View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by NASA Technical Reports Server

NASA-CR-191899
GODDARD
GRANT
IN-61-CR
141147
P.34

# An Investigation of
# Error Characteristics and Coding Performance

NASA GRANT NAG5-2006
July 1, 1992 - June 30, 1993

Semi-Annual Report 1
July 1, 1992 - Dec 30, 1992

(NASA-CR-191899) AN INVESTIGATION OF ERROR CHARACTERISTICS AND CODING PERFORMANCE Semiannual Report No. 1, 1 Jul. - 30 Dec. 1992 (Mississippi State Univ.) 34 p
N93-18373
Unclas
G3/61 0141147

Submitted to:

Mr. Warner Miller
Code 728.4
Instrument Electronic Systems Branch
Engineering Directorate
NASA/Goddard Space Flight Center
Greenbelt, MD 20771
301-286-8183

Submitted by:

William J. Ebel, Ph.D.
Frank M. Ingels, Ph.D.
Mississippi State University
Drawer EE
Mississippi State, MS 39762
601-325-3912

December 1992

# An Investigation of
# Error Characteristics and Coding Performance

NASA GRANT NAG5-2006
July 1, 1992 - June 30, 1993

Semi-Annual Report 1
July 1, 1992 - Dec 30, 1992

Submitted to:

Mr. Warner Miller
Code 728.4
Instrument Electronic Systems Branch
Engineering Directorate
NASA/Goddard Space Flight Center
Greenbelt, MD 20771
301-286-8183

Submitted by:

William J. Ebel, Ph.D.
Frank M. Ingels, Ph.D.
Mississippi State University
Drawer EE
Mississippi State, MS 39762
601-325-3912

December 1992

# Table of Contents


I. INTRODUCTION ........ 1
II. SOURCE CODE GENERAL DESCRIPTION ........ 2
III. PROGRAM DESCRIPTIONS ........ 7
A. Forward Error Correcting Codes ........ 7
1. BlkDecod (Block Decoder) ........ 7
2. Viterbi ........ 8
B. Channel Error Sequences ........ 9
1. BinErrs (Binomial Error Sequence generation) ........ 9
2. BrstErrS (Burst Error Sequence generation) ........ 10
3. BstyErrs (Bursty Errors Sequence generation) ........ 11
C. Interleavers ........ 11
1. BlockInt (Block Deinterleaver) ........ 12
2. BlkArr (Alternate Block Deinterleaver) ........ 13
3. DPCI (Periodic Convolutional Deinterleaver) ........ 13
4. DPCIAlt (Alternate Periodic Convolutional Deinterleaver) ........ 14
D. Error Sequence Analysis ........ 15
1. CVMseq (Cramer Von-Mises sequence distribution test) ........ 15
2. CVMblk (Cramer Von-Mises distribution test on error sequence blocks) ........ 15
3. DeltaEst (Bursty-error parameter estimation via the Δ method) ........ 16
4. IntvHst (Error Interval Histogram) ........ 17
5. GAPEst (fixed GAP burst error distribution Estimation) ........ 17
E. Utilities ........ 18
1. Make Utility for Lahey Fortran v5.0 ........ 18
2. CompSeq (Compare Sequence) ........ 20
3. SetErrS (Set Error Pattern) ........ 20
4. DisplSeq (Display Default Error Sequence) ........ 21
5. DisplFil (Display Error Sequence from user File) ........ 21
IV. NASA GSFC/MSU INTERRELATED CAPABILITIES ........ 22
A. EOS Real Error Sequence Data conversion program ........ 22
B. Error Sequence Archiver using Run Length Encoding ........ 22
C. Error Sequence Unarchiver ........ 23
V. PREVIEW OF EXPECTED RESULTS ........ 24
A. Research Focus 1 ........ 24
B. Research Focus 2 ........ 25
C. Choosing System Parameters ........ 27
BIBLIOGRAPHY ........ 31

# I. INTRODUCTION

This report describes research performed to date on NASA Grant NAG5-2006 for the period July 1, 1992 through December 1, 1992. This work involves studying the performance of forward error correcting coding schemes on errors anticipated for the Earth Observation System (EOS) Ku-band downlink.

The EOS transmits picture frame data to the ground via the Telemetry Data Relay Satellite System (TDRSS) to a ground-based receiver at White Sands. Due to unintentional RF interference from other systems operating in the Ku band, the noise at the receiver is non-Gaussian which may result in non-random errors output by the demodulator. That is, the downlink channel cannot be modeled by a simple memoryless Gaussian-noise channel. From previous experience, it is believed that those errors are bursty.

The research has proceeded by developing a computer based simulation, called Communication Link Error ANalysis (CLEAN), to model the downlink errors, forward error correcting schemes, and interleavers used with TDRSS. To date, the bulk of CLEAN, described in Sections 3, 4, and 5, has been written, documented, debugged, and verified. The procedures for utilizing CLEAN to investigate code performance have been established and will be discussed in Section 5.

## II. SOURCE CODE GENERAL DESCRIPTION

Each system component (decoder, deinterleaver, etc.) has been implemented in CLEAN as separate executable computer programs which interface with each other through data files including an error sequence data file. This allows them to be executed sequentially via a batch file.

All computer programs read parameters from a separate ASCII parameter file with a fixed default name. The default name for the parameter file is the same as the executable but has the extension 'prm'. Also, there is a global parameter file, 'ID.prm', which contains a simulation identifier (ID). Each program generates an output file with an extension identical to this ID. This output file contains all the calculated statistics and estimated parameters from the program. This allows all the files generated by a specific run to be quickly identified and distinguished from data files generated by other runs.

To conduct the studies, a batch file is created which contains a series of executable programs. The type and order of the executables in the batch file implements a particular system configuration. For example, if the user chooses to use a Reed-Solomon (RS) decoder to decode a sequence of random errors, then the batch file contains two executables; the first generates a random error sequence and the second uses an RS decoder to correct them. In general, the batch file contains one of the channel error sequence generation programs which will generate an error sequence stored in file name 'error.seq'. Each program which is executed makes use of and/or modifies that error sequence and generates statistics and other outputs for the error pattern.

The programs have been written with parameter bounds in mind. For example, the programs are designed so that the lowest channel average error probability to be investigated, coded or uncoded, is roughly $10^{-6}$. Along with this, it is assumed that 20 errors are the minimum number required to characterize the statistics of the channel, however, in general many more errors will be generated per sequence. Thus as an upper bound, generating an error sequence, coded or uncoded, with an error probability of $10^{-6}$ requires a minimum of $20/10^{-6} = 2x10^{7}$ error sequence values. The error sequence file is stored in a "packed" format so that 15 error sequence values are stored per two bytes of memory. Therefore, the largest error sequence file is $2x10^{7}/(2/15) = 2.67$Mbytes. This is sufficiently small so that allowable disc space on most computers can accommodate several files at once. In general, error files are ***not*** stored but are generated on the fly. Results can be reproduced by regenerating an error sequence given the proper random number generator and the seed. If it turns out that regenerating the error sequence takes too long, then a set of error sequences can be generated and stored on disc or magnetic tape to be retrieved when required.

All programs have been documented upon completion with a documentation test run. All the generated documentation is stored in a common binder for later reference.

