RICE UNIVERSITY

INTERLACING IN SEQUENTIAL SYSTEMS

by

Bowman Markle Miller

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

Thesis Director's signature:

Paul E. Grefier

Houston, Texas

November, 1965
ABSTRACT

The application of a bit interlacing or time-spread coding to a class of convolutional codes is considered. Investigation is made into the bit spread distances required to facilitate the decoding of various maximum burst length errors. Results of a simulation, obtained using an I.B.M. 7040 digital computer, are presented. The channel model used in the simulation is one derived from the binary symmetric channel, but which provides for the generation of bursts of errors. The error burst lengths are uniformly distributed from one to a variable maximum burst length.
<table>
<thead>
<tr>
<th>SECTION</th>
<th>TITLE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ACKNOWLEDGEMENTS</td>
</tr>
<tr>
<td>I.</td>
<td>INTRODUCTION</td>
</tr>
<tr>
<td>II.</td>
<td>CODE PROPERTIES</td>
</tr>
<tr>
<td>III.</td>
<td>DECODING</td>
</tr>
<tr>
<td>IV.</td>
<td>BIT INTERLACING</td>
</tr>
<tr>
<td>V.</td>
<td>THE SIMULATION AND IIS PROGRAM</td>
</tr>
<tr>
<td>VI.</td>
<td>RESULTS AND DISCUSSION</td>
</tr>
<tr>
<td></td>
<td>BIBLIOGRAPHY</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

There are many to whom my sincerest thanks are extended. My advisor Professor Paul E. Pfeiffer has given inestimable assistance and guidance. I am deeply indebted to him for his apparently limitless patience and concern for those of us who have the privilege to study under him.

To Dr. Shu Lin, whose work provided the basis for this investigation, I give my most appreciative thanks. These thanks also are due to Dr. W.H. Lyne for the computer programs, channel model, and advice which he made available.

My thanks to my fellow graduate students W.H. Ng, C.Y. Ong, and George Paul for the many hours of profitable discussion and for some not so profitable.

I give my loving thanks to my wife Susan for her support and encouragement, for the many evenings with no talking company but the television set, and for cheerfully typing this thesis in its several versions.

This work was supported by the U.S. Army Research Office--Durham.
SECTION I

Introduction

This paper discusses the effectiveness of an interlacing scheme for combating bursts of errors in a sequential coding-decoding system. A particular class of sequential codes was tested under varied noise conditions. The tests were done by simulation using an I.B.M. 7040 digital computer.

The interlacing may be described as a time-spread coding which is applied to the sequentially encoded signal prior to transmission. Upon reception the signal is put back into the sequence acceptable to the sequential decoder. The purpose of this reordering of the encoded signal is to cause adjacent bits of the encoded signal to be separated by a given number of digits when errors are encountered in the channel. Bursts of errors thus tend to be broken into shorter bursts, possibly single errors, in the unlaced received sequence. This is an advantage in terms of cost of implementation, since codes thus far developed for decoding "random" error patterns require less complicated equipment than those designed to combat bursts of errors. Fein, et al (1) have applied interlacing to several block codes on actual, high-frequency radio circuits with some success.

The class of sequential codes considered is one developed from the Wozencraft & Reiffen (2) convolutional codes by Lin and Pfeiffer (3 & 4). In this class, the minimum Hamming distance between the upper and lower halves of the code tree is maximized at each stage. This property can be considered as
analogous to the minimum distance between code vectors in a systematic block code (4).

The model used for the channel noise was one used by Lyne (5). The channel is basically a Binary Symmetric Channel (BSC). Once an error is initiated by the BSC, however, a uniform distribution is sampled to determine the burst-length. Lyne refers to this model as the Binary Symmetric Burst Channel (BSBC).
SECTION II

Code Properties

Convolutional tree codes are the object of this study of the effectiveness of bit interlacing in combating burst-error noise. The type of code considered is that generated by a single generator sequence. For this type of code, a single binary message digit is encoded into $\alpha$ channel input digits. Thus the rate of transmission is $R = 1/\alpha$. Consider the generator sequence $g$ to have the form:

$$g = g(1)g(2)g(3) \ldots g(\nu)g(0)\ldots,$$

(2-1)

where $g(0)$ is the segment consisting of $\alpha$ zeros. Figure 1 shows a general tree constructed out to four stages. The generator sequence is indicated by the asterisks.

The decision rule to select a path through the tree to encode binary messages is a simple one. At any node in the tree, encode a zero message digit by selecting the upper branch segment stemming from that node. In order to encode a message digit one the lower branch segment is selected. This is the same as convolving the message sequence past the generator sequence. A specific example for an $\alpha = 3$ code is shown in Figure 2.

Tree codes constructed in the manner of Figure 1 have a number of useful properties which are developed in detail in two papers by Pfeiffer, Lin, and Lyne (4 & 6) and in the Ph.D. Dissertations of the latter two (5 & 7). Some of these properties are now listed in abbreviated form without proof.

1.) The codes are linear or group codes.
Single Generator Convolutional Code Tree

FIGURE 1.
For $V = 4$ all generator segments past the fourth one will be zero.

$\alpha = 3, \, \gamma = 4, \, m = 01101, \, u = 000 \, 111 \, 110 \, 011 \, 111$

FIGURE 2.
2.) The minimum Hamming distance between the upper and lower halves of a truncated (finite) tree is dependent only upon the length of the truncated tree considered. The minimum distance does not depend upon the position of the truncated tree within the infinite tree. This property is called the fundamental distance property of the codes. To facilitate the use of this property, it is helpful to introduce a distance function \( d(.) \). Consider a truncated tree \( k \) stages long, that is, a tree in which each of the paths is \( k \times \alpha \) channel digits long. This tree will be called a \( k \)-unit. The minimum distance between the upper and lower halves of a \( k \)-unit is \( d(k) \).

3.) In the infinite tree, the tree consisting of the first stages is called the initial truncated tree. Figure 1 is the initial truncated tree for a code having \( \nu = 4 \). In the initial truncated tree and any other truncated tree containing the zero path as the uppermost path, the minimum distance between the upper and lower half trees is just the weight of the path of least weight in the lower half tree.

4.) If two sequences of encoded digits in a tree are the same for \( \alpha(\nu -1) \) digits, the trees stemming from the node following the last encoded digit are identical.

