New Capacity-Approaching Codes for Run-Length-Limited Channels

A Thesis
Presented to
The Academic Faculty

by

Yogesh Sankarasubramaniam

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in Electrical and Computer Engineering

School of Electrical and Computer Engineering
Georgia Institute of Technology
May 2006

New Capacity-Approaching Codes for Run-Length-Limited Channels

Approved by:

Dr. Steven W. McLaughlin, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Thomas P. Barnwell
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. John R. Barry
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Prasad Tetali
School of Mathematics
Georgia Institute of Technology

Dr. Faramarz Fekri
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: March 30, 2006

To my parents

ACKNOWLEDGEMENTS

I have cherished the opportunity to work with Dr. Steven W. McLaughlin. His constant encouragement and enthusiasm have made us all feel comfortable. I am very grateful for his guidance, and wish him and his family the very best in coming years.

My thanks to Dr. John R. Barry, Dr. Faramarz Fekri, Dr. Thomas P. Barnwell and Dr. Prasad Tetali for being part of my dissertation committee, and to Seagate Research for their gracious support; especially to Erozan M. Kurtas and Alexander V. Kuznetsov for several helpful discussions and insights.

Other not-so-insightful, but nevertheless entertaining discussions kept the flame burning through these long years. They are attributed to colleagues and labmates, past and present. My survival has also depended on several friends, who have been a source of inspiration in their own ways. My fond recollections also, of a year spent at Georgia Tech Lorraine - thanks again to Steve - where I did everything else but learn French. Many thanks to the ever-friendly Josyane Roschitz.

Half the world away, my parents have been my biggest strength. During all these years far from home, they have still managed to convey their love and affection every single day. That I have come this far is testament to their resolve and dedication. My efforts have merely followed suit.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

A communication channel is said to be Run-Length-Limited (RLL), if it imposes constraints on runs of consecutive input symbols. RLL channels are found in digital recording systems like the Hard Disk Drive (HDD), Compact Disc (CD), and Digital Versatile Disc (DVD). For a binary recording medium, a typical run-length constraint requires that successive ones in the stored bit sequence be separated by at least d, and at most k consecutive zeros, for non-negative integers d and k. The d constraint is used to regulate inter-symbol interference and the k constraint is important for timing recovery. Such a constrained binary sequence is referred to as a (d, k) sequence. A (d, k) code is defined as an invertible mapping between unconstrained binary sequences and (d, k) sequences. A (d, k) code can use either fixed-rate or variable-rate encoding.

Assuming a noise-free channel, it is known that the encoding rate of any (d, k) code can only be as high as the *Shannon capacity*. Codes that achieve this limit are said to be *optimal*. In the first half of this thesis, we discuss the design of two variable-rate codes that are optimal for certain classes of (d, k) constraints. We introduce the symbol sliding algorithm, which achieves capacity for $(d, 2d + 1)$, (d, ∞), $(d, d + 1)$, and $(2, 4)$ constraints, and comes very close to capacity for all other values of d and k. Symbol sliding is based on the bit stuff algorithm [6], which generates simple and efficient codes for a wide range of constraints. Then, we construct a second class of optimal codes using interleaving. This method is applicable for all (d, k) constraints with $k - d + 1$ not prime. Of particular interest are $(d, d + 2^m - 1)$ constraints, $2 \leq m < \infty$, where the interleaving is especially simple. We conclude the discussion on variable-rate codes by presenting extensions to other RLL constraints.

In the second half of this thesis, we focus on fixed-rate constrained codes. Although variable-rate encoding can generate optimal and near-optimal RLL codes, practical designs require fixed-rate encoding. Several fixed-rate algorithms have been proposed over the years ([17] provides a comprehensive review). However, the design of high-rate codes with simple implementation continues to pose a challenge. From a practical standpoint, even small (1-2%) increases in code rate with simpler encoding/decoding can significantly impact the cost and performance of a digital recording system. Hence, despite the long history of (d, k) codes, there continues to be a need for low-complexity, fixed-rate encoding algorithms that achieve near-capacity rates. In this work, we propose the fixed-rate bit stuff (FRB) algorithm: a fixed-rate version of the variable-rate bit stuff algorithm, for the special class of $(0, k)$ constraints. The key to achieving high encoding rates with the FRB algorithm lies in a novel, iterative pre-processing of the fixed-length input sequence prior to bit stuffing. The encoder then inserts bits to produce a fixed-length output sequence. We provide detailed rate analysis for the proposed FRB algorithm, and derive upper and lower bounds on the asymptotic (in input block length) encoding rate. Our results suggest that near-capacity rates can be achieved by encoding in long, fixed-length, input and output blocks using the FRB algorithm. Then, we proceed to address several system issues, such as encoding complexity, encoding latency, effect of finite block-lengths, DC suppression and error propagation of the proposed FRB codes. We present a performance comparison with existing encoding schemes, and tabulate the FRB code parameters required to design rate 100/101 and rate 200/201, $(0, k)$ codes. We also extend the proposed fixed-rate encoding to $(0, G/I)$ constraints.

CHAPTER I

INTRODUCTION

A communication channel is said to be run-length-limited (RLL) if it imposes constraints on runs of consecutive input symbols. RLL channels have a long history dating back to Shannon's seminal work [46]. Over the years, they have gained in practical importance with the emergence of digital recording systems, namely Hard Disk Drives (HDDs), Compact Discs (CDs), and Digital Versatile Discs (DVDs). A typical run-length constraint used in these systems requires that successive ones in the stored bit sequence be separated by a run of consecutive zeros of length at least d bits, and at most k bits, for non-negative integers d and k. The d constraint is used to regulate inter-symbol interference and the k constraint is important for timing recovery. Such a constrained binary sequence is referred to as a (d, k) sequence.

An encoding algorithm provides an invertible mapping between unconstrained binary sequences and (d, k) sequences. The resulting (d, k) codes can either be variable-rate or fixed-rate. Variable-rate encoding refers to the fact that for a given, finite input block-length, the generated output block-length can vary depending on the actual input bits. For a fixed-rate code, all possible inputs of a given, finite length generate fixed-length outputs. Some variable-rate encoding algorithms for (d, k) constraints are given in [3],[6],[27],[31],[41]. A comprehensive review of fixed-rate codes can be found in [17].

With the assumption of a noise-free channel, the encoding rate, $R(d, k)$, of any (d, k) code is bounded above by the *Shannon capacity*, $C(d, k)$, given by [46]

$$C(d, k) = \log_2 \lambda_{d,k}, \tag{1}$$

where $\lambda_{d,k}$ is the largest, real root of the characteristic equation $H_{d,k}(z) = 1$, and $H_{d,k}(z)$ is the characteristic polynomial of the (d, k) constraint, given by

$$H_{d,k}(z) = \begin{cases} \sum_{j=d+1}^{k+1} z^{-j} & \text{when } k < \infty \\ z^{-1} + z^{-(d+1)} & \text{when } k = \infty. \end{cases} \tag{2}$$

The encoding efficiency, $E(d, k) = \dfrac{R(d, k)}{C(d, k)}$, measures how close the code is to capacity. A (d, k) code is said to be *optimal*, if it is 100% efficient. It can be shown that $C(d, k)$ is irrational, except in the trivial case of (d, k)=$(0, \infty)$. Thus, any (d, k) code which has an encoding rate of the form m/n, m and n being finite integers, is strictly sub-optimal. This includes all fixed-rate codes with finite input and output block lengths. Two existing near-optimal, fixed-rate encoding techniques are based on enumerative coding [10],[19],[43] and arithmetic coding [31],[58]. Some capacity-achieving variable-rate codes are outlined in [3],[6],[27].

Although variable-rate (d, k) codes have the potential to approach capacity, their very nature makes them unsuitable for use in most practical systems. For example, in magnetic and optical disk recording systems, data is written onto fixed-length track sectors, and hence fixed-rate codes are desired. Indeed, variable-rate codes do have some practical use, *e.g.*, frame synchronization with variable-length payloads in certain communication protocols [7], but they are few and far between. Thus, it is fair to say that as of this date, variable-rate codes are mainly of theoretical interest and fixed-rate codes find greater practical use. In this research, we pursue the design of both variable-rate and fixed-rate constrained codes that are based on very simple ideas. The proposed codes are said to be capacity-approaching, in the sense that the encoding rates are either equal to the constraint capacity or come very close to it.

First, we present two new capacity-achieving, variable-rate code constructions for (d, k) constraints. We introduce the symbol sliding algorithm (Chapter 4), which

achieves capacity for $(d, 2d+1)$, (d, ∞), $(d, d+1)$, and $(2, 4)$ constraints, and comes very close to capacity in other cases. It is based on bit stuffing [6],[7], a simple technique that generates efficient codes for a wide range of constraints. Then, we construct a second class of optimal codes based on interleaving (Chapter 5). The interleaving implementation is derived from a certain factorization of characteristic polynomials. This method is applicable for all (d, k) constraints with $k - d + 1$ not prime. Of particular interest are $(d, d + 2^m - 1)$ constructions, $2 \leq m < \infty$, where the interleaving is especially simple. Each of the variable-rate constructions: symbol sliding and interleaving, can be viewed as generalizations of existing algorithms: the bit flipping algorithm [3], and the bit stuff algorithm [6], respectively.

The second half of this research deals with fixed-rate constrained codes. There is a long history of fixed-rate (d, k) codes and they are part of virtually all magnetic and optical disk recording systems today. Several fixed-rate encoding algorithms have been proposed over the years (see [17] for a comprehensive review), with the design goal being two-fold: high encoding rate and simple implementation. However, the nonlinear nature of (d, k) sequences (they do not constitute a linear vector subspace) makes the design of near-optimal, fixed-rate (d, k) codes rather complex. This has meant that practical encoding algorithms have to strike a balance between the conflicting attributes of high encoding rate and simple implementation. Thus, there continues to be a need for low-complexity, fixed-rate encoding algorithms that achieve near-capacity rates.

These factors have motivated us to pursue the design of a fixed-rate version of the simple, variable-rate bit stuff algorithm [6]. We discuss in detail, the fixed-rate bit stuff (FRB) algorithm (Chapter 7), which is applicable for the special class of $(0, k)$ constraints. High encoding efficiency is achieved by iterative pre-processing of the fixed-length input sequence prior to bit stuffing. This has the effect of conforming the input sequence to subsequent bit insertions. The encoder then inserts bits to

produce a fixed-length output sequence. We present a detailed rate analysis for the FRB algorithm, and derive upper and lower bounds on the asymptotic (in input block length) encoding rate. These bounds are found to be very close to the average rate of the variable-rate bit stuff code. Hence, very high-rate $(0, k)$ codes can be constructed using the FRB algorithm by encoding in long, fixed-length input and output blocks.

The FRB algorithm compares favorably with enumerative [15],[[17],Chap.6],[19], and combinatorial [20],[60] encoding: two important existing methods to generate $(0, k)$ sequences. Specifically, the FRB encoding/decoding is simpler than enumeration, while achieving (asymptotically) similarly high encoding rates. The FRB encoding rates are also far greater than that of the combinatorial construction of Immink and Wijngaarden [20], at the cost of slightly higher encoding/decoding complexity.

In theory, the FRB algorithm thus provides an effective means to generate very high-rate $(0, k)$ sequences. However, integrating the FRB codes into a practical recording system raises several other questions. We address these system issues (Chapter 8), namely the encoding complexity, encoding latency, effect of finite block-lengths, DC suppression and error propagation. In particular, we discuss two possible implementations of the iterative pre-processing, with related tradeoffs between the number of computations and encoding latency. We also provide a detailed analysis of the effect of partial pre-processing (Chapter 7.4), which utilizes only a subset of the iterative pre-processing. While partial pre-processing allows simpler implementation and reduces encoding latency, it incurs a penalty on the encoding rate. Hence, quantifying this tradeoff becomes important in system design. Further rate penalties are also incurred while encoding in finite block-lengths. In theory, the asymptotic encoding rates of the FRB algorithm are very close to the $(0, k)$ capacity, but practical systems cannot encode in infinitely long blocks. We study the associated rate penalties, and it is seen that block lengths of a few thousand bits are required to design high-rate $(0, k)$ codes.

Another important issue with the long block codes of the FRB algorithm, is that of error propagation. The FRB algorithm does not code for possible errors caused by the recording channel. However, channel bit errors are inevitable. In fact, in the worst case, it is possible that the entire decoded sequence is in error due to a single channel bit error. Hence, it becomes important to study the effect of channel bit errors on code performance. One existing technique to combat error-propagation is the use of a reverse concatenation configuration [8],[13],[[17],Chap.6], where the error correction code follows the constrained code. We show performance results of FRB codes under both standard and reverse concatenation configurations (Chapter 8.2.1). This serves to illustrate the potential gains of using very high-rate FRB codes in conjunction with reverse concatenation.

However, it is possible to overcome the error-propagation drawback of FRB codes even without using reverse concatenation. The average error-propagation can be significantly reduced by a more careful pre-processing, and eliminating the bit insertions altogether. We call the resultant codes iterative pre-processed (IPP) codes (Chapter 8.2.2). For a given value of k, the IPP codes have lower encoding rates than the FRB codes, but they have reduced error-propagation.

Throughout this research, the reader will note that the presented rate improvements are only slightly (1-2%) higher than those of existing algorithms. One may wonder if such seemingly insignificant improvements are interesting at all. Of course, with variable-rate codes, there is the purely academic interest of achieving the capacity limit. However, even with fixed-rate codes, such small rate improvements can have profound effects on the cost and performance of a manufactured hard disk drive. This is because a 1-2% increase in the coding rate allows the designer to decrease the system bandwidth (or equivalently increase the size of the recorded patterns) by an equivalent amount, which is usually enough to have a large effect on manufacturing

tolerances and system margins. As a concrete example of this, current industry-standard $(0, k)$ codes are rate 8/9 with $k = 3$, rate 16/17 with $k = 6$ and rate 64/65 with $k = 7$; and there is considerable effort being expended to design a rate 200/201 code, which is only a 1% increase in rate and density, but can substantially increase the robustness of the drive. In Chapter 8.1, we present a tabulation of the FRB code parameters required for the design of rate 100/101 and rate 200/201 $(0, k)$ codes.

Although the focus in this thesis is on (d, k) constraints, the presented encoding ideas, both variable-rate and fixed-rate, may be applied to several other RLL constraints. Specifically, we discuss extensions of the variable-rate code constructions to $(0, G/I)$ constraints, asymmetrical run-length constraints and multiple-spacing (d, k) constraints (Chapter 6); and an extension of the fixed-rate encoding to $(0, G/I)$ constraints (Chapter 7.6).

The remainder of this thesis is organized as follows. A brief overview of the digital recording system is presented in Chapter 2. The bit stuffing technique is reviewed in Chapter 3, and an alternate interpretation is provided for its optimality. This helps motivate our variable-rate symbol sliding and interleaving code constructions presented in Chapters 4 and 5, respectively. The variable-rate encoding ideas are extended to a few other RLL constraints in Chapter 6. We then move on to discuss fixed-rate constrained codes. In Chapter 7, we propose the FRB algorithm for $(0, k)$ constraints, and compute upper and lower bounds on the encoding rate. We also extend the fixed-rate encoding to $(0, G/I)$ constraints. Then in Chapter 8, we address the system issues related to FRB codes. Finally, concluding remarks and future research areas are highlighted in Chapter 9.

CHAPTER II

DIGITAL RECORDING SYSTEM

The main application for RLL codes is in digital recording. In this section, we present an overview of a digital recording system, and explain how binary data is stored on the recording medium. The system model described here is by no means an exact representation of an actual recording system, which has numerous components. Our aim is to keep the discussion simple, and extract only the necessary components in order to understand the role of RLL codes in a digital recording system.

2.1 System Block Diagram

Figure 1 shows a simplified block diagram of a typical digital recording system. The source bits (or input data bits) are assumed to be unconstrained, *i.e.*, independent, identically distributed (*i.i.d*), and unbiased ($\mathtt{Pr}\{0\} = \mathtt{Pr}\{1\} = 1/2$). The cascade of the error correction code (ECC) encoder and constrained encoder converts the source bits into what are called the channel bits (or stored bits). The channel bits are then written onto the recording surface by a physical write process. The read-out process comprises of a detector that reads the stored bits, followed by the constrained decoder and ECC decoder.

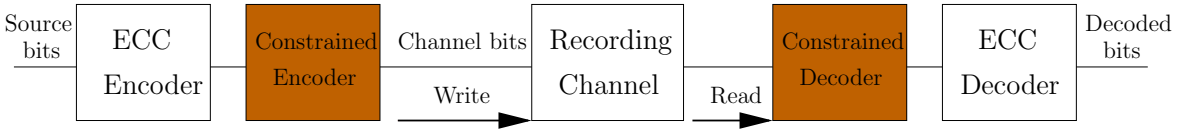

Figure 1: Block diagram of a digital recording system. Our focus is on the design of the shaded blocks, namely the constrained encoder and decoder.

The focus of this research is on the design of simple and efficient constrained

encoders and decoders for the system shown in Fig. 1. We wish to point out that while designing the constrained encoder, we do not code for possible channel bit errors. This is the domain of the ECC encoder and decoder. Hence, for theoretical purposes of optimal and near-optimal constrained code design, we assume that the recording channel is noiseless. Indeed, any practical system is prone to channel bit errors, and we provide a detailed discussion of the error propagation characteristics of our proposed codes in Chapter 8.2.

The first step in constrained encoder/decoder design is to identify a suitable RLL constraint for the recording system. This depends on several factors, including the recording channel model and read-out process. One RLL constraint that has found extensive use is the (d, k) constraint, which is the main subject of this research. There is a long history of the use of (d, k) codes and they are part of virtually all magnetic and optical disk recording systems today. The CD and DVD use $(d = 2, k = 10)$ encoding to increase the storage density by about 50% above that possible with unconstrained coding. Next generation optical recording systems like Blu-ray use the $(1, 7)$ constraint. The $(1, 3), (1, 7)$, and more recently, $(0, k)$ with $5 \leq k \leq 15$, including additional constraints on subsequences and transition-runs find numerous applications in magnetic and optical data storage [17]. These are all examples of recording systems, where the design of simple constrained encoders that generate near-capacity (d, k) sequences is of great interest. Hence, throughout this thesis, our approach will be to first provide detailed discussions on (d, k) constraints, and then present extensions to other RLL constraints. Before proceeding to discuss (d, k) codes however, we briefly describe the write process in a digital recording system, and explain the reason for imposing the (d, k) constraint.

2.2 Writing on a Recording Surface: Relation to (d, k) Sequences

Information is generally stored on a binary recording medium by physically writing into one of two allowable states. For magnetic recording media (e.g. HDDs), the two states are visualized as positive and negative magnetizations, while for their optical counterparts (e.g. CDs,DVDs), they are referred to as marks and nonmarks. In our discussion, we use the terms "marks" and "nonmarks" in a rather generic sense, without alluding to any specific recording medium.

The physics of the write process, along with considerations on inter symbol interference (ISI), prevent the written marks and nonmarks from being arbitrarily small. In other words, there is a certain minimum mark/nonmark size that can be made on the recording surface. On the other hand, written marks and nonmarks cannot be arbitrarily long. This is because the timing information for the read-out clock is derived from the stored data itself, *i.e.*, the stored data must be self-clocking. The timing information resides in state transitions: from marks to nonmarks and vice versa. This in turn means that only those data patterns which have frequent state transitions, can be stored on the recording medium. Hence, we have an upper and lower limit on the physical mark/nonmark sizes in a digital recording system.

To understand this better, let us take a look at how the stored bits are mapped to the physical marks/nonmarks. For convenience, let us assume that the electronic clock scans through the recording surface at a constant speed of one stored bit per unit time. With this reference, let the minimum and maximum mark sizes be d' and k' units, respectively. It must be appreciated that d' can be greater than one, *i.e.*, the electronic clock can run faster than the physical granularity.

Information is stored as a sequence of alternating marks and nonmarks, each of size at least d', and at most k'. Figure 2 shows an example of written data on a recording medium. A naive mapping of the stored bits to physical marks would

be to assign a "1" bit to the minimum mark/nonmark size, and a "0" bit to the next lowest mark/nonmark size. Usually $k' > d' + 1$, and such a scheme is clearly wasteful as it does not utilize the entire range of available mark lengths from d' through to k'. Instead, consider the following assignment. Represent the sequence of alternating marks and nonmarks by a sequence of alternating strings of "1"s and "-1"s of appropriate lengths. For example, the written sequence in Fig. 2: a mark of length 3, followed by a nonmark of length 2, a mark of length 4, a nonmark of length 3 and a mark of length 2, is represented as $1\ 1\ 1\ -1\ -1\ 1\ 1\ 1\ 1\ -1\ -1\ -1\ 1\ 1$. The stored bits are now obtained by mapping 1^i to $0^{i-1}1$, and $(-1)^i$ also to $0^{i-1}1$, $i = d', d'+1, \ldots, k'$. The example stored bit sequence is hence 00101000100101. It is left to the reader to now verify that the stored bit sequence is indeed (d, k)-constrained, where $d = d' - 1$ and $k = k' - 1$. The mapping described above is followed in practice and can be shown to be better than the naive bit assignment described earlier (see [17], pp. 58-60 for details).

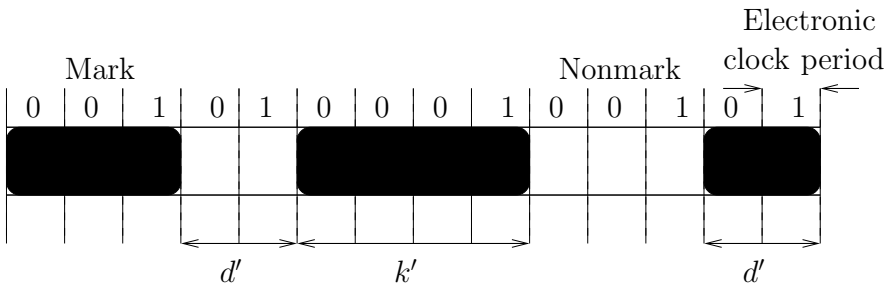

Figure 2: Alternating marks and nonmarks written on a recording surface. Marks/nonmarks can be of length greater than d', but less than k'. In this example, $d' = 2$ and $k' = 4$. Thus, the electronic clock period is half the minimum mark/nonmark size. The corresponding stored data bits are also shown.

From the preceding discussion, it appears that the (d, k)-constrained stored bit sequence is first translated into an intermediate sequence of runs of 1s and -1s, before being written onto the recording surface. Indeed, the intermediate sequence is run-length-limited (an RLL sequence), and is referred to as the *write sequence*. The

above-described differential mapping between the RLL $1/-1$ write sequence and the (d, k)-constrained stored bit sequence, is called the NRZI mapping.

It is now clear that the k constraint arises out of the self-clocking requirement, while the d constraint is used to regulate the space between successive state transitions, and hence the ISI. For several years, recording systems used a peak-detection method (see [[18],Section II-C] for a description) to read-out the stored data. In peak-detection-based systems, the d constraint had a direct impact on performance by regulating the ISI and minimizing detection errors. The conflicting attributes of storage density and timing window (see [[17], pp. 58-60] for a full discussion) meant that typical choices were $d = 1$ and $d = 2$.

2.3 Other RLL Codes in Recording Systems

With the emergence of partial-response equalization with maximum-likelihood sequence detection (PRML) techniques [57],[9],[18], there was a significant change in the read-out methodology. Rather than regulate ISI using a d constraint as in peak detection, PRML detection actually embraced the ISI, and used signal processing techniques to yield dramatic improvements in system performance. With the ISI no longer being a limiting factor, PRML-based magnetic recording systems moved towards higher-rate $d = 0$ codes. With maximum-likelihood sequence detection, the system error-rate mainly depends on the minimum-distance at the detector input. Hence, additional constraints on the stored bit sequence were imposed to increase the minimum-distance, *e.g.* $(0, G/I)$ constraint, $d = 1$ constraint, maximum-transition-run (MTR) constraints. Some other constraints were imposed to match the code spectrum to the channel-response spectrum, according to the theory of matched spectral-nulls codes [24] *e.g.* DC/Nyquist nulls constraints [12]. The advent of perpendicular magnetic recording [28] has brought with it several new problems and possibly different input constraints. Optical recording systems continue to use $d = 1$ codes with

significantly different channel models compared to magnetic recording, and increasing storage density has given rise to asymmetrical run-length constraints in these recording media.

All these examples serve to illustrate that the type of input constraint is highly system-dependent. There can be no single RLL constraint that works best for all systems. However, the need for simple, high-rate constrained codes is common to all these examples. As seen earlier in Chapter 1, even a 1-2% increase in code rate with simpler encoding/decoding can impact the cost and performance of a digital recording system. Hence, despite the long history of constrained codes, there continues to be an effort to improve the encoding rates. Our aim in this research is to develop simple methodologies for the design of optimal and near-optimal constrained codes, with a focus on (d, k) constraints, and subsequent extensions to other relevant constraints. In the next chapter, we review a variable-rate encoding technique called bit stuffing, which forms the basis for the encoding algorithms proposed in the rest of this thesis.

CHAPTER III

REVIEW: VARIABLE-RATE ENCODING AND THE BIT STUFF ALGORITHM

Variable-rate encoding refers to the fact that for a given finite input block length, the output length of the constrained sequence can vary depending on the actual input bits. In other words, different inputs of the same length can give rise to outputs of fluctuating lengths.

In previous work [27],[31], optimal (d, k) codes have been studied from a source coding perspective. This can be understood as follows. For given non-negative integers d and k, $0 \leq d < k$, let $\mathcal{X}_{d,k} = \left\{0^k1, 0^{k-1}1, \ldots, 0^{d+1}1, 0^d1\right\}$ if $k < \infty$, and let $\mathcal{X}_{d,\infty} = \left\{0, 0^d1\right\}$, where 0^t1 represents t consecutive "0"s followed by a "1". The elements of the finite set $\mathcal{X}_{d,k}$ are referred to as (d, k) *phrases*, and any (d, k)-constrained sequence can be described as the concatenation of phrases from $\mathcal{X}_{d,k}$. Furthermore, it is known that a (d, k) code achieves capacity if and only if it generates (d, k) phrases *maxentropically*, which means that the phrase of length l occurs with probability $\lambda_{d,k}^{-l}$, independently of others [61]. Thus, maxentropic (d, k)-constrained sequences can be viewed as the output of a memoryless source which emits phrases from the finite alphabet $\mathcal{X}_{d,k}$, with the phrase 0^t1 occurring with probability $\lambda_{d,k}^{-(t+1)}$.

An ideal, lossless source code removes the redundancy from such a source to form unconstrained and Bernoulli(1/2)-distributed ($\mathtt{Pr}\{0\} = \mathtt{Pr}\{1\} = 1/2$) output. Thus, the ideal, lossless source code acts as a *distribution transformer* (DT), which transforms maxentropically distributed (d, k)-constrained sequences into independent and identically distributed (*i.i.d*) Bernoulli(1/2) sequences. We say that the encoder of

the ideal source code is a $(\mathbf{\Lambda}_{d,k}, \mathtt{b}(1/2))$-DT, where $\mathbf{\Lambda}_{d,k}$ denotes the maxentropic (d,k) phrase distribution, and $\mathtt{b}(1/2)$ denotes the Bernoulli(1/2) distribution. The decoder of the ideal source code is thus a $(\mathtt{b}(1/2), \mathbf{\Lambda}_{d,k})$-DT, and can be used to generate optimal (d,k) codes. However, most source codes in practice are not ideal, and only generate nearly-$\mathtt{b}(1/2)$ output sequences. Nevertheless, the decoder of a suitable source code can be used to encode unconstrained $\mathtt{b}(1/2)$ input into near-maxentropic (d,k)-constrained output. Several authors have examined (d,k) codes from this perspective. Specifically, Kerpez [27] investigated four types of such source encoder-decoder pairs based on the Huffman code [11], enumerative code [55],[10],[29],[19], variable-length-to-block code [21], and a combined source-(d,k) code based on the arithmetic code [31],[58]. Among these only the enumerative code is fixed-rate, while all others use variable-rate encoding. In each case, the rate of the corresponding source decoder was shown to converge to the (d,k) capacity with increasing block length.

In subsequent work by Lee, then Bender and Wolf, the bit stuff algorithm [6] was proposed to construct optimal and near-optimal (d,k) codes using only a $(\mathtt{b}(1/2), \mathtt{b}(p))$-DT, where $\mathtt{b}(p)$ denotes the Bernoulli(p) distribution with $\mathtt{Pr}\{0\} = p$, $p \in [0,1]$. Unlike the $(\mathtt{b}(1/2), \mathbf{\Lambda}_{d,k})$-DT, where $\mathbf{\Lambda}_{d,k}$ is in general non-binary, the $(\mathtt{b}(1/2), \mathtt{b}(p))$-DT has the property that the output distribution is binary. Hence, both the input and the output of a $(\mathtt{b}(1/2), \mathtt{b}(p))$-DT are composed of independent and identically distributed $(i.i.d)$ bits, but the output now has a bias p. Bender and Wolf showed that controlled insertion of additional bits into such an $i.i.d$ biased, bit sequence, could lead to optimal codes for the (d,∞) and $(d,d+1)$ constraints and near-optimal codes for other constraints. This is known as the bit stuff algorithm [6]. The general concept of inserting additional bits so as to satisfy constraints is referred to as bit stuffing [7].

More recently, the bit flipping algorithm [3] was shown to improve bit stuff encoding rates for most (d,k) constraints and additionally achieve $(2,4)$ capacity. For all

values of (d, k), $k \neq d+1$, $k \neq \infty$ and $(d, k) \neq (2, 4)$, both bit stuffing and bit flipping are suboptimal. Our algorithms in Chapters 4 and 5 improve upon the bit stuff and bit flipping algorithms, while still using only $(\mathtt{b}(1/2), \mathtt{b}(p))$-DTs.

To gain the necessary understanding and motivation behind our proposed constructions, we first review the bit stuff algorithm in detail, and discuss some relevant properties. Along the way, we provide an alternate interpretation of the optimality of bit stuffing for certain (d, k) constraints. This motivates the need for our proposed algorithm in Chapter 4. Throughout our discussions in the rest of this thesis, we use the short-hand notation DT(p) to imply the $(\mathtt{b}(1/2), \mathtt{b}(p))$-DT.

3.1 The Bit Stuff Algorithm

The bit stuff algorithm was proposed by Lee, and generalized by Bender and Wolf [6] to produce optimal and near-optimal (d, k) sequences. The block diagram of the bit stuff encoder is shown in Fig. 3. The encoding proceeds in two stages. First, the distribution transformer DT(p) converts the *i.i.d*, unbiased ($\mathtt{Pr}\{0\} = \mathtt{Pr}\{1\} = 1/2$) input bit sequence into an *i.i.d*, p-biased ($\mathtt{Pr}\{0\} = p$) bit sequence. In the second stage, the p-biased bit sequence undergoes bit stuffing according to the following two operations

1. Scan the incoming bit sequence and insert a "1" after every run of $k - d$ consecutive "0"s (skip this step if $k = \infty$)

2. Scan the output bit sequence of the first operation, and stuff d "0"s after every "1".

The first operation produces a $(0, k-d)$-constrained bit sequence, which then acts as input for the second operation. Stuffing d zeros in the second operation translates the $(0, k-d)$ constraint into the required (d, k) constraint. Both these operations are invertible. Hence, with a one-to-one implementation of the distribution transformer

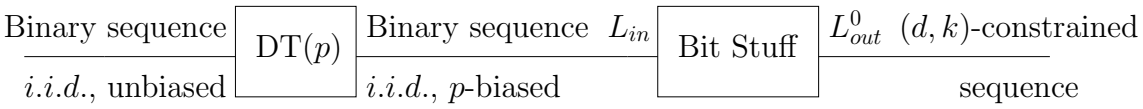

Figure 3: Block diagram of the bit stuff encoder. DT(p) **denotes the** $(\mathtt{b}(1/2), \mathtt{b}(p))$**-distribution transformer.** L_{in} **denotes the average length at the bit stuff input, and** L^0_{out} **denotes the average output length.**

DT(p), the bit stuff decoder is a simple inverse of the encoder.

3.2 Distribution Transformer Implementation

The distribution transformer DT(p) is nothing but the inverse of a lossless source code that transforms *i.i.d*, p-biased ($\mathtt{Pr}\{0\} = p$) bit sequences into *i.i.d*, unbiased ($\mathtt{Pr}\{0\} = \mathtt{Pr}\{1\} = 1/2$) bit sequences. In other words, DT(p) is the decoder of the lossless source code. The source code can be constructed using one of several methods available in the literature. Two examples are the Huffman code [11] and the arithmetic code. The arithmetic codes was derived by Elias, and made useful by Rissanen, Langdon, Jones and Pasco [23],[45]. They are a tree or nonblock-source code, which makes them suitable for coding long sequences. Pasco and Jones [23] outlined separately, the implementation of arithmetic codes using floating-point arithmetic. Indeed, arithmetic coding can be used to directly build a $(\mathtt{b}(1/2), \mathbf{\Lambda}_{d,k})$-DT, as shown in [2],[40],[41]. Even fixed-rate arithmetic codes have been designed for (d, k) constraints in [31],[58]. However, in this thesis, we limit our attention to the design of optimal and near-optimal constrained codes using the simple concept of bit stuffing.

3.3 Computing Rate

Since the bit stuff algorithm is variable-rate, we define the average rate as

$$R_0(p, d, k) = \frac{\text{Average input length}}{\text{Average output length}}.$$

Intuitively, the average rate is the ratio of the number of "bits in" to "bits out" for a very long input sequence.

Recall that the bit stuff encoder has two components: the distribution transformer DT(p), and the bit stuff block, as shown in Fig. 3. The distribution transformation occurs at an average rate of $h(p)$, where $h(p) = -plog_2p - (1-p)log_2(1-p)$ is the binary entropy function. Let us denote the average length at the bit stuff input by L_{in}, and the average output length by L^0_{out}, as indicated in Fig. 3. Then $R_0(p,d,k)$ can be written as the product of the average rates of the two components,

$$R_0(p,d,k) = h(p)\frac{L_{in}}{L^0_{out}}. \tag{3}$$

For given constraint parameters d and k, the maximum average rate of the bit stuff algorithm is now defined as $R^*_0(d,k) = \max_{p\in[0,1]} R_0(p,d,k)$. We note from (3) that for any $p \neq 1/2$, $h(p)$ is strictly less than unity, and hence there is a rate loss in the first stage of bit stuff encoding. However, with a suitable choice of p, the biasing can actually improve the overall rate, $R_0(p,d,k)$, by better fitting input data to the constraint. Essentially, an appropriately biased bit sequence incurs lesser stuffed bits in the second stage. Bender and Wolf [6] showed that the maximum average rate, $R^*_0(d,k)$, equals the (d,k) capacity for $k = d+1$ and $k = \infty$, but is strictly less than capacity for all other cases. In the following discussion, we provide an alternate interpretation of their results. This is based on matching phrase probabilities, and will help motivate the need for our proposed algorithm in Chapter 4.2.

Consider the finite state transition diagram (FSTD) of a (d,k) constraint, which is shown in Fig. 4 for $k < \infty$. Walks on the FSTD can be used to generate all possible (d,k) sequences by reading off the edge-labels. It is well known that there is a *maxentropic* walk, where edges must be traversed according to a set of optimal state transitions in order to achieve the highest possible rate. A code achieves capacity if

and only if it produces a walk on the FSTD with the maxentropic state transition probabilities shown in parentheses in Fig. 4.

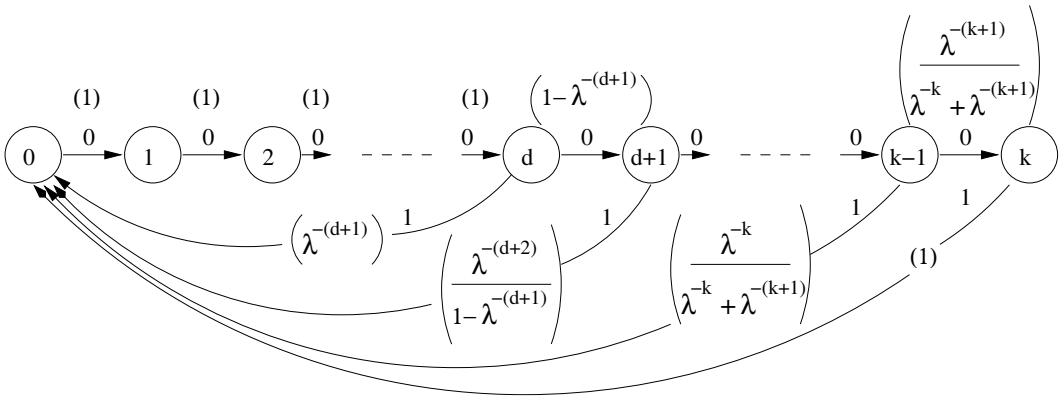

Figure 4: FSTD of the (d, k) constraint, $k < \infty$. Maxentropic state transition probabilities are shown in parentheses. The labels on directed edges indicate the output bit.

Alternatively, one can describe a (d, k) sequence by the concatenation of phrases from the finite set $\mathcal{X}_{d,k} = \left\{0^k1, 0^{k-1}1, \ldots, 0^{d+1}1, 0^d1\right\}$. Each (d, k) phrase, 0^t1, $t = k, k-1, \ldots, d$, corresponds to a cycle on the FSTD (see Fig. 4) that begins and ends in state 0. Consequently, a code achieves capacity if and only if it generates (d, k) phrases with the corresponding maxentropic probabilities. Note that if $k = \infty$, then $\mathcal{X}_{d,\infty} = \left\{0, 0^d1\right\}$ and the FSTD in Fig. 4 can be redrawn with exactly $d+1$ states, and two cycles that begin and end in state 0. Once again, each (d, ∞) phrase corresponds uniquely, to a cycle in the FSTD and any (d, ∞) code is capacity-achieving if and only if it generates the two phrases maxentropically.

It is known from a result of Zehavi and Wolf [61] that the maxentropic probability of the (d, k) phrase of length t is equal to $\lambda_{d,k}^{-t}$. We can hence form a maxentropic phrase probability vector for all (d, k), $k < \infty$, as

$$\mathbf{\Lambda}_{d,k} = \left[\lambda_{d,k}^{-(k+1)} \ \lambda_{d,k}^{-(k)} \ldots \lambda_{d,k}^{-(d+2)} \ \lambda_{d,k}^{-(d+1)}\right]. \tag{4}$$

Next, we write down the corresponding vector of phrase probabilities generated by the bit stuff algorithm

$$\mathbf{v}^0 = \left[v_0^0 \; v_1^0 \ldots v_{k-d-1}^0 \; v_{k-d}^0\right], \tag{5}$$

where v_i^0 denotes the probability of occurrence of the phrase of length $k-i+1$, namely $0^{k-i}1$. Table 1 specifies the mapping between the bit stuff input words and the output (d,k) phrases, as induced by the bit stuff block shown in Fig. 3. Recall that the bit stuff input is p-biased, thereby yielding the corresponding phrase probabilities v_i^0 as a function of p. For the case when $k=\infty$, $\mathbf{\Lambda}_{d,\infty} = \left[\lambda_{d,\infty}^{-1} \; \lambda_{d,\infty}^{-(d+1)}\right]$, and $\mathbf{v}^0 = [p \; 1-p]$.

Table 1: Bit stuff phrase probabilities, $k < \infty$

Index (i)	**Bit stuff input word**	(d,k) **phrase**	**Bit stuff phrase probability** (v_i^0)
0	0^{k-d}	0^k1	p^{k-d}
1	$0^{k-d-1}1$	$0^{k-1}1$	$p^{k-d-1}(1-p)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
j	$0^{k-d-j}1$	$0^{k-j}1$	$p^{k-d-j}(1-p)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$k-d-1$	01	$0^{d+1}1$	$p(1-p)$
$k-d$	1	0^d1	$1-p$

3.4 Interpretation of Optimality

With this initial discussion and setup, we now interpret the Bender-Wolf results [6] as follows. The bit stuff algorithm achieves capacity if and only if $\mathbf{v}^0 = \mathbf{\Lambda}_{d,k}$. It can be verified that for $(d,d+1)$ and (d,∞) constraints, $\mathbf{v}^0$ exactly matches $\mathbf{\Lambda}_{d,k}$ with $p = \lambda_{d,d+1}^{-(d+2)}$ and $p = \lambda_{d,\infty}^{-1}$, respectively. Hence, the bit stuff algorithm is optimal when $k = d+1$ and $k = \infty$. However, for all other (d,k) constraints, bit stuffing is strictly

suboptimal, no matter what value of bias p is chosen. The following proposition restates a result from [6].

