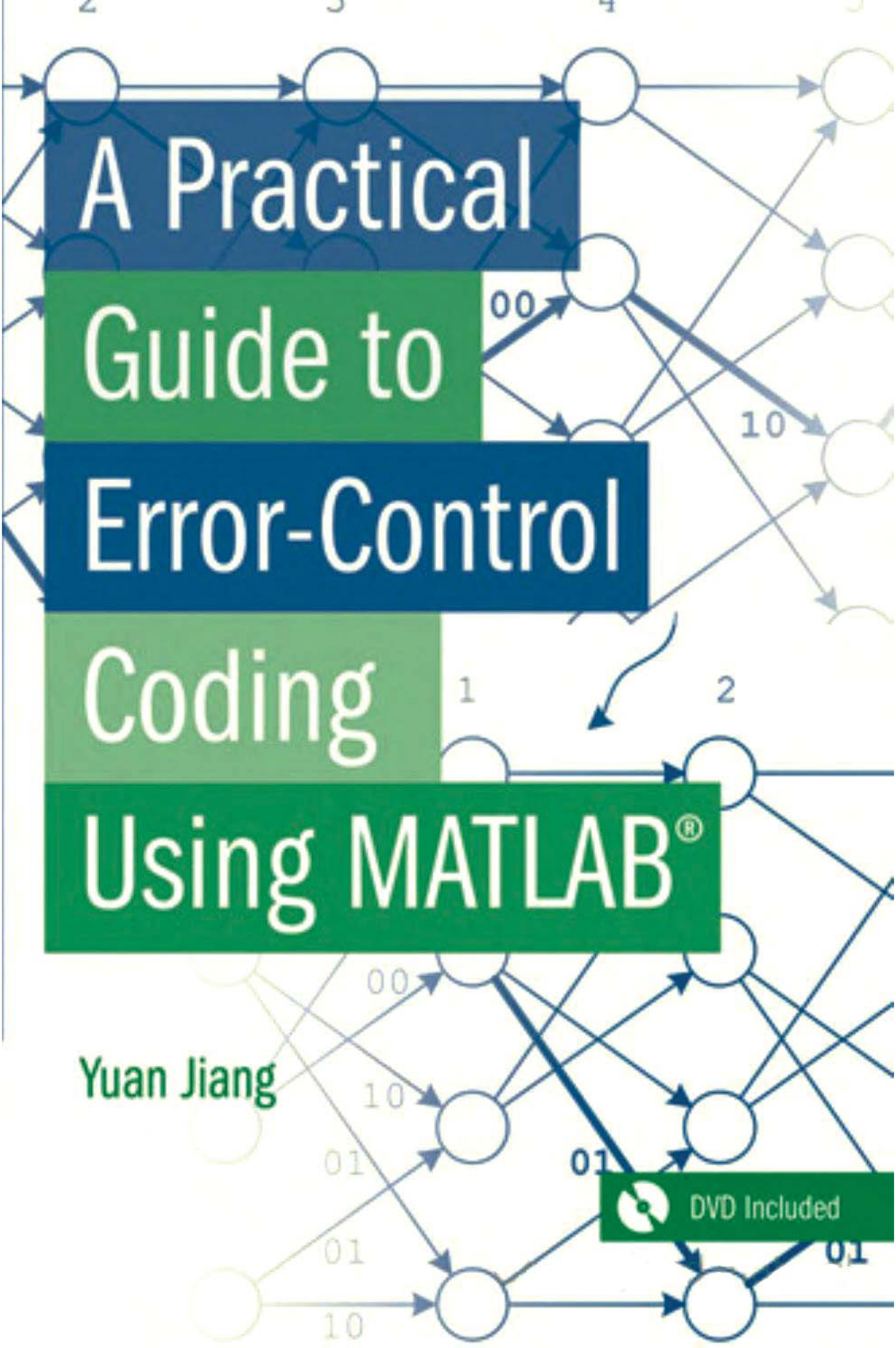
A Practical Guide to Error-Control Coding Using MATLAB®



Yuan Jiang

DVD Included

A Practical Guide to Error-Control Coding Using MATLAB®

DISCLAIMER OF WARRANTY

The technical descriptions, procedures, and computer programs in this book have been developed with the greatest of care and they have been useful to the author in a broad range of applications; however, they are provided as is, without warranty of any kind. Artech House, Inc. and the authors and editors of the book titled *A Practical Guide to Error-Control Coding Using MATLAB®* make no warranties, expressed or implied, that the equations, programs, and procedures in this book or its associated software are free of error, or are consistent with any particular standard of merchantability, or will meet your requirements for any particular application. They should not be relied upon for solving a problem whose incorrect solution could result in injury to a person or loss of property. Any use of the programs or procedures in such a manner is at the user's own risk. The editors, author, and publisher disclaim all liability for direct, incidental, or consequent damages resulting from use of the programs or procedures in this book or the associated software.

A Practical Guide to Error-Control Coding Using MATLAB®

Yuan Jiang

ARTECH
HOUSE

BOSTON | LONDON
artechhouse.com

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the U.S. Library of Congress.

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library.

ISBN 13: 978-1-60807-088-6

Cover design by Patrick McCarthy

685 Canton Street
Norwood, MA 02062

10 9 8 7 6 5 4 3 2 1

Contents

Preface *ix*

1 Error Control in Digital Communications and Storage 1

1.1 Error Control Coding at a Glance 1
1.1.1 Codes for Error Control 1
1.1.2 Important Concepts 5

1.2 Channel Capacity and Shannon's Theorem 14

1.3 Considerations When Selecting Coding Schemes 16

References 17
Selected Bibliography 17

2 Brief Introduction to Abstract Algebra 19

2.1 Elementary Algebraic Structures 19
2.1.1 Group 19
2.1.2 Field 22

2.2 Galois Field and Its Arithmetic 25
2.2.1 Galois Field 25
2.2.2 Arithmetic in *GF* (2^m) 33

2.3 Implementation of *GF* (2^m) Arithmetic 36
2.3.1 Arithmetic with Polynomial Representation 36
2.3.2 Arithmetic with Power Representation 39
2.3.3 A Special Case: Inversion 40

Problems 43
References 44
Selected Bibliography 44

3 Binary Block Codes 45

3.1 Linear Block Codes 45
3.1.1 Code Construction and Properties 45
3.1.2 Decoding Methods 51
3.1.3 Performance of Linear Block Codes 60
3.1.4 Encoder and Decoder Designs 65
3.1.5 Hamming Codes 66

3.2 Cyclic Codes 73
3.2.1 Basic Principles 74
3.2.2 Shift Register–Based Encoder and Decoder 81
3.2.3 Shortened Cyclic Codes and CRC 91

3.3 BCH Codes 95
3.3.1 Introduction 97
3.3.2 BCH Bound and Vandermonde Matrix 100
3.3.3 Decoding BCH Codes 101

Problems 110
References 111
Selected Bibliography 112

4 Reed-Solomon Codes 113

4.1 Introduction to RS Codes 113
4.1.1 Prelude: Nonbinary BCH Codes 113
4.1.2 Reed-Solomon Codes 117

4.2 Decoding of RS Codes 123
4.2.1 General Remarks 123
4.2.2 Determining the Error Location Polynomial 124
4.2.3 Frequency-Domain Decoding 135
4.2.4 Error and Erasure Decoding 140

4.3 RS Decoder: From Algorithm to Architecture 143
4.3.1 Syndrome Computation Circuit 143
4.3.2 Architectures for Berlekamp-Massey Algorithm 143
4.3.3 Circuit for Chien Search and Forney's Algorithm 149

4.4 Standardized RS Codes 149

Problems 150
References 151

5 Convolutional Codes 153

5.1 Fundamentals of Convolutional Codes 153
5.1.1 Code Generation and Representations 153
5.1.2 Additional Matters 161

5.2 Decoding of Convolutional Codes 165
5.2.1 Optimum Convolutional Decoding and Viterbi Algorithm 166
5.2.2 Sequential Decoding 179

5.3 Designing Viterbi Decoders 189
5.3.1 Typical Design Issues 189
5.3.2 Design for High Performance 197

5.4 Good Convolutional Codes 201
5.4.1 Catastrophic Error Propagation 202
5.4.2 Some Known Good Convolutional Codes 202

5.5 Punctured Convolutional Codes 202

Problems 210
References 210
Selected Bibliography 212

6 Modern Codes 213

6.1 Turbo Codes 213
6.1.1 Code Concatenation 213
6.1.2 Concatenating Codes in Parallel: Turbo Code 218
6.1.3 Iterative Decoding of Turbo Codes 228
6.1.4 Implementing MAP 250

6.2 Low-Density Parity-Check Codes 252
6.2.1 Codes with Sparse Parity-Check Matrix 254
6.2.2 Decoding and Encoding Algorithms 259
6.2.3 High-Level Architecture Design for LDPC Decoders 270

Problems 272
References 274
Selected Bibliography 276

About the Author 277

Index 279

Preface

This book attempts to provide a comprehensible and practical introduction to error control coding. The targeted readers are practicing engineers and university students who have already set foot in this territory or plan to. To achieve the goal, this book takes an approach that is somewhat different from the approaches used by the many excellent textbooks currently available.

First, the book introduces MATLAB as a tool to facilitate the presentation of key concepts. The DVD that accompanies this book provides more than 90 MATLAB programs with which readers can experiment. It is the author's hope that this fresh attempt does help readers in mastering the subject.

Second, the book pays attention to the implementation of various decoding algorithms. Readers will find that a few practical issues have received in-depth treatment in the book, such as implementation of Galois field arithmetic, Viterbi decoder design, RS decoder design, and MAP architecture, to name a few.

The organization of the book is standard. Readers may notice, however, that many mathematical proofs and theorems have been omitted. This is because this book emphasizes concepts and rationales. For those who wish to explore further, a comprehensive list of references is given at the end of each chapter. Note that the MATLAB functions marked with asterisks are provided by the book not by the MATLAB software.

It is left to the readers to determine whether the book has served its purpose. The author welcomes feedback of any kind (ecc.book.comments@hotmail.com).

Finally the author would like to express his gratitude to editors Mark Walsh, Lindsey Gendall, and Rebecca Allendorf at Artech House. Without their appreciation and help, publication of this book would have been a lot harder. The author is also indebted to the book reviewer, who remains anonymous to the author, for his valuable comments and suggestions, which enlightened the author a great deal.

1

Error Control in Digital Communications and Storage

The goal of this introductory chapter is to sketch out an overall picture of error control coding for digital communications and storage, so that, after completing the chapter, readers will have a rough idea of what the subject is all about. The emphasis in this chapter is on the concepts and the rationale.

1.1 Error Control Coding at a Glance

1.1.1 Codes for Error Control

1.1.1.1 The Rationale

Digital communications and storage have become part of our daily lives. Robust data transmission and data storage are taken for granted. People hardly realize that errors occur from time to time in data transmission/storage systems, and if it were not for the use of error control techniques, reliable data transmission/storage would be impossible.

Errors in data transmission/storage systems can come from many different sources: random noise, interference, channel fading, or physical defects, just to name a few. These channel errors must be reduced to an acceptable level to ensure the quality of data transmission/storage. To combat the errors, we normally use two strategies, either stand-alone or combined. The first one is the automatic repeat request (ARQ). An ARQ system attempts to detect the presence of errors in the received data. If any errors are found, the receiver notifies the transmitter of the existence of errors. The transmitter then resends the data until they are correctly received.

The second strategy, known as the forward error correction (FEC), not only detects but also corrects the errors, so that data retransmission can be avoided. In many practical applications retransmission may be difficult or not even feasible at all. For example, it is impossible for any receiver in a real-time broadcasting system to request data to be resent. In this case, FEC is the only viable solution.

Either way, error control codes (ECC) are used for detecting the presence of errors and correcting them. To intuitively explain the mechanism of ECC, let us look at a simple example from our daily lives. You and your friend are going for a walk. Before you leave, you recall that rain has been forecast. So you say to your friend, “We should carry an umbrella with us.” Your friend may hear it as “We should carry a banana with us” and gets confused. However, if you instead say, “We should carry an umbrella with us; it’s going to rain,” your friend will know what you said is umbrella not banana, based on the context of your second sentence. Your second sentence in this case is redundancy that facilitates detection and correction of the error. ECC does exactly the same thing. It first adds redundancy to the message to be sent; this process is called *encoding* and is carried out at the transmitter. It then corrects errors based on the redundancy in a process called *decoding* that is performed at the receiver. The output of the encoding process is a codeword that contains both the message and the redundancy (explicitly or implicitly). The redundancy is referred to as the parity check, or simply the parity. Figure 1.1 shows a typical communications system equipped with error control functionality.

Example 1.1

We send a message bit of 1 to the receiver. Due to the channel error, when the bit passes the channel and arrives at the receiver it becomes a 0. Unfortunately there is no indication whatsoever whether the received bit is correct or not.

Now, instead of sending the raw message bit, we send a codeword $\mathbf{c}$ *formed by repeating the message bit three times. The codeword corresponding to a message bit of 0 is* $\mathbf{c}_0 = (000)$*, and the codeword for a message bit of 1 is* $\mathbf{c}_1 = (111)$*. The redundancy here is the two duplicates of the message bit.*

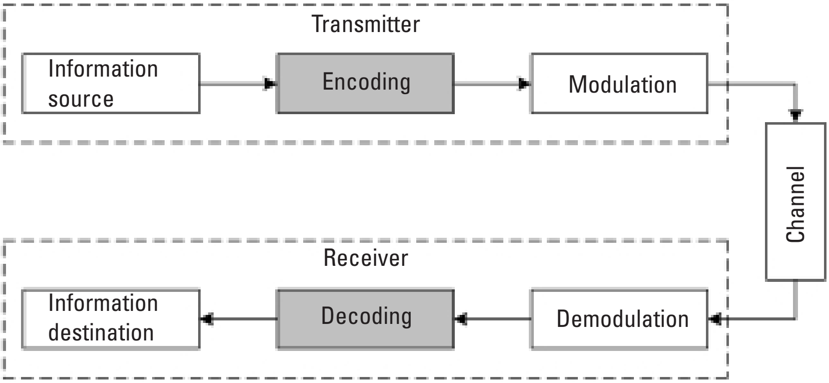


Figure 1.1 A typical communications system with ECC.

> *Suppose that the received word is* $\mathbf{r} = (011)$, *which has an error in its first position. We immediately know that* **r** *is in error, because all three bits are supposed to be identical but they are not.*
>
> *Notice that* **r** *differs from* $\mathbf{c}_0$ *by two bits and differs from* $\mathbf{c}_1$ *by one bit. It is logical to think that the received word is more likely to be* **r** *if* $\mathbf{c}_1$ *is sent. So we can quite confidently conclude that the codeword transmitted is* $\mathbf{c}_1 = (111)$ *and the original message is 1. The two redundant bits have helped us make correct decoding.*
>
> *This trivial repetition code provides both error detection and error correction capability.*

Figure 1.2 illustrates a typical bit error rate (BER) versus signal-to-noise ratio (SNR) curve for coded and uncoded systems.

The use of error correction, however, is not free. The redundancy acts as overhead and it "costs" transmission resources (e.g., channel bandwidth or transmission power). Therefore, we want the redundancy to be as small as possible. To give the redundancy a quantitative measure, the coding rate R is defined as the ratio of the message length to the codeword length. For example, if a coding scheme generates a codeword of length n from a message of length k, the coding rate is:

$$R = {}^{k}\!/_{n} \tag{1.1}$$

The maximum value of the coding rate is 1 when no redundancy is added (i.e., when the message is uncoded). Coding performance and coding rate are two opposing factors. As more redundancy is added, the error

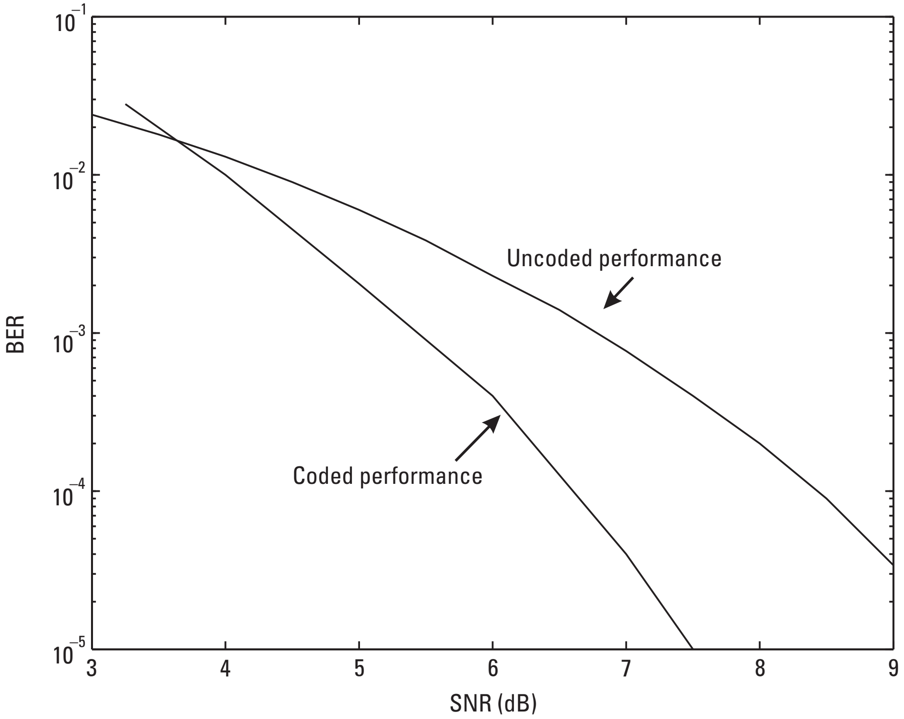


Figure 1.2 Typical BER performance of coded system.

correction capability is strengthened, but the coding rate drops. A good code should maximize the error correction performance while keeping the coding rate close to 1.

1.1.1.2 A Second Look at ECC

From the preceding introduction, it seems that we are definitely better off using error control coding because errors indeed get corrected (as seen in the example). A closer examination of the coding principle, however, shows that this may not be always true, despite the error correction capability provided. As we have just said, the redundancy costs resources. To see if coding is really beneficial, we need to compare coded systems and uncoded systems under the condition of equal resource usage.

Now, say, we use 1 watt of power to transmit the raw message bit in the preceding example. With coding, the transmit power of each bit in the codeword is reduced to 1/3 watt (the total power is kept to 1 watt). Consequently, the probability of errors will increase. We see, on one hand, that coding corrects channel errors and brings down the error probability; on the other hand, reduced power per bit causes the error probability to go higher. So we will be better off only if the coding increases the performance enough to make up for the signal power reduction caused by the redundancy and produces a net gain. Let us reexamine Figure 1.2. We observe that the BER performance of the coded system is actually worse than that of the coded system in the low SNR range (≤3.5 dB in the figure). This is because the coding in that SNR range is not able to offer enough performance improvement to cover the signal power loss due to the redundancy.

As a conclusion, codes must be designed to offer a net performance gain.

1.1.2 Important Concepts

1.1.2.1 Types of Codes

Depending on how redundancy is added, there are two families of codes. One is called the block codes. Block coding encodes and decodes data on a block-by-block basis. Data blocks in this case are independent from each other. Consequently block coding is a memoryless operation and can be implemented using combinational logic. The code in Example 1.1 is a block code because the coding is completely determined by the current data block. In contrast, another family of codes, namely, the convolutional codes, works on a continuous data stream, and its encoding and decoding operations depend not only on the current data but also on the previous data. As such, convolutional coding contains memory and has to be implemented using sequential logic.

1.1.2.2 Systematic Versus Nonsystematic Codes

A complete codeword comprises the message and the redundancy. If the redundancy is implicitly embedded in the codeword, the code is said to be nonsystematic. On the other hand, if the redundancy is explicitly appended to the message, the code is systematic (see Figure 1.3). Systematic codes are always preferred in practice, because the message and the parity are separated so the receiver can directly extract the message from the decoded codeword.

1.1.2.3 Digital Modulation

After error control encoding, we have a sequence of coded digital symbols, which is to be converted to an analog signal (called the carrier) before it can be transmitted over a physical channel (copper wire, optical fiber, or air).

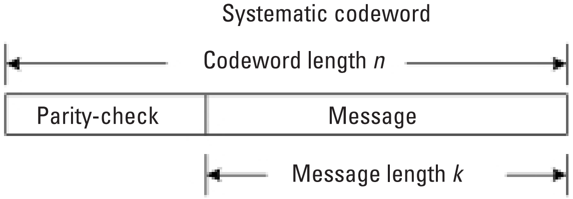


Figure 1.3 Structure of systematic code.

This job is done by modulation. According to the symbols, the modulation process instantaneously alters the amplitude, phase, frequency (or a combination thereof) of the carrier to convey the information to be transmitted. A modulation scheme of particular interest is binary phase shift keying (BPSK) for binary symbols (i.e., bits). BPSK assigns to the carrier 0 phase shift when the bit is a 0, and π phase shift when the bit is a 1. From a baseband point of view, the BPSK modulation is a mapping process: 0 → 1 and 1 → –1. Figure 1.4 shows the waveform of a BPSK modulated signal. At the receiver, the received signal is demodulated back to digital symbols.

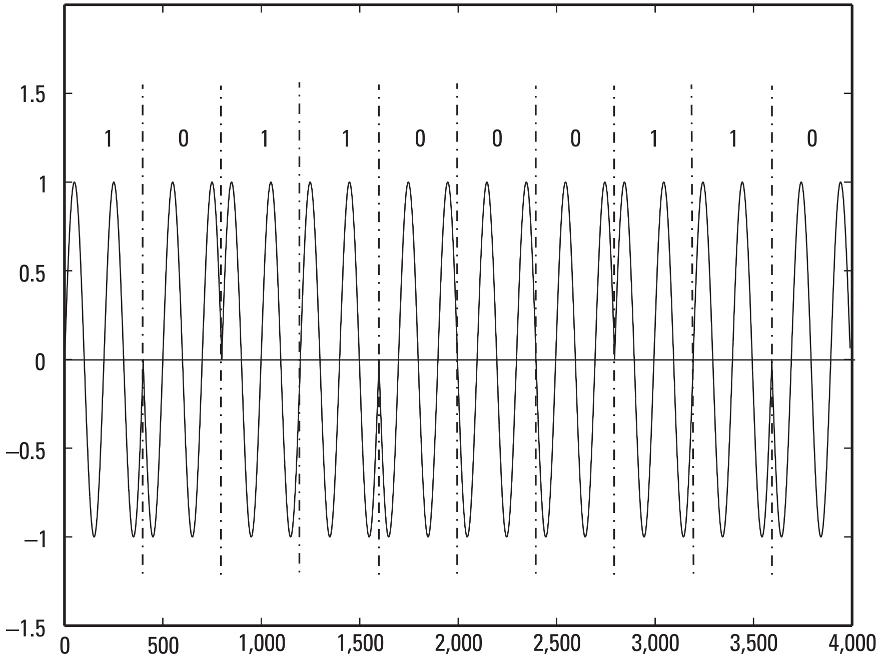


Figure 1.4 BPSK modulated signal.

Extending BPSK to the nonbinary case, let us say that the symbol consists of two bits. Then the symbol has four possible combinations: (00), (01), (11), and (10). Assigning to the carrier four corresponding phase shifts $\pi/4$, $3\pi/4$, $5\pi/4$, and $7\pi/4$, we form so-called quadrature phase-shift keying (QPSK). QPSK maps the symbol as $(00) \rightarrow 1 + j$, $(01) \rightarrow -1 + j$, $(11) \rightarrow -1 - j$, $(10) \rightarrow 1 - j$. The signal space constellations of BPSK and QPSK are depicted in Figure 1.5.

MATLAB Experiment 1.1

The Communications Toolbox in MATLAB provides a pair of functions, `modmap` and `demodmap`, to map a digital signal to and from an analog signal for a given modulation scheme, respectively. Typing in `modmap(‘psk’,2)` and `modmap(‘psk’,4)` generate the BPSK and QPSK constellations, as shown in Figure 1.5.

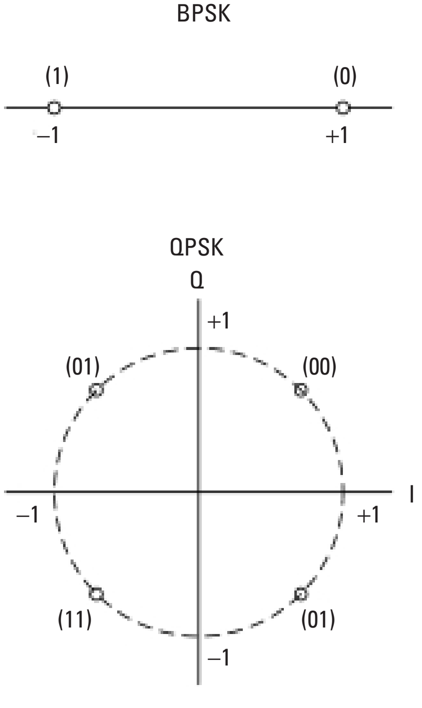


Figure 1.5 Signal constellation of BPSK and QPSK.

1.1.2.4 Channel and Channel Errors

The most common transmission errors are caused by additive white Gaussian noise (AWGN). Because this type of noise is totally random, the AWGN errors are independent from each other; that is, they are memoryless. Such a transmission channel is called an AWGN channel. Most error control codes tackle memoryless errors.

However, in some scenarios channel errors occur in bursts. The bursty channel involves memory, therefore the errors are correlated. Wireless fading channels and defects on the surface of a compact disc are two examples of the channel.

Classical coding theory often views the modulation, channel, and demodulation in Figure 1.2 as being combined as a discrete composite channel. The input to the composite channel consists of binary bits. If the demodulation also outputs binary bits, we may neglect all details inside the composite channel and simply model it as a binary symmetric channel (BSC) (see Figure 1.6) characterized by the crossover probability p_x. The crossover probability is defined as the probability of a bit error. For the particular case of AWGN "internal" channel and BPSK signaling, p_x can be computed as follows:

$$p_x = Q\left(\sqrt{{}^{2E_b}\!/_{N_0}}\right) \tag{1.2}$$

where $Q(x) \triangleq \left({}^{1}\!/_{\sqrt{2\pi}}\right) \cdot \int_x^{\infty} e^{\ y^2/2} dy$ is called the Q-function and E_b/N_0 is the bit SNR.[1] Like an AWGN channel, the BSC is also memoryless.

MATLAB Experiment 1.2

The AWGN channel is modeled in MATLAB by `awgn`. The function adds AWGN noise to transmitted data at a specified SNR.

The companion DVD provides a BSC model `bsc`*, which introduces random bit errors to a binary sequence based on a crossover probability.

[1] E_b denotes the bit energy, and N_0 denotes the AWGN power spectral density.

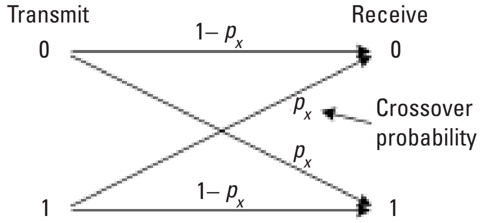


Figure 1.6 Binary symmetric channel.

MATLAB Experiment 1.3

The DVD also includes a simple script `qfunc*` to compute the Q-function. To calculate the crossover probability at $E_b/N_0 = 0$ dB, we type in the following command:

```
>> eb_n0 = 0;                     % dB
>> eb_n0 = 10^(eb_n0/10);         % convert to linear scale
>> px = qfunc(sqrt(2*eb_n0))      % crossover prob.
px =
       0.0786
```

1.1.2.5 Optimal Decoding Principles

Recall that in the previous example, we take $\boldsymbol{c}_1$ as the decoded output, because the received vector is more likely to be $\boldsymbol{r}$ if $\boldsymbol{c}_1$ is sent. Decoding based on this principle is called maximum-likelihood decoding, or simply ML decoding. Mathematically ML decoding can be expressed as follows[2]:

$$\tilde{\boldsymbol{c}} = \begin{cases} \boldsymbol{c}_0, & \text{if } P(\boldsymbol{r} \mid \boldsymbol{c}_0) \geq P(\boldsymbol{r} \mid \boldsymbol{c}_1) \\ \boldsymbol{c}_1, & \text{if } P(\boldsymbol{r} \mid \boldsymbol{c}_0) < P(\boldsymbol{r} \mid \boldsymbol{c}_1) \end{cases} \tag{1.3}$$

where $\tilde{\boldsymbol{c}}$ denotes the decoded word and $P(\boldsymbol{r}|\boldsymbol{c}_0)$ [or $P(\boldsymbol{r}|\boldsymbol{c}_1)$] is the probability that the word $\boldsymbol{r}$ is received given the condition that the codeword

[2] For the sake of simplicity, we assume two codewords in total. The principle remains the same for cases with more codewords.

$\boldsymbol{c}_0$ (or $\boldsymbol{c}_1$) is transmitted. ML decoding is optimal if all codewords (or message symbols, since they have one-to-one correspondence) are equally likely.

When the occurrences of the codewords are not equally probable, so-called maximum a posteriori decoding, or MAP decoding, comes into play. In contrast to ML, MAP selects as the decoded output the codeword that maximizes the a posteriori probability (APP):

$$\tilde{\boldsymbol{c}} = \begin{cases} \boldsymbol{c}_0, & \text{if } P(\boldsymbol{c}_0 \,|\, \boldsymbol{r}) \geq P(\boldsymbol{c}_1 \,|\, \boldsymbol{r}) \\ \boldsymbol{c}_1, & \text{if } P(\boldsymbol{c}_0 \,|\, \boldsymbol{r}) < P(\boldsymbol{c}_1 \,|\, \boldsymbol{r}) \end{cases} \tag{1.4}$$

where $P(\boldsymbol{c}_0|\boldsymbol{r})$ [or $P(\boldsymbol{c}_1|\boldsymbol{r})$] is the probability that $\boldsymbol{c}_0$ (or $\boldsymbol{c}_1$) is transmitted given the condition that the vector $\boldsymbol{r}$ is received.

In the previous example, we implied that the codewords $\boldsymbol{c}_0$ and $\boldsymbol{c}_1$ are equally likely to occur. If, say, $\boldsymbol{c}_0$ has an 80% chance to be sent, and $\boldsymbol{c}_1$ has the remaining 20%, then we need to use MAP to achieve optimal decoding.

When all message symbols are equally probable, ML and MAP decoding techniques are equivalent.

1.1.2.6 Hard-Decision Decoding and Soft-Decision Decoding

With BSC, the input to the decoder is a binary sequence. Decoding based on "hard" binary bits is referred to as hard-decision decoding. In contrast, if the demodulation process uses a multilevel "soft" value to represent an output bit, the decoding then works on (quantized) real values. This type of decoding is called soft-decision decoding. Hard-decision decoding can be viewed as a special case of soft decoding in which single-bit quantization is used. We can imagine that soft-decision decoding performs better because it has more information to exploit.

1.1.2.7 Minimum Hamming Distance and Error Correction Capability

In Example 1.1, we chose as decoded output the codeword to which the received word is "closest" in distance. The distance was measured by counting the number of differing bits in two words. This distance, denoted by d_H and referred to as the Hamming distance, is frequently used in coding theory. Associated with the Hamming distance is the so-called Hamming weight w_H, which is defined as the Hamming distance between a nonzero codeword and the all-zero codeword. For a binary word, the Hamming weight is simply the number of 1s in the word.