The minimum distance between half trees is of the same importance to sequential decoding that the minimum distance between code vectors is to block decoding. By increasing the minimum distance between half trees in the sequential code the error correcting capability is increased. S. Lin (3) has found a way of selecting the generator sequence segment-by-segment so as to
maximize the minimum distance at each stage in the tree using the segments already chosen. This does not guarantee an optimal choice of sequence of a given length. It is quite possible that the optimal choice of sequence for some constraint length will have an intermediate stage at which the minimum distance is not maximized.
SECTION III

Decoding

The basic assumption made in decoding the sequential codes considered in the preceding section is that the noise added to the transmitted sequence in the channel is low enough so the received sequence may be decoded correctly. More explicitly, this assumption takes the form of assuming that the path in the tree, whose Hamming distance from the received sequence is least, is the correct path.

The decoder is assumed to hold $n = \alpha \gamma$ channel output digits from the received sequence for comparison with a truncated tree within the code's infinite tree. A tentative path in the tree is selected to be the most probably correct one in the sense that its Hamming distance is least. The first branch segment, the earliest in time present in the decoder, is then accepted finally as being the correct one and is decoded. The accepted branch is shifted from the decoder and a new segment of the received sequence is shifted into the decoder at the other end. The truncated tree with which the received sequence is compared is now one stemming from the node at the end of the last decoded branch. A tentative path has already been selected for $\gamma - 1$ branches, leaving only the choice between two branch segments remaining. The one closest to the corresponding received sequence branch is chosen. The stage by stage optimal codes always have a distance between two single branches $d(1) = \alpha$. Thus, if the error disturbances create errors in one branch in less than $\alpha/2$ digits,
the branch-by-branch procedure will always choose the correct path. If the noise is greater than $\alpha/2$ an incorrect decision will be made.

This procedure of selecting the tentatively accepted path one branch at a time using $d(l)$ is called the branch-by-branch search. The branch-by-branch search procedure is for use during relatively low noise situations, when the full error correcting capability of the code is not needed. There are several $k$-unit search procedures for higher noise situations which allows searching all possible paths up to $\mathcal{V}$ branches long in the truncated tree which the decoder holds at any given time. The branch-by-branch procedure is used until an error is made; then the broader tree search procedure is begun, since there is no way in which the branch-by-branch procedure can get back on the correct path.

There are two quantities available to the decoder which, used together, can indicate an error has been made. They are the distance properties of the code, in the form of the distance function $d(k)$, and the distance $d(w,v)$ between the received path $v$ and the tentatively accepted path $w$. A threshold function $T(.)$ may be defined as an integer function, which for any positive integer $k$ takes on the value $T(k)$,

$$T(k) < d(k)/2 \leq T(k) + 1. \quad (3-1)$$

$T(k)$ is just the error correcting capability of the code over $k$ branch segments. If $d(w,v)$, as defined above, is greater than $T(k)$ over the corresponding $k$ branches, an incorrect path must have been chosen at some previous time. A search of every path in the $k$-unit should be started to find a more correct one.
It is natural, then, that before the earliest branch in the tentatively accepted path is decoded finally, a check should be made to make sure relation (3-2) holds.

\[ d(w, v) < T(V) \]  

(3-2)

This comparison is called the **Threshold Test**. For large \( y \), it is evident that an error might be shifted several branches into the decoder before \( T(V) \) is violated. For this reason threshold tests are made at several different places from the last of the tentatively accepted path to the earliest branch in time in the decoder. This is termed the **Multiple Threshold Test** or MTT. When one or more thresholds of the MTT are violated, a search is begun of the \( k \)-unit stemming from the node at which the highest threshold is violated. In the search, the weight of each path in the \( k \)-unit is compared with the violated threshold. The path whose distance from the received sequence is least is selected and again compared with the threshold \( T(k) \). If the threshold is violated once again, a larger unit is searched, a \( k+1 \)-unit stemming from the node prior to the one at which the threshold is violated. A search is made of this \( k+1 \)-unit and comparison with the new threshold \( T(k+1) \). The backup and search procedure is continued until the appropriate threshold is no longer violated and the branch-by-branch search procedure is again begun from the end of the path chosen by the \( k+1 \)-unit search. Thresholds may be violated all the way back to and including \( T(V) \), in which case there are two alternatives. Either the decoder may give up, signal a decoding error has been made, and some restart procedure begun to pick up decoding at a later segment of the message; or the path closest to the received path be accepted and the
branch-by-branch search again be attempted from the end of this
new path. A combination of the two alternatives is possible.
The threshold $T(\gamma)$ might be allowed to be violated $m$ times
consecutively before the decoder gives up. It was this sort of
option which was used in our simulation. The paper of Lin and
Pfeiffer (4) may be consulted for a possible restart procedure
to be used when the decoder does give up its search, and for a
more detailed description of decoding.

W.H. Ng (8) has studied means by which the number of compu-
tations in the search may be reduced so the decoder is more ef-
ficient. Ng's study has indicated that certain portions of some
trees, under a given set of threshold violations, need not be
searched at all as they can not possibly contain the correct path.
SECTION IV

Bit Interlacing

In the two previous sections the coding-decoding system has taken the form shown in Figure 3. In it a message from some source is encoded, passed through a noisy channel, and decoded for use in some message sink. In this section the system in Figure 3 is adapted to that shown in Figure 4. The addition of the interlacing and unlacing may be viewed as an addition to the channel by the system in Figure 3, or it may be considered, as shown in Figure 4, as a secondary encoding and decoding which does not change the rate of transmission from $R = 1/\alpha$.

The bit interlacing or spreading technique is a simple one. Its use with sequential codes is similar to that with block codes. Let us illustrate with examples using sequential codes.

Suppose the code is one with $\alpha = 3$ and we wish adjacent bits of the encoded sequence to be BS bits apart in the channel. This BS bit distance separating adjacent encoded bits is termed the bit spread distance or simply the bit spread (BS). For this example let us choose a short bit spread of 5 bits. Assume the output of the encoder is $u$,

$$u = a_1b_1c_1a_2b_2c_2a_3b_3c_3a_4b_4c_4a_5b_5c_5 \cdots,$$

(4-1)

where $a_j$, $b_j$, and $c_j$ represent binary integers in the first, second, and third positions of the $j$th branch segment respectively. The sequence $u$, after interlacing, becomes $t_4$,

$$t_4 = a_1a_2a_3a_4b_1b_2b_3b_4b_5c_1c_2c_3c_4c_5 \cdots.$$

(4-2)

In the channel the sequence $t_4$ is disturbed by noise, becoming $t_0$. 
FIGURE 3.

