A Study on the Effects of Decoder Quantization of Digital Video Broadcasting – Return Channel over Satellite (DVB-RCS) Turbo Codes

A thesis presented to

the faculty of

the Russ College of Engineering and Technology of Ohio University

In partial fulfillment

of the requirements for the degree

Master of Science

Anantha Surya Raghu Gorthy

November 2008

© 2008 Anantha Surya Raghu Gorthy. All Rights Reserved.
This thesis titled
A Study on the Effects of Decoder Quantization of Digital Video Broadcasting – Return
Channel over Satellite (DVB-RCS) Turbo Codes

by

ANANTHA SURYA RAGHU GORTHY

has been approved for
the School of Electrical Engineering and Computer Science
and the Russ College of Engineering and Technology by

Jeffrey C. Dill
Professor of Electrical Engineering and Computer Science

Dennis Irwin
Dean, Russ College of Engineering and Technology
ABSTRACT

GORTHY, ANANTHA SURYA RAGHU, M.S., November 2008, Electrical Engineering

A Study on the Effects of Decoder Quantization of Digital Video Broadcasting – Return Channel over Satellite (DVB-RCS) Turbo Codes (71 pp.)

Director of Thesis: Jeffrey C. Dill

Digital Video Broadcasting – Return Channel over Satellite (DVB-RCS) Turbo Codes have become increasingly popular because of their ability to provide an uplink and downlink on the same path. However, the hardware implementation of these codes remains a challenge to the engineers and designers. This research focuses on the implications of quantizing a decoder’s input to achieve a significant improvement in the hardware implementation of the decoding architecture. The performance issues related to the quantization have been studied in detail. An approach to achieving performances very close to the floating point methodologies has been presented in the form of an algorithm. Results show that by sacrificing very little performance, cost effective and optimal hardware designs can be obtained.

Approved: _____________________________________________________________

Jeffrey C. Dill

Professor of Electrical Engineering and Computer Science
ACKNOWLEDGMENTS

I am grateful to my advisor, Dr. Jeffrey C. Dill who guided me through the entire research and provided valuable suggestions towards achieving my research objectives. This thesis would not have been possible without his support and encouragement.

I thank my committee members Dr. David Matolak, Dr. Maarten Uijt de Haag and Dr. Vardges Melkonian for their time and patience.

I would also like to thank my friends Kamalakar Ganti, Irina and Hunter for providing valuable suggestions throughout this research.

I am extremely thankful to my family members for supporting me throughout my graduate program and my entire education.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
</tr>
<tr>
<td>Acknowledgments</td>
</tr>
<tr>
<td>List of Tables</td>
</tr>
<tr>
<td>List of Figures</td>
</tr>
<tr>
<td>Chapter 1: Error Correcting Codes</td>
</tr>
<tr>
<td>1.1 History</td>
</tr>
<tr>
<td>1.2 Outline of the thesis</td>
</tr>
<tr>
<td>Chapter 2: Digital Communication</td>
</tr>
<tr>
<td>2.1 Digital Communication Systems</td>
</tr>
<tr>
<td>2.2 Shannon’s Thoerem</td>
</tr>
<tr>
<td>2.3 Error Control Codes</td>
</tr>
<tr>
<td>2.3.1 Block Codes</td>
</tr>
<tr>
<td>2.3.2 Convolutional Codes</td>
</tr>
<tr>
<td>2.4 Trellis Codes</td>
</tr>
<tr>
<td>2.4.1 Zero Tailing</td>
</tr>
<tr>
<td>2.4.2 Tail Biting</td>
</tr>
<tr>
<td>Chapter 3: Turbo Codes</td>
</tr>
<tr>
<td>Chapter 4: Literature Review</td>
</tr>
<tr>
<td>Chapter 5: Decoder Quantization</td>
</tr>
<tr>
<td>Chapter 6: Algorithm to Maximize Decoder Quantization</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Page

Table 1: List of LLR values for (144, 48) DVB RCS Code ...........................................31
Table 2: List of LLR values for (56, 48) DVB RCS Code .............................................39
Table 3: List of LLR values for (2592, 864) DVB RCS Code .......................................41
LIST OF FIGURES

Figure 1: General Model of a Basic Communication System ..............................................13

Figure 2: Digital Communication System ........................................................................14

Figure 3: A Basic Trellis Diagram .................................................................................18

Figure 4: A Generic Turbo Encoder .............................................................................21

Figure 5: Turbo Decoding Process .................................................................................22

Figure 6: Constituent Encoder used by DVB-RCS ............................................................25

Figure 7: Encoding technique in DVB-RCS standard turbo code ....................................26

Figure 8: A decoder for DVB-RCS turbo code .................................................................27

Figure 9: FER vs $E_b/N_o$ (144, 48) Floating Point vs. Quantized {-7 7} Decoder Input LLR.................................................................................................................................30

Figure 10: FER vs $E_b/N_o$ (144, 48) Floating Point vs. Quantized {-14 14} Decoder Input LLR..................................................................................................................................................32

Figure 11: FER vs $E_b/N_o$ (144, 48) Floating Point vs. Quantized {-21 21} Decoder Input LLR..................................................................................................................................................32

Figure 12: FER vs $E_b/N_o$ (144, 48) Floating Point vs. Quantized as a function of SNR point ..................................................................................................................................................34

Figure 13: FER vs $E_b/N_o$ Constant log MAP (144, 48) Floating Point vs. Scaled Quantized input LLRs (scaling factor .75 without constants of .375 and 1.125 in the decoding routine) ........................................................................................................................................36

Figure 14: FER vs $E_b/N_o$ Constant log MAP (144, 48) Floating Point vs. Scaled Quantized input LLRs (scaling factor .75 with constants of .375 and 1.125 in the decoding routine) ........................................................................................................................................37

Figure 15: FER vs $E_b/N_o$ Max Log MAP (56, 48) Floating Point vs. Quantized Decoder Input LLR..................................................................................................................................................40
Figure 16: FER vs $E_b/N_0$ Max Log MAP (2592, 864) Floating Point vs. Quantized Decoder Input LLR

Figure 17: FER vs $E_b/N_0$ 48 Couples Variable rate codes Floating Point vs. Quantized Decoder Input LLR

Figure 18: FER vs $E_b/N_0$ 424 Couples Variable rate codes Floating Point vs. Quantized Decoder Input LLR
CHAPTER 1: ERROR CORRECTING CODES

A brief history of error correcting codes will be discussed in this chapter. Encoding and decoding techniques used in Turbo Error Control Coding will be introduced followed by an outline of the thesis.

1.1 History

An ideal communication system (digital or analog) is the one that can receive information sent over a channel as is. Practically speaking it is difficult to find such a system, as there would be some information lost (in most cases) during the transmission of a message over a channel due to noise or other disturbances. The task of communication engineers would then be to detect or better correct the errors that occur during the transmission across a noisy channel. This requires a solid understanding of “Error Control Coding”.

“Error-control coding can be defined as the introduction of redundancy to a message to allow for the detection or correction of errors that can occur during the transmission of that message over a noisy channel” [1]. These coding schemes are broadly divided into two categories, block codes and convolutional codes. To generate a block code, the encoder takes blocks of ‘k’ data bits and transforms these into a block of ‘n’ bits each by adding ‘n-k’ redundant bits which are generated based on a prescribed rule whereas in a convolutional encoder a continuous sequence of message bits is accepted and continuous sequences of code words are generated [2].
Turbo Codes are a class of error correcting codes that were introduced in 1993 by Berrou, Glavieux and Thitimajshima in a paper entitled “Near Shannon Limit Error Correcting Coding and Decoding” [3]. The importance of turbo codes is that they enable reliable communications with power efficiencies close to the theoretical limit predicted by Claude Shannon [4]. Two key innovations, parallel concatenated encoding and iterative decoding, make these codes really powerful.

Parallel concatenated encoding (PCE) consists of two or more component encoders. Its working can be explained with an example. Suppose there are two encoders E1 and E2. A message block 'x' is encoded by both E1 and E2 to produce code words 'c1' and 'c2' respectively. However a shuffled (interleaved) input ‘x'' is fed to the encoder E2 which makes the code words ‘c1’ and ‘c2’ different (in most cases) making the turbo codes so powerful. Then the message block ‘x’ and the code words ‘c1’ and ‘c2’ are multiplexed and sent over the channel to the receiver.

“Decoding with feedback or iterative decoding is an efficient, low-complexity means for "information handover" in any parallel or serial concatenated coding scheme and in related applications” [5]. These decoders use the Max-Log-MAP (Maximum A Posteriori) and SOVA (Soft Output Viterbi Algorithm) algorithms which are considered sub optimal as opposed to the optimal Log-MAP and Viterbi algorithms. The iterative decoders are matched to the corresponding constituent encoders. In MAP decoding, each decoder operates on the systematic input, the parity input and the extrinsic information
iteratively until they reach a convergence factor. Systematic input is the source information in its natural order, parity input is the information obtained by encoding the input data. Extrinsic information is the output of one decoder corrected by the systematic input of the other decoder. The name ‘Turbo’ is derived from this iterative decoding [1].

