Optimal Decoding of Run-Length Limited Codes in a Magnetic Channel

by

Yuns Oh

A thesis
presented to the University of Manitoba
in partial fulfillment of the
requirements for the degree of
Master of Science
in
Electrical Engineering

Winnipeg, Manitoba, 1987

ISBN 0-315-44128-3

OPTIMAL DECODING OF RUN-LENGTH LIMITED CODES IN A MAGNETIC CHANNEL

BY

YUNS OH

A thesis submitted to the Faculty of Graduate Studies of
the University of Manitoba in partial fulfillment of the requirements
of the degree of

MASTER OF SCIENCE

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Yuns Oh

I further authorize the University of Manitoba to reproduce this thesis by photocopying or other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Yuns Oh

Abstract

Many run-length-limited (RLL) codes have been developed recently in order to increase data density on magnetic recording channels. The channel itself represents a further encoding process. Considering this encoding process and incorporating it with the RLL encoding process channel into a single trellis code allows optimal decoding of the received signal. This process is carried out for four popular RLL codes. An analytical estimate of the bit error probability of the decoder is developed and confirmed by simulation of the four codes. In these instances, proper consideration of the channel results in a coding of gain at least 3dB.

Acknowledgements

The author would like to thank his advisor, Prof. E. Shwedyk, for his invaluable assistance and guidance over the course of this work.

The contributions of several associates, including Paul Shufflebotham, Gilbert Soloudre, James White and James Schellenberg are recognized, both in scholarly and in other matters. Life cannot be spent in front of a computer terminal, no matter how tempting it seems. The author would also like to thank the "boys" of the VLSI room, especially Peter Hortensius and Roland Schneider, who often times prevented the author's premature burial by a computer. Finally, S. Song's error control capabilities in his secondary language proved to be quite useful.

The financial support of NSERC is also gratefully acknowledged.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Magnetic data storage devices have represented the most important permanent storage system for computers for about as long as transistors have been the most important switching element. The hysteresis of an appropriately chosen magnetic medium causes a remnant magnetic field to remain in the material after an applied field is removed. This remnant field creates current pulses in a conducting coil that is subsequently moved across the field. New data can be written at the same spot by once again applying an external magnetic field. This combination of indefinite data retention without power, and fast writing and re-writing of data have made magnetic devices the de-facto choice for permanent (or semi-permanent) storage of binary data.

This should not imply that magnetic media is a panacea for all long-term storage problems; it too suffers from limitations. Among them are the problems of data density and timing. The computer industry's insatiable demand for greater capabilities, spurred on by the almost exponential growth in the capabilities of fabricated components, has pressured disk and tape drive manufacturers to increase the storage capacities of these devices.

Saturation recording requires a complete reversal in the magnetic field in the material to represent a "1" in a bit interval. Obviously, the sharper the transition in the field, the smaller the bit interval can be, and the more information that can be packed into a given area. However, the transition cannot be made infinitely sharp. A point is reached where the magnetic field generated by the transition is strong enough to change the magnetic orientation of the material (the same as the write process mechanism), and at this point, the transition cannot be made any sharper. Another problem is

related to timing. It is very important to know where the bit interval boundaries are. However, the pattern recorded on the magnetic medium contains no inherent information about this. The location of a bit interval can be inferred from the position of a transition (which should occur in the center of an interval), but since a transition only occurs when there is a "1" in the data stream, a long sequence of "0"'s will deprive the receiver of this information. A clock can be used to extrapolate the bit intervals, but this requires very accurate clocks and precise control of the movement of the medium relative to the head, and even then, it may not be enough.

A special class of codes called run-length limited (RLL) codes can provide effective solutions to these problems. These codes are usually specified by the parameters (d,k,m,n). An information or data sequence is transformed into a codeword sequence at the rate $(R = \frac{m}{n})$ of n codeword bits for every m data bits. The codeword is constrained so that there are a minimum of d and a maximum of k "0"'s between any consecutive "1"'s in the codeword sequence. The bit interval clock must now extrapolate over a maximum of $k+1$ intervals, and the magnetic transition can now extend over more than one bit interval. These two properties can alleviate the timing problems and increase bit density by over 50% at very little implementation cost. It is emphasized that these codes have been developed specifically for the magnetic channel, and not for error-correcting purposes. Any error-correction encoding is performed by a separate encoder that transforms the input data before being processed by the RLL code.

1.1. Motivation

This thesis is not about how to increase bit density, or about new RLL codes, per se. Much significant work has already been published in these regards, and a good understanding of theoretical limitations and practical implementations of RLL codes

has been achieved. Rather, consideration is given in these pages to the neglected topic of decoding existing RLL codes. Certainly, methods exist to decode present RLL codes, but they are heuristically based (following the heuristic nature of the code design procedure) and are not necessarily optimal. Also, present decoding methods for RLL codes ignore the redundant ternary nature of the output of the magnetic channel.

The question arises as to why this topic should be studied. The most obvious reason is that optimal decoding can reduce the errors associated with magnetic data storage, which is always desirable. By improving error performance, it may also be possible to degrade some part of the system (for instance, the track width) to improve the data density, while maintaining the error rate at its previous level. Related to this point is a less obvious but perhaps more important reason for studying optimal decoding. Many magnetic data storage devices use some form of error control (detection or correction) code in order to provide appropriate error performance in the system. This decreases data density since space for redundancy bits must be allocated.

Now, convolutional codes (although not presently used in magnetic data storage devices) are a form of error control code. They derive their error control properties from the relationship each codeword symbol has with those codeword symbols around it. The codewords of RLL codes also exhibit this relationship, although in a somewhat weaker and less structured form. Present decoding methods throw away, or do not properly exploit this interdependency. By properly considering the relationship, RLL codes may exhibit some error control properties. If this is the case, then the additional error control code can be discarded, improving data density.

There are two major considerations in achieving optimal decoding. The first is the question of optimal decoding of the RLL code itself. There is no all-encompassing mathematical description for these codes as there is for linear block and convolutional codes so a systematic method is, for now, ruled out. Also, the characterisation of the

coding process by the developers of the codes is unsuitable for application to optimal decoding.

The second consideration is the channel itself. That it is not a typical communication channel is highlighted by the fact that while binary information is transmitted to it, the output is ternary. This is not to imply that the channel is not being used to its full capacity (it is), but that the physical process as the signal is passed from the write-head to the magnetic medium and finally to the reading head changes the output into a ternary form while adding redundancy. Present-day receivers choose to ignore this process and simply convert the output back into binary form. An optimal decoding technique must exploit the redundancy of the output and cannot discard any important information.

So the observed problem is that while RLL codes have become an important component of magnetic data storage, optimal decoding of these codes has not ensued. The error performance of these codes could be improved by optimal decoding, and also by incorporating the channel effects as part of this decoding. An analysis of the error performance with optimal decoding may reveal some error correcting properties inherent to the RLL codes.

1.2. Direction

The starting point is to define "optimal" decoding. Then a method of implementing this decoding must be found. Finally, its performance must be analyzed and the improvement, if any, realized.

The magnetic channel is generally involved in the transmission of very long message sequences, so a rational measure of performance would be to minimize the average bit error rate. This can be calculated by dividing the number of bits in error in a sequence by the total number of bits in the sequence for very long sequences, and is

termed bit error rate (BER). It is equivalent to the probability of a bit picked at random from the sequence being in error and is denoted P_{be}.

However, strict BER minimization is a somewhat intangible goal. Satisfactory methods of guaranteeing this are not known for any type of continuous code (such as convolutional codes, as opposed to block codes). A more plausible decoding method is maximum-likelihood sequence detection (MLSD), which is optimal in the sense that the most likely transmitted sequence is found. Additionally, in almost all cases, the chosen sequence minimizes BER as well. This technique is very well defined (by Viterbi's algorithm) for any process that can be modelled as a finite state machine, and under the hypothesis that RLL codes can be represented as such, this appears to be a good avenue.

With the procedure for optimal decoding defined, its performance in terms of BER must be analysed. By analysing the performance rather than simply simulating it, the error mechanisms can be observed and any error control properties identified.

Viterbi's algorithm requires that the process be modelled as a finite state machine (a finite Markov chain). However, the BER analysis requires a stricter characterization of the process. The process must be an irreducible aperiodic finite Markov chain, which has steady-state state probabilities, and these steady-state state probabilities must be found.

The BER analysis must then be performed on some RLL codes that are representative of RLL codes in general. Simulation of the BER should also be performed to verify the analysis. The codes can then be investigated for any error control capabilities that they may possess.

1.3. Contributions

The contributions of this thesis are both in the concrete results presented here, and the avenues that are opened towards further research.

This work represents the first time that a procedure for optimal decoding of RLL codes has been specified. Obviously, it also represents the first time the performance of such decoding has been analysed and simulated.

The effects of the channel are also included in the RLL encoding process and optimal decoding extended to include these effects. Although the channel has previously been optimally considered [1,2], the codes considered with the channel were very simple non-RLL codes that resulted only in changes to the output assignments of the channel state diagram. This is the first time the magnetic channel has been included in an RLL code (or even any code of this complexity).

The analysis shows that $d_{\min}$ for the codes with the channel increases to two, thrusting RLL codes into the class of error detection (though not error correction) codes. By confirming the possibility, it opens the door to the creation of RLL codes with more powerful error control capabilities, without any data density penalty, by appropriately structuring the state diagram of the code and assigning the outputs.

1.4. Outline

The first step is to gain a thorough understanding of the many processes involved. Since this work is about magnetic storage devices and RLL codes, the understanding should begin there. Chapter 2 describes the process of data storage on magnetic medium via saturation recording. The process is broken down into three major blocks; the channel itself, the channel encoder and decoder, and the error control encoder and decoder. The transformation of the input along with the redundancy introduced by the magnetic channel is considered. Both channel and error control codes are described.

Finally, improvements garnered by RLL codes (which belong to the channel code category) are presented and justified.

Chapter 3 goes on to develop MLSD and its implementation in a channel with additive white Gaussian noise (AWGN). An algorithm developed by Viterbi is shown to be equivalent to MLSD, and to be realiseable for real-time (or for that matter any) decoding. This is a very important advancement as MLSD in its basic form is totally impractical. The techniques presented here are those that are used to optimally decode the RLL codes in the magnetic channel. They also provide the basis for the error analysis.

Four popular RLL codes are described in Chapter 4, and they represent most of the actual implementations of RLL codes. The analysis of Chapter 5 is applied to these codes. The coding processes are described as finite state machines, as is the channel. The channel and code is combined into a single model that paves the way for optimal decoding. Each process is also characterized as an irreducible, aperiodic Markov chain, allowing long-term state probabilities to be found. This will be useful in later BER analysis.

An analytic treatment of the BER for the codes is developed in Chapter 5. The resulting expression is preferred over simple simulation of the error rate since it is more general and provides more insight into the error mechanisms. The analysis is actually valid for any code that can be represented as an irreducible aperiodic Markov chain. The analysis is performed by computer for each of the four RLL codes, both with and without the channel. The analysis results are confirmed through computer simulation. They show a significant 3dB coding improvement when the magnetic channel is included in the model. They also show that the RLL codes together with the magnetic channel possess error detection capabilities.

Chapter 6 concludes the study and suggests further areas to consider. Among them are the possible development of better codes, a more systematic treatment of these codes, and the inclusion of intersymbol interference in the models.

Appendices A and B present more information on optimal decoding (in a single symbol interval) and Markov chains. These areas of study are well established, but are significantly related to work presented in this thesis. Appendix C contains a listing of the computer program that analyzed each of the codes, while Appendix D is a listing of the simulation program.

1.5. Prerequisites

In order for the reader to understand the material presented hereinafter, some background knowledge is required. An understanding of several fields of mathematics including vector and metric spaces, group theory, probability, statistics, and graph theory is essential. A rudimentary understanding of digital communications is also required.

It is helpful, but not necessary, for the reader to be familiar with electromagnetics, digital logic, coding theory, and last but not least, magnetic data storage devices.

Chapter 2

The Magnetic Channel and RLL Codes

The process of recording and reading on magnetic medium can be viewed as sending information through a channel, albeit differing somewhat from a typical communication channel. Magnetic saturation recording has some inherent properties that must always be considered and can sometimes be exploited as will be shown in later chapters. In this chapter, the physical process of saturation recording will be described along with models for its behaviour and coding techniques that can improve its performance in several areas.

2.1. Magnetic Data Storage

The process of storing information in a magnetic medium and subsequently retrieving it can be modelled as a communication channel with five basic components, as shown in Figure 2.1. The channel itself represents the actual physical processes associated with writing data to and reading from the magnetic medium. The properties of this process have created problems for designers and, as a result, a class of codes peculiar to magnetic data storage have been developed specifically to deal with these properties. This coding process is represented by the channel encoder and decoder in Figure 2.1. One problem the magnetic channel shares with all other communication channels is that of errors in the transmitted data. To combat these errors, standard error-correcting codes are employed as represented by the error-correcting encoder and decoder blocks in Figure 2.1. These range from simple parity check codes to more complex block (or cyclic block) codes such as the Reed-Solomon codes.

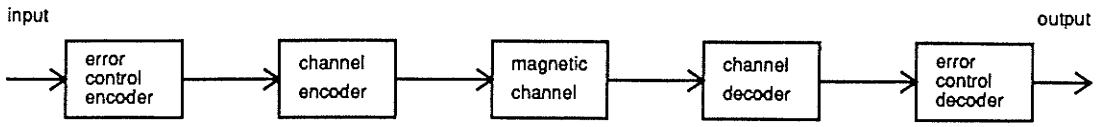


Figure 2.1: Magnetic data storage channel.

2.2. Magnetic Recording

The magnetic medium itself is comprised of a thin layer of microscopic particles, equivalent to electrical dipoles [3, 4]. This layer is bonded onto a substrate that is rigid or flexible according to the application. The medium is moved past a recording head consisting of a conducting coil wrapped around a high permeability construction, as diagrammed in Figure 2.2. When an electric current is passed through the coil a magnetic field is generated in the head with the field lines perpendicular to the current flow. The field lines are confined to the ferric head until the gap in the head is reached, whereupon they extend out from the head and into the magnetic medium The magnetic field causes the dipoles to align themselves parallel to the field lines and the particles remain in this orientation in the absence of the magnetic field. If the magnetic field intensity (i.e. current) is strong enough, all the particles will be aligned and the medium is saturated. This alignment of the particles generates a remnant magnetic field in the magnetic medium. This behaviour is due to the hysteresis of the remnant magnetic field under an applied magnetic field in the medium as shown in Figure 2.3.

When a reading head of similar construction to the recording head is passed over the medium, a field is generated in the ferric head. Any change in this field generates a current in the conduction coil around the reading head. In saturation recording, the particles are aligned along one dimension (horizontally or vertically) due to the characteristics of the head and/or the medium (anisotropic medium), thus the dipoles can have one of two orientations. Whenever the orientation is reversed (due to a reversal in the recording head current), a current pulse is generated in the reading head as it

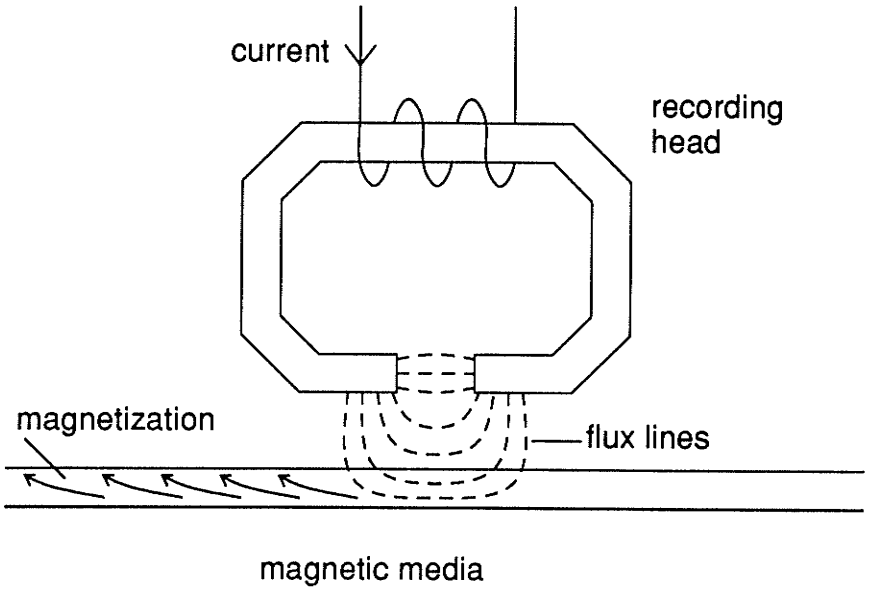


Figure 2.2: The write process.

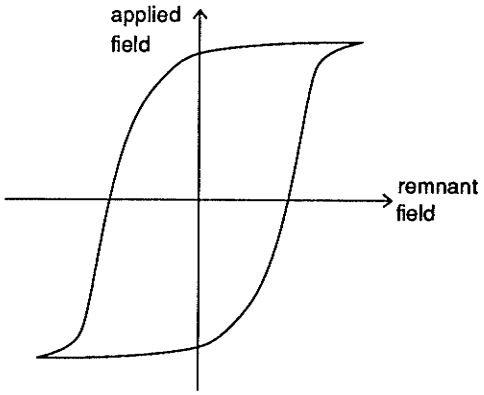


Figure 2.3: Hysteresis.

moves by. The direction of the current depends on the particular flux reversal. Thus the output of the reading head can be a positive, negative or no current pulse.

Considering the recording head, it is noted that the current flow in the coil can be in one of two states; that which is sufficient to saturate the medium in one orientation, or an opposite current flow of the same magnitude such that the medium is saturated in the opposite orientation. Any lower level of current flow will be insufficient to

saturate the medium and an absence of current will leave the medium in its previous state (an undesirable situation). Thus the recording process is naturally binary which complements the binary nature of computer data storage requirements. However, the previous paragraph outlined the ternary nature of the output of the reading head. Concluding that entropy [5] is being added to the channel must be cautioned against since there is some redundancy in the output. After a positive pulse, only a negative pulse or no pulse can follow. This is because a positive pulse implies that a certain type of flux reversal has taken place on the medium, say from dipole orientation A to orientation B. After this event, the dipoles can either remain in orientation B (no pulse) or switch into orientation A resulting in an opposite flux reversal to that which previously occured (negative pulse). Similarly, a negative pulse can only be followed by no pulse or a positive pulse.

2.3. Channel Codes

In order to minimize the problems associated with the channel, the magnetization pattern actually recorded on the medium must be carefully chosen. Towards this end, a number of coding schemes have been developed and are classified as channel codes (more specifically, magnetic channel codes). They transform the input binary data sequence into a current waveform that is applied to the recording head. The waveform for several codes for a sample input is shown in Figure 2.4.

Return-to-zero is a code that was developed in the infancy of magnetic data storage. The magnetic medium is magnetized in one orientation, and a "1" is represented by a reversal in the orientation for half of a bit interval. A "0" indicates no change in the magnetized pattern on the magnetic medium. Two pulses in a bit interval at the output of the reading head indicate a "1", while no output is decoded as a "0". This code suffers from the problem that a long sequence of zeroes result in no

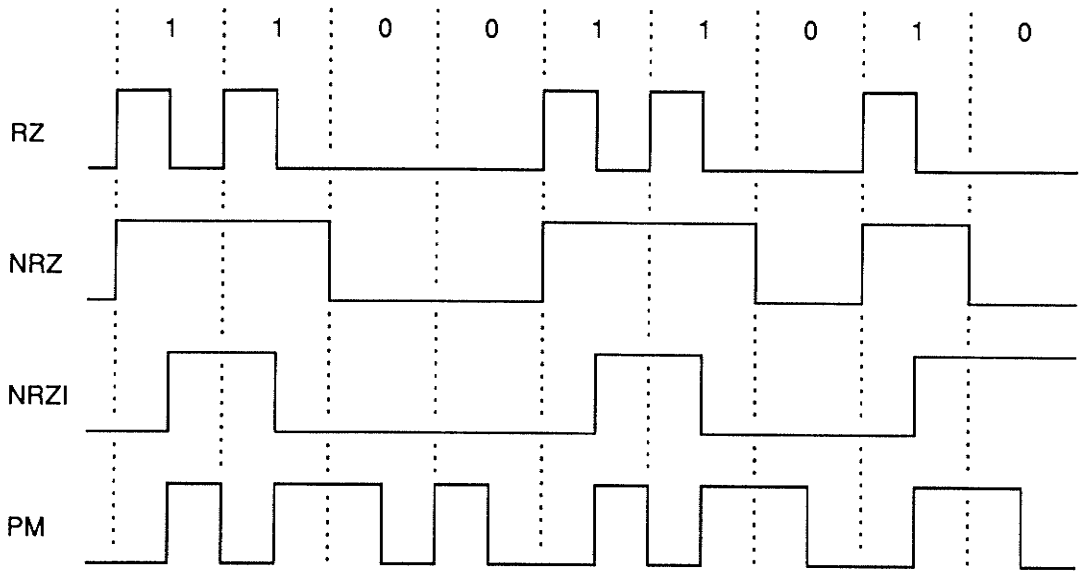


Figure 2.4: Channel codes.

output (at the reading head) at all. This necessitates a very accurate clock, and head motion (relative to the medium) in order to keep track of the bit intervals. Also, when a "1" is input, two transitions are placed very close to each other (inside a single bit interval). In order to compensate for this, the bit intervals must be made larger reducing the data capacity of the medium.

Since the recording head is intrinsically binary, an obvious method of encoding binary data is to assign each symbol to one of the recording head's states. In this way a "1" would be represented by a positive current while a "0" would be represented by a negative current in the recording head. At the output of the reading head, a pulse would denote that the symbol is different than the previous one, while no pulse would denote no change in the output symbol. This method of encoding and decoding is called non-return to zero (NRZ) and is trivial to implement. However it suffers from error propagation since an error in decoding one bit will result in errors in all the following bits until another error occurs [3]. This property alone makes NRZ unacceptable for modern high-density magnetic storage.

A more common method is termed non-return to zero inverse (NRZI). In this code, a "natural" mapping is made from the ternary output of the reading head to the binary symbols. Any pulse (positive or negative) is decoded as a "1" while the absence of a pulse is decoded as a "0". In the recording process, a "1" is represented by the reversal of the head current (i.e. a dipole orientation or flux reversal), while a "0" maintains the previous head current (no change in the orientation). Like NRZ, this method is quite simple to implement. Although NRZI does not suffer from error propagation, it suffers the same timing problems as RZ.

Phase modulation (PM), which is also know as Manchester coding, transforms a "0" into a particular dipole orientation reversal in the magnetic medium (say from orientation A to B), while a "1" is transformed into the opposite reversal (B to A). Thus a "0" represents a reversal in the recording head current from positive to negative, while a "1" represents a negative to positive reversal. Notice that this may necessitate a reversal at the bit interval boundary to accommodate the required reversal in the middle of the bit interval. As a result, the data density suffers. Decoding consists of ignoring any pulses at the bit interval boundaries and detecting the polarity of the pulse in the middle of the interval. If the pulse is positive a "1" is chosen, and if it is negative a "0" is chosen. Thus PM does not suffer from error propagation, and since transitions are separated by no more that one bit interval, timing is not a problem.

2.4. Error Control Codes

Error control codes transform a sequence of information or data symbols into a sequence of codeword symbols with structured redundancy. This redundancy allows the codeword sequence to be decoded into the correct data sequence even when the codeword sequence has been corrupted with errors. There are two fundamental categories of error control codes: block codes and tree codes. Block codes segment

the input sequence into blocks of m symbols and map each block into a codeword of n symbols. Tree codes carry out the same process except that the codeword chosen is also a function of the previous v input blocks. The mapping of the input to the codeword is generally linear since the study of this class of codes is much more mature. For tree codes, if some time invariance properties are added, the result is a class of codes called convolutional codes.

A brief description of the two classes of codes follows. A more complete and formal treatment can be found in [6, 7, 8, 5].

2.4.1. Block Codes

A block code is often described in terms of a matrix. The information block of m bits and the codeword of n bits can be described as row vectors I and C respectively.

$$I = [i_1\ i_2\ \cdots\ i_m]$$

$$C = [c_1\ c_2\ \cdots\ c_n]$$

The code itself is described by a generator matrix G.

$$G = \begin{bmatrix} g_{11} & g_{12} & \cdots & g_{1n} \\ g_{21} & g_{22} & \cdots & g_{2n} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ g_{m1} & g_{m2} & \cdots & g_{mn} \end{bmatrix}$$

Then

$$C = IG$$

If there are q (input and output) symbols, then all operations are carried are carried out in $GF(q)$, that is, a Galois (or finite) field with q elements (note that for some q,

$GF(q)$ may not exist). For a linear block code, the sum of any two codewords is a codeword, and the product of any codeword with an element of $GF(q)$ is also a codeword.

The Hamming distance between any two (information or code) words is the number of positions in which they differ. Block codes derive their error control properties from a parameter $d_{\min}$ which is the minimum Hamming distance between any two valid codewords. In order to illustrate this, consider a case where $d_{\min} = k$. If the codeword C that is transmitted becomes corrupted so that up to $k-1$ symbols are incorrectly received, then the received word R will not match any valid codeword (otherwise, $d_{\min}$ would be less than k). In this way, up to $k-1$ errors per codeword can be detected.

