RECORDING CODES FOR DIGITAL MAGNETIC STORAGE

Paul H. Siegel
IBM Research Laboratory, San Jose, California 95193

Abstract - This paper provides a tutorial introduction to recording codes for magnetic disk storage devices and a review of progress in code construction algorithms. Topics covered include: a brief description of typical magnetic recording channels; motivation for use of recording codes; methods of selecting codes to maximize data density and reliability; and techniques for code design and implementation.

1. INTRODUCTION

Recording codes have been used with great success in magnetic disk storage to increase linear density and improve performance. This paper attempts to provide, in a limited space, a tutorial introduction to these codes.

Section 2 briefly reviews the basic principles of recording and detection in a typical magnetic storage channel, as well as the motivation for the use of codes.

Section 3 focuses on recording codes for magnetic disk storage. The recording codes in most widespread use fall into the class of run-length-limited (RLL) or (d,k) constrained codes.

Two major problems arise in the context of general RLL (d,k) codes. First, one must choose the parameters (d,k) which maximize data density and reliability. In Section 4, we address this issue of recording code performance evaluation and parameter selection.

The second major problem is the design and implementation of (d,k) codes. Section 5 is devoted to a review of code construction techniques, with an emphasis on recent advances in the development of general algorithms for the construction of sliding block RLL codes.

2. THE DIGITAL MAGNETIC RECORDING CHANNEL

Most magnetic disk storage devices in use today employ saturation recording to place the data on the disk, followed by peak detection during the readback step to recover the information. Figure 1 illustrates schematically the essential elements of the process. The data are recorded on the magnetic medium by orienting magnets along a concentric track, as shown at the top left of the figure. The magnets are oriented either in the direction of motion of the head around the track, or in the opposite direction (at least in conventional "horizontal" recording). The remaining portions of the figure show the relationship of the pattern of magnetization to the recorded bits, as dictated by the modulation scheme (known as NRZI precoding) which converts the bit stream to a 2-level write current signal for the recording head. The symbols "1" in the bit stream are recorded as "transitions" or changes in the polarity of the magnets along the track. During the readback process, the inductive read head transforms the sequence of transitions into a stream of pulses of alternating polarity. The clocking circuit (VFO, or variable-frequency oscillator) uses these pulses to maintain a synchronized timing window for the detector, which locates the pulse peaks in time. The information can then be reconstructed by placing a recovered bit "1" in any window in which a peak was detected, and a bit "0" otherwise.

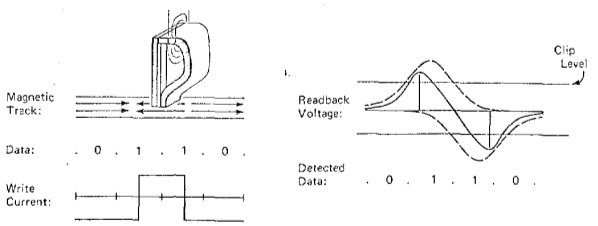


Fig. 1. Digital magnetic recording channel

Manuscript received:

P. H. Siegel
IBM Research Lab, K62/282
5600 Cottle Road
San Jose, CA 95193, U.S.A.

The recording channel has imperfections, however. Thermal noise generated by the electronic circuits and noise arising from magnetic properties of the disk medium can corrupt the readback signal, leading to spurious peaks as well as shifted positions of the genuine peaks. To compensate for these effects, at least in part, enhancements to peak detection, such as the addition of a threshold (clip level) and the tracking of peak polarities, have been introduced. Another major problem is the intersymbol interference (ISI) of neighboring pulses, illustrated in the figure. Magnetic transitions, or recorded symbols "1," which are written too close to one another have an interfering effect during readback that can both reduce the amplitude of the pulse as well as shift the peak position. On the other hand, if the transitions are written too far apart, the clock recovery circuit will not be receiving adequate information to maintain synchronization with the recorded bits.

These detection error mechanisms are combatted by the use of data coding techniques. Two distinct kinds of codes are typically used. A recording code is used to guarantee that the recorded bit sequence does not suffer from the problems described above, namely runs of symbols "0" between consecutive symbols "1" which are either too short or too long. Recording codes are also referred to as modulation codes. Even with an appropriate recording code, detection errors may still occur as a result of channel noise, producing an unacceptable error rate at the desired data density. The second kind of code, an error-correcting (also called error-control) code (ECC), uses redundant bits which have been added to the original user data to detect, locate, and correct all remaining errors with very high probability of success.

