IMPLEMENTATION OF A FORWARD ERROR CORRECTION TECHNIQUE USING
CONVOLUTIONAL ENCODING WITH VITERBI DECODING

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

Sachin Rawat
March 2004
This thesis entitled

IMPLEMENTATION OF A FORWARD ERROR CORRECTION TECHNIQUE

USING

CONVOLUTIONAL ENCODING WITH VITERBI DECODING

BY

SACHIN RAWAT

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, School of Electrical Engineering and Computer Science

R. Dennis Irwin

Dean, Russ College of Engineering and Technology
This thesis, as the name suggests, shows the working of a forward error correction (FEC) coding technique using convolutional encoding with Viterbi decoding. It can be used by anyone interested in designing or understanding wireless digital communications systems.

The thesis initially explains the working of a convolutional encoder. The encoded bit stream, is then passed through an additive white Gaussian noise (AWGN) channel, quantized and received at the decoder. Finally, the original data stream is recovered by either a hard decision Viterbi decoder or a soft decision Viterbi decoder. This entire FEC technique is demonstrated, both practically, using Matlab, and theoretically.

Also shown are simulation plots, characterizing the performance factors affecting the FEC coding technique. These factors include primarily the noise level as well as the encoder memory size.

Approved:

Jeffrey C. Dill

Professor,

School of Electrical Engineering and Computer Science
Acknowledgements

I would like to thank my advisor Dr. Jeffrey C. Dill for his help, guidance and immense knowledge throughout the course of this thesis. I would also like to thank the faculty and staff of the department of Electrical Engineering and Computer Science particularly my committee members Dr. David W. Matolak, Dr. Joseph E. Essman and, Prof. Sergio R. Lopez for all that I have learnt from them, and my graduate secretary Tammy Jordan.

I would also like to thank all my friends I made here at Athens over the past three years.

And finally I would like to thank my parents and my brother for their constant support and words of wisdom and belief that kept me driven and focused throughout my goal of achieving my Masters.
# Table of Contents

Abstract..........................................................................................................................3

Acknowledgements........................................................................................................4

Table of Contents..........................................................................................................5

List of Tables..................................................................................................................8

List of Figures.................................................................................................................9

**Chapter 1**..................................................................................................................12

Introduction....................................................................................................................12

  1.1 Communication.......................................................................................................12

  1.2 Digital Communication.........................................................................................12

  1.3 Error Correction Techniques................................................................................14

  1.4 Channel Noise.......................................................................................................15

  1.5 Scope of the thesis.................................................................................................16

**Chapter 2**..................................................................................................................18

Convolutional Encoding.................................................................................................18

  2.1 Coding...................................................................................................................18

    2.1.1 Block Coding.................................................................................................18

    2.1.2 Convolutional Coding..................................................................................18

  2.2 Convolutional Encoder.........................................................................................19

    2.2.1 Encoder Parameters.....................................................................................19

    2.2.2 Encoder Structure.......................................................................................20

    2.2.3 Encoder States............................................................................................21

  2.3 Special Case Convolutional Encoders...................................................................22
2.3.1 Punctured Codes.................................................................22
2.3.2 Dual-\( k \) Codes.................................................................26
2.3.3 Systematic vs. Non-Systematic Codes.................................27
2.4 Encoder Operation...............................................................28
2.5 Convolutional Representation..............................................30
  2.5.1 State Diagram Representation.........................................32
  2.5.2 Tree Diagram Representation..........................................33
  2.5.3 Trellis Diagram Representation.......................................35
2.6 Encoder – MATLAB Implementation.....................................36

Chapter 3...................................................................................39
Viterbi Decoding........................................................................39
  3.1 Decoding..............................................................................39
    3.1.1 Sequential Decoding.......................................................39
    3.1.2 Viterbi Decoding.............................................................39
  3.2 Hard Decision Viterbi Decoding..........................................41
    3.2.1 Maximum Hamming Distance........................................42
    3.2.2 Trellis Explanation.........................................................42
    3.2.3 Hard Decision Decoder – MATLAB Implementation........50
  3.3 Soft Decision Viterbi Decoding..........................................53
    3.3.1 Quantization.................................................................54
    3.3.2 Minimum Euclidean Distance.......................................54
    3.3.3 Trellis Explanation.........................................................55
    3.3.4 Soft Decision Decoder – MATLAB Implementation........61
3.3.5 Maximum Euclidean Distance......................................................................62

Chapter 4...........................................................................................................66

Performance Factors.........................................................................................66

4.1 Encoder Memory Size...................................................................................66

4.2 Signal to Noise Ratio (SNR).........................................................................67

Chapter 5...........................................................................................................87

Conclusion and Future Work..............................................................................87

5.1 Conclusions...................................................................................................87

5.2 Future Work..................................................................................................88

References.........................................................................................................90

Appendices.........................................................................................................92

Appendix A: Matlab Programs ...........................................................................92

Appendix B: Practical working of a (2, 1, 3) convolutional encoder using
the 3 input bits [1 0 1].........................................................................................110

Appendix C: Matlab Simulation Results............................................................116
List of Tables

Table 2.1: Bits transmitted from a punctured code over a time period ................... 23
Table 2.2: Example illustrating puncturing (code rate = 2/3) ................................. 24
Table 2.3: Result showing the working of a (2, 1, 3) convolutional encoder ................. 30
Table 2.4: State transition table for the (2, 1, 3) convolutional encoder ...................... 31
Table 4.1: SNR ($E_b/N_o$) in dB for the rate 1/2, $m = 2$ encoder .......................... 71
Table 4.2: Coding gain in dB for the rate 1/2, $m = 2$ encoder ............................... 71
Table 4.3: SNR ($E_b/N_o$) in dB for the rate 1/2, $m = 6$ encoder .......................... 74
Table 4.4: Coding gain in dB for the rate 1/2, $m = 6$ encoder ............................... 74
Table 4.5: SNR ($E_b/N_o$) in dB for the rate 1/3, $m = 2$ encoder .......................... 77
Table 4.6: Coding gain in dB for the rate 1/3, $m = 2$ encoder ............................... 77
Table 4.7: SNR ($E_b/N_o$) in dB for the rate 1/3, $m = 6$ encoder .......................... 80
Table 4.8: Coding gain in dB for the rate 1/3, $m = 6$ encoder ............................... 80
Table 4.9: SNR ($E_b/N_o$) in dB using the standard polynomials ............................. 85
Table 4.10: Coding gain in dB using the standard polynomials ............................... 85
Table 4.11: SNR ($E_b/N_o$) in dB using the tested polynomials ............................... 86
Table 4.12: Coding gain in dB using the tested polynomials ............................... 86
List of Figures

Figure 1.1: Block diagram of a digital communication system...............................13
Figure 1.2: Encode/decode and modulate/demodulate portions of a digital
communication link..............................................................................................14
Figure 2.1: A (2, 1, 3) convolutional encoder.......................................................21
Figure 2.2: Punctured code......................................................................................23
Figure 2.3: A (4, 3, 3) convolutional code, which has 3 input bits and 4 output bits......25
Figure 2.4: A dual-3 convolutional code.................................................................26
Figure 2.5: A systematic version of the (4, 3, 3) convolutional code. It has the same
number of memory registers, 3 input bits and 4 output bits. The output bits
consist of the original 3 input bits and a 4th parity bit........................................27
Figure 2.6: Theoretical operation of a (2, 1, 3) convolutional encoder.......................29
Figure 2.7: State diagram for the (2, 1, 3) convolutional encoder.............................32
Figure 2.8: Tree diagram for the (2, 1, 3) convolutional encoder..............................34
Figure 2.9: Trellis diagram for the (2, 1, 3) convolutional encoder...........................35
Figure 2.10: Encoded output bit stream 11 10 00 10 11 for the input bit stream 10100..........................................................37
Figure 3.1: System design of a Viterbi decoder.......................................................41
Figure 3.2: Step 1: Trellis and path metrics at time $t = 0$........................................43
Figure 3.3: Step 2: Trellis and path metrics at time $t = 1$.......................................44
Figure 3.4: Step 3a: Trellis and path metrics at time $t = 2$...................................45
Figure 3.5: Step 3b: Updated trellis and path metrics at time $t = 2$.......................46
Figure 3.6: Step 4a: Trellis and path metrics at time $t = 3$...................................47
Figure 3.7: Step 4b: Updated trellis and path metrics at time $t = 3$.

Figure 3.8: Step 5a: Trellis and path metrics at time $t = 4$.

Figure 3.9: Step 5b: Updated trellis and path metrics at time $t = 4$.

Figure 3.10: Decoded sequence 10100 for the noisy encoded bit stream

\[ 01 \ 10 \ 00 \ 10 \ 11 \]

Figure 3.11: Output of the hard decision Viterbi decoder – 10100.

Figure 3.12: Step 1: Trellis and path metrics at time $t = 0$.

Figure 3.13: Step 2: Trellis and path metrics at time $t = 1$.

Figure 3.14: Step 3a: Trellis and path metrics at time $t = 2$.

Figure 3.15: Step 3b: Updated trellis and path metrics at time $t = 2$.

Figure 3.16: Step 4a: Trellis and path metrics at time $t = 3$.

Figure 3.17: Step 4b: Updated trellis and path metrics at time $t = 3$.

Figure 3.18: Step 5a: Trellis and path metrics at time $t = 4$.

Figure 3.19: Step 5b: Updated trellis and path metrics at time $t = 4$.

Figure 3.20: Decoded sequence 10100 for the noisy encoded bit stream

\[ 3-4 \ -43 \ 33 \ -43 \ -4-4 \]

Figure 3.21: Output of the soft decision Viterbi decoder – 10100.

Figure 3.22: Decoded sequence 10100 for the noisy encoded bit stream

\[ 3-4 \ -43 \ 33 \ -43 \ -4-4 \] (maximum likelihood)

Figure 3.23: Output of the soft decision Viterbi decoder – 10100 (maximum likelihood).

Figure 4.1: Plot of BER vs. $E_b/N_0$ for a rate $1/2$, $m = 2$ encoder, considering a hard decision and a soft decision Viterbi decoder (using standard polynomials).
Figure 4.2: Plot of BER vs. $E_b/N_0$ for a rate 1/2, $m = 2$ encoder, considering a hard decision and a soft decision Viterbi decoder (using tested polynomials)……..69

Figure 4.3: Plot of BER vs. $E_b/N_0$ for a rate 1/2, $m = 6$ encoder, considering a hard decision and a soft decision Viterbi decoder (using standard polynomials)....72

Figure 4.4: Plot of BER vs. $E_b/N_0$ for a rate 1/2, $m = 6$ encoder, considering a hard decision and a soft decision Viterbi decoder (using tested polynomials)……..72

Figure 4.5: Plot of BER vs. $E_b/N_0$ for a rate 1/3, $m = 2$ encoder, considering a hard decision and a soft decision Viterbi decoder (using standard polynomials)....75

Figure 4.6: Plot of BER vs. $E_b/N_0$ for a rate 1/3, $m = 2$ encoder, considering a hard decision and a soft decision Viterbi decoder (using tested polynomials)……..75

Figure 4.7: Plot of BER vs. $E_b/N_0$ for a rate 1/3, $m = 6$ encoder, considering a hard decision and a soft decision Viterbi decoder (using standard polynomials)....78

Figure 4.8: Plot of BER vs. $E_b/N_0$ for a rate 1/3, $m = 6$ encoder, considering a hard decision and a soft decision Viterbi decoder (using tested polynomials)……..78

Figure 4.9: Plot of BER vs. $E_b/N_0$ for various combinations of the encoder, with a hard decision decoder (using standard polynomials).................................81

Figure 4.10: Plot of BER vs. $E_b/N_0$ for various combinations of the encoder, with a hard decision decoder (using tested polynomials).................................81

Figure 4.11: Plot of BER vs. $E_b/N_0$ for various combinations of the encoder, with a soft decision decoder (using standard polynomials)..............................83

Figure 4.12: Plot of BER vs. $E_b/N_0$ for various combinations of the encoder, with a soft decision decoder (using tested polynomials)..............................83
Chapter 1: Introduction

1.1 Communication

The basic subject of communication involves the transmission of information, in analog or digital form, from a source that generates the information to one or more destinations. A common design goal for a communication system is to transmit this information through a channel and receive it with as few errors as possible.

1.2 Digital Communication

Today, there is an ever-growing demand for data communications. This remains evident, from the fact that most data communications are between computers, or between digital instruments or terminals to computers. Such digital terminations are naturally best served by digital communication systems.

In addition, digital systems offer faster data processing and the potential of extremely low error rates. Also, digital circuits are not only more cost-effective but, are less subject to distortion and interference making them more reliable than analog circuits. Thus, we see why digital communication systems are fast-replacing the analog communication systems, and becoming increasingly popular in the field of telecommunications.

A functional block diagram illustrating the signal flow and the signal-processing steps through a typical digital communication system is shown in Figure 1.1 [1].
Figure 1.1. Block diagram of a digital communication system.

As seen, this diagram consists of two blocks. The upper block shows the signal path from the information source to the output of the transmitter (XMT). The lower block shows the signal path from the input of the receiver (RCV) to the information sink. The upper block makes use of coding and modulation techniques before transmitting data over a channel. This channel is a physical medium used to transmit data from the transmitter to the receiver. In the receiver side, the lower block makes use of decoding and demodulation techniques to obtain estimates of the original data.
1.3 Error Correction Techniques

Channel coding and modulation provide the means of mapping information into waveforms such that the receiver (with an appropriate demodulator and decoder) can recover the information in a reliable manner. A block diagram derived from Figure 1.1, which shows this part of the digital communication link, is shown in Figure 1.2.

