Finite-State Modulation Codes for Data Storage

Brian H. Marcus, *Member, IEEE*, Paul H. Siegel, *Senior Member, IEEE*, and Jack K. Wolf, *Fellow, IEEE*

***Abstract*—This paper provides a self-contained exposition of modulation code design methods based upon the state splitting algorithm. The techniques are applied to the design of several codes of interest in digital data recording.**

I. Introduction

Modulation codes in digital recording systems serve a variety of important roles. In the most general terms, their purpose is to improve the performance of the system by matching the characteristics of the recorded signals to those of the channel. Key factors determining the recording code constraints include: channel bandwidth restrictions, the timing recovery method, the gain control method, the detection technique, and the electronic implementation approach.

For example, in magnetic recording systems using peak-detection [1], the main objectives are to compress the signal bandwidth; optimize the trade-off between intersymbol interference, detection window size, and data density; support data-driven timing and gain control; and minimize implementation cost. A widely used family of constraints are the (d, k) or run-length-limited (RLL) constraints, where the run of "0" symbols between consecutive "1" symbols must have length at least d and no more than k [2], [3]. The reasons for the imposition of such constraints are discussed in more detail in Section VI-A. Many commercial systems today use a code with constraints $(d, k) = (2, 7)$. An example of a string satisfying these constraints is

$$\cdots 00100001001000000100 \cdots .$$

The set of all sequences satisfying these constraints is conveniently described by reading the labels off of paths through the diagram in Fig. 1. We will have much more to say about such diagrams in the remaining sections of the paper.

More recently, magnetic recording systems using sampling detectors have appeared on the scene. These employ a scheme, denoted PRML, based upon class-4, partial-response signaling, with maximum-likelihood sequence estimation, [4]–[7]. As discussed in more detail in Section VII, it is desirable to use sequences which satisfy not only a "global" k constraint, denoted $\boldsymbol{G}$, but also a separate "interleaved" k constraint on the even index and odd index substrings, denoted $\boldsymbol{I}$. We will refer to these constraints as $(0, \boldsymbol{G}/\boldsymbol{I})$ constraints. An example of a string satisfying the $(0, 4/4)$ constraints is

$$\cdots 0010000100100001001100 \cdots .$$

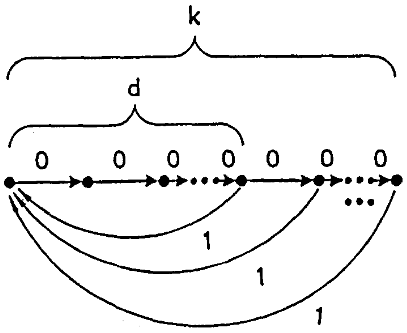


Fig. 1. Run-length limited (d, k) constraint.

Another commonly encountered constraint requires that the channel input waveforms have no spectral content at a particular frequency f, typically zero frequency or the Nyquist frequency (one-half the code symbol frequency). The code is said to have a spectral null at f in the code power density spectrum. Sequences with a spectral null at $f = 0$, often called dc-free sequences, have been used in many tape recording systems employing rotary-type recording heads, such as the R-DAT digital audio tape systems. More recently, it has been shown that dc-free sequences are also of potential value in trellis-coded PRML systems [8]. Sequences with spectral null at $f = 0$ are generated by diagrams of the form shown in Fig. 2. (The more "links" there are in the "sausage," the less attenuation there will be in the low frequencies near the spectral null.)

Once the system of constrained sequences, denoted S, is specified, one needs a code that translates incoming information into the sequences that obey the constraints. We will often refer to such a code as an *S code*. So, a (d, k) code is a mapping that encodes arbitrary binary data into the (d, k) run-length limited constraint.

The encoder typically takes the form of a synchronous finite-state-machine, shown schematically in Fig. 3. A *rate* $p:q$ *finite state encoder* accepts an input block of p user bits and generates a length q codeword depending on the input block and the current internal state of the encoder. There are of course only finitely many states. This structure is ideally suited to digital logic implementation using combinatorial logic or look-up tables.

Manuscript received September 1990; revised August 15, 1991.
B. H. Marcus and P. H. Siegel are with the IBM Research Division, Almaden Research Center, San Jose, CA 95120.
J. K. Wolf is with the Center for Magnetic Recording Research, University of California, San Diego, La Jolla, CA 92093.
IEEE Log Number 9104233.

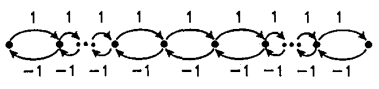


Fig. 2. Charge constraint.

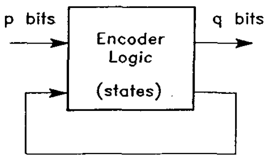


Fig. 3. Finite-state encoder.

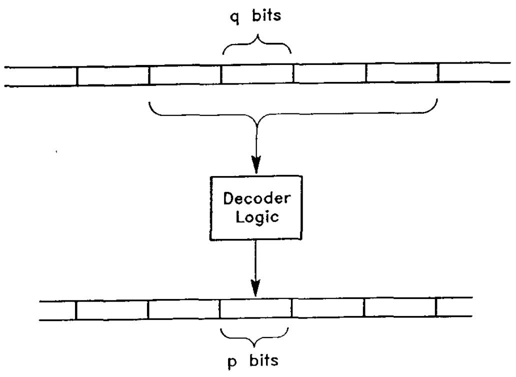


Fig. 4. Sliding-block decoder.

As is customary, we sometimes use the term *rate* to mean the ratio p/q, instead of the pair of block sizes $p:q$. It will be clear from the notation which version of the term "rate" we mean.

The encoders that we construct will automatically be decodable. The weakest type of decoder that we consider is a *state-dependent decoder* which accepts, as input, codewords of length q and generates a block of p user bits depending on the internal state, the input codeword, as well as finitely many upcoming codewords. Such a decoder will invert the encoder when applied to valid code sequences, effectively retracing the state sequence followed by the encoder in generating the code sequence. However, when the code is used in the context of a noisy channel, the state-dependent decoder may run into a serious problem. The noise causes errors in the detection of the code sequences, and the decoder must cope with erroneously detected sequences, including sequences that are not even valid code strings. It is generally very important that the decoder confine the propagation of errors at the decoder output resulting from such an error at the decoder input. Unfortunately, an error at the input to a state-dependent decoder can cause the decoder to lose track of the encoder state sequence, with no guarantee of recovery and with the possibility of unbounded error propagation.

The decoder therefore needs to have additional properties. Specifically, any input symbol error should give rise to a bounded number of output (decoded) bit errors. We call an encoder, for which there is such a decoder, *noncatastrophic*. This is a standard concept in the theory of convolutional codes. In that setting, this definition constrains the time span in which the bounded number of errors must occur. However, in general, it does not. Since, in practice, it is preferable to have these output errors occur within a bounded time interval, we often seek a decoder that is defined by a *sliding-block* mapping. Such a decoder (called a *sliding-block decoder*) makes a decision on a given received word on the basis of the local context of the word in the received sequence: the word itself, as well as a fixed number m of preceding words and a fixed number a of later words. The preceding symbols constitute what is called the *memory* (m) of the decoder, and the following symbols are called the *anticipation* (a). Fig. 4 shows a schematic diagram of a sliding-block decoder. This function can be realized as a shift register with combinatorial logic attached. It is easy to see that a single error at the input to a sliding-block decoder can only affect the decoding of words that fall in a "window" of length at most $m + a + 1$ words.

The problem for the code designer is to construct a code with these encoder/decoder attributes. In addition, it is desirable that the code be *efficient*, where efficiency has a very precise meaning, established by Shannon in his classic paper [9]. Shannon proved that the rate $R = p/q$ of a constrained code cannot exceed a quantity, now referred to as the Shannon capacity C, that depends only upon the constraint. He also gave a nonconstructive proof of the existence of codes at rates less than, but arbitrarily close to, the capacity. The measure of efficiency of a code is the ratio R/C.

Progress was made in the 1950's, 1960's, and 1970's in finding techniques for generating practical, efficient modulation codes. In his pioneering work, Franaszek [10]-[14] (see also Beal [15]) developed construction methods that advanced the theory of code design and invented specific codes that played important roles in the digital data recording industry. In addition, Tang and Bahl [16], Jacoby [17]-[19], Lempel and Cohn [20], Patel [21], and others, made many important contributions. Nevertheless, there remained the following fundamental questions. For a given rate R, what block sizes p, q such that $R = p/q$ can be realized? When and how can encoders with sliding-block decoders be found? If the Shannon capacity is a rational number of $C = p/q$, can a 100% efficient sliding-block code be designed?

In the 1980's, many of these basic questions were answered. The major breakthrough occurred with the introduction by Adler, Coppersmith, and Hassner [22] of techniques that originated from a mathematical discipline called symbolic dynamics. This approach has brought to recording code design a new level of mathematical rigor and generality. In [22], as well as subsequent papers by Marcus [23], and Karabed-Marcus [24], practical properties of encoder and decoder mappings were translated into mathematical terms, families of sequences that play a distinguished role in coding were precisely characterized, and definitive theorems about the existence of code mappings were formulated and proved. Moreover, the proofs of the theorems provided a set of code construction techniques that have proven to be of practical value to code designers. Reference [22] provides an algorithm—the state splitting algorithm—for constructing efficient encoders with sliding-block decoders for the family of *finite type* constraints that includes, for example, all RLL (d, k) constraints and PRML $(0, G/I)$ constraints. References [23] and [24] develop additional techniques, and they extend the sliding-block code existence results to the larger family of *almost-finite type* systems, such as the spectral-null constraints. They also show the existence of *noncatastrophic* codes (which, we recall, are slightly weaker than sliding-block codes in terms of error propagation) for systems not included in the previous families.

As important as these recent papers are for practical code design, however, the reader uninitiated in symbolic dynamics may, at times, find them difficult to follow. The aim of this paper is twofold. First, using only elementary mathematical concepts, we hope to provide a self-contained, yet rigorous exposition of the state splitting algorithm in [22]. We also include a collection of "tricks of the trade" that are of value to the coding practitioner, but have not appeared together in readily accessible prior literature or textbooks [25], [26].

Second, we illustrate the application of these tools in the construction of practical (d, k) codes for recording channels using peak detection and $(0, G/I)$ codes for channels using partial-response with maximum-likelihood (PRML) detection.

We remark that there are several other recent expositions of the state splitting algorithm that the interested reader might wish to consult; Blahut [27, ch. 8]; Immink [28, ch. 5]; Khayrallah and Neuhoff [29]; and Swenson and Cioffi [30].

The remainder of this paper is organized as follows. In Section II, we review the necessary background on finite state transition diagrams, constrained systems, and Shannon capacity. In Section III, we present, in detail, the state splitting algorithm for constructing finite state encoders. In particular, Section III-E summarizes the algorithm in a step-by-step fashion. These encoders automatically have state-dependent decoders. In Section IV, we show that for the class of finite-type constrained systems, the encoders constructed in Section III can be made to have sliding-block decoders. In Section V, we consider practical techniques for reducing the number of encoder states as well as the size of the sliding-block decoder window. In Sections VI and VII, we apply the techniques to run-length limited systems and PRML constraints. In Section VIII, we discuss the class of almost-finite-type systems and state the general results which yield noncatastrophic encoders.

II. Background

The reader is no doubt familiar with the colloquial expression: no pain, no gain. The whole point of this paper is to share the gain, while minimizing the pain, when it comes to constructing finite-state codes for recording channels. However, there are some important concepts and related terminology that we need in order to proceed with the code construction algorithm, as well as to understand its strengths and limitations. The purpose of this section is to present and illustrate in an intuitive manner these few necessary mathematical notions.

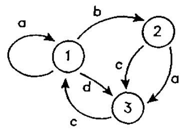


Fig. 5. Typical FSTD.

First, we introduce a convenient diagrammatic method used to represent the set of constrained sequences from which code sequences will be drawn.

A *finite-state transition-diagram* (FSTD) G is a directed graph with a finite number of states (vertices) and edges, and edge labels drawn from an alphabet containing a finite number of symbols. Fig. 5 shows a "typical" example of an FSTD.

There are a few features worth highlighting. Since the graph is directed, each edge can be traversed in only one direction, as indicated by the arrow. Self-loops, meaning edges that start and end at the same state, are allowed. Also there can be more than one edge connecting a given state to another state. (In practice, however, distinct edges that share the same starting and ending states will always have distinct labels.) We will use the symbol $V(G)$ to denote the set of states of G and the symbol $E(G)$ to denote the set of edges of G.

We will sometimes use the symbol G to denote just the graph itself (without the labels), but the meaning should always be clear from the context.

The FSTD can be used, as follows, to generate finite symbol sequences, sometimes referred to as *strings*. We pick a starting state in G, and then follow a path consisting of a sequence of edges, always respecting the arrows indicating the allowed direction. As each edge in the path is traversed, we read off the corresponding label, thereby producing a sequence of symbols. For example, in Fig 5, the symbol sequence $a\ b\ c\ c\ d$ can be generated by following a path along edges with state sequence 1 1 2 3 1 3. We will call a sequence of length n generated by G an *n-block*.

The set of all finite sequences generated by an FSTD will be called a *constrained system* or *constraint*, denoted by S. We say that the FSTD *represents* the constrained system S. Constrained systems are closely related to regular languages in automata theory and sofic systems in symbolic dynamics.

As will become apparent in the next section, an FSTD is the starting point for the code construction procedure. It is therefore important to understand various basic properties of FSTD's and their underlying graphs, as well as certain relationships among FSTD's.

For one thing, a constrained system should not be confused with any particular FSTD, because a given constrained system can be represented by many different FSTD's. For example, the RLL (0, 1) constrained system can be represented by the FSTD's in Fig. 6, which are very different from one another.

For purposes of code construction, it is important to consider FSTD's that have special properties with respect to their labelings. For example, an FSTD is called *deterministic* if at each state the outgoing edges are labeled distinctly. In other words, at each state, any label generated from that state uniquely determines an outgoing edge from that state. The FSTD's in Figs. 1, 2, and 6(a) and (b) are deterministic while the FSTD's in Fig. 6(c) and (d) are not. The reader, familiar with constrained systems in the literature, will notice that constrained systems are usually represented by deterministic FSTD's. It is well known that any constrained system can be represented by some deterministic FSTD [31, sec. 16-3].

We will in fact need to consider the following more general version of the deterministic property. An FSTD has *local anticipation* a if a is the smallest nonnegative integer such that, for every state i, all of the paths of length $a + 1$ that start at i and generate the same sequence begin with the same edge. In other words, knowledge of the initial state of a path and the first $a + 1$ symbols that it generates is sufficient information to determine the initial edge of the path. An FSTD has *finite local anticipation* if it has local anticipation a for some nonnegative integer a. An FSTD that does not have finite local anticipation will be said to have *infinite local anticipation*.

The adjective "local" is used to emphasize dependence on the knowledge of the initial state and also to distinguish this kind of anticipation from a more global kind of anticipation that will be introduced in Section IV. We note that saying that an FSTD is deterministic is equivalent to saying that it has local anticipation 0. The FSTD in Fig. 6(c) is a representation of the RLL (0, 1) constrained system with local anticipation 1 but not 0. Fig. 6(d) depicts an FSTD with infinite local anticipation.

There is also a notion of *finite local memory* obtained by considering paths that end at a given state. Most of what we say can be carried out with this concept rather than finite local anticipation. But by tradition we prefer anticipation over memory. ("It's better to look to the future than to dwell upon the past!") We will need the notion of finite local memory only in Section VIII.

We will be interested primarily in FSTD's that always permit you to "get there from here." That is, there is always a path in G from any specified starting state i to any specified destination state j. An FSTD with this property is called *irreducible*.

An FSTD is *reducible* if it is not *irreducible*. All of the FSTD's in Fig. 6 are irreducible, while Fig. 7(a) shows a reducible FSTD for the system of unconstrained binary sequences, RLL (0, ∞). A special case of reducibility is the property of being disconnected. An FSTD G is *disconnected* if the set of states $V(G)$ can be broken into two disjoint subsets, $V_1(G)$ and $V_2(G)$, in such a way that no edge e in $E(G)$ begins at a state in $V_1(G)$ and ends at a state in $V_2(G)$, or vice-versa. An FSTD that is not disconnected is called *connected*. Fig. 7(a) shows that a reducible FSTD can still be connected. (We also remark that the expression *strongly connected* is sometimes used to mean "irreducible.") The graph in Fig. 7 (b) is disconnected.

It will be useful later to know that any reducible FSTD can, in some sense, be broken down into "maximal" irreducible pieces. To make this more precise we introduce the concept of an *irreducible component* of an FSTD G. An *irreducible component* of an FSTD G is an irreducible FSTD that is contained in G and is not properly contained in any bigger irreducible FSTD that is contained in G. Fig. 7(b) shows the irreducible components of the FSTD in Fig. 7(a). From the point of view of finite-state code construction, we can concern ourselves primarily with irreducible FSTD's by using irreducible components.

As mentioned in the Introduction, a rate $p : q$ finite-state encoder will generate a sequence, composed of length-q codewords (q-blocks), that belongs to the desired, constrained system S. For a system S described by

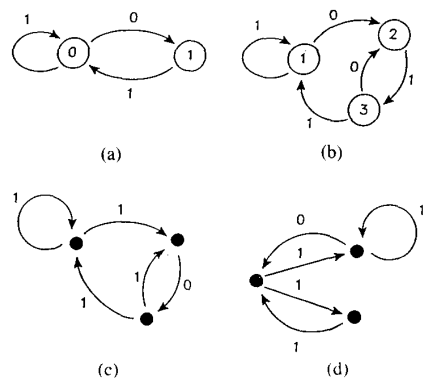


Fig. 6. (a) FSTD for RLL (0, 1). (b) Another FSTD for RLL (0, 1). (c) Yet another FSTD for RLL (0, 1). (d) One more FSTD for RLL (0, 1).

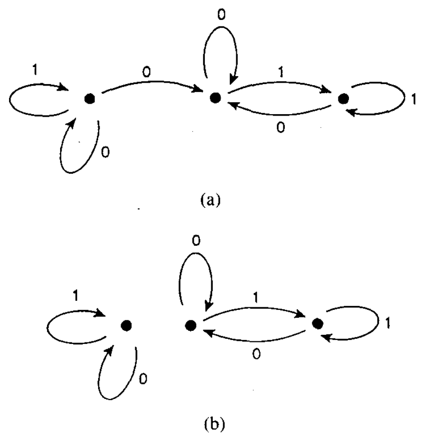


Fig. 7. (a) Reducible FSTD for unconstrained binary sequences. (b) Irreducible components.

an FSTD G, it will be very useful to have an explicit description of the sequences in S, grouped into such nonoverlapping "chunks" of length q. We can obtain such a description by defining another FSTD, called the *qth power of* G and denoted G^q. The FSTD G^q has the exact same set of states as G, but each edge in G^q corresponds to a path of length q in G, and the edge label is the q-block generated by that path. The constrained system generated by G^q is denoted S^q. Its alphabet consists of the q-blocks in S. If G has finite local anticipation, so does G^q.

For example, Fig. 8 shows the third power G^3 of the FSTD G in Fig. 6(a) that represents the RLL (0, 1) constraint.

There are times when the qth power of an FSTD G will not be irreducible, even if G is. For example, Fig. 9 shows an FSTD describing a system S of charge-constrained sequences (with a spectral null at zero frequency). Its second power G^2, shown in Fig. 10, has two irreducible components.

Note that both FSTD's in Figs. 6(a) and 9 are irreducible. These two examples illustrate the general situation: it can be shown that, if G is an irreducible FSTD, then any power G^q is either irreducible, or it is disconnected and decomposes into disjoint, irreducible components (see Figs. 8 and 10). If G^q is irreducible for all powers $q \geq 1$, then G is said to be *aperiodic*. If G^q decomposes into q disjoint components, each of which is aperiodic (and therefore irreducible), then G is said to have *period* q. (In particular, if G is aperiodic, it has period 1.) It can be shown that if G has period p, then the number of disjoint components in G^q will be $gcd(p, q)$, the greatest common divisor of p and q.

The period of G can also be defined as the greatest common divisor of the lengths of all cycles in G, where a cycle is a path whose ending state is the same as its initial state. For example, it is not difficult to check that the charge constraint FSTD in Fig. 9 has period 2. The concept of periodicity, although important in studying certain aspects of constrained systems (see [39]), does not play a role in the remainder of the paper. We refer the interested reader to [29] or any text on graph theory.

We require one more crucial concept, namely the *Shannon capacity* or simply *capacity* of the constrained system S, before we can launch into the coding constructions. The Shannon capacity C measures the growth rate of the number of strings of length n in S, meaning that the number of valid strings of length n, for large enough n, is well approximated by the upper bound 2^{Cn}. More precisely, if we let $N(n; S)$ denote the number of sequences of length n in S, the Shannon capacity, which we will often denote $Cap(S)$, is defined by

$$Cap(S) = \lim_{n \to \infty} \frac{\log_2 (N(n; S))}{n}.$$

Shannon [9] showed that this quantity provides an upper bound on the achievable rate of any finite-state code into the system S. Moreover, given positive integers p, q

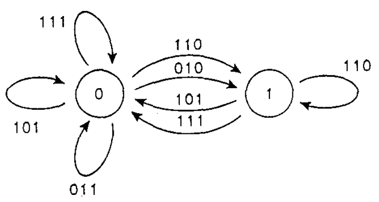


Fig. 8. FSTD for third power of FSTD in Fig. 6(a).