Now consider the codeword C corrupted so that less than $\frac{k}{2}$ symbols are incorrectly received. Again, the received word R will not match any valid codeword, but it will have a smaller Hamming distance to C than to any other valid codeword (otherwise, $d_{\min}$ would again be smaller than k). Thus the code can correct $e < \frac{k}{2}$ errors per codeword.

These error control properties apply to any set of codewords and any mapping of the information words into the codewords (linear or non-linear) as long as the parameter $d_{\min}$ is satisfied. However, imposing certain restrictions allows $d_{\min}$ to be optimized (or at least be made quite large) for a given m and n in a systematic way, and simplifies the encoding and decoding process.

The most basic restriction is that of linearity which has already been mentioned. In fact, almost all error control theory applies to linear codes. Hamming codes fall into this basic category and can be constructed for $m = 2^x - 1 - x$ and $n = 2^x - 1$ for any integer x. For this code, $d_{\min}$ is 3 so that any single error can be corrected or up to

two errors detected. Although not very powerful, the rate $R = \frac{m}{n}$ is quite high.

A somewhat more versatile code is the Reed-Muller code specified for any positive integers x and y such that $y < x$. The rate of this code is given by $m = 1 + \binom{x}{1} + \cdots + \binom{x}{y}$ and $n = 2^x$. Reed-Muller codes have a $d_{\min}$ of 2^{x-y}.

By imposing additional structure, cyclic block codes are reached. In this class of codes, any cyclic shift (a shifting of each symbol by one position) of a codeword is also a codeword. These codes are best described by polynomials, but this will not be covered here.

Fire codes are specified in $GF(q)$ for $m = (q^y-1)(2x-1)-y-2x+1$ and $n = (q^y-1)(2x-1)$ where x and y are integers. Not all choices of x and y will produce a Fire code, but $x \leq y$ is a necessary condition. A Fire code can correct up to x errors provided that the x errors occur in consecutive symbol positions (this is called a burst error).

Bose-Chaudhuri-Hocquenghem (BCH) codes are specified in $GF(q^y)$ with n a factor of q^y-1. A BCH code can correct up to x errors and m will vary according to the choice of x in a manner dependent upon the properties of polynomial rings in $GF(q^y)$.

Reed-Solomon codes are a subclass of BCH codes with $y = 1$. The rate is given by $m = 2x$ and $n = q-1$, and they can correct up to x errors. Reed-Solomon codes have the best possible error correction capability for a given m and n.

Simple, low-capacity disk-drives use extremely simple parity-check error detection (which can be viewed as a simple cyclic block code). Disk and tape drives in larger systems (where data integrity is important) generally use more complex error control schemes, and the Reed-Muller, Fire, and Reed-Solomon codes have all been used in this capacity.

2.4.2. Convolutional Codes

Like block codes, convolutional codes have the parameters m and n and at each time frame m information symbols are received and n codeword symbols are output. Unlike block codes, convolutional codes have an additional parameter v called the constraint length. The codeword C depends not just on the present information word I, but on the previous $m(v-1)$ information symbols. Thus a codeword at any particular time frame bears a (usually complex) relationship to the v codewords before and after it, and indirectly, to every codeword in the entire message. It is through this relationship that convolutional codes derive their error correction capabilities.

The code itself is described by v generator matrices, G_0 to G_{v-1}, where each G_i is an m by n matrix of elements of $GF(q)$. The codeword is given by

$$C = I_0 G_0 + I_1 G_1 + \cdots + I_{v-1} G_{v-1}$$

where I_0 represents the present information word and I_1 to I_{v-1} represent the previous $v-1$ information words. As with block codes, all operations are carried out in $GF(q)$.

Given a set of generator matrices, a schematic of an encoder can easily be constructed. As an example, Figure 2.5 shows the schematic for a code of rate $R = \frac{m}{n} = \frac{1}{2}$ and $v = 3$ as specified in $GF(2)$ by

$$G_0 = [1\ 1]$$

$$G_1 = [1\ 0]$$

$$G_2 = [1\ 1]$$

In the schematic, the boxes represent delay elements or shift registers and the circles represent modulo-2 adders or exclusive-OR gates.

The relationship between the input and the output can be represented as a tree graph as shown in Figure 2.6. Each path from the root of the tree represents a unique

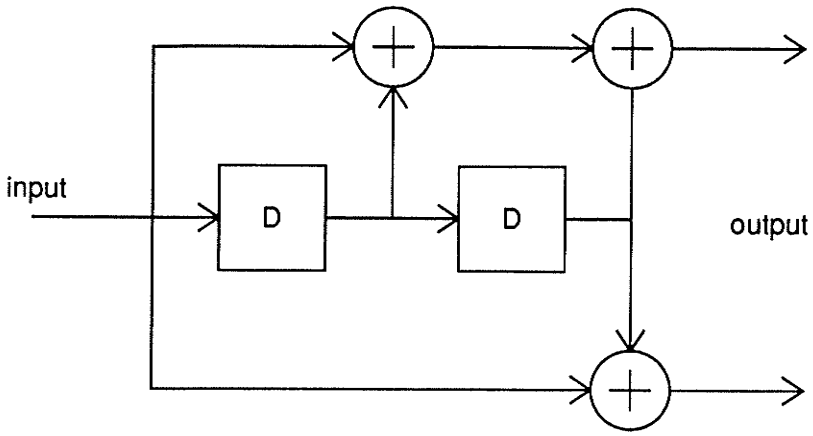


Figure 2.5: Schematic of a convolutional encoder.

input sequence. From each vertex, the upper branch indicates a zero input and the lower branch indicates a one. The codeword is specified beside each edge.

Note that the output patterns along the branches become repetitive after the νth branch. In fact, all the vertices labelled a have identical branches and can be collapsed into a single vertex. Likewise, this is true for the vertices labelled b, c and d. This leads to a new graph called a trellis as shown in Figure 2.7. By relabelling the vertices with symbols from $GF(2)$, the label of a vertex can be made to correspond to the contents of the shift registers in the schematic representation of the encoding process. Each path through the trellis represents a possible information (and codeword) sequence.

By collapsing all the vertices with the same label in the trellis into a single vertex, yet another representation can be reached. This is called a state diagram and is shown in Figure 2.8. All of these diagrams fully represent the encoding process of the convolutional code, differing only in how "time" is represented. A reverse procedure can also be carried out, expanding a state diagram into a trellis.

Although the encoding process of a convolutional code is very simple, the decoding process (at least for optimal decoding) is not. In fact, the only known viable

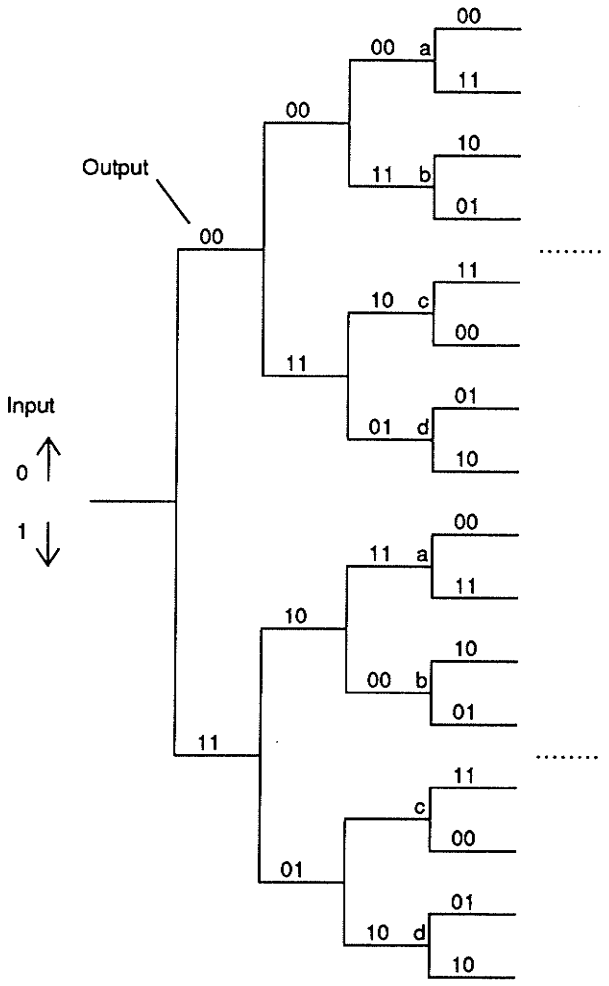


Figure 2.6: Tree graph representation of a convolutional code.

optimal decoding process is Viterbi's algorithm which is explained in Chapter 3. Until then, it is sufficient to note that for a decoding error to occur, a path through the trellis must be chosen that is not identical to the transmitted path. Thus the decoded path must diverge from the transmitted path at some point, and later merge. Over this unmerged segment, the transmitted codeword sequence must be corrupted so that its Hamming distance to the decoded sequence is smaller than its Hamming distance to the transmitted sequence. But inspection of the trellis reveals that the smallest unmerged segment is v edges or time frames long, and this is true for any general v. The important factor is no longer the Hamming distance between single codewords; it

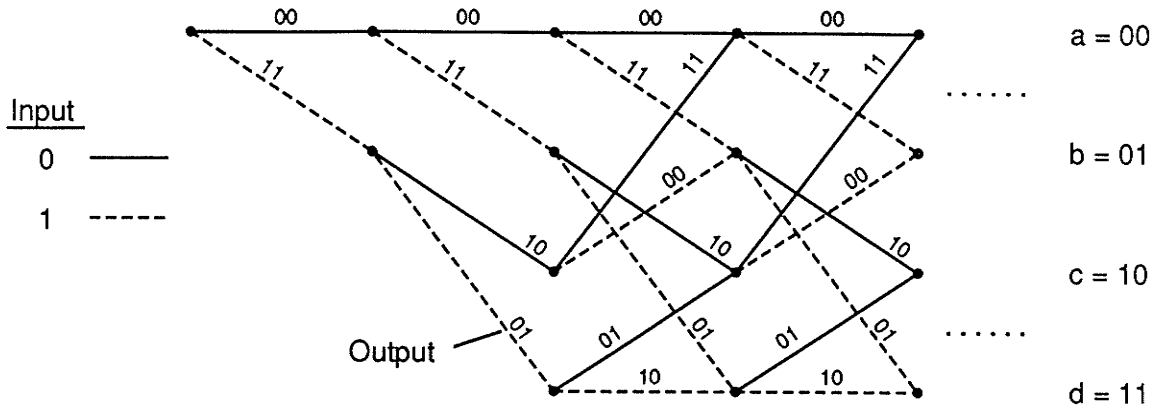


Figure 2.7: Trellis representation of a convolutional code.

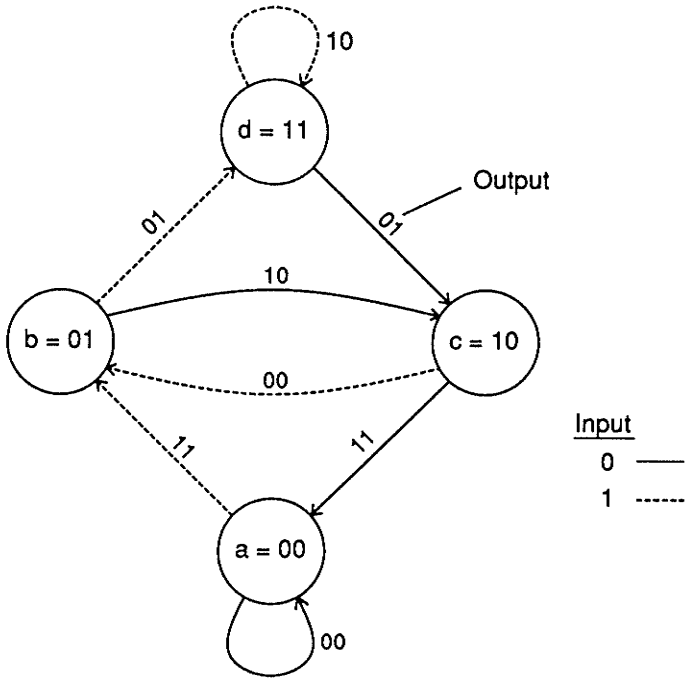


Figure 2.8: State diagram representation of a convolutional code.

is the Hamming distance between specific sequences of ν or more codewords. The smallest such distance is often labelled d_{free}, but in this work it will be labelled $d_{\min}$ to maintain commonality with block codes. Thus a large number of codeword symbols may have to be corrupted before an uncorrectable situation occurs.

The actual nature of the error correcting capabilities of convolutional codes is quite complex (unlike block codes) and beyond the scope of this work. It is clear, however, that careful selection of the generator matrices and v (optimal selection appears to be an NP-complete problem, although it has not been proven) can result in powerful error correction codes, even for high rates. This error correction capability is derived from the structure of the trellis (and suitable codeword assignment) which graphically represents the relationship a codeword has to surrounding codewords.

2.5. High Bit Densities and Other Problems

In the quest for ever higher information density on par with developments in other computer equipment, designers have tried to make the bit interval (that is, the space on the medium that is allocated to a single input bit) as small as possible. Unfortunately, there is a limit as to how far this practice can be taken as displayed in Figure 2.9, especially for horizontally oriented medium. When there is a reversal in the alignment of the dipoles, there is a boundary where like-poles are adjacent to each other. Strong repulsive forces are generated and this tends to cause a local demagnetization which widens the transition region. Typical attempts to overcome this problem have involved finding materials suitable for the medium with a higher coercivity [9] which allows for sharper transitions (and also necessitates higher recording currents, which is generally not a problem but does illustrate that nothing is free). The recording/reading head, and its interaction with the magnetic medium has also been explored [10, 11, 12]. However, both of these methods are expensive solutions to the problem.

One tempting possibility is to totally ignore the fixed transition width and simply make the bit intervals smaller, accepting the resulting intersymbol interference (ISI) - that is the overlapping of the magnetization pattern in one bit interval into adjacent bit

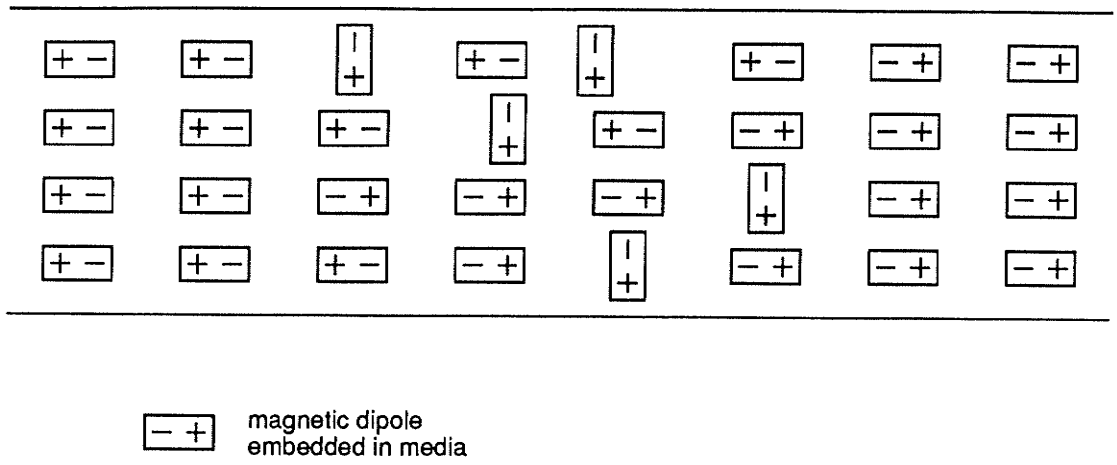


Figure 2.9: The transition region.

intervals. However problems arise in the decoding of the received waveform from the reading head. Present detection schemes use a non-optimal method called peak detection, as it is quite simple and fast. Figure 2.10 outlines the basic procedures involved in peak detection. The signal from the reading head is differentiated and applied to the input of a zero-crossing detector. This output is high whenever the slope of the reading head signal is zero, which occurs when there is no pulse or at the peak of a pulse (in the absence of noise). By applying the reading head signal to a threshold detector set at say half of a pulse peak, and using this signal to gate the output of the zero-crossing detector, a signal with a spike at the peak of each pulse is generated. In NRZI, the existance of this spike denotes a "1" while its absence denotes a "0".

Now, allowing intersymbol interference to occur in the magnetic medium can have significant effects on the performance of peak detection circuitry [13]. Assuming superposition of interfering reading head waveforms (a reasonable assumption at present and future bit densities), examples of two and three adjacent interfering pulses are shown in Figure 2.11. With two adjacent pulses, the peaks of both pulses have moved towards each other. At high bit densities, this may actually move the peak into

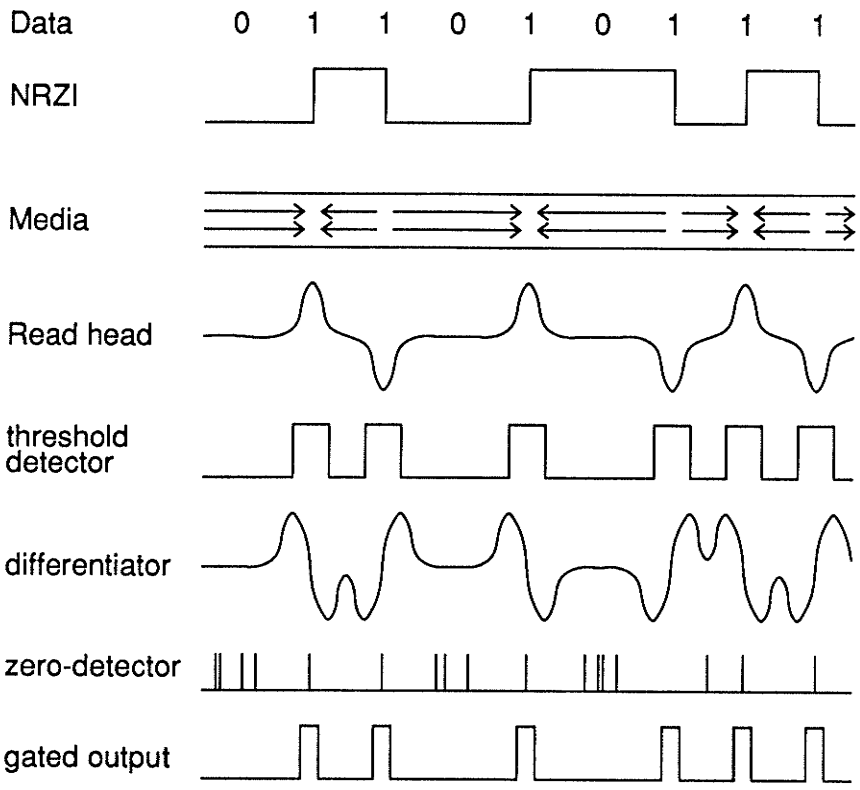


Figure 2.10: Peak detection decoding.

another bit interval, and even at lower densities, it will exacerbate timing problems. With three adjacent pulses, the same problems occur with the outer two pulses, but the centre pulse encounters a more severe problem. It can become so attenuated that it no-longer exceeds the threshold of the gating circuitry, and can be missed entirely. Thus designers of magnetic data storage devices try to avoid any ISI.

Another possibility for NRZI is to constrain the input so that "1"'s cannot occur too closely to each other. In this scenario, ISI would still occur, but not between pulses. Thus the only result would be pulses extending into adjacent bit intervals (devoid of pulses), and no peak-shift or pulse attenuation would occur. This is an excellent situation except for the unfortunate and absurd requirement that the binary input sequence be constrained. This does raise another possibility that perhaps we can map the input sequence onto another sequence of bits that does meet the

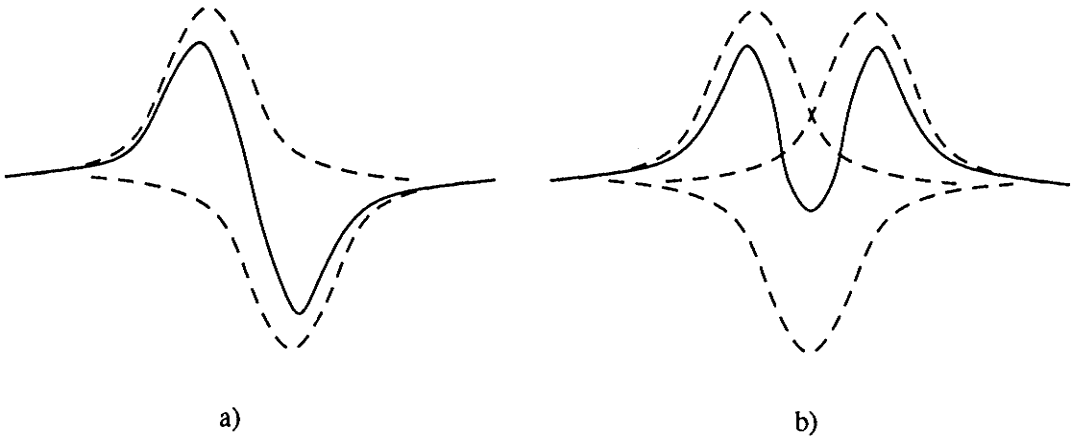


Figure 2.11: a) Two interfering pulses. b) Three interfering pulses.

aforementioned constraints. If the new sequence is not so long as to offset the gain in density made by allowing ISI to occur, then an overall gain in bit density has still occurred. This forms part of the basis of a collection of codes known as run-length-limited (RLL) codes, and is elaborated on in the next section.

The limited transition width is not the only problem inherent in magnetic medium. Saturation recording codes are not inherently self-clocking, as a frequency modulated (FM) signal is, for example. There is no such device as a phase-locked loop that can provide rigid information on the bit intervals. With peak detection circuitry, the only timing information available is the supposed peak of a pulse; there is no timing information available when a pulse does not occur. The best that can be done is to assume that a peak occurs at the centre of the bit interval and use an accurate clock to time the bit intervals until another pulse occurs. Obviously, problems will occur when a pulse does not occur for many intervals (corresponding to a long sequence of "0"'s in NRZI). To overcome this, it would be desirable to constrain the input to the recording head so that "1"'s do not occur too far apart. As mentioned in the previous paragraph, constraining the input sequence in this way is not feasible. However coding the input sequence into another sequence that does meet this

constraint is possible, and forms another part of the basis for RLL codes.

2.6. Run-Length Limited Codes

In the previous section, some desirable constraints on the input to the recording head were discussed. Since coding was purported as a method to solve each of these restrictions, it would be advantageous to find a coding technique that would satisfy all of them at the same time. Such a class of codes has been developed and they are termed run-length limited (RLL) codes [14, 15]. These codes transform an input binary data stream into a binary codeword stream (with more bits) and use NRZI encoding to record the codeword. They restrict the minimum and maximum number of bit intervals between any consecutive "1"'s in order to minimize ISI and timing problems and are necessarily non-linear in nature (observe that an all zero sequence will not produce a like output).

The standard notation for these codes consists of four integer parameters, (d,k,m,n). The minimum and maximum number of zeroes between ones are specified by d and k, respectively. The rate of the code (R) is determined by $R = \frac{m}{n}$; that is, m input bits are mapped into n code bits. The codes are sometimes simply specified by (d,k). Several papers have discussed theoretical aspects of RLL codes and in particular Franaszek [16] found the channel capacities for classes of RLL codes using Shannon's noiseless coding theorems [5]. The graph in Figure 2.12 developed from data provided by Franaszek [16] shows the maximum achievable rate for a given d with the k parameter at infinity; thus $\frac{m}{n}$ cannot be greater than the rate specified by the graph. In practice, it is also desirable to make m and n the smallest integers possible as this simplifies encoder and decoder design. For instance, a (1,7) code can have any rate up to 0.679. However, specifying $m = 679$ and $n = 1000$ would require

extremely complex encoding and decoding circuitry. Previously developed (1,7) codes have set $m = 2$ and $n = 3$ producing a rate very close to the theoretical limit while keeping the complexity at a reasonable level.

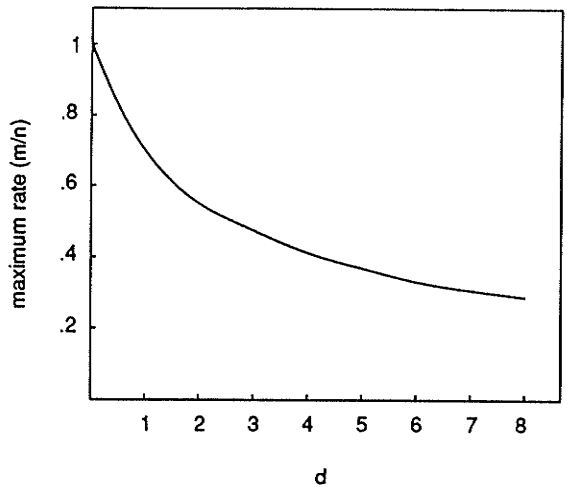


Figure 2.12: Coding rate versus d.

The improvements realiseable with RLL codes can be displayed by considering (1,7,2,3) and (2,7,1,2) codes without (for the moment) even considering the actual code. Both classes of RLL codes have a maximum of seven zeroes between ones, thus accurate timing must be maintained over at most eight bit intervals; quite possible with present technology.

