Project overview
There has been a great deal of interest in novel MEMs-based storage methods, most notably Millipede developed by IBM, for going toward storage densities of 1Tb/in^2. There are a host of coding and signal processing issues in such systems to increase density, correct for errors, perform timing recovery, tracking and servo. We have considerable experience in developing new methods in these areas for conventional magnetic and optical storage and are interested in expanding into MEMs-based storage technology. There are very interesting constrained coding approaches already being used in the AFM approach of IBM. The use of $(d,k)$ constrained runlength limited (RLL) codes allows the net density to increase by about 50%. The constrained code is also very well matched the erasure and re-writability of the polymer material. The codes being used currently are one-dimensional codes (along a row) even though the inherent medium is two-dimensional. We are certainly interested in designing more efficient and easier to use codes, that are both better suited to two-dimensions and error control. Also the issue of ECC is an interesting one — there has been a great deal of recent work in advanced ECC methods using soft information, iterative decoding etc., and we are highly capable at developing efficient ECC methods for channels like these. It is assumed that the target data rates are comparable, even higher than, current HDD applications (in 100-1000 Mbps).

Overview of our accomplishments
Our work focused primarily on the design of new constrained coding theory and practice. The specific details are given in the attached progress reviews. The high level accomplishments were

- A new theory for constrained codes that considers time-domain techniques and facts on factoring polynomials and their role in capacity
- Several new code constructions that use very simple primitives based on bit-stuffing. In some cases these codes achieved capacity and in all cases these codes came within 99% of the maximum achievable limits.

Publications that came from this project


High-Rate \((0,k)\)-Constrained Codes with Reduced Error-Propagation

Progress Report

Yogesh Sankarasubramaniam  Steven W. McLaughlin  
School of Electrical & Computer Engineering  
Georgia Institute of Technology, Atlanta, GA 30332  
E-mail:{yogi,swm}@ece.gatech.edu

Summary: We describe the design of rate > 100/101 \((0,k)\)-constrained codes that have reduced error-propagation.

In previous work [2], we introduced fixed-rate bit stuff (FRB) codes, where

- Iterative pre-processing followed by bit stuffing was used to generate very high-rate codes, with asymptotic rates of the form \(\frac{n-1}{n}, n \sim 2^k\), for any given maximum runlength parameter \(k\). Furthermore, required code parameters, namely: input block length \(m\), number of pre-processing iterations \(r\), and maximum runlength parameter \(k\) were tabulated for the design of rate 100/101 codes.

- The iterative pre-processing is simpler than enumeration, and is key to achieving high encoding rates, which are far greater than the combinatorial construction of Immink and Wijngaarden [1].

- However, the proposed construction had the error-propagation problem, and our initial approach was to look at reverse concatenation as a possible solution.

As it turns out, one can construct high-rate codes with good error-propagation characteristics even without using reverse concatenation. The aim of this report is to describe such a code construction. The main ideas behind the new codes, referred to as Iterative Pre-Processed (IPP) codes, are as follows:

- Insertion/deletion of bits can lead to error propagation, and hence the IPP codes do not use bit stuffing. Only the iterative pre-processing of FRB codes is retained. This is essential for generating high-rate codes. There is a penalty in doing away with bit stuffing: we can now encode only in smaller blocks, and hence for a given maximum runlength parameter \(k\),
the achievable rates are lower than that of the FRB codes. However, there is lesser error propagation.

- Iterative pre-processing is also a source of propagating errors. Hence, we process the input data block only when necessary. On the other hand, the FRB encoder processed every incoming data block - even if the data block did not violate the \( k \) constraint.

The following discussion provides the code construction details.

**\( k = 17, \text{rate} > 100/101 \) code with low error-propagation**

For a given maximum runlength 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 zeros of length greater than or equal to \( k + 1 \). A lower bound on \( m(k+1, k) \) was derived in [2].

\[
m(k + 1, k) \geq \begin{cases} 
1 & \text{when } k = 0 \\
2^{k+2} - 2^k - 1 & \text{when } k \geq 1.
\end{cases}
\]  

(1)

A lower bound on \( m(r, k) \) can be derived as

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

(2)

Let us consider \( k = 17 \) with \( r = 9 \) pre-processing iterations. Using (2), 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 zeros of length greater than or equal to 18 in the processed sequence. We append a “1” bit at the end of the encoded 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. Hence, the rate of this code is \( \frac{1535}{1536+10} \sim 139/140 \).

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.

We use the following definitions and notations:

- Denote the input data sequence by \( x \), and the iterative pre-processing output by \( x_r \).

- Let

\[
u_i = \begin{cases} 
0^i & \text{when } i = 0, 1, \ldots, k \\
0^{k+1} & \text{when } i = k + 1,
\end{cases}
\]  

(3)

and \( u_i^* = 0^{i+1}, i = 0, 1, \ldots, k \). Hence, by definition, \( u_{k+1}^* \equiv u_{k+1} \).
The weight of a binary sequence $s$ with respect to the input word $u^*_i$, denoted by $w_{u^*_i}(s)$, is the number of distinct occurrences of $u^*_i$ in $s$, with $s$ being scanned as a concatenation of words from the set $\{u_0, u_1, \ldots, u_i, u^*_i\}$, $i = 0, 1, \ldots, k$. For example, if $s = 100001010001$, $k \geq 1$ and $i = 1$, $w_{u^*_1}(s) = 3$.

The weight of a binary sequence $s$ with respect to the input word $u_i$, denoted by $w_{u_i}(s)$, is the number of distinct occurrences of $u_i$ in $s$, with $s$ being scanned as a concatenation of words from the set $\{u_0, u_1, \ldots, u_i, u^*_i\}$, $i = 0, 1, \ldots, k$. For example, if $s = 100001010001$, $k \geq 1$ and $i = 1$, $w_{u_1}(s) = 2$.

Denote by $\overline{s}(i)$, the sequence formed by flipping those bit positions in $s$ that follow a string of $i$ consecutive zeros, $i = 0, 1, \ldots, k$. 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 $s$ is flipped. Equivalently, $\overline{s}(i)$ can be described as converting all $u_i$ words to $u^*_i$ words, and all $u^*_i$ words to $u_i$ words in $s$, with $s$ being scanned as a concatenation of words from the set $\{u_0, u_1, \ldots, u_i, u^*_i\}$, $i = 0, 1, \ldots, k$. For example, if $s = 100001010001$, and $k \geq 1$, $\overline{s}(1) = 101011000100$. Note that $\overline{s}(i)$ is an invertible operation.

Then the IPP encoding algorithm A1 (IPP-A1) is defined as follows:

Input $x$;
Set $x_0 = x$;
For $j = 1$ to $r$, do begin
Input $x_{j-1}$;
Scan $x_{j-1}$ as a concatenation of words from the set $\{u_0, u_1, \ldots, u_{j-1}, u^*_{j-1}\}$;
If $w_{u_{j-1}}(x_{j-1}) < w_{u^*_j-1}(x_{j-1})$, output $x_j = \overline{x}_{j-1}(j-1)$; $\alpha_j = 1$;
Else output $x_j = x_{j-1}$; $\alpha_j = 0$;
end
Encoded sequence is $(x_1)$
Index sequence is $\alpha = (\alpha_1 \alpha_2 \ldots \alpha_r 1)$

What remains is to evaluate the error-propagation characteristics. As a first step, we plot the error distribution histogram 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.