Let $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ be any two codewords of a linear code C. The Hamming distance and the Hamming weight have the following relation:

$$d_H(\boldsymbol{c}_1, \boldsymbol{c}_2) = w_H(\boldsymbol{c}_1 + \boldsymbol{c}_2) \tag{1.5}$$

It is now clear that the decoding performed in the previous example is actually a process to choose a codeword $\tilde{\boldsymbol{c}}$ that satisfies the following:

$$\tilde{\boldsymbol{c}} = \arg\min_{\boldsymbol{c} \in C} d_H(\boldsymbol{r}, \boldsymbol{c}) \tag{1.6}$$

For a BSC channel, (1.6) represents ML decoding [1].

The smallest Hamming distance $d_{\min}$ between any two different codewords $\boldsymbol{c}_i$ and $\boldsymbol{c}_j$ in a code C is called the minimum Hamming distance of the code, that is:

$$d_{\min} = \min d_H(\boldsymbol{c}_i, \boldsymbol{c}_j) \tag{1.7}$$

where $d_{\min}$ reflects the error correction capability of a code. To explain this, let us assume that C is a code with a total of eight codewords $\boldsymbol{c}_0$, $\boldsymbol{c}_1$, ... , $\boldsymbol{c}_7$, which are graphically represented as eight points in Figure 1.7. Without loss of generality, we also assume $d_H(\boldsymbol{c}_1, \boldsymbol{c}_3) = d_{\min}$. Now we draw a circle around each codeword point with the same radius and no overlap with the others. Evidently the maximum such radius is $t = \lfloor (d_{\min} - 1)/2 \rfloor$, where $\lfloor x \rfloor$

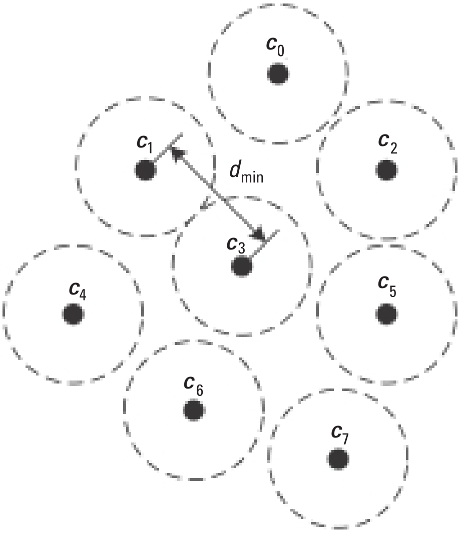


Figure 1.7 Graphical representation of decoding sphere.

denotes the greatest integer no greater than x. These circles are called the *Hamming sphere* (or the decoding sphere) of their corresponding codewords. Now suppose that we send $\boldsymbol{c}_3$ to the receiver. If no channel errors exist, the received word $\boldsymbol{r}$ will coincide with the codeword point $\boldsymbol{c}_3$ in the figure. Otherwise the channel errors will move $\boldsymbol{r}$ away from where $\boldsymbol{c}_3$ is. If in this case $\boldsymbol{r}$ falls within the Hamming sphere of $\boldsymbol{c}_3$, $\boldsymbol{r}$ will still be correctly decoded to $\boldsymbol{c}_3$, simply because it is closer to $\boldsymbol{c}_3$ than any other codeword point (the ML decoding criterion). If $\boldsymbol{r}$ falls out of the sphere, then it will be mistakenly decoded to some other codeword. From this we actually can draw a general conclusion:

Correct decoding is guaranteed if and only if the received word falls within the Hamming sphere of the true codeword.

In other words, the maximum error correction capability of a code quals to the radius of the Hamming sphere t. Therefore,

$$t = \lfloor (d_{\min} - 1)/2 \rfloor \tag{1.8}$$

is the random error correction capability of the code.

Example 1.2

The repetition code in the previous example contains only two codewords, $\boldsymbol{c}_0 = (000)$ and $\boldsymbol{c}_1 = (111)$. Therefore, the Hamming distance between the two codewords is the minimum Hamming distance, which is computed to be:

$$d_{\min} = d_H(\boldsymbol{c}_0, \boldsymbol{c}_1) = 1 + 1 + 1 = 3$$

Based on (1.8), we see that the code is able to correct, at most, one random error. Example 1.1 confirms that it does correct one error. It is also easy to verify that the code cannot correct two or more errors. For instance, if $\boldsymbol{c}_1$ is transmitted and $\boldsymbol{r} = (001)$ (containing two errors) is received, $\boldsymbol{r}$ will be incorrectly decoded to $\boldsymbol{c}_0$.

1.1.2.8 Performance Measures

The most direct measure of performance of an error correcting system is the error rate, defined as the number of errors that the decoder fails to correct divided by the length of the transmitted sequence, at a specified SNR.

However, in many practical applications, exact computation of the error rate is difficult. It is more convenient to use the union bound instead. The union bound is an upper bound and it is computed based on the following observation: If an event is the union of n subevents E_1, E_2, $\cdots$, E_n, then the probability of the event occurring, $P(E_1 \cup E_2 \cup \cdots \cup E_n)$, is at most the sum of the probabilities of all the subevents, $P(E_i)$ $(1 \le i \le n)$, that is:

$$P\left(E_1 \cup E_2 \cup \cdots \cup E_n\right) \le P(E_1) + P(E_2) + \cdots + P(E_n) \tag{1.9}$$

where the equality holds when the subevents are mutually exclusive.

Alternatively an error control system may also be evaluated with coding gain. Coding gain measures the difference in the SNR levels between the coded system and uncoded system at a specified error rate. Go back to Figure 1.2; the difference in E_b/N_0 between the intersections of the two BER curves and the horizontal line of 10^{-4} is the coding gain of the code at the error rate of 10^{-4}. While the coding gain must be evaluated for each individual code of interest, the asymptotic coding gain, an approximation to the coding gain when SNR >> 1, offers a simple and quick measure of the coding performance. It has been shown that, for hard-decision decoding, a code with a rate R and a minimum distance $d_{\min}$ has an asymptotic coding gain of [1, 2]:

$$K = 10 \cdot \log\left({}^{Rd_{\min}}\!\big/_{2}\right) \tag{1.10}$$

For soft-decision decoding, the asymptotic coding gain becomes:

$$K = 10 \cdot \log\left(Rd_{\min}\right) \tag{1.11}$$

which is 3 dB better.

MATLAB Experiment 1.4

The MATLAB function `cgain*` estimates the coding gain given the error probability of the code. Let us find the coding gain at a BER of 10^{-4} for the code in Figure 1.2.

```
>> % bit snr in dB
>> eb_n0 = [3.25 4 5 6 7 7.5];
>> % bit error probability of the code corresponding to eb_n0
>> ber = [2.8e-2 1e-2 2.05e-3 4e-4 4e-5 1e-5];
>> cgain(eb_n0,ber,10^(-4))
ans =
       1.7112
```

Comment: the result is in dB.

1.2 Channel Capacity and Shannon's Theorem

Now we have some idea about the benefits that channel coding can provide. It will be interesting to see the theoretic limit of the coding performance, that is, how small the probability of error can go with a code of rate R. The question was answered by Shannon in his landmark paper published in 1948 [3]. In that paper Shannon proved the following channel coding theorem:

By employing a sufficiently long error correction code, an arbitrarily small probability of error can be achieved if the coded bits are transmitted at a rate less than the channel capacity.

The channel capacity C, defined as the maximum number of bits per unit time that can be transmitted free of error over a channel, is given by the Shannon formula:

$$C = B \cdot \log_2 \left(1 + {}^{S}\!/_{N}\right) \tag{1.12}$$

where B is the channel bandwidth, $S/N = E_b R_T / N_0 B$ is the average SNR expressed in linear scale, and R_T denotes the bit transmission rate. The value of C is given in bits per second. Shannon's channel coding theorem basically states that no such error correction codes exist to guarantee free-of-error transmission if the information bits are transmitted at the rate $R_T > C$; such codes do, however, exist if $R_T \leq C$.

MATLAB Experiment 1.5

The MATLAB script `chcap.m`* plots the relation between the normalized channel capacity C/B and the bit SNR shown in Figure 1.8.

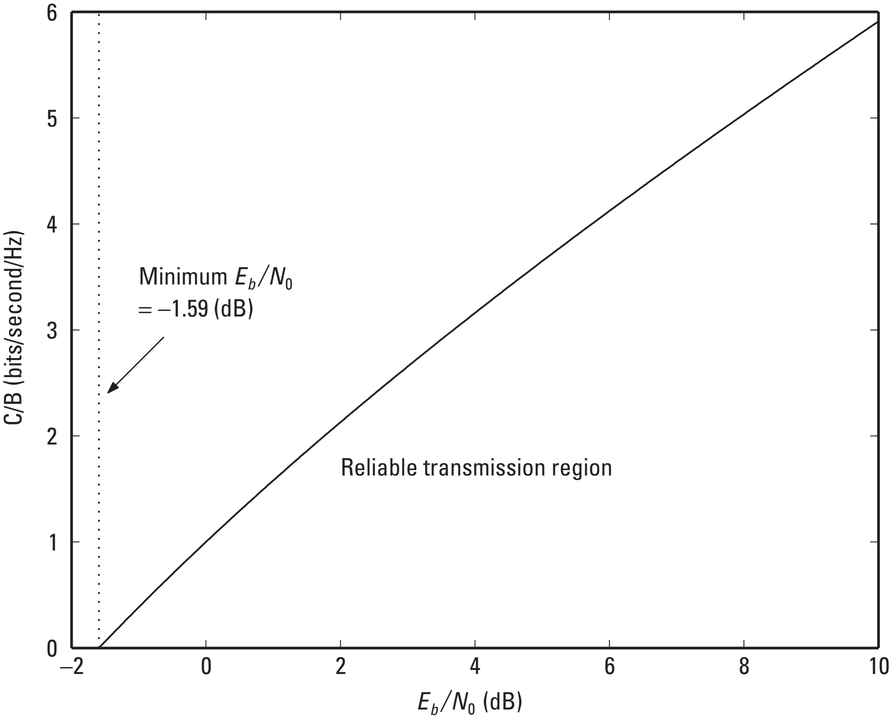


Figure 1.8 Normalized channel capacity.

Figure 1.8 is revealing. Look at the point $E_b/N_0 = -1.59$ dB reached when $C/B = 0$ (i.e., $B \rightarrow \infty$). This is the minimum E_b/N_0 required to achieve reliable communications. For anything above this, as long as we design the system to operate in the reliable region, we are able to obtain error-free transmission (provided that the error control code is sufficiently long).

Shannon's theorem has been keeping coding theorists busy for more than 60 years, because it does not give the actual codes. Coding specialists have seriously attempted to construct codes that are both effective in correcting errors and low in decoding complexity. Their work is rewarding. It has been reported that low-density parity check (LDPC) codes can get to the Shannon channel capacity as close as 0.0045 dB [4].

1.3 Considerations When Selecting Coding Schemes

As practicing engineers, we probably will not need to design codes. We do, however, often need to choose codes—and selection of a coding scheme appropriate for a particular application is also not an easy task. It needs to take into account many factors: error correction capability, decoding complexity, error types, channel bandwidth constraints, signal power constraints and processing latency, and so forth. No single coding scheme works for all applications.

Shannon's theorem tells us that the longer the code, the better the error correcting performance. On the other hand, longer code means a higher decoding complexity and larger processing latency. Decoding dominates the overall computational cost of an error control system. Also, in real-time applications large amounts of latency are not preferred nor tolerated. Therefore, we have a trade-off to make. The ensemble average bound on uncorrected error probability can give us a rough idea of how the performance and the complexity are related. For block codes of length n and coding rate R, the bound P_E is:

$$P_E \leq 2^{-nE_B(R)} \tag{1.13}$$

For convolutional codes with memory length M, it becomes:

$$P_E \leq 2^{-(M+1)nE_C(R)} \tag{1.14}$$

where $E_B(R)$ and $E_C(R)$ are two positive functions of R and are completely determined by the channel characteristics [5].

The type of errors encountered during data transmission is another important consideration. As we already know, there are random errors and busty errors. Random errors affect the data independently. Busty errors are contiguous. An error control code must match the error type in order to be effective. Most codes are designed to combat random errors; only a few codes such as Reed-Solomon codes are good at correcting bursty errors.

With redundancy added, the coding rate $R = k/n$ becomes less than 1 and the effective data rate is reduced. To maintain the same data rate, we need to raise the overall throughput. An increase in throughput translates into more channel bandwidth. Although in deep-space, satellite, and some other wideband communications this required increase is not an issue, it becomes undesirable or even totally impossible in bandwidth-limited applications (e.g., voiceband modem). In this case coding should be used together with multilevel modulation such as m-ary PSK or QAM. Proper combination of coding and modulation can achieve error correction without the need for bandwidth expansion [6].

Note We have just looked at error control coding as the "woods." Now we need to take a look at the "trees." Starting with the next chapter, we will discuss every detail of popular error control codes.

References

[1] Lin, S., and D. J. Castello, *Error Control Coding—Fundamentals and Application*, Upper Saddle River, NJ: Prentice-Hall, 2004.

[2] Clark, G., and J. Cain, *Error-Correcting Codes for Digital Communications*, New York: Plenum Press, 1981.

[3] Shannon, C. E., "A Mathematical Theory of Communication," *Bell Syst. Tech. J.*, July 1948, pp. 379–423 (part 1) and pp. 623–656 (part 2).

[4] Chung, S. Y., et al., "On the Design of Low Density Parity Check Codes Within 0.0045 dB of the Shannon Limit," *IEEE Commun. Lett.*, Vol. 5, No. 2, February 2001, pp. 58–60.

[5] Gallager, R. G., *Information Theory and Reliable Communication*, New York: John Wiley, 1968.

[6] Massey, J. L., "Coding and Modulation in Digital Communication," *Proc. Intl. Zurich Seminar Digital Commun.*, Zurich, Switzerland, March 1974.

Selected Bibliography

Proakis, J. G., *Digital Communications*, 4th ed., New York: McGraw-Hill, 2001.

Wozencraft, J. M., and I. M. Jacob, *Principle of Communications Engineering*, New York: John Wiley & Sons, 1965.

2

Brief Introduction to Abstract Algebra

Starting with the next chapter, individual error control code will be introduced, and we will find that many popular codes are based on abstract algebra. Therefore, this chapter provides readers with basic algebra knowledge to facilitate the study of the codes. The focus of this chapter is not on mathematical rigor; rather it is on comprehension of concepts. Implementation of algebraic operations will also be given attention. The coverage will just be enough to understand the materials presented in this book. For more advanced algebraic theory, the readers may refer to many excellent textbooks, such as [1–3].

2.1 Elementary Algebraic Structures

2.1.1 Group

A *group* is an elementary structure in abstract algebra. Let G be a set on which an algebraic operation * is defined. G is a group if all of the following conditions are met:

1. G is closed under the operation *; that is, for any $a, b \in G$, $a * b \in G$.

2. The operation $*$ is associative; that is, for any $a, b, c \in G$, $(a * b) * c = a * (b * c)$.
3. There exists an element $e \in G$ such that $a * e = e * a = a$ for all $a \in G$, where e is called the *identity element* of G.
4. For every $a \in G$, there exists an element $a^{-1} \in G$ such that $a * a^{-1} = e$, where a^{-1} is called the *inverse* of a.

The group is denoted by $\langle G, *\rangle$. The identity of a group is unique, and so is the inverse of an element in a group.

The group $\langle G, *\rangle$ is also *commutative* if the following condition is satisfied:

$$a * b = b * a \text{ for every } a, b, \in G$$

Example 2.1

Let $G \in \{1, 0\}$ *and the operation* $\oplus$ *be defined as:*

⊕	0	1
0	0	1
1	1	0

The operation $\oplus$ *is called the modulo-2 addition (or the exclusive-or operation, XOR).* $\langle G, \oplus\rangle$ *forms a group. From the preceding table we can easily verify that* $\oplus$ *is associative. The element 0 is the identity element because* $0 \oplus 1 = 1 \oplus 0 = 1$, *and* $0 \oplus 0 = 0$. *The inverse of 1 is* $1 (1 \oplus 1 = 0)$, *and the inverse of 0 is 0* $(0 \oplus 0 = 0)$. *G is closed under* $\oplus$. *Moreover,* $\langle G, \oplus\rangle$ *is communicative.*

Example 2.2

Let $G \in \{1, 0\}$ *and the operation* $\otimes$ *be defined as:*

⊗	0	1
0	0	0
1	0	1

The operation $\otimes$ *is called the modulo-2 multiplication (or the AND operation).* $\langle G, \otimes\rangle$ *also forms a group. It is not difficult to find that* $\otimes$ *is associative, and 1 is the identity element of the group. The inverse of 0 is 1, the inverse of 1 is 0. G is closed under* $\otimes$. $\langle G, \otimes\rangle$ *is communicative, too.*

The number of elements in a group is the order of the group. A group of finite order is called the finite group. Obviously $\langle G, \oplus\rangle$ and $\langle G, \otimes\rangle$ are both finite groups.

Let G' be the subset of G. $\langle G', *\rangle$ is a subgroup of $\langle G, *\rangle$ if and only if $\langle G', *\rangle$ is itself a group. The following theory, Lagrange's theorem, applies to a group and its subgroup [4]:

If $\langle G, *\rangle$ is a finite group of order n, and $\langle G', *\rangle$ is $\langle G, *\rangle$'s subgroup of order m, then m divides n (in other words, n/m has no remainder).

A group $\langle G, *\rangle$ is called a cyclic group if each of its elements equals to some powers of an element α in the same group, where the ith power of α, α^i, is defined as:

$$\alpha^i \triangleq \underbrace{\alpha * \alpha * \cdots * \alpha}_{i\text{ times}}$$

where α is called the generating element of the group.

Example 2.3

Let $G \in \{1, 2, 3, 4\}$. Then $\langle G, \times \text{ mod-5}\rangle$ is a cyclic group, where the operation × mod-5 is defined as:

× mod-5	1	2	3	4
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

We then have the following:

$$2^1 = 2 \text{ mod-5} = 2$$

$$2^2 = 2 \times 2 \text{ mod-5} = 4$$

$$2^3 = 2 \times 2 \times 2 \text{ mod-5} = 3$$

$$2^4 = 2 \times 2 \times 2 \times 2 \text{ mod-5} = 1$$

So, 2 is a generating element. The readers may want to verify that 3 is also a generating element. The generating element is not unique.

Let $a \in G$. The set $a * G' \triangleq \{a * b\}$ for $b \in G'$ is called a left coset of G', and the set $G' * a \triangleq \{b * a\}$ for $b \in G'$ is called a right coset of G'. Cosets can be thought of as the elements of G' shifted by a fixed element a.

2.1.2 Field

Now we introduce another important algebraic structure known as the field. A field is essentially a set of elements in which operations such as addition, subtraction, multiplication and division can be performed without the need to leave the set, *and* the addition and multiplication are associative, communicative, and distributive. Formally a field is defined as follows.

Let F be a set of elements on which two algebraic operations are defined, one is the addition + and the other the multiplication $\cdot$. The set F together with + and $\cdot$ are called a *field* if and only if the following conditions are satisfied:

1. F forms a communicative group under addition +.
2. The set of nonzero elements in F forms a communicative group under multiplication.
3. Multiplication $\cdot$ is distributive over addition +; that is, $a \cdot (b + c) = a \cdot b + a \cdot c$, where a, b, and c are any three elements in F.

The identity element with respect to addition, denoted by 0, is called the zero element, and the identity element with respect to multiplication, denoted by 1, is called the unit element. The total number of elements in a field is referred to as the order of the field.

Example 2.4

> *The set {0, 1} with the modulo-2 addition $\oplus$ and the modulo-2 multiplication $\otimes$ defined in the previous two examples forms the binary field. Based on the tables in the two examples, we can easily verify that all three conditions of a field are satisfied.*

Notice that the binary field has a finite number of elements (that is, 0 and 1). This type of field is extensively used in coding theory. We will have more to say about it later in the chapter.

2.1.2.1 Vector Space

Let $\langle V, +\rangle$ of vectors be a commutative group and F be a field of scalars. A multiplication operation $\cdot$ is defined between the scalars of F and the vectors of V. In this case V becomes a *vector space* over the field F if the following four conditions are met:

1. For any $\alpha \in F$ and any $\boldsymbol{v} \in V$, $\alpha \cdot \boldsymbol{v} \in V$.
2. For any $\boldsymbol{u}, \boldsymbol{v} \in V$ and any $a, b \in F$, $a \cdot (\boldsymbol{u} + \boldsymbol{v}) = a \cdot \boldsymbol{u} + a \cdot \boldsymbol{v}$ and $(a + b) \cdot \boldsymbol{v} = a \cdot \boldsymbol{v} + b \cdot \boldsymbol{v}$ (distributive law).
3. For any $\boldsymbol{v} \in V$ and any $a, b \in F$, $(a \cdot b) \cdot \boldsymbol{u} = a \cdot (b \cdot u)$ (associative law).
4. Let 1 be the unit element in F, then $1 \cdot \boldsymbol{v} = \boldsymbol{v}$ for any $\boldsymbol{v} \in V$.

The addition + defined over V is the vector addition. The multiplication between the scalars in F and the vectors in V is the scalar multiplication. F is the scalar field of the vector space V.

The vector space of particular interest to error control coding is the vector space over the binary field, denoted by $V_2^{(n)}$. Each element of $V_2^{(n)}$ is a binary vector of length n (we call it an ***n*-*tuple***):

$$\boldsymbol{v} = (v_0\ v_1\ v_2 \cdots v_{n-1})$$

where $v_i \in \{0, 1\}$. $V_2^{(n)}$ has a total of 2^n different vectors. Let $\boldsymbol{u} = (u_0\ u_1\ u_2 \cdots u_{n-1})$ and $\boldsymbol{v} = (v_0\ v_1\ v_2 \cdots v_{n-1})$ be two elements of $V_2^{(n)}$. The binary vector addition is defined as:

$$\boldsymbol{u} + \boldsymbol{v} \triangleq \left(u_0 + v_0\ u_1 + v_1\ u_2 + v_2 \cdots u_{n-1} + v_{n-1}\right) \tag{2.1}$$

where $u_i + v_i$ is carried out as modulo-2 addition (XOR). The scalar multiplication is defined as:

$$a \cdot \boldsymbol{u} \triangleq a \cdot \left(u_0\ u_1\ u_2 \cdots u_{n-1}\right) = \left(au_0\ au_1\ au_2 \cdots au_{n-1}\right) \tag{2.2}$$

where au_i is performed as modulo-2 multiplication (AND). The all-zero vector $\mathbf{0} = (000 \cdots 0)$ is the additive identity:

$$\boldsymbol{u} + \mathbf{0} = \boldsymbol{u} \text{ and } \boldsymbol{u} + \boldsymbol{u} = \mathbf{0}$$

Example 2.5

For $n = 3$, the vector space $V_2^{(3)}$ has the following eight vector elements:

$$(000), (001), (010), (011), (100), (101), (110), (111),$$

Pick any two vectors, say, (010) and (100). We then have:

$$(010) + (100) = (0 + 1 \;\; 1 + 0 \;\; 0 + 0) = (110),$$

which is also an element in $V_2^{(3)}$. The scalar multiplication is also easy:

$$0 \cdot (010) = (000) \text{ and } 1 \cdot (010) = (010).$$

For a vector space V defined over a field F, it is possible to find a subset of vectors inside V that also meets the above four conditions to be a vector space. Such a subset V' is called the subspace of the vector space V provided the following conditions are satisfied:

1. For any two vectors in V', $\boldsymbol{u}$ and $\boldsymbol{v}$, the sum vector $\boldsymbol{u} + \boldsymbol{v} \in V'$.
2. For any scalar $a \in F$ and any vector $\boldsymbol{v} \in V'$, the scalar multiplication $a \cdot \boldsymbol{v} \in V'$.

It has been shown that if $\boldsymbol{v}_0, \boldsymbol{v}_1, \cdots, \boldsymbol{v}_{k-1}$ are k vectors in a vector space V over F, all possible linear combinations of $\boldsymbol{v}_0, \boldsymbol{v}_1, \cdots, \boldsymbol{v}_{k-1}$, that is:

$$a_0 \boldsymbol{v}_0 + a_1 \boldsymbol{v}_1 + \cdots + a_{k-1} \boldsymbol{v}_{k-1}$$

constitute a subspace of V, where $a_0, a_1, \cdots, a_{k-1}$ are the scalar elements in the field F [4, p. 58].

Example 2.6

Consider two vectors (101) and (010) in the vector space $V_2^{(3)}$. All linear combinations of the two vectors are:

$$0 \cdot (101) \otimes 0 \cdot (010) = (000)$$

$$0 \cdot (101) \otimes 1 \cdot (010) = (010)$$

$$1 \cdot (101) \otimes 0 \cdot (010) = (101)$$

$$1 \cdot (101) \otimes 1 \cdot (010) = (111)$$

The subset {(000), (010), (101), (111)} not only meets the four conditions to be a vector space but also satisfies the two conditions to be a subspace. So it is a subspace of $V_2^{(3)}$.

2.2 Galois Field and Its Arithmetic

2.2.1 Galois Field

2.2.1.1 Definition

A field with a finite number of elements is a finite field, or *Galois field*, in memory of its discoverer É. Galois. A Galois field of order q is denoted by $GF(q)$. The Galois field is extensively used in coding theory. The binary field in Example 2.3 is a Galois field. If the order of a Galois field is a prime p, then the field is called the *prime field*. Evidently the binary field is moreover a prime field with $p = 2$.

For any Galois field $GF(q)$, there exists a smallest positive integer λ such that:

$$\underbrace{1+1+\cdots+1}_{\lambda\ 1\text{'s}} = 0$$

The integer λ is called the *characteristic* of the field. The characteristic of $GF(2)$ is 2 because $1 \oplus 1 = 0$. For a prime field $GF(p)$, the characteristic is p. It can be shown that the characteristic of any Galois field must be a prime [4, p. 35].

The preceding definition of a characteristic implies that, for any integer $0 < k < \lambda$,

$$\underbrace{1+1+\cdots+1}_{k\ 1\text{'s}} \neq 0$$

The sum is in fact distinct with different k, and these λ distinct sums, $1,(1+1),\cdots,\underbrace{(1+1+\cdots+1)}_{\lambda\ 1\text{'s}}$, form a field $GF(\lambda)$ under addition and multiplication. The resulting field $GF(\lambda)$ is a subfield of $GF(q)$.

The Galois field of order $q = p^m$, where p is a prime and m is a positive integer, is called the *extension Galois field* of $GF(p)$, denoted by $GF(p^m)$. $GF(p)$ is the *ground field* of $GF(p^m)$, where $GF(p^m)$ contains a total of p^m elements (including the zero element 0 and the unit element 1). Of primary interest to error control codes is $GF(2^m)$, which we will describe later in detail.

An important property of a Galois field is that there always exists at least one element whose powers constitute the set of all nonzero elements of the field [4, p. 37], and this element is referred to as a primitive element of the field. Let α denote such a primitive element. All $q-1$ nonzero elements of the field can be obtained as follows:

$$\alpha^0(=1), \alpha, \alpha^2, \alpha^3, \cdots, \alpha^{q-2} \tag{2.3}$$

This equation says that an extension field can be entirely constructed by its primitive element using (2.3). The exponential representation of a field element as in (2.3) is called the power representation of the field element. It has been shown [4, p. 36] that higher powers of α repeat the pattern in (2.3), that is:

$$\alpha^{q-1} = \alpha^0 = 1$$

$$\alpha^q = \alpha$$

$$\alpha^{q+1} = \alpha^2$$

$$\vdots$$

Let $\beta = \alpha^i$ be an arbitrary nonzero element of the field $GF(q)$. The nth power of the element is:

$$\beta^n = (\alpha^i)^n = \alpha^{i \cdot n}$$

where i is some integer. If $i \cdot n$ is a multiple of $q-1$, then:

$$\beta^n = \alpha^{i \cdot n} = \alpha^{q-1} \cdot \alpha^{q-1} \cdots \alpha^{q-1} = 1$$

The *order of field element* β is defined as the smallest number n such that $\beta^n = 1$.

2.2.1.2 Galois Field and Polynomial

An alternative to the power representation of a field element is the polynomial representation. We now introduce the polynomial representation for the extension field $GF(p^m)$. We begin this with the concept of polynomial over $GF(p)$.

A polynomial over a prime Galois field $GF(p)$ of degree n is a single-variate polynomial whose coefficients are from $GF(p)$:

$$f(X) = f_0 + f_1X + f_2X^2 + \cdots + f_nX^n \tag{2.4}$$

where the polynomial coefficient f_i $(i = 0, 1, \cdots, n)$ is one of the p elements in $GF(p)$. The polynomial over $GF(p)$ can be added, subtracted, multiplied, and divided just like an ordinary polynomial except the polynomial coefficients are computed using modulo-p arithmetic.

We must mention the following useful property, which is associated with the polynomials over $GF(2)$, that is:

$$f^2(X) = f\left(X^2\right) \tag{2.5}$$

Equation (2.5) can be derived as follows [5, p. 210]:

$$\begin{aligned}
f^2(X) &= \left(f_0 + f_1X + f_2X^2 + \cdots + f_nX^n\right)^2 \\
&= \left[f_0 + \left(f_1X + f_2X^2 + \cdots + f_nX^n\right)\right]^2 \\
&= f_0^2 + f_0\left(f_1X + f_2X^2 + \cdots + f_nX^n\right) + f_0\left(f_1X + f_2X^2 + \cdots + f_nX^n\right) \\
&\quad + \left(f_1X + f_2X^2 + \cdots + f_nX^n\right)^2 \\
&= f_0^2 + \left(f_1X + f_2X^2 + \cdots + f_nX^n\right)^2
\end{aligned}$$

Repeatedly applying the above procedure yields:

$$f^2(X) = f_0^2 + (f_1X)^2 + (f_2X^2)^2 + \cdots + (f_nX^n)^2$$

Consider that $f_i = 0, 1$ and $f_i^2 = f_i$; hence, the preceding equality is simplified to:

$$f^2(X) = f_0 + f_1X^2 + f_2(X^2)^2 + \cdots + (f_nX^2)^n = f(X^2)$$

In fact, it holds true that, for any $i \geq 0$,

$$f^{2^i}(X) = f\left(X^{2^i}\right) \tag{2.6}$$

An immediate inference from (2.6) is that if β is a root of a polynomial over a prime Galois field then β^{2^i} must also be its root:

$$f(\beta^{2^i}) = f^{2^i}(\beta) = 0$$

So the existence of β as a polynomial root implies the existence of β^2, β^{2^2}, β^{2^3}, $\cdots$ as roots. $\beta^2, \beta^{2^2}, \beta^{2^3}, \cdots$ are called the *conjugates* of β. For example, α is a root of the binary polynomial $X^3 + X + 1$; α^2, α^4, $\cdots$ are also its roots.