Proposition 3.1 *For $d \geq 0$, $d+2 \leq k < \infty$, $\mathbf{v}^0 \neq \mathbf{\Lambda}_{d,k}$ for any $p \in [0,1]$.*

Proposition 3.2 *For $0 \leq d < k < \infty$, and a DT bias p, the average information rate, $R_0(p,d,k)$, of the bit stuff algorithm is given by*

$$R_0(p,d,k) = \begin{cases} h(p)\dfrac{1-p^{k-d}}{1-p^{k-d+1}+d(1-p)} & k < \infty \\ \dfrac{1}{1+d(1-p)} & k = \infty \end{cases} \tag{6}$$

For the special case of $p = 1/2$, i.e.*, without the DT in Fig. 3, (6) reduces to*

$$R_0(d,k) = \begin{cases} \dfrac{2^{k-d+1}-2}{2^{k-d+1}-1+d2^{k-d}} & k < \infty \\ \dfrac{2}{d+2} & k = \infty \end{cases} \tag{7}$$

Proposition 3.1 implies that the maximum average bit stuffing rate, $R_0^*(d,k)$, is strictly less than capacity for all (d,k) constraints with $d+2 \leq k < \infty$. Our objective now is to improve bit stuff encoding rates for $d+2 \leq k < \infty$, while maintaining a similar implementation. To meet this objective, we restrict ourselves to the use of a single distribution transformer as in bit stuffing, and show that a simple switching of bit stuff phrase probabilities improves the encoding rates for several values of d and k. As a first step, we show how this idea leads to the recently proposed bit flipping algorithm [3], and then generalize to symbol sliding in Chapter 4.

3.5 The Bit Flipping Algorithm

For any given (d,k) constraint, $k < \infty$, consider a DT bias of p greater than $1/2$ in the bit stuff algorithm. This means that a "0" is more likely than a "1" at the bit stuff

input (see Fig. 3). Our goal now, is to match the bit stuff phrase probability vector, $\mathbf{v}^0$, to the maxentropic vector $\mathbf{\Lambda}_{d,k}$. Looking at indices $i = 0$ and $i = 1$ in Table 1, we note that $v_0^0 = p^{k-d} > v_1^0 = p^{k-d-1}(1-p)$, but the corresponding maxentropic probabilities are related as $\lambda_{d,k}^{-(k+1)} < \lambda_{d,k}^{-(k)}$. This suggests that swapping the bit stuff phrase probabilities v_0^0 and v_1^0, should result in a better match with $\mathbf{\Lambda}_{d,k}$, thereby improving bit stuff encoding rates. Hence, we would like to replace the the bit stuff block in Fig. 3, by a constrained encoder that performs the following three operations on the biased bit sequence

1. (i) If $k = d + 1$, flip every incoming bit

   (ii) If $k \notin \{d+1, \infty\}$, track the run-length, ρ, of consecutive "0"s in the incoming bit sequence. When $\rho = k - d - 1$, flip the next incoming bit, then reset ρ and goto (ii)

2. Scan the output bit sequence of the first operation, and insert a "1" after every run of $k - d$ consecutive "0"s (skip this step if $k = \infty$)

3. Scan the output bit sequence of the second operation, and stuff d "0"s after every "1".

The first operation performs the bit flipping (change a "1" bit to a "0" bit and *vice versa*), which is equivalent to swapping the phrase probabilities v_0^0 and v_1^0, and their corresponding bit stuff input words in Table 1. The second and third operations are identical to the bit stuff operations described in Chapter 3.1. Since the bit flipping operation is invertible, the decoder once again is simply the encoder's inverse components arranged in the reverse order.

The algorithm described above is precisely the bit flipping algorithm proposed by Aviran *et al.* [3]. Their main results are summarized in the following two propositions

Proposition 3.3 *For $d \geq 1$, $d+2 \leq k < \infty$, the bit flipping algorithm achieves greater maximum average rate than the bit stuff algorithm.*

Proposition 3.4 *For $d \geq 0$, $d+2 \leq k < \infty$, the bit flipping algorithm is optimal if and only if $d = 2$ and $k = 4$.*

Proposition 3.3 mainly depends on the following two facts

(i) Given $k < \infty$, the average bit flipping rate is greater than the average bit stuff encoding rate for all values of bias p such that $1/2 < p < 1$.

(ii) The rate maximizing bit stuffing bias is greater than $1/2$ when $d = 1$, $4 \leq k < \infty$, and for all $d \geq 2$, $d+2 \leq k < \infty$.

Proposition 3.4 states that for $0 \leq d \leq k-2 < \infty$, the new phrase vector, say $\mathbf{v}^1$, formed by swapping the phrase probabilities v_0^0 and v_1^0 in $\mathbf{v}^0$, exactly matches $\mathbf{\Lambda}_{d,k}$ only for the $(2,4)$ constraint. As will be seen later in Chapter 4, this optimality of the bit flipping algorithm is possible because of the capacity equality $C(2,4) = C(1,2)$. For the special case when $k = d+1$, bit flipping with a bias p is equivalent to bit stuffing with bias $1-p$; and when $k = \infty$, bit flipping becomes identical to bit stuffing. Hence, bit flipping continues to be optimal for (d, ∞) and $(d, d+1)$ constraints.

CHAPTER IV

THE SYMBOL SLIDING ALGORITHM

In this chapter, we present the symbol sliding algorithm, which further improves bit flipping rates while still using only a single DT. We prove the optimality of the proposed algorithm for all (d, k) constraints with $k = 2d+1$, and show that bit stuffing and bit flipping can be derived as special cases of symbol sliding. The following discussion on the $(1, 3)$ constraint brings out the principal ideas.

4.1 Motivating Example: The $(1, 3)$ Constraint

Thus far, we have seen a phrase probability interpretation of bit stuffing, and how switching two entries of the phrase probability vector $\mathbf{v}^0$, improved the encoding rates. This prompts us to generalize the idea of switching phrase probabilities to better match the maxentropic vector $\mathbf{\Lambda}_{d,k}$. The following example of the $(1, 3)$ constraint motivates this idea.

Table 2: Phrase probabilities for the $(1, 3)$ constraint

Index (i)	**Bit stuff Input word**	**$(1,3)$ phrase**	**Maxentropic prob.** $(\mathbf{\Lambda}_{1,3}(i))$	**Bit stuff prob.** (v_i^0)	**Bit flipping prob.** (v_i^1)	**Symbol sliding prob. with index 2** (v_i^2)
0	0^2	$0^3 1$	$\lambda_{1,3}^{-4}$	p^2	$p(1-p)$	$p(1-p)$
1	01	$0^2 1$	$\lambda_{1,3}^{-3}$	$p(1-p)$	p^2	$1-p$
2	1	01	$\lambda_{1,3}^{-2}$	$1-p$	$1-p$	p^2

Consider the phrase probabilities listed in Table 2. From Proposition 3.1, it follows that the maximum average bit stuff encoding rate is strictly less than $(1, 3)$ capacity. Proposition 3.3 states that $(1, 3)$ bit flipping rates are also suboptimal. Now consider

the phrase probabilities v_i^2 as listed in the last column of Table 2. We call this *symbol sliding* with index 2. This means that v_0^0 (corresponding bit stuff input word 0^2) is slid down to the index 2 position of v_2^0 (input word 1), with v_2^0 (input word 1) and v_1^0 (input word 01) being pushed up an index each, thus yielding the phrase probability vector $\mathbf{v}^2 = [v_0^2 \ v_1^2 \ v_2^2]$. It can be shown that with a bias of $p = \lambda_{1,3}^{-1}$, $\mathbf{v}^2$ exactly matches $\mathbf{\Lambda}_{1,3}$, and the average rate is equal to the $(1,3)$ capacity. Hence, symbol sliding with index 2 achieves capacity for the $(1,3)$ constraint where both bit stuffing and bit flipping fall short. This prompts us to study symbol sliding in greater depth.

4.2 Encoding Procedure

The main idea behind symbol sliding is to switch the bit stuff phrase probabilities so as to better match the maxentropic vector $\mathbf{\Lambda}_{d,k}$. Symbol sliding is hence a function of a sliding index $j, 0 \leq j \leq k-d$, for any given (d,k) constraint with $k < \infty$. Symbol sliding with index j involves sliding down v_0^0 from index $i = 0$ to $i = j$ and moving each of $v_1^0, v_2^0, \ldots, v_j^0$ up an index each, to yield the phrase probability vector $\mathbf{v}^j = [v_0^j \ v_1^j \ \ldots v_{k-d}^j]$. Table 3 provides the full list of bit stuffing, bit flipping, symbol sliding and maxentropic phrase probabilities. It can be seen that bit stuffing and bit flipping are special cases of symbol sliding with indices $j = 0$ and $j = 1$, respectively.

The symbol sliding encoder is shown in Fig. 5. It has a similar set-up to the bit stuff encoder shown in Fig. 3. The only difference is that the bit stuff block is replaced by a constrained encoder that performs the following two operations on the biased bit sequence

1. (i) If $j = 0$ and $k \neq \infty$, scan the incoming bit sequence and insert a "1" after every run of $k-d$ consecutive "0"s

   (ii) If $j = 1$ and $k = d+1$

   (a) Flip every incoming bit

   (b) Scan the output bit sequence of (a) and insert a "1" after every "0"

(iii) If $j = 1$ and $k \notin \{d+1, \infty\}$

(a) Track the run-length, ρ, of consecutive "0"s in the incoming bit sequence. When $\rho = k - d - 1$, flip the next incoming bit, then reset ρ and goto (a)

(b) Scan the output bit sequence of (a) and insert a "1" after every run of $k - d$ consecutive "0"s

(iv) If $0 \leq d \leq k - 2 < \infty$ and $2 \leq j \leq k - d - 1$

(a) Track the run-length, ρ, of consecutive "0"s in the incoming bit sequence. When $\rho = k - d - j$, insert a "0". When $\rho = k - d$, replace the run of $k - d + 1$ consecutive "0"s (including the inserted "0" when ρ was equal to $k - d - j$), with the phrase $0^{k-d-j}1$, then reset ρ and goto (a)

(v) If $0 \leq d \leq k - 2 < \infty$ and $j = k - d$

(a) Insert a "0" after every string of $k - d$ consecutive "0"s

(b) Scan the output bit sequence of (a) and insert a "0" after every "1"

(c) Track the run-length, ρ, of consecutive "0"s in the output bit sequence of (b). When $\rho = k - d + 1$, replace the run of $k - d + 1$ consecutive "0"s with a single "1" bit, then reset ρ and goto (c)

2. Scan the output bit sequence of the first operation and stuff d "0"s after every "1"

The first operation produces a $(0, k - d)$-constrained sequence with the appropriate phrase matching, and the second operation translates this to a (d, k) constraint by inserting d zeros after each "1" bit. The first operation has been broken down into five cases. The first three cases with $j = 0$ and $j = 1$ correspond to the bit stuffing and bit flipping operations discussed earlier. The generalized symbol sliding operations with $2 \leq j \leq k - d$ are specified in cases (iv) and (v).

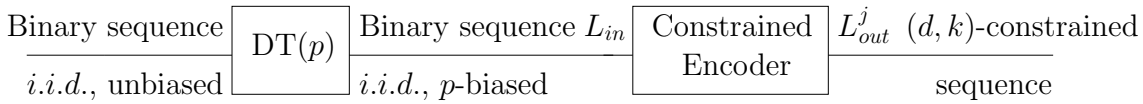

Figure 5: Block diagram of the symbol sliding encoder. L_{in} denotes the average length at the input to the constrained encoder. L^j_{out} denotes the average output length for sliding index j.

The constrained decoder is a simple inverse of the constrained encoder. It performs the following three operations on the (d, k) sequence

1. Delete the d stuffed "0"s after every "1"

2. (i) If $0 \leq d \leq k-2 < \infty$ and $j = k - d$

   (a) After every "1" in the output bit sequence of the first operation, if the next bit is "0", delete the "0" bit. If the next bit is "1", insert a string of $k - d$ consecutive "0"s before the "1" bit

   (b) Scan the output bit sequence of (b), delete the "1" after every run of $k - d$ consecutive "0"s

   (ii) If $0 \leq d \leq k-2 < \infty$ and $2 \leq j \leq k - d - 1$

   (a) Track the run-length, ρ, of consecutive "0"s in the output bit sequence of the first operation. When $\rho = k - d - j$ and the next bit is "0", delete the "0" bit. When $\rho = k - d - j$ and the next bit is "1", replace the phrase $0^{k-d-j}1$ with a string of $k - d$ consecutive "0"s

   (iii) If $j = 1$ and $k \notin \{d + 1, \infty\}$

   (a) Scan the output bit sequence of the first operation, and delete the inserted "1" after every run of $k - d$ consecutive "0"s

   (b) Track the run-length, ρ, of consecutive "0"s in the output bit sequence of (a). When $\rho = k - d - 1$, flip the next incoming bit, then reset ρ and goto (b)

(iv) If $j = 1$ and $k = d + 1$

(a) Scan the output bit sequence of the first operation, and delete the inserted "1" after every "0"

(b) Flip every bit in the output bit sequence of (a)

(v) If $j = 0$ and $k \neq \infty$, delete the "1" after every run of $k - d$ consecutive "0"s in the output bit sequence of the first operation

Having described the symbol sliding algorithm, we now proceed to analyze its performance. Let us denote by $\mathsf{SS}(j)$, the symbol sliding algorithm with index j. Then, the special cases $\mathsf{SS}(0)$ and $\mathsf{SS}(1)$ denote the bit stuffing and bit flipping algorithms, respectively. Hence, we expect that symbol sliding continues to be optimal for $(d, d+1)$, (d, ∞) and $(2, 4)$ constraints. In the following discussion, we derive some important properties and prove the optimality of symbol sliding for an additional class of (d, k) constraints.

4.3 Properties of Symbol Sliding

Lemma 4.1 *Let $0 \leq d < k < \infty$. Then, the maximum average rate achieved by $\mathsf{SS}(j)$ equals (d, k) capacity when $k = 2d + 1$ and sliding index $j = k - d = d + 1$.*

Proof: We use the fact that a code achieves capacity if and only if the generated phrases are maxentropic. We now show that $\mathsf{SS}(j)$ generates maxentropic (d, k) phrases when $k = 2d + 1$ and $j = d + 1$. Let us start with a result of Ashley and Siegel [1], which states that the capacity of the $(d, 2d+1)$ constraint is identical to that of the $(d+1, \infty)$ constraint. Hence $\lambda_{d,2d+1}$ is the positive, real root of each of the following two characteristic equations

$$\sum_{l=d+1}^{2d+2} z^{-l} = 1$$

$$z^{-1} + z^{-(d+2)} = 1. \tag{8}$$

Now, let the sliding index $j = k - d = d + 1$. Consider a bias $p = \lambda_{d,2d+1}^{-1}$. Then, we have

$$v_{k-d}^{d+1} = p^{k-d} = p^{d+1} = \lambda_{d,2d+1}^{-(d+1)} \tag{9}$$

$$v_{k-d-1}^{d+1} = 1 - p = 1 - \lambda_{d,2d+1}^{-1} = \lambda_{d,2d+1}^{-(d+2)} \tag{10}$$

$$v_{k-d-i}^{d+1} = p^{i-1}(1-p) = \lambda_{d,2d+1}^{-(d+i+1)}, \quad 2 \leq i \leq k-d, \tag{11}$$

where (10) follows from (8). Hence, we have $v_i^{d+1} = \lambda_{d,2d+1}^{-(k-i+1)}$, for all $0 \leq i \leq k-d$, whereby $\mathbf{v}^{d+1} = \mathbf{\Lambda}_{d,k}$. This proves the lemma. ■

Lemma 4.1 proves the optimality of symbol sliding for an additional class of (d, k) constraints, namely all $(d, 2d+1)$ constraints. Interestingly, the proof of optimality depends on the capacity equality $C(d, 2d+1) = C(d+1, \infty)$. Recall that the bit stuff algorithm is already capacity-achieving for all $(d+1, \infty)$ constraints. Hence, by a simple modification to bit stuffing, we have been able to extend the optimality property to another class of (d, k) constraints, which have the same capacity as $(d+1, \infty)$ constraints. The following theorem shows that $(d, 2d+1)$ constraints are the only additional class of (d, k) constraints for which symbol sliding is optimal.

Theorem 4.2 *For $0 \leq d < k$, the maximum average rate achieved by SS(j) equals the (d, k) capacity only in the following cases*

1. $j = 0$, $k = d + 1$
2. $j = 1$, $k = d + 1$

3. $j = 1,\ d = 2, k = 4$

4. $j = k - d,\ k = 2d + 1$

5. $k = \infty$.

For all other values of (d, k), the maximum average rate of $\mathsf{SS}(j)$ is strictly less than capacity for each $j, 0 \leq j \leq k - d$.

Proof: We wish to find constraints (d, k) for which $\mathbf{v}^j = \mathbf{\Lambda}_{d,k}$ for some j, $0 \leq j \leq k - d$. We first note that when there is no k constraint, *i.e.*, $k = \infty$, then the symbol sliding operations reduce to simply inserting d zeros after every one in the biased bit sequence. This is identical to the corresponding bit stuffing operation, which has been shown to achieve capacity for (d, ∞) constraints [6]. Case 5) in the theorem statement now follows. In the remainder of this proof, we focus only on (d, k) constraints with $k < \infty$.

Depending on the value of j, we have the following four cases.

Case 1: $j = 0$

This is identical to the bit stuff algorithm. Let us first consider $k > d + 1$. For any such given (d, k) constraint, the following must hold (see Table 3) in order for $\mathbf{v}^0$ to exactly match $\mathbf{\Lambda}_{d,k}$.

$$p = \lambda_{d,k}^{-1} \tag{12}$$

$$1 - p = \lambda_{d,k}^{-(d+1)} \tag{13}$$

$$p^{k-d} = \lambda_{d,k}^{-(k+1)}. \tag{14}$$

(12) and (13) together imply that $\lambda_{d,k}^{-1} + \lambda_{d,k}^{-(d+1)} = 1$. However, this means that $\lambda_{d,k}$ is a root of the characteristic (d, ∞) equation, $H_{d,\infty} = 1$. Hence, (12) and (13) cannot

be simultaneously satisfied for any finite $k > d+1$. This leads us to Proposition 3.1 which was stated without proof in Chapter 3.4.

Next, we look at $k = d+1$. In this case, we only have two possible phrases corresponding to indices $i = 0, 1$ in Table 3. It can be seen that a bias of $p = \lambda_{d,d+1}^{-(d+2)}$ is optimal. This yields Case 1) of the theorem statement.

Table 3: Maxentropic, bit stuff, bit flipping and symbol sliding phrase probabilities for the (d, k) constraint, $k < \infty$

Index (i)	**Bit stuff input word**	(d,k) **phrase**	**Maxentropic probability** $(\mathbf{\Lambda}_{d,k}(i))$	**Bit stuff probability** (v_i^0)	**Bit flipping probability** (v_i^1)	**Symbol sliding probability with index** j (v_i^j)
0	0^{k-d}	0^k1	$\lambda_{d,k}^{-(k+1)}$	p^{k-d}	$p^{k-d-1}(1-p)$	$p^{k-d-1}(1-p)$
1	$0^{k-d-1}1$	$0^{k-1}1$	$\lambda_{d,k}^{-(k)}$	$p^{k-d-1}(1-p)$	p^{k-d}	$p^{k-d-2}(1-p)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$j-1$	$0^{k-d-j+1}1$	$0^{k-j+1}1$	$\lambda_{d,k}^{-(k-j+2)}$	$p^{k-d-j+1}(1-p)$	$p^{k-d-j+1}(1-p)$	$p^{k-d-j}(1-p)$
j	$0^{k-d-j}1$	$0^{k-j}1$	$\lambda_{d,k}^{-(k-j+1)}$	$p^{k-d-j}(1-p)$	$p^{k-d-j}(1-p)$	p^{k-d}
$j+1$	$0^{k-d-j-1}1$	$0^{k-j-1}1$	$\lambda_{d,k}^{-(k-j)}$	$p^{k-d-j-1}(1-p)$	$p^{k-d-j-1}(1-p)$	$p^{k-d-j-1}(1-p)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$k-d-1$	01	$0^{d+1}1$	$\lambda_{d,k}^{-(d+2)}$	$p(1-p)$	$p(1-p)$	$p(1-p)$
$k-d$	1	0^d1	$\lambda_{d,k}^{-(d+1)}$	$1-p$	$1-p$	$1-p$

Case 2: $j = 1$

This is identical to the bit flipping algorithm. We first consider $k > d+2$. For any such given (d, k) constraint, the following must hold (see Table 3) in order for $\mathbf{v}^1$ to exactly match $\mathbf{\Lambda}_{d,k}$.

$$p = \lambda_{d,k}^{-1} \tag{15}$$

$$1 - p = \lambda_{d,k}^{-(d+1)} \tag{16}$$

$$p^{k-d} = \lambda_{d,k}^{-k} \tag{17}$$

$$p^{k-d-1}(1-p) = \lambda_{d,k}^{-(k+1)}. \tag{18}$$

(15) and (16) together imply that $\lambda_{d,k}^{-1} + \lambda_{d,k}^{-(d+1)} = 1$. As in the previous case, this is impossible unless $k = \infty$. Hence, $\mathsf{SS}(1)$ is strictly suboptimal for all (d, k) constraints with $d + 2 < k < \infty$.

Next, let $k = d + 2$. As before, from Table 3, we obtain the following conditions in order for $\mathbf{v}^1$ to exactly match $\mathbf{\Lambda}_{d,k}$.

$$1 - p = \lambda_{d,d+2}^{-(d+1)} \tag{19}$$

$$p^2 = \lambda_{d,d+2}^{-(d+2)} \tag{20}$$

$$p(1 - p) = \lambda_{d,d+2}^{-(d+3)}. \tag{21}$$

From (20) we have $p = \lambda_{d,d+2}^{-(\frac{d}{2}+1)}$. Using this and (19) in (21), we find that $\frac{d}{2}+1+d+1 = d + 3$ or $d = 2$. This implies that $\mathsf{SS}(1)$ is optimal for the $(2, 4)$ constraint, as stated in Case 3) of the theorem.

Finally, let $k = d + 1$. This means that we only have indices $i = 0, 1$ in Table 3. The bit flipping algorithm in this case is exactly the bit stuff algorithm run on the corresponding flipped (ones changed to zeros and *vice versa*) biased bit sequence. Hence, for any bit stuffing bias p, a bit flipping bias of $(1 - p)$ achieves the same average rate. This means that a bias of $1 - \lambda_{d,d+1}^{-(d+2)} = \lambda_{d,d+1}^{-(d+1)}$ is optimal for $(d, d + 1)$ bit flipping, as stated in Case 2) of the theorem.

Case 3: $2 \leq j \leq k - d - 1$

The above range of j implies that we are dealing only with constraints (d, k) for which $k \geq d + 3$. As in the previous two cases, we can derive the following set of conditions for $\mathbf{v}^j$ to exactly match $\mathbf{\Lambda}_{d,k}$ (see Table 3).

$$p = \lambda_{d,k}^{-1} \tag{22}$$

$$1 - p = \lambda_{d,k}^{-(d+1)} \tag{23}$$

$$p^{k-d} = \lambda_{d,k}^{-(k-j+1)}. \tag{24}$$

Once again, the above three conditions cannot be simultaneously satisfied unless $k = \infty$. Hence, we conclude that $\mathsf{SS}(j)$, $2 \leq j \leq k - d - 1$, cannot achieve capacity for any (d, k), $k < \infty$.

Case 4: $j = k - d$ and $j \geq 2$

It was shown in Lemma 4.1 that sliding index $j = k - d$ is optimal for $(d, 2d + 1)$ constraints. We now show that $(d, 2d + 1)$ are the only set of constraints for which $\mathsf{SS}(k - d)$ is capacity achieving. From Table 3, we note that the following conditions need to be satisfied for $\mathsf{SS}(k-d)$ to be optimal for any given (d, k). Recall that $j \geq 2$ and therefore $k - d \geq 2$.

$$p = \lambda_{d,k}^{-1} \tag{25}$$

$$1 - p = \lambda_{d,k}^{-(d+2)} \tag{26}$$

$$p^{k-d} = \lambda_{d,k}^{-(d+1)}. \tag{27}$$

From (27) and (25) above, we require that $k - d = d + 1$ or $k = 2d + 1$. It turns out (see Lemma 4.1) that this value of k satisfies condition (26) by virtue of the capacity equality $C(d, 2d + 1) = C(d + 1, \infty)$. Hence, we see that $\mathsf{SS}(k - d)$ is optimal for all $(d, 2d + 1)$ constraints, and no others.

From our discussions in Cases 1 through 4, we have determined all values of d and k for which $\mathsf{SS}(j)$ is optimal, and also established that for all constraints (d, k), $k \neq d+1$, $k \neq \infty$, $k \neq 2d+1$ and $(d, k) \neq (2, 4)$, the maximum average rate of $\mathsf{SS}(j)$, for each j, $0 \leq j \leq k - d$, is strictly less than capacity. This completes the proof. ■

In Theorem 4.2, we found that the symbol sliding algorithm is optimal for $(d, d+1)$, (d, ∞), $(d, 2d+1)$ and $(2, 4)$ constraints, and strictly suboptimal for all other (d, k) constraints for all values of bias $p \in [0, 1]$. For the suboptimal cases, we now have two degrees of freedom with j and p, that we can tune to improve the encoding rates. The following result establishes a necessary and sufficient condition on the bias p, so that $\mathsf{SS}(j)$ achieves a higher average rate than $\mathsf{SS}(j-1)$.

Theorem 4.3 *Let $0 \leq d < k < \infty$. Then for $0 < j \leq k - d$, the average rate of $\mathsf{SS}(j)$ is greater than the average rate of $\mathsf{SS}(j-1)$ if and only if $p > \lambda_{j-1,\infty}^{-1}$.*

Proof: Let us denote by $R_j(p, d, k)$ the average rate of $\mathsf{SS}(j)$ for a given constraint (d, k) and bias p. We then have

$$R_j(p, d, k) = h(p)\frac{L_{in}}{L_{out}^j}, \tag{28}$$

where L_{in} and L_{out}^j represent the average lengths at the input and output to the $\mathsf{SS}(j)$ constrained encoder, respectively (see Fig. 5). It can be seen that L_{in} does not depend on the sliding index and is identical for all j, $0 \leq j \leq k-d$. Hence, for a given bias p, L_{out}^j is the important factor in comparing the rates of $\mathsf{SS}(j)$ and $\mathsf{SS}(j-1)$. It is given by

$$L_{out}^j = \sum_{i=0}^{k-d} v_i^j l_i^j, \tag{29}$$

where l_i^j is the length of the (d, k) phrase corresponding to the phrase probability v_i^j listed in Table 3. For example, index $i = j - 1$ has $v_{j-1}^j = p^{k-d-j}(1-p)$ and $l_{j-1}^j = k - j + 2$. Now, consider the difference of average output lengths $L_{out}^{j-1} - L_{out}^j$. This is computed from (29) to be

$$L_{out}^{j-1} - L_{out}^{j} = p^{k-d} - p^{k-d-j}(1-p). \tag{30}$$

From (28) and (30), we can derive the condition, $R_j(p,d,k) > R_{j-1}(p,d,k)$ if and only if $p^j + p > 1$. The proof is now completed using the fact that the only positive, real root of $p^j + p = 1$ is $\lambda_{j-1,\infty}^{-1}$. ■

Theorem 4.3 specifies the range of bias under which it is appropriate to use symbol sliding with a higher sliding index. Let us define the maximum average rate of $\mathsf{SS}(j)$ as $R_j^*(d,k) = \max_{p\in[0,1]} R_j(p,d,k)$. Further, let us denote by p_j^*, the value of bias that maximizes $R_j(p,d,k)$. The following result is an immediate consequence of Theorem 4.3.

Corollary 4.4 *For any given (d,k) constraint, $k < \infty$, and sliding index j, $0 < j \leq k-d$, if $p_{j-1}^* > \lambda_{j-1,\infty}^{-1}$, then $R_j^*(d,k) > R_{j-1}^*(d,k)$. Conversely, if $p_j^* < \lambda_{j-1,\infty}^{-1}$, then $R_j^*(d,k) < R_{j-1}^*(d,k)$.*

Corollary 4.4 makes an important connection between the bias and sliding index. If the rate maximizing bias for $\mathsf{SS}(j-1)$, namely p_{j-1}^*, is greater than $\lambda_{j-1,\infty}^{-1}$, it follows from Theorem 4.3 that the average rate of $\mathsf{SS}(j)$ with a bias p_{j-1}^* is greater than $R_{j-1}^*(d,k)$, and hence $R_j^*(d,k) = \max_{p\in[0,1]} R_j(p,d,k) > R_{j-1}^*(d,k)$. For similar reasons, we have the converse that $R_j^*(d,k) < R_{j-1}^*(d,k)$ if $p_j^* < \lambda_{j-1,\infty}^{-1}$. On the other hand, if $p_{j-1}^* < \lambda_{j-1,\infty}^{-1}$, the authors have been unable to obtain any general relationship between $R_j^*(d,k)$ and $R_{j-1}^*(d,k)$.

Thus far, we have proved the optimality of symbol sliding for certain classes of (d,k) constraints, and derived a relationship between the bias p and sliding index j. Our aim now is to jointly optimize the values of p and j, so as to achieve the highest possible symbol sliding rates for the remaining suboptimal cases. The following theorem gives an expression for the average rate, $R_j(p,d,k)$, as a function of j and p.

Theorem 4.5 *The average rate of SS(j) is given by*

$$\begin{aligned} R_j(p,d,k) &= h(p)\frac{1-p^{k-d}}{1-p^{k-d}+(1-p)\left(p^{k-d-j}-jp^{k-d}+d\right)}, \quad \text{when } k<\infty \\ R_j(p,d,\infty) &= \frac{h(p)}{1+d(1-p)}. \end{aligned}$$

Proof: We first consider the case when $k < \infty$, and start with (28) wherein

$$R_j(p,d,k) = h(p)\frac{L_{in}}{L_{out}^j} \quad ,$$

and write out the expressions for L_{in} and L_{out}^j. L_{in} is the average length into the constrained encoder of Fig. 5. Since it is independent of the sliding index j, we can set $j = 0$ without any loss in generality, and compute L_{in} from Table 1. It is given by

$$L_{in} = \sum_{i=0}^{k-d} v_i^0 l_i, \tag{31}$$

where l_i is the length of the corresponding input listed in Table 1. For example, index $i = k-d-1$ has $v_i^0 = p(1-p)$ and $l_i = 2$. Writing this out, we obtain

$$\begin{aligned} L_{in} &= \sum_{i=0}^{k-d} v_i^0 l_i & (32)\\ &= 1-p+2p(1-p)+\ldots+(k-d)p^{k-d-1}(1-p)+(k-d)p^{k-d} & (33)\\ &= 1+p+p^2+p^3+p^4+\ldots+p^{k-d-1} & (34)\\ &= \frac{1-p^{k-d}}{1-p}, & (35) \end{aligned}$$

where (34) is a direct simplification of (33).

Similarly, we now write out the expression for L_{out}^{j}, the average length at the output of the constrained encoder. Clearly, this is dependent on the sliding index j. We start with the expression in (29) and write out the individual terms.

$$\begin{aligned} L_{out}^{j} &= \sum_{i=0}^{k-d} v_i^j l_i^j && (36)\\ &= (1-p)(d+1) + p(1-p)(d+2) + \ldots + p^{k-d-j-1}(1-p)(k-j) && (37)\\ &\quad + p^{k-d}(k-j+1) + p^{k-d-j}(1-p)(k-j+2) + \ldots + p^{k-d-1}(1-p)(k+1). \end{aligned}$$

Now let

$$\begin{aligned} \mathcal{S} &= L_{out}^{0} && (38)\\ &= (1-p)(d+1) + p(1-p)(d+2) + \ldots + p^{k-d-1}(1-p)k + p^{k-d}(k+1) && (39)\\ &= d + 1 + p + p^2 + p^3 + \ldots + p^{k-d} && (40)\\ &= d + \frac{1-p^{k-d+1}}{1-p}, && (41) \end{aligned}$$

where (39) is nothing but the expression for the average output length of the bit stuff algorithm. Using (37), (39) and (41), we get

$$\begin{aligned} L_{out}^{j} &= (1-p)(d+1) + p(1-p)(d+2) + \ldots + p^{k-d-j-1}(1-p)(k-j) && (42)\\ &\quad + p^{k-d}(k-j+1) + p^{k-d-j}(1-p)(k-j+2) + \ldots + p^{k-d-1}(1-p)(k+1)\\ &= \mathcal{S} - jp^{k-d} + p^{k-d-j} - p^{k-d} && (43) \end{aligned}$$

$$= \quad d + \frac{1 - p^{k-d+1}}{1-p} - jp^{k-d} + p^{k-d-j} - p^{k-d} \tag{44}$$

$$= \quad \frac{1 - p^{k-d} + (1-p)\left(p^{k-d-j} - jp^{k-d} + d\right)}{1-p}. \tag{45}$$

Substituting (35) and (45) into (28), we obtain the rate expression when $k < \infty$ as

$$R_j(p, d, k) = h(p) \frac{1 - p^{k-d}}{1 - p^{k-d} + (1-p)\left(p^{k-d-j} - jp^{k-d} + d\right)}. \tag{46}$$

For the $k = \infty$ case, the symbol sliding operations reduce to simply inserting d zeros after every "1" bit in the biased bit sequence. A straightforward computation yields

$$R_j(p, d, \infty) = \frac{h(p)}{1 + d(1-p)} \tag{47}$$

■

Theorem 4.5 gives an expression for the average rate of $\mathsf{SS}(j)$ in terms of the bias p, sliding index j and constraint parameters d, k. For a given (d, k) constraint, $k < \infty$, we are now interested in determining the values of p and j that jointly maximize $R_j(p, d, k)$. However, the complexity of the rate expression in (46) makes further analysis difficult. For this reason, optimization for both p and j is done numerically. The simulation results shown in Table 4 indicate that symbol sliding can improve over bit stuffing and bit flipping for several (d, k) constraints. Furthermore, the maximum average symbol sliding rates are comparatively stable with increasing d. Both these gains are a result of the extra degree of freedom which we have introduced, namely the symbol sliding index j, which can now be tuned in conjunction with the bias p to

achieve higher rates. Note that setting $j = 0$ in (46) yields precisely the expression in (6), Chapter 3 for the average rate of the bit stuff algorithm.

Table 4: Simulation results of rate improvements for some constraints

d	k	**Shannon capacity** $C(d,k)$	**Max. bit stuff efficiency** (%)	**Max. bit flipping efficiency** (%)	**Max. symbol sliding efficiency** (%)	**Maximizing sliding index** j
1	3	0.5515	98.93	99.74	100	2
1	7	0.6793	99.42	99.79	99.79	1
2	5	0.4650	98.47	99.74	100	3
2	10	0.5418	99.39	99.70	99.87	2
3	6	0.3746	98.23	99.57	99.89	2
4	8	0.3432	98.02	99.16	99.91	4
5	9	0.2979	97.82	98.89	99.77	3

CHAPTER V

OPTIMAL CODES USING INTERLEAVING

Thus far, in Chapters 3 and 4, we have studied the bit stuff algorithm, the bit flipping algorithm, and proposed the symbol sliding algorithm to generate (d, k) sequences. All three of these constructions used a single DT(p) to generate an appropriately biased, *i.i.d* bit sequence, which was then encoded into constrained phrases. Recently, it was observed in [3] that with the use of multiple such DTs, optimal bit stuff encoders could be constructed for all values of d and k. The idea is to generate several distinct *i.i.d*, biased bit streams, one each for a state in the FSTD that has two outgoing branches (see Fig. 4). Since the number of such states is $k - d$ for $k < \infty$, we need precisely that many DTs to construct optimal codes in this fashion. We refer to this scheme as the *multiple DT* construction.

In this chapter, we show that certain classes of (d, k) constraints allow optimal encoding using fewer than $k - d$ DTs. This is derived from the factorization of characteristic polynomials, and can be implemented using interleaving. As in the case of symbol sliding, the proposed interleaving code construction is also variable-rate. We first describe such a construction for $(d, d+2^m-1)$ constraints, $2 \leq m < \infty$, and then generalize to all (d, k) constraints with $k - d + 1$ not prime.

5.1 Optimal $(d, d + 2^m - 1)$ Codes, $2 \leq m < \infty$

Consider the set of $(d, d+2^m-1)$ phrases, $\mathcal{X}_{d,d+2^m-1} = \left\{0^{d+2^m-1}1, 0^{d+2^m-2}1, \ldots, 0^{d+1}1, 0^d1\right\}$. The corresponding set of phrase-lengths is $\mathcal{L}_{d,d+2^m-1} = \{d + 2^m, d + 2^m - 1, \ldots, d + 2, d + 1\}$. Consider a random variable X that takes on one of the 2^m possible phrase-lengths according to the maxentropic distribution, *i.e.*, $\texttt{Pr}\{X = t\} = \lambda_{d,d+2^m-1}^{-t}$, for

each $t \in \mathcal{L}_{d,d+2^m-1}$. Then the entropy $H(X)$ is given by

$$H(X) = \sum_{t=d+1}^{d+2^m} \lambda_{d,d+2^m-1}^{-t} log_2 \lambda_{d,d+2^m-1}^{t}. \tag{48}$$

The following result is central to the interleaving code construction. In the rest of this section, we simply use λ to denote $\lambda_{d,d+2^m-1}$, unless otherwise indicated.

Lemma 5.1 *The entropy of the random variable X is equal to the joint entropy of m independent Bernoulli random variables Y_i, with $\mathtt{Pr}\{Y_i = 0\} = \frac{1}{1+\lambda^{-2^i}}$, $i = 0, 1, \ldots, m-1$. That is,*

$$H(X) = \sum_{i=0}^{m-1} h\left(\frac{1}{1+\lambda^{-2^i}}\right),$$

where $h(.)$ denotes the binary entropy function, and λ is used to denote $\lambda_{d,d+2^m-1}$.

Proof: We start by rewriting (48) with $t = d + j$ to obtain

$$H(X) = \sum_{j=1}^{2^m} \lambda^{-(d+j)} log_2 \lambda^{d+j}. \tag{49}$$

To proceed further, we derive the following factorization of the characteristic polynomial of the $(d, d + 2^m - 1)$ constraint

$$\begin{aligned} G_{d,d+2^m-1}(z) &= \sum_{j=1}^{2^m} z^{-(d+j)} & (50)\\ &= z^{-(d+1)} \prod_{i=0}^{m-1} \left(1 + z^{-2^i}\right). & (51) \end{aligned}$$

Since λ is the positive, real root of the equation $G_{d,d+2^m-1}(z) = 1$, we have from (51) that

$$\lambda^{-(d+1)} \prod_{i=0}^{m-1} \left(1 + \lambda^{-2^i}\right) = 1. \tag{52}$$

Using (52), we make the following substitution in (49)

$$\lambda^{-(d+j)} = \frac{\lambda^{-(j-1)}}{\prod_{i=0}^{m-1} \left(1 + \lambda^{-2^i}\right)}, \quad j = 1, 2, \ldots, 2^m. \tag{53}$$

Simplifying the resulting expression leads us to

$$\begin{aligned}
\sum_{j=1}^{2^m} \lambda^{-(d+j)} log_2 \lambda^{d+j} &= \sum_{i=0}^{m-1} log_2 \left(1 + \lambda^{-2^i}\right) + log_2 \lambda \frac{\sum_{j=1}^{2^m-1} j\lambda^{-j}}{\prod_{i=0}^{m-1} \left(1 + \lambda^{-2^i}\right)} && (54) \\
&= \sum_{i=0}^{m-1} log_2 \left(1 + \lambda^{-2^i}\right) + log_2 \lambda \sum_{i=0}^{m-1} \frac{2^i \lambda^{-2^i}}{1 + \lambda^{-2^i}} && (55) \\
&= \sum_{i=0}^{m-1} \frac{1}{1 + \lambda^{-2^i}} log_2 \left(1 + \lambda^{-2^i}\right) + \sum_{i=0}^{m-1} \frac{\lambda^{-2^i}}{1 + \lambda^{-2^i}} log_2 \left(\frac{1 + \lambda^{-2^i}}{\lambda^{-2^i}}\right) \\
&= \sum_{i=0}^{m-1} h\left(\frac{1}{1 + \lambda^{-2^i}}\right), && (56)
\end{aligned}$$

where (55) is obtained from a partial fraction expansion of the second term in (54), and (56) is a regrouping of the terms in (55). From (49) and (56), we conclude that $H(X) = \sum_{i=0}^{m-1} h\left(\frac{1}{1+\lambda^{-2^i}}\right)$. ■

The result of Lemma 5.1 has the following implication. Since X is distributed maxentropically over the set of phrase-lengths $\mathcal{L}_{d,d+2^m-1}$, optimal $(d, d + 2^m - 1)$ codes can be generated using only m *i.i.d*, biased bit streams as opposed to the

$k-d=2^m-1$ *i.i.d*, biased bit streams required with the multiple DT construction. The bias of the m bit streams are given by $\frac{1}{1+\lambda^{-2^i}}$, $i=0,1,\ldots,m-1$. What remains now is to describe how the actual encoding is done. The following discussion on the equivalence between factorization and interleaving provides some insight.