1.2 Outline of the Thesis

Fundamentals of Digital Communication Systems, Shannon’s theorem and Error Control codes are discussed in chapter 2. In chapter 3, a more in depth introduction to Turbo codes, the encoding and decoding processes in turbo coding are explained. The Digital Video Broadcasting – Return Channel over Satellite (DVB-RCS) codes are introduced in chapter 4, along with their encoding and decoding routines. Chapter 5 deals with an in depth analysis of the effects of decoder quantization along with some results of simulation runs. The criterion for developing an efficient algorithm to maximize the performance of quantized decoder input is presented in chapter 6, followed by the algorithm. Also, the results and analyses of performance simulations are presented.
CHAPTER 2: DIGITAL COMMUNICATION

This chapter reviews the basics of digital communication system followed by Shannon’s theorem and error control codes.

2.1 Digital Communication Systems

A Digital communication system basically consists of the following three elements.

1. Transmitter

2. Channel

3. Receiver

Below is a general model of a basic communication system

Figure 1: General Model of a Basic Communication System [1]
The signal that is generated at the source is transmitted through the channel where it is most likely corrupted by the presence of noise. So there is a good chance that the signal received by the receiver is different from the transmitted signal. This makes for an interesting communication system. An Additive White Gaussian Noise (AWGN) model is considered throughout this research.

To further elaborate on the above digital communication system, consider Figure 2 where the transmitter and receiver are broken down into different blocks.

Figure 2: Digital Communication System [2]
The source encoder optimizes the amount of information in the source data (generated by the source) to be transmitted by eliminating redundancy. The channel encoder introduces different redundancy to this data which helps in error correction and detection when the transmission is across a noisy channel. Then the modulator generates the signal suitable for transmission over the channel by mapping each information symbol to a waveform. The transmitted signal, sent over the channel gets corrupted by the presence of noise. This corrupted signal enters the demodulator and is converted back to information symbols. The final destination of the data is the sink after the channel and source decoders perform reverse operations of their respective encoders.

2.2 Shannon’s Theorem:

Shannon defined achievable channel capacity for an AWGN channel as

\[ C = B \log_2 \left(1 + \frac{S}{N}\right) \text{ bits per second} \]

Where B is the Bandwidth of the channel in Hertz, S is the average signal power and N is the average noise power.

It is this upper limit on the channel capacity that led to the advancement of error correction codes in the recent past. Extensive research has been done and still is being carried on in the field of error correcting codes to come up with codes that can achieve channel capacities closer to this theoretical limit.
2.3 Error Control Codes

Transmission of data over a noisy channel in most cases would lead to the received signal being different from the transmitted signal. In order to minimize this difference the transmitted signal is appended with additional bits (redundant bits) of data to produce a codeword that is an extended version of the source data. This type of coding will help in detecting/correcting the errors at the receiving end of the channel and is called error control coding.

There are many error control codes that have made immense progress in the field of Communications like Block Codes and Convolutional Codes. These codes are discussed below

2.3.1 Block Codes

In Block Coding, a sequence of source information bits S, is divided into blocks of fixed length, say K. Redundant bits are added to each of these blocks to make the transmitted sequence of length N bits, where N is of course greater than K. At the receiving end the decoder’s job is to restore the original message sent by decoding the symbol sequence received. The redundant bits (N-K) that have been added to the source bits help in error detection or correction. The information rate of the block codes is given by K/N.
2.3.2 Convolutional Codes

In convolutional coding, redundant bits are added differently than in block codes. A series of linear shift registers provide the redundant bits that are added to the sequence of input bits. The output symbol is then a linear combination of the current set of input bits and the stored bits in the linear shift register.

\[ \text{Output} = \text{conv}(\text{input}, g) \], where \( g \) = generator vectors

Convolutional codes are denoted by a triple (n, k, m) where n output bits are generated whenever k input bits are received; ‘m’ is called the memory order of the convolutional code which designates the number of previous k-input blocks that must be memorized in the encoder.

2.4 Trellis Codes:

Trellis Codes are a class of convolutional codes that can also encode data continuously. Like the convolutional encoder, trellis encoder can be considered as a finite state machine since the output of the encoder at any given instant of time is dependent on the inputs at that time and the current state (the memory bits) of the encoder. The code rate is still given by K/N as in block codes.

The trellis diagram is the most popular representation of the trellis encoder. An example of a trellis diagram is shown below.
Figure 3: A Basic Trellis Diagram

The trellis diagram of a convolutional encoder shown in the figure has four states (00, 01, 10, and 11), a one-bit input and a two-bit output. The states represent the bits stored in the memory devices. Transition of encoder’s state when the current input is one is represented by a solid line in the diagram, and the dotted arrow shows the change of state when the input is zero. For example, if the encoder is in the 01 state and receives an input of 1, the encoder changes its state to 10 and outputs the code symbol 2. If it receives the input of zero, it outputs the code symbol 1 and changes to 00 state.

In most cases, trellis encoding leads to a block structure as the information sequence is truncated to a certain length (L) for convenience, and this is often done to ensure ensuring the starting and ending states of a trellis are the same. This provides the
same level of error protection for the first and last information symbol [2]. Zero Tailing and Tail Biting are the two basic methods we consider that provide trellis termination.

2.4.1: Zero Tailing

Zero tailing is a method where the trellis is forced to return to the zero state by appending a certain number of zeros to the truncated information sequence. The zero state corresponds to the contents of the shift registers being zero initially. An appreciable reduction in the data rate can result as the rate is inversely proportional to the length of the information sequence \( L \), given by \( \frac{L}{L+K} \), where \( K \) is the number of zeros appended to the information sequence [2].

2.4.2: Tail Biting

In this method a certain number of tail (last) bits, say \( T \) of the information sequence are used to initialize the encoder. Tail biting does not introduce any rate loss as the output corresponding to the tail bits \( T \) is ignored. The information sequence \( S \) produces an effective output \( S^1 \) given by \( rS \). The trellis is forced to go back to the initial state by the \( T \) bits initialized.
CHAPTER 3: TURBO CODES

An introduction to turbo coding concepts is discussed in this chapter with focus on the encoding and iterative decoding techniques of these codes.

Turbo Codes

Ever since the group of developers C. Berrou, A. Glavieux, and P. Thitimajshima discovered Turbo Codes in 1993, extensive research effort has been carried out and still is to better the performance of these codes with a clever use of existing algorithms. Turbo codes have become increasingly popular in coding theory because of their ability to achieve energy efficiencies that are within a half decibel of Shannon Capacity. Turbo codes find numerous applications in the standards used by NASA for Digital Video Broadcasting (DVB-T) and deep space communications (CCSDS) [6]. The two important characteristics of Turbo codes are parallel concatenated encoding and iterative decoding.

Parallel Concatenated Encoding

The interesting and powerful fact about Turbo codes is that they are a combination of two codes that work together to achieve synergy that would not be possible with a single code. The two codes (or more) are formed from a parallel concatenation of constituent codes separated by an interleaver [6]. Figure 4 shows the parallel concatenation technique used in turbo codes.
The interleaver, a simple device that rearranges the order of the input data in an irregular but prescribed manner, is a critical device in realizing the parallel concatenation technique. The two encoders shown in the figure realize the same functionality but the inputs that these encoders act on are different. The lower decoder works on the interleaved input which makes parallel concatenation as powerful as it is. The data input, un-interleaved input and the interleaved input are serialized to form an output data block.

**Iterative Decoding**

The first decoder operates on the systematic information received from the first encoder to generate statistical information which is then fed to the second decoder. The second decoder then works on this received information along with the systematic information received from the second encoder. The second decoder then updates the
statistical information which is fed back to the first decoder. The iterative process continues until a final estimate is obtained, thereby generating the decoded output. This process is illustrated in Figure 5.

Figure 5: Turbo Decoding Process [2]
CHAPTER 4: LITERATURE REVIEW

The Digital Video Broadcasting (DVB) Project was founded in 1993 by the European Telecommunications Standards Institute (ETSI) with the goal of standardizing digital television services. Its initial standard for satellite delivery of digital television, dubbed DVB-S, used a concatenation of an outer (204,188) byte shortened Reed Solomon code and an inner constraint length 7, variable rate (r ranges from 1/2 to 7/8) convolutional code [7].

According to Valenti, Cheng and Seshadri, in their introduction to *Turbo Codes and LDPC Codes for Digital Video Broadcasting* state that “The same infrastructure used to deliver television via satellite can also be used to deliver Internet and data services to the subscriber.” Internet coverage over satellite can be very powerful and can serve even the most remote areas all around the world. An added functionality to support interactive applications by incorporating an uplink to such system makes the infrastructure even more powerful and useful. The two channels, uplink and downlink can be on different bandwidths as many internet services require a faster downlink [10].

The uplink can be provided by a telephone modem, but this has its own limitations such as modest data rates, discontinuities in service apart from being very expensive. The authors of [10] hence site a better and cheaper alternative to this: the antenna that is used to receive (downlink) data from satellite could also be used for sending (uplink) information back to the satellite. There are certain challenges that will
have to be faced in implementing this considering how small the antenna aperture is, and the requirement for a cost effective and low power amplifiers. To overcome these challenges the authors suggest the use of strong forward error correcting (FEC) codes for the satellite return channel and turbo codes provide the perfect solution for this.