A polynomial $\vartheta(X)$ over $GF(p)$ of degree m is *irreducible* over $GF(p)$ if it is not divisible by any polynomial over $GF(p)$ of degree less than m. For instance, polynomial $1 + X + X^3$ is an irreducible polynomial over $GF(2)$, but $X + X^3$ is not because $(X + X^3)/X = 1 + X^2$. Irreducible polynomials have an important feature, that is, any irreducible polynomial of degree m divides $X^{p^m - 1} - 1$ [5, p. 207]. This is easy to verify. Dividing $X^{2^3 - 1} - 1 = X^7 - 1$ by $1 + X + X^3$, we find the quotient is $1 + X + X^2 + X^4$ and no remainder.[1]

An irreducible polynomial $\varphi(X)$ of degree m is said to be *primitive* if the smallest positive integer n for which $\varphi(X)$ divides $X^n - 1$ is $n = p^m - 1$. Interestingly, the roots of an mth degree primitive polynomial $\varphi(X)$ over $GF(p)$ are primitive elements of some extension field $GF(p^m)$ [5, p. 208]. We have already stated that an extension Galois field can be completely constructed on its primitive element. Consequently, we can also say that an extension field is built on a primitive polynomial.

MATLAB Experiment 2.1

We use MATLAB to verify that $1 + X + X^3$ is a primitive polynomial; that is, it divides $X^7 - 1$ but not $X^n - 1$ for $0 < n < 7$.

```
>> p1 = [1 1 0 1];               % polynomial 1 + x + x^3
>> p2 = [1 0 0 0 0 0 0 1];       % polynomial 1 + x^7
>> p3 = [1 0 0 0 0 0 1];         % polynomial 1 + x^6
>> % for polynomials over GF(2), x^7 - 1 = 1 + x^7,
>> % x^6 - 1 = 1 + x^6, ...
>> [q,r] = gfdeconv(p2,p1);      % (1 + x^7)/(1 + X + x^3)
>>                               % q: quotient, r: remainder
>> r
r =
      0
>> [q,r] = gfdeconv(p3, p1);     % (1 + x^6)/(1 + x + x^3)
>> r
r =
      0 0 1
```

So, $1 + X + X^3$ divides $X^7 - 1$ but not $X^6 - 1$. Similarly, we can find that $1 + X + X^3$ does not divide $X^5 - 1$, $X^4 - 1$, and so forth.

[1] Note that $X^7 - 1 = X^7 + 1$ over $GF(2)$.

Table 2.1
Some Primitive Polynomials over $GF(2)$

m	Primitive Polynomial
3	$1 + X + X^3$
4	$1 + X + X^4$
5	$1 + X^2 + X^5$
6	$1 + X + X^6$
7	$1 + X^3 + X^7$
8	$1 + X^2 + X^3 + X^4 + X^8$
9	$1 + X^4 + X^9$
10	$1 + X^3 + X^{10}$
11	$1 + X^2 + X^{11}$
12	$1 + X + X^4 + X^6 + X^{12}$
13	$1 + X + X^3 + X^4 + X^{13}$
14	$1 + X + X^6 + X^{10} + X^{14}$
15	$1 + X + X^{15}$
16	$1 + X + X^3 + X^{12} + X^{16}$
17	$1 + X^3 + X^{17}$
18	$1 + X^7 + X^{18}$
19	$1 + X + X^2 + X^5 + X^{19}$
20	$1 + X^3 + X^{20}$

Primitive polynomials play a central role in coding theory. Although theoretically an irreducible but nonprimitive polynomial can also define an extension field, it is more convenient to use a primitive polynomial to generate an extension field since its root is the primitive element of the field. Table 2.1 lists primitive polynomials up to degree 20. Note that, for each *m*, only the default primitive polynomial (i.e., the one that contains the fewest terms) is given.

MATLAB Experiment 2.2

It is convenient to use MATLAB function `gfprimdf(m,p)` to generate the default primitive polynomials over $GF(p)$ of degree `m`.

```
>> p = 2;                      % order of ground field p
>> m = 3;                      % degree of primitive polynomial
>> primpoly = gfprimdf(m,p)
```

```
primpoly =
       1  1  0  1
```

The result corresponds to the polynomial $1 + X + X^3$.

MATLAB Experiment 2.3

To check whether a polynomial `poly` over a Galois field of degree `m` is primitive, we need to examine if `poly` is irreducible and if the smallest positive integer n for which `poly` divides $X^n - 1$ is $n = 2^m - 1$. In practice, we may let the MATLAB function `gfprimck(poly)` do it for us.

```
>> poly = [1 1 0 1];             % polynomial = 1 + x + x^3
>> gfprimck(poly)
ans =
       1
```

The result indicates that $1 + X + X^3$ is a primitive polynomial (see MATLAB Experiment 2.2).

Comment: The function returns –1 if `poly` is not an irreducible polynomial; 0 if `poly` is irreducible, but not a primitive polynomial for $GF(2^m)$; and 1 if `poly` is a primitive polynomial for $GF(2^m)$.

Having defined the polynomial over $GF(p)$, we are now ready to represent all elements of the extension field $GF(p^m)$ by polynomials over $GF(p)$ of degree less than m. Let $\varphi(X)$ be the primitive polynomial that generates $GF(p^m)$, and i be an integer. X^i can be expressed as:

$$X^i = q(X)\varphi(X) + r(X) \tag{2.7}$$

where $q(X)$ and $r(X)$ are quotient and remainder, respectively. Letting α denote the root of $\varphi(X)$ (it is also a primitive element of the extension field) and inserting $X = \alpha$ into (2.7), we have:

$$\alpha^i = q(\alpha)\varphi(\alpha) + r(\alpha) \tag{2.8}$$

Considering $\varphi(\alpha) = 0$, we obtain the following equation from (2.8):

$$\alpha^i = r(\alpha) = r_0 + r_1\alpha + r_2\alpha^2 + \cdots + r_{m-1}\alpha^{m-1} \tag{2.9}$$

Equation (2.9) indicates that an arbitrary element α^i can be represented as the polynomial $r(\alpha)$, which is obtained as:

$$\alpha^i \Rightarrow X^i \bmod \varphi(X)\,|_{X=\alpha} \tag{2.10}$$

Example 2.7

Consider the extension Galois field GF(2^3), or GF(8). The seven nonzero elements of the field are α^0 (= 1), α, α^2, $\cdots$, α^6. The primitive polynomial of degree 3 is $1+X+X^3$ (see Table 2.1). Using the polynomial to construct the field, the polynomial representations of all nonzero elements of the field are as follows:

$$\alpha^0 \Rightarrow 1$$

$$\alpha^1 \Rightarrow \alpha$$

$$\alpha^2 \Rightarrow \alpha^2$$

$$\alpha^3 \Rightarrow X^3 \bmod 1+X+X^3|_{x=\alpha} = 1+\alpha$$

$$\alpha^4 = \alpha \cdot \alpha^3 = \alpha(1+\alpha) = \alpha+\alpha^2$$

$$\alpha^5 = \alpha^1 \cdot \alpha^4 = \alpha(\alpha+\alpha^2) = \alpha^2+\alpha^3 \Rightarrow 1+\alpha+\alpha^2$$

$$\alpha^6 = \alpha^1 \cdot \alpha^5 = \alpha(\alpha^2+\alpha^3) = \alpha^3+\alpha^4 \Rightarrow 1+(1+1)\alpha+\alpha^2 = 1+\alpha^2$$

Often the polynomial representation is abbreviated to a vector. For instance, $\alpha^5 \Rightarrow 1+\alpha+\alpha^2 \Rightarrow (111)$. The mapping between the three representations is summarized in Table 2.2.

MATLAB Experiment 2.4

Mapping between the different representations for a Galois field can be done in MATLAB. Run the following script to convert the power representation to the polynomial representation.

```
>> indx = [0;1;2;3;4;5;6];              % power index
>> % prime polynomial = 1 + x + x^3
>> primpoly = [1 1 0 1];
>> p = 2;                               % over GF(2)
>> poly = gftuple(indx,primpoly,p)      % polynomial in
                                        % vector form
```

Table 2.2
Mapping Between Different Representations of *GF*(8)

Power Representation	Polynomial Representation	Vector Representation
$\alpha^{-\infty}$	0	(000)
$\alpha^0 = 1$	1	(100)
α	α	(010)
α^2	α^2	(001)
α^3	$1 + \alpha$	(110)
α^4	$\alpha + \alpha^2$	(011)
α^5	$1 + \alpha + \alpha^2$	(111)
α^6	$1 + \alpha^2$	(101)

```
poly =
          1  0  0
          0  1  0
          0  0  1
          1  1  0
          0  1  1
          1  1  1
          1  0  1
```

So the polynomial representations of α^0, α^1, α^2, α^3, α^4, $\cdots$ are 1, α, α^2, $1 + \alpha$, $\alpha + \alpha^2$, $\cdots$.

Finally we give one more definition before concluding this section. Let $\phi(X)$ be an irreducible polynomial over $GF(p)$. If $\phi(X)$ has β as its root, where β is an element in $GF(p^m)$, then $\phi(X)$ is the *minimum polynomial* of β. Take Galois field $GF(8)$ as an example. The field is constructed on the primitive polynomial $1 + X + X^3$. Substituting $X = \alpha$ into the polynomial gives:

$$1 + \alpha + \alpha^3 = 1 + \alpha + (1 + \alpha) = 0$$

The polynomial is therefore the minimum polynomial of α.

The minimum polynomial of β can be obtained as [4, p. 51]:

$$\phi(X) = \prod_{i=0}^{L-1}\left(X - \beta^{2^i}\right) \tag{2.11}$$

where L is the smallest integer such that $\beta^{2L} = \beta$.

Example 2.8

We have verified that the minimum polynomial of the primitive element α in $GF(2^3)$ is $1 + X + X^3$. Now we want to find the minimum polynomial of $\beta = \alpha^3$. Notice:

$$\beta^2 = \alpha^6 = 1 + \alpha^2$$

$$\beta^{2^2} = \alpha^{12} = 1 + \alpha + \alpha^2$$

$$\beta^{2^3} = \alpha^{24} = \alpha^3 = \beta$$

The minimum polynomial is then obtained as:

$$\begin{aligned}\phi(X) &= (X - \beta)(X - \beta^2)(X - \beta^{2^2}) = (X - \alpha^3)(X - \alpha^6)(X - \alpha^{12}) \\ &= 1 + X^2 + X^3\end{aligned}$$

The polynomial is listed in the second row of Table 2 3.

MATLAB Experiment 2.5

The size of Table 2.3 grows exponentially as we add more fields into it. Therefore, for more minimum polynomials, we turn to MATLAB. The MATLAB Communications Toolbox offers a function `gfminpoly` that returns the minimal polynomial of an element in a Galois field. The following finds the minimum polynomial of α^{23} in $GF(2^6)$:

```
>> k = 23;                 % the element is alpha^23
>> m = 6;                  % field is GF(2^6)
>> minpoly = gfminpol(k,m)
minpoly =
              1  1  0  0  1  1  1
```

The result is $1 + X + X^4 + X^5 + X^6$.

2.2.2 Arithmetic in *GF*(2^m)

In digital communications and storage, data are represented in binary. As such, only binary field $GF(2)$ and its extension field $GF(2^m)$ are of primary concern to error control coding. This section concentrates on the arithmetic in $GF(2^m)$.

Table 2.3
List of Minimum Polynomials

Galois Field	Element	Minimum Polynomial	Element	Minimum Polynomial
$GF(2^2)$	α	$1 + X + X^2$	—	—
$GF(2^3)$	α	$1 + X + X^3$	α^3	$1 + X^2 + X^3$
$GF(2^4)$	α	$1 + X + X^4$	α^3	$1 + X + X^2 + X^3 + X^4$
	α^5	$1 + X + X^2$	α^7	$1 + X^3 + X^4$
$GF(2^5)$	α	$1 + X^2 + X^5$	α^3	$1 + X^2 + X^3 + X^4 + X^5$
	α^5	$1 + X + X^2 + X^4 + X^5$	α^7	$1 + X + X^2 + X^3 + X^5$
	α^{11}	$1 + X + X^3 + X^4 + X^5$	α^{15}	$1 + X^3 + X^5$
$GF(2^6)$	α	$1 + X + X^6$	α^3	$1 + X + X^2 + X^4 + X^6$
	α^5	$1 + X + X^2 + X^5 + X^6$	α^7	$1 + X^3 + X^6$
	α^9	$1 + X^2 + X^3$	α^{11}	$1 + X^2 + X^3 + X^5 + X^6$
	α^{13}	$1 + X + X^3 + X^4 + X^6$	α^{15}	$1 + X^2 + X^4 + X^5 + X^6$
	α^{21}	$1 + X + X^2$	α^{23}	$1 + X + X^4 + X^5 + X^6$
	α^{27}	$1 + X + X^3$	α^{31}	$1 + X^5 + X^6$

2.2.2.1 Addition and Subtraction

Galois field arithmetic differs from integer arithmetic in that the former has only a finite number of possible values for the operand. Consequently, Galois field arithmetic must be performed with modulo operation.

For binary field $GF(2)$, addition is performed as modulo-2 addition, or exclusive-or (XOR) operation, defined in Example 2.1. Subtraction in $GF(2)$ is exactly the same as addition:

$$u - v = u + v = u \oplus v$$

where $u, v \in GF(2)$.

We have learned that any element of $GF(2^m)$ can be represented as a polynomial with its coefficients taken from $GF(2)$. Therefore, addition and subtraction of any two elements in $GF(2^m)$ can be accomplished as follows:

$$w(\alpha) = u(\alpha) + v(\alpha) = (u_0 + v_0) + (u_1 + v_1)\alpha + \cdots + (u_{m-1} + v_{m-1})\alpha^{m-1} \tag{2.12}$$

where $u(\alpha) = u_0 + u_1\alpha + u_2\alpha^2 + \cdots + u_{m-1}\alpha^{m-1}$ and $v(\alpha) = v_0 + v_1\alpha + v_2\alpha^2 + \cdots + v_{m-1}\alpha^{m-1}$ are two elements of $GF(2^m)$. Calculation of $u_i + v_i$ is carried out using modulo-2 addition, and $w(\alpha) = w_0 + w_1\alpha + w_2\alpha^2 + \cdots + w_{m-1}\alpha^{m-1}$ is the sum.

Example 2.9

Suppose we want to add α^3 and α^5 in GF(8). From Table 2.2 we have their polynomial presentations as:

$$\alpha^3 \Rightarrow 1 + \alpha \text{ and } \alpha^5 \Rightarrow 1 + \alpha + \alpha^2$$

So,

$$\alpha^3 + \alpha^5 = (1 + \alpha) + (1 + \alpha + \alpha^2) = \alpha^2$$

2.2.2.2 Multiplication

Multiplication in a Galois field $GF(2^m)$ is accomplished by adding the power of the multiplicand and the power of the multiplier together, followed by modulo-$(2^m - 1)$ operation:

$$\alpha^i \cdot \alpha^j = \alpha^{(i+j) \bmod 2^m - 1} \tag{2.13}$$

where α^i and α^j are the multiplicand and multiplier, and the right side of the equation is the product.

When the multiplication is performed using the polynomial representation, the product equals the ordinary multiplication of the two polynomials modulo the primitive polynomial:

$$w(\alpha) = u(X)v(X) \bmod \varphi(X)\,|_{x=\alpha} \tag{2.14}$$

where $u(\alpha)$ and $v(\alpha) \in GF(2^m)$ are the two operands, $w(\alpha)$ is the product, and $\varphi(X)$ is the primitive polynomial on which $GF(2^m)$ is built.

Example 2.10

Let us multiply α^3 by α^5 in $GF(2^3)$. If power representation is used, the product is obtained as:

$$\alpha^3 \cdot \alpha^5 = \alpha^{(3+5) \bmod 2^3 - 1} = \alpha$$

On the other hand, if polynomial representation is used, the multiplication should be performed as follows:

$$
\begin{aligned}
\alpha^3 \cdot \alpha^5 &= (1+X)\cdot\ (1+X+X^2) \bmod (1+X+X^3)\big|_{X=\alpha} \\
&= 1 + X^3 \bmod (1+X+X^3)\big|_{X=\alpha} \\
&= \alpha
\end{aligned}
$$

where $1 + X + X^3$ is the primitive polynomial that we used to construct the field.

MATLAB Experiment 2.6

MATLAB contains a set of functions dedicated to Galois field arithmetic such as `gfadd`, `gfsub`, and `gfmul`. Readers should consult the MATLAB User's Guide for details about using the functions. This book's companion DVD provides two simple subroutines: one is `gfmulp*` for polynomial-based multiplication; the other is `gfinvp*` for polynomial-based inversion.

The following script multiplies two elements in polynomial representations.

```
>> u = [1 1];         % multiplicand 1 + alpha = alpha^3
>> v = [1 1 1];       % multiplier 1 + alpha + alpha^2 =
                      % alpha^5
>> p = [1 1 0 1];     % primitive polynomial 1 + alpha +
                      % alpha^3
>> gfmulp(u,v,p)      % the product
ans =
       0  1
```

The product is α.

2.3 Implementation of *GF*(2^m) Arithmetic

2.3.1 Arithmetic with Polynomial Representation

Addition and subtraction in *GF*(2^m) using the polynomial representation are easy. Based on (2.12), the circuit can be built using simple XOR gates as shown in Figure 2.1.

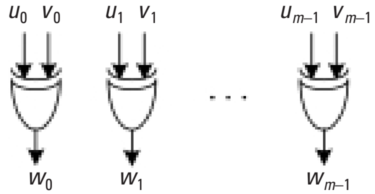


Figure 2.1 Polynomial-based Galois field adder.

Multiplication, however, is more complex. It follows from (2.14):

$$w(X) = u(X)v(X) \bmod \varphi(X) \tag{2.15}$$

Direct implementation of (2.15) involves two separate steps with the second step done iteratively:

1. Polynomial multiplication;
2. Reduction modulo the primitive polynomial.

The above implementation approach is often not favored in high-speed design because it is a multiple-cycle operation and the circuit thus must be clocked at a much higher frequency than the data rate. However, given the primitive polynomial $\varphi(X) = \varphi_0 + \varphi_1 X + \cdots + \varphi_m X^m$, it is possible to complete the second step off-line. Let $u(X) = u_0 + u_1 X + \cdots + u_{m-1} X^{m-1}$, $v(X) = v_0 + v_1 X + \cdots + v_{m-1} X^{m-1}$. We then have [6]:

$$\begin{aligned} w(X) &= u(X) \cdot v(X) \bmod \varphi(X) \\ &= (d_0 + d_1 X + \cdots + d_{m-1} X^{m-1} + d_m X^m + \cdots + d_{2m-2} X^{2m-2}) \bmod \varphi(X) \\ &= d_0 + d_1 X + \cdots + d_{m-1} X^{m-1} \\ &\qquad + d_m [X^m \bmod \varphi(X)] + \cdots + d_{2m-2} [X^{2m-2} \bmod \varphi(X)] \end{aligned} \tag{2.16}$$

where

$$d_k = \begin{cases} \displaystyle\sum_{i=0}^{k} u_i v_{k-i} & k = 0, 1, 2, \cdots, m-1 \\ \displaystyle\sum_{i=k}^{2m-2} u_{k-i+(m-1)} v_{i-(m-1)} & k = m, m+1, \cdots, 2m-2 \end{cases} \tag{2.17}$$

Since the primitive polynomial $\varphi(X)$ is known, terms X^m mod $\varphi(X)$, X^{m+1} mod $\varphi(X)$, ⋯ , X^{2m-2} mod $\varphi(X)$, all can be precomputed. Let $r_i(X) = r_{i,0} + r_{i,1}X + \cdots + r_{i,(m-1)}X^{m-1} = X^i$ mod $\varphi(X)$, where $m \le i \le 2m - 2$. The coefficients of the product $w(\alpha) = w_0 + w_1X + \cdots + w_{m-1}X^{m-1}$ are obtained as follows:

$$w_j = d_j + \sum_{i=m}^{2m-2} d_i r_{i,j} \qquad (j = 0,1,\cdots,m-1) \tag{2.18}$$

Note that the design results in a purely combinational logic circuit.

Example 2.11

Now we use the method to design a multiplier for GF(2^3) with the primitive polynomial $\varphi(X) = 1 + X + X^3$. It is easy to find:

$$r_3(X) = X^3 \bmod \varphi(X) = 1 + X$$

$$r_4(X) = X^4 \bmod \varphi(X) = X + X^2$$

The multiplication is thus realized by the following equations:

$$w_0 = d_0 + d_3$$

$$w_1 = d_1 + d_3 + d_4$$

$$w_2 = d_2 + d_4$$

where

$$d_0 = u_0v_0$$

$$d_1 = u_0v_1 + u_1v_0$$

$$d_2 = u_0v_2 + u_1v_1 + u_2v_0$$

$$d_3 = u_2v_1 + u_1v_2$$

$$d_4 = u_2v_2$$

The schematic of the multiplier is given in Figure 2.2.

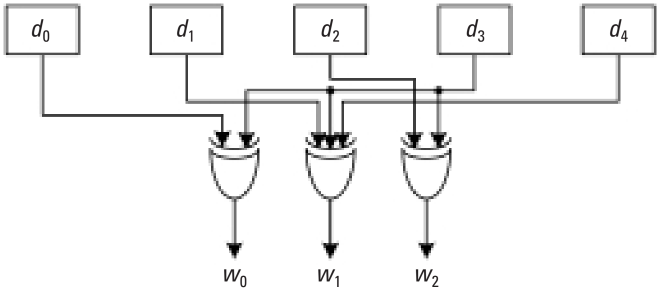


Figure 2.2 *GF*(2^3) multiplier.

2.3.2 Arithmetic with Power Representation

The simplest way to perform multiplication in a Galois field is to represent the operands in the power basis and use (2.13). However, addition in this case becomes awkward. A clever solution to the problem is the *Zech algorithm*. The algorithm defines a variable called the *Zech logarithm* $z(n)$:

$$\alpha^{z(n)} \triangleq 1+\alpha^{n} \qquad (n=1,2,\cdots,2^{m}-2) \tag{2.19}$$

The Zech algorithm transforms addition into multiplication as follows:

$$\alpha^{i}+\alpha^{j}=\alpha^{i}(1+\alpha^{j-i})=\alpha^{i}\alpha^{z(j-i)} \tag{2.20}$$

The algorithm requires one table look-up (to search for the Zech logarithm) and one multiplication. In many implementations this method improves the latency quite significantly.

Example 2.12

The Zech logarithm for GF(8) is listed in the following table:

n	$z(n)$
1	3
2	6
3	1
4	5
5	4
6	2

Using this table, we have:

$$\alpha^3 + \alpha^5 = \alpha^3(1+\alpha^2) = \alpha^3\alpha^{z(2)} = \alpha^3\alpha^6 = \alpha^2$$

The schematic of a general adder based on the Zech algorithm is drawn in Figure 2.3.

2.3.3 A Special Case: Inversion

So far we have not touched division. The common practice for computing the quotient v/u in $GF(2^m)$ is to multiply the dividend v by the multiplicative inverse (or reciprocal), u^{-1}, of the divisor u. So we should examine possible implementations of the inversion operation. Fortunately the majority of error coding applications do not require single-cycle division, leaving some flexibility for our designs.

The most straightforward way to implement element inversion in a Galois field is table look-up. Inverses are precomputed and stored in a $2^m \times m$ read-only memory (ROM). The divisor u is used as the access address to the ROM. The problem with this approach is that for large fields a lot of memory space is required, resulting in high costs.

By definition the inverse of an element in a Galois field must be another element in the same field. As such, we may find the reciprocal of u by testing $bu = 1$ for every nonzero element b in the field. This brute-force search can be realized using a pair of linear feedback shift registers (LFSR). The technique is best explained by using an example.

Example 2.13

*Take as an example the Galois field GF(2^3) defined by the primitive polynomial $\varphi(X) = 1 + X + X^3$. We want to find the reciprocal of **u** = (110). As depicted in Figure 2.4, the two LFSRs of length 3 are individually connected, corresponding to $\varphi(X)$. Upon reset, the left LFSR is loaded with u*

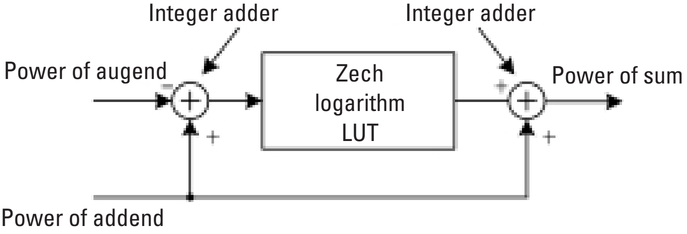


Figure 2.3 Galois field adder based on Zech algorithm.

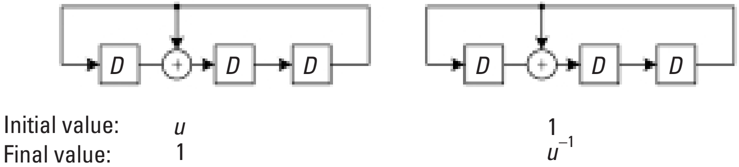


Figure 2.4 $GF(2^3)$ inversion circuit.

> *and the right LFSR with 1. The ratio between the two LFSRs is* $\boldsymbol{u}$*:1. Then both LFSRs shift bit by bit simultaneously. After each shift, the content of the LFSR changes to a different element in the field. During the shifting process, the ratio between the two LFSRs remains* $\boldsymbol{u}$*:1. As a result, when the left LFSR reaches 1, the content of the right LFSR must be* $\boldsymbol{u}^{-1}$*. The process is illustrated in Table 2.4.*

From Example 2.13 we can see that the LFSR is connected in such a way that it becomes the so-called maximum-length LFSR and is able to exhaust all $2^m - 1$ nonzero elements in the field (although in the example we were lucky to obtain the inverse before exhausting all the elements). Also, each shift of the LFSR is equivalent to multiplying its content by X followed by modulo-$\varphi(X)$.

One issue associated with the preceding technique is that the number of shifts needed for different elements is different. This may cause some problem when interfacing with other circuit blocks. The following exponentiation method provides a remedy.

Recall that the order of any nonzero element $\beta \in GF(2^m)$ is $2^m - 1$, that is:

$$\beta^{2^m-1} = 1 \tag{2.21}$$

Table 2.4
An LFSR-Based *GF* Inversion Process

Step	Content of Left LFSR	Content of Right LFSR	Note
0	(110)	(100)	$\boldsymbol{u}$ = (110), $\boldsymbol{b}$ = (100) = 1
1	(011)	(010)	
2	(111)	(001)	
3	(101)	(110)	
4	(100)	(011)	Left LFSR = 1, Right LFSR = $\boldsymbol{u}^{-1}$ = (011)

Multiplying both sides of (2.21) by β^{-1}, we have:

$$\beta^{2^m-2} = \beta^{-1} \tag{2.22}$$

The equation indicates that computing the $(2^m - 2)$-th power of an element gives the inverse of the element. Notice that $2^m - 2$ can be expanded as follows (Fermat's theorem):

$$2^m - 2 = 2 + 2^2 + \cdots + 2^{m-1} \tag{2.23}$$

The inverse can then be calculated quickly as [7]:

$$\beta^{-1} = \beta^{2^m-2} = \beta^2\beta^{2^2}\cdots\beta^{2^{m-1}} \tag{2.24}$$

The implementation of (2.23) is depicted in Figure 2.5 [8]. It consists of a register, a multiplier, and a squaring circuit. The register is initially loaded with a 1. As the circuit shifts in a cyclic manner, the successive values of the register become:

$$1 \Rightarrow \beta^2 \Rightarrow \beta^6 \Rightarrow \beta^{14} \Rightarrow \cdots \Rightarrow \beta^{2^m-2} = \beta^{-1}$$

The final value β^{-1} is obtained in $m - 1$ shifts.

Note This chapter may seem a bit dry, but it is absolutely necessary for understanding the materials presented in the remainder of the book. Our introduction to algebra does not stop here. Several things have yet to be covered, for example, Galois field Fourier transforms and Euclid's algorithm. Because these topics are used for some specific codes or algorithms, they will be presented when needed. Also, readers may have noticed that most of the mathematical proofs are omitted

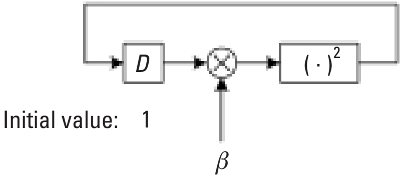


Figure 2.5 Inversion circuit using exponentiation method.

in this chapter. We did this because we think such proofs are the tasks of algebra textbooks.

As for implementation, the designs given in this chapter are rather rudimentary, but they should satisfy many practical needs. Keep in mind that optimal implementations are always considered in the context of applications.

Our official journey of error control coding starts with the next chapter—so be prepared.

Problems

2.1 Show that every extension Galois field has a primitive element. (*Hint:* An extension Galois field is defined by a primitive polynomial.)

2.2 How many roots does $f(X) = X^4 + \alpha^3X^3 + X^2 + \alpha X + \alpha^3$ over $GF(2^3)$ have [where α is a primitive element of $GF(2^3)$]? Find the roots (see Table 2.2). Use the MATLAB function `gfroots` to verify your result.

2.3 Divide $f(X) = X^4 + \alpha^3X^3 + X^2 + \alpha X + \alpha^3$ over $GF(2^3)$ by $g(X) = X^2 + \alpha X + \alpha^5$ over the same field (α is a primitive element of the field). Find the quotient and the remainder (again, use Table 2.2). Use the MATLAB function `gfdeconv` to verify your result.

2.4 $1 + X + X^4$ is a primitive polynomial of $GF(2^4)$ (see Table 2.1).

a. Use the polynomial to construct $GF(2^4)$ by listing all of its elements in both a power representation and a polynomial representation.

b. Solve the following simultaneous equations in $GF(2^4)$:

$$X + \alpha Y + Z = \alpha^3$$

$$X + \alpha Y + \alpha^3 Z = \alpha^5$$

$$\alpha^2 X + \alpha^6 Y + Z = \alpha$$

2.5 Write a MATLAB function to calculate the Zech logarithm.

2.6 When performing element inversion by exponentiation, is there any other way possibly faster than (2.24)?

References

[1] Clark, A., *Elements of Abstract Algebra*, New York: Dover Publications, 1984.

[2] Ash, R. B., *Basic Abstract Algebra for Graduate Students and Advanced Undergraduates*, New York: Dover Publications, 2006.

[3] Hungerford, T. W., *Abstract Algebra—An Introduction*, 2nd ed., New York: Saunders College Publishing, 1997.

[4] Lin, S., and D. J. Castello, *Error Control Coding—Fundamentals and Applications*, 2nd ed., Upper Saddle River, NJ: Prentice-Hall, 2004.

[5] Moon, T. K., *Error Correction Coding—Mathematical Methods and Algorithms*, New York: John Wiley & Sons, 2005.

[6] Deschamps, J.-P. J. L. Imaña, and G. D. Sutter, *Hardware Implementation of Finite-Field Arithmetic*, New York: McGraw-Hill, 2009.

[7] Wang, C. C., et al., "VLSI Architectures for Computing Multiplications and Inverses in $GF(2^m)$," *IEEE Trans. Comput.*, Vol. C-34, No. 8, August 1985, pp. 709–717.

[8] Gill, J., "Galois Field Arithmetic Implementation," Class Handout, ee387, Algebraic Error Control Codes, Stanford University, 2008.

Selected Bibliography

Berlekamp, E. R., *Algebraic Coding Theory*, New York: McGraw-Hill, 1968.