FIGURE 4.
which has the form below, should the first 5 bits be changed by noise.

\[ t_0 = \overline{a_1 b_2 c_3 a_4 b_1 b_2 b_3 b_4 c_1 c_2 c_3 c_4 c_5} \ldots \] (4-3)

The super-bars in the sequence \( t_0 \) indicate the one's complement notation for binary errors (1's changed to 0's and vice versa). Unlacing \( t_0 \) puts it into the form of the received sequence \( v \),

\[ v = \overline{a_1 b_1 c_1 a_2 b_2 c_2 a_3 b_3 a_4 b_4 c_4 a_5 b_5 c_5} \ldots \] (4-4)

The received sequence has one error per branch segment, and is therefore correctable with the branch-by-branch search of an \( \alpha = 3 \) code. If \( v = 10 \), the primary threshold \( T(10) = 5 \) would just be met, and the correct decision would be made, assuming that no additional errors were made in the other five branch segments in the decoder at the time.

Let us consider the above example again, but with some changes in the conditions. Assume the bit spread is increased to 10 and that a burst of 10 consecutive errors is encountered in the channel. In this case \( v \) becomes

\[ v = \overline{a_1 b_1 c_1 a_2 b_2 c_2 a_3 b_3 a_4 b_4 c_4 a_5 b_5 c_5} \ldots \overline{a_{10} b_{10} c_{10}}. \] (4-5)

All the \( a_i, 1 \leq i \leq 10 \), in this sequence would be incorrect. The branch-by-branch search would select the correct path if the search were allowed to continue through all ten branches. However, the primary threshold would be violated by 5 and the tree search with high probability would choose a different path whose distance from the received sequence is less than that of the branch-by-branch path. Unfortunately, the decoder cannot recognize that this type of threshold violation is different from one in which the branch-by-branch search has made an incorrect choice of path due to two errors in a single branch segment.
The sort of error pattern shown in (4-5) could have occurred just as readily if the bit spread had been greater, but then there would be higher probability that further errors would be introduced into the sequence in equation (4-5). This type of error sequence points up a weakness in the interlacing scheme, as presented above and which is present in the test runs made with the \( \alpha = 3 \) code. Not only should adjacent bits in the encoded sequence be some BS bits apart in the channel, adjacent errors in the channel must spread for enough apart in the received sequence \( \mathbf{v} \) so that the error correcting capability of the code is not exceeded. If, instead of interlacing one branch segment of the encoded sequence at a time, three branch segments had been treated as one nine bit word, then the received sequence should have the burst of ten errors spread out in such a manner that three or four errors would be present in each ten consecutive branches in the decoder. By the way of example let us interlace equation (4-1) in this manner with a BS = 10 and assume 10 consecutive errors affect the first ten transmitted digits.

\[
\begin{align*}
\mathbf{u} &= a_1b_1c_1a_2b_2c_2a_3b_3c_3 \cdots a_7b_7c_7a_8b_8c_8a_9b_9c_9 \cdots \\
\mathbf{t}_o &= \overline{a_1}a_4a_7a_{10}a_{13}a_{16}a_{19}a_{22}a_{25}a_{28}b_{14}b_{17}b_{20}b_{23}b_{26}b_{29}b_{32}b_{35}b_{38} \\
&\quad c_1c_4 \cdots c_{28}a_2a_5 \cdots a_{29}b_2 \cdots b_{29}c_2 \cdots c_{29}a_3a_6 \cdots a_{30} \\
&\quad b_3b_6 \cdots b_{30}c_3 \cdots c_{30} \cdots \\
\mathbf{v} &= \overline{a_1}a_4a_7a_{10}a_{13}a_{16}a_{19}a_{22}a_{25}a_{28}b_{14}b_{17}b_{20}b_{23}b_{26}b_{29}b_{32}b_{35}b_{38} \\
&\quad \overline{a_{10}}b_{10}b_{13}b_{16}b_{19} \cdots \overline{a_{16}}b_{16}b_{19} \cdots \overline{a_{19}}b_{19}b_{19} \cdots \\
&\quad \overline{a_{22}}b_{22}c_{22} \cdots \overline{a_{25}}b_{25}c_{25}c_{25} \cdots \overline{a_{28}}b_{28}c_{28}c_{28} \cdots
\end{align*}
\]  

Thus a burst of ten errors has been spread out in the received sequence \( \mathbf{v} \) to one error every third branch segment. This will cause at most four errors to be in the decoder at any one time,
allowing the remaining error correcting capability of one error in each ten segment sequence to correct random errors or error bits due to other bursts which may also affect the first thirty segments of \( v \).

Some codes possess adequate error correcting capability to allow interlacing single branch segments instead of several branches. An example of such a code is an \( \alpha = 5, \nu = 10 \) code. The branch-by-branch search using an \( \alpha = 5 \) code can correct two errors in each branch five bits in length. The primary threshold, \( T(10) = 10 \), allows the correction of the one error per branch pattern which frequently occurs when error bursts are spread by interlacing over a single branch. For a code such as this, or one in which several branches are interlaced as one code word, a single error burst of any prescribed length may be made correctable by making the bit spread equal to or greater than the prescribed length.

Encoded sequences with interlacing have the property that what is usually termed the guard space \((1)\), the distance required between error bursts for both of them to be corrected, may shrink to zero.

As an example of zero guard space, consider an \( \alpha = 5, \nu = 10 \) code with BS = 3, and assume an error burst of 6 consecutive errors occurs. Assume \( u \) has the form

\[
\begin{align*}
  u &= a_1 b_1 c_1 d_1 e_1 a_2 b_2 c_2 d_2 e_2 a_3 b_3 c_3 d_3 e_3 \cdots a_6 b_6 c_6 d_6 e_6 \cdots \\
  \text{(4-7)}
\end{align*}
\]

where \( a_j, b_j, c_j, d_j, \) and \( e_j \) are again binary position indicators in the \( j \)th branch segment.
In this manner a burst twice as long as the bit spread is made correctable. This example is in some respects trivial since an error burst of 6 in ten branch segments would be corrected by a tree search. However, if the bit spread had been 50, the error burst 100 bits in length and located over \( e_1 \ldots e_{50} a_{50} \), the error burst would be correctable by the branch-by-branch search and no thresholds would have been violated.

From these examples it may be seen that bit interlacing can spread bursts at least as long as the bit spread in such a manner that the bursts are correctable with a minimum of tree search, if not by the branch-by-branch search alone. In a code such as the \( \alpha = 3, \nu = 10 \) code, bit spreading must be over several branch segments for the interlacing to increase significantly the error correcting capability of the code.