Each program conforms to a documentation standard which includes a program/subroutine/function header as well as line comments within the code. On average, there should be a comment line per 6 lines of code to indicate the purpose of the next few lines of code. The routine header takes the following form:

```
c*******************************************************************
c*
c* - Program/Subroutine/Function name:  name (Acronym meaning)
c*
c* - Purpose:  This program/subroutine/function ...
c*
c*
c* - Revision History:
c*        Date            Who                          Reason
c*   ----------------------------------------------------------------
c*      May 25, 1992      WE                          Original
c*
c*
c* - Variable/File List:
c*           Name      Type              Description
c*      -------------------------------------------------------------
c*     Inputs:
c*
c*     Outputs:
c*
c*     Internals:
c*
c*
c* - Subroutines called:
c* - Subroutines called by:
c* - Functions called:
c* - Functions called by:
c*
c*********************************************************************
```

As an example, a program written to create a bursty-error sequence may have a header which appears as follows:

```
c*******************************************************************
c*
c* - Program name:  BstyErrS (Bursty-Error Sequence)
c*
c* - Purpose:  This program generates an binary error sequence with
c*      bursty errors.  The error sequence denotes a correct binary channel
c*      transmission with a 0 and denotes an error with a 1.  The error
c*      sequence is partitioned into two main, noncontiguous parts, the burst
c*      error part and the thermal error part.  The method used to generate
c*      each part of the error sequence depends upon the density of errors to
c*      be generated.  For each error sequence part, if the required density of
c*      errors is greater than .01, then the program uses a conditional test on
c*      a uniform random number in the range [0,1]. If the density of errors is
c*      less than .01, then the program will use a sample from the exponential
c*      distribution to generate the next error occurrence time.
c*        This program inputs parameters from an ASCII data file with default
c*      name 'BstyErrs.prm' and outputs the error sequence to a data file
c*      with default name 'error.seq'.  In addition, various statistics are
c*      output to an ASCII data file with default name 'BstyErrs.ID', where
c*      ID is a three letter identifier for the current run which is input from
c*      file 'ID.prm'.
c*        The program is run by editing the parameter file 'BstyErrs.prm' and
c*      selecting the appropriate parameters and by choosing a program ID by
c*      editing file 'ID.prm'.  Executing the program generates the 'error.seq'
c*      file which contains an error sequence (in packed format) with
c*      binomially distributed errors.  It does not matter whether the output
c*      file 'error.seq' exists or not. If it exists, it is overwritten without
c*      a prompt to the user.
c*        Even though Poisson distributed bursts may overlap in theory, this
c*      progam does not allow error bursts to overlap.  The user must take care
c*      to specify input parameters so that the probability of overlapping
c*      burst is negligible.  It is also assumed that Peg<Peb.
```

```
c*
c*
c* - Revision History:
c*       Date           Who                       Reason
c*    --------------------------------------------------------------
c*     Aug 20, 1992     WE                       Original
c*    Sept 14, 1992     WE       Modified to use Makefile to link source
c*                               and updated the documentation
c*     Oct  2, 1992     WE       Output Number of Errors to the error.seq
c*                               file header
c*     Nov  4, 1992     WE       Updated NextBurst function argument list
c*                               to include the previous burst length
c*     Nov 13, 1992     WE       Added write to output Log10(Density)
c*     Nov 16, 1992     WE       Changed all real variables to double precision
c*
c*
c* - Variable/File List:
c*    Inputs:  None (See subroutine ReadParams)
c*
c*    Outputs:
c*         Name       Type              Description
c*      ------------------------------------------------------------
c*       error.seq     file         Error sequence output file
c*                                  (in packed format)
c*         Nerrs     integer*4      Total Number of errors generated
c*      ErrDensity    real*8        Total Error density for generated seq
c*      NBurstyErrs integer*4       Number of errors in the bursts
c*      GenBurstDen   real*8        Error density within the error bursts
c*      GenThermDen   real*8        Error density outside the error bursts
c*        NBursts    integer*4      Total number of bursts generated
c*     GenMeanIntv    real*8        Average burst occurrence
c*    TotalBLength  integer*4       Total sum of burst lengths
c*    GenBDuration    real*8        Average burst length (seq sym)
c*
c*    Internals:
c*          ID      character*3     Identifier for statistics output file
c*           N       integer*4      Error sequence length
c*          Tbs       real*8        Binary channel symbol frequency (freq.)
c*          Peg       real*8        Thermal error density
c*        PegSeed     real*8        Peg random number generator seed
c*          Peb       real*8        Burst error density
c*        PebSeed     real*8        Peb random number generator seed
c*        IntvFlag integer*4        = 1, Periodic error occurrence times
c*                                  = 2, Gaussian error occurrence times
c*                                  = 3, Poisson error occurrence times
c*        IntvMean integer*4        Burst occurrence rate (interval mean)
c*        IntvSeed   real*8         Interval random number generator seed
c*        IntvVar   integer*4       Burst occurrence rate variance
c*                                  (interval statistic variance)
c*       LngthFlag integer*4        = 1, Fixed length error bursts
c*                                  = 2, Gaussian dist. error burst lengths
c*                                  = 3, Exponential error burst lengths
c*       LngthMean integer*4        Burst length distribution mean
c*       LngthSeed   real*8         Length random number generator seed
c*       LngthVar  integer*4        Burst length distribution variance
c*          i,j      integer*4      Do loop indices
c*         RecNum    integer*4      Record number index (error.seq file)
c*        NseqSym    integer*4      Number of DBESS
c*       Error(15)   integer*4      Contains 15 error sequence values
c*         zero      integer*4      Identically the number 0
c*   BurstIntvCount integer*4       Interval Count to next error burst
c*      PrevLength   integer*4      Previous Burst Length
c*  ErrorBurstCount integer*4       Length of next error burst (seq sym)
c*    PegIntvCount   integer*4      Interval Count to next Therm error
c*    PebIntvCount   integer*4      Interval Count to next burst error
c*        DBESS      integer*4      15 consecutive error sequence values
c*                                  stored in a 2 byte integer. Stands
c*                                  for Double Byte Error Sequence Symbol
c*         URV        real*8        Uniform random variable in [0,1]
c*      NSplit(2)    integer*2      A dummy array used to access each
c*                                  double byte of the integer*4
c*                                  number N.
c*     NESplit(2)    integer*2      A dummy array used to access each
c*                                  double byte of the integer*4
c*                                  number Nerrs.
c*
c*
c* - Subroutines called: ReadParams, IterBinErrGen
c* - Functions called: PackErrors, UniformRV, NextBurst, NextLength
c*
c*******************************************************************
```

Figure 1 shows an overall block diagram depicting the CLEAN simulation capability. The CLEAN simulation requires the following assumptions:

1) The transmitted data is all zero
2) Synchronization has been established (i.e. only steady state error statistics are considered)
3) Demodulator performs hard decisions

At each of the points labeled A, B, C, D, and E shown in Figure 1, it is possible to perform statistical analysis including (see Section III.D below):

1) Perform the Cramer Von Mises distribution test to determine if the errors are random.
2) Perform the Cramer Von Mises distribution test on blocks of the error sequence.
3) Estimate burst-error parameters
    a) Average burst-error length
    b) Variance of the burst-error length
    c) List of the burst-error lengths
    d) Average random interval length
    e) Variance of the random interval length
    d) List of the random interval lengths
4) The error interval histogram (for random errors this should be an exponential distribution)
5) Determination of the burst-error distribution 'ala' CLASS

For each program, the calculated statistics are output to the log file as described above.

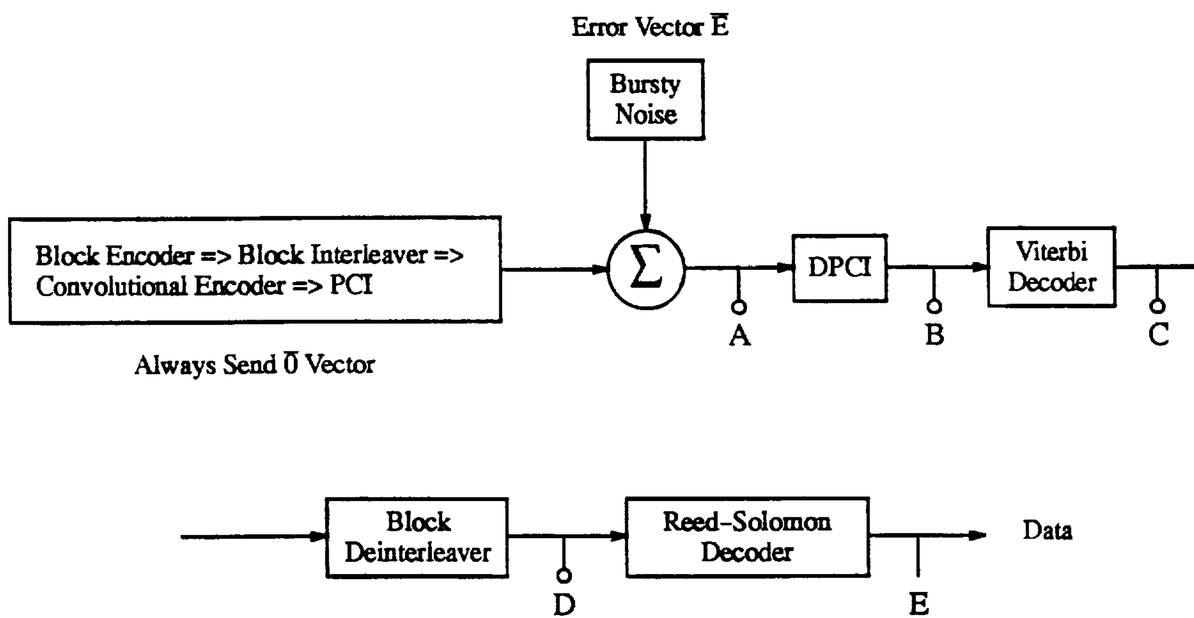


Figure 1. Overall block diagram depicting the CLEAN simulation capability.