“The DVB-RCS (Digital Video Broadcasting-Return Channel over Satellite) turbo codes come in twelve frame sizes ranging from 12 bytes to 216 bytes and seven code rates ranging from $r = 1/3$ to $r = 6/7$. The return link supports data rates from 144 kbps to 2 Mbps and is shared among terminals by using multi-frequency time-division multiple-access (MF-TDMA) and demand assigned multiple-access (DAMA) techniques” [10].

The turbo code architecture remains the same; the encoding is performed by a pair of constituent recursive systematic convolutional (RSC) encoders and either log-MAP or max-log-MAP decoding is used on the receiving end. The only notable difference in these codes is the use of circular recursive systematic convolutional (CRSC) encoding which is based on the concept of tail biting introduced earlier. Tail biting aids in improving the performance of decoding as the encoder’s beginning state is known.

Another distinct feature of the DVB-RCS code compared to the generic turbo codes is the use of Duo-Binary constituent encoders defined over GF (4), which enables faster decoding [10]. The effective code rate is $r = 1/2$, since the encoder works on two
input bits to generate corresponding output parity bits for each. Parallel transitions in the code trellis are avoided by the use of encoders with memory three which corresponds to a constraint length of four [10].

The paper [10] also highlights the advantages of using Duo-Binary encoders in Turbo Codes. Apart from the speed at which the decoding process is performed, the use of efficient Max-Log-MAP algorithm makes these codes very powerful as the performance loss is only about $1/10 - 2/10$ of a dB compared with the log-MAP algorithm. The impact of improper tail biting does not degrade the performance of these codes as compared to their binary counterparts.

**Encoding**

The CRSC constituent encoder used by DVB-RCS is shown in Figure 6

![CRSC Constituent Encoder used by DVB-RCS](image_url)
The encoder is fed with blocks of ‘k’ message bits which are grouped into ‘N’ = \( \frac{k}{2} \) couples where \( N \) can be any of the twelve values in \{48, 64, 212, 220, 228, 424, 432, 440, 752, 848, 856, 864\}. The number of bytes per block is \( \frac{N}{4} \). In the figure, A, B represent the first two bits of the couple and W, Y represent the parity bits. S1, S2 and S3 are the memory devices.

The encoding technique is similar to that used by generic turbo codes, except for the fact that interleaving is performed at two levels, firstly within the couples and secondly between the couples. In the first level of interleaving, every other couple is reversed if ‘k’ is even. In the second level of interleaving couples are permuted in a pseudo random fashion. More information on how this is done can be obtained in [11].

![Diagram](image)

Figure 7: Encoding technique in DVB – RCS standard turbo code
Decoding

The decoding process of the DVB-RCS code involves the exchange of intrinsic information between the two component decoders after each half-iteration. Each binary symbol in a binary turbo code is represented as a single log-likelihood ratio whereas in the duo binary code each symbol requires three log-likelihood ratios. The algorithm used to perform the decoding is the Max-Log-MAP algorithm. A generic decoder than can be used to decode DVB-RCS code is as shown Figure 8.

![Decoder Diagram](image)

**Figure 8:** A decoder for DVB-RCS turbo code [10]

The two decoders work on the input LLR’s of the message couples denoted by $\Lambda^{(i)}_{a,b}(A_k, B_k)$ along with the received parity bits generated by the corresponding encoder to produce the updated LLRs $\Lambda^{(o)}_{a,b}(A_k, B_k)$ at the output. The extension of the Max-Log-
MAP algorithm for DVB-RCS code is straightforward. Details on the method can be found in the literature [10].
CHAPTER 5: DECODER QUANTIZATION

This chapter focuses on the performance of DVB-RCS Duo Binary Turbo Codes when the decoder’s input is quantized to 4 bits. A floating point decoder is used as a baseline for performance comparison for simulation runs performed for several block sizes and rates. For the purpose of this research, only the performance of linear quantization is studied.

Several simulation runs have been performed to ascertain the factors that contribute to the performance of the quantized decoder. Initially, the decoder’s input has been quantized (4 bit) to fixed ranges ({-7 7}, {-14 14}, {-21 21}, etc) and rate (r = 1/3). As an example the performance of a 48 couple 1/3 rate decoder for linearly quantized decoder inputs {-7 to 7} in comparison with the floating point decoder input is shown in Figure 8, which shows frame error rate (FER) versus signal-to-noise ratio (Eb/N0).
Figure 9 shows that there is about $2/10$ dB difference in the performance which suggests that the initial guess of $\{-7 \ 7\}$ input quantization range is a good estimate. However, the performance worsens as the range is increased to higher values of 4 bit quantization as shown in Figures 10 and 11. The maximum and minimum input LLR values for this code are as shown in the table below. As can be seen from the table, the LLR values increase as a function of SNR.
Table 1: List of input LLR values for (144, 48) DVB RCS Code

<table>
<thead>
<tr>
<th>Eb/N₀ (dB)</th>
<th>Maximum Value of LLR</th>
<th>Minimum Value of LLR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5.2367</td>
<td>-5.1179</td>
</tr>
<tr>
<td>.2</td>
<td>5.6612</td>
<td>-5.2645</td>
</tr>
<tr>
<td>.4</td>
<td>5.9315</td>
<td>-5.5953</td>
</tr>
<tr>
<td>.6</td>
<td>6.0656</td>
<td>-5.9380</td>
</tr>
<tr>
<td>.8</td>
<td>6.1346</td>
<td>-6.1313</td>
</tr>
<tr>
<td>1</td>
<td>6.7209</td>
<td>-6.3705</td>
</tr>
<tr>
<td>1.2</td>
<td>6.9393</td>
<td>-6.4148</td>
</tr>
<tr>
<td>1.4</td>
<td>7.2596</td>
<td>-6.7225</td>
</tr>
<tr>
<td>1.6</td>
<td>8.1915</td>
<td>-6.9635</td>
</tr>
<tr>
<td>1.8</td>
<td>8.2195</td>
<td>-7.2516</td>
</tr>
<tr>
<td>2</td>
<td>8.3381</td>
<td>-7.4606</td>
</tr>
<tr>
<td>2.2</td>
<td>8.4967</td>
<td>-7.6099</td>
</tr>
<tr>
<td>2.4</td>
<td>8.8322</td>
<td>-7.8382</td>
</tr>
<tr>
<td>2.6</td>
<td>8.9446</td>
<td>-8.6836</td>
</tr>
<tr>
<td>2.8</td>
<td>9.1945</td>
<td>-9.4324</td>
</tr>
<tr>
<td>3</td>
<td>9.4982</td>
<td>-9.9008</td>
</tr>
<tr>
<td>3.2</td>
<td>9.9511</td>
<td>-10.4261</td>
</tr>
</tbody>
</table>
Figure 10: FER vs $E_b/N_o$ for (144, 48) Floating Point vs. Quantized \{-14, 14\} Decoder Input LLR

Figure 11: FER vs $E_b/N_o$ (144, 48) Floating Point vs. Quantized \{-21, 21\} Decoder Input LLR
An important observation made from the list of values is that the range, to which the decoder’s input can be quantized to, is dependent on the SNR point. The input LLR values are observed to be linearly increasing with the SNR point. A good approach to quantizing the inputs would be to restrict them to lower ranges for low SNR points and higher ranges for high SNR points. Several experiments have been performed to implement this approach.

A variable multiplier factor, which is a factor of the SNR, had been introduced in the decoding routine of the algorithm to account for the linear dependence of the input LLRs on the SNR. The input LLR values are multiplied by this multiplier factor before being sent to the decoding routine. The performances are as shown in Figure 12. These results show good performance which formed the basis for proceeding in the right direction towards achieving good performance for variable rate codes.
Figure 12: (144, 48) Floating Point vs. Quantized FER vs $E_b/N_0$ as a function of SNR

Another experiment tried was to introduce a scaling factor in the decoding routine. The input LLRs that are fed to the decoding algorithm are multiplied by the scaling factor. The performance of the Max-Log-MAP algorithm with a constant correction factor under such scaling has been studied. In Max-Log-MAP algorithm, the trellis is swept in both the forward and backward directions. The compare and select operations of the decoder outputs are estimated based on the Jacobi algorithm also called the max* operator. The max* operator is executed twice, once for the forward sweep of the trellis and a second for the reverse sweep, hence it constitutes a significant portion of
the decoder complexity. There are several operations that estimate this term, given below is one of them [13]

\[
\text{max}^*(x,y) = \max(x,y) + f_c(|y-x|), \text{ where } x, y \text{ are the input LLRs and } f_c(|y-x|) \text{ is the correction function}
\]

For the Max-Log-MAP with a constant correction factor, the max* operator is approximated by the equation below, where it has been showed in [12] that \( C = 0.5 \) and \( T = 1.5 \)

\[
\text{max}^*(x,y) \approx \max(x,y) + \begin{cases} 
0 & \text{if } |y-x| > T \\
C & \text{if } |y-x| \leq T
\end{cases}
\]

The performance of input scaling for the quantized Max log MAP algorithm with constant correction factor has been simulated. When the input was scaled by a scaling factor a significant performance difference had been observed in the FER vs \( E_b/N_0 \) curves. Several experiments have been performed to study the effects of scaling. The performance loss found can be attributed to the dependence of the correction factor on the input scaling. Since the input LLRs have been scaled, best performances for scaled inputs have been observed when the constant correction factor is multiplied by a factor of the scaling factor. For instance, consider the scaling factor to be 0.5, the constants \( C \) and \( T \) of the Max-Log-MAP decoding algorithm have been modified to be 0.5*C and 0.5*T. The figure below shows the performance of such routine with a scaling factor of .75 (with and
without changes made to the constants of the Max-Log-MAP algorithm with constant correction factor.)