Just as bit interlacing can spread consecutive errors apart, it can cluster or bunch otherwise correctable random errors into incorrectable bursts. Take as an example an \( \alpha = 3, \nu = 10 \) code, a bit spread some number several times \( \nu \), like 30, and bit interlace over only one branch.

\[
\begin{align*}
t_0 &= a_1 a_2 a_3 a_4 a_5 a_6 \ldots a_30 b_30 c_30 \ldots \\
v &= a_1 b_1 c_1 d_1 a_2 b_2 c_2 d_2 e_2 a_3 b_3 c_3 d_3 e_3 a_4 b_4 c_4 d_4 e_4 a_5 b_5 c_5 d_5 e_5 \\
\end{align*}
\]

Here two sets of three errors, three bits apart, each with 30
bits between errors of the same set, have bunched together to become two bursts of three close enough together to cause \( T(V) = T(10) = 5 \) to be violated.

The decision to use an interlacing scheme thus depends upon the conditions in the channel in which it is used. If bursts of noise are much more probable than randomly distributed errors, it is more probable that errors will be spread and not clustered and interlacing would be of value.

Further discussion of interlacing, especially as it applies to block codes, may be found in reference (1), a paper by Fein, et al.
SECTION V

The Simulation And Its Program

The simulation of sending an interlaced, sequentially encoded message through a noisy channel and the decoding of the resulting sequence was done using the Rice Computation Laboratory I.B.M. 7040. The programming done to accomplish the simulation was divided into two parts. The first program simulated all the system up to and including the unlacing of the noisy received sequence (see Figure 4 in Section IV). The second program decoded the unlaced message sequence, compared the result with the original message, and printed out an error sequence with a number of calculated parameters and probabilities for each run.

The basic format of the program was to send a sequence of 10,080 message digits, an array of 280 machine words, each of length 36 bits, through the encoder, interlacer, channel, etc. The unlaced sequence was then put in a format acceptable to the decoder and stored on magnetic tape. The program was looped through to send as many blocks of 280 message words as was desired for each set of code parameters. The decoding program treated each of these blocks as a separate message except that some statistics were cumulative for each parameter set.

All programs, with the exception of the input-output programs, were written in the I.B.M 7040's assembly program language MAP. Input-output programs were primarily in FORTRAN.
The Main Encoder-Channel Program--STRBOS.

STRBOS is primarily a sequencing control program calling as subroutines separate programs for each function. Figure 5 is an approximate flow chart of STRBOS. The program is capable of encoding codes for $\alpha$ as great as 12 and $\nu$ as great as 16. With slight input format modification codes with $\nu$ as great as 36 could be encoded. Let us consider some of the subroutine programs in more detail.

Data Input and Run Parameters.

READRE is a FORTRAN program to read data cards containing the basic run parameters. These are the code parameters, $\alpha$, $\nu$, the generator sequence, the minimum weights between upper and lower halves of the tree at each stage up to $\nu$ stages, and the bit spread; the probability of a 1 in the message sequence; a probability of error in the channel; the block length of each segment of the message, always 280 machine words in the test runs; the block length needed for the encoded message; and the total message length to be run with this set of parameters.

The program NUMBRS then calculates other numbers needed by the other programs, especially the decoder. These are primarily the number of times loops are to be completed and how fully the machine 36 bit words are to be packed, etc.

Message Generation.

The heart of the message generator MESGEN is a pseudo-random number generator based on the Boeing Random Number Generator. This uses the method of congruences to generate a set of numbers distributed uniformly between 0 and 1. If a generated member
Flow Chart for STRBOS

1. Initialize tape unit and re-set program constants.

   2. Read-in data cards. (READRE)

   3. Calculate further run parameters from data. (NUMBRS)

   4. Generate the message. (MESGEN)

   5. Encode the message. (ENCODR)

   6. Interlace encoded message. (SPREDR)

   7. Generate error sequence. (BURN03)

   8. Add noise to message, modulo-two. (ADNOIS)

   9. Unlace noisy message. (SQUEZE)

10. Put output into correct word length for decoder. (WRDPAC)

   11. Encode more message??

       a. Yes

          b. No

12. Read-in more data??

       a. Yes

          b. No

STOP

Flow Chart for STRBOS.

Figure 5.
of this set is less than a decimal fraction given originally on a program data card, a message digit 1 is generated. Otherwise a message digit 0 is generated. For all test runs the generation of a 0 or a 1 was equally likely.

The Message Encoder.

The program ENCODR encodes the message sequence generated by MESGEN using the method of Sub-Generator Sequences due to J.J. Busgang (9). Using this method the generator sequence $g$, 

$$g = g(1) g(2) \ldots g(\nu) g(0) \ldots,$$

is divided into sub-generator sequences. To facilitate the description of the encoding, we introduce a different notation for $g$ to show each of the segments $g(i)$ is composed of digits.

Let 

$$g(i) = g_{i1} g_{i2} \ldots g_{i\alpha}.$$  \hspace{1cm} (5-2)

Then $g$ becomes

$$g = g_{11} g_{12} \ldots g_{1\alpha} g_{21} g_{22} \ldots g_{2\alpha} \ldots g_{\nu1} g_{\nu2} \ldots g_{\nu\alpha} 000 \ldots$$  \hspace{1cm} (5-3)

The sub-generator sequences are derived from $g$ in the following manner.

$$G_1 = g_{11} g_{21} g_{31} \ldots g_{\nu1}$$

$$G_2 = g_{12} g_{22} g_{32} \ldots g_{\nu2}$$

$$\vdots$$

$$G_{\alpha} = g_{1\alpha} g_{2\alpha} g_{3\alpha} \ldots g_{\nu\alpha}$$  \hspace{1cm} (5-4)

The message sequence $m = m_1 m_2 \ldots m_n$ is convolved over the generator sequence $g$ in the standard sequential encoding method (Wozencraft & Reiffen (2), and Lin (3)) to produce an encoded sequence,

$$u = u_{11} u_{12} \ldots u_{1\alpha} u_{21} u_{22} \ldots u_{2\alpha} \ldots u_{n1} u_{n2} \ldots u_{n\alpha}.$$  \hspace{1cm} (5-5)
Using sub-generators, \( m \) is convolved over all of the \( g_i \) to produce sequences:

\[
\begin{align*}
\mathbf{u}_1 &= \mathbf{u}_{11} \mathbf{u}_{21} \mathbf{u}_{31} \cdots \mathbf{u}_{n1} \\
\mathbf{u}_2 &= \mathbf{u}_{12} \mathbf{u}_{22} \mathbf{u}_{32} \cdots \mathbf{u}_{n2} \\
\mathbf{u}_\alpha &= \mathbf{u}_{1\alpha} \mathbf{u}_{2\alpha} \mathbf{u}_{3\alpha} \cdots \mathbf{u}_{n\alpha}.
\end{align*}
\]