# III. PROGRAM DESCRIPTIONS

In this section, the programs which deal with the TDRSS system simulation are briefly described.

## A. Forward Error Correcting Codes

The contract requires that Reed-Solomon codes and convolutional codes be considered. Reed-Solomon codes are a class of block codes. To this end, a program is described which implements the effect of an $(n,k,m,t)$ block incomplete, errors-only decoder and a separate program to implement a Viterbi decoder which is used to decode convolutional codes.

### 1. BlkDecod (Block Decoder)

This program performs the effect of an incomplete, errors (erasure) only decoder. The program operates by simply partitioning the error sequence into blocks equivalent to a received codeword. Error statistics are calculated from each block including the number of bit errors and the number of code symbol errors. If the incomplete decoder detects more errors than the error correcting capability of the code, then the errors are not corrected, otherwise they are.

This program inputs parameters from an ASCII data file with default name 'BlkDecod.prm' and inputs the error sequence from the file with default name 'error.seq'. The decoded error sequence is output to the 'error.seq' file and various statistics are output to an ASCII data file with default name 'BlkDecod.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

The program is run by editing the parameter file 'BlkDecod.prm' and selecting the appropriate parameters and by choosing a program ID by editing file 'ID.prm'. Executing the program generates the 'error.seq' file which contains an error sequence (in packed format) with decoded errors. *The 'error.seq' file must exist prior to the execution of this program.*

There is one important assumption associated with the output of this program. It is assumed that the undetected word error probability is negligible. This is important because this program does not implement an actual decoding algorithm, rather the decoded error sequence is constructed by simply counting errors. Under certain circumstances, it is possible for the errors to occur in such a way so that the received codeword is mapped to within a sphere of $t$ (error correcting capability of the code) about the wrong codeword. A decoding algorithm *cannot* detect (all by itself) that error pattern because it thinks that only a few errors occurred which are

then corrected to the wrong codeword. The probability that this event occurs is called the undetected word error probability. The algorithm implemented here cannot tell whether an error pattern is undetectable by a true decoding algorithm. Therefore, this probability is assumed to be negligible which is, in general, a valid assumption.

## 2. Viterbi

This program performs hard decision Viterbi decoding assuming the all zero sequence is transmitted. The Viterbi decoding algorithm assumes that the trellis begins at the all zero state for the first received code symbol. The end of the decoding process does not terminate with flush bits. Instead, steady state Viterbi decoding is performed up to the end of the error sequence.

This program inputs parameters from an ASCII data file with default name 'Viterbi.prm' and outputs the decoded error sequence to data file with default name 'error.seq'. In addition, various statistics are output to an ASCII data file with default name 'Viterbi.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

The program is run by editing the parameter file 'Viterbi.prm' and selecting the appropriate parameters and by choosing a program ID by editing file 'ID.prm'. Executing the program generates the 'error.seq' file which contains an error sequence (in packed format) with the decoded error sequence. *The 'error.seq' file must exist prior to the execution of this program.* There are several assumptions associated with the implementation and output of this program.

1) It is assumed that the all zero sequence is transmitted,

2) The path with the minimum Hamming distance at the $i^{th}$ Trellis stage is used to find the decoded bit for the output,

3) It is assumed that the convolutional encoder is either rate 1/2 or rate 1/3. It is straight forward to extrapolate this program to accommodate a rate $1/n$ encoder, however this has not been done to date. It should also be possible to modify this program to accommodate a rate $m/n$ encoder using the concept of a punctured convolutional code, again however, this has not been done to date.

The Viterbi algorithm, as implemented here, updates the Trellis by iterating through each of the states at the next stage. The Hamming distance for each path entering a given state are computed and the survivor is kept while the other sequence is discarded. In case of a tie, a coin is flipped (via a Uniform RV in [0,1]) to determine the survivor. The survivor is identified by updating the MLStateTrace array. This array contains the state of the previous Trellis stage which connects to the given state being processed. For example, suppose that we are now processing the next stage in the Trellis, we first consider state 1 at the next stage. After

investigating the Hamming distances for the two possible paths entering state 1, we find that the survivor path came from state 3 of the previous Trellis stage. Therefore, MLStateTrace($i$,1) = 3 where $i$ is the stage index.

To prevent overwriting the Metric array, two Metric arrays are alternately processed for each Trellis stage. This is why the algorithm performs two Trellis stage updates for each main loop. In the first Trellis stage update, the metrics are found in array MetricA and the new metrics are stored in MetricB. In the second Trellis stage update, the metrics are found in array MetricB and the new metrics are stored in MetricA.

The Trellis is defined via three arrays; PathCodeSym, PathLink, and PathBit. Since this program only accommodates rate 1/2 or 1/3 encoders, only two paths enter each state at a given trellis stage. therefore, if there are N trellis states, then there are only 2*$N$ possible paths between two trellis stages. These are sequentially numbered from 1 to 2*$N$ where path number 1 and 2 enter state 1, path 3 and 4 enter state 2, etc. Array PathLink($i$) gives the state number from which path $i$ originates. Also, PathCodeSym($i$) gives the code symbol associated with path $i$, and PathBit($i$) gives the bit associated with path $i$. Taken together, these three arrays completely define the steady state trellis.

### B. Channel Error Sequences

The contract requires that several types of channel errors be considered. A program is described which generates Binomial (random) errors which would occur if the channel noise was additive white Gaussian noise (AWGN). Two other programs are described which generate burst errors and bursty errors. These allow the error bursts to have a variety of length statistics and occurrence statistics in addition to a variety of error density statistics.

#### 1. BinErrs (Binomial Error Sequence generation)

This program generates an binary error sequence with binomially distributed errors. The error sequence denotes a correct binary channel transmission with a 0 and denotes an error with a 1. The method used to generate the error sequence depends upon the density of errors to be generated. If the required density of errors is greater than 0.01, then the program uses a conditional test on a uniform random number in the range [0,1]. If the density of errors is less than 0.01, then the program uses a sample from the exponential distribution to generate the next error occurrence time.

This program inputs parameters from an ASCII data file with default name 'BinErrs.prm' and outputs the error sequence to data file with default name 'error.seq'. In addition, various statistics are output to an ASCII data file with default name 'BinErrs.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

The program is run by editing the parameter file 'BinErrs.prm' and selecting the appropriate parameters and by choosing a program ID by editing file 'ID.prm'. Executing the program generates the 'error.seq' file which contains an error sequence (in packed format) with binomially distributed errors. It does not matter whether the output file 'error.seq' exists or not. *If it exists, it is overwritten without a prompt to the user.*

There are no assumptions associated with the implementation or output of this program.

## 2. BrstErrS (Burst Error Sequence generation)

This program generates a binary error sequence with burst errors. The error sequence denotes a correct binary channel transmission with a 0 and denotes an error with a 1. The error sequence is partitioned into two main, noncontiguous parts, the burst error part and the error free part. The method used to generate the burst error part of the error sequence depends upon the density of errors to be generated. If the required density of errors is greater than 0.01, then the program uses a conditional test on a uniform random number in the range [0,1]. If the density of errors is less than 0.01, then the program uses a sample from the exponential distribution to generate the next error occurrence time.