![Diagram](image)

**Figure 1.2.** Encode/decode and modulate/demodulate portions of a digital communication link.

As seen in Figure 1.2, convolutional encoding is one way of performing channel coding. Another method uses block codes. In these methods, redundant bits are used to help determine the occurrence of an error due to noise present in the channel. In the receiver, Viterbi decoding is a way of performing channel decoding. Another method is turbo codes. Turbo codes can be applied to the encoding process too. In these methods, errors can be “automatically” corrected (within specified limitations) to recover the original information.
Error correction is a technique defined by the methods of encoding and decoding. One such technique, called the automatic repeat request (ARQ), simply recognizes the occurrence of an error and requests the sender retransmit the message. Another technique is known as the forward error correction (FEC) technique. This technique allows for “automatic” correction of errors.

1.4 Channel Noise

The term channel refers to the transmission medium connecting the transmitter and receiver. A communications channel can either be a wired channel, for example coaxial cables and fiber optic cables, or a wireless channel, for example atmosphere and empty space.

The term noise refers to unwanted electrical signals associated with electrical systems. Noise can arise from man-made sources, such as spark-plug ignition noise and switching transients, or natural sources such as the atmosphere and the sun. Good engineering designs help minimize this noise through filtering, shielding or modulation. However, there is one source of noise called thermal or Johnson that cannot be eliminated. Thermal noise is generated by the random electron movement in the receiver. Thermal noise can be described by a zero-mean Gaussian random process. Further, since the noise power has a uniform power spectral density over much of the frequency domain of interest, it can be referred to as white noise. This noise is then simply added to the information being transmitted, characterizing the noise additive. The input at the receiver is thus the sum of the transmitted information and this noise.
Thermal noise is present in all communication systems and for many systems it is the most prominent source of noise. The thermal noise with its distinct characteristics: additive, white and Gaussian, thus, make it a useful model for a digital communications channel.

1.5 Scope of the thesis

The scope of the thesis is diagrammatically shown in Figure 1.2. The thesis will help explain one method to achieve forward error correction for a digital communication system. This method incorporates convolutional encoding with Viterbi decoding. The performance of this FEC technique is characterized for an AWGN channel.

Chapter 2 explains the design and implementation of a convolutional encoder. The various representations for a convolutional encoder are shown. The working of the convolutional encoder is studied and Matlab code has been written and tested. There is also a brief description of some special case convolutional encoders. Chapter 3 explains the design and implementation of a Viterbi decoder. The working of both a hard decision and a soft decision Viterbi decoder are studied. This includes studying the characteristics and the differences between the implementation of both these methods. Again, Matlab code written for these two decoders is described. Chapter 4 discusses the performance factors that affect the FEC technique. These include the encoder memory size and the noise level. These factors are studied using an encoder with code rates of 1/2 and 1/3. These code rates are used with an encoder having a memory size of 2 and 6. Also, two sets of generator polynomials are considered with the encoder used. One set was the standard set of polynomials as listed in [1]. The other set was found by us using the
Matlab codes written. Plots characterizing these performance factors are shown and reviewed. Finally, chapter 5 concludes the thesis by summarizing the results. Also, suggestions for future work with the FEC technique are mentioned.
Chapter 2: Convolutional Encoding

2.1 Coding

A useful tool in the design of reliable digital communication systems is channel coding. Channel coding provides improved error performance by adding redundant information to the input data being transmitted through a channel. There are two major forms of channel coding: block coding and convolutional coding.

2.1.1 Block Coding

Block coding techniques map a fixed number of message symbols to a fixed number of code symbols. A block encoder treats each block of data independently and is a memory less device (between blocks though, it can be viewed as having intra-block memory).

2.1.2 Convolutional Coding

A convolutional encoder accepts a sequence of message symbols and produces a sequence of code symbols. Its computations depend not only on the current set of input symbols but, on some of the previous input symbols as well. In practice, convolutional codes operate on a block at a time and so like block codes, have intra-block memory, and possibly no inter-block memory.
2.2 Convolutional Encoder

A convolutional encoder is made of a fixed number of shift registers. Each input bit enters a shift register and the output of the encoder is derived by combining the bits in the shift register. The number of output bits depends on the number of modulo 2-adders used with the shift registers.

2.2.1 Encoder Parameters

Convolutional codes are commonly specified by the three parameters \((n, k, m)\) where,

\[
\begin{align*}
    n & = \text{number of output bits} \\
    k & = \text{number of input bits} \quad \text{and,} \\
    m & = \text{number of memory registers}.
\end{align*}
\]

The quantity \(k/n\) called the code rate is a measure of the bandwidth efficiency of the code. Commonly \(k\) and \(n\) parameters range from 1 to 8, \(m\) from 2 to 10, and the code rate from 1/8 to 7/8 except for deep space applications where code rates as low as 1/100 or even longer can be employed.

Convolutional codes can also be specified by the parameters \((n, k, L)\) where, \(L\) is known as the constraint length of the code and is defined as the number of bits in the encoder memory that affects the generation of the \(n\) output bits. The convolutional codes discussed here will be referred to as \((n, k, L)\) and not as \((n, k, m)\) codes.
2.2.2 Encoder Structure

The term generator polynomial \((g)\) characterizes the encoder connections. The selection of which bits (in the memory registers) are to be added (using modulo-\(q\) adders) to produce the output bits is called the generator polynomial for that output bit. There are many choices for polynomials for any \(m\) order code. Good polynomials are normally found by trial and error through computer simulations. We used two sets of generator polynomials in this thesis which are listed in the Matlab codes attached in Appendix A. One set was that of the standard generator polynomials as listed in [1]. The other was another set, obtained by us, which closely matches the performance characteristics of the standard set of generator polynomials.

In order to understand the working of a convolutional encoder and further the forward error correction technique, the following assumptions have been made:

(a) A \((2, 1, 3)\) convolutional encoder is used.

(b) A 3-bit input sequence is used specified by the bits \([1 0 1]\).

(c) 2 generator polynomials are used, specified by the bits \([1 1 1]\) and \([1 0 1]\).

It is easy to construct a convolutional encoder. We first draw \(m\) boxes representing the \(m\) memory registers. Then we draw \(n\) modulo-2 adders representing the \(n\) output bits. Finally, we connect the memory registers to the adders using the bits specifying the generator polynomials.

Shown in Figure 2.1 is a \((2, 1, 3)\) convolutional encoder. This encoder is going to be used to encode the 3-bit input sequence \([1 0 1]\) with the two generator polynomials specified by the bits \([1 1 1]\) and \([1 0 1]\). \(u_i\) represents the input bit, and \(v_j\) and \(v_2\) represent
the output bits 1 and 2 respectively. $u_0$ and $u_{-1}$ represent the initial state of the memory registers which are initially set to zero.

![Convolutional Encoder Diagram](image)

**Figure 2.1.** A (2, 1, 3) convolutional encoder.

### 2.2.3 Encoder States

The (2, 1, 3) encoder shown in Figure 2.1 has a constraint length of 3. The first memory register holds the incoming bit and the other two memory registers hold the bits, which specify the states of the code. Thus 2 bits or 4 different combinations of these bits can be present in these memory registers. These 4 different combinations determine the output for the coded sequence.

Thus the (2, 1, 3) convolutional encoder has four states, which are: 00, 01, 10, 11.
2.3 Special Case Convolutional Encoders

Discussed in this section are two special cases of a convolutional encoder. Their operation is a representation of all convolutional encoders. The only difference is seen in the structure of these encoders which will be explained.

2.3.1 Punctured Codes

For the special case of $k = 1$, the codes of rates $1/2$ thru $1/7$ are sometimes called the mother codes. These single bit input codes can be combined to produce punctured codes, which give code rates other than $1/n$.

By using two rate $1/2$ codes together as shown in Figure 2.2 and then just not transmitting one of the output bits, a rate $1/2$ code can be converted into a $2/3$ rate code. This concept is called puncturing. On the receiver side, dummy bits that do not affect the decoding metric are inserted in the appropriate places before decoding.
Figure 2.2. Punctured code.

Figure 2.2 also shows a matrix which explains which bits are transmitted and which are not. This can be repeated over a time period as seen in Table 2.1.

<table>
<thead>
<tr>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>- - - - - -</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>T</td>
<td>X</td>
<td>T</td>
<td>X</td>
<td>T</td>
<td>- - - - - -</td>
</tr>
</tbody>
</table>

Table 2.1. Bits transmitted from a punctured code over a time period.
An example illustrating puncturing is as shown in Table 2.2. The term *puncturing sequence* refers to a binary sequence; 0 and 1 means that the corresponding code symbol is not transmitted and transmitted, respectively. The periodic sequence $[11 \ 10 \ 01]$ is used as the puncturing sequence.

<table>
<thead>
<tr>
<th></th>
<th>11</th>
<th>10</th>
<th>00</th>
<th>10</th>
<th>11</th>
<th>00</th>
<th>Original sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>11</td>
<td>10</td>
<td>01</td>
<td>11</td>
<td>10</td>
<td>01</td>
<td>Puncturing sequence</td>
</tr>
<tr>
<td>2nd</td>
<td>11</td>
<td>1X</td>
<td>X0</td>
<td>10</td>
<td>1X</td>
<td>X0</td>
<td>Punctured sequence</td>
</tr>
</tbody>
</table>

(X – Not transmitted)

**Table 2.2.** Example illustrating puncturing (code rate = 2/3).

**Structure of a code for $k > 1$**

Alternately codes with $k > 1$ such as the $(4, 3, 3)$ code can be created as shown in Figure 2.3. This code has 3 input bits and 4 output bits. The number of states is 64. Also, this code requires polynomials of the order of 9.

The procedure for drawing the structure of a $(n, k, m)$ code where $k$ is greater than 1 is as follows. First we draw $k$ sets of $m$ boxes. Then draw $n$ adders. Next, we connect the $n$ adders to the memory registers specified by the order of the polynomial which in this case is 9.

In this way we obtain the structure of the $(4, 3, 3)$ code as shown in Figure 2.3. The shaded boxes represent the constraint length, i.e., 6 and it is in these registers that the code will reside in 1 of its 64 states. Also at every time period, the three input bits are shifted to the next set of memory registers.
Figure 2.3. A (4, 3, 3) convolutional code, which has 3 input bits and 4 output bits.

Although a different technique is shown to construct codes with $k > 1$, punctured codes are preferred. The reason being that with punctured codes, the code rates can be changed dynamically (through software) depending on the channel condition such as rain, etc. This can be explained from Figure 2.2. A fixed implementation, although easier, does not allow this flexibility.
2.3.2 Dual-$k$ Codes

Dual-$k$ codes are a class of nonbinary convolutional codes that are easily decoded by means of the Viterbi algorithm using either hard decision or soft decision decoding. It consists of two $k$-bit memory register stages and $n = 2^k$ order of polynomials. A dual-3 convolutional code is as shown in Figure 2.4.

![Figure 2.4. A dual-3 convolutional code.](image-url)
This dual-3 convolutional code has a code rate of 3/4. Again, at every time period the three input bits are shifted to the next set of memory registers and four output bits are generated. Thus, we see how dual-k codes also help achieve code rates other than 1/n.

### 2.3.3 Systematic vs. Non-Systematic Codes

A special form of convolutional codes in which the output bits contain the entire sequence of the input bits is called the **systematic** form. A systematic version of the (4, 3, 3) code is as shown in Figure 2.5. Of the 4 output bits, 3 are exactly the same as the 3 input bits. The 4th bit is kind of a parity bit produced from a combination of 3 bits using a single polynomial.

![Figure 2.5](image-url)

**Figure 2.5.** A systematic version of the (4, 3, 3) convolutional code. It has the same number of memory registers, 3 input bits and 4 output bits. The output bits consist of the original 3 bits and a 4th ‘parity’ bit.
Systematic codes are often preferred over the non-systematic codes as they require less hardware for encoding. Another important property of systematic codes is that they are non-catastrophic, which means that errors cannot propagate catastrophically. A catastrophic event is defined as an event whereby a finite number of code symbol errors cause an infinite number of decoded data bit errors. All these properties make them very desirable. The error protection properties of the systematic codes however are the same as those of the non-systematic codes.

2.4 Encoder Operation

The encoding of the 3-bit input sequence [1 0 1] for the 6 sequential time periods is as shown in Figure 2.6:

![Diagram](image-url)
Figure 2.6. Theoretical operation of a (2, 1, 3) convolutional encoder.
The result of the above encoding at each of these 6 sequential time periods is as shown in Table 2.3.

<table>
<thead>
<tr>
<th>Time</th>
<th>Input Bits</th>
<th>Output Bits</th>
<th>Encoder Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1 1</td>
<td>0 0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1 0</td>
<td>1 0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0 0</td>
<td>0 1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1 0</td>
<td>1 0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1 1</td>
<td>0 1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
</tbody>
</table>

Table 2.3. Result showing the working of a (2, 1, 3) convolutional encoder.

Thus the encoded output sequence for the 3-bit input sequence [1 0 1] is:

11 10 00 10 11.