(5-6)

Or using the notation of Section IV on interlacing, \( \mathbf{u}, \mathbf{u}_1, \mathbf{u}_2 \) become,

\[
\begin{align*}
\mathbf{u} &= a_1 b_1 c_1 a_2 b_2 c_2 \cdots a_n b_n c_n \\
\mathbf{u}_1 &= a_1 a_2 a_3 \cdots a_n \\
\mathbf{u}_2 &= b_1 b_2 b_3 \cdots b_n
\end{align*}
\]

(5-7)

The sub-generator encoding program is more complicated than the standard procedure, but if interlacing is only over one branch segment, the interlacing programming is simplified. This is because, while using the \( \mathbf{u}_1 \), the number of bits desired in the bit spreading is available by a simple shift. If the standard encoding procedure is used, which should be the case for bit interlacing over several branch segments, a more complicated sorting procedure is needed to select every \( n \)th bit from a sequence and then to process the same sequence again, but with the selected digits shifted over by one from those selected on the previous pass over the sequence.

The program SPREDR which does the actual interlacing, over single branch segments only, is primarily a bookkeeping program to keep track of location in the message and count various word lengths in shifting while doing the interlacing.

**The Noise Source and Channel.**

The model for noise in the channel is Lyne's adaptation (5) of the Binary Symmetric Channel (BSC) such that bursts of errors
are also generated. It is termed the Binary Symmetric Burst Channel (BSBC). The program was written by Lyne.

In this model whether an error occurs, or not, is governed by a BSC. Once an error does occur, a uniform distribution is sampled to find how long a burst will be generated. The maximum burst length is a power of two up to $2^5$. The maximum burst length is a parameter which is set for each test run. A second pseudo-random number generator of the same type used in the message generation is used to generate a 1 with a given probability. The 1 indicates a burst should begin. A third random number generator then generates a binary fraction between 0 and 1. The fraction is truncated down to, at most, the five most significant digits of the fraction. These digits are treated as an integer of at most five digits. For instance had the third number generator yielded the fraction $0.10110_2$, the various burst lengths which could have been generated are shown in Table 1.

<table>
<thead>
<tr>
<th>Maximum Burst Length</th>
<th>$ln\text{M.B.L.}_2$</th>
<th>Truncated Fraction</th>
<th>Burst Length Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>5</td>
<td>10110</td>
<td>22</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>1011</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**TABLE I**

When the whole sequence has been generated by BURN03, it is added word by word modulo-two to the output of SPREDR by the program ADNOIS to form the output of the channel. Modulo-two addition is done using the Exclusive-OR.
The input to SQUEZE, the unlacing program, is an array in the form $t_0$:

$$t_0 = \overline{a_1 a_2 \ldots a_n b_1 b_2 \ldots b_n c_1 c_2 \ldots c_n \ldots} \quad (5-8)$$

First this is put into the same format as the encoded message was at the output of ENCODR.

$$V_1 = v_{11} v_{21} v_{31} \ldots v_{n_1} \ldots$$

$$V_2 = v_{12} v_{22} v_{32} \ldots v_{n_2} \ldots \quad (5-9)$$

$$V_\alpha = v_{1\alpha} v_{2\alpha} v_{3\alpha} \ldots v_{n\alpha} \ldots$$

After the received sequence has this form, SQUEZE calls SPREDR with a bit spread of 1 and $v$ is formed,

$$V = v_{11} v_{12} v_{13} \ldots v_{1\alpha} v_{21} v_{22} v_{23} \ldots v_{2\alpha} \ldots v_{n_1} v_{n_2} \ldots v_{n_\alpha} \ldots \quad (5-10)$$

The decoder program is designed to expect machine words holding an integral number of branch segments and no partial branches. WRDPAC is the program which changes the format of $v$ from fully packed machine words to the format required by the decoder.

The unlaced received sequence is then written on magnetic tape and stored until the generation of all desired received sequences is completed for each run. The tape is then used as the input to the decoder program.
The Main Decoder Program—RECVER.

This author is indebted to W.H. Lyne for this program and its subroutines which were written for Dr. Lyne's Ph.D. Dissertation. Only slight modifications were made to his program. Those modified were primarily input-output subroutines.

The manner of processing data and the main program loop in this program is similar to that in the STRBOS program. This program is capable of decoding codes of up to $\alpha = 12$ and $\gamma = 10$.

An approximate flow chart of the RECVER program is shown in Figure 6. Essentially, the decoder does a branch-by-branch search, tests the resultant tentatively decoded sequence, and proceeds. If a tree search is necessary, it is performed and checked with an MTT. These main functions are in the right-hand column of Figure 6. The left-hand column provides various tables and initiates the required arrays to facilitate the operations in the primary program loop.

The Input Program—REREAD.

This program is one which reads the tape for input data and has several entry points so separate portions of the program may be executed separately. Facility is provided to initialize the run from any suitable starting point in the tape data. Immediately following the first two calls to this subroutine, the memory space holding such things as the decoded sequence and the tentatively accepted path of previous runs is cleared so that it will not interfere with the current run.
POSITION TO DESIRED DATA SET ON INPUT TAPE. READ CODE AND RUN PARAMETERS FROM TAPE. (REREAD 1 & 2)

CONSTRUCT TREE PATH WEIGHT TABLE. (BLWGT)

CONSTRUCT CUMULATIVE SUM OF GENERATOR SEQUENCE TABLE. (BLSGSM)

CONSTRUCT LAST COLUMN OF BRANCH SEGMENTS IN INITIAL TREE. (BLCOLM)

CALCULATE CODE CORRECTION ABILITY AND SET THRESHOLDS FOR MTT. (SETER)

PACK VARIABLE ADDRESSES OF COMMANDS IN LATER SUBROUTINES. (PACKER)

READ MESSAGE AND RECEIVED SEQUENCES FROM TAPE. (REREAD 3)

DECODE FINALLY A MESSAGE DIGIT AND MOVE TENTATIVE RECEIVED, ERROR AND WEIGHT PATHS OVER IN ARRAYS TO ALLOW NEW BRANCHES TO BE SHIFTED INTO ARRAYS. (MOVOVR)

UNPACK A NEW RECEIVED BRANCH SEGMENT FROM RECEIVED SEQUENCE AND PLACE IN THE SPACE VACATED BY MOVOVR. (UNPACK)

BRANCH-BY-BRANCH SEARCH. (BRANCH)

MULTIPLE THRESHOLD TEST. (MULTI)

THRESHOLD VIOLATED??