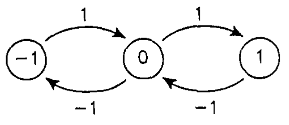


Fig. 9. FSTD for a particular charge constraint.

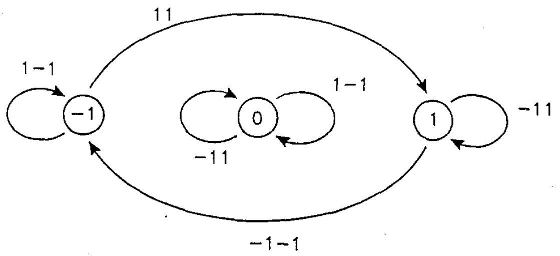


Fig. 10. FSTD for second power of FSTD in Fig. 9.

(relatively prime) satisfying the inequality

$$p/q < Cap(S)$$

Shannon proved (nonconstructively) that there is an integer k and 2^{kp} blocks in S of length kq.

The Shannon capacity $Cap(S)$ is easily computed from any FSTD G representing S, provided that G has finite local anticipation, in particular, if G is deterministic. The capacity is computed from a matrix, describing the interconnections in G, defined as follows. Suppose that the number of states in $V(G)$ is r. The *adjacency matrix* (or *state-transition matrix*) $A = A(G) = \{a_{ij}\}$ is the r-by-r matrix where the entry a_{ij} is the number of edges from state i to state j in G. The matrix therefore will have nonnegative, integer entries.

Given an FSTD G with finite local anticipation and adjacency matrix A, the Shannon capacity turns out to be

$$Cap(S) = \log_2 \lambda(A) \qquad (1)$$

where $\lambda(A)$ is the largest real eigenvalue of A. (See Theorem A3 in Appendix A.) The existence of a positive real eigenvalue is guaranteed by the Perron–Frobenius theory of nonnegative real matrices, but we need not worry about such theoretical details right now. Any good linear algebra software package will gladly find the eigenvalue λ(A) for you. The Perron–Frobenius theory is discussed briefly in Appendix A.

As an example, for the RLL (0, 1) constraint represented by the deterministic FSTD in Fig. 6(a), the adjacency matrix is

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

with largest eigenvalue

$$\lambda = \frac{1 + \sqrt{5}}{2}$$

and capacity

$$Cap(S) = \log_2 (\lambda) = 0.6942 \cdots .$$

The adjacency matrix $A(G^q)$ of G^q is easily seen to be $A(G)^q$, the qth power of the matrix $A(G)$. From linear algebra, it follows that the Shannon capacity $Cap(S^q)$ of the constraint S^q described by G^q is q times the capacity $Cap(S)$ of the constraint S defined by G:

$$Cap(S^q) = qCap(S).$$

We remarked earlier that, for the purposes of constructing a code into the constrained system S, we could restrict our attention to irreducible components of an FSTD representing S. We do not pay any price in terms of achievable code rate in doing so, because if the system S is represented by a reducible FSTD G, then there is an irreducible component of G which represents a subsystem of S whose capacity equals $Cap(S)$ (see Theorems A2 and A3 in Appendix A). The fact is, however, that most constraints of interest can be represented by an irreducible (and, in fact, deterministic) FSTD G.

As mentioned above, it may happen that the power G^q of an irreducible FSTD G decomposes into disjoint, irreducible components. It can be shown that the adjacency matrices of these irreducible components will all share the same largest eigenvalue. In particular, if G has finite local anticipation, then the constrained systems represented by these irreducible pieces of G^q all have the same Shannon capacity $qCap(S)$. For more information on properties of constrained systems and FSTD's, we again refer the reader to [29].

III. Finite-State Code Construction

In this section, we provide a completely self-contained exposition of the state splitting algorithm for constructing finite-state codes. In Section IV, the result will be improved to obtain finite-state codes with sliding-block decoders for a natural class of constrained systems.

Theorem 1 (Finite-State Coding Theorem): Let S be a constrained system with Shannon capacity $Cap(S)$. Let p, q be positive integers satisfying the inequality.

$$p/q \leq Cap(S)$$

Then, there exists a finite-state encoder with state-dependent decoder that encodes binary data into the constraint S at constant rate $p:q$.

The theorem, which can be derived as a weaker version of the main theorem of [22], improves upon earlier coding results in the following two important ways.

1) It proves the existence of finite-state codes that achieve rate equal to the capacity $Cap(S)$, when $Cap(S)$ is rational.

2) For any positive integers p, q satisfying the inequality, there is a code that operates at rate $p:q$. In particular, choosing p, q relatively prime, one can design an encoder/decoder using the smallest possible codeword length compatible with the chosen rate $R = p/q$.

The terminology and presentation we will use is intended to allow the reader to properly formulate practical coding problems, and to use the code construction techniques to explore various code design options for the intended application. The steps in the algorithm are summarized in Section III-E, and they can all be effectively implemented as part of a computer software package (as has been done by several researchers), providing a tool kit for the serious code designer.

Although the design procedure based upon the proof of Theorem 1 can be made completely systematic—in the sense of having the computer automatically generate an encoder and decoder for any valid code rate—the application of the method to just about any nontrivial code design problem will benefit from the interactive involvement of the code designers: there is still plenty of room for the "artist" in coding. There are also several practical tools—"tricks of the trade"—that can help the designer make "good" choices during the construction process. We will discuss these code simplification techniques in Section V, and then demonstrate their effectiveness in the examples described in Sections VI and VII.

It should be stressed, however, that despite these significant advances in code construction methods, the general problem of designing codes that achieve, for example, the minimum number of encoder states, minimum error propagation, or the less precise feature of minimum hardware complexity, still lacks an explicit solution. This remains an active research topic, as exemplified by recent papers where lower bounds on the number of encoder states [32] and bounds on the minimum sliding-block decoder window [33], [34] are derived.

A. State-Splitting

The proof of Theorem 1 relies upon a technique called *state splitting*, which we now motivate and define.

Suppose that a given FSTD G represents the constrained system S from which our code sequences will be drawn. If we select an achievable rate $R = p/q \leq Cap(S)$, and want to code at rate $p:q$, we first look at the qth power of G, denoted G^q, whose states, we recall, are the same as G, but whose edges correspond to paths of length q in G, and whose edge-labels are the corresponding strings of length q in S. The simplest condition that would permit us to use G^q as a rate $p:q$ finite-state encoder would be for G^q to have *outdegree* at least 2^p, that is, at least 2^p edges emanating from each state in G^q. Then, the encoder could generate, from each state and for each data p-block, a q-block in S.

Actually, it would suffice for G^q to have a sub-FSTD

K with this property. This condition can be rephrased nicely in terms of the adjacency matrix A^q for G^q. (Recall that $A^q = A(G^q) = A(G)^q$.) The matrix formulation of this condition is

$$A^q u \geq 2^p u \tag{2}$$

where the vector u is a "0-1" column vector, that is, a vector with components either 0 or 1. The sub-FSTD K determined by the states with corresponding components in u equal to 1 will satisfy the condition that the outdegree is at least 2^p.

Let

$$R(i) \triangleq A(K)_{i,1} + \cdots + A(K)_{i,N}$$

where $A(K)_{i,j}$ is the number of edges in K from state i to state j. The quantity $R(i)$ is often called the *rowsum*, and it indicates the number of edges in K that emanate from state i. When the rowsum $R(i)$ equals or exceeds 2^p, we can select exactly 2^p outgoing edges from state i, and assign to each a distinct binary p-block. For this reason, we call the condition (2), with a "0-1" vector u, the *rowsum condition*. We say that K is an *encoder graph* or an *encoder FSTD*.

Example: Consider the constrained system represented by the FSTD in Fig. 11; the symbols are binary 2-blocks (after the slash). Each state has outdegree 2, and the assignment of data bits (in front of the slash) to labels defines a rate 1:2 encoder from binary data into the constraint.

It is rare that the rowsum condition will be so easily satisfied. To see what can go wrong, consider the FSTD for the RLL (0, 1) constraint that, we recall, has capacity

$$Cap(S) = \log \frac{1+\sqrt{5}}{2} \approx 0.6942 > \frac{2}{3}.$$

For a rate 2:3 encoder, we look at G^3 (in Fig. 8) and $A(G^3)$, where

$$A(G^3) = A(G)^3 = \begin{bmatrix} 3 & 2 \\ 2 & 1 \end{bmatrix}.$$

The rowsums are $R(1) = 5$ and $R(2) = 3$. For $p = 2$, or $2^p = 4$, state 1 satisfies the desired rowsum inequality

$$R(1) = 5 \geq 2^2 = 4$$

but state 2, with $R(2) = 3$, clearly fails. If we remove state 2 (and all edges connected to it), state 1 loses 2 edges. Therefore, the rowsum in the matrix for this subsystem also falls short. In other words, we are stuck!

Having reached this impasse, one approach would be to look at higher powers of G, increasing the block length of codewords and data words, as was done in [10]-[12]. This approach has been successfully applied to generate several RLL codes of practical significance, but for some constrained systems and rates of practical interest, the block length required becomes prohibitively large.

The approach we will follow, in contrast, does not increase the block length, but instead uses graph construction techniques based on state splitting and eigenvectors,

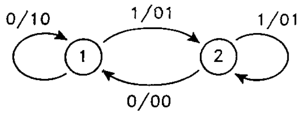


Fig. 11. Rate 1:2 encoder.

which have their roots in symbolic dynamics, where they were introduced by Williams [35] and Adler, Goodwyn, and Weiss [36]. The application of state splitting ideas in modulation coding was foreshadowed by Patel's construction of the zero-modulation (ZM) code [21]. See also Franaszek [13], [14] for related ideas.

The *state splitting algorithm* we describe involves an iterative application of a fundamental FSTD transformation that will allow us to create from G^q a new FSTD representing S^q. The procedure culminates in an FSTD representing S^q that satisfies the desired rowsum condition.

We now define the aforementioned fundamental transformation, which we call a *basic splitting*. We describe this for a general FSTD H and later apply it to the FSTD $H = G^q$.

Let i be a state in the FSTD H, and let E_i be the set of outgoing edges from state i. Let

$$E_i = E_i^1 \cup E_i^2$$

be a partition of E_i into two disjoint sets. This partition is used to define a new FSTD H' that changes the local picture around state i. The set of states of H', $V(H')$, consists of all states $j \neq i$ in H, as well as two new states denoted i_1 and i_2:

$$V(H') = \{j \in V(H) | j \neq i\} \cup \{i_1, i_2\}.$$

The states i_1 and i_2 are called *descendant states of state i*, and i is called the *parent state* of i_1 and i_2. A state in $V(H)$ that is not split is the parent of the corresponding state in $V(H')$.

The interconnections in H' that do not involve states i_1 and i_2 are inherited from H. That is, if there is an edge e from state j to state k in H, (with $j, k \neq i$) there is a corresponding edge in H'. For edges involving state i, we consider three cases.

Case 1: Let edge e in H start at a state $j \neq i$ and terminate in state i. This edge is replicated in H' to produce two edges e_1, from j to i_1, and e_2, from j to i_2.

Case 2: Let edge e in H start at state i and terminate in a state $j \neq i$, and suppose e belongs to the set E_i^k in the partition of E_i. In H' there is a corresponding edge from state i_k to state j.

Case 3: Let edge e be a self-loop at state i in H, and suppose that e belongs to E_i^k. In H' there will be two edges from state i_k corresponding to e, one to state i_1, the other to state i_2.

As with states, we refer to *descendant edges* in H' and *parent edges* in H. In all cases, the edge label of an edge in H' is the edge label of its parent edge in H. The change in the local picture at state i is shown in Fig. 12. In the figure, we have partitioned the set of edges E_i into subsets

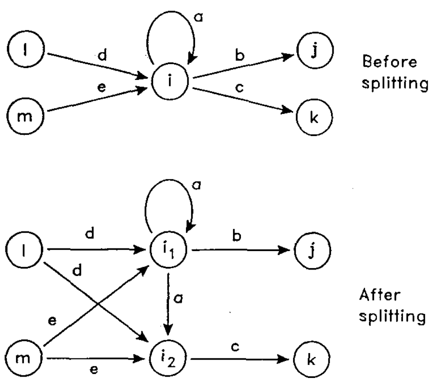


Fig. 12. Local picture of state splitting.

$E_i^1 = \{a, b\}$ and $E_i^2 = \{c\}$, where we are representing the edges by their labels. The state i splits into two states i_1 and i_2 according to the partition.

In general, a state splitting of state i may involve partitions into any number N of subsets. A *generalized splitting* begins with a partition of E_i into N disjoint subsets

$$E_i = E_i^1 \cup \cdots \cup E_i^N. \tag{3}$$

From the partition, we derive a new FSTD H'. The set of states $V(H')$ consists of all states $j \neq i$ in H, as well as descendant states $i_1, \cdots, i_N$. The interconnections are defined in an analogous manner to those in the case of a basic splitting. As in that situation, outgoing edges from state i in H are partitioned among its descendant states, according to (3), and incoming edges to state i in H are replicated in H' to each of the N descendant states. We leave it to the reader to check that this transformation can be broken down into a sequence of basic state splittings. We will make use of such generalized partitions in Section IV.

The FSTD H' obtained from H by a state splitting (basic or generalized) has several important characteristics, enumerated in the following proposition.

Proposition 1:

1) The system of sequences generated by paths in H' is exactly the same as the system generated by H.

2) If H represents the constrained system S with local anticipation a, then H' is a representation of S with local anticipation at most $a + 1$.

3) If H is irreducible, so is H'.

These properties can be verified by examining Fig. 12. For instance, property 2) is verified as follows. Given a state i' in $V(H')$, any sequence of $a + 2$ symbols that can be generated starting from i' by a path $\boldsymbol{e}'$ in H' is also generated by its parent path $\boldsymbol{e}$ (i.e., the path consisting of the parent edges of $\boldsymbol{e}'$) in H. Note that any such path $\boldsymbol{e}$ must start from the parent state i of i'. Since H has local anticipation a, the first two edges of $\boldsymbol{e}$ are uniquely determined. By the definition of state splitting, this in turn uniquely determines the first edge of $\boldsymbol{e}'$. Thus, H' has local anticipation at most $a + 1$. For a more complete proof, see [29], [30].

The fact that the local anticipation may increase under state splitting is the reason that we need to consider FSTD's that may not be deterministic.

To close out the discussion of state splitting, we mention a particular splitting that will be useful later—a *complete splitting*. This refers to the splitting obtained from the partition in which each set in the partition consists of a single, distinct edge. The resulting FSTD, denoted $H^{(2)}$, is called the *edge graph of* H, since it contains one state for each edge in H. It is also referred to as the *higher 2-block graph* of H in [22]. Fig. 6(c) shows the edge graph for the RLL (0, 1) FSTD in Fig. 6(a).

It is clear that, through state splitting, we can change the local picture, in particular the outdegree, of states in an FSTD $H = G^q$ representing the constraint S^q. The remarkable fact is that, when $Cap(S) \geq p/q$, there is a sequence of state splittings that will generate a new FSTD with outdegree at least 2^p. The tool that we will use to guide the evolution of this sequence of transformations is an inequality of the form (2), called an approximate eigenvector inequality, which we now discuss.

B. Approximate Eigenvectors

Earlier, we saw that a rate $p:q$ encoder for S, represented by the FSTD G, could be derived from the FSTD G^q if it contained a sub-FSTD K with outdegree at least 2^p at each state. This condition was captured by the matrix inequality (2)

$$A^q \boldsymbol{u} \geq 2^p \boldsymbol{u}$$

where $\boldsymbol{u}$ is a "0-1" column vector.

What can we say when the capacity condition

$$Cap(S) \geq \frac{p}{q}$$

is satisfied, but G^q does not satisfy the outdegree condition represented by such an inequality? How should the code designer proceed? The answer we will describe relies upon finding a nonnegative, integer vector $\boldsymbol{u}$, not necessarily "0-1", that satisfies the inequality (2).

As mentioned in Section II, we may assume that G is deterministic and thus

$$\lambda(A) = 2^{Cap(S)} \geq 2^{p/q}$$

or, equivalently,

$$\lambda(A^q) \geq 2^p.$$

The Perron–Frobenius theory of nonnegative matrices (see Theorem A4 in Appendix A) then ensures that we can find a nonzero vector $\boldsymbol{v} = [v_1, \cdots, v_r]^T$ with nonnegative integer components, satisfying the inequality

$$A^q \boldsymbol{v} \geq 2^p \boldsymbol{v}. \tag{4}$$

(By a nonzero vector, we mean a vector with at least one nonzero component.) The vector $\boldsymbol{v}$ is called an $(A^q, 2^p)$-*approximate eigenvector* and the inequality is called an *approximate eigenvector inequality*. This has a very simple meaning in terms of the graph G^q. Think of the vector $\boldsymbol{v}$ as assigning *state weights*: the weight of state i is v_i. Now assign weights (called *edge weights*) to the edges of the graph according to their terminal states:

$$w(e) = v_j$$

where j is the terminal state of e. Then the vector inequality (4) can be written as the set of simultaneous scalar inequalities, one for each state i,

$$\sum_{e \in E_i} w(e) \geq 2^p v_i.$$

That is, the sum of the weights of the outgoing edges from a given state i is at least 2^p times the weight of the state i itself.

Example: For the RLL (0, 1) constraint, with $p = 2$ and $q = 3$, we saw earlier that the inequality (2) was not satisfied by any nonzero "0-1" vector $\boldsymbol{u}$. However, it is easy to verify that we have the following inequality:

$$\begin{bmatrix} 3 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 8 \\ 5 \end{bmatrix} \geq 4 \begin{bmatrix} 2 \\ 1 \end{bmatrix}.$$

Therefore, the vector $\boldsymbol{v} = [2, 1]^T$ is an $(A^3, 4)$-approximate eigenvector.

We remark that, in practice, we can always assume that the components of an approximate eigenvector are in fact strictly positive, as follows. If $\boldsymbol{v}$ has components equal to 0, simply delete the states in G^q with weight 0 as well as their incident edges, producing a sub-FSTD K. The vector $\boldsymbol{w}$, obtained by restricting the approximate eigenvector $\boldsymbol{v}$ to K, will be an $(A(K), 2^p)$-approximate eigenvector. In this way, we can delete irrelevant states and thereby reduce bookkeeping.

In the next sections we will show how an approximate eigenvector $\boldsymbol{v}$ specifies a sequence of no more than $\Sigma_i (v_i - 1)$ state splittings starting with $H = G^q$ and leading to a new FSTD $\hat{H}$ with outdegree at least 2^p, and no more than $\Sigma_i v_i$ states. This FSTD can then be used to define a finite state encoder. First, however, we describe a simple algorithm, introduced by Franaszek [13] and based on integer programming, to compute approximate eigenvectors.

We describe Franaszek's algorithm in the setting of a general nonnegative integer matrix $T = (t_{ij})$ and positive integer n. We will then apply this to $T = A^q$ and $n = 2^p$. A (T, n)-approximate eigenvector will mean a nonnegative integer vector $\boldsymbol{v}$ (not all 0's) that satisfies

$$T\boldsymbol{v} \geq n\boldsymbol{v}. \tag{5}$$

If there is a (T, n)-approximate eigenvector with maximum component no larger than a specified integer L, the algorithm will identify one.

Approximate Eigenvector (AE) Algorithm

1) Set $k = 0$.

2) Set $\boldsymbol{v}^{(0)} = (L, L, \cdots, L)$.

3) For each coordinate i, define

$$v_i^{(k+1)} = \min\left(v_i^{(k)}, \left\lfloor \frac{1}{n} \sum_j t_{ij} v_j^{(k)} \right\rfloor\right).$$

4) If $\boldsymbol{v}^{(k+1)} \neq \boldsymbol{v}^{(k)}$, increase k and go to step 3).

5) If $\boldsymbol{v}^{(k+1)} = \boldsymbol{v}^{(k)}$, set $\boldsymbol{v} = \boldsymbol{v}^{(k)}$.

This algorithm always produces a nonnegative integer vector $\boldsymbol{v}$. If $\boldsymbol{v} \equiv 0$, this means that L was chosen too small, and no approximate eigenvector with components at most L exists. The algorithm should then be repeated after increasing L. One approach commonly followed is to start with $L = 1$, and increment it by 1 until an approximate eigenvector is found. This will in fact find an approximate eigenvector with smallest (among all possible approximate eigenvectors) maximal component. If the latter is what is really desired, it is more efficient, on the average, to apply the algorithm with $L = 1, 2, 4, 8, \cdots$, until an approximate eigenvector is found—say with $L = 2^l$. Then perform a binary search on the numbers in between 2^{l-1} and 2^l.

It is important to mention, and not difficult to prove [22, Appendix], that the vector $\boldsymbol{v}$ produced by this algorithm is the "largest" approximate eigenvector that is componentwise less than or equal to the starting vector. In other words, if the vector $\boldsymbol{u}$ is an approximate eigenvector satisfying $\boldsymbol{u} \leq \boldsymbol{v}^{(0)}$, then $\boldsymbol{u} \leq \boldsymbol{v}$. This fact, which follows easily from step 3) in the algorithm, will be used in Appendix B, where we discuss the construction of optimal block codes. These statements remain true if the initial vector, $v^{(0)}$, is a nonnegative vector.

