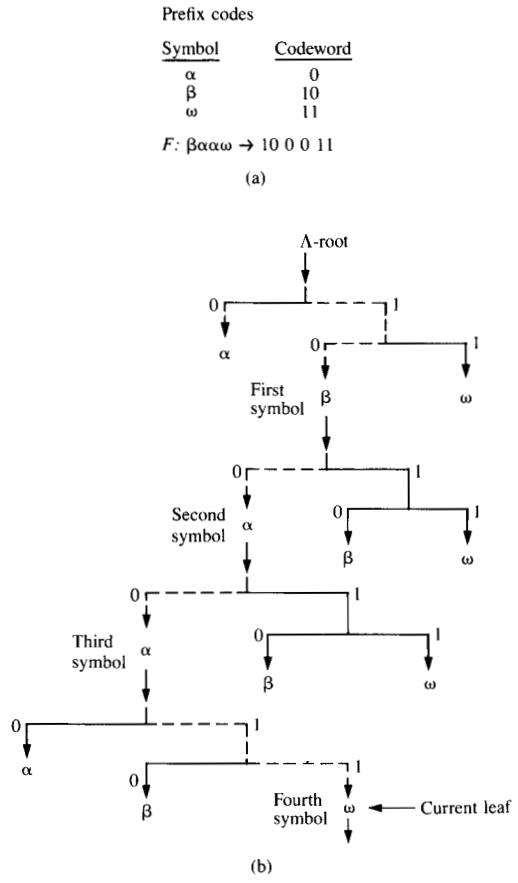

Figure 3 Example prefix code. Image of source string is single leaf. (a) Code table and prefix code; (b) code string tree.

Table 2 Compression coding versus channel coding.

Operation	*Input*	*Output*
Compression encoding (AC* compression)	Source	Code
Compression decoding (AC expansion)	Code	Source
Channel coding (AC expansion)	Data	Channel
Channel decoding (AC compression)	Channel	Data

*AC = arithmetic coding.

coding called *representability*. The Kraft Inequality also has a dual inequality. The analogies of compression versus channel coding are shown in Table 2, which also introduces the terminology employed.

2. Arithmetic coding preliminaries

This section is based on [10] and is included to make the presentation more self-contained. We review arithmetic coding for compressing source strings whose alphabet is $\{\alpha, \beta, \cdots, \omega\}$. Let $\{0,1\}$ be the code alphabet. A coding function F maps each string of the source alphabet to some string in the code alphabet:

$$\{\alpha, \beta, \cdots, \omega\}^* \rightarrow \{0,1\}^*.$$

Here we use * to denote the "star" operator, which generates all possible strings drawn from the alphabet, including null string Λ. To reduce computational complexity, the mapping of an arbitrarily long source string is not made to its image in the set of code strings in a single step. Most coding functions are recursive. The algorithm consists of a series of operations, applied to each successive symbol of the source string, from left to right. Figure 3 shows a mapping from a source string to a binary code string for a prefix code. Figure 3(a) shows a code word table for source symbols α, β, and ω. The coding function F proceeds recursively, handling one source symbol per recursion. The first symbol of string s, which is β, is mapped to 10. For the second symbol, string $\beta\alpha$ is mapped to 100, by concatenating 0 to 10. We handle each next source symbol by a concatenation operation to the code string. Thus the *depth* of the code tree increases each recursion. In Fig. 3(b) we highlight the branches traveled with dashes to demonstrate that the coding process successively identifies the underlined nodes in the code string tree. The root of the code word tree is attached to the leaf at the current depth of the tree, as per the previous source symbol.

Before encoding the first symbol, we are initialized at the root of the code string tree. At initialization, the available *code space* (A) is a set which consists of all finite code strings $\{0,1\}^*$. Following the encoding of the first source symbol β, we are at node 10, i.e., $F(\beta) = 10$. A node is identified by its path from the root. The length of the path corresponds to the depth of the node; here the depth is 2. The current code space available has now been *reduced* from the initial code space (all code strings) to the smaller set of all code strings whose prefix is 10, denoted $10\{0,1\}^*$. Where the alphabet upon which we apply the star operator is understood, we simplify the notation to 10*. At the current state, only code strings in 10* can result from continuing the encoding process. Any string belonging to set 10* is a *continuation* of node 10. We recursively *subdivide*, or subset, the current code space. In the example, the operation following that which gave us 10* delivers a subset; the result is set 100*, a subset of set 10*.

A feature of prefix codes is that a single node in the code space is identified as the result of the *subdivision operation*. In general, each successive source string prefix s of the input string is mapped to a single string in the set of all code strings $\{0,1\}^*$.

• *Arithmetic codes and the code space*

Arithmetic codes do not map individual source string prefixes to individual code strings. Instead, they map successive source string prefixes s to a current code space $A(s)$, which is defined as all continuations of a set of leaves at the current code tree depth. (We discuss later the termination operation which converts the final code space to a single code string.) In arithmetic codes, as opposed to prefix codes, the code space is a set union, e.g., of 011* and 100*.

During the sequential encoding and decoding recursions, arithmetic coding represents the current code space of the current source prefix s as a set of adjacent leaves at the current code tree depth. A set of adjacent leaves may be identified in several ways: (1) the leftmost leaf and the number of leaves to its right; (2) the rightmost leaf and the number of leaves to its left; or (3) the leftmost and the rightmost leaves.

We represent the set by the first method above, by variables $F(s)$ and $A(s)$ defined as follows:

1. F: the path to the first (leftmost) leaf in the code space; and
2. A: the number of leaves in the code space.

Following the step which handles source prefix s, $F(s)$ is analogous to the path to the first leaf and $A(s)$ the number of leaves at the current depth in the current code space. See Fig. 4 for an example where the current depth is 3. As in prefix codes, the code space is successively subdivided, as governed by each source symbol encoded. Note that the right boundary of the current code space is just outside the interval, so the interval is open on the right. Since $F(s)$ belongs to the interval, it is closed on the left.

• *Termination operation*

When the recursion on the last source symbol has taken place, what remains in the code string tree analogy of arithmetic coding is a set of nodes at the current depth. We therefore select a single code string from the current code space during a code string termination operation. In this light, we say prefix coding terminates the current code space to a single node F following the recursion which encodes each source symbol. For prefix codes, the coding process ensures that the number $A(s)$ is always unity.

• *Mapping code strings to the unit interval*

At the code string root Λ the initial code space $\{0,1\}^*$ extends from $00\cdots0$ to $11\cdots1$ at some finite depth. We can map the code space to the unit interval of rational numbers, beginning with value zero and extending to, but not including, the value one. We treat code strings as a fractional magnitude by placing a binary point on the left. The interval shown in Fig. 3 is rewritten as [.010, .101), where the size of A is now the number of nodes (3) multiplied by 2^{-3} or $3 \times 1/8$, which is .011 in binary representation.