Lidl, R., and H. Niederreiter, *Finite Fields*, 2nd ed., Cambridge, U.K.: Cambridge University Press, 1997.

McEliece, R. J., *Finite Field for Computer Scientists and Engineers*, New York: Springer, 1986.

3

Binary Block Codes

Having defined block codes in the first chapter, we are now going to discuss the codes in detail. We take the typical top-down approach in presenting the materials. First, linear block codes are described. Then cyclic codes are introduced as a subclass of linear block codes. The most important cyclic codes, namely, BCH codes, are explained last. We confine our discussion to binary codes. Nonbinary block codes are the theme of the next chapter.

3.1 Linear Block Codes

3.1.1 Code Construction and Properties

3.1.1.1 Construction of Block Codes

To encode data into a block code C, we first need to divide the data sequence into blocks of equal length, say, k bits. Then we take each data block $\boldsymbol{m}$ of k bits (we call it a message word or a message vector) and map it into a codeword $\boldsymbol{c}$ of n bits, where $n > k$. The $n - k$ additional parity-check bits are the redundancies added for error detection and correction use. Different redundancy structures lead to different types of codes.

A block code is specified by a set of two parameters (n, k), where n is the length of the message word and k is the codeword length. The rate of the code is k/n. Each message word is associated with one and only one codeword. The total number of codewords in a code equals that of message words, 2^k.

Example 3.1

Table 3.1 lists all codewords of a (7,4) block code. For every message word of four bits, there exists a corresponding codeword of seven bits. Note that this code is a systematic code. The first four MSBs of the codewords are the message bits. The remaining three bits are the parity bits.

Table 3.1

Codewords of a (7,4) Block Code

Message Word (LSB … MSB)	**Codeword (LSB … MSB)**
$\boldsymbol{m}_0$ = (0000)	$\boldsymbol{c}_0$ = (0000000)
$\boldsymbol{m}_1$ = (1000)	$\boldsymbol{c}_1$ = (1101000)
$\boldsymbol{m}_2$ = (0100)	$\boldsymbol{c}_2$ = (0110100)
$\boldsymbol{m}_3$ = (1100)	$\boldsymbol{c}_3$ = (1011100)
$\boldsymbol{m}_4$ = (0010)	$\boldsymbol{c}_4$ = (1110010)
$\boldsymbol{m}_5$ = (1010)	$\boldsymbol{c}_5$ = (0011010)
$\boldsymbol{m}_6$ = (0110)	$\boldsymbol{c}_6$ = (1000110)
$\boldsymbol{m}_7$ = (1110)	$\boldsymbol{c}_7$ = (0101110)
$\boldsymbol{m}_8$ = (0001)	$\boldsymbol{c}_8$ = (1010001)
$\boldsymbol{m}_9$ = (1001)	$\boldsymbol{c}_9$ = (0111001)
$\boldsymbol{m}_{10}$ = (0101)	$\boldsymbol{c}_{10}$ = (1100101)
$\boldsymbol{m}_{11}$ = (1101)	$\boldsymbol{c}_{11}$ = (0001101)
$\boldsymbol{m}_{12}$ = (0011)	$\boldsymbol{c}_{12}$ = (0100011)
$\boldsymbol{m}_{13}$ = (1011)	$\boldsymbol{c}_{13}$ = (1001011)
$\boldsymbol{m}_{14}$ = (0111)	$\boldsymbol{c}_{14}$ = (0010111)
$\boldsymbol{m}_{15}$ = (1111)	$\boldsymbol{c}_{15}$ = (1111111)

Note: LSB = least significant bit; MSB = most significant bit.

3.1.1.2 Linear Block Codes

A binary block code C is said to be linear if and only if its 2^k codewords form a subspace of the vector space consisting of all binary vectors of length n. The code in Example 3.1 is a linear block code. The 15 codewords form a subspace of the vector space $V_2^{(7)}$.

It follows from the property of subspace (refer to Chapter 2) that linear block codes have the following two important properties:

1. The sum of any two codewords in C is another codeword in C:

$$\boldsymbol{c}_i + \boldsymbol{c}_j = \boldsymbol{c}_k \tag{3.1}$$

where $\boldsymbol{c}_i$, $\boldsymbol{c}_j$ and $\boldsymbol{c}_k \in C$.

2. There exists a set of k codewords in C which are *linearly independent* such that every codeword in C is a linear combination of the k codewords:

$$\boldsymbol{c} = m_0\boldsymbol{g}_0 + m_1\boldsymbol{g}_1 + \cdots + m_{k-1}\boldsymbol{g}_{k-1} \tag{3.2}$$

where $\boldsymbol{g}_0$, $\boldsymbol{g}_1$, $\boldsymbol{g}_2$, $\cdots$ $\boldsymbol{g}_{k-1}$ are the k linearly independent codewords, and m_0, m_1, m_2, $\cdots$ m_{k-1} are some scalars. By linearly independent we mean that $m_0\,\boldsymbol{g}_0 + m_1\,\boldsymbol{g}_1 + \ \cdots + m_{k-1}\boldsymbol{g}_{k-1k} \neq 0$ unless m_0, m_1, $\cdots$, $m_{k-1} = 0$.

Note that the codes discussed in this chapter are binary codes; all multiplications are logic AND operations and all additions/subtractions are logic XOR (exclusive-or) operations.

Example 3.2

Consider the linear block code in the previous example. Take any two codewords, say, $\boldsymbol{c}_6$ *= (1000110) and* $\boldsymbol{c}_8$ *= (1010001), and sum them. The result is (0010111), which is* $\boldsymbol{c}_{14}$*. The four linearly independent codewords may be chosen as* $\boldsymbol{c}_1$ *= (1010001),* $\boldsymbol{c}_2$ *= (1110010),* $\boldsymbol{c}_4$ *= (0110100), and* $\boldsymbol{c}_8$ *= (1101000). All codewords can be constructed from these four codewords using simple AND and XOR operations. For example,* $\boldsymbol{c}_9 = 1 \cdot \boldsymbol{c}_1 + 0 + \boldsymbol{c}_2 + 0 \cdot \boldsymbol{c}_4 + 1 \cdot \boldsymbol{c}_8$*. Another example,* $\boldsymbol{c}_{13} = 1 \cdot \boldsymbol{c}_1 + 0 + \boldsymbol{c}_2 + 1 \cdot \boldsymbol{c}_4 + 1 \cdot \boldsymbol{c}_8$*.*

3.1.1.3 Generator Matrix and Parity-Check Matrix

If we choose k message bits as the k coefficients in (3.2), m_0, m_1, m_2, $\cdots$, m_{k-1}, then the codewords and the message words have one-to-one correspondence. We can rewrite (3.2) in matrix form:

$$
\begin{aligned}
\boldsymbol{c} &= (m_0 m_1 m_2 \cdots m_{k-1}) \cdot \begin{bmatrix} \boldsymbol{g}_0 \\ \boldsymbol{g}_1 \\ \vdots \\ \boldsymbol{g}_{k-1} \end{bmatrix} \\
&= \underbrace{(m_0 m_1 m_2 \cdots m_{k-1})}_{\boldsymbol{m}} \cdot \underbrace{\begin{bmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,n-1} \\ g_{1,0} & g_{1,1} & \cdots & g_{1,n-1} \\ \vdots & & & \\ g_{k-1,0} & g_{k-1,1} & \cdots & g_{k-1,n-1} \end{bmatrix}}_{\boldsymbol{G}} \qquad (3.3) \\
&= \boldsymbol{m} \cdot \boldsymbol{G}
\end{aligned}
$$

where $\boldsymbol{m} = (m_0\ m_1\ m_2\ \cdots\ m_{k-1})$ is the message word, and the $k \times n$ matrix $\boldsymbol{G}$ is called the *generator matrix* of the code C. A linear block code is completely determined by its generator matrix.

For systematic codes, the generator matrix $\boldsymbol{G}$ consists of a $k \times (n - k)$ matrix $\boldsymbol{P}$, which produces the parity-check bits of the code, and a $k \times k$ identity matrix $\boldsymbol{I}_k$, side by side:

$$
\boldsymbol{G} = \left[\begin{array}{c|c} \underbrace{\begin{matrix} p_{0,0} & p_{0,1} & \cdots & p_{0,n-k-1} \\ p_{1,0} & p_{1,1} & \cdots & p_{1,n-k-1} \\ \vdots & & & \\ p_{k-1,0} & p_{k-1,1} & \cdots & p_{k-1,n-k-1} \end{matrix}}_{\boldsymbol{P}} & \underbrace{\begin{matrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ & \vdots & \ddots & & \vdots \\ 0 & 0 & \cdots & 0 & 1 \end{matrix}}_{\boldsymbol{I}_k} \end{array}\right] = [\boldsymbol{P} \mid \boldsymbol{I}_k] \qquad (3.4)
$$

The generator matrix for the (7,4) systematic code example is:

$$
\boldsymbol{G} = \begin{bmatrix} \boldsymbol{g}_0 \\ \boldsymbol{g}_1 \\ \boldsymbol{g}_2 \\ \boldsymbol{g}_3 \end{bmatrix} = \begin{bmatrix} \boldsymbol{c}_8 \\ \boldsymbol{c}_4 \\ \boldsymbol{c}_2 \\ \boldsymbol{c}_1 \end{bmatrix} = \left[\begin{array}{c|c} 110 & 1000 \\ 011 & 0100 \\ 111 & 0010 \\ 101 & 0001 \end{array}\right]
$$

MATLAB Experiment 3.1

The MATLAB function `encode` encodes a data block into a linear block code specified by the generator `G`. The following script is for the (7,4) code.

```
>> n = 7; k = 4;                           % (7,4) code
>> % generator matrix of the (7,4) code
>> G = [1 1 0 1 0 0 0;0 1 1 0 1 0 0;1 1 1 0 0 1 0;
       1 0 1 0 0 0 1];
>> m = [1 0 0 1];                          % message word
>> c = encode(m,n,k,'linear',G);           % encoding
>> c'
ans =
       0  1  1  1  0  0  1
```

The generated codeword is $\boldsymbol{c}_9$ (see Example 3.1).

A matrix closely associated with the generator $\boldsymbol{G}$ is the parity-check matrix $\boldsymbol{H}$:

$$\boldsymbol{H} = \begin{bmatrix} h_{0,0} & h_{0,1} & \cdots & h_{0,n-1} \\ h_{1,0} & h_{1,1} & \cdots & h_{1,n-1} \\ \vdots & & & \\ h_{n-k,0} & h_{n-k,1} & \cdots & h_{n-k,n-1} \end{bmatrix} \qquad (3.5)$$

where $\boldsymbol{G}$ and $\boldsymbol{H}$ satisfy the following:

$$\boldsymbol{G} \cdot \boldsymbol{H}^{\mathrm{T}} = \boldsymbol{0} \qquad (3.6)$$

The superscript T in (3.6) denotes the transpose of the matrix. Based on (3.6) $\boldsymbol{H}$ is related to codeword $\boldsymbol{c}$ as follows:

$$\boldsymbol{c} \cdot \boldsymbol{H}^{\mathrm{T}} = \boldsymbol{m} \cdot \boldsymbol{G} \cdot \boldsymbol{H}^{\mathrm{T}} = \boldsymbol{0} \qquad (3.7)$$

For systematic codes, $\boldsymbol{H}$ takes the form of:

$$\boldsymbol{H} = \left[\boldsymbol{I}_{n-k} \mid \boldsymbol{P}^{T} \right] \qquad (3.8)$$

Similar to the generator matrix, the parity-check matrix alone can also completely specify a linear block code.

MATLAB Experiment 3.2

MATLAB has a dedicated function `gen2par` to convert between the parity matrix and the generator matrix, saving us some time to solve (3.6) manually.

```
>> % generator matrix of the (7,4) code
>> G = [1 1 0 1 0 0 0;0 1 1 0 1 0 0;1 1 1 0 0 1 0;
1 0 1 0 0 0 1];
>> H = gen2par(G)
H =
       1  0  0  1  0  1  1
       0  1  0  1  1  1  0
       0  0  1  0  1  1  1
```

where `H` is the parity-check matrix of the (7,4) code.

3.1.1.4 Encoding with *G* and *H*

With the generator matrix ***G***, block encoding can be accomplished through matrix operation [see (3.3)]. For systematic codes, encoding can be further simplified. Notice that a codeword in systematic form can be expressed as:

$$\boldsymbol{c} = \left(\mu_0 \mu_1 \cdots \mu_{n-k-1} m_0 m_1 \cdots m_{k-1}\right) \tag{3.9}$$

where μ_0, μ_1, $\cdots$, μ_{n-k-1} are the parity bits, and m_0, m_1, $\cdots$, m_{k-1} are the message bits. Consequently only the $n-k$ parity bits need to be computed. If we combine (3.3), (3.4), and (3.9), we have the following equation for calculation of the parity:

$$\begin{aligned}
\mu_0 &= m_0 p_{0,0} + m_1 p_{1,0} + \cdots + m_{k-1} p_{k-1,0} \\
\mu_1 &= m_0 p_{0,1} + m_1 p_{1,1} + \cdots + m_{k-1} p_{k-1,1} \\
\mu_2 &= m_0 p_{0,2} + m_1 p_{1,2} + \cdots + m_{k-1} p_{k-1,2} \\
&\vdots \\
\mu_{n-k-1} &= m_0 p_{0,n-k-1} + m_1 p_{1,n-k-1} + \cdots + m_{k-1} p_{k-1,n-k-1}
\end{aligned} \tag{3.10}$$

Equation (3.10) may also be derived from the parity-check matrix $\boldsymbol{H}$. Substituting (3.8) and (3.9) into (3.7), and solving the equation for μ_i, we obtain the same equation, (3.10).

If expressed in matrix form as follows, (3.10) is more compact:

$$\boldsymbol{\mu} = \boldsymbol{m} \cdot \boldsymbol{P} \tag{3.11}$$

where $\boldsymbol{\mu} = (\mu_0\ \mu_1 \cdots \mu_{n-k-1})$ is the *parity vector*.

3.1.2 Decoding Methods

3.1.2.1 General Description

Let $\boldsymbol{r}$ denote the received word corresponding to the transmitted codeword $\boldsymbol{c}$. Due to the noise existing in the channel, $\boldsymbol{r}$ may not be identical to $\boldsymbol{c}$. Instead, $\boldsymbol{r}$ will equal to $\boldsymbol{c}$ plus an error pattern or error vector $\boldsymbol{e} = (e_0\ e_1 \cdots e_{n-1})$ caused by the channel noise:

$$\boldsymbol{r} = \boldsymbol{c} + \boldsymbol{e} \tag{3.12}$$

where $e_i = 1$ if $r_i \neq c_i$ and $e_i = 0$ otherwise. The decoding task is to determine if the error pattern $\boldsymbol{e}$ is a zero vector (error detection), or find the error pattern $\boldsymbol{e}$ itself so that the original transmitted codeword can be recovered as follows (error correction):

$$\tilde{\boldsymbol{c}} = \boldsymbol{r} - \boldsymbol{e} \tag{3.13}$$

Detection of error(s) is relatively easy. If $\boldsymbol{r}$ is not identical to any of the codewords in C, we know that an error or errors have occurred (although we do not know where they occurred). However, we want to point out that even if $\boldsymbol{r}$ happens to be the same as one of the codewords, it does not necessarily mean that $\boldsymbol{r}$ is free of error. This can be explained using a simple example. Suppose that the codeword $\boldsymbol{c}$ = (1110010) of the (7,4) code is transmitted. If $\boldsymbol{e}$ = (0110100), then $\boldsymbol{r}$ = (1110010) + (0110100) = (1000110). Note that (1000110) is another codeword in C (see Example 3.1). When the decoder sees $\boldsymbol{r}$, it *legitimately* treats it as a valid codeword. In this case the error(s) is undetectable.

Compared with error detection, correction of error(s) is a much more difficult task. Usually there will be multiple codeword/error combinations that give the same received word. For instance, the codeword/error pairs

c = (1110010)/***e*** = (0110100) and ***c*** = (0100011)/***e*** = (1100101) produce the same received word ***r*** = (1000110).The actual transmitted codeword can only be one of them. It is the job of the decoder to determine which one is the true codeword. Based on the maximum-likelihood decoding principle, the codeword that makes ***r*** most likely should be regarded as the true codeword. For a BSC channel, it is the codeword closest in the Hamming distance to ***r***. The error pattern ***e*** in this case will have the smallest number of 1's. The maximum-likelihood decoding principle was explained in Chapter 1. Next we present a method for error correction.

MATLAB Experiment 3.3

The MATLAB function `decode` decodes linear block codes. Take the (7,4) code as an example. Suppose that we have received the word ***r*** = (1001110) containing one error in its fourth position. By running the following script, we will get the correct message word:

```
>> n = 7; k = 4;                    % (7,4) code
>> % generator
>> G = [1 1 0 1 0 0 0;0 1 1 0 1 0 0;1 1 1 0 0 1 0;
      1 0 1 0 0 0 1];
>> % received word
>> r = [1 0 0 1 1 1 0];
>> % decoding, ‘linear’ means linear block code
>> m = decode(r,n,k,’linear’,G);
>> m’                               % decoded message word
ans =
      0  1  1  0
```

3.1.2.2 Error Correction with Standard Array

Standard array decoding is a table look-up (LUT) decoding technique for block codes. For an (n, k) binary block code C, there exist 2^n possible n-tuples that the decoder may receive, 2^k of which are the codewords. The standard array decoding method first arranges all 2^n n-tuples in an array as follows:

1. List all 2^k codewords among the 2^n n-tuples on the first row, starting with the all-zero codeword 0.

$$\begin{array}{ccccl}
c_0=0 & c_1 & \cdots & c_{2^k-1} & \leftarrow \text{Codeword} \\
l_1 & l_1+c_1 & \cdots & l_1+c_{2^k-1} & \leftarrow \text{Coset} \\
\vdots & \vdots & & \vdots & \\
l_{2^{n-k}} & l_{2^{n-k}}+c_1 & \cdots & l_{2^{n-k}}+c_{2^k-1} & \leftarrow \text{Coset} \\
\uparrow & & & & \\
\text{Coset} & & & &
\end{array}$$

Figure 3.1 Standard array.

2. From the remaining n-tuples, choose one with the smallest number of 1's, and put it in the first position of the next row.
3. Add the chosen n-tuple to the codewords at the top of each column, and place the results in the corresponding positions of the row. The results are then removed from the remaining n-tuples.
4. Repeat steps 2 and 3 until all 2^n n-tuples have been used.

At the end of the process, we should have a LUT that looks like that shown in Figure 3.1. This $2^{n-k} \times 2^k$ LUT is called the *standard array* of the code C. Notice that $\boldsymbol{l}_1, \boldsymbol{l}_2, \cdots, \boldsymbol{l}_{2^{n-k}}$ form a subgroup of 2^n n-tuples. By definition all n-tuples on the same row as $\boldsymbol{l}_i$ are the cosets of $\boldsymbol{l}_i$ (see Chapter 2). These $\boldsymbol{l}$'s are called the *coset leaders*.

The coset leaders can be viewed as possible error patterns.[1] The n-tuples in one same column are possible received words corresponding to the codeword at the top. Consequently, upon receiving a word $\boldsymbol{r}$, all we do for decoding is find the coset in the standard array that is identical to $\boldsymbol{r}$, and the corresponding codeword at the top is the decoded codeword.

Example 3.3

Consider a simple (5,2) code with the generator matrix $\boldsymbol{G} = \begin{bmatrix} 10101 \\ 01011 \end{bmatrix}$. *The standard array of the code is constructed as follows:*

[1] In fact the coset leaders are correctable error patterns. There are also other error patterns, but they are just not correctable.

00000	01011	10101	11110	← Codewords
10000	11011	00101	01110	
01000	00011	11101	10110	
00100	01111	10001	11010	
00010	01001	10111	11100	
00001	01010	10100	11111	
11000	10011	01101	00110	
10010	11001	00111	01100	
↑				
Coset Leader				

The first row of the array consists of four codewords. On the second row, the coset leader is the n-tuple with the smallest number of 1's among the remaining n-tuples, which is (10000).[2] *Adding it to the codewords (01011), (10101), and (11110), respectively, we have (11011), (00101), and (01110) placed in the rest positions of the row. The other rows are constructed in a similar manner.*

Suppose the decoder receives **r** *= (01010), which contains an error in its fifth position (counting from the left). In the above standard array we find that the coset leader corresponding to* **r** *is (00001), and the codeword is (01011).*

3.1.2.3 Simplifying a Standard Array Using a Syndrome

The standard array may be simplified using something called a *syndrome*. Syndrome $\boldsymbol{S}$ is defined as the inner product of $\boldsymbol{r}$ and the transpose of the parity-check matrix $\boldsymbol{H}$:

$$\boldsymbol{S} = \left(S_0 S_1 S_2 \cdots S_{n-k-1}\right) \triangleq \boldsymbol{r} \cdot \boldsymbol{H}^{\mathrm{T}} \tag{3.14}$$

Substituting (3.12) into (3.14), we obtain:

$$\boldsymbol{S} = \boldsymbol{r} \cdot \boldsymbol{H}^{\mathrm{T}} = \left(\boldsymbol{c} + \boldsymbol{e}\right) \cdot \boldsymbol{H}^{\mathrm{T}} = \boldsymbol{c} \cdot \boldsymbol{H}^{\mathrm{T}} + \boldsymbol{e} \cdot \boldsymbol{H}^{\mathrm{T}} = \boldsymbol{e} \cdot \boldsymbol{H}^{\mathrm{T}} \tag{3.15}$$

[2] We may also choose other *n*-tuples like (00010) or (01000), but it should not affect the decoding result because they will be chosen eventually.

Example 3.4

The parity-check matrix of the (5,2) code is:

$$\boldsymbol{H} = \begin{bmatrix} 10100 \\ 01010 \\ 11001 \end{bmatrix}$$

The syndrome for $\boldsymbol{r}$ *= (01010) is calculated as:*

$$\boldsymbol{S} = \boldsymbol{r} \cdot \boldsymbol{H}^{\mathrm{T}} = (01010) \cdot \begin{bmatrix} 10100 \\ 01010 \\ 11001 \end{bmatrix}^{\mathrm{T}} = (001)$$

MATLAB Experiment 3.4

A MATLAB function `synd*` is included for syndrome calculation. The following script applies it to the above example:

```
>> H = [1 0 1 0 0; 0 1 0 1 0; 1 1 0 0 1];   % parity-check
                                            % matrix
>> r = [0 1 0 1 0];                         % received word
>> % 'h' indicates H is a parity-check matrix
>> S = synd(r,H,'h')
S =
       0  0  1
```

Reexamining the standard array, it is not difficult to find that all *n*-tuples on the same row have the same syndrome. (This is simply because each row is associated with the one same error pattern which is the coset leader.) As such, the entire row of *n*-tuples can be replaced by their syndrome (see Figure 3.2) and the array still serves the purpose. Decoding in this case is accomplished by finding the coset leader $\boldsymbol{l}$ with the same syndrome as the one calculated from $\boldsymbol{r}$, and correcting the error(s) as $\tilde{\boldsymbol{c}} = \boldsymbol{r} - \boldsymbol{l}$.

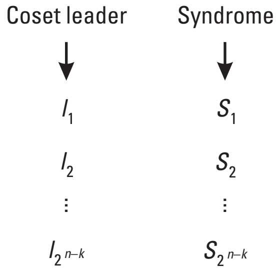


Figure 3.2 Syndrome-based standard array.

Example 3.5

The new standard array for the (5,2) code is:

00000	000
10000	101
01000	011
00100	100
00010	010
00001	001
11000	110
10010	111

The syndromes are computed using (3.15). From the array we see that the coset leader $\boldsymbol{l}$ *corresponding to the syndrome of* $\boldsymbol{r}$ = *01010 [= (001) as highlighted] is (00001). So* $\bar{\boldsymbol{c}} = \boldsymbol{r} - \boldsymbol{l}$ *= (01011).*

MATLAB Experiment 3.5

The MATLAB function `syndtable` in MATLAB Communications Toolbox returns the syndrome-based standard array. Here we use it to build the array for the (5,2) code:

```
>> G = [1 0 1 0 1;0 1 0 1 1];  % generator matrix
>> H = gen2par(G);              % convert to parity-check matrix
>> t = syndtable(H)             % construct the standard array
t =
       0  0  0  0  0
       0  0  0  0  1
       0  0  0  1  0
       0  1  0  0  0
```

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

Each row is a coset leader whose syndrome equals the row number − 1 (in decimal).

3.1.2.4 Elaboration on Syndrome Decoding

The syndrome decoding technique presented in the preceding section, in its essence, is to solve (3.15) for the error pattern $\boldsymbol{e}$. One issue with the approach is that (3.15) are indeterminate equations and, thus, will have multiple solutions. This is more evident after we write out (3.15) as follows:

$$\begin{aligned}
S_0 &= e_0h_{0,0} + e_1h_{1,0} + \cdots + e_{n-1}h_{n-1,0}\\
S_1 &= e_0h_{0,1} + e_1h_{1,1} + \cdots + e_{n-1}h_{n-1,1}\\
&\vdots\\
S_{n-k-1} &= e_0h_{0,n-k-1} + e_1h_{1,n-k-1} + \cdots e_{n-1}h_{n-1,n-k-1}
\end{aligned} \tag{3.16}$$

Equation (3.16) has n unknowns $e_0, e_1, \cdots, e_{n-1}$ but only $n - k$ equations. The standard array technique, by design, picks the error pattern with the fewest 1's as the true error pattern. According to Section 3.1.2.1, this error pattern is the most probable error pattern.

Syndromes are also used for error detection. A nonzero syndrome indicates error(s). Note, however, that a zero syndrome does not necessarily mean no error exists. As a matter fact, two possibilities can lead to a zero syndrome: $\boldsymbol{e}$ is a zero vector, or $\boldsymbol{e}$ is identical to a codeword. Errors of the latter kind are undetectable.

Combining (3.8) and (3.14), we have the following syndrome computation equation for systematic codes:

$$\begin{aligned}
S_0 &= r_0 + r_{n-k}p_{0,0} + r_{n-k+1}p_{1,0} + \cdots + r_{n-1}p_{k-1,0}\\
S_1 &= r_1 + r_{n-k}p_{0,1} + r_{n-k+1}p_{1,1} + \cdots + r_{n-1}p_{k-1,1}\\
&\vdots\\
S_{n-k-1} &= r_{n-k-1} + r_{n-k}p_{0,n-k-1} + r_{n-k+1}p_{1,n-k-1} + \cdots + r_{n-1}p_{k-1,n-k-1}
\end{aligned} \tag{3.17}$$

The corresponding combinational logic circuit is shown in Figure 3.3.

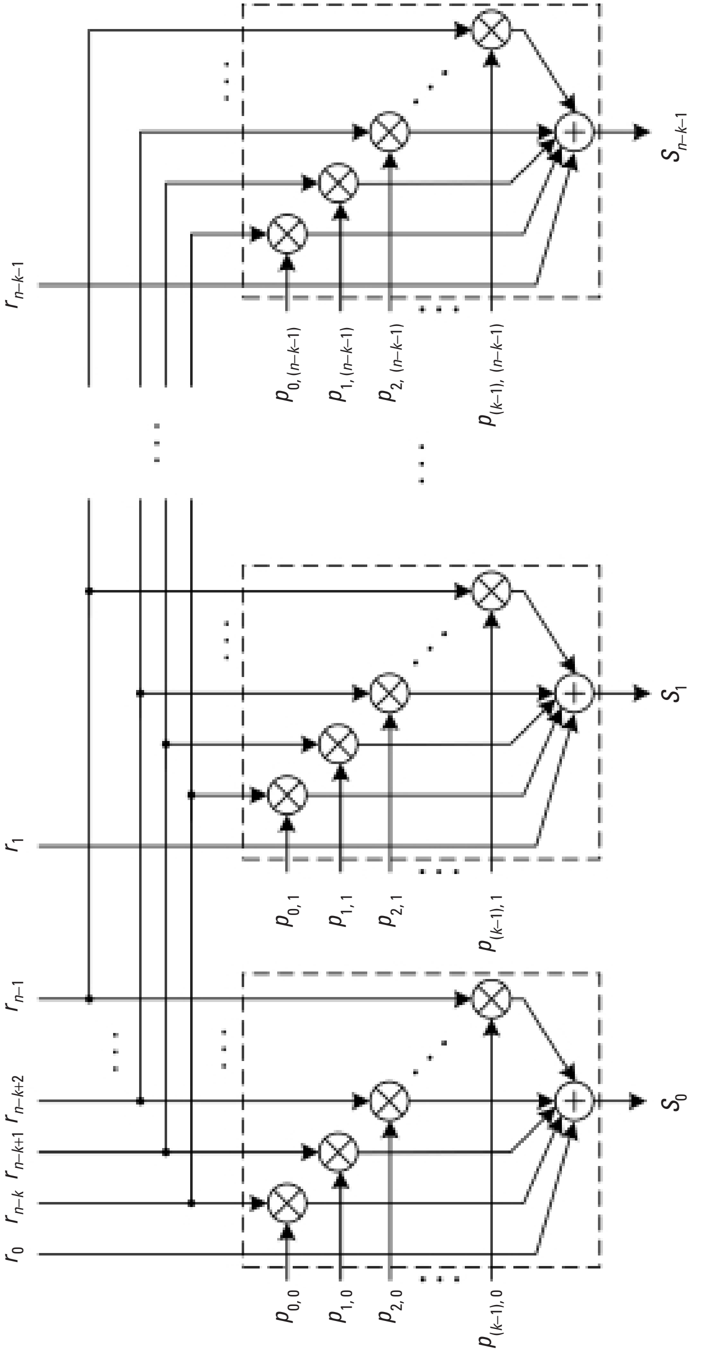


Figure 3.3 Syndrome computation circuit.

3.1.2.5 Erasure Decoding

Erasure is a special type of error whose location is known (whereas the location of a regular error is unknown). An erasure is produced when the receiver receives a signal that is not considered reliable. For example, when BPSK modulation is employed, the demodulator maps positive value → 0 and negative value → 1. If a received signal is too close to 0, the demodulator may consider the signal not reliable enough and assign an erasure to it (instead of 1 or 0). In this case, the discrete composite channel (see Chapter 1) will have a ternary output; that is, the channel output consists of 0, 1 and ×, where × denotes an erasure (Figure 3.4).

For binary linear codes, erasures can be corrected by following three steps [1, p. 229]:

1. Replace all erasures with 0's and decode to a codeword $\tilde{\boldsymbol{c}}^0$.
2. Replace all erasures with 1's and decode to a codeword $\tilde{\boldsymbol{c}}^1$.
3. Compare $\tilde{\boldsymbol{c}}^0$ and $\tilde{\boldsymbol{c}}^1$ with the received word, and choose as the decoded output the codeword closer to the received word $\boldsymbol{r}$ in the Hamming distance. Note that erasures are excluded when computing the Hamming distance.