C. Basic $\boldsymbol{v}$-Consistent Splitting

Let H be an FSTD, $T = A(H)$ and n be a positive integer. Given a (T, n)-approximate eigenvector $\boldsymbol{v}$, we define the concept of a *basic $\boldsymbol{v}$-consistent splitting.*

Recall that E_i denotes the set of edges outgoing from state i. A *basic $\boldsymbol{v}$-consistent partition of E_i* is a partition

$$E_i = E_i^1 \cup E_i^2$$

with the property that

$$\|E_i^1\| \triangleq \sum_{e \in E_i^1} w(e) \geq y_1 n$$

and

$$\|E_i^2\| \triangleq \sum_{e \in E_i^2} w(e) \geq y_2 n$$

where y_1 and y_2 are integers satisfying

$$y_1 \geq 1 \quad \text{and} \quad y_2 \geq 1$$

and

$$y_1 + y_2 = v_i.$$

Note that the conditions on $\|E_i^1\|$ and $\|E_i^2\|$ are inequalities. However, in the next subsection, we will obtain partitions so that the condition on $\|E_i^1\|$ is actually an equality.

The splitting determined by this partition is called a *basic $\boldsymbol{v}$-consistent state splitting* of state i, and we denote the resulting FSTD and adjacency matrix by H' and T'. It is straightforward to check that the vector $\boldsymbol{v}'$, indexed by the states of H', defined by

$$v'_j = \begin{cases} v_j & \text{if } j \neq i \\ y_1 & \text{if } j = i_1 \\ y_2 & \text{if } j = i_2 \end{cases}$$

is a (T', n)-approximate eigenvector.

We summarize in Proposition 2 the important features of basic $\boldsymbol{v}$-consistent state splittings, which the reader may enjoy verifying.

Proposition 2: Let the FSTD H' be obtained from the FSTD H by a basic $\boldsymbol{v}$-consistent state splitting.

1) The FSTD H' has one more state than H.

2) $\Sigma_k v_k = \Sigma_k v'_k$ (and v_i in $\boldsymbol{v}$ is replaced in $\boldsymbol{v}'$ by two strictly smaller positive integer components that sum to v_i).

Example: Fig. 13 shows the result of a basic $\boldsymbol{v}$-consistent splitting for the third power (shown in Fig. 8) of the RLL (0, 1) FSTD. Here, $v = [2, 1]^T$. State 0 is split into two descendant states, 0_1 and 0_2, according to the partition $E_i^1 = \{011, 110, 010\}$ and $E_i^2 = \{101, 111\}$. (Note that, strictly speaking, the elements of these sets are edges, but in this example we can denote them by their labels without any ambiguity.) The induced approximate eigenvector for the split graph is $\boldsymbol{v}' = [1, 1, 1]^T$, and the resulting FSTD is therefore an encoder graph.

Remark: It is worth noting that in [22], the transition matrix T was required to have entries that were either 0 or 1. In terms of the FSTD H, this translates into the requirement that, for states i and j, there be at most one edge in H from i to j. For any FSTD H, one can obtain another FSTD, representing the same constraint and having this property, by applying the edge graph construction, mentioned earlier. In general, we have found that practical code design is greatly simplified by extending the state splitting technique so that it can be applied to *all* FSTD's with nonnegative, integer adjacency matrices, as we have done above.

D. Constructing the Encoder

We have seen that for the RLL (0, 1) system, we could generate an encoder graph, suitable for encoding at rate 2 : 3, by a basic $\boldsymbol{v}$-consistent state splitting. Can we do this in general? That is, can we always find a sequence of basic $\boldsymbol{v}$-consistent splittings leading to a graph with adjacency matrix T having a "0-1" (T, n)-approximate eigenvector? As indicated earlier, the answer is yes, as we now show.

The proof that we can find basic $\boldsymbol{v}$-consistent splittings is due to [37] in the case of rational capacity, and it is extended to the general case in [22]. The key result we need is stated in Proposition 3.

Proposition 3: Let T be the adjacency matrix of an irreducible FSTD H, and assume that the all-1's vector is not a (T, n)-approximate eigenvector. Let $\boldsymbol{v}$ be a positive (T, n)-approximate eigenvector. Then, there is a basic $\boldsymbol{v}$-consistent splitting of H.

Before giving the proof of the proposition, we describe how we will make use of it in an iterative fashion to construct an encoder FSTD assuming $Cap(s) \geq p/q$.

Let G^q denote the FSTD describing S^q, and assume that there is no "0-1" $(A^q, 2^p)$-approximate eigenvector. Let $\boldsymbol{x}$ be an $(A^q, 2^p)$-approximate eigenvector. If $\boldsymbol{x}$ contains any components equal to 0, we can restrict our attention to the FSTD K determined by the nonzero components.

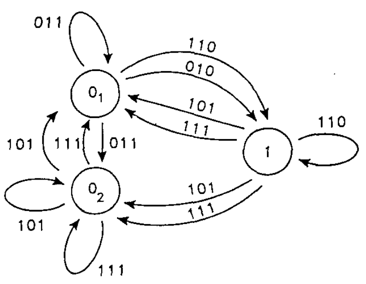


Fig. 13. Example of basic $\boldsymbol{v}$-consistent state splitting.

The vector $\boldsymbol{w}$ obtained by restricting $\boldsymbol{x}$ to the states in K will be an $(A(K), 2^p)$-approximate eigenvector.

If K is irreducible, then we will be in a position to apply the proposition. However, it might happen that K is reducible. In that case, the following argument, which may be skipped on a first reading, shows that we can restrict to an irreducible component H where the proposition is then applicable. We look at the irreducible components of K, and find one, say H, that acts like a "sink," meaning that any edge that originates in H also terminates in H. Such an irreducible "sink" component can always be found in the following manner. Pick an irreducible component and check if it is a sink. If so, stop. If not, there must be a path leading to another irreducible component. If it is a sink, great. If not, continue the procedure. The process must eventually terminate at a sink component H; otherwise, the original decomposition into irreducible components would be contradicted, as the reader can verify.

Since H is an irreducible component of K, and, by assumption, there is no "0-1" $(A(G^q), 2^p)$-approximate eigenvector, it follows that the all-1's vector is not an $(A(H), 2^p)$-approximate eigenvector. Moreover, since H is a sink component, it follows that the restriction $\boldsymbol{v}$ of $\boldsymbol{w}$ to H is an $(A(H), 2^p)$-approximate eigenvector with positive components. The proposition can now be applied with $n = 2^p$ to carry out a $\boldsymbol{v}$-consistent splitting of H, generating an irreducible FSTD H'.

Since $\boldsymbol{v}$-consistent splittings decompose components of $\boldsymbol{v}$ into strictly smaller positive integers, iteration of this state splitting procedure would ultimately produce an FSTD $\hat{H}$ whose adjacency matrix $\hat{T}$ has an all-1's $(\hat{T}, n)$-approximate eigenvector, implying that $\hat{H}$ has outdegree at least $n=2^p$. The number of iterations required to achieve this is no more than $\Sigma_i(v_i - 1)$, since a state i with component v_i will be split into at most v_i descendant states, and this can take no more than $(v_i - 1)$ splitting operations. For the same reason, the number of states in the encoder graph would be at most $\Sigma_i v_i$.

We now proceed with the proof of Proposition 3. The proof requires only elementary concepts, which is why we give full details here. The reader can certainly skip the proof on a first reading, and remain in a position to understand and apply the code construction techniques. However, we want to emphasize here a corollary of the proof that is very important in practice: *Under the con-*

ditions of the proposition, one can always find a splittable state among those states with ***maximum*** *approximate eigenvector component.*

Proof of Proposition 3: Let $v_{\max} \neq 1$ be the maximum of the components of $\boldsymbol{v}$. We will show that there is a state i with the following properties:

1) $v_i = v_{\max}$;

2) $t_{ij} \neq 0$ for some state j with $v_j < v_{\max}$.

We then show that any such state i has a basic $\boldsymbol{v}$-consistent splitting.

Assume that no such state exists. Then, the outgoing edges E_i for every state i with component $v_{\max}$ must terminate only at states with the component $v_{\max}$. Since the FSTD H is assumed to be irreducible, this implies that the approximate eigenvector v is a constant vector, with all of its components equal to $v_{\max}$. If we divide both sides of the approximate eigenvector inequality by $v_{\max}$, we see that this condition implies that the all-1's vector is also an approximate eigenvector. However, this contradicts our assumption about T.

Given the state i satisfying properties 1) and 2), we now construct a $\boldsymbol{v}$-consistent splitting. First, we claim that the number of edges in E_i, denoted $|E_i|$, is at least n. To see this, observe that the approximate eigenvector inequality asserts

$$\sum_k t_{ik} v_k \geq n v_i.$$

Thus, by property 1) above

$$|E_i| v_{\max} \geq \sum_k t_{ik} v_k \geq n v_i = n v_{\max}.$$

Dividing by $v_{\max}$ gives the desired conclusion $|E_i| \geq n$ (actually, with the help of property 2) above one can see that $|E_i| \geq n + 1$, but this is not really needed now).

Let $M = |E_i| \geq n$. Write $E_i = \{e_1, \cdots, e_M\}$, and assume that e_1 terminates in state j, so $w(e_1) < v_{\max}$. Consider the partial accumulated weights

$$W_m = \sum_{k=1}^{m} w(e_k), \qquad m = 1, \cdots, M$$

and their residues modulo n

$$R_m \equiv W_m \ (\text{mod } n), \qquad m = 1, \cdots, M.$$

The *pigeon-hole principle*—which states that if one distributes n pigeons into n pigeon holes, then either every hole has a pigeon, or some hole contains two or more pigeons—implies that the n residues satisfy one of the following conditions:

1) $R_m \equiv 0 \ (\text{mod } n)$, for some $1 \leq m \leq n$;

2) $R_{m_1} \equiv R_{m_2}$ (mod n), for some $1 \leq m_1 < m_2 \leq n$.

In the former case, we define a partition of E_i by setting

$$E_i^1 = \{e_k | k = 1, \cdots, m\}$$

and

$$E_i^2 = E_i - E_i^1.$$

In the latter case, we set

$$E_i^1 = \{e_k | k = m_1 + 1, \cdots, m_2\}$$

and

$$E_i^2 = E_i - E_i^1.$$

In either case, the sum of weights of the edges in E_i^1 is divisible by n,

$$\sum_{e \in E_i^1} w(e) = rn.$$

We claim that

$$1 \leq r < v_{\max}.$$

Clearly $1 \leq r$ since E_i^1 is nonempty. To see that $r < v_{\max}$ observe that in the first case, E_i^1 contains at most n edges and includes e_1 for which $w(e_1) < v_{\max}$; and in the second situation, E_i^1 has strictly less than n edges, each contributing at most $v_{\max}$ to the sum.

Now, we can assert that

$$\begin{aligned}\sum_{e \in E_i^2} w(e) &= \sum_{e \in E_i} w(e) - \sum_{e \in E_i^1} w(e)\\ &\geq v_i n - rn\\ &= (v_i - r)n.\end{aligned}$$

Letting $y_1 = r$ and $y_2 = v_i - r$, we conclude that the partition

$$E_i = E_i^1 \cup E_i^2$$

defines a (nontrivial) basic $\boldsymbol{v}$-consistent splitting.

Q.E.D.

Thus, by an iterative state splitting procedure, the FSTD G^q can be transformed into an FSTD $\hat{H}$ that describes a constraint contained in S^q and has outdegree at least 2^p at each state. Given this FSTD $\hat{H}$, we can easily construct a rate $p:q$ encoder. We first select, for each state i, exactly 2^p edges, discarding any excess beyond this amount. The selection can be completely arbitrary, although in practice, as we indicate in the following sections, judicious selection can simplify the final encoder/decoder implementation.

At each state, we then assign to each of the 2^p edges a unique binary p-block, called an *input tag*, to distinguish it from the labeling already on the graph. This assignment can be completely arbitrary but, again, a careful assignment can yield benefits in the form of reduced complexity in the implementation, as we will see in Section V-B. We loosely refer to this procedure as the *data-to-codeword assignment.*

The encoding procedure is as follows.

1) Choose an initial encoder state i_0.

2) If the current state is i, and the data word is the p-block b, find the edge e in E_i which has the input tag b. The codeword generated is the q-block labeling the edge e. The next encoder state is the state j at which the edge e terminates.

3) Repeat step 2) as long as data words are provided.

Applying this procedure to the RLL (0, 1) encoder graph in Fig. 13 we obtain the encoder graph shown in Fig. 14. The labels are of the form s/t, where s denotes the input tag, and t denotes the codeword.

If we initialize to state 0_1, the data string 00 10 10 11 encodes to the RLL (0, 1) string

011 110 101 111.

E. The State-Splitting Algorithm

We now summarize the steps in the encoder construction procedure.

1) Find a deterministic FSTD G (or, more generally, an FSTD with finite local anticipation) which represents the given constrained system S (most constrained systems have a natural deterministic representation that is used to describe them in the first place).

2) Find the adjacency matrix $A = A(G)$ of G.

3) Compute the capacity $Cap(S)$ as $\log_2$ of the largest eigenvalue $\lambda(A)$ of A.

4) Select a desired code rate $p:q$ satisfying

$$Cap(S) \geq \frac{p}{q}$$

(one usually wants to keep p, q relatively small for complexity reasons).

5) Construct G^q.

6) Using the approximate eigenvector algorithm, find an $(A^q, 2^p)$-approximate eigenvector $\boldsymbol{v}$.

7) Eliminate all states i with $v_i = 0$, and restrict to an irreducible sink component H if necessary.

8) Find a basic $\boldsymbol{v}$-consistent partition for some state in H.

9) Find the basic $\boldsymbol{v}$-consistent state splitting corresponding to this partition, creating FSTD H' and approximate eigenvector $\boldsymbol{v}'$.

10) Iterate steps 8) and 9) until you obtain a graph H with minimum outdegree at least 2^p.

11) At each state of $\hat{H}$, delete all but 2^p outgoing edges and tag these edges with the binary p-blocks, one for each edge.

12) Congratulate yourself with a nice banana "split."

We remark that "generalized $\boldsymbol{v}$-consistent state splittings" (splitting a state into many states) and "rounds of splitting" (splitting several states simultaneously), that will be discussed in Section IV, can be used to shorten the code construction procedure. Also, there is a variable length state splitting approach that produces codes of fixed rate $p:q$, and in many cases this also shortens the code construction procedure. See [38] and [39].

F. State-Dependent Decoders

Having shown how to construct an encoder, it is now time to consider the other half of the code—the decoder. In this section, we show that our encoders always have state-dependent decoders. In order to see this, first recall that we started the code construction procedure with a deterministic FSTD G or one that at least has finite local anticipation. Recall that this latter property is preserved under taking powers. Thus, G^q will also have finite local anticipation (the anticipation would be measured in numbers of q-blocks). Recall from Proposition 1 that state splitting also preserves finite local anticipation, although it may increase the anticipation. Thus, the encoder FSTD $\hat{H}$, obtained from state splitting, will have local anticipation a for some a.

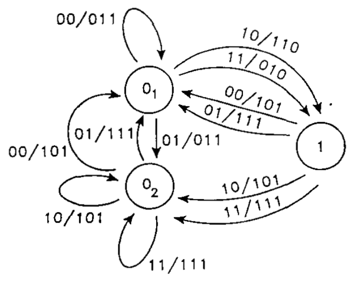


Fig. 14. Encoder graph with input tags.

Now, we can decode as follows.

1) Use the initial state i_0 of the encoder as the initial state of the decoder.

2) If the current state is i, the current codeword to be decoded together with the a upcoming codewords constitute a sequence of length $a + 1$ (measured in q-blocks) that is generated by a path that starts at i; by definition of local anticipation, the initial edge e of such a path is uniquely determined; the data word generated is the input tag of e; the next decoder state is the terminal state of e.

3) Repeat step 2) as long as codewords are provided.

Having now completely described the construction of the encoder and decoder, we have completed the proof of Theorem 1.

As an example, we decode the RLL (0, 1) code string generated above using the encoder in Fig. 14. Starting at state 0_1, the edge determined by the codeword 011, with upcoming codeword 110 is the edge from state 0_1 to state 0_1 with code label 011, so the decoder will generate the input tag 00. Proceeding, the codeword 110 (with upcoming word 101) determines the edge from state 0_1 to state 1 with label 110 and input tag 10. The reader can decode the next codeword 101 in a similar manner, and that is as far as we can go without knowing more upcoming codewords.

IV. Systems of Finite Type and Sliding Block Decoders

As mentioned in the Introduction, the kind of state-dependent decoding discussed in the last section introduces the possibility of catastrophic error propagation when the code is used in a noisy environment. Consider the simple (and, admittedly, artificial) example in Fig. 15.

If we set the initial state to be state 1 and encode the sequence 0000000 $\cdots$, we obtain the constrained sequence *aaaaaaa* $\cdots$. If the first symbol a is corrupted

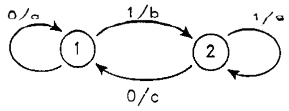


Fig. 15. Encoder susceptible to catastrophic decoder failure.

into the symbol b, then the received sequence will be $baaaaaa \cdots$, which is decoded to the sequence $1111111 \cdots$. Thus, one error caused the decoder to make an unbounded number of errors because the decoder lost track of the correct state.

Thus, it is desirable that the encoder be decodable in a somewhat state-independent manner. We now show that this is always possible for a natural class of constrained systems that includes RLL (d, k) constraints and PRML $(0, G/I)$ constraints.

Recall that a sliding-block decoder, as described in the Introduction, may require some "look-back" and/or "look-ahead" when decoding a codeword, but otherwise requires no state information. More precisely, a finite state encoder with rate $p:q$ has a *sliding-block decoder* if for some nonnegative integers m and a, there is a mapping f from $(m + a + 1)$-blocks in $(\mathfrak{A}(S))^q$ (the alphabet of q-blocks) to $\{0, 1\}^p$ (the alphabet of binary p-blocks)

$$f\colon ((\mathfrak{A}(S))^q)^{m+a+1} \rightarrow \{0, 1\}^p$$

such that, if $\boldsymbol{y} = y_0, y_1, \cdots$, is any sequence of q-blocks generated by the encoder from the input sequence of p-blocks $\boldsymbol{x} = x_0, x_1, \cdots$, then, for $i \geq m$,

$$x_i = f(y_{i-m}, \cdots, y_i, \cdots, y_{i+a}).$$

See Fig. 4 in the Introduction.

We now define a class of constrained systems for which we can construct encoders with sliding-block decoders at rates up to capacity. First we need to introduce some ideas that provide a way to talk precisely about these systems.

An FSTD that represents a constrained system S is said to have *memory* m and *anticipation* a if, given any sequence $\boldsymbol{x} = x_{-m}, \cdots, x_0, \cdots, x_a$ in S, the set of paths $\boldsymbol{e} = e_{-m}, \cdots, e_0, \cdots, e_a$ that generate $\boldsymbol{x}$ all agree in the edge e_0. We say that an FSTD has *finite memory and anticipation* if it has memory m and anticipation a for some m and a. Note the difference between this concept and the concept of finite local anticipation: we have replaced knowledge of an initial state with knowledge of some finite memory. Actually, finite memory and anticipation is a stronger condition, as shown in the following proposition.

Proposition 4: If an FSTD has finite memory and anticipation, then it has finite local anticipation.

Proof: The FSTD has memory m and anticipation a for some m and a. Let i be a state of the FSTD, and let $\gamma = e_0, \cdots, e_a, \gamma' = e_0', \cdots, e_a'$ be paths of length $a + 1$ which start at i and generate the same sequence. We need to show that $e_0 = e_0'$. Well, let η be any path of length m which ends at state i. Then, the concatenated paths $\eta\gamma$ and $\eta\gamma'$ both generate the same sequence. Since the FSTD has memory m and anticipation a, $e_0 = e_0'$ as desired. Q.E.D.

We will see that if a constrained system S is represented by an FSTD with finite memory and anticipation, then we can construct a finite-state encoder with a sliding-block decoder at any rate $p:q$ such that $p/q \leq Cap(S)$. It is, therefore, important to identify the constrained systems that have such a representation.

A constrained system S is *finite-type* if it can be represented by an FSTD with finite memory and anticipation. As an example, the RLL (d, k) constraint is finite-type. The FSTD in Fig. 1 (in the Introduction) has memory k and anticipation 0, i.e., for any given sequence s of length at least $k + 1$ all paths that generate s end with the same edge.

In fact, it can be shown that any finite-type constrained system can be represented by an FSTD that has memory m and anticipation 0 (for some m). This follows from the characterization 1) below (see also the discussion in Section VIII). For this reason, finite-type constrained systems are sometimes called "finite memory" constrained systems. Another sense in which these systems have finite memory is the characterization 3) below. Constrained systems which are not finite-type are sometimes called "infinite memory" systems.

It is important to recognize that there are "bad" representations of finite-type constraints, meaning FSTD's that do not have finite memory and anticipation. For example, the FSTD in Fig. 6(d) represents the RLL (0, 1) constraint, but does not have finite memory and anticipation, as can be seen by considering the paths that generate substrings consisting of all 1's.