Recollect that the equality of Lemma 5.1 was obtained using a factorization of the $(d,d+2^m-1)$ characteristic polynomial in (51). In general, the characteristic polynomial of the (d,k) constraint, $k<\infty$, is given by (2)

$$G_{d,k}(z)=\sum_{j=d+1}^{k+1} z^{-j}.$$

$G_{d,k}(z)$ is indicative of the fact that a (d,k) sequence is a concatenation of phrases from the finite set $\mathcal{X}_{d,k}=\left\{0^k1,0^{k-1}1,\ldots,0^{d+1}1,0^d1\right\}$. Factorization of $G_{d,k}(z)$ has the interpretation of interleaving phrases corresponding to the individual factors. For example, consider the characteristic polynomial of the $(1,4)$ constraint, $G_{1,4}(z)=z^{-2}+z^{-3}+z^{-4}+z^{-5}$. This can be factored as $G_{1,4}(z)=\left(z^{-1}+z^{-2}\right)\left(z^{-1}+z^{-3}\right)=G_{1,\infty}(z)G_{2,\infty}(z)$. The term $(z^{-1}+z^{-2})$ represents phrases of length one or two bits (corresponding to a $(1,\infty)$-constrained sequence). Similarly, $(z^{-1}+z^{-3})$ represents phrases of length one or three bits (corresponding to a $(2,\infty)$-constrained sequence). Interleaving the phrases corresponding to these two terms yields phrases of length two, three, four or five bits, which is in turn described by $z^{-2}+z^{-3}+z^{-4}+z^{-5}$, the characteristic $(1,4)$ polynomial. This gives the equivalence between interleaving and factorization. Note that the interleaving is based on the length of individual phrases and not their representations.

Now consider the characteristic polynomial of the $(d,d+2^m-1)$ constraint, $G_{d,d+2^m-1}(z)=\sum_{j=d+1}^{d+2^m} z^{-j}$, which can be factored as

$$G_{d,d+2^m-1}(z) \;=\; z^{-(d+1)}\prod_{i=0}^{m-1}\left(1+z^{-2^i}\right) \tag{57}$$

$$= \quad z^{-(d-m+1)} \prod_{i=0}^{m-1} G_{2^i,\infty}(z). \tag{58}$$

Equation (58) shows that $G_{d,d+2^m-1}(z)$ can be written as the product of m characteristic polynomials, each with $k = \infty$, and an additional "delay" term $z^{-(d-m+1)}$. This means that we can generate the $(d, d + 2^m - 1)$ phrases by an interleaving of the m $(2^i, \infty)$ phrases, $i = 0, 1, \ldots, m-1$. Each $(2^i, \infty)$ phrase is in turn one of two possibilities from the set $\mathcal{X}_{2^i,\infty} = \left\{0, 0^{2^i}1\right\}$. Hence, in order to generate maxentropic $(d, d + 2^m - 1)$ phrases, we need to generate the $(2^i, \infty)$ phrases with appropriate, non-maxentropic probabilities. From previous discussions in Chapter 3, we note that by using a single DT with bias p, and then bit stuffing, we can generate the corresponding $(2^i, \infty)$ phrases with $\mathtt{Pr}\{0\} = p$ and $\mathtt{Pr}\{0^{2^i}1\} = 1 - p$, for any $p \in [0, 1]$. In fact, setting $p = \lambda^{-1}_{2^i,\infty}$ generates maxentropic $(2^i, \infty)$ phrases, but for our purpose we need to generate non-maxentropic phrases, which can again be done by suitably setting p. It turns out that the required bias p to generate $(2^i, \infty)$ phrases for our code construction is given by $\frac{1}{1+\lambda^{-2^i}}$, $i = 0, 1, \ldots, m-1$.

Hence, we see that optimal $(d, d+2^m-1)$ codes can be constructed using exactly m DTs, one each for factors $G_{2^i,\infty}(z)$, $i = 0, 1, \ldots, m-1$ in (58), and then bit stuffing and interleaving. In our construction, we make a further modification. Rather than first generate the $(2^i, \infty)$ phrases, $i = 0, 1, \ldots, m-1$ and then interleave the m phrases, we directly interleave the m biased bit streams themselves, and then suitably encode the interleaved bit stream. This removes the need for individually bit stuffing each of the m biased bit streams. Our proposed code construction is shown in Fig. 6.

First, the input is split into m distinct streams using a serial-to-parallel (S/P) converter. These m streams then act as inputs to the m DTs. As before, DT(p) denotes a distribution transformer that outputs a binary *i.i.d* stream with bias p ($\mathtt{Pr}\{0\} = p$) in response to an unbiased, binary *i.i.d* input stream. The bias of the

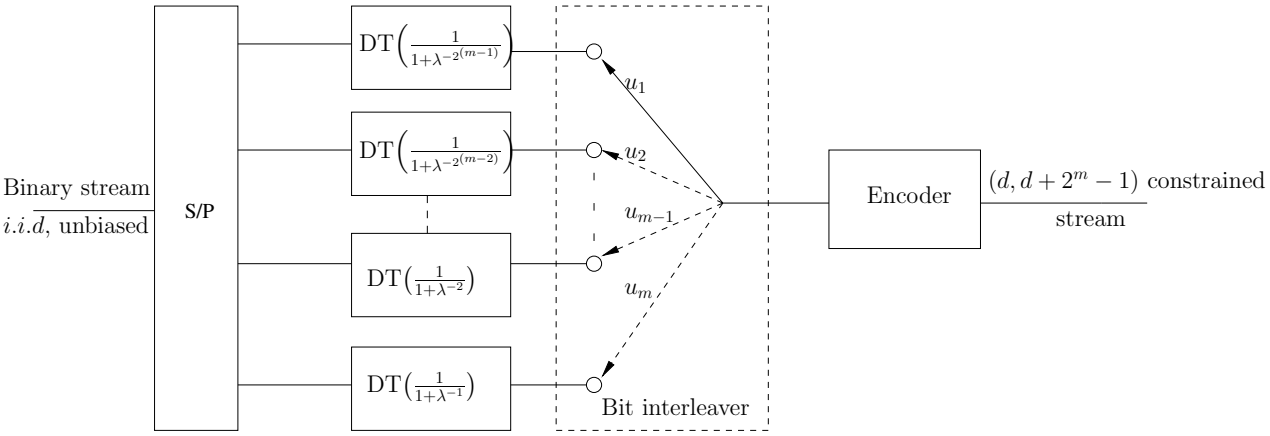

Figure 6: Block diagram of the $(d, d+2^m-1)$ code construction using interleaving. λ denotes the positive real root of $G_{d,d+2^m-1}(z) = 1$.

m DTs are chosen as $\frac{1}{1+\lambda^{-2^i}}$, $i = 0, 1, \ldots, m-1$ from Lemma 5.1, so as to generate optimal $(d, d+2^m-1)$ codes.

The m biased bit streams then act as inputs to the bit interleaver. The bit interleaver produces binary sequences $\mathbf{u} = (u_1 u_2 \ldots u_m) \in \{0,1\}^m$ by interleaving the m biased streams one bit at a time, in the specified order (u_1 is the MSB and u_m the LSB). The result of Lemma 5.1 implies that the interleaver generates the m-bit sequences according to the maxentropic $(d, d+2^m-1)$ phrase distribution, *i.e.*, the 2^m distinct interleaved sequences $(u_1 u_2 \ldots u_m)$ are generated according to the 2^m distinct maxentropic phrase probabilities of the $(d, d+2^m-1)$ constraint. Finally, the encoder performs the "phrase shaping", whereby the binary sequence $\mathbf{u}$ of decimal value j is mapped to the $(d, d+2^m-1)$ phrase $0^{d+j}1$, $j = 0, 1, 2, \ldots, 2^m-1$. Table 5 specifies the encoder mapping for $(d, d+7)$ constraints. The size of this table is eight in the example and $k-d+1 = 2^m$ in general.

The construction described above requires m DTs, one m-bit interleaver and one fixed-length to variable-length encoder. Hence, the number of required DTs is $log_2(k-d+1)$, as opposed to $k-d$ with the multiple DT construction. The average encoding rate of the interleaving construction is given by

Table 5: Encoder mapping for the $(d, d+7)$ constraint

Interleaved binary sequence u=$(u_1u_2u_3)$	**Corresponding $(d, d+7)$ constrained phrase**
000	0^d1
001	$0^{d+1}1$
010	$0^{d+2}1$
011	$0^{d+3}1$
100	$0^{d+4}1$
101	$0^{d+5}1$
110	$0^{d+6}1$
111	$0^{d+7}1$

$$R(d, d+2^m-1) = \sum_{i=0}^{m-1} \frac{h\left(\frac{1}{1+\lambda^{-2^i}}\right)}{L_{out}}, \tag{59}$$

where $L_{out} = \sum_{j=1}^{2^m}(d+j)\lambda^{-(d+j)}$ is the average phrase length at the output of the encoder. The capacity of the $(d, d+2^m-1)$ constraint can be expressed as [46]

$$C(d, d+2^m-1) = log_2\lambda = \sum_{j=1}^{2^m} \frac{\lambda^{-(d+j)} log_2\lambda^{d+j}}{L_{out}}. \tag{60}$$

It immediately follows that $R(d, d+2^m-1) = C(d, d+2^m-1)$, since $\sum_{j=1}^{2^m} \lambda^{-(d+j)} log_2\lambda^{d+j} = \sum_{i=0}^{m-1} h\left(\frac{1}{1+\lambda^{-2^i}}\right)$ from Lemma 5.1. Hence, the interleaving code construction is capacity-achieving for all $(d, d+2^m-1)$ constraints, $2 \leq m < \infty$.

5.2 Generalization to Other (d, k) Constraints

We now extend the interleaving construction proposed in Chapter 5.1 to a wider class of (d, k) constraints. As before, the idea is to derive an appropriate factorization of characteristic polynomials. For constraint parameters d and k, $0 \leq d < k < \infty$, let $k-d+1$ be written as the product of primes

$$k - d + 1 = \prod_{i=1}^{n} P_i. \tag{61}$$

Now define $\eta_i = \prod_{j=1}^{i} P_j$, $i = 1, 2, \ldots, n$, with $\eta_0 = 1$. Then, the characteristic polynomial can be factored as

$$\begin{aligned} G_{d,k}(z) &= \sum_{j=d+1}^{k+1} z^{-j} \\ &= z^{-(d+1)} \prod_{i=1}^{n} F_{d,k}^{i}(z) \quad , \end{aligned} \tag{62}$$

where each factor $F_{d,k}^{i}(z)$, $i = 1, 2, \ldots, n$, is of the form

$$F_{d,k}^{i}(z) = 1 + z^{-\eta_{i-1}} + z^{-2\eta_{i-1}} + \ldots + z^{-(P_i - 1)\eta_{i-1}}. \tag{63}$$

Similar to the discussion in Chapter 5.1, let us now define X to be a random variable that is maxentropically distributed over the set of phrase-lengths $\mathcal{L}_{d,k} = \{k+1, k, \ldots, d+1\}$. Then $\mathtt{Pr}\{X = t\} = \lambda_{d,k}^{-t}$ for each $t \in \mathcal{L}_{d,k}$, and the entropy $H(X)$ is given by

$$H(X) = \sum_{t=d+1}^{k+1} \lambda_{d,k}^{-t} log_2 \lambda_{d,k}^{t}. \tag{64}$$

Using the factorization in (62), we obtain the following result. The proof proceeds along similar lines to Lemma 5.1, and is hence omitted.

Lemma 5.2 *The entropy of the random variable X is equal to the joint entropy of n independent random variables Y_i with the following properties*

1. *Y_i is a P_i-ary random variable, $i = 1, 2, \ldots, n$.*

2. *Let $Y_i \in \mathcal{Y}_i = \{0, 1, \ldots, P_i - 1\}$. Then* $\mathtt{Pr}\{Y_i = j\} = \frac{\lambda_{d,k}^{-j\eta_{i-1}}}{F_{d,k}^{i}(\lambda_{d,k})}$, $j = 0, 1, \ldots, P_i - 1$.

According to Lemma 5.2, $H(X) = \sum_{i=1}^{n} H(Y_i)$, and since X is maxentropic, we can generate optimal (d, k) codes by interleaving the random variables Y_i, $i = 1, 2, \ldots, n$, and then suitably encoding. Now, each P_i-ary random variable Y_i can be realized using $(P_i - 1)$ DTs[1]. Hence, the total number of DTs required is $\sum_{i=1}^{n} (P_i - 1)$. As long as $k - d + 1$ is not prime, *i.e.*, the number of factors n is greater than one, this is strictly less than the $k - d$ DTs required in the multiple DT construction. In what follows, we describe how the interleaving is done using the $\sum_{i=1}^{n} (P_i - 1)$ DTs.

Similar to the previous construction in Fig. 6, Chapter 5.1, we start with a S/P converter that splits the input bit stream into $\sum_{i=1}^{n} (P_i - 1)$ distinct streams, each of which is then fed into a DT. Let the $(P_i - 1)$ DTs corresponding to the random variable Y_i, be indexed as DT_1^i, DT_2^i,..., $\mathrm{DT}_{P_i-1}^i$. The bias of DT_l^i is chosen as $\frac{1}{1+\lambda_{d,k}^{-\eta_{i-1}}+\ldots+\lambda_{d,k}^{-(P_i-l)\eta_{i-1}}}$, for $l = 1, 2, \ldots, P_i - 1$ and $i = 1, 2, \ldots, n$. Let the output bit stream of DT_l^i be indexed as $\mathcal{B}_l^i$. Then the interleaver functionality is described by the finite state transition diagram (FSTD) shown in Fig. 7. The interleaver starts in state S_1^1, and takes a bit from stream $\mathcal{B}_l^i$ when in state S_l^i. The labels on edges going out of state S_l^i in Fig. 7 denote the bits from stream $\mathcal{B}_l^i$. Finally, the code construction is completed using an encoder that suitably maps the interleaved sequences to (d, k) phrases. As an example, we now describe the code construction for the $(0, 11)$ constraint.

The characteristic $(0, 11)$ polynomial can be factored as

[1]It is as yet unknown to this author, if for any specific cases, the random variable Y_i can be implemented with even fewer than $(P_i - 1)$ DTs.

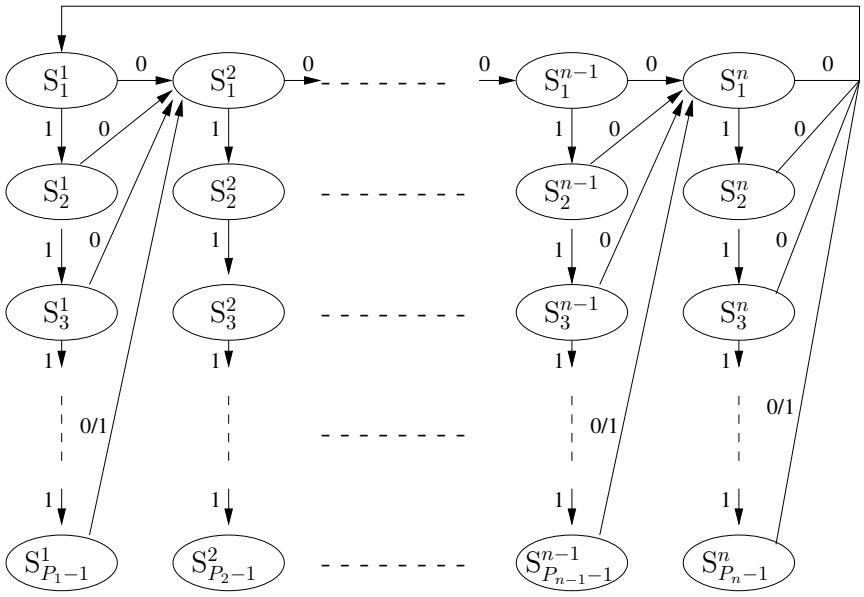

Figure 7: FSTD of the interleaver for (d, k) code construction, $k-d+1$ not prime. The interleaver takes a bit from the biased bit stream $\mathcal{B}_l^i$ when in state S_l^i. The labels on edges going out of state S_l^i denote the bits from stream $\mathcal{B}_l^i$.

$$G_{0,11}(z) = \sum_{j=1}^{12} z^{-j} \tag{65}$$

$$= z^{-1}\left(1+z^{-1}\right)\left(1+z^{-2}\right)\left(1+z^{-4}+z^{-8}\right). \tag{66}$$

Fig. 8 shows the code construction that uses four DTs, one four-bit interleaver and one variable-length encoder. The DTs with bias $\frac{1}{1+\lambda_{0,11}^{-1}}$ and $\frac{1}{1+\lambda_{0,11}^{-2}}$ correspond to factors $(1+z^{-1})$ and $(1+z^{-2})$, respectively (or to the two binary random variables Y_1 and Y_2). The remaining two DTs with bias $\frac{1}{1+\lambda_{0,11}^{-4}}$ and $\frac{1}{1+\lambda_{0,11}^{-4}+\lambda_{0,11}^{-8}}$ both correspond to the factor $(1+z^{-4}+z^{-8})$, and are effectively used to generate the ternary random variable Y_3.

The interleaver functionality can be determined from the FSTD in Fig. 7 by setting $P_1 = P_2 = 2$ and $P_3 = 3$. The interleaver starts with the biased bit stream $\mathcal{B}_1^1$ and generates a binary sequence $\mathbf{u} = (u_1u_2u_3u_4)$ by interleaving the four biased

streams one bit at a time in the specified order (u_1 is the MSB and u_4 the LSB), if $u_3 = 1$. If $u_3 = 0$, the interleaver skips $\mathcal{B}_2^3$ (shown in dotted lines) and outputs the binary sequence $\mathbf{u} = (u_1 u_2 u_3)$. The encoder then maps the binary sequence $\mathbf{u}$ to $(0, 11)$-constrained phrases as specified in Table 6. The size of this table is 12 for this example and $k - d + 1$ in general. The code construction described above requires four DTs, as opposed to 11 DTs required with the multiple DT construction.

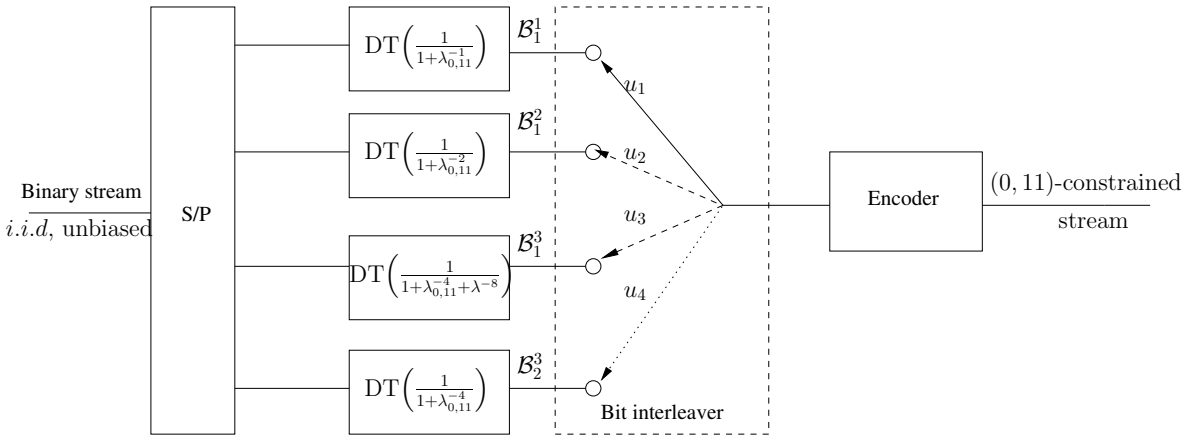

Figure 8: Block diagram of the $(0, 11)$ code construction using interleaving. $\lambda_{0,11}$ denotes the positive real root of $G_{0,11}(z) = 1$.

Table 6: Encoder mapping for the $(0, 11)$ constraint

Interleaved binary sequence u	**Corresponding $(0, 11)$ constrained phrase**
000	1
100	01
010	$0^2 1$
110	$0^3 1$
0010	$0^4 1$
1010	$0^5 1$
0110	$0^6 1$
1110	$0^7 1$
0011	$0^8 1$
1011	$0^9 1$
0111	$0^{10} 1$
1111	$0^{11} 1$

CHAPTER VI

VARIABLE-RATE CODES: EXTENSION TO OTHER RLL CONSTRAINTS

In Chapters 4 and 5, we described two variable-rate encoding algorithms that achieved capacity for certain classes of (d, k) constraints. Both algorithms were inspired by the simple concept of bit stuffing, whereby additional bits were inserted into the data sequence so as to satisfy the d and k constraints. In this chapter, we show that bit stuff encoding may be applied to other RLL constraints as well. First, we present a simple and efficient bit stuff algorithm for $(0, G/I)$ constraints. Then, we consider asymmetrical run-length constraints and multiple-spacing (d, k) constraints, and describe capacity-achieving interleaving codes in each case.

6.1 $(0, G/I)$ Constraints

$(0, G/I)$ sequences are constrained binary sequences that find applications in magnetic recording [18],[30]. "G" refers to the global constraint, which limits the run-length of consecutive "0"s that separate successive "1"s in the $(0, G/I)$ sequence, to at most G. This makes the stored data self-clocking. Thus, the "G" constraint plays exactly the same role as the k constraint described in Chapter 2.2. "I" refers to a constraint on the even and odd subsequences[1], where the maximum run-length of consecutive "0"s that separate successive "1"s in each of the subsequences is I. This is used to limit the path memory of the Viterbi detector in partial response equalization with maximum-likelihood sequence detection (PRML) based magnetic recording systems

[1]The subsequences are also called the interleaves, and hence the symbol I

[9, 18] (Chapter 8.2.1 has more details on a PRML model). Finally, the "0" in $(0, G/I)$ indicates that there is no minimum run-length constraint.

An example of a $(0, G/I)$ sequence with $G = 2$ and $I = 2$ is

$$11010010011010010111 1.$$

The even subsequence is

$$10010110011.$$

The odd subsequence is

$$1100100111.$$

Both the even and odd subsequences have at most $I = 2$ consecutive zeros, and the global $(0, G/I)$ sequence has at most $G = 2$ consecutive zeros.

A $(0, G/I)$ code is defined as an invertible mapping between unconstrained binary sequences and $(0, G/I)$ sequences. For our purpose, an unconstrained binary sequence refers to a sequence of independent, identically distributed ($i.i.d$), and equally likely ($\texttt{Pr}\{0\}=\texttt{Pr}\{1\}=1/2$) bits, as seen in Chapter 3 earlier. It is well known that the maximum possible encoding rate of any $(0, G/I)$ code is equal to the capacity $C_{G/I}$, of the $(0, G/I)$ constraint. A general method for calculating $C_{G/I}$ is presented in [56], where the authors also provide a list of $C_{G/I}$ values for $0 < G \leq 15$ and $0 < I \leq 10$.

In the following section, we describe a variable-rate algorithm to generate near-capacity $(0, G/I)$ codes. As in all our previous constructions, we use the simple concept of bit stuffing to guide our encoding.

6.1.1 A Variable-Rate Bit Stuff Algorithm

Bit stuffing [7] refers to a controlled addition of bits into the data sequence so as to satisfy certain constraints. We presented a detailed review of the bit stuff algorithm [6] for (d, k) constraints in Chapter 3. Along similar lines, we propose the following

bit stuff algorithm for encoding $(0, G/I)$ sequences, $0 < I < \infty$, $0 < G \leq 2I$. We wish to clarify that the proposed bit stuff encoder for $(0, G/I)$ constraints does not include any biasing, *i.e.*, there is no DT in our set-up here, as opposed to that shown in Fig. 3, Chapter 3.1 for (d, k) constraints.

Let us assume that the bit stuff encoder scans the data sequence at the rate of one bit per unit time. Let $b(n) \in \{0, 1\}$ denote the bit scanned at time n. The encoder keeps track of the following three variables.

1. $i_c(n)$: number of trailing zeros in the current subsequence.

2. $i_{\overline{c}}(n)$: number of trailing zeros in the alternate subsequence.

3. $g(n)$: number of trailing zeros in the global sequence.

We use the term "trailing zeros" to imply the number of consecutive zeros since the last "1", and the term "current subsequence" to refer to the subsequence (even or odd) that contains $b(n)$. Hence, if the current subsequence happens to be the even subsequence, then the alternate subsequence is the odd subsequence, and vice versa. The three variables, $i_c(n)$, $i_{\overline{c}}(n)$ and $g(n)$ are initialized to zero, and recursively computed as

$$i_c(n) = \begin{cases} i_{\overline{c}}(n-1) + 1 & \text{if } b(n) = 0 \\ 0 & \text{if } b(n) = 1, \end{cases} \tag{67}$$

$$i_{\overline{c}}(n) = i_c(n-1) \tag{68}$$

$$g(n) = \begin{cases} g(n-1) + 1 & \text{if } b(n) = 0 \\ 0 & \text{if } b(n) = 1, \end{cases} \tag{69}$$

$g(n)$ may also be computed from $i_c(n)$ and $i_{\overline{c}}(n)$ as follows

$$g(n) \;=\; \begin{cases} 2i_{\overline{c}}(n)+1 & \text{when } i_c(n) > i_{\overline{c}}(n) \\ 2i_c(n) & \text{when } i_c(n) \le i_{\overline{c}}(n). \end{cases} \tag{70}$$

With knowledge of $i_c(n)$, $i_{\overline{c}}(n)$ and $g(n)$, the encoder then performs the following bit insertion steps at each time instant n. For simplicity, we do not explicitly include the time index in the rest of our discussions.

```
If i_c = I
  If i_c̄ < I and g = G
    Insert ‘‘11’’ in the global sequence
  Else
    Insert ‘‘1’’ in the current subsequence
  end
Elseif g = G
  If i_c̄ < I
    Insert ‘‘1’’ in the global sequence
  end
end
```

The above algorithm is applicable for the entire range of G and I with $0 < I < \infty$, $0 < G \le 2I$. Essentially, at each time instant, the encoder computes i_c, $i_{\overline{c}}$ and g, then checks for impending violations of the I and G constraints, and inserts bits in a controlled manner so as to prevent these violations. Decoding involves removal of the inserted bits and is a simple inverse of the encoding.

The above-stated bit insertion rules can be further simplified by breaking down the algorithm into four classes.

Class 1: $G = 2I$

```
If i_c = I
    Insert ‘‘1’’ in the current subsequence
end
```

Class 2: $G < 2I$, G even

```
If i_c = I
    Insert ‘‘1’’ in the current subsequence
Elseif g = G
  If i_c̄ < I
    Insert ‘‘1’’ in the global sequence
  end
end
```

Class 3: $G < 2I - 1$, G odd

```
If i_c = I
  If g = G
    Insert ‘‘11’’ in the global sequence
  Else
    Insert ‘‘1’’ in the current subsequence
  end
Elseif g = G
    Insert ‘‘1’’ in the global sequence
end
```

Class 4: $G = 2I - 1$

```
If i_c = I
  If i_c̄ = I − 1
```

```
      Insert ‘‘11’’ in the global sequence
    Else
      Insert ‘‘1’’ in the current subsequence
    end
  end
```

As an example, let us encode the following 18-bit data sequence using the above-described algorithm for $G = 2$ and $I = 2$.

$$100001010111001001$$

Clearly, the $G = 2$ constraint is violated in the data sequence as it contains runs of four and three consecutive zeros. The $I = 2$ constraint is also violated by the even subsequence. Using the bit stuff encoding algorithm for Class 2, the encoded sequence is

$$100\mathbf{1}00\mathbf{1}10101\mathbf{1}1100\mathbf{1}100\mathbf{1}1$$

The inserted bits are shown in bold. The even subsequence after encoding is

$$100\mathbf{1}00\mathbf{1}10101.$$

The odd subsequence after encoding is

$$0\mathbf{1}011110\mathbf{1}0\mathbf{1}.$$

Both the $G = 2$ and $I = 2$ constraints are satisfied in the encoded sequence. Note that there are 5 inserted bits in this case. It can be verified that for the all-zero data sequence of length 18 bits, the corresponding number of bit insertions is 9. This is an example of the variable-rate nature of the proposed algorithm.

6.1.2 Rate Computation

Having described a bit stuff algorithm for $(0, G/I)$ constraints, we now proceed to compute its encoding rate. Since the encoding is variable-rate, we define the average encoding rate as in Chapter 3.3

$$R_{G/I} = \frac{\text{Average input length}}{\text{Average output length}} = \frac{L_{in}}{L_{out}}.$$

The special case of Class 1 is rather simple. We note that the algorithm for Class 1 inserts a "1" whenever the subsequence maximum run-length I is attained. This in effect, ensures that the global constraint is also satisfied. Hence, with respect to the individual subsequences, the Class 1 algorithm is identical to the bit stuff algorithm for $(0, k)$ constraints [6] with $k = I$. Since the global sequence is formed by interleaving the subsequences, it follows that the average encoding rate for $(0, 2I/I)$ constraints is the same as the average encoding rate for $(0, I)$ constraints, as derived in [6]. Hence, we have

$$R_{2I/I} = \frac{2^{I+1} - 2}{2^{I+1} - 1}. \tag{71}$$

To compute the average encoding rate for Classes 2, 3 and 4, we use a Markov chain interpretation of the bit stuff encoder. The states in the finite state transition diagram (FSTD) of the Markov chain, all correspond to $i_c = 0$, *i.e.*, $b = 1$ from (67). The state labels are σ_j, $j = 0, 1, \ldots, I-1$, which represent the possible values of $i_{\overline{c}}$. Hence, the encoder is said to be in state σ_j when $i_{\overline{c}} = j$ and $i_c = 0$. A state transition occurs when a certain input word from the input alphabet, is encoded into the corresponding output word. The input alphabet for state σ_j is

$$X_j = \left\{1,\ 01,\ \ldots,\ 0^{\mathcal{J}-1}1,\ 0^{\mathcal{J}}\right\}, \quad j = 0, 1, \ldots, I-1,$$

where $\mathcal{J} = \min\{G, \overline{j}\}$, and $\overline{j} = 2(I - j)$. The corresponding output words are class-dependent and shown in Table 7 for Classes 2 and 3, and in Table 8 for Class 4. The inserted bits are marked in bold. The corresponding next-states can also be computed from the output words in Tables 7 and 8.

The individual input words of the alphabet X_j in state j, $j = 0, 1, \ldots, I-1$, are denoted as

$$X_j(i) = \begin{cases} 0^i 1 & \text{when } i = 0, 1, \ldots, \mathcal{J} - 1 \\ 0^{\mathcal{J}} & \text{when } i = \mathcal{J}. \end{cases}$$

$Y_j(i)$ and $Z_j(i)$ are used to denote the corresponding output word and next-state, respectively. Assuming the data bits to be independent, identically distributed ($i.i.d$), and equally likely ($\mathtt{Pr}\{0\}$=$\mathtt{Pr}\{1\}$=1/2), we can construct the transition probability matrix $\mathcal{Q}$, with elements

$$\begin{aligned} q_{jl} &= \text{Transition probability from state } j \text{ to state } l && (72) \\ &= \sum_{X_j(i):Z_j(i)=l} \mathtt{Pr}\{X_j(i)\} && (73) \\ &= \sum_{X_j(i):Z_j(i)=l} \begin{cases} 2^{-(i+1)} & \text{when } i = 0, 1, \ldots, \mathcal{J} - 1 \\ 2^{-\mathcal{J}} & \text{when } i = \mathcal{J}, \end{cases} && (74) \end{aligned}$$

for $0 \leq j, l \leq I-1$. The steady state probability distribution vector $\boldsymbol{\pi} = [\pi_0\ \pi_1\ \ldots\ \pi_{I-1}]$, where π_j is the steady-state probability of state j, can be found by simultaneously solving the equations

$$\boldsymbol{\pi}\mathcal{Q} = \boldsymbol{\pi} \tag{75}$$

Table 7: Input-output mapping for state transitions from state j in the FSTD: Classes 2 and 3

Input word	**Output word Class 2** $\overline{j} \leq G$	**Output word Class 2** $\overline{j} > G$	**Output word Class 3** $\overline{j} - 1 = G$	**Output word Class 3** $\overline{j} - 1 > G$	**Output word Class 3** $\overline{j} - 1 < G$
1	1	1	1	1	1
01	01	01	01	01	01
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$0^{\mathcal{J}-2}1$	$0^{\mathcal{J}-2}1$	$0^{\mathcal{J}-2}1$	$0^{\mathcal{J}-2}1$	$0^{\mathcal{J}-2}1$	$0^{\mathcal{J}-2}1$
$0^{\mathcal{J}-1}1$	$0^{\mathcal{J}-1}\mathbf{1}\mathbf{1}$	$0^{\mathcal{J}-1}1$	$0^{\mathcal{J}-1}1$	$0^{\mathcal{J}-1}1$	$0^{\mathcal{J}-1}\mathbf{1}\mathbf{1}$
$0^{\mathcal{J}}$	$0^{\mathcal{J}}\mathbf{1}$	$0^{\mathcal{J}}\mathbf{1}$	$0^{\mathcal{J}}\mathbf{1}\mathbf{1}$	$0^{\mathcal{J}}\mathbf{1}$	$0^{\mathcal{J}}\mathbf{1}$

Table 8: Input-output mapping for state transitions from state j in the FSTD: Class 4

Input word	**Output word Class 4** $j = 0$	**Output word Class 4** $j > 0$
1	1	1
01	01	01
$\vdots$	$\vdots$	$\vdots$
$0^{\mathcal{J}-2}1$	$0^{\mathcal{J}-2}1$	$0^{\mathcal{J}-2}1$
$0^{\mathcal{J}-1}1$	$0^{\mathcal{J}-1}1$	$0^{\mathcal{J}-1}\mathbf{1}\mathbf{1}$
$0^{\mathcal{J}}$	$0^{\mathcal{J}}\mathbf{1}\mathbf{1}$	$0^{\mathcal{J}}\mathbf{1}$

$$\sum_{j=0}^{I-1} \pi_j = 1. \tag{76}$$

The average encoding rate can now be computed as $R_{G/I} = \frac{L_{in}}{L_{out}}$ where

$$L_{in} = \sum_{j=0}^{I-1} \pi_j L_{in}^j, \tag{77}$$

$$L_{out} = \sum_{j=0}^{I-1} \pi_j L_{out}^j, \tag{78}$$

where

$$L_{in}^{j} = 2(1-2^{-\mathcal{J}}), \quad j=0,1,\ldots,I-1, \tag{79}$$

$$L_{out}^{j} = L_{in}^{j}+\delta^{j}, \quad j=0,1,\ldots,I-1, \tag{80}$$

with δ^j being class-dependent as shown below.

$$\delta^{j}=\begin{cases} 2^{-(\mathcal{J}-1)} & \text{for Class } 2, \overline{j} \leq G \\ 2^{-\mathcal{J}} & \text{for Class } 2, \overline{j} > G \\ 2^{-(\mathcal{J}-1)} & \text{for Class } 3, \overline{j}-1 = G \\ 2^{-\mathcal{J}} & \text{for Class } 3, \overline{j}-1 > G \\ 2^{-(\mathcal{J}-1)} & \text{for Class } 3, \overline{j}-1 < G \\ 2^{-(\mathcal{J}-1)} & \text{for Class } 4, j=0 \\ 2^{-(\mathcal{J}-1)} & \text{for Class } 4, j>0 \end{cases} \tag{81}$$

Table 9: $r_{G/I}$ values for $I \leq 5$ and $G \leq 10$, where $R_{G/I} = \frac{r_{G/I}}{r_{G/I}+1}$

$G\downarrow /I\rightarrow$	1	2	3	4	5
1	1.0000	1.5000	1.7500	1.8750	1.9375
2	2.0000	4.2857	5.4783	5.8621	5.9650
3		5.0000	8.0000	10.4000	12.0000
4		6.0000	12.1370	19.1623	24.4714
5			13.0000	22.8000	34.0000
6			14.0000	28.0659	49.0581
7				29.0000	54.0000
8				30.0000	60.0322
9					61.0000
10					62.0000

Table 10: $r_{G/I}$ values for $6 \leq I \leq 10$ and $G \leq 15$, where $R_{G/I} = \frac{r_{G/I}}{r_{G/I}+1}$

$G \downarrow / I \rightarrow$	6	7	8	9	10
1	1.9688	1.9844	1.9922	1.9961	1.9980
2	5.9912	5.9978	5.9995	5.9999	6.0000
3	12.9412	13.4545	13.7231	13.8605	13.9300
4	27.5068	28.9411	29.5630	29.8218	29.9277
5	44.1818	51.6842	56.4000	59.0746	60.5038
6	73.6496	95.0661	109.6340	117.9166	122.1513
7	90.3636	134.0000	175.7391	207.8462	228.6479
8	111.6845	188.3366	280.2618	365.7298	428.5850
9	117.5294	212.7368	353.4783	526.0000	694.8511
10	124.0159	238.8873	434.7865	725.9833	1081.0072
11	125.0000	245.2727	465.2000	837.3846	1391.7021
12	126.0000	252.0079	494.4559	940.8926	1700.0972
13		253.0000	501.1385	975.1940	1843.1831
14		254.0000	508.0039	1006.2311	1961.5864
15			509.0000	1013.0698	1998.1221

Table 11: Efficiency $\left(\frac{R_{G/I}}{C_{G/I}}(\%)\right)$ of $I = 6$ codes

G	$R_{G/6}$	$C_{G/6}$	**Efficiency (%)**
1	0.663158	0.693471	95.6288
2	0.856963	0.878850	97.5096
3	0.928270	0.944540	98.2775
4	0.964921	0.972930	99.1774
5	0.977867	0.984320	99.3444
6	0.986604	0.990114	99.6455
7	0.989055	0.992304	99.6726
8	0.991126	0.993509	99.7601
9	0.991563	0.993899	99.7650
10	0.992001	0.994119	99.7869
11	0.992063	0.994167	99.7883
12	0.992126	0.994192	99.7922

As an example, we show the FSTD of the $(0, 3/2)$ bit stuff encoder in Fig. 9. This belongs to Class 4 encoding. The input alphabets for the two states are $X_0 = \{1, 01, 001, 000\}$ and $X_1 = \{1, 01, 00\}$. The entries of the transition probability matrix are $q_{00} = 0.625$, $q_{01} = 0.375$, $q_{10} = 0.75$ and $q_{11} = 0.25$. Steady-state probabilities are

$\pi_0 = 2/3$, $\pi_1 = 1/3$, and $R_{3/2}$ is computed to be 5/6.

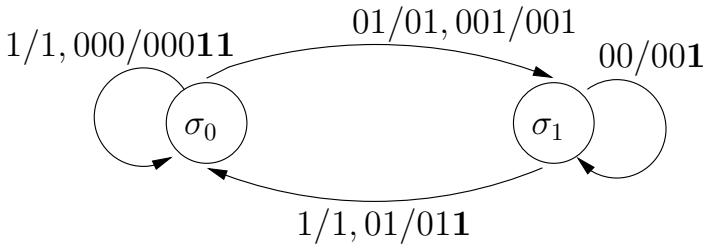

Figure 9: FSTD of the $(0, 3/2)$ bit stuff encoder. The transition labels are specified as input/output. Inserted bits are shown in bold.

Let

$$R_{G/I} = \frac{r_{G/I}}{r_{G/I} + 1}.$$

A full list of $r_{G/I}$ values is provided in Tables 9 and 10 for $1 \leq I \leq 10$ and $G \leq 15$. The $r_{G/I}$ values for trivial constraints $G > 2I$, are not given because they are equal to those when $G = 2I$. The familiar reader may also recollect that $C_{2I/I} = C(0, I) = \lim_{G\to\infty} C_{G/I} = \lim_{G\to\infty} C_{I/G}$, where $C(0, I)$ is the capacity of the $(0, I)$ constraint, as introduced earlier in (1), Chapter 1 with $d = 0$ and $k = I$. It is seen from Tables 9 and 10 that very high-rate $(0, G/I)$ codes can be generated using the proposed bit stuff algorithm. Table 11 shows the average rate, capacity (taken from [56]), and the efficiency (average rate/capacity (%)) for $I = 6$ codes as an example.