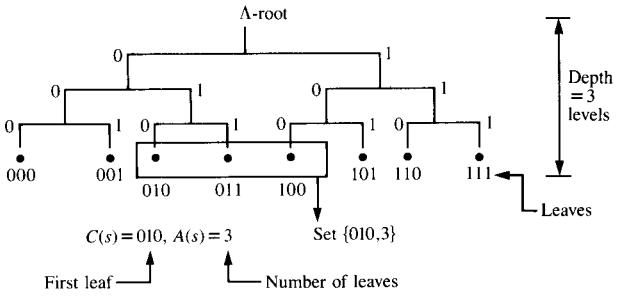

Figure 4 Arithmetic codes permit image of source string in code string tree to be a set of leaves.

This conversion to a new representation that we have just made permits arithmetic operations on code strings. In the example tree of Fig. 4, at a depth of 3, string $F(s)$ is .010, and the value of $A(s)$ is .011. Code string prefix $F(s) + A(s) =$ .101 is just outside the interval defining the code space at the current depth.

Having a correspondence between nodes of the code string tree and the number interval, we subdivide the interval by arithmetic operations. The arithmetic nature of the subdivision recursion gives arithmetic coding its name. Each source symbol ξ is handled by adding (instead of concatenating) a value $D(s.\xi)$ called an *augend* to the right end of the code string $F(s)$, forming $F(s.\xi) = F(s) + D(s.\xi)$. The "." between the prefix of a source string and the next symbol denotes concatenation and illustrates the source symbol ξ involved in the recursion.

• *Subdivision operation*

For data compression, one property we wish to avoid in subdividing the code space (current interval) is *overlap*. If the code spaces for two distinct source strings overlap, then there will be a problem for the decoder upon receipt of a string in the overlapping space—which of two possible source string prefixes was encoded? Thus, for decoding, we require a subdivision into nonoverlapping parts. The danger of overlap exists between strings which are adjacent in the ordering.

Consider the alphabet $\{\alpha,\beta,\omega\}$ and the set of strings of length 3 under the ordering α,β,ω. The code space is subdivided according to this ordering. We use $s + 1$ to denote the next string after s in the ordering of the same length. If string $s = \beta\beta\omega$, then $s + 1 = \beta\omega\alpha$. Thus the code space assigned to $s + 1$ is immediately to the right of the code space assigned to

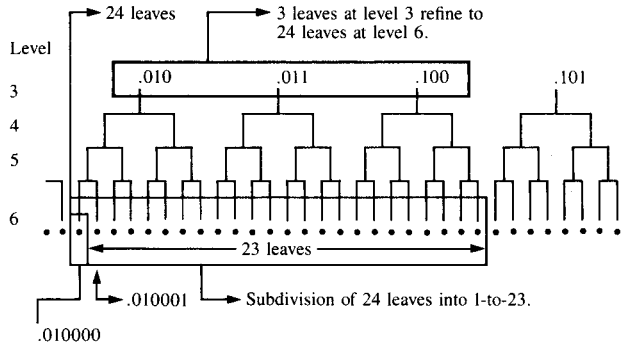

Figure 5 Code tree refinement and subdivision of current space.

s. Similarly, $s - 1$ indicates the immediately preceding string, i.e., $\beta\omega\alpha - 1$ is $\beta\beta\omega$.

Consider the example of Fig. 4 and assume that the subdivision operation does not produce either an overlap or a gap in the leaves of the tree between the code space for string s and the code space for $s + 1$. Such a subdivision is a *partition*. For a partition, the sum $F(s) + A(s)$ gives the value of the code tree leaf just to the right of the current code space. For Fig. 4, $F(s + 1) = .101$. We summarize this below:

Partition (no gap or overlap): $F(s + 1) = F(s) + A(s)$.

In any event, the value $F(s + 1) - F(s)$ defines the size of the code space which encoded continuations s^* of string s can use without overlapping the code space assigned to string $s + 1$.

The $+1$ notation, when applied to nodes on the right boundary of the code string tree, is a special case. There are no nodes to the right of the rightmost node at each depth or level. Thus what do we mean by $\omega\omega\omega + 1$? In reality, the $s + 1$ notation is used to identify some property involving the left boundary of the code space to the right of string s, e.g., $F(s + 1)$. When $s = \omega, \omega\omega, \omega\omega, \cdots, \omega$, etc., that property is obtained by defining

$$F(\Lambda + 1) = F(\omega + 1) = F(\omega\omega + 1) = \cdots = 1.000.$$

In arithmetic coding, of the initial code space, the leftmost nodes at any level are identified as $.0000 \cdots 0$, and the rightmost nodes have the form $.1111 \cdots 1$. We initialize the F and A values to $F(\Lambda) = .0$, and $A(\Lambda) = 1.0$. With $F(\Lambda + 1) = 1.00$, note that $F(\Lambda) + A(\Lambda) = 1.00$ and that 1.00 is just outside the current code space, as desired.

- *Code space refinement and realignment*

In handling the next source symbol, a problem in subdividing the code space occurs when it is represented by too few leaves. Consequently we must increase the tree depth. Suppose the two-symbol source alphabet is $\{\alpha,\omega\}$ and the code space in Fig. 4 is to be partitioned according to a split for the less probable symbol α in the range $1/16$ to $1/32$. We lack the precision to split three leaves in this proportion. We can solve the precision problem by increasing the depth from 3 to 6; see Fig. 5. This figure is very instructive. Each leaf (node) at depth 2 is *refined* to 2^3 leaves at depth 6, without changing the value of the current code space. By increasing the depth from 3 to 6 for greater arithmetic precision, the code space becomes [.010000, .101000). The effect of an increase in depth on the code string F is to add more 0's on the right. There are now $3 \times 8 = 24$ nodes at the new depth, but size A is the same, 24×2^{-6}, because the depth is now 6. At depth 6, with 24 leaves in the code space, we have enough precision to split the code space in the desired range. In Fig. 5, we assign α as the first symbol in the ordering, hence the left part of a partition is assigned to α and the right part to ω.

The partition shown has a single leaf in one part (in this case the leftmost) and 23 leaves in the second (rightmost) part and achieves a subdivision whose proportion lies within $1/16$ to $1/32$. If the current source symbol is α, we keep the smaller part, so that $F(s.\alpha) = .010000$ and $A(s.\alpha)$ is reduced to $1/2^6$ or .000001. On the other hand, if the current symbol is the more probable ω, we keep the current code space as defined by $F(s.\omega) = .010001$ and $A(s.\omega) = .011000 - .000001 = .010111$.

