An-Najah National University

Faculty of Graduated Studies

Decoding Turbo Codes with Linear Programming

By

Hisham Hamed Abdel-Raouf Salahat

Supervisor

Dr. Mohammad Assa`d

Co supervisor

Dr. Mohammad Omran

This Thesis is Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Mathematics, Faculty of Graduate Studies, An-Najah National University, Nablus, Palestine.

2013

Decoding Turbo Codes with Linear Programming

By

Hisham Hamed Abdel-Raouf Salahat

This thesis was defended successfully on 30/12/2013 and approved by:

Defense Committee Members		signature
1- Dr. Mohammad Assa`d	(supervisor)	.....
2- Dr. Mohammad Omran	(Co supervisor)	.....
3- Dr. Saed Mallak	(external examiner)	.....
4- Dr. Sameer Matar	(internal examiner)	.....
5- Dr. Loai Malhees	(internal examiner)	.....

Dedication

To my parents and sisters.

To the souls of the martyrs of Palestine.

Acknowledgments

My first thanks go to my adviser Dr. Mohammad Assa`d. As I reflect on the past two years, I see that his efforts have had the goal of training me not only how to conjecture and prove, but also how to be a better colleague, and mathematical citizen. He had inspired me to the idea and methods of research and construction in order to reach the goals that we have set together. I am certain that his example will be one that I turn to in the years to come.

Second, I would like to thank Dr. Mohammad Omran for his Supervision of encouragement and guidance. He introduced me to the field of coding theory.

Thanks to Dr. Mohammad Assa`d and Mohammad Omran, both had made it easy and possible for me to accomplish this work successfully, with yours supervision, evaluations and encouragement.

A special thank-you goes to all my friends at An-Najah National University, who have been my second family during my time in Mathematics Department.

Thanks to the members of the committee discussion: Dr. Dr. Loai Malhees, Dr. Sameer Matar and Dr. Saed Mallak.

I give my final thanks to my parents. Although they may not understand the finer points of my work, their support and encouragement were essential to the completion of this dissertation. Thank you so much – I love you.

الإقرار

أنا الموقع أدناه مقدم الرسالة التي تحمل العنوان:

Decoding Turbo Codes with Linear Programming

أقر بأن ما اشتملت عليه هذه الرسالة إنما هي نتاج جهدي الخاص، باستثناء ما تم الإشارة إليه حيثما ورد، وأن هذه الرسالة ككل، أو أي جزء منها لم يقدم لنيل أي درجة أو لقب علمي أو بحثي لدى أية مؤسسة تعليمية أو بحثية أخرى.

Declaration

The work provided in this thesis, unless otherwise referenced, is the researcher's own work, and has not been submitted elsewhere for any other degree or qualification.

Student's Name: **اسم الطالب:**

Signature **التوقيع:**

Date: **التاريخ:**

List of Content

List of Figures

List of Tables

Decoding Turbo Codes with Linear Programming

By
Hisham Hamed Abdel-Raouf Salahat
Supervisor
Dr. Mohammad Assa`d
Co supervisor
Dr. Mohammad Omran

Abstract

In this thesis we investigate the application of *Linear Programming* LP relaxation to the problem of decoding an error-correcting code. LP relaxation is a standard technique in approximation algorithms and operation research, and it is used to find good suboptimal solution to very difficult optimization problems.

The method of a posteriori probability and iterative decoding algorithm is used to decode product codes (a special type of turbo codes). We investigate a program using Matlab to make computations to our algorithm. The logistic distribution with variance one is used. We compare the results of our computations to those of other authors; we find that our results are the best all over the others.

The LP method has its place in the generic turbo code, which is made up of asset of simpler" trellis-based" codes, we formulate the LP for a single trellis-based code as a min-cost flow problem, using the trellis as a directed flow. We extend this formulation to any turbo codes by applying constraints between the LP variables used in each component code.

One of the most advantages for LP decoding is that whenever the decoder output a result it is guaranteed to be the optimal solution, the most likely (ML) information sent over the channel, we refer to this property as the ML certificate property.

Preface

History of Coding Theory

Coding theory originated with the arrival of computers. Early computers were based on large banks of mechanical relays and their reliability was very low compared to the computers of today. If a single relay failed to work the entire calculation was in error. The engineers of that time developed ways to detect faulty relays so that they could be replaced. While working for Bell Labs, R.W. Hamming [25] had the idea that if the machine was capable of knowing there was an error, maybe it is also possible for the machine to correct that error. Based on this concept, he developed a way of encoding information so that if an error was detected, it could also be corrected. Based in part on this work, Claude Elwood Shannon developed the theoretical framework for the science of coding theory [13].

Shannon is considered as the founding father of electronic communications age. Shannon joined Bell Telephone Laboratories as a research mathematician in 1941 and has spent many years teaching at MIT. Shannon's most famous paper was the theory of communication which established the basis of the today’s communications [5]. Shannon's creation in the 1940's of the information theory is considered one of the great intellectual achievements of the twentieth century. As a mathematician, whose work pioneered digital communication and artificial intelligence

and was influential in cryptography and probability, he was one of the most important people of the 20th century.

The other vital figure in the history of coding theory was Richard Hamming, who set the basis of error correcting codes. After the end of World War II, Hamming joined the Bell Telephone Laboratories in 1946, where he was able to work with Shannon. Hamming is best known for his work on error- detecting and error- correcting codes. Hamming codes are of fundamental significance in coding theory.

Early days of Error Correcting Codes

When transmitting a message along a channel (such as a telephone line) in which errors occur randomly, we need to transmit more bits than there are in the original message in order to be able to detect and correct errors which occur in the transmission. One of the simplest methods for detecting errors in binary message (a message consisting only of 0's and 1's) is the parity code which transmits an extra "parity" bit after every 7 bits from the source message. However, this method can only detect errors and the only way to correct them is to ask for the data to be transmitted again, and also it can detect only up to one error.

A simple way to correct as well as detect errors is to repeat each bit several times. The decoder checks which value occurs more often and assumes it as the intended value. Each block of repeated symbols is called a codeword, i.e., a codeword is what is transmitted in place of one piece of information in the original

message. The set of all codewords is called a Code. If all the codewords in a code have the same length, then the code is called a Block Code. The repeat code is a block code. For example in transmitting the message: 01001100 over a noisy channel, using 5 repetitions for each bit the transmitted message would be: (00000 11111 00000 00000 11111 11111 00000 00000 00000 is a Codeword). Even if the message is altered in the transmission and the decoder receives the following message: 00100 11101 10000 00000 10011 01111 00000 011000, through a process of majority decoding the original message could be recovered. In the case of more than 2 errors per 5 consecutive bits the information would be decoded incorrectly (1 encoded and transmitted as 11111 and received as 00101 would be decoded as 0). The repeat code with codewords having length 5 can always detect from 1 to 4 errors made in the transmission of a codeword, and it can correct up to 2, and therefore it is called 2-error correcting and 4-error detecting code. If the received codeword is 11110 and 1 is sent then we have 1 error, it is detecting and correcting to 11111 which is the sent codeword but if 11110 is received and 0 is sent then it is detecting but not correcting.

Remark: It can be seen that if the number of repetitions is increased so is the number of errors capable of being detected and corrected. But there is a heavy price paid for this, and that is the efficiency of the transmission. *The increased the length of the*

transmitted code, the more time and energy is required to transmit it and to decode the message correctly. The efficiency of a coding procedure depends upon the coding scheme used in encoding and decoding and this is where advanced coding algorithms play an essential role. The disadvantage of the repetition scheme is that it multiplies the number of bits transmitted by a factor which is unacceptably high.

In 1948, Shannon, working at Bell Laboratories, showed that it was possible to encode messages in such a way that the number of extra bits transmitted was as small as possible using explicit error-correcting codes with information transmission rates more efficient than simple repetition.

Chapter One

Coding Theory Background

1.1 Introduction:

Coding theory is the branch of mathematics concerned with accurate and efficient transfer of data across noisy channels as well as the recovery of the message sent. A transmission channel is the physical medium through which the information is transmitted, such as telephone lines, or atmosphere in the case of wireless communication. Undesirable disturbances (noise) can occur across the communication channel, causing the received information to be different from the original information sent. Coding theory deals with detection and correction of the transmission errors caused by the noise in the channel. The primary goal of coding theory is efficient encoding of information, easy transmission of encoded messages, fast decoding of received information and correction of errors introduced in the channel. Coding Theory is used all the time: in reading CDs, receiving transmissions from satellites, or in cell phones. Coding theory should not be confused with cryptography, which is the art of encrypting messages (making them secure and thus hard to read for the unintended listeners).

The purpose of the transmission environment is *communication,* transfer of information from one place to another (phone conversation, data transfer over the internet, radio transmission, wireless data transfer etc.).

1.2 Components of an Error-Correcting Code

Error-correcting codes are the basic tools used to transmit digital information over an unreliable communication channel. The channel can take on a variety of different forms. For example, if you are sending voice information over a cellular phone, the channel is the air; this channel is unreliable because the transmission path varies, and there is interference from other users.

A common abstraction of channel noise works as follows. We first assume that the information is a block of k bits (a block of *k 0s* and *1s*). When the sending party transmits the bits over the channel, some of the bits are flipped; some of the *0s* are turned into *1s*, and vice-versa. The receiving party must then try to recover the original information from this corrupt block of bits.

In order to counter the effect of the "noise" in the channel, we send more information. ***For example***, we can use a repetition code: for each bit we want to send, we send it many times, say five times. This longer block of bits we send is referred to as the codeword. The receiving party then uses the following process to recover the original information: for every group of five bits received, if there are more *0s* than *1s*, assume a 0 was transmitted, otherwise assume a 1 is transmitted. Using this scheme, as long as no more than two out of every group of five bits are flipped, the original information is recovered.

Below we list the components of an Error-Correcting Code to familiarize the reader with the terminology used throughout this thesis, and in all coding theory references.

- The ***information word*** is a block of symbols that the sending party wishes to transmit. This information could be a sound, a picture of Mars, or some data on a disk. For our purposes, the information word $x \in \{0,1\}^k$ is simply an arbitrary binary word of k bits.

  There are some error-correcting codes built on information streams of arbitrary (***infinite***) length, but we will restrict ourselves to block codes in this thesis (codes defined for some ***finite length n***). With a ***block code***, in order to send more than k bits of information, multiple blocks are transmitted. It is assumed that each block is independent with respect to information content, and the effect of noise; therefore, we concentrate on sending a single block over the channel. We also note that some codes use non-binary alphabets; i.e., the symbols of the information word are not just bits, but are symbols from a larger alphabet. We restrict ourselves to binary codes in this thesis; those built on the alphabet $\{0,1\}$.

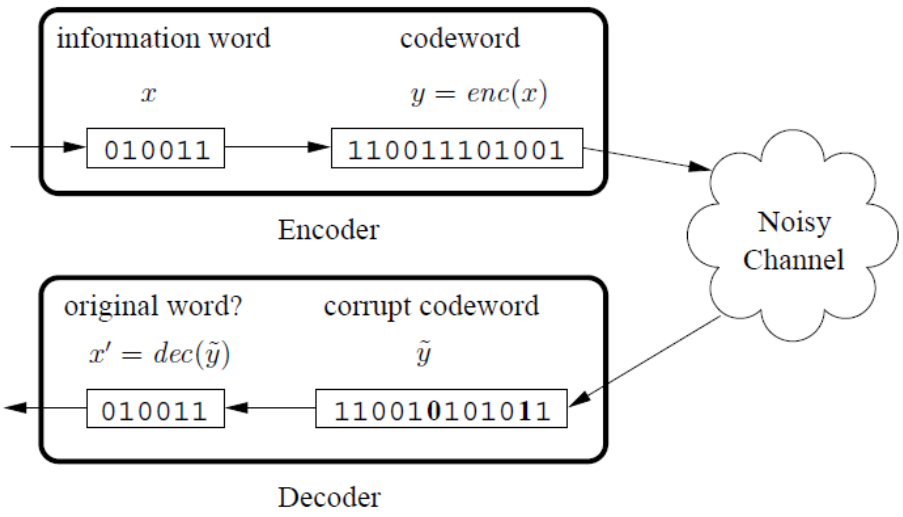

Figure (1): The high-level model of an error-correcting code. An information word *x* of length *k* is encoded into a longer code word *y* = *enc(x)* of length *n* > *k*. This code word *y* is sent over a noisy channel, and a corrupt word $\tilde{y}$ is received at the other end. This corrupt word is then decoded to a word *x′* = *dec*($\tilde{y}$), which hopefully is equal to the original information *x*.

- The ***encoder*** is the process the sending party uses to build redundancy into the information word. The encoder is simply a function $enc:\{0,1\}^k \rightarrow \{0,1\}^n$ that given an information word, outputs a longer binary word *y* of length *n* > *k*. In our repetition code example, the encoder is the function that repeats every bit five times, and so *n* = 5*k*. For example, if *x* = 101, then our encoder would output *y* = *enc*(101) = 111110000011111.
- The ***codeword*** is the binary word output by the encoder. This is the word that is transmitted over the channel.
- ***The code C*** is the set of codewords that could possibly be transmitted, i.e., they are the binary words that are encodings of some information word. Formally, C ={*y*: *y*= *enc(x)*, $x \in \{0,1\}^k$}. In our repetition code example, the code is exactly the set of binary

words of length *n* where the first five bits are the same, and the next five bits are the same, etc.

- The ***(block) length*** of the code is equal to *n*, the length of the codewords output by the encoder. The parameter *k*, the length of the information word, is often referred to as the dimension of the code. This term is used for linear codes, when the code is a vector space.
- The ***rate*** r of the code is the ratio between the dimension and the length. More precisely, $r = k/n$. The rate of our example code is 1/5. It is desirable to have a high rate, since then information can be transmitted more efficiently. One of the main goals in the study of error-correcting codes is to transmit reliably using a code of high rate (close to 1).
- The ***channel*** is the model of the communication medium over which the transmissions sent. We have already described a channel that flips bits of the codeword arbitrarily. Another common model is called the binary symmetric channel (BSC), where each bit is flipped independently with some fixed crossover probability *p*. There are many other models, some of which we will discuss in later parts of this chapter.
- The ***received word*** $\tilde{y}$ is the output of the channel, the corrupted form of the codeword *y*. Each symbol $\tilde{y}_i$ of the received word is drawn from some alphabet Σ. In the **BSC**, $\Sigma = \{0, 1\}$. In the additive white Gaussian noise(AWGN) channel (the details of which we review in

Section 1.3.2), each symbol of the received word is a real number, so $\Sigma = R$.

- The ***decoder*** is the algorithm that receiving party uses to recover the original information from the received word $\tilde{y}$. In the repetition code example, the decoder is the following algorithm:

1. Examine every five bits of the codeword.
2. For each set of five bits, if there are more 0*s* than 1*s*, output a 0, otherwise output a 1. We can think of the decoder as a function:

$$\text{dec}:\Sigma^{n} \rightarrow \{0,1\}^{k}$$

Which takes as input the corrupt code word $\tilde{y}$ and outputs a decoded information word x'.

Usually the difficulty of the decoding process is not in translating from codewords back into information words, but rather in finding a codeword that was likely sent, given the received word $\tilde{y}$. In most codes used in practice, and in all the codes of this thesis, given a codeword $y \in C$, finding the information word x such that $enc(x) = y$ is *straightforward*. Thus, in the discussion we will simply talk about a decoder finding a codeword, rather than an information word.

1.3 Channels

1.3.1 Channels Decoding Methods

The task of channel coding is to encode the information sent over a communication channel in such a way that in the presence of channel noise, errors can be detected and/or corrected. We distinguish between two coding methods:

Method 1: Backward Error Correction (BEC)

This method requires only error detection: if an error is detected, the sender is requested to retransmit the message. While this method is simple and sets lower requirements on the code's error-correcting properties, it on the other hand requires duplex communication and causes undesirable delays in transmission.

Method 2: Forward Error Correction (FEC)

This method requires that the decoder should also be capable of correcting a certain number of errors, i.e. it should be capable of locating the positions where the errors occurred. Since FEC codes require only simplex communication, they are especially attractive in wireless communication systems, helping to improve the energy efficiency of the system.

1.3.2 Channels Types

We consider the following three channels, the **Binary Symmetric Channel (BSC)**, the Binary Erasure Channel (BEC) and the **Binary Input Additive White Gaussian Noise Channel (BI-AWGNC).**

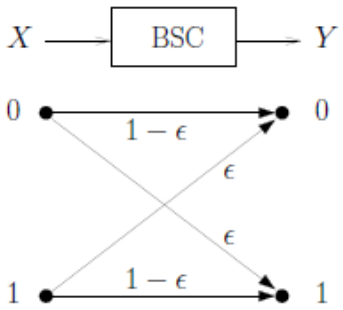

Figure (2): The BSC with cross over probability ϵ.

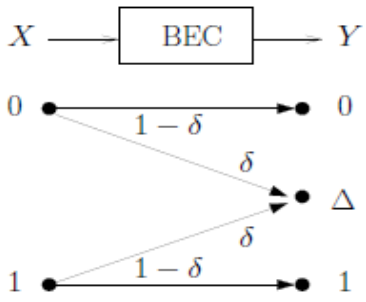

Figure (3): The BEC with erasure probability δ.

For the BSC in Figure (2) some input values are with probability ϵ flipped at the output, Y ∈ {0, 1}. In the BEC Figure (3) the input values can be erased (Δ) with probability δ , mapping X into Y ∈ {0, 1}.

For the (normalized) BI-AWGNC in Figure (4), the input X is mapped into X′ ∈ {0, 1} which is added with Gaussian white noise, resulting in the output Y = X′ +W, where W ∼ $\mathcal{N}(0, N_0/2E_s)$. The conditional distribution of Y is

$$\Pr(Y|X') = \Pr W(Y - X') = \frac{1}{\sqrt{2\pi(N_0/2E_s)}} \exp\left(-\frac{(Y-X')^2}{N_0/E_s}\right) \quad (1)$$

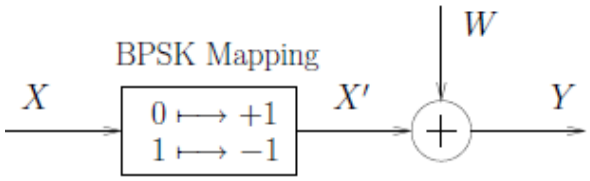

Figure (4): The normalized BI-AWGNC.

Definition 1.1: if the length of the output codewords is finite and stay the same for all codewords "say *n*" then the code is called a **Block Code**.

1.4 Examples of Block codes

1-Linear Block Codes

Some of the most important error correcting codes are the linear codes

Definition 1.2: A binary block code is said to be linear provided that the sum of arbitrary two codewords is a codeword.

Linear block codes form a vector space. A vector space is a collection of vectors, here codewords, which is closed under vector

addition and scalar multiplication, in other words contains all the possible linear combinations of its codewords.

Definition 1.3: If the vector space of all codewords is *n* dimensional and the subspace formed by the codewords of a code is *k* dimensional then the code is described as an (*n, k*)-linear code.

The dimension of a vector space is given by the number of vectors in the basis of the vector space; a basis for a vector space or subspace is a minimum collection of linearly independent vectors which generate the entire vector space; linearly independent vectors are vectors such that none of them can be expressed as a linear combination of the others).

There are two ways of describing a linear code C.

1- Using a ***generator matrix*** G which has as its rows a set of basis vectors of the linear subspace C. For every linear code there is an equivalent code which has a generator matrix of the form $G = [I_k \ P]$, where I_k is the *k* by *k* identity matrix and P is a *k* by *n-k* matrix.

2- Description of a linear code C consists in specifying not vectors in C but rather the vectors orthogonal to C (orthogonal vectors are vectors whose inner or dot product is 0). The orthogonal complement of C is a subspace and in fact is another code called the dual code of C denoted by $C^{\perp}$.

Definition 1.4: If P is a generator matrix for $C^{\perp}$ then P is called a parity check matrix for C. In general the rows of the parity check matrix for a code C are orthogonal to all codewords of C and any matrix P is a parity

check matrix if the rows of P generate the dual code of C. Therefore, a code C is defined by such a parity check matrix in the following way:

$$C = \{ x \mid x.P^T = 0 \}.$$

The parity check matrix can be obtained from the null space of the generator matrix and the generator matrix can be obtained by finding the null space of the parity check matrix, (the null space of a matrix is the collection of all the orthogonal vectors to the vector space generated by the rows of the matrix, also known as the row space)

2-Hamming Codes

Discovered by Richard Hamming in 1950, the Hamming codes are the most famous of all error-correcting codes. Hamming code is an error-detecting and error-correcting code. We restrict to binary Hamming codes in this discussion, which is used in data transmission, and can detect all single-bit and double-bit errors and correct all single-bit errors. For a linear (n, k)-code C, the parity-check matrix for C is the generator matrix P of the dual code $C^{\perp}$. Furthermore, we can use P to determine the codewords of C by all c such that $Pc^t = 0$.

A Hamming codeword is generated by multiplying the data bits by a generator matrix G using mod 2 arithmetic (in mod 2 arithmetic 0+0=0, 0+1=1, 1+0=1, 1+1=0, 0*0=0, 0*1=0, 1*0=0, and 1*1=1). The result of this multiplication is called the codeword vector [c_1 c_2

c_3 .....c_n], consisting of the original data bits and the additional bits used for error correcting. The generator matrix G used in constructing Hamming codes can be written as I (the identity matrix) and a parity generation matrix A:

$$G = [IA]$$

An example of Hamming (7, 4) code generator matrix:

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

If a 4-bit message vector (d_1 d_2 d_3 d_4) is multiplied by G, the result is a 7-bit codeword of the form (d_1 d_2 d_3 d_4 p_1 p_2 p_3). It can be seen that each codeword contains the original message and three additional check bits, which are a linear combination of the message bits based on the columns of A. Validating the received codeword r, involves multiplying it by a parity check matrix P, to form s, the parity check vector.

$$P = [A^T I_3]$$

We can either multiply r * P^T, or equivalent P * r^T. We choose to do the latter, for instance if r = (1 0 0 1 0 0 1) then s = P * r^T =

$$s = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

If all elements of s are zero, the codeword assumed to be received correctly. If s contains non-zero elements, the bit in error can be determined by analyzing which check bits have failed, as long as the error involves only a single bit. For instance if r=[10*1*1001], s computes to $[101]^T$, which matches the third column in P that corresponds to the third bit of r - the bit in error.

Using the Hamming (7, 4) code we can detect up to two errors and correct up to one error, with only three additional bits for every four bits of data.

3-Cyclic Codes

One of the most important classes of linear codes is the class of cyclic codes. In general these codes are much easier to implement and have great practical value. Cyclic codes have the property that there exists a generator of the entire code. There are two approaches to describe such codes. A code, as mentioned above, is cyclic if whenever $[c_1\ c_2\ \ldots c_n]$ is in the code, so is the shift $[c_n\ c_1\ \ldots c_{n-1}]$. A more indirect way is to use polynomials, which we will illustrate briefly. If C is an ideal in R_n (ring R_n being a principal ideal domain, $R_n = F_q[x] / \langle x^n - 1 \rangle$), then there is a monic polynomial g(x) of minimal degree in C, such that g(x) generates C, $C=\langle g(x) \rangle$.

Such code C is called cyclic code and g(x) is called the generator polynomial.

4-BCH Codes

Discovered independently by R.C.Bose, D.K. Ray-Chaudhuri in 1960 and by A. Hocquenghem in 1959, BCH codes are one of the most important classes of cyclic codes, with good error correcting capabilities and a relatively fast and easy encoding and decoding procedure. BCH codes are a generalization of Hamming codes, but a lot more efficient since the designed distance of the BCH codes can be d=2 or greater, which allows correcting up to two errors. A kind of BCH code is used in reading CDs. BCH codes are best explained using polynomials, as in the case of cyclic codes.

In this section we define a Block Code and give some examples, in the following section we will introduce two types of code which is not a **B**lock Code.

1.5 Non "linear Block Codes"

1.5.1 Convolutional Codes

One important difference between convolutional codes and block codes is that the encoder contains memory. Encoders of convolutional codes can also be divided into two categories, namely feedforward and feedbackward. In both of these categories the encoder can be systematic or nonsystematic.

Definition 1.5: ***systematic encoder*** is an encoder in which one of the outputs is equal to the input.

Definition 1.6: ***nonsystematic encoder*** is an encoder in which no one of the outputs is equal to the input values.