2.5 Convolutional Representation

The previous section demonstrated the logical functioning of the convolutional encoder. The convolutional encoder can use a look-up table, otherwise called the state transition table to do the encoding. The state transition table consists of four items:

(a) The input bit.
(b) The state of the encoder, which is one of the 4 possible states (00 01 10 11) for the (2, 1, 3) convolutional encoder.

(c) The output bits, which for the (2, 1, 3) convolutional encoder are: 00 01 10 11, since only two bits are output.

(d) The output state which is the input state for the next bit.

The state transition table for the (2, 1, 3) convolutional encoder is as shown in Table 2.4:

<table>
<thead>
<tr>
<th>Input Bit</th>
<th>Input State</th>
<th>Output Bits</th>
<th>Output State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>1</td>
<td>0 0</td>
<td>1 1</td>
<td>1 0</td>
</tr>
<tr>
<td>0</td>
<td>0 1</td>
<td>1 1</td>
<td>0 0</td>
</tr>
<tr>
<td>1</td>
<td>0 1</td>
<td>0 0</td>
<td>1 0</td>
</tr>
<tr>
<td>0</td>
<td>1 0</td>
<td>1 0</td>
<td>0 1</td>
</tr>
<tr>
<td>1</td>
<td>1 0</td>
<td>0 1</td>
<td>1 1</td>
</tr>
<tr>
<td>0</td>
<td>1 1</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1</td>
<td>1 1</td>
<td>1 0</td>
<td>1 1</td>
</tr>
</tbody>
</table>

Table 2.4. State transition table for the (2, 1, 3) convolutional encoder.
Graphically there are three ways to represent the encoder to gain a better understanding of its operation. These are:

(1) State Diagram Representation
(2) Tree Diagram Representation
(3) Trellis Diagram Representation

2.5.1 State Diagram Representation

A state diagram for the (2, 1, 3) convolutional encoder is as shown in the Figure 2.7:

Figure 2.7. State diagram for the (2, 1, 3) convolutional encoder.
A state refers to the contents of 2 cells of the shift register. Each circle represents a state. At any one time, the encoder resides in one of these states. The lines to and from it show state transitions that are possible as bits arrive. Only two events can happen at each time, arrival of the bit 1 or arrival of the bit 0. Each of these two events allows the encoder to “jump” into a different state.

When compared with the state transition table, as seen, the state diagram contains the same information but in a graphical representation. The solid lines indicate the arrival of the bit 0 and the dashed lines indicate the arrival of the bit 1. The output bits for each case are shown on the line and the arrow indicates the state transition.

2.5.2 Tree Diagram Representation

A tree diagram for the (2, 1, 3) convolutional encoder is as shown in Figure 2.8. The state diagram completely characterizes the encoder but one cannot easily use it for tracking the encoder transitions as a function of time. This is evident as the diagram does not represent time history. The tree diagram adds the dimension of time to the state diagram, i.e., it shows the passage of time as we go deeper into the tree branches. This makes it a better approach than the state diagram but for representing convolutional codes.

With the tree diagram, instead of “jumping” from one state to another, branches of the tree are travelled depending on the arrival of the bit 1 or the arrival of the bit 0. If the bit 0 is received at the encoder input, we go up and if the bit 1 is received we go down in the tree. The first two bits represent the output bits and the bits within the parentheses represent the output state.
Figure 2.8. Tree diagram for the (2, 1, 3) convolutional encoder.
2.5.3 Trellis Diagram Representation

A trellis diagram for the (2, 1, 3) convolutional encoder is as shown in Figure 2.9:

![Trellis Diagram](image)

**Figure 2.9.** Trellis diagram for the (2, 1, 3) convolutional encoder.

Trellis diagrams are generally preferred over both the tree diagrams and the state diagrams because they represent linear time sequencing of events. The horizontal or $x$-axis represents time and the vertical or $y$-axis all possible states of the code. We move horizontally through the trellis with the passage of time. Each transition represents the arrival of new bits.

The trellis diagram is drawn by lining up all the possible states in the vertical axis. Then we connect each state to the next by the allowable codewords for that state. Since the code is binary there are only two choices possible at each state. These are determined by the arrival of either the bit 0 or the bit 1. The arrows show the input bit and the output
bits are shown in parentheses. The solid lines going upwards represent the bit 0 and the
dashed lines going downwards represent the bit 1.

The trellis diagram is unique to each code, just as both the state diagram and the
state diagram are. The trellis diagram can be drawn for as many periods as needed and
each period repeats the possible transitions.

The trellis diagram conventionally begins at state 00. Starting from here, the
trellis expands and in $L$ bits becomes fully “populated” such that all transitions are
possible. The possible transitions then repeat from this point on.

2.6 Encoder - MATLAB Implementation

The operation of the convolutional encoder was as demonstrated in Section 2.4.
This procedure was implemented using Matlab. The Matlab codes written for the FEC
technique are attached in Appendix A. The main code Forward_Error_Correction_technique.m calls the functions which perform the processes
of encoding and decoding. This section discusses the functions written that performs
convolutional encoding. The two functions we wrote were encode_block and encode_bit
and are as described.

Function encode_block:

(i) receives the generator polynomial and the input bit stream to be encoded, in our
example [1 0 1].

(ii) zero pads the input bit stream. In our example, the sequence [1 0 1] is zero padded
such that we have the new sequence [1 0 1 0 0].
(iii) calls the function `encode_bit`.

(iv) generates the output of the convolutional encoder, in our example the codeword `[11 10 00 10 11]`.

Function `encode_bit`:

(i) receives the generator polynomial, the input bit to be encoded and input state bits.

(ii) generates the encoded output bits corresponding to the input bit and input state bits.

(iii) generates the next state bits corresponding to the input bit and input state bits.

The encoded sequence for the 3-bit input sequence `[1 0 1]` is shown by the trellis diagram in Figure 2.10:

![Trellis Diagram](image)

**Figure 2.10.** Encoded output bit stream 11 10 00 10 11 for the input bit stream 10100.
As seen, coding is easy. We go up for the bit 0 and down for the bit 1. The path taken by the bits in our example [1 0 1 0 0], is shown by the lines. Thus we see how we achieve the encoded output bit stream [11 10 00 10 11] for the 3-bit input stream [1 0 1].
Chapter 3: Viterbi Decoding

3.1 Decoding

Channel decoding is defined as the process of recovering the encoded input data stream, at the receiver, once transmitted through a channel. There are two major forms of channel decoding for convolutional codes: sequential decoding and the maximum likelihood decoding or Viterbi decoding.

3.1.1 Sequential Decoding

Sequential decoding was one of the first methods proposed for decoding a convolutionally encoded bit stream and is best described by analogy. It was first proposed by Wozencraft and later a better version was proposed by Fano.

In sequential decoding, we deal with just one path at a time, thus enabling both forward and backward movements through the trellis. The main disadvantage with this decoding technique is its variable decoding time, since the number of calculations increases with the number of input bits.

3.1.2 Viterbi Decoding

Viterbi decoding, also often referred to as the maximum likelihood decoding was developed by Andrew J. Viterbi, a founder of Qualcomm Corporation. His seminal paper
In Viterbi decoding, unlike in sequential decoding, the decoding time is fixed and not variable thus making it more suitable to hardware implementation. Here, we narrow the options systematically at each trellis stage. The principal used to reduce the choices are:

(i) The errors occur infrequently. The probability of error is small.

(ii) The probability of two errors in a row is much smaller than a single error that is the errors are distributed randomly.

The Viterbi decoder examines an entire received sequence of a given length. The decoder computes a metric for each path and makes a decision based on this metric. All paths are followed until two paths converge on one node. Henceforth based on a decision, discussed ahead, one of the two paths is chosen. The paths selected are called the survivors.

For an $N$ bit sequence, the total number of possible received sequence is $2^N$. Of these only $2^{kl}$ are valid.

A practical system design of the Viterbi decoder demonstrating the data flow is as shown in Figure 3.1. This shows the necessary computational paths for a working system. The working of these blocks will become clearer in the sections to follow.
As discussed, when two paths converge on one node, only one survivor path is chosen based on a decision. This decision can be achieved in two ways, resulting in the following two types of Viterbi decoding:

(i) Hard decision Viterbi decoding

(ii) Soft decision Viterbi decoding

### 3.2 Hard Decision Viterbi Decoding

Also referred to as the soft input Viterbi decoding technique, this uses a path metric called the *Hamming Distance* metric, to determine the survivor paths as we move through the trellis.
3.2.1 Maximum Hamming Distance

Hamming distance between the received codeword and the allowable codeword is calculated by checking the corresponding bit positions of the two codewords. For example the Hamming distance between the codewords 00 and 11 is 0 or the Hamming distance between the codewords 00 and 00 is 2. The Hamming distance metric is cumulative so that the path with the largest total metric is the final winner. Thus the hard decision Viterbi decoding makes use of the maximum Hamming distance in order to determine the output of the decoder.

*Note:* Hamming distances as explained, can only be calculated when we use the binary bits 0 and 1. Thus, before the corrupted data signal is fed to the input of the hard decision Viterbi decoder, voltages that are less than and equal to zero are represented by the bit 0, and voltages that are greater than 0 are represented by the bit 1. This is also known as one-bit quantization. Quantization is discussed more in detail in Section 3.3.1.

3.2.2 Trellis Explanation

The actual working of the hard decision Viterbi decoder is as explained in the following figures. The trellis is drawn for each time tick, based on the example considered, of the encoded 3-bit input stream [1 0 1].

The corrupted data bit stream at the input of the hard decision Viterbi decoder is assumed as [01 10 00 10 11].
Step 1: At time $t = 0$, we have received the bits 01. The decoder always starts at the initial state of 00. From this point on it has two paths available, but neither matches the incoming bits. The decoder computes the branch metric for both of these and will continue simultaneously along both of these branches, in contrast to the sequential decoding where a choice is made at every decision point. The metric for both branches is equal to 1, which means that one of the two bits was “matched” with the incoming bits. The corresponding trellis and path metric are as shown in Figure 3.2:

![Trellis and path metrics at time $t = 0$.](image)

Step 2: At time $t = 1$, we have received the bits 10. The decoder fans out from these two states to all four of the possible states. The branch metrics for these branches are computed and added to the previous branch metrics. The corresponding trellis new path metrics are as shown in Figure 3.3:
Step 3: At time $t = 2$, we have received the bits 00. The paths progress forward and now begin to converge at the nodes. Two metrics are computed for each of the paths coming into a node. Considering the maximum Hamming distance principle, at each node we discard the path with the lower metric because it is less likely. This is as shown in Figure 3.4:
Figure 3.4. Step 3a: Trellis and path metrics at time $t = 2$.

This discarding of paths at each node helps to reduce the number of paths that have to be examined and thus, gives the Viterbi method of decoding its strength. The corresponding trellis and new path metrics are as shown in Figure 3.5:
Step 4: At time $t = 3$, we have received the bits 10. Again the metrics are computed for all paths. We discard all smaller metrics but keep both paths if they have equal metrics. The corresponding trellis and new path metrics are as shown in Figure 3.6 and Figure 3.7:
Figure 3.6. Step 4a: Trellis and path metrics at time $t = 3$.

Figure 3.7. Step 4b: Updated trellis and path metrics at time $t = 3$. 
Step 5: At time $t = 4$, we have received the bits 11. The procedure from Step 4 is repeated. But now, the trellis is complete. The corresponding trellis and new path metrics are as shown in Figure 3.8 and Figure 3.9:

![Trellis and path metrics at time $t = 4$.](image)

**Figure 3.8.** Step 5a: Trellis and path metrics at time $t = 4$. 
Figure 3.9. Step 5b: Updated trellis and path metrics at time $t = 4$.

The path with the highest metric is looked for and a winner path is traced. The path traced by the states 00, 10, 01, 10, 01, 00 and corresponding to the bits 10100 is the decoded sequence and is as shown in Figure 3.10:
Figure 3.10. Decoded sequence 10100 for the noisy encoded bit stream 01 10 00 10 11.

Thus, we see how the hard decision Viterbi decoder, using maximum Hamming distances, works and achieves the decoded data bit stream, from a convolutionally encoded input data bit stream transmitted over an AWGN channel from the transmitter. This process is actually shown in the Matlab codes written which are as explained in the next section.

3.2.3 Hard Decision Decoder – MATLAB Implementation

The working of the hard decision Viterbi decoder was as demonstrated in Section 3.2.2.

This procedure was implemented using Matlab. The Matlab codes written for the FEC technique are attached in Appendix A. The main code Forward_Error_Correction_technique.m calls the functions which perform the processes
of encoding and decoding. This section discusses the functions written that performs hard
decision Viterbi decoding. The main function we wrote was hard_decision_Viterbi and is
as described.

Function *hard_decision_Viterbi*:

(i) receives the generator polynomial and the corrupted encoded data bit stream, in our
example [01 10 00 10 11].
(ii) initializes matrices for storing the maximum path metric (trellis) and the
corresponding path (path).
(iii) initializes an output matrix (output) which represents the output bits, given the input
states and received bits.
(iv) initializes a transition matrix (transition) which represents the output states, given the
input states and received bits.

*Note:* steps (iii) and (iv) are implemented by calling the functions *bin_state* and
*encode_bit*. The function *bin_state* generates all the states, in our example 00, 01, 10, 11
and the function *encode_bit* works as explained in Section 2.6. Step (iv) further calls on
the function *int_state* which gets an integer value for the binary denoted state i.e. 00 = 0,
01 = 1, 10 = 2, 11 = 3.
(v) for each time interval:

(a) from the matrices output and transition, the corresponding branch metrics are
calculated.