In arithmetic coding, the source and code symbols are ordered, an ordering which extends lexicographically to the strings. The arithmetic coding compression operation recursively maps a source string to a subset of code strings of a given length. For each new source symbol, this code string subset is subdivided. The length of the code string subset is possibly increased by an operation called *realignment*. Realignment essentially makes more strings available in the subset, so that the subdivision operation can achieve finer proportions.

The code string is generated recursively. A first arithmetic recursion adds a value D (augend) to the previous code string. A second arithmetic recursion on code space A controls the realignment. In probability-based arithmetic codes, the second recursion is a product of probabilities. In length-based arithmetic codes, the second recursion is a sum of lengths. See [11].

- *Arithmetic coding equations*

The number A is a fraction of q significant digits, whose value corresponds roughly to the probability $p(s)$ of the

source string. More formally, the code string subset is the set of code strings c such that $\ell(c) = \ell(F(s))$, and $F(s) \leq c < F(s) + A(s)$.

Given a next symbol ξ to encode, $[F(s),F(s) + A(s))$ is subdivided into as many subsets as there are symbols in the alphabet. For compression purposes $A(s.\xi)$ should be such that $A(s.\xi)/A(s) \simeq p(\xi/s)$, where $p(\xi/s)$ denotes the state dependent probability of symbol ξ given the context defined by string s.

Let subdivision operation S act upon value $A(s)$ to determine values $A(s.\alpha), A(s.\beta), \cdots, A(s.\omega)$: $S(A(s),\beta) = A(s.\beta)$. Consider the case where the space $A(s)$ is partitioned; we have

$$A(s) = A(s.\alpha) + A(s.\beta) + \cdots + A(s.\xi) + \cdots + A(s.\omega).$$

For the first symbol, $F(s.\alpha)$ is $F(s)$ so the subset is $[F(s),F(s) + A(s.\alpha))$. For the second symbol β, the subset is $[F(s) + A(s.\alpha),A(s.\beta))$. In general,

$$F(s.\xi) = F(s) + D(s,\xi), \tag{1}$$

where $D(s,\xi)$ is the sum of the code space sizes $A(s.\gamma)$ of all symbols $\gamma < \xi$. The term addend may be applied to values $A(s.\gamma)$. Also,

$$A(s.\xi) = D(s,\xi + 1) - D(s,\xi). \tag{2}$$

Except for the last symbol, Eq. (2) serves as a definition of the code space available for string $s.\xi$. The available code space for $s.\omega$ is

$$A(s.\omega) = F(s + 1) - (F(s) + D(s,\omega)).$$

For the decompression operation, we begin with a code string F treated as a fraction. To decode the first symbol, we apply the subdivision operation to $A(\Lambda)$ and determine the largest augend $D(\Lambda,\xi)$ less than F. We decode ξ such that

$$D(\Lambda,\xi) \leq F < D(\Lambda,\xi + 1). \tag{3}$$

To decode the second symbol, it is convenient to subtract augend $D(\Lambda,\xi)$ from F and compare the augends $D(\xi,\gamma)$ against $F - D(\Lambda,\xi)$.

In compression coding, the transformation is decipherable if the subdivision operation satisfies the following inequality for the addends:

$$A(s) \geq A(s.\alpha) + A(s.\beta) + \cdots + A(s.\omega). \tag{4}$$

Here Eq. (4) is called the *decodability criterion.*

- *P-based coding equations*

There are basically two ways to subdivide the interval $A(s)$ according to a set of probabilities. We first perform the subdivision with the probability-based (P-based) technique

$$A(s.\xi) \simeq A(s) \times p(\xi/s), \tag{5}$$

where "$\simeq$" includes the necessary approximation such that the precision of $A(s.\xi)$ is not allowed to grow, and where $p(\xi/s)$ is the portion of the current code space allocated to symbol ξ for the iteration following those which have handled string prefix s. In general, the binary representation of $A(s.\xi)$ will have a number of leading 0's followed by a q-bit mantissa whose leading bit value is 1. The subdivision operation is then performed on the mantissa. Equation (1) is used to encode, where augend $D(s,\xi)$ is the sum of $A(s.\gamma)$ for symbols $\gamma < \xi$.

- *L-based coding equations*

A second approach to the subdivision operation is the basis for the length-based (L-based) arithmetic codes. Each value $p(\xi/s)$ is represented as a length $\ell(\xi/s)$, where

$$\ell(\xi/s) \simeq -\log p(\xi/s). \tag{6}$$

The lengths are rational, to some integer denominator C. The size of the code space for continuations of string s is kept as the sum $L(s)$ of the lengths of the encoded symbols comprising string s, where $E(s)$ is the integer part and $X(s)/C$ is the fractional part. $X(s)$ takes on values $0, 1, \cdots, C - 1$. The $E(s)$ of L-based arithmetic codes corresponds to the number of leading 0's of $A(s)$ in P-based arithmetic codes, and the mantissa of $A(s)$ corresponds roughly to the value $2^{-X(s)/C}$. The augends $D(s,\xi)$ are of the form

$$D(s.\xi) = M(X,\xi) \times 2^{-E(s)}, \tag{7}$$

where augend mantissas M are typically precomputed for each symbol and each value of X and are stored in an augend table.

3. An arithmetic coding overview of channel coding

In compression coding, if the subdivision operation partitions the subset and assigns a partition (no gap, no overlap) to each source symbol, then the code is decipherable and the available code space is completely utilized. If the subdivision operation leaves no overlap but allows a gap (a subset assigned to no next input symbol), the code is still decipherable. However the gaps do not utilize code space and give less compression. If the subdivision operation permits the subsets corresponding to two distinct input strings to overlap, then the code is undecipherable.

In channel coding, if the subdivision operation leaves gaps, the code is not representable. If the channel code subdivision operation partitions the subset at each step, the code is representable. If the subdivision operation leaves overlapping subsets, the code is still representable but there is a reduction in information rate.

In the following sections we describe the coding recursions and establish the requirements for representability. Arithmetic coding provides more than one solution for channel coding. One solution employs approximations to the Shannon probabilities and a subdivision operation which recursively partitions the subset of channel strings. A second solution is reminiscent of a length-based arithmetic code [7]. The second solution offers a fixed rate code.