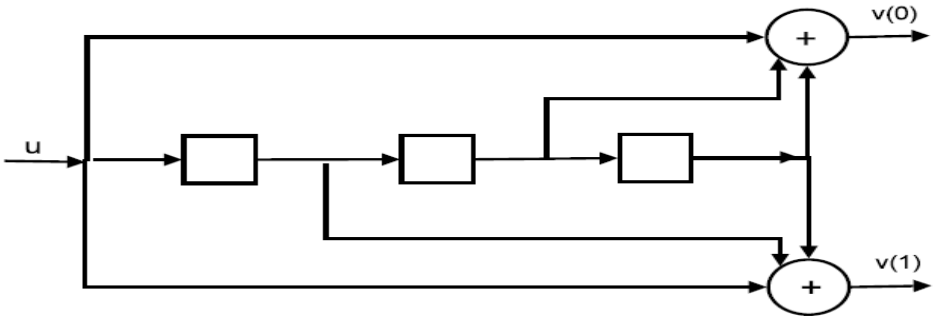

Figure (5) : A binary nonsystematic feedforward convolutional encoder

Figure (5) shows how a simple convolutional encoder with rate R = 1/2 might work. The figure can be viewed as a shift register with u as input, and $v^{(0)}$and $v^{(1)}$as output. The information sequence u = (u_0, u_1, u_2, . . .) enters an encoder, one bit at the time. From the figure the reader can see that the encoder holds memory. This encoder actually contains memory of order $m = 3$, which is shown by the boxes in the figure. The circle with a plus inside XORs the bits from the boxes. It is often easier to represent the encoder figure as generator sequences instead. From the figure the generator sequences, $g^{(0)}$and $g^{(1)}$, will be:

$$g^{(0)} = (1\ 0\ 1\ 1\ ), \tag{1}$$

$$g^{(1)} = (1\ 1\ 0\ 1\ ), \tag{2}$$

$g^{(0)}$and $g^{(1)}$ are prescribed by the connections shown in figure(5).

(We have lines join $v^{(0)}$ after u and m_2 and m_3 now it is clear that it corresponds to $g^{(0)} = (1\ 0\ 1\ 1\ )$, similarly see $g^{(1)} = (1\ 1\ 0\ 1\ )$). Computing the outputs, $v^{(0)}$ and $v^{(1)}$, is done by performing discrete

convolution, denoted ⊗ as defined below. All the operations are modulo-2. The two output sequences will then be denoted by the following encoding equations:

$$v^{(0)} = u \otimes g^{(0)} \quad (3)$$

$$v^{(1)} = u \otimes g^{(1)} \quad (4)$$

Now, Let us consider a finite case, say $u = (u_0 u_1 \dots u_k)$, $g = (g_0 g_1 \dots g_m)$ and $v = (v_0 v_1 \dots v_{k+m})$, where k is the length of the message, m is the number of the memories the encoder contains.

Then discrete convolution can be written as a sum of products, it is defined as follows, for all $l: 0,1, \dots, k+m$ we have:

$$v_l^{(j)} = \sum_{i=0}^{m} u_{l-i} g_i^{(j)} = u_l g_0^{(j)} + u_{l-1} g_1^{(j)} + \dots + u_{l-m} g_m^{(j)}, j=0, 1 \quad (5)$$

Example: Let us pick a random message, namely u = (1 0 1 0 1). Thus encoding this message can be expressed as follows:

$$v^{(0)} = (1\ 0\ 1\ 0\ 1) \otimes (1\ 0\ 1\ 1) = (1\ 0\ 0\ 1\ 0\ 1\ 1\ 1), \quad (6)$$

$$v^{(1)} = (1\ 0\ 1\ 0\ 1) \otimes (1\ 1\ 0\ 1) = (1\ 1\ 1\ 0\ 1\ 0\ 0\ 1), \quad (7)$$

One can verify this easily for example in $v^{(0)}$ we have:

$$v_2^{(0)} = u_2 g_0^{(0)} + u_1 g_1^{(0)} + u_0 g_2^{(0)} + u_{-1} g_3^{(0)}$$

$$= 1.1 + 0.0 + 1.1 = 0$$

The transmitted coded signal is then the concatenated of $v^{(0)}$ *and* $v^{(1)}$ defined by $v = (v_0^{(0)} v_0^{(1)}, v_1^{(0)} v_1^{(1)}, \dots, v_{m+k}^{(0)} v_{m+k}^{(1)})$

So v = (1 1, 0 1, 0 1, 1 0, 0 1, 1 0, 1 0, 1 1). (8)

The matrix form:

It is sometimes convenient to represent the two generator sequences by a matrix. This matrix, denoted ***G***, is constructed by interlacing the generator sequences, in this case $g^{(0)}$ and $g^{(1)}$. Every interleaved row in ***G*** will be exactly the same as the first row of ***G***, the only difference being a shift of length *l*, where *l* is the number of generator sequences. So in this example each shift will be 2. The number of rows of ***G*** will be the same as the length of the information sequence *u*.

$$\boldsymbol{G} = \begin{bmatrix} g_0{}^{(0)}g_0{}^{(1)} & g_1{}^{(0)}g_1{}^{(1)} & g_2{}^{(0)}g_2{}^{(1)} & \cdots & g_m{}^{(0)}g_m{}^{(1)} & & \\ & g_0{}^{(0)}g_0{}^{(1)} & g_1{}^{(0)}g_1{}^{(1)} & \cdots & g_{m-1}{}^{(0)}g_{m-1}{}^{(1)} & g_m{}^{(0)}g_m{}^{(1)} & \\ & & g_0{}^{(0)}g_0{}^{(1)} & \cdots & g_{m-2}{}^{(0)}g_{m-2}{}^{(1)} & g_{m-1}{}^{(0)}g_{m-1}{}^{(1)} & g_m{}^{(0)}g_m{}^{(1)} \\ & & \ddots & & & & \ddots \end{bmatrix}$$

The codeword can now be defined as a matrix-vector multiplication,

$$v = u\boldsymbol{G} = (1\,0\,1\,0\,1)\begin{bmatrix} 1\,1 & 0\,1 & 1\,0 & 1\,1 & & & \\ & 1\,1 & 0\,1 & 1\,0 & 1\,1 & & \\ & & 1\,1 & 0\,1 & 1\,0 & 1\,1 & \\ & & & 1\,1 & 0\,1 & 1\,0 & 1\,1 \end{bmatrix}$$

$$= (1\,1, 0\,1, 0\,1, 1\,0, 0\,1, 1\,0, 1\,0, 1\,1)$$

which is exactly the same as found in (8).

The general form of the matrix G when the encoder has *m* memories and *k* generator sequences is: ***G***=

$$\begin{bmatrix} g_0{}^{(0)}g_0{}^{(1)}..g_0{}^{(k)} & g_1{}^{(0)}g_1{}^{(1)}..g_1{}^{(k)} & g_2{}^{(0)}g_2{}^{(1)}..g_2{}^{(k)} & \cdots & g_m{}^{(0)}g_m{}^{(1)}..g_m{}^{(k)} & & \\ & g_0{}^{(0)}g_0{}^{(1)}..g_0{}^{(k)} & g_1{}^{(0)}g_1{}^{(1)}..g_1{}^{(k)} & \cdots & g_{m-1}{}^{(0)}g_{m-1}{}^{(1)}..g_{m-1}{}^{(k)} & g_m{}^{(0)}g_m{}^{(1)}..g_m{}^{(k)} & \\ & & g_0{}^{(0)}g_0{}^{(1)}..g_0{}^{(k)} & \cdots & g_{m-2}{}^{(0)}g_{m-2}{}^{(1)} & g_{m-1}{}^{(0)}g_{m-1}{}^{(1)} & \cdots \\ & & \ddots & & & & \ddots \end{bmatrix}$$

Convolutional encoders can be constructed in many different ways. The previous example was a code with rate R = 1/2. Other encoders with rate 2/3 might have for instance two input sequences and three output sequences.

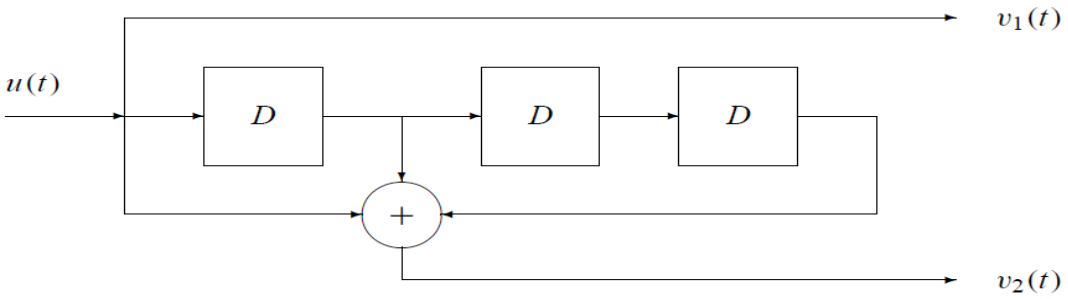

Figure (6) :A systematic encoder

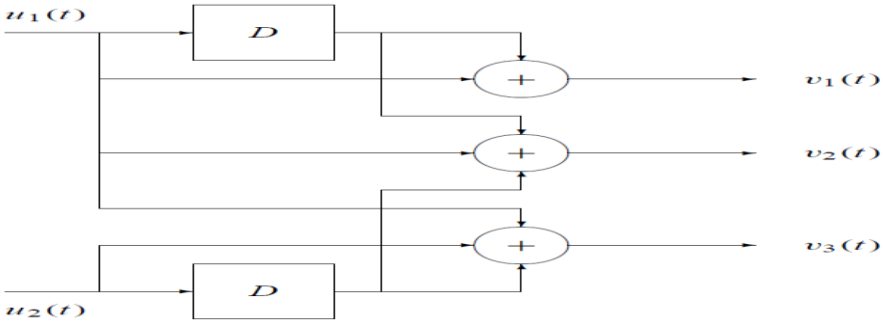

Figure (7) : A rate=2/3 encoder

The encoder in figure (6) is ***systematic*** since one of its outputs is equal to the input i.e. $v_1(t) = u(t)$. The rate *R* of this code is 1/2, its memory *M* =3. For every *k* = 2 binary input symbols the encoder in figure (7) produces *n* = 3 binary output symbols. Therefore its rate *R* = *k*/*n* = 2/3. The memory *M* of this encoder is 1 since only u_1(t-1) and u_2(t-1) are used to produce $v_1(t)$, $v_2(t)$ and $v_3(t)$, in addition to u_1(t) and u_2(t).

1.5.2 Turbo Codes: Encoding with Interleaving

The first turbo code, based on convolutional encoding, was introduced in 1993 by Berrou. Since then, several schemes have been proposed and the term"turbo codes" has been generalized to cover block codes as well as convolutional codes. Simply put,

Definition 1.7*:* a Turbo Code is formed from the parallel concatenation of two codes separated by an interleaver.

The generic design of a turbo code is depicted in Figure (8). Although the general concept allows for free choice of the encoders and the interleaver, most designs follow the ideas presented in [26].

- The two encoders used are normally identical;
- The code is in a systematic form, i.e. the input bits also occur in the output (see Figure (8)).
- The interleaver reads the bits in a pseudo-random order.

The choice of the interleaver is a crucial part in the turbo code design. The task of the interleaver is to "scramble" bits in a (pseudo-)random, albeit predetermined fashion. This serves two purposes. Firstly, if the input to the second encoder is interleaved, its output is usually quite different from the output of the first encoder. This means that even if one of the output code words has low weight, the other usually does not, and there is a smaller chance of producing an output with very low weight. Higher weight, as we saw above, is beneficial for the performance of the decoder. Secondly, since the code is a parallel concatenation of two codes, the divide-and-conquer strategy can be employed for decoding. If the input to the second decoder is scrambled, also its output will be different, or "uncorrelated" from the output of the first encoder. This means that the corresponding two decoders will gain more from information exchange.

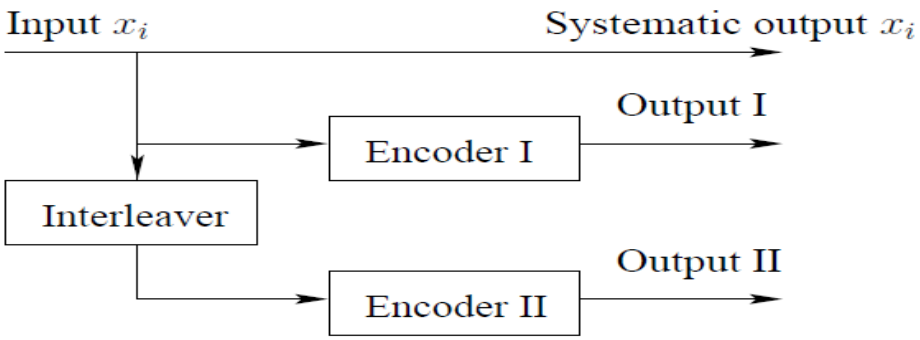

Figure (8) : The generic turbo encoder

We now briefly review some interleaver design ideas, stressing that the list is by no means complete. The first three designs are illustrated in Figure (9) with a sample input size of 15 bits.

1. **A *"row-column" interleaver***: data is written row-wise and read column wise. While very simple, it also provides little randomness.
2. **A *"helical" interleaver***: data is written row-wise and read diagonally.
3. **An *"odd-even" interleaver***: first, the bits are left uninterleaved and encoded, but only the odd-positioned coded bits are stored. Then, the bits are scrambled and encoded, but now only the even-positioned coded bits are restored. Odd-even encoders can be used, when the second encoder produces one output bit per one input bit.
4. **A *pseudo-random interleaver*** defined by a pseudo-random number generator or a look-up table.

Input				
x_1	x_2	x_3	x_4	x_5
x_6	x_7	x_8	x_9	x_{10}
x_{11}	x_{12}	x_{13}	x_{14}	x_{15}

Row-column interleaver output														
x_1	x_6	x_{11}	x_2	x_7	x_{12}	x_3	x_8	x_{13}	x_4	x_9	x_{14}	x_5	x_{10}	x_{15}

Helical interleaver output														
x_{11}	x_7	x_3	x_{14}	x_{10}	x_1	x_{12}	x_8	x_4	x_{15}	x_6	x_2	x_{13}	x_9	x_5

Odd-even interleaver output														
Encoder output without interleaving														
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}
y_1	-	y_3	-	y_5	-	y_7	-	y_9	-	y_{11}	-	y_{13}	-	y_{15}
Encoder output with row-column interleaving														
x_1	x_6	x_{11}	x_2	x_7	x_{12}	x_3	x_8	x_{13}	x_4	x_9	x_{14}	x_5	x_{10}	x_{15}
-	z_6	-	z_2	-	z_{12}	-	z_8	-	z_4	-	z_{14}	-	z_{10}	-
Final output of the encoder														
y_1	z_6	y_3	z_2	y	z_{12}	y_7	z_8	y_9	z_4	y_{11}	z_{14}	y_{13}	z_{10}	y_{15}

Figure (9): Interleaver designs

There is no such thing as a universally best interleaver. For short block sizes, the odd-even interleaver has been found to outperform the pseudo-random interleaver, and vice versa. The choice of the interleaver has a key part in the success of the code and the best choice is dependent on the code design. For further reading, several articles on interleaver design can be found for example at [26].

Some Notes on Decoding

In the traditional decoding approach, the demodulator makes a “hard” decision of the received symbol, and passes to the error control decoder a discrete value, either a 0 or a 1. The disadvantage of this approach is that

while the value of some bits is determined with greater certainty than that of others, the decoder cannot make use of this information.

A ***soft-in-soft-out (SISO)*** decoder receives as input a “soft” (i.e. real) value of the signal. The decoder then outputs for each data bit an estimate expressing the probability that the transmitted data bit was equal to one. In the case of turbo codes, there are two decoders for outputs from both encoders. Both decoders provide estimates of the same set of data bits, albeit in a different order. If all intermediate values in the decoding process are soft values, the decoders can gain greatly from exchanging information, after appropriate reordering of values. Information exchange can be iterated a number of times to enhance performance. At each round, decoders re-evaluate their estimates, using information from the other decoder, and only in the final stage will hard decisions be made, i.e. each bit is assigned the value 1 or 0. Such decoders, although more difficult to implement, are essential in the design of turbo codes.

Turbo codes can be achieved by serial or parallel concatenation of two (or more) codes called the constituent codes. The constituent codes can be either block codes or convolutional codes. Currently, most of the work on turbo codes has essentially focused on convolutional Turbo Code (CTC)'s and Block Turbo Code (BTC)'s have been partially neglected.

Remark 1.1:

1. BTC resulted from the combination of three ideas that were known to all in the coding community.

2. The utilization of block codes instead of commonly used nonsystematic or systematic convolutional codes.
3. The utilization of soft input soft output decoding instead of using hard decisions, the decoder uses the probabilities of the received data to generate soft output which also contain information about the degree of certainty of the output bits.
4. Encoders and decoders working on permuted versions of the same information. This is achieved by using an interleaver.

1.5.3 Concatenated Codes

The power of Forward Error Correction codes can be enhanced by using the concatenated codes, which are shown in Figure (10). Concatenated codes were first introduced by Elias in 1954 [3]. The principle is to feed the output of one encoder (called the outer encoder) to the input of another encoder, and so on, as required. The final encoder before the channel is known as the inner encoder. The resulting composite code is clearly much more complex than any of the individual codes. However it can readily be decoded: we simply apply each of the component decoders in turn, from the inner to the outer.

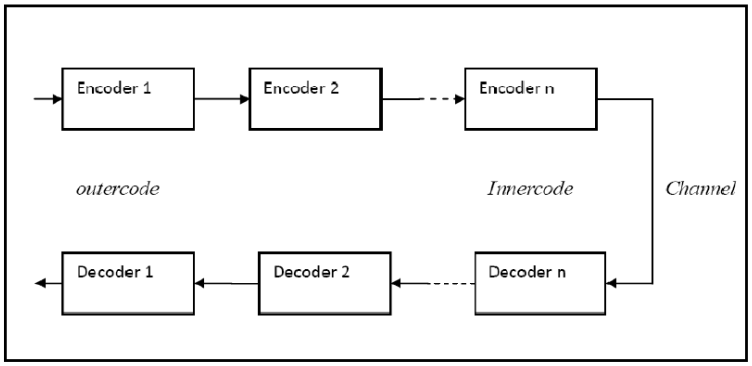

Figure (10): Principle of Concatenated codes

1.5.4 Block Turbo Code

Block Turbo Codes (or product codes) are serially concatenated codes [4] which were introduced by Elias in 1954 [3]. The concept of product codes is very simple and relatively efficient for building very long block codes by using two or more short block codes. Let us consider two systematic linear block codes c^1 with parameters (n_1 , k_1 , δ_1) and c^2 with parameters (n_2 , k_2 , δ_2), where n_i , k_i and δ_i stand for codeword length, number of information bits, and Minimum Hamming Distance, respectively. The product code is obtained (as shown in Figure (11)) by:

1. Placing ($k_1 \times k_2$) information bits in an array of k_1 rows and k_2 columns;
2. Coding the k_1 rows using code c^2;
3. Coding the k_2 columns using codec^1.

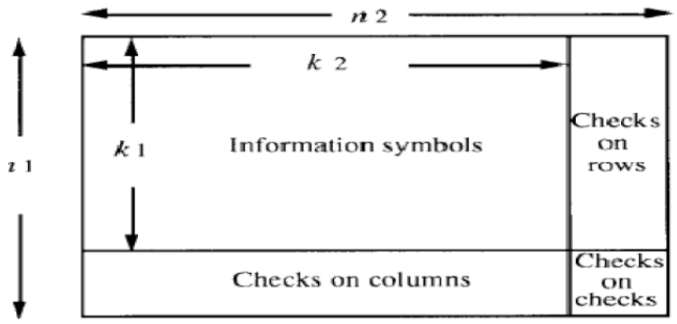

Figure (11): Construction of product code P = $c^1 \otimes c^2$

The parameters of the product code P are $n = n_1 \text{x}\, n_2$, $k = k_1 \text{ x}\, k_2$, $\delta = \delta_1 \text{x}\, \delta_2$ and the code rate R is given by $R = R_1 \text{x}\, R_2$, where R_i is the code rate of code c^i [12]. Thus, we can build very long block codes with large minimum Hamming distance by combining short codes with small minimum Hamming distance. Given the procedure used to construct the product code, it is clear that the $(n_2 - k_2)$ last columns of the matrix are

codewords of c^1. By using the matrix generator [13], one can show that the $(n_1 - k_1)$ last rows of matrix P are codewords of c^2.

Hence, all of the rows of matrix P are codewords of c^1 and all of the columns of matrix P are codewords of c^2.

1.6 System of Model Decoding

Now we consider the system model and different channel models that are used throughout this thesis. Moreover, we introduce commonly used assumptions and definitions that will be used in our work.

1.6.1 Basic System Model

One may be given the message $x \in F_2^n$, then ideal observer decoding generates the codeword $y \in C$ .The process results in this solution:

$$\text{P}(\ y \text{ sent} \mid x \text{ received}\ )$$

For example, a person can choose the codeword *y* that is most likely to be received as the message *x* after transmission.

1.6.2 Decoding conventions

Each codeword does not have an expected possibility: there may be more than one codeword with an equal likelihood of mutating into the received message. In such a case, the sender and receiver(s) must agree ahead of time on a decoding convention. Popular conventions include:

1. Request that the codeword be resent – automatic repeat-request.
2. Choose any random codeword from the set of most likely codewords which is nearer to that.

1.6.3 Maximum likelihood decoding (MLD)

Given a received codeword $x \in F_2^n$ ***maximum likelihood decoding*** picks a codeword $y \in C$ to *maximize*:

$$\mathrm{P}(\, x \text{ received} \mid y \text{ sent} \,)$$

i.e. choose the codeword y that maximizes the probability that x was received, given that y was sent. Note that if all codewords are equally likely to be sent then this scheme is equivalent to ***ideal observer decoding***. In fact, by Bayes Theorem we have:

$$\mathrm{P}(x \text{ received} \mid y \text{ sent}) = \frac{P(x \text{ received}, y \text{ sent})}{P(y \text{ sent})}$$

$$= P(\, y \text{ sent} \mid x \text{ received} \,) \cdot \frac{P(x \text{ received})}{P(y \text{ sent})}$$

Upon fixing $P(x \text{ received})$, x is restructured and $P(y \text{ sent})$ is constant as all codewords are equally likely to be sent. Therefore $\mathrm{P}(x \text{ received} \mid y \text{ sent})$ is maximized as a function of the variable y precisely when $P(y \text{ sent} | x \text{ received})$ is maximized, and the claim follows.

As with *ideal observer decoding*, a convention must be agreed to for non-unique decoding.

The ML decoding problem can also be modeled as an integer programming problem.

Chapter 2

Decoding Block Turbo Codes (BTC)

2.1 Likelihood Functions

Most of the different iterative decoding algorithms used on turbo codes uses log-likelihood algebra in the decoding process. This is a brief introduction on the subject required.

The mathematical foundations of hypothesis testing rest on Bayes' theorem. For communications engineering, where applications involving an AWGN channel are of great interest, the most useful form of Bayes' theorem expresses the *a posteriori probability* (*APP*) of a decision in terms of a continuous-valued random variable *x* in the following form:

$$P(d = i \mid x) = \frac{P(x \mid d = i)\, P(d=i)}{P(x)}, \quad i = 1, \ldots, M \tag{1}$$

and

$$P(x) = \sum_{i=1}^{M} P(x \mid d = i)\, P(d = i) \tag{2}$$

Where $P(d = i \mid x)$ is the APP, and $d = i$ represents data d belonging to the ith signal class from a set of M classes. Further, $p(x \mid d = i)$ represents the probability density function (pdf) of a received continuous-valued data-plus-noise signal x, conditioned on the signal class $d = i$. Also, $P(d = i)$, called the *a priori probability*, is the probability of occurrence of the ith signal class. Typically x is an "observable" random variable or a test statistic that is obtained at the output of a demodulator or some other signal processor.

Therefore, $p(x)$ is the pdf of the received signal x, yielding the test statistic over the entire space of signal classes. In Equation (1), for a particular observation, $p(x)$ is a scaling factor, since it is obtained by averaging over all the classes in the space. Lowercase p is used to designate the pdf of a continuous-valued random variable, and uppercase P is used to designate probability (a priori and APP). Determining the APP of a received signal from Equation (1) can be thought of as the result of an experiment. Before the experiment, there generally exists (or one can estimate) an a priori probability $P(d = i)$. The experiment consists of using Equation (1) for computing the APP, $P(d = i \mid x)$, which can be thought of as a "refinement" of the prior knowledge about the data, brought about by examining the received signal x.