Figure 13: FER vs $E_b/N_0$ Constant log MAP (144, 48) Floating Point vs. Scaled Quantized input LLRs (scaling factor .75 without constants of .375 and 1.125 in the decoding routine)
Figure 14: FER vs $E_b/N_o$ Constant log MAP (144, 48) Floating Point vs. Scaled Quantized input LLRs (scaling factor .75 with constants of .375 and 1.125 in the decoding routine)

Earlier experimental results for quantizing the decoder input range as a factor of SNR showed us that the performance can closely match that of the floating point code. As an additional example, the performance of quantized (to a range {-7 to 7}) decoder inputs of the (56, 48) DVB-RCS Duo Binary Turbo Code is compared to the floating
point (56, 48) DVB-RCS Duo Binary Turbo Code. Figure 15 shows that the quantized code performance is far from the floating point performance particularly as SNR increases. The result is explained by a close look at the input LLRs as shown in table 2, the input LLRs at low SNR are comparably higher than those observed with the lower rate code. So, the range to which the LLRs were restricted to turned out to be a bad estimate as most of the LLR’s were higher than 7 or less than -7. The approach followed earlier to vary the input LLR range as a function of SNR has been observed to give good results, but the best performance was obtained when the code rate is also taken into account. The quantization range had been scaled by a factor of rate times SNR. So, it can be stated that the quantization range of {-7 to 7} alone does not give good performance results for all codes, and code rate is one of the factors that the range is dependent on.
Table 2: List of LLR values for (56, 48) DVB RCS Code

<table>
<thead>
<tr>
<th>$E_b/N_o$ (dB)</th>
<th>Maximum Value of LLR</th>
<th>Minimum value of LLR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8.4624</td>
<td>-8.0173</td>
</tr>
<tr>
<td>.2</td>
<td>9.6508</td>
<td>-8.2637</td>
</tr>
<tr>
<td>.4</td>
<td>10.1341</td>
<td>-9.9141</td>
</tr>
<tr>
<td>.6</td>
<td>10.5832</td>
<td>-10.5476</td>
</tr>
<tr>
<td>.8</td>
<td>11.4035</td>
<td>-11.3302</td>
</tr>
<tr>
<td>1</td>
<td>12.0796</td>
<td>-11.5248</td>
</tr>
<tr>
<td>1.2</td>
<td>12.5540</td>
<td>-11.6109</td>
</tr>
<tr>
<td>1.4</td>
<td>13.2094</td>
<td>-12.0319</td>
</tr>
<tr>
<td>1.6</td>
<td>13.4157</td>
<td>-12.3251</td>
</tr>
<tr>
<td>1.8</td>
<td>13.7496</td>
<td>-12.7578</td>
</tr>
<tr>
<td>2</td>
<td>13.7926</td>
<td>-13.0981</td>
</tr>
<tr>
<td>2.2</td>
<td>12.5921</td>
<td>-13.2856</td>
</tr>
<tr>
<td>2.4</td>
<td>12.6580</td>
<td>-14.5631</td>
</tr>
<tr>
<td>2.6</td>
<td>13.9182</td>
<td>-16.2689</td>
</tr>
<tr>
<td>2.8</td>
<td>14.4175</td>
<td>-16.8983</td>
</tr>
<tr>
<td>3</td>
<td>14.8758</td>
<td>-17.0241</td>
</tr>
<tr>
<td>3.2</td>
<td>15.6362</td>
<td>-17.8188</td>
</tr>
<tr>
<td>3.4</td>
<td>15.8788</td>
<td>-18.0149</td>
</tr>
<tr>
<td>3.6</td>
<td>16.1631</td>
<td>-18.3221</td>
</tr>
<tr>
<td>3.8</td>
<td>16.5192</td>
<td>-19.2316</td>
</tr>
<tr>
<td>4</td>
<td>17.3470</td>
<td>-20.0886</td>
</tr>
<tr>
<td>4.2</td>
<td>19.1282</td>
<td>-20.9135</td>
</tr>
<tr>
<td>4.4</td>
<td>19.8076</td>
<td>-20.8467</td>
</tr>
<tr>
<td>4.6</td>
<td>21.4751</td>
<td>-21.6886</td>
</tr>
<tr>
<td>4.8</td>
<td>22.8135</td>
<td>-21.6476</td>
</tr>
<tr>
<td>5</td>
<td>23.8681</td>
<td>-23.8772</td>
</tr>
<tr>
<td>5.2</td>
<td>23.8453</td>
<td>-23.9442</td>
</tr>
<tr>
<td>5.4</td>
<td>24.3078</td>
<td>-23.9770</td>
</tr>
<tr>
<td>5.6</td>
<td>24.3987</td>
<td>-24.8265</td>
</tr>
<tr>
<td>5.8</td>
<td>24.7163</td>
<td>-25.5341</td>
</tr>
<tr>
<td>6</td>
<td>25.7513</td>
<td>-26.0986</td>
</tr>
</tbody>
</table>
To further study the factors that will have to considered, the performance of large block codes needs to be analyzed. The performance of a (2592, 864) DVB-RCS Duo Binary Turbo Code is compared with that of the floating point code. Figure 17 shows the comparison in the FER versus Signal-To-Noise-Ratio of the two codes. There is about 3/10 dB difference in the performance of the quantized code which is relatively better than the higher rate code discussed earlier. Below is the table that shows the range of LLR’s for this large (2592, 864) block code. The estimate of the range of LLR values of {-7 to 7} turned out to be a reasonable estimate in this case, but there is still a significant
loss in the performance as compared to the floating point code. Table 3 shows that the
LLRs compare very closely with the LLRs for small block codes and hence the
performance that is within 1/10dB of the small block codes. A performance that closely
resembles the floating point code can be obtained by considering the approach followed
earlier. It can be observed that the quantization range is strongly affected by the code
rate, but not as strongly as block size, yet the block size is still a factor that needs to be
considered in designing the quantization range.

Table 3: List of LLR values for (2592, 864) DVB RCS Code

<table>
<thead>
<tr>
<th>(\frac{E_b}{N_0}) (dB)</th>
<th>Maximum Value of LLR</th>
<th>Minimum Value of LLR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7.4994</td>
<td>-7.6273</td>
</tr>
<tr>
<td>.2</td>
<td>8.3993</td>
<td>-7.8242</td>
</tr>
<tr>
<td>.4</td>
<td>8.6687</td>
<td>-8.3816</td>
</tr>
<tr>
<td>.6</td>
<td>8.8755</td>
<td>-9.0058</td>
</tr>
<tr>
<td>.8</td>
<td>9.1651</td>
<td>-9.1226</td>
</tr>
<tr>
<td>1</td>
<td>9.7469</td>
<td>-9.6396</td>
</tr>
<tr>
<td>1.2</td>
<td>9.8002</td>
<td>-9.6691</td>
</tr>
</tbody>
</table>
To summarize, in order to maximize the performance of quantized input decoder DVB-RCS turbo codes certain factors need to be accounted for, of which code rate and block size are the two most important ones. A criterion also needs to be developed to determine an efficient technique to achieve good performance that can be obtained for any block code and code rate.
CHAPTER 6: ALGORITHM TO MAXIMIZE DECODER QUANTIZATION

Earlier experiments conducted have been very useful in analyzing the performance of decoder quantized turbo codes. It has been noticed that a fixed input decoder range does not give good performance for variable rate codes. For a given code rate, a smaller decoder input range affects the performance of the code, whereas a higher decoder input range affects the resolution. So, there is a tradeoff between resolution and performance that needs attention. In order to account for this an algorithm has been put forth that will be explained in this chapter.

In the previous chapter the performance of the DVB-RCS Duo Binary turbo codes for floating point decoder inputs versus quantized decoder inputs has been studied. Apart from the block size and code rates, there are other important factors that affect the performance of quantized codes. Below are the three most important factors to be considered for optimal performance.

1. Block Size
2. Code Rate
3. Signal-To-Noise-Ratio Point

Apart from these three, the decoding algorithm is also a factor that will affect the performance. In this research, only the MAP-Decoding Algorithm is considered. Several research papers [7] [8] [9] and texts have been published that describe the MAP Algorithm for Turbo Codes.
6.1 **Criteria for estimating the Decoder Input Range**

Several Experiments have been conducted during the current research to develop a criterion that needs to be followed to obtain a good estimate of the quantization range for decoder inputs.