Instead of determining how to encode data strings to the channel, let us first consider translating channel strings back to data strings, i.e., the mapping X^2 of Fig. 1. This mapping is the arithmetic coding operation. A constrained channel can only accept a subset of strings consisting of symbols drawn from the channel alphabet. The channel alphabet is ordered, so we can order the channel strings u. The data string corresponding to u is $F(u)$. In ordinary arithmetic coding where the probability of each string is known, $F(u)$ is the probability of all strings less than u in the ordering. For channel coding, the corresponding intuitive property is: of all long, acceptable channel strings starting with prefix u, a fraction $F(u)$ of acceptable channel strings precede u in the ordering. This fraction can be expressed as a fractional number whose radix is the cardinality of the data alphabet. For practical purposes, we assume the data string alphabet to be binary and the data string to be a binary fraction. As with compression coding, the mapping is order preserving; thus, $u^1 < u^2$ implies $F(u^1) < F(u^2)$.

To transform a data string into a channel string u, we treat the data string as a binary fraction $F(u)$. We then find channel string u such that of all long, acceptable channel strings, the portion $F(u)$ have an $\ell(u)$-symbol prefix which is less than u. We perform this transformation recursively, generating one channel symbol per recursion. The binary representation of the fraction is decompressed by arithmetic coding; each newly decompressed channel symbol is the next symbol of the channel string.

Thus to transform a channel string u back into a data string, we find the binary representation of fraction $F(u)$ such that of all long, acceptable channel strings $F(u)$ have an $\ell(u)$-symbol prefix less than u. Again, the transformation is performed recursively on each channel symbol, this time by AC compression.

Some notation is needed. Let string $u.\xi$ result from concatenating symbol ξ to string u. Let strings $u + 1$ and $u - 1$, respectively, denote the strings of length $\ell(u)$ next higher than u in the ordering and next lower than u in the ordering. Let Λ denote the empty string. The initial conditions are

$$F(\Lambda) = 0 \text{ and } A(\Lambda) = 1.00, \tag{8}$$

so that the initial space is the number line of fractional numbers including 0: [0, 1). If u_ω is the highest string of length $\ell(u_\omega)$ in the ordering, then $u_\omega + 1$ is undefined. By definition

$$F(u_\omega + 1) = F(\Lambda + 1) = 1.00. \tag{9}$$

4. Channel coding with arithmetic codes

The arithmetic decompression recursion generates the channel string from a data string. The compression recursion recovers the data string from the channel string. We assume that the channel finite state machine model for the channel constraints has been calculated. For convenience, we assume that the CFSM begins in state 1, and we receive channel string u. We can simplify the next state function notation $T(1,u)$ by simply using $T(u)$. The state dependent Shannon probability for symbol ξ following string u may be denoted $p(\xi/T(u))$.

The key idea in the application of arithmetic coding to the constrained channel is the definition of F as the fraction of long, acceptable channel strings whose $\ell(u)$-symbol prefix is lower than u. It can be shown that from the channel state $T(u)$, in which channel code symbol ξ occurs, the value of Shannon probability $p(\xi,T(u))$ is the fraction of all long, allowable channel strings from state $T(u)$ under symbols preceding symbol ξ. Arithmetic coding provides a simple way to implement the correspondence between an acceptable channel string u and the number representation $F(u)$. The coding recursion is as follows:

$$F(u.\xi) = F(u) + D(u,\xi), \tag{10}$$

where $D(u,\xi)$ is the state dependent augend representing the portion of all channel strings which are continuations of channel strings $u.\alpha$ through $u.\xi - 1$.

- *Representability of data strings*

In applying the arithmetic encoding and decoding functions to channel coding, we must ensure that all data strings can be represented as channel strings and that the data string can be recovered from the channel string.

Just as channel coding is a dual of compression coding, the inverted order of the compression and expansion phases creates a dual to the notion of decodability. Each data string must be representable as a channel string. If there is a data string prefix which is not the image $F(u)$ of some channel string u, that data string is not representable.

Decodability implies that the subdivision operation must not create an overlap of code intervals. Similarly, in the dual situation of channel coding, representability states that the subdivision operation must not leave any gaps in its range, the set of all the data strings. For channel coding, we may subdivide a current code space $A(u)$ such that the sum of the sizes of the subintervals exceeds the original interval.

We study the representability criterion for the subdivision operation with the aid of Fig. 6. We consider an interval between $F(u)$ and $F(u+1)$ and the subdivision which defines $F(u.\xi)$ and $F(u.\xi+1)$. Following the subdivision operation for channel string $u.\xi$, the size of the interval taken up by continuations of $u.\xi$ is seen to be $D(u.\xi,\omega) + A(u.\xi.\omega)$. If this value does not exceed the value $F(u.\xi+1) - F(u.\xi)$, corresponding to the interval to be spanned

$$[F(u.\xi), F(u.\xi+1)),$$

then a nonzero gap appears as identified in Fig. 6. This gap is zero if

$$F(u.\xi+1) - F(u.\xi) = D(u.\xi,\omega) + A(u.\xi.\omega). \tag{11a}$$

The left-hand side may be rewritten in terms of the augend and addend values as $D(u,\xi+1) - D(u,\xi)$, but the right-hand side contains term $A(u.\xi.\omega)$. Since ω is the last symbol in the alphabet, we have not yet defined $D(u.\xi,\omega+1)$ and hence $A(u.\xi.\omega)$. Define

$$D(u,\omega+1) = D(u,\omega) + D(u.\omega,\omega) + D(u.\omega.\omega,\omega) + \cdots \tag{11b}$$

such that

$$\begin{aligned} A(u.\omega) &= D(u,\omega+1) - D(u,\omega) \\ &= D(u.\omega,\omega) + D(u.\omega.\omega,\omega) + \cdots . \end{aligned} \tag{11c}$$

In view of Fig. 5 and the above, we may rewrite Eq. (11a) as

$$D(u,\xi+1) - D(u,\xi) = D(u.\xi,\omega+1). \tag{12}$$

We leave a gap if $D(u,\xi+1) - D(u,\xi) > D(u.\xi,\omega+1)$. In compression coding, the problem to avoid is an overlap during the subdivision operation, but gaps are not disastrous. In channel coding, the overlap is not harmful, but gaps are disastrous. Since the decompression recursion is performed on all data strings, the effect of overlap is to exclude certain channel strings from the output of the channel encoder. The greater the overlap, the less the rate. For representability, overlap is permitted, so the equal sign of Eq. (12) can be replaced by the $\leq$ sign. The result is an important inequality due to [5] (a dual to a generalized Kraft Inequality for compression):

$$D(u,\xi+1) - D(u,\xi) \leq D(u.\xi,\omega+1). \tag{13}$$

When Eq. (13) is met with equality for all values of u and ξ, the subdivision operation always yields a partition with zero overlap and zero gap. Subdivision operations which partition the code space are suitable for either compression coding or channel coding. Such compression codes, e.g., Guazzo [9], are also suitable for channel coding. Also in this category is an arithmetic compression code due to Martin [12]. In this case the code is designed to use the Shannon probabilities directly [9]. However, it is more interesting to consider techniques which provide a fixed rate.