2.1.1 The Two Signal Class Case

Let the binary logical elements 1 and 0 be represented electronically by voltages +1 and -1, respectively. The variable d is used to represent the transmitted data bit, whether it appears as a voltage or as a logical element. Sometimes one format is more convenient than the other; the reader should be able to recognize the difference from the context. Let the binary 0 (or the voltage value -1) be the null element under addition. For signal transmission over an AWGN channel, Figure (12) shows the conditional pdfs referred to as *likelihood functions*. The rightmost function, $P(x \mid d = +1)$, shows the pdf of the random variable x conditioned on $d = +1$ being

transmitted. The leftmost function, $P(x \mid d = -1)$, illustrates a similar pdf conditioned on $d = -1$ being transmitted. The abscissa represents the full range of possible values of the test statistic x generated at the receiver. In Figure (12), one such arbitrary value x_k is shown, where the index denotes an observation in the k^{th} time interval. A line subtended from x_k intercepts the two likelihood functions, yielding two likelihood values $\ell_1 = P(x_k \mid d_k = +1)$ and $\ell_2 = P(x_k \mid d_k = -1)$. A well-known hard decision rule, known as *maximum likelihood* (ML), is to choose the data d_k=+1 or d_k= -1 associated with the larger of the two intercept values, ℓ_1 or ℓ_2, respectively. For each data bit at time k, this is tantamount to deciding that d_k= +1 if x_k falls on the right side of the decision line labeled γ_0, otherwise deciding that d_k= -1.

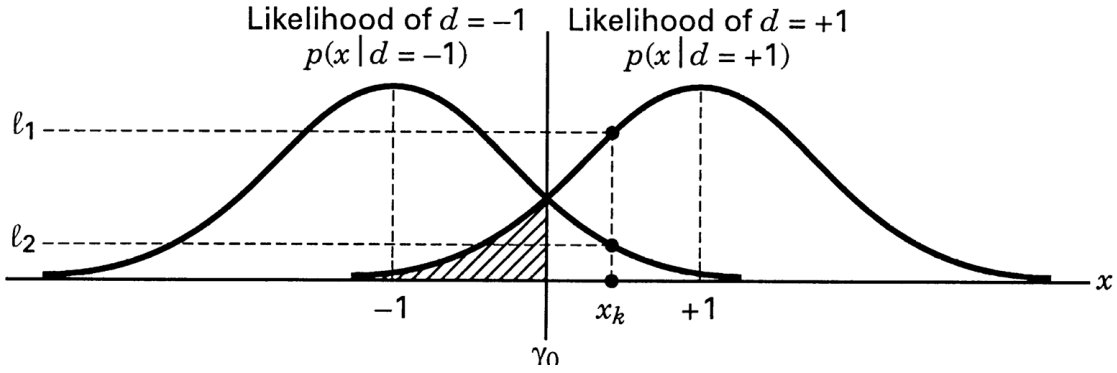

Figure (12) : Likelihood function

A similar decision rule, known as *maximum a posteriori* (MAP), which can be shown to be a *minimum probability of error rule*,

takes into account the a priori probabilities of the data. The general expression for the MAP rule in terms of APPs is as follows:

$$P\,(\,d = +1 \mid x\,) \underset{H_1}{\overset{H_0}{\gtrless}} P\,(\,d = -1 \mid x\,) \tag{3}$$

Equation (3) states that you should choose the hypothesis H_0, (d = +1), if the APP $P(d = +1 \mid x)$, is greater than the APP $P(d = -1 \mid x)$. Otherwise, you should choose hypothesis H_1, (d = -1). Using the Bayes' theorem of Equation (1), the APPs in Equation (3) can be replaced by their equivalent expressions, yielding the following:

$$P\,(\,x \mid d = +1)\,P(d = +1\,) \underset{H_1}{\overset{H_0}{\gtrless}} P\,(\,x \mid d = -1)\,P(d = -1\,) \tag{4}$$

where the pdf $p(x)$ appearing on both sides of the inequality in Equation (1) has been canceled. Equation (4) is generally expressed in terms of a ratio, yielding the so-called *likelihood ratio test*, as follows:

$$\frac{P(x \mid d = +1)}{P(x \mid d = -1)} \underset{H_1}{\overset{H_0}{\gtrless}} \frac{P(d = -1)}{P(d = +1)} \quad or \quad \frac{P(x \mid d = +1)\,P(d = +1)}{P(x \mid d = -1)P(d = -1)} \underset{H_1}{\overset{H_0}{\gtrless}} 1 \tag{5}$$

2.1.2 Log-Likelihood Ratio

By taking the logarithm of the likelihood ratio developed in Equations (3) through (5), we obtain a useful metric called the *log-likelihood ratio* (*LLR*). It is a real number representing a soft decision out of a detector, designated by as follows:

$$L(d \mid x) = log\left[\frac{P(d=+1 \mid x\ )}{P(d=-1 \mid x\ )}\right] = log\left[\frac{P(x \mid d=+1\ )P(d=+1)}{P(x \mid d=-1\ )P(d=-1)}\right] \quad (6)$$

$$L(d \mid x) = log\left[\frac{P(x \mid d=+1\ )}{P(x \mid d=-1\ )}\right] + log\left[\frac{P(d=+1)}{P(d=-1)}\right] \quad (7)$$

$$L(d \mid x) = L(x \mid d) + L(d) \quad (8)$$

Where $L(x|d)$ is the LLR of the test statistic x obtained by measurements of the channel output x under the alternate conditions that d = +1 or d = -1 may have been transmitted, and $L(d)$ is the a priori LLR of the data bit d.

To simplify the notation , equation (8) is rewritten as follows:

$$L'(\hat{d}) = L_c(x) + L(d) \quad (9)$$

where the notation $L_c(x)$ emphasizes that this LLR term is the result of a channel measurement made at the receiver. Equations (1) through (9) were developed with only a data detector in mind. Next, the introduction of a decoder will typically yield decision-making benefits. For a systematic code, it can be shown [3] that the LLR (soft output) $L(\hat{d})$ out of the decoder is equal to Equation 10:

$$L(\hat{d}) = L'(\hat{d}) + L_e(\hat{d}\ ) \quad (10)$$

Where $L'(\hat{d})$ is the LLR of a data bit out of the demodulator (input to the decoder), and $L_e(\hat{d})$, called the *extrinsic* LLR, represents extra knowledge gleaned from the decoding process. The output sequence of a systematic decoder is made up of values representing data bits and parity bits. From Equations (9) and (10), the output LLR $L(\hat{d})$ of the decoder is now written as follows:

$$L(\hat{d}) = L_c(x) + L(d) + L_e(\hat{d}) \quad (11)$$

Equation (11) shows that the output LLR of a systematic decoder can be represented as having three LLR elements: a channel measurement, a priori knowledge of the data, and an extrinsic LLR stemming solely from the decoder. To yield the final $L(\hat{d})$, each of the individual LLRs can be added as shown in Equation (11), because the three terms are statistically independent [3, 13]. This soft decoder output $L(\hat{d})$ is a real number that provides a hard decision as well as the reliability of that decision. The sign of $L(\hat{d})$ denotes the hard decision; that is, for positive values of $L(\hat{d})$ decide that d = +1, and for negative values decide that d = -1. The magnitude of $L(\hat{d})$ denotes the reliability of that decision. Often, the value of $L_e(\hat{d})$ due to the decoding has the same sign as $L_c(x) + L(d)$,and therefore acts to improve the reliability of $L(\hat{d})$.

2.1.3 Principles of Iterative (Turbo) Decoding

In a typical communications receiver, a demodulator is often designed to produce soft decisions, which are then transferred to a decoder. The error-performance improvement of systems utilizing such soft decisions compared to hard decisions is typically approximated as 2 dB in AWGN. Such a decoder could be called a *soft input/hard output* decoder, because the final decoding process out of the decoder must terminate in bits (hard decisions). With turbo codes, where two or more component codes are used, and decoding involves feeding outputs from one decoder to the inputs of

other decoders in an iterative fashion, a hard-output decoder would not be suitable. That is because hard decisions into a decoder degrade system performance (compared to soft decisions). Hence, what is needed for the decoding of turbo codes is a *soft input/soft output* decoder. For the first decoding iteration of such a soft input/soft output decoder, illustrated in Figure(13), we generally assume the binary data to be equally likely, yielding an initial a priori LLR value of $L(d) = 0$ for the third term in Equation (7). The channel LLR value, $L_c(x)$, is measured by forming the logarithm of the ratio of the values of ℓ_1 and ℓ_2 for a particular observation of x (see Figure (12)), which appears as the second term in Equation (7). The output $L(\hat{d})$ of the decoder in Figure (13) is made up of the LLR from the detector, $L'(\hat{d})$, and the extrinsic LLR output, $L_e(\hat{d})$ representing knowledge gleaned from the decoding process as in equation (10). As illustrated in Figure (13), for iterative decoding, the extrinsic likelihood is fed back to the decoder input, to serve as a refinement of the a priori probability of the data for the next iteration.

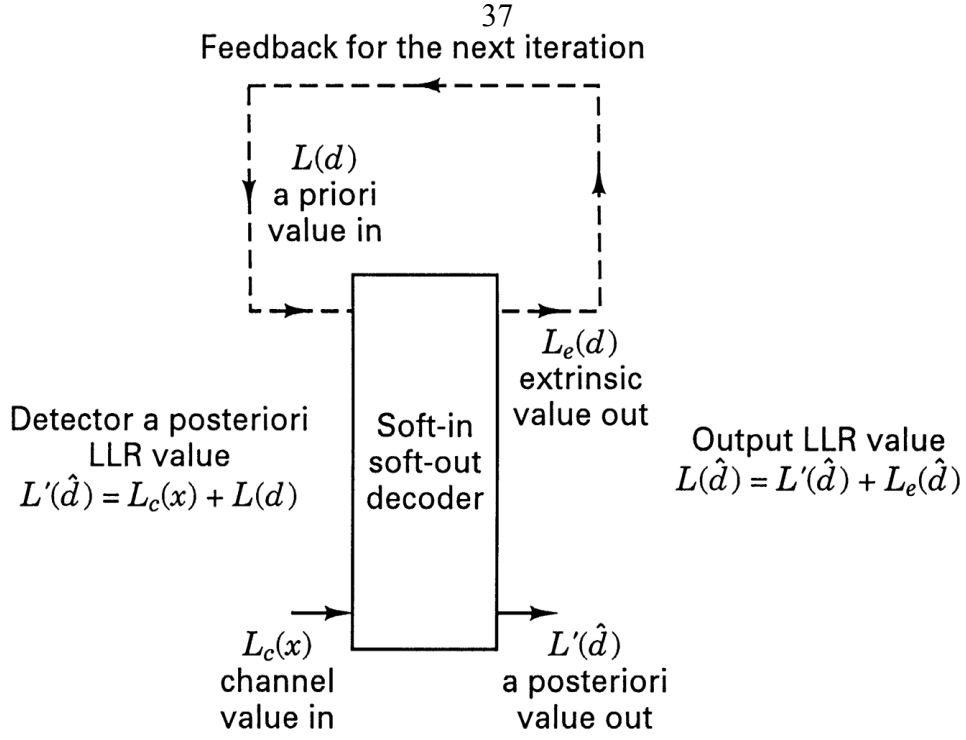

Figure (13) : soft input/soft output decoder (for a systematic code).

2.2 Log-Likelihood Algebra

To best explain the iterative feedback of soft decoder outputs, the concept of loglikelihood algebra [13] is introduced. For statistically independent data *d*, the sum of two log-likelihood ratios (LLRs) is defined as follows:

$$L(d_1) \boxplus L(d_2) \equiv L(d_1 \oplus d_2) = \ln\left[\frac{e^{L(d_1)} + e^{L(d_2)}}{1 + e^{L(d_1)} e^{L(d_2)}}\right] \quad (12)$$

$$\approx (-1) \times \operatorname{sgn}[L(d_1)] \times \operatorname{sgn}[L(d_2)] \times \min\{|L(d_1)|, |L(d_2)|\} \quad (13)$$

Where the natural logarithm is used, and the function sign (·) represents "the polarity of." There are three addition operations in Equation (12). The + sign is used for ordinary addition. The $\oplus$ sign is used to denote the modulo-2 sum of data expressed as binary digits. The $\boxplus$ sign denotes log-likelihood addition or, equivalently, the mathematical operation described by Equation (12). The sum of

two LLRs denoted by the operator ⊞is defined as the LLR of the modulo-2 sum of the underlying statistically independent data bits [13]. Equation (13) is an approximation of Equation (12) that will prove useful later in a numerical example. The sum of LLRs as described by Equations (12) or (13) yields the following interesting results when one of the LLRs is very large or very small:

$$L(d) \boxplus \infty = -L(d)$$

and

$$L(d) \boxplus 0 = 0$$

Note that the log-likelihood algebra described here differs slightly from that used in [13] because of a different choice of the null element. In this treatment, the null element of the binary set (1, 0) has been chosen to be 0.

2.2.1 Single Parity Check Product Code (Example)

Consider the two-dimensional code (product code) depicted in the below figure. The configuration can be described as a data array made up of k_1 rows and k_2 columns. The k_1 rows contain codewords made up of k_2 data bits and n_2 - k_2 parity bits. Thus, each row (except the last ones) represents a codeword from an (n_2, k_2) code. Similarly, the k_2 columns contain codewords made up of k_1 data bits and n_1 - k_1 parity bits. Thus, each column (except the last ones) represents a codeword from an (n_1, k_1) code. The various portions of the structure are labeled *d* for data, p_h for horizontal parity (along the rows), and p_v for vertical parity (along the columns). In

effect, the block of k_1 x k_2 data bits is encoded with two codes—a horizontal code, and a vertical code. Additionally, in Figure (14) , there are blocks labeled L_{eh} and L_{ev} that contain the extrinsic LLR values learned from the horizontal and vertical decoding steps, respectively. Error-correction codes generally provide some improved performance. We will see that the extrinsic LLRs represent a measure of that improvement. Notice that this product code is a simple example of a concatenated code. Its structure encompasses two separate encoding steps—horizontal and vertical.

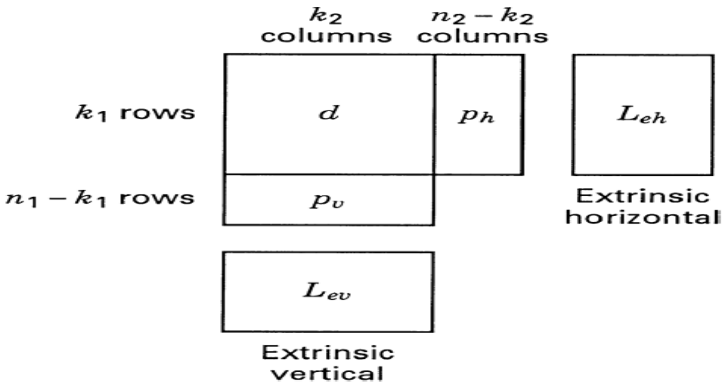

Figure (14) : Product code

2.2.2 Iterative Decoding Algorithm for Product Codes

This algorithm can be found in [5]:

1. Set the a priori LLR $L(d) = 0$ (unless the a priori probabilities of the data bits are other than equally likely).
2. Decode horizontally, and using Equation (11) obtain the horizontal extrinsic LLR as shown below:

$$L_{eh}(\hat{d}) = L(\hat{d}) - L_c(x) - L(d)$$

3. Set $L(d) = L_{eh}(\hat{d})$ for the vertical decoding of step 4.
4. Decode vertically, and using Equation (11) obtain the vertical extrinsic LLR as shown below:

$$L_{ev}(\hat{d}) = L(\hat{d}) - L_c(x) - L(d)$$

5. Set $L(d) = L_{ev}(\hat{d})$ and repeat steps 2 through 5.
6. After enough iterations (that is, repetitions of steps 2 through 5) to yield a reliable decision, go to step 7.
7. The soft output is

$$L(\hat{d}) = L_c(x) + L_{eh}(\hat{d}) + L_{ev}(\hat{d}) \tag{14}$$

An example is next used to demonstrate the application of this algorithm to a very simple product code.

2.2.3 Two-Dimensional Single-Parity Code Example

At the encoder, let the data bits and parity bits take on the values shown in Figure 15(a), where the relationships between data and parity bits within a particular row (or column) expressed as the binary digits (1 , 0) are as follows:

$$\begin{cases} p_{12} = d_1 \oplus d_2 \\ p_{34} = d_3 \oplus d_4 \\ p_{13} = d_1 \oplus d_3 \\ p_{24} = d_2 \oplus d_4 \end{cases} \tag{15}$$

Where $\oplus$ denotes modulo-2 addition. The transmitted bits are represented by the sequence d_1 d_2 d_3 d_4 p_{12} p_{34} p_{13} p_{24}. At the receiver input, the noise-corrupted bits are represented by the sequence $\{x_i\}$, $\{x_{ij}\}$, where $x_i = d_i + n$ for each received data bit, d_i &

$x_{ij} = p_{ij} + n$ for each received parity bit, and n represents the noise contribution that is statistically independent for both d_i and p_{ij}. The indices i and j represent position in the encoder output array shown in Figure 15(a). However, it is often more useful to denote the received sequence as $\{x_k\}$, where k is a time index. Both conventions will be followed below—using i and j when focusing on the positional relationships within the product code, and using k when focusing on the more general aspect of a time-related signal. The distinction as to which convention is being used should be clear from the context. Using the relationships developed in Equations (7) through (9), and assuming an AWGN interference model, the LLR for the channel measurement of a signal x_k received at time k is written as follows:

$$L_c(x_k) = ln\left[\frac{p(x_k|d_k=+1)}{p(x_k|d_k=-1)}\right] \tag{16a}$$

$$= ln\left[\frac{\frac{1}{\sigma\sqrt{2\pi}}exp\left(-\frac{1}{2}\left(\frac{x_k-1}{\sigma}\right)^2\right)}{\frac{1}{\sigma\sqrt{2\pi}}exp\left(-\frac{1}{2}\left(\frac{x_k+1}{\sigma}\right)^2\right)}\right] \tag{16b}$$

$$= -\frac{1}{2}\left(\frac{x_k-1}{\sigma}\right)^2 + \frac{1}{2}\left(\frac{x_k+1}{\sigma}\right)^2 = \frac{2}{\sigma^2}x_k \tag{16c}$$

Where the natural logarithm is used, if a further simplifying assumption is made that the noise variance σ^2 is unity, then

$$L_c(x_k) = 2x_k \tag{17}$$

Consider the following example, where the data sequence $d_1d_2d_3d_4$ is made up of the binary digits 1 0 0 1, as shown in Figure (15). By the use of Equation (15), it is seen that the parity sequence

$p_{12}p_{34}p_{13}p_{24}$ must be equal to the digits 1 1 1 1. Thus, the transmitted sequence is

$$\{d_i\}\ ,\ \{p_{ij}\} = 1\ 0\ 0\ 1\ 1\ 1\ 1\ 1 \tag{18}$$

When the data bits are expressed as bipolar voltage values of +1 and -1 corresponding to the binary logic levels 1 and 0, the transmitted sequence is:

$$\{d_i\}\ ,\ \{p_{ij}\} = +1\ -1\ -1\ +1\ +1\ +1\ +1\ +1$$

Assume now that the noise transforms this data-plus-parity sequence into the receive sequence

$$\{x_i\}\ ,\ \{x_{ij}\} = 0.75,\ 0.05,\ 0.10,\ 0.15,\ 1.25,\ 1.0,\ 3.0,\ 0.5 \tag{19}$$

Where the members of $\{x_i\}$, $\{x_{ij}\}$ positionally correspond to the data and parity $\{d_i\}$, $\{p_{ij}\}$ that was transmitted. Thus, in terms of the positional subscripts, the received sequence can be denoted as:

$$\{x_i\}\ ,\ \{x_{ij}\} = x_1,\ x_2,\ x_3,\ x_4,\ x_{12},\ x_{34},\ x_{13},\ x_{24}$$

From Equation (17), the assumed channel measurements yield the LLR values

$$\{L_c(x_i)\}\ ,\ \{L_c(x_{ij})\} = 1.5,\ 0.1,\ 0.20,\ 0.3,\ 2.5,\ 2.0,\ 6.0,\ 1.0 \tag{20}$$

These values are shown in Figure (15b) as the decoder input measurements. It should be noted that, given equal prior probabilities for the transmitted data, if hard decisions are made based on the $\{x_k\}$ or the $\{L_c(x_k)\}$ values shown above, such a

process would result in two errors, since d_2 and d_3 would each be incorrectly classified as binary 1.

$d_1 = 1$	$d_2 = 0$	$p_{12} = 1$
$d_3 = 0$	$d_4 = 1$	$p_{34} = 1$
$p_{13} = 1$	$p_{24} = 1$	

(a) Encoder output binary digits

$L_c\,(x_1) = 1.5$	$L_c\,(x_2) = 0.1$	$L_c\,(x_{12}) = 2.5$
$L_c\,(x_3) = 0.2$	$L_c\,(x_4) = 0.3$	$L_c\,(x_{34}) = 2.0$
$L_c\,(x_{13}) = 6.0$	$L_c\,(x_{24}) = 1.0$	

(b) Decoder input log-likelihood ratios $L_c\,(x)$

Figure (15) : Product code example.

2.2.4 Extrinsic Likelihoods

For the product-code example in Figure (15), we use Equation (11) to express the soft output $L(\hat{d})$ for the received signal corresponding to data d_1, as follows:

$$L(\hat{d}) = \mathrm{L_c}(x_1) + \mathrm{L}(d_1) + \{[\,\mathrm{L_c}(x_2) + \mathrm{L}(d_2)\,] \boxplus \mathrm{L_c}(x_{12})\,\} \tag{21}$$

where the terms $\{[\,\mathrm{L_c}(x_2) + \mathrm{L}(d_2)\,] \boxplus \mathrm{L_c}(x_{12})\,\}$ represent the extrinsic LLR contributed by the code (that is, the reception corresponding to data d_2 and it's a priori probability, in conjunction with the reception corresponding to parity P_{12}. In general, the soft output $L\,(\hat{d})$ for the received signal corresponding to data d_i is:

$$L(\hat{d}) = \mathrm{L_c}(x_i) + \mathrm{L}(d_i) + \{[\,\mathrm{L_c}(x_j) + \mathrm{L}(d_j)\,] \boxplus \mathrm{L_c}(x_{ij})\,\} \tag{22}$$

Where $L_c(x_i)$, $L_c(x_j)$, and $L_c(x_{ij})$ are the channel LLR measurements of the reception corresponding to d_i, d_j, and p_{ij}, respectively. $L(d_i)$ and $L(d_j)$ are the LLRs of the a priori probabilities of d_i and d_j, respectively, and

$\{[\ L_c(x_j) + L(d_j)\ ] \boxplus L_c(x_{ij})\ \}$ is the extrinsic LLR contribution from the code. Equations (21) and (22) can best be understood in the context of Figure (15)b. For this example, assuming equally-likely signaling, the soft output $L(\hat{d})$is represented by the detector LLR measurement of $L_c(x_1) = 1.5$ for the reception corresponding to data d_1, plus the extrinsic LLR of $[L_c(x_2) = 0.1] \boxplus [L_c(x_{12}) = 2.5]$ gleaned from the fact that the data d_2 and the parity p_{12} also provide knowledge about the data d_1, as seen from Equations (15).