As seen from Fig. 1, the error propagation of IPP-A1 codes is rather high, eventhough it does not have the insertion/deletion errors of the FRB codes. Hence, we define the following improved algorithm, IPP-A2:

Input $x$;
If $x$ does not violate the $(0,k)$ constraint
$x_r = x$; $\alpha = (0 0 \ldots 0 1)$;
Else
Set $x_0 = x$;
For $j = 1$ to $r$, do begin
Input $x_{j-1}$;
Scan $x_{j-1}$ as a concatenation of words from the set $\{u_0, u_1, \ldots, u_{j-1}, u_{j-1}^*\}$;
If $w_{u_{j-1}}(x_{j-1}) < w_{u_{j-1}^*}(x_{j-1})$, output $x_j = \overline{x}_{j-1}(j-1)$; $\alpha_j = 1$;
Else output $x_j = x_{j-1}$; $\alpha_j = 0$;
end
Encoded sequence is $(x, 1)$
Index sequence is $\alpha = (\alpha_1 \alpha_2 \ldots \alpha_r \ 1)$

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). Hence, although the worst case error-propagation is unaltered, the average error propagation is substantially improved, as shown in Fig. 2.

![Figure 1: IPP-A1 algorithm](image1)
![Figure 2: IPP-A2 algorithm](image2)

A further improvement to algorithm IPP-A2 is possible. One can also check if the flipped data block, $\overline{x}$, satisfies the $(0, k)$ constraint. If it does, then the sequence $\overline{x}$ is transmitted. Clearly, a single bit error in $\overline{x}$ is recoverable as a single data bit error. Hence, IPP-A3 further improves the performance of IPP-A2. This is illustrated in Figs. 3 and 4, where a certain input sequence undergoes pre-processing with algorithm IPP-A2 (which leads to some propagation of errors), but since $\overline{x}$ 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. The average error propagation for the IPP-A3 algorithm is shown in Fig. 5 on the left. Note the small increase in the percentage of single data bit errors, as compared to Fig. 2.
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 3: Error distribution due to a certain input sequence with algorithm IPP-A2.

Figure 4: The same sequence has reduced error propagation with algorithm IPP-A3.

Figure 5: Comparison of error propagation characteristics of algorithm IPP-A3 and the combinatorial construction of Immink and Wijngaarden.

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 [1] of rate 48/49, as shown in Fig. 5.

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. 6, a single index-bit error can propagate through to several data bits. From the IPP-A3 histogram in Fig. 5, 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.

![Error propagation due to a single index-bit error. It is possible that the entire data block (of length 1535 bits) is in error.](image)

**Further Research**

1. Build up a simulation model as described in [3] for the magnetic recording channel, which goes beyond the simple RS-encode+EPR4 model we used earlier.

2. Conduct the BER test using the above simulation model: compare BERs before and after the (0, k) decoder

3. Carry out a simple analysis that quantifies error-propagation improvements due to IPP algorithms.

**References**


Progress Report
October-December 2005

Yogesh Sankarasubramaniam  Steven W. McLaughlin
School of Electrical & Computer Engineering
Georgia Institute of Technology, Atlanta, GA 30332
E-mail: yogi,swm}@ece.gatech.edu

Summary
During this quarter we worked on the following items:

- Analysis and reduction of error propagation in the proposed $(0,k)$ fixed-rate codes.
- A parallel implementation of the pre-processing iterations.
- Extension of encoding ideas to $(0,G/I)$ constraints.

In what follows, we provide details on our findings.

I. ANALYSIS AND REDUCTION OF ERROR PROPAGATION IN THE PROPOSED $(0,k)$ CODES

In previous work [2], we introduced the fixed-rate bit stuff (FRB) codes, with very high rate and relatively simple implementation, but the FRB codes suffered from error propagation. In the worst case, it is possible that the entire data block is in error due to a single channel-bit error. To overcome this drawback, we make a slight modification to FRB codes and propose the new codes, referred to as Iterative Pre-Processed (IPP) codes. The main ideas behind the IPP codes are as follows:

- Insertion/deletion of bits can lead to error propagation, and hence the IPP codes do not use bit stuffing. Only the iterative pre-processing of FRB codes is retained. This is essential for generating high-rate codes.
- There is a penalty in doing away with bit stuffing: we can now encode only in smaller blocks, and hence for a given maximum runlength parameter $k$, the achievable rates are lower than that of the FRB codes. However, there is reduced error propagation.
- Iterative pre-processing is also a source of propagating errors. Hence, we process the input data block only when necessary.

On the other hand, the FRB encoder processed every incoming data block - even if the data block did not violate the $k$ constraint.

As an example, let us consider $k = 17$ with $r = 9$ pre-processing iterations. We encode in input blocks of length 1535 bits. From our results in [2], [3], we see that using 9 pre-processing iterations on 1535 bits guarantees that there will be no strings of zeros of length greater than or equal to 18 in the processed sequence. We append a “1” bit at the end of the encoded 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. Hence, the rate of this code is $\frac{1535}{1536+10} \approx 139/140$. Note that there is no bit stuffing involved in this construction.

We define the encoding algorithm IPP-A2 as follows (see [3] for more details):

Input $x$;
If $x$ does not violate the $(0,k)$ constraint
$\quad x_r = x; \ \alpha = (0 \ 0 \ldots \ 0 \ 1)$;
Else
Set $x_0 = x$;
For $j = 1$ to $r$, do begin
Input $x_{j-1}$;
Scan $x_{j-1}$ as a concatenation of words from the set $(u_0, u_1, \ldots, u_{j-1}, u_{j-1}^*)$;
If $u_{j-1}(x_{j-1}) < u_{j-1}^*(x_{j-1})$, output $x_j = x_{j-1}(j - 1); \ \alpha_j = 1$;
Else output $x_j = x_{j-1}; \ \alpha_j = 0$;
end
Encoded sequence is $(x_1)$
Index sequence is $\alpha = (\alpha_1 \ \alpha_2 \ldots \ \alpha_r \ 1)$

Essentially, the IPP-A2 algorithm improves over the FRB algorithm by eliminating bit insertions, and 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). Hence, although the worst case error-propagation is unaltered, the average error propagation is substantially improved. An upper bound for the probability of worst-case error propagation is given in (3).

A further improvement to algorithm IPP-A2 is possible. One can also check if the flipped data block, $\bar{x}$, satisfies the $(0,k)$ constraint. If it does, then the sequence $\bar{x}$ is transmitted. Clearly, a single bit error in $\bar{x}$ is recoverable as a single data bit error. We call this further modification, the IPP-A3 algorithm. Essentially IPP-A3 lowers the probability that a data sequence has to undergo multiple pre-processing iterations, and hence reduces the IPP-A2 error propagation. A comparison of the error distribution histogram of the rate $139/140, k = 17$, IPP-A3 code with that of the corresponding $k = 17$ combinatorial code [1] of rate 48/49 is shown in Fig. 1.
One issue we have not addressed in our discussion thus far, is that of error propagation due to index-bit errors. A single index-bit error can propagate through to several data bits. From the IPP-A3 histogram in Fig. 1, 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.