Given the existence of bad FSTD's one might begin to worry about potential problems in determining whether or not a constraint is finite-type. In Section VIII, we will see that there is a distinguished representation G_S of any constrained system S such that S is finite type if and only if G_S itself has finite memory and anticipation. For now, we simply point out that there are several intrinsic characterizations of finite-type constraints that are very useful in making this determination. (see also [29].)

1) There is an integer L and a list $\mathfrak{L}$ of words of length L with the property that a sequence $\boldsymbol{x}$ satisfies the specified constraint if and only if each substring of length L in $\boldsymbol{x}$ belongs to the list $\mathfrak{L}$.

For example, for the RLL (0, 1) constraint, we may take $L = 2$ and $\mathfrak{L} = \{11, 01, 10\}$.

2) There is a finite list $\mathfrak{F}$ of forbidden strings; meaning that a sequence $\boldsymbol{x}$ satisfies the constraint if and only if it contains no substrings in the list $\mathfrak{F}$. Note that the forbidden strings need not have the same length.

For example, for RLL (0, 1), we may take $\mathfrak{F} = \{00\}$.

3) Let $\mathfrak{A}$ be the symbol alphabet. There is an integer N such that, for any symbol $a \in \mathfrak{A}$, there is a list of strings of length no more than N, denoted $S(a)$, with the property that, if $\boldsymbol{w}$ is any constrained sequence, then the concatenation $\boldsymbol{w}a$ is a sequence allowed by the constraint if and only if $\boldsymbol{w}$ ends in a string $\boldsymbol{u} \in S(a)$.

For RLL (0, 1), we may take $N = 1$, $S(0) = \{1\}$, and $S(1) = \{0, 1\}$.

Not every constraint of practical interest is finite-type. For example, the system of charge-constrained sequences described by Fig. 9 is not. This can be seen easily by considering condition 3) above: the symbol 1 can be appended to the string

$$-1\ 1\quad -1\ 1\quad -1\ 1\quad \cdots\quad -1\ 1$$

but not to the string

$$1\ 1\quad -1\ 1\quad -1\ 1\quad -1\ 1\ \cdots\ -1\ 1.$$

Now, we improve Theorem 1 for finite-type constrained systems.

Theorem 2: Let S be a finite-type constrained system with Shannon capacity $Cap(S)$. Let p, q be positive integers satisfying the inequality

$$p/q \leq Cap(S).$$

Then, there exists a finite-state encoder, with a sliding-block decoder, that encodes binary data into the constraint S at constant rate $p:q$.

Proof: The proof of the theorem is obtained by applying the finite-state code construction procedure of Section III to any FSTD with finite memory and anticipation that represents S (in particular, such an FSTD has finite local anticipation by Proposition 4 and thus the formula (1) of Section II holds). Just as taking higher powers preserves finite local anticipation, it also preserves finite memory and anticipation. And the reader can verify that just as state splitting preserves finite local anticipation, it also preserves finite memory and anticipation, although the anticipation, but not the memory, may increase.

Thus, the encoder FSTD $\hat{H}$ constructed in Section III, has finite memory ($\hat{m}$) and anticipation ($\hat{a}$). Now, we can decode a q-block b as follows: observe the $\hat{m}$ previous q-blocks and the $\hat{a}$ upcoming q-blocks to determine the unique edge that produced b; then read off the input tag on this edge. This defines a sliding-block decoder with memory $\hat{m}$ and anticipation $\hat{a}$, and this completes the proof of Theorem 2. Q.E.D.

When the code is used in conjunction with a noisy channel, such as a magnetic recording channel, the extent of error propagation is controlled by the size of the decoder window $\hat{m} + \hat{a} + 1$. How large is this window? Well, suppose that we start the code construction procedure with some FSTD G and G^q has memory m and anticipation a (measured in q-blocks).

If x is the number of state splittings used to construct the encoder, then the encoder FSTD $\hat{H}$ has memory m and anticipation no more than $a + x$ (measured in q-blocks). It follows that we can design a sliding-block decoder with decoding window length W satisfying the bound

$$W \leq m + a + x + 1$$

(again, measured in q-blocks). Recall from Section III that an upper bound on the number of state splittings required is

$$x \leq \sum_i (v_i - 1)$$

so

$$W \leq m + a + \sum_i (v_i - 1) + 1.$$

For the RLL (0, 1) encoder in Fig. 14, where $m = 1$, $a = 0$, (refer to Fig. 8) and $v_1 = 2$, $v_2 = 1$, this expression gives an upper bound on the window length of 3 (codewords).

The guarantee of a sliding-block decoder when S is finite-type and the explicit bound on the decoder window length represent key strengths of the state splitting algorithm. In practice, however, this bound on the window length often is larger—sometimes much larger— than the shortest possible. As an example, by examining Fig. 14, one can see that the RLL (0, 1) encoder has a sliding-block decoder with reduced window length $W = 2$, as shown in Table I [2]. (For example, according to the table, the codeword 010 decodes to 11; the codeword 011 decodes to 01 if it is followed by 101 or 111, and it decodes to 00 if it is followed by 010, 011, or 110.)

We next develop the concept of a round of splitting and use it to derive a refined bound on W.

The main transformation in the construction of the finite-state encoder was the basic $\boldsymbol{v}$-consistent state splitting. In that operation, the state i was split into two descendant states. In practice, however, one can combine a sequence of several basic state splittings of state i and its descendants (through several generations) into a single generalized state splitting, as defined in Section III-A, based upon a particular generalized partition of the outgoing edges E_i, as we now describe.

Given a graph H with adjacency matrix T, a positive integer n, a (T, n)-approximate eigenvector $\boldsymbol{v}$, and a state i, a *generalized $\boldsymbol{v}$-consistent partition of E_i* is a partition

$$E_i = E_i^1 \cup E_i^2 \cup \cdots \cup E_i^N$$

with the property that

$$\|E_i^k\| = \sum_{e \in E_i^k} w(e) \geq y_k n, \qquad k = 1, \cdots, N$$

where y_k, for $k = 1, \cdots, N$, are integers

$$y_k \geq 1, \qquad \text{for } k = 1, \cdots, N$$

and

$$\sum_{k=1}^{N} y_k = v_i.$$

The state splitting based upon this partition is called a *generalized $\boldsymbol{v}$-consistent state splitting* of state i. We leave it to the reader to check that this operation can be broken down into a sequence of basic $\boldsymbol{v}$-consistent state splittings.

One can also combine into one transformation several generalized $\boldsymbol{v}$-consistent splittings of different states into a round of splittings, provided that the splitting of each

TABLE I
SLIDING-BLOCK DECODER FOR RLL (0, 1) CODE

Current Codeword	Next Codeword	Decoded Data
010	—	11
011	{101, 111}	01
011	{010, 011, 110}	00
101	{101, 111}	10
101	{010, 011, 110}	00
110	—	10
111	{101, 111}	11
111	{010, 011, 110}	01

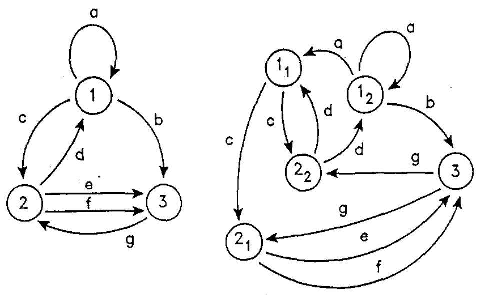


Fig. 16. Independent splittings.

state is independent of the splittings of each of the other states, in a sense which we now make more precise. Suppose that states i and j in H both have a nontrivial generalized $\boldsymbol{v}$-consistent partition of their respective outgoing edges E_i and E_j. Without loss of generality, we first split, say, state i according to its partition, creating an FSTD H' with approximate eigenvector $\boldsymbol{v}'$. The memory remains m and the anticipation increases by at most one to $a + 1$. The partition of E_j in H induces in a natural way a $\boldsymbol{v}'$-consistent partition of the outgoing edges from j in H'; in particular, for any edge e from j to i in H, the descendant edges in H' all lie in one subset of the induced partition of E_j. It follows from this that if we now split state j according to this induced partition, the memory of the presentation by the new FSTD H'' again remains m and the anticipation is still at most $a + 1$. We get the same result if we split state j before we split state i. So, we can view this as a simultaneous splitting of states i and j. It is not difficult to extend this argument to more than two states in H. A sequence of such independent splittings constitutes what we call a *round of splittings*.

Fig. 16 gives an example of a round of splitting in which two states, 1 and 2, are independently split in the pictured FSTD G. An $(A(G), 2)$-approximate eigenvector is $\boldsymbol{v} = [2, 2, 1]^T$. A $\boldsymbol{v}$-consistent splitting for each of states 1 and 2 can be carried out as follows. State 1 splits according to the partition $E_1^1 = \{c\}$ and $E_1^2 = \{a, b\}$. (We denote edges by their labels, since there is no ambiguity in doing so in this example.) Note that the subset E_1^2 has "excess" weight. State 2 splits, independently, according to the partition $E_2^1 = \{e, f\}$ and $E_2^2 = \{d\}$. It is easily checked that the anticipation has increased by only 1, from 0 to 1.

Assume that the construction of the encoder requires r rounds of splittings. Clearly,

$$r \leq x.$$

We can now refine our bound on the sliding-block decoder window length, namely

$$W \leq m + a + r + 1.$$

For the RLL (0, 1) example, in which only 1 basic splitting (and, therefore, only 1 round) was needed to construct the encoder graph, this bound does not improve upon the previous bound $W \leq 3$.

In other cases, however, the refined bound can lead to substantial reduction in the upper bound on W. To illustrate the potential improvement this bound can provide, we look at the rate 2/3 RLL (1, 7) code with a four-state encoder, described in Section VI. The FSTD G^3 has memory $m = 3$ and anticipation $a = 0$. For the code construction, we used the approximate eigenvector

$$\boldsymbol{v} = [2, 3, 3, 3, 2, 2, 2, 1].$$

The number of basic splittings required is $x = 10$, implying the bound

$$W \leq m + a + x + 1 = 14.$$

The actual splitting breaks down into only two rounds, implying

$$W \leq m + a + 3 = 6$$

a substantial reduction.

For both the RLL (0, 1) and RLL (1, 7) codes, however, the refined bounds are not tight. Recall that the RLL (0, 1) encoder that was constructed has a sliding-block decoder with window length $W = 2$. The sliding-block decoder for the RLL (1, 7) code mentioned above can be realized with window length $W = 3$. In the next section, specifically Section V-B, we discuss the methods that were used to achieve these even smaller decoder window lengths.

V. ENCODER/DECODER SIMPLIFICATION TECHNIQUES

A. State Merging

In practice, it is often desirable to design an encoder FSTD with the smallest possible number of states. For a given FSTD, with a (T, n)-approximate eigenvector $\boldsymbol{v}$, we have shown that state splitting can produce an M-state encoder FSTD, where

$$M \leq \sum_i v_i.$$

Often, however, one can reduce this number substantially by means of *state merging*.

Specifically, let i and j be states in an encoder FSTD, with outdegree n (so any edges in excess of n have been deleted). Suppose that their edge sets $E_i = \{e_1, \cdots, e_n\}$

and $E_j = \{f_1, \cdots, f_n\}$ can be arranged so that, for each $k = 1, \cdots, n$, e_k and f_k have the same terminating state and label. Then, it is not difficult to see that we can combine states i and j to derive a new encoder FSTD with one less state. Note that this procedure is precisely the inverse of a state splitting determined by partitioning incoming edges.

As an example, we again consider the RLL (0, 1) constraint. In the encoder FSTD underlying the encoder in Fig. 14, i.e., ignoring input tags, we can see that states 0_2 and 1 can be merged, according to the merging criterion just discussed. The resulting two-state FSTD is shown in Fig. 17.

The sequence of state splittings affects the possible state mergings available in the final FSTD, and, to some extent, it is messy having to deal with the large number of states that may arise. It would be nice if we could identify potential state mergings *during* the state splitting process, using that information to guide the choice of $\boldsymbol{v}$-consistent partitions and splittings, and suggesting ways of reducing the number of states along the way.

Although there is not yet a definitive solution to this problem, there are techniques and heuristics that have proved to be very effective in the construction of codes for recording channels using peak detection as well as channels using partial-response signaling with maximum-likelihood detection (PRML). We now describe some of these techniques which were introduced in [25], [26]. We will illustrate their application in Sections VI and VII.

The key idea is that of ordering the states in an FSTD to reflect the inclusion relations among the corresponding sets of sequences generated from each state. Consider an arbitrary FSTD. A partial ordering $<$ can be imposed on the states of the FSTD by consideration of *follower sets.* For a state i we define the follower set $F(i)$ to be the set of all words of all lengths generated from i. Given two states i and j, we specify that $i < j$ if $F(i) \subseteq F(j)$, where, as usual, $\subseteq$ denotes set inclusion. This partial ordering of sets was used by Freiman and Wyner [40] in the construction of optimal block codes, about which we will have more to say shortly.

We generalize the merging operation illustrated above in terms of the partial ordering. Let G be an FSTD and let i, j be two states. Assume $i < j$; that is, the follower sets $F(i)$ and $F(j)$ satisfy $F(i) \subseteq F(j)$. The (i, j) *merger of* G is the FSTD H obtained from G by the following:

1) eliminating all edges in E_j, the edges emanating from state j;

2) redirecting into state i all remaining edges coming into state j;

3) eliminating the state j.

Fig. 18 shows a schematic representation of an (i, j)-merger.

The following proposition shows how we can reduce the final number of encoder states by merging.

Proposition 5: Let G be an FSTD, with a (T, n)-approximate eigenvector $\boldsymbol{v}$, and let i and j be states in G satisfying:

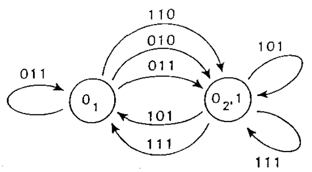


Fig. 17. Merging of RLL (0, 1) encoder graph.

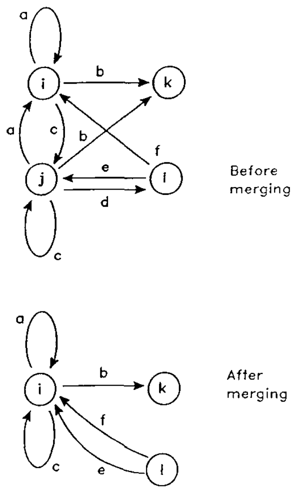


Fig. 18. (i, j)-merger.

a) $i < j$ [that is, $F(i) \subseteq F(j)$], and

b) $v_i = v_j$.

Let H denote the (i, j)-merger of G. Then:

1) the set of sequences generated by H is a subset of the sequences generated by G; and

2) the vector $\boldsymbol{w}$ with $w_k = v_k$ for all vertices k of H is an $(A(H), n)$-approximate eigenvector.

Proof:

1. Let $\boldsymbol{s} = s_1, \cdots, s_k$ be a sequence generated by H. Let $\boldsymbol{e} = e_1, \cdots, e_k$ be a path in H that generates $\boldsymbol{s}$. If $\boldsymbol{e}$ does not pass through the merged state i, then there is clearly a path $\hat{\boldsymbol{e}}$ in G corresponding to $\boldsymbol{e}$ that generates the sequence $\boldsymbol{s}$. Similarly, if $\boldsymbol{e}$ does pass through the state i in H, but does not contain any edge derived from an edge in G ending in state j, one can immediately find a corresponding path $\hat{\boldsymbol{e}}$ in G. Otherwise, let e_l be the last edge of $\boldsymbol{e}$ that terminates at state i in H and comes from an edge $\hat{e}_l$ in G that ends in state j. By hypothesis a), $F(i) \subseteq F(j)$, so there is a path $\hat{e}_{l+1}, \cdots, \hat{e}_k$ in G emanating from state j and generating $s_{l+1}, \cdots, s_k$. If $e_1, \cdots, e_{l-1}$ contains no redirected edges, then $e_1, \cdots, e_{l-1}, \hat{e}_l, \hat{e}_{l+1}, \cdots, \hat{e}_k$ is a path in G generating $\boldsymbol{s}$. If it does, let e_u be the last such edge. Then $e_{u+1}, \cdots, e_{l-1}, \hat{e}_l, \hat{e}_{l+1}, \cdots, \hat{e}_k$ is a path in G that begins at state i and generates $s_{u+1}, \cdots, s_k$. By hypothesis a), there is another path $f_{u+1}, \cdots, f_k$ in G emanating from j that also generates $s_{u+1}, \cdots, s_k$. Continuing in this manner, we eventually produce a path in G that generates the entire sequence $\boldsymbol{s}$.

2) Let T and U be the state-transition matrices for G and H, respectively. Let r be a state in H. By hypothesis b)

$$(Uw)_r = (Tv)_r \geq nv_r = nw_r$$

so w is a (U, n)-approximate eigenvector, as desired Q.E.D.

In a set with partial ordering, there is the possibility of having minimal elements. Recall that the approximate eigenvector v assigns a value v_i, called the *weight of i*, to each state i. In this context, we now introduce the concept of *weight minimal states*.

A state i is *weight minimal* with respect to the partial ordering by follower sets if, for any other state j, the conditions $j < i$ and $v_j = v_i$ imply that j and i are the same state. Proposition 5 shows that, by means of preliminary state merging, code construction by state splitting can be accomplished using only the weight minimal states.

The partial ordering on weight minimal states in the approximate eigenvector component groups also suggests possible state splittings and state mergings that can simplify the final encoder FSTD. Proposition 6 describes a situation in which the "suggestions" from the ordering are guaranteed to be implementable.

Proposition 6: Suppose that the FSTD G has capacity exactly $\log_2(n)$, so that $Tv = nv$, for some nonnegative integer eigenvector v. Let i and j be states in G satisfying $i < j$ and $v_i \leq v_j$. Finally, assume that, if any edges $e \in E_i$ and $f \in E_j$ have the same label, they terminate in the same state.

Then, state j can be split into two states, one of which can be merged with state i. The sum of the components of the induced eigenvector w is reduced by v_i relative to the sum of components in v. Therefore, the upper bound on the final number of encoder states is also reduced by v_i.

Proof: We construct a v-consistent partition $E_j = E_j^1 \cup E_j^2$ as follows. Let E_j^1 be the set of edges emanating from state j that have labels that can also be generated by edges in E_i. Let E_j^2 be the complementary subset $E_j^2 = E_j - E_j^1$. The hypotheses imply that

$$\|E_j^1\| = nv_i$$

and

$$\|E_j^2\| = n(v_j - v_i).$$

Setting $y_1 = v_i$ and $y_2 = v_j - v_i$, we see that the partition defines a basic v-consistent splitting of state j, with descendent states j_1 and j_2, and corresponding eigenvector components y_1 and y_2. Moreover, from the definition of the partition, it is clear that state i can be merged with state j_1, and the sum of the components of the resulting eigenvector is reduced by exactly v_i. This completes the proof. Q.E.D.

We also remark that there are examples that show that this combined splitting/merging cannot always be carried out for an arbitrary choice of approximate eigenvector when the capacity of G is not $\log_2(n)$. However, the examples in Sections VI and VII (as well as the RLL (0, 1) encoder graph that we have constructed) demonstrate that, in many cases, the splitting/merging operations suggested by the partial ordering of the weight minimal states can be implemented, even when the capacity condition in Proposition 6 does not hold. Thus, the merging principle is a valuable heuristic in code design.

The state merging operation is really a special case of a more general method in which states are merged provided that the intersection of their follower sets satisfies certain conditions. The intersection of follower sets also plays an important role in the construction of optimal block codes described by Freiman and Wyner [40]. Their construction can be interpreted nicely in terms of the ideas we have just introduced, as we now describe.

Let G be an FSTD with memory m and anticipation 0, representing a finite-type constraint S. (As mentioned earlier, in Section VIII we will discuss the special FSTD called the *Shannon cover* that is deterministic and, for a finite-type constraint S, provides a representation with memory m and anticipation 0.)

A *block code for length q* is a list of q-blocks in S that are freely concatenable; that is, any concatenation of blocks in the code generates a string in S. A block code is *optimal* if there is no other block code for length q with more codewords in it.

We assume that q is at least as large as the memory m of G. This means that any q-block in S uniquely determines the final state of any path that generates it. For any subset of states τ in $V(G)$, let $\mathcal{L}(\tau)$ denote the block code for length q obtained by taking the intersection of the lists of q-blocks that are generated by the states in τ and that end in some state in τ. To construct an optimal block code for length q we now proceed as follows.

Suppose we wish to determine if there is a block code with at least 2^p words. We first see if there is a "0-1" $(A^q, 2^p)$-approximate eigenvector, by applying the AE algorithm with $L = 1$. If not, then there is no block code for length q with 2^p or more codewords, so we need to reduce p and try again. (This follows from the fact that such a code can be used to define a "0-1" approximate eigenvector: set the components corresponding to the final states of the codewords equal to 1, and set all others to 0.)

If, however, we find such an approximate eigenvector v (which, incidentally, will be the largest "0-1" approximate eigenvector), we look at the set of states τ whose corresponding components in v are 1. We then consider the set of states that are minimal with respect to the partial ordering, restricted to τ, and we call these states τ-minimal states. We then find the common intersection of the lists of q-blocks that are generated by the τ-minimal states and that end in some state in τ. The resulting (possibly empty) collection of words, which is precisely $\mathcal{L}(\tau)$, is a candidate for an optimal block code.