The (1,7,2,3) codes have at least one zero between ones in the codeword. This allows a transition to extend over the bit interval containing the one, plus half of the bit intervals on either side (i.e. two codeword bit intervals in total) without any ISI. Because of this, two codeword bits can be placed where one raw input bit would have been, increasing the bit density by a factor of two. However, the coding process itself occurs at a rate of $\frac{2}{3}$ causing a decrease in the bit density by a factor of $\frac{2}{3}$. Thus the overall gain in bit density is $\frac{4}{3}$ representing a 33% increase. Significantly, this increase has been achieved without changing the hardware (except for adding a simple

encoder and decoder). Since the bit interval has been cut in half, acceptable timing errors are half that of the uncoded data. However this is more than offset by the fact that sequences of zeroes are limited in length.

The (2,7,1,2) codes have a minimum of two zeroes between ones in the code-word, allowing transitions to extend over three bit intervals (a factored gain of three). The coding process has a rate of only $\frac{1}{2}$ so the overall gain in bit density is $\frac{3}{2}$, that is, a 50% increase. The (2,7,1,2) codes are obviously superior in terms of bit density. However their timing requirements are more stringent, since the bit interval has been reduced to one-third that of the uncoded data. Acceptable timing errors are 33% smaller than that of the (1,7,2,3) codes.

The previous analysis hints at a very simple formula based on the (d,k,m,n) parameters to determine any increase in data density, and this is the case.

$$I = \frac{(d+1)m}{n} \tag{2.1}$$

The symbol I represents the increase in input data density relative to recording the uncoded data. In practice, the data density is the overriding parameter as the k constraint has made timing problems relatively small.

Chapter 3

Maximum-Likelihood Decoding and the Viterbi Algorithm

Convolutional codes, and in fact any code that can be represented as a finite state machine (with more than one state) present some unique problems in decoding. Many optimal decoding methods have been developed for classes of codes such as linear block codes. However, the memory inherent in a finite state machine precludes decoding small sections of the codeword independently of the other sections. One possible method would be to consider all possible complete received sequences, and compare them to the actual received sequence. This is called maximum-likelihood sequence detection (MLSD), but it involves unmanageably large amounts of computations for even very small messages. Viterbi developed an algorithm that is equivalent to MLSD, but involves a manageable number of computations, even for real time applications. In this chapter, optimal decoding (in the sense of MLSD) and Viterbi's algorithm are reviewed.

3.1. Finite State Machine Codes

A finite state machine is any process that can be represented by a finite state diagram. Based upon a present state (S_n) and an input, the output and the next state (S_{n+1}) are determined. Each possible input is represented by a path leaving a state, and the destination of that path is the new state corresponding to the present state and input combination. The output is also specified by the path. Convolutional codes are a special case of this and one of the most important constraints is that the process must be linear [6, 7, 8, 5].

In the previous chapter, it is shown that a convolutional code can be represented by a state diagram. This classifies it as a finite state machine process. It is also shown that a convolutional code can be represented by a trellis as well. Likewise, other finite state machine codes can be represented by a trellis which can be obtained from the state diagram. In order to apply Viterbi's algorithm, it is important that the coding process be representable as a trellis, as is shown later in this chapter. In Chapter 4, it is shown that the RLL codes can be represented as finite state machine processes. In a manner similar to that for convolutional codes, a trellis representation can be obtained from the finite state machine representation, allowing Viterbi's algorithm to be used.

3.2. Decoding

The previous chapter shows that the output of a magnetic channel can be a positive or negative pulse, or no pulse. In order to proceed further, a technique must be found to establish which of the three possibilities has occurred. In the presence of noise, the receiver must decide which signal actually occurred in the interval. Such techniques have been developed based on Bayes criterion [17], and are explained in more detail in Appendix A. In this section, it will suffice to note that the signal set has a single orthonormal basis, and it is a normalized transition pulse (either positive or negative). The signal pulses are located at $\pm\sqrt{E}$ and the no-pulse signal is located at origin in the single-dimensional vector space generated by the orthonormal basis. To decode the received signal which is corrupted with additive white Gaussian noise (AWGN), it is multiplied by a noiseless positive transition pulse, and the result is integrated over a bit interval. The output of the integrator represents a point in the vector space. The Euclidean norm is used to measure the distance of the received signal to the three possible transmitted signals. This distance decreases monotonically

with increasing probability of a signal being transmitted, so an optimal decision can be made for the bit interval by choosing the transmitted signal that has the smallest Euclidean distance from the received signal.

3.3. Maximum-Likelihood Sequence Detection

Extending optimal decoding in a single bit interval to optimal decoding over several symbol intervals is relatively straightforward. Since signals that exist at different times are obviously orthogonal to each other, the observations over other bit intervals can simply be considered as a subspace of a larger vector space (in the sense that there are more dimensions). The received signal can now be considered as a vector of length p if p symbol intervals are to be considered. The received signal can be plotted in the p-dimensional space and its Euclidean distance to all possible transmitted signals found. The procedure is an exact analogy to that for a single symbol interval, except that the received signal now has p components.

For block codes, optimal decoding consists of carrying out this procedure over the interval of a codeword. The number of codewords is not unwieldy and the codeword has no relationship to any of the other codewords around it. Finite state machine codes (and, of course, convolutional codes) are not so easily dealt with. Obviously, optimum decisions can be made concerning just a single frame (the output bits associated with a single path or transition from one state to another). But the vagaries of random noise may cause paths in consecutive frames to be chosen so that they do not share a common state. Each path may have been optimally chosen when considered on its own, but the chosen path cannot be considered optimum since the path cannot even have been taken (i.e., the decoded sequence could not even have been transmitted). It is evident that any optimum receiver must be able to consider the relationship that successive frames have.

One obvious way to do this would be to consider the entire message as a single signal. The orthonormal representation for the entire received signal could be found and compared to that of the transmitted signals for all possible transmitted signals. This would, of course, guarantee that the entire message would be optimally decoded, in the maximum-likelihood sense. In fact, this procedure is called maximum-likelihood sequence detection (MLSD) since it finds the path sequence most likely to have occurred, given the received signal. However, as outlined previously, the technique is unrealiseable since it involves an enormous number of computations. Even for an extremely short message of 30 binary information symbols, the received signal must be compared to over a billion possible message sequences. To be practical, a receiver must involve far less processing, but to be optimal in the sense of MLSD, it must always make the same decision as MLSD.

3.4. The Viterbi Algorithm

The coding process of a finite state machine can be modelled as a trellis, as shown previously in Chapter 2 for a convolutional code. Each distinct input (information or message) sequence is represented as a distinct path through the trellis, and the path that has the minimum Euclidean distance from the received signal will be one of them. In Figure 3.1, an arbitrary time or frame interval t is shown, and an arbitrary node is highlighted. The node has a set of paths entering it (E) and a set of paths leaving it (F). The complete set of paths I intersecting the node is found by pairing each and every element of E with each and every element of F; that is, $I = E \times F = \{ (e, f); e \in E, f \in F \}$. The Euclidean distance of an arbitrary path $i \in I$ from the received vector R is given by

$$d_e^2(R, i) = \sum_{x=1}^{L} (r_x - i_x)^2$$

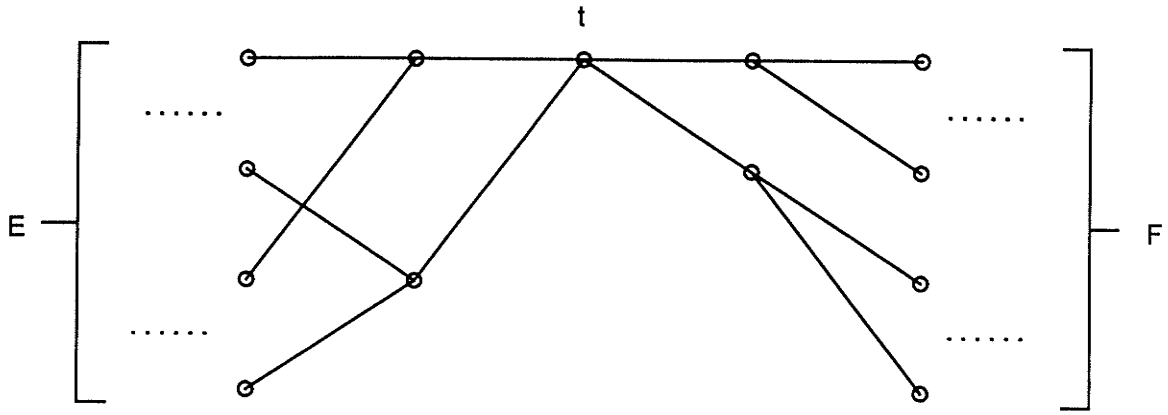


Figure 3.1: Arbitrary part of a set of possible coding sequences.

$$= \sum_{x=1}^{t} (r_x - i_x)^2 + \sum_{x=t+1}^{L} (r_x - i_x)^2$$

The first summation is simply the Euclidean distance of a path in E from the portion of the received vector R_a, up to the node and the second summation is likewise for the path in F from the portion of R, R_b after the node. Thus

$$d_e^2(R,i) = d_e^2(R_a,e) + d_e^2(R_b,f)$$

From the set I of all paths going through a particular node, a subset can be created of all the paths going through the node with a common path after the node. Thus for a particular $f_1 \in F$, $I_{f_1} = \{(e,f_1); e \in E\}$. The distance of any of these paths is given by

$$d_e^2(R,i) = d_e^2(R_a,e) + d_e^2(R_b,f_1)$$

What is the minimum distance path in this set? Since $d_e^2(R_b,f_1)$ is constant for all members of the set, only $d_e^2(R_a,e)$ has to be minimized. In order to find the portion of the minimum distance path up to the node, that portion can be considered independently (from the rest of the path). The distance of each element of E from R_a can be found, and the minimum distance element $e_{\min}$ can be paired with f_1 to find the

minimum distance element of I_{f_1}.

However, f_1 is an arbitrary element of F, so the argument holds true for all subsets I_{f_1}. That is, for all $f_b \in F$, the minimum distance path is $(e_{\min}, f_b)$. All subsets I_{f_b} form a partition of I, so the minimum distance path in I must be one of the minimum distance paths of the partitioning set of subsets. However, all of these minimum distance paths have a common first element, $e_{\min}$. Thus, no matter what the minimum distance path through this node is, it must contain the minimum distance path up to that node. The node used to create I is arbitrary, so the argument holds true for any node in the trellis.

Suppose that the minimum distance paths up to all the nodes at a certain frame in the trellis are known. How can the minimum distance path up to node a in the next frame of the trellis be found? Let node a be as in Figure 3.1. It has i segments entering it, where these i segments originate from nodes $b_1, b_2, \cdots, b_i$. Each of these nodes has a single best path up to that node, and if the overall minimum distance path intersects that node, then it must coincide with the chosen path up to that node. However, this is also true for node a. If the minimum distance paths up to the nodes $b_1, b_2, \cdots, b_i$ are known (as well as their respective distances from the received vector), then there are only i paths to consider when choosing the minimum distance path at node a. These i paths are made up of the best path up to the i nodes, and the segment that connects each of these nodes to node a. The distance of each of these paths from the received vector is simply the sum of the distance up to the segment and the distance of the segment from the received vector. Thus the minimum distance path up to node a can be found by comparing the i paths, and choosing the best one. This procedure can be repeated for each node in the next time frame, and the minimum distance paths up to each node in the next time frame will be known, along with their respective distances from the received vector.

This procedure can be repeated for each frame of the trellis in succession (starting with the first), and for a trellis of a state diagram with s states, there will be s chosen paths at each time frame in the trellis. At the end of the message sequence, the best of the s paths is selected, and this will be the minimum distance path for the entire message. Often, the message will end in a known node, in which case the best path will automatically be known once that node is reached.

This, then is Viterbi's algorithm. It can be proven by induction that it finds the minimum distance path out of all possible paths in an arbitrarily long trellis, by following the previous development [6, 7]. It's primary advantage is that the computational load grows linearly with trellis length, since, at each and every time frame, a fixed number of computations are performed. Initially, it would appear that the best path cannot be chosen until the end of the message (although it will be known immediately after the end of the message as opposed to several billion computations later), however this is not necessarily true. At reasonable signal to noise ratios, the best paths of each node tend to merge into a single path several frames before the latest decision. For convolutional codes, they are almost guaranteed to merge at 4 to 5 times v frames previous to the latest decisions, where v is the constraint length of the code. When all the paths merge, no matter what the final decision on the best path, that path will coincide with any node path up to the point of the merge. Previous work [7] has shown that storing the path history (called the survivor sequence) for this length of time degrades performance by only 0.1dB. Even if the survivor sequences have not merged by this point, a decision is made based on the survivor sequence with the lowest Euclidean distance at that point. In most cases, the slight decrease in performance is more than offset by the ability to produce a decoded sequence in a steady, continuous fashion.

Chapter 4

Four Codes

In this chapter, four popular RLL codes are described and modelled as finite state machines. The four codes are modified frequency modulation (MFM), two codes used by IBM with parameters (2,7,1,2) and (1,7,2,3), and a code used by Santa Clara Systems with parameters (1,7,2,3). The labels MFM (1,3,1,2), IBM (2,7,1,2), IBM (1,7,2,3), and ISS (1,7,2,3) will denote each of the four codes, respectively. The latter three codes are widely used in high capacity disk drives, while MFM is almost universal in smaller drives. These codes are chosen for analysis because they represent most of the actual implementations of RLL codes, and because of the availability of detailed information about the coding process. For the same reasons, these codes are almost always chosen as examples in publications that discuss RLL codes.

4.1. Markov Chains

In order to use Viterbi's algorithm, the coding process must be represented as a state or trellis diagram. The mechanics of Viterbi's algorithm also lead to a better understanding of the error process. This allows the state diagram to be used as a tool in the analysis of the error rates of the codes, which is developed in the next chapter. However, this error analysis (although theoretically applicable to any state diagram) provides useful results for only certain classes of state diagrams.

A Markov chain is a countable state space where the probability of entering a given state depends only on the probabilities of being in each state in the immediately preceding time interval. A coding process that can be represented by a finite-state machine (that is, a finite state or trellis diagram) is a Markov chain. If the property of

time-invariance is added to the coding process and to the probability distribution of the input sequence, then the process becomes a stationary Markov chain. In this case, the probability of a transition between any two given states, i and j, in one time interval is a constant and is simply denoted by p_{ij}. Given a state space with m elements, there are obviously m^2 transition probabilities, and they can be conveniently denoted by a transition probability matrix (or simply transition matrix) T.

$$T = [p_{ij}] = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1m} \\ p_{21} & p_{22} & \cdots & p_{2m} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ p_{m1} & p_{m2} & \cdots & p_{mm} \end{bmatrix}$$

The error analysis in Chapter 5 depends on the probability that the process lies in a specific state at a given time (the state probabilities). Denoting the state probabilities at time n by the column vector $\Pi_n = [P_n(1)\ P_n(2) \cdots P_n(m)]^T$, the state probabilities at time $n+1$ can be found by premultiplying the state probability vector by the transition matrix T.

$$\Pi_{n+1} = T\Pi_n$$

In order for the error analysis to be viable, the state probabilities must converge to a constant vector of steady-state state probabilities, Π, as the time approaches infinity ($n \to \infty$). Since this is not guaranteed to occur for a Markov chain, the conditions under which it does occur must be found. It can be shown (a brief proof can be found in Appendix B, and more detailed proofs in [18, 19]) that Π exists for irreducible aperiodic stationary finite Markov chains. A Markov chain is irreducible if it does not have any set of states in which the process can get "stuck"; that is, to enter said set of states never to leave again. Every state in a Markov chain has a probability of the process leaving that state and returning in a specific number of time intervals. If this

probability is zero in a periodic fashion with respect to the time intervals, then the Markov chain is periodic. An aperiodic Markov chain is one in which no state is periodic.

This topic is already well-studied, so two theorems useful in classifying stationary finite Markov chains will be presented, along with a consequence of the theorems. The reader is again referred to Appendix B or [18, 19] for a more detailed treatment.

Theorem 4.1

If T is the transition matrix for a finite Markov chain, then the multiplicity of the eigenvalue 1 is equal to the number of irreducible closed subsets of the chain.

Theorem 4.2

If T is the transition matrix for an irreducible Markov chain with period d, then the dth roots of unity are eigenvalues of T.

These two theorems allow a stationary finite Markov chain to be classified as irreducible and aperiodic by showing that only one eigenvalue has a magnitude of one. Once a time-invariant finite-state machine code has been appropriately classified by Theorems 4.1 and 4.2, the steady-state state probabilities can be found by solving

$$\Pi = T\Pi$$

under the additional constraint that $\sum_{s=1}^{m} P(s) = 1$. These probabilities form an integral part of the error analysis presented in Chapter 5.

4.2. Modified Frequency Modulation

This early code was developed when the problems previously mentioned with magnetic storage became evident, but the concept of RLL codes was not. Its performance does not quite come up to the standards of the other codes, since it actually provides no increase in data density over NRZI [14, 15]. However, there are a maximum of three bit intervals between transitions, and this alone represents a significant improvement over NRZI. It is also extremely simple to implement (almost as simple as NRZI) and this factor has ensured its popularity in smaller capacity disk drives today.

The coding process itself is described in Table 4.1. The table requires that the previous output bit be known and this suggests that a state diagram representation ought to have two states corresponding to the two possibilities for the previous output bit. The state that each path enters is determined by the last bit of the output corresponding to that path. The state diagram is shown in Figure 4.1(a). A trellis diagram follows naturally from the state diagram and is shown in Figure 4.1(b).

Table 4.1. MFM (1,3,1,3) coding table.

Data	Codeword
0	X0
1	01

X denotes the complement of the previous output bit

In order to include the effects of the channel, the channel itself must be modelled appropriately. The channel was described in Chapter 2, but here, a state diagram representation is desirable. The channel has two restrictions on its ternary output; that any two positive pulses be separated by a negative pulse, and that any two negative pulses be separated by a positive pulse. This can be modelled with two states. One

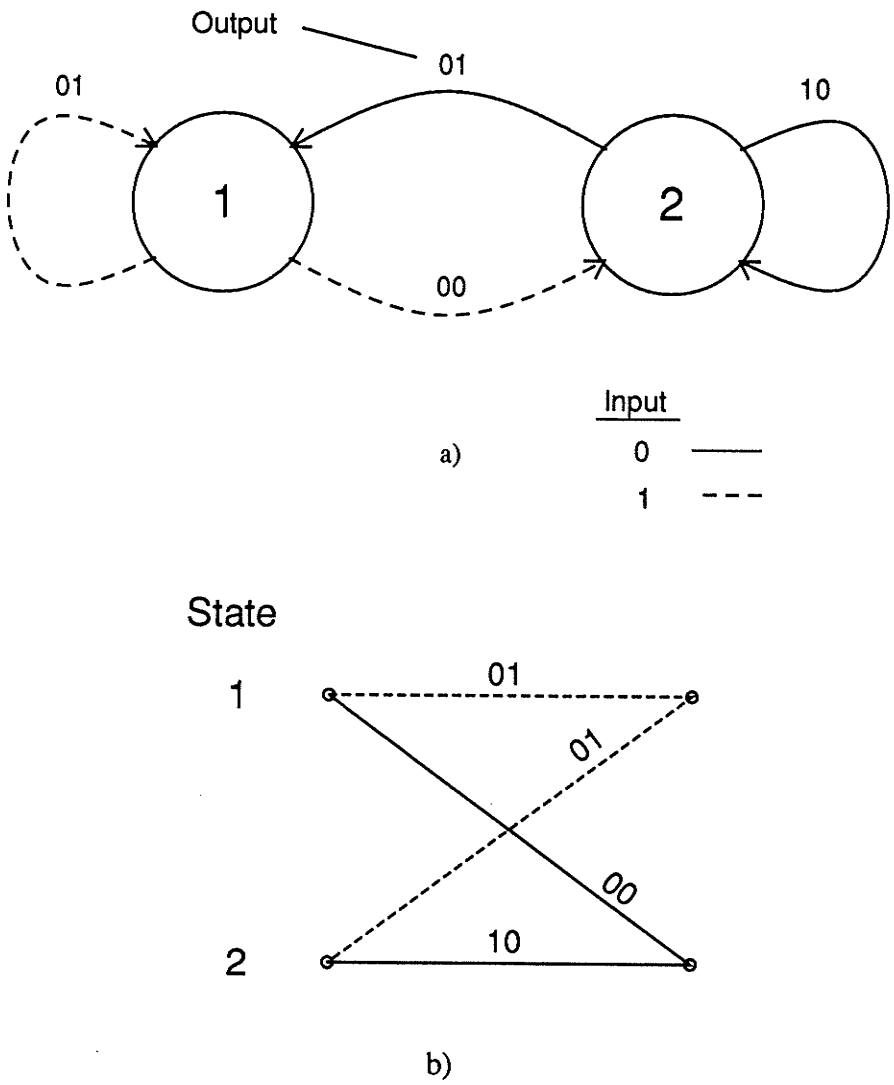


Figure 4.1: a) MFM state diagram. b) MFM trellis.

state (a) signifies that the last pulse to occur was positive and that the next one must be negative, while the other state (b) signifies the reverse situation. If a path leaving one of the states outputs a pulse (of the appropriate type, of course), then that path will enter the other state. If the path outputs no pulse (a "0"), then the state remains unchanged. The state diagram is shown in Figure 4.2, with the output symbol "2" representing a negative pulse.

To combine the state diagrams of the code and the channel, each state in the code state diagram is replaced by the two states of the channel state diagram. The paths

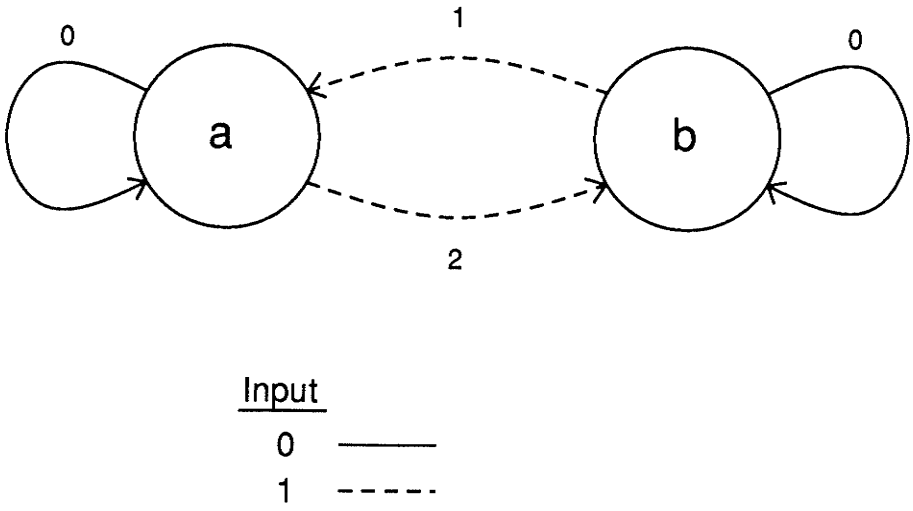


Figure 4.2: State diagram for a magnetic channel.

leaving the original code state are connected to state a of the channel state diagram. If any path contains more than one output symbol "1", then the second and succeeding alternate "1"'s are replaced with the output symbol "2". These paths are replicated for state b, with any output symbol "1"'s being changed to "2"'s, and "2"'s changed to "1"'s. The terminus of the paths are one of the channel states that now make up the path's previous code terminus. The actual state is determined by the final transition to occur on that path.

This process can also be visualized as replacing each state in the state diagram of the channel with the complete state diagram (paths included) of the code. The replacement for state a has any second and succeeding alternate output "1" symbols replaced with the symbol "2". The replacement for state b has the symbols "1" and "2" reversed. In the replacement for a, a connection is preserved if the final transition on that path is positive (i.e., the final non-zero symbol is a "1"), or if that path contains no transitions (all output symbols are "0"). If the final transition is a negative pulse (the final non-zero output symbol is a "2"), then the path is changed so that the terminus is now the equivalent state in the replacement for b. In the replacement for b,

the opposite is true. A connection is preserved if the final transition on that path is negative (i.e., the final non-zero symbol is a "2"), or if that path contains no transitions (all output symbols are "0"). If the final transition is a positive pulse (the final non-zero output symbol is a "1"), then the path is changed so that the terminus is now the equivalent state in the replacement for a. The completed process for MFM is displayed in Figure 4.3.