A quick analysis of the IPP-A3 algorithm yields the following results.

- The worst case error propagation with \( r \) pre-processing iterations for a maximum runlength parameter \( k \), is given by

\[
E(r, k) \leq \max_{x \in \{2, 3, \ldots, r\}} \frac{2x + (x + 1)}{\frac{x+1}{x} m(x, x - 1)} + 1,
\]

where \( m(r, k) \) is defined as the maximum input block length for which the output of the iterative pre-processing with \( r \) iterations, has no strings of consecutive zeros of length greater than or equal to \( k + 1 \). The following lower bound on \( m(k+1,k) \) was derived in [2].

\[
m(k+1,k) \geq \begin{cases} 
1 & \text{when } k = 0 \\
2^{k+2} - 2^{k} - 2 & \text{when } k \geq 1.
\end{cases}
\]

A lower bound on \( m(r, k) \) can be derived as

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

- For large \( k \) and suitable \( m \), a union bound on the probability that the data block undergoes multiple pre-processing iterations with IPP-A3 is

\[
P(E) \leq (m-k)(m-2k-1)1/2 2^{k+2}
\]

This gives us an estimate of the probability that a single channel-bit error leads to more than one data bit error, \textit{i.e.}, a propagation of errors.

II. Other Results

Some of our other findings during this quarter are:

- A parallel and semi-parallel implementation of the pre-processing iterations that might be useful in possible hardware implementation of the FRB/IPP algorithms. The code for parallel implementation was executed in MATLAB (albeit serially) and the encoding results were verified to match that of the serial implementation.

- Extension of the proposed encoding to \((0, G/l)\) constraints. We propose a new variable-rate and fixed-rate algorithm to generate near-capacity \((0, G/l)\) codes. We are currently preparing a submission to ISIT'06 based on these ideas.

- We studied the modeling of electronics and media noise in both longitudinal and perpendicular recording. The next step is to incorporate them in our earlier channel model, and re-evaluate the FRB/IPP code performance.

REFERENCES

Summary
During this quarter we worked on the following items:

- System performance of proposed $(0, k)$ fixed-rate bit stuff codes.
- Submission of journal paper on $(0, k)$ fixed-rate bit stuff codes to IEEE Trans. Inform. Theory.
- Revision of earlier submitted journal paper on variable-rate capacity-achieving codes.

In what follows, we provide details on the simulation setup and performance results of the proposed $(0, k)$ codes on a magnetic recording channel.

1. System Performance of Proposed $(0, k)$ Fixed-Rate Bit Stuff Codes

The $(0, k)$ constraint is used in magnetic recording systems to make the stored data self-clocking. In recent work [1], we proposed an algorithm to generate near-capacity $(0, k)$ codes. The idea is to derive efficient, fixed-rate codes from a simple, variable-rate technique called bit stuffing. However, the proposed long block codes may be prone to error propagation effects. This motivates us to study the performance of these codes under a suitable reverse-concatenation scheme that combats propagation of errors.

The reverse-concatenation scheme we use is the one proposed by Bliss, and analyzed by Fan and Calderbank [2]. Figs. 1 and 2 show the standard concatenation and reverse-concatenation configurations, respectively. The EPR4 model, with a transfer function of $1 + D - D^2 - D^3$, is used to model the magnetic recording channel. This is appropriate for PW50/T in the range (1.6, 2.2). The corresponding ML Viterbi detector has an 8-state trellis. A byte-oriented RS code capable of correcting at most 8 byte errors, is used in both the standard and reverse concatenation schemes. The modulation code is specified by parameters $(r, k, k_m, n_m)$, where $r$ denotes the number of pre-processing iterations, $k_m$ is the fixed input-length in bits and $n_m$ is the fixed output-length in bits. The parameter values shown in Figs. 1 and 2 generate $(0, k)$ codes with rate close to 100/101. Note that our $(0, k)$ code construction allows flexible parameter choices that can be used to trade off encoding/decoding latency for the maximum runlength, 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 [1]). The short modulation code shown in Fig. 2 is used only for the RS parity bits, and can be constructed using the combinatorial technique of [3], where error propagation can be limited to atmost one byte. This code will be of a low rate 42/43 for $k = 15$. In our preliminary simulations, we do not explicitly include this short $(0, 15)$ code. For simplicity, we assume that the parity bits are directly affected by the channel ISI and noise. The overall encoding rates for the EPR4 channel are 0.9264 for the standard concatenation, and 0.9241 for the reverse concatenation.

![System model for standard concatenation using the $(0, k)$ fixed-rate bit stuff codes.](image1)

![System model for Bliss's reverse-concatenation using the $(0, k)$ fixed-rate bit stuff codes.](image2)

II. Preliminary Results for Low-SNR Region

Our preliminary results in Figs. 3, 4 and 5 show the improved performance of reverse-concatenation. The PER shows a steady improvement with increasing SNR. The SNR is measured at the input of the MLSD.
Ongoing and Future Work:

1) Complete the simulations for the higher SNR region.
2) In our simulations, we fixed the RS code rate to 239/255. The next step is to optimize the RS code rates for different ranges of SNR.
3) We noticed a small anomaly in the BER values. The BER3 value, for each SNR, is slightly greater than the BER1 value for the corresponding standard concatenation. However since they are both the channel BERs, such a trend is not expected. The nature of dummy-bit padding and the non-implementation of the short code are possible reasons. We intend to investigate further. Note that the reverse-concatenation gains are lesser due to this anomaly.
4) Repeat the simulations for rate ~ 200/201 fixed-rate codes, and possibly other parameter values.
5) Analysis using error event polynomials for the EPR4 and possibly higher order channel models.

REFERENCES

Progress Report

Yogesh Sankarasubramaniam  Steven W. McLaughlin
School of Electrical & Computer Engineering
Georgia Institute of Technology, Atlanta, GA 30332
E-mail: {yogi,swm}@ece.gatech.edu

Summary
During this quarter we worked on the following items:

- Analysis and coding for constrained channels using time-domain methods
- Analysis and coding for constrained channels using symbol sliding
- Two teleconference presentations on M-ary recording

In what follows we give more details on our activities in these areas and some of our findings.

1. Symbol Sliding: Improved Codes for the $(d, k)$ Constraint

Bender and Wolf [1] first proposed the simple bit stuffing algorithm to generate $(d, k)$ constrained sequences by inserting bits into an arbitrary data stream. The bit stuffing algorithm is interesting because of its simple implementation but surprisingly high efficiency. More recently, bit flipping [2] was suggested as an improvement to bit stuffing.

We propose the symbol sliding algorithm which further improves bit stuffing and bit flipping rates while maintaining similar implementation complexity. In fact, the bit stuffing and bit flipping algorithms can be derived as special cases of symbol sliding. Symbol sliding is additionally optimal for all $(d, k)$ pairs with $k = 2d + 1$. In the following discussion, we summarize some important properties of the algorithm.