This program inputs parameters from an ASCII data file with default name 'BurstErrs.prm' and outputs the error sequence to a data file with default name 'error.seq'. In addition, various statistics are output to an ASCII data file with default name 'BurstErrs.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

The program is run by editing the parameter file 'BurstErrs.prm' and selecting the appropriate parameters and by choosing a program ID by editing file 'ID.prm'. Executing the program generates the 'error.seq' file which contains an error sequence (in packed format) with binomially distributed errors. It does not matter whether the output file 'error.seq' exists or not. *If it exists, it is overwritten without a prompt to the user.*

Even though Poisson distributed bursts may overlap in theory, this program does not allow error bursts to overlap. The user must take care to specify input parameters so that the probability of overlapping bursts is negligible.

### 3. BstyErrs (Bursty Errors Sequence generation)

This program generates an binary error sequence with bursty errors; that is, a combination of random and burst errors. The error sequence denotes a correct binary channel transmission with a 0 and denotes an error with a 1. The error sequence is partitioned into two main, noncontiguous parts, the burst error part and the random error part. The method used to generate each part of the error sequence depends upon the density of errors to be generated. For each error sequence part, if the required density of errors is greater than 0.01, then the program uses a conditional test on a uniform random number in the range [0,1]. If the density of errors is less than 0.01, then the program uses a sample from the exponential distribution to generate the next error occurrence time.

This program inputs parameters from an ASCII data file with default name 'BurstyErrs.prm' and outputs the error sequence to a data file with default name 'error.seq'. In addition, various statistics are output to an ASCII data file with default name 'BurstyErrs.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

The program is run by editing the parameter file 'BurstyErrs.prm' and selecting the appropriate parameters and by choosing a program ID by editing file 'ID.prm'. Executing the program generates the 'error.seq' file which contains an error sequence (in packed format) with binomially distributed errors. It does not matter whether the output file 'error.seq' exists or not. *If it exists, it is overwritten without a prompt to the user.*

Even though Poisson distributed bursts may overlap in theory, this program does not allow error bursts to overlap. The user must take care to specify input parameters so that the probability of overlapping bursts is negligible. It is also assumed that $P_{eg} < P_{eb}$.

## C. Interleavers

The contract requires that block interleavers and periodic convolutional interleavers be considered. To this end, a program is described which implements the effect of a block interleaver and a separate program is described which implements the effect of a periodic convolutional interleaver. Also, there are two versions of each program. The two versions implement the same operation but trade off computer code complexity for execution speed.

### 1. BlockInt (Block Deinterleaver)

This program performs block deinterleaving of the error sequence found in file 'error.seq'. It is assumed that the channel symbols corresponding to those errors have already been interleaved using an $(C,R,m)$ block interleaver. The deinterleaver groups every $m$ error sequence values together and deinterleaves them as a group. The method used to implement the function of the block interleaver is to read in a block of the error sequence and to use a series of formulas to perform the block deinterleaving. These formulas are described below.

Let $b_K$ denote the error sequence input to the deinterleaver and let $d_L$ denote the error sequence output by the deinterleaver. Note: the subscripts are assumed to be incremented starting with zero. Then $b_K$ is read into the deinterleaver memory array (by columns) at location:

Symbol index = $\text{int}(K/m) == Y$
Row of $b_K = \text{Mod}(Y,R) == i$
Column of $b_K = \text{int}(Y/R) == j$
Depth of $b_K = \text{Mod}(K,m) == p$

Given $i$, $j$, and $p$ the deinterleaved value location (read out by rows) is found to be

$$L = m * (i*C+j) + p$$

The implementation found below actually calculates $K$ given $L$. The actual value $b_K$ is found in a buffer which is loaded with error sequence values. The calculation is as follows:

1) $L$ points to location BuffL in the buffer, BuffL = Mod($L$,BuffLength)

2) The interleaved location for BuffL is BuffK where

$ll$ = Mod(BuffL,$m$)
$X$ = BuffL/$m$
BuffK = $m$ * ($R$*Mod($X$,$C$) + int4($X/C$)) + $ll$

where BuffLength=$R*C*m$. Note that there is a problem deinterleaving the end of the 'error.seq' file due to a possible partial interleaver block at the end of the sequence. The program attempts to partially deinterleave this last partial block. An error sequence could be zero padded to fill a partial block, thereby changing slightly the overall error statistics.

This program inputs parameters from an ASCII data file with default name 'BlockInt.prm' and outputs the error sequence to data file with default name 'error.seq'. In addition, various statistics are output to an ASCII data file with default name 'BlockInt.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

The program is run by editing the parameter file 'BlockInt.prm' and selecting the appropriate parameters and by choosing a program ID by editing file 'ID.prm'. Executing the program generates the 'error.seq' file which contains an error sequence (in packed format) with deinterleaved errors. *The 'error.seq' file must exist prior to the execution of this program.*

There are no assumptions associated with the implementation or output of this program.

### 2. BlkArr (Alternate Block Deinterleaver)

This program performs block deinterleaving of the error sequence found in file 'error.seq'. It is assumed that the channel symbols corresponding to those errors have already been interleaved using an $(C,R,m)$ block interleaver. The deinterleaver groups every $m$ error sequence values together and deinterleaves them as a group. The method used to implement the function of the block interleaver is to read in a block of the error sequence into a buffer which mimics the block interleaver memory array. The error sequence is read in by rows and deinterleaving is performed by reading the error sequence out by columns.

This program inputs parameters from an ASCII data file with default name 'BlockInt.prm' and outputs the error sequence to data file with default name 'error.seq'. In addition, various statistics are output to an ASCII data file with default name 'BlockInt.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

The program is run by editing the parameter file 'BlockInt.prm' and selecting the appropriate parameters and by choosing a program ID by editing file 'ID.prm'. Executing the program generates the 'error.seq' file which contains an error sequence (in packed format) with deinterleaved errors. *The 'error.seq' file must exist prior to the execution of this program.*

There are no assumptions associated with the implementation or output of this program.

### 3. DPCI (Periodic Convolutional Deinterleaver)

This program performs deinterleaving of the error sequence found in file 'error.seq'. It is assumed that the channel symbols corresponding to those errors have already been interleaved using an (Ntaps,$M$) periodic convolution interleaver. The method used to implement the function of the periodic convolutional interleaver is a series of formulas as described below. These functions are applied to a portion of the error.seq array which is stored in a ring buffer.

Let $b_K$ denote the error sequence input to the DPCI and let $d_L$ denote the error sequence output by the DPCI. Then the index $L$ relates to the index $K$ as follows,

$K = \text{Mod}((L\text{-}1),\text{Ntaps}) * M * \text{Ntaps} + L$

Note that there is a problem deinterleaving the end of the 'error.seq' file due to the sequential nature of the algorithm. The DPCI error sequence file is truncated to eliminate the "don't cares".

This program inputs parameters from an ASCII data file with default name 'DPCI.prm' and outputs the error sequence to data file with default name 'error.seq'. In addition, various statistics are output to an ASCII data file with default name 'DPCI.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

The program is run by editing the parameter file 'DPCI.prm' and selecting the appropriate parameters and by choosing a program ID by editing file 'ID.prm'. Executing the program generates the 'error.seq' file which contains an error sequence (in packed format) with deinterleaved errors. *The 'error.seq' file must exist prior to the execution of this program.*

There are no assumptions associated with the implementation or output of this program.

### 4. DPCIAlt (Alternate Periodic Convolutional Deinterleaver)

This program performs deinterleaving of the error sequence found in file 'error.seq'. It is assumed that the channel symbols corresponding to those errors have already been interleaved using an $(n,M)$ periodic convolution interleaver. The method used to implement the function of the periodic convolutional interleaver is a series of formulas as described below.