The decoder input LLR values vary as the code rate and block size varies as observed earlier. It is important to note that the values change as a factor of Signal-To-Noise-Ratio (SNR). So, to account for the performance of Quantized decoder input codes for these parameter variations it is important to have a quantized range that changes as a factor of the SNR. Below is an efficient algorithm that accounts for all the parameter variations.

**Algorithm**

1. For Every SNR value, find the maximum and minimum LLR Values and round them to the nearest integers accounting for the code rate and block sizes.
2. From the maximum and minimum values estimate which of these values is larger i.e. find Max (maximum LLR, (-minimum LLR)). Say it is MAX_TERM
3. Estimate 15 linearly spaced values from MAX_TERM to –MAX_TERM. This will be the final range of values that the input LLR’s will be restricted to.
4. Quantize the floating point LLR values to the final estimated input LLR values from step 3.
5. Use the Quantized values as the new LLR inputs to the Turbo Decoder.
This is a simple algorithm that accounts for all the factors that need to be considered and that might impact the performance of the quantized version of the code. It yields a very good estimate of the range that the input LLR’s can be quantized to.

Simulation Results

The performance of 48 block variable rate codes were compared to the quantized version of the codes. The range to which the inputs were restricted to was obtained from the algorithm presented previously. From Figure 17 it can be said that there is very little performance difference between the floating point and quantized versions of the code. The performance of the quantized code is within 1/10dB to that of the floating point code.
The performance of a larger block size code was also compared. Figure 18 shows comparison curves for variable rate codes of a 424 block DVB-RCS turbo code. Even in this case there is very little performance difference between the two versions of the code, and the quantized decoder input codes perform to within a 1/10dB to that of the floating point codes. The performance of the largest DVB RCS Turbo code for quantized inputs
was also observed to be very close to floating point input performance. The algorithm hence produces good quantization results for all rate variable size codes.

Figure 18: FER vs $E_b/N_0$ 424 Couples Variable rate codes Floating Point vs. Quantized

Decoder Input LLR
CONCLUSIONS AND FUTURE WORK

In this thesis, an extensive study of the effects of decoder quantization on the performance of DVB-RCS duo binary turbo codes has been performed and an efficient algorithm to maximize the performance of such codes has been presented. The algorithm focused on the technique to linearly quantize the decoder input. Performance simulations for varying code rates and block sizes for such codes have been studied and presented. Results show that the algorithm presented essentially matches the performance of the floating point decoder input codes.

Important lessons learnt about decoder quantization from this research are as follows

1. The decoder quantization is dependent on the SNR, at low SNR the input log likelihood ratios were observed to be lower and at high SNR the ratios were observed to be higher.

2. Changes needed to the Max Log MAP decoding algorithm to account for input scaling has been presented.

3. Performance very similar to the floating point codes (within 0.1 dB) can be obtained by properly quantizing the input to the decoder.

The algorithm presented in this research can be extended to various other iterative decoding algorithms like Max-Log-MAP algorithm with a constant factor from a small look-up table or constants obtained from interpolation. The extension of this algorithm to these decoding iterative routines is straight forward.
Considering the flexible nature of the algorithm, the hardware implementation of such codes becomes relatively easy. The size, cost and the material involved in designing the decoder hardware would be drastically reduced by hardwiring the quantized inputs as the decoding architecture would have to work on only 4 bits of information, which is considered the standard implementation as compared to the floating point inputs. Also, the adaptive nature of the algorithm to various code rates and block sizes makes the decoding structure very flexible.

An additional improvement in the performance of these codes can be obtained by using a non-linear quantization technique. Future work could involve developing algorithms that can improve the performance even further by quantizing the decoder input to non-linear values.
REFERENCES


APPENDIX A: MATLAB ROUTINES

The following code has been obtained from the Iterative Solutions Coded Modulation Library (CML). It is free software that can be modified under the terms of the GNU Lesser General Public License as published by the Free Software Foundation. The files below have been modified to obtain results pertaining to this research. More information can be found at

http://www.iterativesolutions.com/Matlab.htm

1. The following code provides the list of Scenarios to be executed for a particular run.

```matlab
% File DVBRCSScenarios
%
% Raghu Gorthy, This program has been modified to simulate for % different scenarios. This is the scenarios file for which % the simulation runs. The code has been modified to only consider BPSK % modulation and AWGN noise model %
%
% The basic code has been provided by the Coded Modulation Library (CML)
%
% determine where your root directory is
load( 'CmlHome.mat' );

% determine where to store the output files
base_name = 'DVBRCS';
if ispc
    data_directory = strcat( '\output\', base_name, '\' );
else
    data_directory = strcat( '/output/', base_name, '/' );
end

full_directory = strcat( cml_home, data_directory );
if ~exist( full_directory, 'dir' )
    mkdir( full_directory);
end

MINBER = 1e-5;
num_errors = 100;

% A range of Block Codes can be chosen for simulation runs
% couples = [ 48, 64, 212 220, 228, 424, 432, 440, 848, 856, 864, 752 ];
couples = 864;

% A range of Rates can be chosen
% rates = [1/3, 2/5, 1/2, 2/3, 3/4, 4/5, 6/7 ];
```
rates = [1/3];

% The simulation can be performed for various Decdoing Algorithms
% The current research is done for MAP-Algorithms which is
% decoder 1 as per the documentation provided by CML
% decoders = [0 1 2 3 4];
decoders = 1;
colors = ['k' 'r' 'b' 'm' 'c' 'g' 'y'];

% This program creates records according to a loop.
% the record number depends on the size (number of couples) and rate
% of the code, as well as the type of decoder.
record = 0;
for decoder_index = 1:length(decoders)
    for couple_index = 1:length(couples)
        for rate_index = 1:length(rates)
            % increment the record number
            record = record + 1;

            % All cases use BPSK over AWGN in 0.2 dB increments
            sim_param(record).modulation = 'BPSK';
            sim_param(record).mapping = 'gray';
            sim_param(record).mod_order = 2;
            sim_param(record).channel = 'AWGN';
            sim_param(record).SNR = [0:0.2:10];

            % particular to this rate and number of couples
            sim_param(record).framesize = 2*couples(couple_index);
            sim_param(record).code_bits_per_frame = ceil(couples(couple_index)/rates(rate_index)) *2;

            % filename
            sim_param(record).filename = strcat( data_directory,
                sim_param(record).modulation, ...
                int2str( sim_param(record).code_bits_per_frame),
                'comma_864couples_rate13_quantized-7to7', int2str( sim_param(record).framesize), ...
                sim_param(record).channel, '.mat');

% select the decoder type
    sim_param(record).decoder_type = decoders( decoder_index );
    sim_param(record).linetype = colors(rate_index);
    if (decoders( decoder_index ) == 0)
        sim_param(record).filename = strcat( sim_param(record).filename, 'LogMap');
    sim_param(record).linetype = strcat( sim_param(record).linetype, '-' );
    elseif (decoders( decoder_index ) == 1)
        sim_param(record).filename = strcat( sim_param(record).filename, 'MaxLogMap');
    else
        sim_param(record).filename = sim_param(record).filename;
    end
}
sim_param(record).linetype = strcat(sim_param(record).linetype, '-+' );
else
  sim_param(record).filename = strcat(sim_param(record).filename, 'OtherLogMap');
  sim_param(record).linetype = strcat(sim_param(record).linetype, '-v' );
end
sim_param(record).filename = strcat(sim_param(record).filename, '.mat');
sim_param(record).comment = sprintf( '(%d,%d) %s %s',
  sim_param(record).code_bits_per_frame/2, sim_param(record).framesize/2, ...  
  sim_param(record).modulation, sim_param(record).channel );

% Same for all
sim_param(record).legend = sim_param(record).comment;
sim_param(record).sim_type = 'coded';
sim_param(record).code_configuration = 6; % DVB-RCS
sim_param(record).SNR_Type = 'Eb/No in dB';
sim_param(record).bicm = 0;
sim_param(record).demod_type = 0; % linear-log-MAP demapper
sim_param(record).max_iterations = 10;
sim_param(record).reset = 0;
sim_param(record).max_trials = le10*ones(size(sim_param(record).SNR) );
sim_param(record).minBER = MINBER;
sim_param(record).max_frame_errors = num_errors*ones(size(sim_param(record).SNR) );
sim_param(record).plot_iterations = sim_param(record).max_iterations;
sim_param(record).save_rate = floor(le6/sim_param(record).framesize); % save every million data bits
end
end