- The main idea is to generate constrained phrases with probabilities that match those of the maxentropic sequence. The finite state transition diagram (FSTD) of the $(d, k)$ constraint is shown in Fig. 1. A code achieves capacity if and only if it produces a $(d, k)$ constrained walk on the FSTD with the maxentropic state transition probabilities shown [3]. The maxentropic phrase probability vector is given by $\Lambda = [\lambda^{-(k+1)} \lambda^{-(k)} \ldots \lambda^{-(d+2)} \lambda^{-(d+1)}]^T$, where $\lambda^{-i}$ denotes the probability of occurrence of a $(d, k)$ constrained phrase of length $i$ ($i - 1$ zeros followed by one, henceforth denoted as $0^{(i-1)}1$).

![Fig. 1. FSTD with the maxentropic state transition probabilities in parentheses.](image)

- Any encoding algorithm achieves capacity if and only if the phrase probability vector it generates is identical to $\Lambda$. It can be shown that simply exchanging the roles of some phrase probabilities in the bit stuffing algorithm can result in rate improvements. In particular, we employ a sliding rule to remap the phrase probabilities. This is illustrated in the following example of the $(1, 3)$ constraint. Table I lists the phrase probabilities. $p$ denotes the bias of the input bit stream ($Pr[0] = p$).

<table>
<thead>
<tr>
<th>Index $(i)$</th>
<th>$(1,3)$ constrained phrase</th>
<th>Maxentropic prob. $\Lambda(i)$</th>
<th>Bit stuffing $(v_i^0)$</th>
<th>Bit flipping $(v_i^1)$</th>
<th>Symbol sliding with index 2 $(v_i^2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$0^31$</td>
<td>$\lambda^{-4}$</td>
<td>$p^3$</td>
<td>$p(1 - p)$</td>
<td>$p^2$</td>
</tr>
<tr>
<td>1</td>
<td>$0^21$</td>
<td>$\lambda^{-5}$</td>
<td>$p(1 - p)$</td>
<td>$p^2$</td>
<td>$1 - p$</td>
</tr>
<tr>
<td>2</td>
<td>$0^11$</td>
<td>$\lambda^{-6}$</td>
<td>$1 - p$</td>
<td>$p^2$</td>
<td></td>
</tr>
</tbody>
</table>

It is known that the maximum average bit stuffing and bit flipping rates are less than the $(1, 3)$ capacity. Now consider the probabilities $v_i^2$ as listed in Table I. We call this symbol sliding with index 2. It is easy to see that with a bias of $p = \frac{1}{2}$, the probability vector $v^2 = [p(1 - p) \ 1 - p \ p^2]^T$ exactly matches $\Lambda$ 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 idea of swapping phrase probabilities is generalized in the symbol sliding algorithm.

- **Implementation**: Requires a constrained encoder that sequentially performs the following two operations on the biased bit stream. Index $j, 0 \leq j \leq k - d$, for a given $(d, k)$ pair, denotes the symbol sliding index.

1) Track the runlength $(\rho)$ of consecutive zeros
   - If current bit is zero and $\rho < k - d$, goto 1
   - If current bit is zero and $\rho = k - d$, replace the run of $k - d$ zeros with the phrase $0^{k-d-j-1}$, reset $\rho$ and goto 1
   - If current bit is one and $k - d - j \leq \rho < k - d$, insert a zero, reset $\rho$ and goto 1
2) Stuff $d$ zeros after every one

- Let us denote symbol sliding with index $j$ as $SS(j)$. Note that $SS(0)$ and $SS(1)$ are identical to the bit stuffing and bit flipping algorithms, respectively. The following properties can be shown.

1) Let $d \geq 0$, $d < k < \infty$. Then, the maximum average rate achieved by $SS(j)$ equals the $(d, k)$ capacity when $k = 2d + 1$ and $j = k - d - d + 1$.

2) For $0 < d < k$, the maximum average rate achieved by $SS(j)$ equals the $(d, k)$ capacity only in the following cases: $j = 0$, $k = d + 1$; $j = 1$, $k = d + 1$; $j = 1$, $k = \infty$; $j = 1$, $d = 2$, $k = 4$ and $j = k - d$, $k = 2d + 1$.

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

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

4) A closed form expression for the average rate of $SS(j)$ is given by

$$R_j(p, d, k) = \frac{1 - p^{k-d}}{1 - p^{k-d} + (1-p)(p^{k-d-j} - jp^{k-d} + d)}$$

Optimization for both $p$ and $j$ is done numerically due to the complexity of the rate expression. Rate improvements for some important constraint pairs are summarized in Table II.

### Table II

**Examples of Some Constraints**

<table>
<thead>
<tr>
<th>$d$</th>
<th>$k$</th>
<th>Shannon Capacity $C(d, k)$</th>
<th>Maximum bit stuffing efficiency (%)</th>
<th>Maximum bit flipping efficiency (%)</th>
<th>Maximum symbol sliding efficiency (%)</th>
<th>Maximizing symbol sliding index $j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>0.5515</td>
<td>98.93</td>
<td>99.74</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>0.6793</td>
<td>99.42</td>
<td>99.79</td>
<td>99.79</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>0.6450</td>
<td>98.97</td>
<td>99.74</td>
<td>100</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>0.5418</td>
<td>99.39</td>
<td>99.70</td>
<td>99.87</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>0.3746</td>
<td>98.23</td>
<td>99.57</td>
<td>99.89</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>0.3432</td>
<td>98.02</td>
<td>99.16</td>
<td>99.91</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>0.2979</td>
<td>97.82</td>
<td>98.89</td>
<td>99.77</td>
<td>3</td>
</tr>
</tbody>
</table>

**Future Work:**

1) Implementation of the symbol sliding algorithm in MATLAB.

2) Possible fixed rate implementations of symbol sliding.

3) Optimal code constructions using interleaving was shown in previous work. A combination of symbol sliding and interleaving is worth investigating.

4) Possible extension of these ideas to constrained encoding in two dimensions.

II. A CONVOLUTION INTERPRETATION OF $(M, d, k)$ CAPACITY

From $Z$-transforms, we know that the characteristic $(d, k)$ polynomial has a corresponding interpretation as a discrete-time sequence, i.e., $Z(d, k) \overset{Z^{-1}}{\rightarrow} \sum_{j=0}^{k-1} \delta(n-j)$, for $k < \infty$ and $Z(d, k) \overset{Z^{-1}}{\rightarrow} \delta(n-(d+1)) + \delta(n-1)$. For $k = \infty$. We had earlier used this interpretation for code construction using interleaving. Now, we show its effectiveness in determining capacity equivalences of constraint pairs.

We consider the problem of capacity equivalence for a general class of $(M, d, k)$ run-length-limited constraints, namely, we look for pairs $(M, d, k)$ and $(M', d', k')$ that have the same capacity. Using the convolution interpretation, we first derive the known binary equivalence relations and then generalize to the $M$-ary case. The following three $M$-ary equivalences were found [4].

$$C(M, 0, \infty) = C(M^d(M - 1) + 1, d', \infty), \quad M \geq 2, d' \geq 1$$

$$C(M, 0, \infty) = C(M^{d+1} + 1, d', d''), \quad M \geq 2, d' \geq 0$$

$$C(M, d, d) = C((M - 1)^p + 1, pd + p - 1, pd + p - 1), \quad M \geq 2, d \geq 0, p \geq 1$$

**Future Work:**

- We conjecture that there are $M$-ary equivalences beyond those in (1). The entire set of equivalences remains to be determined.

- The time domain approach may be used to show that the two known binary equivalences are the only ones possible.