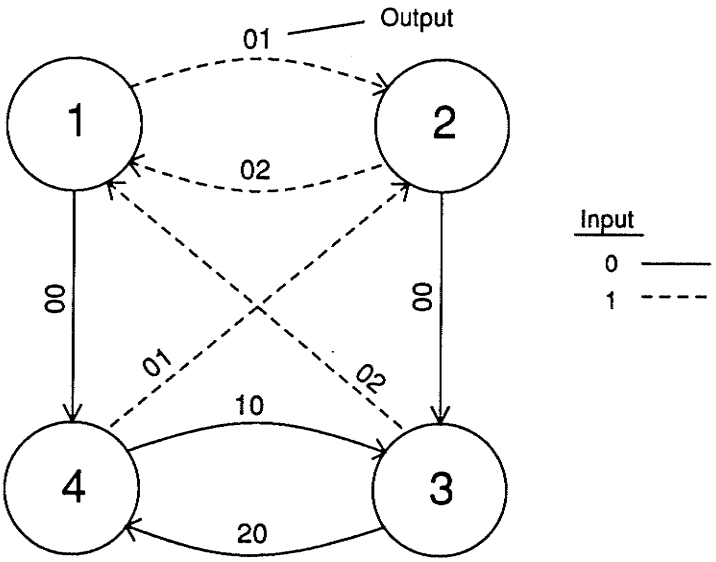


Figure 4.3: State diagram for MFM in a magnetic channel.

The first step in calculating the steady-state state probabilities is to find the transition matrix, which can be obtained directly from the state diagram. Note that, assuming equiprobable input symbols (as always will be the case), all the paths leaving a state are equiprobable, and that probability is $\frac{1}{2^m} = \frac{1}{2}$ for MFM. Since both the binary and ternary case will be analysed, both transition matrices are calculated and denoted by T_{MFMB} and T_{MFMT}, respectively.

$$T_{MFMB} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

$$T_{MFMT} = \begin{bmatrix} 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \end{bmatrix}$$

For T_{MFMB} the eigenvalues are found to be 1 and 0, and it follows from theorems 4.1 and 4.2 that the state diagram is a finite irreducible aperiodic Markov chain. Likewise, the same conclusion can be made for T_{MFMT}.

From the transition matrix, the steady-state state probabilities are found using the procedure described in Section 4.1. Carrying out this procedure, it is found that the state probabilities for MFM are all $\frac{1}{2}$ without the channel, and $\frac{1}{4}$ with the channel.

4.3. IBM (2,7,1,2)

This code was developed by Franaszek in 1972, and implemented by IBM for its larger 3370 and 3380 disk drives. The code allows the capacities of the drives to be increased by 67%, and the rate is very close to the theoretical limit for codes with a (2,7) constraint [16].

The rate of $\frac{1}{2}$ is actually nominal, as the data is actually broken up into blocks of 2, 3 or 4 bits. These are mapped to a set of codewords according to Table 4.2, that are 4, 6, or 8 bits long. This variable rate, as it will be termed, results in the codeword bits being output unevenly and they must be buffered before being written onto the disk surface. Buffering applies during decoding also. While this may be acceptable for IBM's implementation, it is inappropriate for a state diagram representation of the coding process. Such a model must have a single fixed rate for all the paths.

This feature requires that the output be delayed relative to the original coding process. It is desirable to fix the rate at $\frac{1}{2}$ (as opposed to $\frac{2}{4}$, etc) in order to reduce

Table 4.2: IBM (2,7,1,2) Coding Table.

Information Block	Codeword
10	0100
11	1000
000	000100
010	100100
011	001000
0010	00100100
0011	00001000

the complexity of the state diagram. With only one input bit at each frame, each state will have just two paths leaving it. This implies that the output sequence should be delayed by some multiple of 2 bits. Note that redefining the coding process as rate $\frac{1}{2}$ with no delay is not possible as seen by a quick inspection of Table 4.2.

As an initial attempt, consider a state diagram that delays the output by 2 bits. In this situation, the first bit of an information block would output the last two bits of the previous codeword, the second bit of the information block would output the first two bits of the codeword, and so forth. Inspection of the coding table reveals that all the codewords end with "00". Define a state that is entered after the last bit of an information block (which may be the second, third, or fourth bit of that block), so that the input for any path leaving the state is the first bit of an information block. With no delay, the output of each path entering this state would be "00", however with the delay of one frame (two output bits), every path (there are two) leaving the state will output "00". Since the rest of the state diagram is unknown at this point, a new state will be defined at the terminus of each of the two paths leaving the first state. This is shown in Figure 4.4. Each of these two states will also have two paths exitting them (to destinations unknown) and the inputs associated with each path correspond to the

second bit of an information block (so actually the destinations of two of the paths are known; they enter the "end" state...but this is not relevant at this point). The outputs associated with each path are the first two bits of the appropriate codeword. Thus, the first frame (two bits) of each codeword must be completely specified by the first two input bits of an information block. An inspection of the coding table reveals that the data block "010" maps onto the codeword "100100" while "011" maps onto "001000", leading to ambiguity. Knowing only that the first two input bits are "01", it cannot be determined whether the output should be "10" or "00". Obviously, a delay of one frame is insufficient.

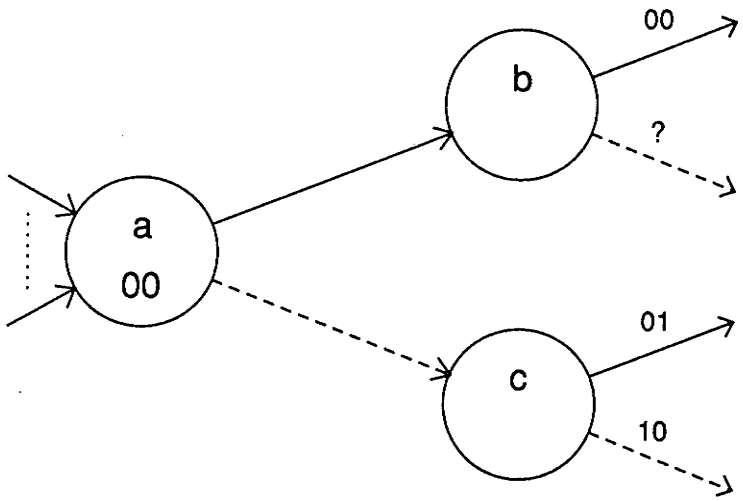


Figure 4.4: Delay of one frame for IBM (2,7,1,2).

Logically, the next step would be to attempt a delay of two frames. Further inspection of the code shows that all codewords end in "0100" or "1000", and they correspond to information blocks that end in "0" and "1" respectively. Two states, a and b, can be defined, so that one or the other is entered after the last bit of an information block. If the last bit is "0" then state a is entered, otherwise, state b is entered. Each state has two paths leaving it, and for now they all terminate in distinct states, c, d, e, f as shown if Figure 4.5. All of the paths (of two segments in length)

leaving state a output "0100" over the two segments, while the paths leaving state b output "1000". Since the last output frame is the same for all the paths, state c is actually equivalent to state e, and d is equivalent to f. These states can be merged, leaving only two states c, and d. The inputs to these paths are the first two bits of an information block. For the input sequence "10" and "11", the second bit also represents the last bit, so these two paths enter state a and b respectively after the second input bit. The resulting state diagram is shown in Figure 4.6.

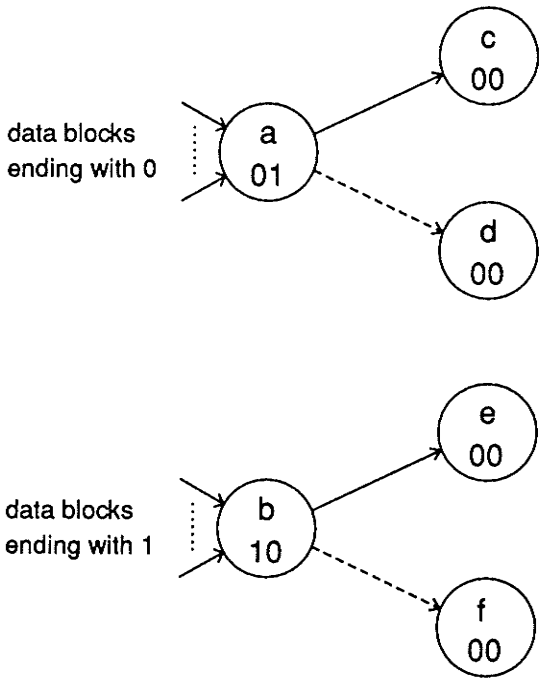


Figure 4.5: Delay of two frames for IBM (2,7,1,2).

The other paths will enter two new states, g and h. The next input bit represents the third bit of an information block, and the first two bits of the codeword must be output at this time. A check of the coding table reveals no ambiguities, thus the state diagram can be continued. In three of the four cases, the third input bit is also the last bit of the information block. These cases correspond to the input sequences "000", "010", and "011". The paths are sent to the appropriate ending state according to the

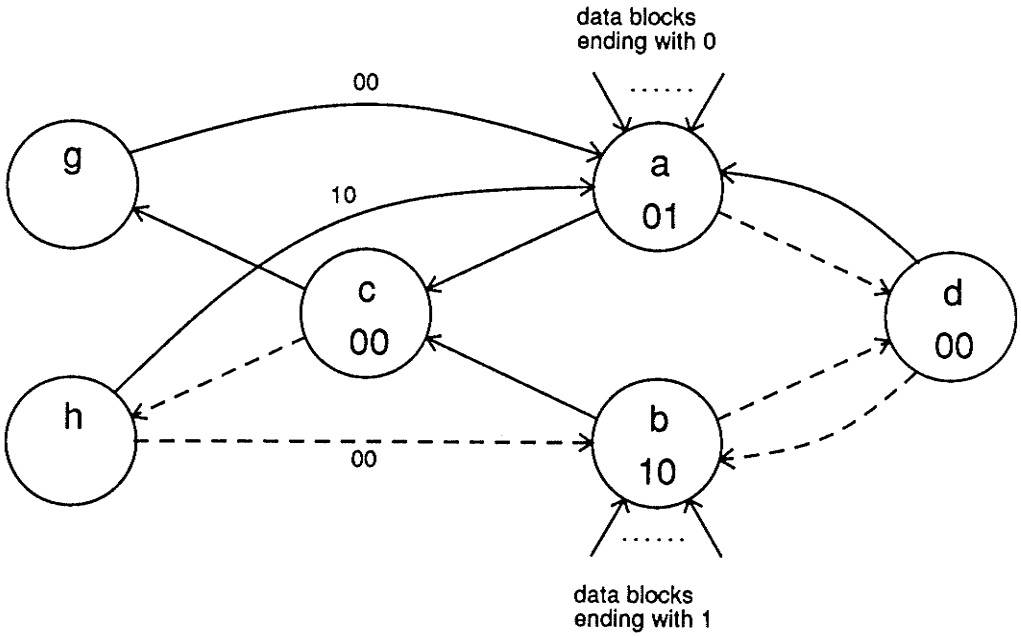


Figure 4.6: Extended state diagram (two-frame delay) for IBM (2,7,1,2).

last four bits of the output; that is, the first two sequences enter state a, while the last sequence enters b.

The exception is the input sequence "001". This is directed towards a new state i. The paths leaving it represent the fourth and last bit of the remaining two information blocks, thus there are no more ambiguities (it was determined earlier that the last bit of an information block is free from ambiguity). The sequence "0010" is directed to state a, while "0011" enters b. This completes the state diagram, and the final form is shown in Figure 4.7.

It is significant to note that the coding process has now been represented by a fixed rate finite state machine (or trellis) code with only seven states. It does have a delay of two frames at the output, but rather than worrying about synchronization with the original code, it can be treated as a separate code with no delay. However, other than the delay, it is entirely equivalent to the original code. Extensive computer simulations verified this model of the coding process. A single stage of the trellis

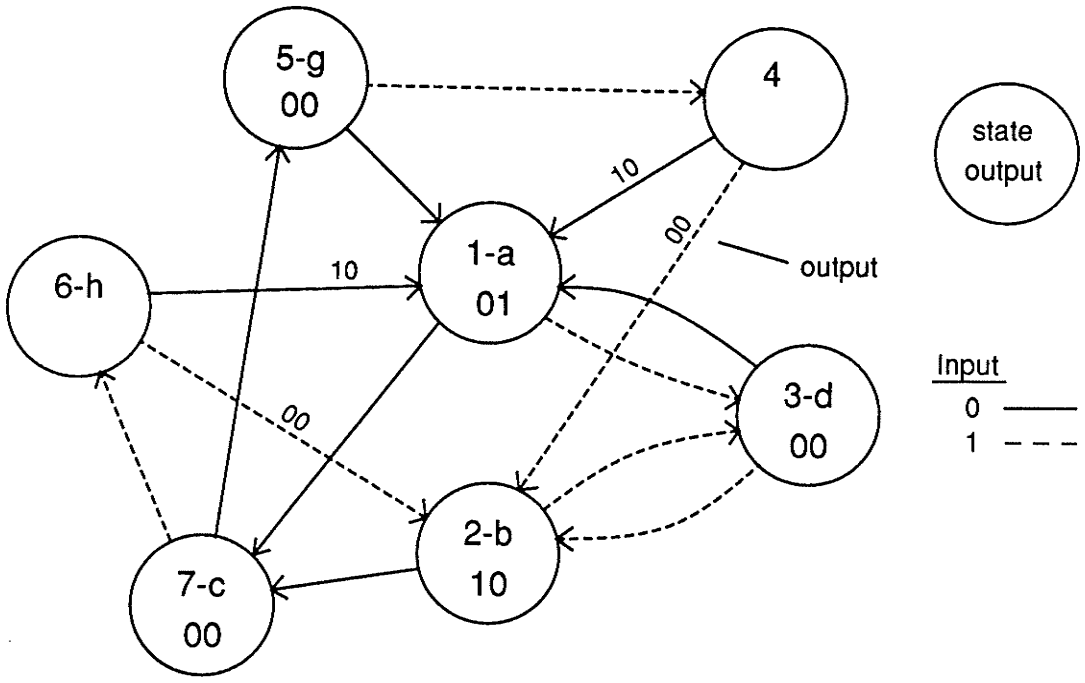


Figure 4.7: State diagram for IBM (2,7,1,2).

representation of the code is shown in Figure 4.8. The state diagram is combined with that of the channel as outlined in the previous section, and the result is shown in Figure 4.9.

The next step is to create the transition matrices T_{IBM27B}, and T_{IBM27T} for the binary (without the channel) and ternary (with the channel) versions, respectively, of the state diagrams. They are derived directly from the state diagrams, noting that each path leaving a given state represents a single input bit and thus has a probability of $\frac{1}{2}$ of occurring (assuming, as usual, that the input symbols are equiprobable). The state diagrams are relabeled with numerical symbols so that they can correspond to rows and columns in the transition matrix. The transition matrices corresponding to the labels in Figures 4.7 and 4.9 are

$$T_{IBM27B} = \begin{bmatrix} 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \end{bmatrix}$$

$$T_{IBM27T} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

A check of the eigenvalues for the two matrices confirms that the two state diagrams represent finite irreducible aperiodic Markov chains. The long-term state probabilities can be calculated and are presented in Table 4.3.

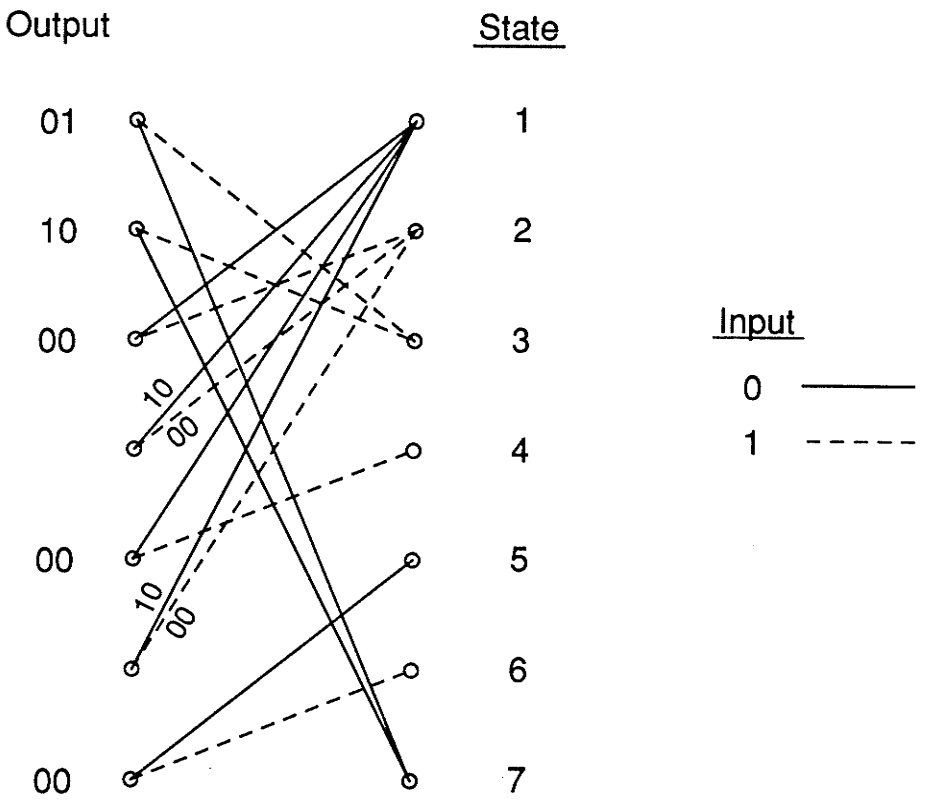


Figure 4.8: Trellis for IBM (2,7,1,2).

4.4. IBM (1,7,2,3)

Engineered and patented by Franaszek, this code increases data density by 50%; not as much as the (2,7,1,2) code, but it does have relaxed timing constraints. Like its (2,7,1,2) brethren, it is a variable rate code and therefore, it must be dealt with in a manner similar to the previous code.

The nominal rate of this code is $\frac{2}{3}$, with the actual information block being two or four bits long. The coding process is described in Table 4.4. Some of the code-words have an "X" (complement of the preceding codeword bit) as the first symbol requiring that the last bit of the previous codeword be known. This complicates the state diagram slightly.

To simplify the state diagram, the rate will be fixed at $\frac{2}{3}$. With two input bits per time step, each state has four paths leaving it. An inspection of the coding table

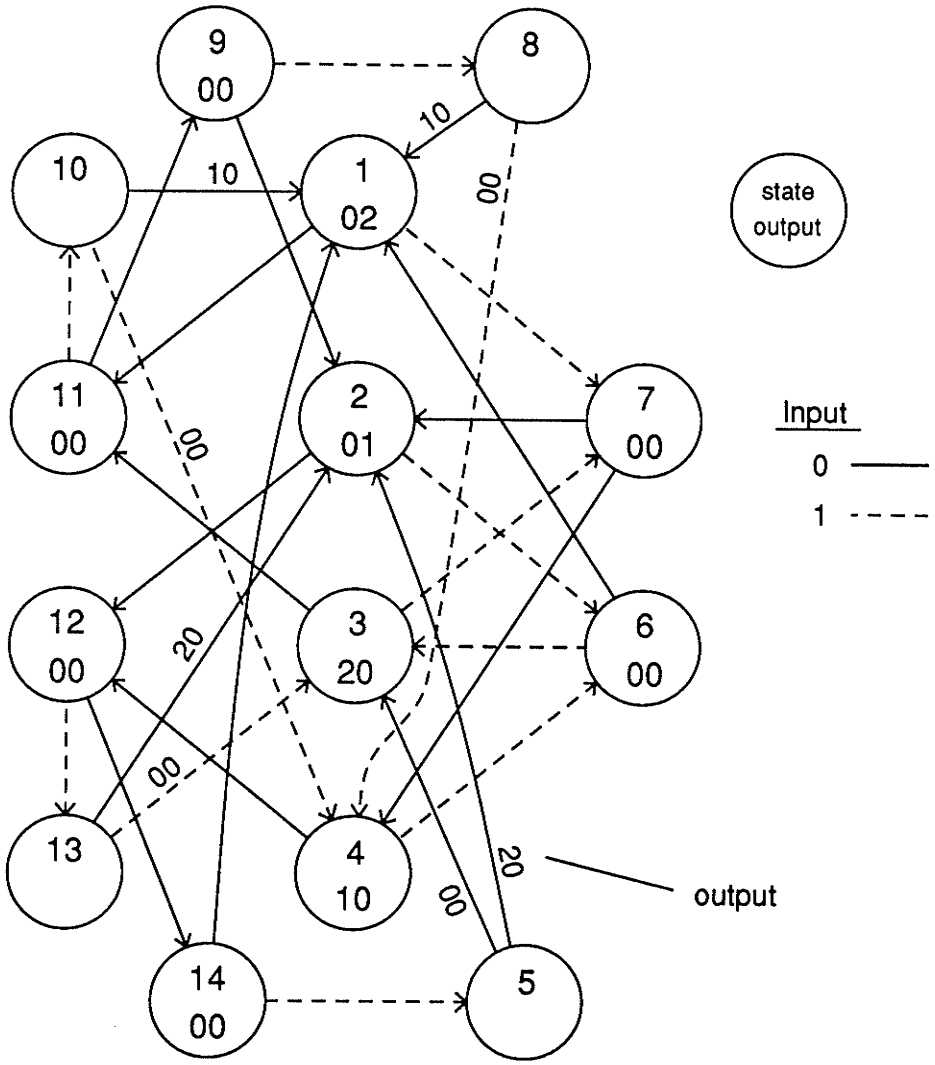


Figure 4.9: State diagram for IBM (2,7,1,2) with channel.

shows that the first two input bits are not sufficient to specify the first three output bits. This means that the output needs to be delayed by some multiple of three bits.

The natural starting point is a single frame delay (corresponding to three output bits). In the same manner as for the (2,7,1,2) code, a state is defined for each possible sequence in the last frame of a codeword. The situation is somewhat more complex than before, as five states *a* to *e* have to be created for each of the outputs "000", "001", "010", "100", and "101" (keeping in mind that "X" can represent a "0" or a

Table 4.3: State probabilities for IBM (2,7,1,2).

Without channel	
State	$P(state)$
1	.214286
2	.166667
3	.190476
4	.047619
5	.095238
6	.095238
7	.190476

With channel	
State	$P(state)$
1	.107143
2	.107143
3	.083333
4	.083333
5	.023809
6	.095238
7	.095238
8	.023809
9	.047619
10	.047619
11	.095238
12	.095238
13	.047619
14	.047619

Table 4.4. IBM (1,7,2,3) coding table.

Information Block	**Codeword**
01	X00
10	010
11	X01
0001	X00001
0010	X00000
0011	010001
0000	010000

"1"). In fact, every three bit sequence that satisfies d=1 is represented. These states are entered after the last two bits of an information block. The output normally associated with these two information bits are produced by any path leaving the state, representing a one frame delay. Note that for the information blocks "01" and "11",

there are two distinct paths entering two different states. This is to allow for the fact that the first bit of the codeword (the entire codeword being only one frame long) is specified as "X", thus the codeword may be one of two sequences depending on the last bit of the previous codeword.

The inputs to these paths are the first two bits of an information block, but the coding table indicates that three of the information blocks are only two bits in length. The segments with these inputs ("01", "10", and "11") are directed back to one of the five existing states according to their codeword ("X00", "010", and "X01" respectively), so that the correct codeword is generated in the next frame. This is displayed in Figure 4.10.

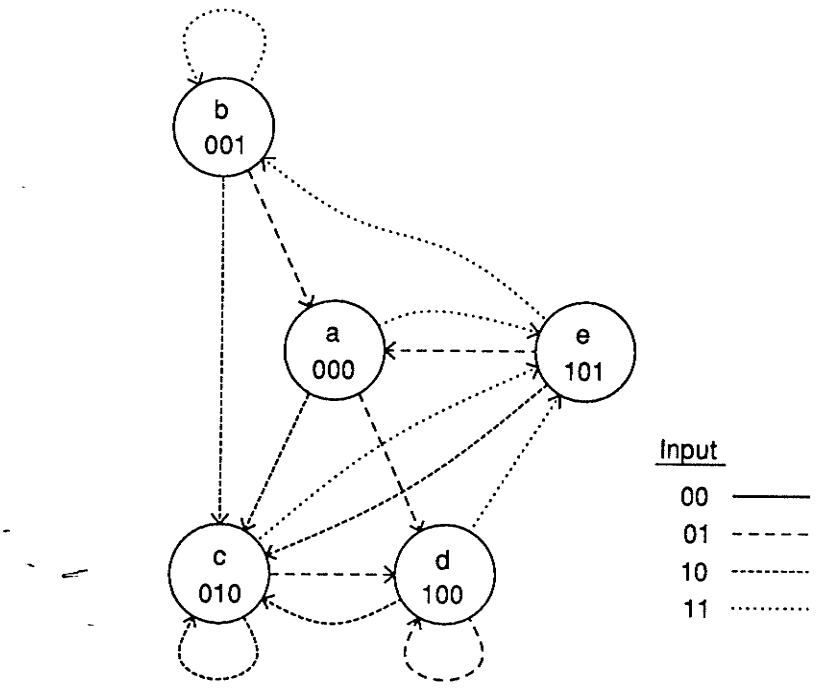


Figure 4.10: Delay of one frame for IBM (1,7,2,3).

Now there is only one path out of each state to account for, the input being "00" in each case. These five segments temporarily now terminate in five new states, f to j. Again, each of the five states has four exitting paths. The input is the second

frame of the information block, while the output is the first frame of the codeword. Checking the coding table reveals that the output can be specified down to the symbol "X" in the first bit of two of the codewords. The output of this symbol requires knowledge of the last bit of the previous codeword. However, the state from which the segments exits provides that information, as there is only one path into that state. So the output can be completely specified as shown in Figure 4.11.