2.2.5 Computing the Extrinsic Likelihoods

For the example in Figure (15), the horizontal calculations for $L_{eh}(\hat{d})$ and the vertical calculations for $L_{ev}(\hat{d})$ are expressed as follows:

$$L_{eh}(\hat{d}_1) = [\ L_c(x_2) + L(\hat{d}_2)] \boxplus L_c(x_{12}) \quad (23a)$$

$$L_{ev}(\hat{d}_1) = [\ L_c(x_3) + L(\hat{d}_3)] \boxplus L_c(x_{13}) \quad (23b)$$

$$L_{eh}(\hat{d}_2) = [\ L_c(x_1) + L(\hat{d}_1)] \boxplus L_c(x_{12}) \quad (24a)$$

$$L_{ev}(\hat{d}_2) = [\ L_c(x_4) + L(\hat{d}_4)] \boxplus L_c(x_{24}) \quad (24b)$$

$$L_{eh}(\hat{d}_3) = [\ L_c(x_4) + L(\hat{d}_4)] \boxplus L_c(x_{34}) \quad (25a)$$

$$L_{ev}(\hat{d}_3) = [\ L_c(x_1) + L(\hat{d}_1)] \boxplus L_c(x_{13}) \quad (25b)$$

$$L_{eh}(\hat{d}_4) = [\ L_c(x_3) + L(\hat{d}_3)] \boxplus L_c(x_{34}) \quad (26a)$$

$$L_{ev}(\hat{d}_4) = [\ L_c(x_2) + L(\hat{d}_2)] \boxplus L_c(x_{24}) \quad (26b)$$

The LLR values shown in Figure (15) are entered into the $L_{eh}(\hat{d})$ expressions in Equations (23) through (26) and, assuming equally-likely signaling, the $L(d)$ values are initially set equal to zero, yielding (using eqn. (13))

$$L_{eh}(\hat{d}_1) = (0.1 + 0) \boxplus 2.5 \approx -0.1 = \text{new } L(d_1) \quad (27)$$

$$L_{eh}(\hat{d}_2) = (1.5 + 0) \boxplus 2.5 \approx -1.5 = \text{new } L(d_2) \quad (28)$$

$$L_{eh}(\hat{d}_3) = (0.3 + 0) \boxplus 2.0 \approx -0.3 = \text{new } L(d_3) \quad (39)$$

$$L_{eh}(\hat{d}_4) = (0.2 + 0) \boxplus 2.0 \approx -0.2 = \text{new } L(d_4) \quad (30)$$

Where the log-likelihood addition has been calculated using the approximation in Equation (13). Next, we proceed to obtain the first vertical calculations using the $L_{ev}(\hat{d})$ expressions in Equations (23) through (26). Now, the values of $L(d)$ can be refined by using the new $L(d)$ values gleaned from the first horizontal calculations, shown in Equations (27) through (30). That is,

$$L_{ev}(\hat{d}_1) = (0.2 - 0.3) \boxplus 6 \approx 0.1 = \text{new } L(d_1) \quad (31)$$

$$L_{ev}(\hat{d}_2) = (0.3 - 0.2) \boxplus 1.0 \approx -0.1 = \text{new } L(d_2) \quad (32)$$

$$L_{ev}(\hat{d}_3) = (1.5 - 0.1) \boxplus 6.0 \approx -1.4 = \text{new } L(d_3) \quad (33)$$

$$L_{ev}(\hat{d}_4) = (0.1 - 1.5) \boxplus 1.0 \approx 1.0 = \text{new } L(d_4) \quad (34)$$

The results of the first full iteration of the two decoding steps (horizontal and vertical) are shown below.

1.5	0.1
0.2	0.3

Original $L_c(x_k)$ measurements

-0.1	-1.5
-0.3	-0.2

$L_{eh}(\hat{d})$ after first horizontal decoding

0.1	-0.1
-1.4	1.0

$L_{ev}(\hat{d})$ after first vertical decoding

Each decoding step improves the original LLRs that are based on channel measurements only. This is seen by calculating the decoder output LLR using Equation (14). The original LLR plus the horizontal extrinsic LLRs yields the following improvement (the extrinsic vertical terms are not yet being considered):

1.4	-1.4
-0.1	0.1

Improved LLRs due to $L_{eh}(\hat{d})$

The original LLR plus both the horizontal and vertical extrinsic LLRs yield the following improvement:

1.5	-1.5
-1.5	1.1

Improved LLRs due to $L_{eh}(\hat{d})$+ $L_{ev}(\hat{d})$

For this example, the knowledge gained from horizontal decoding alone is sufficient to yield the correct hard decisions out of the decoder, but with very low confidence for data bits d_3 and d_4. After incorporating the vertical extrinsic LLRs into the decoder, the new LLR values exhibit a higher level of reliability or confidence. Let's pursue one additional horizontal and vertical decoding iteration to

determine if there are any significant changes in the results. We again use the relationships shown in Equations (23) through (26) and proceed with the second horizontal calculations for $L_{eh}(\hat{d})$, using the new $L(d)$ from the first vertical calculations shown in Equations (31) through (34), so that

$$L_{eh}(\hat{d}_1) = (0.1 - 0.1) \boxplus 2.5 \approx 0 = \text{new } L(d_1) \quad (35)$$

$$L_{eh}(\hat{d}_2) = (1.5 + 0.1) \boxplus 2.5 \approx -1.6 = \text{new } L(d_2) \quad (36)$$

$$L_{eh}(\hat{d}_3) = (0.3 + 1) \boxplus 2.0 \approx -1.3 = \text{new } L(d_3) \quad (37)$$

$$L_{eh}(\hat{d}_4) = (0.2 - 1.4) \boxplus 2.0 \approx 1.2 = \text{new } L(d_4) \quad (38)$$

Next, we proceed with the second vertical calculations for $L_{ev}v$, using the new $L(d)$ from the second horizontal calculations, shown in Equations (35) through (38). This yields

$$L_{ev}(\hat{d}_1) = (0.2 - 1.3) \boxplus 6 \approx 1.1 = \text{new } L(d_1) \quad (39)$$

$$L_{ev}(\hat{d}_2) = (0.3 + 1.2) \boxplus 1.0 \approx -1.0 = \text{new } L(d_2) \quad (40)$$

$$L_{ev}(\hat{d}_3) = (1.5 + 0) \boxplus 6.0 \approx -1.5 = \text{new } L(d_3) \quad (41)$$

$$L_{ev}(\hat{d}_4) = (0.1 - 1.6) \boxplus 1.0 \approx 1.0 = \text{new } L(d_4) \quad (42)$$

The second iteration of horizontal and vertical decoding yielding the above values results in soft-output LLRs that are again calculated from Equation (14), which is rewritten below:

$$L(\hat{d}) = Lc(x) + L_{eh}(\hat{d}) + L_{ev}(\hat{d}) \quad (43)$$

The horizontal and vertical extrinsic LLRs of Equations (35) through (42) and the resulting decoder LLRs are displayed below.

For this example, the second horizontal and vertical iteration (yielding a total of four iterations) suggests a modest improvement over a single horizontal and vertical iteration. The results show a balancing of the confidence values among each of the four data decisions.

1.5	0.1
0.2	0.3

Original $L_c(x_k)$ measurements

0	-1.6
-1.3	1.2

$L_{eh}(\hat{d})$after second horizontal decoding

1.1	-0.1
-1.5	1.0

$L_{ev}(\hat{d})$ after second vertical decoding

The soft output is

$$L(\hat{d}) = L_c(x) + L_{eh}(\hat{d}) + L_{ev}(\hat{d})$$

Which after two iterations yields the following values for $L(\hat{d})$

2.6	-2.5
-2.6	2.5

Observe that correct decisions about the four data bits will result, and the level of confidence about these decisions is high. The iterative decoding of turbo codes is similar to the process used when solving a crossword puzzle. The first pass through the puzzle is likely to contain a few errors. Some words seem to fit, but when

the letters intersecting a row and column do not match, it is necessary to go back and correct the first-pass answers.

2.3 Modification

An AWGN interference model was used for the channel, under this and using the Iterative Decoding Algorithm for product codes, an algorithm was written and a code using Matlab was developed, our computation shows the following results:

First for an AWGN interference model with normal distribution having unity variance "$\sigma^2 = 1$" the program was run for one iteration and the result was as shown in the following table for L(d)

$$L(d) = \begin{bmatrix} 1.5 & -1.5 \\ -1.5 & 1.5 \end{bmatrix}$$

Then after 4 iterations, the following values for L(d) were obtained

$$L(d) = \begin{bmatrix} 3.5 & -3.4 \\ -3.5 & 3.3 \end{bmatrix}$$

Also, after 1000 iterations, we find that the values are stable with:

$$L(d) = \begin{bmatrix} 3.5 & -3.4 \\ -3.5 & 3.3 \end{bmatrix}$$

We notice that the algorithm was stable after 4 iterations with:

$$L(d) = \begin{bmatrix} 3.5 & -3.4 \\ -3.5 & 3.3 \end{bmatrix}$$

Secondly a comparison using the logistic distribution with variance 1 instead of the normal distribution had been done.

Remark 2.1: The logistic distribution has a P.d.f

$$\text{P.d.f : } f(x) = \frac{e^{-\left(\frac{x-\mu}{\beta}\right)}}{\beta.\left(1+e^{-\frac{x-\mu}{\beta}}\right)^2} \quad , \quad \begin{matrix} -\infty < x < \infty \\ -\infty < \mu < \infty \\ \beta > 0 \end{matrix}$$

Where μ is the mean and the variance is $\sigma^2 = \frac{\pi^2 \beta^2}{3}$

$$\ln\left(\frac{pr(x=1)}{pr(x=-1)}\right) = \ln\left(\frac{e^{-\left(\frac{x-1}{\beta}\right)}}{\beta.\left(1+e^{-\frac{x-1}{\beta}}\right)^2} \quad . \frac{\beta.\left(1+e^{-\frac{x+1}{\beta}}\right)^2}{e^{-\left(\frac{x+1}{\beta}\right)}} \right)$$

$$= \ln\left(e^{-\frac{x}{\beta}+\frac{1}{\beta}+\frac{x}{\beta}+\frac{1}{\beta}} \quad . \frac{\left(1+e^{-\frac{x+1}{\beta}}\right)^2}{\left(1+e^{-\frac{x-1}{\beta}}\right)^2} \right)$$

$$= \frac{2}{\beta} + 2 \ln\left(\frac{\left(1+e^{-\frac{x+1}{\beta}}\right)}{\left(1+e^{-\frac{x-1}{\beta}}\right)} \right) \qquad (44)$$

When $\sigma^2 = 1$ then $\beta = \frac{\sqrt{3}}{\pi}$.

We ran the program using the logistic distribution under the condition mentioned before, after one iteration we found that:

$$L(d) = \begin{bmatrix} 1.8181 & -1.8181 \\ -1.8181 & 1.3927 \end{bmatrix}$$

Also, it is found to be stable after 4 iterations (see Appendix B) with:

$$L(d) = \begin{bmatrix} 4.3440 & -3.8098 \\ -4.3440 & 3.9472 \end{bmatrix}$$

Our computations show that the logistic distribution is more performance than the normal distribution, the results is shown in the following table:

channel / L(d)	AWGN channel (normal distribution)	Channel with logistic distribution
L(d)	$L(d) = \begin{bmatrix} 3.5 & -3.4 \\ -3.5 & 3.3 \end{bmatrix}$	$L(d) = \begin{bmatrix} 4.3440 & -3.8098 \\ -4.3440 & 3.9472 \end{bmatrix}$

Notice that with variance 1 the logistic distribution is approximately a normal distribution with the same variance value.

In all the previous calculations an AWGN channel with variance 1 was used. Now we will make a modification; a normal distribution with variance $\sigma^2 = \frac{\pi}{\sqrt{3}}$ will be used to make calculations:

We use equation (17c), hence, $Lc(x_i) = 2\frac{\pi}{\sqrt{3}}x_i$, using the Matlab code we get:

1- After one iteration:

$$L(d) = \begin{bmatrix} 2.72 & -2.72 \\ -2.72 & 1.99 \end{bmatrix}$$

2- After 4 iterations:

$$L(d) = \begin{bmatrix} 6.348 & -6.167 \\ -6.348 & 5.985 \end{bmatrix}$$

Remark 2.2 One can verify that decreasing the variance values gives better results, But still with performance to logistic distribution over normal distribution.

2.4 Encoding with Recursive Systematic Codes

The basic concepts of concatenation, iteration, and soft-decision decoding using a simple product-code example have been described. These ideas are next applied to the implementation of turbo codes that are formed by the parallel concatenation of component convolutional codes [3, 7].

Definition: Finite Impulse Response (FIR): the output is a weighted sum of the current and a finite number of previous values of the input.

A short review of simple binary rate 1/2 convolutional encoders with constraint length K and memory K-1 is in order. The input to the encoder at time k is a bit d_k, and the corresponding codeword is the bit pair (u_k, v_k), such that

$$u_k = \sum_{i=1}^{k-1} g_{1i}\, d_{k-i} \qquad \text{mod } 2\ ;\ \ g_{1i} = 0,1 \tag{45}$$

$$v_k = \sum_{i=1}^{k-1} g_{2i}\, d_{k-i} \qquad \text{mod } 2\ ;\ \ g_{2i} = 0,1 \tag{46}$$

Where **G1** = $\{g_{1i}\}$ and **G2** = $\{g_{2i}\}$ are the code generators, and d_k is represented as a binary digit. This encoder can be visualized as a discrete-time finite impulse response (FIR) linear system, giving rise to the familiar **nonsystematic convolutional (NSC)** code, ***an example*** of which is shown in Figure (16). In this example, the constraint length is K = 3, and the two code generators are described by **G1** = { 1 1 1 } and **G2** = { 1 0 1 }. It is well known that at large E_b/N_0 (the energy per bit to noise power spectral density, for more details we refer the reader to [29]) values, the error performance of an NSC is better than that of a systematic code having the same memory. At small E_b/N_0 values, it is generally the other way around [3]. A class of **infinite impulse response (IIR)** convolutional codes [3] has been proposed as building blocks for a turbo code. Such building blocks are also referred to as **recursive systematic convolutional (RSC)** codes because previously encoded information bits are continually fed back to the encoder's input. For high code rates, RSC codes result in better error

performance than the best NSC codes at any value of E_b/N_0. A binary, rate 1/2, RSC code is obtained from an NSC code by using a feedback loop and setting one of the two outputs (u_k or v_k) equal to d_k. Figure 17(a) illustrates an example of such an RSC code, with K = 3, where a_k is recursively calculated as

$$a_k = d_k + \sum_{i=1}^{k-1} g'_i \, a_{k-i} \mod 2 \qquad (47)$$

and g'_i is respectively equal to g_{1i} if $u_k = d_k$, and to g_{2i} if $v_k = d_k$. Figure 17(b) shows the trellis structure for the RSC code in Figure 17(a).

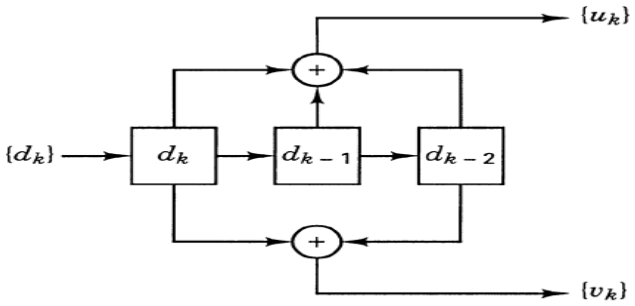

Figure (16) : Nonsystematic convolutional (NSC) code.

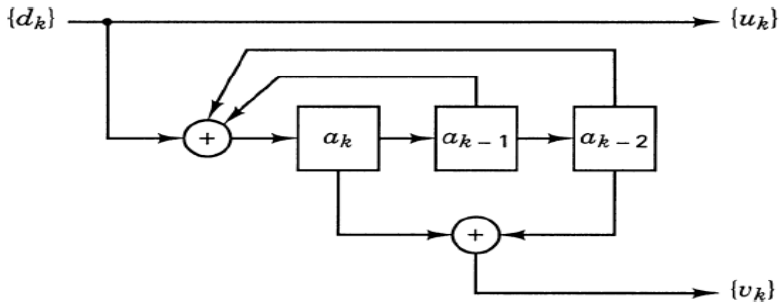

Figure 17(a) :Recursive systematic convolutional (RSC) code.

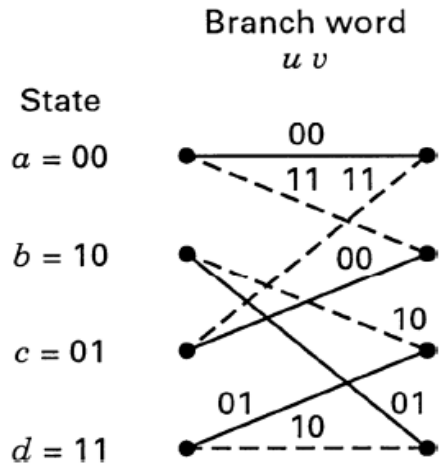

Figure 17(b): Trellis structure for the RSC code in Figure 17(a).

It is assumed that an input bit d_k takes on values of 1 or 0 with equal probability. Furthermore, $\{a_k\}$ exhibits the same statistical properties as $\{ d_k \}$ [3]. The free distance is identical for the RSC code of Figure (16) and the NSC code of Figure (16). Similarly, their trellis structures are identical with respect to state transitions and their corresponding output bits. However, the two output sequences $\{u_k\}$ and $\{v_k\}$ do not correspond to the same input sequence $\{d_k\}$ for RSC and NSC codes. For the same code generators, it can be said that the weight distribution of the output codewords from an RSC encoder is not modified compared to the weight distribution from the NSC counterpart. The only change is the mapping between input data sequences and output codeword sequences.

2.4.1 Example: Recursive Encoders and Their Trellis Diagrams

a) Using the RSC encoder in Figure 17(a), verify the section of the trellis structure (diagram) shown in Figure 17(b).

b) For the encoder in part a, start with the input data sequence $\{d_k\}$ = 1 1 1 0, and show the step-by-step encoder procedure for finding the output codeword.

Solution

a) For NSC encoders, keeping track of the register contents and state transitions is a straightforward procedure. However, when the encoders are recursive, more care must be taken. Table 1 is made up of eight rows corresponding to the eight possible transitions in this four-state machine. The first four rows represent transitions when the input data bit, d_k, is a binary zero, and the last four rows represent transitions when $d_k$is a one. For this example, the step-by-step encoding procedure can be described with reference to Table 1 and Figure 17, as follows.

1. At any input-bit time k, the (starting) state of a transition is denoted by the contents of the two rightmost stages in the register, namely a_{k-1} and $a_{k2.}$
2. for any row (transition), the contents of the a_k stage is found by the modulo-2 addition of bits d_k, a_{k-1}, and a_{k-2} on that row.
3. The output code-bit sequence u_kv_k for each possible starting state (that is, a = 00, b = 10, c = 01, and d = 11) is found by appending the modulo-2 addition of a_k and a_{k-2} to the current data bit, d_k= u_k.

It is easy to verify that the details in Table 1 correspond to the trellis section of Figure 17b. An interesting property of the most useful recursive shift registers used as component codes for turbo encoders is that the two transitions entering a state *should not* correspond to the same input bit value (that is, two solid lines or two dashed lines should not enter a given state). This property is assured if the polynomial describing the feedback in the shift register is of full degree, which means that one of the feedback lines must emanate from the high-order stage; in this example, stage a_{k-2}.

Table 1: Validation of the figure 17(b) Trellis Section

Input bit $d_k = u_k$	Current bit a_k	Starting state		Code bits $u_k v_k$	Ending state	
		a_{k-1}	a_{k-2}		a_k	a_{k-1}
0	0	0	0	0 0	0	0
	1	1	0	0 1	1	1
	1	0	1	0 0	1	0
	0	1	1	0 1	0	1
1	1	0	0	1 1	1	0
	0	1	0	1 0	0	1
	0	0	1	1 1	0	0
	1	1	1	1 0	1	1

b) There are two ways to proceed with encoding the input data sequence $\{d_k\}$ = 1 1 1 0. One way uses the trellis diagram, and the other way uses the encoder circuit. Using the trellis section in Figure 17(b), we choose the dashed-line transition (representing input bit binary 1) from the state a = 00 (a natural choice for the starting state) to the next state b = 10 (which becomes the starting state for the next input bit). We denote the bits shown on that transition as the output coded-bit sequence 11. This procedure is repeated for each input

bit. Another way to proceed is to build a table, such as Table 2, by using the encoder circuit in Figure 17(a). Here, time, k, is shown from start to finish (five time instances and four time intervals). Table 2 is read as follows.

1. At any instant of time, a data bit d_k becomes transformed to a_k by summing it (modulo-2) to the bits a_{k-1} and a_{k-2} on the same row.
2. For example, at time k = 2, the data bit d_k= 1 is transformed to a_k= 0 by summing it to the bits a_{k-1} and a_{k-2} on the same k = 2 row.
3. The resulting output, $u_k v_k$= 10, dictated by the encoder logic circuitry, is the coded-bit sequence associated with time k = 2 (actually, the time interval between times k = 2 and k = 3).
4. At time k = 2, the contents (10) of the rightmost two stages, a_{k-1} and a_{k-2}, represents the state of the machine at the start of that transition.
5. The state at the end of that transition is seen as the contents (01) in the two leftmost stages, a_k a_{k-1}, on that same row. Since the bits shift from left to right, this transition-terminating state reappears as the starting state for time k = 3 on the next row.
6. Each row can be described in the same way. Thus, the encoded sequence seen in the final column of Table 2 is 1 1 1 0 1 1 0 0.

Table 2: Encoding a Bit Sequence with the Figure 17(a) Encoder

Time ($_k$)	Input $d_k = u_k$	First Stage a_k	State at Time k		Output $u_k v_k$
			a_{k-1}	a_{k-2}	
1	1	1	0	0	1 1
2	1	0	1	0	1 0
3	1	0	0	1	1 1
4	0	0	0	0	0 0
5			0	0	

2.4.2 Concatenation of RSC Codes

Consider the parallel concatenation of two RSC encoders of the type shown in Figure (16). Good turbo codes have been constructed from component codes having short constraint lengths (K = 3 to 5). An example of such a turbo encoder is shown in Figure (18), where the switch yielding v_k provides ***puncturing*** (puncturing systematically removes some of the parity bits after encoding. This is, for example, applied to constraint code. Both encoders might want to send the information. This repetition is inefficient so one encoder may puncture its information bits. The information bits in say, the second encoder are therefore ignored, and the rate of the code is increased.), making the overall code rate 1/2. Without the switch, the code would be rate 1/3. There is no limit to the number of encoders that may be concatenated, and in general the component codes need not be identical with regard to constraint length and rate. The goal in designing turbo codes is to choose the best component codes by maximizing the effective free distance of the code [8]. At large values of E_b/N_0, this is tantamount to maximizing the minimum-weight codeword. However, at low

values of E_b/N_0 (the region of greatest interest), optimizing the weight distribution of the codewords is more important than maximizing the minimum-weight codeword [7].

The turbo encoder in Figure (18) produces codewords from each of two component encoders. The weight distribution for the codewords out of this parallel concatenation depends on how the codewords from one of the component encoders are combined with codewords from the other encoder. Intuitively, we should avoid pairing low-weight codewords from one encoder with low-weight codewords from the other encoder. Many such pairings can be avoided by proper design of the interleaver. An interleaver that permutes the data in a random fashion provides better performance than the familiar block interleaver [9].

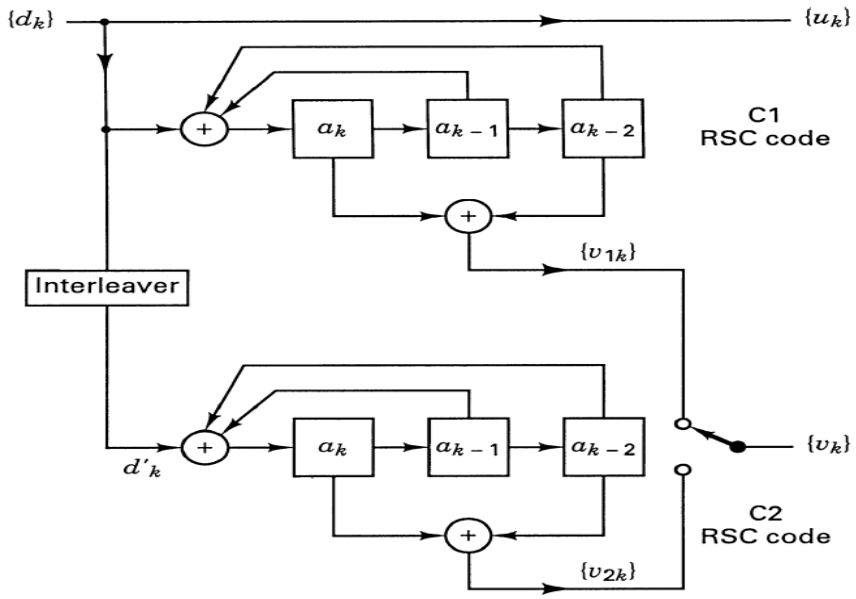

Figure (18): Parallel concatenation of two RSC codes.

If the component encoders are not recursive, the unit weight input sequence 0 0 … 0 0 1 0 0 … 0 0 will always generate a low-weight codeword at the input of a second encoder for any interleaver design. In other words, the interleaver would not influence the output-codeword weight distribution if the component codes were not recursive. However if the component codes are recursive, a weight-1 input sequence generates an infinite impulse response (infinite-weight output). Therefore, for the case of recursive codes, the weight-1 input sequence does not yield the minimum-weight codeword out of the encoder. The encoded output weight is kept finite only by trellis termination, a process that forces the coded sequence to terminate in such a way that the encoder returns to the zero state. In effect, the convolutional code is converted to a block code.

For the encoder of Figure (18), the minimum-weight codeword for each component encoder is generated by the weight-3 input sequence (0 0 … 0 0 1 1 1 0 0 0 … 0 0) with three consecutive 1s. Another input that produces fairly low-weight codewords is the weight-2 sequence (0 0 … 0 0 1 0 0 1 0 0 … 0 0).

However, after the permutations introduced by an interleaver, either of these deleterious input patterns is unlikely to appear again at the input to another encoder, making it unlikely that a minimum-weight codeword will be combined with another minimum-weight codeword.