- In general, the convolution interpretation is promising and has potential applications in constrained code design.

**References**


Progress Report
April-June 2004

Yogesh Sankarasubramaniam  Steven W. McLaughlin
School of Electrical & Computer Engineering
Georgia Institute of Technology, Atlanta, GA 30332
E-mail: {yogi,swm}@ece.gatech.edu

Summary
During this quarter we worked on the following items:

- Journal submission [1]
- Extension of interleaving code construction to a wider range of \((d,k)\) constraints
- Further analysis of time-domain approach for capacity equivalences

In what follows we give more details on our activities in these areas.

1. Extension of Interleaving Construction

In earlier work, we had presented a novel code construction using interleaving. Our construction was optimal for the class of \((d,d+2^m-1)\) constraints. We now extend this idea to a wider class of constraints, \((d,k)\), with \(k-d+1\) not prime.

- We first derive appropriate factorizations for general characteristic \((d,k)\) polynomials, \(k < \infty\). Let us start with the characteristic polynomial

\[
H_{d,k}(z) = \sum_{j=d+1}^{k+1} z^{-j}
\]

(1)

The number of terms in the summation in (1) is equal to \(k-d+1\). Let \(k-d+1\) be factored into the product of primes as

\[
k-d+1 = \prod_{i=1}^{n} P_i
\]

(2)

Now define \(\eta_i = \prod_{j=1}^{i} P_i\), \(i = 1, 2, \ldots, n\), with \(\eta_0 = 1\). It follows (with the help of convolution interpretation introduced in prior work [3]) that \(H_{d,k}(z)\) can be factored as

\[
H_{d,k}(z) = z^{-(d+1)} \prod_{i=1}^{n} F_{d,k}^i(z),
\]

(3)

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

\[
F_{d,k}^i(z) = 1 + z^{-\eta_{i-1}} + z^{-2\eta_{i-1}} + \ldots + z^{-(P_i-1)\eta_{i-1}}
\]

(4)

- Each factor \(F_{d,k}^i(z)\) has \(P_i\) terms and can be realized using \((P_i-1)\) DTs. Hence, the total number of DTs required is \(\sum_{i=1}^{n} (P_i-1)\). As long as \(k-d+1\) is not prime, and the number of factors \(n\) is greater than one, this is strictly less than the \(k-d\) DTs required in a recent construction (we refer to this as the multiple DT construction) described by Wolf [4].

- As an example, we now describe our construction for the \((0,11)\) constraint. The characteristic \((0,11)\) polynomial can be factored as

\[
H_{0,11}(z) = \sum_{j=1}^{12} z^{-j}
\]

(5)

\[
= z^{-1} (1 + z^{-1}) (1 + z^{-2})(1 + z^{-4} + z^{-8})
\]

(6)

Fig. 1 shows the code construction that uses 4 DTs, one 4-bit interleave and one variable length encoder. The bias of the 4 DTs are determined by working backwards from a maxentropic output. This ensures that the codes constructed are optimal. In Fig. 1, the DTs with bias \(\frac{1}{3}\) correspond to factors \((1 + z^{-1})\) and \((1 + z^{-2})\), respectively. The remaining two DTs with bias \(\frac{1}{3}\) both correspond to the factor \((1 + z^{-4} + z^{-8})\). The interleave functionality is as follows. If \(u_1 = 1\), the interleave generates a binary sequence \(u = (u_1u_2u_3u_4)\) by interleaving the 4 biased streams one bit at a time in the specified order \((u_1\) is the MSB and \(u_4\) the LSB). If \(u_1 = 0\), the interleave skips the second biased stream (shown in dotted lines) and outputs the binary sequence \(u = (u_1u_3u_4)\). The encoder then maps the binary sequence \(u\) to \((0,11)\) constrained phrases as specified in Table I. The size of this table is 12 for this example and \(k-d+1\) in general.
The code construction described above requires 4 DTs, as opposed to 11 DTs \((k - d\) in general) required by the multiple DT construction.

**Future Work:**

1) Combination of symbol sliding (proposed in prior work [2]) and interleaving to further improve information rates.
2) Extension of these ideas to constrained encoding in two dimensions.

II. PROOF OF EXHAUSTIVENESS FOR BINARY CAPACITY EQUIVALENCES

In earlier work [3], we had proposed a time-domain approach to identify \((M, d, k)\) capacity equivalences. We now provide a proof of exhaustiveness for capacity equalities between binary \((d, \infty)\) and \((d', k')\) constraints.

**Theorem 1:** Between constraints of the type \((d, \infty)\) and \((d', k')\), the only possible capacity equivalence relation is \(C(d + 1, \infty) = C(d, 2d + 1)\).

**Sketch of proof:** We make use of the following two facts

- The region of convergence (ROC) of \(G(z)\) includes the circle \(|z| = \lambda\), where \(\lambda\) is the positive, real root of \(H_{d,k}(z) = 1\).
- With \(k = \infty\), for given \(d\) and \(d'\), \(k' \leq d + d'\)

We then write out \(G(z)\) and analyze its pole-zero characteristics. Recall that \(G(z)\) has neither poles nor zeros positive, real. This constraint leads us to \(d' = d - 1\) and \(k' = 2d - 1\).

**REFERENCES**

Progress Report  
July-September 2004

Yogesh Sankarasubramaniam  Steven W. McLaughlin  
School of Electrical & Computer Engineering  
Georgia Institute of Technology, Atlanta, GA 30332  
E-mail: {yogi,swm}@ece.gatech.edu

Summary
During this quarter we worked on fixed-rate bit stuff encoding for the noiseless \((d, k)\) constrained channel. The main idea is to use multiple (input word-constrained phrase) mappings instead of the bit stuff mapping alone. In what follows, we propose fixed-rate code constructions for \((d, \infty)\) and \((d, d+1)\) constraints and discuss possible extensions for all values \((d, k)\).

1. Fixed-rate Bit Stuff Codes for \((d, k)\) Constraints

The bit stuffing algorithm [1] is a simple and efficient method of generating \((d, k)\) sequences. It uses the following two bit insertion rules sequentially: insert a one after every run of \(k-d\) consecutive zeros; stuff \(d\) zeros after every one. The main stumbling block in implementation has been the variable-rate nature of bit stuff encoding. In this report, we summarize our design of fixed-rate bit stuff codes for the noiseless \((d, \infty)\) and \((d, d+1)\) constrained channels, and then propose extensions for all values \((d, k)\).

- We start with the \((1, \infty)\) constraint as a motivating example. The \((1, \infty)\) bit stuff mapping is specified in Table I, with individual mapping rates given in the rightmost column. Let us assume that the fixed-rate code has input block length \(m\) and output block length \(n\). Since the encoding is variable-length, the \(m\) input bits will produce outputs of varying lengths, the largest of which is \(n\). For any output of length \(n' < n\) bits, the remaining \(n - n'\) are filled as dummy bits that are ignored in the decoding process. From Table I, the code rate \(R = \frac{m}{n} = \frac{2n}{n} = 2\) is limited to \(\frac{1}{2}\) (only 72% efficient), which happens when all input bits are one. Our design goal is to have \(n < 2m\) so that \(R > \frac{1}{2}\).