Let $b_i$ denote the error sequence input to the DPCI and let $d_j$ denote the error sequence output by the DPCI. Then the index $j$ relates to the index $i$ as follows,

$$j = i - [(i\text{-}1) \bmod n]*M*n$$

Note that there is a problem deinterleaving the end of the 'error.seq' file due to the sequential nature of the algorithm. For this case, the 'error.seq' file is filled with zeroes for those deinterleaved positions which result from locations which are beyond the end of the 'error.seq' file.

This program inputs parameters from an ASCII data file with default name 'DPCI.prm' and outputs the error sequence to data file with default name 'error.seq'. In addition, various statistics are output to an ASCII data file with default name 'DPCI.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

The program is run by editing the parameter file 'DPCI.prm' and selecting the appropriate parameters and by choosing a program ID by editing file 'ID.prm'. Executing the program generates the 'error.seq' file which contains an error sequence (in packed format) with deinterleaved errors. *The 'error.seq' file must exist prior to the execution of this program.*

There are no assumptions associated with the implementation or output of this program.

## D. Error Sequence Analysis

The contract requires that error sequence be characterized. This amounts to modeling the errors by a predefined mathematical model. Several mathematical models are considered; one which models the errors as bursty errors, and one which models the errors as burst errors. Bursty errors are characterized by errors which occur within bursts as well as errors which occur outside bursts. Burst errors are characterized by errors which occur only within bursts. In addition, two programs have been written to implement distribution tests for the purpose of determining if an error sequence, or a segment of an error sequence, resulted from random errors.

### 1. CVMseq (Cramer Von-Mises sequence distribution test)

This program uses the Cramer Von-Mises (CVM) distribution test to determine whether the error sequence (in default file 'error.seq') is binomially distributed with confidence level alpha. The method implemented is simple. The error sequence is read in by blocks and the overall CVM test statistic is calculated. At the end of the program, the test statistic for the complete sequence along with a preselected set of critical values is output to the user. The results are also output to 'CVMseq.ID' file where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

Executing the program causes the 'error.seq' file to be read which contains an error sequence (in packed format). *The 'error.seq' file must exist prior to the execution of this program.* There are no assumptions associated with the implementation or output of this program.

### 2. CVMblk (Cramer Von-Mises distribution test on error sequence blocks)

This program uses the Cramer Von-Mises (CVM) distribution test to determine whether the error sequence (in default file 'error.seq') is binomially distributed with confidence level alpha. The error sequence is read in by blocks and the CVM test statistic is calculated for each

block. At the end of the program, the test statistics for each block along with a preselected set of critical values are ordered and output to the user. The results are also output to 'CVMblk.ID' file where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

Executing the program causes the 'error.seq' file to be read which contains an error sequence (in packed format). *The 'error.seq' file must exist prior to the execution of this program.* There are no assumptions associated with the implementation or output of this program.

### 3. DeltaEst (Bursty-error parameter estimation via the Δ method)

This program estimates parameters associated with a bursty error sequence. The method employed segments the error sequence into random error regions and burst error regions. The algorithm implemented operates on the error sequence iteratively. For each iteration, the algorithm is either tracking a burst segment or a random segment. At each iteration, the error sequence interval to the next error is found. If the algorithm is tracking a random segment, then an attempt is made to begin a burst by comparing the error density for the $i^{th}$ interval (surrounded by 2 errors which gives an effective error density of 2/[interval+2]) with a threshold (Delta). If the error density for the $i^{th}$ interval is greater than Delta, then the algorithm begins tracking a burst segment, if not then the random segment is continued. If the algorithm is tracking a burst segment, then the segment is continued until the error density within the total burst segment falls below the threshold, Delta. In this way, the entire sequence is partitioned. Initializing the processes is particularly troublesome because of the various combinations for the beginning of the error seq.

This program inputs parameters from an ASCII data file with default name 'DeltaEst.prm' and outputs the error sequence to data file with default name 'error.seq'. In addition, various statistics are output to an ASCII data file with default name 'DeltaEst.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

The program is run by editing the parameter file 'DeltaEst.prm' and selecting the appropriate parameters and by choosing a program ID by editing file 'ID.prm'. Executing the program generates the 'error.seq' file which contains an error sequence (in packed format) with deinterleaved errors. *The 'error.seq' file must exist prior to the execution of this program.*

There are no assumptions associated with the implementation or output of this program.

## 4. IntvHst (Error Interval Histogram)

This program calculates the error interval probability density function for an error sequence. The error sequence is partitioned into error free segments and a histogram of the interval length calculated. Note that the two error free intervals occurring at the beginning of the error sequence and at the end are ignored. Only intervals between errors are counted.

The program outputs the histogram to file 'Interval.hst' which (for now) is an ASCII file with each histogram value stored per record. For each record, the interval index appears first followed by the probability of occurrence.

Note that there are NO parameters to be read in for this program. However, various statistics are output to an ASCII data file with default name 'IntvHst.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

The program is run by editing the parameter file 'IntvHst.prm' and selecting the appropriate parameters and by choosing a program ID by editing file 'ID.prm'. Executing the program generates the 'Interval.hst' file which contains the histogram of the error intervals found in the error sequence. *The 'error.seq' file must exist prior to the execution of this program.*

There are no assumptions associated with the implementation or output of this program.

## 5. GAPEst (fixed GAP burst error distribution Estimation)

This program estimates parameters associated with a bursty error sequence. The method employed segments the error sequence into error free regions and burst error regions. A burst error region is defined to be a region which contains errors no two of which are separated by more than the prespecified GAP number of error free symbols. In addition, the burst error region is preceded and followed by error free regions of minimum width specified by GAP. The algorithm implemented operates on the error sequence iteratively. For each iteration, the algorithm determines the width of the next error free interval, if it is less than GAP then the next error is included in the current burst, if it is greater than GAP then the previous burst is terminated and the next burst is started. In this way, the entire sequence is partitioned. If the first error sequence value is a '0' then the process always begins with an error free region. If the first error sequence value is a '1' then the process always begins with an error burst.

This program inputs parameters from an ASCII data file with default name 'GAPEst.prm' and outputs the error sequence to data file with default name 'error.seq'. In addition, various statistics are output to an ASCII data file with default name 'GAPEst.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

The program is run by editing the parameter file 'GAPEst.prm' and selecting the appropriate parameters and by choosing a program ID by editing file 'ID.prm'. Executing the program causes the 'error.seq' file to be read which contains an error sequence (in packed format). *The 'error.seq' file must exist prior to the execution of this program.*

There are no assumptions associated with the implementation or output of this program.

## E. Utilities

Several utilities have been developed to support CLEAN. The *makefile* given in the following section can be used to compile the source code with a single command by typing 'make all'. The programs which follow allow the user to compare error sequences, set error sequences, and display error sequences.