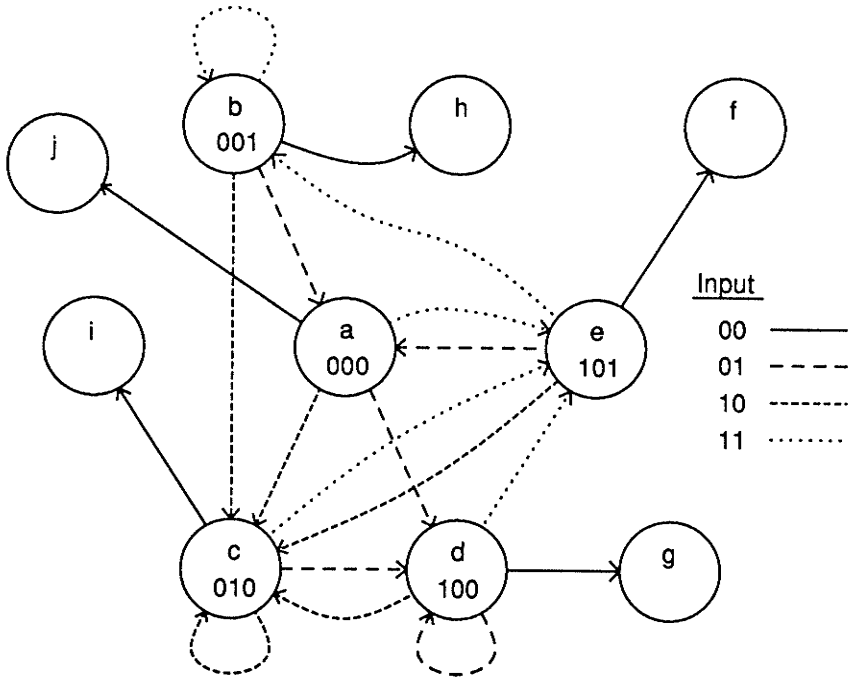


Figure 4.11: Extended state diagram for IBM (1,7,2,3).

The input to these paths is also the last frame of an information block, so their destinations are one of the five states, a to e, as previously explained. The state diagram is now complete. Is it minimal? Note that the three states f, g, and h have identical exitting paths (that is, they have the same input/output combinations, and they go to the same states. The same is true for the two states i and j. The first three states can be combined into state k, while the last two become state l. Now all states are truly distinct as shown in Figure 4.12. Finally, the channel properties are added to

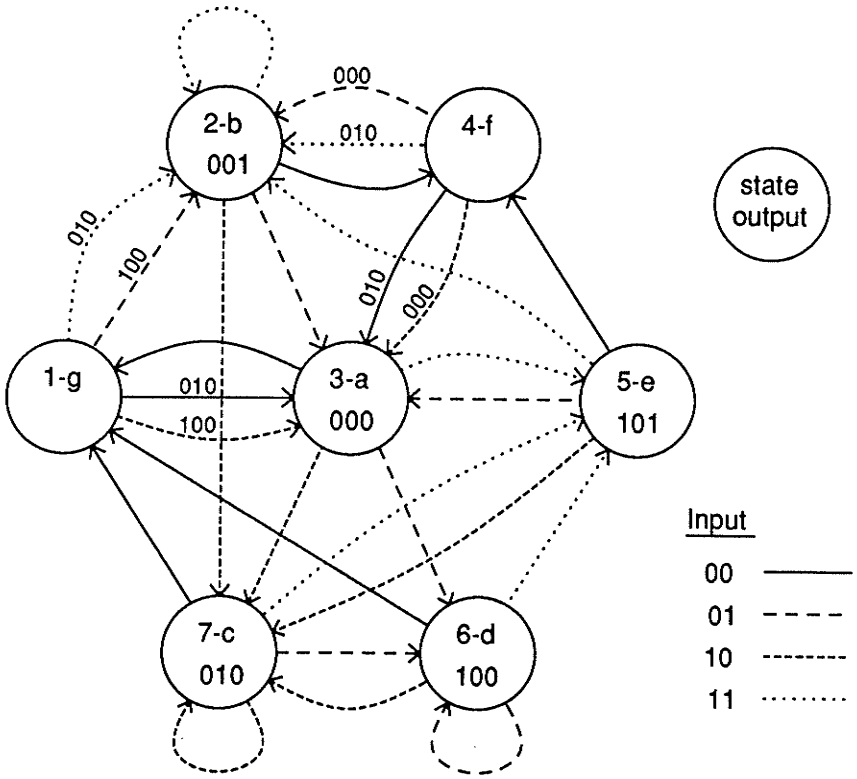


Figure 4.12: State diagram for IBM (1,7,2,3).

the coding process and the result shown in Figure 4.13.

Once again, the transition matrices are obtained from the state diagrams, with the states relabeled as numbers to correspond to rows and columns of the matrix.

$$T_{IBM\,17B} = \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & \frac{1}{4} \\ \frac{1}{4} & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & \frac{1}{4} \\ \frac{1}{4} & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \end{bmatrix}$$

$$T_{IBM17T} = \begin{bmatrix}
0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 \\
0 & 0 & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 \\
\frac{1}{4} & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 \\
0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{4} & 0 \\
0 & \frac{1}{4} & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & \frac{1}{4} \\
\frac{1}{4} & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & \frac{1}{4} \\
\frac{1}{4} & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 \\
0 & 0 & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 \\
0 & \frac{1}{4} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 \\
0 & 0 & 0 & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & \frac{1}{4} \\
0 & \frac{1}{4} & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & 0 & 0 \\
0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0
\end{bmatrix}$$

The eigenvalues reveal that the processes are irreducible aperiodic Markov chains. The long-term state probabilities are shown in Table 4.5.

4.5. ISS (1,7,2,3)

In terms of data density, this code is identical to the IBM (1,7,2,3) code, but its implementation is somewhat different. This code was invented by Jacoby, Cohn and Bates in 1982, but the work was published by Jacoby and Kost [20] in 1984. The code has been implemented in the ISS 8470 disk drives.

The code actually resembles a simple (albeit non-linear) block code of rate $\frac{2}{3}$ as described by Table 4.6. In this form, however, there are four cases where the d-constraint would be violated over two frames. When these four cases occur, the coding process refers to a secondary special-case table shown in Table 4.6. Obviously, in order to detect these exceptions, the encoder must have information on the next

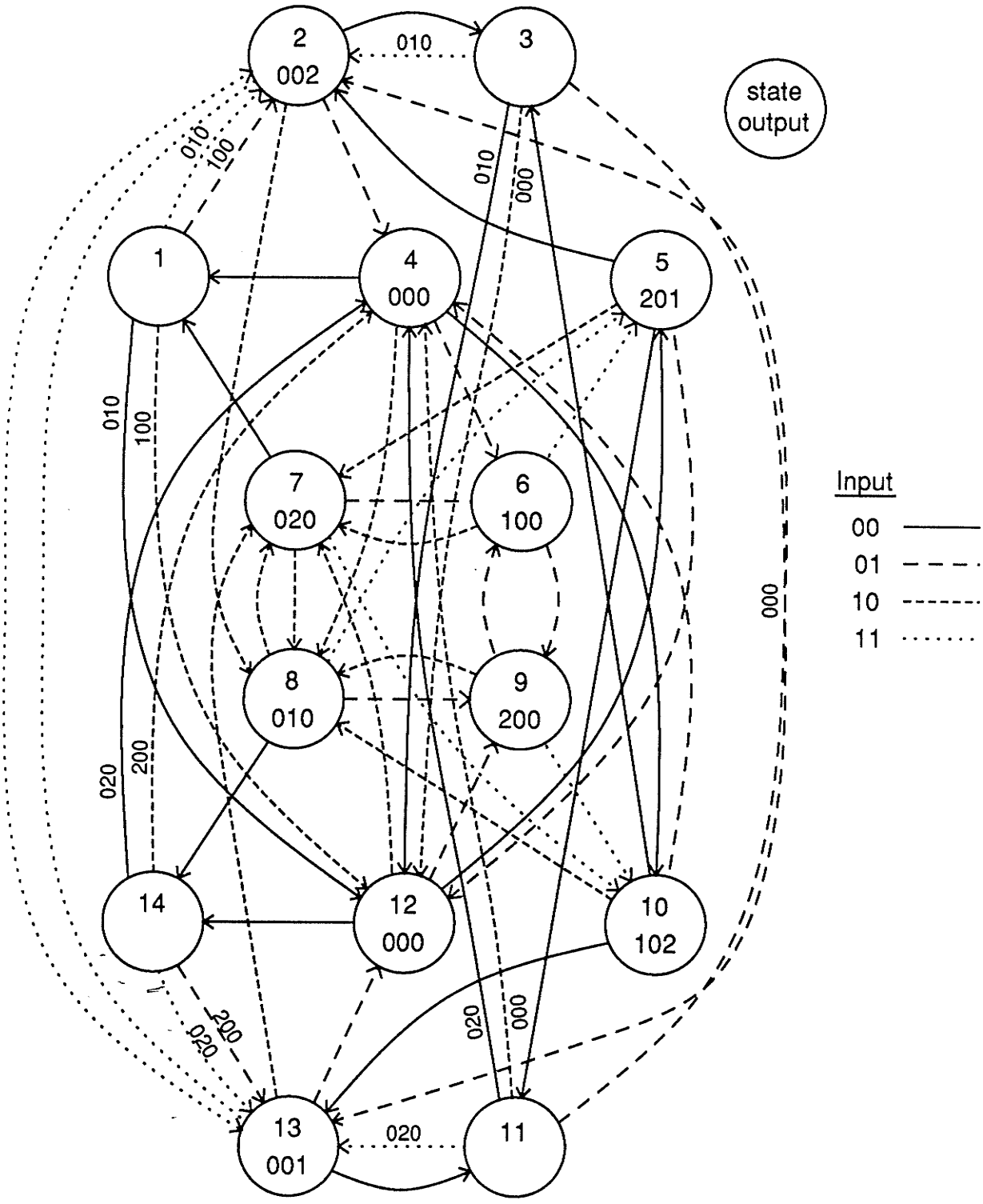


Figure 4.13: State diagram for IBM (1,7,2,3) with channel.

Table 4.5: State probabilities for IBM (1,7,2,3).

Without channel	
State	*P* (*state*)
1	.125
2	.175
3	.175
4	.075
5	.125
6	.125
7	.200

With channel	
State	*P* (*state*)
1	.0625
2	.0875
3	.0875
4	.0375
5	.0625
6	.0625
7	.1000
8	.1000
9	.0625
10	.0625
11	.0375
12	.0875
13	.0875
14	.0625

(future) frame of data. Jacoby and Kost termed this requirement "full-word look-ahead". The special-case table modifies the output of the present and the succeeding frame.

Since the rate is already fixed with co-prime integers, this aspect does not have to be changed for the state diagram representation. However, the full-word look-ahead means that the output should be delayed by a frame or more. (Actually, this is already the case in the original implementation; it has just been relabelled full-word look-ahead).

The simplest case is just a single frame delay, and this can be the starting point. The basic encoding table is represented first, and is a simple four-state state diagram as shown in Figure 4.14. The input associated with a segment determines the destination of that segment, and all segments exitting a state will have an output associated with

Table 4.6: ISS (1,7,2,3) coding table.

Basic Coding Table	
Information Block	**Codeword**
00	101
01	100
10	001
11	010

Special Coding Table			
Information Block		**Codeword**	
Present	Next	Present	Next
00	00	101	000
00	01	100	000
10	00	001	000
10	01	010	000

that state. For instance, any segment for the input "00" goes to state a, and any segment leaving state a outputs 101, which is the codeword for the information block "00". This output now has the appropriate delay of one frame.

The next step is to incorporate the special-case encoding table into the state diagram. Since the output is delayed by one frame, the present information block becomes the previous information block, while the next becomes the present. The previous information block is represented by the state that the process is in. As in the previous example, if the process is in state a, then the previous input is known to be "00". Now, a necessary condition for invoking the special-case encoding table is that the previous input be "00" or "10". This means that only paths leaving states a and c need be considered. In addition, the present input must be "00" or "01", so only two segments out of these states are changed.

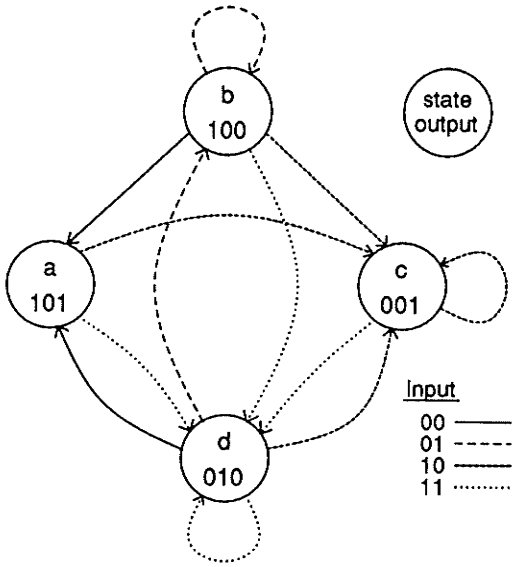


Figure 4.14: Delay of one frame for ISS (1,7,2,3).

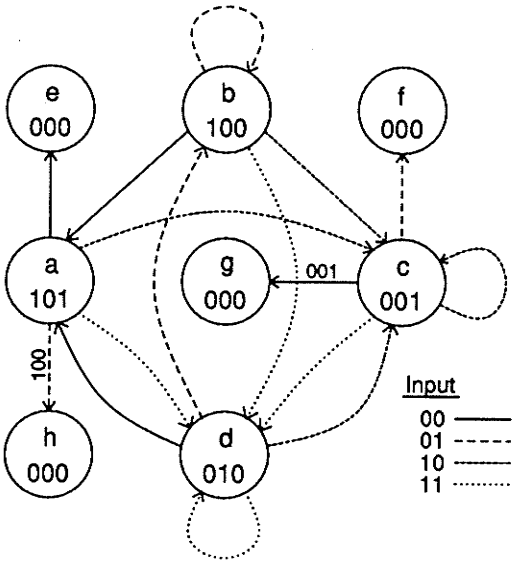


Figure 4.15: Extended state diagram for ISS (1,7,2,3).

When one of these four special cases is encountered, the present and the next output bit must be changed. To incorporate this, a new state is created for each of the four segments to enter as shown in Figure 4.15. The outputs associated with these segments are changed to match the present codeword. Each of the new states, e to h, has four exitting segments all outputting "000" and entering the state appropriate for

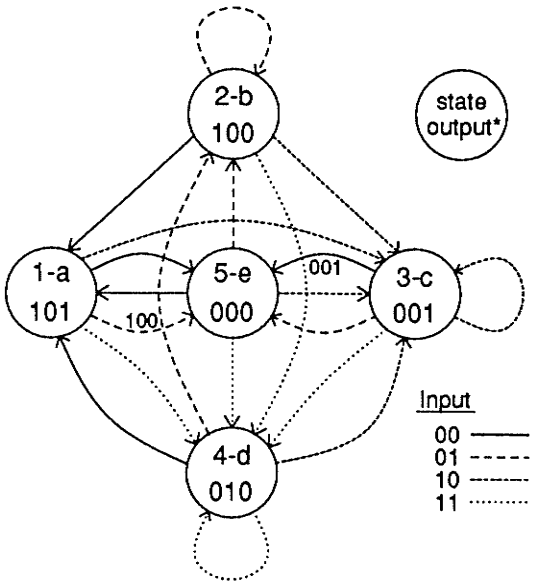
the input. The coding process has now been completely specified. However, the states e to h are actually equivalent, since they all have identical exitting segments that go to identical states. Thus, they can all be combined into one state i as shown in Figure 4.16.

The final step is to incorporate the channel into the model as described in section 4.2. The resulting state diagram is shown in Figure 4.17.

Figures 4.17 and 4.18 also show the states relabelled as integers. Using these as column and row indices, the transition matrices T_{ISSB} and T_{ISST} can be created for the models without and with the channel respectively.

$$T_{ISSB} = \begin{bmatrix} 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 \\ 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 \end{bmatrix}$$

$$T_{ISST} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & 0 & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & 0 \\ 0 & \frac{1}{4} & \frac{1}{4} & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & \frac{1}{4} & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & 0 & \frac{1}{2} \\ \frac{1}{4} & 0 & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & 0 \\ 0 & \frac{1}{4} & \frac{1}{4} & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & 0 & 0 \\ 0 & \frac{1}{4} & \frac{1}{4} & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & 0 & 0 \\ \frac{1}{4} & 0 & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & 0 \end{bmatrix}$$



*except as noted on path

Figure 4.16: State diagram for ISS (1,7,2,3).

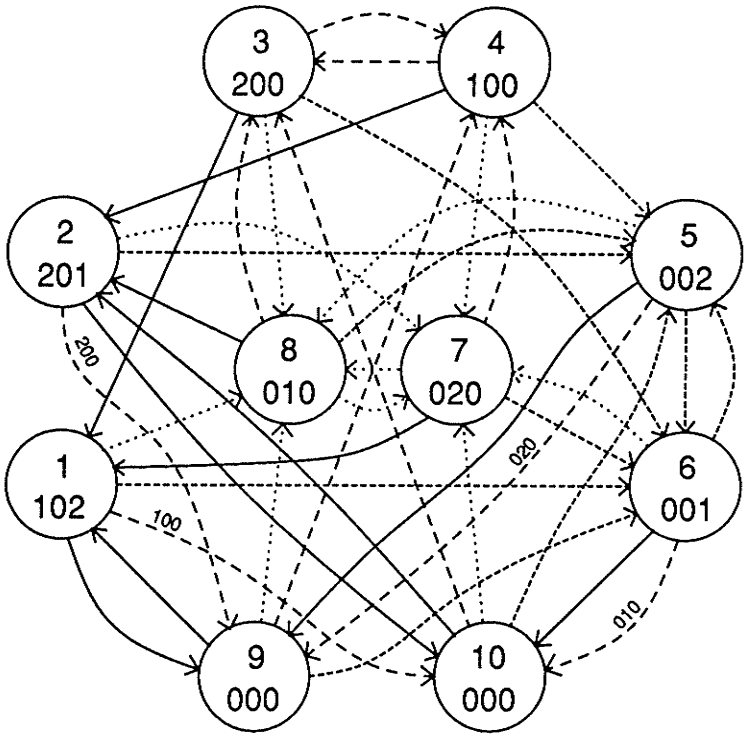


Figure 4.17: State diagram for ISS (1,7,2,3) with channel.

Once again, the eigenvalues confirm that the processes are irreducible aperiodic Markov chains. Thus the long-term state probabilities can be calculated and they are shown in Table 4.7.

Table 4.7: State probabilities for ISS (1,7,2,3).

Without channel	
State	P (*state*)
1	.15
2	.15
3	.25
4	.25
5	.20

With channel	
State	P (*state*)
1	.075
2	.075
3	.075
4	.075
5	.125
6	.125
7	.125
8	.125
9	.100
10	.100

Chapter 5

Modelling the Behaviour

The desired goal is an analytic method for reliably predicting the error rate of a code for various signal to noise ratios. The work in the previous chapter provides information on each coding process in a trellis form. Now this information must be transformed into an expression that models the error performance of the codes. In convolutional codes, there exist techniques that provide exact polynomial expressions for the error performance of these codes from a state diagram model of the coding process [7, 21]. However these techniques depend heavily on the linearity of the code, and therefore, are inappropriate for RLL codes. These techniques, however, do suggest similar avenues that may be applicable to RLL codes.

5.1. Error Events Defined

In order to model error rates, the mechanics of an error must be known. That Viterbi's algorithm finds the minimum distance path is already known, but analysing the behaviour from the standpoint of entire data sequences is far too complex. A smaller, more manageable event is needed. Consider the (arbitrary) recorded sequence in Figure 5.1. For any reasonable (and most unreasonable) signal to noise ratios, the decoded path will match the recorded sequence for significant portions of this path. There will be portions, isolated from each other, where the decoded path diverges from and subsequently merges to the recorded path, as illustrated by the dashed line in the figure. Each one of these occurrences is termed an "error event".

Consider one of these events in isolation. Up to the point at which they diverge, both paths will have the same Euclidean distance from the received signal. Thus, the

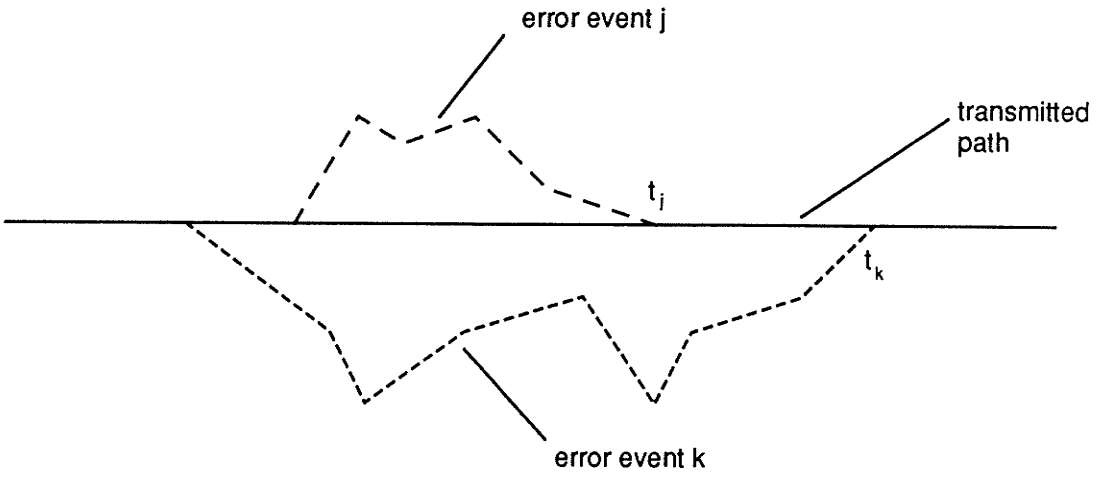


Figure 5.1: Error events on an arbitrary path.

past history does not affect the error event. Over the unmerged sections of the paths, the distances for the two paths will start to differ. For the error event path to be chosen over the recorded path, the error event path must obviously have a smaller Euclidean distance to the received signal over the unmerged segments than the recorded path. If this occurs, then Viterbi's algorithm (and thus MLSD) will select the error event path over the recorded path at the point at which the two paths merge. If the recorded path has a smaller distance to the received signal over the unmerged segment, then this path will be chosen, and the particular error event cannot occur. So a necessary condition for the error event j to occur is that the noise must be of such a nature as to bring the received signal closer to path j than to the recorded path i over the duration of path j. The probability of this occurring depends on the probability density function of the noise, the metric used, and the distance between paths i and j. This probability is denoted $P_d(j/i)$.

Although a necessary condition has been determined for error event j to occur, it is not a sufficient condition. This can be illustrated by considering yet another error event k as shown in Figure 5.1. This error event diverges from the recorded path before error event j and merges with the recorded path after j. It is quite possible for

the received signal to be closer to path j than the recorded path i over the duration of j, and for it to be closer to path k than i over the duration of k. The first situation will cause path j to be selected at state S_j at time t_j over path i over the duration of j, as previously mentioned. However, the second situation will similarly cause path k to be selected over path iji (that is, path i except where it has been superseded by path j) at state S_k at time t_k. Thus, despite the fact that the conditions necessary for error event j to occur are present, error event j does not occur, due to error event k.

5.2. In Search of Performance

The goal of this chapter is to find a method of creating an expression to model the performance of these codes. Convolutional codes have Hamming distance properties that can be described as polynomials. This corresponds to a precise description of the distance properties from the all-zero path and allows a tight upper bound for the error probabilities from this path to be found. The linear property of the codes then allows the model to be generalised to all paths [7, 21].

Such is not the case for RLL codes. First, the absence of an all-zero path implies that analysis of the weight structure does not provide any useful information. Second, the non-linear nature of the code means that all paths have to be individually analysed. In general, these problems result in an intractable situation, since extensive computations would be required for each code for each signal- to-noise ratio (SNR). However, insight into the problem allows approximations to be made that result in an expression that is easily evaluated for various SNR's. Extensive calculations are still required, but far fewer than before and most significantly, only once for a particular code. The resulting upper bound on an estimate of the error performance is accurate at any reasonable SNR and is asymptotically tight.

5.3. Bit Error Probability Bounds

Though RLL codes are non-linear and the upper bounds on BER or probability of bit error (P_{be}) derived for convolutional codes are not directly applicable, it is possible to obtain an upper bound on an estimate of the probability of bit error (denoted P_{be}^{*}) by a method that parallels the analysis for convolutional codes [7, 21].

In order to model the BER, the expected number of information bit errors must be found, i.e.,

$$E\{n_{be}\} \tag{5.1}$$

P_{be} can then be obtained by simply dividing (5.1) by the number of information bits per time step (m).

$$P_{be} = \frac{E\{n_{be}\}}{m} \tag{5.2}$$

Considering (5.1), note that the expected number of bit errors can be found by averaging the expected number of bit errors given a state, $E\{n_{be}/s\}$, over all the states.

$$E\{n_{be}\} = \sum_{s \in S} E\{n_{be}/s\} P(s) \tag{5.3}$$

where S is the set of all states and $P(s)$ denotes the probability of being in state s. These state probabilities were found for all of the codes in Chapter 4.