*Note:* branch metrics are calculated by calling the function *hard_dist.*
(b) the corresponding path metrics are then calculated from the matrix trellis and branch metrics.

(c) the matrices trellis and path are updated corresponding to the path with the maximum Hamming distance.

(vi) after all the path metrics are calculated and stored in the matrix trellis, with their corresponding paths in the matrix path, we make sure that the first row of the matrices trellis and path holds the information required to determine the decoded data bit stream.

(vii) generates the output of the hard decision Viterbi decoder, which in our example is the codeword [1 0 1 0 0].

The practical working of the hard decision Viterbi decoder using the functions just discussed is as shown in Appendix B. Thus as seen, is the working of a hard decision Viterbi decoder, both theoretically and practically.
The output of the hard decision Viterbi decoder is as shown in Figure 3.11:

![Diagram of Viterbi decoder output with states and transitions labeled with numbers.]

**Figure 3.11.** Output of the hard decision Viterbi decoder – 10100.

Thus, demonstrated is the working of the hard decision Viterbi decoder and we see how the input data bit stream [1 0 1] is obtained.

### 3.3 Soft Decision Viterbi Decoding

Also referred to as the soft input Viterbi decoding technique, this uses a path metric called the *Euclidean Distance* metric, to determine the survivor paths as we move through the trellis.
### 3.3.1 Quantization

In practical systems, we quantize the received channel data sequence with one or a few bits of precision in order to reduce the complexity of the Viterbi decoder. If the received data stream is quantized to one-bit of precision, i.e., voltages that are less than or equal to 0 are represented by the bit 0, and voltages that are greater than 0 are represented by the bit 1, then we get as what was explained before, the hard decision Viterbi decoder. If however the received data stream is quantized with two or more bits of precision we get the soft decision Viterbi decoder.

The soft decision Viterbi decoder discussed in this report uses a 3-bit quantizer to quantize the received channel data stream. A Viterbi decoder with soft decision data inputs quantized to three or four bits of precision can perform about 2 dB better than one working with hard decision inputs. The usual quantization precision is three bits as more bits provide little additional improvement. The quantized number is represented in 2's complement giving it a range of -4 to 3.

Soft decision Viterbi decoding offers better performance results than hard decision Viterbi decoding since it provides a better estimate of the noise, i.e., less quantization noise is introduced.

### 3.3.2 Minimum Euclidean Distance

Euclidean distance is the basic measurement used for calculating metrics when we use the soft decision Viterbi decoder. The squared Euclidean distance (ED) is measured by the following formula:
\[ ED(n,i) = \sum_{allk} [S_k(n) - G_k(n)]^2 \]  

(1)

where,

\( n \) = current state
\( i \) = input bit
\( k \) = encoded bits associated with a given input
\( S \) = received quantized bits
\( G \) = output bits

The hard decision Viterbi decoder is implemented by using the maximum Hamming distance for calculating the metrics. However, the soft decision Viterbi decoder is implemented using the minimum Euclidean distance for calculating the metrics.

### 3.3.3 Trellis Explanation

The working of the soft decision Viterbi decoder follows almost the same principles as that of the hard decision Viterbi decoder. The only difference is the calculation of the path metrics. Shown in the figures are the trellises' at each time tick, based on the example considered, of the encoded 3-bit input stream [1 0 1]. Since the working mechanism is the same as that of the hard decision Viterbi decoder, the explanation of the trellises is analogous to that discussed in Section 3.2.2.

The corrupted, and in this case quantized too, data bit stream at the input of the soft decision Viterbi decoder is assumed as [3-4 -43 33 -43 -4-4].
Step 1: At time $t = 0$

Step 2: At time $t = 1$
Step 3: At time $t = 2$

Figure 3.14. Step 3a: Trellis and path metrics at time $t = 2$.

Figure 3.15. Step 3b: Updated trellis and path metrics at time $t = 2$. 

Step 4: At time $t = 3$

![Trellis diagram](image)

**Figure 3.16.** Step 4a: Trellis and path metrics at time $t = 3$.

![Trellis diagram](image)

**Figure 3.17.** Step 4b: Updated trellis and path metrics at time $t = 3$. 
**Step 5:** At time $t = 4$

<table>
<thead>
<tr>
<th>Step</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 -4</td>
<td>11 (29)</td>
<td>11 (58)</td>
<td>11 (66)</td>
<td>11 (95)</td>
</tr>
<tr>
<td>-4 3</td>
<td>11 (11) (58)</td>
<td>11 (66)</td>
<td>11 (95)</td>
<td>11 (145)</td>
</tr>
<tr>
<td>3 3</td>
<td>11 (11) (113)</td>
<td>11 (11) (113)</td>
<td>11 (11) (113)</td>
<td>11 (11) (113)</td>
</tr>
<tr>
<td>-4 3</td>
<td>11 (11) (113)</td>
<td>11 (11) (113)</td>
<td>11 (11) (113)</td>
<td>11 (11) (113)</td>
</tr>
<tr>
<td>-4 -4</td>
<td>11 (11) (113)</td>
<td>11 (11) (113)</td>
<td>11 (11) (113)</td>
<td>11 (11) (113)</td>
</tr>
</tbody>
</table>

**Figure 3.18.** Step 5a: Trellis and path metrics at time $t = 4$.

<table>
<thead>
<tr>
<th>Step</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 -4</td>
<td>11 (29)</td>
<td>11 (58)</td>
<td>11 (66)</td>
<td>11 (95)</td>
</tr>
<tr>
<td>-4 3</td>
<td>11 (11) (58)</td>
<td>11 (66)</td>
<td>11 (95)</td>
<td>11 (145)</td>
</tr>
<tr>
<td>3 3</td>
<td>11 (11) (91)</td>
<td>11 (95)</td>
<td>11 (145)</td>
<td>145, 77</td>
</tr>
<tr>
<td>-4 3</td>
<td>11 (11) (91)</td>
<td>11 (95)</td>
<td>11 (145)</td>
<td>145, 77</td>
</tr>
<tr>
<td>-4 -4</td>
<td>11 (11) (91)</td>
<td>11 (95)</td>
<td>11 (145)</td>
<td>145, 77</td>
</tr>
</tbody>
</table>

**Figure 3.19.** Step 5b: Updated trellis and path metrics at time $t = 4$. 
The path with the lowest metric is looked for and a winner path is traced. The path traced by the states 00, 10, 01, 10, 01, 00 and corresponding to the bits 10100 is the decoded sequence and is as shown in Figure 3.20:

![Diagram](image-url)

**Figure 3.20.** Decoded sequence 10100 for the noisy encoded bit stream 3-4 –43 33 –43 -4 -4.

Thus, we see how the soft decision Viterbi decoder, using minimum Euclidean distances, works and achieves the decoded data bit stream, from a convolutionally encoded input data bit stream transmitted over an AWGN channel from the transmitter. This process is actually shown in the Matlab codes written which are as explained in the next section.
3.3.4 Soft Decision Decoder – MATLAB Implementation

The working of the soft decision Viterbi decoder was as demonstrated in Section 3.3.3.

This procedure was implemented using Matlab. The Matlab codes written for the FEC technique are attached in Appendix A. The main code Forward_Error_Correction_technique.m calls the functions which perform the processes of encoding and decoding. This section discusses the functions written that performs soft decision Viterbi decoding. The main function we wrote was soft_decision_Viterbi and is as described.

As seen from the previous sections, since the working of the soft decision Viterbi decoder is analogous to that of the hard decision Viterbi decoder, the only difference seen in the function soft_decision_Viterbi is as explained:

(i) - (iv) refer to Section 3.2.3.

(v) for each time interval:

(a) from the matrices output and transition, the corresponding branch metrics are calculated.

(b) the corresponding path metrics are then calculated from the matrix trellis and branch metrics.

(c) the matrices trellis and path are updated corresponding to the path with the minimum Euclidean distance.

(vi) refer to Section 3.2.3.

(vii) generates the output of the soft decision Viterbi decoder, which in our example is the codeword [1 0 1 0 0].
The practical working of the soft decision Viterbi decoder using the functions just discussed is as shown in Appendix B. Thus as seen, is the working of a soft decision Viterbi decoder, both theoretically and practically.

The output of the soft decision Viterbi decoder is as shown in Figure 3.21:

![Diagram of the output of the soft decision Viterbi decoder](image)

**Figure 3.21.** Output of the soft decision Viterbi decoder – 10100.

Thus, demonstrated is the working of the soft decision Viterbi decoder and we see how the input data bit stream [1 0 1] is obtained.

### 3.3.5 Maximum Euclidean Distance

Viterbi decoding is most often referred to as the maximum likelihood technique of decoding. The soft decision Viterbi decoder however uses minimum Euclidean distances to calculate path metrics.
To obtain maximum Euclidean distances we simplify the formula used to calculate the Euclidean distance. We have:

\[ ED(n, i) = \sum_{allk} [S_k^i(n) - G_k^i(n)]^2 \quad \text{from (1)} \]

\[ i.e. ED(n, i) = \sum_{allk} [S_k^2(n) - 2S_k(n)G_k(n) + G_k^2(n)] \quad (2) \]

Noting that \( S_k^2(n) \) and \( G_k^2(n) \) are constants in a given symbol period, these terms can be ignored since Euclidean distances are concerned with finding the minimum distances. We then have:

\[ \text{metric} = \sum_{allk} [-2S_k(n)G_k(n)] \quad (3) \]

Next the -2 is taken out which implies that instead of finding the path with the minimum Euclidean distance, we would look for the path with the maximum Euclidean distance. We have:

\[ \text{metric} = \sum_{allk} [S_k(n)G_k(n)] \quad (4) \]

Thus, by simplifying the formula for calculating the minimum Euclidean distance, the soft decision Viterbi decoder can also be referred to as a maximum likelihood technique of decoding.

The trellis of a soft decision Viterbi decoder using maximum Euclidean distances is as shown in Figure 3.22. The explanation of its working is analogous to that of the soft
decision Viterbi decoder discussed in Section 3.3.3 and that of the hard decision Viterbi decoder discussed in Section 3.2.2.

![Diagram](image)

**Figure 3.22.** Decoded sequence 10100 for the noisy encoded bit stream 3-4 -43 33 -43 -4-4 (maximum likelihood).

Thus we see how the soft decision Viterbi decoder, using maximum Euclidean distances, works and achieves the decoded data bit stream, from a convolutionally encoded input data bit stream transmitted over an AWGN channel from the transmitter.

The practical working of the soft-decision Viterbi decoder using the maximum likelihood is as shown in Appendix B. Thus as seen, is the working of a maximum-likelihood soft-decision Viterbi decoder, both theoretically and practically.
The corresponding output of the soft output Viterbi decoder using maximum Euclidean distances is as shown in Figure 3.23:

![Viterbi Decoder Diagram](image.png)

**Figure 3.23.** Output of the soft decision Viterbi decoder - 10100 (maximum likelihood).

Thus, demonstrated is the working of the soft decision Viterbi decoder with maximum likelihood and we see how the input data bit stream [1 0 1] is obtained.
Chapter 4: Performance Factors

This section discusses the performance of the convolutional encoding/Viterbi decoding forward error correction technique. These performance factors help to study how the forward error correction technique can be applied to daily applications.

The following performance factors are considered:

(i) Encoder Memory Size

(ii) Signal to Noise Ratio (SNR)

4.1 Encoder Memory Size

The forward error correction (FEC) technique makes use of a convolutional encoder. This encoder has memory to store the previous bits/state information. The memory size of an encoder is the number of bits/states that can be stored in the encoder. The FEC technique implemented in this thesis uses memory sizes of 2 and 6.

For the convolutional encoder, for larger memory size, the FEC technique has better performance as the coding algorithm becomes more sophisticated. However, the complexity of the decoder is exponentially increased. This is due to the fact that the number of states $n$ is exponentially related to the encoder memory size $m$. Thus, the size of the trellis formed is also exponentially related to the encoder memory size. This results in the decoding time to increase significantly with the memory size. Thus, the encoder memory size is kept in smaller values typically between 2 and 6.
4.2 Signal to Noise Ratio (SNR)

The most direct factor to affect the performance of the FEC technique is the signal to noise ratio.

For a fixed received signal power, noise level can be represented by the signal energy per bit to the noise power spectral density ($E_b/N_0$). As $E_b/N_0$ increases, the noise level decreases. Thus, the performance of the FEC technique improves or the bit error rate (BER) decreases as $E_b/N_0$ increases.

To analyze how the noise level affects the performance of the FEC technique, a number of simulation tests were conducted using the Matlab codes attached in Appendix A. The number of input bits considered was 10000. These input bits were encoded using a rate 1/2 and a rate 1/3 convolutional encoder. The encoder was also tested for values of $m = 2$ and $m = 6$ for both the code rates.

As mentioned before, generator polynomials characterize the encoder connections. Also, good generator polynomials are found by trial and error through computer simulations. The performance of our FEC technique is analyzed by two such sets of generator polynomials. The first set considered is a set of standard generator polynomials listed in [1]. The second set is another set, found and tested by us, which produces results that closely match those obtained with the set of standard generator polynomials used. These polynomials are listed in our Matlab codes attached in Appendix A.