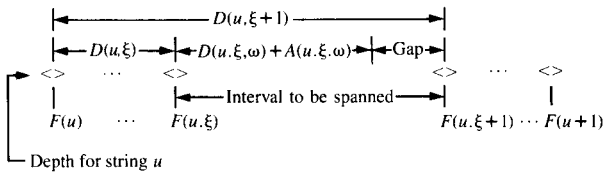

Figure 6 Subdivision notation and illustration of gap.

For arithmetic codes, the subdivision operation is suitable for channel coding if the sense of the decodability inequality of Eq. (4) is reversed:

$$A(u) \leq A(u.1) + \cdots + A(u.\omega). \tag{14}$$

For length-based arithmetic codes which use augend tables, the addend values in Eq. (14) may be obtained from Eqs. (2) or (11c).

• *Fixed rate channel codes for (d,k) constraints*

The arithmetic codes of the previous section can be employed in channel coding provided the interval subdivision process conforms to (13). There is no guarantee, however, that the result provides a fixed rate. Moreover, the general approaches are designed for handling general probabilities. Since the Shannon probabilities are not arbitrary, i.e., are derived in a structured way, some implementation advantage is to be gained.

In this section, we study fixed rate codes in the context of the (d,k) constraints. There is a particularly simple way to take advantage of the nature of the Shannon probabilities while at the same time providing a fixed rate code. The basis for this approach depends upon the following:

> For any CFSM, if string u of length $\ell(u)$ takes the CFSM from initial state 1 back to state 1, then current interval size $A(u)$ is $W^{-\ell(u)}$. Thus $A(u)$ is dependent only on value $\ell(u)$.

To show the above, at initial state 1, the code space is the set of all allowable channel strings from that state. Asymptotically, there are $W^{\ell(u)}$ such strings of length $\ell(u)$. Since u is one such string, u represents $W^{-\ell(u)}$ of the code space from state 1, because we have returned to state 1. (State 1 where we ended "spawns" the same relative number of channel strings per unit length as the state we started with.)

For (d,k) codes, the CFSM is systematically returning to state 1 upon receipt of a **1**. We can take advantage of this fact by assigning to symbol **1** the left part of the result of a subdivision operation.

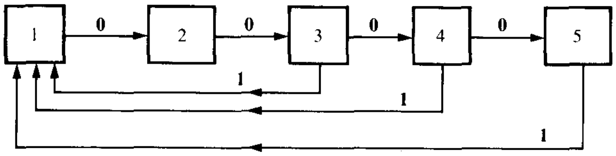

Figure 7 Finite state machine for (2,4) channel constraint. The channel string tree appears as in Fig. 8.

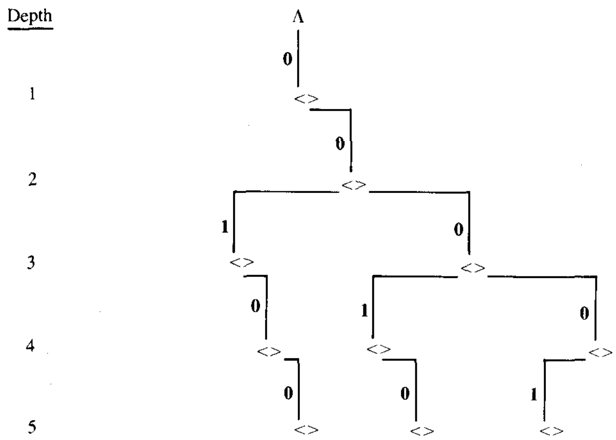

Figure 8 Channel string tree for (2,4) constraints, **1** on left, to depth 5.

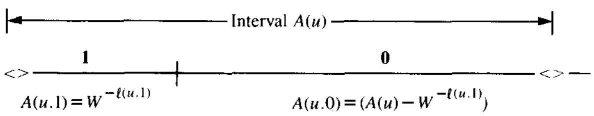

Figure 9 Subdivision operation for (d,k) constraints.

For the (2,4) constraints, the CFSM appears in Fig. 7. This CFSM acts in unit time, that is, upon the basic channel time unit symbols **0** and **1.** A one-state three-symbol "extended alphabet" or parsed phrase CFSM could be defined using the three phrases **001, 0001,** and **00001.** The channel string tree appears as in Fig. 8.

The data string corresponding to a (d,k) channel string is constructed as follows. For nodes (states) where the channel symbol is certain (no choice, only one possible value), the interval is not subdivided and nothing is added to the data string. For states where there is a choice, then the interval is subdivided into two parts, and the size of the left part is $W^{-\ell(u)}$. In other words, whenever the current interval is subdivided, it is subdivided as shown in Fig. 9.

The only time the data string value is changed is when a nonzero augend is added to it. Since the nonzero augend is always $W^{-\ell(u.1)}$, the only interval subdivision operation required is to maintain the current value of

$$A(u.1) = D(u.0) = W^{-\ell(u.0)} = W^{-\ell(u)+1}. \tag{15}$$

$A(u.1)$ is implementable as follows with a symbol-wise recursion:

$$W^{-\ell(u)+1} = W^{-\ell(u)} \times W^{-1}. \tag{16}$$

Since W^{-1} is constant, the value of recursion variable $A(u.1)$ depends only on $\ell(u) + 1$: we denote the recursion variable as $A(\ell(u) + 1)$. At any channel time unit, we add to $F(u)$ either nothing or the value of recursion variable A. If the binary choice selects the right part of Fig. 9 (the symbol is a **0**), we add $A(\ell(u) + 1)$ to $F(u)$ to obtain $F(u.0)$. For the (2,4) constraints, a choice can only occur for transitions from states *3* or *4*. Suppose a (2,4) channel string u is as follows:

001000100001001. (17)

The symbols governing the transitions from states *3* or *4* are checked and are those at lengths 3, 6, 7, 10, 11, and 15. The symbols at lengths 1, 2, 4, 5, 8, 9, 12, 13, and 14 are *constrained transitions* and do not cause a subdivision; there is no choice as to the symbol value. Of the unconstrained transitions (involving a binary choice), only the symbols at lengths 6, 10, and 11 are assigned the right part of a subdivision. At these lengths, the subdivision operation creates strings of portions