Continuing, we select a τ-minimal state. We set to 0 the component in v corresponding to the minimal state, giving a vector $\boldsymbol{v}'$. Using the modified vector as a starting point, we apply the approximate eigenvector algorithm, and compute a new "0-1" approximate eigenvector $\boldsymbol{w}$. If the algorithm generates a vector $\boldsymbol{w}$ of all-0's, we restore the vector $\boldsymbol{v}$ and repeat the procedure with a different τ-minimal state, if there is one. If the vector $\boldsymbol{w}$ is not all-0's we look at the set of states μ whose corresponding components in $\boldsymbol{w}$ are 1. We then find $\mathcal{L}(\mu)$, by taking the intersection of the lists of q-blocks that are generated from the μ-minimal states and that end in some state in μ.

This process is applied recursively. Each time this process is applied, we either get the all-0's vector or a proper subset of states. Thus, eventually the procedure must terminate. If there is a block code for length q with at least 2^p words, then this process will identify an optimal one. The proof of this statement is given in Appendix B. In the application sections, we use this procedure to construct several optimal block codes of practical interest.

B. Sliding Block Decoder Window

Once the data-to-codeword assignment on the encoder FSTD is selected and the encoding finite-state machine is completely specified, the decoder mapping is effectively fixed.

If the decoder mapping can, in fact, be represented in sliding-block form, one can, in principle, calculate the minimum decoder window length required to correctly decode valid code sequences. This length may depend on the specific choice of input tags, however, as can be seen from the example in Fig. 19(a) and (b), based upon the rate 1/2, RLL (1, 3) code described in the next section. In Fig. 19(a), the decoder window is 1 codeword, but in Fig. 19(b), the minimum decoder window is 2 codewords (1 codeword look-back).

The general question we wish to answer is: assuming that a sliding-block decoder is possible, what choice(s) of data-to-codeword assignment will minimize the decoder window length W?

If the encoder FSTD has finite memory ($\hat{m}$) and anticipation ($\hat{a}$), as is the case when the encoder results from the application of the state splitting algorithm to a finite-type constraint, then the encoder finite-state machine produced by *any* data-to-codeword assignment has a sliding-block decoder with window length W satisfying the bound

$$W \leq \hat{m} + \hat{a} + 1.$$

This bound is essentially the same as the bound

$$W \leq m + a + r + 1$$

derived in Section IV for an encoder FSTD obtained by the application of r rounds of state splitting to an initial FSTD with memory m and anticipation a.

It is quite possible, however, that these upper bounds will not be tight and the window size can be reduced. A nice illustration of decoder window reduction, relative to these bounds, can be found in the construction of the rate 1/2, (2, 7) code in [22]. In that case, the nominal memory of the encoder FSTD (arising from the (2, 7) constraint memory) was $\hat{m} = 4$ codewords and the anticipation (corresponding to the three rounds of state splitting used to construct the encoder FSTD) was $\hat{a} = 3$ codewords. However, a data-to-codeword assignment was demonstrated that eliminates all look-back, reducing the sliding-block decoder window to only 4 codewords, far less than the 8 codewords suggested by the bounds mentioned above. This reduction in decoder window length was achieved by making consistent assignments of input tags to edges that have the same labels, but that are not completely distinguished by the possible contents of the reduced decoder window.

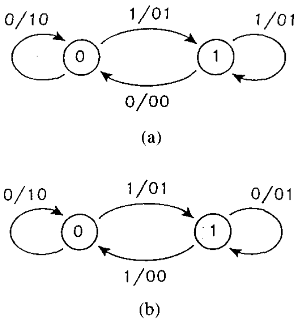


Fig. 19. (a) One choice of input tags. (b) Another choice of input tags.

For the RLL (0, 1) code, it is precisely this type of consistent input tag assignment that enabled the reduction of the sliding-block decoder window length to $W = 2$, eliminating the need for 1 codeword look-back that would be expected from the memory $\hat{m} = 1$ of the encoder graph derived from Fig. 13. This can be verified by referring to the RLL (0, 1) encoder shown in Fig. 14. There, we can see that the edges with label 101 (respectively, 111), emanating from states 1 and 0_2 and ending in state 0_1, have the same input tag, namely 00 (respectively, 01). Similarly, the edges with label 101 (respectively, 111), running from states 1 and 0_2 to state 0_2, are assigned the same input tag, 10 (respectively, 11).

We now elaborate upon this idea of "consistent" assignment and describe an iterative, albeit brute-force, approach to defining a data-to-codeword assignment that allows the minimum possible decoder window to be achieved. The approach is built upon the following procedure, which, for a specified m' and a', determines if a decoder with look-back m' and look-ahead a' is feasible for any data-to-codeword assignment.

Select targets for the look-back m' and look-ahead a' for the decoder. For each edge e in the encoder FSTD, one can list the codewords generated by paths $e_{-m'}, \cdots, e_{-1}, e, e_1, \cdots, e_{a'}$. Denote this list by $L(e; m', a')$. Clearly, in order for a sliding-block decoder with look-back m' and look-ahead a' to exist, any two edges e and f must be assigned the same input tag whenever the corresponding lists are not disjoint

$$L(e; m', a') \cap L(f; m', a') \neq \phi.$$

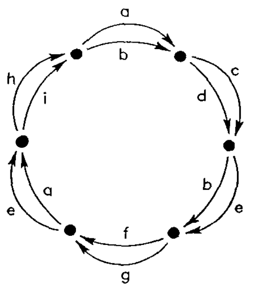


Fig. 20. Encoder FSTD requiring decoder window of length at least 2.

Of course, there is another requirement the assignment must satisfy: the set of data words assigned to the 2^p edges at each state must comprise the entire list of binary p-blocks. If inconsistencies arise in the labeling and this condition cannot be met, then the proposed decoder window parameters m' and a' are not achievable.

The execution of this procedure, although conceptually very straightforward, has been shown to be equivalent to a graph-coloring problem that is NP-complete [41]. From a practical standpoint, this means that the optimization of the decoder window length, through iterative application of the procedure for varying values of m' and a', may require either some good hunches or a good computer program, or both.

It is also worth pointing out that these techniques are not restricted to encoder FSTD's describing finite-type constraints. They have been successfully applied in the design of sliding-block decoders for codes into constrained systems that are not finite-memory. See, for example, the discussion in [42] of a 100% efficient sliding-block code for the charge-constrained, run-length limited constraint with parameters $(d, k; c) = (1, 3; 3)$, also mentioned in Section VIII.

In Fig. 20, we show an FSTD for which it is impossible to assign 1 bit input tags in such a way that the sliding-block decoder will have window length equal to one codeword (that is, no look-back or look-ahead). Using the procedure just outlined, the reader may enjoy verifying this fact, as well as finding an input tag assignment that allows a decoder window of length 2 codewords.

To conclude this section, we remark that, ideally, the objective of the code designer is to simultaneously optimize the number of states in the encoder FSTD and the length of the decoder window, resulting in lowest complexity and error propagation. As mentioned earlier, this problem of optimal code design is extremely difficult, in general, although progress continues to be made. It should be stressed, however, that an encoder FSTD that has the minimal number of states may not necessarily be the same as an encoder FSTD that, with the appropriate input tag assignment, minimizes the sliding-block decoder window.

VI. Applications to RLL Codes

In this section, we apply the state splitting algorithm to construct several RLL codes. We first provide an overview of the recording channel. Another discussion of RLL codes in the context of digital data recording can be found in [2].

A. Background on Peak Detection Recording Channels

Digital magnetic recording systems are communications channels. They have input signals and output signals, the output signal being a transformed and noisy version of the input signal. To quote from a paper by Berlekamp [43] "Communication links transmit information from here to there. Computer memories transmit information from now to then."

For all practical digital magnetic recording systems, saturation recording is utilized whereby the magnetic medium is magnetized in one of two (opposite) directions. The input signal (i.e., the write current) is then simply a two-level signal that takes on the values $+A$ and $-A$. The positions of the transitions in this input signal (from $+A$ to $-A$ or vice versa) carry the digital information. Here we assume that the input waveform is such that transitions can only occur at integer multiples of a fixed time duration T_b called a channel bit period.

At the densities of present day systems, a linear model [44], [45] gives a reasonably accurate description of the input–output signal characteristics of the channel. That is, if a single positive-going transition (from $-A$ to $+A$) which occurs at time $t = 0$ produces the output $2Ag(t)$, then, in the absence of noise and assuming a two-level input, the channel acts as a linear channel with impulse response $h(t) = g'(t)$, where the prime denotes differentiation with respect to time. The most commonly assumed form for $g(t)$ is the so-called Lorentzian isolated step response given as $g(t) = 1/(1 + (2t/\tau)^2)$. The constant τ is the pulse width of the isolated transition response when its amplitude is decreased by a factor of 50% from its peak value.

The design of the modulation, coding, and signal processing in magnetic recording products has been driven by the detector chosen to detect the transitions in the input waveform. This detector, called a peak detector [1], has the advantage of being extremely simple. However, by its very nature, it can only perform reliably at low densities.

The peak detector works as follows. It looks for peaks in the output waveform whose magnitude exceeds some predetermined threshold. Each such peak is thought to be due to a transition in the input waveform. A device called a phase lock loop (PLL) is used to derive timing. That is, the PLL produces a clock of period T_b seconds by which to identify channel bit cells. Then, if an output pulse is located in a bit cell, that bit cell is said to contain a transition. The PLL, in turn, is driven by the peak detector and the clock generated by the PLL is adjusted so that, on average, the peaks occur in the centers of the bit cells.

If one assumes a linear channel and examines the waveform produced by the linear superposition of two Lorentzian pulses (of opposite sign) separated by αPW50 seconds, one finds that this waveform will contain two peaks separated by βPW50 seconds where

$$\beta = \sqrt{\frac{\alpha^2 - 1 + 2\sqrt{(\alpha^4 + \alpha^2 + 1)}}{3}}.$$

If α is much greater than 1, β is approximately equal to α, but as α approaches zero, β approaches the fixed value $\sqrt{3}/3 \approx 0.5774$. Peaks will be centered in their bit cell only at low densities (values of α in excess of 1).

To make the system work better at moderate information densities (say $\alpha = 0.5$), several techniques are used. One such technique is to use a linear equalizer at the output of the channel which modifies the transition response of the system before the peak detector. This equalizer is often called a pulse slimmer [46] since its purpose is to create a channel whose isolated transition response is a thinner pulse than that produced by the unequalized channel. Many different equalization techniques are utilized but, in all of them, the benefits derived from slimming the pulses must be weighed against the possible penalty in signal-to-noise ratio arising from the effect of the equalizer on the noise.

Another technique for allowing a peak detector to perform reliably at moderate densities makes use of run-length limited constraints. These constrained sequences result in a two-level input waveform for which the minimum and maximum time interval between transitions are fixed. In particular, for a given choice of the parameters d and k, the input waveform is constrained so that the minimum time between transitions is $(d + 1)T_b$ and the maximum time between transitions is $(k + 1)T_b$. For no constraint on the input waveform, d is zero and k is infinite.

The reason for choosing the parameter k to be finite is to ensure the proper operation of the PLL timing recovery circuit. Since the maximum time interval between input transitions is constrained, so also is the maximum time interval between peaks in the output waveform. Therefore, the PLL can only proceed for so long without an output peak occurring to correct its estimate of the timing.

The reason for choosing the parameter d to be greater than zero is less obvious (and thus more interesting). Assume that $T_{\min}$ is the smallest time interval that can be allowed between neighboring transitions in the two-level input waveform. If d is equal to zero, then we must choose $T_b = T_{\min}$. For this choice of T_b, the maximum information rate that can be supported by the two-level input waveform would be $(T_b)^{-1} = (T_{\min})^{-1}$ bits/second. Now assume that d is chosen as an integer strictly greater than zero. Since the minimum time interval between transitions is now $(d + 1)T_b$, $T_{\min}$ can be chosen equal to $(d + 1)T_b$ and the constrained binary digits occur at a rate of $(T_b)^{-1} = (d + 1)(T_{\min})^{-1}$ binary digits/seconds. For a fixed value of $T_{\min}$, this corresponds to an increase in symbol rate by a factor of $(d + 1)$. Unfortunately, this increase is not in the true information rate but in the constrained binary symbol rate since more than one constrained binary digit is required to represent one information bit. Let R be the rate of a given finite-state (d, k) code. Then the true information rate for this system using the (d, k) code is $R(T_b)^{-1} = R(d + 1)(T_{\min})^{-1}$ bits/second. The product $R(d + 1)$, called the density ratio [21] of the code, represents the increase (if $R(d + 1) > 1$) or decrease (if $R(d + 1) < 1$) in true information rate using a (d, k) code as compared to an unconstrained or uncoded system. We are particularly interested in systems where the density ratio is strictly greater than 1 for then we are storing information at a higher rate than the highest possible rate of the transitions in the two-level input waveform.

We assume that every T_b seconds there is either a transition or nontransition in the input waveform. We represent this sequence of transitions and nontransitions by a sequence of ones and zeros, respectively. The constraint on the minimum and maximum time interval between transitions described above then translates to the following constraint on the binary sequence: successive ones in the binary stream are separated by at least d zeros and at most k zeros. A deterministic FSTD representing the (d, k) constraint, for k finite was shown in the Introduction (Fig. 1). In the literature, the (d, k) constraint is often referred to as the *maxentropic* (d, k) *code*.

The Shannon capacity of the (d, k) constraint, denoted $C(d, k)$, is given as

$$C(d, k) = \log_2 \lambda$$

where λ is the largest real root of the equation

$$x^{k+2} - x^{k+1} - x^{k+1-d} + 1 = 0.$$

(The polynomial on the left-hand side of the equation is $(x - 1)$ times the characteristic polynomial of the adjacency matrix of the standard (d, k) FSTD.) Furthermore, it can be shown that for k infinite, $C(d, \infty) = C(d - 1, 2d - 1)$ [47].

The state splitting algorithm described in Sections III and IV can be utilized to find finite-state encoders with sliding-block decoders at any rate $p : q$ where $p/q \le C(d, k)$.

Some common (d, k) code parameters and practical rates p/q are listed in Table II.

In the remainder of this section we will use the methods described in the previous sections to derive several codes that have played important roles in magnetic recording: a rate 4/5, (0, 2) block code; a rate 8/9, (0, 3) block code; a rate 1/2, (1, 3) code; and two distinct rate 2/3, (1, 7) codes.

B. Block Codes for (0, 2) and (0, 3) Constraints

We begin the design of specific codes with the first two entries in this table, rate 4/5 (0, 2) and rate 8/9 (0, 3). These codes were introduced in IBM 3420 and 3480 tape drives, and became industry standards [28, pp. 120–122]. Both of the codes are block codes; that is, codes for which a single codebook serves as the encoder and decoder, so there is, in effect, only one state. We will use the methods of Section III, which reduce to the methods of Freiman

TABLE II
PARAMETERS OF SOME COMMON CODES

d	k	$C(d, k)$	p	q	p/q
0	2	0.8792	4	5	0.8
0	3	0.9468	8	9	0.8889
1	3	0.5515	1	2	0.5
1	7	0.6793	2	3	0.6667
2	7	0.5174	1	2	0.5

and Wyner [40], to derive optimal block codes that contain the two block codes of interest.

The codebook for the rate 4/5 (0, 2) code (the so-called GCR code) is given in Table III.

To verify that the channel sequences satisfy the (0, 2) constraint, note that no codeword by itself violates the (0, 2) constraint nor does the concatenation of any two codewords.

The methods described earlier can be applied to generate this code, as follows. We begin with the natural FSTD for the $(d, k) = (0, 2)$ constraint as shown in Fig. 21.

The follower sets satisfy the inclusion relations

$$F(2) \subseteq F(1) \subseteq F(0)$$

so the partial ordering of states is given by

$$2 < 1 < 0.$$

The adjacency matrix for this FSTD is

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}.$$

We are interested in a rate 4/5 code so we calculate A^5 to be

$$A^5 = \begin{bmatrix} 13 & 7 & 4 \\ 11 & 6 & 3 \\ 7 & 4 & 2 \end{bmatrix}.$$

It is easy to see that the vector

$$v = [1, 1, 0]^T$$

is the $(A^5, 2^4)$-approximate eigenvector found by the AE algorithm when we set $L = 1$. Using the terminology of Section V-A, state 1 is τ-minimal, where $\tau = \{0, 1\}$. There are exactly 17 words of length 5 generated from state 1, ending in state 0 or state 1. This list contains the 16 codewords in the GCR code in Table III, as well as the word 11111.

We can now apply the procedure described in Section V-A to prove that this list of 17 words is an optimal block code for length 5. Recalling that state 1 is $\{0, 1\}$-minimal, we set to 0 the corresponding component, and apply the AE algorithm. The resulting vector is all-0's. It follows that the list of 17 words is optimal (and unique). Note also that the list is invariant under reversal of the symbol order in each word.

TABLE III
ENCODING/DECODING TABLE FOR (0, 2) CODE

User Bits	Code Bits
0000	11001
0001	11011
0010	10010
0011	10011
0100	11101
0101	10101
0110	10110
0111	10111
1000	11010
1001	01001
1010	01010
1011	01011
1100	11110
1101	01101
1110	01110
1111	01111

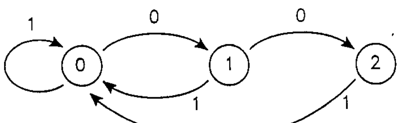

$$\begin{tikzcd} 0 \arrow[loop left, "1"] \arrow[r, bend left, "0"] & 1 \arrow[l, bend left, "1"] \arrow[r, bend left, "0"] & 2 \arrow[ll, bend left, "1"] \end{tikzcd}$$

Fig. 21. FSTD for $(d, k) = (0, 2)$ constraint.

The selection of 16 words for a rate 4:5 block code, and the mapping between the user bits and the codewords are arbitrary. However, the codewords and the mapping can be chosen for various system-related reasons; for example, to minimize the hardware for implementing this codebook.

The construction of an optimal block code with length 9 for the $(d, k) = (0, 3)$ constraint proceeds in a similar fashion, so we will only sketch the code design. An optimal block code contains 293 words, so a rate 8/9 (0, 3) code is supported.

An FSTD for the $(d, k) = (0, 3)$ constraint is shown in Fig. 22.

The adjacency matrix for this graph is

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

The 9th power of this adjacency matrix is

$$A^9 = \begin{bmatrix} 208 & 108 & 56 & 29 \\ 193 & 100 & 52 & 27 \\ 164 & 85 & 44 & 23 \\ 108 & 56 & 29 & 15 \end{bmatrix}.$$

The two $(A^9, 2^8)$-approximate eigenvectors $v = [1, 1, 1, 0]^T$ and $v = [1, 1, 0, 0]^T$ both describe sets of states in which there is a single minimal state generating 293

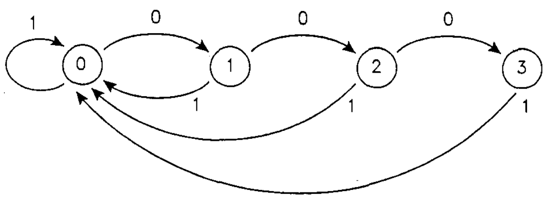


Fig. 22. FSTD for $(d, k) = (0, 3)$ constraint.

words. The two lists constitute the two optimal block codes with length 9, and we note that if the symbol order is reversed in the words in one list, the words in the other list are generated.

Remark: The $(d, k) = (0, 2)$ and $(d, k) = (0, 3)$ codes illustrate a more general result about $(0, k)$ codes [40]. For $k = 2l$, there is a unique optimal block code for length q, where $q \geq k$. The code consists of the q-blocks generated from state l and terminating in states $\{0, 1, \cdots, l\}$. The code is reversal-symmetric.

For $k = 2l + 1$, there are two optimal block codes for length q, where $q \geq k$. The first code contains the q-blocks generated from state $l + 1$ and terminating in states $\{0, 1, \cdots, l + 1\}$. The second contains the q-blocks generated from state l and terminating in states $\{0, 1, \cdots, l\}$. Reversing the symbol order in the codewords of one of these codes yields the codewords of the other.

C. $(d, k) = (1, 3)$

In this section we derive a rate $1/2$ (1, 3) code, known variously as Miller code, modified frequency modulation (MFM) code, and delay modulation [28, p. 69]. It has found wide use in the data storage industry.

We begin with the finite-state transition diagram (FSTD) G for (1, 3) constrained binary sequences shown in Fig. 23.

The capacity of this constraint is $C \approx 0.5515$. We desire a code of rate $R = 1/2$. The second power of the FSTD of Fig. 23 is shown in Fig. 24.

Note that state 3 in this graph is deficient in that it has only one edge emanating from it. However, $v = [1, 1, 1, 0]^T$ is an $(A(G)^2, 2)$-approximate eigenvector found using the AE algorithm with $L = 1$. The sub-FSTD H determined by this vector, shown in Fig. 25, satisfies the rowsum condition needed to support a rate 1 : 2 code.

Moreover, since states 1 and 2 have identical follower sets, we construct the (1, 2)-merger according to Section V-A, as shown in Fig. 26.

Finally, we obtain a rate 1 : 2 encoder, shown in Fig. 27, where, as before, the edge labels are of the form s/t where s is the (1 bit) input tag and t is the pair of (1, 3)-constrained binary symbols produced by the encoder.