The configuration of these codes in the magnetic recording channel is shown in Figure 2. In typical high end applications, the recording code reduces the bit error rate (BER) to approximately 1 detection error in 10 billion bits. In addition, practical recording codes limit error propagation in the decoding process. This is crucial, because the ECC typically used has the capability of correcting only a small number of multi-bit bursts of errors per track.

Fig. 2. Configuration of codes

For the remainder of this paper, we focus on the aspects of recording code selection and design.

3. RUN-LENGTH-LIMITED (RLL) CODES FOR MAGNETIC STORAGE

Many popular recording codes for peak detection channels fall into the class of run-length-limited (RLL) codes. These codes, in their general form, were pioneered by P. Franaszek [1] in the late 1960's. Since then, a considerable body of engineering and mathematical literature has been written on the subject. RLL codes are characterized by two parameters *(d,k)*, which represent, respectively, the minimum and maximum number of symbols "0" between consecutive symbols "1" in the allowable code strings. The parameter *d* controls high frequency content in the associated write current signals, and so reduces the effects of intersymbol interference. The parameter *k* controls low frequency content, and ensures frequent information for the clock synchronization loop.

For example, most flexible and low-end rigid disk files, as well as some high-end drives, today incorporate a code known as Modified Frequency Modulation (MFM). It also goes by other names, such as Delay Modulation or Miller code. MFM is an RLL code with (d,k)=(1,3).

The problem faced by the coding theorist is the construction of a simple, efficient correspondence, or code mapping, between the arbitrary binary strings that a user might want to store on a magnetic disk and the (d,k) constrained code strings which the peak detector can more easily recover correctly. We now give the term "efficient" quantitative meaning by introducing the third important code parameter, the *code rate*.

The conversion of arbitrary strings to constrained (d,k) strings could be accomplished as follows. Pick a codeword

0018-9464/85/0900-1344$01.00

length n. List all the strings of length n which satisfy the (d,k) constraint. If there are at least 2^m such strings, assign a unique codeword to each of the 2^m possible binary input words of length m. This kind of code mapping is commonly referred to as a *block code*. The ratio, m/n, of input word length m to codeword length n is called the code rate. Since there are only 2^n unconstrained binary strings of length n, there will be less than this number of constrained codewords. Therefore, the rate must satisfy $m/n<1$. In fact, there is a maximum achievable rate, called the *Shannon capacity* C, which we now discuss.

In 1948, Shannon proved that, as the codeword length grows, the number of constrained codewords approaches 2^{Cn} from below, for some constant C which depends on the code constraints. This result implies that the rate m/n of any code mapping for that constraint must satisfy $m/n \leq C$. Roughly speaking, a code is called *efficient* if the rate m/n is close to C. Shannon's proof also showed that a block code is possible at any rate $m/n<C$, provided long enough codewords are used. We remark that practical applications usually require code mappings involving shorter codewords. Section 5 discusses techniques which have been developed to construct practical code mappings, including codes with rate *equal* to the capacity C when C is a rational fraction.

One can employ a finite-state transition diagram (FSTD) in order to conveniently represent the infinitude of binary strings satisfying the (d,k) constraint. This graph representation for constrained channel strings dates back to Shannon's seminal paper [2], and it was exploited by Franaszek in his work on RLL codes. Figure 3 shows an FSTD for the (1,3) constraint. It consists of a graph with 4 nodes, called states, and directed edges between them, called state transitions, represented by arrows. The edges are labeled with channel bits. Paths through the graph correspond precisely to the binary strings satisfying the (1,3) constraint. A similar FSTD having k+1 states can be used to describe any (d,k) constraint, as the reader can easily verify.