On the decoding side, the corrupted encoded data bit stream was decoded using both a hard decision and a soft decision Viterbi decoder. Both these decoders were used, with each of the convolutional encoders considered, to study the performance of the FEC
technique. The simulation plots are as shown and the corresponding values obtained for the bit error rate are attached in Appendix C. This value was calculated by dividing the number of bit errors at the output of the decoder with the number of bits transmitted at the encoder side. Tables 4.9 through 4.12 at the end of this section summarize the noise level characteristics of the FEC technique. This result appears to agree well with those obtained by others [1].

The term *coding gain* refers to the reduction, usually expressed in decibels (dB), in the required $E_b/N_0$ to achieve a specific bit error rate of the coded system over an uncoded system with the same modulation and channel characteristics.
Figure 4.1. Plot of BER vs. $E_b/N_0$ for a rate $1/2$, $m = 2$ encoder, considering a hard decision and a soft decision Viterbi decoder (using standard polynomials).

Figure 4.2. Plot of BER vs. $E_b/N_0$ for a rate $1/2$, $m = 2$ encoder, considering a hard decision and a soft decision Viterbi decoder (using tested polynomials).
Figures 4.1 and 4.2 compare the performance of a hard decision Viterbi decoder and a soft decision Viterbi decoder with an encoder having a code rate of 1/2 and memory $m = 2$. Table 4.1 lists the SNR ($E_b/N_o$) in dB achieved with our Matlab implementation using both, the set of standard and the set of tested polynomials. Table 4.2 lists the coding gain in dB corresponding to Table 4.1. Thus from our results obtained, we see how the performance of the FEC technique improves using a soft decision Viterbi decoder.
<table>
<thead>
<tr>
<th>Bit Error Rate (BER)</th>
<th>Uncoded BPSK</th>
<th>Generator Polynomial Used</th>
<th>Decoder Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>4.2</td>
<td>4.2</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>3.3</td>
<td>3.2</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>5.8</td>
<td>5.7</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>5.0</td>
<td>4.8</td>
</tr>
</tbody>
</table>

Table 4.1. SNR ($E_b/N_o$) in dB for the rate 1/2, $m = 2$ encoder.

<table>
<thead>
<tr>
<th>Bit Error Rate (BER)</th>
<th>Uncoded BPSK</th>
<th>Generator Polynomial Used</th>
<th>Decoder Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>1.1</td>
<td>1.2</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>1.0</td>
<td>1.1</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>1.8</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Table 4.2. Coding gain in dB for the rate 1/2, $m = 2$ encoder.
Figure 4.3. Plot of BER vs. $E_b/N_0$ for a rate $1/2$, $m = 6$ encoder, considering a hard decision and a soft decision Viterbi decoder (using standard polynomials).

Figure 4.4. Plot of BER vs. $E_b/N_0$ for a rate $1/2$, $m = 6$ encoder, considering a hard decision and a soft decision Viterbi decoder (using tested polynomials).
Figures 4.3 and 4.4 compare the performance of a hard-decision Viterbi decoder and a soft-decision Viterbi decoder with an encoder having a code rate of 1/2 and memory $m = 6$. Table 4.3 lists the SNR ($E_b/\overline{N}_o$) in dB achieved with our Matlab implementation using both, the set of standard and the set of tested polynomials. Table 4.4 lists the coding gain in dB corresponding to Table 4.3. Thus from our results obtained, we see how the performance of the FEC technique improves using a soft decision Viterbi decoder. Also, when compared with Figures/Tables 4.1 and 4.2 it is seen that the performance of the FEC technique further improves with an encoder having a greater number of memory registers, i.e., a larger value for $m$. 
<table>
<thead>
<tr>
<th>Bit Error Rate (BER)</th>
<th>Uncoded BPSK</th>
<th>Generator Polynomial Used</th>
<th>Decoder Used</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Standard</td>
<td>Tested</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>3.7</td>
<td>3.5</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>2.2</td>
<td>2.3</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>4.7</td>
<td>4.4</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>3.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

**Table 4.3.** SNR ($E_b/N_o$) in dB for the rate 1/2, $m = 6$ encoder.

<table>
<thead>
<tr>
<th>Bit Error Rate (BER)</th>
<th>Uncoded BPSK</th>
<th>Generator Polynomial Used</th>
<th>Decoder Used</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Standard</td>
<td>Tested</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>0.7</td>
<td>0.9</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>2.2</td>
<td>2.1</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>2.1</td>
<td>2.4</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>3.8</td>
<td>3.8</td>
</tr>
</tbody>
</table>

**Table 4.4.** Coding gain in dB for the rate 1/2, $m = 6$ encoder.
Figure 4.5. Plot of BER vs. $E_b/N_0$ for a rate 1/3, $m = 2$ encoder, considering a hard decision and a soft decision Viterbi decoder (using standard polynomials).

Figure 4.6. Plot of BER vs. $E_b/N_0$ for a rate 1/3, $m = 2$ encoder, considering a hard decision and a soft decision Viterbi decoder (using tested polynomials).
Figures 4.5 and 4.6 compare the performance of a hard decision Viterbi decoder and a soft decision Viterbi decoder with an encoder having a code rate of 1/3 and memory $m = 2$. Table 4.5 lists the SNR $(E_b/N_o)$ in dB achieved with our Matlab implementation using both, the set of standard and the set of tested polynomials. Table 4.6 lists the coding gain in dB corresponding to Table 4.5. Thus from our results obtained, we see how the performance of the FEC technique improves using a soft decision Viterbi decoder. Also, when compared with Figures/Tables 4.1 and 4.2 it is seen that the performance of the FEC technique further improves with an encoder having a smaller code rate of 1/3.
### Table 4.5. SNR ($E_b/N_o$) in dB for the rate 1/3, $m = 2$ encoder.

<table>
<thead>
<tr>
<th>Bit Error Rate (BER)</th>
<th>Uncoded BPSK</th>
<th>Generator Polynomial Used</th>
<th>Decoder Used</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Standard</td>
<td>Tested</td>
<td></td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>4.0</td>
<td>4.0</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>3.0</td>
<td>3.2</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>5.6</td>
<td>5.4</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>4.7</td>
<td>4.6</td>
</tr>
</tbody>
</table>

### Table 4.6. Coding gain in dB for the rate 1/3, $m = 2$ encoder.

<table>
<thead>
<tr>
<th>Bit Error Rate (BER)</th>
<th>Uncoded BPSK</th>
<th>Generator Polynomial Used</th>
<th>Decoder Used</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Standard</td>
<td>Tested</td>
<td></td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>1.4</td>
<td>1.2</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>1.2</td>
<td>1.4</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>2.1</td>
<td>2.2</td>
</tr>
</tbody>
</table>
Figure 4.7. Plot of BER vs. $E_b/N_0$ for a rate $1/3$, $m = 6$ encoder, considering a hard decision and a soft decision Viterbi decoder (using standard polynomials).

Figure 4.8. Plot of BER vs. $E_b/N_0$ for a rate $1/3$, $m = 6$ encoder, considering a hard decision and a soft decision Viterbi decoder (using tested polynomials).
Figures 4.7 and 4.8 compare the performance of a hard decision Viterbi decoder and a soft decision Viterbi decoder with an encoder having a code rate of 1/3 and memory $m = 6$. Table 4.7 lists the SNR ($E_b/N_0$) in dB achieved with our Matlab implementation using both, the set of standard and the set of tested polynomials. Table 4.8 lists the coding gain in dB corresponding to Table 4.7. Thus from our results obtained, we see how the performance of the FEC technique improves using a soft decision Viterbi decoder. Also, when compared with Figures/Tables 4.3 and 4.4 and Figures/Tables 4.5 and 4.6 it is seen that the performance of the FEC technique further improves with an encoder having a smaller code rate of 1/3 and an encoder having a greater number of memory registers, i.e., a larger value for $m$, respectively.
<table>
<thead>
<tr>
<th>Bit Error Rate (BER)</th>
<th>Uncoded BPSK</th>
<th>Generator Polynomial Used</th>
<th>Decoder Used</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Standard</td>
<td>Tested</td>
<td></td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>3.1</td>
<td>3.0</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>3.9</td>
<td>4.2</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>2.8</td>
<td>2.5</td>
</tr>
</tbody>
</table>

**Table 4.7.** SNR ($E_b/N_0$) in dB for the rate 1/3, $m = 6$ encoder.

<table>
<thead>
<tr>
<th>Bit Error Rate (BER)</th>
<th>Uncoded BPSK</th>
<th>Generator Polynomial Used</th>
<th>Decoder Used</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Standard</td>
<td>Tested</td>
<td></td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>1.3</td>
<td>1.4</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>2.9</td>
<td>2.6</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>4.0</td>
<td>4.3</td>
</tr>
</tbody>
</table>

**Table 4.8.** Coding gain in dB for the rate 1/3, $m = 6$ encoder.
Figure 4.9. Plot of BER vs. $E_b/N_0$ for various combinations of the encoder, with a hard decision decoder (using the standard polynomials).

Figure 4.10. Plot of BER vs. $E_b/N_0$ for various combinations of the encoder, with a hard decision decoder (using tested polynomials).
Figures 4.9 and 4.10 compare the performance of a hard decision Viterbi decoder for various combinations of the encoder. As can be seen from the figure, the performance of a rate 1/2 and a rate 1/3 convolutional encoder closely match each other. However, when $m$ is set to 6 the performance of the FEC technique is improved not only by a larger value for $m$, but also for the encoder with a code rate of 1/3.
Figure 4.11. Plot of BER vs. $E_b/N_0$ for various combinations of the encoder, with a soft decision decoder (using standard polynomials).

Figure 4.12. Plot of BER vs. $E_b/N_0$ for various combinations of the encoder, with a soft decision decoder (using tested polynomials).
Figures 4.11 and 4.12 compare the performance of a soft decision Viterbi decoder for various combinations of the encoder. As can be seen from the figure, the performance of a rate 1/2 and a rate 1/3 convolutional encoder closely match each other. However, when $m$ is set to 6 the performance of the FEC technique is improved not only by a larger value for $m$, but also for the encoder with a code rate of 1/3. Also, when compared with Figures 4.9 and 4.10, it is seen that the performance of the FEC technique significantly improves using a soft decision Viterbi decoder.

Tables 4.9 through 4.12 summarize all the results we achieved with the Matlab implementation of the FEC technique over a Gaussian channel. These tables are similar to the tables already discussed in this section. They allow an easy comparison between our different implementations of the encoder and decoder. Tables 4.9 and 4.11 list the SNR ($E_b/N_0$) in dB using the set of standard and tested generator polynomials, respectively. Tables 4.10 and 4.12 list the coding gains in dB corresponding to Tables 4.9 and 4.11.
<table>
<thead>
<tr>
<th>Bit Error Rate (BER)</th>
<th>Uncoded BPSK</th>
<th>Rate 1/2</th>
<th>Rate 1/3</th>
<th>Decoder Used</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$m = 2$</td>
<td>$m = 6$</td>
<td>$m = 2$</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>4.2</td>
<td>3.7</td>
<td>4.0</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>3.3</td>
<td>2.2</td>
<td>3.0</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>5.8</td>
<td>4.7</td>
<td>5.6</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>5.0</td>
<td>3.0</td>
<td>4.7</td>
</tr>
</tbody>
</table>

**Table 4.9.** SNR ($E_b/N_0$) in dB using the standard polynomials.

<table>
<thead>
<tr>
<th>Bit Error Rate (BER)</th>
<th>Uncoded BPSK</th>
<th>Rate 1/2</th>
<th>Rate 1/3</th>
<th>Decoder Used</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$m = 2$</td>
<td>$m = 6$</td>
<td>$m = 2$</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>0.2</td>
<td>0.7</td>
<td>0.4</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>1.1</td>
<td>2.2</td>
<td>1.4</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>1.0</td>
<td>2.1</td>
<td>1.2</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>1.8</td>
<td>3.8</td>
<td>2.1</td>
</tr>
</tbody>
</table>

**Table 4.10.** Coding gain in dB using the standard polynomials.
<table>
<thead>
<tr>
<th>Bit Error Rate (BER)</th>
<th>Uncoded BPSK</th>
<th>Rate 1/2</th>
<th>Rate 1/3</th>
<th>Decoder Used</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$m = 2$</td>
<td>$m = 6$</td>
<td>$m = 2$</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>4.2</td>
<td>3.5</td>
<td>4.0</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>3.2</td>
<td>2.3</td>
<td>3.2</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>5.7</td>
<td>4.4</td>
<td>5.4</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>4.8</td>
<td>3.0</td>
<td>4.6</td>
</tr>
</tbody>
</table>

**Table 4.11.** SNR ($E_b/N_0$) in dB using the tested polynomials.

<table>
<thead>
<tr>
<th>Bit Error Rate (BER)</th>
<th>Uncoded BPSK</th>
<th>Rate 1/2</th>
<th>Rate 1/3</th>
<th>Decoder Used</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$m = 2$</td>
<td>$m = 6$</td>
<td>$m = 2$</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>0.2</td>
<td>0.9</td>
<td>0.4</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>4.4</td>
<td>1.2</td>
<td>2.1</td>
<td>1.2</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>1.1</td>
<td>2.4</td>
<td>1.4</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.8</td>
<td>2.0</td>
<td>3.8</td>
<td>2.2</td>
</tr>
</tbody>
</table>

**Table 4.12.** Coding gain in dB using the tested polynomials.
Chapter 5: Conclusion and Future Work

5.1 Conclusions

The forward error correction technique (FEC) technique is a technique, particularly suited for a white Gaussian noise (AWGN) channel, which has been implemented using convolutional encoding with Viterbi decoding.