Now consider $E\{n_{be}/s\}$. The number of bit errors for a given state depends on which path is taken out of that state, since each path may have its own unique distance distribution to other paths. Thus the bit errors must be averaged over the paths to find $E\{n_{be}/s\}$.

$$E\{n_{be}/s\} = \sum_{s \in S} P(s) \sum_{i \in P_s} P(i) E\{n_{be}/i\} \tag{5.4}$$

where P_s is the set of paths that originate from state s, and $P(i)$ is the probability of taking path i.

An error event occurs when path $j \in P_s$, $j \neq i$ is chosen instead of path i, and that event creates a number of bit errors corresponding to the number of information bit positions in which i and j disagree (the Hamming distance of the information bits between paths i and j). This event can happen only when paths i and j merge into the same state, and in the case of first error events, when i and j merge for first time. Thus

$$E\{n_{be}/i\} = \bigcup_{j \in Q_i} P_d(j/i) d_i(i,j) \tag{5.5}$$

where $P_d(j/i)$ is the probability that path j is chosen when path i is transmitted, Q_i is the set of paths that diverge from path i at state s and subsequently merge, and $d_i(i,j)$ is the Hamming distance of the information bits between i and j. Note that for an exact equality, the union of these events must be considered since it is possible for some other path k (diverging from path i before path j and merging after path j) to be chosen thus causing this error event to supersede the error event associated with path j. However, an upper bound for this expression can be found by considering the error events to be disjoint.

$$E\{n_{be}/i\} \leq \sum_{j \in Q_i} P_d(j/i) d_i(i,j) \tag{5.6}$$

Substituting into (5.4).

$$E\{n_{be}\} \leq \sum_{s \in S} P(s) \sum_{i \in P_s} P(i) \sum_{j \in Q_i} P_d(j/i) d_i(i,j) \tag{5.7}$$

P_s can be divided into smaller sets P_{sl} that consist of all the paths out of state s of length l frames (or segments). Similarly, Q_i can be broken down into sets Q_{il}. Equation (5.7) then becomes

$$E\{n_{be}\} \leq \sum_{s \in S} P(s) \sum_{l=1}^{\infty} \sum_{i \in P_{sl}} P(i) \sum_{j \in Q_{il}} P_d(j/i) d_i(i,j) \tag{5.8}$$

Assuming equiprobable information bit sequences, $P(i)$ is constant for all $i \in P_{sl}$ and $P(i) = \frac{1}{(2^m)^l}$.

$$E\{n_{be}\} \leq \sum_{s \in S} P(s) \sum_{l=1}^{\infty} 2^{-ml} \sum_{i \in P_{sl}} \sum_{j \in Q_{il}} P_d(j/i) d_i(i,j) \tag{5.9}$$

For additive white Gaussian noise (AWGN),

$$P_d(j/i) = Q\left[\frac{d_c(i,j)}{2}\sqrt{SNR}\right] \tag{5.10}$$

where

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^{\infty} exp(-\frac{x^2}{2})dx$$

that is, one-half of the complementary error function. SNR is the signal to noise ratio and $d_c(i,j)$ is the distance between the codewords represented by paths i and j. Therefore,

$$E\{n_{be}\} \leq \sum_{s \in S} P(s) \sum_{l=1}^{\infty} 2^{-ml} \sum_{i \in P_{sl}} \sum_{j \in Q_{il}} Q\left[\frac{d_c(i,j)}{2}\sqrt{SNR}\right] d_i(i,j) \tag{5.11}$$

The argument of the Q function can take on a set of discrete values and for low values of $d_c(i,j)$ (say <5) the difference between two adjacent values represents a significant portion of the value itself. Also, for arguments $x>3$, $Q(x)$ is a rapidly decreasing function. Thus for most useful SNR's (> 6dB) and $d_c(i,j)$ patterns (i.e., $d_{cmin}(i,j) < 5$), the contribution to bit errors by paths that are far apart (i.e., have large $d_c(i,j)$) is negligible. In fact, any pair of paths, i and j for which

$$d_c(i,j) > d_{\min} = \min_{\forall\, i,j} d_c(i,j)$$

makes only a very small contribution to $E\{n_{be}\}$. Therefore the number of bit errors can be approximated by considering only those paths that are $d_{\min}$ away. After this

approximation, the expression inside the error function becomes a constant.

$$E^*\{n_{be}\} \le \sum_{s\in S} P(s) \sum_{l=1}^{\infty} 2^{-ml} \sum_{i\in P_{sl}} \sum_{j\in Q_{il}} Q\left[\frac{d_{\min}}{2}\sqrt{SNR}\right] d_i(i,j)$$

$$= Q\left[\frac{d_{\min}}{2}\sqrt{SNR}\right]\left[\sum_{s\in S} P(s) \sum_{l=1}^{\infty} 2^{-ml} \sum_{i\in P_{sl}} \sum_{j\in Q_{ilmin}} d_i(i,j)\right] \qquad (5.12)$$

where Q_{ilmin} is now a subset of Q_{il} that includes only the paths $d_{\min}$ away from path i. This removes the error function from the summations leaving only the terms in the brackets which are amenable to computer evaluation. P_{be}^*, an upper bound on an approximation to the probability of bit error now becomes

$$P_{be}^* = E^*\{n_{be}\} = Q\left[\frac{d_{\min}}{2}\sqrt{SNR}\right]\left[\frac{1}{m}\sum_{s\in S} P(s) \sum_{l=1}^{\infty} 2^{-ml} \sum_{i\in P_{sl}} \sum_{j\in Q_{ilmin}} d_i(i,j)\right]$$

$$= c\, Q\left[\frac{d_{\min}}{2}\sqrt{SNR}\right] \qquad (5.13)$$

where c is a constant.

5.4. The Performance of the Codes

The error performance of each code can now be characterized by two parameters, $d_{\min}$ and c. These parameters can be found by a search of the trellis representing the code. The number of paths to search for each code is still quite large (in the thousands), so a manual analysis of each code is out of the question. Instead, a computer program is written to find $d_{\min}$ and c for each code. The state diagram for each code is represented as a path list. In this form, a list is created of all the segments in the state diagram, with each segment represented by two states (source and destination)

and an input and output. This form is easily digested by a computer program. The program was written using the C language primarily for its speed and ability to handle Boolean operations. A complete program listing is given in Appendix C.

The results of the analysis are summarized in Table 5.1. The analysis consumed between 5 minutes and 45 minutes of CPU time on an Amdhal 5870 mainframe computer for each of the codes. The MFM code without the channel took the least amount time, while the most complex was the IBM (1,7,2,3) code with the channel. The table shows that all of the codes have a $d_{\min}$ of 1 without the channel and this increases to 2 with the channel. As a comparison, the best convolutional code that can be represented by four states (i.e., with a constraint length of 3), has a $d_{\min}$ of 5. Obviously, these RLL codes do not perform as well as error correcting codes, but that was not the objective when these particular codes were designed. The results are plotted as probability of error versus signal to noise ratio curves in Figures 5.2 to 5.5.

Table 5.1: Results of computer analysis

Code	c	$d_{\min}$
MFM without channel	0.25	1
MFM with channel	0.75	2
IBM (2,7,1,2) without channel	0.3810	1
IBM (2,7,1,2) with channel	1.8512	2
IBM (1,7,2,3) without channel	1.2375	1
IBM (1,7,2,3) with channel	2.6422	2
ISS (1,7,2,3) without channel	0.925	1
ISS (1,7,2,3) with channel	2.4469	2

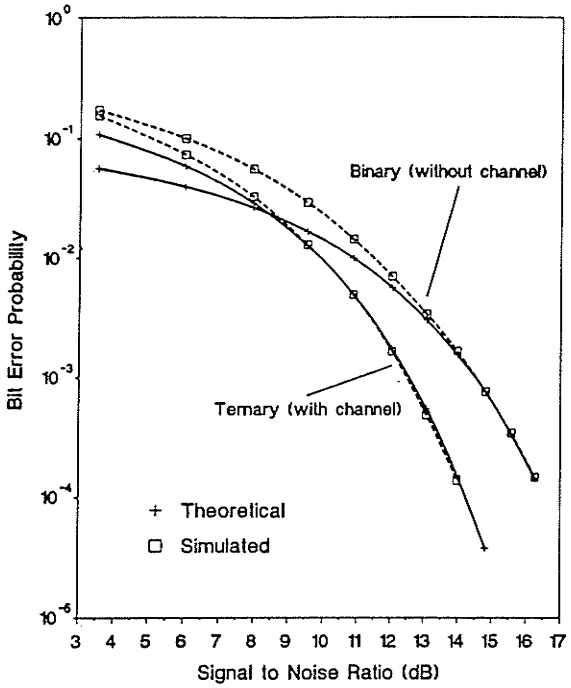


Figure 5.2: MFM (1,3,1,2).

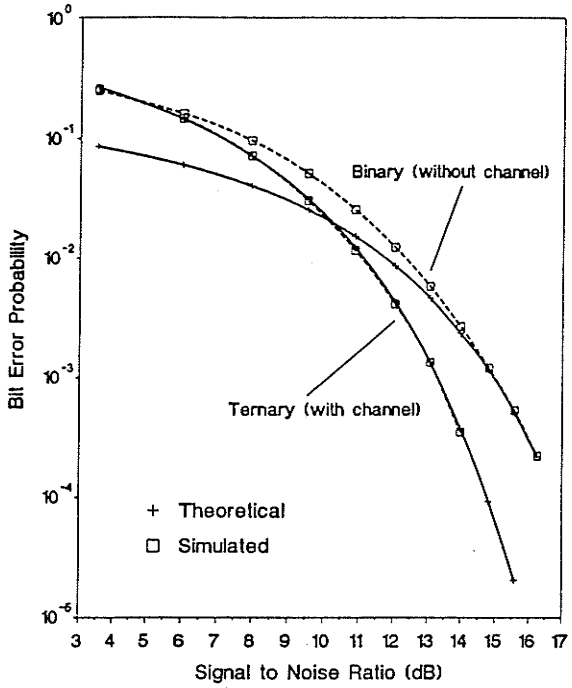


Figure 5.3: IBM (2,7,1,2).

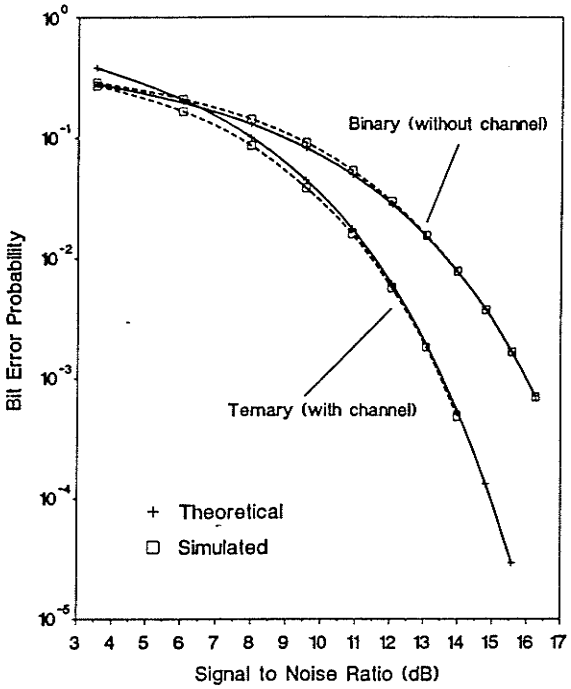


Figure 5.4: IBM (1,7,2,3).

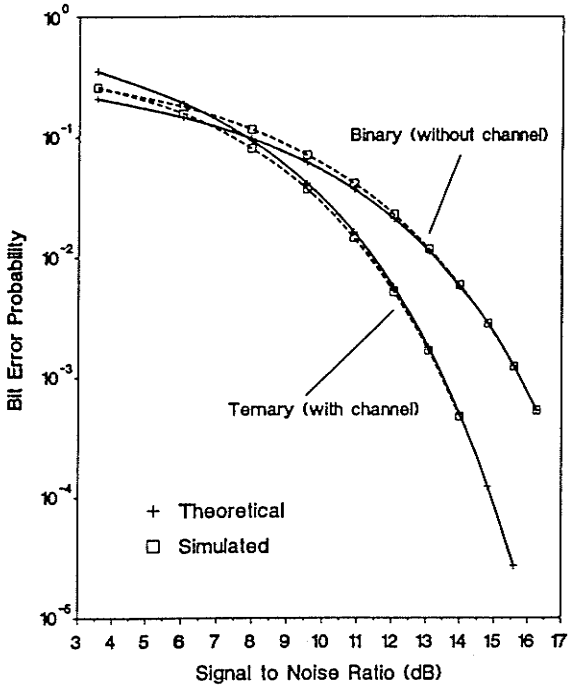


Figure 5.5: ISS (1,7,2,3).

5.5. Simulation

In order to verify these results, a computer simulation of the encoding process, the channel, and a Viterbi decoder was performed. A computer simulation rather than actual hardware testing was chosen for two reasons. The first is that the equipment to carry out and measure the hardware testing was unavailable. The second reason is that computer simulations allows just the effects of AWGN to be included. With a hardware test, many other effects that contribute to the error rate would be included (and that contribution, unknown) so that the validity of this model would still be in question. This is not to trivialize any other potential sources of error (they are certainly important in an implemented disk drive), but a complete model of error performance is well beyond the scope of this work. In some cases, models of error behaviour are so specific to the particular hardware that they have no place in any general work.

This program was written in the C language for the same reasons as the analysis program (speed and Boolean operators). It consists of three major parts. The first portion generates a pseudo-random binary input sequence using a prime polynomial (a monic irreducible polynomial of degree greater than one) in $GF(2)$ (Galois Field of two elements). The particular polynomial used has a degree of 31, thus the generated sequence has a length of 2^{31} bits. It then converts this into a coded sequence. The state diagram model is used rather than the original coding process, since Viterbi's algorithm has to use this model for decoding anyway. The binary input sequence is stored for later comparison with the decoded sequence.

The second portion represents the channel. The codeword sequence is represented using an appropriate orthonormal basis. Since this representation has only one dimension, the output of the channel (without the noise) is simply a scalar as described in Chapter 3. AWGN is simulated by the sine and cosine method [22] using

a C library function to generate pseudo-random numbers between 0 and 1. This function has a sequence length of 2^{32} which is adequate. Since the memory in the channel is modelled as part of the state diagram, it is not included here.

The final portion implements a Viterbi receiver using the same path list that was used in the analysis program, to specify the structure of the trellis. The program was written so that the survivor sequence could be set arbitrarily, and a length that did not affect the error rate was found by trial and error. For all cases, a survivor sequence 32 frames long was found to be sufficient. This length is quite reasonable in an actual implementation of a receiver. This portion also compares the decoded sequence to the data sequence stored in the first portion, and counts the errors.

The simulations were run for each of the codes (with and without the channel) for signal to noise ratios of 3dB to 15dB. They required between 50 minutes and 8 hours of CPU time on an Amdhal 5870 mainframe computer. The 99% confidence intervals are less than 5% of the actual error rate in all cases. The results are plotted in Figures 5.2 to 5.5, and a complete program listing is provided in Appendix D.

5.6. Comparison

The results of both the analysis and the simulations are contained in Figures 5.2 to 5.5. The curves are asymptotically tight, as expected in section 5.2, and show close agreement above a signal to noise ratio of 10dB or so. Since disk drives typically have signal to noise ratios above 15dB, the results of the analysis prove to be useful.

At low signal to noise ratios, the curves diverge somewhat (although they are still reasonably close). There are two possible reasons for this. The first is that the analysis made the approximation that only paths with distance $d_{\min}$ would contribute to error rates, thus discarding all the other paths. At high signal to noise ratios, this is certainly valid. However, at very low signal to noise ratios, the error function (Q)

almost describes a flat line, so this is no longer valid. The second reason is the union bound used in equation 5.6. At high signal to noise ratios this is a very tight bound as error events become more disjoint, but at low signal to noise ratios, the error events begin to overlap.

For useful signal to noise ratios, the analysis provides results that almost match the simulations exactly. Thus, the analysis can be considered valid and can be applied to other codes. It does require a computer search (not providing, for example, an elegant polynomial solution) but the time required is quite reasonable. It is far less than the CPU time required to simulate a codes' performance (by an order of magnitude or more) and gives a result applicable for arbitrary signal to noise ratios.

Chapter 6

Conclusion

6.1. Error Performance

The results in Table 5.1 show that all of the codes have the same $d_{\min}$, so there are no large differences between the codes themselves. The IBM (1,7,2,3) code and the ISS (1,7,2,3) code also have similar constants c. However, the ISS (1,7,2,3) code has a slightly lower value of c and a less complex state diagram representation, thus it would appear superior. MFM has an error rate three to five times lower than the (1,7,2,3) codes and two to three times lower than the IBM (2,7,1,2) code. However, it is significantly poorer in terms of data density, so it should only be chosen when error performance is paramount. The IBM (2,7,1,2) code has an error rate somewhere between MFM and the (1,7,2,3) codes. Its high data density and moderately simple trellis (it has as many, or more states than the (1,7,2,3) codes, but significantly fewer arcs) make it a good coding choice.

It is of note that the values of $d_{\min}$ are the lowest possible (except for codes with identical codewords for distinct paths: a situation that will result in errors in the total absence of noise). As mentioned earlier, the prime consideration in designing RLL codes was data density, not error performance. However, now that these codes can be modelled in a format where the error mechanisms are clear, it should be possible to improve upon this worst-case performance. Since these codes can be considered as a trellis, it would be a matter of designing a trellis with associated outputs such that $d_{\min}$ is some other (non-zero) value. This is not necessarily a simple matter (many hours expended for exactly such an end has convinced this author), but any such realisation

will garner at least a 3dB coding gain over these present codes. The fact that convolutional codes base their error correction capabilities on the structure of a trellis provides the motivation for this.

6.2. About the Channel

In Chapter 1, it was intuitively felt that properly considering the effects of the channel should improve the performance of a receiver. The results from the analysis and the simulation bear this out. In all cases, including the channel model increased $d_{\min}$ from one to two. In fact, the nature of the channel forces the distance between any two (diverging and converging) paths to be even. This is due to the fact that the outputs of two paths can differ in only one of two ways, as shown in Figure 6.1. One alternative is to have exactly the same output except that one or more non-zero outputs are displaced in one path (relative to the other). In this case, there is a distance of 2 for each displaced symbol. The other alternative is that one path is the same as the other, except that it has both a "1" and a "2" inserted somewhere. Again the distance is two each time this pair of symbols is inserted.

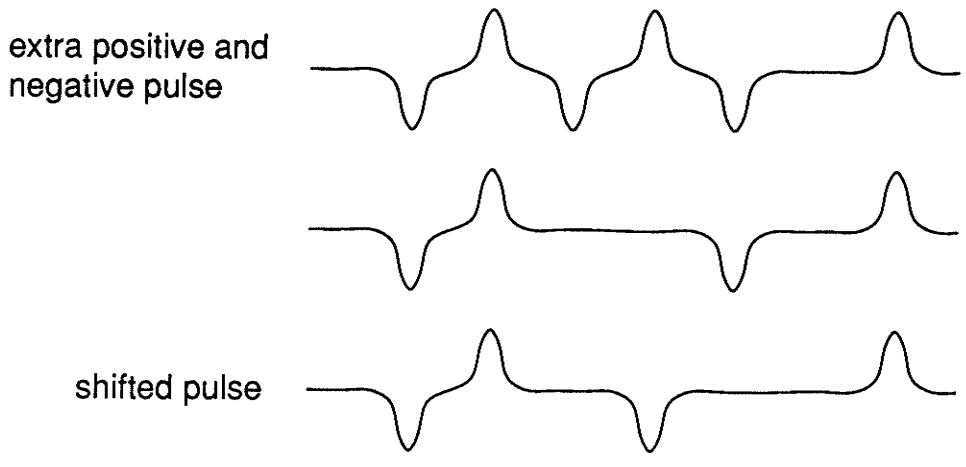


Figure 6.1: Differing paths.

For these four codes, the properties of the channel results in a coding gain of 3dB (corresponding to a doubling of d_{min}) when it is appropriately considered. This is a very significant gain, and thrusts RLL codes (which are by design just channel codes) into the class of error detection codes. (Since the entire decoding procedure is carried out in a single step, the actual gain is not the error detection capability, but rather, increased noise immunity which is illustrated in the aforementioned 3dB gain.). It is especially remarkable in that no changes have to be made to the storage system, or indeed, the stored data itself. The coding process that provides the 3dB gain is inherent in the magnetic channel and thus, in all magnetic storage systems. The only requirement is a receiver that can properly exploit this process. In this case MLSD, realised by Viterbi's algorithm, is used to optimally utilise the potential coding gain.

6.3. Implementability

This analytical treatment and simulation are all very nice, but still raises the question as to whether or not such a receiver can be implemented. Viterbi receivers have already been built for convolutional codes, so this is not in question. However, such receivers are usually designed as single-chip VLSI circuits because of the speed and computational load required [23]. General-purpose CPU's could be used for slower channels, but such a solution is very expensive. The high speed of magnetic channels dictates that a specialised VLSI chip be used to decode the output.

Can such a receiver be implemented on a single chip? Northern Telecom's 3 micron CMOS fabrication technology has proved more than adequate for a four-node receiver. This would suffice for MFM, but not for the other codes. However, present state-of-the art technology has feature sizes below one micron. Such a fabrication process would be able to support a 64 node receiver, which is considerably larger than required for any of the codes covered in this work.

6.4. Contributions

Optimal decoding of RLL codes is explored for the first time. A procedure whereby optimal decoding can be implemented is outlined, and the BER performance of such a system analytically derived. Simulation confirms the validity of the analysis for all reasonable (and most unreasonable) signal to noise ratios.

The analysis reveals the trellis structure of the RLL encoding process, much like the structure that provides the error control capabilities of convolutional codes. This opens the possibility of providing error control capabilities to RLL codes which, by design, are simply channel codes.

The unique properties of the magnetic channel are included in the RLL encoding process, thus extending the optimal decoding to include the channel itself. In doing so, the analysis reveals that $d_{\min}$ increases from one to two for each of the codes. This thrusts each of the RLL codes into the class of error detection codes, proving the previous contention. Since the entire decoding process is actually done in one stage, the actual benefit is increased noise immunity which translates into a 3dB effective coding gain in signal to noise ratio.

6.5. Further Study

The insight that this work provides into the error mechanisms of RLL codes and magnetic channels opens several avenues for further research. The observation was made that these codes have $d_{\min}$'s that are the lowest possible for codes of their type. Improving this situation is certainly desirable and would involve finding a trellis structure and output assignment that increases $d_{\min}$. Incorporating the channel into the encoding process increased $d_{\min}$ from one to two, so this is obviously possible.

In Chapter 2, it was mentioned that ISI is a very undesirable phenomenon to be avoided at all costs. However, ISI does not present as severe a problem to a trellis

structure (as long as the ISI is accounted for). Bit densities could be increased to the point where ISI does occur (even with the RLL code), and with a properly designed receiver, the performance may be degraded by an acceptable amount. This would involve designing trellises that model not only the code and the channel, but the ISI as well. Obviously, the complexity would increase, but the storage gain may make the endeavor worthwhile.

Work could also be directed at providing a more systematic basis for RLL codes. They will always be non-linear, so perhaps, a strict mathematical basis (such as is available for convolutional codes) may never be possible. However, Ungerboeck [24] developed a systematic non-linear treatment based on trellises for a variety of signals that can be represented by two orthonormal signals. Such a treatment for RLL codes would be a significant improvement over what is presently available in this field.

References

[1]. H. Kobayashi and D. T. Tang, "Application of Partial-response Channel Coding to Magnetic Recording Systems," *IBM J. Res. Develop.*, pp. 368-75, July 1970.

[2]. Roger W. Wood and David A. Petersen, "Viterbi Detection of Class IV Partial Response on a Magnetic Recording Channel," *IEEE Trans. Commun.*, vol. COM-34, no. 5, pp. 454-61, May 1986.

[3]. L. G. Sebestyen, *Digital Magnetic Tape Recording for Computer Applications,* Chapman and Hall, London, 1973.

[4]. Finn Jorgensen, *The Complete Handbook of Magnetic Recording,* Tab Books, Blue Ridge Summit, PA, 1980.

[5]. Richard W. Hamming, *Coding and Information Theory,* Prentice-Hall, Englewood Cliffs, NJ, 1980.

[6]. Richard E. Blahut, *Theory and Practice of Error Control Codes,* Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.

[7]. Andrew J. Viterbi and Jim K. Omura, *Principles of Digital Communication and Coding,* McGraw-Hill Book Company, New York, 1979.

[8]. Arnold M. Michelson and Allen H. Levesque, *Error-Control Techniques for Digital Communication,* John Wiley & Sons, New York, 1985.

[9]. John C. Mallinson, "The Next Decade in Magnetic Recording," *IEEE Trans. Magnetics*, vol. MAG-21, no. 3, pp. 1217-20, May, 1985.

[10]. L. D. Stevens, "The Evolution of Magnetic Storage," *IBM J. Res. Develop.*, vol. 25, no. 5, pp. 663-75, Sept. 1981.