From the encoder, one can see that this code has an interesting and uncommon property: it is a systematic code. That is, each time an information bit is encoded by the encoder, this bit appears as the second code symbol produced by the encoder. The rule by which the encoder

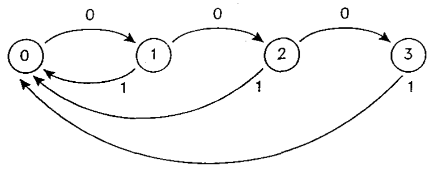


Fig. 23. FSTD for $(d, k) = (1, 3)$ constraint.

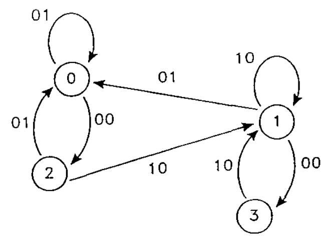


Fig. 24. Second power of FSTD in Fig. 23.

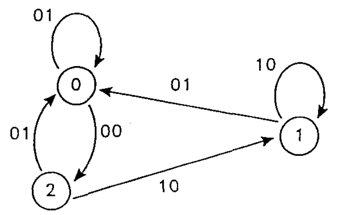


Fig. 25. RLL (1, 3) encoder graph.

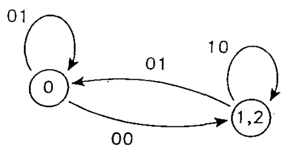


Fig. 26. Merger of RLL (1, 3) encoder graph.

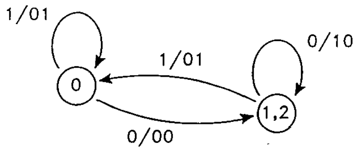


Fig. 27. RLL (1, 3) encoder.

produces the first code symbol can be described easily in words: the first symbol is a 0, except when it appears between two information bits which are 0's, in which case it is a 1.

We remark that the two optimal two-block codes for the (1, 3) constraint, namely $\{01\}$ and $\{10\}$, contain only one word each. Therefore, there is no rate 1 : 2, (1, 3) block code.

D. $(d, k) = (1, 7)$

Rate $2/3$ (1, 7) codes are used today in many digital recording devices. Several codes with these parameters have been derived (see, for example, [19], [48], [49]). The finite-state encoders for two of the more popular codes, developed independently by Jacoby [18], [19] and Adler *et al.* [48], respectively, were shown in [2] to have identical, underlying five-state encoder FSTD's. The only difference between the codes was the choice of data-to-codeword assignments.

The construction in [18] does not use state splitting methods. Rather, the encoding makes use of a basic 2:3 encoding table, with a substitution table for violations of the $d = 1$ constraint, as anticipated by means of one word look-ahead. The construction in [48] uses state splitting applied to a five-state sub-FSTD of the third power of the edge graph of the usual eight-state (1, 7) FSTD. This five-state FSTD represents a constrained system that has capacity exactly 2, and was found by somewhat *ad hoc* methods by examining concatenations of binary 3-tuples that do not violate the $d = 1$ or $k = 7$ constraints. The state merging methods described in Section V-A can be immediately applied to reduce the five-state FSTD to a three-state FSTD (which is not an encoder FSTD) shown in Fig. 28, that describes exactly the same constrained system.

Recently, state splitting was used to design a different rate 2/3, (1, 7) code with a four-state encoder FSTD [49], the minimum number of encoder states possible [32]. The construction applied the state splitting algorithm to the three-state FSTD shown in Fig. 29.

The sequences described by this FSTD can be interpreted as a "phase-shifted" version of the sequences produced by the FSTD in Fig. 28, as discussed in Howell [50]. More precisely, if the sequences from Fig. 28 are written with the time index

$$\cdots x_0x_1x_2 \cdot x_3x_4x_5 \cdot x_6 \cdots$$

then the sequences from Fig. 29 are

$$\cdots x_0 \cdot x_1x_2x_3 \cdot x_4x_5x_6 \cdot \cdots .$$

The main purpose of this subsection is to show that both of these codes can be constructed "from scratch," so to speak, using the techniques discussed in Sections III–V. No inspired observations or *a priori* knowledge of previous rate 2:3 codes is necessary. What is interesting is that we obtain the two codes by using two distinct approximate eigenvectors produced by the AE algorithm, when the maximum component limits are set to $L = 3$ and $L = 5$.

The construction of the four-state encoder FSTD proceeds as follows. The third power of the RLL (1, 7) FSTD is shown in Fig. 30.

In the partial ordering based upon follower sets, we find that

$$7 < 6 < 5 < 4 < 3 < 2 < 1$$

while 0 is not comparable to any of the other states; that is, there is no inclusion relationship involving $F(0)$ and any of the other follower sets.

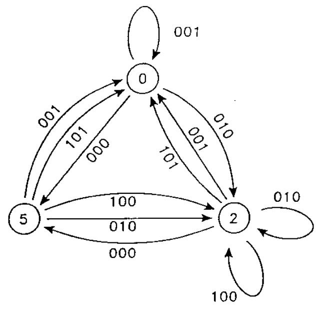


Fig. 28. A three-state FSTD for (1, 7) code sequences.

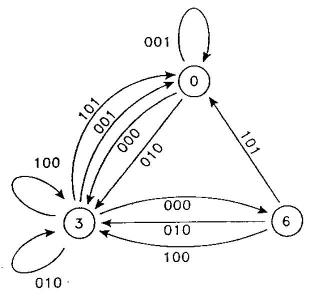


Fig. 29. A "phase-shifted" version of FSTD in Fig. 28.

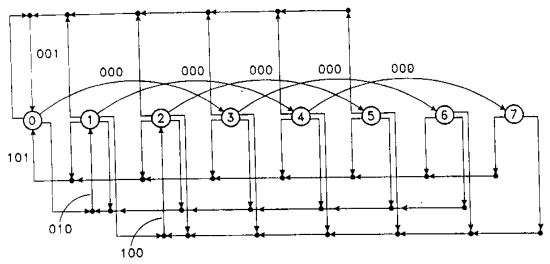


Fig. 30. Third power of the RLL (1, 7) FSTD.

The first $(A(G^3), 4)$-approximate eigenvector found by the AE algorithm, with $L = 3$, is

$$v = [2, 3, 3, 3, 2, 2, 2, 1]^T.$$

The weight-minimal states are 0, 3, 6, and 7. Applying Proposition 5, we can apply mergings and reduce the FSTD G^3 to a four-state FSTD, shown in Fig. 31, with induced approximate eigenvector

$$w = [2, 3, 2, 1]^T.$$

This FSTD is reducible, however, since state 7 has no incoming edges. We, therefore, reduce to an irreducible component that is a "sink" component (there is only one, here, derived from states 0, 3, and 6). The resulting FSTD is precisely the three-state diagram in Fig. 29 from which the four-state encoder FSTD can now be derived, as described in [49]. The derivation requires two rounds of splitting. The sliding-block decoder window requires three words: the current word and two words of look-ahead

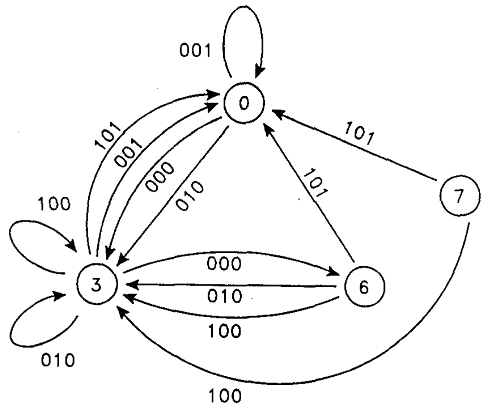


Fig. 31. A merger of the FSTD in Fig. 30.

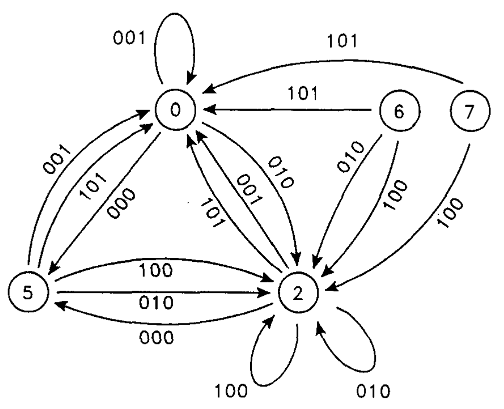


Fig. 32. A different merger of the FSTD in Fig. 30.

(from the two rounds of splitting). The data-to-codeword assignment was chosen to eliminate the need for any look-back in the decoder. Consequently, the propagation of errors can be no more than 6 user bits, but one can check that, in fact, it never exceeds 5 bits, again thanks to the judicious choice of data-to-codeword assignment.

The derivation of the five-state encoder FSTD is quite similar. As mentioned above, we use the first $(A(G^3), 4)$-approximate eigenvector found by the AE algorithm, with $L = 5$, namely

$$v = [3, 5, 5, 4, 4, 4, 3, 2]^T.$$

The weight-minimal states are now 0, 2, 5, 6, and 7. Applying Proposition 5, we reduce the FSTD G^3 to a five-state FSTD, shown in Fig. 32, with induced approximate eigenvector

$$w = [3, 5, 4, 3, 2]^T.$$

We find that this FSTD is also reducible. We, therefore, find an irreducible component that is a "sink" component (again, there is only one, derived from states 0, 2, and 5). The resulting FSTD is precisely the three-state diagram in Fig. 28 from which the five-state encoder FSTD can now be derived, as described in [48].

E. $(d, k, s) = (2, 18, 2)$

The previous sections demonstrated how the state splitting algorithm and related code design methods can be used to derive a variety of RLL codes, including several codes originally found by interesting, but sometimes *ad hoc* techniques.

The usefulness of the state splitting approach becomes clearest, however, when one attempts to design codes for more complicated constrained systems. As an example, we mention the multiple-spaced RLL constraint with parameters $(d, k, s) = (2, 18, 2)$. The parameter s indicates that the run-lengths of 0's must be of the form $d + is$, where i is a nonnegative integer. For $d = s = 2$, this implies that the 1's are separated by an even number of 0's. These constraints were originally investigated by

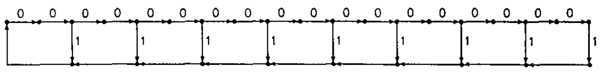


Fig. 33. $(d, k, s) = (2, 18, 2)$ FSTD.

Funk [51]. In the context of magnetic recording, he showed that multiple-spaced RLL codes with $s = 2$ might have some practical value. More recently, and independently, $(d, k, 2)$ constraints were shown to play a natural role in magnetooptic recording systems using a resonant-bias coil direct-overwrite technique [52], [53].

Weigandt [54] applied state splitting techniques to design a rate 2/5 (2, 18, 2) code. It is not unfair to say that the design of such a code would have been close to impossible without the use of state splitting, as well as a computer software package to help with the bookkeeping. Details of the encoder and decoder can be found in [54]. Our aim in this subsection is simply to give the reader an indication of the scope of the design problem and the measure of success achieved using state splitting.

An FSTD G for the $(d, k, s) = (2, 18, 2)$ constraint is shown in Fig. 33.

The Shannon capacity of this system is $C = 0.40403 \cdots$. In order to design a rate 2:5 code, the AE algorithm was used to find an $(A(G^5), 4)$-approximate eigenvector:

$$v = [7, 9, 12, 9, 12, 9, 12, 9, 11, 8, 11, 8, 11, 7, 10, 6, 8, 4, 5]^T.$$

A sequence of state splittings requiring only four rounds of splitting was found. The resulting encoder FSTD had 25 states. (Note that the approximate eigenvector established an upper bound of 168 states!) The sliding-block decoder window spanned only four codewords, after a judicious selection of a data-to-codeword assignment.

VII. Applications to PRML $(0, G/I)$ Codes

A. Code Constraints for Partial Response Channels

Partial-response signaling with maximum-likelihood sequence estimation (which we will denote by PRML) is a technique, using amplitude detection, which has been investigated for high-density magnetic recording of digital data. The technique combines partial-response class IV signaling with maximum likelihood detection in the form of a Viterbi decoder. For more details, see, for example, [4]–[7]. There is a need for constrained codes in this context: to improve timing and gain control, as well as to limit the path memory limitations on samples with value "0" in the channel output signal. When Interleaved NRZI recording [INREI] [4], [25], [26] is used, constraints on the runs of samples with value "0" in the channel output sequence correspond directly to similar constraints on the runs of symbols "0" at the channel input [55]. In the PRML context, these constraints are described naturally by two parameters $\boldsymbol{G}$ and $\boldsymbol{I}$, where $\boldsymbol{G}$ is the maximum number of adjacent symbols "0" allowed in the code string, and $\boldsymbol{I}$ is the maximum number of adjacent symbols "0" allowed in both the even and odd substrings which interleave to make up the codestring. The $\boldsymbol{G}$ constraint helps to improve timing and gain control. The $\boldsymbol{I}$ constraint is used to limit the path memory of the Viterbi decoder: decoding is accomplished by deinterleaving the received string and applying two Viterbi decoders based on $1 - D$ trellises—one for the substring determined by even coordinates and the other for the substring determined by odd coordinates. To help distinguish the PRML $(0, \boldsymbol{G}/\boldsymbol{I})$ constraints from the (d, k) RLL constraints, we will use the notation $(0, \boldsymbol{G}/\boldsymbol{I})$. The first parameter can be thought of as a d constraint which, for our purposes, is always set to $d = 0$. The $\boldsymbol{G}$ and $\boldsymbol{I}$ parameters resemble the k constraint in RLL codes, governing maximum run-lengths of 0's.

We can represent $(0, \boldsymbol{G}/\boldsymbol{I})$ constraints by diagrams based on states which reflect the three relevant quantities, the number g of symbols "0" since the last symbol "1" in the global string and the numbers a and b which denote the number of symbols "0" since the last symbol "1" in the two interleaved substrings. Note that g is a function of a and b, denoted $g(a, b)$

$$g(a, b) = \begin{cases} 2a + 1 & \text{if } a < b \\ 2b & \text{if } a \geq b \end{cases}.$$

We label the states with 2-tuples (a, b), where a is the number of symbols "0" in the interleaved substring containing the next to last bit, and b is the number in the substring containing the last bit. In the (a, b) notation, the set of states V for a $(0, \boldsymbol{G}/\boldsymbol{I})$ constraint is given by

$$V = \{(a, b)\colon \ 0 \leq a, b \leq \boldsymbol{I} \ \text{ and } \ g(a, b) \leq \boldsymbol{G}\}$$

and the transitions between states are given by the rules

$$\text{“0”: } (a, b) \rightarrow (b, a + 1), \ \text{ provided } (b, a + 1) \in V$$

$$\text{“1”: } (a, b) \rightarrow (b, 0).$$

The reader may verify that this FSTD has finite memory and anticipation, and, therefore, the PRML $(0, \boldsymbol{G}/\boldsymbol{I})$ constraints are finite-type.

The partial ordering of states for the $(0, \boldsymbol{G}/\boldsymbol{I})$ constraints is expressed in a particularly simple, geometric way when the (a, b) nomenclature based on substring runs is used. Let $s_1 = (a_1, b_1)$ and $s_2 = (a_2, b_2)$. Then, $s_1 < s_2$ if either

$$a_1 \geq a_2 \ \text{ and } \ b_1 \geq b_2$$

or

$$b_1 \geq b_2 \ \text{ and } \ g(a_1, b_1) = \boldsymbol{G}.$$

The ordering is interpreted geometrically by placing the states on the integer lattice in the plane. Each state (a, b) with $g(a, b) < \boldsymbol{G}$ is placed at the grid point with coordinates (a, b), while states with $g(a, b) = \boldsymbol{G}$ are placed at grid point $(\boldsymbol{I}, b)$. The ordering can then be described in words by the simple rule

$$s_1 < s_2 \quad \text{if } s_2 \text{ is below and to the left of } s_1.$$

From the appropriate adjacency matrix T, we can determine the feasible rates for $(0, \boldsymbol{G}/\boldsymbol{I})$ codes. For example, for the $(0, \boldsymbol{G}/\boldsymbol{I}) = (0, 3/3)$ case, we first compute the capacity C of the constraint by finding the largest eigenvalue λ of T: $C = \log_2 \lambda = 0.9157 \cdots$. This value of C indicates that a code is possible at rate $8/9 = 0.8888 \cdots$. The rate $8/9$ is a good choice of rate because it is close to the Shannon capacity, and it is well adapted to standard byte-oriented data processing. We also note that $(0, \boldsymbol{G}/\boldsymbol{I}) = (0, 3/3)$ has the smallest $\boldsymbol{G}$ and $\boldsymbol{I}$ values for which coding at rate $8/9$ is possible. (We may assume $\boldsymbol{G} \leq 2\boldsymbol{I}$ and, by computation, observe that $(0, \boldsymbol{G}/\boldsymbol{I}) = (0, 2/\infty)$ and $(0, \boldsymbol{G}/\boldsymbol{I}) = (0, 4/2)$ have capacity $C = 0.8791 \cdots < 8/9$.)

With this selection of code rate, the next step is to take the 9th power of T, and find a $(T^9, 2^8)$-approximate eigenvector $\boldsymbol{v}$. For example, for the $(0, \boldsymbol{G}/\boldsymbol{I}) = (0, 3/3)$ parameters, one such approximate eigenvector is

$$\boldsymbol{v} = [3, 2, 2, 1, 3, 2, 1, 0, 2, 2, 0, 1]^T$$

with components corresponding to the states in lexicographic order. The upper bound of final states is the sum of the components, or 19. The lattice of states for $(0, \boldsymbol{G}/\boldsymbol{I}) = (0, 3/3)$ is shown in Fig. 34, along with the approximate eigenvector components and weight minimal states (circled). States with weight 0 are ignored in the code construction process. Note that states (1, 2) and (1, 3) are shifted from their normal grid positions to the far right edge because the global run achieves the maximum value $g = 3$.

Table IV gives the Shannon capacity of selected $(0, \boldsymbol{G}/\boldsymbol{I})$ constraints. The techniques described in Sections III–V were used to construct rate $8/9$, finite-state $(0, \boldsymbol{G}/\boldsymbol{I})$ codes with sliding-block decoders for each of these constraints. A summary of the resulting code parameters is included in Table IV. As an example of the effectiveness of these techniques, the $(0, \boldsymbol{G}/\boldsymbol{I}) = (0, 3/3)$ code was reduced in a systematic way to a four-state encoder, far less complex than the original upper bound of 19.

We remark that the techniques described here were applied to the design of a rate 8:9, modified $(0, \boldsymbol{G}/\boldsymbol{I})$ code

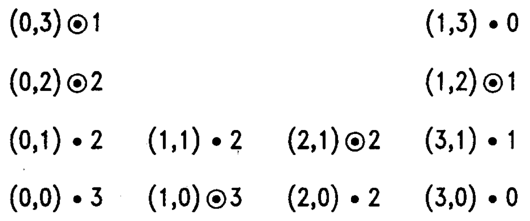


Fig. 34. Lattice of states for (0, G/I) = (0, 3/3). (Approximate eigenvector shown, with weight minimal states circled.)

TABLE IV
Compendium of (0, G/I) Codes

(0, G/I)	Capacity	Rate	Efficiency (%)	Encoder States	Decoder Look-ahead (bits)
(0, 4/4)	0.961366	8/9	92.4	1	0
(0, 4/3)	0.939505	8/9	94.6	3	0
(0, 3/6)	0.944539	8/9	94.1	1	0
(0, 3/5)	0.941533	8/9	94.4	2	0
(0, 3/4)	0.934253	8/9	95.1	3	8
(0, 3/3)	0.915723	8/9	97.0	4	7

for the PRML channel incorporated into a recent IBM disk drive. We now discuss the design of some of the codes listed in Table IV. Details about the other codes can be found in [25].

B. (0, G/I) = (0, 3/6) and (0, 4/4)

Consider first the constraint (0, G/I) = (0, 4/4). The states form a lattice structure as shown in Fig. 35, where state labels have been omitted where they agree with grid coordinates, starting with (0, 0) at the lower left.

The three states in position (4, 2) are, from left to right, (2, 2), (3, 2), and (4, 2). All have global run length g of maximum value 4 and so have been moved to the far right and lumped together in one state. The capacity of this constraint is 0.961366, and it is again reasonable to attempt to construct a code with rate 8/9. For the adjacency matrix T, the AE algorithm indicates that there is a (T^9, 2^8)-approximate eigenvector with components all equal to 0 or 1, indicating that a code at rate 8/9 can be constructed without state splitting. We use the approximate eigenvector $\boldsymbol{v}$ = [1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0]^T with components corresponding to states in lexicographic order. Eliminating those states corresponding to 0 components we are left with the irreducible FSTD based upon the seven states shown in Fig. 36.

There are two minimal states, (0, 2) and (2, 1), in Fig. 35. This reduction leaves only two states, which we continue to label (0, 2) and (2, 1), but the merging of nonminimal states with minimal states here involves some choices. We can merge (0, 0) and (0, 1) with either (0, 2) or (2, 1). We arbitrarily choose the latter. In some cases, the choice can be made to simplify the relationship between codewords and next state functions in the encoder [25].