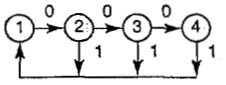


Fig. 3. (1,3) FSTD

(d,k)	Capacity C	Practical Rate
(0,1)	0.6942	1/2, 2/3
(1,3)	0.5515	1/2
(1,7)	0.679	2/3
(2,7)	0.5172	1/2

Fig. 4. Code Capacities

The capacity C of the RLL (d,k) constrained channel is directly related to the structure of the FSTD. We define the state-transition matrix $T = (t_{ij})$ associated to the (d,k) FSTD with states 1,..., k+1 as follows:

$$t_{ij} = x, \text{ if there are x edges from state } i \text{ to state } j. \quad = 0, \text{ otherwise.} \tag{1}$$

For example, for the (d,k)=(1,3) case:

$$T = \begin{matrix} 0\,1\,0\,0 \\ 1\,0\,1\,0 \\ 1\,0\,0\,1 \\ 1\,0\,0\,0 \end{matrix}$$

The capacity C, in units of user bits per channel bit, was shown by Shannon to be:

$$C = \log_2 \lambda$$

where λ is the largest positive eigenvalue of T, that is the largest root of f(t), the characteristic polynomial of T. For the (d,k) constraint, the polynomial f(t) is $f(t)=t^{k+1}-t^{k-d}-\ldots-t-1$. The roots can be easily found by computer calculation. In the (1,3) case, we find $\lambda=1.465..$, so $C=0.5515...$

In practice, one chooses for the rate a rational number $m/n \leq C$. To help keep the codeword length small, the integers m and n are often selected to be small. Thus, for the (1,3) constraint, it would be natural to look for a code mapping at rate 1/2, which uses codewords of length 2 bits. Figure 4 gives a list of some (d,k) constraints of historical and current interest, along with Shannon capacity C, and choices of practical code rates. We note here that the rate 1/2 for (d,k)=(0,1) is included for historical reasons which will be clarified in section

For a given user bit frequency, or data rate, the choice of (d,k) constraint and code rate determines the code bit frequency and the power spectrum of the write current signals produced by the code. Figure 5 shows the power spectra of write current signals that would be obtained from a (d,k) code that is 100% efficient, for the (d,k) parameters in Figure 4.

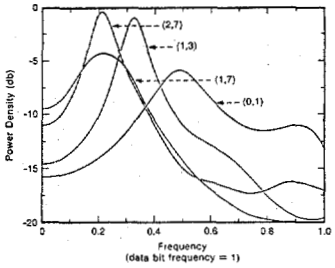


Fig. 5. RLL code spectra

4. CODE PERFORMANCE EVALUATION AND SELECTION

We are now in a position to evaluate, to some extent, the relative abilities of RLL (d,k) codes to prevent detection errors in peak detection channels.

One can get a clue as to the best (d,k) code to use at a specified data rate with a particular head and disk combination by seeing which code spectrum in Figure 5 "matches" the head/disk transfer function best. This comparison can be strengthened by taking a closer look at code patterns and their impact on the peak detection process. To start, we do a comparison which will explain the prevalence of MFM today. When peak detection was introduced along with a variable-frequency oscillator (VFO) clocking circuit, a coding technique was needed to give very frequent clock recovery information to the VFO. It is known as Frequency Modulation (FM), or Double Frequency, and is related to Phase Encoding and Manchester code. The code was at rate 1/2, and the code mapping rules were simple. To encode, write 2 bits for each data bit, the first always being "1," the second being the data bit. Decoding involves simply grouping the detected bits into pairs, and dropping the first bit of each pair. Using RLL code nomenclature, FM is a rate 1/2 (0,1) RLL code. This is the reason that rate 1/2 for (d,k)=(0,1) was included in Figure 4.

It should be mentioned that FM is also a *DC-balanced* code, meaning that the associated write signals have zero average power at DC. In fact, at any point in time, the number of preceding clock intervals in which the signal is positive differs from the number in which it is negative by at most 1. With this additional constraint, the FM code is 100% efficient. This DC-balanced property is not required, however, for most magnetic disk storage applications.

In FM, the k=1 parameter made accurate clock synchronization possible. Figure 6 shows uncoded data and rate 1/2 (0,1) bit patterns which correspond to the same linear user bit density. The minimum allowable magnetic transition spacing is indicated by the solid bar at left above each pattern. The detection window in which the peak must be sensed is indicated by the dotted bar at right above each pattern. We see that the FM code cuts the size of the detection window in half, and simultaneously reduces the minimum transition spacing by a factor of two. The drastic increase in intersymbol interference certainly negated to some extent the potential benefits obtained by the introduction of peak sensing and the VFO.