YES

SEARCH A K-UNIT AND CHECK RESULTS WITH AN MTT. SEARCH)

GIVE UP DECODING.

PROCEED TO NEW BRANCH.

CHECK DECODED SEQUENCE WITH ORIGINAL MESSAGE AND PRINT OUT ERROR MESSAGE AND RESULTING PROBABILITIES. (CHECK & RESULT)

SET DONE

YES

GO TO NEXT DATA SET??

DECODE MORE OF THIS PARAMETER SET.

NO

STOP

RECOVER PROGRAM FLOW CHART

FIGURE 6.
The Initializing Program—BLWGHT, BLSGSM, BLCOLM, SETER, and PACKER.

Let us consider these programs separately. BLWGHT is a program to create an array $2^{12}$ words long containing the weight of the binary numbers from 0 to $2^{12}$. This table is consulted to find the weight of the error sequence between the received sequence and the tentatively decoded sequence. The table is readily accessible since the weight of the error sequence, considered as a binary integer, is also the position number of the weight within the table.

The program BLSGSM generates the lowest path in the tree. Branch-by-branch this path consists of the sum modulo-two of all the generator sequences used up to and including that of the current stage (see Figure 1). This path is stored one branch per machine word in an array for use in the tree search program.

For the decoder to work, it must be capable of generating up to $V$ segments of any possible code tree path. BLCOLM is the program which provides this capability. It generates an array containing the last branch segment of each of the $2^V$ paths in the tree. It is the size of this array which currently limits the maximum $V$ of a code which the decoder can decode. The manner in which the tree is generated insures that this array contains all branches of the tree and that a path through the tree may be traced by hopping up and down this column. The manner in which the path is traced is discussed later in this section with the branch-by-branch search procedure.

SETFR. In order for the MTT to be performed, the error correction capability of the code at each stage cut through the
initial truncated tree must be known. This is calculated from the minimum weights between the upper and lower half trees given in the input data. From this the threshold function $T(.)$ is set for the desired nodes in the $\gamma$ stages.

The program PACKER is strictly a bookkeeping program. In it the addresses for some machine operation commands, which vary with $\alpha$ and $\gamma$, are set. Following PACKER, the received sequence is read from the input tape along with the original message and decoding is begun.

The Main Loop Program—MOVOVR, UNPACK, BRANCH, DCDR2, MULTI, SEARCH, CHECK and RESULT.

MOVOVR and UNPACK decode finally a message digit, shift over all the tentative sequences, and shift in a new segment or new segments of the received sequence. This program has the facility to decode finally several branches at a time and shift in a like number of unprocessed received sequence segments. The number of segments shifted at one time was 1 for the test runs. Shifting more segments than 1 should speed up decoding in lower noise situations where the tree search is infrequently needed. The arrays which require shifting are the received path in the decoder, the tentatively decoded path, the error path between the received and tentatively accepted paths, the weight of the error path, and the tentatively accepted message path. UNPACK shifts new received sequence segments into the vacated words in the received path array in the decoder, from the array read from the input tape.
The Branch-By-Branch Search--BRANCH

BRANCH uses the COLUMN array from the program BLCOLM to find the branch segments stemming from the last tentatively decoded path. The location of the branch segments in the COLUMN array are numbered from 0 to $2^V - 1$. In order to reach the upper branch stemming from branch number $x$, branch number $2x$ is chosen. To reach the lower branch corresponding to a 1 message digit branch, $2x + 1$ is chosen. Should the last number $y$ be in the bottom half of the COLUMN array, there is no path numbered $2y$. BRANCH calls as a subroutine DCODR2 to determine to which path $2y$ corresponds. As an example, assume $y = 12 \sim 011002$ in the $4^{th}$ stage of Figure 3. It is the $4^{th}$ column which would be stored for decoding this $= 4$ code. Row 12 doubled is row 24, or 110002. DCODR2 merely truncates away the lead 1 in the binary numeral, leaving the corresponding row number to be row 8 in the $4^{th}$ column. Observing Figure 2 will verify the validity of the truncation method.

The row in the COLUMN array, obtained as in the above example, is tried as the tentative branch segment. It is added to the received path modulo-two to generate the error sequence. The weight of this segment is found, using the BLIGHT array, and compared with the correction capability of the code in a 1-unit. If the correction capability is greater than or equal to the determined weight, the upper branch is chosen. Otherwise, 1 is added to the row number and the corresponding tentative branch is selected. Proper error and weight paths are generated. Just before exiting from the program a tentative message digit is placed in an array for future use if the path selected is the correct one.
The Multiple Threshold Test.

The program which conducts the MTT is MULTI. In it the weights generated in BRANCH are made cumulative from the input end of the decoder to each of the thresholds. For testing purposes the thresholds were taken only after the branches in which the error correcting capability increased from what it had been previously. Thus for the $\alpha = 3$, $\nu = 10$ code, thresholds were taken after the $3^{rd}$, $5^{th}$, $7^{th}$, and $10^{th}$ branches from the input end of the decoder. All thresholds are checked against the respective weights of the error path. If there are no violations, MULTI is terminated and control transferred to MOVOVR to be decoded. If the current block is completely decoded, program control proceeds to the subroutines CHECK and RESULT. However, if a threshold has been violated, the tree search program SEARCH is called. The tree search is begun from the highest threshold violated, that is, the threshold nearest to the output end of the decoder. The topmost path in the $k_0$-unit to be searched is generated together with its mirror image in the lower half of the $k_0$-unit. The closer of these two paths to the received path is chosen. Ties go to the first path tested. The mirror image path is used to generate a third path, its own mirror image path altered by one row number towards the axis of reflection. The newly generated path is then compared with the current "best" path and selection made in the same way as it was initially. The third path generates a fourth path, this one in the lower half tree, closer to reflection axis by the path than the second path. The process is continued until the $k_0$-unit has been completely searched in order from its outermost paths to the paths just above
and below the middle of the tree. It is in generating the mirror image paths that the array of sums of the generator segments created in BLSGSM is used.