The encoding process was demonstrated using a (2, 1, 3) convolutional encoder. A 3-bit input stream was encoded as an example to show the working of this encoder. This process was shown and verified both theoretically, as mentioned in Chapter 2, and practically, as mentioned in Appendix B.

The decoding process was demonstrated using a hard decision Viterbi decoder and a soft decision Viterbi decoder. The noise corrupted and encoded 3-bit input sequence, was decoded and recovered using the concepts of Hamming distances and Euclidean distances for the hard decision Viterbi decoder and soft decision Viterbi decoder respectively. This process was also shown and verified both theoretically as, mentioned in Chapter 3, and practically as mentioned in Appendix B.

Performance factors affecting the FEC technique were discussed. These included the encoder memory size and the more significant factor of SNR.

To study how these performance factors affect the working of the FEC technique, the encoder for the FEC technique was considered for different values of code rates and the number of memory registers. The code rates considered were 1/2 and 1/3 and the number of memory registers considered were 2 and 6. These code rates were used with
each of the two values of memory. The decoder for the FEC technique was then tested with both a hard decision Viterbi decoder and a soft decision Viterbi decoder, for each one of the combinations considered at the encoder. These combinations included both a standard set of generator polynomials as listed in [1] and another set, which we found, by a trial and error method through computer simulations. This set of generator polynomials produced results which matched very closely with those obtained from the set of standard generator polynomials.

As was seen from the simulation results obtained, the performance of the FEC technique was greatly improved using:

(i) a convolutional encoder with a larger encoder memory size

(ii) a convolutional encoder with a smaller code rate

(iii) a soft decision Viterbi decoder.

Thus we see that the performance of the FEC technique improves for an encoder having a code rate of 1/3 and an increased number of memory registers. Also, a soft decision Viterbi decoder, which makes use of more precision in quantization, is a more powerful decoder than a hard decision Viterbi decoder and significantly improves the performance of the FEC technique.

5.2 Future Work

For years, convolutional encoding with Viterbi decoding has been the predominant FEC technique used in space applications, particularly in geostationary satellite communication networks, such as VSAT (very small aperture terminal) networks [3]. These networks most commonly use a rate 1/2 convolutional encoder with memory
size of 6 and can transmit signals with at least 5 dB less power. This makes it useful in reducing transmitter and/or antenna cost or permitting increased data rates given the same transmitter power and antenna sizes.

If we further supplement the FEC technique with Reed-Solomon coding in the geostationary satellite communication arena, one can transmit signals with even lower power. The two coding techniques are implemented as serially concatenated block and convolutional coding, i.e., concatenated Reed-Solomon coding and convolutional coding with Viterbi decoding. Typically, the information to be transmitted is first encoded with the Reed-Solomon code, then with the convolutional code. On the receiving end, Viterbi decoding is performed first, followed by Reed-Solomon decoding. This is the technique that is used in most if not all of the direct broadcast satellite (DBS) systems, and in several of the newer VSAT products as well.

The FEC technique can be used to implement Turbo codes, codes which are capable of achieving near Shannon capacity performance [1]. The encoding scheme uses simple convolutional codes separated by interleaving stages to produce generally low rate block codes. The decoding is done by decoding the convolutional encoder separately using soft output Viterbi decoding and sharing bit reliability in an iterative manner.
References


Appendices

Appendix A: Matlab Programs

Appendix A lists the Matlab codes written by us to demonstrate the working of the FEC coding technique. These codes also help study the performance factors that characterize the FEC technique.

- Main Matlab code – Forward Error Correction Technique.m

```matlab
% ********************************************
% * Forward Error Correction Technique     *
% * ------------------------------------   *
% * (main code)                             *
% ********************************************

clear all
clc
close all
iterat = 3 % number of trials
N = 10000 % number of input bits
Eb_No = [2:1:7]; % signal to noise ratio
LE = length(Eb_No);

for kk = 1:iterat
    d0 = rand(1,N); % Generation of random input bits of length N used for testing
    d0(find(d0 >= 0.5)) = 1; % Conversion to orthogonal signal elements
    d0(find(d0 < 0.5)) = 0;

% -------------------------------------------------------------------------------------------------
% * Standard generator polynomials for which the FEC technique is implemented *
% -------------------------------------------------------------------------------------------------

% g = [1,1,1;0,1] % generator polynomial for rate 1/2 m = 2
% convolutional encoder
```
% g = [1,0,0,1,1,1,1,0,1,0,1,0,1] % generator polynomial for rate 1/2 m = 6
  % convolutional encoder
% g = [1,1,1,1,1,1;1,1,0,1,1,0] % generator polynomial for rate 1/3 m = 2
  % convolutional encoder
% g = [1,0,0,1,1,1,1;1,0,1,1,0,1,1;1,1,0,1,1,0,1] % generator polynomial for rate 1/3
  % m = 6 convolutional encoder

% * Tested generator polynomials for which the FEC technique is implemented *

% g = [1,0,1;1,1,1]           % generator polynomial for rate 1/2 m = 2 convolutional encoder
% g = [1,0,1,1,0,1;1,1,1,1,0,0,1] % generator polynomial for rate 1/2 m = 6
  % convolutional encoder
% g = [1,0,1; 0,1,0; 1,1,0]                % generator polynomial for rate 1/3 m = 2
  % convolutional encoder
% g = [1,0,1,1,0,1,1;1,1,1,1,0,0,1;1,1,0,0,1,0,1]  % generator polynomial for rate 1/3
  % m = 6 convolutional encoder

% Convolutional Encoding *
% ---------------------------------
% - Start -

% refer to hard_decision_Viterbi.m and soft_decision_Viterbi.m

% - End -

antipodal_conv(find(antipodal_conv >= 1)) = 1; % conversion of encoded bits to
  % antipodal signal
antipodal_conv(find(antipodal_conv <= 0)) = -1;
antipodal_conv;

Ltx = length(antipodal_conv);

Eb_No;
Es_No = Eb_No + 10 * log10 (1 / 2);   % Es_No calculated depending on the
  % convolutional encoder used
No = 1 ./ (10 .^ (Es_No ./ 10));

for i = 1:LE
  i;
  % -------------------------------------
  % * Generating AWGN channel *
  % -------------------------------------
% - Start -

No(i);
nnoise = sqrt(No(i) * 0.5) * randn(1, Ltx);

% - End -

rec_bits = antipodal_conv + noise(1:length(antipodal_conv)); % Addition of noise to transmitted bits

% -------------------------
% * Viterbi Decoding *
% -------------------------
%
% refer to hard_decision_Viterbi.m and soft_decision_Viterbi.m
% the corresponding lines of code are shown and the function calls, depending
% on the decoding process used
%
% - End -

% ------------------------------------
% * Bit Error Rate Calculation *
% ------------------------------------
%
% refer to hard_decision_Viterbi.m and soft_decision_Viterbi.m
% the corresponding lines of code are shown and the function calls, depending
% on the decoding process used
%
% - End -

% ------------------------------------
% * Analytical Calculations *
% ------------------------------------
%
Ebn_db = [0:1:10];
EBp = 0:0.1:max(Ebn_db);
EBpn = 10. ^ (EBp / 10); % Numeric value of Eb/No
BER = Qf(sqrt(2 * EBpn));

% - End -
% ----------------------------
% * Plotting Commands *
% ----------------------------

% - Start -

figure(1)
semilogy (EBp, BER, 'k-', Eb_No, avg_ber, 'r^-', 'Linewidth', 2)
title (' Plot of BER vs. Eb / No ')
xlabel (' Eb / No ')
ylabel (' Bit Error Rate (BER) ')
legend (' Uncoded BPSK ', ' code rate = 1/2 or 1/3, m = 2 or 6, hard or soft decision 
        (standard or tested ) ')
axis ([0 max(Ebn_db) 1E-6 1])
grid

% - End -

- hard_decision_Viterbi.m

% ***************************************************************
% * Hard - Decision Viterbi Decoder with Convolutional Encoding                    *
% * ---------------------------------------------------------------------------                     *
% ***************************************************************

clear all
cle
close all
iterat = 3;                        % number of trials
N = 10000                      % number of input bits
Eb_No = [2:1:7];            % noise level
LE = length(Eb_No);

for kk = 1:iterat
    kk

d0 = rand(1,N);                  % Generation of random input bits of length N used for testing the 
                                    % FEC technique practically
    d0(find(d0 >= 0.5)) = 1;
d0(find(d0 < 0.5)) = 0;        % Conversion to orthogonal signal elements
d0;

    % ***************************************************************
    % * Standard generator polynomials for which the FEC technique is implemented *
    % ***************************************************************
% g = [1,1,1;1,0,1]  % generator polynomial for rate 1/2 m = 2 convolutional encoder
% g = [1,0,0,1,1,1,1,0,1,0,1,1]  % generator polynomial for rate 1/2 m = 6 convolutional encoder
% g = [1,1,1;1,1,1,1,0,1,0,1]  % generator polynomial for rate 1/3 m = 2 convolutional encoder
% g = [1,0,0,1,1,1,1,0,1,0,1,1,1,1,0,1,0,1]  % generator polynomial for rate 1/3 m = 6 convolutional encoder

% -----------------------------------------------------------------------------------------------
% * Tested generator polynomials for which the FEC technique is implemented *
% *-----------------------------------------------------------------------------------------------

% g = [1,1,1;1,0,1]  % generator polynomial for rate 1/2 m = 2 convolutional encoder
% g = [1,0,1,1,0,1,1;1,1,1,1,0,0,1]  % generator polynomial for rate 1/2 m = 6 convolutional encoder
% g = [1,0,1; 0,1,0; 1,1,0]  % generator polynomial for rate 1/3 m = 2 convolutional encoder
% g = [1,0,1,1,0,1,1,1,1,1,0,0,1;1,1,0,0,1,0,1]  % generator polynomial for rate 1/3 m = 6 convolutional encoder

% *-----------------------------------------------------------------------------------------------
% * Convolutional Encoding *
% *-----------------------------------------------------------------------------------------------

% antipodal_conv = encode_block(g,d0);  % output from the convolutional encoder used
% - Start -

antipodal_conv(find(antipodal_conv >= 1)) = 1;  % conversion of encoded bits to antipodal signal
antipodal_conv(find(antipodal_conv <= 0)) = -1;
antipodal_conv;

Ltx = length(antipodal_conv);

Eb_No;
Es_No = Eb_No + 10 * log10 (1 / 2);  % Es_No calculated depending on the convolutional encoder used
No = 1 ./ (10 .^ (Es_No ./ 10));

for i = 1:LE
    i;
% * Generating AWGN channel *
% --------------------------------------
% - Start -

No(i);
noise = sqrt(No(i) * 0.5) * randn(1, Ltx);

% - End -

rec_bits = antipodal_conv + noise(1:length(antipodal_conv)); % Addition of noise % to transmitted bits

% ---------------------------------------------
% * Hard - Decision Viterbi Decoding *
% ---------------------------------------------
% - Start -

rec_bits(find(rec_bits <= 0)) = 0;
rec_bits(find(rec_bits > 0)) = 1;
rec_bits;

estimated_bits = hard_decode(g, rec_bits);

% - End -

% ---------------------------------------------
% * Bit Error Rate Calculation *
% ---------------------------------------------
% - Start -

out = d0 - estimated_bits;
ber(:,i) = sum(abs(out)) / (N);
end % LE
ber_it(kk,:) = ber;
end % kk
avg_ber = sum(ber_it,1) ./ (iterat)

% - End -

% * Analytical Calculations *
% ---------------------------------------------
% - Start -

Ebn_db = [0:1:10];
EBp = 0:0.1:max(Ebn_db);
EBpn = 10 .^ (EBp / 10); % Numeric value of Eb/No
BER = Qf(sqrt(2 * EBpn));

% - End -

% -----------------------------
% * Plotting Commands *
% -----------------------------
% - Start -

figure(1)
semilogy (EBp, BER, 'k-', Eb_No, avg_ber, 'r^-', 'Linewidth', 2)
title (' Plot of BER vs. Eb / No ')
xlabel (' Eb / No ')
ylabel (' Bit Error Rate (BER) ')
legend(' Uncoded BPSK ', ' code rate = 1/2 or 1/3 , m = 2 or 6, hard-decision (standard or tested) ')
axis([0 max(Ebn_db) 1E-6 1])
grid