6.2 Asymmetrical Run-Length Constraints

Asymmetrical RLL constraints are found in high-density optical storage. With increasing information density, the lengths of the nonmarks diminish, but the written marks cannot be made arbitrarily small, due to limitations on the physical write process. For example, significant asymmetry exists between marks and nonmarks in

write-once and erasable optical recording [17]. There are also other instances where asymmetrical RLL coding may be used purely to combat the effects of intersymbol interference [24, 38, 39].

Asymmetrical RLL sequences are characterized by four parameters (d^+, k^+) and (d^-, k^-), $0 \leq d^+ < k^+$, $0 \leq d^- < k^-$, which describe the constraints on alternate run-lengths of "1"s and "-1"s in the $1/-1$ RLL write sequence (see Chapter 2.2 for details). Hence, an asymmetrical RLL sequence is composed of alternate strings of the form 1^i, $i \in \{d^+ + 1, d^+ + 2, \ldots, k^+ + 1\}$, and $(-1)^i$, $i \in \{d^- + 1, d^- + 2, \ldots, k^- + 1\}$. In Chapter 2.2, we had described symmetrical 1/-1 RLL sequences, where $d^+ = d^- = d$ and $k^+ = k^- = k$. An example of asymmetrical written data is shown in Fig. 10. This must be compared with the symmetrical marks and nonmarks of Fig. 2, Chapter 2.2.

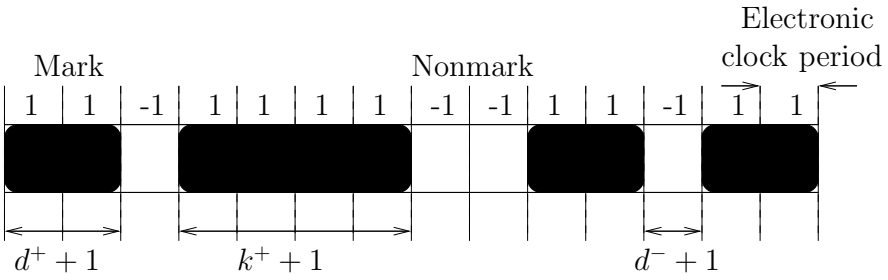

Figure 10: Asymmetrical marks and nonmarks written on a recording surface. In this example, the marks can be of length greater than $d^+ + 1 = 2$, but less than $k^+ + 1 = 4$, similar to that in Fig. 2, Chapter 2.2. However, the marks are now packed closer together, thus decreasing the minimum nonmark size to $d^- + 1 = 1$.

By now, the reader might have grasped that an asymmetrical RLL sequence can be thought of as interleaved 1^i and $(-1)^i$ "phrases". The notion of phrases here is similar to that of the (d, k) phrases in Chapter 3, and the interleaving corresponds to a factorization of the characteristic polynomial, as seen in Chapter 5. Hence, the characteristic equation of the asymmetrical RLL constraint is given by

$$\left(\sum_{i=d^+ +1}^{k^+ +1} z^{-i}\right)\left(\sum_{j=d^- +1}^{k^- +1} z^{-j}\right) = 1 \tag{82}$$

We can now use similar ideas to the interleaving construction in Chapter 5.2. According to the discussion therein, the term $\sum_{i=d^+ +1}^{k^+ +1} z^{-i}$ can be realized using $D^+ = \sum_{i^+ =1}^{n^+} (P_{i^+} - 1)$ DTs, where n^+ and P_{i^+} play a similar role to n and P_i in the discussion in Chapter 5.2. As long as $k^+ - d^+ + 1$ is not prime, D^+ is guaranteed to be less than $k^+ - d^+$. We have a similar interpretation of the term $\sum_{i=d^- +1}^{k^- +1} z^{-i}$, where $D^- = \sum_{i^- =1}^{n^-} (P_{i^-} - 1)$ is the number of required DTs. Hence, maxentropic phrases corresponding to the product of the two terms, *i.e.*, $\left(\sum_{i=d^+ +1}^{k^+ +1} z^{-i}\right)\left(\sum_{j=d^- +1}^{k^- +1} z^{-j}\right)$ can be realized using $D^+ + D^-$ DTs. This is a method of generating capacity-achieving asymmetrical RLL codes for any $0 \leq d^+ < k^+$ and $0 \leq d^- < k^-$.

6.3 Multiple-Spacing Run-Length Constraints

Multiple-spacing RLL constraints provide an extension of (d, k) constraints. They are mentioned in [17], although no known practical application exists to date. Originally due to Funk [16], multiple-spacing (d, k) constraints have an additional degree of freedom, namely the spacing of the sequence s. The phrases for such a (d, k, s) constrained system are of the form $0^i 1$, with $i = d + js$, $j = 0, 1, 2, \ldots, (k - d)/s$. It is required that $k - d$ be a multiple of the spacing s. As usual, d and k represent the minimum and maximum allowable run-length, respectively. The finite state transition diagram for the (d, k, s)=$(1, 15, 2)$ constraint in shown in Fig. 11.

The capacity $C(d, k, s)$, of the multiple-spacing (d, k) constraint is given by

$$C(d, k, s) = \log_2 \lambda, \tag{83}$$

Table 12: Encoder mapping for the $(d, d+14, 2)$ constraint

Interleaved binary sequence u=$(u_1u_2u_3)$	**Corresponding $(d, d+14, 2)$ constrained phrase**
000	0^d1
001	$0^{d+2}1$
010	$0^{d+4}1$
011	$0^{d+6}1$
100	$0^{d+8}1$
101	$0^{d+10}1$
110	$0^{d+12}1$
111	$0^{d+14}1$

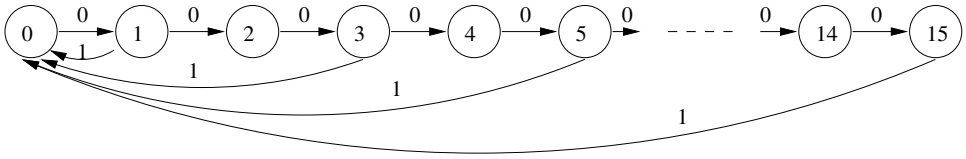

Figure 11: Finite state transition diagram for the (d, k, s)=$(1, 15, 2)$ constraint. Note the difference from Fig. 4, Chapter 3.3, in the branches returning to state 0.

where λ is the largest, real root of the characteristic equation

$$\sum_{j=0}^{(k-d)/s} z^{-(d+js+1)} = 1 \tag{84}$$

The interleaving constructions of Chapter 5 are also applicable to multiple-spacing constraints. The only difference arises in the encoder mappings between the biased, interleaved words and the phrase-lengths, where we now incorporate the spacing s. For example, the encoder mapping for the $(d, d+14, 2)$ interleaving code construction is shown in Table 12, whereas the encoder mapping for the $(d, d+7, 1)$ interleaving code construction is exactly the same as that given in Table 5, Chapter 5.1. The extension of the general interleaving constructions of Chapter 5.2 is also similarly straightforward.

CHAPTER VII

FIXED-RATE BIT STUFF (FRB) CODES

Thus far, we have developed two variable-rate code constructions based on symbol sliding and interleaving. Both constructions were inspired by the bit stuff algorithm [6], which employs the simple idea of inserting additional bits to satisfy the required constraints. Although the bit stuff algorithm is simple, and can lead to efficient codes for a wide range of constraints, the encoding is variable-rate, which is unacceptable in most practical systems. For example, magnetic and optical disk recording systems are designed to operate with fixed-length track-sectors, which means that any given fixed-length input sequence must be translated into a fixed-length channel bit sequence. As a consequence, bit stuffing, symbol sliding and interleaving-based codes discussed thus far, are mainly of theoretical interest and of limited practical value in today's recording systems.

As an aside, we point out that variable-rate bit stuffing has indeed been used in digital communication systems [7] for frame synchronization. Users in such a communication system transmit and receive data in frames. The beginning of a frame is identified by a reserved marker pattern, which must not occur elsewhere within the data frame. Hence, the data is "constrained" to omit this marker pattern. Bit stuffing eliminates any marker patterns by suitably inserting additional bits into the data frame. Indeed, such a scheme is variable-rate, but several digital communication protocols operate with variable-length frames, and hence variable-rate bit stuffing is acceptable in these systems. On the other hand, recording systems are designed to only operate with fixed-rate encoding and are intolerant to fluctuating-length channel bit sequences.

This part of our research is devoted to addressing the variable-rate problem of bit stuffing. Unless otherwise stated, the remaining discussions in this thesis assume that the bit stuff encoder does not include any biasing, *i.e.*, there is no DT in our set-up for fixed-rate encoding (see Fig. 3).

In this chapter, we introduce the fixed-rate bit stuff (FRB) algorithm for efficiently encoding and decoding $(0, k)$ sequences, which find applications in magnetic recording systems. The FRB algorithm can be viewed as a fixed-rate version of the variable-rate bit stuff algorithm proposed by Bender and Wolf [6]. High encoding efficiency is achieved by iterative pre-processing of the fixed-length input data sequence, so as to conform it to subsequent bit insertion. The encoder then inserts bits to produce a fixed-length output word. Rate computations for the proposed encoding algorithm suggest that encoding rates very close to the average rate of the variable-rate bit stuff code are possible with long, fixed-length input and output blocks. Variable-rate bit stuffing is in turn near-optimal for the special class of $(0, k)$ constraints. Hence, near-capacity $(0, k)$ codes can be designed using the FRB algorithm by encoding in long, fixed-length input and output blocks.

As explained earlier in Chapter 2, there is a long history of fixed-rate (d, k) codes and they are part of virtually all magnetic and optical disk recording systems today. Several encoding algorithms have been proposed over the years ([17] provides a comprehensive review), with the design goal being two-fold: high encoding rate and simple implementation. However, since (d, k) sequences are nonlinear (they do not constitute a linear vector sub-space), the design of near-optimal, fixed-rate code constructions is often restricted by their complexity, more so than their linear error correcting counterparts [5]. Thus, there continues to be a need for low-complexity algorithms that achieve high encoding rates.

Specifically, the design of very high-rate codes, *e.g.*, rate 100/101 and rate 200/201

$(0, k)$ codes, is of considerable interest in current recording systems. In the past, low-rate $(0, k)$ codes were easily designed using look-up tables. Examples are the rate 4/5, $k = 2$ code and the rate 8/9, $k = 3$ code [[17], Sec. 5.6]. However, current recording systems are capable of working with higher values of k, in which case the above codes are prohibitively low-rate. For typical values of $k \geq 5$ used in practice, the capacity $C(0, k)$ is well-approximated by [17]

$$C(0, k) \simeq \frac{2^{k+2} \ln 2 - 1}{2^{k+2} \ln 2}. \tag{85}$$

Hence, the design of $(0, k)$ codes with rate $(n-1)/n$, for large integers n (preferably close to $2^{k+2} \ln 2$), is of considerable interest. However, encoding using direct look-up becomes impractical with increasing n.

Immink and van Wijngaarden proposed an elegant construction [20] to solve this problem. They designed rate $(n-1)/n$ codes for any odd n, $n \geq 9$, such that at most eight bits from the source word need to be altered to obtain the constrained bit sequence. The maximum run-length of their construction is $k = 1 + \lfloor n/3 \rfloor$. Apart from simple encoding and decoding, their code has the added virtue that a single channel bit error propagates through to at most eight data bits. Details of a rate 16/17, $k = 6$ code based on this technique can be found in [[17], pp. 102-103]. Some other combinatorial constructions for high-rate $(0, k)$ codes are discussed in [60].

An alternative to combinatorial techniques is to use enumerative coding (see [[17], Chap.6] for a summary,[10],[43],[19] for more details), which has been shown to achieve very high encoding rates that approach capacity with increasing codeword length n. However, the disadvantages of enumerative coding are bitwise encoding and decoding; and additions and comparisons with pre-stored, n-bit weighting coefficients. A detailed comparison of the FRB codes with the corresponding enumeration and combinatorial codes is presented in Chapter 8.5. For the moment, it suffices to say that

the FRB encoding/decoding is simpler than that of enumeration, and it achieves higher rates than that of the combinatorial construction of [20].

Before proceeding to describe the FRB algorithm, we revisit the variable-rate bit stuff algorithm for the special case of $(0, k)$ constraints. With the minimum run-length $d = 0$, the bit stuff algorithm takes an especially simple form, and is surprisingly efficient. The bit stuff algorithm, as presented in the early work of Lee, then Bender and Wolf in [6], generates a $(0, k)$ sequence from an arbitrary binary sequence by simply inserting a "1" after every run of k consecutive "0"s. It is well known [61] that $(0, k)$ sequences can be described as a concatenation of phrases from the set $\mathcal{X}_k = \left\{1, 01, 0^2 1, \ldots, 0^{k-1}1, 0^k 1\right\}$, where as usual, 0^i denotes a run of i consecutive zeros. Hence, the bit stuff algorithm induces a reversible mapping from input words to the $(0, k)$ phrases as shown in Table 13.

Table 13: Bit stuff mapping for $(0, k)$ constraints

Input word	**$(0, k)$ phrase**
1	1
01	01
$\vdots$	$\vdots$
0^j1	0^j1
$\vdots$	$\vdots$
$0^{k-1}1$	$0^{k-1}1$
0^k	0^k1

The above mapping is said to be *variable-rate* because the ratio of the input word length to the corresponding $(0, k)$ phrase length is not the same for all the $(0, k)$ phrases. Hence, different input sequences of the same length can give rise to $(0, k)$-constrained sequences of fluctuating lengths. For instance, with $k = 3$, input sequences $0^9 1$ and $1^9 0$, each of length 10 bits, produce $(0, k)$ bit stuff output sequences $0^3 10^3 10^3 11$ and $1^9 0$, of lengths 13 bits and 10 bits, respectively. Such rate fluctuations are unacceptable in recording systems. However, there are two useful properties of

the bit stuff algorithm: stream encoding, and high average encoding rate. Stream encoding means that the encoding process can proceed as the input streams along, without the need for any look-up tables. This implies that encoding in long input blocks is feasible. In this limit of long input blocks, the bit stuff algorithm can yield rates very close to capacity. Assuming the input bits to be unconstrained, *i.e.*, independent, identically distributed (*i.i.d*) and unbiased ($\mathtt{Pr}\{0\} = \mathtt{Pr}\{1\} = 1/2$), the average rate of the bit stuff algorithm, $R_0(0,k)$, can be expressed as [6]

$$R_0(0,k) = \frac{2^{k+1}-2}{2^{k+1}-1} \tag{86}$$

This is exactly the same as equation (7) in Chapter 3.4 with $p = 1/2$ (absence of DT) and $d = 0$. Comparing (86) and (85), we see that $R_0(0,k)/C(0,k)$ is very close to unity for typical maximum-run-length parameters $5 \leq k \leq 15$ used in practice.

One simple solution to generate fixed-rate bit stuff codes is to use additional dummy bits to pad-up the variable-length bit stuff outputs to an appropriate, fixed output length. These dummy bits can then be ignored during the constrained decoding. We refer to such a simple, fixed-rate encoding as plain bit stuffing. This method has been adopted for the design of weakly-constrained codes in [22], where constraint violation is permitted with a small probability. However, the constraints considered in this work are not weak (no constraint violations permitted), and in such cases plain bit stuffing results in sizeable rate loss. Hence, for strict $(0,k)$ constraints, we are faced with the dual problem of remedying variable-rates and maintaining high encoding rate at the same time. In what follows, we outline a systematic procedure to build near-capacity, fixed-rate $(0,k)$ codes based on the bit stuff algorithm. While using dummy bits similar to [22], we introduce an additional iterative pre-processing of the fixed-length input that is central to achieving high encoding rates. Essentially, the role of pre-processing is to better conform the input data to bit insertion. As a

result, the processed input incurs lesser stuffed bits, thus leading to higher encoding rates. We now illustrate this idea using the simplest case of $k = 1$ codes, and then generalize in Chapter 7.2.

7.1 Motivating Example: Fixed-Rate $k = 1$ Codes

The $(0, 1)$ bit stuff mapping is specified in Table 14, with the individual rates given in the rightmost column. With $k = 1$, the bit stuff algorithm inserts a "1" after every "0" in the input data sequence, thus mapping an input "0" bit to the constrained phrase "01", and leaving the input "1" bits unchanged.

Table 14: Bit stuff mapping for the $k = 1$ constraint

Input word	$k = 1$ **constrained phrase**	**Rate**
1	1	1
0	01	1/2

Clearly, the rate of any $k = 1$ fixed input-length, fixed output-length, plain bit stuff encoder is limited to 1/2, by the worst case "all zeros" input. This must be compared to the average rate, $R_0(0, 1) = 2/3$, of the corresponding variable-rate bit stuff code. In what follows, we describe how to generate fixed-rate $k = 1$ codes with encoding rates approaching 2/3.

The block diagram of our proposed construction is shown in Fig. 12. The input to the fixed-rate encoder is a binary sequence $\mathbf{x}$ of length m bits, *i.e.*, $\mathbf{x} \in \mathbb{Z}_2^m$, where $\mathbb{Z}_2 = \{0, 1\}$. The output is a constrained sequence of length n bits. The encoding proceeds in three steps. The input sequence first undergoes pre-processing, followed by variable-rate bit stuff encoding, and finally dummy-bit padding to the fixed output length n. The pre-processing operation involves scanning the input sequence $\mathbf{x}$, and counting the number of "0"s, denoted by $w_0(\mathbf{x})$. If the number of "0"s in $\mathbf{x}$ is greater than the number of "1"s, then the *pre-processing output sequence* $\mathbf{v}$ is formed by

flipping every bit of $\mathbf{x}$ and appending a "0" bit as index. Else, the sequence $\mathbf{v}$ is formed by appending a "1" bit as index to the input sequence $\mathbf{x}$. Here, the flipping operation refers to a "0" bit being changed to a "1" bit and vice versa. In Fig. 12, $\overline{\mathbf{x}}$ denotes the flipped input, and $\overline{\mathbf{x}} \parallel 0$ indicates the concatenation of sequence $\overline{\mathbf{x}}$ with the index bit "0". The index bits are used to identify the two encoding branches to ensure unique decoding. The above-described pre-processing can also be implemented in two parallel branches, as discussed later in Chapter 7.

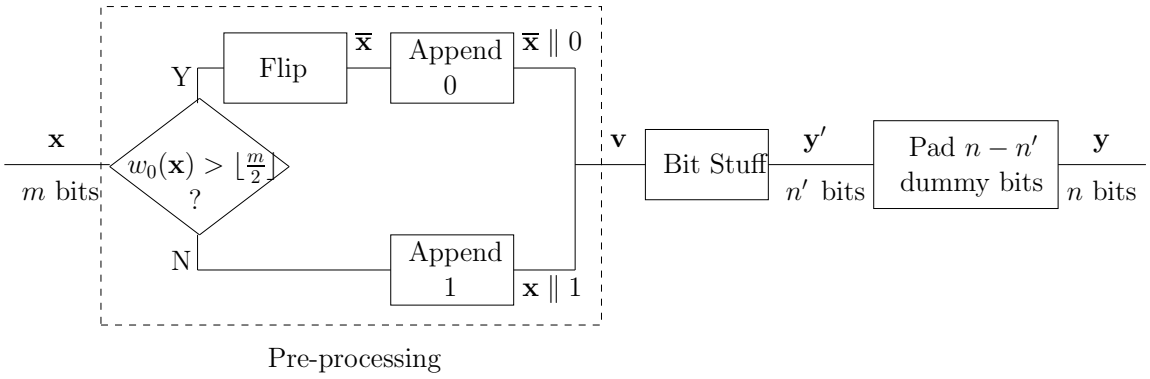

Figure 12: Block diagram of the $k = 1$ fixed-rate encoder. It accepts an m-bit input and generates an n-bit constrained output. Depending on its weight, the input is either flipped or retained as shown in the upper and lower branches, respectively. The effect of such pre-processing is to better conform the input sequence x to bit stuff encoding. This is responsible for rate improvements from $1/2$ up towards $2/3$.

The pre-processing output sequence $\mathbf{v}$ is hence of a fixed length $m+1$ bits, and has the property that the number of "0"s is at most $\lceil m/2 \rceil$. This is the key to improving rates beyond plain bit stuffing, where the encoding rate is constricted when all input bits are "0"s. However, a simple pre-processing has eliminated this, and several other undesirable input sequences. Essentially, the pre-processing has converted the input sequence $\mathbf{x}$ of length m into a sequence $\mathbf{v}$ of length $m + 1$, that is better conformed to bit stuffing. This leads to fewer bit insertions, and hence greater encoding rates.

The sequence $\mathbf{v}$ is then bit stuff encoded according to the variable-rate bit insertion rule in Table 14, to produce an output sequence $\mathcal{B}(\mathbf{v}) = \mathbf{y}'$. Here, we use $\mathcal{B}(.)$ to

denote the variable-rate bit stuff operation. Hence, the bit stuff output sequence $\mathbf{y}'$ is now of a variable length $l(\mathbf{y}') = n'$. Finally, the fixed-length constrained output, $\mathbf{y}$, is generated by padding sequence $\mathbf{y}'$ with $n - n'$ dummy bits, where

$$\begin{aligned} n &= \max_{\mathbf{x}\in\mathbb{Z}_2^m}\{n'\} \\ &= \max_{\mathbf{x}\in\mathbb{Z}_2^m}\{\min\{l(\mathcal{B}(\mathbf{x} \parallel 1)), l(\mathcal{B}(\overline{\mathbf{x}} \parallel 0))\}\} \\ &= \max_{\mathbf{x}\in\mathbb{Z}_2^m}\{\min\{m + w_0(\mathbf{x}) + 1, 2m - w_0(\mathbf{x}) + 2\}\}, \end{aligned} \tag{87}$$

and $w_0(\mathbf{x})$ denotes the number of "0"s in $\mathbf{x}$.

The inserted $n - n'$ dummy bits can be ignored during decoding. Several assignments are possible for this purpose. One possibility is to choose all the dummy bits as "1"s. Let us assume that the string of $n - n'$ "1"s is appended at the end of the sequence $\mathbf{y}'$ to form the length-n sequence $\mathbf{y}$ during encoding. Furthermore, let the recording channel be noise-free, or indeed if there are channel bit errors, then we assume that they are all corrected by a powerful error correction code (ECC) (see Fig. 1, Chapter 2). Thus, the stored bit sequence $\mathbf{y}$ is read-out error-free and fed into the constrained decoder input. The first decoding step is to remove the stuffed "1" bits that follow strings of k consecutive "0"s. This leaves a sequence of length $m + 1 + n - n'$ bits, the first $m + 1$ of which correspond to the pre-processing output $\mathbf{v}$. Since the decoder has prior knowledge of the value of m, the last $n - n'$ bits can be correctly ignored as dummy-bits. The sequence $\mathbf{x}$ can then be recovered from $\mathbf{v}$, by reading the index bit, and then using a simple inverse of the pre-processing.

The encoding rate of the above-described construction for the $(0, 1)$ constraint, with an input block length m, is simply given by $R(0, 1, m) = m/n$, where n is obtained from (87). A little thought leads us to the following rate expression

$$R(0,1,m) = \begin{cases} \frac{2m}{3m+2} & \text{when } m \text{ even} \\ \frac{2m}{3m+3} & \text{when } m \text{ odd.} \end{cases} \tag{88}$$

It follows that $\lim_{m\to\infty} R(0,1,m) = 2/3$. Hence, we see that the asymptotic (in input block length) encoding rate of the proposed fixed-rate construction is equal to the average rate of the variable-rate bit stuff code. Moreover, by encoding in input block lengths $m \geq 4$ bits, we can obtain encoding rates greater than the plain bit stuffing rate, which is limited to 1/2.

In the following section, we generalize the fixed-rate encoding to all values of $k < \infty$. As with $k = 1$ codes, the main idea is to include a suitable pre-processing component to maintain high encoding efficiency. However, unlike the simple $k = 1$ codes, the required pre-processing for values of $k \geq 2$ is more complex, and iterative in nature. The reason for this disparity lies in the nature of the bit stuff algorithm, which maps bits to variable-length phrases for the $k = 1$ constraint, but variable length input words to variable length phrases for all other values of $k < \infty$ (see Table 13).

7.2 The Fixed-Rate Bit Stuff (FRB) Algorithm

The algorithm accepts an unconstrained binary input sequence, $\mathbf{x} = (x_0 x_1 \ldots, x_{m-1})$, of fixed-length m bits and outputs a $(0,k)$ binary sequence, $\mathbf{y} = (y_0 y_1 \ldots, y_{n-1})$, of fixed-length n bits. The objective is to design a fixed-rate encoder for all possible m-bit inputs, such that the rate m/n, is close to the average rate $R_0(0,k) = (2^{k+1} - 2)/(2^{k+1} - 1)$, of the variable-rate bit stuff code.

Similar to the discussion in Chapter 7.1, the proposed encoding consists of three stages. This is illustrated in the block diagram in Fig. 13. The input sequence $\mathbf{x}$ first undergoes iterative pre-processing, followed by bit stuff encoding, and finally

dummy-bit padding to a fixed output length n. The pre-processing operations now involve k iterations, and are key to building efficient fixed-rate codes. The main idea here is to repeatedly scan and parse the input to identify certain undesirable (from bit stuffing point of view) patterns. They are subsequently eliminated/curtailed by a reversible, selective inversion. Essentially, the pre-processing transforms the input sequence $\mathbf{x}$ into a sequence $\mathbf{x}_k$, which is of the same length m, but is better conformed to bit stuffing. There is an inherent penalty for such a pre-processing in the form of additional index bits that need to be conveyed to the decoder for suitable post-processing. The entire sequence of these index bits is referred to as the *index sequence*, denoted by $\boldsymbol{\alpha}_{\mathbf{x}}$. The *pre-processing output* $\mathbf{v}$ shown in Fig. 13, is composed of the *pre-processed sequence* $\mathbf{x}_k$, appended with the index sequence $\boldsymbol{\alpha}_{\mathbf{x}}$. Thus, sequence $\mathbf{v}$ is longer than m bits, and there is a rate penalty in pre-processing. However, this rate penalty does not grow with input block length m. Hence, for large m, the cascade of pre-processing and bit stuffing can result in rate improvements over plain bit stuffing.

Figure 13: Block diagram of the fixed-rate bit stuff (FRB) encoder. It accepts an m-bit input and generates an n-bit constrained output. The key to achieving high encoding rates lies in the iterative pre-processing, which has k iterations. The effect of such a pre-processing is to better conform the input sequence to subsequent bit insertions. One can say that the pre-processing output v is "better prepared" for bit stuffing, as compared to the input sequence x.

Note that plain bit stuffing of the input sequence $\mathbf{x}$, followed by dummy-bit padding would yield $n = m + \lfloor m/k \rfloor$ for the worst-case "all-zeros" input. This pushes the rate toward $\frac{k}{k+1}$ for large m, still a fair distance away from the average rate $R_0(0, k)$ of the variable-rate bit stuff code. The proposed FRB algorithm thus provides a means of bridging this gap between fixed-rate, plain bit stuffing and

variable-rate bit stuffing. Before describing the algorithm, we introduce the following notations and definitions.

Notations and Definitions:

- Let

$$\mathbf{u}_i = 0^i 1, \quad i = 0, 1, \ldots, k-1 \tag{89}$$

$$\mathbf{u}_i^* = 0^{i+1}, \quad i = 0, 1, \ldots, k-1. \tag{90}$$

- The weight of a binary sequence $\mathbf{s}$ with respect to the input word $\mathbf{u}_i^*$, denoted by $w_{\mathbf{u}_i^*}(\mathbf{s})$, is the number of distinct occurrences of $\mathbf{u}_i^*$ in sequence $\mathbf{s}$, with $\mathbf{s}$ being scanned as a concatenation of words from the set $\{\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_i, \mathbf{u}_i^*\}$, $i = 0, 1, \ldots, k-1$. For example, if $\mathbf{s} = 100001010001$, $k \geq 2$ and $i = 1$, $w_{\mathbf{u}_1^*}(\mathbf{s}) = 3$.

- The weight of a binary sequence $\mathbf{s}$ with respect to the input word $\mathbf{u}_i$, denoted by $w_{\mathbf{u}_i}(\mathbf{s})$, is the number of distinct occurrences of $\mathbf{u}_i$ in sequence $\mathbf{s}$, with $\mathbf{s}$ being scanned as a concatenation of words from the set $\{\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_i, \mathbf{u}_i^*\}$, $i = 0, 1, \ldots, k-1$. For example, if $\mathbf{s} = 100001010001$, $k \geq 2$ and $i = 1$, $w_{\mathbf{u}_1}(\mathbf{s}) = 2$.

- Denote by $\overline{\mathbf{s}}(i)$, the sequence formed by converting all $\mathbf{u}_i$ words to $\mathbf{u}_i^*$ words, and all $\mathbf{u}_i^*$ words to $\mathbf{u}_i$ words in $\mathbf{s}$, with $\mathbf{s}$ being scanned as a concatenation of words from the set $\{\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_i, \mathbf{u}_i^*\}$, $i = 0, 1, \ldots, k-1$. The operation $\overline{\mathbf{s}}(i)$ is equivalent to flipping those bit positions in $\mathbf{s}$ that follow a string of i consecutive zeros, $i = 0, 1, \ldots, k-1$. The flipping operation refers to a "0" bit being changed to a "1" bit and vice versa. The special case of $i = 0$ means that every bit in $\mathbf{s}$ is flipped. For example, if $\mathbf{s} = 100001010001$, and $k \geq 2$, $\overline{\mathbf{s}}(1) = 101011000100$. Note that $\overline{\mathbf{s}}(i)$ is an invertible operation.

- Denote by $\mathbf{s}_1 \parallel \mathbf{s}_2$, the concatenation of two sequences $\mathbf{s}_1$ and $\mathbf{s}_2$.
- Denote by $\boldsymbol{\alpha}_{\mathbf{x}} = [\alpha_1\ \alpha_2 \ldots \alpha_k]$, the index sequence corresponding to input sequence $\mathbf{x}$, and by δ^t a string of t dummy bits. The index bits are used to convey to the decoder whether or not the flipping operation is performed in each of the k iterations. The dummy bits are used to pad the output up to length n bits, and can be ignored during decoding (refer to the discussion in Chapter 7.1).
- Denote by $\mathcal{B}(\mathbf{s})$, the bit stuff encoding of a sequence $\mathbf{s}$.
- Denote by $l(\mathbf{s})$, the length in bits of a binary sequence $\mathbf{s}$.

The encoding algorithm is described next.

The FRB Encoding Algorithm:

Input $\mathbf{x}$

Set $\mathbf{x}_0 = \mathbf{x}$

For $j = 1$ to k

  Input $\mathbf{x}_{j-1}$

  Scan $\mathbf{x}_{j-1}$ as a concatenation of words from the set $\left\{\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{j-1}, \mathbf{u}^*_{j-1}\right\}$

  If $w_{\mathbf{u}_{j-1}}(\mathbf{x}_{j-1}) < w_{\mathbf{u}^*_{j-1}}(\mathbf{x}_{j-1})$

    $\mathbf{x}_j = \overline{\mathbf{x}}_{j-1}(j-1)$

    $\alpha_j = 1$

  Else

    $\mathbf{x}_j = \mathbf{x}_{j-1}$

    $\alpha_j = 0$

  end

end

$n' = l(\mathcal{B}(\mathbf{x}_k \parallel \boldsymbol{\alpha}_{\mathbf{x}}))$

Output $\mathbf{y} = \mathcal{B}\left(\mathbf{x}_k \parallel \boldsymbol{\alpha}_{\mathbf{x}}\right)) \parallel \delta^{n-n'}$.

The "for" loop in the encoding algorithm performs the k pre-processing iterations, while the bit stuffing and dummy-bit padding operations are carried out in the last step. Since each of the encoding operations is reversible, the decoder functionality is a simple inverse of the encoding.

In the rest of this thesis, we use the following short-hand notation to denote the k pre-processing iterations.

$1)1 \geq 0$

$2)01 \geq 00$

$\vdots$

$k)0^{k-1}1 \geq 0^k$

The short-hand notation emphasizes that at the end of each iteration j, $j = 1, 2, \ldots, k$, the sequence $\mathbf{x}_j$ satisfies $w_{\mathbf{u}_{j-1}}(\mathbf{x}_j) \geq w_{\mathbf{u}^*_{j-1}}(\mathbf{x}_j)$, *i.e.*, $\mathbf{x}_j$ has at least as many $0^{j-1}1$ words as 0^j words, with $\mathbf{x}_j$ being scanned as a concatenation of words from the set $\left\{\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{j-1}, \mathbf{u}^*_{j-1}\right\}$.

7.2.1 An Encoding Example

We track the iterative fixed-rate encoding for $k = 9$. Consider the 50-bit input sequence

$$\mathbf{x} = 0^2 10^9 1^3 01^{11} 01^3 0^{10} 1^9.$$

Since the number of "1"s in $\mathbf{x}_0 = \mathbf{x}$ is greater than the number of "0"s, we have $w_{\mathbf{u}_0}(\mathbf{x}_0) > w_{\mathbf{u}^*_0}(\mathbf{x}_0)$, and hence $\mathbf{x}_1 = \mathbf{x}_0$ after the first pre-processing iteration, *i.e.*, $j = 1$. Next, when $j = 2$, we scan $\mathbf{x}_1$ as a concatenation of words from the set $\{\mathbf{u}_0 = 1, \mathbf{u}_1 = 01, \mathbf{u}^*_1 = 00\}$, to find that $w_{\mathbf{u}_1}(\mathbf{x}_1) = 3 < 10 = w_{\mathbf{u}^*_1}(\mathbf{x}_1)$. Hence, $\mathbf{x}_2 = \overline{\mathbf{x}}_1(1) = 01^2(01)^4 001^2 001^{10} 001^2 (01)^5 1^9$. It so happens in this example that $\mathbf{x}_2$ only has runs of consecutive zeros of length at most 2. Hence, we find at the end of $k = 9$ iterations that $\mathbf{x}_9 = \mathbf{x}_2$.

In this example, we have performed the flipping operation, *i.e.*, $\mathbf{x}_j = \overline{\mathbf{x}}_{j-1}(j-1)$, only in iteration 2, and retained the sequence, *i.e.*, $\mathbf{x}_j = \mathbf{x}_{j-1}$, in iteration 1 and iterations 3 through 9. For our index assignment, let us use a "0" bit to indicate flipping and a "1" bit otherwise. The index sequence $\boldsymbol{\alpha}_{\mathbf{x}}$ is then a concatenation of index bits for the 9 iterations[1]. Hence, for the input sequence in our example, we have, $\boldsymbol{\alpha}_{\mathbf{x}} = 101111111$. Bit stuff encoding of $\mathbf{x}_k \parallel \boldsymbol{\alpha}_{\mathbf{x}}$ does not incur any additional bits. Assuming that the output block length n is 65 bits[2], we can compute that 6 dummy bits will be used for padding. Hence, the output $\mathbf{y}$ of the algorithm can now be given as $\mathbf{y} = \mathbf{x}_9 \parallel \boldsymbol{\alpha}_{\mathbf{x}} \parallel \delta^6$. Another encoding example can be found in [51].

7.3 Rate Computation

Having described the FRB algorithm, we now proceed to compute its rate. The encoding rate, $R(0, k, m)$, for $d = 0$, maximum-run-length parameter k, and input block length m, is $R(0, k, m) = m/n$, where the output block length n is given by

$$n = \max_{\mathbf{x} \in \mathbb{Z}_2^m} l(\mathcal{B}(\mathbf{x}_k)) + k + 1. \tag{91}$$

Note that $k+1$ is the maximum length of the portion of the codeword corresponding to the index sequence $\boldsymbol{\alpha}_{\mathbf{x}}$. Recall that the index bits are used to convey to the post-processor, whether or not the flipping operation is performed in each of the k iterations, *i.e.*, whether $\mathbf{x}_i = \overline{\mathbf{x}}_{i-1}(i-1)$ or $\mathbf{x}_i = \mathbf{x}_{i-1}$, for $i = 1, 2, \ldots, k$. Hence, the index sequence $\boldsymbol{\alpha}_{\mathbf{x}}$ is of length k bits, and can be involved in at most one additional bit insertion. Thus, the output block length n, as computed in (91) is in fact an upper bound on the maximum possible output length for any input sequence of length m

[1] Alternatively, any other suitable index assignment may be chosen.

[2] The choice of numbers m and n in this example is purely for illustration purposes. Typically, we expect m and n to be a few thousand bits each.

bits. Indeed, it can be verified that

$$\begin{aligned} n &= \max_{\mathbf{x}\in\mathbb{Z}_2^m} l(\mathcal{B}(\mathbf{x}_k)) + k + 1 \\ &\geq \max_{\mathbf{x}\in\mathbb{Z}_2^m} l(\mathcal{B}(\mathbf{x}_k \parallel \boldsymbol{\alpha}_{\mathbf{x}})) \end{aligned} \tag{92}$$

We now proceed to compute the term $\max_{\mathbf{x}\in\mathbb{Z}_2^m} l(\mathcal{B}(\mathbf{x}_k))$ in (91). This is nothing but the maximum length of the bit stuff encoding of sequence $\mathbf{x}_k$, which in turn is derived from iterative pre-processing of the input $\mathbf{x}$. As seen earlier in Table 13, the bit stuff algorithm inserts a "1" after every string of k consecutive "0"s in $\mathbf{x}_k$. Hence, we have

$$\max_{\mathbf{x}\in\mathbb{Z}_2^m} l(\mathcal{B}(\mathbf{x}_k)) = m + \max_{\mathbf{x}\in\mathbb{Z}_2^m} w_{\mathbf{u}_{k-1}^*}(\mathbf{x}_k). \tag{93}$$

Using (93) and (91), we find that the encoding rate $R(0,k,m)$ is given by

$$R(0,k,m) = \frac{m}{m + \max_{\mathbf{x}\in\mathbb{Z}_2^m} w_{\mathbf{u}_{k-1}^*}(\mathbf{x}_k) + k + 1}. \tag{94}$$

Our main interest in this section will be to evaluate $R(0,k,m)$ for very long input blocks, *i.e.*, as $m \to \infty$. Rate computation for finite input blocks is discussed later in Chapter 8.1.

Let us now define

$$\beta(k,m) = \max_{\mathbf{x}\in\mathbb{Z}_2^m} w_{\mathbf{u}_{k-1}^*}(\mathbf{x}_k). \tag{95}$$

For any given $k < \infty$, the function $\beta(k, m)$ has the following properties (see Appendix A)

1. $\beta(k, m)$ is non-decreasing in m.

2. $\beta(k, m)$ is unbounded, that is $\beta(k, m) \to \infty$ as $m \to \infty$.

Let m_0 be the smallest positive integer such that $\beta(k, m) > 0$. Equation (94) can now be re-written for all $m \geq m_0$ as

$$R(0, k, m) = \frac{1}{1 + \frac{\beta(k,m)}{m} + \frac{k+1}{m}}. \tag{96}$$

The term $\frac{k+1}{m}$ vanishes as $m \to \infty$. Thus, the asymptotics of $R(0, k, m)$ are determined by the asymptotics of $\frac{\beta(k,m)}{m}$. This relationship is quite intuitive, since $\frac{\beta(k,m)}{m}$ is nothing but the maximum number of bit insertions per input bit in the FRB algorithm, for a given input block length m.

To facilitate further analysis, we define the following function

$$\theta(k, m) = \frac{m}{\beta(k, m)} \tag{97}$$

Note that $\theta(k, m) = \left(\frac{\beta(k,m)}{m}\right)^{-1}$. A direct analysis of $\theta(k, m)$ involves searching over the entire set of 2^m input sequences, $\mathbf{x} \in \mathbb{Z}_2^m$ (see equation (95)). Clearly, such an analysis is intractable for large m. This prompts us to take the following, alternative approach.