Example 3.6

Consider the (7,4) linear block code. Assume a received word $\boldsymbol{r}$ *= (1 × 100 × 1), where × represents an erasure. We first replace the erasures with 0's and decode it to* $\tilde{\boldsymbol{c}}^0$ *= (1010001). Then we replace the erasures with 1's and decode it to* $\tilde{\boldsymbol{c}}^1$ *= (1110010).*

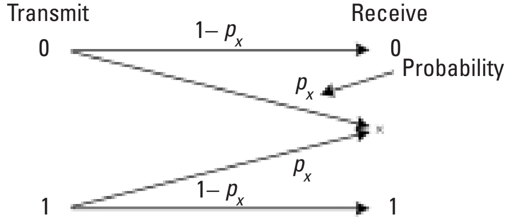


Figure 3.4 Binary erasure channel.

$$\begin{array}{rl@{\qquad\qquad}rl}
\boldsymbol{r} = & (1\,\boxed{\times}\,100\,\boxed{\times}\,1) & \boldsymbol{r} = & (1\,\boxed{\times}\,100\,\boxed{\times}\,1) \\
\tilde{\boldsymbol{c}}^0 = & (1\,\boxed{0}\,100\,\boxed{0}\,1) & \tilde{\boldsymbol{c}}^1 = & (1\,\boxed{1}\,100\,\boxed{1}\,0) \\
\hline
d_{\mathrm{H}} = & \qquad\qquad 0 & & \qquad\qquad\quad 1
\end{array}$$

$\tilde{c}^0$ is therefore selected as the decoded output.

3.1.3 Performance of Linear Block Codes

3.1.3.1 Minimum Distance

We stated in Chapter 1 that the minimum distance $d_{\min}$ determines the error detection and correction capability of a code. The minimum distance for binary linear block codes is derived next.

Equation (1.5) indicated that the Hamming distance between any two codewords in a code C is equal to the Hamming weight of the sum of the two codewords:

$$d_H(\boldsymbol{c}_i, \boldsymbol{c}_j) = w(\boldsymbol{c}_i + \boldsymbol{c}_j) \tag{3.18}$$

where $\boldsymbol{c}_i$, $\boldsymbol{c}_j \in C$ and $\boldsymbol{c}_i \neq \boldsymbol{c}_j$. Consequently we have:

$$d_{\min} = \min d_H(\boldsymbol{c}_i, \boldsymbol{c}_j) = \min w(\boldsymbol{c}_i + \boldsymbol{c}) \tag{3.19}$$

By the definition of linear block codes: $\boldsymbol{c}_i + \boldsymbol{c}_j = \boldsymbol{c}_k \in C$, (3.19) can be rewritten as:

$$d_{\min} = \min w(\boldsymbol{c}_k) \tag{3.20}$$

Equation (3.20) tells us that the minimum Hamming distance of a linear block code C equals the minimum weight (i.e., the minimum number of 1's) of its nonzero codewords. It can be shown that the minimum weight equals the minimum number of columns in the parity-check matrix of C that sums to $\mathbf{0}$ [2, p. 77]. For instance, the (7,4) code has a minimum distance of 3 because the smallest number of such columns is 3 (i.e., columns 4, 5, and 7 in the parity-check matrix).

MATLAB Experiment 3.6

Finding the minimum distance for a code involves counting the number of 1’s in every codeword of the code and comparing the numbers. We should leave this tedious work to a machine.

```
>> % generator matrix of the (7,4) linear block code
>> G = [1 1 0 1 0 0 0;0 1 1 0 1 0 0;1 1 1 0 0 1 0;
      1 0 1 0 0 0 1];
>> % compute the minimum distance
>> dmin = gfweight(G,‘gen’)      % ‘gen’: generator matrix
dmin =
        3
```

The function also works with the parity-check matrix ***H***.

3.1.3.2 Error Detection and Error Correction Capabilities

As explained earlier, errors are detectable if and only if they do not change the transmitted codeword into another codeword. Notice that the received word is the sum of the transmitted codeword and the error pattern [see (3.12)]. One situation that always meets this condition is that the number of errors is less than the code minimum distance $d_{\min}$. By the definition of minimum distance, any two codewords of the same code differ in at least $d_{\min}$ bits. The transmitted codeword in this case will by no means be turned into another codeword. Therefore, up to τ errors can be detected, where:

$$\tau = d_{\min} - 1 \tag{3.21}$$

This *random error detection capability* in (3.21) is guaranteed. However, in some cases a linear block code may be able to go beyond this. Consider a scenario in which the received word contains more than $d_{\min} - 1$ errors but falls outside of the codeword set. In this case the received word also differs from any of the codewords and the errors are detectable.

Now let us determine the probability of an error going undetected. We already know that undetectable error patterns are the ones identical to codewords. In a BSC with the crossover p_x, the probability of a particular pattern of i errors is $p_x^i \cdot (1 - p_x)^{n-i}$. There are a total of W_i cases in which a pattern of *i* errors can be identical to a codeword, where W_i is the weight distribution of the codeword, defined as the number of codewords with weight *i*. Summing all of the possibilities yields the probability of undetected error as follows:

$$\begin{aligned} P_{\mathrm{ud}} &= \underbrace{W_{d_{\min}} p_x^{d_{\min}} \cdot (1 - p_X)^{n-d_{\min}}}_{d_{\min}\ \text{error}} + \underbrace{W_{d_{\min}+1} \cdot p_x^{d_{\min}+1} \cdot (1 - p_X)^{n-d_{\min}-1}}_{d_{\min}+1\ \text{errors}} + \cdots + \underbrace{W_n \cdot p_x^n}_{n\ \text{errors}} \\ &= \sum\nolimits_{i=d_{\min}}^{n} W_i \cdot p_X^i \cdot (1 - p_X)^{n-i} \end{aligned} \tag{3.22}$$

The summation starts from $i = d_{\min}$ because no codeword has a weight of less than $d_{\min}$ [see (3.20)].

The probability of a detected error, P_{d}, is the probability that errors occur minus the probability that the errors are undetected, and is given by [3, p. 99]:

$$P_{\mathrm{d}} = \sum\nolimits_{i=1}^{n} \binom{n}{i} p_X^i (1 - p_X)^{n-i} - P_{\mathrm{ud}} \tag{3.23}$$

where the term $\binom{n}{i} p_X^i (1 - p_X)^{n-i}$ is the probability that *i* errors occur.

Turn now to error correction. The minimum distance of a block code $d_{\min}$ is a positive integer, and is either an odd or an even number. So, $d_{\min}$ can always be expressed as:

$$2t + 1 \le d_{\min} \le 2t + 2 \tag{3.24}$$

where *t* is some positive integer. We now show that a block code is capable of correcting up to *t* errors.

Let $\boldsymbol{c}$ be a codeword of *C* and $\boldsymbol{r}$ be the corresponding received word. Denote as $\boldsymbol{w}$ another codeword in *C*. The following relation is based on the simple triangle inequality:

$$d_{\mathrm{H}}(\boldsymbol{c}, \boldsymbol{r}) + d_{\mathrm{H}}(\boldsymbol{r}, \boldsymbol{w}) \ge d_{\mathrm{H}}(\boldsymbol{c}, \boldsymbol{w}) \tag{3.25}$$

This can also be easily seen in Figure 3.5.

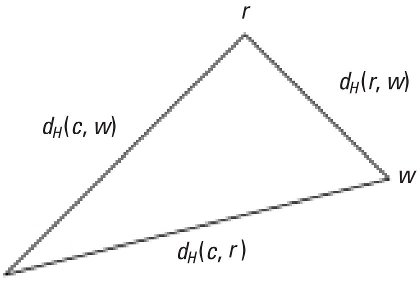


Figure 3.5 Triangle inequality.

Suppose the received word $\boldsymbol{r}$ contains t' errors; $\boldsymbol{r}$ then differs from $\boldsymbol{c}$ in t' positions: $d_{\mathrm{H}}(\boldsymbol{c}, \boldsymbol{r}) = t'$. Also, because $\boldsymbol{c}$ and $\boldsymbol{w}$ are both codewords, we have:

$$d_{\mathrm{H}}(\boldsymbol{c},\boldsymbol{w}) \geq d_{\min} \geq 2t+1 \tag{3.26}$$

Combining (3.25), (3.26), and $d_{\mathrm{H}}(\boldsymbol{c}, \boldsymbol{r}) = t'$, we obtain:

$$d_{\mathrm{H}}(\boldsymbol{r},\boldsymbol{w}) \geq 2t+1-t' \tag{3.27}$$

If $t' \leq t$, the preceding inequality becomes $d_{\mathrm{H}}(\boldsymbol{r}, \boldsymbol{w}) \geq t+1 > t$. Therefore, under the condition $t' \leq t$, the received word is closer in the Hamming distance to the true codeword $\boldsymbol{c}$ than to any other codewords, or in the maximum-likelihood terminology, the probability $P(\boldsymbol{r}|\boldsymbol{c})$ is greater than $P(\boldsymbol{r}|\boldsymbol{w})$. Thus, $\boldsymbol{r}$ is correctly decoded. This means that the block code is able to correct up to t errors. From (3.24) we have $t = \lfloor (d_{\min} - 1)/2 \rfloor$, where $\lfloor x \rfloor$ signifies the largest integer no greater than x. So, as a conclusion, a block code is able to correct up to t errors, where:

$$t = \lfloor (d_{\min} - 1)/2 \rfloor \tag{3.28}$$

The parameter t is often called the *random error correcting capability* of linear block codes. In fact, this error correction capability was obtained intuitively in Chapter 1. Similar to the situation in error detection, a linear block code may occasionally correct more errors (but there is no guarantee).

For a BSC with the crossover probability p_X, the upper bound of the probability of uncorrected error can be calculated using the union bound introduced in Chapter 1. As mentioned earlier, the probability of a pattern of i errors is $p_X^i \cdot (1 - p_X)^{n-1}$, and there are a total of $\binom{n}{i}$ distinct such cases. Because only up to t errors can be corrected for sure, the bound is simply the sum of all the probabilities that a received word contains more than t errors:

$$P_{\mathrm{uc}} \leq \sum\nolimits_{i=t+1}^{n} \binom{n}{i} p_X^i (1 - p_X)^{n-i} \tag{3.29}$$

Normally in (3.29) only the first few terms are significant.

To get a sense of how block codes can perform, we show in Figure 3.6 the BER versus SNR per bit (E_b/N_0, where E_b is the energy per bit and N_0 is the AWGN power spectral density) for several popular binary block codes.

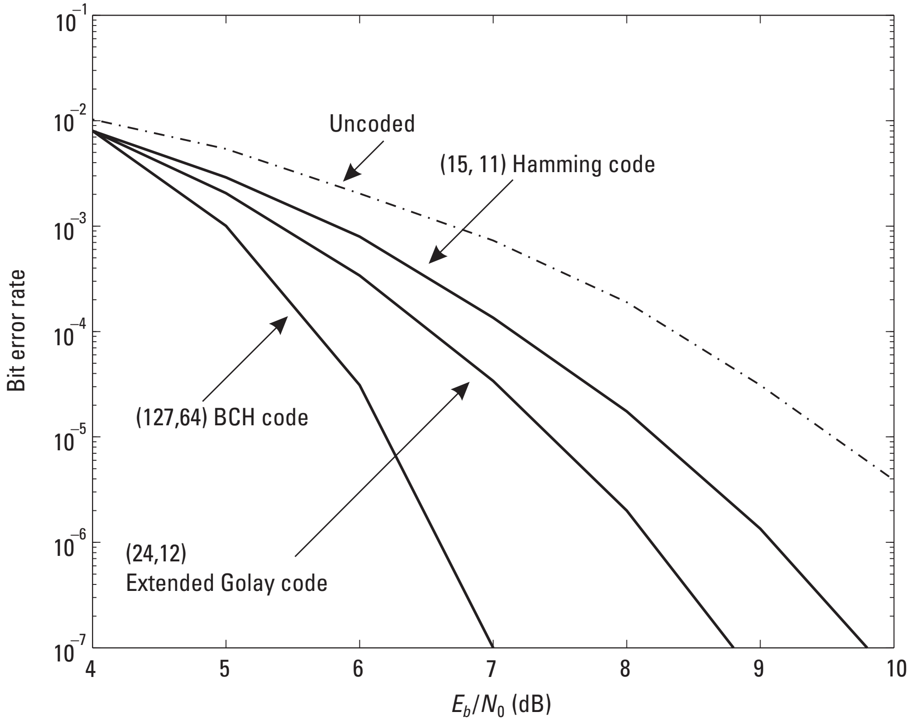


Figure 3.6 Error performances of block codes. (*After:* [4].)

MATLAB Experiment 3.7

This MATLAB experiment simulates the probability that the (7,4) block code fails to correct error(s) and compares the result with the bound and the performance of the uncoded BPSK. Three m-files are provided in the companion DVD: `hmsim.m*` [simulation of the (7,4) code], `hmtheory.m*` (bit error probability bound), and `bpsksim.m*` (simulation of uncoded BPSK).

It is worth noting that we may trade error correction for error detection. According to (3.21) and (3.28), one bit of error correction can be exchanged for two bits of error detection. Therefore, for example, a two-error correcting code may be used to detect four errors, or correct one error and at the same time detect two errors.

3.1.4 Encoder and Decoder Designs

3.1.4.1 Encoder

The most primeval block encoder is perhaps an LUT storing 2^k codewords of length n. For systematic codes, the width of the LUT can be reduced to $n - k$ because only the parity portion of a codeword needs to be looked up. Such an encoder is depicted in Figure 3.7. The ROM of size $2^k \times (n - k)$ is

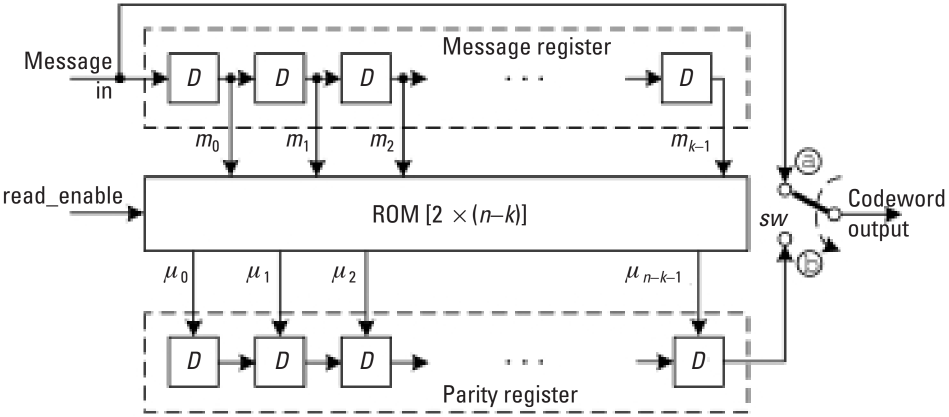


Figure 3.7 LUT-based systematic block encoder.

needed to hold the 2^k parity checks of length $n - k$. The `read_enable` signal is asserted when the message word $\boldsymbol{m} = (m_0\, m_1 \cdots m_{k-1})$ is completely moved into the register buffer. Upon the assertion of `read_enable`, the parity bits $\mu_0, \mu_1, \cdots \mu_{n-k-1}$ are loaded into the parity register. The switch is initially placed at position ⓐ to let message bits shift into the channel, and turned to ⓑ when `read_enable` becomes active. The codeword output is of the form $\boldsymbol{c} = (\mu_0\, \mu_1 \cdots \mu_{n-k-1} m_0\, m_1 \cdots m_{k-1})$.

The encoder can also be built using combinational logic as in Figure 3.8. The circuit block highlighted is based on (3.10) and has the same functionality as the ROM in Figure 3.7. The `sw` switches from ⓐ to ⓑ when all message bits are shifted into the registers. Note the resemblance between this encoder and the syndrome computation circuit in Figure 3.3.

3.1.4.2 Decoder

The syndrome decoding circuit is sketched in Figure 3.9. The ROM is arranged as the simplified standard array of the code. Once the received word is clocked into the receive register, the syndrome is computed and fed to the ROM as a read address. Immediately the `read_enable` signal is asserted to load the selected coset leader (i.e., the error pattern) from the ROM into the error register. The received bit and the error bit are shifted out of their respective registers simultaneously, one at a time. The two bits are then XORed to correct any errors.

3.1.5 Hamming Codes

Hamming codes are binary linear block codes that can correct one error or detect two errors. The code is named after its inventor R. Hamming. Although invented in 1950s, Hamming codes are still widely used today, especially in computer memory systems, for their simplicity and effectiveness.[3]

Hamming codes always have m ($m \geq 3$) parity bits and $2^m - m - 1$ message bits. Therefore, the code rate of Hamming codes is $(2^m - m - 1)/(2^m - 1)$. All Hamming codes have a minimum distance of $d_{\min} = 3$.

The parity-check matrix $\boldsymbol{H}$ of a Hamming code is constructed by taking all $2^m - 1$ nonzero binary vectors of length m as columns of the matrix.

[3] Hamming codes are among the easiest error control codes to construct.

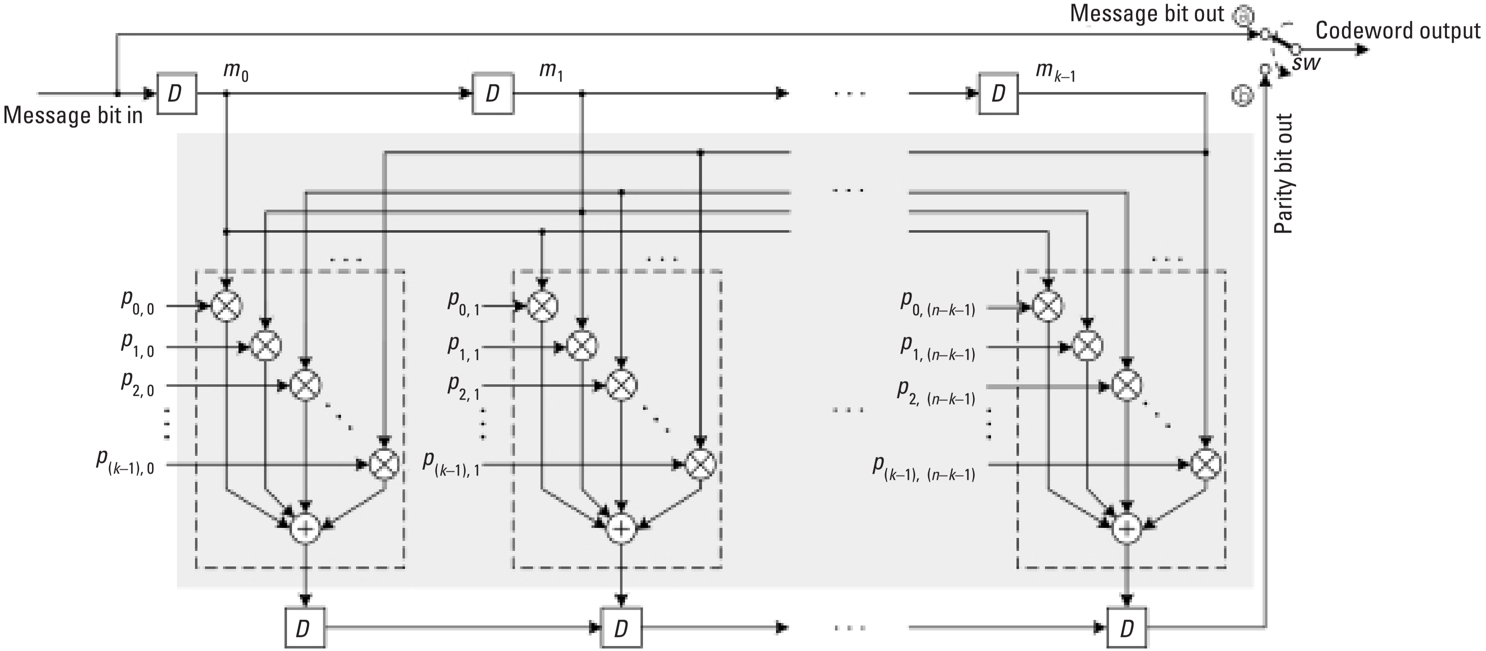


Figure 3.8 Combinational logic-based block encoder.

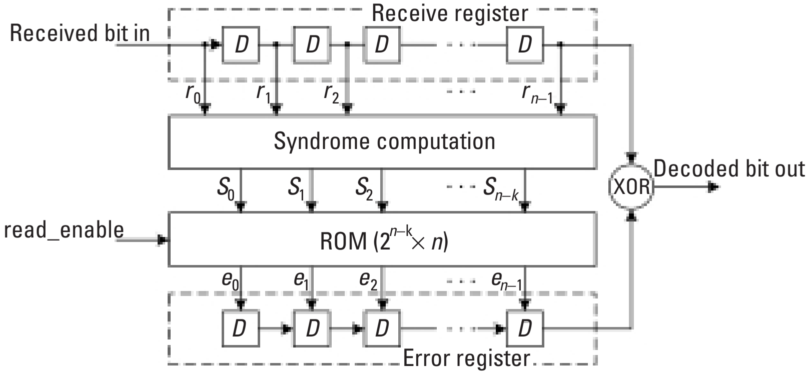


Figure 3.9 Block diagram of a syndrome-based decoder.

Example 3.7

Taking all seven nonzero three-bit binary vectors and putting them column by column to form a matrix generates the parity-check matrix of the (7,4) Hamming code:

$$\boldsymbol{H} = \begin{bmatrix} 0001111 \\ 0110011 \\ 1010101 \end{bmatrix}$$

The code has three parity bits.

Hamming codes can also be constructed by using an ad hoc method as follows.

Construction of Hamming Codes

1. All bit positions in the codeword that are powers of 2 (i.e., positions 1, 2, 4, 8, …) are for parity bits $\mu_{1,}$ μ_2, μ_3, μ_4, ….[4]
2. All of the rest of the positions (i.e., positions 3, 5, 7, 9, $\cdots$) are for message bits m_1, m_2, m_3, m_4, $\cdots$.
3. The parity bits are governed by the following rules:
   (a) μ_1 checks every other bit starting from where μ_1 is located (i.e., position 1).

[4] Note that the message bits and the parity bits are numbered from 1 instead of from 0 to facilitate our discussion.

(b) μ_2 checks every other 2 bits starting from where μ_2 is located (i.e., position 2).

(c) μ_3 checks every other 4 bits starting from where μ_3 is located (i.e., position 4).

(d) μ_4 checks every other 8 bits starting from where μ_4 is located (i.e., position 8).

. . .

4. Set a parity bit to 1 if the total number of 1's in the bits it checks (excluding itself) is odd, to 0 otherwise.

The method is pictorially illustrated in Table 3.2. The × sign signifies that the parity bit checks the bit in that position. For example, parity μ_2 checks the bits in positions 2, 3, 6, $\cdots$. So $\mu_2 = m_1 \oplus m_3 \oplus m_4 \oplus \cdots$. The highlighted area in the table corresponds to the (7,4) Hamming code. Note the resemblance between the × pattern in the area and the parity-check matrix of the code in Example 3.7.

Decoding of Hamming codes can be accomplished by using a standard array because Hamming codes are linear block codes. However, the way that Hamming codes are built entitles us to decode it more straightforwardly as follows:

Decoding of Hamming Codes

1. Based on Table 3.2, compute the parity checks of the received bits.
2. Bitwise add (i.e., bitwise XOR) the received parity bits to the computed parity bits.

Table 3.2
Construction of Hamming Codes

Bit Position		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	
Codeword Content		μ_1	μ_2	m_1	μ_3	m_2	m_3	m_4	μ_4	m_5	m_6	m_7	m_8	m_9	m_{10}	m_{11}	
Parity Bit	μ_1	×		×		×		×		×		×		×		×	⋮
	μ_2		×	×			×	×			×	×			×	×	
	μ_3				×	×	×	×					×	×	×	×	
	μ_4								×	×	×	×	×	×	×	×	
									⋮								

3. Sum the result, which gives the position where the received bit is in error.
4. Complement the bit in the position.

Example 3.8

Assume the (7,4) Hamming code. Suppose we have the following four message words to be sent: (1011), (1001), (0011), and (1011). Based on Table 3.2, the corresponding Hamming codewords are (0110011), (0011001), (1000011), and (1010011). The four received words are (0110011), (1011001), (1001010), and (0011001), which have 0, 1, 2, and 3 errors, respectively. Table 3 3 shows the decoding process and the decoding result.

As we can see, the (7,4) Hamming code is able to correct one error or detect up to two errors. Beyond this, the code can neither detect nor correct any errors.

Hamming codes can also be arranged into a systematic form. If we manipulate the parity-check matrix ***H*** of the (7,4) Hamming code by column

Table 3.3
Decoding Process for (7,4) Hamming Code

Transmitted Codeword	**(0110011)**	**(0011001)**	**(1000011)**	**(1010011)**
Case	0 Error	1 Error	2 Errors	3 Errors
Received Word *r**	(0110011)	(1011001)	(1001010)	(0011001)
Received Parity ($\mu_1 \mu_2 \mu_3$)	(010)	(101)	(101)	(001)
Computed Parity ($\mu_1' \mu_2' \mu_3'$)	(010)	(001)	(011)	(001)
($\mu_1 \mu_2 \mu_3$) $\oplus$ ($\mu_1' \mu_2' \mu_3'$)	(000)	(100)	(110)	(000)
Error Position	—	$= 1 + 0 \cdot 2^1 + 0 \cdot 2^2 = 1$	$= 1 + 1 \cdot 2^1 + 0 \cdot 2^2 = 3$	—
Decoded Word	(0110011)	(0011001)	(101100)	(0011001)
Error Detected?	—	Yes	Yes	No
Error Corrected?	—	Yes	No	No

* The highlighted bits are the erroneous bits.

permutation and elementary row operation, we obtain the equivalent parity-check matrix $\boldsymbol{H'}$ in systematic form as follows:[5]

$$\boldsymbol{H'} = \left[\begin{array}{c|c} 100 & 1011 \\ 010 & 1110 \\ 001 & 0111 \end{array}\right]$$

Compare the above $\boldsymbol{H'}$ with `H` obtained in MATLAB Experiment 3.2, we find that the code introduced in Example 3.1 is actually the (7,4) systematic Hamming code.

MATLAB Experiment 3.8

We can use MATLAB function `hammgen(m)` to produce systematic parity-check and generator matrices for a particular Hamming code specified by the parity-check length `m`.

```
>> m = 3;                   % parity-check length = 7 - 4
>> % produce parity-check and generator matrices
>> [H,G] = hammgen(m)
H =
       1  0  0  1  0  1  1
       0  1  0  1  1  1  0
       0  0  1  0  1  1  1
G =
       1  1  0  1  0  0  0
       0  1  1  0  1  0  0
       1  1  1  0  0  1  0
       1  0  1  0  0  0  1
```

[5] Let $\boldsymbol{G}$ (resp. $\boldsymbol{H}$) and $\boldsymbol{G'}$ (resp. $\boldsymbol{H'}$) be the generator (resp. parity-check) matrices of two codes. If they can be related merely via simple column operation and elementary row operation, the two codes are *equivalent,* and their distance structures are identical. An elementary row operation (a simple column operation) on a matrix is to sum two rows (columns) or scale a row (column) with a nonzero element.

Table 3.4 lists some of the Hamming codes with different (n,k).

Hamming codes can be expanded to *extended Hamming codes* by adding one extra bit. This bit increases the minimum distance of the extended Hamming code to $d_{\min} = 4$, giving the code the ability to detect up to three errors. The extra bit μ_x checks the parity of the whole codeword:

$$\mu_x = c_0 \oplus c_1 \oplus \cdots \oplus c_{n-1} \tag{3.30}$$

where $c_0, c_1, \cdots, c_{n-1}$ are the n bits of the original codeword. For instance, the codeword of the (7,4) Hamming code (0110011) is expanded to (01100110), (0011001) is expanded to (00110011), and so forth.

The parity-check bits alone in a Hamming code are not able to tell if the received word $\boldsymbol{r}$ contains no error or three errors, because in both cases the received parity is identical to the computed parity (see Example 3.8). However, the extra bit computed from $\boldsymbol{r}$ and the extra bit received in $\boldsymbol{r}$ will not match in the case of three errors. So, μ_x serves as an indication of whether $\boldsymbol{r}$ is error free or has three errors in it.

Example 3.9

We continue with Example 3.8. (01100110), (00110011), (10000111), and (10100110) are the codewords extended from the four Hamming code-

Table 3.4
Hamming Codes with Different (n, k)

n	k	$g(X)$
7	4	$1 + X + X^3$
15	11	$1 + X + X^4$
31	26	$1 + X^2 + X^5$
63	57	$1 + X + X^6$
127	120	$1 + X^3 + X^7$
255	247	$1 + X^2 + X^3 + X^4 + X^8$
511	502	$1 + X^4 + X^9$
1,023	1,013	$1 + X^3 + X^{10}$
2,047	2,036	$1 + X^2 + X^{11}$
4,095	4,083	$1 + X + X^4 + X^6 + X^{12}$

words (0110011), (0011001), (1000011), and (1010011), respectively. Let us assume that the same errors occur as in the previous example. Decoding of the extended Hamming code is illustrated in Table 3.5.

The received and computed parities match in both cases of no error and three errors, but, in the latter case, the calculated and received extra parity bits do not, that is, $\mu_x \neq \mu_x'$. So detection of up to three errors is now feasible with the extended code.

3.2 Cyclic Codes

As a subclass of linear block codes, cyclic codes are attractive for two reasons. First, due to their unique algebraic structure, cyclic encoding is easy to implement and efficient decoding algorithms can be devised. Second, the codes are

Table 3.5
Decoding Process for Example 3.9

Transmitted Codeword	**(01100110)**	**(00110011)**	**(10000111)**	**(10100110)**
Case	**0 Error**	**1 Error**	**2 Errors**	**3 Errors**
Received Word r*	(01100110)	(10110011)	(10010101)	(00110010)
Received Original Parity ($\mu_1 \mu_2 \mu_3$)	(010)	(101)	(101)	(001)
Computed Original Parity ($\mu_1' \mu_2' \mu_3'$)	(010)	(001)	(011)	(001)
($\mu_1 \mu_2 \mu_3$) $\oplus$ ($\mu_1' \mu_2' \mu_3'$)	(000)	(100)	(110)	(000)
Error Position	–	$= 1 + 0 \cdot 2^1 + 0 \cdot 2^2 = 1$	$= 1 + 1 \cdot 2^1 + 0 \cdot 2^2 = 3$	–
Received Extra Parity μ_x	0	0	1	0
Computed Extra Parity μ_x'	0	0	1	1
$\mu_x \oplus \mu_x'$	0	0	0	1
Decoded Word	(0110011)	(0011001)	(1011010)	(0011001)
Error Detected?	–	Yes	Yes	Yes
Error Corrected?	–	Yes	No	No

* The highlighted bits are the erroneous bits.

very effective in error detection. Needless to say, all properties associated with linear block codes apply equally to cyclic codes.

3.2.1 Basic Principles

3.2.1.1 Definition of Cyclic Codes

Let C be an (n, k) linear code. Then C is a cyclic code if every cyclic shift of a codeword in C is another codeword in C. For codeword $\boldsymbol{c} = (c_0\ c_1\ c_2\ \cdots\ c_{n-1})$, if a shift is performed l times, the shifted version of $\boldsymbol{c}$ beomes:

$$\boldsymbol{c}^{(l)} = (c_{n-l}\ c_{n-l+1}\ \cdots\ c_{n-1} c_0\ \cdots\ c_{n-l-1}) \tag{3.31}$$

The preceding definition of cyclic codes does not mean that all codewords can be generated from one codeword by cyclic shift; rather, it says that all codewords can be generated from one codeword by the combination of cyclic shift *and* addition. The former operation is due to cyclicality, and the latter comes from linearity.