The row number of the best path found by the search of the $k_0$-unit is saved during the search and it is used to generate again that path's elements, error path, and error weight path. The MTT is performed in SEARCH on this best path. If thresholds are again violated, the next larger $k$-unit, a $k_0 + 1$-unit, is searched. The backup and search process is continued until the thresholds are met. If the primary threshold is violated, the decoder is allowed to process a new branch segment with the BRANCH program. In Lyne's program this violation is allowed to occur $2V$ times before the decoder gives up. In the tests run with bit interlacing, the decoder gave up too quickly in cases where the error correcting capability was being heavily taxed, but in which predictions showed that difficult error patterns should only be occasional. For this reason SEARCH is adapted in most data runs to allow $3V$ consecutive violations. This modification did not cause the decoder to get back onto the correct path any more quickly than the $2V$ backup case. But it did allow the decoder to get back onto the correct path in instances where the $2V$ backup procedure gave up decoding efforts. If the SEARCH program does give up decoding, control is transferred directly to CHECK and RESULT. At that time decoding is begun on the next received sequence on the input tape which has a different set of parameters.
The Bookkeeping Program To Determine Decoder Effectiveness.

The CHECK program adds modulo-two the decoded sequence and the original message sequence, ignoring the first digits of the decoded sequence. It ignores these because they are the product of the start procedure in decoding and are not message digits. The addition forms an error sequence printed out by RESULT, unless the array is non-zero. RESULT also prints out a summary of the code data used for the run; error generation statistics; actual counts of generated errors and errors in decoding; error probabilities in the received sequence; and error probabilities for the errors actually made. RESULT returns control to the main program to continue decoding with a new set of data or to end the run if there is no more data.
SECTION VI

Results and Discussion

The test runs made with the programs previously described were not intended to be exhaustive of all possible combinations of noise conditions and bit spread distances. The results bear out expectations both in the expected suppression due to interlacing and in the short-comings of this particular single branch interlacing scheme for some codes.

Two codes were used in the simulation tests. One of these was an $\alpha = 3$ code with $V = 10$ whose generator sequence $g$ was

$$g = 1110010010100101000001,$$  \hspace{1cm} (5-1)

which yielded a minimum weight of 11 between half trees at the tenth stage. The other code was an $\alpha = 5, \nu = 10$ code with generator sequence $g$,

$$g = 011111000011000101001001001110001000 \quad (5-2)$$

$$0010100011000110000101,$$

whose minimum weight is 21 at stage ten.

For each combination of input parameters, a run of 20,160 message digits was made. This corresponds to transmitting 60,480 channel digits for the $\alpha = 3$ code or 100,800 channel digits for the $\alpha = 5$ code. Longer runs would be preferable from a statistical point of view. However, the cost of computer time made runs of greater length comparatively high. For example, for the $\alpha = 3$ code, the difference between the probability of an error in the transmitted channel sequence, including bursts (POEW/B), and the actual frequency of errors generated in the channel,
including bursts (FOEW/B), was about 2% after 60,480 channel digits. For this $\alpha = 3$ code the difference after 120,960 channel digits was about 0.9%; for 151,200 channel digits it was about 0.75%. Considering the reduction of the difference between the actual noise generation of the pseudo-random number generator and its predicted values, more than doubling the computer time did not seem feasible.

There are three quantities which are the primary basis for determining the effectiveness of any given run with a particular code and a particular set of error parameters for the channel. We have mentioned the FOEW/B. This is the ratio of the total number of error digits generated in the channel to the total number of channel digits processed in each run. FOEW/B is cumulative over all the loops through the program for each run. The ratio FOEW/B is compared with the frequency of decoding errors (FDE), in order to determine the effects of the error-correcting capability of the code and/or bit spreading. FDE is the ratio of the total number of decoded message digits in error to the total number of message digits processed in each run. FDE is also cumulative for each run. The third quantity is the normalized average number of computations (NAVNOC) which is a measure of the number of times the decoder is forced to use the tree search procedure. If the value of NAVNOC is 1.0 units, the branch-by-branch search has chosen the correct tentative branch segment each time. NAVNOC is an average, so it is possible that if only a small number of searches of small k-units were required, there would be no observable increase from 1.0 in NAVNOC. In some situations there would be no difference in FDE.
with and without bit spread. But in the case of spread it may be possible for the decoder to operate on a branch-by-branch basis, with a resulting reduction of NAVNOC and hence of computer processing time.

From Table 2, it may be seen that no amount of bit spreading caused an appreciable decrease in the FDE for the \( \alpha = 3 \) code in comparison with the no spread case, BS = 1. Quite frequently for these same \( \alpha = 3 \) codes NAVNOC is increased by several units while there is a minor reduction in FDE of approximately 10% to 25%. The increase of processing time may be too great for bit spreading to be feasible in these codes.

For the MBL = 2 case, the two decoding errors made with each of the bit spreads were caused by the same error configuration in the channel. This configuration occurred too early in the received sequence to be modified by any of the bit spreads.

For \( \alpha = 3 \), MBL = 4, the MBL was just less than the error-correcting capability of the code, so the code with no bit interleaving yielded an FDE lower than the FOEW/B by an appreciable amount. For this parameter set, FDE was an order of magnitude less than FOEW/B. But since interleaving was over a single branch, bit spreading could not be expected to spread channel errors over wide enough portions of the received sequence for the interleaving to further reduce FDE. A BS = 360 did reduce FDE by a factor of one-half, but at the expense of increasing NAVNOC by nearly 0.55 units.

For the \( \alpha = 3 \), MBL = 8 case, the error-correcting capability of the code was exceeded by the MBL, so with no spreading, FDE was slightly more than one-half of FOEW/B. No amount of bit
spreading improved much on the reduction of error, and bit spreading increased NAVNCC by as much as 8.4 units.

From these cases we can see that bit spreading over a single branch, where one error per branch in the received sequence is not within the error-correcting capability, is of quite questionable value. However, in looking at the results with an \( \alpha = 5 \) code, where one error per branch is acceptable, the value of bit spreading may be seen.

In the case MBL = 8 for the \( \alpha = 5 \) code, we see that interlacing is sensitive to some error patterns. The code itself provided nearly an order of magnitude decrease in FDE from FOEW/B. A BS = 40 reduced FDE to 1/4 the value it had for the case without interlacing. NAVNOC was reduced by more than 2.0 units. But for BS = 50 and BS = 280, the errors were gathered together in such a manner that the decoder gave up attempting to decode the message after 1800 to 3600 message digits.