We make the assumption that $\theta(k, m)$ converges as $m \to \infty$. This assumption is supported by a discussion on the properties of $\theta(k, m)$ in Appendix A, which leads us to the following result

$$\lim_{m\to\infty} \theta(k,m) = \inf_{w_{\mathbf{u}^*_{k-1}}(\mathbf{x}_k)\in\mathbb{Z}^+} \frac{l(\mathbf{x})}{w_{\mathbf{u}^*_{k-1}}(\mathbf{x}_k)}, \tag{98}$$

where $\mathbb{Z}^+$ denotes the set of all positive integers, and $l(\mathbf{x})$ denotes the length of input sequence $\mathbf{x}$ (not fixed at m anymore). Using (98) in (96), we find that

$$\lim_{m\to\infty} R(0,k,m) \;\; = \;\; \frac{\eta(k)}{\eta(k)+1},$$

where

$$\eta(k) = \inf_{w_{\mathbf{u}^*_{k-1}}(\mathbf{x}_k)\in\mathbb{Z}^+} \frac{l(\mathbf{x})}{w_{\mathbf{u}^*_{k-1}}(\mathbf{x}_k)}. \tag{99}$$

Hence, the problem of computing the asymptotic rate has now been reduced to finding the infimum of the ratio $l(\mathbf{x})/w_{\mathbf{u}^*_{k-1}}(\mathbf{x}_k)$, over all non-zero values of the weight $w_{\mathbf{u}^*_{k-1}}(\mathbf{x}_k)$.

The result in (98) is fairly intuitive, with both expressions formulating the minimum number of input bits per bit insertion, *i.e.*, per 0^k word in the pre-processed sequence $\mathbf{x}_k$. However, there is a significant difference in the search spaces of (95) and (98). Rather than search over the potentially intractable space of all input sequences $\mathbf{x} \in \mathbb{Z}_2^m$ in (95), our search-space in (98) is the set of all non-zero values of the weight $w_{\mathbf{u}^*_{k-1}}(\mathbf{x}_k)$. Noticeably in (98), we now search by starting with the pre-processed vectors $\mathbf{x}_k$ (rather than the input sequence $\mathbf{x}$) and *backtrack* through the iterative pre-processing to determine the input length $l(\mathbf{x})$. This means that we now start at the output of the pre-processing block of Fig. 13, and work our way backwards through the k pre-processing iterations, ending up at the input. Such a backtracking

over the pre-processing iterations, *i.e.,* moving from $\mathbf{x}_k$ towards $\mathbf{x}$, is more conducive to analysis, and is an important theme in the rest of our discussions.

This point is further illustrated with the help of the binary search-tree in Fig. 14. Here the root node denotes the input sequence $\mathbf{x}$ and the 2^k leaf nodes represent the pre-processed sequences $\mathbf{x}_k$, each corresponding to a possible path through the iterative pre-processing. The result of every iteration, $i = 1, 2, \ldots, k$, is denoted on the branches by either $\mathbf{F}$ or $\overline{\mathbf{F}}$, to indicate $\mathbf{x}_i = \overline{\mathbf{x}}_{i-1}(i-1)$ or $\mathbf{x}_i = \mathbf{x}_{i-1}$, respectively. Indeed, for a given input sequence $\mathbf{x}$, only one of these 2^k paths is traversed by the iterative pre-processing.

However, for our search of $\eta(k)$ as in (99), we need to determine $\inf_{w_{\mathbf{u}^*_{k-1}}(\mathbf{x}_k)\in\mathbb{Z}^+} \frac{l(\mathbf{x})}{w_{\mathbf{u}^*_{k-1}}(\mathbf{x}_k)}$ by backtracking through the binary search-tree along each of the 2^k paths. In other words, we work backwards starting from a leaf node up towards the root, and determine the infimum amongst all inputs that traverse the corresponding path from the root to the leaf node; and repeat this process for all 2^k leaf nodes to determine the global infimum. Such an approach eliminates the exhaustive search over all possible input sequences as required in (95). However, it could still be prohibitively complex for large k. This is because we need to backtrack over 2^k paths, and this number grows exponentially with k. Thus, we have been unable to proceed with the exact computation of $\eta(k)$, except for a few select cases of small k, $k = 1, 2, 3, 4$ (see Appendix A). In what follows, we derive upper and lower bounds on $\eta(k)$.

7.3.1 Upper Bound on Asymptotic Encoding Rate

If we limit our search for $\inf_{w_{\mathbf{u}^*_{k-1}}(\mathbf{x}_k)\in\mathbb{Z}^+} \frac{l(\mathbf{x})}{w_{\mathbf{u}^*_{k-1}}(\mathbf{x}_k)}$ to a single path in the binary search-tree rather than all the 2^k paths, then we obtain an upper bound on $\eta(k)$ in (99), and hence an upper bound on the asymptotic encoding rate.

The path we have chosen[3] for our analysis here is the path marked $\mathcal{P}$ in Fig. 14.

[3]In fact, any tractable subset of the 2^k paths may be chosen to derive the upper bound. We have

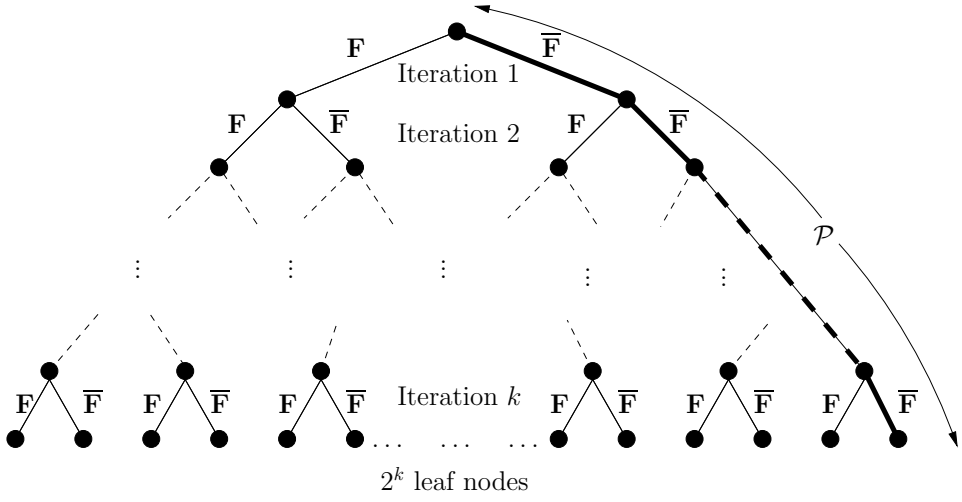

Figure 14: Binary search-tree to determine $\inf_{w_{\mathbf{u}^*_{k-1}}(\mathbf{x}_k)\in\mathbb{Z}^+} \frac{l(\mathbf{x})}{w_{\mathbf{u}^*_{k-1}}(\mathbf{x}_k)}$**. The path** $\mathcal{P}$ **marked in bold is the one traversed to determine an upper bound on the asymptotic rate.**

This corresponds to retaining the input sequence at each iteration (equivalently, the result of each iteration is $\overline{\mathbf{F}}$), whereby $\mathbf{x}_k = \mathbf{x}$.

Thus, in order to determine the upper bound, we start with the rightmost leaf-node in Fig. 14 (which represents $\mathbf{x}_k$ for path $\mathcal{P}$) and work our way upward along path $\mathcal{P}$ towards the root (which represents $\mathbf{x}$). Essentially, for each $w_{\mathbf{u}^*_{k-1}}(\mathbf{x}_k) \in \mathbb{Z}^+$, we now need to backtrack along path $\mathcal{P}$ to determine the minimum corresponding $l(\mathbf{x})$, according to (98). While backtracking from the leaf-node to the root, the intermediate nodes represent sequences $\mathbf{x}_{k-1}, \mathbf{x}_{k-2}, \ldots, \mathbf{x}_2, \mathbf{x}_1$, in that order. The following definition connects properties of the sequence $\mathbf{x}_1$ to sequence $\mathbf{x}_k$. Unless otherwise stated, all subsequent discussions on the upper bound are with respect to the path $\mathcal{P}$ in Fig. 14.

Definition 7.1 *For given $k < \infty$ and path $\mathcal{P}$ shown in Fig. 14, the* effective weight*, a_i^k, of the word 0^i1, $i \in \mathbb{Z}^+$, is the minimum number of zeros in $\mathbf{x}_1$ resulting from the presence of a single 0^i1 word in the pre-processed sequence $\mathbf{x}_k$, with $\mathbf{x}_k$ being scanned*

chosen path $\mathcal{P}$ here, as it leads to simpler computations and reasonably tight bounds.

as a concatenation of words $0^{t-1}1$, $t \in \mathbb{Z}^+$.

The effective weight is a measure of the reverse cascading, from $\mathbf{x}_k$ through to $\mathbf{x}_1$ in the iterative pre-processing. As we will see shortly, it relates to the minimum number of bits, $l(\mathbf{x}) = 2a_i^k$, necessary at the input to produce a single 0^i1 word in the sequence $\mathbf{x}_k$. Clearly, $a_1^k = 1$ for all k. By working backwards through the binary search-tree along path $\mathcal{P}$, we obtain the following result.

Lemma 7.2 *The effective weights,* a_i^k, $i \geq 2$, *are related by the recursion*

$$\begin{aligned} a_i^k &= \left(\sum_{j=1}^{k-1} \left[w_{\mathbf{u}_j^*}(0^i1) - w_{\mathbf{u}_j}(0^i1) \right] a_j^k \right) + w_{\mathbf{u}_0^*}(0^i1) \\ &= \left(\sum_{j=1}^{k-1} q(i,j) a_j^k \right) + i, \end{aligned} \tag{100}$$

where the function $q(i,j)$ *for* $i, j \geq 1$ *is given by*

$$q(i,j) = \begin{cases} \max\left\{0, \frac{i+1}{j+1} - 2\right\} & \text{when } \frac{i+1}{j+1} \in \mathbb{Z}^+ \\ \lfloor \frac{i+1}{j+1} \rfloor & \text{when } \frac{i+1}{j+1} \notin \mathbb{Z}^+. \end{cases} \tag{101}$$

Proof: We backtrack through the binary search-tree along path $\mathcal{P}$ shown in Fig. 14. We start at the rightmost leaf-node and move up towards the root, one iteration at a time. Let us assume that the pre-processed sequence $\mathbf{x}_k$ is scanned as a concatenation of words $0^{t-1}1$, $t \in \mathbb{Z}^+$. Let this yield a 0^i1 word, $i \geq 2$. By virtue of the pre-processing iterations, we deduce that any 0^i1 word, $i \geq 2$ cannot "stand alone" in $\mathbf{x}_k$, and must be accompanied by other words. This follows from the fact that at the end of each iteration j, $j = 1, 2, \ldots, k$, the sequence $\mathbf{x}_j$ satisfies

$w_{\mathbf{u}_{j-1}}(\mathbf{x}_j) \geq w_{\mathbf{u}^*_{j-1}}(\mathbf{x}_j)$, *i.e.*, $\mathbf{x}_j$ has at least as many $0^{j-1}1$ words as 0^j words, with $\mathbf{x}_j$ being scanned as a concatenation of words from the set $\left\{\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{j-1}, \mathbf{u}^*_{j-1}\right\}$. For instance, if $k = 2$, then a 0^21 word cannot stand alone in $\mathbf{x}_2$, and must be accompanied by at least one 01 word; similarly, a 0^41 word in $\mathbf{x}_2$ must be accompanied by at least two 01 words. On the other hand, a 1 or a 01 word can stand alone in $\mathbf{x}_2$.

Our aim now is to track the minimum number of such accompanied words for a given 0^i1 word, $i \geq 2$, in $\mathbf{x}_k$. This will eventually help us determine a_i^k, the minimum number of zeros in $\mathbf{x}_1$. Thus, we backtrack along path $\mathcal{P}$ in the binary search-tree: starting from the rightmost leaf-node $\mathbf{x}_k$, moving one pre-processing iteration at a time, until we reach $\mathbf{x}_1$. It is important to note here that we are able to track the minimum number of accompanying words all the way up to $\mathbf{x}_1$ only because of the favorable nature of the chosen path $\mathcal{P}$, which ensures that

$$\mathbf{x}_k = \mathbf{x}_{k-1} = \mathbf{x}_{k-2} = \ldots = \mathbf{x}_2 = \mathbf{x}_1.$$

Consider a 0^i1 word, $i \geq 2$ in $\mathbf{x}_k$. Iteration k ensures that $w_{\mathbf{u}_{k-1}}(\mathbf{x}_k) \geq w_{\mathbf{u}^*_{k-1}}(\mathbf{x}_k)$, *i.e.*, $\mathbf{x}_k$ has at least as many $0^{k-1}1$ words as 0^k words, with $\mathbf{x}_k$ being scanned as a concatenation of words from the set $\left\{\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{k-1}, \mathbf{u}^*_{k-1}\right\}$. Hence, the minimum number of accompanying $0^{k-1}1$ words present in sequence $\mathbf{x}_k$ is given by $\left[w_{\mathbf{u}^*_{k-1}}(0^i1) - w_{\mathbf{u}_{k-1}}(0^i1)\right]$. Furthermore, by virtue of path $\mathcal{P}$, we have that $\mathbf{x}_{k-1} = \mathbf{x}_k$. Thus, we infer that the sequence $\mathbf{x}_{k-1}$ contains at least $\left[w_{\mathbf{u}^*_{k-1}}(0^i1) - w_{\mathbf{u}_{k-1}}(0^i1)\right]$ $0^{k-1}1$ words in addition to the 0^i1 word.

By definition, each of the $\left[w_{\mathbf{u}^*_{k-1}}(0^i1) - w_{\mathbf{u}_{k-1}}(0^i1)\right]$ $0^{k-1}1$ words in $\mathbf{x}_{k-1}$ results in at least a_{k-1}^{k-1} zeros in sequence $\mathbf{x}_1$. Note that the subscript $k-1$, in a_{k-1}^{k-1} indicates the length of the string of zeros in the word $0^{k-1}1$, while the superscript $k-1$, points to the number of remaining iterations to backtrack. From the definition of $q(i,j)$ in (101), we see that $a_t^t = a_t^k$ for all $t \leq k$. Hence, we can re-write a_{k-1}^{k-1} as a_{k-1}^k for

convenience.

We are now left with the 0^i1 word in $\mathbf{x}_{k-1}$. We repeat the above arguments for iterations $k-1$ through to 2. This yields the minimum number of zeros in $\mathbf{x}_1$ as the sum $\sum_{j=1}^{k-1}\left(\left[w_{\mathbf{u}_j^*}(0^i1) - w_{\mathbf{u}_j}(0^i1)\right]a_j^k\right) + w_{\mathbf{u}_0^*}(0^i1)$, where the second term is simply the contribution of i zeros from the 0^i1 word. Thus, the effective weight a_i^k is given by $\sum_{j=1}^{k-1}\left(\left[w_{\mathbf{u}_j^*}(0^i1) - w_{\mathbf{u}_j}(0^i1)\right]a_j^k\right) + i$. ■

We now return to the computation of the upper bound. Definition 7.1 makes a connection between properties of the pre-processed sequence $\mathbf{x}_k$ and the number of zeros in the sequence $\mathbf{x}_1$. From pre-processing iteration 1, we have that $w_{\mathbf{u}_0}(\mathbf{x}_1) \geq w_{\mathbf{u}_0^*}(\mathbf{x}_1)$, *i.e.*, the number of ones in $\mathbf{x_1}$ is at least equal to the number of zeros in $\mathbf{x_1}$. Thus, we find that the input length $l(\mathbf{x})$ must be at least $2a_i^k$ bits in order to produce a single 0^i1 word in $\mathbf{x}_k$.

From (99) and Definition 7.1, we now have

$$\eta(k) = \inf_{w_{\mathbf{u}_{k-1}^*}(\mathbf{x}_k)\in\mathbb{Z}^+} \frac{l(\mathbf{x})}{w_{\mathbf{u}_{k-1}^*}(\mathbf{x}_k)} \leq \inf_{i\geq k} \frac{2a_i^k}{\lfloor \frac{i}{k} \rfloor} \tag{102}$$

The inequality in (102) arises from the premise for the upper bound: there might be paths other than $\mathcal{P}$ in Fig. 14, that could yield a lower value of $\frac{l(\mathbf{x})}{w_{\mathbf{u}_{k-1}^*}(\mathbf{x}_k)}$. The denominator in (102) follows from the simple relation $w_{\mathbf{u}_{k-1}^*}(0^i1) = \lfloor \frac{i}{k} \rfloor$. This relation also points out that we only need to consider the search space $i \geq k$, so that $w_{\mathbf{u}_{k-1}^*}(\mathbf{x}_k) \in \mathbb{Z}^+$, as required in (99).

For further analysis, we will find it convenient to denote $f_i(k) = \frac{2a_i^k}{\lfloor \frac{i}{k} \rfloor}$. Our goal now is to compute for a given k, the infimum of $f_i(k)$ over all $i \geq k$. However, because of the nature of the expression for $f_i(k)$, we have been unable to obtain a closed form solution. Instead, we use the following result to limit our search space.

Theorem 7.3 *For any given k, $1 \leq k < \infty$,*

$$\inf_{i \geq k} f_i(k) \quad = \quad \min_{k \leq i \leq k+b(k)-1} f_i(k), \tag{103}$$

where $b(k)$ denotes the LCM of positive integers upto and including k, i.e.*, $b(k) = LCM(1, 2, \ldots, k)$.*

Proof: Theorem 7.3 states that the required infimum of $f_i(k)$ over all positive integers $i \geq k$, can be obtained by searching over $b(k)$ values of i that lie within $[k, k + b(k) - 1]$. The proof proceeds by exploring certain monotonicity properties of $f_i(k)$.

Consider the sequence $\{f_i(k)\}_{i \geq k}$. Clearly, this sequence by itself is not monotonic in i (see (101) and (100)). However, $\{f_i(k)\}_{i \geq k}$ can be broken up into subsequences

$$\{f_{k+tb(k)}(k)\}_{t \geq 0}, \{f_{k+1+tb(k)}(k)\}_{t \geq 0}, \{f_{k+2+tb(k)}(k)\}_{t \geq 0}, \ldots, \{f_{k+(b(k)-1)+tb(k)}(k)\}_{t \geq 0},$$

each of which is either monotonic in t, or is constant. This can be proved as follows.

Consider any subsequence $\{f_{c+tb(k)}(k)\}_{t \geq 0}$, $k \leq c \leq k + b(k) - 1$. $f_{c+tb(k)}(k)$ can be expanded out as

$$f_{c+tb(k)}(k) \quad = \quad \frac{2a^k_{c+tb(k)}}{tb_k(k) + \lfloor \frac{c}{k} \rfloor}, \tag{104}$$

where $b_k(k) = \frac{b(k)}{k}$. Now, consider the sequence, $\left\{\mathsf{Den}(f_{c+tb(k)}(k))\right\}_{t \geq 0} = \left\{tb_k(k) + \lfloor \frac{c}{k} \rfloor\right\}_{t \geq 0}$, of the denominators of $f_{c+tb(k)}(k)$ in (104). Clearly, $\left\{\mathsf{Den}(f_{c+tb(k)}(k))\right\}_{t \geq 0}$ is an arithmetic progression with initial term $\lfloor \frac{c}{k} \rfloor$ and common difference $b_k(k)$.

The corresponding numerator sequence is $\left\{\mathsf{Num}(f_{c+tb(k)}(k))\right\}_{t\geq 0} = \left\{2a^k_{c+tb(k)}\right\}_{t\geq 0}$. We now show that this forms an arithmetic progression as well. Consider the difference $a^k_{c+(t+1)b(k)} - a^k_{c+tb(k)}$, for any $t \geq 0$. From (100), we have

$$\begin{aligned} a^k_{c+(t+1)b(k)} - a^k_{c+tb(k)} &= \left(\sum_{j=1}^{k-1} q(c+(t+1)b(k),j)a^k_j\right) - \\ &\quad \left(\sum_{j=1}^{k-1} q(c+tb(k),j)a^k_j\right) + b(k) \qquad (105) \\ &\qquad (106) \\ &= \left(\sum_{j=1}^{k-1} b_{j+1}(k)a^k_j\right) + b(k), \qquad (107) \end{aligned}$$

where $b_{j+1}(k) = \frac{b(k)}{j+1}$, and (107) follows from (101). From (107), we note that $a^k_{c+(t+1)b(k)} - a^k_{c+tb(k)}$ is independent of both c and t. Hence, we infer that the numerator sequence $\{2a^k_{c+tb(k)}\}_{t\geq 0}$ is an arithmetic progression with initial term $2a^k_c$ and common difference $2\left(\left(\sum_{j=1}^{k-1} b_{j+1}(k)a^k_j\right) + b(k)\right)$.

The subsequence $\{f_{c+tb(k)}(k)\}_{t\geq 0}$ is then the term-by-term ratio of two arithmetic progressions, each of which has initial term and common difference as positive integers. As a consequence, we have that $\{f_{c+tb(k)}(k)\}_{t\geq 0}$ is monotonically increasing if $\frac{a^k_c}{\left\lfloor\frac{c}{k}\right\rfloor} < \frac{\left(\sum_{j=1}^{k-1} b_{j+1}(k)a^k_j\right)+b(k)}{b_k(k)}$, monotonically decreasing if $\frac{a^k_c}{\left\lfloor\frac{c}{k}\right\rfloor} > \frac{\left(\sum_{j=1}^{k-1} b_{j+1}(k)a^k_j\right)+b(k)}{b_k(k)}$ and a constant if $\frac{a^k_c}{\left\lfloor\frac{c}{k}\right\rfloor} = \frac{\left(\sum_{j=1}^{k-1} b_{j+1}(k)a^k_j\right)+b(k)}{b_k(k)}$.

Thus far, we have shown that every subsequence $\{f_{c+tb(k)}(k)\}_{t\geq 0}$, $k \leq c \leq k + b(k) - 1$, is either monotonic or a constant. Since we are interested in the determining the minimum of $f_i(k)$, we now consider the minimum term in each of the subsequences. Those subsequences that are monotonically increasing (constant) have their minimum (constant term) equal to $f_c(k)$, which occurs at $t = 0$. Since $k \leq c \leq k + b(k) - 1$, these are already included in the search space in (103). Hence, we are left with the monotonically decreasing subsequences. The minimum term in this case is given by

$\lim_{t\to\infty} f_{c+tb(k)}(k)$, which can be evaluated as $2\frac{\left(\sum_{j=1}^{k-1} b_{j+1}(k)a_j^k\right)+b(k)}{b_k(k)}$. Note that this expression is independent of c. To complete the proof, we now show that $f_k(k) \leq \lim_{t\to\infty} f_{c+tb(k)}(k)$. We have

$$\begin{aligned}
\frac{f_k(k)}{2} &= a_k^k && (108)\\
&= \left(\sum_{j=1}^{k-1} q(k,j)a_j^k\right) + k && (109)\\
&= \frac{\left(\sum_{j=1}^{k-1} b_k(k)q(k,j)a_j^k\right) + b(k)}{b_k(k)} && (110)\\
&\leq \frac{\left(\sum_{j=1}^{k-1} b_{j+1}(k)a_j^k\right) + b(k)}{b_k(k)} && (111)\\
&= \frac{1}{2}\lim_{t\to\infty} f_{c+tb(k)}(k), && (112)
\end{aligned}$$

where (108) and (109) follow directly from definition, (110) is obtained by multiplying and dividing (109) by $b_k(k)$, and (111) results from the following fact

$$b_k(k)q(k,j) \leq \begin{cases} \frac{b(k)}{k}\left(\frac{k+1}{j+1} - 2\right) & \text{when } \frac{k+1}{j+1} \in \mathbb{Z}^+ \\ \frac{b(k)}{j+1} & \text{when } \frac{k+1}{j+1} \notin \mathbb{Z}^+ \end{cases}$$

To summarize, we have shown that the subsequences $\{f_{c+tb(k)}(k)\}_{t\geq 0}$, $k \leq c \leq k + b(k) - 1$ are all either monotonic or constant, and further that the monotonically decreasing subsequences are lower bounded by $f_k(k)$. These together imply that the search space $i \in [k, k + b(k) - 1]$ is sufficient to locate the minimum $f_i(k)$. Hence, we conclude that $\inf_{i\geq k}\{f_i(k)\} = \min_{k\leq i\leq k+b(k)-1}\{f_i(k)\}$. ■

Theorem 7.3 reduces our search space from the set of all positive integers $i \geq k$ to the finite set of integers $k \leq i \leq k+b(k)-1$. Let us denote $f(k) = \min_{k\leq i\leq k+b(k)-1} f_i(k)$ and $i_k = \arg\min_{k\leq i\leq k+b(k)-1} f_i(k)$, *i.e.*, i_k is the corresponding minimizing value of

integer i. Note that there is no condition on i_k being unique. In cases where several values of i all yield $f(k)$, we pick i_k to be the smallest of all such arguments.

By means of computer search within $[k, k+b(k)-1]$, we have determined $f(k)$ and i_k, for values of k up to 15. They are listed in Table 15 along with the corresponding asymptotic rate upper bound values, $R_u(0,k)$, which can be computed from (99) as

$$R_u(0,k) \;=\; \frac{f(k)}{f(k)+1} \geq \lim_{m\to\infty} R(0,k,m) = \frac{\eta(k)}{\eta(k)+1}. \tag{113}$$

It is seen that $R_u(0,k)$ values are only slightly lesser than corresponding average bit stuff rates, $R_0(0,k)$.

7.3.2 Lower bound on Asymptotic Encoding Rate

Next, we compute a lower bound on $\eta(k) = \inf_{w_{\mathbf{u}^*_{k-1}}(\mathbf{x}_k)\in\mathbb{Z}^+} \frac{l(\mathbf{x})}{w_{\mathbf{u}^*_{k-1}}(\mathbf{x}_k)}$ as in (99). For small values of k, $k = 1, 2, 3, 4$, the computed lower bounds are seen to be equal to the upper bounds obtained in Chapter 7.3.1. Thus, we are able to determine the exact asymptotic rate of the FRB algorithm for these values of k. In the special cases of $k = 1, 2$, these bounds are also seen to be equal to the average rate of the variable-rate bit stuff algorithm.

First, let us consider the case when $k = 1$. It can be verified that $\inf_{w_{\mathbf{u}_1}(\mathbf{x}_1)\in\mathbb{Z}^+} \frac{l(\mathbf{x})}{w_{\mathbf{u}_1}(\mathbf{x}_1)} = 2$. This follows from the pre-processing, which ensures that every "0" bit in $\mathbf{x}_1$ has at least one corresponding "1" bit in $\mathbf{x}_1$. Hence, $\eta(1) = 2$, and the exact asymptotic rate $R(0,1) = \lim_{m\to\infty} R(0,1,m) = \frac{2}{3}$, as seen earlier in Chapter 7.1.

Next, with $k = 2$, we have two pre-processing iterations and hence $2^k = 4$ leaf nodes in Fig. 14. The upper bound obtained in Chapter 7.3.1 traverses path $\mathcal{P}$, thereby leaving us with three more paths to trace. Backtracking on each of these three other paths, we find that the sequence $\mathbf{x}_1$ in each case contains at least three

Table 15: Summary of rate computations for the FRB algorithm

k	$f(k)$	i_k	**Upper bound** $R_u(0,k)$	**Lower bound** $R_l(0,k)$	**Bit stuff average rate** $R_0(0,k)$	$\frac{R_l(k)}{R_0(k)}$ (%)	$\frac{R_l(k)}{C(k)}$ (%)
1	2	1	2/3	2/3	2/3	100	96.02
2	6	2	6/7	6/7	6/7	100	97.49
3	12	3	12/13	12/13	14/15	98.90	97.49
4	24	5	24/25	24/25	30/31	99.20	98.43
5	54	5	54/55	48/49	62/63	99.54	99.13
6	114	7	114/115	96/97	126/127	99.75	99.54
7	240	7	240/241	192/193	254/255	99.87	99.76
8	480	11	480/481	384/385	510/511	99.93	99.88
9	984	11	984/985	768/769	1022/1023	99.96	99.94
10	1974	11	1974/1975	1536/1537	2046/2047	99.98	99.97
11	4020	11	4020/4021	3072/3073	4094/4095	99.99	99.98
12	8052	17	8052/8053	6144/6145	8190/8191	99.99	99.99
13	16242	17	16242/16243	12288/12289	16382/16383	99.99	99.99
14	32496	17	32496/32497	24576/24577	32766/32767	99.99	99.99
15	65226	17	65226/65227	49152/49153	65534/65535	99.99	99.99

"0" bits for every 0^2 word in $\mathbf{x}_2$ (with $\mathbf{x}_2$ being scanned as a concatenation of words $\{1, 01, 0^2\}$). From the previous argument on $k = 1$, we infer that there are also at least three "1" bits in $\mathbf{x}_1$, and hence $\inf_{w_{\mathbf{u}_2}(\mathbf{x}_2)\in\mathbb{Z}^+} \frac{l(\mathbf{x})}{w_{\mathbf{u}_2}(\mathbf{x}_2)} = 6$. This means that the exact asymptotic rate $R(0,2) = \lim_{m\to\infty} R(2,m) = \frac{6}{7}$.

With $k = 3$, such a computation involves backtracking along 8 possible paths and becomes rather tedious. Instead, let us look at iteration 3 alone. There are two possibilities: either $\mathbf{x}_3 = \overline{\mathbf{x}}_2(2)$ or $\mathbf{x}_3 = \mathbf{x}_2$, denoted by the branches $\overline{\mathbf{F}}$ and $\mathbf{F}$, respectively, in Fig. 14. Regardless of which branch we take in iteration 3, the resulting sequence $\mathbf{x}_2$ contains at least two 0^2 words (with $\mathbf{x}_2$ being scanned as a concatenation of words $\{1, 01, 0^2\}$) for every 0^3 word in $\mathbf{x}_3$ (with $\mathbf{x}_3$ being scanned as a concatenation of words $\{1, 01, 0^21, 0^3\}$). Thus, using our earlier result for $k = 2$, we conclude that $\inf_{w_{\mathbf{u}_3}(\mathbf{x}_3)\in\mathbb{Z}^+} \frac{l(\mathbf{x})}{w_{\mathbf{u}_3}(\mathbf{x}_3)} \geq 12$, and a lower bound on the asymptotic encoding rate for $k = 3$ is $R_l(0,3) = \frac{12}{13}$. This turns out to be exactly equal to the upper bound $R_u(0,3)$, and hence $R_l(0,3) = R_u(0,3) = \frac{12}{13} = R(0,3)$, thereby

yielding the exact asymptotic rate of the FRB algorithm for $k = 3$. This is a welcome coincedence since we did not backtrack through all the $2^k = 8$ paths. Proceeding similarly with $k = 4$, it can be shown that $R_l(0,4) = R_u(0,4) = \frac{24}{25} = R(0,4)$.

Next, with $k = 5$, we have from iteration 5 that the sequence $\mathbf{x}_4$ contains at least two 0^4 words (with $\mathbf{x}_4$ being scanned as a concatenation of words $\{1, 01, 0^21, 0^31, 0^4\}$) for every 0^5 word in $\mathbf{x}_5$ (with $\mathbf{x}_5$ being scanned as a concatenation of words $\{1, 01, 0^21, 0^31, 0^41, 0^5\}$). Now, since we have $\inf_{w_{\mathbf{u}_4}(\mathbf{x}_4)\in\mathbb{Z}^+} \frac{l(\mathbf{x})}{w_{\mathbf{u}_4}(\mathbf{x}_4)} = 24$, this implies that $\inf_{w_{\mathbf{u}_5}(\mathbf{x}_5)\in\mathbb{Z}^+} \frac{l(\mathbf{x})}{w_{\mathbf{u}_5}(\mathbf{x}_5)} \geq 48$. Hence, $R_l(0,5) = \frac{48}{49}$. This is less than the upper bound $R_u(5) = \frac{54}{55}$ determined in Chapter 7.3.1. Thus, there is a gap between the upper and lower bounds in this case.

Proceeding in a similar fashion, we obtain the general result that $\inf_{w_{\mathbf{u}^*_{k-1}}(\mathbf{x}_k)\in\mathbb{Z}^+} \frac{l(\mathbf{x})}{w_{\mathbf{u}^*_{k-1}}(\mathbf{x}_k)} \geq 2^{k+1} - 2^{k-1}$ for all $k \geq 2$. This yields the following lower bound, $R_l(0,k)$, on the asymptotic encoding rate. Our results are summarized in Table 15.

$$R_l(0,k) = \begin{cases} \frac{2}{3} & \text{when } k = 1 \\ \frac{2^{k+1} - 2^{k-1}}{2^{k+1} - 2^{k-1} + 1} & \text{when } k \geq 2. \end{cases} \tag{114}$$

From Table 15, it is seen that the asymptotic rate lower bounds for the FRB algorithm are very close to the corresponding upper bounds, and also to the average rate of the variable-rate bit stuff algorithm. We have already seen in earlier discussions (refer to equations (86) and (85)) that the variable-rate bit stuff algorithm is near-optimal for $(0,k)$ constraints. Thus, we conclude that the asymptotic encoding rate of the FRB algorithm gets very close to the $(0,k)$ capacity.

7.4 Asymptotic Partial Rates

Thus far, we have introduced the FRB algorithm for $(0,k)$ constraints, and computed bounds on its asymptotic encoding rate. High encoding rates were achieved

by iterative pre-processing of the input sequence to better conform it to bit insertion. For $k \geq 2$, the described pre-processing consisted of k iterations. The resulting asymptotic encoding rates were found to get close to the average rate, $R_0(0, k)$, of the variable-rate bit stuff algorithm. In this section, we investigate the effect of reducing the number of pre-processing iterations.

This is a practical consideration as we move to higher values of k. Recall that each of the k pre-processing iterations involves scanning, parsing and selective inversion of a potentially long input data block. In view of the related computations and latency, we can associate an implementation cost with each iteration. For a detailed analysis of the expected computations and latency of the FRB algorithm, the reader is directed to Chapter 8.4.

The combined implementation cost of k iterations could become significant as we move towards higher values of k. Hence, we would like to study the effect of partial pre-processing with $r < k$ iterations for $k \geq 2$. We consider $k - 1$ possibilities for partial pre-processing, which consist of the first r iterations, $r = 1, 2, \ldots, k - 1$, respectively. We derive upper and lower bounds on the asymptotic encoding rate for each r. Smaller values of r allow for very low-cost encoders and decoders, but the resulting codes are relatively inefficient. By adding more pre-processing iterations (increasing r), one can increase the encoding rate towards $R_0(0, k)$, thus making way for possible trade-offs between high encoding rate and implementation cost.

For a given maximum-run-length parameter k, let us denote by $R^r(0, k)$, the asymptotic r^{th} partial rate, *i.e.*, the asymptotic encoding rate with the first r pre-processing iterations, $r < k$. Hence, the pre-processing in short-hand notation now becomes

$1)1 \geq 0$

$2)01 \geq 00$

$\vdots$

$r)0^{r-1}1 \geq 0^r$

The rate computation proceeds in a similar vein to the earlier discussions in Chapter 7.3. An upper bound, $R_u^r(0,k)$, on the asymptotic r^{th} partial rate is now given by $R_u^r(0,k) = \frac{g^r(k)}{g^r(k)+1}$ where

$$g^r(k) = \inf_{i \geq k} g_i^r(k) = \inf_{i \geq k} \frac{2a_i^r}{\lfloor \frac{i}{k} \rfloor}, \tag{115}$$

where

$$a_i^r = \left(\sum_{j=1}^{r-1} q(i,j) a_j^r \right) + i, \tag{116}$$

and the function $q(i,j)$ is as specified in (101). The expression for $g^r(k)$ in (115) is similar to (102), with a_i^k replaced by a_i^r. We now use a result similar to Theorem 7.3 to limit our search space for $g^r(k)$.

Theorem 7.4 *For any given k, $2 \leq k < \infty$, and any $r < k$,*

$$\inf_{i \geq k} g_i^r(k) = \min_{k \leq i \leq k+b(k)-1} g_i^r(k), \tag{117}$$

where $b(k)$ denotes the LCM of positive integers up to and including k, i.e., *$b(k) = LCM(1,2,\ldots,k)$.*

The proof proceeds along similar lines to the proof of Theorem 7.3, and is hence omitted.

Theorem 7.4 reduces our search space for the positive integer $i \geq k$ that minimizes $g_i^r(k)$, to $k \leq i \leq k + b(k) - 1$. By means of computer search within this range, one can determine $g^r(k)$, and hence the upper bound $R_u^r(0,k)$ for any $2 \leq k < \infty$ and $1 \leq r < k$. As examples, we list these values for $k = 9$ and $k = 10$ in Tables 16 and 17, respectively.

A simple lower bound, $R_l^r(0,k)$, on the asymptotic r^{th} partial rate is given by

$$R_l^r(0,k) = \frac{h^r(k)}{h^r(k)+1}, \tag{118}$$

where

$$h^r(k) = \left\lfloor \frac{k}{r} \right\rfloor \left(\frac{R_l(0,r)}{1 - R_l(0,r)} \right), \tag{119}$$

and $R_l(0,r)$ is the lower bound on the asymptotic rate of the r constraint, as determined earlier in (114). Equation (119) follows from the simple fact that a 0^k word is composed of at least $\lfloor \frac{k}{r} \rfloor$ 0^r words. We leave the determination of tighter lower bounds as an open problem. In what follows, we summarize some important properties and observed trends in the asymptotic partial rates

1. The asymptotic 1^{st} partial rate, $R^2(0,k)$, is equal to $\frac{2k}{2k+1}$. The asymptotic 2^{nd} partial rate, $R^2(0,k)$, for any $k \geq 3$, is equal to $\frac{3k-3}{3k-2}$ for odd k, and $\frac{3k}{3k+1}$ for even k.

2. $R_u^r(k)$ is non-decreasing with increasing r. This follows directly from the fact that a_i^r is a non-decreasing function of r for any given $i \in \mathbb{Z}^+$.

3. For given k, the increase in $R_u^r(0,k)$ can be quite irregular. However the lower

Table 16: Summary of partial-rate computations for $k = 9$

r	$g^r(9)$	**Upper bound on asymptotic r^{th} partial rate $R_u^r(0,9)$**	**Lower bound on asymptotic r^{th} partial rate $R_l^r(0,9)$**
1	18	18/19	18/19
2	24	24/25	24/25
3	42	42/43	36/37
4	54	54/55	48/49
5	66	66/67	48/49
6	114	114/115	96/97
7	240	240/241	192/193
8	480	480/481	384/385
9	984	984/985	768/769

Table 17: Summary of partial-rate computations for $k = 10$

r	$g^r(10)$	**Upper bound on asymptotic r^{th} partial rate $R_u^r(0,10)$**	**Lower bound on asymptotic r^{th} partial rate $R_l^r(0,10)$**
1	20	20/21	20/21
2	30	30/31	30/31
3	42	42/43	36/37
4	54	54/55	48/49
5	114	114/115	96/97
6	114	114/115	96/97
7	240	240/241	192/193
8	480	480/481	384/385
9	984	984/985	768/769
10	1974	1974/1975	1536/1537

bound, $R_l^r(0,k)$, shows a pattern in the increase, with the function $h^r(k)$ doubling with each increasing r, starting from $r = \lfloor \frac{k}{2} \rfloor + 1$.

4. Smaller values of r imply low-cost encoding and decoding. For any given r, using a maximum-run-length parameter $k = vr$, $v \in \mathbb{Z}^+$, yields $h^r(k) = h^r(vr) = v\left(\frac{R_l(0,r)}{1-R_l(0,r)}\right)$, which guarantees a certain minimum code rate.

5. Suppose one wishes to design a code with asymptotic rate $\sim \frac{\rho}{\rho+1}$ (we use the

notation $x \sim y$ to imply x "close to" y), there could be several possibilities trading off the number of pre-processing iterations, r, for the maximum-run-length parameter k, such that $h^r(k) \sim \rho$. Suitable values for $\rho = 100$ are $r = 3, 4, 5, 6$ and $k = 24, 16, 10, 6$, respectively; and for $\rho = 200$ are $r = 5, 6, 7$ and $k = 20, 12, 7$, respectively.

7.5 Asymptotic Excess Rates

In contrast to partial pre-processing, one can also extend the pre-processing to beyond k iterations for any given $k < \infty$. This gives us the notion of s^{th} excess rate, which refers to the encoding rate with $s > k$ pre-processing iterations. In short-hand notation, the pre-processing now becomes

$1)1 > 0$

$2)01 > 00$

$\vdots$

$k)0^{k-1}1 > 0^k$

$k+1)0^k1 > 0^{k+1}$

$\vdots$

$s)0^{s-1}1 > 0^s$

The main result in this section is that only very small gains, if any, are possible by increasing the number of pre-processing iterations beyond k. The following discussion provides the details.