Binary cyclic codewords are best described by a code polynomial over $GF(2)$ whose coefficients are the codeword bits. The code polynomial corresponding to the codeword $\boldsymbol{c} = (c_0\ c_1\ c_2\ \cdots\ c_{n-1})$ is:

$$c(X) = c_0 + c_1 X + c_2 X^2 + \cdots + c_{n-1} X^{n-1} \qquad (c_i \in \{0,1\}) \tag{3.32}$$

The code polynomial of the cyclically shifted version of $\boldsymbol{c}$, $\boldsymbol{c}^{(l)}$, is then expressed as:

$$c^{(l)}(X) = c_{n-l} + c_{n-l+1} X + \cdots + c_{n-1} X^{l-1} + c_0 X^l + \cdots + c_{n-l-1} X^{n-1} \tag{3.33}$$

$c^{(l)}(X)$ can be obtained from $c(X)$ through the following operation [3, p. 114]:

$$c^{(l)}(X) = X^l c(X) \bmod (X^n - 1) \tag{3.34}$$

This is easy to verify. For instance, the code polynomial of $\boldsymbol{c} = (1010001)$ is $c(X) = 1 + X^2 + X^6$. The code polynomial of $\boldsymbol{c}^{(2)} = (0110100)$ is obtained as:

$$\begin{array}{rl}
 & X \\
X^7 - 1 \big) & \overline{X^8 + X^4 + X^2} \\
 & \underline{X^8 + \qquad\qquad X} \\
 & \qquad X^4 + X^2 + X \leftarrow \text{code polynomial of } \boldsymbol{c}^{(2)}
\end{array}$$

3.2.1.2 Generator Polynomial and Parity-Check Polynomial

It can be shown [2, p. 140] that, for an (n, k) cyclic code C, there always exists a unique codeword with the code polynomial $g(X)$ as follows:

$$g(X) = g_0 + g_1X + g_2X^2 + \cdots + g_{n-k}X^{n-k} \tag{3.35}$$

where $g_0, g_{n-k} = 1$. By the definition of cyclic codes, $g(X), g^{(1)}(X), g^{(2)}(X), \cdots, g^{(k-1)}(X)$ are all code polynomials of the code C, and any linear combination of $g(X)$, $Xg(X)\text{mod } (X^n - 1)$, $X^2g(X)\text{mod } (X^n - 1)$, $\cdots$, $X^{k-1}\, g(X)\text{mod } (X^n - 1)$ is also a valid code polynomial of C (recall that all codewords can be generated from a single codeword by cyclic shift and addition). That is,

$$\begin{aligned} c(X) &= m_0g(X) + m_1Xg(X)\text{mod } (X^n - 1) + m_2X^2g(X)\text{mod } (X^n - 1) + \cdots \\ &\quad + m_{k-1}X^{k-1}g(X)\text{mod } (X^n - 1) \\ &= [m_0g(X) + m_1Xg(X) + m_2X^2g(X) + \cdots + m_{k-1}X^{k-1}g(X)] \text{ mod } (X^n - 1) \\ &= (m_0 + m_1X + m_2X^2 + \cdots + m_{k-1}X^{k-1})g(X) \text{ mod } (X^n - 1) \end{aligned} \tag{3.36}$$

is a codeword polynomial of C, where $m_0, m_1, m_2, \cdots, m_{k-1}, \in (0,1)$ are some coefficients. Notice that the degrees of $g(X)$, $Xg(X)$, $X^2g(X)$, $\cdots$, $X^{k-1}g(X)$ are all less than n. Therefore, the modulo operation in (3.36) can be omitted:

$$\begin{aligned} c(X) &= (m_0 + m_1X + m_2X^2 + \cdots + m_{k-1}X^{k-1}) \cdot g(X) \\ &= m(X) \cdot g(X) \end{aligned} \tag{3.37}$$

where $m(X) = m_0 + m_1X + m_2X^2 + \cdots + m_{k-1}X^{k-1}$. We see that $c(X)$ and $m(X)$ have a one-to-one correspondence. If $m_0, m_1, m_2, \cdots, m_{k-1}$ are chosen to be k message bits, then $c(X)$ becomes the codeword of $m(X)$. Comparing (3.37) with (3.3), we observe the analogy between $g(X)$ and the generator matrix $\boldsymbol{G}$. $g(X)$ is therefore called the *generator polynomial* of the code C, $m(X)$ and is the *message polynomial.*

It has been proven that the generator polynomial of a cyclic code $g(X)$ is a polynomial factor of $X^n - 1$. In fact, any factor of $X^n - 1$ can generate a cyclic code [2, p. 140]; it is just that the resultant code may or may not be a good code.

Example 3.10

Polynomial $1 + X + X^3$ *is a factor of* $X^7 - 1$ *[i.e.,* $X^7 - 1 = (1 + X + X^3)(1 + X + X^2 + X^4)$*]. We now take it as the generator polynomial of a (7,4) cyclic code. Let* $\boldsymbol{m} = (1011)$ *be the message word. The corresponding message polynomial is* $m(X) = 1 + X^2 + X^3$*. Multiplying* $m(X)$ *by* $g(X)$ *we obtain the following codeword:*

$$c(X) = 1 + X + X^2 + X^3 + X^4 + X^6 \text{ , or } \boldsymbol{c} = (1111101)$$

MATLAB Experiment 3.9

The MATLAB function `cyclpoly` in the Communications Toolbox produces all possible generator polynomials for an (n, k) cyclic code. Let us find the generator polynomial of the (7,4) code now:

```
>> n = 7; k = 4;          % (7,4) code
>> % to produce generator polynomial
>> % ‘all’ returns all generator polynomials for a given
>> % n and k
>> genpoly = cyclpoly(n,k,’all’)
genpoly =
             1  0  1  1
             1  1  0  1
```

The second row of `genpoly` is the generator polynomial used in Example 3.10, $g(X) = 1 + X + X^3$.

Similar to the parity-check matrix, a *parity-check polynomial* is associated with cyclic codes:

$$h(X) = h_0 + h_1 X + h_2 X^2 + \cdots + h_k X^k \tag{3.38}$$

where $h_0, h_k = 1$. Like the generator polynomial $g(X)$, the parity-check polynomial $h(X)$ also determines a code. The polynomials $h(X)$ and $g(X)$ are related to each other as follows:

$$g(X)h(X) = X^n - 1 \tag{3.39}$$

Similar to (3.7):

$$\begin{aligned} c(X)h(X) \bmod (X^n - 1) &= m(X)g(X)h(X) \bmod (X^n - 1) \\ &= m(X)(X^n - 1) \bmod (X^n - 1) = 0 \end{aligned} \tag{3.40}$$

Using (3.39), the parity-check polynomial of the (7,4) Hamming code can be obtained as follows:

$$h(X) = (X^7 - 1)/g(X) = (X^7 + 1)/(X^3 + X + 1) = 1 + X + X^2 + X^4$$

MATLAB Experiment 3.10

This book's companion DVD provides a MATLAB function `g2hpoly*` that produces a generator polynomial given a parity-check polynomial and vice versa.

```
>> g = [1 1 0 1];                   % generator poly.
>> h = g2hpoly(7,g)
>> h =                              % parity-check poly.
        1   1   1   0   1
```

3.2.1.3 Generator Matrix and Parity-Check Matrix

As a subclass of linear block codes, cyclic codes can also be represented by their generator matrices or parity-check matrices. Based on (3.37) we have:

$$\begin{aligned} c(X) &= m_0 g(X) + m_1 X g(X) + m_2 X^2 g(X) + \cdots + m_{k-1} X^{k-1} g(X) \\ &= (m_0\ m_1\ m_2 \cdots m_{k-1}) \cdot [g(X)\ Xg(X)\ X^2 g(X) \cdots X^{k-1} g(X)]^{\mathrm{T}} \end{aligned} \tag{3.41}$$

Expressing (3.41) in matrix form, we obtain the following equation, which is identical to (3.3):

$$\boldsymbol{c} = \boldsymbol{m} \cdot \boldsymbol{G} \tag{3.42}$$

where $\boldsymbol{m} = (m_0\ m_1\ m_2\ \cdots\ m_{k-1})$, and the $k \times n$ generator matrix $\boldsymbol{G}$ is:

$$\boldsymbol{G} = \begin{bmatrix} g_0 & g_1 & g_2 & \cdot & \cdot & \cdot & \cdot & \cdot & g_{n-k} & 0 & 0 & 0 & \cdot & \cdot & 0 \\ 0 & g_0 & g_1 & g_2 & \cdot & \cdot & \cdot & \cdot & \cdot & g_{n-k} & 0 & 0 & \cdot & \cdot & 0 \\ 0 & 0 & g_0 & g_1 & g_2 & \cdot & \cdot & \cdot & \cdot & \cdot & g_{n-k} & 0 & \cdot & \cdot & 0 \\ \vdots \\ 0 & 0 & \cdot & \cdot & \cdot & 0 & g_0 & g_1 & g_2 & \cdot & \cdot & \cdot & \cdot & \cdot & g_{n-k} \end{bmatrix} \tag{3.43}$$

It follows from (3.39) that $c(X)h(X) = m(X)g(X)h(X) = m(X)(X^n - 1) = m(X)X^n - m(X)$. The highest order in $m(X)$ is X^{k-1} and the lowest order in $m(X)\ X^n$ is X^n, therefore $c(X)h(X)$ should not contain the terms X^k, X^{k+1}, $\cdots$ X^{n-1}, or in other words, the coefficients of the terms are zero. Consequently, we have:

$$\sum\nolimits_{i=0}^{k} h_i c_{l-i} = 0 \quad (l = k, k+1, \cdots, n-1) \tag{3.44}$$

Rearranging (3.44) in matrix form gives us the following equation, which is exactly the same as (3.7):

$$\boldsymbol{c} \cdot \boldsymbol{H}^{\mathrm{T}} = \boldsymbol{0} \tag{3.45}$$

where $\boldsymbol{c} = c_0\ c_1\ c_2\ \cdots\ c_{n-1})$, and the $(n - k) \times n$ parity-check matrix $\boldsymbol{H}$ is:

$$\boldsymbol{H} = \begin{bmatrix} h_k & h_{k-1} & h_{k-2} & \cdot & \cdot & \cdot & \cdot & \cdot & h_0 & 0 & 0 & 0 & \cdot & \cdot & 0 \\ 0 & h_k & h_{k-1} & h_{k-2} & \cdot & \cdot & \cdot & \cdot & \cdot & h_0 & 0 & 0 & \cdot & \cdot & 0 \\ 0 & 0 & h_k & h_{k-1} & h_{k-2} & \cdot & \cdot & \cdot & \cdot & \cdot & h_0 & 0 & \cdot & \cdot & 0 \\ \vdots \\ 0 & 0 & \cdot & \cdot & \cdot & 0 & h_k & h_{k-1} & h_{k-2} & \cdot & \cdot & \cdot & \cdot & \cdot & h_0 \end{bmatrix} \tag{3.46}$$

Note that the two matrices $\boldsymbol{G}$ and $\boldsymbol{H}$ are Toeplitz with the property $\varepsilon_{i,j} = \varepsilon_{i\ -1,j-1}$, where $\varepsilon_{i,j}$ denotes the matrix element in the ith row and jth column. Also, there is a one-to-one correspondence between the generator (or parity-check) polynomial and the generator (or parity-check) matrix.

The two matrices for the (7,4) cyclic code are easily obtained:

$$\boldsymbol{G} = \begin{bmatrix} 1101000 \\ 0110100 \\ 0011010 \\ 0001101 \end{bmatrix} \text{ and } \boldsymbol{H} = \begin{bmatrix} 1011100 \\ 0101110 \\ 0010111 \end{bmatrix}$$

3.2.1.4 Cyclic Codes in Systematic Form

Using the generator polynomial $g(X)$ we can also produce cyclic codes in systematic form. Recall that a systematic codeword $\boldsymbol{c}$ is in the form of $\boldsymbol{c} = (\boldsymbol{\mu}|\boldsymbol{m}) = (\mu_0\,\mu_1 \cdots \mu_{n-k-1}|m_0\,m_1\,m_2 \cdots m_{k-1})$. Correspondingly its code polynomial is:

$$c(X) = \underbrace{\mu_0 + \mu_1 X + \cdots + \mu_{n-k-1}X^{n-k-1}}_{\mu(X)} + \underbrace{m_0 X^{n-k} + m_1 X^{n-k+1} + \cdots + m_{k-1}X^{n-1}}_{m(X)} \tag{3.47}$$

where $\mu(X)$ is termed the *parity polynomial.* As long as we find $\mu(X)$, we obtain the codeword polynomial in systematic form by appending $m(X)$ to $\mu(X)$. Next we show that $\mu(X)$, as calculated next, is the parity polynomial:

$$\mu(X) = X^{n-k}m(X) \bmod g(X) \tag{3.48}$$

Rewrite (3.48) as:

$$X^{n-k}m(X) = q(X)g(X) + \mu(X) \tag{3.49}$$

where $q(X)$ is the quotient. Adding $\mu(X)$ to both sides of (3.49), we obtain:

$$\mu(X) + X^{n-k}m(X) = q(X)g(X) \tag{3.50}$$

We realize that $\mu(X) + X^{n-k}m(X)$ has to be a codeword, because $q(X)$ $g(X)$ is a codeword [according to (3.37)]. Writing out $\mu(X) + X^{n-k}m(X)$, we have:

$$\begin{aligned} \mu(X) + X^{n-k}m(X) &= \mu_0 + \mu_1 X + \cdots + \mu_{n-k-1}X^{n-k-1} \\ &+ m_0 X^{n-k} + m_1 X^{n-k+1} + \cdots + m_{k-1}X^{n-1} \end{aligned} \tag{3.51}$$

which corresponds to the systematic codeword $\boldsymbol{c} = (\mu_0\ \mu_1\ \cdots\ \mu_{n-k-1}\ m_0\ m_1\ \cdots\ m_{k-1})$.

For the generator and parity-check in matrix form, their systematic representations can be obtained through column permutation and elementary row operation. Applying such manipulation to matrices $\boldsymbol{G}$ and $\boldsymbol{H}$ of the (7,4) cyclic code, we have their systematic forms as follows:

$$\boldsymbol{G} = \left[\begin{array}{c|c} 110 & 1000 \\ 011 & 0100 \\ 111 & 0010 \\ 101 & 0001 \end{array}\right] \text{ and } \boldsymbol{H} = \left[\begin{array}{c|c} 100 & 1011 \\ 010 & 1110 \\ 001 & 0111 \end{array}\right]$$

The above $\boldsymbol{G}$ and $\boldsymbol{H}$ are completely identical to the `G` and `H` obtained in MATLAB Experiment 3.8, indicating that the (7,4) Hamming code is a cyclic code. In fact, it has been observed that all Hamming codes have cyclic equivalences [5, p. 184].

MATLAB Experiment 3.11

The function `cyclgen(n,g,opt)` built-in MATLAB produces a generator matrix and parity-check matrix for cyclic codes, given a codeword length of `n` and the generator polynomial `g`. The parameter `opt` takes the following values:

`opt = 'nonsys'`: the function produces the matrices in nonsystematic form.

`'system'`: the function produces the matrices in systematic form.

For the (7,4) cyclic code, we have:

```
>> n = 7;                         % code length n = 7
>> g = [1 1 0 1];                 % generator polynomial
>> [H,G] = cyclgen(n,g,'nonsys')  % nonsystematic G and H
H =
     1  0  1  1  1  0  0
     0  1  0  1  1  1  0
     0  0  1  0  1  1  1
G =
     1  1  0  1  0  0  0
     0  1  1  0  1  0  0
     0  0  1  1  0  1  0
     0  0  0  1  1  0  1
```

or in systematic form:

```
>> [H,G] = cyclgen(n,g,'system')        % systematic G and H
H =
      1  0  0  1  0  1  1
      0  1  0  1  1  1  0
      0  0  1  0  1  1  1
G =
      1  1  0  1  0  0  0
      0  1  1  0  1  0  0
      1  1  1  0  0  1  0
      1  0  1  0  0  0  1
```

3.2.2 Shift Register–Based Encoder and Decoder

Considering the fact that every linear block code has a systematic equivalence [6, pp. 53–54], we focus on systematic encoders and decoders.

3.2.2.1 Cyclic Encoder

The key to systematic encoding of cyclic codes is the computation of parity polynomial $\mu(X)$ using (3.48). The equation is essentially a polynomial division operation, therefore we first consider a generic circuit to divide polynomial $b(X) = b_0 + b_1X + b_2X^2 + \cdots + b_nX^n$ by polynomial $a(X) = a_0 + a_1X + a_2X^2 + \cdots + a_mX^m$ ($n \geq m$). To carry out the division, we first subtract $(b_na_m^{-1})X^{n-m}a(X)$ from $b(X)$ so that the highest-order term in $b(X)$, b_mX^m, is eliminated and leave $b^{(1)}(X)$. Then we subtract $a_m^{-1}(b_{n-1} - b_n a_m^{-1})X^{n-m-1}a(X)$ from $b^{(1)}(X)$ so that the highest order $(b_{n-1} - b_n a_m^{-1})$ X^{n-1} in $b^{(1)}(X)$ is removed. The procedure is performed for a total of $n - m + 1$ times. Then the remaining $b^{(n-m+1)}(X)$ is the remainder of $b(X)/a(X)$. The division process can be realized by the circuit in Figure 3.10. The registers are initially reset to zeros. When the last term in $b(X)$, b_0, is shifted in, what is left in the registers is the remainder. The quotient is at the output of the circuit with the highest order appearing first. If $a(X)$ and $b(X)$ are over $GF(2)$, then $a_m^{-1} = a_m$ (a_m nonzero) and $-a_m = a_m$.

Notice that (3.48) computes the remainder of $X^{n-k}m(X)/g(X)$. Letting $a(X) = g(X)$, $b(X) = X^{n-k}m(X)$, we can design the cyclic encoder shown in Figure 3.11. Inputting $m(X)$ from the right side of the circuit corresponds to multiplication of $m(X)$ by X^{n-k}. The encoder operates as follows:

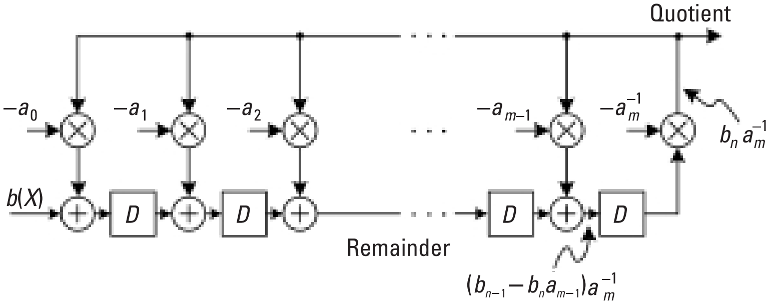


Figure 3.10 Polynomial division circuit.

1. Turn both switches sw_1 and sw_2 to ⓐ. The message polynomial is shifted into the encoder with the highest order first, and simultaneously into the channel. Once the lowest order of the message is shifted in, the registers contain the parity polynomial $\mu(X)$.
2. Turn sw_1 and sw_2 to ⓑ. The encoder is clocked $n - k - 1$ more times to shift the parity-check bits into the channel.

Systematic cyclic encoding may also be implemented using the parity-check polynomial $h(X)$. Because $h_k = 1$ [see (3.38)], we can rewrite (3.44) as:

$$c_{l-k} = h_0 c_l + h_1 c_{l-1} + \cdots + h_{k-1} c_{l-k+1} = \sum_{i=0}^{k-1} h_i c_{l-i} \quad (l = k, k+1, \cdots, n-1) \tag{3.52}$$

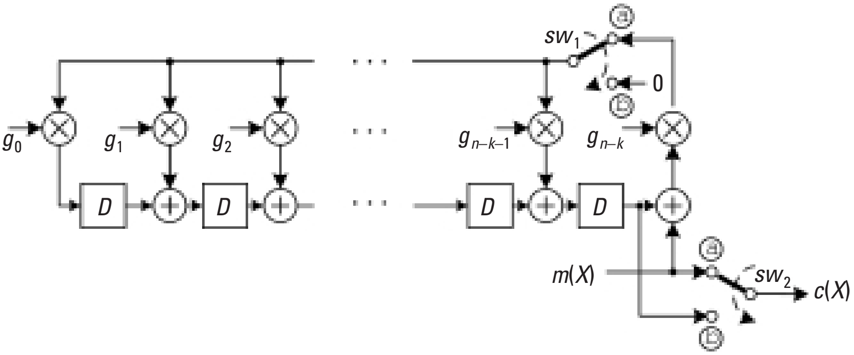


Figure 3.11 Systematic cyclic encoder using a generator polynomial.

Notice that $c_{n-k} = m_0$, $c_{n-k+1} = m_1$, $\cdots$, $c_{n-1} = m_{k-1}$ are already known. The parity checks $\mu_0 = c_0$, $\mu_1 = c_1$, $\cdots$, $\mu_{n-k-1} = c_{n-k-1}$ can then be computed using (3.52) starting from $l = n - 1$. Figure 3.12 is the schematic of the encoder. The circuit consists of the following four operational steps:

1. First switch sw_1 is closed and switch sw_2 open. The message $m(X) = m_0 + m_1X + m_2X^2 + \cdots + m_{k-1}X^{k-1}$ is shifted into both the register and the channel, starting with the highest order. At the end of k clocks, the register line contains the entire message.
2. Then sw_1 is turned open and sw_2 closed. The first parity-check bit $\mu_{n-k-1} = c_{n-k-1} = \Sigma_{i=0}^{k-1} h_i\, c_{n-i-1}$ is generated and appears at point $\textcircled{\Sigma}$.
3. When the next clock arrives, the register is shifted once more and the second parity-check bit $\mu_{n-k-2} = c_{n-k-2} = \Sigma_{i=0}^{k-1} h_i c_{n-i-2}$ is produced and appears at point $\textcircled{\Sigma}$.
4. The computation continues until all $n - k$ parity-check bits have been formed.

Example 3.11

Figure 3.13 shows the encoder for the (7,4) code with the generator $g(X) = 1 + X + X^3$. The encoding process for the message polynomial $m(X) = 1 + X^3$ is listed in Table 3.6.

The encoded output is $c(X) = X + X^2 + X^3 + X^6$.

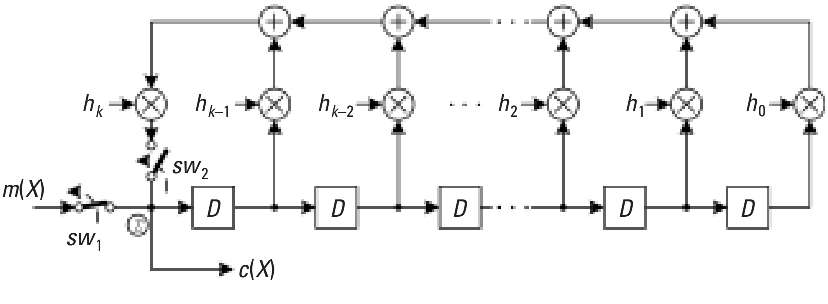


Figure 3.12 Systematic cyclic encoder using parity-check polynomial.

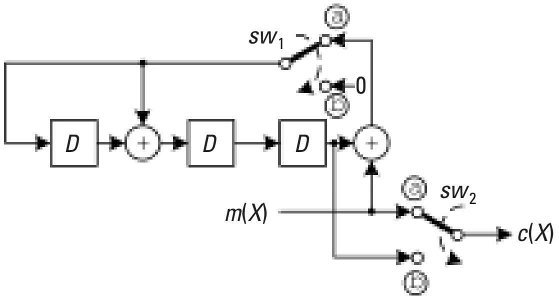


Figure 3.13 Example cyclic encoder.

MATLAB Experiment 3.12

Given a generator polynomial, the function `encode` used in MATLAB Experiment 3.1 can also encode a message into the desired cyclic code. The following script encodes $\boldsymbol{m}$ = (001) into the codeword $\boldsymbol{c}$ = (0100011) of the (7,4) cyclic code.

```
>> n = 7; k = 4;                        % (7,4) code
>> g = [1 1 0 1];                       % generator polynomial
>> m = [0 0 1 1];                       % message
>> c = encode(m,n,k,’cyclic’,g);        % ‘cyclic’ encoding
>> c’                                   % codeword
ans =
      0  1  0  0  0  1  1
```

Table 3.6
Example Cyclic Encoding Process

Step	Message Input	sw_1	sw_2	Register	Output	Note
0				000		Initialization
1	1			110	1	
2	0	ⓐ	ⓐ	011	0	
3	0			111	0	
4(a)	1			011	1	
4(b)				011	1	
5		ⓑ	ⓑ	001	1	
6				000	0	Encoding complete

3.2.2.2 Cyclic Decoder

Due to the cyclic nature of the codes, the syndrome computation for cyclic codes can be implemented in a much simpler manner with a shift register.

Let $r(X) = r_0 + r_1X^1 + r_2X^2 + \cdots + r_{n-1}X^{n-1}$ be the received polynomial corresponding to the received word $\boldsymbol{r} = (r_0\ r_1\ r_2\ \cdots\ \ r_{n-1})$. Then $r(X)$ can be expressed as:

$$r(X) = c(X) + e(X) = m(X)g(X) + e(X) \tag{3.53}$$

where $e(X) = e_0 + e_1X + e_2X + \cdots + e_{n-1}X^{n-1}$ is the error polynomial corresponding to the error pattern $\boldsymbol{e} = (e_0\ e_1\ e_2 \cdots\ e_{n-1})$. Computing $r(X)$ modulo-$g(X)$, we obtain a remainder $S(X)$:

$$S(X) = r(X) \bmod g(X) = [c(X) + e(X)] \bmod g(X) = e(X) \bmod g(X) \tag{3.54}$$

Notice that the polynomial $S(X)$ has the same functionality as the syndrome matrix $\boldsymbol{S}$: it equals zero if $r(X)$ contains no error, and nonzero otherwise. Therefore, $S(X)$ is called the *syndrome polynomial.*

The syndrome computation in (3.54) basically involves polynomial division; therefore, it can be done using a circuit similar to that shown in Figure 3.10. The architecture is sketched in Figure 3.14. Initially, the registers are reset to zero. After the received polynomial $r(X)$ has been totally shifted into the registers, the content of the registers forms the syndrome.

Example 3.12

The syndrome computation circuit for the (7,4) Hamming code is depicted in Figure 3.15. Table 3.7 provides a step-by-step list of how the circuit operates when it receives the polynomial r(X) = X + X² + X⁵ +X⁶.

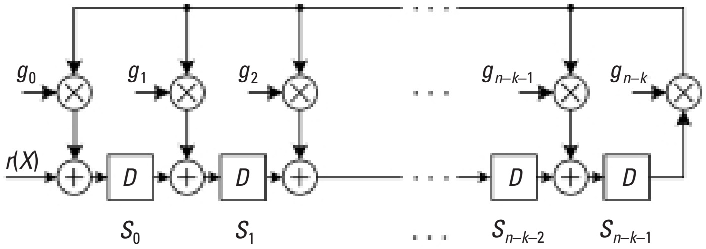


Figure 3.14 Syndrome computation circuit for cyclic codes.

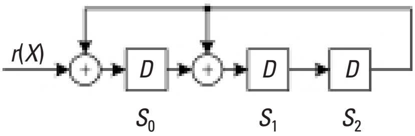


Figure 3.15 Syndrome computation circuit for cyclic Hamming code.

MATLAB Experiment 3.13

The syndrome computation function that we have provided can also be used to find the syndrome polynomial with the option set to `‘g’`:

```
>> r = [0 1 1 0 0 1 1];    % r(x) = x + x^2 + x^5 + x^6
>> g = [1 1 0 1];          % g(x) = 1 + x + x^3
>> syndpoly = synd(r,g,’g’) % ‘g’ means g is a generator poly.
syndpoly =
             0  0  1
```

The result corresponds to X^2 and is exactly the same as in the example.

Once the syndrome is computed, we can readily tell if the received polynomial $r(X)$ contains any error. For error correction, however, we need to find the error polynomial $e(X)$ from $S(X)$. Certainly we can apply the simplified standard array decoding method introduced earlier to cyclic codes, but the decoder proposed by Meggitt is more efficient [7].

The basic idea of the Meggitt decoder is to decode only the highest position in $r(X)$ and its cyclically shifted versions. After $r(X)$ is shifted

Table 3.7
Operation Process of Example Syndrome Computational Circuit

Step	***r*(*X*)**	**Register**	**Note**
0		000	Initialization
1	1	100	
2	1	110	
3	0	011	
4	0	111	
5	1	001	
6	1	010	
7	0	001	$S(X) = X^2$

$n-1$ times, every bit of $r(X)$ is in that position once and has the chance to be decoded. The presence of error is determined by checking if the related syndrome corresponds to an error polynomial with a 1 in its highest order.

It has been shown that if the syndrome of $r(X)$ is $S(X)$, the syndrome of $r^{(i)}(X)$ is $S^{(i)}(X)$, where the superscript (i) denotes the cyclic shift by i times [3, p. 138]. That is to say, only the first syndrome needs to be computed from scratch.

The Meggitt decoding process is illustrated in Figure 3.16. If the highest position is found to be correct in the current version of the received polynomial $r^{(i)}(X)$, nothing is changed except that $r^{(i)}(X)$ and $S^{(i)}(X)$ are cyclically shifted once at the same time. If the highest position in $r^{(i)}(X)$ is found to be in error, the position is complemented (i.e., corrected) and the resultant polynomial $\hat{r}^{(i)}(X)$ is shifted. Note that the new syndrome in this case can no longer be obtained by shifting $S^{(i)}(X)$ [because $\hat{r}^{(i)}(X) \neq r^{(i)}(X)$]; rather it equals the shifted version of $S^{(i)}(X)$, that is, $S^{(i+1)}(X)$, plus one:

$$S_{\text{new}}^{(i+1)}(X) = \text{shift of } [S^{(i)}(X)] + 1 \tag{3.55}$$

The loop continues until all bits in $r(X)$ have been decoded.

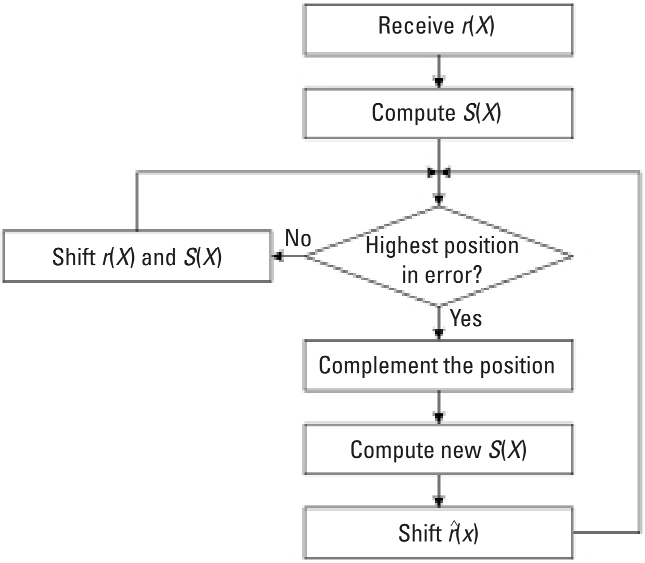


Figure 3.16 Flowchart of Meggitt decoding.