W^{-6}, W^{-10}, and W^{-11},

which are lower than string u in the ordering and require a nonzero augend. We deduce

$$F(u) = W^{-6} + W^{-10} + W^{-11}. \tag{18}$$

Our observations for the (d,k) constraints are summarized by Eq. (19). We describe the recursion on $\ell(v)$, the length of the prefix of string u. If $\ell(v)$ is 1, then v is the first symbol of u.

$$F(u) = \sum_{\ell(v)=1}^{\ell(u)} \delta_v \times W^{-\ell(v)}, \tag{19}$$

where δ_v is zero if the last symbol of v is **1** and δ_v is unity only if v ends in **0** and $T(v - 1,0)$ is among states $d + 1$ through k. δ_v is normally zero, but it is one when the $\ell(v)$th channel string symbol is **0** and not constrained to be **0.**

Equation (19) is valid for all (d,k) constraints and is an interesting way of viewing (d,k) codes from an arithmetic coding perspective. Eq. (19) generalizes to constrained

channels whose allowable strings can be parsed to give a one-state CFSM. When weighted by eigenvector components of multi-state CFSM's, Eq. (19) generalizes to more economical coding implementations in a companion paper [13].

The implementation of channel coding for (d,k) codes is thus quite simple. The internal variable for the subdivision recursion at each channel time unit involves only multiplying the current value of $W^{-\ell(u)}$ by W^{-1} to obtain the new value. Although this value only coincides with addend $A(u.1)$ from states which have a binary choice, we use the notation $A(u)$ for the internal recursion variable.

There are three problems:

1. W is an irrational number.
2. We must avoid a "growing precision" problem. In using finite precision arithmetic, we would prefer to keep a fixed rate.
3. We must approximate W^{-1} and carry out the subdivision operation such that no gaps are left, i.e., we must satisfy the representability inequality of Martin [5].

Two means of using finite precision and performing the subdivision suggest themselves—a multiplicative approach and a length-oriented approach. In either case, we should approximate W with a number slightly smaller, which means W^{-1} should be approximated with a number slightly larger. This ensures that the size of the interval will be slightly larger than the ideal.

- *P-based binary codes for (d,k) constrained channels.*

In the P-based approach, we seek a q-bit binary factor P such that

$$P > W^{-1}. \tag{20}$$

For each channel time unit, if the symbol indicates a binary choice **0,** recursion variable A is added to $F(u)$. We also multiply A by P to obtain the intermediate (unnormalized) product $A \times P$. This product may have a leading 0, which means the intermediate product is normalized by a left shift before placing it in A. Normalizing $A(\ell(u) + 1)$ also calls for a left shift of $F(u)$. Without a leading 0 in the intermediate product, no normalization shift is needed.

Since $\ell(\Lambda) = 0$, we initialize recursion variable $A(\Lambda)$ to P. Note that were we to initialize A to 1.00, then the result of the first product would be P. By initializing to P, we are one channel time unit "ahead" in calculating the augend. That is, after ℓ time units the value in A corresponds to $W^{-\ell+1}$. In this way, we always have the nonzero augend precalculated. This yields an implementation advantage; the product for the new $A(\ell(u) + 2)$ can be performed concurrently with the sum $F(u.0) = F(u) + A(\ell(u) + 1)$ if required.

- *L-based binary codes for (d,k) channels*

The L-based (d,k) approach [5] precedes the P-based approach. The factor W^{-1} is approximated to the number base 2, as a negative rational power $-J/C$:

$$2^{-J/C} < W^{-1}. \tag{21}$$

For best efficiency, J/C should be such that $2^{J/C}$ is very close to, but slightly smaller than, W. For ease of implementation the value of C should be small, e.g., 2, 3, or 4.

In a sense, the information value of each channel time unit is J/C bits. For a channel string of length u, the value $W^{-\ell(u)}$ is approximately $2^{-J\times\ell(u)/C}$, which we can represent as

$$2^{-E(u)} \times 2^{-X(u)/C},$$

where the integral division of $J \times \ell(u)$ by C yields quotient $E(u)$ and remainder $X(u)$. Values of X belong to the set $\{0, 1, \cdots, C-1\}$. Note that for fixed rate, L-based arithmetic channel codes, $E(u)$ and $X(u)$ are actually independent of the particular sequence of channel symbols comprising u and depend only on the channel string length $\ell(u)$. In this context, we may equivalently employ the argument $\ell(u)$ in functions E and X without introducing ambiguities.

We generate the data string from the channel string according to Eq. (22) below, which is Eq. (19) with our approximation:

$$F(u) = \sum_{\ell(v)=1}^{\ell(u)} \delta_v \times 2^{-J\times\ell(v)/C}. \tag{22}$$

We can factor augend terms

$$D(u) = 2^{-J\times\ell(u)/C}$$

into an integral shift $2^{-E(u)}$ and fractional power $2^{-X(u)/C}$. Unfortunately, most values $2^{-X(u)/C}$ are irrational and must be approximated by rational augend factor $M(X,0)$:

$$M(X,0) \simeq K \times 2^{-X/C}, \qquad X = 0, 1, \cdots, C-1, \tag{23}$$

where K is a scale factor. (The augend factors are relative, i.e., each can undergo the same relative shift.) Now $D(u.0)$ is represented as

$$D(u.0) = M(X(u),0) \times 2^{-E(u)}. \tag{24}$$

We can rewrite Eq. (22) in light of this second approximation:

$$F(u) = \sum_{\ell(v)=1}^{\ell(u)} \delta_v \times M(X(v),0) \times 2^{-E(v)}. \tag{25}$$

As in [10], from one recursion to the next we need only remember the numerator X of the fractional power, provided

we shift left $F(u)$ to maintain proper relative alignment of the string F with the augend factors M. Since the shift is relative, alternatively the augend factors M may be viewed as being shifted right. Following treatment of symbol ξ, the value of the shift is $E(u.\xi) - E(u)$, and the new value of the numerator of the fractional power is $X(u.\xi)$. These values are determined from the previous value of fractional part $X(u)$ as follows:

$$E(u.\xi) - E(u) = \lfloor (J \times \ell(\xi) + X(u))/C \rfloor ,$$

$$X(u.\xi) = (J \times \ell(\xi) + X(u)) \bmod C.$$

The notation "$\lfloor\ \rfloor$" denotes the integer part of the number inside.