<table>
<thead>
<tr>
<th>Input word</th>
<th>Corresponding ((1, \infty)) phrase</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>(\frac{1}{2})</td>
</tr>
</tbody>
</table>

- The main idea is as follows. Consider the bit stuff codewords generated by \((m + 1)\)-bit inputs. From the set of \(2^m+1\) possible codewords, we pick the \(2^m\) codewords that have the least length. They are then mapped to the \(2^m\) \(m\)-bit inputs. It will be shown that this simple logic leads to highly efficient fixed-rate codes for large \(m\). More importantly, there is an efficient implementation of this procedure, which does not depend on input block length \(m\).

- As an example, consider \(m + 1 = 8\). Table II lists the 8-bit input word types along with corresponding bit stuff codeword lengths. Input word type \((i, 8 - i)\) refers to 8-bit inputs with \(i\) zeros and \(8 - i\) ones. The \(2^7\) codewords of least length are the

<table>
<thead>
<tr>
<th>Input word type</th>
<th>Number of such input words</th>
<th>Corresponding codeword length</th>
</tr>
</thead>
<tbody>
<tr>
<td>(8,0)</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>(7,1)</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>(6,2)</td>
<td>96</td>
<td>10</td>
</tr>
<tr>
<td>(5,3)</td>
<td>56</td>
<td>11</td>
</tr>
<tr>
<td>(4,4)</td>
<td>70</td>
<td>12</td>
</tr>
<tr>
<td>(3,5)</td>
<td>56</td>
<td>13</td>
</tr>
<tr>
<td>(2,6)</td>
<td>28</td>
<td>14</td>
</tr>
<tr>
<td>(1,7)</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>(0,8)</td>
<td>1</td>
<td>16</td>
</tr>
</tbody>
</table>

ones corresponding to input word types \((8,0), (7,1), (6,2), (5,3)\) and \((4,4)\). With this choice of codewords, the maximum codeword length is 12, and hence we can construct an \(m = 7, n = 12\), \((1, \infty)\) constrained code of rate greater than \(\frac{1}{2}\).

- The block diagram of the fixed-rate \((1, \infty)\) encoder is shown in Fig. 1. It is made up of two parallel branches. The encoder accepts an \(m\)-bit input word and generates \((1, \infty)\) constrained sequences of length \(n_1\) and \(n_2\) in the two branches. The
output word is chosen to be the one that has the smaller length. Hence, \( n = \max \{ \min(n_1, n_2) \} \), where the maximization is over all \( m \)-bit input words. In the example with \( m = 7 \), it can be verified that \( n = 12 \). Typically, \( m \approx \text{few thousands} \).

- The rate \( R_m(1, \infty) \) of the fixed-rate encoder for an \( m \)-bit input can be computed as

\[
R_m(1, \infty) = \begin{cases} 
\frac{m}{2^m + \frac{m}{2}} & \text{when } m \text{ even} \\
\frac{m}{2^m + \frac{m}{2}} & \text{when } m \text{ odd} 
\end{cases}
\]

(1)

It can be seen that \( \lim_{m \to \infty} R_m(1, \infty) = \frac{2}{3} \). This is exactly equal to the \((1, \infty)\) bit stuff average information rate.

- It is now straightforward to extend to all \((d, \infty)\) constraints, \( d > 0 \). The encoder block diagram can be redrawn as in Fig. 2. The mappings \( M_1 \) and \( M_2 \) are specified in Table III. The encoding rate, \( R_m(d, \infty) \), can be computed as

\[
R_m(d, \infty) = \begin{cases} 
\frac{m}{2^{d+2} + \frac{m}{2}} & \text{when } m \text{ even} \\
\frac{m}{2^{d+2} + \frac{m}{2}} & \text{when } m \text{ odd} 
\end{cases}
\]

(2)

Hence, \( \lim_{m \to \infty} R_m(d, \infty) = \frac{2}{3d+2} \), which is identical to the average information rate of the bit stuff algorithm. \((d, d+1)\) codes can be constructed along similar lines.

**Future Work:**

1. Extend the above encoding procedure to all values of \((d, k)\). We expect that similar fixed-rate codes can be designed using \( k - d + 1 \) mappings for any \( 0 \leq d < k < \infty \).
2. Investigate trade-offs between asymptotic rate and number of mappings.
3. Design fixed-rate codes for typical values of output block length, for e.g., \( n = 4096 \).
4. Pursue a more general approach to design fixed-rate codes. Consider the entire codespace made up of concatenated constrained phrases and pick only those codevectors that are shorter than a certain truncation length \( n \). The codebook, in general, might be complex for such a design. However, rates even greater than those of the bit stuff algorithm might be possible. Similar design techniques have been used before for the SVQ [3].
5. Extend to noisy constrained channels and weakly constrained codes.

**REFERENCES**


I. FIXED-RATE ITERATIVE ENCODING FOR $(0, k)$ CONSTRAINTS

We design fixed-rate codes for the $(0, k)$ constraint based on a simple, variable-rate encoding technique called bit stuffing [1]. In previous work, we had reported the design of fixed-rate $(d, \infty)$ and $(d, d+1)$ codes. Here, we extend similar ideas to $(0, k)$ constraints. In related prior work by Jin et al. [4], fixed-rate bit stuff codes were considered for weak $(0, k)$ constraints. There are significant differences in our approach in that the constraints are not weak, and we build fixed-rate codes iteratively.

The bit stuff algorithm [1] generates a $(0, k)$ constrained sequence from an arbitrary binary sequence by simply inserting a one after every run of $k$ consecutive zeros. It is well known that $(0, k)$ constrained sequences can be described as a concatenation of independent phrases from the set $V = \{0^i1, 0^21, \ldots, 0^{k-1}1 \}$, where $0^i$ denotes a run of $i$ consecutive zeros [3]. For convenience, let us further denote the phrase $0^i1$ by $v_i$, $i = 0, 1, \ldots, k$. Hence, the bit stuff algorithm induces a reversible mapping from input words, $u_i$, to $(0, k)$ constrained phrases, $v_i$, such that

$$u_i = \begin{cases} v_i & \text{for } i = 0, 1, \ldots, k - 1 \\ 0^k & \text{for } i = k \end{cases}$$

Such an encoding is said to be variable-rate because different input sequences of the same length can give rise to constrained sequences of fluctuating lengths. Such rate fluctuations are unacceptable in practice and thus far, the interest in bit stuffing has been mainly theoretical. However, there are two very 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 stream 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 independent, identically distributed (i.i.d) and unbiased ($Pr\{0\} = Pr\{1\} = 0.5$), the average information rate of the bit stuff algorithm, $R^*(k)$, can be expressed as [1]

$$R^*(k) = \frac{2^{k+1} - 2}{2^{k+1} - 1}$$

For typical values of $k \geq 5$ used in practice, the capacity $C(k)$ is well-approximated by [2]

$$C(k) \approx 1 - \frac{2^{-k}}{4 \ln 2}$$

(2)

From (1) and (2), it can be seen that $\frac{R^*(k)}{C(k)}$ is very close to unity. In the remainder of this paper, we describe how to remedy the variable-rate nature of bit stuffing for $(0, k)$ constraints.