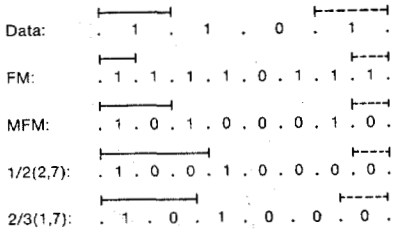


Fig. 6. RLL code patterns

Modified Frequency Modulation (MFM) was developed, as the name suggests, to improve upon the FM recording code. The observation was made that not all of the redundant bits need to be "1". Inserting a "1" in the first position of each

for clock recovery, and the d=1 represented a doubling of the minimum transition spacing compared to FM. Figure 6 shows the comparison of "worst case" rate 1/2 (0,1) and rate 1/2 (1,3) bit patterns. At a specified linear user bit density, one would anticipate considerable improvement in detector performance from MFM. MFM became an industry standard in flexible and "Winchester"-technology drives.

More recently, other RLL (d,k) constraints have been introduced. For example, ISS used a rate 1/2 (2,11) code called 3PM [3] in its 8434 disk drive and a rate 2/3 (1,7) code in its 8470 drive [4]. Also, IBM utilized a rate 1/2 (2,7) code [5] in its 3370-3380 family of high-end drives.

Figure 6 gives heuristic justification of the use of these codes instead of MFM. In comparing 1/2 (2,7) to MFM, at fixed linear density, we see that the "worst case" intersymbol interference for the (2,7) code gives a minimum transition spacing which is 50% larger than that of MFM. The detection window size remains unchanged. Provided that the VFO can handle a k=7 parameter, we conclude that use of the (2,7) code leads to a considerable improvement in detector performance. Similarly, in comparing the 2/3 (1,7) code to MFM, we find that the (1,7) has 33% larger minimum transition spacing, as well as a 33% larger detection window. Again, we can conclude that the error rate can be reduced by use of (1,7) instead of MFM.

The code comparison method breaks down when we try to choose between the 1/2 (2,7) and the 2/3 (1,7) codes. The 2/3(1,7) code has a 33% larger detection window than the 1/2(2,7), but the minimum transition spacing is 11% less. The optimal choice depends on the specific signal and noise characteristics of the head-disk combination and other channel components, as well as additional signal processing options such as readback equalization and write precompensation. More sophisticated performance evaluation tools are required, and several studies have been devoted to the development of such tools and their application to code selection. See, for example, Huber [6]. Using the peak detection channel model reported in [7], error-rate curves were computed for FM, MFM, and (2,7) codes to illustrate the progress achieved via improved coding. These calculations assumed conventional thin-film head, particulate medium with no significant disk defects, low-pass filtering, and no write precompensation. The results are shown in Figure 7. In addition, Figure 7 shows a projected range of improved performance that can be achieved by use of (2,7) or (1,7) codes in conjunction with write precompensation and readback equalization. These projections are consistent with simulated and experimental results given in several published studies. For example, Jacoby and Kost [4] reported that a data rate increase of 10% was achieved in an equalized channel by use of a rate 2/3 (1,7) code in place of a rate 1/2 (2,7) code.

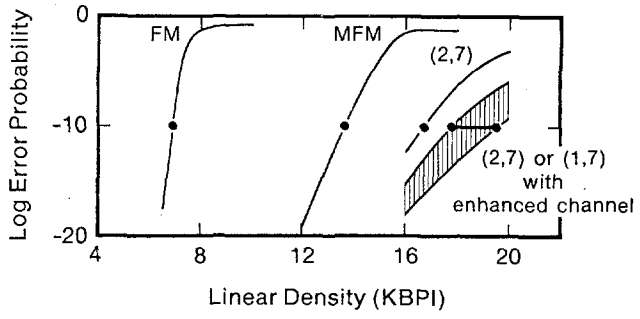


Fig. 7. RLL code performance calculation

These calculations indicate that recording code progress since 1966 has increased linear density by a factor of approximately 2.5. This represents a significant contribution to the overall factor of increase due to improvements in storage technology, for example, as reported for IBM disk drives in [8].

5. RECORDING CODE CONSTRUCTION AND IMPLEMENTATION

Once the optimal code parameters are selected, based on modeling or experimentation, it is necessary to devise encoding and decoding rules for an efficient code which can be implemented in simple logic circuits or look-up tables. This section addresses the problem of code construction and implementation.

To illustrate some of the techniques and algorithms that [illegible] code. Then, we will examine some of the methods developed to design (d,k) codes, emphasizing (2,7) and (1,7) codes. The sequence state coding methods of Franaszek [1] and the look-ahead techniques of Patel [9], Jacoby [3], Franaszek [10], Cohn and Jacoby [11], Jacoby and Kost [4], and Lempel and Cohn [12] will be discussed.