We must make approximations such that the $M(X,0)$ satisfy the representability criterion of Eq. (13). The test which guarantees representability is a self-consistency test. Equation (26) below is derived from Eq. (13). Setting ξ to **1** and $\xi + 1$ to **0**, we have

$D(u,0) - D(u,1) \leq D(u.1,\omega + 1)$, which, since $D(u,1) = 0$,

yields $D(u,0) \leq D(u.1,\omega + 1)$. (26)

[Note that if we set ξ to ω in Eq. (13), the self-consistency test is $D(u,\omega + 1) - D(u,\omega) \leq D(u.\omega,\omega + 1)$, which is true by definition since $D(u,\omega + 1) = D(u,\omega) + D(u.\omega,\omega + 1)$.]

Equation (26) may be rewritten using Eq. (24) and factoring out the integral power of 2:

$$M(X,0) \leq M(X,\omega + 1). \quad (27)$$

Factor $M(X,0)$ is the augend factor corresponding to $D(u,0)$ where u is any channel string such that $X = J \times \ell(u) \bmod C$. Equation (27) gives rise to C inequalities. Factor $M(X,\omega + 1)$ is calculated with the help of Eq. (11b). String $u.1$ takes the CFSM to state *1*. We can begin in state *1* with numerator X, assume receipt each channel unit time of symbol $\omega = \mathbf{0}$ (when allowed), and accumulate the sum of Eq. (25). In forming the sum we must add nothing when in states for which δ_v is 0. We may take the first C nonzero terms of the sum, or until the CFSM returns to state *1*. There is some leeway here because the set of augend factors are self-consistent if each $M(X,0)$ is less than or equal to a number which itself is slightly less than $M(X,\omega + 1)$.

We use Eq. (23) to assign values to $M(X,0)$. Let the values of M be q-bit binary representations of integers by a suitable pick of K:

$$M(X,0) = round\,(K \times 2^{-X}). \quad (28)$$

We next test for self-consistency using Eq. (27). This requires the calculation of $M(X,\omega + 1)$ from Eq. (25), assuming proper initial conditions, as a finite number of terms $M(X(v),\omega)$ shifted by $2^{-E(v)}$. If Eq. (27) fails, perhaps some of the values of M can be adjusted. Otherwise, the number of bits in the representation may be increased. Alternatively, the rate could be reduced.

- *An example with (5,12) constraints*

Consider the (5,12) code, whose rate is 0.3369 bits per channel time unit. Let rate $J/C = 1/3$. This means a rate of one data bit for three channel time units. The unit-time CFSM has states *1*, *2*, $\cdots$, *13*.

Symbol **0** is allowed in any state except *13*:

$0 \leq i \leq 11$: $T(i,0) = i + 1$.

Symbol **1** is not allowed in states *1* through *5*. When symbol **1** is allowed, the next state is always state *1*:

$6 \leq i \leq 13$: $T(i,1) = 1$.

The nonzero augends D are approximations to the values $W^{-\ell(u)}$ which correspond to symbol **0** occurring when there is a binary choice; recall Eq. (24).

With denominator C for the rate, there are only three distinct values for the augend factors: $M(0,0)$, $M(1,0)$, and $M(2,0)$. To be self-consistent these values must satisfy inequality (27). We employ Eq. (11b) as the basis for conservatively approximating $M(X,\omega + 1)$ by taking the first three nonzero terms of Eq. (25).

We can relate this approach to the derivation of Eq. (27) from Eq. (26) as follows. For the (5,12) code, five **0**'s are required following the occurrence of a **1**. Therefore, the first nonzero augend term following $u.1$ occurs at the sixth **0**, i.e., at string $u.1.0.0.0.0.0.0$. $E(u)$ introduces a common factor on each side of Eq. (26) which cancels, and the left-hand side of Eq. (26) thus becomes the same as in Eq. (27). Equation (29) below results, showing the approximation to $M(X,\omega + 1)$ as a sum of three terms:

$$\begin{aligned} M(X,0) \leq\ & M(X + 6 \bmod 3,0) \times 2^{-\lfloor X+6 \rfloor} \\ & + M(X + 7 \bmod 3,0) \times 2^{-\lfloor X+7 \rfloor} \\ & + M(X + 8 \bmod 3,0) \times 2^{-\lfloor X+8 \rfloor} \\ & < M(X,\omega + 1). \end{aligned} \quad (29)$$

Let $M(X,0)$ be integers, and try to find the smallest value for $M(0,0)$. Smaller integer values for M minimize the maximum number of bits needed for the binary representation to represent each $M(X,0)$. For the (5,12) code example, the smallest integer augend factor $M(0,0)$ is 5, which works with $M(1,0) = 4$ and $M(2,0) = 3$.

• *Example channel string*

Let the data string to be encoded be 101010101010101010 ⋯. The column headed i in Table 3 gives the state of the channel, assumed to be "home" state 1 initially. The column headed X gives the value of the internal variable X, which need not be initialized to 0. Column M gives the aligned value of the augend. Column ξ gives the channel symbol generated. The initial conditions for E and X must be such that $D(\Lambda,\omega + 1)$ equals or exceeds unity when the data string is viewed as a fraction. (Otherwise, data string 111111 ⋯ will not be representable as a channel string.) In this case, $E(\Lambda) = 2$.

5. Multistate, multisymbol constrained channels

An approach for a fixed rate L-based code suitable for multistate, multisymbol channels appears in [5]. The important notion is the approximation to W^{-1}, as in Eq. (21). Consider the case of a (2,7,8) code. It is convenient to consider that the symbol alphabet is not binary, but to use an extended alphabet of symbols **001, 0001, ⋯, 00000001.** Thus we can treat the channel as having six symbols of lengths in channel time units of 3, 4, 5, 6, 7, and 8. For these symbols, A. Patel has determined a 14-state CFSM.

Let string u applied to the channel from state 1 leave recursion variables $F(u)$, $E(u)$, and $X(u)$, and let the channel be in state $T(u)$. To transform the next symbol ξ we have

$$F(u.a) = F(u) + M(T(u), X(u),\xi) \times 2^{-E(u)}, \tag{30}$$

$$E(u.a) = \lfloor (\ell(u) + \ell(\xi)) \times J/C \rfloor , \tag{31a}$$

$$X(u.a = (\ell(u) + \ell(\xi)) \times J) \bmod C. \tag{31b}$$

Equation (31) constitutes the length recursion, a property of L-based arithmetic codes. The value $M(T(u),X,\xi) \times 2^{-E(u)}$ is the augend. Values of the type $M(i,X,\xi)$, where i is the state, X the channel unit time modulo C, and ξ a channel symbol, are the augend factors. The procedure for the calculation of the augend factors $M(i,X,\xi)$ is essentially an approximation to $P(i,\xi) \times 2^{-X/C}$, and such a procedure is given in [5].