A Meggitt decoder consists of a buffer, a syndrome computation circuit, and a pattern recognizer, as depicted in Figure 3.17. Before the decoding starts, sw_1 is closed and sw_2 (including sw_2_a and sw_2_b) is open. The decoder first moves the message into the syndrome register and into the buffer simultaneously. When this is completed, the syndrome register contains the syndrome of $r(X)$. Then sw_1 opens and sw_2 closes. If the pattern recognizer detects that the current syndrome corresponds to an error polynomial with a 1 in its highest position, it produces a 1; otherwise, it produces a 0. Accordingly, the decoder does either of following:

1. Correct the highest position and cyclically shift the resultant received polynomial once. Also, at the same time, calculate the new syndrome.
2. Cyclically shift both the received polynomial and the syndrome once.

Either way, the pattern recognizer then checks the highest position in the new received polynomial and the decoding proceeds in a similar manner.

The Meggitt decoder can also be configured to move in $r(X)$ from the right side of the circuit. This is left as a problem for readers to solve.

Example 3.13

Let us design a Meggitt decoder for the (7,4) Hamming code. The key circuit is the pattern recognizer. Recall that Hamming codes can correct one error at most. As such, there is only one error polynomial that corresponds to the received polynomial having an error in the highest position, that is, e(X)

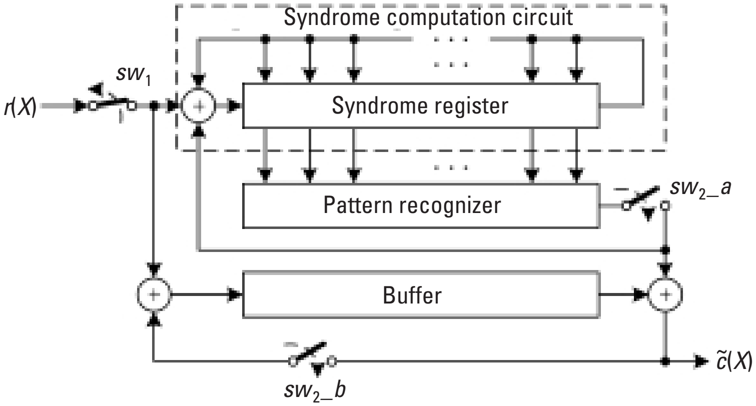


Figure 3.17 Block diagram of Meggitt decoder.

Table 3.8
Logic Function of the Example Pattern Recognizer

Input (= ***S***)	Output
(000)	0
(001)	0
(010)	0
(011)	0
(100)	0
(101)	1
(110)	0
(111)	0

$= 0 + 0X + 0X^2 + 0X^3 + 0X^4 + 0X^5 + X^6$. The syndrome corresponding to the error is calculated to be $S(X) = 1 + X$, or $S = (101)$ in vector form. Consequently, the logic function of the pattern recognizer is as shown in Table 3.8.

The circuit is simply a single three-input AND gate with the middle input inverted.

The complete decoder is depicted in Figure 3.18. Table 3.9 lists the operations of the decoder for decoding on $r(X) = X + X^2 + X^4 + X^5 + X^6$ with an error in its term X.

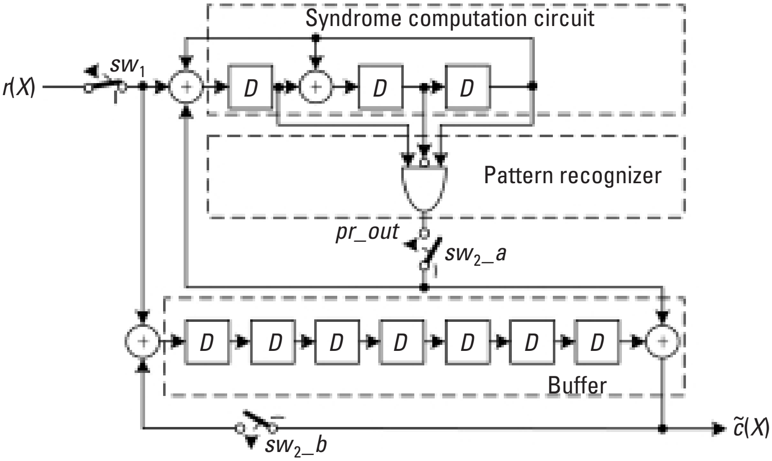


Figure 3.18 Meggitt decoder for Example 3.13.

Table 3.9
Decoding Process for Example 3.13

Step	***Input*** r_i	sw_1	sw_2	***pr_out***	**Syndrome Register**	**Buffer**	**Note**
0					000	0000000	
1	1			0	100	1000000	
2	1			0	110	1100000	
3	1			0	111	1110000	
4	0	Close	Open	0	101	0111000	
5	1			0	000	1011100	
6	1			0	100	1101110	Error bit moves in
7	0			0	010	0110111	
8				0	001	1011011	
9				0	110	1101101	
10				0	011	1110110	
11		Open	Close	0	111	0111011	
12				1	101	1011101	Corrected to 1011100
13				0	000	0101110	
14				0	000	0010111	← Decode output

An alternative decoder is the error trapping decoder of Kasami [8], which is a modified version of the Meggitt decoder. Because the error trapping decoder is most suited to Fire codes and those codes are not introduced in this book, we do not cover it. Interested readers will find details in [2, 3].

MATLAB Experiment 3.14

This book's companion DVD provides an `errpat*` function that lists syndrome patterns for all possible errors having erroneous MSBs. Together with the syndrome computation function `synd*` we can easily simulate the Meggitt decoder in MATLAB.

If the decoding method is not a concern, we can use the MATLAB function `decode` to decode a cyclic code. The following commands decode $\boldsymbol{r}$ = (0100001), which has an error in its sixth position. The original message is $\boldsymbol{m}$ = (0011).

```
>> n = 7; k = 4;                     % code parameters
>> g = [1 1 0 1];                    % generator polynomial
>> r = [0 1 0 0 0 0 1];              % received word
>> m = decode(r,n,k,'cyclic',g);     % decoding
>> m'
ans =
       0  0  1  1
```

3.2.3 Shortened Cyclic Codes and CRC

3.2.3.1 Shortening Cyclic Codes

The need for shortened codes arises when the original code parameters do not match the system design. Code shortening is accomplished by giving up some of the message bits in a code. To be more specific, suppose that we want to encode a message of length $k - L$:

$$m'(X) = m_0 + m_1X + \cdots + m_{k-L-1}X^{k-L}$$

but the selected code is an (n, k) code. In this case we may add L 0's to $m'(X)$ to make up the length k. The resultant message polynomial is:

$$m(X) = m_0 + m_1X + \cdots + m_{k-L-1}X^{k-L-1} + \underbrace{0 \cdot X^{k-L} + 0 \cdot X^{k-L+1} + \cdots + 0 \cdot X^{k-1}}_{L\ 0\text{'s}}$$

The corresponding codeword is[6]:

$$\begin{aligned} c(X) &= c_0 + c_1X + \cdots + c_{n-1}X^{n-1} \\ &= \mu_0 + \mu_1X + \cdots + \mu_{n-k-1}X^{n-k-1} + m_0X^{n-k} + m_1X^{n-k+1} + \cdots + m_{k-L-1}X^{n-L} \\ &\quad + \underbrace{0 \cdot X^{k-L} + 0 \cdot X^{k-L+1} + \cdots + 0 \cdot X^{n-1}}_{L\ 0\text{'s}} \end{aligned}$$

The L 0's in $c(X)$ are not transmitted. So the codeword actually transmitted is the shortened version of $c(X)$ [shortened from (n, k) to $(n - L, k - L)$]:

$$\begin{aligned} c'(X) &= c_0 + c_1X + \cdots + c_{n-L-1}X^{n-L-1} \\ &= \mu_0 + \mu_1X + \cdots + \mu_{n-k-1}X^{n-k} + m_0X^{n-k} + m_1X^{n-k+1} + \cdots + m_{k-L-1}X^{n-L-1} \end{aligned}$$

[6] Without loss of generality, we assume a systematic code.

By shortening the code we are actually using a subset of the original codewords. The error detection and correction capabilities of the shortened codes are at least the same as their original codes.

Shortened cyclic codes (sometimes also referred to as *polynomial codes*) are still linear block codes, but in general they are no longer cyclic. However, if we pad the L 0's to make up the length, the shift register–based encoder and the Meggitt decoder can still be used.

The L 0's apparently do not impact the syndrome computation. So it should also be possible for the Meggitt decoder to decode the shortened codes without adding the L 0's. Attention must be paid, however, to the proper alignment of the syndrome with the bit to be decoded. Let $r'(X) = r_0 + r_1X + r_2X^2 + \cdots r_{n-L-1}X^{n-L-1}$ be the received polynomial. To decode the first received bit r_{n-L}, we actually need the syndrome calculated as $X^{n-k+L}r'(X)\text{mod } g(X)$, rather than $r'(X)\text{mod } g(X)$. (Think about it; why?) The term X^{n-k+L} can be made up by either shifting the syndrome register $n - k + L$ times after the original syndrome is ready, or by modifying the syndrome computation circuit so that it directly computes $X^{n-k+L}r'(X)\text{mod } g(X)$. The first approach needs additional $n - k + L$ clocks to shift $S(X)$, and the second one introduces no extra delay.

Modification to the syndrome computation circuit for shortened codes is straightforward. We notice that $X^{n-k+L}r'(X)\text{mod } g(X)$ can be equivalently calculated as $r'(X)\rho(X)\text{mod } g(X)$, where $\rho(X)$ is is defined as:

$$\rho(X) = \rho_0 + \rho_1X + \cdots + \rho_{n-k-1}X^{n-k-1} \triangleq X^{n-k+L} \text{ mod } g(X)$$

As a result, the new circuit involves both polynomial multiplication $[r'(X)\rho(X)]$ and polynomial division $[r'(X)\rho(X)\text{mod } g(X)]$, as illustrated in Figure 3.19.

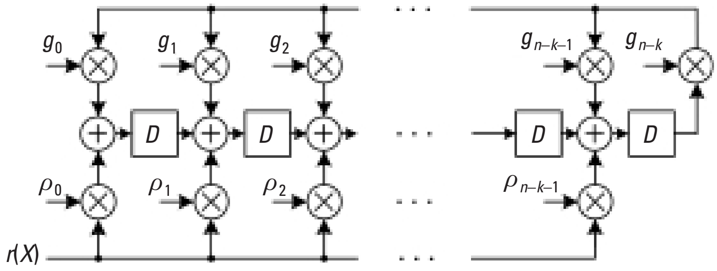


Figure 3.19 Modified syndrome computation circuit for shortened cyclic codes.

Example 3.14

Consider a (5,2) code shortened from the (7,4) Hamming code with the generator polynomial $g(X) = 1 + X + X^3$. The polynomial $\rho(X)$ is calculated to be:

$$\rho(X) = X^{n-k-L} \bmod g(X) = x^5 \bmod g(x) = 1 + X^2$$

The syndrome register in the Meggitt decoder is now revised as in Figure 3.20. The connection enclosed by the dashed line implements $r'(X)\rho(X)$. The switches are initially set to off and turned to on when $r'(X)$ is clocked in. The content of the resultant circuit is $r'(X)\rho(X) \bmod g(X)$.

3.2.3.2 Cyclic Redundancy Check

One of the most important shortened cyclic codes for error detection is the *cyclic redundancy check* (CRC) codes. By design, a CRC code has a shorter message length than cyclic codes. A CRC code is particularly good at detecting burst errors. So the codes are exclusively used together with ARQ for error detection in practice, especially in computer communications (although they do have some error correction capabilities).

CRC has generator polynomials of the form $g(X) = (1 + X)a(X)$, where $a(X)$ is a primitive polynomial over $GF(2)$. Different $a(X)$ results in different error detection capability of the code. Table 3.10 is a list of some CRC codes that have been standardized and used in practice.

Error detection with CRC is simple. We do not even need a syndrome. We calculate the parity check of the k received bits corresponding to the message bits in the codeword, and compare it with the received parity check. If they do not match, an error or errors have occurred.

CRC codes are evaluated in terms of their error pattern coverage, burst error detection capability, or probability of undetected error. The error pattern

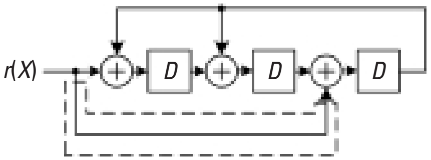


Figure 3.20 Syndrome computation circuit revised for shortened (5,2) code.

Table 3.10
Some Standardized CRC Codes

CRC Code	**Generator Polynomial $g(X)$**
CRC-8	$X^8 + X^2 + X + 1$
CRC-12	$X^{12} + X^{11} + X^3 + X^2 + X + 1$
CRC-ANSI	$X^{16} + X^{15} + X^2 + 1$
CRC-CCITT	$X^{16} + X^{12} + X^5 + 1$
CRC-SDLC	$X^{16} + X^{15} + X^{13} + X^7 + X^4 + X^2 + X + 1$
CRC-32	$X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X^2 + X + 1$

coverage of an (n, k) CRC code, η, is defined as the ratio of the number of noncodeword n-tuples to the total number of n-tuples:

$$\eta = \frac{(2^n - 2^k)}{2^n} = 1 - 2^{-(n-k)} \tag{3.56}$$

The measure reflects the probability that a received word, if in error, is not a codeword, or in other words, it is detectable. Obviously we want this parameter as close to 1 as possible.

An (n, k) CRC code has the following burst error detection capability [9, p. 189]:

1. Detect all error bursts of length $b \le n - k$. The length of an error burst is defined as the number of bits counted from the first error to the last, inclusive.
2. Detect the fraction of $1 - 2^{-(n-k-1)}$ of all bursts of length $b = n - k + 1$.
3. Detect the fraction of $1 - 2^{-(n-k)}$ of all bursts of length $b > n - k + 1$.

The probability of undetected error may be calculated from (3.22). For a BSC with small crossover probability and large code length, The probability approaches $2^{-(n-k)}$, independent of the channel quality [5, p. 175].

Example 3.15

Consider the CRC-CCITT, whose generator polynomial is:

$$
\begin{aligned}
g(X) &= X^{16} + X^{12} + X^5 + 1 \\
&= (X+1)\underbrace{(X^{15} + X^{14} + X^{13} + X^{12} + X^4 + X^3 + X^2 + X + 1)}_{a(X)}
\end{aligned}
$$

This is a (k + 16, k) code with the minimum distance $d_{min} = 4$. The code is able to correct a single error and at the same time detect two errors, or detect up to three errors. The codeword is formed by appending the 16-bit parity check to the original message. The encoder/detector is depicted in Figure 3.21. CRC-CCITT is widely used in many communications protocols such as X.25 and Bluetooth.

MATLAB Experiment 3.15

Our function `crcchk*` generates and checks CRC. Taking CRC-ANSI as an example, we assume a message word of ***m*** = (1110010010110010).

```
>> m = [1 1 1 0 0 1 0 0 1 0 1 1 0 0 1 0];
>> g = [1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1];    % gen. poly.
>> crc = crcchk(m,g)
crc =
      1 1 1 0 1 0 1 1 0 1 1 1 0 1 0 0
```

Appending the parity bits to the message, we have the complete codeword as (1110101101110100|0111001001011010110). If no error occurs in the transmission, we then have:

```
>> r = [1 1 1 0 1 0 1 1 0 1 1 1 0 1 0 0 1 1 1 0 0 1 0 0...
      1 0 1 1 0 0 1 0];
>> crc = crcchk(r,g)
crc =
      0
```

The outcome confirms that no error exists.

3.3 BCH Codes

BCH codes, named after their discoverers Bose, Chaudhuri, and Hocquenghen [10, 11], are undoubtedly the most important cyclic codes. The codes

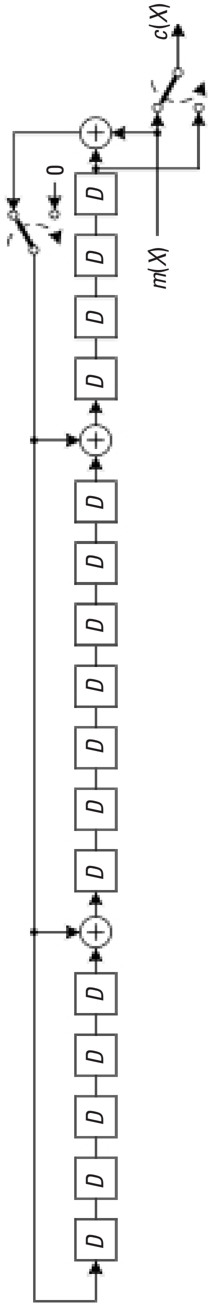


Figure 3.21 CRC-CCITT encoder/detector.

are among the best codes of moderate length. The biggest advantage of BCH codes is the existence of efficient decoding methods due to the special algebraic structure introduced in the codes. The two main types of interest are binary BCH codes and Reed-Solomon codes. In this section we present binary BCH codes. Reed-Solomon codes are the topic of the next chapter.

3.3.1 Introduction

3.3.1.1 Designing BCH Codes

A binary BCH code of length $n = 2^m - 1 (m \geq 3)$ is defined as a cyclic code whose code polynomials take $\alpha, \alpha^2, \alpha^3, \cdots, \alpha^{2t}$ as their roots, where α is the primitive element of $GF(2^m)$ and $t < 2^{m-1}$.

From the preceding statement it follows that the generator polynomial of a BCH code is the least common multiple (LCM) of the minimum polynomials of $\alpha^i (i = 1, 2, \cdots, 2t)$:

$$g(X) = \mathrm{LCM}[\phi_1(X), \phi_2(X), \cdots, \phi_{2t}(X)] \tag{3.57}$$

where $\phi_i(X)$ represents the minimum polynomial of α^i.

The minimal polynomials $\phi_1(X), \phi_2(X), \cdots, \phi_{2t}(X)$ have an interesting property: for every $\phi_i(X)$, where i is an even number, there exists $\phi_{i'}(X) = \phi_i(X)$, where i' is an odd number [2, p. 195]. Based on this finding, (3.57) can be simplified as follows:

$$g(X) = \mathrm{LCM}[\phi_1(X), \phi_3(X), \cdots, \phi_{2t-1}(X)] \tag{3.58}$$

Because each minimal polynomial is of degree m or less, the degree of $g(X)$ is at most mt, and so is the number of parity-check bits, that is, $n - k \leq mt$.[7]

Table 3.11 lists the parameters and generator polynomials of some BCH codes.

The binary BCH codes defined in Table 3.11 are *primitive BCH codes* because they are constructed using a primitive element of $GF(2^m)$.

[7] If t is small, $n - k = mt$ [2, 3].

Table 3.11
Some BCH Codes

n	*k*	*t*	**Generator Polynomial**
15	7	2	$1 + X^4 + X^6 + X^7 + X^8$
15	5	3	$1 + X + X^2 + X^4 + X^5 + X^8 + X^{10}$
31	21	2	$1 + X^3 + X^5 + X^6 + X^8 + X^9 + X^{10}$
31	16	3	$1 + X + X^2 + X^3 + X^5 + X^7 + X^8 + X^9 + X^{10} + X^{11} + X^{15}$
31	11	5	$1 + X^2 + X^4 + X^6 + X^7 + X^9 + X^{10} + X^{13} + X^{17} + X^{18} + X^{20}$
63	51	2	$1 + X^3 + X^4 + X^5 + X^8 + X^{10} + X^{12}$
63	45	3	$1 + X + X^2 + X^3 + X^6 + X^7 + X^9 + X^{15} + X^{16} + X^{17} + X^{18}$
63	39	4	$1 + X + X^2 + X^4 + X^5 + X^6 + X^8 + X^9 + X^{10} + X^{13} + X^{16} + X^{17} + X^{19} + X^{20} + X^{22} + X^{23} + X^{24}$
63	36	5	$1 + X + X^4 + X^8 + X^{15} + X^{17} + X^{18} + X^{19} + X^{21} + X^{22} + X^{27}$
63	30	6	$1 + X + X^2 + X^5 + X^6 + X^8 + X^9 + X^{11} + X^{13} + X^{14} + X^{15} + X^{20} + X^{22} + X^{23} + X^{26} + X^{27} + X^{28} + X^{29} + X^{30} + X^{32} + X^{33}$
127	113	2	$1 + X + X^2 + X^4 + X^5 + X^6 + X^8 + X^9 + X^{14}$
127	106	3	$1 + X + X^5 + X^6 + X^7 + X^8 + X^{11} + X^{12} + X^{14} + X^{15} + X^{17} + X^{18} + X^{21}$
255	239	2	$1 + X + X^5 + X^6 + X^8 + X^9 + X^{10} + X^{11} + X^{13} + X^{14} + X^{16}$
255	231	3	$1 + X^2 + X^4 + X^5 + X^7 + X^8 + X^{13} + X^{15} + X^{16} + X^{17} + X^{19} + X^{20} + X^{21} + X^{23} + X^{24}$

Example 3.16

Let α be a primitive element of GF(2^3) and t = 1. The minimum polynomials corresponding to α and α^2 are:

$$\phi_1(X) = \phi_2(X) = X^3 + X + 1$$

So, the generator polynomial is obtained as:

$$g(X) = \mathrm{LCM}[\phi_1(X), \phi_2(X)] = X^3 + X + 1$$

Notice that g(X) is exactly that same generator polynomial of the (7,4) cyclic Hamming code. Therefore the (7,4) cyclic Hamming code is also a BCH code.

With the generator polynomial ready, encoding of a BCH code is done just the same as that of a cyclic code.

MATLAB Experiment 3.16

The function `bchpoly` in the MATLAB Communications Toolbox is designed to find the generator polynomial of a binary BCH code with codeword length `n` and message length `k`.

```
>> n = 7; k = 4;                   % (7,4) BCH code
>> genpoly = bchpoly(n,k)          % generator polynomial
genploy =
             1  1  0  1
```

The corresponding generator polynomial for the result is $1 + X + X^3$.

MATLAB Experiment 3.17

Run the following script and encode a message word `m` into an (n, k) BCH code specified by the generator polynomial `g`.

```
>> n = 7; k = 4;              % (7,4) BCH code
>> g = [1 1 0 1];             % from last experiment
>> m = [1 0 0 1];             % message, you may change
                              % it to any 4-tuple
>> c = bchenco(m,n,k,g)       % encoding
c =
       0  1  1  1  0  0  1
```

3.3.1.2 Parity-Check Matrix of BCH Codes

BCH codes can also be specified by their generator matrices or parity-check matrices. Stated previously, a BCH code takes α^i $(i = 1, 2, \cdots, 2t)$ as its roots. This implies $c(\alpha^i) = 0$. Writing it out, we have:

$$\begin{aligned}
&c_0 + c_1\alpha^1 + c_2\alpha^2 + \cdots + c_{n-1}\alpha^{n-1} = 0\\
&c_0 + c_1(\alpha^2)^1 + c_2(\alpha^2)^2 + \cdots + c_{n-1}(\alpha^2)^{n-1} = 0\\
&\vdots\\
&c_0 + c_1(\alpha^{2t})^1 + c_2(\alpha^{2t})^2 + \cdots + c_{n-1}(\alpha^{2t})^{n-1} = 0
\end{aligned} \tag{3.59}$$

or in matrix form:

$$(c_0 c_1 c_2 \cdots c_{n-1}) \cdot \begin{bmatrix} 1 & 1 & \cdots & 1 \\ \alpha^1 & (\alpha^2)^1 & \cdots & (\alpha^{2t})^1 \\ \alpha^2 & (\alpha^2)^2 & \cdots & (\alpha^{2t})^2 \\ \vdots & & & \\ \alpha^{n-1} & (\alpha^2)^{n-1} & \cdots & (\alpha^{2t})^{n-1} \end{bmatrix} = \mathbf{0} \qquad (3.60)$$

On the other hand, the parity-check matrix satisfies the following [see (3.7)]:

$$\boldsymbol{c} \cdot \boldsymbol{H}^{\mathrm{T}} = \mathbf{0} \qquad (3.61)$$

Comparing (3.60) with (3.61), we obtain the parity-check matrix of the BCH code:

$$\boldsymbol{H} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ \alpha^1 & (\alpha^2)^1 & \cdots & (\alpha^{2t})^1 \\ \alpha^2 & (\alpha^2)^2 & \cdots & (\alpha^{2t})^2 \\ \vdots & & & \\ \alpha^{n-1} & (\alpha^2)^{n-1} & \cdots & (\alpha^{2t})^{n-1} \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} 1 & \alpha^1 & \alpha^2 & \cdots & \alpha^{n-1} \\ 1 & (\alpha^2)^1 & (\alpha^2)^2 & \cdots & (\alpha^2)^{n-1} \\ 1 & (\alpha^3)^1 & (\alpha^3)^2 & \cdots & (\alpha^3)^{n-1} \\ \vdots & & & & \\ 1 & (\alpha^{2t})^1 & (\alpha^{2t})^2 & \cdots & (\alpha^{2t})^{n-1} \end{bmatrix} \qquad (3.62)$$

3.3.2 BCH Bound and Vandermonde Matrix

We first state that, by design, the minimum distance of a BCH code is $d_{\min} = 2t + 1$. As such, a BCH code is able to correct up to $\lfloor (d_{\min} - 1)/2 \rfloor = t$ errors. This error correction capability is called the *BCH bound*.

Now we prove the bound by using *reductio ad absurdum*. BCH codes are linear block codes. So the minimum distance is the minimum weight of nonzero codewords. Suppose that there exists a nonzero codeword of weight $w \le 2t$. Let $c_{j_1} = 1$, $c_{j_2} = 1$, $\cdots$, $c_{j_w} = 1$ be the nonzero components in the codeword $\boldsymbol{c}$. It follows from (3.61) that:

$$\begin{aligned}
\boldsymbol{c}\cdot\boldsymbol{H}^{\mathrm{T}}=\mathbf{0}\Rightarrow\left(c_{i_1}\ c_{i_2}\ \cdots c_{i_w}\right)&\begin{bmatrix}\alpha^{j_1} & (\alpha^2)^{j_1} & \cdots & (\alpha^{2t})^{j_1}\\ \alpha^{j_2} & (\alpha^2)^{j_2} & \cdots & (\alpha^{2t})^{j_2}\\ \vdots & & & \\ \alpha^{j_w} & (\alpha^2)^{j_w} & \cdots & (\alpha^{2t})^{j_w}\end{bmatrix}=\mathbf{0}\\
\Rightarrow\underbrace{(1\,1\cdots1)}_{w\ 1\text{'s}}&\begin{bmatrix}\alpha^{j_1} & (\alpha^{j_1})^2 & \cdots & (\alpha^{j_1})^{2t}\\ \alpha^{j_2} & (\alpha^{j_2})^2 & \cdots & (\alpha^{j_2})^{2t}\\ \vdots & & & \\ \alpha^{j_w} & (\alpha^{j_w})^2 & \cdots & (\alpha^{j_w})^{2t}\end{bmatrix}=\mathbf{0}
\end{aligned}\tag{3.63}$$

Equation (3.63) implies that every column in the $w \times 2t$ matrix must sum to zero. However, we also find that any $w \times w$ submatrix of the preceding matrix is a *Vandermonde matrix* whose columns never sum to zero. So our assumption that $d_{\min} \leq 2t$ must be invalid, and the code distance of BCH codes exceeds $2t$. This proves that the minimum distance $d_{\min} = 2t + 1$.

Note that although we have proven that $d_{\min} = 2t + 1$, the true minimum distance of a BCH code may be larger and the code may be able to correct more than t errors. However, despite that potential, most BCH decoding methods will correct only up to errors anyway, and any extra capability of the code goes into error detection.

3.3.3 Decoding BCH Codes

3.3.3.1 General Approach

Thanks to the unique algebraic structure in BCH codes, the syndrome computation for the codes can be simplified to a great extent. Using the definition of syndrome $\boldsymbol{S} = S_0\ S_1\ S_2 \cdots S_{n-1} = \boldsymbol{r} \cdot \boldsymbol{H}^{\mathrm{T}}$ and the parity-check matrix for BCH codes $\boldsymbol{H}$ specified in (3.62), each component of the syndrome, S_i, can be easily calculated as:

$$S_i = r(\alpha^i) = r_0 + r_1\alpha^i + r_2(\alpha^i)^2 + \cdots + r_{n-1}(\alpha^i)^{n-1} \qquad (i = 1, 2, \cdots, 2t) \tag{3.64}$$

Substitute $r(X) = c(X) + e(X)$ into (3.64):

$$S_i = c(\alpha^i) + e(\alpha^i) = e(\alpha^i) \tag{3.65}$$

Assume that $r(X)$ contains $\nu \le t$ errors at locations $j_1, j_2, \cdots, j_\nu$. Then the error polynomial is:

$$e(X) = e_{j1} \cdot X^{j1} + e_{j2} \cdot X^{j2} + \cdots + e_{j\nu} \cdot X^{j\nu} \tag{3.66}$$

Combining (3.65) and (3.66), we obtain the syndrome:

$$\begin{aligned} S_i &= e_{j_1} \cdot (\alpha^i)^{j_1} + e_{j_2} \cdot (\alpha^i)^{j_2} + \cdots + e_{j_\nu} \cdot (\alpha^i)^{j\nu} \\ &= e_{j_1} \cdot (\alpha^{j_1})^i + e_{j_2} \cdot (\alpha^{j_2})^i + \cdots + e_{j_\nu} \cdot (\alpha^{j\nu})^i \quad (i = 1, 2, \cdots, 2t) \end{aligned} \tag{3.67}$$

Letting β_k denote α^{jk}, (3.67) is rewritten as:

$$\begin{aligned} S_1 &= e_{j_1}\beta_1 + e_{j_2}\beta_2 + e_{j_3}\beta_3 + \cdots + e_{j_\nu}\beta_\nu \\ S_2 &= e_{j_1}\beta_1^2 + e_{j_2}\beta_2^2 + e_{j_3}\beta_3^2 + \cdots + e_{j_\nu}\beta_\nu^2 \\ S_3 &= e_{j_1}\beta_1^3 + e_{j_2}\beta_2^3 + e_{j_3}\beta_3^3 + \cdots + e_{j_\nu}\beta_\nu^3 \\ &\vdots \\ S_{2t} &= e_{j_1}\beta_1^{2t} + e_{j_2}\beta_2^{2t} + e_{j_3}\beta_3^{t} + \cdots + e_{j_\nu}\beta_\nu^{2t} \end{aligned} \tag{3.68}$$

Notice that for binary codes, $e_{jk} = 1$ ($k = 1, 2, \cdots, \nu$). Equation (3.68) is further simplified to:

$$\begin{aligned} S_1 &= \beta_1 + \beta_2 + \beta_3 + \cdots + \beta_\nu \\ S_2 &= \beta_1^2 + \beta_2^2 + \beta_3^2 + \cdots + \beta_\nu^2 \\ S_3 &= \beta_1^3 + \beta_2^3 + \beta_3^3 + \cdots + \beta_\nu^3 \\ &\vdots \\ S_{2t} &= \beta_1^{2t} + \beta_2^{2t} + \beta_3^{2t} + \cdots + \beta_\nu^{2t} \end{aligned} \tag{3.69}$$

Decoding of BCH codes is essentially done to find β_k's. Once all β_k's are known, the location of the error(s) is also known. For example, say, $\beta_1 = \alpha^3$, by definition $\alpha^{j_1} = \alpha^3$ or $j_1 = 3$, which means that $r(X)$ has an error with the term X^3. The bit r_3 is then complemented and the error is corrected. The solution to the simultaneous equations in (3.69) is usually not unique. However, if there are t or fewer errors, the most probable error pattern is the one with the smallest number of 1's.