The important aspect of the building blocks used in turbo codes is that they are recursive (the systematic aspect is merely incidental). It is the RSC code's IIR property that protects against the generation of low-weight codewords that cannot be remedied by an interleaver. One can argue that turbo code performance is largely influenced by minimum-weight codewords that result from the weight-2 input sequence. The argument is that weight-1 inputs can be ignored, since they yield large codeword weights due to the IIR encoder structure. For input sequences having weight-3 and larger, a properly designed interleaver makes the occurrence of low-weight output codewords relatively rare [8-10].

2.5 A Feedback Decoder

The APP of a decoded data bit d_k can be derived from the joint probability $\lambda_k^{i,m}$ defined by

$$\lambda_k^{i,m} = P\{ d_k = i, S_k = m | R_1^N \} \tag{48}$$

Where S_k is the encoder state at time k, and R_1^N is a received binary sequence from time $k = 1$ through some time N.

Thus, the APP for a decoded data bit d_k, represented as a binary digit, is equal to

$$P\{d_k = i | R_1^N \} = \sum_m \lambda_k^{i,m} \quad , \text{i= 0, 1.} \tag{49}$$

The log-likelihood ratio (LLR) is written as the logarithm of the ratio of APPs, as follows:

$$L(\hat{d}) = \log\left[\frac{\sum_m \lambda_k^{1,m}}{\sum_m \lambda_k^{0,m}}\right] \quad (50)$$

The decoder makes a decision, known as the *maximum a posteriori* (*MAP*) decision rule, by comparing $L(\hat{d})$to a zero threshold. That is,

$$\begin{cases} \hat{d} = 1 & if \quad L(\hat{d}) > 0 \\ \hat{d} = 0 & if \quad L(\hat{d}) < 0 \end{cases} \quad (51)$$

For a systematic code, the LLR $L(\hat{d}_k)$ associated with each decoded bit $\hat{d}_k$can be described as the sum of the LLR of $\hat{d}_k$out of the demodulator and of other LLRs generated by the decoder (extrinsic information), as expressed in Equations (12) and (13).

Consider the detection of a noisy data sequence that stems from the encoder of Figure (18), with the use of a decoder shown in Figure (19). Assume binary modulation and a discrete memoryless Gaussian channel. The decoder input is made up of a set R_k of two random variables x_k and y_k. For the bits d_k and v_k at time k, expressed as binary numbers (1, 0), the conversion to received bipolar (+1, -1) pulses can be expressed as follows:

$$x_k = (2d_k - 1) + i_k \quad (52)$$

$$y_k = (2v_k - 1) + q_k \quad (53)$$

where i_k and q_k are two statistically-independent random variables with the same variance σ^2, accounting for the noise contribution. The redundant information, y_k, is demultiplexed - (a method by which multiple analogue message signals or digital data streams are combined into one signal) - and sent to decoder DEC1 as $y_{1k}$when

$v_k = v_{1k}$, and to decoder DEC2 as $y_{2k}$when $v_k = v_{2k}$. When the redundant information of a given encoder (C1 or C2) is not emitted, the corresponding decoder input is set to zero. Note that the output of DEC1 has an interleaver structure identical to the one used at the transmitter between the two encoders. This is because the information processed by DEC1 is the noninterleaved output of C1 (corrupted by channel noise). Conversely, the information processed by DEC2 is the noisy output of C2 whose input is the same data going into C1, however permuted by the interleaver. DEC2 makes use of the DEC1 output, provided that this output is time-ordered in the same way as the input to C2 (that is, the two sequences into DEC2 must appear "in step" with respect to the positional arrangement of the signals in each sequence). For more details we refer the reader to see [4]

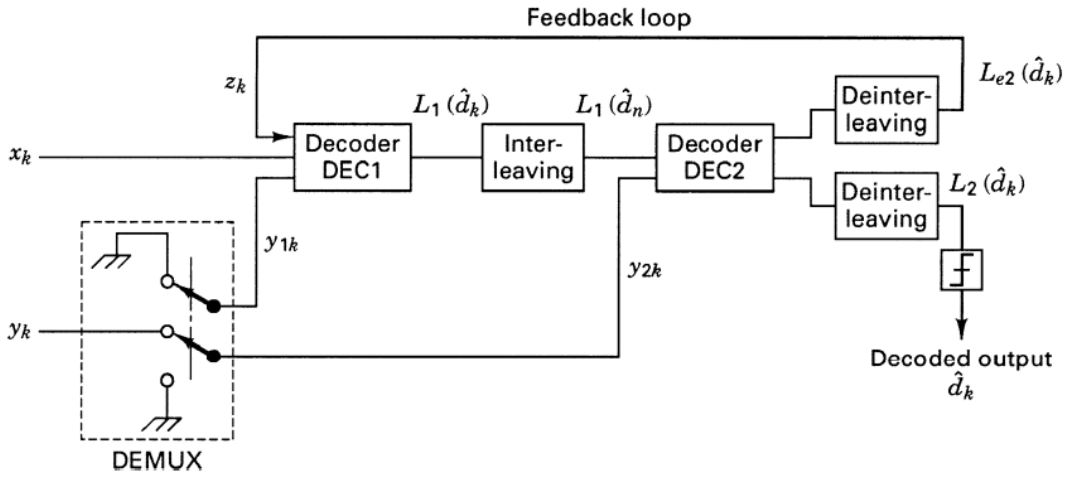

Figure (19): Feedback decoder.

2.5.1 Decoding with a Feedback Loop

We rewrite Equation (11) for the soft-decision output at time *k*, with the a priori LLR $L(d_k)$ initially set to zero. This follows from the assumption that the data bits are equally likely. Therefore,

$$L(\hat{d}_k) = L_c(x_k) + L_e(\hat{d}_k)$$
$$= \log\left[\frac{P(x_k|d_{k=1})}{P(x_k|d_{k=0})}\right] + L_e(\hat{d}_k) \quad (54)$$

Where $L(\hat{d}_k)$ is the soft-decision output at the decoder, and $L_c(x_k)$ is the LLR channel measurement, stemming from the ratio of likelihood functions $P(x_k|d_k=i)$ associated with the discrete memoryless channel model. $L_e(\hat{d}_k) = L(\hat{d}_k)|_{xk=0}$ is a function of the redundant information. It is the extrinsic information supplied by the decoder, and does not depend on the decoder input x_k. Ideally, $L_c(x_k)$ and $L_e(\hat{d}_k)$ are corrupted by uncorrelated noise, and thus $L_e(\hat{d}_k)$ may be used as a new observation of d_k by another decoder for an iterative process. The fundamental principle for feeding back information to another decoder is that a decoder should never be supplied with information that stems from itself (because the input and output corruption will be highly correlated).

For the Gaussian channel, the natural logarithm in Equation (54) is used to describe the channel LLR, $L_c(x_k)$, as in Equations (16a-c). We rewrite the Equation (16c) LLR result below:

$$L_c(x_k) = -\frac{1}{2}\left(\frac{x_k-1}{\sigma}\right)^2 + \frac{1}{2}\left(\frac{x_k+1}{\sigma}\right)^2 = \frac{2}{\sigma^2}x_k \quad (55)$$

Both decoders, DEC1 and DEC2, use a method of decoding, for example, the modified Bahl algorithm [2]. If the inputs $L_1(\hat{d}_k)$ and y_{2k} to decoder DEC2 are statistically independent, the LLR $L_2(\hat{d}_k)$ at the output of DEC2 can be written as

$$L_2(\hat{d}_k) = f\,[L_1(\hat{d}_k)] + L_{e2}(\hat{d}_k) \tag{56}$$

With

$$L_1(\hat{d}_k) = \frac{2}{\sigma_0{}^2} x_k \quad + \quad L_{e1}(\hat{d}_k) \tag{57}$$

Where f [·] indicates a functional relationship. The extrinsic information $L_{e2}(\hat{d}_k)$ out of DEC2 is a function of the sequence $\{L_1(\hat{d}_k)\}_{n\neq k}$. Since $L_1(\hat{d}_k)$ depends on the observation R_1^N, the extrinsic information $L_{e2}(\hat{d}_k)$is correlated with observations x_k and y_{1k}. Nevertheless, the greater $|n\text{-}k|$ is, the less correlated are $L_1(\hat{d}_k)$ and the observations x_k, y_k. Thus, due to the interleaving between DEC1 and DEC2, the extrinsic information $L_{e2}(\hat{d}_k)$ and the observations x_k, y_{1k} are weakly correlated. Therefore, they can be jointly used for the decoding of bit d_k. In Figure (19), the parameter $z_k = L_{e2}(\hat{d}_k)$ feeding into DEC1 acts as a diversity effect in an iterative process. In general, $L_{e2}(\hat{d}_k)$ will have the same sign as d_k.Therefore, $L_{e2}(\hat{d}_k)$ may increase the associated LLR and thus improve the reliability of each decoded data bit.

The algorithmic details for computing the LLR, $L(\hat{d}_k)$, of the a posteriori probability (APP) for each data bit has been described by several authors [3-4,20]. Suggestions for decreasing the implementational complexity of the algorithms can be found in [18-24].

Chapter 3

Linear Programming Decoding of Low Density Parity Check Codes (LDPC)

3.1 Introduction

LDPC codes can be classified in two categories, regular and irregular LDPC codes.

Definition 3.1: A regular LDPC code is characterized **by** two values, d_v and d_c, where d_v is the number of ones in each column of the parity check matrix H $\in F_2^{M \times N}$, and d_c represents the numbers of ones in each row.

Generally, the decoding of linear block codes, and particularly the LDPC codes can be carried out by several methods. One of the well-known method, the message passing algorithm (MPA). Another method will be described in the following, the *linear programming* decoding.

The Linear Programming (LP) method is an optimization method to solve a problem defined by a linear *objective function* using *linear* constraints. The optimization problem can be considered as follows, Given a binary linear code $C \in F_2^n$, a codeword $x = [x_1 \quad x_2 \quad \cdots \quad x_N]^T$ is transmitted over a memoryless channel, the received vector is $y = [y_1 \quad y_2 \quad \cdots \quad y_N]^T$ and the log likelihood ratio (LLR) vector is $\lambda = [\lambda_1 \quad \lambda_2 \quad \cdots \quad \lambda_n]$. The parity check matrix H $\in F_2^{m \times n}$. The goal of the optimization problem is to find the maximum likelihood codeword

$$\text{Minimize, } \lambda^T x \qquad (1)$$

$$\text{subject to } x \in C$$

From now onwards the optimization problem is written with the notation min. as minimum and s.t. as subject to. If $x \in C$, then $H.\ x = 0$. So, the equation (1) becomes,

$$\text{min. } \lambda^T x \qquad (2)$$

$$\text{s.t. } H.\ x = 0$$

$$x \in \{0,1\}$$

The different parts of this optimization problem are,

- An *objective function* which is minimized (or sometimes maximized):

$$\text{min. } \lambda^T x \qquad (3)$$

- The *problem constraint*,

$$H\ .x = 0 \qquad (4)$$

If we split H into its M rows, the equation (4) will be,

$$h_m \otimes\ x = 0; \qquad m = \underbrace{A, B, C, \ldots.}_{M} \qquad (5)$$

We have now M equations, each corresponding to :

$$a_{i1}x_1 \oplus a_{i2}x_2 \oplus a_{i3}x_3 \oplus \ldots\ldots \oplus a_{iN}x_N = 0 \qquad (6)$$

With $a_{ij} \in \{0,1\}$, $j = 1, \ldots\ldots, N, \quad i = 1, \ldots, m$

- The *integer (binary) variable*, $x_n \in \{0,1\}$.

3.2 Maximum Likelihood Decoding for LP

After the transmission of an error-correcting code, the receiver has to decode the transmitted codeword. So, one way to decode is to choose a codeword which has the maximum likelihood probability of a received word given a transmitted codeword. It means that it will find a *maximum likelihood codeword*. We will show how the maximum likelihood codeword can be derived in a linear form in order to be used in the LP formulation.

If a codeword $x \in C$ is transmitted over a memoryless channel and the corresponding received vector is y, then the maximum likelihood codeword can be :

$$x = \underset{x \in C}{argmax} \Pr(y|x) \tag{8}$$

Since, the variables are independent and the channel is memoryless without feedback, equation (8) becomes:

$$x = \underset{x \in C}{argmax} \prod_{n=1}^{N} \Pr(y_n | x_n) \tag{9}$$

$$= \underset{x \in C}{argmin} (-\ln \prod_{n=1}^{N} \Pr(y_n | x_n)) \tag{10}$$

$$= \underset{x \in C}{argmin} (-\sum_{n=1}^{N} \ln \Pr(y_n | x_n)) \tag{11}$$

One trick is now used. The term $\sum_{n=1}^{N} \ln \Pr(y_n | 0)$ is independent of *x*. So, it can be considered as a constant since minimization is done over *x* equation (11) will be :

$$x = \underset{x \in C}{argmin} \sum_{n=1}^{N} \ln \Pr(y_n | 0) - \sum_{n=1}^{N} \ln \Pr(y_n | x_n) \tag{12}$$

$$= \underset{x \in C}{argmin} \sum_{n=1}^{N} ln \frac{\Pr(y_n|0)}{\Pr(y_n|x_n)} \quad (13)$$

The sum, $\sum_{n=1}^{N} ln \frac{\Pr(y_n|0)}{\Pr(y_n|x_n)}$ is equal to 0 when $x_n = 0$, and for $x_n = 1$ it is equal to

$$\sum_{n=1}^{N} ln \frac{\Pr(y_n|x_n=0)}{\Pr(y_n|x_n=1)} = \sum_{n=1}^{N} \lambda_n$$

So equation (13) is now :

$$x = \underset{x \in C}{argmin} \sum_{n=1}^{N} x_n \cdot \left(\ln \frac{\Pr(y_n|0)}{\Pr(y_n|x_n)} \right) \quad (14)$$

Finally, the log likelihood ratio can be replaced by its value such that

$$x = \underset{x \in C}{argmin} \sum_{n=1}^{n} \lambda_n x_n \quad (15)$$

$$= \underset{x \in C}{argmin} \lambda^T . \boldsymbol{x} \quad (16)$$

We see that the equation (16) is exactly equal to our optimization problem described in equation (1).

3.3 Linear Programming Formulation

Now we will first state the problem of linear programming decoding and then describe the results.

3.3.1 Problem Formulation

An example will be taken to derive a formulation of LP decoding. Afterwards the general formulation is shown. Our goal is still,

$$\text{min. } \lambda^T x \quad (17)$$

$$\text{s.t. } h_m \otimes x = 0; \qquad m = \underbrace{A, B, C, \ldots.}_{M}$$

$$x_n \in \{0,1\}$$

We want to use a linear programming decoder such that the variables and constraint function are in R. A simple example is considered to formulate the integer LP problem before relaxation of the constraints. The relaxation of the constraints means that the constraint $x_n \in \{0,1\}$is changed to $x_n \in [0,1]$.

The check equations in F_2will be reformulated as linear equations in R which is described further. So, consider the following parity check matrix H,

$$H = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{18}$$

If $x = [x_1 \quad x_2 \quad x_3 \quad x_4 \; x_5 \quad x_6]$, the set of check equations corresponding to the H matrix will be,

$$H \,.\, x = \left\{ \begin{array}{l} chk(A): x_1 + x_4 + x_5 = 0 \\ chk(B): x_2 + x_4 + x_6 = 0 \\ chk(c): x_3 + x_5 + x_6 = 0 \end{array} \right\} \tag{19}$$

In order to use the parity check equation in R, a new formulation with new variables is used. Let us define them for example for chk(A),

- $S_A$is the set of indices of the code symbols used in the check equation chk(A)

$$S_A = \{ 1 , 4, 5 \} \tag{20}$$

- $x_{S_A}$is the local codeword formed with the corresponding code symbols used in check equation chk(A):

$$x_{S_A} = (x_1 , x_4 , x_5)^{\mathrm{T}} \tag{21}$$

- The matrix which will be able to extract $x_{S_A}$from x is called B_A

$$B_A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad (22)$$

It can be verified that

$$x_{S_A} = B_A x \quad (23)$$

Note : in the above equation the addition and multiplication in R are used instead of the modulo 2 operation.

- Now, another matrix A_A is defined as follows:

The columns of A_A are composed of the local codewords satisfying chk(A). These local codewor's are even weight vectors of length d_c, including the zero vector. In our case d_c = 3, so:

$$X_{S_A} \in \{ (0 , 0, 0)^T, (0 , 1, 1)^T, (1 , 0, 1)^T , (1 , 1, 0)^T\}$$

Thus,

$$A_A = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \quad (24)$$

An indicator vector w_A that select the right configuration for chk(A) is defined in this way

$$w_A = \begin{bmatrix} w_A, \{\Phi\} \\ w_A, \{4,5\} \\ w_A, \{1,5\} \\ w_A, \{1,4\} \end{bmatrix} \in F_2^4 \ , \ \mathbf{1}^T w_A = 1 \quad (25)$$

It shall be noted that the 4 in F_2^4 corresponds to the number of different local codewords satisfying chk(A).and also to the number of columns in the matrix A_A .And 1 is the all-ones vector,

$$\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad (26)$$

So, w_A belongs to the set because of the definition in equation (25),

$$w_A \in \left\{ \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix} \right\} \quad (27)$$

All these vectors will satisfy, $\mathbf{1}^T w_A = 1$

It can be seen that

$$x_{S_A} = A_A w_A \quad (28)$$

In equation (28), the multiplication and addition are used in R. If for example if $w_A = [0 \quad 1 \quad 0 \quad 0]^T$ then $w_{A,\{4,5\}} = 1$ and $w_{A,\{\Phi\}} = w_{A,\{1,5\}} = w_{A,\{1,4\}} = 0$ such that $x_{S_A} = [0 \ 1 \ 1]^T$. w_A is an indicator/auxiliary variable which selects the correct configuration for the local codeword from the columns of A_A.

Equation (23) and (28) are combined such that

$$B_A x = A_A w_A, \qquad w_A \in F_2^4, \quad \mathbf{1}^T w_A = 1 \quad (29)$$

Equation (29) has been formed in such a way that it satisfies the check equation chk(A). Thus, equation (29) is equivalent to chk(A)

For each check equation of (19), the following settings for the new linear equations can be deduced,

$$\text{Chk(A)} = \left\{ \begin{array}{c} x_1 + x_4 + x_5 = 0, S_A = \{1,4,5\} \\ w_A = [w_{A,\{\Phi\}} \quad w_{A,\{4,5\}} w_{A,\{1,5\}} \quad w_{A,\{1,4\}} \quad]^T \\ B_A = \begin{bmatrix}1&0&0&0&0&0\\0&0&0&1&0&0\\0&0&0&0&1&0\end{bmatrix}, \quad A_A = \begin{bmatrix}0&0&1&1\\0&1&0&1\\0&1&1&0\end{bmatrix} \end{array} \right\} \quad (30)$$

$$\text{Chk(B)} = \left\{ \begin{array}{c} x_2 + x_4 + x_6 = 0, S_B = \{2,4,6\} \\ w_B = [w_{B,\{\Phi\}} \quad w_{B,\{4,6\}} w_{B,\{2,6\}} \quad w_{B,\{2,4\}} \quad]^T \\ B_B = \begin{bmatrix}0&1&0&0&0&0\\0&0&0&1&0&0\\0&0&0&0&0&1\end{bmatrix}, \quad A_B = \begin{bmatrix}0&0&1&1\\0&1&0&1\\0&1&1&0\end{bmatrix} \end{array} \right\} \quad (31)$$

$$\text{Chk(C)}=\left\{\begin{array}{c} x_3+x_5+x_6=0\ , S_C=\{3,5,6\} \\ w_C=[\mathrm{w}_{C,\{\Phi\}}\quad \mathrm{w}_{C,\{5,6\}}\mathrm{w}_{C,\{3,6\}}\quad \mathrm{w}_{C,\{3,5\}}\quad]^T \\ B_C=\begin{bmatrix}0&0&1&0&0&0\\0&0&0&0&1&0\\0&0&0&0&0&1\end{bmatrix},\quad A_C=\begin{bmatrix}0&0&1&1\\0&1&0&1\\0&1&1&0\end{bmatrix} \end{array}\right\} \tag{32}$$

So, if the number of one's is the same in each row of H (d_c = 3), A_m will be the same matrix for each check equation (m corresponds to the index of the check equation). Let us generalize the formulation. For each m^{th} row of the parity check matrix, let $d_c(m)$ be the number of ones in this row, S_m be the set of indices of the code symbols participating in the check equation m, B_m be a $d_c(m)$ *x N* matrix which extracts the local codeword x_{S_A} from the codeword$x$ and A_m be a matrix $d_c(m)$ x $2^{d_c(m)-1}$ whose columns are formed by the local codewords satisfying m^{th} check equation. The m^{th} parity check equation can be written as

$$B_m\, x = A_m w_m, \qquad w_m \in \{0,1\}^{\,2^{dc(m)-1}},\ \mathbf{1}^{\mathrm{T}} w_m = 1 \tag{33}$$

It can be noticed that all the operations used in this equation are over R. By replacing the check constraint in (17) by this new equation (33), the optimization formulation becomes

$$\begin{array}{ll} \min.\ \lambda^{T} x & \\ \text{s.t.} \quad B_m\, x = A_m w_m, & m = \underbrace{A, B, C, \ldots.}_{M} \\ w_m \in \{0,1\}^{\,2^{dc(m)-1}}, & \mathbf{1}^{\mathrm{T}} \mathrm{w_m} = 1 \\ x_n \in \{0,1\}, & n = 1, \ldots, N \end{array} \tag{31}$$

3.4 Alternative Formulation

In this section, an alternative way of defining the constraints of the LP are given. The motivation for the *alternative formulation* is to reduce the

complexity of LP by removing the auxiliary variable $w_{m,s}$. We know that for each check equation, the number of different auxiliary variables $w_{m,s}$ grows exponentially (2^{d_c-1}), so removing these could simplify the problem significantly. However, we are minimizing $\lambda^T x$ and the objective function is independent of the $w_{m,s}$, so it can be possible to remove the auxiliary variable.

3.4.1 Exemplification of the Alternative Formulation

In this section, an example is given to show how the alternative formulation can be derived. Consider the constraints for the degree 3 check equation *A* in equation (32), where the constraint $\mathbf{1}^T w_A = 1$ has been added to the matrix.

$$\begin{bmatrix} 1 \\ x_1 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} w_{A,\{\phi\}} \\ w_{A,\{1,4\}} \\ w_{A,\{1,5\}} \\ w_{A,\{4,5\}} \end{bmatrix} \quad , \quad w_{A,S} \geq 0 \tag{32}$$

Formulating this in an augmented matrix form and bringing it into row reduced echelon form.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & x_1 \\ 0 & 1 & 0 & 1 & x_4 \\ 0 & 0 & 1 & 1 & x_5 \end{bmatrix} \overrightarrow{RREF} \begin{bmatrix} 1 & 0 & 0 & 0 & \frac{2-x_1-x_4-x_5}{2} \\ 0 & 1 & 0 & 0 & \frac{x_1+x_4-x_5}{2} \\ 0 & 0 & 1 & 0 & \frac{x_1-x_4+x_5}{2} \\ 0 & 0 & 0 & 1 & \frac{-x_1+x_4+x_5}{2} \end{bmatrix} \tag{33}$$

If we add the constraint that $w_{A,S} \geq 0$ and multiply with -1, we obtain the following 4 equations in (34)

$$-2 + x_1 + x_4 + x_5 \leq 0 \qquad \text{(hs1)}$$

$$-x_1 - x_4 + x_5 \leq 0 \qquad \text{(hs2)}$$

$$-x_1 + x_4 - x_5 \leq 0 \qquad \text{(hs3)}$$

$$+x_1 - x_4 - x_5 \leq 0 \qquad \text{(hs4)} \qquad (34)$$

First note that the solution space of x_n in (34) should be equivalent with the solution space of x_n in (32) (all the constraints are utilized). We can now describe the local check equation constraints in another way without the indicator variable $w_{A,S}$.Further; the new constraints have a nice interpretation. The equation (34) describes 4 *half-spaces* (hs1) - (hs4). Contemplate now the tetrahedron in the Figure (20).

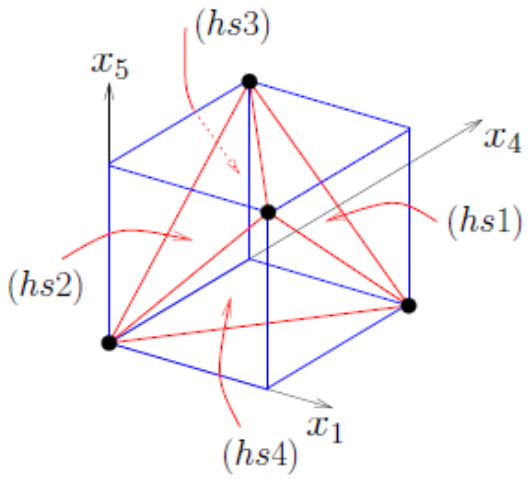

Figure (20): the 4 half-spaces defining the tetrahedron

For equality in the equations (34), the faces of the tetrahedron are described by the4 planes, and the solid tetrahedron is described by the 4 half-spaces (hs1) - (hs4).