A general length-based channel code needs an augend table for $M(i,X,\xi)$ of $n \times C \times (|\xi| - 1)$ entries, where $|\xi|$ is the cardinality of the set of channel symbols $\alpha, \cdots, \xi, \cdots, \omega$, n is the cardinality of the set of states (typical state i), and C is the denominator of the rational fixed rate J/C. (J and C are integers.) There are C possible retained fractional lengths, with numerators $0, 1, \cdots, C - 1$ (typical numerator X). The multiplier involving the cardinality $|\xi|$ of the symbol alphabet $\alpha, \cdots, \omega$ is decreased by one because any augend $M(i,X,\alpha)$ corresponding to the first symbol α has value 0. The (2,7,8) code at rate ½ yields $14 \times 2 \times (6 - 1)$ nonzero augends. A more economical alternative approach in terms of table size, which also offers other implementation advantages, is presented in a subsequent work [13].

Table 3 Example: channel coding by decompression algorithm.

i	X	M	ξ	*Data string*
1	2	0		1010101010101 ⋯
			0	*00*
2	0	0		1010101010101 ⋯
			0	*000*
3	1	0		1010101010101 ⋯
			0	*000*
4	2	0		1010101010101 ⋯
			0	*000*
5	0	0		1010101010101 ⋯
			0	*000*
6	1	4		1010101010101 ⋯
			0	*100*
7	2	3		0110101010101 ⋯
			0	*011*
8	0	5		0011101010101 ⋯
			0	*101*
9	1	4		0001001010101 ⋯
			1	*100*
1 etc.	2	0		0001001010101 ⋯

6. Summary

We have adopted the notion of a constrained channel, as defined by a finite state machine of allowed transitions. The CFSM is subject to the calculation procedure of Shannon, resulting in knowledge of the growth rate W of channel strings, the achievable channel capacity log W, and the Shannon probabilities. Decodability is to compression coding what representability is to channel coding. Arithmetic string coding is briefly reviewed. We map elements of one set of strings to elements of another set of strings, one source symbol at a time. A connecting viewpoint is that of the subdivision operation on a code space defined on the unit intervel. The L-based and P-based arithmetic coding approaches are covered. The notions of a subdivision gap and subdivision overlap are explained. Overlap results in nondecipherability of compression codes whereas a gap results in nonrepresentability for constrained channel codes. This duality results from the inversion of the compression and expansion operations (see Table 1) inherent in channel coding. Most of Section 4 is drawn from [5]. A simple (2,4) constraint is used as an example. The recursions for arithmetic codes are covered, and the problem of a subdivision gap is illustrated (see Fig. 5). The inequality for representability is given [Eqs. (13) and (14)]. The (d,k) constraints are particularly simple for studying the effect of the growth rate W. This study provides the basis for the simple fixed rate implementation of Martin's unit-time L-based code for run length limited (d,k) codes.

Acknowledgments

The basic ideas of the application of arithmetic coding to the constrained channel appear in an unpublished report by the senior author [5]. This paper explains, extends, and reformulates some of those ideas. We bring out the interesting decodability-representability duality, and base the explanation on ideas in [10].

W. M. Beynon helped in an earlier version of the paper. We have received much encouragement from Arvind Patel and Paul Siegel. Peter Franaszek provided support, discussions, and several helpful suggestions. J. Rissanen provided some valuable insights and discussions.

References

1. P. A. Franaszek, "Sequence-State Coding for Digital Transmission," *Bell Syst. Tech. J.* **47,** 1, 143–157 (1968).
2. P. A. Franaszek, "A General Method for Channel Coding," *IBM J. Res. Develop.* **24,** 5, 638–641 (1980).
3. P. A. Franaszek, "Construction of Bounded Delay Codes for Discrete Noiseless Channels," *IBM J. Res. Develop.*, **26,** 4, 506–514 (1982).
4. R. Adler, D. Coppersmith, and M. Hassner, "Algorithms for Sliding Block Codes," *IEEE Trans. Info. Theory*, **IT-9,** 1, (1983).
5. G. Nigel N. Martin, "Range Encoding for Discrete Noiseless Channels," unpublished internal memorandum, IBM UK Scientific Centre, March 1980.
6. C. E. Shannon, "A Mathematical Theory of Communication," *Bell Syst. Tech. J.* **27,** 379–423 (1948).
7. P. A. Franaszek, "On Future-Dependent Block Coding for Input-Restricted Channels," *IBM J. Res. Develop.* **23,** 1, 75–81 (1979).
8. S. J. Hong and D. L. Ostapko, "Codes for Self-Clocking AC-coupled Transmission: Aspects of Synthesis and Analysis," *IBM J. Res. Develop.* **19,** 4, 358–365 (1975).
9. M. Guazzo, "A General Minimum-Redundancy Source-Coding Algorithm," *IEEE Trans. Info. Theory* **IT-26,** 15–25 (1980).
10. Glen G. Langdon, Jr., "Tutorial on Arithmetic Coding," *Research Report RJ-3128*, IBM San Jose Research Laboratory, San Jose, CA, 1981.
11. J. J. Rissanen and G. G. Langdon, Jr., "Arithmetic Coding," *IBM J. Res. Develop.* **23,** 2, 149–162 (1979).
12. G. Nigel N. Martin, "Range Encoding: An Algorithm for Removing Redundancy for a Digitised Message," *Video and Data Recording Conference*, Southampton, England, July 24–27, 1979.
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,** 2, 107–115 (1983, this issue).

Received January 22, 1982; revised September 29, 1982

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. 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 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.

G. Nigel N. Martin *IBM United Kingdom Laboratories Ltd., Hursley House, Hursley Park, Winchester, Hampshire S021 2JN, England.* Mr. Martin joined IBM Information Services in 1968 as a systems programmer. In 1973, he joined the IBM United Kingdom Scientific Center, where he worked with others to provide a relational database interface to IMS. He took a year at Warwick University in 1978 to research data access techniques. He is currently working on data channels at Hursley. Mr. Martin received a B.A. from Cambridge University and an M.A. in 1968.

Stephen J. P. Todd *IBM United Kingdom Scientific Centre, Athelstan House, St. Clement Street, Winchester, Hants S023 9DR, England.* Mr. Todd joined the IBM Scientific Center in 1971 and has worked there since except for a two-year assignment at the Research Division in San Jose, California. During this assignment, he researched text processing and coding theory. Most of his other work with IBM has been on relational data base systems, with some work on automated offices and image processing. He is currently working on graphics. Mr. Todd received a B.A. in mathematics from Oxford University, England, in 1968, and an M.A. in 1969.