### 1. Make Utility for Lahey Fortran v5.0

```
FFLAGS = /3 /B /nA1 /C1 /P /R /Z1

CorrCW   = CorrCW.obj Unpack.obj Pack.obj
GaussRV  = GaussRV.obj UnifRV.obj
IterBin  = IterBin.obj UnifRV.obj
LdBuff1  = LdBuff1.obj Unpack.obj
LdBuff4  = LdBuff4.obj Unpack.obj
NextBrst = NextBrst.obj UnifRV.obj GaussRV.obj
NextInt  = NextInt.obj LdBuff4.obj
NextLnth = NextLnth.obj UnifRV.obj GaussRV.obj
SvBuff1  = SvBuff1.obj Unpack.obj Pack.obj
SvBuff4  = SvBuff4.obj Unpack.obj Pack.obj
TotalPe  = TotalPe.obj Unpack.obj

ALL : BinErrs BlkArr BlkDecod Blockint BrstErrs \
      BstyErrs CompSeq CVMblk CVMseq DeltaEst DisplFil DisplSeq \
      DPCI DPCIOld GAPest IntvHst SetErrs

BINERRS : BINERRS.obj $(IterBin) Pack.obj UnifRV.obj
  Optlink BINERRS.obj $(IterBin) Pack.obj UnifRV.obj, \
          BINERRS.exe,,c:\compiler\lahey\F77L.LIB

BLKARR :  BLKARR.obj $(LdBuff1) $(SvBuff1) DispBuf1.obj
  Optlink BLKARR.obj $(LdBuff1) $(SvBuff1) DispBuf1.obj, \
          BLKARR.exe,,c:\compiler\lahey\F77L.LIB

BLKDECOD : BLKDECOD.obj $(LdBuff4) $(CorrCW)
  Optlink  BLKDECOD.obj $(LdBuff4) $(CorrCW), \
           BLKDECOD.exe,,c:\compiler\lahey\F77L.LIB

BLOCKINT : BLOCKINT.obj $(LdBuff1) $(SvBuff1) Pack.obj
  Optlink  BLOCKINT.obj $(LdBuff1) $(SvBuff1) Pack.obj, \
           BLOCKINT.exe,,c:\compiler\lahey\F77L.LIB

BRSTERRS : BRSTERRS.obj $(IterBin) Pack.obj UnifRV.obj $(NextBrst) $(NextLnth)
  Optlink  BRSTERRS.obj $(IterBin) Pack.obj UnifRV.obj $(NextBrst) $(NextLnth) , \
           BRSTERRS.exe,,c:\compiler\lahey\F77L.LIB

BSTYERRS : BSTYERRS.obj $(IterBin) Pack.obj UnifRV.obj $(NextBrst) $(NextLnth)
  Optlink  BSTYERRS.obj $(IterBin) Pack.obj UnifRV.obj $(NextBrst) $(NextLnth) , \
           BSTYERRS.exe,,c:\compiler\lahey\F77L.LIB
```

```
COMPSEQ: COMPSEQ.obj Unpack.obj
  Optlink COMPSEQ.obj Unpack.obj, \
          COMPSEQ.exe,,c:\compiler\lahey\F77L.LIB

CVMblk :  CVMblk.obj $(LdBuff4) RdStats.obj
  Optlink CVMblk.obj $(LdBuff4) RdStats.obj, \
          CVMblk.exe,,c:\compiler\lahey\F77L.LIB

CVMseq:  CVMseq.obj $(LdBuff4) $(NextInt) RdStats.obj
  Optlink CVMseq.obj $(LdBuff4) $(NextInt) RdStats.obj, \
          CVMseq.exe,,c:\compiler\lahey\F77L.LIB

DELTAEST : DELTAEST.obj $(LDBuff4) $(NextInt) $(TotalPe)
  Optlink  DELTAEST.obj $(LDBuff4) $(NextInt) $(TotalPe) , \
           DELTAEST.exe,,c:\compiler\lahey\F77L.LIB

DISPLFIL : DISPLFIL.obj Unpack.obj
  Optlink  DISPLFIL.obj Unpack.obj, \
           DISPLFIL.exe,,c:\compiler\lahey\F77L.LIB

DISPLSEQ: DISPLSEQ.obj Unpack.obj
  Optlink  DISPLSEQ.obj Unpack.obj, \
           DISPLSEQ.exe,,c:\compiler\lahey\F77L.LIB

DPCI :    DPCI.obj Unpack.obj Pack.obj
  Optlink DPCI.obj Unpack.obj Pack.obj, \
          DPCI.exe,,c:\compiler\lahey\F77L.LIB

DPCIOLD : DPCIOLD.obj Unpack.obj Pack.obj
  Optlink DPCIOLD.obj Unpack.obj Pack.obj, \
          DPCIOLD.exe,,c:\compiler\lahey\F77L.LIB

GAPEST : GAPEST.obj $(LdBuff4) $(NextInt)
  Optlink  GAPEST.obj $(LdBuff4) $(NextInt) , \
           GAPEST.exe,,c:\compiler\lahey\F77L.LIB

IntvHst :  IntvHst.obj $(LDBuff4) $(NextInt)
  Optlink  IntvHst.obj $(LDBuff4) $(NextInt) , \
           IntvHst.exe,,c:\compiler\lahey\F77L.LIB

SETERRS : SETERRS.obj Pack.obj
  Optlink SETERRS.obj Pack.obj, \
          SETERRS.exe,,c:\compiler\lahey\F77L.LIB


BINERRS.obj : BINERRS.for
  F77L BINERRS.for $(FFLAGS)

BLKARR.obj : BLKARR.for
  F77L BLKARR.for $(FFLAGS)

BLKDECOD.obj : BLKDECOD.for
  F77L BLKDECOD.for $(FFLAGS)

BLOCKINT.obj : BLOCKINT.for
  F77L BLOCKINT.for $(FFLAGS)

BRSTERRS.obj : BRSTERRS.for
  F77L BRSTERRS.for $(FFLAGS)

BSTYERRS.obj : BSTYERRS.for
  F77L BSTYERRS.for $(FFLAGS)

COMPSEQ.obj : COMPSEQ.for
  F77L COMPSEQ.for $(FFLAGS)

CVMblk.obj : CVMblk.for
  F77L CVMblk.for $(FFLAGS)

CVMseq.obj : CVMseq.for
  F77L CVMseq.for $(FFLAGS)

CorrCW.obj : CorrCW.for
  F77L CorrCW.for $(FFLAGS)

DELTAEST.obj : DELTAEST.for
  F77L DELTAEST.for $(FFLAGS)

DispBuf1.obj : DispBuf1.for
  F77L DispBuf1.for $(FFLAGS)

DispBuf4.obj : DispBuf4.for
  F77L DispBuf4.for $(FFLAGS)

DISPLFIL.obj : DISPLFIL.for
  F77L DISPLFIL.for $(FFLAGS)

DISPLSEQ.obj : DISPLSEQ.for
  F77L DISPLSEQ.for $(FFLAGS)

DPCI.obj : DPCI.for
  F77L DPCI.for $(FFLAGS)

DPCIOLD.obj : DPCIOLD.for
  F77L DPCIOLD.for $(FFLAGS)
```

```
GAPEST.obj : GAPEST.for
  F77L GAPEST.for $(FFLAGS)
GAUSSRV.obj : GAUSSRV.for
  F77L GAUSSRV.for $(FFLAGS)
IntvHst.obj : IntvHst.for
  F77L IntvHst.for $(FFLAGS)
ITERBIN.obj : ITERBIN.for
  F77L ITERBIN.for $(FFLAGS)
LDBUFF1.obj : LDBUFF1.for
  F77L LDBUFF1.for $(FFLAGS)
LDBUFF4.obj : LDBUFF4.for
  F77L LDBUFF4.for $(FFLAGS)
NEXTBRST.obj : NEXTBRST.for
  F77L NEXTBRST.for $(FFLAGS)
NEXTINT.obj : NEXTINT.for
  F77L NEXTINT.for $(FFLAGS)
NEXTLNTH.obj : NEXTLNTH.for
  F77L NEXTLNTH.for $(FFLAGS)
PACK.obj : PACK.for
  F77L PACK.for $(FFLAGS)
RdStats.obj : RdStats.for
  F77L RdStats.for $(FFLAGS)
SETERRS.obj : SETERRS.for
  F77L SETERRS.for $(FFLAGS)
SVBUFF1.obj : SVBUFF1.for
  F77L SVBUFF1.for $(FFLAGS)
SVBUFF4.obj : SVBUFF4.for
  F77L SVBUFF4.for $(FFLAGS)
TotalPe.obj : TotalPe.for
  F77L TotalPe.for $(FFLAGS)
UNIFRV.obj : UNIFRV.for
  F77L UNIFRV.for $(FFLAGS)
UNPACK.obj : UNPACK.for
  F77L UNPACK.for $(FFLAGS)
```

## 2. CompSeq (Compare Sequence)

This program compares two error sequences and identifies those error locations where the two are different. The user is prompted for the two error sequence filenames. It is assumed that the errors stored in error.seq are in the DBESS (Double Byte Error Sequence Symbol) packed format.

## 3. SetErrS (Set Error Pattern)

This program interactively allows the user to input an error sequence. All parameters and the error sequence are input directly from the user so that there is no parameter file associated with this program. The errors are stored in the DBESS packed format.

There are no assumptions associated with the implementation or output of this program.