3.4.2 The Alternative Formulation in General

From the previous section's results, it can be construed that the convex hull can be described as a set of inequalities without using the auxiliary variable $w_{m,S}$. Firstly, the example of the previous section with half space

(hs2) is described after which the alternative formulation is derived in general. In Figure (21), the example with (hs2) is shown.

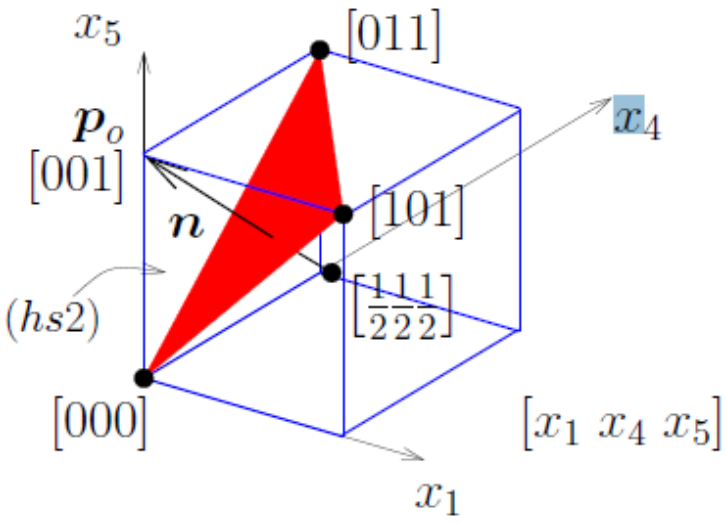

Figure (21) : Half-space 2 (hs2) with the normal *n*.

The normal to the half-space is $n = p_0 - \frac{1}{2}\mathbf{1}$, i.e. the vector from the center of the unit cube to an odd weighted vertex p_0. Then, the plane can be defined such that all x_{s_m} satisfy equation (35).

$$n^T(x_{s_m} - x_{0,s_m}) = 0 \tag{35}$$

Where n is the normal, and x_{0,s_m} is a point on the plane. Inserting the example from (hs2) yields.

$$2.\ n^T(x_{s_m} - x_{0,s_m}) = 2.\ (p_0 - \tfrac{1}{2}\mathbf{1})^T (x_{s_m} - x_{0,s_m}) = 0 \tag{36}$$

$$= 2.\left(\begin{bmatrix}0\\0\\1\end{bmatrix} - \begin{bmatrix}1/2\\1/2\\1/2\end{bmatrix}\right)^T \left(\begin{bmatrix}x_1\\x_4\\x_5\end{bmatrix} - \begin{bmatrix}0\\0\\0\end{bmatrix}\right) = 0 \tag{37}$$

$$= \begin{bmatrix}-1\\-1\\+1\end{bmatrix}^T \begin{bmatrix}x_1\\x_4\\x_5\end{bmatrix} = 0 \tag{38}$$

$$= -x_1 - x_4 + x_5 \tag{39}$$

Not surprisingly, we have obtained the surface corresponding to half space (hs2) in equation (34). From this example, we now generalize this procedure for the halfspaces.

$$n^T(x_{s_m} - x_{0,s_m}) \leq 0 \tag{40}$$

$$2.n^T x_{s_m} \leq 2.x_{0,s_m} \tag{41}$$

As *n* always point towards an odd weighted vertex, it will have the following form.

$$2.n^T = \left[\overbrace{\underbrace{+1\ +1\ \ldots +1}_{v,|v|\ is\ odd}\ \underbrace{-1\ -1\ldots\ -1}_{N(m)\backslash v}}^{N(m)}\right] \tag{42}$$

$2\,n^T$ can be permuted and just shown in this regular form for convenience. *V* is defined as a subset of indices from *N(m)* where n^T has positive signs. The negative signs (the rest) are then *N(m)* excluding *V* . We can further see that we can select x_{0,s_m} as one of the even weighted vertices defining the plane. The number of positions where the symbols in x_{0,s_m} and p_0 differs is **1** i.e., Hamming weight or distance $w_H(x_{0,s_m} \oplus p_0) = 1$. This fact can be construed by observing the Figure (21). Any walk from an even weight vertex along an edge in the hyper cube will give an odd weight vertex, and vice versa. The total weight change from an odd to an even vertex is ± 1, such that the sum of x_{0,s_m} is $\mathbf{1}^T x_{0,s_m} = |V| \pm 1$.

$$x_{0,s_m} = \left[\overbrace{\underbrace{1\ 1\ \ldots 1}_{\#even\ =\ |v|\pm 1}\ \ 0\,0\ldots\ 0}^{N(m)}\right] \tag{43}$$

x_{0,s_m} can be permuted and just shown in this regular form for convenience. It can be comprehended that $2\,n^T x_{0,s_m} = |V| - 1$ from the contention. Continuing from equation (41).

$$2.\,n^T\,x_{s_m} \leq 2.\,x_{0,s_m} \tag{44}$$

$$\sum_{n\in V} x_n - \sum_{n\in N(m)\setminus V} x_n \leq |V| - 1 \tag{45}$$

Since we want a set of inequalities for all the surfaces, we need to find the respective half-spaces for each odd weighted vertex in the unit cube, which corresponds to the all odd combinations of the subset *V* taken from *N(m).* All the solutions should be within the unit cube, so the general check constraint in the alternative formulation is given by [16], [18]:

$$\sum_{n\in V} x_n - \sum_{n\in N(m)\setminus V} x_n \leq |V| - 1 \qquad \forall\, V \subseteq N(m)\,,\ |V| \text{ odd} \quad 0 \leq x_n \leq 1 \tag{46}$$

Equation (46) is bit hard to interpret, so if these inequalities are written in matrix form then they have a nice structure. The constraints from the degree 4 check equation *A* of the (7, 4, 3) Hamming code is considered such that *N(A)* = {1, 2, 4, 5}.

$$\begin{bmatrix} 1 & -1 & -1 & -1 \\ -1 & 1 & -1 & -1 \\ -1 & -1 & 1 & -1 \\ -1 & -1 & -1 & 1 \\ 1 & 1 & 1 & -1 \\ 1 & 1 & -1 & 1 \\ 1 & -1 & 1 & 1 \\ -1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_4 \\ x_5 \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 2 \\ 2 \\ 2 \\ 2 \end{bmatrix} \begin{matrix} V = \{1\} \\ V = \{2\} \\ V = \{4\} \\ V = \{5\} \\ V = \{1,2,4\} \\ V = \{1,2,5\} \\ V = \{1,4,5\} \\ V = \{2,4,5\} \end{matrix} \tag{47}$$

Note that the matrix has a nice diagonal symmetry with the signs when the rows are in the order shown.

3.4.3 Special Properties for a Degree 3 Check Equation

Now turn back to the degree 3 check equation, which turns out to have special properties. Consider now a different way of deriving the alternative formulation.

$$\begin{bmatrix}1\\x_1\\x_4\\x_5\end{bmatrix} = \begin{bmatrix}1&1&1&1\\0&1&1&0\\0&1&0&1\\0&0&1&1\end{bmatrix}\begin{bmatrix}w_{A,\{\phi\}}\\w_{A,\{1,4\}}\\w_{A,\{1,5\}}\\w_{A,\{4,5\}}\end{bmatrix},\ w_{A,S} \geq 0 \tag{48}$$

$$\begin{bmatrix}1\\x_{S_A}\end{bmatrix} = A_{A.e}w \tag{49}$$

where, $A_{A.e}$ is the extended *(e)* version of the A_A with $\mathbf{1}^T w$ added in the first row.$A_{A.e}$is non-singular and square, so calculating the inverse yields.

$$2w = 2A_{A,e}^{-1}\begin{bmatrix}1\\x_{S_A}\end{bmatrix} = \begin{bmatrix}2&-1&-1&-1\\0&1&1&-1\\0&1&-1&1\\0&-1&1&1\end{bmatrix}\begin{bmatrix}1\\x_{S_A}\end{bmatrix} \tag{50}$$

The factor 2 is used to bring the matrix entries to integer values. Again using the side constraint $w_{m,s} \geq 0$ and multiply by -1,

$$\begin{bmatrix}-2&1&1&1\\0&-1&-1&1\\0&-1&1&-1\\0&1&-1&-1\end{bmatrix}\begin{bmatrix}1\\x_{S_A}\end{bmatrix} \leq \begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{matrix}(hs1)\\(hs2)\\(hs3)\\(hs4)\end{matrix} \tag{51}$$

which will give us the four half spaces. This manoeuvre can only be accomplished for a degree 3 check equation, because only this matrix is square. For higher degrees more unknown occurs since $A_{A.e}$has dimension $(d_c + 1) \times (2^{d_c - 1})$. Further, a check equation of degree 3 has a special property. To see this, add now the equations corresponding to the two half-spaces (hs2) and (hs3).

$$\underbrace{(-x_1 - x_4 + x_5)}_{(hs2)} + \underbrace{(-x_1 + x_4 - x_5)}_{(hs3)} \leq 0 + 0 \Leftrightarrow x_1 \geq 0 \tag{52}$$

Further by adding (hs1) and (hs2)

$$\underbrace{(-2 + x_1 + x_4 + x_5)}_{(hs1)} + \underbrace{(x_1 - x_4 - x_5)}_{(hs4)} \leq 0 + 0 \Leftrightarrow x_1 \leq 1 \qquad (53)$$

This means that the bounding $0 \leq x_1 \leq 1$ of the variables is implied in the equations.It is also implied for x_4 and x_5. If one observes the tetrahedron in the Figure (20) and imagine the four planes, it is possible to see that no solutions are outside the unit cube $[0,1]^3$. That is, all the variables in a degree 3 check equation are bounded automatically because it is implied in the definition. The side constraints $0 \leq x_1 \leq 1$ are not needed if x_n participates in a check equation of degree 3.

The consequence of this observation is that the complexity of LP decoding may be reduced. If all variable x_n participates in a check equation of degree 3, then no bounding is needed. The complexity in terms of the number of constraints can be reduced by *2N* since there are 2 inequalities for each variable and *N* variables.

3.5 Multiple Optima in the BSC

This section is on the existence of multiple optima in the BSC. It will be derived how multiple optima occurs while decoding the (7, 4, 3) Hamming code.

A leaf node is a variable node having degree one such that it is only connected with one check node. By example it was identified with the LP decoder, that if an error occurs in a leaf node of the (7, 4, 3) Hamming code, the LP decoder may fail because of the occurrence of multiple

optima. For an error anywhere else, decoding always succeeds. In this section we will investigate this observation.

We will start by convincing us that for ML decoding, a leaf node error is possible to decode. The (7, 4, 3) Hamming code has d_{min} = 3, which means that we should be able to correct at minimum t = $\left\lfloor \frac{d_{min}-1}{2} \right\rfloor$ = 1 error anywhere in the codeword [23] .So the loss of decoding performance is due to the degradation of the LP decoder compared to the ML decoder.

For the example, the alternative formulation is used for decoding and MOSEK 4 is used as a solver to yield two solutions, a basic point solution x_b^* and an interior point solution x_i^*. If an error is placed in the leaf node for this code under the BSC, then we have,

$$\lambda^T x_b^* = \lambda^T x_i^* \quad \text{where} \quad x_b^* \neq x_i^* \tag{54}$$

It states that both solutions have the same cost and there exist there by multiple optima for this optimization problem. Both of them should be considered as correct solutions. However, there are more than two solutions to an error in the leaf node for the Hamming code. In fact there are infinitely many solutions as we will show now. Because there is equal cost in equation (54), then

$$\lambda^T (x_b^* - x_i^*) = \lambda^T r = 0 \tag{55}$$

The vector of direction r is orthogonal to the cost vector λ . The problem of multiple optima could look like the Figure (22).

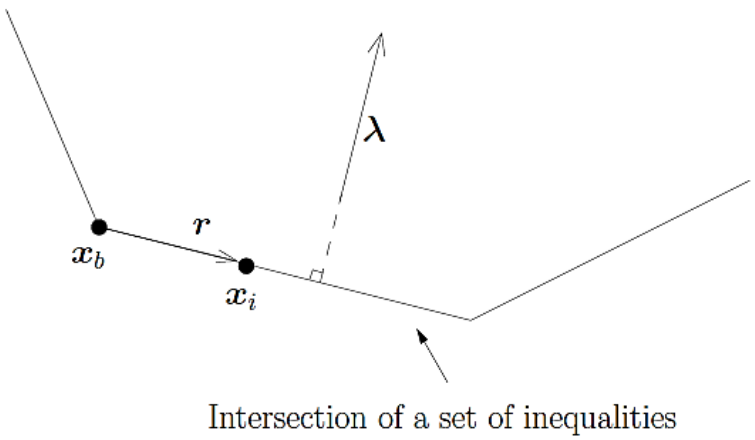

Figure (22): overview of multiple optima and MOSEK solution

The reason why x_b^* is a vertex, is because by example x_b^* has always given an integer point solution. We now want to find the set of inequalities such that we can find the intersection that seems to be orthogonal to the cost vector. Say that we are solving the following problem, where *G* and *h* defines the inequalities from the alternative formulation.

$$\min\ \lambda^T . x \tag{56}$$

$$\text{s.t. } Gx \leq h$$

$$0 \leq x_n \leq 1 \qquad \forall n = 1, 2, \ldots N$$

Let now each row equation of $G\ x \leq h$ be $g_k^T \leq h_k$. The set of inequalities defining the intersection, is all the active constraints for the solution $x = x_i^*$. An active constraint is where equality holds such that the solution is on the bound of the polytope, i.e., the equations *k* for which $g_k^T x_i^* = h_k$ instead of just $g_k^T x_i^* \leq h_k$. All the active constraints are then the constraints for which one of the following equations holds.

$$g_k^T x_i^* = h_k \quad \text{or} \quad x_n = 0 \quad \text{or} \quad x_n = 1 \tag{57}$$

Chapter 4

The Method of LP Decoding

4.1 Linear Programming Relaxation

A *linear program* (LP) consists of a set of linear inequalities (constraints) and a linear objective function over a set of variables. Solving the linear program means finding a setting of the variables that satisfies the inequalities, and optimizes the objective function. Linear programs can be solved efficiently using the simplex algorithm [1], which runs efficiently in practice, or the ellipsoid algorithm [17], which has worst-case run-time guarantees.

Although many important problems can be solved as LPs, not all problems are directly amenable to this treatment. One issue is that LP solutions can be real-valued, whereas the variables (in certain problems) may only be meaningful as integers (e.g., number of seats in an airplane). If we add the restriction that all variables must be integers, we obtain an **Integer Linear Programming (ILP)** problem, which (unfortunately) is NP-hard in general [27].

A natural strategy for finding an *approximate* solution to an ILP, then, is to remove the integer constraints, solve the resulting LP, and then transform the solution into a meaningful one. (For example, rounding techniques, often randomized, are one method of transforming an LP solution into a decent solution to the ILP of interest.) This generic technique is referred to as linear programming relaxation, and many successful approximation algorithms to NP-hard optimization problems are based on it [11].

4.2 An LP Relaxation of ML Decoding

Suppose we wish to decode a binary code C⊆ $\{0,1\}^n$ under some binary-input memoryless channel. Let y∈ C denote the transmitted codeword, And let $\tilde{y}$ denote the received codeword.

Let γ_i be the *log-likelihood ratio* of the *ith* code bit:

$$\gamma_i = \ln\left(\frac{\Pr(\tilde{y}_i|\, y_i=0}{\Pr(\tilde{y}_i|\, y_i=1}\right) \tag{58}$$

The sign of the log-likelihood ratio$\gamma_i$determines whether transmitted bit y_i is more likely to be a 0 or a 1. (In particular, if y_i is more likely to be a 1, then γ_i will be negative, whereas if $y_i$is more likely to be a 0, then γ_i will be positive.) We will refer to γ_i as the *cost* of code bit y_i, where γ_i represents the cost incurred by setting a particular bit y_i to 1, and to the sum$\sum_i \gamma_i\, y_i$ as the cost of a particular codeword y. With these definitions, the ML codeword is exactly the codeword of minimum cost [14].

Our LP relaxations for decoding will have LP variables f_i for each code bit, where i∈ {1, … , n}. Suppose we were able to solve the following problem:

$$\text{Minimize } \sum_{i=1}^{n} \gamma_i\, f_i \qquad \text{s.t} \quad f \in C \tag{59}$$

Any optimal solution f to this system is an ML codeword. However, optimizing over C is too complex in general. Therefore, we optimize instead over a less complex *polytope* ⊆ $[0,1]^n$, defined by a set of linear constraints on the variables f_i. The particular nature of the constraints will depend on the underlying code. In [16, 15], polytopes for turbo codes, LDPC codes, and arbitrary binary linear codes was defined. In each of

these cases, polytopes contain a linear (in n) number of constraints, and are therefore solvable efficiently.

Since we are looking for codewords, it should be the case that our polytope includes all the codewords, and does not include any non-codewords.

Definition 4.1: *A polytope $\mathcal{P}$is* proper *for code C if the integral points in $\mathcal{P}$are exactly the codewords of* C*; i.e., $\mathcal{P}$ is proper if* $\mathrm{P} \cap \{0,1\}^n = C$.

Given a proper polytope $\mathcal{P}$, our LP decoder solves the following linear program:

$$\text{Minimize } \sum_{i=1}^{n} \gamma_i f_i \qquad \text{s.t} \quad f \in \mathcal{P} \tag{60}$$

Define the *cost* of a point $f \in \mathcal{P}$ as $\sum_{i=1}^{n} \gamma_i f_i$. The LP in equation (59) will find the point in$\mathcal{P}$ with minimum cost. If the LP solution is integral (i.e., all f_i are either 0 or 1), then the LP decoder outputs the codeword *f*. In contrast, if the LP solution is fractional (i.e., some f_i is non-integral), then the decoder outputs "error."

Notice that the only part of the LP relaxation that depends on the received vector is the objective function.

Figure (22) provides an abstract visualization of the relaxation $\mathcal{P}$. This figure is of course two-dimensional, but all the elements of the LP relaxation can still be seen. The dotted line represents the polytope $\mathcal{P}$, and the circles represent vertices of the polytope. The black circles in the figure represent codewords, and the gray circles represent fractional vertices that are not codewords. The inner solid line encloses the convex hull of the codewords, the set of points that are convex combinations of codewords. The arrows inside the polytope represent various cases for the objective

function, which depend on the noise in the channel. We will go through these cases explicitly later in the discussion.

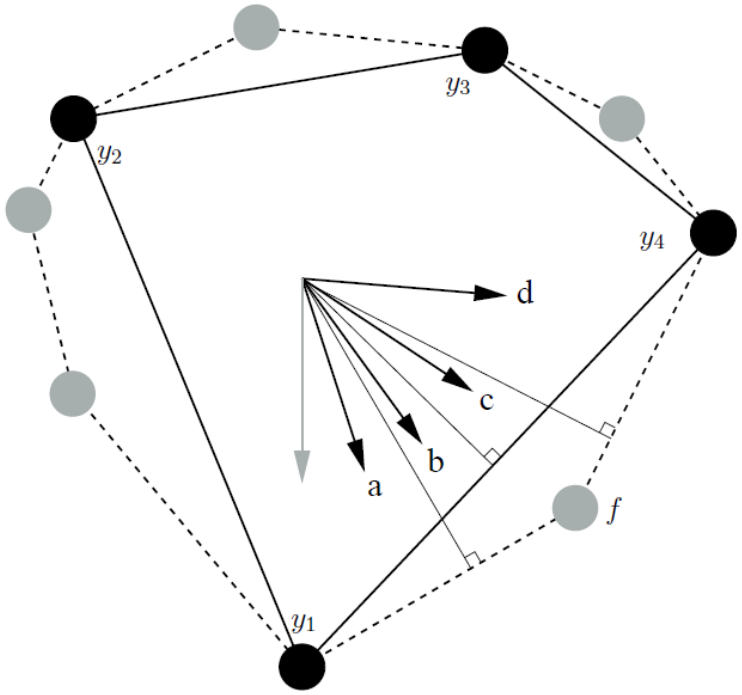

Figure (22): A decoding polytope $\mathcal{P}$ (dotted line) and the convex hull [28] P_{ML}(solid line) of the codewords y_1 through y_4. Also shown are the four possible cases (a-d) for the objective function.[14]

4.3 The LP Relaxation as a Decoder

Maximum-likelihood (ML) decoding can be seen as optimizing the objective function over points inside the convex hull of the codewords (solid line in Figure (22)); since every point in this convex hull is a convex combination of codewords, then the optimum will obtained at a codeword. Unfortunately, this convex hull will be too complex to represent explicitly for any code for which ML decoding is NP-hard (unless P = NP).

Our decoding algorithm using the relaxed polytope $\mathcal{P}$ is the following: solve the LP given in equation (60). If the LP solution is integral (a black vertex in Figure (22)), output the corresponding codeword. If the LP

solution is fractional (a gray vertex in Figure (22)), output "error." Note that if this algorithm outputs an integral solution (codeword), then we know it outputs the ML codeword. This is because the cost of the codeword found is at most the cost of all the points in $\mathcal{P}$, including all the othercodewords. Therefore this decoder has what we call the ML certificate property: if it outputs a codeword, it is guaranteed to be the ML codeword. This property is one of the unique advantages of LP decoding.

Theorem 4.1: *An LP decoder using a proper polytope has the* ML certificate *property: if the decoder outputs a codeword, it is guaranteed to be an ML codeword.*

***Proof**:* If the LP decoder outputs a codeword $f \in C$, then the cost of the point $f \in \mathcal{P}$is at most the cost of any point in $\mathcal{P}$. Since $\mathcal{P}$ is proper, we have $P \supseteq C$, and so f has cost at most the cost of any code word y∈ C. We conclude that f is the ML codeword.

Example 4.1: Suppose we have the linear code C = {0000, 1101, 1011, 0110}. This code can be characterized by the parity check equations: $(y_1 \oplus y_2 \oplus y_3) = 0$ and $(y_2 \oplus y_3 \oplus y_4) = 0$. We define a polytope $\mathcal{R}$ on four variables $\{f_1, f_2, f_3, f_4\}$ as the set of points that satisfy the following linear inequalities:

$$\begin{cases} f_1 \leq f_2 + f_3 \\ f_2 \leq f_1 + f_3 \\ f_3 \leq f_1 + f_2 \\ f_1 + f_2 + f_3 \leq 2 \end{cases} \quad \text{(A)}$$

$$\begin{cases} f_2 \leq f_3 + f_4 \\ f_3 \leq f_2 + f_4 \\ f_4 \leq f_2 + f_3 \\ f_2 + f_3 + f_4 \leq 2 \end{cases} \quad (B)$$

$$\begin{cases} 0 \leq f_1 \leq 1 \\ 0 \leq f_2 \leq 1 \\ 0 \leq f_3 \leq 1 \\ 0 \leq f_4 \leq 1 \end{cases} \quad (C)$$

The (C) constraints ensure that all f_i take on values between zero and one. The (A) and (B) constraints ensure that the above constraints are exactly the set of codewords of *C*. To see this, consider the (A) constraints; the binary words that satisfy these the polytope $\mathcal{R}$ is proper; i.e., the set of binary words of length four the constraints are exactly the words that satisfy the parity check equation $(y_1 \oplus y_2 \oplus y_3) = 0$. Similarly, the (B) constraints correspond to the parity check equation $(y_2 \oplus y_3 \oplus y_4) = 0$. This polytope is a special case of a general-purpose polytope for binary linear codes and LDPC codes [15, 14].

4.3.1 Noise as a perturbation of the LP objective

Suppose the relaxation has the following reasonable property: when there is no noise in the channel, the transmitted codeword *y* will be the optimum point of the LP, and thus the LP decoder will succeed. (All the relaxations we give in this thesis have this property.) Noise in the channel then amounts to a perturbation of the objective function away from the "no noise" direction. If the perturbation is small, then y will remain the optimal point of the LP. If the perturbation is large (there is a lot of noise in the channel), then *y* will no longer be optimal, and the LP decoder will fail.

In Figure (22), an objective function can be seen as a direction inside the polytope; solving the LP amounts to finding the point in the polytope that is furthest in that direction. The following mental exercise often helps visualize linear programming. Rotate the polytope such that the objective function points "down." Then, the objective function acts like gravity; if we drop a ball inside this polytope, it will settle at the point that optimizes the objective function. In Figure (22) we have rotated the polytope so that when there is no noise in the channel, then the objective function points "down," directly at the transmitted codeword (y_1 in figure(22)).