% - End -

- soft_decision_Viterbi.m

% ********************************************************************************
% * Soft - Output Viterbi Decoder with Convolutional Encoding                     *
% * ********************************************************************************
% ********************************************************************************
clear all
clecloseclose all
iterat = 3 % number of trials
N = 10000 % number of input bits
Eb_No = [2:1:7];
LE = length(Eb_No);

for kk = 1:iterat
    kk;
    d0 = rand(1, N); % Generation of random input bits of length N used for testing the
                     % FEC technique practically
    d0(find(d0 >= 0.5)) = 1;
    d0(find(d0 < 0.5)) = 0; % Conversion to orthogonal signal elements
% * Standard generator polynomials for which the FEC technique is implemented *
% -------------------------------------------------------------------------------------------------
% g = [1,1,1;1,0,1]                           % generator polynomial for rate 1/2 m = 2
% convolutional encoder
% g = [1,0,0,1,1,1;1,0,1,1,0,1]   % generator polynomial for rate 1/2 m = 6
% convolutional encoder
% g = [1,1,1;1,1,1;1,0,1]                  % generator polynomial for rate 1/3 m = 2
% convolutional encoder
% g = [1,0,0,1,1,1;1,0,1,1,1;1,1,0,1,1,1;1,1,0,1,1,1]  % generator polynomial for rate 1/3
% m = 6 convolutional encoder
% -------------------------------------------------------------------------------------------------
% * Tested generator polynomials for which the FEC technique is implemented *
% -------------------------------------------------------------------------------------------------
% g = [1,1,1;1,0,1]           % generator polynomial for rate 1/2 m = 2 convolutional encoder
% g = [1,0,1,1,0,1,1;1,1,1,1,0,0,1]    % generator polynomial for rate 1/2 m = 6
% convolutional encoder
% g = [1,0,1; 0,1,0; 1,1,0]                  % generator polynomial for rate 1/3 m = 2
% convolutional encoder
% g = [1,0,1,0,1,1;1,1,1,0,1,1,1;1,1,0,1,0,1,1,1]  % generator polynomial for rate 1/3
% m = 6 convolutional encoder
% -------------------------------------------------------------------------------------------------
% * Convolutional Encoding *
% ----------------------------------
% - Start -
antipodal_conv = encode_block(g,d0);   % output from the convolutional encoder used
% - End -
antipodal_conv(find(antipodal_conv >= 1)) = 1;    % conversion of encoded bits to
antipodal_conv(find(antipodal_conv <= 0)) = -1;
antipodal_conv;
Ltx = length(antipodal_conv);
Eb_No;
Es_No = Eb_No + 10 * log10 (1 / 2);      % Es_No calculated depending on the
No = 1 ./ (10 .^ (Es_No ./ 10));
for i = 1:LE

% Generating AWGN channel

No(i);
noise = sqrt(No(i) * 0.5) * randn(1, Ltx);

rec_bits = antipodal_conv + noise(1:length(antipodal_conv)); % Addition of noise to transmitted bits

estimated_bits = soft_decode(g, rec_bits);

out = d0 - estimated_bits;
ber(:, i) = sum(abs(out)) / (N);
ber_it(kk, :) = ber;
avg_ber = sum(ber_it, 1) ./ (iterat)
Ebn_db = [0:1:10];
EBp = 0:0.1:max(Ebn_db);
EBpn = 10 .^ (EBp / 10);          % Numeric value of Eb/No
BER = Qf(sqrt(2 * EBpn));

% - Start -

figure (1)
semilogy (EBp, BER, 'k-', Eb_No, avg_ber, 'r^-', 'Linewidth', 2)
title (' Plot of BER vs. Eb / No ')
xlabel (' Eb / No ')
ylabel (' Bit Error Rate (BER) ')
legend (' Uncoded BPSK ', ' code rate = 1/2 or 1/3 , m = 2 or 6, soft-decision (standard or tested) ')
axis ([0 max(Ebn_db) 1E-6 1])
grid
% - End -

- encode_block.m

function [y] = encode_block (g, x)

% receives the generator polynomial and the input bit stream to be encoded
% zero pads the input bit stream
% calls the function encode_bit
% generates the output of the convolutional encoder

[n, K] = size(g); % g = 1 1 1 n = 2, K = 3
    % 1 0 1
m = K - 1;        % no. of states = 2^m = 2^2 = 4 i.e. (00, 01, 10, 11)

[temp, L_info] = size(x);
state = zeros(1, m); % initializes the state vector i.e. state = [0 0]
x = [x state]; % zero pads the input bit stream
L_total = L_info + m;
% generates the encoded output bit stream
for i = 1:L_total
    input_bit = x(1, i);
    [output_bits, state] = encode_bit (g, input_bit, state);
    y(n * (i-1) + 1:n * i) = output_bits;
end

y;                     % output of the convolutional encoder

- encode_bit.m

function [output, state] = encode_bit (g, input, state)

% receives the generator polynomial, the input bit to be encoded and input state bits
% generates the encoded output bits corresponding to the input bit and input state bits
% generates the next state bits corresponding to the input bit and input state bits

[n, k] = size(g);
m = k-1;

% determines the next output bit
for i = 1:n
    output(i) = g(i, 1) * input; % generates the first & second output bit
    for j = 2:k
        output(i) = xor (output(i), g(i, j) * state(j - 1));
    end;
end
state = [input, state(1:m - 1)];     % generates the next state corresponding to the input bit
                                      % and previous state

- hard_decode.m

% -------------------------------------------
% * Hard - Decision Viterbi Decoder *
% -------------------------------------------

function [x_hat] = hard_decode(g, r)

% receives the generator polynomial and the corrupted encoded data bit stream
% initializes and updates matrices for storing the maximum path metric (trellis)
% and the corresponding path (path)
% calls the functions bin_state, encode_bit, int_state, hard_dist
% generates the output of the hard-decision Viterbi decoder
% r = [0 1 1 0 0 0 1 0 1 1]    % used to verify the theoretical working of the (2, 1, 3)
% convolutional encoder

[n, K] = size(g);
m = K - 1;
max_state = 2 ^ m;               % number of states
[temp, rec_size] = size(r);
L_total = rec_size / n;          % number of input bits including tail bits
L_info = L_total - m;           % number of input bits excluding tail bits

% set infinity to an arbitrarily small value
inf = -10;

% initializes trellis and path matrices
trellis = inf * ones(max_state, L_total);
path = zeros(max_state, L_total);

% this matrix stores the maximum hamming distances as the trellis is formed
path = zeros(max_state, L_total);

% this matrix keeps a track of the paths traced as we receive the codeword
new_path = path;

% ----------------------------------------------------
% * Initializes output and transition matrices *
% ----------------------------------------------------
% - Start -
for state = 1:max_state
    state_vector = bin_state(state - 1, m);
    % generates the states i.e. state_vector = 00,01,10,11 when variable state = 1,2,3,4
    [out_0, state_0] = encode_bit(g, 0, state_vector);
    % determines o/p and next state bits given current state(state_vector) and input bit = 0
    [out_1, state_1] = encode_bit(g, 1, state_vector);
    % determines o/p and next state bits given current state(state_vector) and input bit = 1
    output(state, :) = [out_0 out_1];
    %  0    1    (received bits)
    % 00 00 11    (output bits given input states and received bits)
    % 01 11 00
    % 10 10 01
    % 11 01 10
    transition(state, :) = [(int_state(state_0) + 1) (int_state(state_1) + 1)];
    %  0    1    (received bits)
    % 00 1 3    (outputs states given input states and received bits)
    % 01 1 3
    % 10 2 4
    % 11 2 4
end

% - End -

% -----------------------------------------------------------
% * Determines trellis and path matrices at time 1 *
% -----------------------------------------------------------
% - Start -

y_segment = r(1, 1:n);
for i = 0:1
    hypothesis = output(1, n * i + 1:n * (i + 1));
    next_state = transition(1, i + 1);
    hamming_dist = hard_dist(hypothesis, y_segment);
    path_metric = hamming_dist;
    trellis(next_state, 1) = path_metric;
    path(next_state, 1) = i;
end

% - End -

% --------------------------------------------------------------------------------------
% * Now determines trellis and path matrices for times 2 through L_total *
% --------------------------------------------------------------------------------------
% - Start -

counter = n + 1;
for time = 2:L_total
    y_segment = r(1, counter:counter + n - 1);
    counter = counter + n;
    for state = 1:max_state
        for i = 0:1
            hypothesis = output(state, n * i + 1:n * (i + 1));
            % determines output bits depending on input bit(i) from output matrix
            next_state = transition(state, i + 1);
            % determines next state depending on input bit(i) from transition matrix
            hamming = hard_dist(hypothesis, y_segment);
            % computes Hamming Distance
            % path metric and trellis are updated
            path_metric = hamming + trellis(state, time - 1);
            if path_metric > trellis(next_state, time) % maximum hamming distance check
                trellis(next_state, time) = path_metric;
                % maximum path metric or hamming distance stored and updated in matrix
                % trellis
            end
        end
    end
end
new_path(next_state, 1:time ) = [path(state, 1:time - 1) i];
  % path corresponding to maximum path metric or hamming distance
  % stored and updated in matrix new_path
  end     % (if)
  end     % (i)
  end       % (state)
  path = new_path; % path is updated with each new state (formation of trellis)
  % with the help of matrix new_path
  end         % (time)

  % - End -

  % * To make sure first row of path holds the information bits *
  % *---------------------------------------------------------------------------
  % - Start -

  k2 = 1;
  for j = 2:max_state
    if trellis (j, L_total) > trellis (k2, L_total)
      k2 = j;
    end
  end
  trellis;
  path (1, 1:L_total) = path (k2, 1:L_total);
  path;

  % - End -

  x_hat = path(1, 1:L_info); % output of the hard-decision Viterbi decoder

  **soft_decode.m**

  % *---------------------------------------------------------------------------
  % * Soft - Decision Viterbi Decoder *
  % *---------------------------------------------------------------------------

  function [x_hat] = soft_decode (g, r)

  % receives the generator polynomial and the corrupted encoded data bit stream
  % initializes and updates matrices for storing the path metric (trellis)
  % and the corresponding path (path)
  % calls the functions bin_state, encode_bit, int_state
  % generates the output of the soft-decision Viterbi decoder
% r = [3 -4 -4 3 3 -4 3 -4 -4] % used to verify the theoretical working of the (2, 1, 3)
% convolutional encoder

[n, K] = size(g);
m = K - 1;
max_state = 2 ^ m; % number of states
[temp, rec_size] = size(r);
L_total = rec_size / n; % number of input bits including tail bits
L_info = L_total - m; % number of input bits excluding tail bits

% set infinity to an arbitrarily large value
inf = 10 ^ 5;

% initializes trellis and path matrices
trellis = inf * ones(max_state, L_total);
path = zeros(max_state, L_total);

% this matrix stores the minimum euclidean distances as the trellis is formed
path = zeros(max_state, L_total);

% this matrix keeps a track of the paths traced as we receive the codeword
new_path = path;

% -----------------------------------------------------
% * Initializes output and transition matrices *
% -----------------------------------------------------
% - Start -
for state = 1:max_state
    state_vector = bin_state(state - 1, m);
    % generates the states i.e. state_vector = 00,01,10,11 when variable state = 1,2,3,4
    [out_0, state_0] = encode_bit(g, 0, state_vector);
    % determines o/p and next state bits given current state(state_vector) and input bit = 0
    [out_1, state_1] = encode_bit(g, 1, state_vector);
    % determines o/p and next state bits given current state(state_vector) and input bit = 1
    output(state, :) = [out_0 out_1];
    % 0 1 (received bits)
    % 00 00 11 (output bits given input states and received bits)
    % 01 11 00
    % 10 10 01
    % 11 01 10
    transition(state, :) = [(int_state(state_0) + 1) (int_state(state_1) + 1)];
    % 0 1 (received bits)
    % 00 1 3 (outputs states given input states and received bits)
    % 01 2 4
    % 10 2 4
    % 11 2 4
end
% - End -

% -----------------------------------------------------------
% * Determines trellis and path matrices at time 1 *
% -----------------------------------------------------------
% - Start -

y_segment = r(1, 1:n);
for i = 0:1
    hypothesis = 2 * output(1, n * i + 1:n * (i + 1)) - 1;
    next_state = transition(1, i + 1);
    path_metric = 0;
    for j = 0:1
        path_metric = path_metric + (y_segment(1, j + 1) - hypothesis(1, j + 1)) ^ 2;
    end
    trellis(next_state, 1) = path_metric;
    path(next_state, 1) = i;
end

% - End -

flag = 0;

% -----------------------------------------------------------
% * Now determines trellis and path matrices for times 2 through L_total *
% -----------------------------------------------------------
% - Start -

counter = n + 1;
for time = 2:L_total
    y_segment = r(1, counter:counter + n - 1);
    counter = counter + n;
    for state = 1:max_state
        for i = 0:1
            hypothesis = 2 * output(state, n * i + 1:n * (i + 1)) - 1;
            next_state = transition( state, i + 1 );
            % determines output bits depending on input bit(i) from output matrix

            % computes squared Euclidian distance
            square_dist = 0;
            for j = 1:n
                if y_segment(j)
square_dist = square_dist + (y_segment(j) - hypothesis(j))^2; % Euclidean distance
end
end

% path metric and trellis are updated
path_metric = square_dist + trellis(state, time - 1);
if path_metric < trellis(next_state, time) % minimum Euclidean distance check
    trellis(next_state, time) = path_metric;
    % minimum path metric or Euclidean distance stored and updated in matrix trellis
    new_path(next_state, 1:time) = [path(state, 1:time - 1) i];
    % path corresponding to minimum path metric or Euclidean distance stored and updated in matrix new_path
end % (if)
end % (i)
end % (state)
path = new_path; % path is updated with each new state (formation of trellis) with the help of matrix new_path
end % (time)

% - End -

% ----------------------------------------------------------------------------------------------------------------------------------
% * To make sure first row of path holds the information bits *
% ----------------------------------------------------------------------------------------------------------------------------------
% - Start -

k2 = 1;
for j = 2:max_state
    if trellis(j, L_total) < trellis(k2, L_total)
        k2 = j;
    end
end

% trellis;
path(1, 1:L_total) = path(k2, 1:L_total);
path;

% - End -

x_hat = path(1,1:L_info); % output of the soft-decision Viterbi decoder
- **bin_state.m**

```matlab
function [hard_state] = bin_state (int_state, m)

% generates all the states i.e 00, 01, 10, 11

for i = m:-1:1
    state(m - i + 1) = fix(int_state / (2 ^ (i - 1)));
    int_state = int_state - state(m - i + 1) * 2 ^ (i - 1);
end

hard_state = state;
```

- **int_state.m**

```matlab
function [int_states] = int_state (state)

% generates an integer value for the binary denoted state
% i.e. 00=0, 01=1, 10=2, 11=3

[dummy, m] = size(state);

for i = 1:m
    vect(i) = 2 ^ (m - i);
end

state;

int_states = state * vect';    % by multiplying state and vect' we get an integer value
% for the binary denoted state i.e. 00=0, 01=1, 10=2, 11=3
```

- **hard_dist.m**

```matlab
function [y] = hard_dist (w,p)

% calculates the hamming distance

[r1, c1] = size(w);
[r2, c2] = size(p);
hd = 0;
if (c1 ~= c2), error('Matrix sizes do not match.'), end
for i = 1:c1
    if w(i) == p(i)
        hd = hd + 1;
    end
end
y = hd;
```
Appendix B: Practical working of a (2, 1, 3) convolutional encoder using the 3 input bits [1 0 1]

Appendix B shows the practical demonstration of the (2, 1, 3) convolutional encoder as explained in this thesis with the 3-input bits [1 0 1] using a hard decision and a soft decision Viterbi decoder.