It is in the case in which the MBL is greater than the error-correcting capability of the code that interlacing is of most assistance. With no interlacing, BS = 1, there was almost no reduction of FDE from FOEW/B. For this MBL = 32 and for the few different values of BS tested, the greater the value of BS, the greater the reduction in FDE and NAVNOC from their values for the BS = 1 test run. With BS = 840, FDE was reduced to less than 1/10 of the no spread value and NAVNCC was decreased by more than 9.0 units.

To summarize quickly, bit interlacing can be used quite effectively to increase the error correcting capability of
convolutional codes, when burst errors are highly probable in the channel, and if the interleaving spreads adjacent errors in the channel out over enough branch segments so the error-correcting capability of the code is not exceeded.

**Suggestions For Future Investigation.**

There are two areas in which further investigation would be useful to make the simulation more meaningful. The first suggestion would be to program the interleaving so that it may be done over a variable number of branch segments. The performance of the $\alpha = 5$ code with bit spreading suggests that interleaving over several branches in an $\alpha = 3$ code would allow similar performance with the higher rate of transmission of $R = 1/3$.

Of special interest would be the selection of optimal bit spread distance for a given maximum burst length or similar noise parameter. The codes should be tested under a greater variety of error frequency, especially noise conditions taxing the correction capability more severely.

Future studies will be limited by the adequacy of the channel noise model, just as is this study. Thought might be given to possible changes in the BSBC. One change of possible merit would be to sample one-half of a normal distribution for the burst length, greater weight being given to the shorter length errors. The random number generator may be adapted easily to yield normally distributed numbers but at the sacrifice of requiring greater generation time. Such a change might also encompass modification of the process by which an error burst is begun, such that bursts of bursts are more probable.
Some statistics reflecting the clustering of bursts are given by Fein, et al, (1), from their study of high-frequency radio channels while using interlacing on block codes.
THE KEY TO ABBREVIATIONS IN TABLE 2.

POEB    -- The Probability Of an Error Burst Occurring.
MBL     -- The Maximum Burst Length.
BS      -- The Bit Spread Distance.
POEW/B  -- The Probability Of an Error in the transmitted channel sequence, including all Bursts.
POEW/B  -- The actual Frequency Of Errors generated in the channel, including Bursts.
FDE     -- The Frequency of Decoding Errors in the decoded sequence.
NAVNOC  -- The Normalized Average Number Of Computations.
NO.DE   -- The Number of Decoding Errors in the decoded sequence.
NO.DEB  -- The Number of Decoding Error Bursts in the decoded sequence.
NO.CHDT -- The Number of Channel Digits Transmitted.
MD/R    -- The number of Message Digits per Run.
<table>
<thead>
<tr>
<th>POEB</th>
<th>MBL</th>
<th>BS</th>
<th>POE W/B</th>
<th>FDE</th>
<th>NAVNOC</th>
<th>NO. DE</th>
<th>NO. DEB</th>
<th>CHDT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.010</td>
<td>2</td>
<td>1</td>
<td>0.01500</td>
<td>0.01526</td>
<td>0.00010</td>
<td>1.55795</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>40</td>
<td></td>
<td>0.00010</td>
<td>1.31217</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>70</td>
<td></td>
<td>0.00010</td>
<td>1.39044</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>280</td>
<td></td>
<td>0.00010</td>
<td>1.21901</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0.010</td>
<td>4</td>
<td>1</td>
<td>0.02500</td>
<td>0.02550</td>
<td>0.00342</td>
<td>6.07416</td>
<td>69</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td></td>
<td>0.00255</td>
<td>6.66091</td>
<td>51</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20</td>
<td></td>
<td>0.00248</td>
<td>7.38547</td>
<td>50</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>40</td>
<td></td>
<td>0.00208</td>
<td>5.59483</td>
<td>42</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>120</td>
<td></td>
<td>0.00293</td>
<td>6.14055</td>
<td>59</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>280</td>
<td></td>
<td>0.00184</td>
<td>6.09332</td>
<td>37</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>360</td>
<td></td>
<td>0.00169</td>
<td>6.61920</td>
<td>34</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>840</td>
<td></td>
<td>0.00337</td>
<td>6.50654</td>
<td>68</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1680</td>
<td></td>
<td>0.00283</td>
<td>6.55729</td>
<td>57</td>
<td>12</td>
</tr>
<tr>
<td>0.010</td>
<td>4</td>
<td>1</td>
<td>0.04500</td>
<td>0.04463</td>
<td>0.02391</td>
<td>32.45436</td>
<td>482</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>40</td>
<td></td>
<td>0.02103</td>
<td>35.02371</td>
<td>424</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>70</td>
<td></td>
<td>0.02212</td>
<td>38.68170</td>
<td>446</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>280</td>
<td></td>
<td>0.02217</td>
<td>40.81748</td>
<td>447</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>840</td>
<td></td>
<td>0.02257</td>
<td>39.78748</td>
<td>455</td>
<td>72</td>
</tr>
<tr>
<td>0.010</td>
<td>8</td>
<td>1</td>
<td>0.04500</td>
<td>0.04529</td>
<td>0.00546</td>
<td>9.34462</td>
<td>110</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>40</td>
<td></td>
<td>0.00268</td>
<td>7.33789</td>
<td>54</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20</td>
<td></td>
<td>0.00238</td>
<td>7.77801</td>
<td>48</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>40</td>
<td></td>
<td>0.00139</td>
<td>7.26867</td>
<td>28</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>50</td>
<td>gave up</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>280</td>
<td>gave up</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1400</td>
<td></td>
<td>0.00268</td>
<td>8.18249</td>
<td>54</td>
<td>9</td>
</tr>
<tr>
<td>0.001</td>
<td>32</td>
<td>1</td>
<td>0.01650</td>
<td>0.01618</td>
<td>0.01478</td>
<td>11.18562</td>
<td>298</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>40</td>
<td></td>
<td>0.00193</td>
<td>2.38832</td>
<td>39</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>280</td>
<td></td>
<td>0.00188</td>
<td>2.91713</td>
<td>38</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>840</td>
<td></td>
<td>0.00129</td>
<td>1.82020</td>
<td>26</td>
<td>2</td>
</tr>
</tbody>
</table>

**RUN STATISTICS**

**TABLE 2.**
BIBLIOGRAPHY