[11]. J.M. Harker, D. W. Brede, R. E. Pattison, G. R. Santana, and L. G. Taft, "Quarter Century of Disk File Innovation," *IBM J. Res. Develop.*, vol. 25, no. 5, pp. 677-89, Sept. 1981.

[12]. N. H. Yeh, "Review of Head-Media Coupling in Magnetic Recording," *IEEE Trans. Magnetics*, vol. MAG-21, no. 5, pp. 1338-43, Sept. 1985.

[13]. P. H. Siegel, "Applications of a Peak Detection Channel Model," *IEEE Trans. Magnetics*, vol. MAG-18, no. 6, pp. 1250-52, Nov. 1982.

[14]. Paul H. Siegel, "Recording Codes for Digital Magnetic Storage," *IEEE Trans. Magnetics*, vol. MAG-21, no. 5, pp. 1344-49, Sept. 1985.

[15]. Max Roth, "Mass Storage Gets New Boost," *Electronics Week*, pp. 55-59, April 8, 1985.

[16]. P. A. Franaszek, "Sequence-state Methods for Run-length-limited Coding," *IBM J. Res. Develop.*, pp. 376-83, July 1970.

[17]. Harry L. Van Trees, *Detection, Estimation, and Modulation Theory*, John Wiley & Sons, New York, 1968.

[18]. Dean L. Isaacson and Richard W. Madsen, *Markov Chains: Theory and Applications*, John Wiley & Sons, New York, 1976.

[19]. V. I. Romanovsky, *Discrete Markov Chains*, Wolters-Hoordhoff Publishing, Groningen, Netherlands, 1970.

[20]. George V. Jacoby and Robert Kost, "Binary Two-thirds Rate Code with Full Word Look-ahead," *IEEE Trans. Magnetics*, vol. MAG-20, no. 5, pp. 709-14, Sept. 1984.

[21]. John G. Proakis, *Digital Communications*, McGraw-Hill Book Company, New York, 1983.

[22]. G. E. P. Box, M. E. Muller, and G. Marsaglia, *Annals of Math. Stat.*, vol. 28, p. 610, 1958.

[23]. R. M. Orndorff, P. C. Chou, J. D. Krcmarik, T. W. Doak, and Dr. R. Koralek, *Viterbi Decoder VLSI Integrated Circuit for Bit Error Correction.*

[24]. G. Ungerboeck, "Channel Coding With Multilevel/Phase Signals," *IEEE Trans. Info. Theory*, vol. IT-28, no. 1, pp. 55-67, Jan. 1982.

Appendix A

Decoding

Assuming that a signal has been transmitted, the received signal corrupted by noise must be decoded. In optimum decoding using Bayes criteria, the most probable input signal for the given received signal must be chosen. The decision must include the a priori knowledge of the input signal probabilities and the probability density function of the corrupting noise. In order to reach this step, the signal must have a manageable representation over a symbol interval; the analog waveform representing the transmitted or received signal is very difficult to deal with. One such method is called an orthonormal representation. Two signals, $\phi_i(t)$ and $\phi_j(t)$, are considered orthogonal to each other over the time interval T if and only if

$$\int_{kT}^{(k+1)T} \phi_i(t)\phi_j(t)dt = 0 \qquad k = 0,1,\cdots$$

and a signal $\phi_i(t)$ is normalized if and only if

$$\int_{kT}^{(k+1)T} \phi_i^2(t)dt = 1 \qquad k = 0,1,\cdots$$

The set $\Phi = \{\phi_1(t), \phi_2(t), \cdots, \phi_m(t)\}$ is an orthonormal set if all pairs of members are orthogonal to each other, and if every member is normalized.

Let $S = \{s_1(t), s_2(t), \cdots, s_n(t)\}$ be an arbitrary set of signals over a time interval T. The orthonormal set Φ is called a spanning set for S if for every $s_i(t) \in S$,

$$s_i(t) = c_{i1}\phi_1(t) + c_{i2}\phi_2(t) + \cdots + c_{im}\phi_m(t) \qquad t \in [0,T]$$

If the spanning set has the additional property that for every j, there exists some i such that $k_{ij} \neq 0$, then Φ is called an orthonormal basis for S.

If S is the set of signals transmitted in a bit interval T, then an orthonormal basis Φ of S provides a method of completely specifying each signal as m scalars or a single (length m) vector C_i of real constants. In addition, each unique signal in S will have its own unique representation. This representation can be found by

$$c_{ij} = \int_{kT}^{(k+1)T} s_i(t)\phi_j(t)dt$$

In the absence of noise, a transmitted signal can be multiplied by each signal in the orthonormal basis, and integrated over the symbol interval T. At the end of the symbol interval, the values in the m integrators correspond to the m scalars, or the vector of length m, or the point in m-dimensional space, associated with the transmitted signal. Each unique signal is represented by one of n points in an m-dimensional space, and this technique provides a complete representation of each signal.

This techniques provides a method to distinguish between the signals, but still raises the question of optimal decoding in the presence of random noise. One property of this representation is that white noise will (by definition) have independent, identically-distributed components along each dimension of the orthonormal basis. Passing the received signal into m multipliers and integrators will result in m scalars that consist of m components due to the transmitted signal added to m independent, identically-distributed components due to the noise (if it is additive). Instead of the received vector lying exactly on one of the n points in m-dimensional space, as was the case in the absence of noise, the received vector can lie at any point in the space with a probability varying according to which signal was transmitted. Although it may seem rather convenient that the noise has been defined to have a property that is quite useful, white noise is the predominant type of noise dealt with in communication systems. In fact, the noise present at the reading head of a disk drive can be very accurately modelled as being additive white Gaussian noise (AWGN). This type of noise

has the additional property that the noise along each of the components is also Guassian.

At this stage, any received signal can be represented as a point in m- dimensional space, without any loss of information. As previously mentioned, in order to optimally decide which signal was transmitted, the most probable transmitted signal must be found given a point in this space representing the received signal. Each input signal will be considered equally probable; such a receiver is called a maximum-likehood receiver. Let the received vector be $R = [r_1, r_2, \cdots, r_m]$, and the transmitted signal set S, and the orthonormal representations C_i be as before. Then, the receiver must find the i that gives

$$\max_i Pr[s_i(t) \text{ sent } /R]$$

Using the Bayes equality a couple of times, the test becomes

$$\max_i \frac{Pr[R/s_i(t) \text{ sent}]Pr[s_i(t) \text{ sent}]}{Pr[R]}$$

Since all signals are equiprobable

$$Pr[s_i(t) \text{ sent}] = \frac{1}{n}$$

for all $s_i(t) \in S$. Also $Pr[R]$ is independent of $s_i(t)$, and thus is a constant over i. Since the receiver must merely find the maximum, the two constants can be eliminated from all the expressions without affecting the maximum. Now, the $s_i(t)$ must be found that maximizes

$$\max_i Pr[R/s_i(t) \text{ sent}]$$

For AWGN with two sided noise power $\frac{N_0}{2}$, the value of a single orthonormal component has probability density function

$$Pr[r_j = a / s_i(t) \text{ sent}] = \frac{1}{\sqrt{2\pi N_0}} \exp \frac{-(a-k_{ij})^2}{2N_0}$$

Since the noise in each component is independent of the noise in the other components, the probability of the entire vector R occurring is just the product of the probabilities of each scalar r_j occurring.

$$Pr[R/s_i(t) \text{ sent}] = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi N_0}} \exp \frac{-(r_j-c_{ij})^2}{2N_0} \tag{A.1}$$

When maximizing an expression over a parameter, it is sufficient to maximize a function that increases monotonically with that expression. Thus, in order to maximize a, an equivalent process would be to maximize $f(a)$ if $f(a)$ increases monotonically with a, since $f(a) > f(b)$ implies $a > b$. The function $\ln(a)$ increases monotonically with a for $a \in (0,\infty)$, so rather than maximizing (A.1), the natural logarithm of that expression can be maximized.

$$\max_i \ln Pr[R/s_i(t) \text{ sent}] = \max_i \ln \left[\prod_{j=1}^{n} \frac{1}{\sqrt{2\pi N_0}} \exp \frac{-(r_j-c_{ij})^2}{2N_0} \right]$$

$$= \max_i \sum_{j=1}^{n} \ln \left[\frac{1}{\sqrt{2\pi N_0}} \exp \frac{-(r_j-c_{ij})^2}{2N_0} \right]$$

$$= \max_i \left[\sum_{j=1}^{n} \ln \frac{1}{\sqrt{2\pi N_0}} + \sum_{j=1}^{n} \ln \exp \frac{-(r_j-c_{ij})^2}{2N_0} \right]$$

Since the first term is constant for all $s_i(t)$, this term can be removed from each expression without affecting the choice of the maximum.

$$\max_i \ln Pr[R/s_i(t) \text{ sent}] = \max_i \sum_{j=1}^{n} \frac{-(r_j-c_{ij})^2}{2N_0}$$

$$= \max_i -\frac{1}{2N_0} \sum_{j=1}^{n} (r_j-c_{ij})^2$$

The constant in front of the summation is constant over $s_i(t)$, so it can be removed from the expression. The negative sign can also be removed by specifying the minimum be found instead of the maximum.

$$\max_i \ln Pr[R/s_i(t) \text{ sent}] = \min_i \sum_{j=1}^{n} (r_j - c_{ij})^2$$

The observant reader will note that if the vectors R and C_i are considered vectors in m-dimensional space, the procedure is equivalent to finding the C_i (that is, the transmitted signal as represented in the space) that is closest to the received signal R, in the sense of Euclidean distance. Thus the optimal decoding over a symbol interval of a set of arbitrary (but distinct) time signals has been broken down into a series of precise mathematical operations that can be implemented in practice.

Appendix B

Markov Chains

Consider a random event. The set of all possible outcomes from this event is called the sample space of the event. Now, a function that maps a sample space into the real number set is called a random variable. Thus a random variable assigns a real number label to each outcome. A collection of random variables defined on some sample space is called a stochastic process. If the members of the collection are countable, then the process is discrete-time and the members can be denoted by $X_1, X_2, \cdots$. The state space is the set of distinct values assumed by the stochastic process, and it is denoted by S. If the state space is countable, then the process is called a chain. (Since the word chain forms half of this section heading, the reader may get an idea of where this "stochastic chain" is heading).

Now, the Markov property must be explained. Basically, this property requires that the probability that a process is in a given state at a given time, depends only on the probabilities that it was in each of the states at the immediately preceding time. More formally, a discrete-time stochastic chain $\{X_i\}$, $i = 1, 2, \cdots$ with state space $S = \{1, 2, \cdots\}$ that also satisfies

$$P[X_n = s_n \mid X_{n-1} = s_{n-1}, X_{n-2} = s_{n-2}, \cdots, X_1 = s_1] = P[X_n = s_n \mid X_{n-1} = s_{n-1}]$$

for all n and all $s_1, s_2, \cdots, s_n \in S$ is said to be a discrete-time Markov chain. Henceforth, the Markov chain will always be assumed to be discrete-time.

A Markov chain is homogeneous or stationary if the probability of a transition from any state to any other state is independent of the time of the transition, that is

$$P[X_n = j \mid X_{n-1} = i] = P[X_m = j \mid X_{m-1} = i] \qquad \forall n, m$$

In this case, the transition probability is simply denoted p_{ij}. Given a state space S with m elements, there are obviously m^2 transition probabilities. They are most conveniently denoted as a transition probability matrix (or simply transition matrix) T.

$$T = [p_{ij}] = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1m} \\ p_{21} & p_{22} & \cdots & p_{2m} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ p_{m1} & p_{m2} & \cdots & p_{mm} \end{bmatrix}$$

where p_{ij} is the probability of a transition from state i to state j in one time interval. A stochastic matrix is one that is non-negative and has all row sums equal to 1 ($\sum_j p_{ij} = 1 \quad \forall\, i$). Thus T is a stochastic matrix.

Let $p_{ij}^{(2)}$ denote the probability of a transition from state i to state j in 2 steps. State i has a certain probability of entering each state in S in the first step. In the second step, each state in S has a certain probability of entering state j. Thus, $p_{ij}^{(2)}$ is just the sum of $p_{ik}p_{kj}$ (the probability of taking the path from i to j via k) over all $k \in S$.

$$p_{ij}^{(2)} = \sum_{k \in S} p_{ik}p_{kj}$$

However, this summation is the same as taking the ith row of T and multiplying it by the jth column of T. In other words, multiplying the transition matrix T by itself results in a matrix whose elements correspond to $p_{ij}^{(2)}$. This concept can be extended to an arbitrary n transitions, thus

$$T^n = [p_{ij}^{(n)}] = \begin{bmatrix} p_{11}^{(n)} & p_{12}^{(n)} & \cdots & p_{1m}^{(n)} \\ p_{21}^{(n)} & p_{22}^{(n)} & \cdots & p_{2m}^{(n)} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ p_{m1}^{(n)} & p_{m2}^{(n)} & \cdots & p_{mm}^{(n)} \end{bmatrix}$$

A Markov Chain is irreducible if there is no non-empty subset C of the states S (other than S itself) such that the probability for a transition from some state in C to some state in S, but not in C, is zero. That is,

$$(\forall\, C \subset S, C \neq \varnothing) \qquad p_{ij} \neq 0 \qquad i \in C, j \notin C$$

In "layman's" terms, an irreducible Markov chain does not have any group of states in which a process may get stuck; that is, enter said group of states never to leave again.

Two states, i and j, are said to intercommunicate if there is some transition sequence from i to j and from j to i with non-zero probability. More precisely, if there exists some $n \geq 0$ such that $p_{ij}^{(n)} > 0$ and some $m \geq 0$ such that $p_{ji}^{(m)} > 0$, then states i and j intercommunicate. Irreducibility can be stated in terms of intercommunicability; a Markov chain is irreducible if and only if all pairs of states intercommunicate. This is intuitively reasonable since a reducible Markov chain must have a subset C such that there is no possibility of a transition from within the subset to without; that is, the members of C must not intercommunicate with non-members of C. But if all pairs of states intercommunicate, such a subset cannot be created.

A state j has a period of d if,

i) $p_{jj}^{(n)} = 0$ unless $n = md$ for $m = 1,2,...$

ii) d is the largest integer with the above property.

If $d=1$, then the state is aperiodic. Denote the probability that the first visit to state j from state i occurs at time n by $p_{ij}^{f(n)}$. Thus,

$$p_{ij}^{f(n)} = P\left[X_{n+k}=j \mid X_{n+k-1}\neq j, X_{n+k-2}\neq j, \cdots, X_{k+1}\neq j, X_k = i\right]$$

Now denote p_{ij}^{f*} as the probability of ever visiting state j from state i. Thus,

$$p_{ij}^{f*} = \sum_{n=1}^{\infty} p_{ij}^{f(n)}$$

If $p_{ii}^{f*} = 1$ then state i is called persistent. If a state i is persistent, then the expected

return time to that state is called μ_i, and,

$$\mu_i = \sum_{n=1}^{\infty} n p_{ii}^{f(n)}$$

If $\mu_i < \infty$, then the state i is positive persistent. Two states that intercommunicate will be of the same type in regard to periodicity and persistency. Thus, in an irreducible Markov Chain, all states are of the same type.

At this point, the renewal theorem should be mentioned, since important properties of Markov chains follow from it. Let $\{a_i\}$, $\{b_i\}$, and $\{u_i\}$ be sequences of non-negative real numbers such that

$$\sum_{i=0}^{\infty} a_i = 1$$

$$\sum_{i=0}^{\infty} b_i < \infty$$

$$-\infty < x \le u_i \le y < \infty \qquad \forall\, i$$

$$GCD\,\{\, i : i > 0 \,\} = 1$$

$$u_n - \sum_{i=0}^{n} a_{n-i} u_i = b_n \qquad n = 1,2,3, \cdots$$

Then

$$\lim_{n \to \infty} u_n = \begin{cases} \dfrac{\sum_{i=0}^{\infty} b_i}{\sum_{i=1}^{\infty} i a_i} & \text{if} \quad \sum_{i=1}^{\infty} i a_i < \infty \\ 0 & \text{if} \quad \sum_{i=1}^{\infty} i a_i = \infty \end{cases}$$

The following theorem about Markov chains can be stated as a consequence of the renewal theorem. For an irreducible, persistent, aperiodic Markov chain with transition matrix $T = [p_{ij}]$,

$$\lim_{n\to\infty} p_{ij}^{(n)} = \begin{cases} \dfrac{1}{\sum_{k=0}^{\infty} kp^{f(k)_{jj}}} & \text{if} \quad \sum_{k=0}^{\infty} kp^{f(k)_{jj}} < \infty \\ 0 & \text{if} \quad \sum_{k=0}^{\infty} kp^{f(k)_{jj}} = \infty \end{cases}$$

If the Markov chain is further restricted to being positive persistent, then the theorem is simplified to

$$\lim_{n\to\infty} p_{ij}^{(n)} = \frac{1}{\sum_{k=0}^{\infty} kp^{f(k)_{jj}}}$$

The most significant thing about this theorem is that $p_{ij}^{(n)}$ in fact will converge to a constant value as n goes to infinity. This theorem can also be used to show that in any finite irreducible Markov chain, all the states are positive persistent.

The analysis so far has shown that the state transition probabilities will converge (and thus the state probabilities) for a finite, irreducible, aperiodic Markov chain. Theorems have also been developed to classify a given process, but they can be very difficult to prove. However, a more powerful technique can be developed starting with the Perron-Frobenius theorems on non-negative matrices.

Let A be a non-negative square matrix. Then

i) A has a positive dominant eigenvalue λ_0 with corresponding left eigenvector x_0 such that x_0 is non-negative and non-zero;

ii) if λ is any other eigenvalue of A, then $|\lambda| \leq \lambda_0$;

iii) if λ is an eigenvalue of A and $|\lambda| = |\lambda_0|$, then $\eta = \dfrac{\lambda}{\lambda_0}$ is a root of unity and $\eta^k \lambda_0$ is an eigenvalue of A for $k=1,2,\cdots$.

Let A be a non-negative square matrix such that A^m is positive for some m. Then

i) A has a positive eigenvalue λ_0 with corresponding left eigenvector x_0 such that x_0 is positive;

ii) if λ is any other eigenvalue of A, then $|\lambda| < \lambda_0$;

iii) λ_0 has multiplicity one.

These two theorems can be used directly to prove the following two theorems (mentioned in Chapter 4) about Markov chains.

Theorem 4.1

If T is the transition matrix for a finite Markov chain, then the multiplicity of the eigenvalue 1 is equal to the number of irreducible closed subsets of the chain.

Theorem 4.2

If T is the transition matrix for an irreducible Markov chain with period d, then the dth roots of unity are eigenvalues of T.