Finally, we focus on the recent sliding block code construction algorithm of Adler, Coppersmith, Hassner [13], based on work of Marcus [14]. This technique, derived from the branch of mathematics known as symbolic dynamics, represents a theoretical breakthrough in code construction, with significant practical implications. For the first time, the algorithm provides an explicit recipe, justified by mathematical proofs, for construction of simple, efficient RLL codes with limited error propagation. The method incorporates many of the key ideas which appear in the work of Franaszek, Patel, Jacoby, Cohn and Lempel, generalizing them and making precise the construction steps and the scope of their applicability.

Properties of Practical Code Mappings

We now discuss some properties that practical code mappings possess, using MFM to illustrate them. The essential properties are:

1) high efficiency,
2) simple encoder and decoder implementations, and
3) limited error propagation.

The MFM code, as defined earlier, is a rate 1/2 (1,3) code. It has *high efficiency*, 0.5/0.5515, or approximately 91%. The encoder is characterized by two encoding rules. The first rule, which we call rule A, is used to encode a data bit if the previous data bit was "0". The second rule B, is used if the previous bit was "1". Both rules take the form of a block code, mapping 1 bit to 2 bits. The MFM encoder can be represented in table form, as shown in Figure 8.

State Data	A	B	Code	Data
0	10/A	00/A	N0	0
1	01/B	01/B	N1	1

Fig. 8. Encoder and decoder tables for MFM

The columns labeled A and B describe the block codes for the two rules. Each entry in these columns also indicates the next encoder state, that is, the rule which is to be used to encode the next bit.

The MFM encoder is an example of a *finite-state machine* (FSM), with fixed length inputs and outputs. A FSM is simply a set of encoding rules which indicate how to map input words to output words, and also define which encoding rule to use after each encoding operation. A FSM also has a graphical representation which will play an important role in the later discussion of code construction techniques. The nodes (states) in the graph correspond to columns in the encoder table, and the graph edges have labels "x/y" where x is an input word, and y is the corresponding codeword. The labeled edges emanating from a state define the encoding rule, and the state in which an edge terminates indicates the state, or encoding rule, to use next. The FSM graph for MFM is shown in Figure 9.

The FSM encoder structure is particularly convenient because it is quite easily implemented as a sequential logic network or ROM-based look-up table. Each encoder cycle encodes a single data bit into 2 code bits, which are a function of the data bit and the contents of a 1-bit state indicator register, and then updates the state indicator.

Decoding is accomplished by grouping the code string into 2-bit blocks and dropping the redundant first bit in each block. The decoding table is shown in Figure 8 also. The symbol "N" represents a "not care" bit position in which the actual bit value does not affect decoding. For example, both "10" and "00" decode to "0". This block decoder ensures that error propagation is no more than 1 user bit.

The MFM decoder is an example of a *sliding block decoder*, which decodes a codeword by looking at a "window" containing the codeword and a finite number of preceeding and following codewords in the string. In the MFM case, the window only contains the codeword being decoded. One can think of such a decoder in terms of a sliding window that shifts along the codestring one codeword at a time, producing at each shift a decoded data word, depending only on the window contents. This structure ensures *finite error propagation*, since an erroneous codeword can only affect the decoding decisions

decoding cycle shifts 2 code bits into the register and generates a decoded data bit.

Fig. 9. MFM encoder

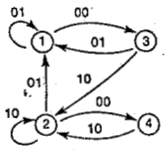


Fig. 10. G^2 for (1,3)

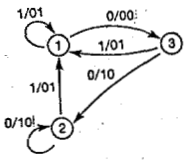


Fig. 11. Subgraph H

Sequence State Coding

Franaszek [1] introduced sequence state methods into the investigation of efficient constrained code construction. The essential idea is to build a FSM encoder in which the states (encoding rules) correspond to the states in a FSTD representation of the constraints. The codewords produced by a particular encoding rule in the FSM must be generated by following paths through the FSTD which start at the corresponding state.