The resulting adjacency matrix for the two-state FSTD,

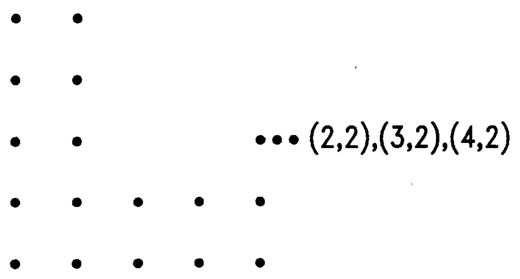


Fig. 35. Lattice of states for (0, G/I) = (0, 4/4).

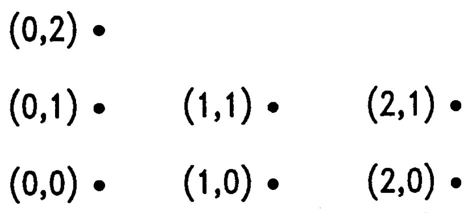


Fig. 36. Reduced lattice for 9th power of (0, G/I) = (0, 4/4).

and its rowsums, are given by

$$\begin{array}{cccc} & (0,2) & (2,1) & \text{Total} \\ (0,2) & 27 & 298 & 325 \\ (2,1) & 28 & 269 & 297. \end{array}$$

The number of codewords in the intersection of the labels of the outgoing edges are shown in the Venn diagram in Fig. 37.

A two-state code could now be constructed by selecting a list of 256 codewords from each row. (In fact, the astute reader will notice that, as a result of the merging, a block code can be derived quite simply by choosing 256 codewords from the 269 codewords that start and end at the state (2, 1) in the merged FSTD.) To design a block decoder, one first finds the codewords in the common intersection of the two states. The assignment of 8-bit input patterns to codewords should then be made consistently; that is, if a codeword in the intersection is included in both lists of 256 words, it should correspond to the same 8-bit input tag. In this case, we find that the lists for the merged states (0, 2) and (2, 1) have a common intersection which contains more than 256 codewords, as indicated by the intersections shown in Fig. 37. Specifically, there are 279 codewords of length 9 emanating from both states (0, 2) and (2, 1). These codewords can be freely concatenated without violating the constraints so a block code (that is, code with one state) for (0, G/I) = (0, 4/4) at rate 8/9 can be based on any subset of these 279 words of size 256. Moreover, applying the technique described in Section V-A, one can verify that these 279 form the unique optimal block code of length 9 for this constraint.

These 279 codewords were first found by J. Eggenberger [55] by eliminating all 9-bit blocks that have more than four consecutive symbols "0" anywhere in the block, more than two consecutive symbols "0" at the beginning or end, or more than two consecutive symbols "0" at the beginning or end in even or odd substrings.

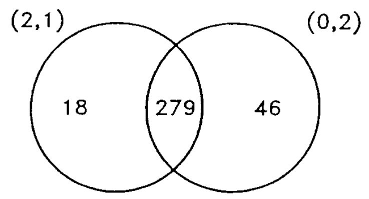


Fig. 37. Intersection diagram for $(0, G/I) = (0, 4/4)$.

We remark here that, in a similar fashion, one can use these minimal state techniques to rederive the two optimal block codes of length 9 for the $(0, G/I) = (0, 3/6)$ constraint. These codes, originally discovered by J. Eggenberger, consist of 272 words.

The ordered states in the original $(0, G/I) = (0, 3/6)$ diagram are shown in Fig. 38. The first code was derived from the approximate eigenvector $v = [1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0]^T$. The states with component 1 in the approximate eigenvector are enclosed in the box, and the minimal states which had a common intersection of 272 words are circled.

The minimal state techniques also generate the approximate eigenvector $v = [1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0]^T$. This alternative approximate eigenvector leads to the 272 words which constitute the second optimal block code. The words in this list are precisely the reverse binary representations of those in the first one. This symmetry was first pointed out by Eggenberger, who gave a simple description of these codes, similar in nature to that of the (0/4, 4) code. Namely, the first code consists of all 9-bit blocks that have no more than three consecutive 0's anywhere in the block, no more than two consecutive 0's at the beginning of the block, no more than one symbol 0 at the end of the block, and no more than three consecutive 0's anywhere in the even or odd substrings. The second optimal block code is described similarly, the only difference being that the blocks have no more than one consecutive 0 at the beginning, and no more than two consecutive 0's at the end.

C. $(0, G/I) = (0, 3/5)$

Of the PRML constraints for which a rate 8:9 *block* code is not possible, the $(0, G/I) = (0, 3/5)$ constraint turns out to have the simplest encoder. The (0, 3/5) constraint has the ordered state lattice shown in Fig. 39. The Shannon capacity is 0.941533, and once again there is an approximate eigenvector for the 9th power of the adjacency matrix, with $n = 256$, which has only 0 and 1 components. The set of states τ with component 1 are enclosed in the box, with the two τ-minimal states circled.

The common intersection is 251 codewords, as seen in the intersection diagram in Fig. 40, and one can check that this is optimal for length 9.

A two state code can be constructed by merging nonminimal states (0, 0), (1, 0), and (2, 0) with minimal state (3, 0), and merging state (0, 1) with minimal state (0, 2). This amalgamation then makes the next state function depend only on the last bit of the codeword, namely final bit "1" implies next state is (3, 0) and "0" implies (0, 2).

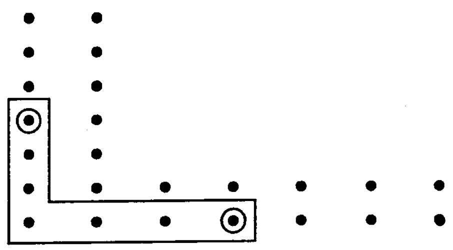


Fig. 38. Lattice of states for $(0, G/I) = (0, 3/6)$.

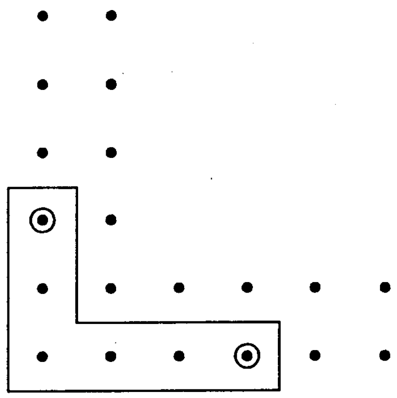


Fig. 39. Lattice and minimal states for $(0, G/I) = (0, 3/5)$.

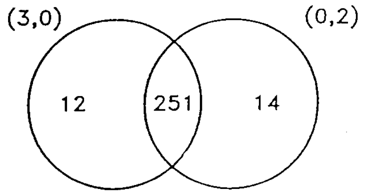


Fig. 40. Intersection diagram for $(0, G/I) = (0, 3/5)$.

The code is very "close" to a block code; the two stages agree on 250 codewords. Moreover, the codewords which differ do so only on their second and fifth bits. For state (3, 0), bit 2 of these words is always "0", whereas for state (0, 2) it is "1." Similarly, for state (3, 0), bit 5 of these words is "1," while it is "0" for state (0, 2).

The decoder for this code is state independent, taking the form of a look-up table with 262 entries. The details of a rate 8:9, (0, 3/5) code are given in [56].

D. $(0, G/I) = (0, 3/3)$

The minimal-state reduction in the $(0, G/I) = (0, 3/3)$ case leaves the states which are circled in Fig. 41 (extracted, for convenience, from the lattice diagram in Fig. 34).

The weight minimal states in the group with approximate eigenvector component 1 are (0, 3) and the right-shifted (1, 2). For component value 2, the minimal states are (0, 2) and (2, 1). Finally, for value 3, the only minimal state is (1, 0). The new upper bound on the number of states after splitting is 9, versus the original 19.

The ordering on the weight minimal states provides in-

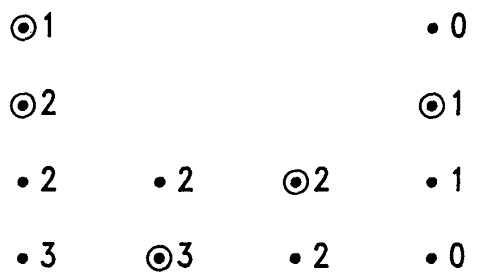


Fig. 41. Weight minimal states for $(0, G/I) = (0, 3/3)$.

sight into how the states should be split so as to further simplify the code structure. In the sense of the ordering, state (0, 2) with component value 2 exceeds states (0, 3) and right-shifted (1, 2), both with value 1. Similarly, state (1, 0) of value 3 exceeds state (2, 1) of value 2, which in turn also exceeds right-shifted (1, 2) of value 1. These relationships suggest splitting (0, 2) into two states which could then be merged with (0, 3) and right-shifted (1, 2), respectively. Likewise, state (2, 1) could split into two states, one of which could merge with right-shifted (1, 2). These two states would themselves merge with two of the three states into which we split (1, 0). Ideally, then, the code would reduce to only four states, represented by the three states split from (1, 0) and the one state from (0, 3). This plan for splitting states was in fact carried out, resulting in a surprisingly simple four-state encoder for the $(0, \boldsymbol{G/I}) = (0, 3/3)$ constraint at rate 8/9. Details of a four-state encoder for $(0, \boldsymbol{G/I}) = (0, 3/3)$ can be found in [25], [26].

VIII. Almost-Finite-Type Systems and Noncatastrophic Encoders

Recall from Section IV that the charge-constrained systems are not finite-type systems. However, they do belong to the more general class of almost-finite-type systems, which we now describe. The almost-finite-type systems can be thought of as "locally finite-type." Recall the notions of finite local anticipation and finite local memory that were first introduced in Section II. A constrained system is *almost-finite-type* if it can be represented by an FSTD that has both finite local anticipation and finite local memory. From Proposition 4, we know that finite memory and anticipation implies finite local anticipation, and the same proof shows that it also implies finite local memory. Thus, every constrained system which is finite-type is also almost-finite-type, and so the almost-finite-type systems do indeed include the finite-type systems. From Fig. 2 (in the Introduction) we see that the charge-constrained systems are represented by FSTD's with local anticipation 0 and local memory 0; thus, these systems are almost-finite-type, but not finite-type. Most every constrained system that has been used in practical applications is almost-finite-type.

As with finite-type systems, we have the problem that a given constrained system may have some representation that satisfies the finite local memory and finite local anticipation conditions and another representation that does not. Unfortunately, in contrast to finite-type systems, the intrinsic condition that defines almost-finite-type is very hard to state (see [57]). So, at this point it is most convenient to introduce a distinguished representation called the *Shannon cover*. We give only a brief discussion of the Shannon cover, and we refer the reader to [29] for more details.

The Shannon cover G_s is the minimal (in terms of number of states) deterministic representation of S. It turns out that for constrained systems that can be represented by an irreducible FSTD (e.g., run-length limited and charge constrained systems), the Shannon cover is unique. In fact, the Shannon cover can be constructed from any irreducible deterministic representation G in the following well-known way. If two states i, j in G have the same follower set, merge them, i.e., form the (i, j)-merger as described in Section V-A. It is easy to see that the merger is still deterministic, and a modification of the proof of Proposition 5 shows that the merger represents the same constrained system S. Continue this procedure until the follower sets of all states are distinct, i.e., continue until you reach an FSTD that has the property that for every pair of states, i, j, there is a sequence that can be generated from i but not from j or vice versa. This procedure must terminate since there are only finitely many states to begin with. The result of this procedure turns out to be the Shannon cover G_S. In particular (see [58]) the Shannon cover is characterized by the following properties:

1) G_S is deterministic;

2) the follower sets of the states of G_S are distinct.

There is an algorithm [31, sect. 10-3] to determine whether or not the follower sets of an FSTD are distinct, but usually a casual glance at the FSTD is sufficient to decide. Also, there is an algorithm [31, sect. 16-3] to construct a deterministic representation of any constrained system from any given representation, so that we can start the procedure for finding the Shannon cover (and so that we can also start the code construction procedure). But as mentioned earlier, typically most constrained systems are represented deterministically in the first place.

As an example, the FSTD in Fig. 6(b) is a deterministic representation of the RLL (0, 1) constrained system, but it is not the Shannon cover because states 1 and 3 have the same follower set. And indeed the FSTD in Fig. 6(a) is the Shannon cover of the RLL (0, 1) constrained system because it is deterministic and 0 is the label of an outgoing edge from state 1, but not from state 2. Note that if we merge states 1 and 3 in the FSTD of Fig. 6(b), we get the Shannon cover in Fig. 6(a). The reader can verify from the characterization above that the Shannon cover of a run-length limited constraint is the usual FSTD depicted in Fig. 1 of the Introduction and that Fig. 2 displays the Shannon cover for the charge-constrained systems.

Now, we mention one of the main uses of the Shannon cover. It can be used to tell if a given constrained system is finite-type or almost-finite-type or neither.

Proposition 7: A constrained system representable by an irreducible FSTD is finite-type (respectively, almost-finite-type) if and only if its Shannon cover has finite memory and anticipation (respectively, has both finite local anticipation and finite local memory) [58, theorem 4(iv)], [59, corollary 11], [60].

From this, one can see that the constrained system represented by the FSTD in Fig. 42 is not almost-finite-type. Specifically, one can easily verify that the FSTD in the figure is the Shannon cover of the constraint. However, it does not have finite local memory, as can be confirmed by looking at the paths that generate strings of the form $\cdots aaaab$.

As a consequence of this proposition, we also get a fact that was claimed in Section IV: since the Shannon cover is deterministic, every finite-type constrained system has a representation (the Shannon cover) with memory m and anticipation 0 (for some m).

Finally, we state a result that generalizes both Theorems 1 and 2. This result appears in [24], but the groundwork was laid by [22].

Theorem 3: Let S be a constrained system with Shannon capacity $Cap(S)$. Let p, q be positive integers satisfying the inequality

$$p/q \leq Cap(S).$$

Then, there exists a noncatastrophic finite-state encoder that encodes binary data into the constraint S at constant rate $p:q$. Moreover, if S is almost-finite-type (in particular, if S is finite-type), then the encoder has a sliding-block decoder.

Thus, for almost-finite-type constrained systems, which include the charge-constrained systems, the decoder can be made sliding-block, ensuring that decoder error propagation is limited in both number and time extent by a fixed constant. For general constrained systems, the error propagation is guaranteed only to be limited in number. Indeed, [24] describes an example of a constrained system with rational capacity, for which a finite-state encoder with sliding-block decoder *cannot* be constructed at rate equal to capacity (of course, by Theorem 3, such a constrained system cannot be almost-finite-type). However, it is shown in [23] (see also [29]) that if one replaces the assumption

$$p/q \leq Cap(S)$$

by

$$p/q < Cap(S),$$

then the encoders can always be constructed to have sliding-block decoders. But even in this case, it may be that there are noncatastrophic encoders for some constraint which are much simpler than any encoder with the same rate that has a sliding-block decoder.

The proof of Theorem 3 for the finite-type case was given in Sections III and IV, and it effectively provides a practical algorithm for the construction of efficient, finite-state codes with sliding-block decoders.

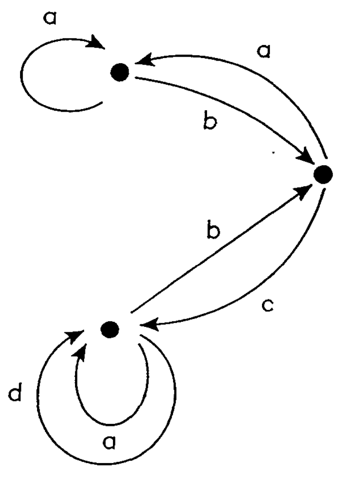


Fig. 42. A system which is not almost-finite-type.

The proof for the almost-finite-type case is substantially deeper mathematically, and therefore more complicated. Although it does not exactly provide a practical code construction algorithm, the proof makes use of some very powerful techniques that can be brought to bear in particular applications. A nontrivial example of a code with 'reasonable' complexity for a particular combined run-length limited/charge-constrained system called the $(d, k; c) = (1, 3; 3)$ constraint, is described by Karabed and Siegel [42]. Several of the new ideas in the generalization to almost-finite-type systems have also played a role in the recent design of coded-modulation schemes based upon spectral null constraints (see, for example, [8]).

In general, from the complexity point of view, there is still a great deal of work left to be done in understanding how to construct finite-state encoders with sliding-block decoders for charge constraints and more generally, for spectral null constraints.

IX. Conclusions

A self-contained treatment of modulation code design methods based upon the state splitting algorithm has been presented. The methods were applied to the construction of recording codes for digital data storage.

Appendix A
Perron–Frobenius Theory

The Perron–Frobenius Theory is a rich collection of results concerning the eigenvalues, eigenvectors, and general structure of nonnegative matrices. Here we state only the most relevant parts of the theory as it applies to FSTD's, and we derive some consequences that are used in the text of this paper. We refer the reader to one of the many excellent texts [61], [62] on this subject for further information.

Recall that for an FSTD G, $\lambda(A(G))$ denotes the largest real eigenvalue of $A(G)$.

Theorems A1 and A2 are both parts of the classical Perron–Frobenius Theory.

Theorem A1: Let G be an irreducible FSTD. Then:

1) $\lambda(A(G)) > 0$ and $A(G)$ has a positive (i.e., all components are positive) eigenvector associated with the eigenvalue $\lambda(A(G))$.

2) The multiplicity of $\lambda(A(G))$ as an eigenvalue of $A(G)$ is 1; i.e., $\lambda(A(G))$ appears as a root of the characteristic polynomial of $A(G)$ with multiplicity 1. In particular, all eigenvectors associated with the eigenvalue $\lambda(A(G))$ are scalar multiples of one another.

3) $\lambda(A(G)) \geq |\mu|$ for all eigenvalues μ of $A(G)$.

Proof: See [61, p. 53].

Theorem A2: Let G be an FSTD. Then G has an irreducible component H such that $\lambda(A(H)) = \lambda(A(G))$.

Proof: See [61, p. 69].

Using these results we can now verify the capacity formula (1) given in Section II.

Theorem A3: Let G be an FSTD which represents a constrained system S. If G has finite local anticipation (in particular, if G is deterministic), then

$$Cap(S) = \log_2 (\lambda(A(G))).$$

Proof: Let $P(n; G)$ be the number of paths of length n in G. The proof is broken into two parts. We first show

$$\lim_{n \to \infty} \frac{\log_2 (P(n; G))}{n} = \log_2 \lambda(A(G)) \tag{6}$$

and then we show

$$\lim_{n \to \infty} \frac{\log_2 (P(n; G))}{n} = \lim_{n \to \infty} \frac{\log_2 (N(n; S))}{n} = Cap(S) \tag{7}$$

(recall that $N(n; S)$ is the number of sequences of length n in S). The desired result is obtained by combining (6) and (7).

Equation (6) can be obtained by applying the theory of difference equations; namely $P(n; G)$ is the sum of the components of a solution to a system of linear difference equations.

As an alternative proof of equation (6) in the irreducible case, we make use of the positive eigenvector x associated with the eigenvalue $\lambda(A(G))$. Let $x_{\max}$, $x_{\min}$ denote the maximal, minimal components of x. So $0 < x_{\min} \leq x_{\max}$. Then for each state i

$$x_{\min} \sum_j (A(G)^n)_{ij} \leq \sum_j (A(G)^n)_{ij} x_j = \lambda(A(G))^n x_i. \tag{8}$$

Thus,

$$P(n; G) = \sum_{ij} (A(G)^n)_{ij} \leq \lambda\, (A(G))^n \frac{\sum x_i}{x_{\min}}. \tag{9}$$

Replacing $x_{\min}$ by $x_{\max}$ and reversing the direction of the inequalities, we get

$$P(n; G) = \sum_{ij} (A(G)^n)_{ij} \geq \lambda(A(G))^n \frac{\sum x_i}{x_{\max}}. \tag{10}$$

Thus, by (9) and (10), the ratio of $P(n; G)$ to $(\lambda(A(G)))^n$ is bounded above and below by positive constants. So, these two quantities grow at the same rate, i.e., (6) holds. This result can be extended to the reducible case by making use of the Jordan canonical form of the matrix $A(G)$ [63]. See also [29].

To verify (7), first observe that since each sequence generated by G is produced by at least one path in G of the same length

$$P(n, G) \geq N(n; S). \tag{11}$$

Now, we claim that there is some number M such that any sequence can be generated by at most M paths. This is a consequence of the assumption that G has finite local anticipation $(=a)$: M is the product of the number of states of G and the maximal number of paths of length a that can generate the same sequence starting at the same state. Thus

$$MN(n; S) \geq P(n; G). \tag{12}$$

Now, (11) and (12) yield (7). Q.E.D.

Note that the proof of Theorem A3 shows that the capacity formula given by (1) holds more generally: namely, when the number of paths that generate the same sequence is bounded above, independent of the sequence.

The next result is used in Section III of this paper. It shows that approximate eigenvectors exist when we need them, and thus that if, in the approximate eigenvector algorithm, one sets L sufficiently large, then the algorithm will indeed find an approximate eigenvector.

Theorem A4: Let G be an FSTD and n a positive integer with $\lambda(A(G)) \geq n$. Then there is an $(A(G), n)$-approximate eigenvector, i.e., a nonnegative integer vector $\boldsymbol{v} \not\equiv \boldsymbol{0}$ such that

$$A\boldsymbol{v} \geq n\boldsymbol{v}.$$

Proof: We first prove this under the assumption that G is irreducible. By Theorem A1 (part 1), $A(G)$ has a positive eigenvector $\boldsymbol{x}$ associated with $\lambda(A(G))$. There are the following two cases.