A. Iterative fixed-rate algorithm

The algorithm accepts an unconstrained binary input block, $x = (x_0, x_1, \ldots, x_{m-1})$, of length $l(x) = m$ bits and outputs a $(0, k)$ constrained binary sequence, $y = (y_0, y_1, \ldots, y_{n-1})$, of length $l(y) = n$ bits. The following notations and definitions will be used in the algorithm.

Notations and Definitions:

- Denote by $u_i^*$, the word $0^{i+1}$, $i = 0, 1, \ldots, k - 1$. As before, $v_i$ denotes the $(0, k)$ constrained phrase $0^i1$ and $u_i$, the corresponding bit stuff input word, $i = 0, 1, \ldots, k$. Hence, by definition, $u_i^{*k-1} = u_k$.
- The weight of a binary vector $s$ with respect to the input word $u_i^*$, denoted by $w_{u_i^*}(s)$, is the number of occurrences of $u_i^*$ in vector $s$, with $s$ being scanned as a concatenation of words from the set $\{u_0, u_1, \ldots, u_i, u_i^*\}$, $i = 0, 1, \ldots, k - 1$.
- The weight of vector $s$ with respect to the input word $u_k$, denoted by $w_{u_k}(s)$, is the number of occurrences of $u_k$ in vector $s$, with $s$ being scanned as a concatenation of words from the set $\{u_0, u_1, \ldots, u_k, u_k^*\}$, $i = 0, 1, \ldots, k - 1$.
- Denote by $s(i)$, the vector formed by flipping the bits in $s$ that follow a string of $i$ consecutive zeros, $i = 0, 1, \ldots, k - 1$.
- Denote by $s_1 \| s_2$, the concatenation of two vectors $s_1$ and $s_2$. 
Denote by $i(x)$ the index vector corresponding to input vector $x$, and by $\delta^t$ a string of $t$ dummy bits. The index bits are used to convey to the decoder whether or not bits are flipped in each of the $k$ iterations. The dummy bits are used to pad the output up to length $n$ bits, and are ignored during decoding.

* Denote by $C_2(s)$, the bit stuff encoding of a vector $s$.

The encoding algorithm is described next.

**Algorithm:**

1. Input $x$. Set $x_0 = x$.
   
   Scan $x_0$ as a concatenation of words from the set $\{u_0, u_0^0\}$.
   
   If $w_{u_0}(x_0) < w_{u_0^0}(x_0)$, output $x_1 = x_0(0)$.
   
   Else output $x_1 = x_0$.

2. For $j = 2$ to $k - 1$,
   
   Input $x_{j-1}$.
   
   Scan $x_j$ as a concatenation of words from the set $\{u_0, u_1, \ldots, u_{j-1}, u_{j-1}^0\}$.
   
   If $w_{u_0^j}(x_j) < w_{u_{j-1}^0}(x_{j-1})$, output $x_j = x_{j-1}(j - 1)$.
   
   Else output $x_j = x_{j-1}$.

3. Input $x_{k-1}$.
   
   Scan $x_k$ as a concatenation of words from the set $\{u_0, u_1, \ldots, u_{k-1}, u_{k-1}^0\}$.
   
   If $w_{u_{k-1}}(x_{k-1}) < w_{u_{k-1}^0}(x_{k-1})$, output $x_k = x_{k-1}(k - 1)$.
   
   Else output $x_k = x_{k-1}$.

4. Output $y = C_2(x_k) || C_2(i(x)) || \delta^{n-n'}$, where $n' = l(C_2(x_k)) + l(C_2(i(x)))$.

**B. Rate Computation**

The encoding rate of the iterative algorithm is given by $R(k,m) = \frac{m}{n}$, where the output block length $n$ is given by

$$n = \max_{x \in Z_m^n} \{l(C_2(x)) + l(C_2(i(x)))\}$$

(3)

$$n = \max_{x \in Z_m^n} \{l(C_2(x)) + l(C_2(i))\}$$

(4)

where $Z_m^n$ is the set of all binary $m$-tuples, and $i^*$ denotes the maximum-length index vector. Recall that the index vector, $i(x)$, is used to convey to the decoder whether or not the flipping operation is performed in each of the $k$ iterations, i.e., whether $x_t = x_{t-1}(t - 1)$ or $x_t = x_{t-1}$, for $i = 1, 2, \ldots, k$. Hence, we expect that the maximum-length index vector, $i^*$, grows with $k$, but is independent of $m$ for given $k$.

We now proceed to compute the term $\max_{x \in Z_m^n} \{l(C_2(x))\}$ in (4). This is nothing but the maximum length of the bit stuff encoding of vector $x_k$, which in turn is derived from the iterative process on the input $x$. The bit stuff algorithm inserts a one after every string of $k$ consecutive zeros in $x_k$. Hence, we have

$$\max_{x \in Z_m^n} \{l(C_2(x))\} = \max_{x \in Z_m^n} \{m + w_{u_k}(x_k)\}$$

(5)

$$= m + \max_{x \in Z_m^n} \{w_{u_k}(x_k)\}$$

(6)

Substituting (6) in (4), we find that the encoding rate $R(k,m)$ is given by

$$R(k,m) = \frac{m}{m + \max_{x \in Z_m^n} \{w_{u_k}(x_k)\} + l(C_2(i^*))}$$

(7)

**Ongoing Work:**

1. Complete the above rate computations by evaluating the term $\max_{x \in Z_m^n} \{w_{u_k}(x_k)\}$. Our main interest will be to evaluate $R(k,m)$ for large values of input block length $m$. Here, we can make the reasonable assumption that the index term $l(C_2(i^*))$ in (7) grows only with the number of iterations (equal to $k$), and is independent of $m$.

2. Evaluate the asymptotic rate $\lim_{m \to \infty} R(k,m)$ and compare with the bit stuff average information rate $R^*(k)$. Initial results suggest a very small difference. For the particular case of $k = 2$, we have found that $\lim_{m \to \infty} R(2,m) = R^*(2) = \frac{1}{2}$.

3. Evaluate the DC content in the codeword spectrum, preferably by analysis. Integrate an adequate DC balancing component into the iterative encoding algorithm.

**References**


Summary
During this quarter we worked on the following items:

- Rate computations for the proposed fixed-rate \((0, k)\) encoding algorithm.
- Paper submission of \((0, k)\) encoding results to ISIT'05.
- Initial ideas on extension to all \((d, k)\).

In the previous report, we had described an iterative fixed-rate encoding procedure for \((0, k)\) constraints. We now provide details on rate analysis for the proposed algorithm.

I. RATE COMPUTATIONS FOR THE FIXED-RATE \((0, k)\) ITERATIVE ENCODING ALGORITHM

For an input block length \(m\) bits, the encoding rate of the iterative algorithm is given by

\[
R(k, m) = \frac{m}{m + \max_{x \in \mathbb{Z}^n} \{w_{uk}(x_k)\} + l(B(\alpha^*))}.
\]

Our main interest will be to evaluate \(R(k, m)\) for large values of input block length \(m\). Since the index term \(l(B(\alpha^*))\) is assumed to be independent of \(m\), we find that for given \(k\), \(R(k, m)\) directly depends on only the maximum number of distinct \(u_k\) words in the vector \(x_k\) of length \(m\) bits. Furthermore, the following two computations are equivalent for large \(m\)

\[
\frac{1}{m} \max_{x \in \mathbb{Z}^n} \{w_{uk}(x_k)\} \equiv \min_{w_{uk}(x_k) \in \mathbb{Z}^+} \left\{ \frac{m}{w_{uk}(x_k)} \right\},
\]

where \(\mathbb{Z}^+\) is used to denote the set of all positive integers. We will find it easier to work with the latter expression.