### 4. DisplSeq (Display Default Error Sequence)

This program displays the error sequence found in file 'error.seq'. It is assumed that the errors stored in error.seq are in the DBESS packed format.

### 5. DisplFil (Display Error Sequence from user File)

This program displays the error sequence found in a file specified by the user. It is assumed that the errors stored in the file are in the DBESS packed format.

## IV. NASA GSFC/MSU INTERRELATED CAPABILITIES

To enhance the research efforts at both MSU and NASA GSFC, several interrelated capabilities have been established. The first author visited GSFC in August of 1992 to learn how to use the Communications Link And System Simulation (CLASS) software tool. CLASS performs a signal level simulation of the TDRS downlink and predicts coded system performance using theoretical analysis. In addition, the first author learned how to use the OMV bit-by-bit simulator which uses the same signal level simulation nucleus as CLASS but also incorporates actual deinterleaving and decoding algorithms to simulate the operation of the deinterleavers and decoders at White Sands. After learning how to use these software tools, analyst level access was granted and has been established. It is now possible for MSU personnel to exercise CLASS and the OMV bit-by-bit simulator remotely from MSU via internet. MSU appreciates the support given by the NASA/GSFC CLASS group.

Furthermore, real EOS Ku-band downlink data (validity of the data pending) has been acquired by Victor Sank at GSFC. A program was written to convert from the GSFC error sequence data format into the format required by CLEAN. Since these data files are sometimes rather large which requires large storage spaces, a second program was written to archive the GSFC data using run length encoding, a *lossless* compression scheme. For an error sequence with an error probability of $10^{-3}$, this provides about 3:1 lossless compression. For an error sequence with an error probability of $10^{-4}$, this provides about 30:1 lossless compression. In addition, a third program was written to unarchive the run length encoded data into the DBESS format required by CLEAN. Mr. Sank's help has been invaluable to this project.

### A. EOS Real Error Sequence Data conversion program

This program inputs the real EOS downlink data obtained from Victor Sank and converts it into the DBESS packed format required by the programs in CLEAN.

*It is assumed that the input file accessed by this program exists prior to its execution.*

### B. Error Sequence Archiver using Run Length Encoding

This program inputs the real EOS downlink data obtained from Victor Sank and converts it into an archival format. The archival format only stores the location of each error in the file. This is *not* the format which is necessary for CLEAN. Another program called SeqUnarc can be executed to convert from the Archival format to the DBESS format required by CLEAN.

*It is assumed that the input file accessed by this program exists prior to its execution.*

## C. Error Sequence Unarchiver

This program inputs data in the archival format (run length encoding) via the SeqArc program and unarchives it to the DBESS format required by CLEAN.

*It is assumed that the input file accessed by this program exists prior to its execution.*

## V. PREVIEW OF EXPECTED RESULTS

The problem of interest is that of choosing/evaluating a good forward error correcting coding (FEC) scheme for the Ku-band TDRS downlink which will be used for the Earth Observation System (EOS). There are many issues to be considered when choosing a "good" FEC including required error probability, required data rate, and data loss during synchronization cycles just to name a few.

For example, suppose it is proposed to use a (255,223) Reed-Solomon (RS) code with a block interleaver for the 150Mbps Ku-band TDRS downlink. If this code meets the required error probability, say $10^{-5}$, for the types and density of errors anticipated on the link and if it can accommodate the required data rate, 150Mbpsx(223/255)=131Mbps, then this code can be considered acceptable. If the decision is made to concatenate a rate 1/2 convolutional encoder and periodic convolutional interleaver with the RS code and block interleaver, then several undesirable side effects will take place. First, the hardware complexity will increase which will increase cost, size, weight, power, etc. Second, the periodic convolutional deinterleaver and Viterbi decoder at the receiver must synchronize to the received data. The synchronization process can result in significant data loss. In addition, the convolutional code rate results in a decrease in the system data rate to 131Mbpsx(1/2)=65.6Mbps, assuming a fixed channel rate. Although this concatenated scheme may provide a lower error probability which exceeds the requirement, it is achieved at a significant cost. Therefore, the studies developed for this contract focus on determining and evaluating the minimum complexity coding scheme for EOS to satisfy the system requirements. This requires an understanding of the nature of the Ku-band downlink errors and of the achievable performance for various coding schemes in various types of error environments.

To this end, the research is being focussed along two main lines as discussed in the following sections.

### A. Research Focus 1

First, the nature of the downlink errors is being investigated. The expected results are a consequence of discussions with NASA/GSFC and STEL personnel concerning the nature of the Ku-band downlink errors. The expected results are:

1) Determine that the expected errors which occur in a received block of data are *not* random. This is accomplished by applying the Cramer Von-Mises distribution test (see CVMblk in Section III.D.2) to the actual data.

2) Estimate the error parameters for the actual channel data assuming that the errors are bursty in nature. These will be estimated by applying the bursty-error parameter estimation via the Δ method (see DeltaEst in Section III.D.3) to the actual data. It is expected that the burst locations follow a Poisson distribution. The estimated parameters are:

   a) Average rate of burst occurrence and the burst occurrence interval probability density function (*pdf*). It is expected that this *pdf* is exponential which means that the burst locations follow a Poisson distribution.
   b) The average burst length (in channel symbols) and the burst length *pdf*. It is expected the variance of this *pdf* is small.
   c) The average error density during the bursts and the burst error density *pdf*. It is expected that the variance of this *pdf* is small.
   d) The average error density outside the bursts. It is expected that this error density will be very nearly the random error rate.

Because the actual data has not been received to date, this work has not been completed.

## B. Research Focus 2

The second focus of this research is the investigation of performance for various coding schemes in a bursty-error environment. The expected result will be plots similar to the one shown in Figure 2. Several coding schemes will be considered including:

1) Reed-Solomon (RS)
2) RS, block interleaver (interleave depth of 5)
3) RS, block interleaver (interleave depth of 8)
4) RS outer code, block interleaver (interleave depth of 5), convolutional inner code
5) RS outer code, block interleaver (interleave depth of 5), convolutional inner code, periodic convolutional interleaver.

The curves drawn are for illustration only but *do* indicate to some degree the expected shape. The *error ratio* $R_\varepsilon$, as defined in this research, is

$$R_\varepsilon = \frac{\text{Total Random Errors}}{\text{Total Errors}}$$

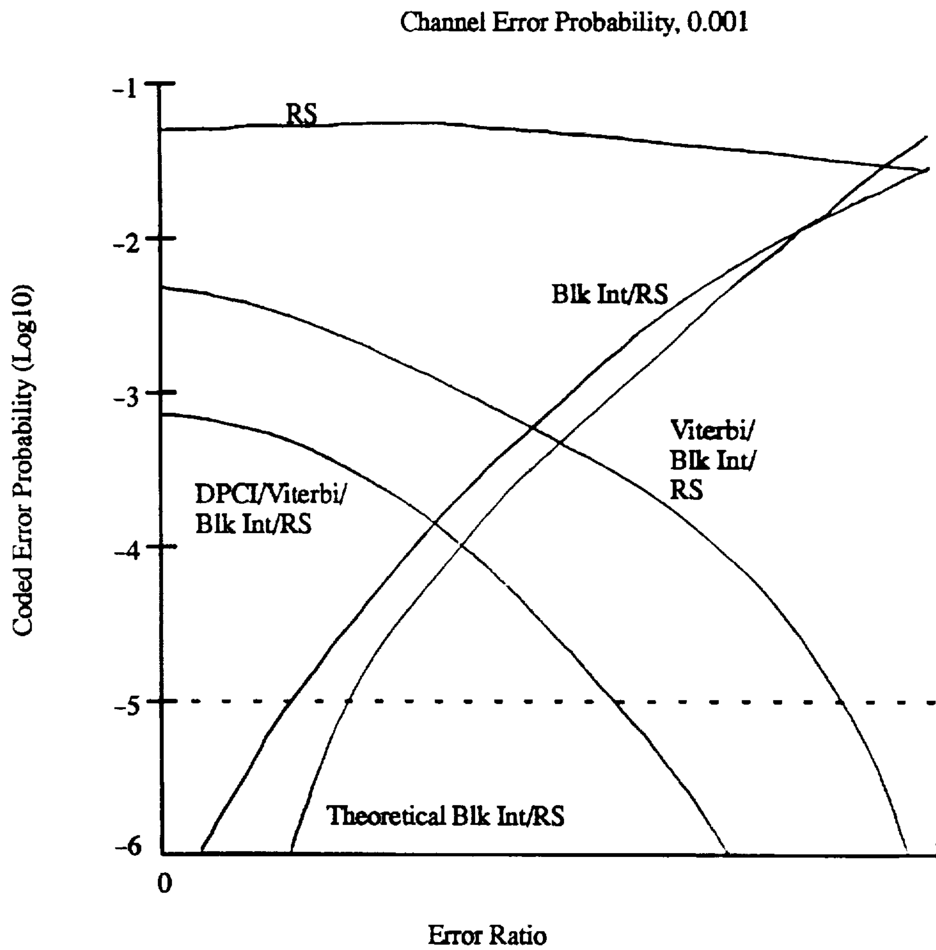


Figure 2. An expected output performance data product (for illustration only).