We illustrate the sequence state methods by rederiving the FSM encoder for the MFM code. We begin with the FSTD for the (1,3) constraint, shown in Figure 3. Denote this graph by G. We want the FSM encoding rules to map 1 bit to 2 bits, so an edge in the FSM graph should have a codeword label which is 2 bits long. Therefore, an edge in the FSM graph must correspond to a sequence of 2 edges in the FSTD, each of which had a 1 bit codeword label. With this in mind, we now define the graph called the square of the FSTD G, and denoted G^2, as a candidate for the underlying graph of the FSM encoder. This is the FSTD which has the same states as G, but in which each edge corresponds to a consecutive pair of edges in G, and the edge label is the corresponding sequence of 2 edge labels from G. The FSTD G^2 for this case is shown in Figure 10. Note that if each state in G^2 had at least 2 edges emanating from it, we could construct an FSM encoder by simply assigning the input words "0" and "1" to two distinct edges from each state, discarding any excess edges.

Here, though, G^2 does not have this property. This can be seen from Figure 10. It can also be seen by noting that the matrix T^2, the matrix square of the state-transition matrix T for FSTD G, is the state-transition matrix for the FSTD G^2. This matrix is:

$$T^2 = \begin{matrix} 1\,0\,1\,0 \\ 1\,1\,0\,1 \\ 1\,1\,0\,0 \\ 0\,1\,0\,0 \end{matrix}$$

The entry in (row i, column j) can be seen to count the total number of edges from state i to state j in G^2. The *row sum* for each row must be at least 2 if each state has at least 2 outgoing edges. Clearly the row sum for row 4 is only 1.

However, we can observe that there is a subgraph H of G^2 which does have the desired number of outgoing edges from each state. Namely, we eliminate state 4 and all edges which enter or leave it. Alternatively, we note that the submatrix obtained from T^2 by eliminating the last row and column has row sums exactly 2. Input labels can now be assigned to the edges in the subgraph H to yield a FSM encoder structure, shown in Figure 11. States 2 and 3 have the property that their outgoing edges are identical in terms of output labels and next states. In other words, they describe the same encoding rule. They may be merged, therefore, into a single aggregate state, 2'. The outgoing edges from 2' are those common to states 2 and 3, while the incoming edges are simply redirected to 2' from states 2 and 3. If we relabel state 1 as state B and state 2' as state A, the resulting FSM is exactly the FSM shown in Figure 9.

For the RLL (2,7) constraint, the Shannon capacity is given by C=0.5172... If we try to construct a FSM encoder which encodes 1 bit into 2 bits, as we did with MFM above, we find that there is no subgraph of the square of the standard (2,7) FSTD with at least 2 outgoing edges from each state. We know from Shannon's result that a code is possible at rate 1/2, so one strategy is to take higher powers of the graph, say G^{2m}, and try to derive from it a FSM encoder in which one encodes m input bits at a time into 2m code bits. A computer calculation of powers of the matrix T shows that the smallest value of m for which this is possible is m=17. In the resulting FSM encoder, each state would require 2^{17} outgoing

Data	Code
10	0100
11	1000
000	000100
010	100100
011	001000
0010	00100100
0011	00001000

Fig. 12. Variable-length (2,7) code

In general, this technique can be used to construct efficient (d,k) codes, with finite error propagation. The drawback, as seen in the (2,7) example, is that the encoder and decoder implementations can still be very complex, involving long codewords and potentially large error propagation.

Franaszek [5] found a considerably simpler code mapping by utilizing FSM structures with variable-length input words and codewords. Space limitations preclude a detailed discussion of the technique, but we will describe the 1/2 (2,7) code he constructed, as well as properties of the variable-length FSM codes generated by the method.

The variable-length FSM encoder for the 1/2 (2,7) code reduces to a single state, yielding a variable-length block code as shown in Figure 12. The codewords form a *prefix-free list*, that is, no word is a prefix of another. This ensures that any concatenation of codewords has a unique decomposition into a string of codewords. In addition, Franaszek selected the codewords so that the patterns "1000" and "0100" delimit codeword boundaries. These two properties ensure decodability with finite error propagation. The corresponding data word list is also prefix-free, and additionally has the property of being *full*. This means that every semi-infinite binary data string has a unique decomposition into input words, guaranteeing encodability of data. It is interesting to note that the actual implementation of a (2,7) encoder based on this variable-length code involved the translation of this code description into an equivalent fixed-length FSM encoder mapping 1 data bit to 2 code bits during each encoding operation. The number of states used required a 4-bit state indicator memory. The decoder implementation took the form of a sliding block decoder, decoding 2 code bits into 1 data bit during each decoding operation. The sliding window was a shift register of 8 code bits. The error propagation was therefore no more than 4 data bits. [15].