% End Program

2. The routine used to run a single coded/un-coded modulation simulation scenario

function sim_state = SimulateMod( sim_param, sim_state, code_param )
% Raghu Gorthy - The Code has been modified to quantize the symbol_likelihood % values to 15 linearly spaced values based on the SNR point. The code
% accounts for variable rates and block sizes
% The calling syntax is:
%   sim_state = SimulateMod( sim_param, sim_state )
% sim_param = A structure containing simulation parameters.
```matlab
% sim_state = A structure containing the simulation state.
% code_param = A structure containing derived information.
% Note: See readme.txt for a description of the structure formats.

% create a random channel (BICM) interleaver
if (code_param.coded)
    if ( sim_param.bicm > 0 )
        code_param.bicm_interleaver = randperm(code_param.code_bits_per_frame)-1;
    end
end

% determine Es/No
if ( sim_param.SNR_type(2) == 'b' ) % Eb/No
    EbNo = 10.^(sim_param.SNR/10);
    EsNo = EbNo*code_param.rate;
else % Es/No
    EsNo = 10.^(sim_param.SNR/10);
end

% temporary filename
tempfile = 'tempsave.mat';

% FOR PROFILING RUNTIME
t0 = clock;

% simulate
for snrpoint = 1:length(EsNo)
    fprintf( strcat( '
', sim_param.SNR_type, ' = %f dB
'),
        sim_param.SNR(snrpoint) );
    current_time = fix(clock);
    fprintf( 'Clock %2d:%2d:%2d
', current_time(4), current_time(5),
        current_time(6) );

    % loop until either there are enough trials or enough errors
    while ( ( sim_state.trials( code_param.max_iterations, snrpoint ) <
        sim_param.max_trials( snrpoint ) ) &
        ( sim_state.frame_errors(code_param.max_iterations, snrpoint) <
        sim_param.max_frame_errors(snrpoint) ) )

        % increment the trials counter
        sim_state.trials(1:code_param.max_iterations, snrpoint) =
        sim_state.trials(1:code_param.max_iterations, snrpoint) + 1;

        % generate random data
        data = round( rand( 1, code_param.data_bits_per_frame ) );

        % code and modulate
        s = CmlEncode( data, sim_param, code_param );

        % Put through the channel
```
symbol_likelihood = CmlChannel( s, sim_param, code_param, EsNo(snrpoint));

% Raghu Gorthy Determine the Quantization Range and restrict the input LLR's to this range %
symbol_likelihood = 1*symbol_likelihood;
max_likelihood = round(max(symbol_likelihood));
min_likelihood = round(min(symbol_likelihood));

if (max_likelihood < (-min_likelihood))
    max_likelihood = -min_likelihood;
end

partition = sim_param.SNR(snrpoint)*(-max_likelihood:((2*max_likelihood)/13):max_likelihood);

if sim_param.SNR(snrpoint) > 0
    partition = (1/sqrt(sim_param.SNR(snrpoint)))*(-max_likelihood:((2*max_likelihood)/13):max_likelihood);
    codebook = (1/sqrt(sim_param.SNR(snrpoint)))*(-max_likelihood:((2*max_likelihood)/14):max_likelihood);
end

[index,symbol likelihood] = quantiz(symbol_likelihood,partition,codebook);

% Raghu Gorthy Modification End %

if (code_param.outage == 0)
    % Decode
    [detected_data, errors] = CmlDecode( symbol_likelihood, data, sim_param, code_param );

    % Echo an x if there was an error
    if ( errors( code_param.max_iterations ) );
        fprintf( 'x' );
    end

    % update frame error and bit error counters
    sim_state.bit_errors( 1:code_param.max_iterations, snrpoint ) = sim_state.bit_errors( 1:code_param.max_iterations, snrpoint ) + errors;
    sim_state.frame_errors( 1:code_param.max_iterations, snrpoint ) = sim_state.frame_errors( 1:code_param.max_iterations, snrpoint ) + (errors>0);
    sim_state.BER(1:code_param.max_iterations, snrpoint) = sim_state.bit_errors(1:code_param.max_iterations,}
57

```matlab
% if uncoded, update symbol error rate, too.
if ~code_param.coded
    if ( sim_param.mod_order > 2 )
        error_positions = xor(detected_data(1:code_param.data_bits_per_frame), data);
    
        % update symbol, frame, and bit error counters
        sim_state.symbol_errors(snrpoint) = sim_state.symbol_errors(snrpoint) + sum( max( reshape(error_positions, code_param.bits_per_symbol, code_param.symbols_per_frame ),[],1 ) );
        sim_state.SER(snrpoint) = sim_state.symbol_errors(snrpoint)/sim_state.trials(snrpoint)/code_param.symbols_per_frame;
    else
        sim_state.symbol_errors(snrpoint) = sim_state.bit_errors(snrpoint);
        sim_state.SER(snrpoint) = sim_state.BER(snrpoint);
    end
else
    % determine capacity
    if ( sim_param.bicm )
        % BICM capacity
        if (code_param.bpsk)
            bit_likelihood = symbol_likelihood; % later this should be moved to Somap function
        else
            bit_likelihood = Somap( symbol_likelihood, sim_param.demod_type );
        end
        % BICM capacity (added log2(mod_order) on 12/23/07)
        cap = log2(sim_param.mod_order)*Capacity(bit_likelihood, data );
    else
        % CM capacity (added log2(mod_order) on 12/23/07)
        cap = log2(sim_param.mod_order)*Capacity(symbol_likelihood, data );
    end
    % compare to threshold and update FER counter
    if ( cap < code_param.rate )
        sim_state.frame_errors( 1, snrpoint ) = sim_state.frame_errors( 1, snrpoint ) + 1;
        sim_state.FER(1, snrpoint) = sim_state.frame_errors(1, snrpoint)/sim_state.trials(1, snrpoint);
        % Echo an x if there was an error
        fprintf( 'x' );
    end
```
end

% determine if it is time to save (either (1) last error, (2) last frame, or (3) once per save_rate)
condition1 = (sim_state.frame_errors(code_param.max_iterations, snrpoint) == sim_param.max_frame_errors(snrpoint));
condition2 = (sim_state.trials(code_param.max_iterations, snrpoint) == sim_param.max_trials(snrpoint));
condition3 = ~mod(sim_state.trials(code_param.max_iterations, snrpoint),sim_param.save_rate);
if (condition1|condition2|condition3)
% FOR PROFILING RUNTIME
% fprintf( '%f
', etime(clock,t0) );
% t0=clock;

fprintf('.');
save_state = sim_state;
save_param = sim_param;

% Aded on April 22, 2006 in case system crashes during save
save( tempfile, code_param.save_flag, 'save_state', 'save_param');

% Store into local directory (if running locally)
if (sim_param.compiled_mode)
    copyfile( tempfile, sim_param.filename, 'f' );
end

movefile( tempfile, code_param.filename, 'f');

% redraw the BICM interleaver (so that it is uniform)
if (code_param.coded)
    if (sim_param.bicm > 0)
        code_param.bicm_interleaver = randperm(code_param.code_bits_per_frame)-1;
    end
end

end

% halt if BER or FER is low enough
if (~code_param.outage & (sim_state.BER(code_param.max_iterations, snrpoint) < sim_param.minBER ) )
    % adjust max_iterations to be the last iteration that has not yet dropped below the BER threshold
    % Logic has changed on 7-28-06
    iteration_index = max( find( sim_state.BER(sim_param.plot_iterations,snrpoint) >= sim_param.minBER ) );
end
if isempty(iteration_index)
    break;
else
    code_param.max_iterations = sim_param.plot_iterations(iteration_index);
    fprintf( 'Number of iterations = %d\n', code_param.max_iterations);
end

elseif ( code_param.outage & (sim_state.FER(code_param.max_iterations, snrpoint) < sim_param.minFER))
    % break when the FER is low enough (changed on 12-31-07)
elseif ( sim_state.FER(code_param.max_iterations, snrpoint) < sim_param.minFER^)
    break;
end
end

fprintf( 'Simulation Complete\n' );
current_time = fix(clock);
fprintf( 'Clock %2d:%2d:%2d\n', current_time(4), current_time(5), current_time(6) );

3. Routine to Plot Simulation Results

% Raghu Gorthy - The plotting routine has been modified to plot various
% scenarios on a single plot
% CmlPlot plots simulation results
% function [sim_param, sim_state] = CmlPlot( varargin )
% The calling syntax is:
% [sim_param, sim_state] = CmlPlot( scenario_filename, cases )
% Outputs:
% sim_param = A structure containing simulation parameters.
% sim_state = A structure containing the simulation state.
% Required inputs:
% scenario_filename = the name of the file containing an array of
% sim_param structures.
% cases = a list of the array indices to plot.
% Note: Multiple scenario files can be specified. In this case, the argument list
% should contain each scenario file to be used followed by the list of array indices
to read from that file.
% setup structures are retrieve data
% give an extra argument to force sim_param.reset = 0
[sim_param, sim_state] = ReadScenario( varargin{:}, [] );
number_cases = length( sim_param );

% determine the simulation types
sim_types = zeros( 8, number_cases );
for ( case_number=1:number_cases )
    if ( strcmp( sim_param(case_number).sim_type, 'capacity' ) )
        sim_types(1,case_number) = 1; % capacity simulation
    elseif ( strcmp( sim_param(case_number).sim_type, 'exit' ) )
        sim_types(2,case_number) = 1; % EXIT
    elseif ( strcmp( sim_param(case_number).sim_type, 'uncoded' ) )
        sim_types(3,case_number) = 1; % uncoded modulation
    elseif ( strcmp( sim_param(case_number).sim_type, 'coded' ) )
        sim_types(4,case_number) = 1; % coded modulation
    elseif ( strcmpi( sim_param(case_number).sim_type, 'outage' ) )
        sim_types(5,case_number) = 1; % outage probability
    elseif ( strcmp( sim_param(case_number).sim_type, 'throughput' ) )
        sim_types(6,case_number) = 1; % throughput of hybrid-ARQ
    elseif ( strcmp( sim_param(case_number).sim_type, 'bwcapacity' ) )
        sim_types(7,case_number) = 1; % capacity of FSK under BW constraint
    elseif ( strcmp( sim_param(case_number).sim_type, 'minSNRvsB' ) )
        sim_types(8,case_number) = 1; % Min SNR as a function of B
    end
end

fig_number = 0;