As before, we denote the output of the i^{th} iteration by $\mathbf{x}_i$, $i = 1, 2, \ldots, s$. First, we find an upper bound, $R_u^s(0, k)$, on the asymptotic s^{th} excess rate for the k-constraint. Similar to the discussion in Section 7.3.1, the upper bound is found by traversing a specific path on the binary search-tree shown in Fig. 15. The search-tree in this case consists of 2^s leaf nodes and the chosen path $\mathcal{P}$ is again the one that corresponds to retaining the input sequence at each iteration. The following definitions and results

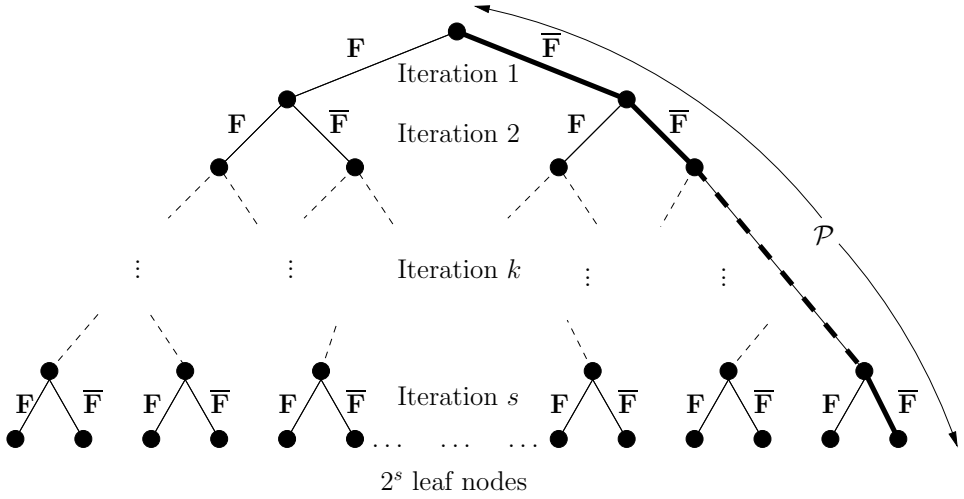

Figure 15: Binary search-tree to determine $\inf_{w_{\mathbf{u}^*_{k-1}}(\mathbf{x}_s)\in\mathbb{Z}^+} \frac{l(\mathbf{x})}{w_{\mathbf{u}^*_{k-1}}(\mathbf{x}_s)}$**. The path** $\mathcal{P}$ **marked in bold is the one traversed to determine an upper bound on the asymptotic excess rate.**

concerning the upper bound are all with respect to path $\mathcal{P}$.

Definition 7.5 *For given* $s, k \in \mathbb{Z}^+$, $s > k$, *and path* $\mathcal{P}$ *shown in Fig. 14, the* effective count, $\beta_i^s(k)$, *of the word* 0^i1, $i \in \mathbb{Z}^+$, *is the minimum number of* 0^k *words in* $\mathbf{x}_k$ *resulting from the presence of a single* 0^i1 *word in* $\mathbf{x}_s$, *with* $\mathbf{x}_s$ *being scanned as a concatenation of words* $0^{t-1}1$, $t \in \mathbb{Z}^+$, *and* $\mathbf{x}_k$ *being scanned as a concatenation of words from the set* $\left\{\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{k-1}, \mathbf{u}^*_{k-1}\right\}$.

We now *backtrack* through the binary search-tree in a similar fashion to the discussions in Chapter 7.3. By working backward from iteration s through to iteration k, we derive the following expression for $\beta_i^s(k)$. The details of this derivation are similar to those of Lemma 7.2, and are hence omitted.

Proposition 7.6 *For any* $s, k, i \in \mathbb{Z}^+$, $s > k$, $i \geq k$, *the effective count,* $\beta_i^s(k)$ *is given by*

$$\beta_i^s(k) \;=\; \left(\sum_{j=k}^{s-1} q(i,j)\beta_j^j(k)\right) + \left\lfloor \frac{i}{k} \right\rfloor, \tag{120}$$

where the function $q(i,j)$ is as specified earlier in (101). We rewrite the expression here for convenience.

$$q(i,j) \;=\; \begin{cases} \max\left\{0, \frac{i+1}{j+1} - 2\right\} & \text{when } \frac{i+1}{j+1} \in \mathbb{Z}^+ \\ \lfloor \frac{i+1}{j+1} \rfloor & \text{when } \frac{i+1}{j+1} \notin \mathbb{Z}^+. \end{cases}$$

The effective count is essentially a measure of the number of 0^k words that ultimately lead to the effective weight (see Definition 7.1, Section 7.3.1) number of zeros in $\mathbf{x}_1$. Hence, it is a means of connecting properties of $\mathbf{x}_s$ to an upper bound on the asymptotic rate, via the properties of $\mathbf{x}_k$. This is formalized in the following definition.

Definition 7.7 *For given $s, k \in \mathbb{Z}^+$, $s > k$, the function $g_i^s(k)$ is defined as two times the ratio of the* effective weight *to the* effective count*, of the word 0^i1, $i \in \mathbb{Z}^+$, in $\mathbf{x}_s$, with $\mathbf{x}_s$ being scanned as a concatenation of words $0^{t-1}1$, $t \in \mathbb{Z}^+$. Mathematically, $g_i^s(k) = \frac{2a_i^s}{\beta_i^s(k)}$.*

Hence, rather convolutely, $g_i^s(k)$ represents the minimum number of input bits per 0^k word in $\mathbf{x}_k$, resulting from a single 0^i1 word in $\mathbf{x}_s$. The reader may note that the number of 0^k words in $\mathbf{x}_k$ gives us exactly, the number of 0^k words in $\mathbf{x}_s$ because of our choice of path $\mathcal{P}$. Recall that in the path $\mathcal{P}$, the input is retained as it is at each iteration, *i.e.*,

$$\mathbf{x}_s = \mathbf{x}_{s-1} = \mathbf{x}_{s-2} = \ldots = \mathbf{x}_2 = \mathbf{x}_1 = \mathbf{x}.$$

Thus, in order to determine the number of 0^k words in $\mathbf{x}_s$, we had to work backward starting from iteration s up toward iteration k.

The infimum of $g_i^s(k)$ over all $i \geq k$ will give us the necessary minimum input block length $l(\mathbf{x})$. Hence, an upper bound on the asymptotic s^{th} excess rate is given by

$$R_u^s(0,k) = \frac{g^s(k)}{g^s(k)+1}, \quad s > k, \tag{121}$$

where $g^s(k) = \inf_{i \geq k} g_i^s(k)$. If for a given k and some $s > k$, this infimum is greater than $f(k)$, then it implies that there are upper bound improvements in the asymptotic rate by increasing the number of pre-processing iterations to s. Our aim now is to determine the range of values of s for which such an improvement is possible. We start with the following proposition.

Proposition 7.8 *For any $s, k, i \in \mathbb{Z}^+$, $s > k$, the effective weights, a_i^k and a_i^s, satisfy*

$$a_i^k = a_i^s = a_i^i \quad \text{when } i \leq k \tag{122}$$

$$a_i^k < a_i^s = a_i^i \quad \text{when } k < i \leq s \tag{123}$$

$$a_i^k < a_i^s \quad \text{when } i > s. \tag{124}$$

The above relations follow directly from Lemma 7.2, and are useful in proving the following result.

Lemma 7.9 *For any $s, k, j \in \mathbb{Z}^+$, $s > k$, $k \leq j \leq s-1$, the following relationship holds*

$$\frac{a_j^s}{\beta_j^j(k)} \geq \frac{f(k)}{2}, \tag{125}$$

where $f(k) = \min_{k \leq i \leq k+b(k)-1} \frac{2a_i^k}{\lfloor \frac{i}{k} \rfloor}$ from Section 7.3.1.

Proof: We provide a proof by induction. When $j = k$, we have from Proposition 7.8 that $a_j^s = a_j^k = a_k^k$, and $\beta_k^k(k) = 1$. Hence, $\frac{a_j^s}{\beta_j^j(k)} = a_k^k \geq \frac{f(k)}{2}$ as required. Let us now assume that the induction hypothesis is true, that is $\frac{a_j^s}{\beta_j^j(k)} \geq \frac{f(k)}{2}$, for each $j = k, k+1, \ldots, t-1$, where $t < s$. Our task is to show that $\frac{a_j^s}{\beta_j^j(k)} \geq \frac{f(k)}{2}$, for $j = t$. This can be done using the following steps

$$\begin{aligned} \frac{a_t^s}{\beta_t^t(k)} &= \frac{a_t^t}{\beta_t^t(k)} && (126) \\ &= \frac{a_t^k + \left(\sum_{l=k}^{t-1} q(t,l) a_l^t\right)}{\lfloor \frac{t}{k} \rfloor + \left(\sum_{l=k}^{t-1} q(t,l) \beta_l^l(k)\right)}, && (127) \end{aligned}$$

where (127) is obtained using (100) and (120). Now since $t > l$, we have from Proposition 7.8 that $a_l^t = a_l^l$, for each $l = k, k+1, \ldots, t-1$. Further from the hypothesis assumption, we have that $\frac{a_l^l}{\beta_l^l(k)} \geq \frac{f(k)}{2}$, for each $l = k, k+1, \ldots, t-1$. We also know from the definition of $f(k)$ that $\frac{a_t^k}{\lfloor \frac{t}{k} \rfloor} \geq \frac{f(k)}{2}$. Hence, we conclude from (127) that $\frac{a_t^s}{\beta_t^t(k)} \geq \frac{f(k)}{2}$. ■

The inequality of Lemma 7.9 is useful in proving the following result, which implies that the asymptotic excess-rate upper bound, for any $s > k$, is not less than the asymptotic rate upper bound with k iterations.

Theorem 7.10 *For any $s, k \in \mathbb{Z}^+$, $s > k$, let $g^s(k) = \inf_{i \geq k} g_i^s(k)$. Then the following is true*

$$g^s(k) \;\; \geq \;\; f(k), \tag{128}$$

where $f(k) = \inf_{i \geq k} f_i(k)$ as specified earlier in Section 7.3.1.

Proof: Let us start with

$$\frac{g^s(k)}{2} \;\; = \;\; \inf_{i \geq k} \frac{a_i^s}{\beta_i^s(k)} \tag{129}$$

$$= \;\; \inf_{i \geq k} \frac{\left(\sum_{j=1}^{s-1} q(i,j) a_j^s\right) + i}{\left(\sum_{j=k}^{s-1} q(i,j) \beta_j^j(k)\right) + \left\lfloor \frac{i}{k} \right\rfloor}, \tag{130}$$

where (130) follows from (100) and (120). The numerator of (130) can be rewritten to get

$$\frac{g^s(k)}{2} \;\; = \;\; \inf_{i \geq k} \frac{\left(\sum_{j=1}^{k-1} q(i,j) a_j^k\right) + i + \left(\sum_{j=k}^{s-1} q(i,j) a_j^s\right)}{\left\lfloor \frac{i}{k} \right\rfloor + \left(\sum_{j=k}^{s-1} q(i,j) \beta_j^j(k)\right)}. \tag{131}$$

Part of the numerator, $\left(\sum_{j=1}^{k-1} q(i,j) a_j^k\right) + i$, is nothing but the expression for a_i^k (refer to equation (100)). Further, we know that for any $i \geq k$

$$\frac{a_i^k}{\left\lfloor \frac{i}{k} \right\rfloor} = \frac{f_i(k)}{2} \geq \frac{f(k)}{2}. \tag{132}$$

Hence, in order to prove the inequality of (128), it remains to be shown that $\frac{a_j^s}{\beta_j^j(k)} \geq \frac{f(k)}{2}$, for each $j = k, k+1, \ldots, s-1$. However, this is exactly the result of Lemma 7.9, thus completing the proof. ■

The following important corollaries can now be drawn.

Corollary 7.11 *Let* $i_k = \arg\min_{k \le i \le k+b(k)-1} f_i(k)$ *as defined in Chapter 7.3.1. If* $i_k = k$*, then* $R_u^s(0,k) = R_u(0,k)$ *for all* $s > k$*.*

Corollary 7.12 $R_u^s(k) > R_u(k)$ *if and only if* $i_k > k$*.*

The above Corollaries can be proved as follows. The upper bound on the asymptotic s^{th} excess rate is given by (121). Theorem 7.10 states that $g^s(k) \geq f(k)$, which implies that $R_u^s(0,k) \geq R_u(0,k)$ for each $s > k$. However, for the special case when $i_k = k$, we see from (131) and Proposition 7.8 that $g_k^s(k) = f_k(k)$, and hence $R_u^s(0,k) = R_u(0,k)$ as in Corollary 7.11. This leads to the important fact that increasing the number of pre-processing iterations yields no improvements in the asymptotic rate upper bound if $i_k = k$. In other words, improved upper bounds may be possible only if $i_k > k$, as stated in the forward part of Corollary 7.12. To prove the reverse part of Corollary 7.12, consider equation (131). We have already seen in the proof of Theorem 7.10 that $\frac{\left(\sum_{j=1}^{k-1} q(i,j)a_j^k\right)+i}{\lfloor \frac{i}{k} \rfloor} = \frac{a_i^k}{\lfloor \frac{i}{k} \rfloor} \geq \frac{f(k)}{2}$ for any $i \geq k$. Further from Lemma 7.9, we have that $\frac{a_j^s}{\beta_j^j(k)} \geq \frac{f(k)}{2}$, for each $j = k, k+1, \ldots, s-1$. Hence, for $i_k > k$, we not only have $g^s(k) \geq f(k)$ as in Theorem 7.8, but also that $g^s(k) = f(k)$ if and only if the following two conditions are simultaneously satisfied for some $i \geq k$

1. $\frac{a_i^k}{\lfloor \frac{i}{k} \rfloor} = \frac{f(k)}{2}$
2. Either $q(i,j) = 0$ or $\frac{a_j^s}{\beta_j^j(k)} = \frac{f(k)}{2}$, for each $j = k, k+1, \ldots, s-1$,

Let the first condition be satisfied for some $i = i^*$. Since i_k is defined to be the minimum of all $\arg\min_{k \le i \le k+b(k)-1} f_i(k)$, we have that $i^* \geq i_k$. Let us assume that $i^* \neq 2k+1$. Since $i_k > k$, we have that $q(i^*, j)$ (refer to (101)) is non-zero for at least one value of j, namely $j = k$. For the same reason, we have that $\frac{a_j^s}{\beta_j^j(k)} > \frac{f(k)}{2}$ for $j = k$.

For the case when $i^* = 2k+1$, we have that $q(i^*,k) = 0$, but then $q(i^*,k+1) \neq 0$, and $\frac{a_j^s}{\beta_j^j(k)} > \frac{f(k)}{2}$ for $j = k+1$. Hence, we conclude that the above two conditions cannot be jointly satisfied for any $i \geq k$. This implies that $g^s(k)$ is strictly greater than $f(k)$ when $i_k > k$, thus completing the sufficiency argument.

We now derive a result similar to Theorem 7.10, but one that is much stronger. Theorem 7.13 implies that increasing the number of pre-processing iterations can only, if at all, increase $g^s(k)$.

Theorem 7.13 *For any given $s, k \in \mathbb{Z}^+$, $s > k$, we have $g^s(k) \leq g^{s+1}(k)$.*

Proof: Starting with (131), we can write down

$$g^{s+1}(k) = \inf_{i \geq k} \frac{a_i^k + \left(\sum_{j=k}^{s} q(i,j) a_j^{s+1}\right)}{\lfloor \frac{i}{k} \rfloor + \left(\sum_{j=k}^{s} q(i,j)\beta_j^j(k)\right)} \tag{133}$$

$$= \inf_{i \geq k} \frac{a_i^k + \left(\sum_{j=k}^{s-1} q(i,j) a_j^{s}\right) + q(i,s)a_s^s}{\lfloor \frac{i}{k} \rfloor + \left(\sum_{j=k}^{s-1} q(i,j)\beta_j^j(k)\right) + q(i,s)\beta_s^s(k)}. \tag{134}$$

Similarly, we have that

$$g^s(k) = \inf_{i \geq k} \frac{a_i^s}{\beta_i^s(k)} \tag{135}$$

$$= \inf_{i \geq k} \frac{a_i^k + \left(\sum_{j=k}^{s-1} q(i,j) a_j^{s}\right)}{\lfloor \frac{i}{k} \rfloor + \left(\sum_{j=k}^{s-1} q(i,j)\beta_j^j(k)\right)}. \tag{136}$$

Comparing (134) and (136), we see that proving $g^s(k) \leq g^{s+1}(k)$ reduces to proving $\frac{a_s^s}{\beta_s^s(k)} \geq g^s(k)$. However, this follows directly from (135). ■

Next, we obtain the following result to help us evaluate $R_u^s(0,k)$.

Theorem 7.14 *For any given $s, k \in \mathbb{Z}^+$,*

$$\inf_{i \geq k} g_i^s(k) \quad = \quad \min_{k \leq i \leq k+b(s)-1} g_i^s(k), \tag{137}$$

where $b(s)$ denotes the LCM of positive integers upto and including s, i.e., *$b(s) = LCM(1, 2, \ldots, s)$.*

Proof: The proof uses ideas similar to that of Theorem 7.2, and is hence omitted. ■

The following corollaries are direct consequences of Theorems 7.13 and 7.14

Corollary 7.15 *If for some $s > k$, the minimizing index $i_s = \arg\min_{i \geq k} g_i^s(k)$ is not greater than s, then for each $s' > s > k$, we have $R_u^{s'}(0, k) = R_u^s(0, k)$ with corresponding $i_{s'} = s$.*

Corollary 7.16 *$R_u^{s+1}(0, k) > R_u^s(0, k)$ if and only if $i_s > s$.*

Corollary 7.17 *For any given k, the largest asymptotic excess rate upper bound is obtained by setting $s = s^*$, where $i_{s^*} \leq s^*$ and $i_s > s$ for each $s < s^*$.*

As examples, consider $k = 1, 2, 3, 5, 7, 11$. We note from Table 15 that $i_k = k$ in each of these cases. Hence, increasing the number of pre-processing iterations cannot improve upon $R_u(k)$ values for these k. On the other hand, for a value of $k = 4$, we find that $i_4 = 5 > k$, and hence, there is scope for upper bound improvements. $R_u^s(0, k)$ values for $k = 4$ and $k = 6$ are listed in Table 18 and Table 19, respectively. For corresponding lower bounds, we use $R_l^s(k) = R_l(k)$. Improving upon the lower bound is once again left open.

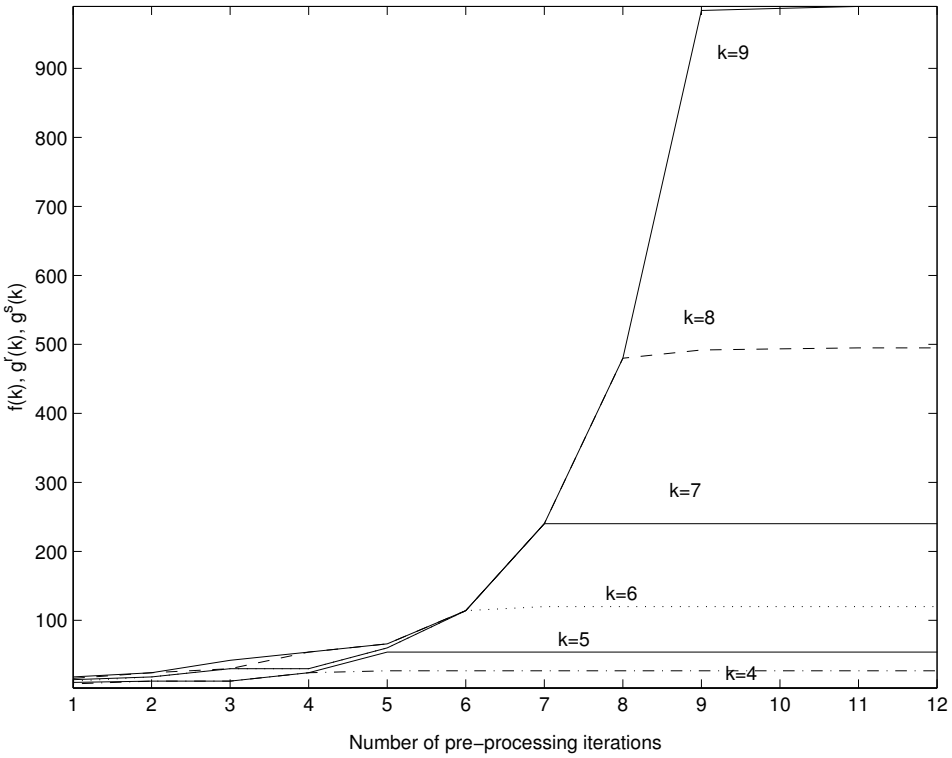

Figure 16: The figure shows the behavior of the entire range of partial and excess pre-processing upper bounds for values of $k = 4$ through 9. It plots $y \in \{f(k), g^r(k), g^s(k)\}$ as a function of the number of pre-processing iterations. Recall that $f(k) = R_u(k)/(1 - R_u(k))$, $g^r(k) = R_u^r(k)/(1 - R_u^r(k))$ and $g^s(k) = R_u^s(k)/(1 - R_u^s(k))$. It is seen that y tapers off after k iterations, which confirms that very small gains, if any, are possible from excess pre-processing.

Table 18: Summary of asymptotic excess rate computations for $k = 4$

s	i_s	$R_u^s(0,4)$
5	5	27/28
6	5	27/28
7	5	27/28

The preceding study suggests that only marginal rate improvements, if any, are possible with the addition of pre-processing iterations beyond k. This is confirmed in Fig. 16, which shows $y \in \{f(k), g^r(k), g^s(k)\}$ as a function of the number of pre-processing iterations, for values of $k = 4$ through 9. This gives us an idea of the

Table 19: Summary of asymptotic excess rate computations for $k = 6$

s	i_s	$R_u^s(0,4)$
7	7	120/121
8	7	120/121

corresponding asymptotic-rate upper bound, which is nothing but $y/(y+1)$. There is a steep increase in y until k iterations, after which it only shows very slight increase, if any. Hence, only very small gains are possible from excess pre-processing. Considering the increase in pre-processing and post-processing complexity, this is probably not an encouraging option. One would rather work with a greater maximum-run-length parameter k, and use k iterations to get significantly more gains.

This concludes our discussion on the asymptotic (in input block length) performance of the FRB algorithm for $(0, k)$ constraints. We have carried out a detailed rate analysis and computed upper and lower bounds on the asymptotic encoding rates. We have also computed similar performance bounds for the special cases when fewer/additional pre-processing iterations are used. In our analysis thus far, we have allowed the input blocks to be infinitely long. Needless to say, this is impossible in any practical system. Several practical issues including FRB encoding with finite input-blocks are considered in Chapter 8.

Before proceeding to discuss the system issues however, we present an extension of the fixed-rate bit stuff encoding ideas to the class of $(0, G/I)$ constraints. Once again, the key to generating efficient fixed-rate codes is to use iterative pre-processing prior to bit stuffing.

7.6 Fixed-Rate Codes: Extension to $(0, G/I)$ Constraints

In Chapter 6.1, we presented a variable-rate bit stuff algorithm for $(0, G/I)$ constraints, and outlined a procedure to compute its average encoding rate. It was seen that the proposed encoding generates near-capacity codes for a wide range of values of G and I. In what follows, we describe a fixed-rate bit stuff algorithm for $(0, G/I)$ constraints that is based on the variable-rate algorithm of Chapter 6.1. We restrict ourselves to Classes 1 and 2, *i.e.* G even, $0 \leq G \leq 2I$ for the fixed-rate codes.

To design high-rate codes, we use iterative pre-processing similar to the $(0, k)$ FRB codes in Chapter 7.2. However, the pre-processing in this case applies to each of the even and odd subsequences rather than the global sequence. The block diagram of the fixed-rate encoder for $(0, G/I)$ constraints, $0 < I < \infty$, $0 < G \leq 2I$, G even, is shown in Fig. 17. It accepts an unconstrained binary sequence, $\mathbf{x} = (x_0 x_1 \ldots, x_{2m-1})$, of fixed-length $2m$ bits and outputs a $(0, G/I)$ sequence, $\mathbf{y} = (y_0 y_1 \ldots, y_{n-1})$, of a fixed-length n bits. The encoding proceeds in five stages. First, the even and odd subsequences, $\mathbf{x}_e = (x_0 x_2 \ldots, x_{2m-2})$ and $\mathbf{x}_o = (x_1 x_3 \ldots, x_{2m-1})$, are extracted (DEMUX) from the input $\mathbf{x}$. Each of the subsequences $\mathbf{x}_e$ and $\mathbf{x}_o$, then undergoes I iterations of pre-processing, which is shown as $PP(I)$ in Fig. 17. The individual pre-processing outputs $\mathbf{v}_e$ and $\mathbf{v}_o$, are then interleaved bit-by-bit (MUX) to yield the sequence $\mathbf{v}$. In the fourth stage, $\mathbf{v}$ undergoes variable-rate bit stuffing, as discussed in Section 6.1.1, thus producing variable-length sequences $\mathbf{y}'$. Finally, the output sequence $\mathbf{y}$ is formed by dummy-bit padding $\mathbf{y}'$ to a fixed output-length n bits. These dummy bits can be ignored during decoding as explained in Chapter 7.1.

The operation $PP(I)$ is exactly the same as the iterative pre-processing in the FRB algorithm of Chapter 7.2. The main idea once again is to repeatedly scan and parse the input subsequence so as to identify undesirable bit stuff patterns. These patterns then undergo a reversible, selective inversion. Once again, we use index

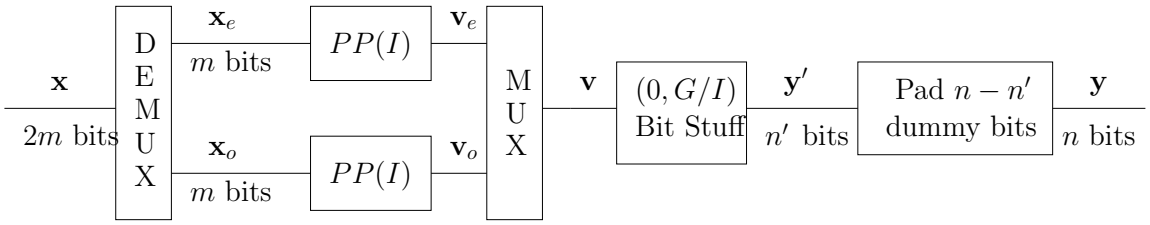

Figure 17: Block diagram of the proposed fixed-rate encoder for $(0, G/I)$ constraints. The proposed encoding can be viewed in three stages. The input $\mathbf{x}$ is first separated into even and odd subsequences, $\mathbf{x}_e$ and $\mathbf{x}_o$, each of which undergoes iterative pre-processing. The second stage is the variable-rate bit stuff encoding as discussed in Section 6.1.1. This produces variable-length output sequences $\mathbf{y}'$. Finally $\mathbf{y}'$ is padded-up using dummy-bits to a fixed output-length n bits. The pre-processing operations $PP(I)$ involve I iterations, and are key to building efficient fixed-rate codes.

sequences to convey the pre-processing information to the decoder. As before, these index sequences are included in $\mathbf{v}_e$ and $\mathbf{v}_o$ (refer to the discussion in Chapter 7.2).

Essentially, the combination of demuxing, pre-processing and muxing in Fig. 17, transforms the input sequence $\mathbf{x}$ into a sequence $\mathbf{v}$, that is better conformed to bit stuffing. This gain is possible only at the price of the index penalty. However, the index penalty does not grow with m, thus enabling the construction of efficient fixed-rate $(0, G/I)$ codes.

We now formally describe the pre-processing $PP(I)$ to make this discussion self-contained. We remind the reader of the following notations and definitions.

Notations and Definitions

- Let

$$\mathbf{u}_i = 0^i 1, \quad i = 0, 1, \ldots, I-1 \tag{138}$$

$$\mathbf{u}_i^* = 0^{i+1}, \quad i = 0, 1, \ldots, I-1. \tag{139}$$

and $\mathbf{u}_i^* = 0^{i+1}$, $i = 0, 1, \ldots, I-1$. Hence, by definition, $\mathbf{u}_{I-1}^* \equiv \mathbf{u}_I$.

- The weight of a binary sequence $\mathbf{s}$ with respect to the input word $\mathbf{u}_i^*$, denoted by $w_{\mathbf{u}_i^*}(\mathbf{s})$, is the number of distinct occurrences of $\mathbf{u}_i^*$ in $\mathbf{s}$, with $\mathbf{s}$ being scanned as a concatenation of words from the set $\{\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_i, \mathbf{u}_i^*\}$, $i = 0, 1, \ldots, I-1$. For example, if $\mathbf{s} = 100001010001$, $I \geq 1$ and $i = 1$, $w_{\mathbf{u}_1^*}(\mathbf{s}) = 3$.

- The weight of a binary sequence $\mathbf{s}$ with respect to the input word $\mathbf{u}_i$, denoted by $w_{\mathbf{u}_i}(\mathbf{s})$, is the number of distinct occurrences of $\mathbf{u}_i$ in $\mathbf{s}$, with $\mathbf{s}$ being scanned as a concatenation of words from the set $\{\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_i, \mathbf{u}_i^*\}$, $i = 0, 1, \ldots, I-1$. For example, if $\mathbf{s} = 100001010001$, $I \geq 1$ and $i = 1$, $w_{\mathbf{u}_1}(\mathbf{s}) = 2$.

- Denote by $\overline{\mathbf{s}}(i)$, the sequence formed by converting all $\mathbf{u}_i$ words to $\mathbf{u}_i^*$ words, and all $\mathbf{u}_i^*$ words to $\mathbf{u}_i$ words in $\mathbf{s}$, with $\mathbf{s}$ being scanned as a concatenation of words from the set $\{\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_i, \mathbf{u}_i^*\}$, $i = 0, 1, \ldots, I-1$. For example, if $\mathbf{s} = 100001010001$, and $I \geq 1$, $\overline{\mathbf{s}}(1) = 101011000100$. Note that $\overline{\mathbf{s}}(i)$ is an invertible operation.

- Denote by $\mathbf{s}_1 \parallel \mathbf{s}_2$, the concatenation of two sequences $\mathbf{s}_1$ and $\mathbf{s}_2$.

- Denote by $\boldsymbol{\alpha}_{\mathbf{s}} = [\alpha_1 \ \alpha_2 \ \ldots \ \alpha_I]$ the index sequence corresponding to input sequence $\mathbf{s}$. The index bits are used to convey to the decoder whether or not the flipping operation is performed in each of the I iterations.

Table 20: Lower bound on the asymptotic rate of fixed-rate $(0, G/I)$ codes

G	I	**Lower bound on $\mathcal{R}_{G/I}$**
12	6	96/97
8	6	32/33
14	7	192/193
10	7	64/65
16	8	384/385
12	8	128/129
10	8	768/778

The iterative pre-processing $PP(I)$ is defined as follows. The input is $\mathbf{x}_{e/o}$: either the even subsequence $\mathbf{x}_e$ or the odd subsequence $\mathbf{x}_o$ as appropriate.

Input $\mathbf{x}_{e/o}$

Set $\mathbf{x}_0 = \mathbf{x}_{e/o}$

For $j = 1$ to I

  Input $\mathbf{x}_{j-1}$

  Scan $\mathbf{x}_{j-1}$ as a concatenation of words from the set $\left\{\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{j-1}, \mathbf{u}^*_{j-1}\right\}$

  If $w_{\mathbf{u}_{j-1}}(\mathbf{x}_{j-1}) < w_{\mathbf{u}^*_{j-1}}(\mathbf{x}_{j-1})$

    $\mathbf{x}_j = \overline{\mathbf{x}}_{j-1}(j-1)$

    $\alpha_j = 1$

  Else

    $\mathbf{x}_j = \mathbf{x}_{j-1}$

    $\alpha_j = 0$

  end

end

$\mathbf{v}_{e/o} = \mathbf{x}_I \parallel \boldsymbol{\alpha}_{\mathbf{x}_{e/o}}$

The encoding rate of the fixed-rate algorithm is given by $\mathcal{R}_{G/I}(m) = \frac{2m}{n}$. Let us denote the asymptotic encoding rate by $\mathcal{R}_{G/I} = \lim_{m\to\infty} \frac{2m}{n}$. The computation of $\mathcal{R}_{G/I}$ proceeds along similar lines to the discussion on fixed-rate $(0, k)$ codes in Chapter 7.3. Lower bounds on $\mathcal{R}_{G/I}$ for some selected values of G and I are given in Table 20.

CHAPTER VIII

FRB CODES: INTEGRATION INTO A DIGITAL RECORDING SYSTEM

In the previous chapter, we introduced the fixed-rate bit stuff (FRB) algorithm to generate very high-rate $(0, k)$ codes. We presented a detailed rate analysis, and showed that the asymptotic (in input block length) encoding rate was very close to the average rate of the variable-rate bit stuff code, which in turn approached capacity for values of $k \geq 5$. Thus, in theory, the FRB algorithm provides an effective means to generate near-capacity $(0, k)$ sequences. However, integrating the FRB codes into a practical recording system raises several other questions. These system issues, namely the effect of finite block-lengths, error propagation, DC suppression and implementation complexity, are the subject of our discussions in this chapter.

8.1 Encoding Rates for Finite Block Lengths

Thus far, our primary interest has been to analyze the asymptotic encoding rate of the FRB algorithm. Very conveniently, the effect of additional index bits could be neglected in all asymptotic rate computations. However, such an index overhead shows up for any finite input block length. In this section, we incorporate the rate loss due to pre-processing index overhead, and compute the input block lengths required to design codes with rate close to 100/101 and 200/201.

We start over from the expression in (94) for the encoding rate, $R(0, k, m)$. For finite m, the term $\beta(k, m) = \max_{\mathbf{x} \in \mathbb{Z}_2^m} w_{\mathbf{u}_{k-1}^*}(\mathbf{x}_k)$ is related to the asymptotic rate lower bound, $R_l(0, k)$, as

Table 21: Lower bound on encoding rates for finite input block lengths with $k = 9$

m	**Lower bound on** $R(0, 9, m)$
100	10/11
500	50/51
1000	90/91
2000	166/167
3000	230/231
4000	266/267
∞	768/769

Table 22: Possible r,k,m values that achieve encoding rates close to 100/101

r	k	m
4	20	3000
5	15	2000
6	12	1500
7	14	1100
8	16	1000

Table 23: Possible r,k,m values that achieve encoding rates close to 200/201

r	k	m
6	18	4600
7	14	3400
8	16	2500
9	18	2300
9	9	2700

Table 24: Possible G,I,m that achieve encoding rates close to 100/101

G	I	m
14	7	1670
16	8	1220
12	8	4350
18	9	1150
14	9	1730
12	9	3010

Table 25: Possible G,I,m that achieve encoding rates close to 200/201

G	I	m
16	8	3760
18	9	2710
16	10	3780
18	11	3110

$$\beta(k,m) \;\leq\; \Big\lfloor \frac{m(1-R_l(0,k))}{R_l(0,k)} \Big\rfloor, \tag{140}$$

where $R_l(0,k)$ is as specified in (114). For $k = 1, 2, 3, 4$, (140) is satisfied with equality (see Appendix A), and we have an exact expression for $R(0,k,m)$ in these cases. For $k \geq 5$, we obtain the following lower bound using (140), (114) and (94).

$$R(0,k,m) \;\geq\; \frac{m}{m + \lfloor \frac{m}{2^{k+1}-2^{k-1}} \rfloor + (k+1)} \tag{141}$$

Once again, we extend our computations to include partial rates, $R^r(0,k,m)$, using (119).

$$R^r(0,k,m) \;\geq\; \frac{m}{m + \left\lfloor \frac{m}{\lfloor \frac{k}{r} \rfloor \left(\frac{R_l(0,r)}{1-R_l(0,r)} \right)} \right\rfloor + (r+1)} \tag{142}$$

Comparing (114), (141) and (142), one can estimate the loss in encoding rate caused by the index overhead for finite values of m. Table 21 lists the lower bounds on $R(0,k,m)$ for $k = 9$. For comparison purposes, they are truncated to the form $\frac{y}{y+1}$, $y \in \mathbb{Z}^+$. In Tables 22 and 23, we specify some possible r, k, m values that can be

used to design fixed-rate codes with encoding rate $\sim \frac{\rho}{\rho+1}$ for $\rho = 100$ and $\rho = 200$, respectively. We point out that current state-of-the-art codes: rate $8/9, k = 3$; rate $16/17, k = 6$; and rate $64/65, k = 7$, can be designed using the fixed-rate algorithm with input block lengths $m = 96$ bits; $m = 128$ bits; and $m = 704$ bits, respectively.

It is straightforward to extend the above computations to the fixed-rate algorithm for $(0, G/I)$ constraints, which was discussed in Chapter 7.6. Tables 24 and 25 show a set of possible G, I, m values which can be used to design fixed-rate $(0, G/I)$ codes with rate close to 100/101 and 200/201, respectively.

8.2 Error Propagation

The FRB algorithm generates near-capacity $(0, k)$ codes when the encoding is performed in long, input and output blocks. In Chapter 8.1, we saw that input block lengths of a few thousand bits are required to design rate 100/101 and rate 200/201 codes for practical recording systems. However, the use of such long block codes opens a new set of problems: that of error propagation.

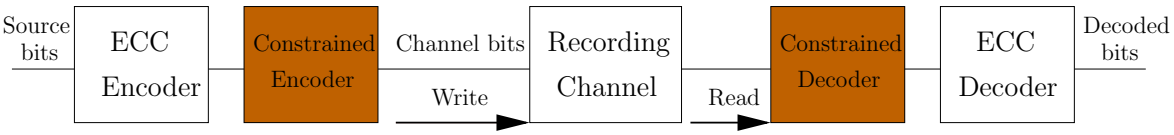

Figure 18: Block diagram of a digital recording system.

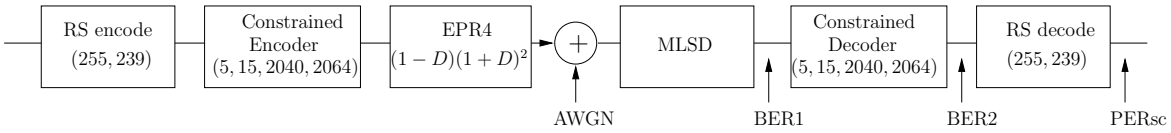

Figure 19: System model for standard concatenation using the $(0, k)$ fixed-rate bit stuff codes.

Let us go back to our model of the digital recording system in Fig. 1, Chapter 2. This is redrawn in Fig. 18 for convenience. Our aim was to design the constrained

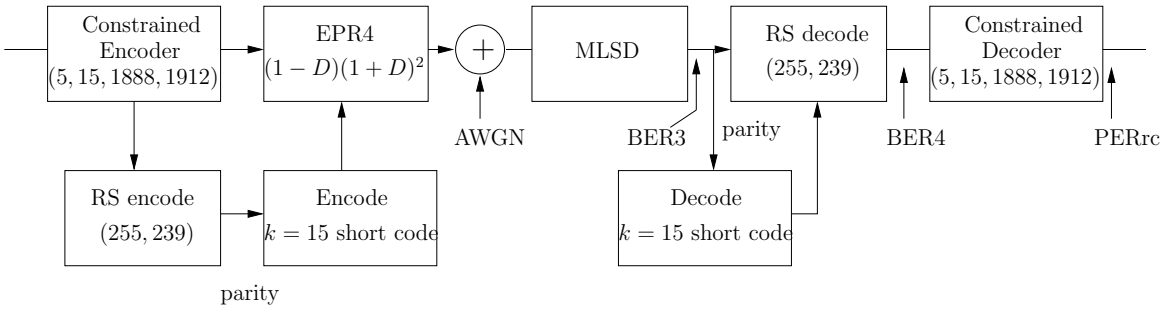

Figure 20: System model for Bliss's reverse-concatenation using the $(0, k)$ fixed-rate bit stuff codes.

encoder and decoder blocks. This has been the subject of our discussions in Chapters 4, 5, 6 and 7. While coding for constraints, we have implicitly assumed that the data written onto the recording surface can be correctly read back. In other words, for theoretical purposes, we have made the assumption of a noise-free recording channel, or that in the case of noisy channels, all channel bit errors are rectified by a powerful error control code (ECC). This assumption is hardly valid in any practical communication system, and more so, in magnetic and optical storage systems that are prone to severe intersymbol interference (ISI) and noise [28]. Thus, it is inevitable that one or more of the channel bits will be read-out in error. The term error propagation refers to the fact that a single channel bit error can lead to multiple bit-errors in the decoded sequence. This effect is typical of variable-rate constrained codes, and fixed-rate constrained codes that operate with very long blocks. Indeed, in the worst case it is possible that a large portion of the decoded sequence is in error due to a single channel bit error. Hence, it is important to evaluate the error propagation performance of any constrained code design.