In general, the variable-length sequence state coding method produces efficient (d,k) codes with rate $m/n<C$ and finite error propagation. The variable-length FSM encoder can be translated into a fixed-length FSM structure with an associated sliding block decoder, just as was done for the (2,7) code, by introducing additional states. The codeword length in such a fixed-length FSM description can still grow very large, although this did not occur in the (2,7) case. The codeword length will typically be smaller than would be obtained by using fixed-length sequence state methods, however.

Franaszek also constructed a 2/3 (1,8) variable length block code. Other applications of variable length coding techniques to RLL codes may be found in [16], where a rate 2/3, (1,7) variable-length code having 2 states is presented.

Look-Ahead Coding Techniques

Another class of techniques found in the work of several authors, including Patel [9], Jacoby [3], Franaszek [10], Cohn and Jacoby [11], Lempel and Cohn [12], and Jacoby and Kost [4], is called future-dependent or look-ahead (LA) coding, and has been used to produce several practical, efficient codes.

One objective of LA codes, as described in [12], is to overcome the codeword length restrictions encountered in the sequence state coding methods. For any code rate $m/n \leq C$, one would like a fixed-length FSM encoder which maps m bits to n bits, and a decoder with finite error propagation. The tradeoffs between codeword length, implementation complexity, and maximum error propagation could then be better evaluated.

The idea is to design an encoder (often with structure similar to a FSM derived from the graph G^n, where G is the FSTD for the constraints) in which the encoding rules allow several alternative encodings for given input data words. The alternative chosen to encode the input word depends on "look-ahead" at a finite number of upcoming input words.

There are several important examples of practical look-ahead codes. Jacoby [3] developed a rate 3/6 (2,11) code called 3PM, which was later modified into a rate 3/6 (2,7)

published a very elegant rate 2/3 RLL (1,7) code, invented by Cohn, Jacoby, and Bates [18].

To illustrate the look-ahead encoding technique, we describe the Cohn, Jacoby, and Bates code in more detail. The encoder takes the form of two tables, shown in Figure 13. The Basic Encoding table provides a mapping from 2-bit input words to 3-bit codewords which satisfy the (1,7) constraint. If this table was used for encoding, however, certain sequences of 2-bit input patterns would produce code strings which violate the (1,7) constraint. For example, 00.00 would encode to 101.101. Exactly four such violation patterns occur. To handle these, a Violation Substitution table provides alternative encodings that should be used when a violation pattern is detected by look-ahead at the upcoming 2-bit input word. The decoder has a maximum error propagation of 5 data bits.

— Basic encoding table

Data	Code
0 0	1 0 1
0 1	1 0 0
1 0	0 0 1
1 1	0 1 0

— Violation substitution table

Data	Code
0 0•0 0	1 0 1•0 0 0
0 0•0 1	1 0 0•0 0 0
1 0•0 0	0 0 1•0 0 0
1 0•0 1	0 1 0•0 0 0

Fig. 13. Look-ahead (1,7) code

Just as in the case of the variable-length codes described above, sequential implementations of these codes can be obtained by the use of an equivalent fixed-length FSM encoder and a sliding block decoder. For example, a FSM implementation of the encoder for the Cohn, Jacoby, and Bates (1,7) code is shown in Figure 14 [18].

State / Data	A	B	C	D	V
00	101/V	100/A	001/V	010/A	000/A
01	100/V	100/B	010/V	010/B	000/B
10	101/C	100/C	001/C	010/C	000/C
11	101/D	100/D	001/D	010/D	000/D

Fig. 14. (1,7) code FSM

Several key ideas appear in the context of look-ahead coding. First, there is the introduction of an *approximate eigenvector inequality* to guide the code construction [12]. If we are interested in a code at rate m/n, then an approximate eigenvector v is a non-negative integer vector satisfying the inequality:

$$T^n v \geq 2^m v$$

where, as before, T is the state-transition matrix for the RLL constraint. The existence of such an eigenvector is guaranteed by the Perron-Frobenius theory of non-negative matrices [19].

This vector can be used to guide the assignment of input words to edges in G^n. If the matrix T^n does not have a submatrix with row sums at least 2^m, then some component of v will be larger than 1 and encoder look-ahead is required. The amount of look-ahead required to encode the input label of an edge is related to the eigenvector component of the state in which the edge terminates. This input assignment process is illustrated in [12]. The literature on the subject describes the construction technique primarily by means of examples. A systematic procedure for designing look-ahead (d,k) codes with guaranteed finite error propagation has not yet been published.