Case 1: $\lambda(A(G)) > n$.

In this case, we can change the entries of $\boldsymbol{x}$ slightly to obtain a new vector $\boldsymbol{x}'$, with positive rational entries, that satisfies the inequality $A(G)\boldsymbol{x}' \geq n\boldsymbol{x}'$.

Now, let $\boldsymbol{v}$ be the vector obtained from $\boldsymbol{x}'$ by clearing denominators, i.e., by multiplying $\boldsymbol{x}'$ by any common multiple of the denominators of its entries. The vector $\boldsymbol{v}$ has positive integer entries, and it satisfies the approximate eigenvector inequality since $\boldsymbol{x}'$ does. Thus, $\boldsymbol{v}$ is an $(A(G), n)$-approximate eigenvector.

Case 2: $\lambda(A(G)) = n$.

By Theorem A1 (part 1), there is a nontrivial solution to the homogeneous linear system of equations

$$(A - nI)\boldsymbol{x} = \boldsymbol{0}.$$

Since the coefficients of this linear system are rational numbers, we can apply Gaussian elimination to obtain a solution to the system with rational entries. Clearing denominators, we obtain a solution $\boldsymbol{v}$ with integer entries. But combining parts 1 and 2 of Theorem A1, we see that every solution to the system is a multiple of a positive vector. Thus, either $\boldsymbol{v}$ or $-\boldsymbol{v}$ is a positive integer eigenvector associated with eigenvalue $\lambda(A(G)) = n$; in particular, it is an $(A(G), n)$-approximate eigenvector. This completes the proof in case G is irreducible.

If G is reducible, then there is, by Theorem A2, an irreducible component H of G with $\lambda(A(H)) = \lambda(A(G)) \geq n$. Thus, by what we have just proven, there is an $(A(H), n)$-approximate eigenvector $\boldsymbol{v}$. Now, the vector $\boldsymbol{v}$ is indexed by the states of H. We extend $\boldsymbol{v}$ to a vector indexed by the states of G by simply setting to zero the entries corresponding to those states of G that are not states of H. This new vector is an $(A(G), n)$-approximate eigenvector. Q.E.D.

Appendix B
Optimal Block Codes

To prove that the procedure outlined at the end of Section V-A produces an optimal block code for length q, we briefly review the results of Freiman and Wyner [40]. The design method they develop relies upon the partial ordering of follower sets by inclusion. The code construction is simplified by the introduction of the concept of a complete set of states, as we now describe.

Let τ denote a subset of states of G, $\tau \subseteq V(G)$. We say that τ is *complete* with respect to the partial ordering of follower sets if, whenever a state i belongs to τ and $i < j$, then j belongs to τ.

We assume that the FSTD G has memory m and anticipation 0, and choose a block length $q \geq m$. We can also assume that no two states in G have identical follower sets; otherwise, we can merge them. Recall that $\mathcal{L}(\tau)$ denotes the block code for length q obtained by taking the intersection of the lists of q-blocks that are generated by the states in τ, and that end in some state in τ. A main result in [40] is that an optimal block code will be $\mathcal{L}(\tau)$ for some complete set of states τ.

Suppose that the optimal block code for length q has $M \geq 2^p$ codewords, and is generated by the complete set τ. For convenience, we will say that a "0-1" vector is complete if the set of states τ corresponding to the components with value 1 is complete. We will show that the procedure described at the end of Section V-A generates every complete "0-1" $(A^q, 2^p)$-approximate eigenvector. By [40], we conclude that we will find an optimal block code for length q.

So, let $\boldsymbol{v}$ and $\boldsymbol{w}$ be distinct "0-1" $(A^q, 2^p)$-approximate eigenvectors satisfying $\boldsymbol{w} \leq \boldsymbol{v}$, and assume $\boldsymbol{w}$ is complete. For any state k in the set τ determined by $\boldsymbol{v}$, let $\boldsymbol{v}^{[k]}$ be the vector obtained from $\boldsymbol{v}$ by setting $v_k = 0$

$$\boldsymbol{v}_i^{[k]} = \begin{cases} v_i & \text{if } i \neq k \\ 0 & \text{if } i = k. \end{cases}$$

We now prove that $\boldsymbol{w} \leq \boldsymbol{v}^{[k]}$, for some k that is τ-minimal. The desired result then follows easily.

Since $\boldsymbol{w} \neq \boldsymbol{v}$, we know that $\boldsymbol{w} \leq \boldsymbol{v}^{[k]}$, for some state k in τ. If k is τ-minimal, we are done. So, suppose it is not. Then, there must be a state $j \neq k$ in τ, implying $v_j = 1$, such that $j < k$. If $w_j = 1$, we contradict the assumption that $\boldsymbol{w}$ is complete. So $w_j = 0$, and $\boldsymbol{w} \leq \boldsymbol{v}^{[j]}$. If j is τ-minimal, we are done. If not, we iterate this argument until we run into a τ-minimal state (since no two states have the same follower set, we are indeed bound to run into a τ-minimal state).

It is interesting to note that there is a sort of "converse" result. Namely, each approximate eigenvector obtained during the procedure is in fact complete. To prove this, we will make frequent use of a fact mentioned in Section III-B: the approximate eigenvector algorithm generates the largest approximate eigenvector dominated by the initial vector $\boldsymbol{v}^{(0)}$.

It is easy to see that the largest "0-1" approximate eigenvector $\boldsymbol{v}$ is complete. If it were not, we could find states s and t such that, $v_s = 1$, $v_t = 0$, and $s < t$. If we flip position t to $v_t = 1$, the corresponding vector $\boldsymbol{w}$ will certainly satisfy the approximate eigenvector inequality (2). The vector $\boldsymbol{w}$ is strictly larger than $\boldsymbol{v}$. However, the vector $\boldsymbol{v}$ is larger than any other approximate eigenvector, a contradiction. Therefore, $\boldsymbol{v}$ must be complete.

Now, let $\boldsymbol{v}$ be an approximate eigenvector generated during the course of the procedure. By induction, we may assume that it is complete. Let k be a τ-minimal state in the set τ defined by $\boldsymbol{v}$, and let $\boldsymbol{w}$ denote the vector that differs only in position k, $w_k = 0$. Apply the approximate eigenvector algorithm to get the vector $\boldsymbol{z}$.

If $\boldsymbol{z}$ is not complete, then there is a pair of states $i < j$ with $z_i = 1$ and $z_j = 0$. Setting $z_j = 1$ gives a vector $\boldsymbol{x}$ that is strictly larger than z and that also must satisfy the approximate eigenvector inequality. We will show that this leads to a contradiction.

First, we claim that $j \neq k$. If $j = k$, then we would have $i < k$. By τ-minimality of k, and the assumption that no two states have the same follower set, we would conclude that $i = k$, so $z_i = z_k$. But by construction, $z_k = 0$, and by definition of i, $z_i = 1$. It follows that, indeed, $j \neq k$.

Since $\boldsymbol{v}$ dominates $\boldsymbol{z}$, and $\boldsymbol{v}$ is complete, we know that $v_j = 1$. This implies, in turn, that $w_j = 1$, since $\boldsymbol{w}$ differs from $\boldsymbol{v}$ only in position k, and we have seen that $k \neq j$. However, this means that $\boldsymbol{x} \leq \boldsymbol{w}$. Combining this relation with $\boldsymbol{z} \leq \boldsymbol{x}$ and $\boldsymbol{z} \neq \boldsymbol{x}$ leads to a contradiction, since $\boldsymbol{z}$ is the largest approximate eigenvector smaller than $\boldsymbol{w}$, by its definition. Therefore, $\boldsymbol{z}$ must in fact be complete, proving the claim.

To summarize, the procedure finds, for a given p, the length-q block codes generated by the complete sets that support a finite-state code at rate $p:q$. If an optimal code has $M \geq 2^p$ words, the procedure will find one.

Acknowledgment

The authors gratefully acknowledge useful discussions with A. Khayrallah, D. Neuhoff, R. Roth, and N. Swenson. They also thank J. Fitzpatrick and T. Weigandt for carefully reading and commenting upon an earlier version of this manuscript.

References

[1] P. Siegel, "Applications of a peak detection channel model," *IEEE Trans. Magnet.*, vol. MAG-18, no. 6, pp. 1250-1252, Nov. 1984.

[2] —, "Recording codes for digital magnetic storage," *IEEE Trans. Magnet.*, vol. MAG-21, no. 5, pp. 1344-1349, Sept. 1985.

[3] K. A. Schouhamer Immink, "Runlength-limited sequences," *Proc. IEEE*, vol. 78, no. 11, pp. 1745-1759, Nov. 1990.

[4] H. Kobayashi and D. T. Tang, "Application of partial-response channel coding to magnetic recording systems," *IBM J. Res. Develop.*, vol. 14, pp. 368-374, 1970.

[5] F. Dolivo, D. Maiwald, and G. Ungerboeck, "Partial-response class-IV signaling with Viterbi decoding versus conventional modified frequency modulation in magnetic recording," IBM Res., Zurich Res. Lab., IBM Res. Rep. RZ 973-33865, Switzerland, Aug. 1979.

[6] R. Wood and D. Petersen, "Viterbi detection of class IV partial response on a magnetic recording channel," *IEEE Trans. Commun.*, vol. COM-34, no, 5, pp. 454-461, May 1986.

[7] F. Dolivo, "Signal processing for high density digital magnetic recording," in *Proc. COMPEURO 89*, Hamburg, Germany, May 1989.

[8] R. Karabed and P. Siegel, "Matched spectral-null codes for partial-response channels," *IEEE Trans. Inform. Theory*, vol. 37, no. 3, pt. II, pp. 818-855, May 1991.

[9] C. Shannon, "A mathematical theory of communication," *Bell Syst. Tech. J.*, vol. 27, pp. 379-423, 623-656, Oct. 1948.

[10] P. Franaszek, "Sequence-state coding for digital transmission," *Bell Syst. Tech. J.*, pp. 113-157, 1968.

[11] —, "On synchronous variable length coding for discrete noiseless channels," *Inform. Contr.*, vol. 15, pp. 155-164, 1969.

[12] —, "Sequence-state methods for run-length-limited coding," *IBM J. Res. Develop.*, vol. 14, pp. 375-383, July 1970.

[13] —, "A general method for channel coding," *IBM J. Res. Develop.*, vol. 24, pp. 638-691, 1980.

[14] —, "Construction of bounded delay codes for discrete noiseless channels," *IBM J. Res. Develop.*, vol. 26, pp. 506-514, 1982.

[15] M.-P. Beal, "The method of poles: a coding method for constrained channels," *IEEE Trans. Inform. Theory*, vol. 36, no. 4, pp. 763-772, July 1990.

[16] D. Tang and L. Bahl, "Block codes for a class of constrained noiseless channels," *Inform. Contr.*, vol. 17, pp. 436-461, 1970; and pp. 462-474, 1973.

[17] G. Jacoby, "A new look-ahead code for increased data density," *IEEE Trans. Magnet.*, vol. MAG-13, no. 5, pp. 1202-1204, Sept. 1977.

[18] G. Jacoby and R. Kost, "Binary two-thirds rate code with full word look-ahead," *IEEE Trans. Magnet.*, vol. MAG-20, no. 5, pp. 709-714, Sept. 1984.

[19] M. Cohn, G. Jacoby, and A. Bates, III, "Data encoding method and system employing two-thirds code rate with full word look-ahead," U.S. Patent 4,337,458, 1982.

[20] A. Lempel and M. Cohn, "Lookahead coding for input restricted channels," *IEEE Trans. Inform. Theory*, vol. IT-28, pp. 933-937, Nov. 1982.

[21] A. Patel, "Zero modulation encoding in magnetic recording," *IBM J. Res. Develop.*, vol. 19, no. 4, pp. 366-378, July 1975.

[22] R. Adler, D. Coppersmith, and M. Hassner, "Algorithms for sliding-block codes," *IEEE Trans. Inform. Theory*, vol. IT-29, no. 1, pp. 5-22, Jan. 1983.

[23] B. Marcus, "Sofic systems and encoding data," *IEEE Trans. Inform. Theory*, vol. IT-31, no. 3, pp. 366-377, May 1985.

[24] R. Karabed and B. Marcus, "Sliding-block coding for input-restricted channels," *IEEE Trans. Inform. Theory.*, vol. 34, no. 1, pp. 2-26, Jan. 1988.

[25] B. Marcus and P. Siegel, "Constrained codes for PRML," IBM Res. Rep. RJ 4371, July 1984.

[26] —, "Constrained codes for partial response channels," in *Proc. Beijing Int. Workshop Inform. Theory*, July 1988, pp. DI1.1-DI1.4.

[27] R. E. Blahut, *Digital Transmission of Information*. Reading, MA: Addison-Wesley, 1990.

[28] K. A. Schouhamer Immink, *Coding Techniques for Digital Recorders*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

[29] A. Khayrallah and D. Neuhoff, "Subshift models and finite-state codes for input-constrained noiseless channels: A tutorial," preprint (based upon Ph.D. dissertation by A. Khayrallah, Univ. Mich., 1989).

[30] N. Swenson and J. Cioffi, "A simplified design approach for run-length limited sliding block codes," preprint (based upon Ph.D. dissertation by N. Swenson, Stanford Univ., 1991).

[31] Z. Kohavi, *Switching and Finite Automata Theory*. New York: McGraw-Hill, 1970.

[32] B. Marcus and R. Roth, "Bounds on the number of states in encoder graphs for input-constrained channels," *IEEE Trans. Inform. Theory*, vol. 37, no. 3, pt. II, pp. 742-758, May 1991.

[33] J. Ashley, "A linear bound for sliding-block decoder window size," *IEEE Trans. Inform. Theory*, vol. 34, no. 3, pp. 389-399, May 1988.

[34] H. Kamabe, "Minimum scope for sliding-block decoder mappings," *IEEE Trans. Inform. Theory*, vol. 35, no. 6, pp. 1335-1340, Nov. 1989.

[35] R. Williams, "Classification of shifts of finite type," *Ann. Math.*, vol. 98, pp. 120-153, 1973; and "Errata," *Ann. Math.*, vol. 99, pp. 380-381, 1974.

[36] R. Adler, L. Goodwyn, and B. Weiss, "Equivalence of topological Markov shifts," *Israel J. Math.*, vol. 27, pp. 49-63, 1977.

[37] B. Marcus, "Factors and extensions of full shifts," *Monatshefte fur Math.*, vol. 88, pp. 239-247, 1979.

[38] R. Adler, J. Friedman, B. Kitchens, and B. Marcus, "State splitting for variable-length graphs," *IEEE Trans. Inform. Theory*, vol. IT-32, no. 1, pp. 108-113, Jan. 1986.

[39] C. Heegard, B. Marcus, and P. Siegel, "Variable length state splitting with applications to average runlength constrained (ARC) codes," *IEEE Trans. Inform. Theory*, vol. 37, no. 3, pt. II, pp. 759-777, May 1991.

[40] C. Freiman and A. Wyner, "Optimum block codes for noiseless input restricted channels," *Inform. Contr.*, vol. 7, pp. 398-415, 1964.

[41] P. Siegel, "On the complexity of limiting error propagation in sliding-block codes," unpublished memorandum.

[42] R. Karabed and P. Siegel, "A 100% efficient sliding-block code for the charge-constrained, runlength limited channel with parameters (d, k; c) = (1, 3; 3)," presented at the IEEE Int. Symp. Inform. Theory, Budapest, June 1991; also in IBM Res. Rep. RJ 7092, Dec. 1990.

[43] E. Berlekamp, "The technology of error-correcting codes," *Proc. IEEE*, vol. 68, no. 5, pp. 564-593, May 1980.

[44] J. Mallinson and C. Steele, "Theory of linear superposition in tape recording," *IEEE Trans. Magnet.*, vol. MAG-5, pp. 886-890, Dec. 1969.

[45] K. A. Schouhamer Immink, "Coding techniques for the noisy magnetic recording channel," *IEEE Trans. Commun.*, vol. 37, pp. 413-419, May 1989.

[46] L. Barbosa, "Minimum noise pulse slimmer," *IEEE Trans. Magnet.*, vol. MAG-17, no. 6, pp. 3340-3342, Nov. 1981.

[47] J. Ashley and P. Siegel, "A note on the Shannon capacity of run-length-limited codes," *IEEE Trans. Inform., Theory*, vol. IT-33, no. 4, pp. 601-605, July 1987.

[48] R. Adler, M. Hassner, and J. Moussouris, "Method and apparatus for generating a noiseless sliding block code for a (1, 7) channel with rate 2/3," U.S. Patent 4,413,251, 1982.

[49] A. Weathers and J. Wolf, "A new rate 2/3 sliding-block code for the (1, 7) runlength constraint with the minimal number of encoder states," *IEEE Trans. Inform. Theory*, vol. 37, no. 3, pt. II, pp. 908-913, May 1991.

[50] T. Howell, "Statistical properties of selected recording codes," *IBM J. Res. Develop.*, vol. 33, no. 1, pp. 60-73, Jan. 1989.

[51] P. Funk, "Run-length-limited codes with multiple spacing," *IEEE Trans. Magnet.*, vol. MAG-18, no. 2, pp. 772-775, Mar. 1982.

[52] D. Rugar, "Magnetooptic direct overwrite using a resonant bias coil," *IEEE Trans. Magnet.*, vol. 24, no. 1, pp. 666-669, Jan. 1988.

[53] D. Rugar and P. Siegel, "Recording results and coding considerations for the resonant bias coil overwrite technique," in G. R. Knight and C. N. Kurtz, Eds, *Topical Meeting on Optical Data Storage, Proc. SPIE*, 1989, vol. 1078, pp. 265-270.

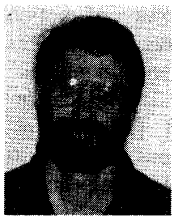
[54] T. Weigandt, "Magneto-optic recording using a (2, 18, 2) run-length-limited code," S.M. thesis, Mass. Inst. Technol., Cambridge, MA, 1991.

[55] J. Eggenberger and A. M. Patel, "Method and apparatus for implementing optimum PRML codes," U.S. Patent 4,707,681, Nov. 17, 1987.

[56] B. Marcus, A. Patel, and P. Siegel, "Method and apparatus for implementing a PRML code," U.S. Patent 4,786,890, Nov. 1988.

[57] S. Williams, "Covers of non-almost-finite-type systems," *Proc. AMS*, vol. 104, pp. 245-252, 1988.

[58] R. Fischer, "Graphs and symbolic dynamics," *Colloquia Math. Societatis Janos Bolyai*, (*Topics in Information Theory*), vol. 16, pp. 229-243, 1975.
[59] M. Boyle, B. Kitchens, and B. Marcus, "A note on minimal covers for sofic systems," *Proc. Amer. Math. Soc.*, vol. 95, no. 3, pp. 403-411, Nov. 1985.
[60] M. Nasu, "An invariant for bounded-to-one factor maps between transitive sofic subshifts," *Ergod. Theory Dynam. Syst.*, vol. 5, pp. 89-105, 1985.
[61] F. R. Gantmacher, *The Theory of Matrices, Vol. II.* New York: Chelsea, 1959.
[62] E. Seneta, *Non-Negative Matrices and Markov Chains*, 2nd ed. New York: Springer-Verlag, 1981.
[63] I. N. Herstein, *Topics in Algebra*, 2nd ed. New York: Wiley, 1975.



Brian H. Marcus (M'84) attended Claremont Men's College, Claremont, CA, received the B.A. degree from Pomona College, Claremont, CA, and the M.A. and Ph.D. degrees in mathematics from the University of California, Berkeley, in 1975.

From 1975 to 1983 he was Assistant Professor and then Associate Professor of Mathematics at the University of North Carolina, Chapel Hill. Since 1983 he has been a Research Staff Member at IBM Almaden Research Center, San Jose, CA.

Paul H. Siegel (M'82-SM'90) was born in Berkeley, CA, in 1953. He received the B.S. degree in mathematics in 1975 and the Ph.D. degree in mathematics in 1979, both from the Massachusetts Institute of Technology, Cambridge.

He joined the Research Staff at IBM in 1980. He is currently manager of the Signal Processing and Coding Project at the IBM Almaden Research Center, San Jose,CA. His primary research interest is the mathematical foundations of signal processing and coding, especially as applicable to digital data storage channels. He holds several patents in the area of coding and detection for digital recording systems. He has taught courses in information and coding at the University of California, Santa Cruz, and at Santa Clara University, and was a Visiting Associate Professor at the University of California, San Diego, while at the Center for Magnetic Recording Research during the 1989-1990 academic year. He was elected by Phi Beta Kappa in 1974. He held a Chaim Weizmann Fellowship during a year of postdoctoral study at the Courant Institute, New York University. He is currently a member of the Board of Governors of the IEEE Information Theory Society. He was a Co-Guest Editor of the May 1991 Special Issue on Coding for Storage Devices of the IEEE TRANSACTIONS ON INFORMATION THEORY, and was a member of the Program Committee for the 1991 International Symposium on Information Theory.

Jack K. Wolf (S'54-M'60-F'73), for a photograph and biography, see this issue, p. 4.