To construct Figure 2, a channel error probability, $P(\varepsilon_{ch})$, is chosen. For each possible error rate, the bursty-error parameters are calculated and CLEAN is used to calculate the decoded error probability. For example, to simulate system (5) identified above, the following programs are sequentially executed:

1) BstyErrs (see Section III.B.3)
2) DPCI (see Section III.C.3)
3) Viterbi (see Section III.A.2)
4) BlockInt (see Section III.C.1)
5) BlkDecod (see Section III.A.1)

The input parameters must be chosen and input to the appropriate parameter files. The choice for the input parameters are discussed in the following section. The file 'BlkDecod.ID' where ID is the 3 letter identifier found in file 'ID.prm' gives the final decoded error probability. Note that CLEAN performs a Monte Carlo simulation.

It is expected that the actual plot, similar to that shown in Figure 2, will show that the Reed-Solomon code used with a block interleaver (interleave depth of 5) is sufficient to provide the required decoded error probability and, therefore, constitutes the "best" coding scheme.

To date, about 10% of the actual plot has been developed for the choice of parameters discussed in the following section. The required execution time of some of the programs is on the order of hours per data point for a SPARC workstation.

## C. Choosing System Parameters

Of interest in this research are performance results for codes which are used for space based communication systems. The Consultative Committee for Space Data Systems (CCSDS) [1] defines a concatenated coding scheme for space based communication systems consisting of a (255,223) RS outer code followed by an interleaver and a rate 1/2 constraint length 7 convolutional inner code. Therefore, these are the code parameters chosen for study in this research. To summarize

1) Reed-Solomon code (BlkDecod program)
    a) Blocklength, $n$=255
    b) Information codeword length, $k$=223
    c) Number of binary symbols per codeword, $m$=8
    c) Error correcting capability, $t$=16 code symbols per codeword
2) Convolutional code (Viterbi program)

a) Constraint length, $K$=7
b) Number of code generators, 2 (code rate = 1/2)
c) Tap weights for code generator #1, 1011011
d) Tap weights for code generator #2, 1111001
e) Number of constraint lengths for decoder memory, 4

Also of interest are the interleaver parameters. The Framing and Multiplex Equipment (FAME) defines a standard architecture for space based communication systems which involves multiplexing 8 (only 5 are utilized) data streams together to form a single data stream for transmission to earth. This results in a block interleaving effect for the demultiplexed data input to the RS decoder. Therefore, the block interleaver imitates the multiplex operation. For the (255,223) RS code defined above, this requires the block interleaver parameters to be chosen as

3) Block interleaver (BlockInt)
   a) Number of rows, 5 (This is alternately chosen to be 8)
   b) Number of columns, 255
   c) Number of binary symbols per memory array element, 8

In addition, the periodic convolutional interleaver currently used has parameters given by

4) Periodic Convolutional Interleaver (DPCI)
   a) Number of taps, 30
   b) Number of delays for the 2nd tap, 2

The only parameters remaining to be specified are the bursty-error parameters. This requires choosing the burst duration *pdf* be chosen along with the mean and possibly the variance, the burst location *pdf* be chosen along with the mean and possibly the variance, the error density within the bursts, and the error density outside the bursts. These parameters must be chosen for the given raw channel error probability, $P(\varepsilon_{ch})$, and for each possible value for the *error ratio*.

It is known that the Ku-band downlink is characterized by essentially error free transmission interrupted by short, fixed periods of high interference. The interference is probably less than 0.3μsec in duration. Although the average time between error bursts is unknown, the duty cycle of the interference is probably less than 0.025. Given this information, a worse case scenario can be constructed. If the worse case interference duration is 0.3μsec and the channel symbol rate is 75Mbps (2 binary symbols per channel symbol for QPSK gives rise to the required 150Mbps), then $(0.3\times10^{-6})(75\times10^{6}\text{bps})$(2bits/channel symbol)=45 binary symbols is the length of each error burst. As an aside, it is easy to determine that a (255,223) RS code with

a depth 5 block interleaver can correct an error burst of 45 binary symbols. However, it is possible for multiple error bursts to occur within one interleaved block. In light of this characterization, some of the bursty-error parameters are chosen as follows

5) Bursty-error Generation (BstyErrS)
   a) Burst occurrence location *pdf*, IntvFlag=3 (Poisson)
   b) Burst occurrence interval mean, IntvMean=4500 binary symbols
   c) Burst occurrence duration *pdf*, LngthFlag=1 (Fixed)
   d) Burst occurrence duration mean, LngthMean=45 binary symbols

The only two parameters remaining to be chosen are the error probability during the error bursts, $P_{eb}$, and the error probability outside the error bursts, $P_{eg}$. Choosing these is more involved than the previous parameters because they must be calculated for the predefined raw channel error probability, $P(\varepsilon_{ch})$, and because they must be changed to adjust the *error ratio*.

The method for calculating $P_{eb}$ and $P_{eg}$ in terms of $P(\varepsilon_{ch})$ and $R_\varepsilon$ is as follows. From [2], the raw channel error probability for a bursty-error channel is given by

$$P(\varepsilon_{ch}) = P_{eg}(1 - d/M_v) + P_{eb}(d/M_v)$$

where $M_v$ is the average interval between error bursts (denoted IntvMean in part 4.b above) and where $d$ is the burst duration (denoted LngthMean in part 4.d above). The error ratio can be expressed in terms of these symbols to be

$$R_\varepsilon = \frac{P_{eg}(1 - d/M_v)}{P(\varepsilon_{ch})}$$

Solving the previous two equations for $P_{eg}$ and $P_{eb}$ gives

$$P_{eg} = \frac{R_\varepsilon P(\varepsilon_{ch})}{1 - d/M_v}$$

and

$$P_{eb} = \frac{M_v}{d}(1 - R_\varepsilon)P(\varepsilon_{ch})$$

These are valid provided $P_{eb} \geq P_{eg}$. Note that for given values of $d$ and $M_v$, it is generally not possible for the *error ratio* to take on all values from 0 to 1. Clearly, $P_{eg} \leq P_{eb} \leq 1/2$, from which it can be determined that

$$0 \leq \frac{P(\varepsilon_{ch}) - d/(2M_v)}{P(\varepsilon_{ch})} \leq R_\varepsilon \leq 1 - d/M_v \leq 1$$

which implies that we must have

$$P(\varepsilon_{ch}) \geq \frac{d}{2M_v}$$

Note that if we choose $P(\varepsilon_{ch}) = d/(2M_v)$ then it is possible to achieve a range for the error ratio of $0 \leq R_\varepsilon \leq 1$ by selecting appropriate values for $P_{eg}$ and $P_{eb}$.

## BIBLIOGRAPHY

1. Consultative Committee for Space Data System, "Recommendations for space data system standards: Telemetry channel coding." *Blue Book*, May 1984.

2. Ebel, W.J., Simulation and Evaluation of Reed-Solomon Codes in a Burst Noise Environment, Ph.D. Dissertation, University of Missouri-Rolla, 1991.