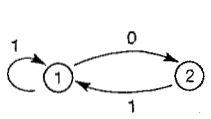
Patel [9] also utilized a look-ahead approach in his design of the Zero-Modulation (ZM) code, a DC-free code with RLL (1,3) constraints and Shannon capacity C=1/2. Patel introduced another very important idea in his construction. He modified the FSTD G^2 by *splitting and merging some of the states* to obtain a representation of the constraint by a new FSTD

Sliding Block Code Algorithm

The sliding block code algorithm of Adler, Coppersmith, and Hassner [13], based on work of Marcus [14], used techniques developed independently from recording technology in the branch of abstract mathematics known as symbolic dynamics. The application of these ideas to the construction of recording codes was inspired by Patel's paper on ZM [9]. The ideas of "approximate eigenvector" and "state-splitting and state-merging," seen in look-ahead coding examples, play an important role in the algorithm. Instead of using the approximate eigenvector v to guide the assignment of m-bit input words to the edges of the FSTD G^n to enable look-ahead encoding at rate m/n, the algorithm uses the vector v to guide the construction of a new FSTD that underlies a FSM encoder. The eigenvector component for each state s in G^n is called its *weight*, v_s. The algorithm produces the new FSTD, which we call H, by iteratively splitting each state s into v_s states and specifying labeled state transitions among the new collection of states. The state-splitting procedure ensures that the resulting FSTD H generates the same set of constrained code strings as G^n, but, in addition, each state has at least 2^m outgoing edges. The assignment of distinct m-bit input words to 2^m outgoing edges at each state of H then provides a FSM encoder which maps m-bit input words to n-bit codewords. Moreover, the state-splitting algorithm in [13] guarantees that the decoder for this code will have a finite sliding block structure, ensuring limited error propagation. We will give an example shortly, but the reader is encouraged to see [13] for details.

The key advance achieved by the algorithm is that it is systematic and is supported by a rigorous mathematical proof. The procedure successfully constructs a code at any rate m/n $\leq$ C, where C is the Shannon capacity of the RLL constraint. Note especially that the construction of codes with rate m/n=C is equally well handled by the algorithm. Computer programs have been written which apply the algorithm to automatically generate practical, efficient RLL (d,k) codes.

To illustrate the ideas, we construct a rate 2/3, RLL (0,1) sliding block code. The FSTD, called G, and its third power G^3 are shown in Figures 15 and 16.



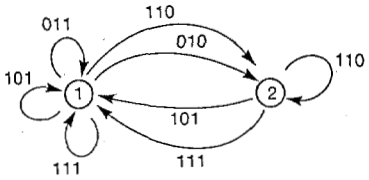


Fig. 15. (0,1) FSTD G

Fig. 16. G^3 for (0,1)

An eigenvector inequality is given by:

$$T^3 v = \begin{bmatrix} 3 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} \geq 2^2 \begin{bmatrix} 2 \\ 1 \end{bmatrix} = 2^2 v .$$

This approximate eigenvector v=(2 1) indicates that state 1 will be split into 2 states, while state 2 will not be split. The two states into which state 1 is split are called 1^1 and 1^2. The outgoing edges of state 1 are partitioned into two groups which are assigned to the two "offspring" states. In addition, all edges which entered state 1 are redirected to both offspring states in the split FSTD. The splitting rule requires that the sum of the weights of the terminal states of edges in a group must be an integer multiple of the approximate eigenvalue, 2^2, with the possible exception of one group; we split the edges into groups [011,110,010] and [101,111], both of which have total weight exactly 4. Note that there can be more than one choice of edge partitioning which satisfies the splitting rule. The resulting FSTD, called H, is shown in Figure 17. It generates the same set of strings as G^3, but has at least 4 outgoing edges from each state. By discarding the loop at state 2, we can label the edges at each state with 2-bit input words to get a FSM encoder, for example, as shown in Figure 18. States 1^2 and 2 represent the same encoding rule, so they can be merged, as was done in the case of the MFM construction, to further simplify the FSM encoder to include only two states. The algorithm guarantees that, regardless of the input word assignment and subsequent state-mergings, the decoder will

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.