8.2.1 Performance under Reverse Concatenation

The configuration shown in Fig. 18 is commonly referred to as the standard concatenation configuration. In standard concatenation, the constrained encoder follows the

ECC encoder. One possible solution to the error propagation problem of large block-length constrained codes is to reverse this configuration. This is referred to as the reverse concatenation configuration, wherein the ECC encoder now follows the constrained encoder. In this subsection, we study the performance of a rate $\sim 100/101$ $(0, k)$ FRB code in a magnetic recording system with reverse concatenation, and compare the performance to that with standard concatenation.

The reverse-concatenation scheme we use is the one proposed by Bliss, and analyzed in detail by Fan and Calderbank [13]. Figs. 19 and 20 show the standard concatenation and reverse-concatenation configurations, respectively, that are used in our simulations. In order to analyze the error propagation performance of the FRB codes, we need to incorporate a model for the detection process during read-out (marked "Read" in Fig. 18). Historically, magnetic recording systems used a peak detection scheme [18], but current high-density recording systems use a technique referred to as PRML [57],[9],[18], short for partial-response (PR) equalization with maximum-likelihood (ML) sequence detection. For our purposes, the extended partial-response Class-4 (EPR4) model, with a discrete-time transfer function of $1 + D - D^2 - D^3$, is used to model the ISI in the magnetic recording channel. This is appropriate for a density ratio in the range 1.6 to 2.2 [57],[28], which represents a medium-to-high recording density. Furthermore, we restrict ourselves to an additive white gaussian noise (AWGN) model.

Forney [14] proved the general result that the maximum-likelihood sequence detector (MLSD) for any uncoded, linear intersymbol interference channel with AWGN comprises a whitened matched filter, whose output is sampled at the symbol rate, followed by a Viterbi detector whose trellis structure reflects the memory of the ISI channel. For the discrete-time EPR4 channel model, the corresponding Viterbi detector has an 8-state trellis. For more details on Viterbi detection, the reader is referred to [44], [5].

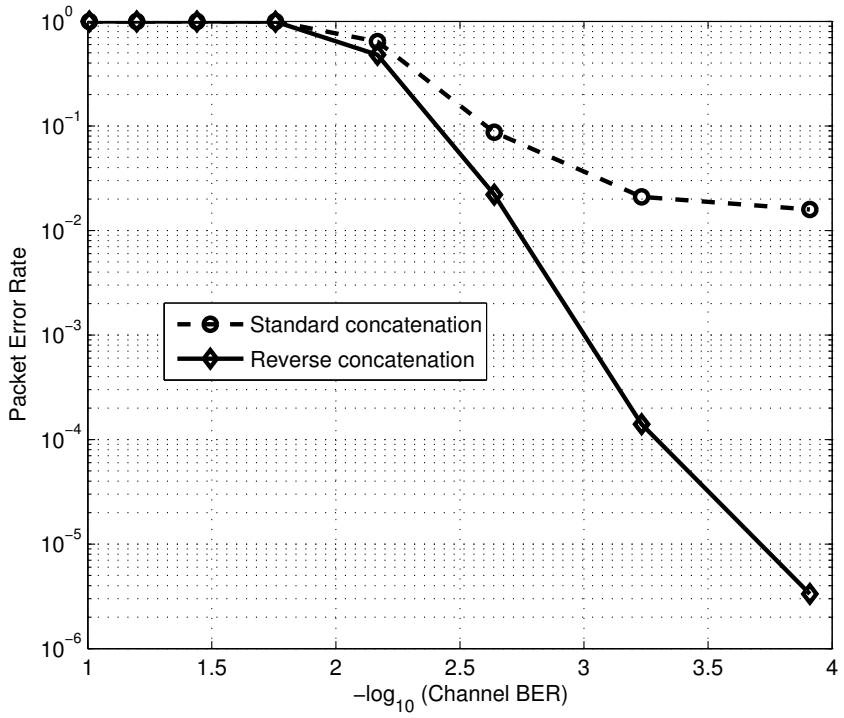

Figure 21: Comparison of packet error rates between standard concatenation (PERsc) and reverse concatenation (PERrc). A packet is declared to be in error if one/more bits in the packet are in error. The packet sizes are 1912 bits for standard concatenation, and 1888 bits for reverse concatenation. At very high channel BER (high noise), there are far too many errors for the RS code to make any difference between PERsc and PERrc. Reverse concatenation gains are seen from channel BER $\sim 10^{-2}$ onwards.

For the ECC, a byte-oriented RS code with bounded-distance decoding [59] capable of correcting at most 8 byte errors, is used in both the standard and reverse concatenation schemes. The constrained encoder is specified by parameters (r, k, m, n), where r denotes the number of pre-processing iterations, m is the fixed input-length in bits and n is the fixed output-length in bits. The parameter values shown in Figs. 19 and 20 generate FRB $(0, k)$ codes with rate close to 100/101. Note that the FRB $(0, k)$ code construction allows flexible parameter choices that can be used to trade

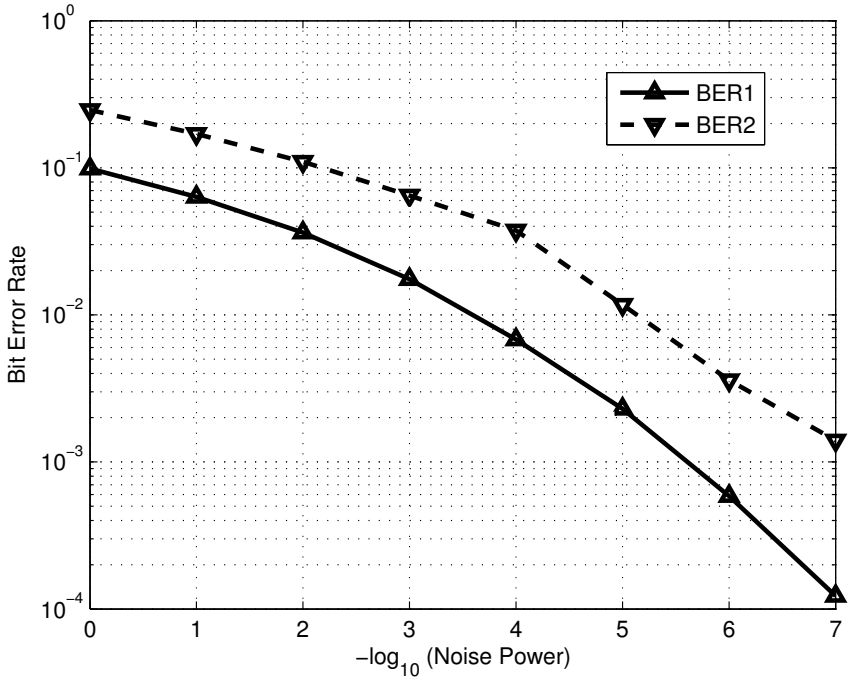

Figure 22: Channel bit error rate (BER1) vs. bit error rate at constrained decoder output (BER2), under standard concatenation. This captures the average effect of error propagation, as can be observed from BER2>BER1. Since the long constrained code is based on bit stuffing with iterative pre-processing, a single channel bit error can cause multiple constrained-decoding errors. Indeed, in the worst scenario, an entire decoded block can be in error due to a single channel bit error.

off encoding/decoding latency for the maximum run-length, and also to choose input/output lengths that better match the byte oriented RS code - all at the cost of very slight variations in rate (details in Chapters 7.4 and 8.1). The $k = 15$ short code shown in Fig. 20 is used only for the RS parity bits, and can be constructed using the combinatorial technique of [20], where error propagation is limited to at most one byte. This code will be of a low rate 42/43 for $k = 15$. The overall encoding rates in our simulations are 0.9264 with standard concatenation, and 0.9241 with reverse concatenation.

The results of our simulation are shown in Figs. 21, 22 and 23. The performance

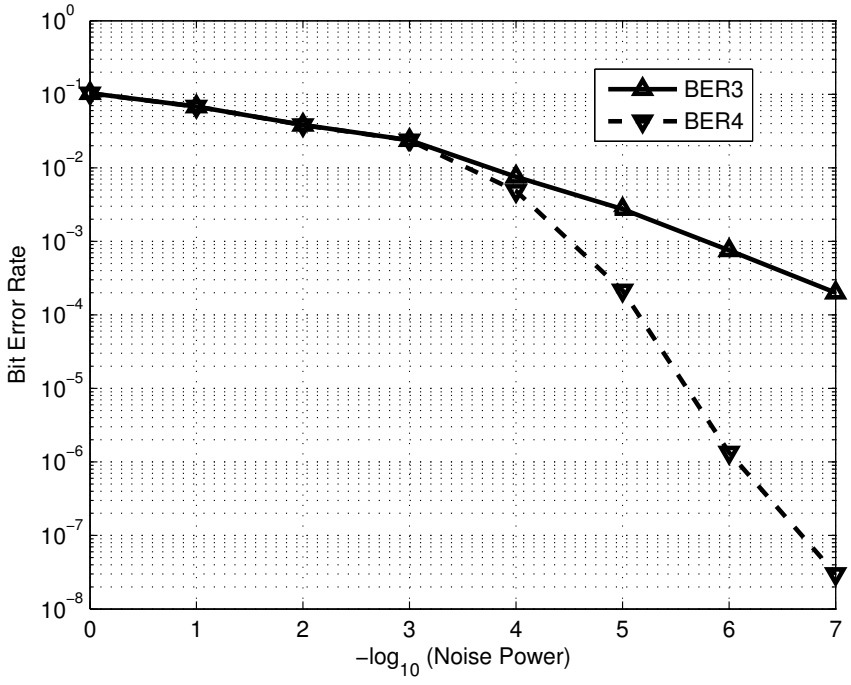

Figure 23: Channel bit error rate (BER3) vs. RS decoder bit error rate (BER4) with reverse-concatenation. The RS decoder corrects some of the channel errors, and hence provides the constrained decoder with a cleaner input, as compared with standard concatenation. Thus the chance of constrained decoder error propagation is reduced by decreasing the error-rate at the constrained decoder input.

improvement with reverse concatenation. is obvious from Fig. 21. It compares the output packet error rates (marked PERsc and PERrc in Figs. 19 and 20, respectively) of the standard and reverse concatenation configurations. The abscissa of Fig. 21 represents decreasing channel bit error rate (BER) values to the right, on a log scale. The channel BERs are the bit error rates at the output of the MLSD. They are marked as BER1 in Fig. 19 for standard concatenation, and BER3 in Fig. 20 for reverse concatenation. For our model the channel BER is independent of the input, and we have that $BER1 = BER3$.

It is seen from Fig. 21 that PERrc shows a steady improvement over PERsc with decreasing channel BER. The reverse-concatenation gains are significant at channel

BERs around 10^{-3}, which are typical of real systems. The main reason for improved PER performance with reverse concatenation is the reduced BER at the constrained decoder input. From Fig. 19, we see that the input BER into the constrained decoder for standard concatenation is the channel BER, namely BER1. However, with reverse concatenation, the BER at the constrained decoder input is BER4, the BER at the output of the RS decoder. The $k = 15$ short code has limited error propagation, and hence limits the bit error rate at the RS decoder input. Thus, the RS code can now correct some of the channel errors before constrained decoding.

When the noise power is very high, there is little difference between BER4 and BER3 (which equals BER1), as the error rate is beyond the correction capability of the $(255, 239)$ RS code. Thus, there are practically no gains at channel BERs higher than 10^{-2} in Fig. 21. However, at lower channel BERs, the RS code corrects some of the channel bit errors, and BER4<BER3, as seen in Fig. 23. This means that there are now fewer erroneous bits at the constrained decoder input with reverse concatenation, *i.e.*, BER4<BER1, and hence there is lesser chance of error propagation with the FRB code. With standard concatenation, the error propagation effects of the FRB code can be seen in Fig. 22.

8.2.2 High-Rate $(0, k)$ Codes with Reduced Error-Propagation

In this section, we describe the design of $(0, k)$ codes that have reduced error-propagation under the standard concatenation configuration, as compared to that of the FRB codes. The principal ideas behind the new codes, referred to as iterative pre-processed (IPP) codes, are as follows

- Insertion/deletion of bits can lead to large propagation of errors, and hence the IPP codes do not use bit stuffing. Only the iterative pre-processing of FRB

codes is retained. Thus, we do not use any variable-rate encoding or dummy-bit padding in the design of IPP codes. As seen in Chapter 7, iterative pre-processing is essential for generating high-rate codes. However, there is a price to pay for the elimination of bit stuffing. There is now a limit on the maximum allowable input block length for each maximum run-length parameter k, and hence, for any given k, the achievable rates are lower than that of the FRB codes. However, the IPP codes have significantly lesser error propagation.

- Iterative pre-processing is also a source of propagating errors. Hence, we process the input data sequence only when necessary. On the other hand, the FRB encoder performs the iterative pre-processing on every incoming data sequence, regardless of the fact that the input sequence may not violate the $(0, k)$ constraint in the first place.

The following discussion provides the code-construction details for a $k = 17$, rate $> 100/101$ code with reduced error-propagation. In general, the IPP codes have encoding rate close to $\frac{\frac{2^k}{k}}{\frac{2^k}{k}+1}$ for a given maximum run-length parameter k.

Let us define $m(r, k)$ to be the maximum input block length, for which the output of the iterative pre-processing with r iterations, has no strings of consecutive "0"s of length greater than or equal to $k + 1$. A lower bound on $m(k, k-1)$ can be obtained from (114), Chapter 7.3.2, since $m(k, k-1)$ is related to the asymptotic rate lower bound $R_l(k)$ as $m(k, k-1) = \frac{R_l(k)}{1-R_l(k)} - 1$. Thus, we have

$$m(k, k-1) \geq \begin{cases} 1 & \text{when } k = 1 \\ 2^{k+1} - 2^{k-1} - 1 & \text{when } k \geq 2. \end{cases} \tag{143}$$

A lower bound on $m(r, k)$ now follows from (119), Chapter 7.4

$$m(r,k) \geq \left(\left\lfloor \frac{k+1}{r} \right\rfloor (m(r,r-1)+1) \right) - 1, \quad r < k+1 \tag{144}$$

Let us consider $k = 17$ with $r = 9$ pre-processing iterations. Using (144), we obtain $m(9,17) = 1535$. Hence, we encode in input blocks of length 1535 bits. Using 9 pre-processing iterations on 1535 bits guarantees that there will be no strings of "0"s of length greater than or equal to 18 in the pre-processed sequence. We append a "1" bit at the end of the pre-processed sequence for merging with neighboring sequences. With 9 pre-processing iterations, the number of index bits is $r = 9$. Once again, we append a "1" bit at the end of each index sequence for merging purposes. Hence, the rate of this code is $\frac{1535}{1536+10} \sim 139/140$.

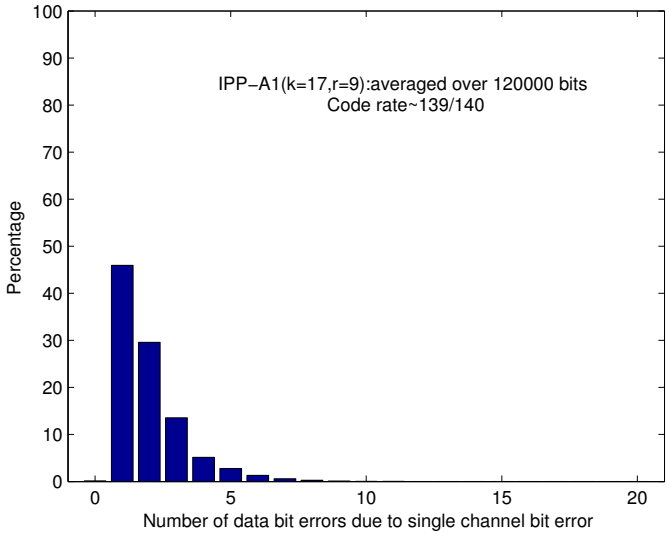

Figure 24: IPP-A1 algorithm

We have thus accomplished our first step: to design a rate$>$ 100/101 code without the bit insertions/deletions of bit stuffing. In doing so, we only used the iterative pre-processing stage of the FRB codes. Next, we formally write down the IPP encoding algorithm, and then proceed to study its error-propagation characteristics.

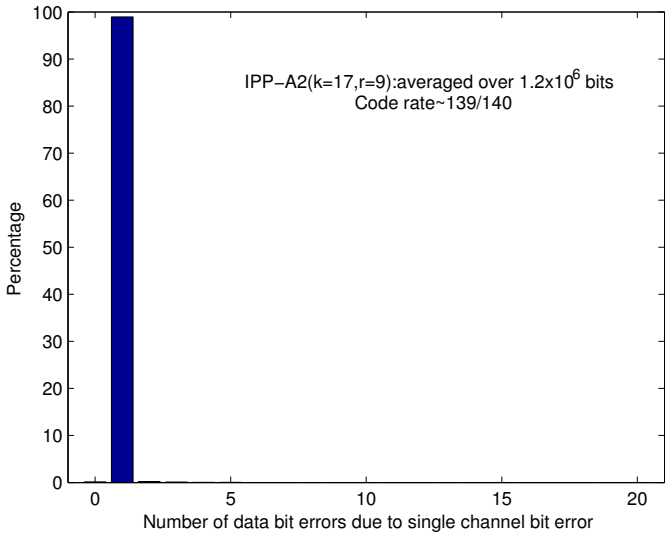

Figure 25: IPP-A2 algorithm

Encoding Algorithm IPP-A1:

Input $\mathbf{x}$

Set $\mathbf{x}_0 = \mathbf{x}$

For $j = 1$ to r

  Input $\mathbf{x}_{j-1}$

  Scan $\mathbf{x}_{j-1}$ as a concatenation of words from the set $\left\{\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{j-1}, \mathbf{u}_{j-1}^*\right\}$

  If $w_{\mathbf{u}_{j-1}}(\mathbf{x}_{j-1}) < w_{\mathbf{u}_{j-1}^*}(\mathbf{x}_{j-1})$

    $\mathbf{x}_j = \overline{\mathbf{x}}_{j-1}(j-1)$

    $\alpha_j = 1$

  Else

    $\mathbf{x}_j = \mathbf{x}_{j-1}$

    $\alpha_j = 0$

  end

end

Pre-processed sequence is $\mathbf{x}_r \parallel 1$

Index sequence is $\boldsymbol{\alpha}_{\mathbf{x}} = (\alpha_1\ \alpha_2\ \ldots\ \alpha_r\ 1)$

Encoded sequence is $\mathbf{x}_r \parallel 1 \parallel \boldsymbol{\alpha}_{\mathbf{x}}$.

Let us now evaluate the error-propagation characteristics of the IPP-A1 codes. We plot the error distribution histogram[1] that shows the distribution of the number of data bit errors due a single channel bit error. A good histogram has a heavy concentration of single data bit errors.

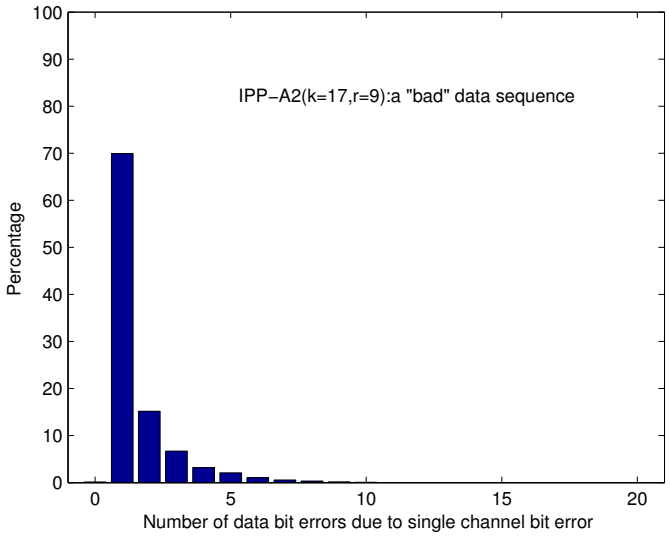

Figure 26: Error distribution due to a certain input sequence with algorithm IPP-A2.

As seen from Fig. 24, the error propagation of IPP-A1 codes is rather high, even though it does not have the insertion/deletion errors of the FRB codes. Hence, we define the following improved algorithm, IPP-A2.

Encoding Algorithm IPP-A2:

Input $\mathbf{x}$;

If $\mathbf{x}$ does not violate the $(0, k)$ constraint

$\mathbf{x}_r = \mathbf{x}$

[1]The author is grateful to Alexander V. Kuznetsov for his suggestions in this regard.

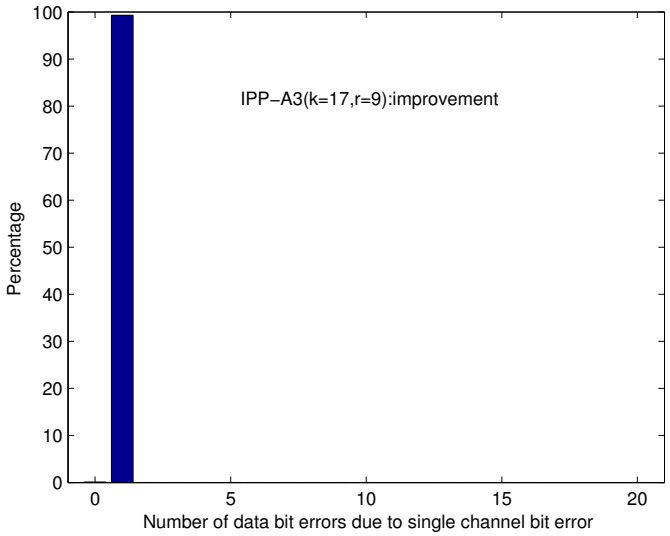

Figure 27: The same sequence as in Fig. 26 has reduced error propagation with algorithm IPP-A3.

$\alpha = (0\ 0\ \ldots\ 0\ 1)$

`Else`

`Set` $\mathbf{x}_0 = \mathbf{x}$

`For` $j = 1$ to r

`Input` $\mathbf{x}_{j-1}$

`Scan` $\mathbf{x}_{j-1}$ `as a concatenation of words from the set` $\left\{\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{j-1}, \mathbf{u}^*_{j-1}\right\}$

`If` $w_{\mathbf{u}_{j-1}}(\mathbf{x}_{j-1}) < w_{\mathbf{u}^*_{j-1}}(\mathbf{x}_{j-1})$

$\mathbf{x}_j = \overline{\mathbf{x}}_{j-1}(j-1)$

$\alpha_j = 1$

`Else`

$\mathbf{x}_j = \mathbf{x}_{j-1}$

$\alpha_j = 0$

`end`

`end`

`Pre-processed sequence is` $\mathbf{x}_r \parallel 1$

`Index sequence is` $\boldsymbol{\alpha}_{\mathbf{x}} = (\alpha_1\ \alpha_2\ \ldots\ \alpha_r\ 1)$

`Encoded sequence is` $\mathbf{x}_r \parallel 1 \parallel \boldsymbol{\alpha}_{\mathbf{x}}$.

Essentially, the IPP-A2 algorithm improves over the IPP-A1 algorithm by processing the incoming data block only if it violates the $(0, k)$ constraint. In the more probable event that the incoming data already satisfies the $(0, k)$ constraint, the data block is transmitted as is, along with the appropriate index sequence (r zeros appended by a "1" merge bit). The probability, $P(V)$, that input data block violates the k constraint can be upper bounded using a simple union bound.

$$P(V) \leq (m-k)(1/2)^{k+1} \tag{145}$$

For $m = 1535$ and $k = 17$, the union bound evaluates to 5.791×10^{-3}. Thus, at most one in 172 input sequences undergoes the iterative pre-processing on an average. Hence, although the worst case error-propagation is unaltered, the average error propagation is substantially improved, as shown in Fig. 25.

A further improvement to algorithm IPP-A2 is possible. One can also check if the flipped data block, $\overline{\mathbf{x}}$, satisfies the $(0, k)$ constraint. If it does, then the sequence $\overline{\mathbf{x}}$ is transmitted along with the appropriate index sequence ($r-1$ zeros followed by a "1" and then appended by a "1" merge bit). Clearly, a single bit error in $\overline{\mathbf{x}}$ is recoverable as a single data bit error. Hence, IPP-A3 further improves the performance of IPP-A2. This is illustrated in Figs. 26 and 27, where a certain input sequence undergoes pre-processing with algorithm IPP-A2 (which leads to some propagation of errors), but since $\overline{\mathbf{x}}$ satisfies the $(0, 17)$ constraint, it does not have to go through the rest of the pre-processing in IPP-A3, thereby reducing error propagation.

In fact, it can be shown that in the maximum number of data bit errors due to a single channel bit error in the encoded sequence with r pre-processing iterations for

a maximum run-length parameter k, is given by[2]

$$E(r,k) \leq \max_{x \in \{2,3,\ldots,r\}} 2x + (x+1)\left\lfloor \frac{m(r,k)}{\lfloor \frac{x+1}{x} \rfloor m(x,x-1)} \right\rfloor + 1, \tag{146}$$

where $m(r,k)$ is as specified in (144). The proof of inequality (146) is beyond our present scope. We are more concerned with a union bound on the probability, $P(E)$, that the input sequence undergoes multiple pre-processing iterations with IPP-A3, and hence, in the worst case, is liable to the worst-case error propagation as in (146). This is given by

$$P(E) \leq (m-k)(m-2k-1)(1/2)^{2k+2}. \tag{147}$$

For $m = 1535$ and $k = 17$, the union bound evaluates to 3.313×10^{-5}. Thus, on the average, at most one in 30179 input sequences is susceptible to the worst-case error propagation of $E(9,17)$ data bits due to a single channel bit error.

The average error propagation for the IPP-A3 algorithm is shown in Fig. 28 on the left. Note the small increase in the percentage of single data bit errors, as compared to Fig. 25. Finally, we compare the error distribution histogram of the rate $139/140, k = 17$, IPP-A3 code with that of the corresponding $k = 17$ combinatorial code [20] of rate 48/49, as shown in Fig. 28.

Encoding Algorithm IPP-A3:

`Input x;`

`If x does not violate the (0,k) constraint`

[2]It is assumed that the index bits are coded to correct single errors, as mentioned at the end of this subsection. Indeed, if the index bits were left uncoded, then the worst case error propagation would lead to all data bits being erroneous as a result of a single index bit error.

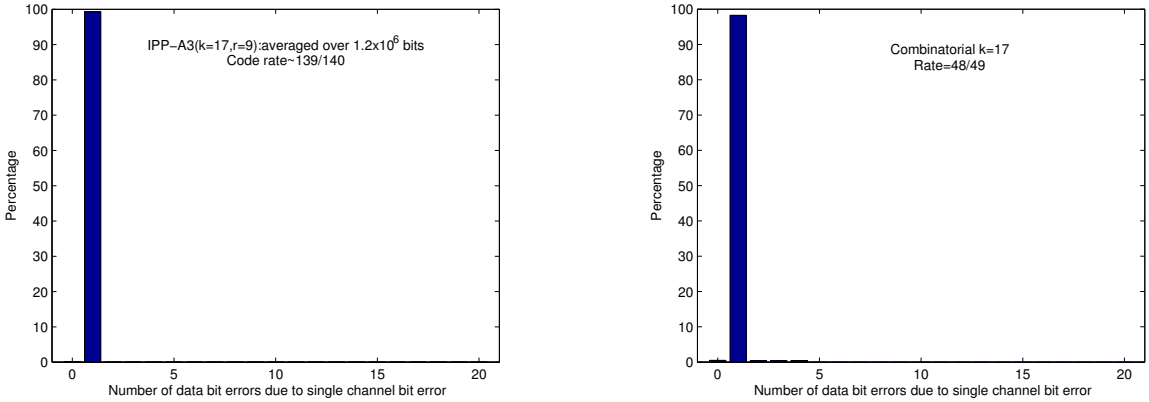

Figure 28: Comparison of error propagation characteristics of algorithm IPP-A3 and the combinatorial construction of Immink and Wijngaarden.

$\mathbf{x}_r = \mathbf{x}$

$\alpha = (0\ 0\ \ldots\ 0\ 1)$

`Elseif`

`If` $\overline{\mathbf{x}}$ `does not violate the` $(0, k)$ `constraint`

$\mathbf{x}_r = \overline{\mathbf{x}}$

$\alpha = (0\ 0\ \ldots\ 1\ 1)$

`Else`

`Set` $\mathbf{x}_0 = \mathbf{x}$

`For` $j = 1$ to r

`Input` $\mathbf{x}_{j-1}$

`Scan` $\mathbf{x}_{j-1}$ `as a concatenation of words from the set` $\left\{\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{j-1}, \mathbf{u}^*_{j-1}\right\}$

`If` $w_{\mathbf{u}_{j-1}}(\mathbf{x}_{j-1}) < w_{\mathbf{u}^*_{j-1}}(\mathbf{x}_{j-1})$

$\mathbf{x}_j = \overline{\mathbf{x}}_{j-1}(j-1)$

$\alpha_j = 1$

`Else`

$\mathbf{x}_j = \mathbf{x}_{j-1}$

$\alpha_j = 0$

end

end

Pre-processed sequence is $\mathbf{x}_r \parallel 1$

Index sequence is $\boldsymbol{\alpha}_{\mathbf{x}} = (\alpha_1\ \alpha_2\ \ldots\ \alpha_r\ 1)$

Encoded sequence is $\mathbf{x}_r \parallel 1 \parallel \boldsymbol{\alpha}_{\mathbf{x}}$.

Further improvements over IPP-A3 may be possible by continually checking for violations with each pre-processing iteration so that the next iteration is carried out only if necessary. However, we found that such improvements are too small to be interesting.

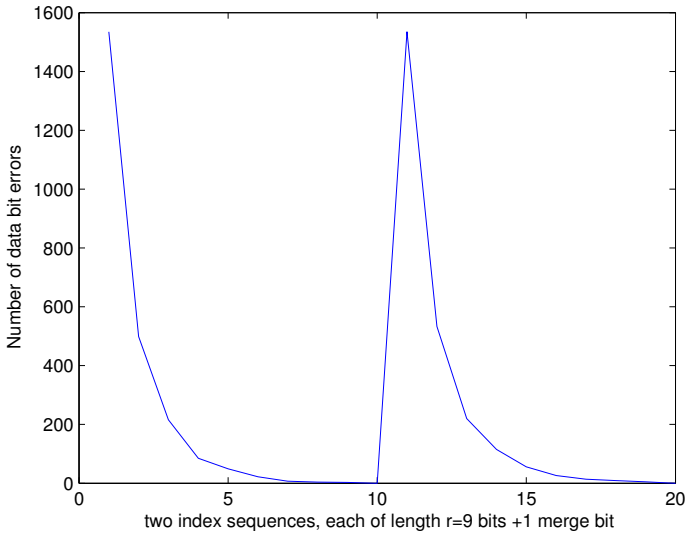

Figure 29: Error propagation due to a single index-bit error. It is possible that the entire data block (of length 1535 bits) is in error.

One issue we have not addressed in our discussion thus far, is that of error propagation due to index-bit errors. As shown in Fig. 29, a single index-bit error can propagate through to several data bits. From the IPP-A3 histogram in Fig. 28, the percentage of the index-bit errors is rather small ($< 0.5\%$), but the large number of

resulting errors could alter performance. In such a case, we propose to additionally encode the index bits alone, so as to correct single errors. Since we have $r = 9$ index bits, by using a $(13, 9)$ shortened Hamming/BCH code, we can correct a single index-bit error. Once again, merge bits can be used appropriately to prevent violation of the $(0, 17)$ constraint. This reduces the code rate from $139/140$ to $102/103$.

8.3 DC Suppression

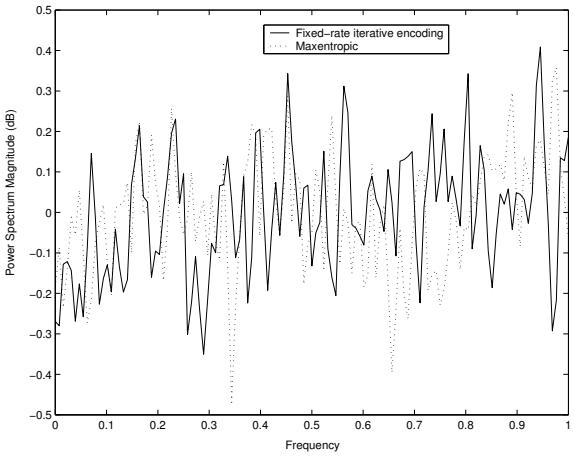

Figure 30: PSD of the $(0, 9)$ codewords generated by the FRB algorithm, as compared to that of the maxentropic sequence. The two are very similar.

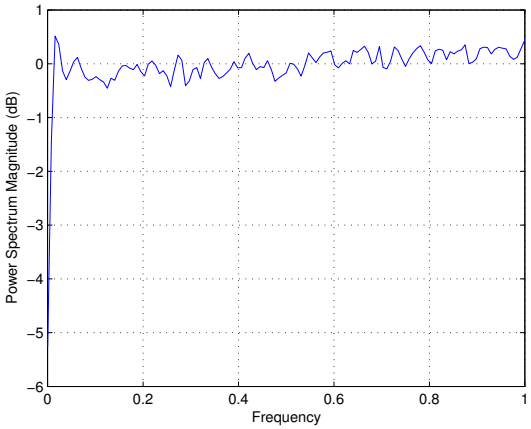

Figure 31: DC suppression of $(0, 9)$ FRB codes using a rate $23/24$ adapted polarity bit scheme.

DC suppression refers to the reduction in low-frequency code spectrum content. Figure 8.3 shows the power spectral density (PSD) of the FRB $(0, 9)$ codewords as compared to the maxentropic PSD. Since the FRB algorithm generates near-maxentropic $(0, k)$ sequences, we see that their PSDs are similar. Specifically, there is high DC content, which can lead to write-signal distortion in magnetic-tape systems that use rotary-type recording heads. Another example is in optical recording, where DC-free codes that have zero DC content are desired since servo information is stored at the low frequencies. The use of DC-free codes also enables filtering of low-frequency

noise arising from the wear and tear of the disk surface. Hence, DC-free codes have been the subject of much attention in recording literature, a summary of which can be found in [[17],Chap.10] and [18].

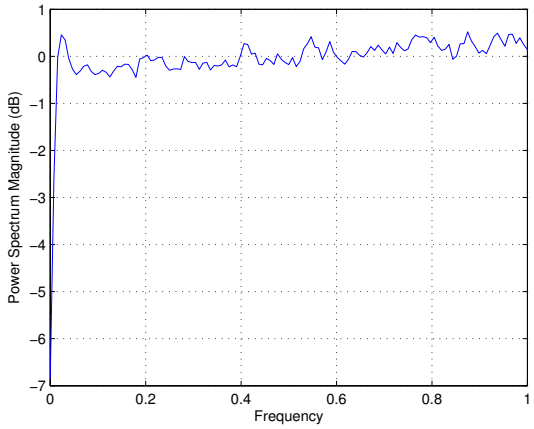

Figure 32: DC suppression of $(0, 9)$ FRB codes using a rate $16/17$ adapted polarity bit scheme.

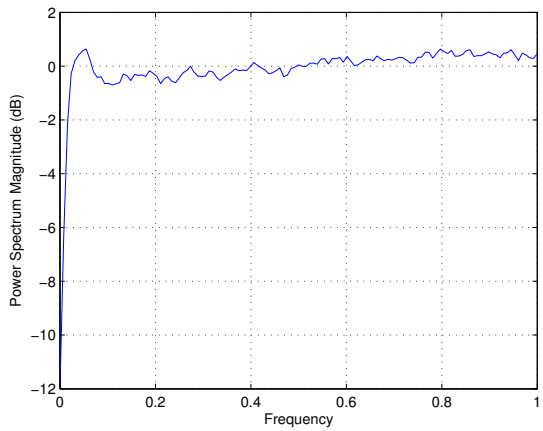

Figure 33: DC suppression of $(0, 9)$ FRB codes using a rate $8/9$ adapted polarity bit scheme.

However, the design of near-capacity DC-free RLL codes: codes that satisfy both the RLL constraints and the running digital sum (RDS) constraints is severely hampered by the large number of states in the underlying finite state transition diagram (FSTD). Here, we use a simple ad-hoc method to obtain suppression of DC content. Figure 31 shows the DC-suppression obtained with a simple polarity bit scheme of rate 23/24, over and above the $(0, 9)$ FRB code. The polarity bit scheme used was adapted from the one proposed by Bowers and Carter (see [[17],Chap.10] for a description) to incorporate the maximum run-length constraint. Specifically, we use a window size of 46 bits and two supplementary bits: one to indicate the polarity, and the other to limit the maximum run-length. The rate of the adapted polarity bit scheme is hence $46/48 = 23/24$. The adapted scheme increases the maximum run-length from k to $k + 1$. Hence, by using an $m = 2000$, $k = 9$ FRB code in conjunction with the polarity bit scheme, we can obtain a rate $\sim 20/21$, $k = 10$ code

with $\sim 6dB$ DC-suppression. Better low-frequency suppression can be obtained at a reduced rate, as shown in Figs. 32 and 33. Here, the adapted polarity bit scheme is used with reduced window sizes of 32 and 16 bits, respectively. This limits the RDS to lower values thereby improving low-frequency suppression. A comparison of the RDS of the FRB codes to that of a rate 16/17 DC-suppressed FRB codes is shown in Fig. 34.

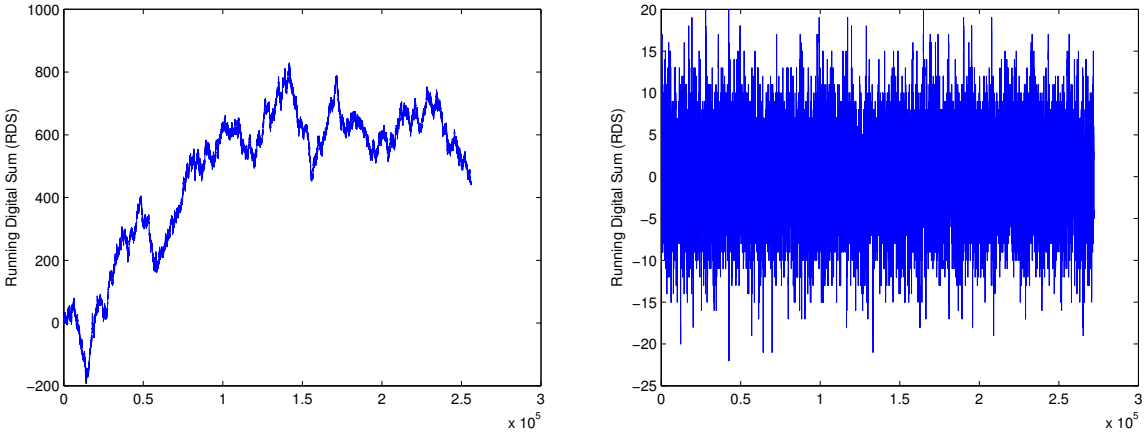

Figure 34: Comparison of the running digital sum (RDS) of the $(0, 9)$ FRB codewords and the rate $16/17$ DC-suppressed FRB codewords.

8.4 Implementation Complexity of the FRB algorithm

The FRB algorithm generates high-rate $(0, k)$ codes by using iterative pre-processing followed by bit stuffing. As seen in Chapter 7, the bit stuff operations are very simple, and only involve scanning of the pre-processed sequence, counting the run-length of consecutive "0"s, and then inserting bits if there is an impending constraint violation. Let us assume that the number of bit-level computations required for the counting and bit insertion, each grows linearly with the input block length m. Then, we say that bit stuffing requires $\sim 2m$ computations on the whole. This must be compared with look-up table based encoding which requires $\sim 2^m$ computations. In what follows, we

discuss the implementation complexity of the iterative pre-processing, and show that the required number of computations, once again grows only linearly in m.