There are four cases describing the success of LP decoding, related to ML decoding. These cases are illustrated in Figure (22) by four arrows (a, b, c, d), representing directions inside the polytope. The gray arrow is the objective function without noise in the channel. The cases are described as follows:

(a) If there is very little noise, then both ML decoding and LP decoding succeed, since both still have y_1 as the optimal point.

(b) If more noise is introduced, then ML decoding succeeds, but LP decoding fails, since the fractional vertex f is optimal for the relaxation.

(c) With still more noise, ML decoding fails, since y_2 is now optimal; LP decoding still has a fractional optimum (f), so this error is detected.

(d) Finally, with a lot of noise, both ML decoding and LP decoding have y_2 as the optimum, so both fail, and the error is undetected.

Remark 4.1: in the last two cases when ML decoding fails, this is in some sense the fault of the code itself, rather than the decoder.

4.4 Success Conditions for LP Decoding

Overall, the LP decoder succeeds if the transmitted codeword is the unique optimal solution to the LP. The decoder fails if the transmitted codeword is not an optimal solution to the LP. In the case of multiple LP optima (which for many noise models has zero probability), we will be conservative and assume that the LP decoder fails. Therefore, we have the following theorem:

Theorem 4.2: For any binary-input memoryless channel, an LP decoder using polytope $\mathcal{P}$ will fail if and only if there is some point in $\mathcal{P}$ other than the transmitted codeword y with cost less than or equal to the cost of y.

Remark 4.2: We use WER_y to denote the word error rate (WER) of the LP decoder, given a particular transmitted codeword y. By Theorem, we have:

$$WER_y = \Pr\left[\ \exists\ f \in \mathcal{P}, f \neq y\colon \sum_i \gamma_i f_i \ \leq \sum_i \gamma_i y_i\ \right] \tag{61}$$

4.5 Vertices, Codewords and Pseudocodewords

Definition 4.2: An ***extreme point***, or equivalently a *vertex* of a polytope is a point that cannot be expressed as the convex combination of other points in the polytope.

Let $\mathcal{V}(\mathcal{P})$ be the set of vertices of the polytope $\mathcal{P}$. A fundamental fact of linear programming is that the optimal solution to an LP can always be found at a vertex of the polytope associated with the LP [1]. Therefore, the LP decoder will always find the *lowest cost* vertex of the polytope $\mathcal{P}$.

Theorem 4.3: *For any polytope* $\mathcal{P} \subseteq [0,1]^n$ *that is proper for* C*, every codeword* $y \in C$ *is avertex of* $\mathcal{P}$.

It is important to note that the converse statement (i.e., every polytope vertex is a codeword) may *not* hold, however, since the polytope could have *fractional* (non-integral) vertices. So, in general we have

$$C \subseteq \mathcal{V}(\mathcal{P}) \subseteq \mathcal{P} \subseteq [0,1]^n.$$

In LP decoding, vertices take on the role of *pseudocodewords* : the set of possible results that a sub-optimal decoder may produce.

Definition 4.3: Pseudocodewords are a superset of the codewords, and may contain "false" codewords that "fool" the algorithm. While the set of codewords is a function of the code itself, the set of pseudocodewords is a function of the sub-optimal decoding algorithm being used.

Example 4.2: Consider the polytope $\mathcal{R}$ designed earlier for the code C = {0000, 1101, 1011, 0110}. The vertices of this polytope include the codewords, as well as the fractional vertices $(1, \frac{1}{2}, \frac{1}{2}, 0)$ and $(0, \frac{1}{2}, \frac{1}{2}, 1)$.

Note that neither of the fractional vertices can be expressed as convex combinations of codewords. We have $\mathcal{V}(\mathcal{R})$ = {(0, 0, 0, 0), (1, 1, 0, 1),(1, 0, 1, 1), (1, 1, 1, 1), $(1, \frac{1}{2}, \frac{1}{2}, 0)$, $(0, \frac{1}{2}, \frac{1}{2}, 1)$}.This is the set of *pseudocodewords* for the LP decoder using $\mathcal{R}$ on this code.

4.6 The Fractional Distance

We motivate the definition of fractional distance by providing an alternate definition for the (classical) distance in terms of a proper polytope $\mathcal{P}$. Recall that in a proper polytope $\mathcal{P}$, there is a one-to-one correspondence between codewords and integral vertices of $\mathcal{P}$; i.e.,

$\boldsymbol{C}= \mathcal{P} \cap \{0,1\}^n$. The Hamming distance between two points in the discrete space $\{0,1\}^n$ is equivalent to the l_1 distance between the points in the space$[0,1]^n$.Therefore, given a proper polytope $\mathcal{P}$, we may define the distance of a code as the minimum l_1 distance between two integral vertices, i.e.,

$$d = min_{y,y' \in (\mathcal{V}(\mathcal{P}) \cap \{0,1\}^n) \atop y \neq y'} \quad \sum_{i=1}^{n} | y_i - y_i' |$$

The LP polytope $\mathcal{P}$ may have additional non-integral vertices, as illustrated in Figure (22). We define the fractional distance d_{frac} of a polytope $\mathcal{P}$ as the minimum $l_1$distance between an integral vertex (codeword) and any other vertex of $\mathcal{P}$; i.e.,

$$d_{frac} = min_{\substack{y \in \boldsymbol{C} \\ f \in \mathcal{V}(\mathcal{P}) \\ f \neq y}} \sum_{i=1}^{n} |y_i - f_i|$$

Note that this fractional distance is always a lower bound on the classical distance of the code, since every codeword is a polytope vertex (in the set $\mathcal{V}(\mathcal{P})$). Moreover, the performance of LP decoding is tied to this fractional distance, as we make precision the following:

Theorem 4.4: Let $\boldsymbol{C}$ be a binary code and $\mathcal{P}$ a proper polytope in an LP relaxation for $\boldsymbol{C}$. If the fractional distance of $\mathcal{P}$ is d_{frac}, then the LP decoder using $\mathcal{P}$ is successful if at most $\lceil d_{frac}/2 \rceil$ - 1 bits are flipped by the binary symmetric channel.

Proof: Let *y* be the codeword transmitted over the channel. Suppose the LP decoder fails; i.e., *y* is not the unique optimum solution to the LP. Then there must be some other vertex $f^* \in \mathcal{V}(\mathcal{P})$ (where $f^* \neq$ y) that is an

optimum solution to the LP, since the LP optimum is always obtained at a vertex. By the definition of fractional distance, we have

$$\sum_{i=1}^{n} |f_i^* - y_i| \geq d_{frac}$$

For all bits $i \in \{1, \ldots n\}$, let $f_i = |f_i^* - y_i|$. From the above equation, we have:

$$\sum_{i=1}^{n} f_i \geq d_{frac} \quad (62)$$

Let $\varepsilon = \{i: \tilde{y}_i \neq y_i\}$ be the set of bits flipped by the channel. By assumption, we have that:

$$|\varepsilon| \leq \lceil d_{frac}/2 \rceil - 1$$

and so

$$\sum_{i \in \varepsilon} f_i \leq \lceil d_{frac}/2 \rceil - 1 \quad (63)$$

Since all $f_i \leq 1$. from (62) and (63), it follows that

$$\sum_{i \notin \varepsilon} f_i \geq \lfloor d_{frac}/2 \rfloor + 1 \quad (64)$$

Therefore from (63) and (64), we have

$$\sum_{i \notin \varepsilon} f_i - \sum_{i \in \varepsilon} f_i > 0 \quad (65)$$

Since f^* is an optimum solution to the LP, its cost must be less than or equal to the cost of y under the objective function γ; i.e.,

$$\sum_{i=1}^{n} \gamma_i f_i^* - \sum_{i=1}^{n} \gamma_i y_i \leq 0 \quad (66)$$

We can rewrite the left side of equation (66) as follows

$$\sum_{i=1}^{n} \gamma_i f_i^* - \sum_{i=1}^{n} \gamma_i y_i = \sum_{i=1}^{n} \gamma_i (f_i^* - y_i)$$

$$= \sum_{i:y_i=0} \gamma_i f_i^* - \sum_{i:y_i=1} \gamma_i (1 - f_i^*)$$

$$= \sum_{i:y_i=0} \gamma_i f_i - \sum_{i:y_i=1} \gamma_i f_i \qquad (67)$$

$$= \sum_{i \notin \varepsilon} f_i - \sum_{i \in \varepsilon} f_i . \qquad (68)$$

Equation (67) and (68) follows from the fact that

$$f_i = \begin{cases} f_i^* & if \quad y_i = 0 \\ 1 - f_i^* & if \quad y_i = 1 \end{cases}$$

Equation (68) follows from the fact under the BSC, we have $\gamma_i = -1$ if $\tilde{y}_i = 1,$ and $\gamma_i = +1$ if $\tilde{y}_i = 0$. from (66) and (68), it follows that

$\sum_{i \notin \varepsilon} f_i - \sum_{i \in \varepsilon} f_i \leq 0$, which contradicts (65).

Chapter 5

LP Decoding of Turbo Codes

In this chapter we define a linear program (LP) to decode any turbo code. We will motivate the general LP by first defining an LP for any convolutional code. In fact, we will give an LP for a more general class of codes: any code that can be defined by a finite state machine. Then, we use this LP as a component in a general LP for any turbo code. We discuss the success conditions of this decoder, which are related to the optimality conditions for the network flow problem.

5.1 Trellis-Based Codes

In this section we define a linear program for codes represented by a trellis. This family of codes is quite general; in fact trellises in their most general form may be used to model any code (even non-binary codes). The trellis is a directed graph where paths in the graph represent codewords. The trellis representation is most useful when the code admits a small-sized trellis; in this case, ML decoding can be performed using the Viterbi algorithm, which essentially finds the shortest path in the trellis.

5.1.1 Finite State Machine Codes and the Trellis

Definition 5.1: Let M be a finite-state machine (FSM) over the input alphabet $\{0, 1\}$. An FSM M is simply a directed graph made up of states. Each state (node) in the graph has two outgoing edges, an input-0 edge and an input-1 edge. For a given edge e, we use $type(e) \in \{0, 1\}$ to denote the "type" of the edge. In diagrams of the FSM, we use solid lines to denote input-0 edges, and dotted lines to denote input-1 edges. Each edge has an

associated output label made up of exactly *R* bits. For an edge *e*, we use *label (e)* ∈ $\{0, 1\}^R$ to denote the output label.

The FSM can be seen as an encoder as follows. When the machine receives a block of input bits, it examines each bit in order; if the bit is a zero, the machine follows the input-0 edge from its current state to determine its new state, and outputs the output label associated with the edge it just traveled. If the bit is a one, it follows the input-1 edge. The overall rate of this encoding process is *r = 1/R*.

For example, consider the FSM in figure (23). Suppose we start in state 00, and would like to encode the information bits 1010. Our first information bit is a 1, so we follow an input-1 edge (dotted line) to the new state 10, and output the label 11. The next information bit is 0, so we follow the input-0 edge (solid line) to the new state 01, and output the label 10. This continues for the next two information bits, and overall, we travel the path 00 →10 →01 →10 →01, and output the code bits11100010.

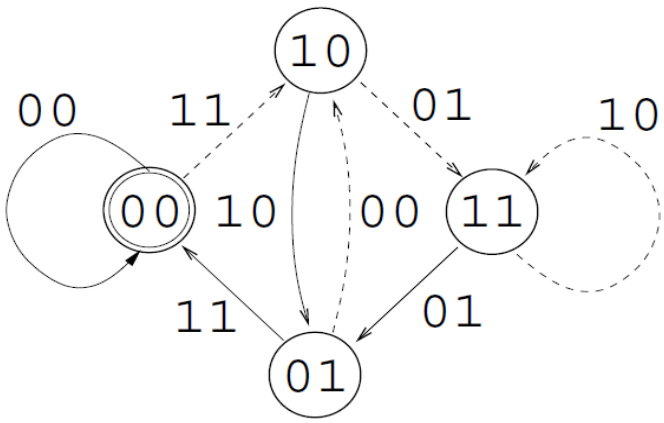

Figure (23) : A state machine code with a rate-1/2. The bit pairs on the transition edges represent encoded output bits.

Definition 5.2: The Trellis In order to simulate the operation of a single FSM *M* over some number *k* of input bits; we will define a trellis *T*. The trellis *T* is a graph with *k + 1* copies ($M_0, \ldots, M_k$) of the states of *M* as nodes.

The trellis *T* has edges between every consecutive copy of *M*, representing the transitions of the encoder at each time step, as it encodes an information word of length *k*. So, for each edge (*s*, *s'*) connecting state *s* to state *s'* in the FSM *M*, the trellis contains, for each time step *t* where $1 \leq t \leq k$, an edge from state *s* in M_{t-1} to state *s'* in M_t. This edge will inherit the same type and output label from edge (*s*, *s'*) in *M*. Figure (24) gives the trellis for the FSM in Figure (23), where $k = 4$. The bold path in Figure (23) indicates the path taken by the encoder in the example given earlier, when encoding 1010**.**

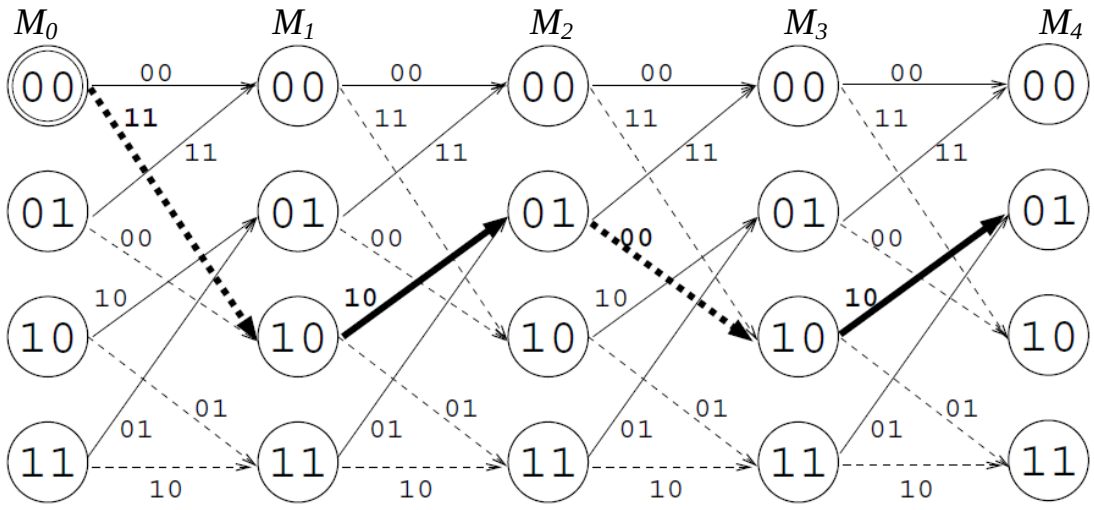

Figure (24): A trellis for the rate-1/2 FSM code in figure (23). The trellis has length $k = 4$, and five node layers (M_0, M_1, M_2, M_3, M_4). As in the state transition table, the bit pairs on the transition edges represent encoded output bits. Each layer of the trellis represents one input bit. The bold path is the path taken by the encoder while encoding the information bits 1010.

We can view the encoder as building a path through the trellis, as follows. Denote a "start-state" of the FSM. The trellis length *k* is equal to the length of the information word. The FSM encoder begins in the start state in M_0, uses the information word *x* as input, and outputs a codeword *y*. The length of this codeword will be $kR = k/r = n$. Since every state has exactly two outgoing edges, there are exactly 2^k paths of length *k* from the start state, all ending in trellis layer M_k. So, the set of codewords is in one-to-one correspondence the set of paths of length *k* from the start state in M_0.

Now we need to establish some notation we will use through our study of trellis-based codes and turbo codes.

Definition 5.3: for each node *s* in a trellis, define:

1- *out(s)*: the set of outgoing edges from s.

2- *in(s)*: the set of incoming edges.

3- for a set of nodes *S*, define *out(s)* and *in(s)* to be the set of outgoing and incoming edges from the node set *S*.

Let I_t be the set of " input- 1" edges entering layer *t*. formally,

$$I_t = \{ e \in in(M_t) \quad : \quad type(e) = 1 \}$$

we also define edge sets O_i, where an edge is in O_i if it outputs a 1 for the i^{th} code bit. Formally,

$$O_i = \{ e \in in(M_t) \quad : \quad label(e)_{\ell} = 1 \}$$

where $t = \lfloor (i-1)/R \rfloor + 1$ and $\ell = i - R(t-1)$.

5.1.2 Decoding Trellis-Based Codes

Assign a cost to each edge of the trellis, and then finding the lowest-cost path of length *k* from the start state of the trellis.

For each edge *e* in *T*, define a cost γ_e. This is also referred to as the *branch metric*. This cost will be the sum of the costs *i* of the code bits y_i in the output label (denoted *label (e)*) of the edge that are set to 1. Where the cost γ_i of a code bit is the log-likelihood ratio of the bit i.e

$$\gamma_i = \ln\left(\frac{\Pr(\tilde{y}_i \mid y_i=0)}{\Pr(\tilde{y}_i \mid y_i=1)}\right)$$

more formally,

$$\gamma_e = \sum_{i:e \in O_i} \gamma_i.$$

5.1.3 Convolutional Codes

Convolutional codes can be seen as a particular class of FSM-based codes. Their simple encoder and decoder have made them a very popular class of codes. In this section we describe the basics of convolutional codes.

The state of a convolutional encoder is described simply by the last *k-1* bits fed into it, where *k* is defined as the constraint length of the code. So, as the encoder processes each new information bit, it remembers the last *k- 1* information bits.

The output (code bits) at each step is simply the sum (mod 2) of certain subsets of the last *k- 1* input bits. (We note that convolutional codes with feedback cannot be described this way, but can still be expressed as FSM codes.)

Convolutional encoders are often described in terms of a circuit diagram, such as the one in Figure (3.1). These diagrams also illustrate how simple these encoders are to build into hardware (or software). The circuit for a convolutional encoder with constraint length *k* has *k-1* registers, each holding one bit. In general, at time *t* of the encoding process, the i_{th} register holds the input bit seen at time *t - i*. In addition, there are *R* output bits, each one the mod-2 sum of a subset of the registers and possibly the input bit. The connections in the circuit indicate which registers (and/or the input bit) are included in this sum.

For example, in Figure (25), the constraint length is 3, and so there are 2 registers. The rate is 1/2, so there are two output bits. The bits $(x_1, x_2, \ldots, x_k)$ are fed into the circuit one at a time. Suppose the current 'time" is *t*. The first output bit is the sum of the current input bit (at time *t*) and the contents of the second register: the bit seen at time *t-2*. The second output bit is the sum of the input bit, and the two registers: the bits seen at time *t - 1* and *t - 2*.

In figure (25) The actions of a convolutional encoder for a rate-1/2 convolutional code From its initial state 00, given an input stream of 100. Diagram (a) shows the initial state of the encoder, with both memory elements set to 0. The first input bit is a 1. Upon seeing this bit, the encoder outputs 1 + 0 = 1 for the first code bit, and 1 + 0 + 0 = 1 for the second code bit. Diagram (b) shows the encoder after processing the first input bit; the memory elements have slid over one step, so the new state is 10. The new input bit is 0, and so the next output bits are 0 + 0 = 0 and 0 + 1 + 0 =

1. Diagram (c) shows the next state (01), the new input bit (0) and the next output bits (1 and 1). Finally, diagram (d) shows the final state 00 after processing all three input bits.

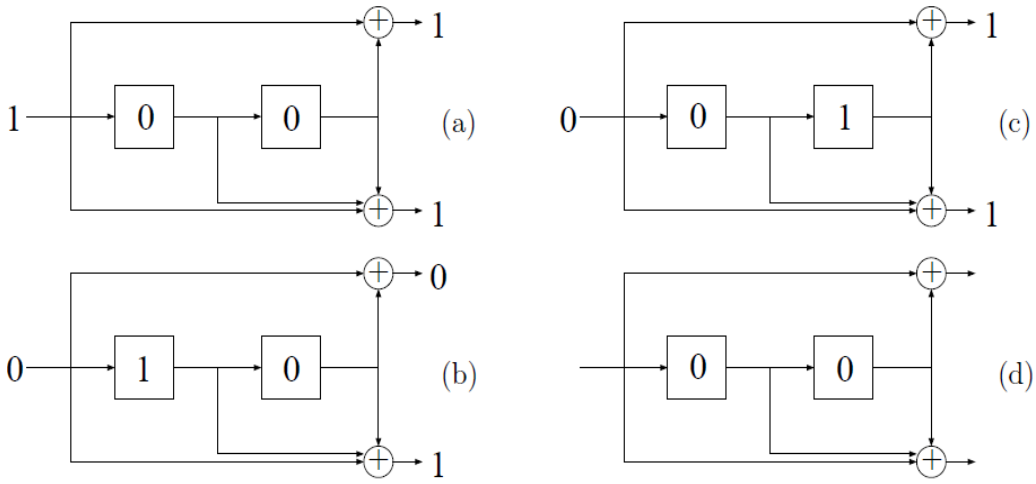

Figure (25): The actions of a convolutional encoder for a rate-1/2 convolutional code

The FSM and Trellis For Convolutional Codes: We can also describe a convolutional code using a finite state machine. The state s of a convolutional encoder can be described by the contents of the registers in the circuit, so $s \in \{0,1\}^{k-1}$.

The FSM for our running example is the same one shown in Figure (23). This is simply a graph with one node for each possible state $s \in \{0,1\}^{k-1}$of the circuit in Figure (25). For each node with state s, we draw two edges to other states, representing the two possible transitions from that state (one for an input bit of 1, one for an input bit of 0). As before, the edges are solid lines if the transition occurs for an input bit of 0 and dotted lines for an input bit of 1.

The edges are labeled as before, with the bits output by the encoder circuit making this transition. For example, if we take the first step in Figure (25),

the current state is 00, the input bit is a 1, the next state is 10, and the output bits are 11. So, in our FSM graph we make an edge from state 00, written with a dotted line, into state 10, and label the edge with code bits 11. We derive the trellis for a convolutional code from the FSM, as we did in the previous section. The trellis for the code in Figure (25) is shown in Figure (24).

Feedback If a convolutional code has feedback, then its new state is not determined by the previous *k-1* information bits, but rather is determined by the previous *k-1* ***feedback*** bits. The ***feedback*** bit at time *t* is the sum (mod 2) of the information bit at time *t* and some subset of the previous *k-1* ***feedback*** bits. These codes still have a simple state transition table, and can be decoded with a trellis.

5.2 LP Formulation of Trellis Decoding

Since every cycle in the trellis passes through M_0, the ML decoding problem on tail-biting trellises is easily solved by performing $|M_0|$ shortest path computations. However, we present a sub-optimal LP (min-cost flow) formulation of decoding in order to make it generalize more easily to turbo codes, where we have a set of trellises with dependencies between them.

We define a variable f_e for each edge *e* in the trellis *T*, where $0 \leq f_e \leq 1$. Our LP is simply a min-cost circulation LP applied to the trellis *T*, with costs *e*. Specifically, our objective function is:

$$\text{Minimize } \sum_{e \in T} \gamma_e f_e$$

We have the conversation constraints on each node:

$$\sum_{e \in out(s)} f_e = \sum_{e \in in(s)} f_e \tag{68}$$

We also force the total flow around the trellis to be one. Since every cycle in *T* must pass between layers M_0 and M_1, it suffices to enforce:

$$\sum_{e \in out(M_0)} f_e = 1 \tag{69}$$

We define auxiliary LP variables ($x_1, \ldots, x_k$) for each information bit, and LP variables ($y_1, \ldots, y_n$) for each code bit. These variables indicate the value that the LP assigns to each bit. They are not necessary in the definition of the linear program, but make the connection to decoding much clearer. The value of an information bit x_t should indicate what "type" of edge is used at layer *t* of the trellis; if $x_i = 1$, this indicates that the encoder used an input-1 edge at the i^{th} trellis layer. Accordingly, we set

$$x_t = \sum_{e \in I_t} f_e \tag{70}$$

The value of y_i indicates the i^{th} bit output by the encoder. Therefore, if $y_i = 1$, then the edge taken by the encoder that outputs the i^{th} bit should have a 1 in the proper position of the edge label. To enforce this in the LP, we set

$$y_i = \sum_{e \in O_i} f_e \tag{71}$$

Since all variables are indicators, we enforce the constraints

$$\begin{aligned} 0 \le f_e \le 1 \quad & \text{for all } e \in T \\ 0 \le x_t \le 1 \quad & \text{for all } t \in \{1, \ldots, k\} \quad \text{and} \\ 0 \le y_i \le 1 \quad & \text{for all } i \in \{1, \ldots, n\} \end{aligned} \tag{72}$$

We use the notation *poly(T)* to denote the polytope corresponding to these constraints; formally,

$$\text{ploy}(T) = \{(f, x, y): \text{ equations (68) – (71) holds}\}$$

Overall, our LP can be stated as:

$$\text{Minimize } \sum_{e \in T} \gamma_e f_e \quad \text{s.t. } (f, x, y) \in \text{ploy}(T)$$

Or , equivalently,

$$\text{Minimize } \sum_{i=1}^{n} \gamma_i f_i \quad \text{s.t.} \quad (f, x, y) \in \text{ploy}(T)$$

5.3 A Linear Program for Turbo Codes

In general, a turbo code is any set of convolutional codes, concatenated in serial or parallel, with interleavers between them. These are often referred to in the literature as "turbo-like" codes [1], because they are a generalization of the original turbo code [3]. In this section we describe an LP to decode any turbo code. In fact, we will describe an LP for a more general class of code realizations, defined on trellises built from arbitrary finite-state machines, and associations between sets of edges in the trellises.