- **Using a hard – decision Viterbi decoder**

iterat = 1

N = 3

kk = 1

d0 = 1 0 1

g = 1 1 1
    1 0 1

x = 1 0 1 0 0

y = 1 1 1 0 0 0 1 0 1 1

antipodal_conv = 1 1 1 0 0 0 1 0 1 1

antipodal_conv = 1 1 1 -1 -1 -1 -1 1 -1 1 1

r = 0 1 1 0 0 0 1 0 1 1

trellis =

-10 -10 -10 -10 -10
-10 -10 -10 -10 -10
-10 -10 -10 -10 -10
-10 -10 -10 -10 -10
-10 -10 -10 -10 -10

path =

0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
new_path =

0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

output =

0 0 1 1
1 1 0 0
1 0 0 1
0 1 1 0

transition =

1 3
1 3
2 4
2 4

trellis =

1 2 4 5 9
-10 3 3 7 6
1 2 5 5 7
-10 1 3 5 6

path =

1 0 1 0 0
0 0 0 1 0
0 0 0 0 1
0 0 0 1 1

path =

1 0 1 0 0
0 0 0 1 0
0 0 0 0 1
0 0 0 1 1

x_hat = 1 0 1

estimated_bits = 1 0 1
avg_ber = 0

- Using a soft – decision Viterbi decoder

iterat = 1

N = 3

kk = 1

d0 = 1 0 1

g = 1 1 1

1 0 1

x = 1 0 1 0 0

y = 1 1 1 0 0 0 1 0 1 1

antipodal_conv = 1 1 1 0 0 0 1 0 1 1

antipodal_conv = 1 1 1 -1 -1 -1 1 -1 1 1

r = 3 -4 -4 3 3 3 -4 3 -4 -4

trellis =

100000 100000 100000 100000 100000
100000 100000 100000 100000 100000
100000 100000 100000 100000 100000
100000 100000 100000 100000 100000

path =

0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0

new_path =

0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
output =

\[
\begin{array}{cccc}
0 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 \\
\end{array}
\]

transition =

\[
\begin{array}{ccc}
1 & 3 \\
1 & 3 \\
2 & 4 \\
2 & 4 \\
\end{array}
\]

trellis =

\[
\begin{array}{ccccccc}
29 & 58 & 66 & 95 & 77 \\
100000 & 38 & 74 & 59 & 121 \\
25 & 54 & 46 & 91 & 109 \\
100000 & 66 & 74 & 87 & 121 \\
\end{array}
\]

path =

\[
\begin{array}{cccccc}
1 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccc}
1 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 \\
\end{array}
\]

x_hat = 1 0 1

estimated_bits = 1 0 1

avg_ber = 0

- Using a soft – decision Viterbi decoder (maximum likelihood)

iterat = 1
N = 3

kk = 1

d0 = 1 0 1

g = 1 1 1
   1 0 1

x = 1 0 1 0 0

y = 1 1 1 0 0 0 1 0 1 1

antipodal_conv = 1 1 1 0 0 0 1 0 1 1

antipodal_conv = 1 1 1 -1 -1 -1 1 -1 1 1

r = 3 -4 -4 3 3 3 -4 3 -4 -4

trellis =

-10 -10 -10 -10 -10 -10 -10 -10 -10

path =

0 0 0 0 0 0 0 0 0 0

new_path =

0 0 0 0 0 0 0 0 0 0

output =

0 0 1 1
1 1 0 0
transition =

```
1 3
1 3
2 4
2 4
```

trellis =

```
-1 -2 4 3 29
-10 8 0 21 7
1 0 14 5 13
-10 -3 0 7 7
```

path =

```
1 0 1 0 0
1 0 1 1 0
1 0 1 0 1
1 0 1 1 1
```

```
1 0 1 0 0
1 0 1 1 0
1 0 1 0 1
1 0 1 1 1
```

x_hat = 1 0 1

estimated_bits = 1 0 1

avg_ber = 0
Appendix C: Matlab Simulation Results

Appendix C lists the values for the bit error rates calculated by our Matlab codes. These bit error rates are calculated as per the encoder and decoder used.

- for figure 4.1

(\textit{encoder: code rate} = 1/2, \textit{m} = 2 \textit{decoder: hard-decision})

\begin{verbatim}
N = 10000
iterat = 3

kk = 1
g = 1 1 1
  1 0 1

kk = 2
g = 1 1 1
  1 0 1

kk = 3
g = 1 1 1
  1 0 1

avg_ber = 0.0741 0.0339 0.0117 0.0034 0.0008 0.0001
\end{verbatim}

(\textit{encoder: code rate} = 1/2, \textit{m} = 2 \textit{decoder: soft-decision})

\begin{verbatim}
N = 10000
iterat = 3

kk = 1
g = 1 1 1
  1 0 1

kk = 2
g = 1 1 1
  1 0 1

kk = 3
g = 1 1 1
  1 0 1

avg_ber = 0.0296 0.0129 0.0038 0.0011 0.0001 0
\end{verbatim}
- for figure 4.2

(encoder: code rate = 1/2, m = 2 decoder: hard-decision)

N = 10000
iterat = 3

kk = 1
g = 1 0 1
  1 1 1

kk = 2
g = 1 0 1
  1 1 1

kk = 3
g = 1 0 1
  1 1 1

avg_ber = 0.0734  0.0304  0.0115  0.0034  0.0005  0.0001

(encoder: code rate = 1/2, m = 2 decoder: soft-decision)

N = 10000
iterat = 3

kk = 1
g = 1 0 1
  1 1 1

kk = 2
g = 1 0 1
  1 1 1

kk = 3
g = 1 0 1
  1 1 1

avg_ber = 0.0259  0.0115  0.0031  0.0008  0.0001  0

- for figure 4.3

(encoder: code rate = 1/2, m = 6 decoder: hard-decision)

N = 10000
iterat = 3

kk = 1
g = 1
0 0 1 1 1 1
1 1 0 1 1 0 1

kk = 2
g = 1
0 0 1 1 1 1
1 1 0 1 1 0 1

kk = 3
g = 1
0 0 1 1 1 1
1 1 0 1 1 0 1

avg_ber = 0.1104 0.0285 0.0057 0.0004 0 0

(encoder: code rate = 1/2, m = 6 decoder: soft-decision)

N = 10000
iterat = 3

kk = 1
g = 1
0 0 1 1 1 1
1 1 0 1 1 0 1

kk = 2
g = 1
0 0 1 1 1 1
1 1 0 1 1 0 1

kk = 3
g = 1
0 0 1 1 1 1
1 1 0 1 1 0 1

avg_ber = 0.0202 0.001 0 0 0 0

- for figure 4.4

(encoder: code rate = 1/2, m = 6 decoder: hard-decision)

N = 10000
iterat = 3

kk = 1
g = 1
0 1 1 0 1 1
1 1 1 1 0 0 1
\[ \text{kk} = 2 \]
\[ g = 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \]
\[ 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \]

\[ \text{kk} = 3 \]
\[ g = 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \]
\[ 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \]

\[ \text{avg\_ber} = 0.1168 \quad 0.0307 \quad 0.0027 \quad 0.0002 \quad 0 \quad 0 \]

\[(\text{encoder: code rate} = 1/2, \text{m} = 6 \text{ decoder: soft-decision})\]

\[ N = 10000 \]
\[ \text{iterat} = 3 \]

\[ \text{kk} = 1 \]
\[ g = 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \]
\[ 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \]

\[ \text{kk} = 2 \]
\[ g = 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \]
\[ 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \]

\[ \text{kk} = 3 \]
\[ g = 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \]
\[ 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \]

\[ \text{avg\_ber} = 0.0183 \quad 0.001 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \]

\text{for figure 4.5}

\[(\text{encoder: code rate} = 1/3, \text{m} = 2 \text{ decoder: hard-decision})\]

\[ N = 10000 \]
\[ \text{iterat} = 3 \]

\[ \text{kk} = 1 \]
\[ g = 1 \quad 1 \quad 1 \]
\[ 1 \quad 1 \quad 1 \]
\[ 1 \quad 0 \quad 1 \]

\[ \text{kk} = 2 \]
\[ g = 1 \quad 1 \quad 1 \]
\[ 1 \quad 1 \quad 1 \]
\[ 1 \quad 0 \quad 1 \]
\begin{verbatim}
kk = 3
  g = 1 1 1
     1 1 1
     1 0 1

avg_ber = 0.0764 0.0327 0.0107 0.0027 0.0005 0
  (encoder: code rate = 1/3, m = 2 decoder: soft-decision)

N = 10000
iterat = 3

kk = 1
  g = 1 1 1
     1 1 1
     1 0 1

kk = 2
  g = 1 1 1
     1 1 1
     1 0 1

kk = 3
  g = 1 1 1
     1 1 1
     1 0 1

avg_ber = 0.0271 0.0093 0.0021 0.0008 0.0001 0
  - for figure 4.6

  (encoder: code rate = 1/3, m = 2 decoder: hard-decision)

N = 10000
iterat = 3

kk = 1
  g = 1 0 1
     0 1 0
     1 1 0

kk = 2
  g = 1 0 1
     0 1 0
     1 1 0
\end{verbatim}
kk = 3
g = 1 0 1 0 1 0 1 1 0

avg_ber = 0.0593 0.0336 0.0106 0.0022 0.0002 0

(encoder: code rate = 1/3, m = 2 decoder: soft-decision)

N = 10000
iterat = 3

kk = 1
g = 1 0 1 0 1 0 1 1 0

kk = 2
g = 1 0 1 0 1 0 1 1 0

kk = 3
g = 1 0 1 0 1 0 1 1 0

avg_ber = 0.0243 0.0119 0.0034 0.0004 0 0

- for figure 4.7

(encoder: code rate = 1/3, m = 6 decoder: hard-decision)

N = 10000
iterat = 3

kk = 1
g = 1 0 0 1 1 1 1
1 0 1 0 1 1 1
1 1 0 1 1 0 1

kk = 2
g = 1 0 0 1 1 1 1
1 0 1 0 1 1 1
1 1 0 1 1 0 1
kk = 3
  g = 1 0 0 1 1 1 1
       1 0 1 0 1 1 1
       1 1 0 1 1 0 1

avg_ber = 0.0664  0.0142  0.0007  0.0001  0  0

(encoder: code rate = 1/3, m = 6 decoder: soft-decision)

N = 10000
iterat = 3

kk = 1
  g = 1 0 0 1 1 1 1
       1 0 1 0 1 1 1
       1 1 0 1 1 0 1

kk = 2
  g = 1 0 0 1 1 1 1
       1 0 1 0 1 1 1
       1 1 0 1 1 0 1

kk = 3
  g = 1 0 0 1 1 1 1
       1 0 1 0 1 1 1
       1 1 0 1 1 0 1

avg_ber = 0.0037  0.0007  0  0  0  0  0

- for figure 4.8

(encoder: code rate = 1/3, m = 6 decoder: hard-decision)

N = 10000
iterat = 3

kk = 1
  g = 1 0 1 0 1 1 1
       1 1 1 1 0 0 1
       1 1 0 0 1 0 1

kk = 2
  g = 1 0 1 0 1 1 1
       1 1 1 1 0 0 1
       1 1 0 0 1 0 1
kk = 3
g = 1  0  1  1  0  1  1
   1  1  1  1  0  0  1
   1  1  0  0  1  0  1

avg_ber = 0.0644   0.0106   0.0016   0.0001   0   0

(encoder: code rate = 1/3, m = 6 decoder: soft-decision)

N = 10000
iterat = 3

kk = 1
g = 1  0  1  1  0  1  1
   1  1  1  1  0  0  1
   1  1  0  0  1  0  1

kk = 2
g = 1  0  1  1  0  1  1
   1  1  1  1  0  0  1
   1  1  0  0  1  0  1

kk = 3
g = 1  0  1  1  0  1  1
   1  1  1  1  0  0  1
   1  1  0  0  1  0  1

avg_ber = 0.0037   0.0002   0   0   0   0