% first plot capacity vs. Eb/No and Es/No, if there are any capacity curves requested
if ( sum( sim_types(1,:) ) )
    fig_number = fig_number + 1;
    figure( fig_number );
    for (i=find( sim_types(1,:) == 1 ) )
        EsNo = 10.^(sim_param(i).SNR/10); % assume SNR is Es/No in dB
        EbNo = EsNo./(sim_state(i).capacity_avg*log2(sim_param(i).mod_order));
        EbNoDB = 10*log10( EbNo );
        plot( EbNoDB,
             log2(sim_param(i).mod_order)*sim_state(i).capacity_avg,
             sim_param(i).linetype );
        hold on;
    end
end

% compute unconstrained (Gaussian input) capacity
EsNoDB = sim_param(i).SNR;
EsNo = 10.^(EsNoDB/10);
cap_unconstrained = log2(1+EsNo);
EbNo = EsNo./cap_unconstrained;
EbNodB = 10*log10( EbNo );
legend( sim_param( find( sim_types(1,:) == 1 ) ).legend );

% uncomment if you want to show unconstrained
% plot( EbNodB, cap_unconstrained, '-' );
% legend( sim_param( find( sim_types(1,:) == 1 ) ).legend, 'Unconstrained', 2 );
xlabel( 'Eb/No in dB' );
ylabel( 'Capacity' );
hold off;

% Eb/No vs. Capacity (useful for FSK modulation)
fig_number = fig_number + 1;
figure( fig_number );

for (i=find( sim_types(1,:) == 1 ) )
    EsNo = 10.^(sim_param(i).SNR/10); % assume SNR is Es/No in dB
    EbNo = EsNo./(sim_state(i).capacity_avg*log2(sim_param(i).mod_order));
    EbNodB = 10*log10( EbNo );
    plot( sim_state(i).capacity_avg, EbNodB, sim_param(i).linetype );
    hold on;
end
ylabel( 'Eb/No in dB' );
xlabel( 'Rate' );
hold off;

% plot capacity vs. Es/No
fig_number = fig_number + 1;
figure( fig_number );
for (i=find( sim_types(1,:) == 1 ) )
    plot( sim_param(i).SNR, log2(sim_param(i).mod_order)*sim_state(i).capacity_avg, sim_param(i).linetype );
    hold on;
end
legend( sim_param( find( sim_types(1,:) == 1 ) ).legend );

% uncomment if you want to show unconstrained
% plot( EsNodB, cap_unconstrained, '-' );
% legend( sim_param( find( sim_types(1,:) == 1 ) ).legend, 'Unconstrained', 2 );
xlabel( 'Es/No in dB' );
ylabel( 'Capacity' );
hold off;
end
% Raghu Gorthy Modification Start %

% next plot BER vs. Eb/No if this is a coded or uncoded simulation
% if ( sum( sum( sim_types(3:4,:)) ) )
%     % plot BER vs. Eb/No
%     fig_number = fig_number + 1;
%     figure( fig_number );
%     % BER of uncoded modulation
%     for (i=find( sim_types(3,:) == 1 ) )
%         % Can only plot against Eb/No (add logic later)
%         if ( sim_param(i).SNR_type(2) ~= 'b' )
%             error( 'Uncoded modulation results must use SNR_type of
%                     Eb/No in dB' );
%         end
%         figure( fig_number );
%         semilogy( sim_param(i).SNR, sim_state(i).BER,
%                   sim_param(i).linetype );
%         hold on;
%     end
%     % BER of coded modulation
%     for (i=find( sim_types(4,:) == 1 ) )
%         % Convert to Eb/No (dB) if stored as Es/No (dB)
%         if ( sim_param(i).SNR_type(2) ~= 'b' )
%             % This is Es/No
%             EsNodB = sim_param(i).SNR;
%             % Convert to Eb/No
%             EsNo = 10.^((EsNodB/10);
%             EbNo = EsNo./sim_param(i).rate;
%             EbNodB = 10*log10(EbNo);
%         else
%             EbNodB = sim_param(i).SNR;
%         end
%         % only plot the last iteration
%         if ( length( sim_param(i).max_iterations ) )
%             max_iter = sim_param(i).max_iterations;
%         else
%             max_iter = 1;
%         end
%         semilogy( EbNodB, sim_state(i).BER( max_iter, : ),
%                   sim_param(i).linetype );
%         hold on;
%     end
%     legend( sim_param( find( sim_types(3,:) == 1 ) ).legend,
%             sim_param( find( sim_types(4,:) == 1 ) ).legend, 0 );
%     xlabel( 'Eb/No in dB' );
%     ylabel( 'BER' );
%     for (i=find( sim_types(4,:) == 1 ) )
if ( length( sim_param(i).plot_iterations ) )
    % Convert to Eb/No (dB) if stored as Es/No (dB)
    if ( sim_param(i).SNR_type(2) ~= 'b' )
        % This is Es/No
        EsNodB = sim_param(i).SNR;
        % Convert to Eb/No
        EsNo = 10.^(EsNodB/10);
        EbNo = EsNo./sim_param(i).rate;
        EbNodB = 10*log10(EbNo);
    else
        EbNodB = sim_param(i).SNR;
    end
else
    % This is already in Es/No
    EsNodB = sim_param(i).SNR;
end
% plot the other iterations
semilogy( EbNodB,
    sim_state(i).BER(sim_param(i).plot_iterations,:), sim_param(i).linetype );
end

hold off;

% Now plot against Es/No, if uncoded
if sum( sim_types(4,:) )
    fig_number = fig_number + 1;
    figure( fig_number );
    for (i=find( sim_types(4,:) == 1 ) )
        % see if Es/No is defined, otherwise derive
        if ( sim_param(i).SNR_type(2) == 'b' )
            % This is Eb/No
            EbNodB = sim_param(i).SNR;
            % Convert to Es/No
            EbNo = 10.^((EbNodB/10);
            EsNo = sim_param(i).rate*EbNo;
            EsNodB = 10*log10(EsNo);
        else
            % This is already in Es/No
            EsNodB = sim_param(i).SNR;
        end

        % only plot the last iteration
        if ( length( sim_param(i).max_iterations ) )
            max_iter = sim_param(i).max_iterations;
        else
            max_iter = 1;
        end

        % Plot FER versus Es/No in dB
        figure( fig_number );
        semilogy( EsNodB, sim_state(i).BER( max_iter, : ), sim_param(i).linetype );
        hold on;
    end
end
figure( fig_number );
legend( sim_param( find( sim_types(4,:) == 1 ) ).legend, 0 );
xlabel( 'Es/No in dB' );
ylabel( 'BER' );

for (i=find( sim_types(4,:) == 1 ) )
  if ( length( sim_param(i).plot_iterations ) )
    % see if Es/No is defined, otherwise derive
    if ( sim_param(i).SNR_type(2) == 'b' )
      % This is Eb/No
      EbNodB = sim_param(i).SNR;
      % Convert to Es/No
      EbNo = 10.^((EbNodB)/10);
      EsNo = sim_param(i).rate*EbNo;
      EsNodB = 10*log10(EsNo);
    else
      % This is Es/No
      EsNodB = sim_param(i).SNR;
      % Convert to Eb/No
      EbNo = 10.^((EsNodB)/10);
      EbNo = EsNo./sim_param(i).rate;
      EbNodB = 10*log10(EbNo);
    end
    % plot the other iterations
    figure( fig_number );
    semilogy( EsNodB,
              sim_state(i).BER(sim_param(i).plot_iterations,:), sim_param(i).linetype);
    end
  end
end

figure( fig_number );
hold off;
end

% Raghu Gorthy Modification End %

% Plot the SER if uncoded
if ( sum( sum( sim_types(3,:) ) ) )
  % plot SER vs. Eb/No
  fig_number = fig_number + 1;
  figure( fig_number );

  % SER of uncoded modulation
  for (i=find( sim_types(3,:) == 1 ) )
    % Convert to Eb/No (dB) if stored as Es/No (dB)
    if ( sim_param(i).SNR_type(2) == 'b' )
      error( 'The SNR should be stored as Eb/No' );
    end
end
semilogy( sim_param(i).SNR, sim_state(i).SER, sim_param(i).linetype );
    hold on;
end

legend( sim_param( find( sim_types(3,:) == 1 ) ).legend, 0 );
xlabel( 'Eb/No in dB' );
ylabel( 'SER' );

    hold off;
end

% Plot the FER of coded and outage simulations
if ( sum( sum( sim_types(4:5,:) ) ) )