Appendix C

Computer Program for Code Analysis

The following pages contain a source listing of the C program used to evaluate $d_{\min}$ and c for each of the codes, both with and without the channel. The program was run using the Lattice C compiler on an Amdhal 5870 computer in a JCL (TSO) environment.

```
//          JOB  ',,TIME=35M'
// EXEC C370CLG
//C.SYSIN  DD  *

#define MAXSTATES      24
#define MAXARCS        48
#define MAXPATHS      512
#define MAXDEPTH        8
#define BITSIN          1  /* Bits needed to represent an input symbol */
#define BITSOUT         2  /* Bits needed for an output symbol         */
#define SIFTIN          1  /* Bit mask to sift a single input symbol   */
#define SIFTOUT         3  /* Bit mask to sift a single output symbol  */
#define M               1  /* Parameter m of (d,k,m,n)                 */
#define N               2  /* Parameter n of (d,k,m,n)                 */
#define INFNTY 0x7FFFFFFF  /* For initializing best path to infinity   */
#define OK              1  /* Loop control comparator                  */


main()

 /*************************************************************************/
 /*  Main program accepts a trellis or state diagram (in the form of      */
 /*  an edge or adjacency list) and finds the minimum distance paths      */
 /*  for all starting and ending nodes over a specified range of          */
 /*  depths.  The probability coefficient (c) is calculated for all       */
 /*  paths whose distance matches dfree.  These are summed and            */
 /*  the final coefficient output.  Once all the paths at distance d      */
 /*  are found, the paths for distance d+1 are found by "growing" all     */
 /*  the d paths, and repositioning then to the appropriate end node.     */
 /*                                                                       */
 /*  INPUT:#Nodes,#Arcs,#Input symbols,#Output symbols                    */
 /*        #distinct output sequences,output sequencs                     */
 /*        Edge or adjacency list of trellis (state diagram)              */
 /*        Steady-state node probability for each node                    */
 /*  OUTPUT:Echo of input                                                 */
 /*         dmin,#path with dmin,probability of paths for each start      */
 /*             and end state, for each depth                             */
 /*         Overall coefficient c for dfree                               */
 /*************************************************************************/

{
int x;
float coeffc,evaluate();

x=getlist();                                  /* read the inputs          */
coeffc=evaluate(1,6,2);                       /* startdepth,enddepth,dfree */
printf("\fCoefficient of the error function is %.10f",coeffc);
return OK;
}

 /*************************************************************************/
 /*  The following information is accessed by almost all functions,       */
 /*  thus they are stored as global variables (for the functions).        */
 /*************************************************************************/

unsigned int from[MAXARCS];     /* starting node of arc          */
unsigned int to[MAXARCS];       /* ending node of arc            */
unsigned int in[MAXARCS];       /* input associated with arc     */
unsigned int out[MAXARCS];      /* output associated with arc    */
unsigned int statenum;          /* #states in trellis            */
unsigned int arcnum;            /* #arcs in trellis              */
unsigned int pathsout;          /* #paths exiting each node      */
float prob[MAXSTATES];          /* steady state probabilities    */
```

```
650. getlist()              /* get and print arclst (from,to,in,out) */
660.
670.  /*************************************************************************/
680.  /*  Function to read all necessary inputs (trellis and parameters).  */
690.  /*                                                                       */
700.  /*  INPUT:None (except external)                                         */
710.  /*  OUTPUT:Stores trellis and parameter info in global variables      */
720.  /*************************************************************************/
730.
740. {
750.    unsigned int x,y,xx,yy;                          /* dummy variables   */
760.    unsigned int s,i;                                /* loop variables    */
770.    unsigned int arc,outnum,chkout[20];
780.
790.    scanf("%d %d %*d %*d %*d",&statenum,&arcnum);
800.    scanf("%d",&outnum);
810.    printf("\n\nThe %d valid outputs are:",outnum);
820.    for (i=0;i<outnum;i++) {
830.       printf(" ");
840.       scanf("%x",&x);
850.       for (xx=0,y=0;y<N;y++,x>>=4)          /* ouputs read symbol by   */
860.          xx=(xx<<BITSOUT)+(x&SIFTOUT);      /* symbol in reverse order */
870.       for (y=0;y<N;y++,xx>>=BITSOUT) {
880.          chkout[i]=(chkout[i]<<BITSOUT)+(xx&SIFTOUT);
890.          printf("%x",xx&SIFTOUT);
900.       }
910.    }
920.    printf("\n\n\nArc From To In Out\n--- ---- -- -- ---");
930.    for (arc=0;arc<arcnum;arc++) {
940.       scanf("%d %d %x %x",&from[arc],&to[arc],&x,&y);
950.       printf("\n%2d  %3d %3d  ",arc,from[arc],to[arc]);
960.       for (xx=0,i=0;i<M;i++,x>>=4)
970.          xx=(xx<<BITSIN)+(x&SIFTIN);
980.       for (yy=0,i=0;i<N;i++,y>>=4)
990.          yy=(yy<<BITSOUT)+(y&SIFTOUT);
1000.      for (i=0;i<M;i++,xx>>=BITSIN) {
1010.         in[arc]=(in[arc]<<BITSIN)+(xx&SIFTIN);
1020.         printf("%x",xx&SIFTIN);
1030.      }
1040.      printf(" ");
1050.      for (i=0;i<N;i++,yy>>=BITSOUT) {
1060.         out[arc]=(out[arc]<<BITSOUT)+(yy&SIFTOUT);
1070.         printf("%x",yy&SIFTOUT);
1080.      }
1090.   }
1100.   printf("\n\n\nState  Probability\n-----  -----------");
1110.   for (s=0;s<statenum;s++) {
1120.      scanf("%f",&prob[s]);
1130.      printf("\n%3d      %f",s,prob[s]);
1140.   }
1150.   pathsout=1<<M;
1160.   printf("\n\nThere are %d paths out of each state.",pathsout);
1170.   return OK;
1180. }
1190.
1200. float evaluate(mind,maxd,dfree)
1210.
1220.  /*************************************************************************/
1230.  /*  Evaluate the paths in the trellis from depth mind to max d, for  */
1240.  /*  all start and end nodes.  Sum coefficents for dfree paths.       */
1250.  /*                                                                   */
1260.  /*  INPUT:mind - minimum depth                                       */
1270.  /*        maxd - maximum depth                                       */
1280.  /*        dfree - sum for paths with distance=dfree                  */
```

```
1290.   /*  OUTPUT:return coefficient c, probability of taking dfree path    */
1300.   /*********************************************************************/
1310.
1320.  unsigned int mind,maxd,dfree;
1330.  {
1340.     unsigned int x;
1350.     unsigned ss,es,ed;          /* start state, end state, end depth   */
1360.     float coeff=0,search();
1370.
1380.     for (ss=0;ss<statenum;ss++) {
1390.        printf("\fStarting state is: %2d\n\n Endstate:      ",ss);
1400.        for (es=0;es<statenum;printf(" %5d",es++));
1410.        x=initplist(mind,ss);
1420.        if (mind<=1)
1430.           coeff+=search(1,dfree,ss);    /* find paths of depth 1       */
1440.        for (ed=2;ed<=maxd;ed++) {
1450.           x=extend(ed);                 /* extend ("grow") the paths   */
1460.           coeff+=search(ed,dfree,ss);   /* find paths for other depths */
1470.        }
1480.     }
1490.     return coeff;
1500.  }
1510.
1520.   /*********************************************************************/
1530.   /*  Variables that are global for following functions.  2 complete   */
1540.   /*  pathlists with pointers to each one so the current one can be    */
1550.   /*  flip-flopped as the paths are grown.  Stored by end state, path  */
1560.   /*  index, and edge for each depth.                                  */
1570.   /*********************************************************************/
1580.
1590.  unsigned int plista[MAXSTATES][MAXPATHS][MAXDEPTH];
1600.  unsigned int plistb[MAXSTATES][MAXPATHS][MAXDEPTH];
1610.  unsigned int *nplist=&plista[0][0][0],*oplist=&plistb[0][0][0];
1620.  unsigned int pna[MAXSTATES],pnb[MAXSTATES],*npn=pna,*opn=pnb;
1630.
1640.  initplist(mind,ss)
1650.
1660.   /*********************************************************************/
1670.   /*  Initialize the lists by growing in to the minimum depth.  This   */
1680.   /*  list is only good for the given starting state.                  */
1690.   /*                                                                   */
1700.   /*  INPUT:mind - minimum depth                                       */
1710.   /*        ss   - start state                                         */
1720.   /*  OUTPUT:initializes the global pathlists and associated pointers  */
1730.   /*********************************************************************/
1740.
1750.  unsigned int mind,ss;
1760.  {
1770.     unsigned int arc,d,x;
1780.     unsigned int es;            /* end state                           */
1790.     unsigned int top;           /* last edge originating from given node */
1800.
1810.     for (es=0;es<statenum;opn[es++]=0);
1820.     for (arc=0;from[arc]!=ss;arc++);
1830.     for (top=arc+pathsout;arc<top;arc++)
1840.        *(oplist+(to[arc]*MAXPATHS+opn[to[arc]]++)*MAXDEPTH+1)=arc;
1850.     for (d=2;d<mind;x=extend(d++));
1860.     return OK;
1870.  }
1880.
1890.  extend(ed)
1900.
1910.   /*********************************************************************/
1920.   /*  Functions "grows" previously grown paths up to a depth ed        */
```

```
1930.  /*                                                                    */
1940.  /*  INPUT:ed - depth to "grow" paths to                                */
1950.  /*  OUTPUT:puts "grown" paths into global pathlists                    */
1960.  /*********************************************************************/
1970.
1980. unsigned int ed;
1990. {
2000.    unsigned int *otemp,*ntemp;        /* old, new pathlist pointers    */
2010.    unsigned int arc,i,j,d;
2020.    unsigned int es;                   /* end state                     */
2030.    unsigned int top;                  /* last edge originating from given node */
2040.
2050.    for (es=0;es<statenum;npn[es++]=0);
2060.    for (es=0;es<statenum;es++) {
2070.       for (arc=0;from[arc]!=es;arc++);
2080.       for (top=arc+pathsout;arc<top;arc++) {
2090.          for (j=npn[to[arc]],i=0;i<opn[es];i++) {
2100.             otemp=oplist+(es*MAXPATHS+i)*MAXDEPTH;
2110.             ntemp=nplist+(to[arc]*MAXPATHS+i+j)*MAXDEPTH;
2120.             for (d=1;d<ed;d++)
2130.                *(ntemp+d)=*(otemp+d);
2140.             *(ntemp+ed)=arc;
2150.          }
2160.          npn[to[arc]]+=i;
2170.       }
2180.    }
2190.    ntemp=nplist;                      /* switch the old and new lists */
2200.    nplist=oplist;                     /* by switching the pointers    */
2210.    oplist=ntemp;
2220.    ntemp=npn;
2230.    npn=opn;
2240.    opn=ntemp;
2250.    return OK;
2260. }
2270.
2280. float search(ed,dfree,ss)
2290.
2300.  /*********************************************************************/
2310.  /*  search all the paths for disjoint paths (with common end nodes:  */
2320.  /*  all start nodes are the same since they were all grown from the  */
2330.  /*  same node, and all paths are depth ed).  Find paths with min.    */
2340.  /*  distance.  If that distance = dree, find the probability of      */
2350.  /*  taking that path, and associated errors.                         */
2360.  /*                                                                   */
2370.  /*  INPUT:ed   - depth of the paths                                  */
2380.  /*        dree - sum paths with distance dree                        */
2390.  /*        ss   - starting state of all paths                         */
2400.  /*  OUTPUT:return coefficients for all dree paths                    */
2410.  /*         :print #minimum distance paths, minimum distance          */
2420.  /*********************************************************************/
2430.
2440. unsigned int ed,dfree,ss;
2450. {
2460.    unsigned int es;                   /* end state                     */
2470.    unsigned int pa,pb;                /* paths to be compared a & b    */
2480.    unsigned d,dis;
2490.    unsigned int mindis[MAXSTATES];    /* array of min. distances       */
2500.    unsigned int sumerr[MAXSTATES];    /* array of errors               */
2510.    unsigned int outa[MAXDEPTH];       /* output of path a to depth d   */
2520.    unsigned int mdpn[MAXSTATES];
2530.    unsigned int ina,inb;              /* inputs of paths a & b         */
2540.    unsigned int probpath;             /* probability of taking path    */
2550.    unsigned int nomatch;              /* are paths disjoint?           */
2560.    unsigned int err,evec;             /* errors (XOR of two paths)     */
```

```
2570.     unsigned int count,counted;         /* count path only once          */
2580.     unsigned int *tempa,*tempb;
2590.     float coeff=0;
2600.
2610.     for (probpath=1,d=0;d<ed;d++,probpath*=pathsout);
2620.     for (es=0;es<statenum;es++) {
2630.        mindis[es]=INFNTY;
2640.        mdpn[es]=0;
2650.        sumerr[es]=0;
2660.        for (pa=0;pa<opn[es];pa++) {                    /* select path a   */
2670.           tempa=oplist+(es*MAXPATHS+pa)*MAXDEPTH;
2680.           count=0;
2690.           counted=0;
2700.           for (ina=0,d=1;d<=ed;d++) {
2710.              outa[d]=out[tempa[d]];                    /* find its output */
2720.              ina=(ina<<M*BITSIN)+in[tempa[d]];         /* find its input  */
2730.           }
2740.           for (pb=0;pb<opn[es];pb++) {                /* select path b   */
2750.              if (pa!=pb) {                            /* not same path?  */
2760.                 tempb=oplist+(es*MAXPATHS+pb)*MAXDEPTH;
2770.                 for (dis=0,nomatch=1,inb=0,d=1;d<=ed&&nomatch;d++) {
2780.                    nomatch=to[tempa[d]]¬to[tempb[d]];     /* disjoint? */
2790.                    for (evec=outa[d]¬out[tempb[d]];evec;evec>>=BITSOUT)
2800.                       dis+=((evec&SIFTOUT)>2)?4:((evec&SIFTOUT)>0)?1:0;
2810.                    inb=(inb<<M*BITSIN)+in[tempb[d]];
2820.                 }
2830.                 if (d>ed&&dis<=mindis[es]) {          /* min. dist path? */
2840.                    for (err=0,evec=ina¬inb;evec;evec>>=BITSIN)
2850.                       err+=evec&SIFTIN;               /* find errors     */
2860.                    if (dis<mindis[es]) {              /* new min. dist?  */
2870.                       mindis[es]=dis;
2880.                       sumerr[es]=err;
2890.                       mdpn[es]=1;
2900.                       counted=1;
2910.                    }
2920.                    else                               /* same min. dist? */
2930.                       sumerr[es]+=err;
2940.                       count=1;
2950.                 }
2960.              }
2970.           }
2980.           if (count&&!counted)
2990.              mdpn[es]++;
3000.        }
3010.        if (mindis[es]==dfree)
3020.           coeff+=prob[ss]*sumerr[es]/probpath/M;       /* add to coeff */
3030.     }
3040.     printf("\n\nDepth is %d",ed);
3050.     printf("\n Number of paths:");
3060.     for (es=0;es<statenum;printf(" %5d",opn[es++]));
3070.     printf("\nMin. dist. paths:");
3080.     for (es=0;es<statenum;printf(" %5d",mdpn[es++]));
3090.     printf("\n   Sum of errors:");
3100.     for (es=0;es<statenum;printf(" %5d",sumerr[es++]));
3110.     printf("\nMinimum distance:");
3120.     for (es=0;es<statenum;es++)
3130.        printf(" %5d",mindis[es]>9999?9999:mindis[es]);
3140.     return coeff;
3150. }
3160.
3170. /*
3180. //GO.SYSIN  DD  *
3190. ++EMBED YUNS27 NOSEQ PERF='++'
```

Appendix D

Computer Program for Simulation

The following pages contain a source listing of the C program used to simulate the bit error performance of each of the RLL codes in a magnetic channel with AWGN. The program was run using the Lattice C compiler on an Amdhal 5870 computer in a JCL (TSO) environment.

```
//          JOB  ',,TIME=100M'
// EXEC C370CLG
//C.SYSIN  DD  *

#define MAXPATHIN       8
#define MAXSTATES      20
#define MAXARCS        50
#define MAXOUTNUM      16
#define BITSIN          1  /* Bits needed to represent an input symbol */
#define BITSOUT         2  /* Bits needed for an output symbol         */
#define SIFTIN          1  /* Bit mask to sift a single input symbol   */
#define SIFTOUT         3  /* Bit mask to sift a single output symbol  */
#define M               1  /* Parameter m of (d,k,m,n)                 */
#define N               2  /* Parameter n of (d,k,m,n)                 */
#define NO           1024  /* Variance of the noise                    */
#define TRUNCMS         0  /* #word in survivor sequence               */
#define TRUNCLS        30  /* #bits per word in survivor sequence      */
#define BITLEN         32  /* Bit lenght of machine                    */
#define INFNTY 0x7FFFFFFF  /* Infinity                                 */
#define BIG    0x40000000  /* A big number, but not infinity           */
#define OK              1  /* Loop and function control                */

#include <stdlib.h>
#include <math.h>

int channel[3];          /* The pulse transmitted for an output symbol */
unsigned int statenum,arcnum;      /* #states,#arcs                    */

main()

 /*************************************************************************/
 /*  Main program accepts a trellis or state diagram (in the form of      */
 /*  and edge or adjacency list) and simulates the coding process,        */
 /*  channel (magnetic, in this case), and decoding process in the        */
 /*  Viterbi's algorithm.  The error rate is found for a range of         */
 /*  signal to noise ratios (SNR).                                        */
 /*                                                                       */
 /*  INPUT:#Nodes,#Arcs,#Input symbols,#Output symbols,start state        */
 /*        #distinct output sequences,output sequencs                     */
 /*        Edge or adjacency list of trellis (state diagram)              */
 /*        Steady-state node probability for each node                    */
 /*  OUTPUT:Echo of input                                                 */
 /*         dmin,#path with dmin,probability of paths for each start      */
 /*             and end state, for each depth                             */
 /*         Bit error rate for several SNR                                */
 /*************************************************************************/

{
unsigned int frame;          /* frame (stage of trellis) number          */
unsigned startstate,x;
unsigned primitive;          /* primitive element of GF(2**32)           */
int errors,testlen;
int signal;                  /* signal strength                          */

scanf("%d %d %*d %*d %d",&statenum,&arcnum,&startstate);
x=getlist();                                          /* read trellis */
x=initialize(startstate,1);
x=findchkarc();                                /* find arcs to be checked */
printf("\f");
for (signal=3072;signal<6145;signal+=512) {    /***********************/
   channel[1]=signal;                          /* Specify signal size */
   channel[2]=-signal;                         /* for each symbol.    */
   srand(rand());                              /* Specify seed.       */
   testlen=(signal<5121)?10000:100000;         /* Specify # of frames */
```

```
    primitive=rand()+1;                                  /***********************/
    x=initialize(startstate,primitive);
    for (frame=0;frame<(TRUNCMS*BITLEN+TRUNCLS)/BITSIN/M;frame++) {
       x=output();
       x=distance();                          /* Execute code until survivor */
       x=viterbi();                           /* sequence registers are full */
    }
    for (errors=0,frame=0;frame<testlen;frame++) {
       x=output();
       x=distance();                          /* Execute code and count the  */
       errors+=viterbi();                     /* errors.                     */
    }
    printf("\n\n\nS/N Ratio: %.2f",(float)signal/NO);
    printf("\n   Errors: %d\n   Frames: %d",errors,frame);
}
return OK;
}

 /*********************************************************************/
 /* Start of functions:                                               */
 /* getlist: gets and prints valid outputs (outnum,chkout[]) and      */
 /*          arclist (from[],to[],in[],out[]).                        */
 /* findchkarc: finds the paths to check for each state and puts the  */
 /*             number of them in pthinum[], and the arcs themselves  */
 /*             into chkarc[state][index] (pointed to by chkarcp[]).  */
 /* output: finds the psuedo-random binary input sequence (M bits per */
 /*         frame) from GF(2**31) using the minimal polynomial        */
 /*         x**31+x**3+1, and finds the output (outpt). These symbols */
 /*         are then translated to channel symbols (channel[]) and    */
 /*         AWGN of energy NO is added to the N symbols.              */
 /* distance: the distance of each valid output (chkout[]) is         */
 /*           calculated from the recieved signal (r[]) and put into  */
 /*           dist[].                                                 */
 /* viterbi: performs Vitirbi's algorithm for one trellis frame.  The */
 /*          updated metrics are calculated from oldmetp[] and stored */
 /*          in newmetp[].  The best arc is stored in decisn and the  */
 /*          updated survivor sequence (newssp[][]) is calculated     */
 /*          from the input represented by decisn (in[decisn]) and    */
 /*          the old survivor sequence (oldssp[][]).  The best state  */
 /*          metric is found (minmet) and the survivor sequence of    */
 /*          that state is used to decide on the input x frames ago   */
 /*          (newssp[minste][]).  Normalization to prevent overflow   */
 /*          is carried out.                                          */
 /* initialize: initializes the pointers and arrays of pointers to    */
 /*             the correct locations. Initializes the state metrics  */
 /*             and the polynomial (gf).                              */
 /* print: will print information about the process at each frame, if */
 /*        necessary.  If the program is working properly, it will    */
 /*        not normally be used.  0 selects no decoded input and      */
 /*        error count, while any other number does.                  */
 /*********************************************************************/

 /*********************************************************************/
 /*  Global variables                                                 */
 /*********************************************************************/

int *newmetp,*oldmetp;                  /* new & old state metric pointers */
int meta[MAXSTATES],metb[MAXSTATES];    /* state metrics                   */
int dist[0x22],r[3];
unsigned int to[MAXARCS];               /* ending node of arc              */
unsigned int in[MAXARCS];               /* input associated with arc       */
unsigned int out[MAXARCS];              /* output associated with arc      */
unsigned int statenum;                  /* #states in trellis              */
unsigned int chkarc[MAXSTATES][MAXPATHIN],*chkarcp[MAXSTATES];
```

```
1290.                                   /* list of arcs to check for each state */
1300. unsigned int **newssp,**oldssp;     /* pointers to pointers                 */
1310. unsigned int *ssap[MAXSTATES];      /* pointers to old & new survivor       */
1320. unsigned int *ssbp[MAXSTATES];      /* sequences.                           */
1330. unsigned int ss[2][MAXSTATES][TRUNCMS+1];  /* survivor sequences           */
1340. unsigned int ins[TRUNCMS+1];                /* input sequence               */
1350. unsigned int pthinum[MAXSTATES];            /* #path into each state        */
1360. unsigned int chkout[MAXOUTNUM],;            /* outputs to check             */
1370. unsigned int minste,inste,inpt,outpt,outnum;
1380. unsigned int gf;                           /* prime polynomial of order 31  */
1390.
1400. getlist()                          /* get and print arclst (from,to,in,out) */
1410. {
1420.    unsigned int x,y,xx,yy,i,arc;
1430.
1440.    scanf("%d",&outnum);
1450.    printf("\n\nThe %d valid outputs are:",outnum);
1460.    for (i=0;i<outnum;i++) {
1470.       printf(" ");
1480.       scanf("%x",&x);
1490.       for (xx=0,y=0;y<N;y++,x>>=4)
1500.          xx=(xx<<BITSOUT)+(x&SIFTOUT);
1510.       for (y=0;y<N;y++,xx>>=BITSOUT) {
1520.          chkout[i]=(chkout[i]<<BITSOUT)+(xx&SIFTOUT);
1530.          printf("%x",xx&SIFTOUT);
1540.       }
1550.    }
1560.    printf("\n\n\nArc From To In Out\n--- ---- -- -- ---\n");
1570.    for (arc=0;arc<arcnum;arc++) {
1580.       scanf("%d %d %x %x",&from[arc],&to[arc],&x,&y);
1590.       printf("\n%2d  %3d %3d  ",arc,from[arc],to[arc]);
1600.       for (xx=0,i=0;i<M;i++,x>>=4)
1610.          xx=(xx<<BITSIN)+(x&SIFTIN);
1620.       for (yy=0,i=0;i<N;i++,y>>=4)
1630.          yy=(yy<<BITSOUT)+(y&SIFTOUT);
1640.       for (i=0;i<M;i++,xx>>=BITSIN) {
1650.          in[arc]=(in[arc]<<BITSIN)+(xx&SIFTIN);
1660.          printf("%x",xx&SIFTIN);
1670.       }
1680.       printf(" ");
1690.       for (i=0;i<N;i++,yy>>=BITSOUT) {
1700.          out[arc]=(out[arc]<<BITSOUT)+(yy&SIFTOUT);
1710.          printf("%x",yy&SIFTOUT);
1720.       }
1730.    }
1740.    return OK;
1750. }
1760.
1770. findchkarc()              /* Find the paths to check for each state */
1780. {
1790.    unsigned int arc,path,state;
1800.
1810.    for (arc=0;arc<arcnum;arc++)
1820.       chkarc[to[arc]][pthinum[to[arc]]++]=arc;
1830.    printf("\n\n\nState #Pathsin Arcnumbers\n----- -------- ----------");
1840.    for (state=0;state<statenum;state++) {
1850.       printf("\n  %2d     %2d    ",state,pthinum[state]);
1860.       for (path=0;path<pthinum[state];path++)
1870.          printf(" %2d",*(chkarcp[state]+path));
1880.    }
1890.    return OK;
1900. }
1910.
1920. output()                                  /* find the output of the channel */
```

```
1930.
1940. #define NORM 0x7FFF
1950. #define PI   3.14159265358979323846264338328
1960. {
1970.    unsigned int i,temp,arc;
1980.    double ra,u;
1990.
2000.    inpt=0;
2010.    for (i=0;i<M;i++) {                          /* find the input       */
2020.       temp=(gf^gf>>3^gf>>31)&1;                 /* psuedo-random binary */
2030.       inpt=(inpt<<BITSIN)+temp;                 /* sequence.            */
2040.       gf=(gf<<1)+temp;
2050.    }
2060.    for (arc=0;(from[arc]!=inste)||(in[arc]!=inpt);arc++);
2070.    inste=to[arc];
2080.    for (i=TRUNCMS;i;i--)
2090.       ins[i]=(ins[i]<<BITSIN*M)+(ins[i-1]>>BITLEN-BITSIN*M);
2100.    ins[0]=(ins[0]<<BITSIN*M)+inpt;
2110.    outpt=out[arc];                                 /* find the output */
2120.    for (temp=outpt,i=0;i<N;) {                     /* find AWGN       */
2130.       ra=sqrt(-2*NO*NO*log((double)(rand()+1)/(NORM+1)));
2140.       u=(double)rand()/NORM*2*PI;
2150.       r[i++]=(int)(ra*cos(u))+channel[temp&SIFTOUT];
2160.       temp>>=BITSOUT;
2170.       if (i<N) {
2180.          r[i++]=(int)(ra*sin(u))+channel[temp&SIFTOUT];
2190.          temp>>=BITSOUT;
2200.       }
2210.    }
2220.    return OK;
2230. }
2240.
2250. distance()                  /* find the distance to all possible outputs */
2260. {
2270.    int i,j,x,y;
2280.    for (i=0;i<outnum;i++) {
2290.       x=chkout[i];
2300.       dist[x]=0;
2310.       for (j=0;x;x>>=BITSOUT) {
2320.          y=channel[x&SIFTOUT];
2330.          dist[chkout[i]]+=((y>0)?1:(y<0)?-1:0)*(-r[j++]+(y>>1));
2340.       }
2350.    }
2360.    return OK;
2370. }
2380.
2390. viterbi()                                  /* perform Viterbi's algorithm */
2400. {
2410.    unsigned int arc,path,state,i,decisn,errorvec,fste,**tmp;
2420.    int otrmet,minmet,*temp,errors;
2430.
2440.    minmet=INFNTY;
2450.    for (state=0;state<statenum;state++) {
2460.       *(newmetp+state)=INFNTY;
2470.       for (path=0;path<pthinum[state];) {          /* check all input */
2480.          arc=*(chkarcp[state]+path++);             /* paths.          */
2490.          otrmet=*(oldmetp+from[arc])+dist[out[arc]];
2500.          if (otrmet<*(newmetp+state)) {            /* find best path  */
2510.             *(newmetp+state)=otrmet;
2520.             decisn=arc;
2530.             if (otrmet<minmet) {                   /* find best state */
2540.                minmet=otrmet;                      /* to output surv. */
2550.                minste=state;                       /* seq.            */
2560.             }
```

```
2570.           }
2580.        }
2590.        fste=from[decisn];
2600.        for (i=TRUNCMS;i;i--)                          /* update surv. seq. */
2610.           *(*(newssp+state)+i)=(*(*(oldssp+fste)+i)<<BITSIN*M)+
2620.                                (*(*(oldssp+fste)+i-1)>>BITLEN-BITSIN*M);
2630.        **(newssp+state)=(**(oldssp+fste)<<BITSIN*M)+in[decisn];
2640.     }
2650.     errorvec=(*(*(newssp+minste)+TRUNCMS)^ins[TRUNCMS])>>TRUNCLS;
2660.     for (errors=0,i=0;i<M;i++,errorvec>>=BITSIN)       /* count errors */
2670.        errors+=((errorvec&SIFTIN)==0)?0:1;
2680.     for (state=0;state<statenum;)
2690.        *(newmetp+state++)-=minmet;
2700.     temp=newmetp;                                     /* flip-flop state */
2710.     newmetp=oldmetp;                                  /* metrics.        */
2720.     oldmetp=temp;
2730.     tmp=newssp;
2740.     newssp=oldssp;
2750.     oldssp=tmp;
2760.     return errors;
2770. }
2780.
2790. initialize(startstate,primitive)                    /* initialize pointers */
2800.                                                     /* and primitive       */
2810. unsigned int startstate,primitive;
2820.
2830. {
2840.    unsigned int state;
2850.
2860.    newssp=ssap;
2870.    oldssp=ssbp;
2880.    newmetp=meta;
2890.    oldmetp=metb;
2900.    for (state=0;state<statenum;state++) {
2910.       chkarcp[state]=chkarc[state];
2920.       ssap[state]=ss[0][state];
2930.       ssbp[state]=ss[1][state];
2940.       metb[state]=(state==startstate)?0:BIG;
2950.    }
2960.    inste=startstate;
2970.    gf=primitive;
2980.    return OK;
2990. }
3000.
3010. print(choice,frame,errors)                          /* print pertinant info. */
3020.
3030. int choice,errors;
3040. unsigned int frame;
3050. {
3060.    int i;
3070.
3080.    printf("\n%3d ",frame);
3090.    if (choice) {
3100.       for (i=M-1;i>=0;i--)
3110.       printf("%x",*(*(oldssp+minste)+TRUNCMS)>>TRUNCLS+i*BITSIN&SIFTIN);
3120.    }
3130.    else {
3140.       for (i=0;i<=M;i++)
3150.          printf(" ");
3160.    }
3170.    printf(" ");
3180.    for (i=M-1;i>=0;i--)
3190.       printf("%x",inpt>>i*BITSIN&SIFTIN);
3200.    printf(" ");
```

```
3210.     for (i=N-1;i>=0;i--)
3220.        printf("%x",outpt>>i*BITSOUT&SIFTOUT);
3230.     for (i=N-1;i>=0;)
3240.        printf(" %4d",r[i--]);
3250.     for (i=0;i<7;)
3260.        printf(" %5d",*(oldmetp+i++));
3270.     printf("   ,%2d",minste);
3280.     if (choice) printf("   ,%d",errors);
3290.     return OK;
3300. }
3310.
3320. /*
3330. //GO.SYSIN  DD  *
3340. ++EMBED DATAIBMBC NOSEQ PERF='++'
```