Hence, the problem of computing rate has now been reduced to finding the minimum of the ratio \(\frac{m}{w_{uk}(x_k)}\) over all non-zero values of the weight \(w_{uk}(x_k)\). Unfortunately, the complexity of the search for this minimum is still quite high. This is illustrated with the help of the binary tree in Fig. 1, where the root node denotes the input vector \(x\) and the \(2^k\) leaf nodes represent vectors \(x_k\), each corresponding to a possible path through the iterative algorithm. The result of every iteration, \(i = 1, 2, \ldots, k\), is denoted either by \(F\) or \(F\), to indicate \(x_i = x_{i-1}(i - 1)\) or \(x_i = x_{i-1}\), respectively. Due to the complexity of this search, we do not proceed with the exact rate computation. Instead, we derive upper and lower bounds on the encoding rate.

![Fig. 1. Binary search-tree to determine \(\min_{w_{uk}(x_k) \in \mathbb{Z}^+} \left\{ \frac{m}{w_{uk}(x_k)} \right\}\). The path \(P_1\) marked in bold is the one traversed to determine an upper bound on the encoding rate.](image)

An upper bound on the encoding rate can be obtained by traversing the path \(P_1\), marked in Fig. 1. We only provide a brief sketch here. The complete derivation with proofs can be found in [3].

**Definition 1:** For a given constraint parameter \(k\), the effective weight, \(a^k_i\), of the word \(0^i1\), \(i \in \mathbb{Z}^+\), is the minimum number of zeros in \(x\) resulting from the presence of a single \(0^i1\) word in the vector \(x_k\), with \(x_k\) being scanned as a concatenation of independent words \(0^{i-1}, t \in \mathbb{Z}^+\).

**Lemma 1:** The effective weights, \(a^k_i\), \(i \geq 2\), are related by the recursion

\[
a^k_i = \sum_{j=1}^{\min\{i,k\}-1} \left( w_{uj}(0^j) - w_{uj}(0^{j+1}) \right) a^j_i + w_{uj}(0^{i+1})
\]
From (2) and Definition 1, we find that \( \min_{\omega_{u_k}(x_k) \in \mathbb{Z}^+} \left\{ \frac{m_{\omega_{u_k}(x_k)}}{\omega_{u_k}(x_k)} \right\} \leq \min_{i \geq k} \left\{ \frac{2a_i^{+}}{1/k} \right\} \). For convenience, we denote \( f_i^k = \frac{2a_i^{+}}{1/k} \). Our goal now is to compute for a given \( k \), the minimum of \( f_i^k \) over all \( i \geq k \). We derive and use the following result.

**Theorem 1:** For any given \( k \), \( 1 \leq k < \infty \),

\[
\min_{i \geq k} \{ f_i^k \} = \min_{k \leq i \leq k+b(k)-1} \{ f_i^k \},
\]

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

Finally, we have \( R_u(k) = \frac{\sum_{i=1}^{f^*(k)}}{f^*(k)} \), where \( f^*(k) = \min_{i \geq k} \{ f_i^k \} \) is found by computer search over a reduced range according to Theorem 1.

In addition, a lower bound can be found as \( R_l(k) = \frac{2^{k+1} - 2^{k-1}}{2^{k+1} - 2^{k-1} + 1} \). Details are provided in [3]. Our results are summarized in Table 1.

### Table 1

**Summary of Rate Computations for the Iterative Encoding Algorithm**

<table>
<thead>
<tr>
<th>( k )</th>
<th>( \min_{i \geq k} { f_i^k } )</th>
<th>Minimizing value of ( i ) ( (i^*) )</th>
<th>Upper bound on asymptotic rate ( R_u(k) )</th>
<th>Lower bound on asymptotic rate ( R_l(k) )</th>
<th>Bit stuff average rate ( R^*(k) )</th>
<th>( R_u(k) ) (%)</th>
<th>( R_l(k) ) (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>100</td>
<td>96.02</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>100</td>
<td>97.49</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>3</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>100</td>
<td>97.49</td>
</tr>
<tr>
<td>4</td>
<td>24</td>
<td>5</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>100</td>
<td>98.43</td>
</tr>
<tr>
<td>5</td>
<td>54</td>
<td>5</td>
<td>54</td>
<td>54</td>
<td>54</td>
<td>100</td>
<td>99.13</td>
</tr>
<tr>
<td>6</td>
<td>114</td>
<td>7</td>
<td>114</td>
<td>114</td>
<td>114</td>
<td>99.54</td>
<td>99.13</td>
</tr>
<tr>
<td>7</td>
<td>240</td>
<td>7</td>
<td>240</td>
<td>240</td>
<td>240</td>
<td>99.75</td>
<td>99.54</td>
</tr>
<tr>
<td>8</td>
<td>480</td>
<td>11</td>
<td>480</td>
<td>480</td>
<td>480</td>
<td>99.97</td>
<td>99.76</td>
</tr>
<tr>
<td>9</td>
<td>984</td>
<td>11</td>
<td>984</td>
<td>984</td>
<td>984</td>
<td>99.93</td>
<td>99.88</td>
</tr>
<tr>
<td>11</td>
<td>4020</td>
<td>11</td>
<td>4020</td>
<td>4020</td>
<td>4020</td>
<td>99.99</td>
<td>99.99</td>
</tr>
<tr>
<td>12</td>
<td>8052</td>
<td>17</td>
<td>8052</td>
<td>8052</td>
<td>8052</td>
<td>99.99</td>
<td>99.99</td>
</tr>
<tr>
<td>13</td>
<td>16242</td>
<td>17</td>
<td>16242</td>
<td>16242</td>
<td>16242</td>
<td>99.98</td>
<td>99.98</td>
</tr>
<tr>
<td>14</td>
<td>32496</td>
<td>17</td>
<td>32496</td>
<td>32496</td>
<td>32496</td>
<td>99.99</td>
<td>99.99</td>
</tr>
<tr>
<td>15</td>
<td>65226</td>
<td>17</td>
<td>65226</td>
<td>65226</td>
<td>65226</td>
<td>99.99</td>
<td>99.99</td>
</tr>
</tbody>
</table>

**Ongoing Work:**

1. Extend the encoding product to all \((d, k)\). Since \((d, k)\) bit stuffing is a two step process, the first of which produces a \((0, k - d)\)-constrained sequence, the above analysis on \((0, k)\) encoding will provide useful starting points.

2. Initial attempts at DC suppression for the generated \((0, k)\) sequences have been aimed at suitably altering the pre-processing. However, this has turned out to be very complex. Hence, we will investigate ad-hoc DC suppression schemes that go with the proposed iterative encoding.

3. Design fixed-rate \((0, k)\), \((1, k)\) and \((2, k)\) codes with practical parameter values, e.g., output length 4096 bits, adequate DC suppression. This will also include the effect of index bits, which could be overlooked for the asymptotic analysis thusfar.

### References