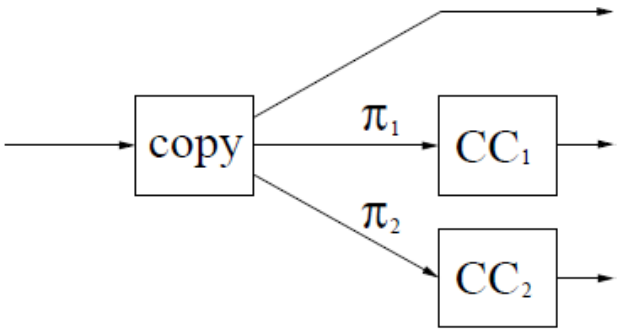

Figure (26): A circuit diagram for a classic rate-1/3 Turbo code. The "copy" FSM simply copies the input to the three outputs. The two component codes CC1 and CC2 are convolutional codes.

Turbo Codes

The original turbo code consists of two component rate-1 convolutional codes concatenated in parallel, with an interleaver in front of each of them. Additionally, the information bits themselves are added to the codeword. Figure (26), shows a tree representation of this code. The encoder for the turbo code in Figure (26) takes as input an information word *x* of length *k*. Two separate interleaves (fixed permutations) π_1 and π_2 are applied to the information word. Then, the length-*k* words π_1 (*x*) and π_2 (*x*) are sent to the encoders for rate-1 convolutional codes CC1 and CC2. The output of these encoders, along with a copy of the information word *x*, is output as the codeword.

In general we will define a turbo code by a directed out-tree $\boldsymbol{\mathcal{T}}$,whose nodes correspond to trellises *T*. Each edge in the tree has an associated interleaver π ; this is a permutation on *k* elements, where *k* is the length of the trellis whose corresponding node the edge enters.

Formally, let $\boldsymbol{\mathcal{T}}$ be a directed out-tree, where the nodes {1, … , $|\boldsymbol{\mathcal{T}}|$} correspond to trellises $\{T^1, \ldots, T^{|\mathcal{T}|}\}$. We assume that each trellis has tailbiting. By convention, we have node 1 in $\boldsymbol{\mathcal{T}}$ as the root, corresponding to trellis T^1. Let k_m denote the length of trellis T^m, $1 \leq m \leq |\boldsymbol{\mathcal{T}}|$, and let R_m denote the length of the output labels on edges of trellis T^m. For each edge from *m* to *m′* in the tree , there is an interleaver π_1 [m, m'], a permutation on $k_{m'}$ elements. Let L(T) denote the set of leaves of $\boldsymbol{\mathcal{T}}$ (the nodes in $\boldsymbol{\mathcal{T}}$ with no outgoing edges).

The encoder for a turbo code takes a block of *k* information bits, feeds it into the trellis at the root, and sends it through the tree. The codeword is output at the leaves of the tree. An individual trellis T^m of size k_m receives a block of bits of size k_m from its parent in , applies its encoding process to the block, and sends a copy of its output block of size $R_m k_m$ to each of its children in $\mathcal{T}$. Each edge applies its permutation to the bits sent across it. For the encoder to work properly, it must be the case that for every edge from *m* to *m'* in, we have $R_m k_m = k_{m'}$, so the number of output bits of trellis T^m is equal to the number of input bits for trellis $T^{m'}$.

The overall codeword is the concatenation of all the outputs of the leaf trellises of . For a trellis T^m where $m \in L(\mathcal{T})$, we use y_m to denote the string of $R_m k_m$ code bits output by trellis T^m. The overall code length is then n $= \sum_{m \in L(\mathcal{T})} R_m k_m$, and the overall rate is *k/n*. This codeword is transmitted over the channel, and a corrupt codeword $\tilde{y}$ is received. We use $\tilde{y}^m$ to denote the corrupt bits corresponding to the code bits y^m, for some $m \in L(\mathcal{T})$.

5.3.1 Turbo-Code Linear Program (TLCP)

We will define the Turbo Code Linear Program (TCLP), a generic LP that decodes any turbo code. We will first outline all the variables of TCLP, then give the objective function and the constraints. Basically, the LP will consist of the min-cost flow LPs associated with each trellis, as well as "agreeability constraints," tying together adjacent trellises in $\mathcal{T}$.

i. **Variables**: All variables in TCLP are indicator variables in {0, 1}, relaxed to be between 0 and 1. For each m $\in \mathcal{T}$, we have variables x^m

= (x_1^m ,... , $x_{k_m}^m$) to denote the bits entering trellis T^m. The TCLP variables x^1 = (x_1^1 ,... , $x_{k_1=k}^1$) represent the information bits, since they are fed into the root trellis T^1. Additionally, we have variables y^m = (y_1^m ,... , $y_{R_m k_m}^m$)to denote the output bits of each trellis T^m. Finally, for each m ∈ $\mathcal{T}$, and edge e in trellis T^m, we have a flow variable f_e; we use the notation f^m to denote the vector of flow variables f_e for all edges e in trellis T^m.

ii. **Objective Function**: The objective function will be to minimize the cost of the code bits. Since the code bits of the overall code are only those output by the leaves of the encoder, we only have costs associated with trellises that are leaves in $\boldsymbol{\mathcal{T}}$**.** For each trellis T^m where m ∈ L($\boldsymbol{\mathcal{T}}$),we have a cost vector $\gamma^m$with a cost γ_i^m for all i∈ {1, … ,$R_m k_m$ }. These costs are associated with the bits γ_i^m output by the encoder for trellis T^m. The value of γ_i^m is the log-likelihood ratio for the bit y_i, given the received bit $\tilde{y}_i^m$. Thus our objective function is simply:

$$\text{minimize } \sum_{m \in L(T)} \sum_{i=1}^{R_m k_m} \gamma_i^m \, y_i^m.$$

iii. **Constraints**: Recall that for a trellis *T*, the polytope poly(*T*) is the set of unit circulations around the trellis *T* (defined before). In TCLP, we have all the constraints of our original trellis polytope poly(T^m) for each trellis T^m in the tree $\boldsymbol{\mathcal{T}}$. In addition, we have equality constraints to enforce that the output of a trellis is consistent with the input to each of the child trellises. We define the LP constraints formally as follows:

1. **Individual trellis constraints:** for all m ∈ $\mathcal{T}$ **,**

$$(f^m, x^m, y^m) \in \text{ply}(T^m).$$

These constraints enforce that the values f^m are a unit circulation around the trellis T^m, and that x^m and y^m are consistent with that circulation

2. **Interleaver consistency:** For all edges $(m, m') \in \boldsymbol{\mathcal{T}}$, for all t ∈ $\{1,\ldots, k_{m'}\}$,

$$y_t^m = x_{\pi[m,m'](t)}^{m'}$$

These constraints enforce consistency between trellises in the tree $\boldsymbol{\mathcal{T}}$ that sharean edge, using the interleaver associated with the edge.

A decoder based on TCLP has the ML certificate property for the following reasons. Every integral solution to TCLP corresponds to a single cycle in each trellis Furthermore, the consistency constraints enforce a correspondence between adjacent trellises in , and the output bits $\{y^m$: m ∈ L($\boldsymbol{\mathcal{T}}$)$\}$. We may conclude that every integral solution to this LP represents a valid set of encoding paths for an information word *x*, and the cost of the solution is exactly the cost of the codeword output by this encoding at the leaves. Thus this LP has the ML certificate property.

Chapter 6

Conclusions and Future Work

1- The work in this thesis represents the consideration of the application of linear programming relaxation to the problem of decoding an error-correcting code. We have been successful in applying the technique to the modern code families of turbo codes and LDPC codes, and have proved a number of results on error-correcting performance. However, there is much to understand even within these families, let alone in other code families we have yet to explore. In this final chapter we survey some of the major open questions in LP decoding of turbo codes and LPDC codes, and suggest a number of general ideas for future research in this area.

2- There are many interesting open questions regarding the relationship between message-passing algorithms and LP decoding. One needs to run an interior point algorithm to solve the LP. It would be interesting to see if there was a primal-dual or other combinatorial algorithm with a provably efficient running time. An interesting question is how modifying belief propagation to be "cost-balanced" affects performance.

3- Improving the Running Time a drawback of the LP approach to turbo decoding is the complexity of solving a linear program. Even though the simplex algorithm runs quite fast in practice, most applications of error-correcting codes require a more efficient decoding algorithm. The specialized structure of TCLP may allow a

combinatorial solution. In general, it is an interesting question to determine how the min-cost agreeable flow problem must be restricted in order to make it solvable combinatorially.

4- In this thesis we study binary linear codes in memoryless symmetric channels. There is certainly a way to generalize the notion of LP decoding for more general codes and channels, and this opens up a whole new set of questions.

In many applications, we operate over larger alphabets than binary (for example in the transmission of internet packets). We could model this in an LP by having a variable range over this larger space, or by using several 0 - 1 variables as indicators of a symbol taking on a particular value. Alternatively, we could map the code to a binary code, and use an LP relaxation for the binary code. It would be interesting to see if anything is gained by representing the larger alphabet explicitly.

In practice, channels are generally not memoryless due to physical effects in the communication channel. Even coming up with a proper linear cost function for an LP to use in these channels in an interesting question. The notions of pseudocodeword and fractional distance would also need to be reconsidered for this setting.

5- In all the published research we found that Additive White Gaussian Channel with normal distribution having variance one is used, we know that in general the logistic distribution is better than the normal distribution when having the same mean and variance making, we

study the case of channel having a logistic distribution with variance one and compare our results with the normal distribution, we found that our results were more accurate to determine wither the receiving digits are 0 or 1. For example, it's clear that in the case of normal distribution the probability of choosing 1 was 0.95 while we were more accurate and we choose 1 under a probability exactly 0.98 .

6- A Final Remark

The work in this thesis represents the exploration of a computer science theorist into the world of error-correcting codes. We have discovered that many standard techniques in theoretical computer science can help shed light on the algorithmic issues in coding theory. The author therefore encourages communication between these two fields, and hopes that this work serves as an example of the gains that can result.

References

[1] A. Schrijver. **Theory of Linear Programming**. John Wiely, 1987.

[2] Bahl, L. R., Cocke, J., Jeinek, F., and Raviv, J., **"Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate,"** ***Trans. Inform. Theory,*** vol. IT-20, March 1974, pp. 248-287.

[3] Berrou, C., Glavieux, A., and Thitimajshima, P., **"Near Shannon Limit Error-Correcting Coding and Decoding: Turbo Codes,"** ***IEEE Proceedings of the Int. Conf. on Communications,*** Geneva, Switzerland, May 1993 (ICC'93), pp. 1064-1070.

[4] Berrou, C. and Glavieux, A., **"Near Optimum Error Correcting Coding and Decoding: Turbo-Codes,"** ***IEEE Trans. on Communications,*** vol. 44, no. 10, October 1996, pp. 1261-1271.

[5] C. E. Shannon. **A mathematical theory of communication**. ***Bell System Technical Journal,*** 27:379-423, 623-656, 1948.

[6] Derand Sklar. Fundamental of turbo codes http://www.informit.com/articles,2002.

[7] Divsalar, D. and McEliece, R. J., **"Effective Free Distance of Turbo Codes,"** ***Electronic Letters,*** vol. 32, no. 5, Feb. 29, 1996, pp. 445-446.

[8] Divsalar, D. and Pollara, F., **"On the Design of Turbo Codes,"** ***TDA Progress Report 42-123,*** Jet Propulsion Laboratory, Pasadena, California, November 15, 1995, pp. 99-121.

[9] Divsalar, D. and Pollara, F., **"Turbo Codes for PCS Applications,"** ***Proc. ICC '95,*** Seattle, Washington, June 18-22, 1995.

[10] Dolinar, S. and Divsalar, D., **"Weight Distributions for Turbo Codes Using Random and Nonrandom Permutations,"** ***TDA Progress Report 42-122,*** Jet Propulsion Laboratory, Pasadena, California, August 15, 1995, pp. 56-65.

[11] D. Hochbaum, editor. ***Approximation Algorithms for NP-hard Problems*****.** PWS Publishing, 1995.

[12] F. J. Macwilliams and N. J. A. Sloane, **The Theory of Error Correcting Codes. Amsterdam,** The Netherlands: North-Holland, 1978, pp. 567–580.

[13] Hagenauer, J., **"Iterative Decoding of Binary Block and Convolutional Codes,"** ***IEEE Trans. on Information Theory,*** vol. 42, no. 2, March 1996, pp. 429-445.

[14] Jon Feldman, ***Decoding error-correcting codes via linear programming,*** Ph.D. thesis, Massachusetts Institute of Technology, 2003.

[15] J. Feldman, M. J. Wainwright, and D. R. Karger. **Using linear programming to decode linear codes**. 37th annual Conference on Information Sciences and Systems (CISS '03) , March 2003. Submitted to IEEE Transactions on Information Theory , May, 2003.

[16] Jon Feldman, Martin J.Wainwright, and David R. Karger, ***Using linear programming to decode binary linear codes,*** **IEEE Transactions on Information Theory**, 51(3), pp. 954.972. (2005).

[17] M. Grotschel, L. Lov´asz, and A. Schrijver. **The ellipsoid method and its consequences in combi-natorial optimization. Combinatorica ,** 1(2):169–197, 1981.

[18] Mohammad H. Taghavi N. and Paul H. Siegel**,** ***Adaptive linear programming decoding***, (2006).

[19] Moon T. Error correction coding: **Mathematical Methods and Algorithms. United States of America:** John Wiley and Sons; 2005.

[20] Pietrobon, S., "Implementation and Performance of a Turbo/MAP Decoder," *Int'l. J. Satellite Commun.*, vol. 16, Jan-Feb 1998, pp. 23-46.

[21] P. Elias, **"Error-free coding," IRE Trans. Inform Theory,** vol. IT-4, pp. 29–37, Sept. 1954.

[22] Ramjee Prasad, **"OFDM for Wireless Communications systems"**, Artech House Publishers, 2004.

[23] Robert H. Morelos-Zaragoza, ***The art of error correcting coding*, John Wiley & Sons,**Ltd, 2002, ISBN: 0471 49581 6.

[24] Robertson, P., Villebrun, E., and Hoeher, P**., "A Comparison of Optimal and Sub-Optimal MAP Decoding Algorithms Operating in the Log Domain,"** ***Proc. of ICC '95***, Seattle, Washington, June 1995, pp. 1009-1013

[25] R. W. Hamming. **Error detecting and error correcting codes**. ***The Bell System Tech. Journal***, XXIX(2):147-160, 1950.

[26] University of South Australia, Institute for Telecommunications Research, ***Turbo coding research group***. http://www.itr.unisa.edu.au/~steven/turbo/.

[27] **Wikipedia.org** , *Np-hard*, 2006, Internet: http://en.wikipedia.org/wiki/NP-hard.

[28] **Wikipedia.org**, *Convex hull*, 2006, Internet: http://en.wikipedia.org/wiki/Convex_hull.

[29] http://en.wikipedia.org/wiki/Eb/N0

Appendix A

Matlab program for Additive White Gaussian Noise channel (normal distribution with variance one).

```
clc
clear
t = input('received message')
n= input(' number of iterations')
l(1)=0;
l(2)=0;
l(3)=0;
l(4)=0;
x(1) = t(1);
x(2) = t(2);
x(3) = t(3);
x(4) = t(4);
y(1,2)= t(5);
y(3,4)= t(6);
y(1,3)= t(7);
y(2,4)= t(8);
lc(1)= 2*x(1)
lc(2)= 2*x(2)
lc(3)= 2*x(3)
lc(4)= 2*x(4)
ly(1,2)= 2*y(1,2)
ly(3,4)= 2*y(3,4)
ly(1,3)= 2*y(1,3)
ly(2,4)= 2*y(2,4)
for k= 1:n
for i= 1 : 4;
    if mod(i,2)==1
        lh(i)=-1*sign(lc(i+1) + l(i+1) ) *sign(ly(i,i+1))
*min(abs(lc(i+1) + l(i+1)),(ly(i,i+1)))
    else
        lh(i)=-1*sign(lc(i-1) + l(i-1) )*sign(ly(i-1,i))
*min(abs(lc(i-1) + l(i-1)),(ly(i-1,i)))
    end
end
for i = 1 :4;
    l(i) = lh(i)
end
for i= 1 : 4;
    if abs(i -1.5) == .5
```

```
        lv(i)=-1*sign(lc(i+2) + l(i+2) )
*sign(ly(i,i+2))*min(abs(lc(i+2) + l(i+2)),(ly(i,i+2)))
    else
        lv(i)=-1*sign(lc(i-2) + l(i-2) )*sign(ly(i-2,i))
*min(abs(lc(i-2) + l(i-2)),(ly(i-2,i)))
    end
end
for i = 1 :4;
    l(i) = lv(i)
end
end
for i = 1 :4;
    l(i) = lv(i) + lh(i) + lc(i)
end
clc
   Lc(1,1)= lc(1);
      Lc(1,2)= lc(2);
         Lc(2,1)= lc(3);
            Lc(2,2)= lc(4)
Leh(1,1)= lh(1);
      Leh(1,2)= lh(2);
         Leh(2,1)= lh(3);
            Leh(2,2)= lh(4)
Lev(1,1)= lv(1);
      Lev(1,2)= lv(2);
         Lev(2,1)= lv(3);
            Lev(2,2)= lv(4)
Ld(1,1)= l(1);
      Ld(1,2)= l(2);
         Ld(2,1)= l(3);
            Ld(2,2)= l(4)
```

Appendix B

Matlab program for a channel using Logistic Distribution with variance one.

```
clc
clear
t = input('received message')
n= input(' number of iterations')
l(1)=0;
l(2)=0;
l(3)=0;
l(4)=0;
x(1) = t(1);
x(2) = t(2);
x(3) = t(3);
x(4) = t(4);
y(1,2)= t(5);
y(3,4)= t(6);
y(1,3)= t(7);
y(2,4)= t(8);
lc(1)= 2*pi/sqrt(3)+2*log((1+exp((pi/sqrt(3))*
(-x(1)-1)))/(1+exp((pi/sqrt(3))*(1-x(1)))))
lc(2)= 2*pi/sqrt(3)+2*log((1+exp((pi/sqrt(3))*
(-x(2)-1)))/(1+exp((pi/sqrt(3))*(1-x(2)))))
lc(3)= 2*pi/sqrt(3)+2*log((1+exp((pi/sqrt(3))*
(-x(3)-1)))/(1+exp((pi/sqrt(3))*(1-x(3)))))
lc(4)= 2*pi/sqrt(3)+2*log((1+exp((pi/sqrt(3))*
(-x(4)-1)))/(1+exp((pi/sqrt(3))*(1-x(4)))))
ly(1,2)=2*pi/sqrt(3)+2*log((1+exp((pi/sqrt(3))*
(-y(1,2)-1)))/(1+exp((pi/sqrt(3))*(1-y(1,2)))))
ly(3,4)= 2*pi/sqrt(3)+2*log((1+exp((pi/sqrt(3))*
(-y(3,4)-1)))/(1+exp((pi/sqrt(3))*(1-y(3,4)))))
ly(1,3)= 2*pi/sqrt(3)+2*log((1+exp((pi/sqrt(3))*
(-y(1,3)-1)))/(1+exp((pi/sqrt(3))*(1-y(1,3)))))
ly(2,4)= 2*pi/sqrt(3)+2*log((1+exp((pi/sqrt(3))*
(-y(2,4)-1)))/(1+exp((pi/sqrt(3))*(1-y(2,4)))))
for k= 1:n
for i= 1 : 4;
    if mod(i,2)==1
        lh(i)=-1*sign(lc(i+1) + l(i+1)
)*sign(ly(i,i+1))*min(abs(lc(i+1) + l(i+1)),(ly(i,i+1)))
    else
        lh(i)=-1*sign(lc(i-1) + l(i-1) )*sign(ly(i-1,i))*
min(abs(lc(i-1) + l(i-1)),(ly(i-1,i)))
```

```
    end
end
for i = 1 :4;
    l(i) = lh(i)
end
for i= 1 : 4;
    if abs(i -1.5) == .5
        lv(i)=-1*sign(lc(i+2) + l(i+2) )*sign(ly(i,i+2))*
min(abs(lc(i+2) + l(i+2)),(ly(i,i+2)))
    else
        lv(i)=-1*sign(lc(i-2) + l(i-2) )*sign(ly(i-2,i))*
min(abs(lc(i-2) + l(i-2)),(ly(i-2,i)))
    end
end
for i = 1 :4;
    l(i) = lv(i)
end
end
for i = 1 :4;
    l(i) = lv(i) + lh(i) + lc(i)
end
clc
   Lc(1,1)= lc(1);
      Lc(1,2)= lc(2);
         Lc(2,1)= lc(3);
            Lc(2,2)= lc(4)
Leh(1,1)= lh(1);
      Leh(1,2)= lh(2);
         Leh(2,1)= lh(3);
            Leh(2,2)= lh(4)
Lev(1,1)= lv(1);
      Lev(1,2)= lv(2);
         Lev(2,1)= lv(3);
            Lev(2,2)= lv(4)
Ld(1,1)= l(1);
      Ld(1,2)= l(2);
         Ld(2,1)= l(3);
            Ld(2,2)= l(4)
```

جامعة النجاح الوطنية
كلية الدراسات العليا

فك الشيفرة من النوع تيربو باستخدام البرمجة الخطية

إعداد
هشام حامد عبد الرؤوف صلاحات

إشراف
د. محمد نجيب أسعد
د. محمد عمران

قدمت هذه الأطروحة استكمالا لمتطلبات درجة الماجستير في الرياضيات بكلية الدراسات العليا في جامعة النجاح الوطنية في نابلس، فلسطين.

2013

فك الشيفرة من النوع تيربو باستخدام البرمجة الخطية

إعداد
هشام حامد عبد الرؤوف صلاحات
إشراف
د. محمد نجيب أسعد
د. محمد عمران

الملخص

نفحص في هذه الرسالة تطبيقات البرمجة الخطية الموسعة في اكتشاف وتصحيح الأخطاء في بعض أنواع الشيفرة. طريقة البرمجة الخطية الموسعة هي طريقة قياسية في خورازميات التقريب وبحوث العمليات،وتستخدم أيضا في إيجاد الحل الجيد شبه المثالي لمسائل الأفضلية الصعبة.

لقد تم استخدام طريقة الاحتمالات البعدية وخورازميات الفك العددية لفك الشيفرة الضربية (حالة خاصة من شيفرات التيربو). تم كتابة برنامج على الماتلاب يقوم بعمل حسابات للخوارزمية السابقة (خوارزمية الفك العددية)، حيث تم تطبيق البرنامج على التوزيع اللوجستي باستخدم التباين 1، وبمقارنة نتائج حساباتنا مع أبحاث آخرين كانت نتائجنا الأفضل.

طريقة البرمجة الخطية احتلت مكانها في شيفرات التيربو العامة المكونة من تعريشات مبسطة أصيلة لهذه الشيفرة، حيث تم تشكيل صيغة مشكلة البرمجة الخطية لمعالجة تعريشة واحدة للشيفرة كمسألة إيجاد الحد الأدنى لتكلفة التدفق، حيث كانت التعريشة تدفق متجه، تم توسيع هذه الصيغة لأي شيفرة من نوع تيربو بتطبيق تقيدات بين المتغيرات الموجودة في مسألة البرمجة الخطية في كل وحدة شيفرة.

إحدى الفوائد المهمة في استخدام البرمجة الخطية في فك الشيفرات أن الناتج من فك الشيفرة هو الحل الأمثل، أي أن المستلم من المعلومات هو الأكثر احتمالية أنه المرسل، هذه الخاصية تسمى خاصية التصديق الأكثر احتمالية.