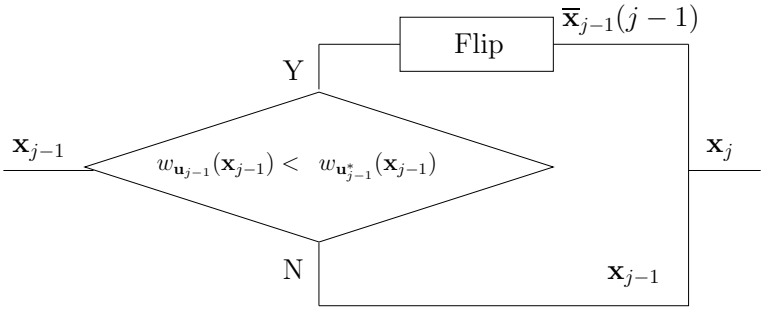

Figure 35: Serial implementation of pre-processing iteration j. Here, the input $\mathbf{x}_{j-1}$ is first scanned to determine if $w_{\mathbf{u}_{j-1}}(\mathbf{x}_{j-1}) < w_{\mathbf{u}^*_{j-1}}(\mathbf{x}_{j-1})$, and then the appropriate branch is taken.

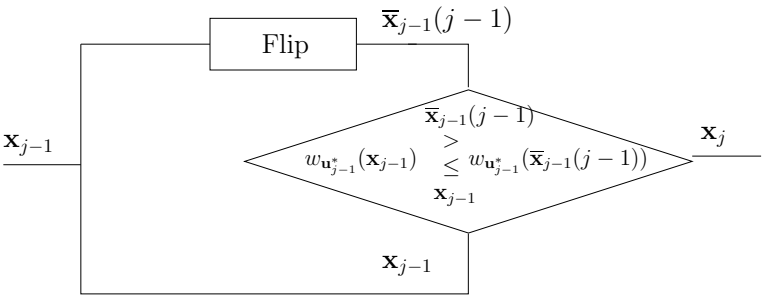

Figure 36: Parallel implementation of pre-processing iteration j. Here the decision is made after the input is processed in the upper and lower branches.

Recall from Chapter 7 that the pre-processing involves k iterations of scanning, parsing and selective inversion. Two possible implementations of a pre-processing iteration are shown in Fig. 35 and 36. The block diagrams depict iteration j, with the "Flip" block performing the selective inversion.

Fig. 35 shows a serial implementation similar to that of Fig. 12, Chapter 7.1. Here, the input $\mathbf{x}_{j-1}$ is first scanned to determine if $w_{\mathbf{u}_{j-1}}(\mathbf{x}_{j-1}) < w_{\mathbf{u}^*_{j-1}}(\mathbf{x}_{j-1})$, and then either the upper or the lower branch is taken depending on the outcome. Let us

compute the average number of computations and latency with such an implementation. In order to determine if $w_{\mathbf{u}_{j-1}}(\mathbf{x}_{j-1}) < w_{\mathbf{u}^*_{j-1}}(\mathbf{x}_{j-1})$, the required parsing and counting each take $\sim m$ computations. If the upper branch is taken, further parsing and selective inversion, each require $\sim m$ computations. Thus, if the probability of taking the upper branch is q, the average number of required computations with the serial implementation is $\sim 2m(1+p)$, still linear in m. The average latency for the serial implementation can be similarly computed as $\sim m(1+p)$, assuming that most of the delay is incurred in scanning.

In contrast, the number of computations and latency for the parallel implementation shown in Fig. 36, are $\sim 3m$ and $\sim m$, respectively. The difference arises because the parallel implementation first duplicates the input, and performs the inversion operation, $\overline{\mathbf{x}}_{j-1}(j-1)$, in the upper branch. This involves $2m$ computations. The lower branch allows $\mathbf{x}_{j-1}$ to pass through as is. The decision box then picks exactly one of $\overline{\mathbf{x}}_{j-1}(j-1)$ or $\mathbf{x}_{j-1}$, depending on which sequence has the least number of $\mathbf{u}^*_{j-1}$ words. However, the counting operations to determine if $w_{\mathbf{u}^*_{j-1}}(\mathbf{x}_{j-1}) > w_{\mathbf{u}^*_{j-1}}(\overline{\mathbf{x}}_{j-1}(j-1))$ can be done simultaneously on the upper branch, rather than scan the entire sequence again. Hence the total number of required computations is $\sim 3m$. Because of the parallel structure, scanning is performed only once and hence the latency is lesser than the average latency of the serial implementation.

Thus, we see that each iteration can be implemented in either serial or parallel fashion. While the parallel implementation has lower latency, the serial implementation performs fewer computations on the average if $p < 0.5$. Indeed, in our simulations, we have found that $p \in (0.3, 0.4)$, and hence there is a trade-off between the latency and computations required in the serial and parallel implementations. The entire set of k iterations can be implemented by serially cascading the individual iteration blocks. Thus, the total number of required computations (including bit insertions) for FRB encoding is $\sim 2mk(1+p) + 2m$ with serial implementation, and

$\sim 3mk + 2m$ with the parallel implementation. Very recently, we have devised a completely parallel implementation of the entire set of k pre-processing iterations, which requires $\sim 2^{k+1}m(k+1)$ computations.

8.5 Comparison with Existing Schemes

Recall that for $(0,k)$ constraints, $k \geq 5$, the capacity $C(0,k)$ is well-approximated by

$$C(0,k) \simeq \frac{2^{k+2}\ln 2 - 1}{2^{k+2}\ln 2}. \tag{148}$$

Hence, the design of $(0,k)$ codes with rate $(n-1)/n$, for large integers n (preferably close to $2^{k+2}\ln 2$), is of considerable theoretical interest. From a practical view-point, the design of high-rate, fixed-rate $(0,k)$ codes is interesting if the encoding/decoding is simple. This motivated us to pursue the design of a fixed-rate version of the simple, variable-rate bit stuff algorithm in Chapter 7. The resultant fixed-rate bit stuff (FRB) codes were shown to be asymptotically very efficient, and the required code parameters for the design of rate 100/101 and rate 200/201 $(0,k)$ codes were tabulated in Chapter 8.1. In the following discussion, we present a comparitive study of the FRB algorithm with two other existing methods of designing high-rate $(0,k)$ codes: enumerative coding [15],[[17],Chap.6],[19], and combinatorial coding [20].

Table 26: Performance comparison: FRB *vs.* Enumeration *vs.* Combinatorial

	Fixed-rate Bit Stuff	**Enumeration**	**Combinatorial [20]**
Encoding Rate	Near-capacity	Near-capacity	$\frac{n-1}{n}, n \sim 3k$
Storage	–	$O(m^2)$	–
Computation	$O(m)$	$O(m^2)$	$O(1)$
Error Propagation	Yes	Yes	No

A summary of the performance comparison is given in Table 26. Let us first look at the encoding rates, which are shown in greater detail in Fig. 37. The ordinate

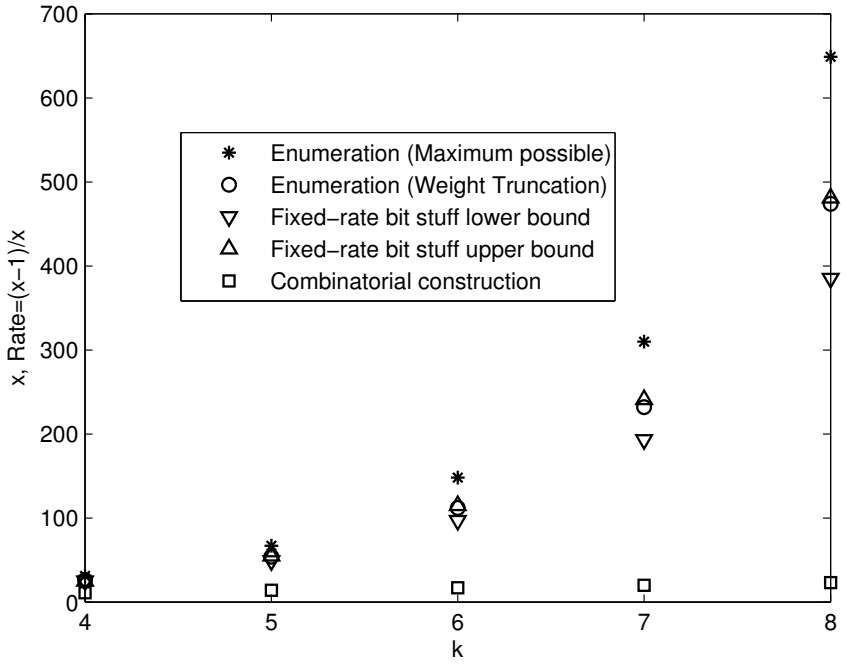

Figure 37: Encoding rates of enumerative, FRB and combinatorial $(0, k)$ codes for $4 \leq k \leq 8$. The ordinate shows the value of x, where the encoding rate is given by $\frac{x-1}{x}$. Note that x may or may not be equal to the output block length n, depending on the choice of encoding scheme.

shows the values of x, where the encoding rate is given by $\frac{x-1}{x}$. In the case of the combinatorial code [20], x is equal to the output block length n, which only grows linearly with k. Indeed, for any given n, n odd, the maximum run-length of the combinatorial code is $k = 1 + \lfloor n/3 \rfloor$. In contrast, the maximum possible value of x grows exponentially with k, as seen from equation (148). Figure 37 shows that the best possible enumeration codes (taken from Table 6.4, [17]) get quite close to this exponential scaling. The asymptotic upper and lower bounds with the FRB algorithm are poorer than the best enumeration codes, but are in close agreement with weight truncation based enumerative coding (taken from Table 6.5, [17]). However, both FRB and enumerative coding achieve significantly higher encoding rates compared to combinatorial $(0, k)$ codes. A more detailed comparison of the FRB partial rates to

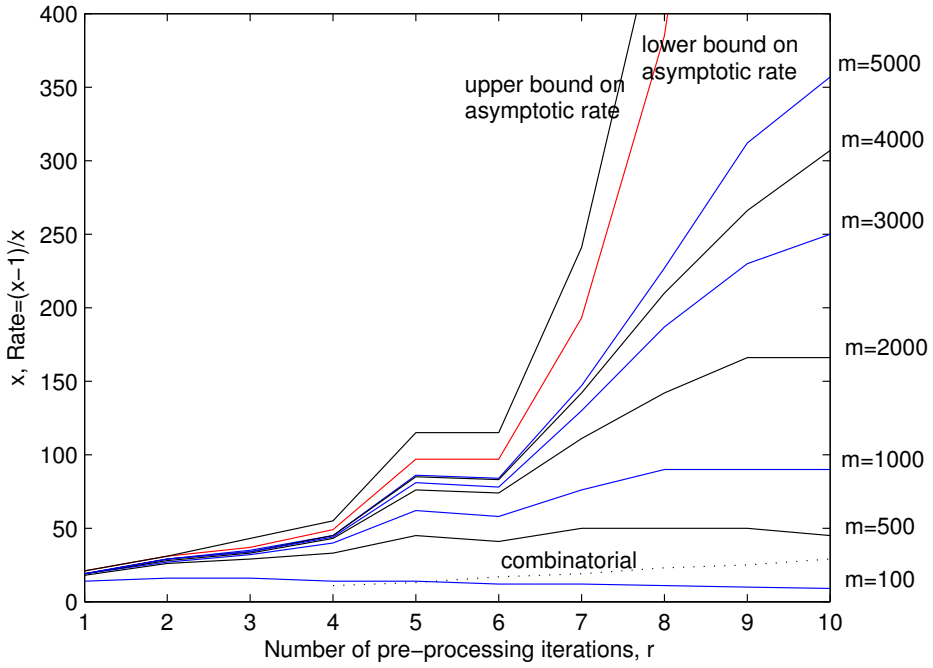

Figure 38: Partial encoding rates of the FRB algorithm, as compared to the combinatorial code (shown in the dotted line). The ordinate shows the value of x, where the encoding rate is given by $\frac{x-1}{x}$. The abscissa shows the number of pre-processing iterations $r = 1$ through 9 for the FRB algorithm for $k = 10$. For the combinatorial code, the abscissa denotes the value of k. The values of x for the asymptotic upper and lower bounds on the partial encoding rates are shown, along with the partial encoding rates for several finite block lengths m. Clearly, higher encoding rates are achievable using the FRB algorithm.

that of the combinatorial code is shown in Fig. 38.

One advantage that the combinatorial code does have over the others, is that of simpler implementation. At most eight bits need to be altered during the encoding, regardless of the input block length. Moreover, a single channel bit error propagates through to at most one data byte during decoding. In contrast, FRB and enumeration codes are prone to error propagation effects, whereby several data bits can be in error due to a single channel bit error. One possible solution is offered by the IPP codes, which were discussed in Chapter 8.2. The IPP codes have lower encoding rates

compared to the the FRB codes, but they have reduced error propagation.

Finally, we compare the implementation complexities of enumerative coding and the FRB algorithm. Enumeration is based on computations using pre-stored weighting coefficients, and hence there is a storage cost involved. For a desired input block length m, a fixed-point representation of the weights requires storage of $O(m^2)$. This can be reduced to $O(m \log m)$ using a floating-point representation. In the case of FRB codes, there are no such pre-computational storage requirements. Furthermore, we have already seen in Chapter 8.4 that the number of computations required to execute the FRB algorithm is $O(m)$, *i.e.*, grows only linearly with input block length m. On the other hand, enumerative coding using fixed-point arithmetic requires $O(m^2)$ computations, which can be prohibitive for large m required in the design of very high-rate codes.

CHAPTER IX

CONCLUSIONS AND FUTURE RESEARCH

9.1 Conclusions

Despite its long history, the design of near-capacity run-length-limited (RLL) codes continues to merit significant attention. Even small improvements (1-2%) in the encoding rate are much sought after in today's digital recording systems. Rather than directly increase the storage density, such small rate increments are beneficial in improving manufacturing tolerances and system margins. As an example of this, current industry-standard $(0, k)$ codes used in magnetic recording systems are of rate 8/9, 16/17 and 64/65; and there is considerable effort being expended to design rate 100/101 and rate 200/201 codes, which is only a 1% increase in rate and density, but can substantially impact the cost and performance of a manufactured disk drive. Thus, there continues to be a need for low-complexity algorithms that achieve higher encoding rates. Furthermore, there is the purely theoretical interest of achieving the constraint capacity.

These factors motivated us to pursue the design of new capacity-approaching coding methods in this research. The proposed algorithms are based on a very simple technique called bit stuffing. Building on the existing bit stuff [6] and bit flipping [3] algorithms for (d, k) constraints, we introduced the symbol sliding algorithm in Chapter 4. Symbol sliding is a variable-rate encoding algorithm that achieves capacity for $(d, 2d{+}1)$, (d, ∞), $(d, d{+}1)$, and $(2, 4)$ constraints, and comes very close to capacity for all other values of d and k. In Chapter 5, we introduced another variable-rate code construction based on interleaving, which was capacity-achieving for all (d, k) constraints with $k - d + 1$ not prime. We also discussed a new, near-capacity bit stuff

algorithm for $(0, G/I)$ constraints, and presented capacity-achieving, variable-rate encoding algorithms for asymmetrical run-length constraints and multiple-spacing (d, k) constraints in Chapter 6.

While the aforementioned variable-rate codes are capacity-approaching, they are of limited practical use in current digital recording systems, which require fixed-rate encoding. In Chapter 7, we derived a fixed-rate version of the variable-rate bit stuff algorithm for the special class of $(0, k)$ constraints. To the best of our knowledge, this is the first attempt at creating a fixed-rate code using bit stuffing. To achieve high encoding rates, the proposed fixed-rate bit stuff (FRB) algorithm used k iterations of pre-processing prior to bit stuffing. The iterative pre-processing ideas were also extended to build fixed-rate $(0, G/I)$ codes in Chapter 7.6.

We presented a detailed rate analysis of the FRB algorithm in Chapter 7.3, and computed upper and lower bounds on the asymptotic (in input block length) encoding rate. Our results suggest that near-capacity $(0, k)$ codes can be constructed using the FRB algorithm, by encoding in very long, fixed-length input and output blocks. In Chapter 8.1, we listed the FRB code parameters required for the design of rate 100/101 and rate 200/201 $(0, k)$ codes.

Two important existing methods to generate $(0, k)$ sequences use enumerative [[17],Chap.6],[19], and combinatorial [20],[60] encoding. The FRB encoding/decoding is simpler than enumeration, while achieving (asymptotically) similarly high encoding rates. The FRB encoding rates are also far greater than that of the combinatorial construction of Immink and Wijngaarden [20], at the cost of slightly higher encoding/decoding complexity.

Thus, in theory, the FRB algorithm provides an effective means to generate very high-rate $(0, k)$ sequences. However, there are several practical issues to be addressed while incorporating the FRB codes into a recording system. For large values of k,

running k pre-processing iterations could lead to excessive encoding delay. This motivated us to study the encoding rates with partial pre-processing in Chapter 7.4. Encoding in very long input and output blocks also raises the possibility of error propagation. With this in mind, we studied the performance of FRB codes under reverse concatenation using a simple magnetic recording channel model. We also described in Chapter 8.2, the construction of a rate 102/103, $(0, 17)$ iterative pre-processed (IPP) code, which trades off encoding rate for reduced error propagation. In general, the IPP codes have lower encoding rates ($\sim \frac{\frac{2^k}{k}}{\frac{2^k}{k}+1}$) as compared to the FRB codes (asymptotically $\sim \frac{2^k}{2^k+1}$), but they have reduced error propagation. Finally, in Chapters 8.4 and 8.5, we discussed the implementation complexity of the FRB algorithm, and presented performance comparisons with enumerative and combinatorial encoding.

9.2 Future Research

The symbol sliding and interleaving algorithms, presented in Chapters 4 and 5, both rely on a phrase interpretation of (d, k) sequences. A similar phrase interpretation applies to several other RLL constraints, in which case we can find extensions of the proposed algorithms. Specifically, we presented extensions of the interleaving construction to asymmetrical run-length constraints and multiple-spacing (d, k) constraints in Chapter 6. Another example of a constraint that admits a phrase interpretation is the M-ary (d, k) or (M, d, k) constraint [33],[34],[35],[36],[32], $M > 2$. It would be interesting to explore the applicability of symbol sliding and interleaving algorithms for (M, d, k) constraints. Since symbol sliding derives its optimality property from a capacity equality, determining (M, d, k) capacity equivalences [48] becomes an important sub-problem.

We also note that the optimality of the symbol sliding and interleaving algorithms have different origins. Symbol sliding is optimal for $(d, 2d+1)$ constraints only because

of the binary capacity equality $C(d, 2d{+}1) = C(d{+}1, \infty)$, and the fact that bit stuffing on a biased bit stream achieves $(d+1, \infty)$ capacity. With interleaving, the proof of optimality lies in the factorization of characteristic (d, k) polynomials. Hence, with the two different origins of optimality, we believe that further improvements might be possible with a combination of symbol sliding and interleaving. Interestingly, a very recent work by Aviran *et al.* [4] has already built upon symbol sliding, by considering other input word-parsing assignments rather than just the bit stuff word-parsing.

Bit stuffing, symbol sliding and interleaving code constructions, all rely on a distribution transformer (DT) to generate the biased bit sequence. In prior work, arithmetic coding has been considered as a means of implementing such a distribution conversion [23],[45],[31],[58],[2],[41]. However, in the context of bit stuffing, precise implementation algorithms for finite block length DTs, along with computation of the resulting encoding rates could be quite insightful. The design and analysis of fixed-rate DTs is also potentially valuable.

The fixed-rate bit stuff (FRB) algorithm, presented in Chapter 7 enables the construction of very high-rate $(0, k)$ codes by using a cascade of pre-processing and bit stuffing. The pre-processing involves k iterations, which were assumed to be executed serially. An equivalent parallel structure conducive to hardware implementation could be of considerable interest.

A potential weak link in the performance of the FRB codes lies in error propagation. Thus, further analysis on the error propagation of iterative pre-processed (IPP) codes is important. More significantly, we had used a rather simple model of the magnetic recording channel in our simulations in Chapter 8.2. For high-density recording, more accurate channel models incorporate a signal-dependent media noise component [28]. The performance of FRB and IPP codes under these models remains to be studied.

The design of very high-rate constrained codes that admit iterative, soft-decision

decoding [5],[44] could be quite valuable. Thus, the design of combined error control and constrained codes is of considerable interest.

Finally, we mention that the iterative pre-processing technique could be extended to other RLL constraints. Specifically, for $(0, G/I)$ constraints, we showed in Chapter 7.6 that using iterative pre-processing on each of the even and odd subsequences can lead to very high-rate codes. Further research needs to be carried out to determine the appropriate pre-processing structure for other important constraints like maximum-transition-run (MTR) constraints, $(1, k)$, $(2, k)$ constraints and in general, constraints that prohibit specific difference sequences [37].

APPENDIX A

PROPERTIES OF $\theta(k, m)$

For a given maximum-run-length parameter $k < \infty$, the function $\theta(k, m)$ is given by

$$\theta(k, m) = \frac{m}{\beta(k, m)}, \quad m \geq m_0 \tag{149}$$

where $\beta(k, m) = \max_{\mathbf{x} \in \mathbb{Z}_2^m} \left\{ w_{\mathbf{u}_{k-1}^*}(\mathbf{x}_k) \right\}$, and m_0 is the smallest positive integer such that $\beta(k, m) > 0$, for all $m \geq m_0$.

First, we note that $\beta(k, m)$ is a non-decreasing function of m. Let us define $\mathbf{x}' = \arg\max_{\mathbf{x} \in \mathbb{Z}_2^m} w_{\mathbf{u}_{k-1}^*}(\mathbf{x}_k)$. While $\mathbf{x}'$ may not be unique, it can be seen there exists at least one $\mathbf{x}'$ such that $w_{\mathbf{u}_{k-1}^*}(1 \parallel \mathbf{x}') = \beta(k, m)$ for any given k and m. Here, the sequence $1 \parallel \mathbf{x}'$ is of length $m + 1$ bits. Thus, $\beta(k, m+1) = \max_{\mathbf{x} \in \mathbb{Z}_2^{m+1}} w_{\mathbf{u}_{k-1}^*}(\mathbf{x}_k) \geq \beta(k, m)$, which proves the monotonicity of $\beta(k, m)$. Furthermore, $\beta(k, m)$ is unbounded, that is, $\beta(k, m) \to \infty$ as $m \to \infty$.

Although $\beta(k, m)$ is monotonic, the function $\theta(k, m) = m/\beta(k, m)$ need not be monotonic in general. To see this, consider the sequence $\{\theta(k, m)\}_{m=m_0}^{\infty}$. Let us define $\mathcal{S}_j = \{m : \beta(k, m) = j + 1\}$ and $m_j = \min\{\mathcal{S}_j\}$, for all non-negative integers j. This is illustrated in Fig. 39. Essentially, we are partitioning the entire range of values of block length $m \geq m_0$, into several bins, each bin corresponding to a unique value of the function $\beta(.)$.

The size of bin $j + 1$ is $|\mathcal{S}_j|$, which gives us the increase in block length that is necessary for a corresponding increment in $\beta(.)$ from $j + 1$ to $j + 2$. For each j, the bin-size $|\mathcal{S}_j|$ is finite, and can be bounded as $k \leq |\mathcal{S}_j| \leq 2^{k+1} - 2$. The lower bound

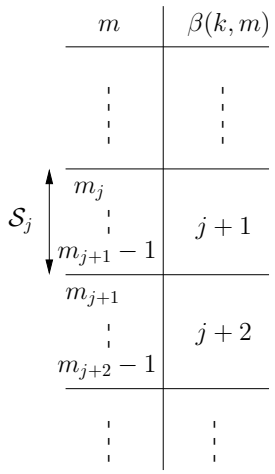

Figure 39: Since the function $\beta(k,m)$ is non-decreasing, we can partition the entire range of input block lengths m, into bins as shown above. Each bin is associated with a unique value of $\beta(.)$.

is obtained from plain bit stuffing. Since every $\mathbf{u}_{k-1}^*$ word is of length k bits, we have that $|\mathcal{S}_j| \geq k$. The upper bound follows from variable-rate bit stuffing, which represents the best one can do with the fixed-rate code. From (86) we have that $|\mathcal{S}_j| \leq 2^{k+1} - 2$.

Fig. 39 helps us understand the properties of the function $\theta(k,m) = \frac{m}{\beta(k,m)}$. $\theta(k,m)$ attains local minimum values at $m = m_j$, and then follows a linear increase from $m = m_j$ up until $m = m_{j+1} - 1$, which is a point of local maximum, before dropping down again at $m = m_{j+1}$. This gives $\theta(k,m)$ a resemblance to a saw-toothed waveform. Since $|\mathcal{S}_j|$ is bounded above, the "saw-teeth" become flatter with increasing j. Our aim now is to study the convergence of $\theta(k,m)$.

First consider the case when $k = 1$. By definition, m_j is the smallest block length m, such that $\beta(1,m) = j + 1$. This means that the sequence $\mathbf{x}_1$ has $j + 1$ "0"s. In such a case, the iterative pre-processing ensures that there are also at least $j+1$ "1"s in $\mathbf{x}_1$. Hence, $m_j = 2(j + 1)$, and $|\mathcal{S}_j| = 2$. This implies that $\beta(1,m) = \lfloor \frac{m}{2} \rfloor$, and hence $\theta(1,m) = \frac{m}{\lfloor \frac{m}{2} \rfloor}$ converges as $m \to \infty$.

Next, we consider $k = 2$. By backtracking through pre-processing iteration 2, we

find that the sequence $\mathbf{x}_1$ contains at least three "0" bits for every 0^2 word in $\mathbf{x}_2$ (with $\mathbf{x}_2$ being scanned as a concatenation of words $\{1, 01, 0^2\}$). Further, from the previous discussion on $k = 1$, we know that $\mathbf{x}_1$ has at least one "1" bit for every "0" bit. Together, these imply that $m_j \geq 6(j+1)$. Now, consider the pre-processed sequence $\mathbf{x}_2^* = (0^2 1101)^{j+1}$, which is of length $6(j+1)$ bits and contains $(j+1)$ 0^2 words (with $\mathbf{x}_2^*$ being scanned as a concatenation of words $\{1, 01, 0^2\}$). It can be verified that $\mathbf{x}_2^*$ is indeed a valid sequence, *i.e.*, it satisfies the required conditions in both iterations. This proves that $m_j \leq 6(j+1)$. Hence, we conclude that $m_j = 6(j+1)$, and $|\mathcal{S}_j| = 6$. It follows that $\beta(2, m) = \lfloor \frac{m}{6} \rfloor$, and $\theta(2, m) = \frac{m}{\lfloor \frac{m}{6} \rfloor}$ converges as $m \to \infty$.

Similar arguments can be used for $k = 3$, with $\mathbf{x}_3^* = (1^3 0^3 1 0^2 101)^{j+1}$, and for $k = 4$ with $\mathbf{x}_4^* = (1^7 0^5 1 0^3 1 0^2 1 (01)^2)^{j+1}$. Hence, $\beta(3, m) = \lfloor \frac{m}{12} \rfloor$, and $\beta(4, m) = \lfloor \frac{m}{24} \rfloor$, thus proving the convergence of both $\theta(3, m)$ and $\theta(4, m)$, as $m \to \infty$. Unfortunately, such an analysis is hard to extend to higher values of k, as it relies on finding an appropriate pre-processed sequence $\mathbf{x}_k^*$. However, the above discussion gives us reason to believe that $\theta(k, m)$ is in general well-behaved, and converges for all values of k.

Assuming that $\lim_{m \to \infty} \theta(k, m)$ exists, we now show that this limit is equal to $\inf_{w_{\mathbf{u}_{k-1}^*}(\mathbf{x}_k) \in \mathbb{Z}^+} \frac{l(\mathbf{x})}{w_{\mathbf{u}_{k-1}^*}(\mathbf{x}_k)}$, where $\mathbb{Z}^+$ denotes the set of all positive integers and $l(\mathbf{x})$ denotes the length of sequence $\mathbf{x}$. Consider the subsequence, $\{\psi(k, j)\}_{j=1}^{\infty}$, of $\{\theta(k, m)\}_{m=m_0(k)}^{\infty}$ defined by the local minimum points as follows

$$\psi(k, j) = \theta(k, m_{j-1}).$$

Then, the following relation holds

$$\psi(k, j) \;=\; \inf_{t \in \mathcal{T}_j} \psi(k, t), \tag{150}$$

where $\mathcal{T}_j = \{t : \frac{j}{t} \in \mathbb{Z}^+\}$. Essentially, (150) is a formulation of the fact that a sequence $\mathbf{x}_k$ with j $\mathbf{u}^*_{k-1}$ words can be constructed by concatenating j/t shorter $\mathbf{x}_k$ sequences each with t $\mathbf{u}^*_{k-1}$ words. The desired relation can now be obtained using the following steps

$$\begin{aligned}
\lim_{m\to\infty} \theta(k,m) &= \lim_{j\to\infty} \psi(k,j) \\
&= \lim_{j\to\infty} \inf_{t\in\mathcal{T}_j} \psi(k,t) \\
&= \inf_{w_{\mathbf{u}^*_{k-1}}(\mathbf{x}_k)\in\mathbb{Z}^+} \frac{l(\mathbf{x})}{w_{\mathbf{u}^*_{k-1}}(\mathbf{x}_k)}.
\end{aligned}$$

REFERENCES

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

[2] E. Arikan, "An implementation of Elias coding for input-restricted channels," *IEEE Trans. Inform. Theory*, vol. 36, no. 1, pp. 162-165, Jan. 1990.

[3] S. Aviran, P.H. Siegel and J.K. Wolf, "An improvement to the bit stuffing algorithm," *IEEE Trans. Inform. Theory*, vol. 51, no. 8, pp. 2885-2891, Aug. 2005.

[4] S. Aviran, P.H. Siegel and J.K. Wolf, "Optimal parsing trees for run-length coding of biased data," *submitted to IEEE Trans. Inform. Theory*, Jan. 2006.

[5] J.R. Barry, E.A. Lee and D.G. Messerschmitt, *Digital Communication*, Kluwer Academic Publishers, Third Edition, 2004.

[6] P. Bender and J.K. Wolf, "A universal algorithm for generating optimal and nearly optimal run-length-limited, charge constrained binary sequences," in *Proc. IEEE Int. Symp. Inform. Theory (ISIT),* p. 6, Jan. 1993.

[7] D. Bertsekas and R.G. Gallager, *Data Networks,* Englewood Cliffs, NJ; Prentice-Hall 1987.

[8] W.G. Bliss, "Circuitry for performing error correction calculations on baseband encoded data to eliminate error propagation," *IBM Techn. Discl. Bul.*, vol. 23, pp. 4633-4634, 1981.

[9] R.D. Cidediyan, F. Dolivo, R. Hermann, W. Hirt and W. Schott, "A PRML system for digital magnetic recording," *IEEE J. Sel. Areas Commun.,* vol. 10, no. 1, pp. 38-56, Jan. 1992.

[10] T.M. Cover, "Enumerative source coding," *IEEE Trans. Inform. Theory,* vol. 19, no. 1, pp. 73-77, Jan. 1973.

[11] T.M. Cover and J.A. Thomas, *Elements of Information Theory,* John Wiley, 1991.

[12] E. Eleftheoriou and R.D. Cideciyan, "On codes satisfying Mth order running digital sum constraints," *IEEE Trans. Inform. Theory,* vol. 37, pp. 1294-1313, Sept. 1991.

[13] J.L. Fan and A.R. Calderbank, "A modified concatenated coding scheme, with applications to magnetic data storage," *IEEE Trans. Inform. Theory*, vol. 44, no. 4, pp. 1565-1574, July 1998.

[14] G.D. Forney, Jr. "Maximum likelihood sequence detection in the presence of intersymbol interference," *IEEE Trans. Inform. Theory,* vol. 18, pp. 363-378, May 1972.

[15] K. Forsberg and I.F. Blake, "The enumeration of (d, k) sequences," in *Proc. 26th Allerton Conf. on Communications, Control, and Computing* pp. 471-472, Sept. 1988.

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

[17] K.A.S. Immink, *Codes for Mass Data Storage Systems*. Eindhoven,The Netherlands: Shannon Foundation, 1999.

[18] K.A.S. Immink, P.H. Siegel and J.K. Wolf, "Codes for digital recorders," *IEEE Trans. Inform. Theory,* vol. 44, no. 6, Oct. 1998.

[19] K.A.S. Immink, "A practical method for approaching the channel capacity of constrained channels," *IEEE Trans. Inform. Theory,* vol. 43, no. 5, pp. 1389-1399, Sept. 1997.

[20] K.A.S Immink and A.J. van Wijngaarden, "Simple high-rate constrained codes," *Electronics Letters,* vol. 32, no. 20, p. 1877, Sept. 1996.

[21] F. Jelinek and K.S. Schneider, "On variable-length-to-block coding," *IEEE Trans. Inform. Theory,* vol. 18, no. 6, pp. 765-774, Nov. 1972.

[22] M. Jin, K.A.S. Immink and B. Farhang-Boroujeny, "Design techniques for weakly constrained codes," *IEEE Trans. Comm.*, vol. 51, no. 5, pp 709-714, May 2003.

[23] C.B. Jones, "An efficient coding system for long source sequences," *IEEE Trans. Inform. Theory,* vol. 27, no. 3, pp. 280-291, May 1981.

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

[25] N. Kashyap and P.H. Siegel, "Sliding-block decodable encoders between (d,k) runlength-limited constraints of equal capacity," *IEEE Trans. Inform. Theory*, vol. 50, no. 6, pp. 1327-1331, June 2004.

[26] N. Kashyap and P.H. Siegel, "Equalities among capacities of (d,k)-constrained systems" *SIAM J. Discrete Math*, vol. 17, no. 2, pp. 276-297, 2003.

[27] K.J. Kerpez, "Runlength codes from source codes," *IEEE Trans. Inform. Theory,* vol. 37, pp.682-687, May 1991.

[28] *Coding and Signal Processing for Magnetic Recording Systems,* Editors: B. Vasic and E.M. Kurtas, CRC Press 2005.

[29] J.C. Lawrence, "A new universal coding scheme for the binary memoryless source," *IEEE Trans. Inform. Theory,* vol. 23, no. 4, pp. 466-471, July 1977.

[30] B.H. Marcus, P.H. Siegel and J.K. Wolf, "Finite state modulation codes for data storage," *IEEE J. Sel. Areas Commun.,* vol. 10, no. 1, pp. 5-37, Jan. 1992.

[31] G.N.N. Martin, G.G. Langdon Jr. and S.J.P. Todd, "Arithmetic codes for constrained channels," *IBM J. Res. Develop.,* vol. 27, no. 2, pp. 94-106, March 1983.

[32] S.W. McLaughlin, J. Luo and Q. Xie, "On the capacity of M-ary runlength-limited codes," *IEEE Trans. Inform. Theory,* vol. 41, no. 5, pp. 1508-1511, Sept. 1995.

[33] S.W. McLaughlin, "Improved distance M-ary (d, k) codes for high density recording," *IEEE Trans. Magn.,* vol. 31, no. 2, pp. 1155-1160, March 1995.

[34] S.W. McLaughlin, "Five runlength-limited codes for M-ary recording channels," *IEEE Trans. Magn.,* vol. 33, no. 3, pp. 2442-2450, March 1997.

[35] S.W. McLaughlin, "The construction of M-ary (d, ∞) codes that achieve capacity and have the fewest number of encoder states," *IEEE Trans. Inform. Theory,* vol. 43, no. 2, pp. 699-703, March 1997.

[36] S. Datta and S.W. McLaughlin, "Optimal block codes for M-ary runlength-constrained channels," *IEEE Trans. Inform. Theory,* vol. 47, no. 5, pp. 2069-2078, July 2001.

[37] B.E. Moision, A. Orlitsky and P.H. Siegel, "Bounds on the rate of codes which forbid specified difference sequences," in *Proc. Globecom'99* vol. 18, pp. 878-882.

[38] J. Moon and B. Brickner, "Design of a rate 5/6 maximum transition run code," *IEEE Trans. Magn.,* vol. 33, pp. 2749-2751, Sept. 1997.

[39] J. Moon and B. Brickner, "Maximum transition run codes for data storage systems," *IEEE Trans. Magn.,* vol. 32, no. 5, pt. 1, pp. 3992-3994, Sept. 1994.

[40] H. Morita, M. Hoshi and K. Kobayashi, "A reversible distribution converter with finite precision arithmetic," in *Proc. Int. Symp. Inform. Theory (ISIT)* Lausanne, Switzerland, June-July 2002.

[41] H. Morita, M. Hoshi and K. Kobayashi, "On source conversion using arithmetic codes," in *Proc. AEW4,* Viareggio, Italy, Oct. 2004.

[42] K. Norris and D.S. Bloomberg, "Channel capacity of charge-constrained runlength-limited codes," *IEEE Trans. Magn.,* vol. Mag-17, N0. 6, pp. 3452-3455, Nov. 1981.

[43] L. Patrovics and K.A.S. Immink, "Encoding of *dklr*-sequences using one weight set," *IEE Trans. Inform. Theory,* vol. 42, no. 5, pp. 1553-1554, Sept. 1996.

[44] *Digital Communications,* McGraw-Hill, Fourth edition, 2001.

[45] J. Rissanen and G.G. Langdon Jr., "Arithmetic coding" *IBM J. Res. Develop.,* vol. 23, pp. 149-162, March 1979.

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

[47] Y. Sankarasubramaniam and S.W. McLaughlin, "Capacity achieving code constructions for two classes of (d, k) constraints," *accepted for publication in IEEE Trans. Inform. Theory.*

[48] Y. Sankarasubramaniam and S.W. McLaughlin, "A convolution interpretation of (M, d, k) run-length-limited capacity," in *Proc. Int. Symp. Inform. Theory (ISIT)*, Chicago, USA, June-July 2004.

[49] Y. Sankarasubramaniam and S.W. McLaughlin, "Symbol sliding: improved codes for the (d, k) constraint," in *Proc. Int. Symp. Inform. Theory and Applic. (ISITA)*, Parma, Italy, Oct. 2004.

[50] Y. Sankarasubramaniam and S.W. McLaughlin, "A fixed-rate bit stuffing approach for high efficiency k-constrained codes," in *Proc. Int. Symp. Inform. Theory (ISIT)*, Adelaide, Australia, Sept. 2005.

[51] Y. Sankarasubramaniam and S.W. McLaughlin, "Fixed-rate maximum runlength limited codes from variable-rate bit stuffing," in *Proc. Forty-Third Annual Allerton Conference on Communication, Control, and Computing,* Sept. 2005.

[52] Y. Sankarasubramaniam and S.W. McLaughlin, "Fixed-rate maximum runlength limited codes from variable-rate bit stuffing," *submitted to IEEE Trans. Inform. Theory,* Aug. 2005.

[53] Y. Sankarasubramaniam and S.W. McLaughlin, "Bit stuff encoding for $(0, G/I)$ constraints," *submitted to Int. Symp. Inform. Theory (ISIT)*, Jan. 2006.

[54] Y. Sankarasubramaniam and S.W. McLaughlin, "High-rate $(0, k)$-constrained codes with reduced error propagation," *Progress report submitted to Seagate Research,* Nov. 2005.

[55] J.P.M. Schalkwijk, "An algorithm for source coding," *IEEE Trans. Inform. Theory,* vol. 18, no. 3, pp. 395-399, May 1972.

[56] M.C. Stefanovic and B.V. Vasic, "Channel capacity of $(0, G/I)$ codes," *IEEE Electronics Letters,* vol. 29, no. 2, pp. 243-245, Jan. 1993.

[57] H. Thapar and A. Patel, "A class of partial response systems for increasing storage density in magnetic recording," *IEEE Trans. Magn.,* vol. 23, no. 5, pp. 3666-3668, Sep. 1987.

[58] S.J.P. Todd, G.G. Langdon Jr. and G.N.N. Martin, "A general fixed rate arithmetic coding method for constrained channels," *IBM J. Res. Develop.,* vol. 27, no. 2, pp. 107-115, March 1983.

[59] S.W. Wicker, *Error Control Systems for Digital Communication and Storage,* Prentice-Hall, 1995.

[60] A.J. van Wijngaarden and K.A.S. Immink, "Combinatorial construction of high rate runlength-limited codes," in *Proc. Globecom'96*, vol. 1, pp. 343-347, Nov. 1996.

[61] E. Zehavi and J.K. Wolf, "On runlength codes," *IEEE Trans. Inform. Theory,* vol. 34, no. 1, pp. 45-54, Jan. 1988.

VITA

Yogesh Sankarasubramaniam was born in Chennai, India on August 2, 1979. He received the B.Tech. degree in Electrical Engineering from the Indian Institute of Technology, Madras, India, in May 2001. In August 2001, he started graduate studies at the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, Georgia, USA, where he received the M.S. degree in 2003, and the Ph.D. degree in 2006. His current research interests include information theory, error control coding and digital signal processing.