    % First plot FER vs. Eb/No
    fig_number = fig_number + 1;
    figure( fig_number );
    hold on
    % Outage Probability
    for (i=find( sim_types(5,:) == 1 ) )
        % If stored as Es/No, convert to Eb/No
        if ( sim_param(i).SNR_type(2) == 'b' )
            % This is Eb/No
            EbNodB = sim_param(i).SNR;
        else
            % This is stored as Es/No
            EsNodB = sim_param(i).SNR;
            % Convert to Eb/No
            EsNo = 10.^(EsNodB/10);
            EbNo = EsNo./sim_param(i).rate;
            EbNodB = 10*log10(EbNo);
        end

        % Plot FER versus Eb/No in dB
        semilogy( EbNodB, sim_state(i).FER, sim_param(i).linetype );
        hold on;
    end

    % FER of coded modulation
    for (i=find( sim_types(4,:) == 1 ) )
        % If stored as Es/No, convert to Eb/No
        if ( sim_param(i).SNR_type(2) == 'b' )
            % This is Eb/No
            EbNodB = sim_param(i).SNR;
        else
            % This is stored as Es/No
            EsNodB = sim_param(i).SNR;
            % Convert to Eb/No
            EsNo = 10.^(EsNodB/10);
            EbNo = EsNo./sim_param(i).rate;
            EbNodB = 10*log10(EbNo);
% only plot the last iteration
if ( length( sim_param(i).max_iterations ) )
    max_iter = sim_param(i).max_iterations;
else
    max_iter = 1;
end

% Plot FER versus Eb/No in dB
%testing figure( fig_number );
semilogy( EbNodB, sim_state(i).FER( max_iter, : ),
    sim_param(i).linetype );
hold on;
end
%testing figure( fig_number );
legend( sim_param( find( sim_types(5,:) == 1 ) ).legend, sim_param( find( sim_types(4,:) == 1 ) ).legend, 0 );
xlabel( 'Eb/No in dB' );
ylabel( 'FER' );

% Now plot the other iterations
for (i=find( sim_types(4,:) == 1 ) )
    if ( length( sim_param(i).plot_iterations ) )
        % make sure that we get both Es/No and Eb/No
        if ( sim_param(i).SNR_type(2) == 'b' )
            % This is Eb/No
            EbNodB = sim_param(i).SNR;
        else
            % This is stored as Es/No
            EsNodB = sim_param(i).SNR;
            % Convert to Eb/No
            EsNo = 10.^((EsNodB/10);
            EbNo = EsNo./sim_param(i).rate;
            EbNodB = 10*log10(EbNo);
        end

        % plot the other iterations
        %testing figure( fig_number );
        semilogy( EbNodB,
            sim_state(i).FER(sim_param(i).plot_iterations,:), sim_param(i).linetype );
    end
end
%testing figure( fig_number );
hold off;

% Raghu Gorthy Modification Start %
% plot FER vs. Es/No
% fig_number = fig_number + 1;
figure( fig_number );

% Outage Probability
for (i=find( sim_types(5,:) == 1 ) )
% If stored as Eb/No, convert to Es/No
if ( sim_param(i).SNR_type(2) == 'b' )
    % This is Eb/No
    EbNodB = sim_param(i).SNR;
    % Convert to Es/No
    EbNo = 10.*(EbNodB/10);
    EsNo = sim_param(i).rate*EbNo;
    EsNodB = 10*log10(EsNo);
else
    % This is Es/No
    EsNodB = sim_param(i).SNR;
end

% Plot FER versus Es/No in dB
figure( fig_number );
semilogy( EsNodB, sim_state(i).FER, sim_param(i).linetype );
hold on;
end

% FER of coded modulation
for (i=find( sim_types(4,:) == 1 ) )
% If stored as Eb/No, convert to Es/No
if ( sim_param(i).SNR_type(2) == 'b' )
    % This is Eb/No
    EbNodB = sim_param(i).SNR;
    % Convert to Es/No
    EbNo = 10.*(EbNodB/10);
    EsNo = sim_param(i).rate*EbNo;
    EsNodB = 10*log10(EsNo);
else
    % This is Es/No
    EsNodB = sim_param(i).SNR;
end

% only plot the last iteration
if ( length( sim_param(i).max_iterations ) )
    max_iter = sim_param(i).max_iterations;
else
    max_iter = 1;
end

% Plot FER versus Es/No in dB
figure( fig_number );
semilogy( EsNodB, sim_state(i).FER( max_iter, : ), sim_param(i).linetype );
hold on;
end

figure( fig_number );
legend( sim_param( find( sim_types(5,:) == 1 ) ).legend, sim_param( find( sim_types(4,:) == 1 ) ).legend, 0 );
% xlabel( 'Es/No in dB' );
% ylabel( 'FER' );

% Now plot the other iterations
for (i=find( sim_types(4,:) == 1 ) )
    if ( length( sim_param(i).plot_iterations ) )
        % If stored as Eb/No, convert to Es/No
        if ( sim_param(i).SNR_type(2) == 'b' )
            % This is Eb/No
            EbNodB = sim_param(i).SNR;
            % Convert to Es/No
            EbNo = 10.^((EbNodB/10);
            EsNo = sim_param(i).rate*EbNo;
            EsNodB = 10*log10(EsNo);
        else
            % This is Es/No
            EsNodB = sim_param(i).SNR;
        end
        % plot the other iterations
        figure( fig_number );
        semilogy( EsNodB,
            sim_state(i).FER(sim_param(i).plot_iterations,:), sim_param(i).linetype );
    end
end

% Raghu Gorthy Modification End %

end

% plot throughput vs. Es/No, if there are any throughput curves requested
if ( sum( sim_types(6,:) ) )
    fig_number = fig_number + 1;
    figure( fig_number );

    % plot throughput vs. Es/No
    for (i=find( sim_types(6,:) == 1 ) )
        plot( sim_param(i).SNR, sim_state(i).throughput,
            sim_param(i).linetype );
        hold on;
    end
    legend( sim_param( find( sim_types(6,:) == 1 ) ).legend, 2 );
    xlabel( 'Es/No in dB' );
    ylabel( 'Normalized throughput' );
    hold off;
end

% plot min Eb/No vs. h for nonorthogonal FSK under BW constraints.
if ( sum( sim_types(7,:) ) )
    fig_number = fig_number + 1;
figure( fig_number );

    % plot min Eb/No vs. h
    for (i=find( sim_types(7,:) == 1 ) )
        [Y,I] = sort( sim_param(i).h );
        plot( sim_param(i).h(I), sim_state(i).min_EbNodB(I),
            sim_param(i).linetype );
        hold on;
    end
    legend( sim_param( find( sim_types(7,:) == 1 ) ).legend, 2 );
xlabel( 'h' );
ylabel( 'min Eb/No (in dB)' );
hold off;

fig_number = fig_number + 1;
figure( fig_number );

    % plot min Eb/No vs. rate
    for (i=find( sim_types(7,:) == 1 ) )
        [Y,I] = sort( sim_state(i).best_rate );
        plot( sim_state(i).best_rate(I), sim_state(i).min_EbNodB(I),
            sim_param(i).linetype );
        hold on;
    end
    legend( sim_param( find( sim_types(7,:) == 1 ) ).legend, 2 );
xlabel( 'code rate r' );
ylabel( 'min Eb/No (in dB)' );
hold off;
end

% plot min Eb/No vs. B for nonorthogonal FSK under BW constraint B.
if ( sum( sim_types(8,:) ) )
    fig_number = fig_number + 1;
figure( fig_number );

    % plot min Eb/No vs. B
    for (i=find( sim_types(8,:) == 1 ) )
        [Y,I] = sort( sim_param(i).bwconstraint );
        plot( sim_param(i).bwconstraint(I), sim_state(i).min_EbNodB(I),
            sim_param(i).linetype );
        hold on;
    end
    legend( sim_param( find( sim_types(8,:) == 1 ) ).legend, 2 );
xlabel( 'Bandwidth B' );
ylabel( 'min Eb/No (in dB)' );
hold off;
fig_number = fig_number + 1;
figure( fig_number );

% plot eta vs. min Eb/No
for (i=find( sim_types(8,:) == 1 ) )
    [Y,I] = sort( sim_param(i).bwconstraint );
    % plot( sim_state(i).min_EbNodB(I),
    1./sim_param(i).bwconstraint(I), sim_param(i).linetype );
    plot( 1./sim_param(i).bwconstraint(I), sim_state(i).min_EbNodB(I), sim_param(i).linetype );
    hold on;
end

legend( sim_param( find( sim_types(8,:) == 1 ) ).legend, 2 );

% plot optimal h vs. B
for (i=find( sim_types(8,:) == 1 ) )
    [Y,I] = sort( sim_param(i).bwconstraint );
    plot( sim_param(i).bwconstraint(I), sim_param(i).h(I),
    sim_param(i).linetype );
    hold on;
end

legend( sim_param( find( sim_types(8,:) == 1 ) ).legend, 2 );

% plot optimal rate vs. B
for (i=find( sim_types(8,:) == 1 ) )
    [Y,I] = sort( sim_param(i).bwconstraint );
    plot( sim_param(i).bwconstraint(I), sim_state(i).best_rate(I),
    sim_param(i).linetype );
    hold on;
end

legend( sim_param( find( sim_types(8,:) == 1 ) ).legend, 2 );
hold off;
end