Equation (3.69) is a set of nonlinear functions; solving it directly appears not to be a simple task especially for large t. However, things will be easier if we construct a polynomial $\sigma(X)$ whose roots are the reciprocals of β_k's:

$$\sigma(X) = \sigma_0 + \sigma_1 X + \sigma_2 X^2 + \cdots + \sigma_\nu X^\nu \triangleq (1-\beta_1 X)(1-\beta_2 X)\cdots(1-\beta_\nu X) \tag{3.70}$$

where

$$\begin{aligned} \sigma_0 &= 1 \\ -\sigma_1 &= \beta_1 + \beta_2 + \cdots \beta_\nu \\ \sigma_2 &= \beta_1\beta_2 + \beta_2\beta_3 + \cdots + \beta_{\nu-1}\beta_\nu \\ &\vdots \\ (-1)^\nu \sigma_\nu &= \beta_1\beta_2 \cdots \beta_{\nu-1}\beta_\nu \end{aligned} \tag{3.71}$$

As we will see shortly an efficient algorithm is available with which to build the polynomial. Polynomial $\sigma(X)$ is called the *error location polynomial*. Equation (3.71) is referred to as the *elementary symmetric function*.

Summarizing, decoding of BCH codes involves:

1. Calculating a syndrome;
2. Determining the error location polynomial $\sigma(X)$;
3. Finding the roots of the error location polynomial (i.e., error positions);
4. Correcting the errors.

Step 1 is done by evaluating (3.64). Step 4 is trivial for binary codes. In the next two subsections we focus on steps 2 and 3 in detail.

Based on the decoding steps just listed, the block diagram of a BCH decoder is illustrated in Figure 3.22. The delay is used to compensate for the processing latency in steps 1, 2, and 3, such that the delayed $r(X)$ aligns

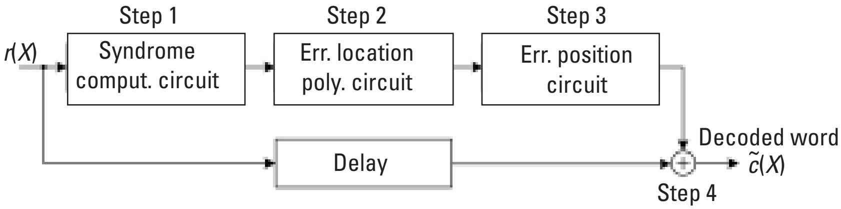


Figure 3.22 Block diagram of binary BCH decoder.

properly with the error position signal. The signal is 1 whenever an error is found, and simultaneously is used to correct the error bit in $r(X)$ using XOR.

3.3.3.2 Error Location Polynomial and Peterson's Algorithm

We now come down to the algorithm for deriving the error location polynomial. It has been shown that the coefficients of the error locator polynomial $\sigma(X)$ are related to the syndrome by the so-called *Newton's identity* [3, p. 285]:

$$\begin{aligned}
&S_1 + \sigma_1 = 0 \\
&S_2 + \sigma_1 S_1 + 2\sigma_2 = 0 \\
&S_3 + \sigma_2 S_1 + \sigma_1 S_2 + 3\sigma_3 = 0 \\
&\vdots \\
&S_v + \sigma_1 S_{v-1} + \sigma_2 S_{v-2} + \cdots + \sigma_{v-1} S_1 + v\sigma_v = 0 \\
&S_{v+1} + \sigma_1 S_v + \sigma_2 S_{v-1} + \cdots + \sigma_{v-1} S_2 + \sigma_v S_1 = 0 \\
&S_{v+2} + \sigma_1 S_{v+1} + \sigma_2 S_v + \cdots + \sigma_{v-1} S_3 + \sigma_v S_2 = 0 \\
&\vdots \\
&S_{2t} + \sigma_1 S_{2t-1} + \sigma_2 S_{2t-2} + \cdots + \sigma_{v-1} S_{2t-v+1} + \sigma_v S_{2t-v} = 0
\end{aligned} \tag{3.72}$$

Considering the fact that for binary codes the syndrome component S_t is either 1 or 0, we have:

$$\sigma S_i = \begin{cases} 0 & \text{if } \sigma \text{ is even} \\ S_i & \text{if } \sigma \text{ is odd} \end{cases} \tag{3.73}$$

Moreover, based on (2.5) in Chapter 2, we also have:

$$S_{2i} = \sum_{k=1}^{v} \beta_k^{2i} = \left(\sum_{k=1}^{v} \beta_k^{i} \right)^2 = S_i^2 \tag{3.74}$$

Substituting (3.73) and (3.74) into (3.72), we find that every second equation in (3.72) is redundant and can be omitted. As a result, a simplified version of (3.72) is obtained:

$$\begin{aligned}
&S_1 + \sigma_1 = 0 \\
&S_3 + \sigma_2 S_1 + \sigma_1 S_2 + \sigma_3 = 0 \\
&S_5 + \sigma_4 S_1 + \sigma_3 S_2 + \sigma_2 S_3 + \sigma_1 S_4 + \sigma_5 = 0 \\
&\vdots \\
&S_{2t-1} + \sigma_1 S_{2t-2} + \sigma_2 S_{2t-3} + \cdots + \sigma_{v-1} S_{2t-v} + \sigma_v S_{2t-v-1} = 0
\end{aligned} \tag{3.75}$$

Equation (3.75) is a linear equation. The error location polynomial coefficients $\sigma_0, \sigma_1, \sigma_2, \cdots, \sigma_v$ are readily obtained by solving the equation. This is the algorithm proposed by Peterson in 1960 [12] to derive the error location polynomial. Equation (3.75) can be compactly arranged in the following matrix form:

$$\underbrace{\begin{bmatrix}
1 & 0 & 0 & 0 & \cdots & 0 & 0 \\
S_2 & S_1 & 1 & 0 & \cdots & 0 & 0 \\
S_4 & S_3 & S_2 & S_1 & \cdots & 0 & 0 \\
\vdots & & & & & & \\
S_{2t-2} & S_{2t-3} & S_{2t-4} & S_{2t-5} & \cdots & S_{2t-v} & S_{2t-v-1}
\end{bmatrix}}_{\boldsymbol{A}}
\underbrace{\begin{bmatrix}
\sigma_1 \\ \sigma_2 \\ \sigma_3 \\ \vdots \\ \sigma_v
\end{bmatrix}}_{\Lambda}
=
\underbrace{\begin{bmatrix}
-S_1 \\ -S_3 \\ -S_5 \\ \vdots \\ -S_{2t-1}
\end{bmatrix}}_{\boldsymbol{S}}$$

or

$$\boldsymbol{A} \cdot \Lambda = \boldsymbol{S} \tag{3.76}$$

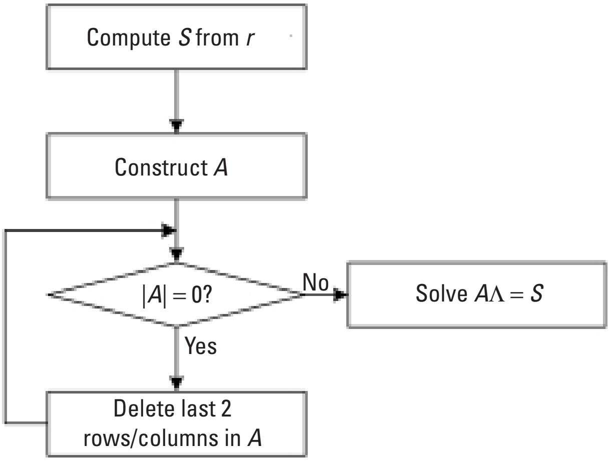


Figure 3.23 Flowchart of Peterson's algorithm.

One small issue remaining is that we do not know ν. This can be solved by initially assuming $\nu = t$. If indeed there are t errors, matrix $\boldsymbol{A}$ is nonsingular and has a nonzero determinant $|\boldsymbol{A}|$, and (3.75) [or (3.76)] can be solved as is. If there are fewer than t errors (i.e., $\nu < t$), then $\boldsymbol{A}$ is singular. In this case we remove the last two rows and the rightmost two columns from $\boldsymbol{A}$,[8] and then check for singularity of the new matrix. The procedure continues until the remaining matrix eventually becomes nonsingular. Then we solve (3.76) with the nonsingular matrix and obtain Λ.

Figure 3.23 is the flowchart of *Peterson's algorithm*.

For a small number of errors, it may be easier to solve the equations directly. The following lists some of the results [13]:

Single Error Correction

$$\sigma_1 = S_1$$

Double Error Correction

$$\sigma_1 = S_1, \quad \sigma_2 = \frac{S_3 + S_1^3}{S_1}$$

[8] Accordingly, the last two elements in A are also removed.

Triple Error Correction

$$\sigma_1 = S_1, \quad \sigma_2 = \frac{S_1^2 S_3 + S_5}{S_1^3 + S_3}, \quad \sigma_3 = (S_1^3 + S_3) + S_1\sigma_2$$

Four Error Correction

$$\sigma_1 = S_1, \quad \sigma_2 = \frac{S_1(S_7 + S_1^7) + S_3(S_1^5 + S_5)}{S_3(S_1^3 + S_3) + S_1(S_1^5 + S_5)}, \quad \sigma_3 = (S_1^3 + S_3) + S_1\sigma_2$$

$$\sigma_4 = \frac{(S_1^2 S_3 + S_5) + (S_1^3 + S_3)\sigma_2}{S_1}$$

Five Error Correction

$$\sigma_1 = S_1$$

$$\sigma_2 = \frac{(S_1^3 + S_3)\left[(S_1^9 + S_9) + S_1^4(S_5 + S_1^2 S_3) + S_3^2(S_1^3 + S_3)\right] + (S_1^5 + S_5)(S_7 + S_1^7) + S_1(S_3^2 + S_1 S_5)}{(S_1^3 + S_3)\left[(S_7 + S_1^7) + S_1 S_3(S_1^3 + S_3)\right] + (S_5 + S_1^2 S_3)(S_1^5 + S_5)}$$

$$\sigma_3 = (S_1^3 + S_3) + S_1\sigma_2$$

$$\sigma_4 = \frac{(S_1^9 + S_9) + S_3^2(S_1^3 + S_3) + S_1^4(S_5 + S_1^2 S_3) + \sigma_2\left[(S_7 + S_1^7) + S_1 S_3(S_1^3 + S_3)\right]}{S_1^5 + S_5}$$

$$\sigma_5 = S_5 + S_1^2 S_3 + S_1 S_4 + \sigma_2(S_1^3 + S_3)$$

Although Peterson's method is conceptually straightforward, it is computationally complex. In practice, a more effective algorithm such as the Berlekamp-Massey algorithm or Euclid's method are often used instead (see the next chapter).

3.3.3.3 Finding Error Locations and Chien Search

Now we have the error location polynomial ready. The next step is to find the roots of the error location polynomial $\sigma(X)$. One simple yet effective method is to do an exhaustive search.

Notice that any root of $\sigma(X)$ must be one of the elements in the Galois field of the code: $1, \alpha, \alpha^2, \cdots, \alpha^{n-1}$, where $n = 2^m - 1$. As such, we can evaluate one by one if an element α^i is a root; that is, we evaluate $\sigma(X)$ at each nonzero element of the field. If $\sigma(\alpha^i) = 0$, α^i is a root of $\sigma(X)$. This algorithm is by Chien and is called the *Chien search* [14].

A hardware realization of the algorithm is drawn in Figure 3.24. The jth register contains $\sigma_j \cdot (\alpha^j)^i$, where $i = 0, 1, 2, \cdots, n - 1$.The output of the circuit is $\Sigma = \sigma(\alpha^i) = \sigma_0 + \sigma_1 \cdot \alpha^i + \sigma_2 \cdot (\alpha^2)^i + \cdots + \sigma_v \cdot (\alpha^v)^i = \sigma_0 + \sigma_1 \cdot \alpha^i + \sigma_2 \cdot (\alpha^i)^2 + \cdots + \sigma_v \cdot (\alpha^i)^v$. After n clocks, all elements of the field are tested.

Example 3.17

Now we use the (15,5) two-error-correcting BCH code as an example to demonstrate BCH decoding. The generator polynomial of the code is as follows (see Table 3.11):

$$g(X) = 1 + X^4 + X^6 + X^7 + X^8$$

which has as roots α, α^2, α^3, and α^4. Suppose the message polynomial $m(X) = X + X^2$. The corresponding code polynomial is $c(X) = X + X^2 + X^4 + X^5 + X^6 + X^7 + X^8 + X^{11} + X^{12}$. The received polynomial $r(X) = X + X^2 + X^5 + X^6 + X^7 + X^8 + X^{12}$ contains two errors. The syndrome of $r(X)$ is calculated to be:

$$S_1 = r(\alpha) = \alpha^{13},\ S_2 = r(\alpha^2) = \alpha^{11},\ S_3 = r(\alpha^3) = \alpha^{10} \text{ and } S_4 = r(\alpha^4) = \alpha$$

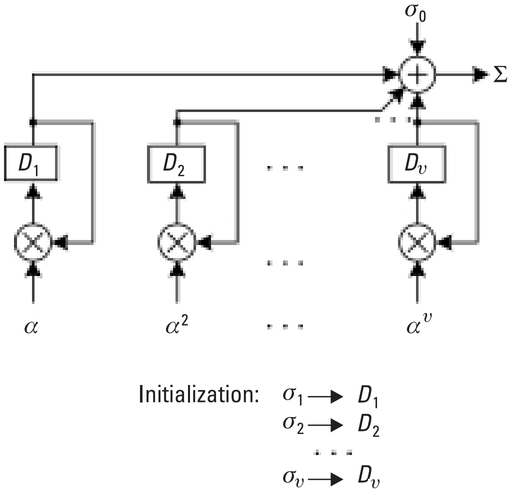

σ_0
Σ
D_1
D_2
D_v
α
α^2
α^v
Initialization: $\sigma_1 \rightarrow D_1$
$\sigma_2 \rightarrow D_2$
$\sigma_v \rightarrow D_v$

Figure 3.24 Hardware implementation of the Chien search.

Using the formulas for double error correction listed earlier, we have:

$$\sigma_1 = S_1 = \alpha^{13}, \quad \sigma_2 = \frac{S_3 + S_1^3}{S_1} = 1$$

So the error location polynomial is:

$$\sigma(X) = 1 + \alpha^{13}X + X^2$$

Inserting 1, α, α^2, $\cdots$, α^{14} into $\sigma(X)$ and using the result of Problem 2.4(a) in Chapter 2, we find that $\sigma(\alpha^{11})$ and $\sigma(\alpha^4)$are equal to zero. The inverses of the two roots are α^4 and α^{11}. So the error polynomial is:

$$e(X) = X^4 + X^{11}$$

Summing up, e(X) and r(X) indeed give the correct codeword c(X).

MATLAB Experiment 3.18

The MATLAB built-in function `bchdeco(r,k,t)` decodes BCH code `r`. Parameter `k` is the message length and `t` is the error-correction capability.

The following experiment uses the function to decode the (15,5) BCH code:

```
>> k = 5; t = 2;                             % code parameters
>> r = [0 1 1 0 0 1 1 1 1 0 0 0 1 0 0];    % = r(X) in example
>> m = bchdeco(r,k,t)                        % decoding
m =
      0  1  1  0  0
```

Note The focus of this chapter has been on the generalities of block codes. Therefore, in terms of specific codes, we have only presented Hamming codes and CRC codes. Many other types of block codes are widely used in practice, such as Reed-Muller codes, Golay codes and Fire codes, to name a few. They are by no means less important. Interested readers are referred to [2, 3, 5].

The chapter discusses binary codes. The most important nonbinary BCH codes, Reed-Solomon codes, will be presented in the next chapter.

Problems

3.1 This is a (30,20) binary block code. The codeword is formed by first arranging the 20 message bits into a 5 × 4 array, and then assigning a parity bit to each row and column. The value of the parity is XOR of all bits in the row or column. For instance, if the message is $\boldsymbol{m}$ = (10110100011011000011), the two-dimensional codeword $\boldsymbol{c}$ is as follows:

	1	2	3	4	5
1	1	0	1	1	1
2	0	1	0	0	1
3	0	1	1	0	0
4	1	1	0	0	0
5	0	0	1	1	0
6	0	1	1	0	0

← Parity

↑
Parity

The codeword bits are read out of the array row by row.

The code has certain error correction capability. Suppose that during the transmission the bit at location (3,2) is flipped into 0. The row parity and the column parity associated with the bit are recomputed as 1 and 0, and do not match the corresponding received parities. [They are 0 and 1; see locations (3,5) and (6,2), respectively.] Based on the observation, we know that the bit at the intersection of the row and the column [i.e., the bit at (3,2)] is in error. The bit is then complemented (i.e., corrected).

(a) Determine the generator matrix and the parity-check matrix of the code.

(b) What is the number of errors the code guarantees to correct? What is the maximum number of errors the code may correct (though not guaranteed)?

(c) Determine the code rate.

(d) Compare the code with the (31,21) BCH code (see Table 3.11). Which one is more efficient?

3.2 Assume a channel that generates only the following six different error patterns:

(1001010), (0110111), (1001000), (0011011), (1010111), (0100010).

Determine the largest possible k for a (7, k) block code that can detect all of the above error patterns.

3.3 Consider the (15,7) BCH code in Table 3.11. Could the polynomial $X + X^5 + X^8$ possibly be a syndrome polynomial of the code? Why?

3.4 Use MATLAB to simulate the probability that the (7,4) example code fails to detect errors, and compare the result with the bound in (3.25).

3.5 Implement the Meggitt decoder in MATLAB. (*Hint:* Use the `synd*` and `errpat*` function provided in this book's companion DVD.)

3.6 For a cyclic code, if an error pattern is detectable, is its cyclically shifted version also detectable? Justify your conclusion [2].

3.7 Show that all CRC are guaranteed to correct up to one error. [*Hint:* $g(X)|_{x=1} = 0 \Rightarrow c(X)|_{x=1} = 0 \Rightarrow d_{\min} = 2$].

3.8 What are the correctable error patterns of the (7,4) Hamming code? (*Hint:* Use a standard array.)

References

[1] Peterson, W. W., and E. J. Weldon, Jr., *Error-Correcting Codes*, 2nd ed., Cambridge, MA: The MIT Press, 1972.

[2] Lin, S., and D. J. Castello, *Error Control Coding—Fundamentals and Application*, Upper Saddle River, NJ: Prentice-Hall, 2004.

[3] Moon, T. K., *Error Control Coding—Mathematical Methods and Algorithms*, New York: John Wiley & Sons, 2005.

[4] Sklar, B., *Digital Communications—Fundamentals and Applications*, 2nd ed., Upper Saddle River, NJ: Prentice-Hall, 2001.

[5] Reed, I. S., and X. M. Chen, *Error-Control Coding for Data Network*, Norwell, MA: Kluwer, 1999.

[6] Blahut, R. E., *Algebraic Codes for Data Transmissions*, Cambridge, U.K.: Cambridge University Press, 2003.

[7] Meggitt, J. E., "Error Correcting Codes and Their Implementation," *IRE Trans. Inform. Theory*, Vol. IT-7, October 1961, pp. 232–244.

[8] Kasami, T., "A Decoding Procedure for Multiple-Error-Correction Cyclic Codes," *IEEE Trans. Inform. Theory*, Vol. 10, April 1964, pp. 134–139.

[9] Wells, R. B., *Applied Information Theory and Coding for Engineers*, Upper Saddle River, NJ: Prentice-Hall, 1998.

[10] Hocquenghem, A., "Code Corecteurs D'Erreurs," *Chiffres*, Vol. 2, 1959, pp. 147–156.

[11] Bose, R. C., and D. K. Ray-Chaudhuri, "On a Class of Error Correcting Binary Group Codes," *Inform. Control*, Vol. 3, March 1960, pp. 68–79.

[12] Peterson, W. W., "Encoding and Error-Correction Procedures for Bose-Chaudhuri Codes," *IRE Trans. Inform. Theory*, Vol. IT-6, September 1960, pp. 459–470.

[13] Michelson, A. M., and A. H. Levesque, *Error Control Techniques for Digital Communication*, New York: John Wiley & Sons, 1985.

[14] Chien, R. T., "Cyclic Decoding Procedure for the Bose-Chaudhuri-Hocquenghem Codes," *IEEE Trans. Inform. Theory*, Vol. IT-10, October 1964, pp. 357–363.

Selected Bibliography

Clark, G., and J. Cain, *Error-Correcting Codes for Digital Communications*, New York: Plenum Press, 1981.

Hamming, R. W., "Error Detecting and Error Correcting Codes," *Bell Syst. Tech. J.*, Vol. 29, April 1950, pp. 41–56.

Wicker, S. B., *Error Control Systems for Digital Communication and Storage*, Englewood Cliffs, NJ: Prentice-Hall, 1995.

4

Reed-Solomon Codes

In Chapter 3 we constructed all BCH codes over $GF(2)$, which resulted in binary BCH codes. There are also BCH codes that are over $GF(q)$, where $q > 2$. Such codes are called *nonbinary BCH codes*. A special class of nonbinary BCH codes that is overwhelmingly popular is *Reed-Solomon* (RS) codes. Discovered by I. Reed and G. Solomon in 1960 [1], RS codes are very powerful in correcting both random errors and bursty errors. RS codes were initially applied in deep-space communications; today they are found in numerous consumer products, such as mass storage devices (e.g., CDs and DVDs), broadband modems (e.g., xDSL and cable modems), wireless mobile communications systems (e.g., WiMax), and so forth. Some of the literature claims that RS codes are the most widely used error correcting codes in the world.

RS codes (or nonbinary BCH codes as a whole) are a generalization of binary BCH codes. Many similarities exist between the two. In this chapter, we assume that readers are already familiar with the materials in the previous chapter and concentrate on specifics for RS codes.

4.1 Introduction to RS Codes

4.1.1 Prelude: Nonbinary BCH Codes

4.1.1.1 From Binary BCH Codes to Nonbinary BCH Codes

Nonbinary BCH codes differ from binary BCH codes in that nonbinary BCH codewords consist of *symbols* over $GF(2^m)$ $(m \geq 2)$,[1] whereas binary

BCH codewords contain binary bits of 1's and 0's. As a consequence, arithmetic operations in nonbinary codes are no longer the simple XORs and ANDs; rather, they are performed over $GF(2^m)$.[2] Other than that, all properties associated with binary BCH codes apply equally to nonbinary codes.

Similar to binary BCH codes, a t-error-correcting nonbinary BCH code of length $n = q^z - 1$ ($q = 2^m$ and $z \geq 3$) contains roots $\alpha^1, \alpha^2, \alpha^3, \ldots, \alpha^{2t}$,[3] where α is the primitive element of $GF(q^z)$. The generator polynomial of the code is constructed as follows:

$$g(X) = g_0 + g_1X + g_2X^2 + \cdots + g_{k-1}X^{k-1} = \mathrm{LCM}[\phi_1(X), \phi_2(X), \cdots, \phi_{2t}(X)] \tag{4.1}$$

where $g_i \in GF(2^m)$ and $\phi_i(X)$ is the minimum polynomial of α^i. The minimum distance of the code is $d_{\min} \geq 2t + 1$. Note that in nonbinary codes "t errors" means "*t symbol* errors." [Do not confuse the two Galois fields associated with a nonbinary BCH code: $GF(2^m)$ and $GF(q^z)$ where $q = 2^m$. The former is where the generator polynomial gets its coefficients and also where the codeword symbols are from, and the latter is the field over which the generator polynomial has its roots.]

The encoding process for nonbinary BCH codes is the same as that for binary BCH codes. The encoding methods presented in Chapter 3 can be directly ported to nonbinary BCH codes, as long as the computations are performed over $GF(2^m)$.

Compared with binary BCH codes, decoding of nonbinary BCH codes involves one extra step: finding the magnitude of the error(s). The complete decoding procedure now consists of a total of five steps:

Decoding of Nonbinary BCH Codes

1. Calculate syndrome S_i for $i = 1, 2, \ldots, 2t$ [see (3.64)].
2. Determine the error location polynomial $\sigma(X)$.

[1] In a broader sense, the codeword symbols can be over $GF(p^m)$, where p is a prime number. However, in almost all practical applications $GF(2^m)$ is adopted because our systems live on binary logic.

[2] The message word comprises symbols that are also over Galois field $GF(2^m)$.

[3] Notice that the root α^i starts with $i = 1$. Such a code is said to be *narrow sense*. If the root starts with $i > 1$, the code is *nonnarrow sense*.

3. Find the roots of the error location polynomial (Chien search).
4. Compute the error magnitude.
5. Correct the errors [see (3.13)].

Note that the computation in the first step is performed over $GF(q^z)$, all others are over $GF(2^m)$.

A nonbinary BCH decoder based on the procedure here is sketched in Figure 4.1. The error position determination unit controls switch sw. When an error position is determined, the unit closes sw to let the error magnitude pass through to the $GF(2^m)$ adder for correction of the error.

The Peterson-Gorenstein-Zierler (PGZ) algorithm [2, 3], extended from Peterson's algorithm to the nonbinary case, can be used in step 2 above to obtain the error location polynomial. However, like its original version, PGZ algorithm becomes computationally costly as the number of errors increases. Therefore, we should instead turn to some more efficient algorithms (although they may not be as straightforward conceptually). The presentation of these algorithms is deferred to Section 4.2. In the next section we introduce an algorithm for the error magnitude calculation performed in step 4.

4.1.1.2 Calculating Error Magnitude and Forney's Algorithm

The relationship between the syndrome and the error magnitude for a binary BCH code with v errors is specified by (3.68) in Chapter 3, which we duplicate here for convenience:

$$S_i = e_{j_1}\beta_1^i + e_{j_2}\beta_2^i + e_{j_3}\beta_3^i + \cdots + e_{j_v}\beta_v^i \qquad (i = 1, 2, \cdots, 2t) \tag{4.2}$$

where $\beta_k = \alpha^{j_k}$. The same equation holds for nonbinary codes except $e_{j_k} \in GF(2^m)$ in this case. The error magnitude e_{j_k} sought in step 4 of the above

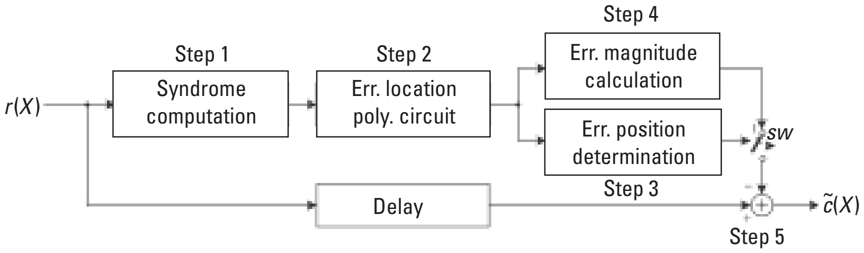

Step 4
Step 1
Step 2
Err. magnitude calculation
r(X)
Syndrome computation
Err. location poly. circuit
Err. position determination
sw
Step 3
Delay
$\tilde{c}(X)$
Step 5

Figure 4.1 Structure of nonbinary BCH decoder.

decoding procedure can then be obtained by solving (4.2) over $GF(2^m)$ after the error location is determined (which is done in step 3).

However, a more efficient algorithm for calculating the error magnitude has been developed by Forney [4]. The algorithm defines an *error evaluation polynomial* $\Omega(X)$ as follows:

$$\Omega(X) \triangleq S(X)\sigma(X) \bmod X^{2t} \tag{4.3}$$

where

$$S(X) = S_1 + S_2X + S_3X^2 + \cdots + S_{2t}X^{2t-1}$$

is the syndrome polynomial, and

$$\sigma(X) = (1-\beta_1 X)(1-\beta_2 X)\cdots(1-\beta_v X) = \sigma_0 + \sigma_1 X + \sigma_2 X^2 + \cdots + \sigma_v X^v$$

is the error location polynomial. We call (4.3) the *key equation*. The modulo-X^{2t} operation in the equation is to remove all terms of order ≥ $2t$.

Forney's algorithm computes the error magnitude as follows:

$$e_{j_i} = \left.\frac{\Omega(X)}{\sigma'(X)}\right|_{X=\beta_i^{-1}} \qquad (i = 1, 2, \cdots, v) \tag{4.4}$$

where $\sigma'(X)$ is the formal derivative of $\sigma(X)$. Proof of the algorithm can be found in [5] or [6].

The value of $\sigma'(X)$ can be calculated as follows:

$$\sigma'(X) \triangleq \frac{d\sigma(X)}{dX} = \frac{d(\sigma_0 + \sigma_1 X + \sigma_2 X^2 + \cdots + \sigma_{v-1}X^{v-1})}{dX}$$

$$= \sigma_1 + 2\sigma_2 X + 3\sigma_3 X^2 + \cdots + (v-1)\sigma_v X^{v-1}$$

Observe that $\sigma(X)$ is a polynomial over $GF(2^m)$. It follows from the $GF(2^m)$ arithmetic (see Section 2.2.2 in Chapter 2):

$$i \cdot \sigma_i = \begin{cases} 0 & \text{if } i \text{ is even} \\ \sigma_i & \text{if } i \text{ is odd} \end{cases}$$

Consequently, $\sigma'(X)$ can be formed by taking the coefficients of the odd power terms of $\sigma(X)$, and assigning them to the next lower power terms (which are even power):

$$\sigma'(X) = \sigma_1 + \sigma_3 X^2 + \sigma_5 X^4 + \cdots \tag{4.5}$$

Later in Section 4.2.4 we will provide an example that illustrates how the algorithm is used in decoding RS codes.

In terms of computational complexity, for the nonbinary BCH codes over $GF(2^m)$ in which we are interested, Forney's algorithm requires about $1.25v^2$ multiplications and v inversions to find all v error magnitudes.

4.1.2 Reed-Solomon Codes

4.1.2.1 Definition and Particulars

Reed-Solomon codes are a special case of nonbinary BCH codes with $z = 1$, that is, the roots of the generator polynomial are also over $GF(2^m)$. For a t-error-correcting RS code of length $n = 2^m - 1$, the generator polynomial is constructed as follows:[4]

$$\begin{aligned} g(X) &= g_0 + g_1 X + g_2 X^2 + \cdots + g_{2t-1} X^{2t-1} + X^{2t} \\ &= (X + \alpha^1)(X + \alpha^2) \cdots (X + \alpha^{2t}) \end{aligned} \tag{4.6}$$

where α is a primitive element of $GF(2^m)$ and α^1, α^2, …, α^{2t} are elements of the same field. Notice that, since $g(X)$ is of degree exactly $2t$, all (n, k) RS codes satisfy the following equation:

$$n - k = 2t \tag{4.7}$$

Example 4.1

The generator polynomial of the (7,3) *RS code is constructed as follows:*

$$g(X) = (X + \alpha)(X + \alpha^2)(X + \alpha^3)(X + \alpha^4)$$

$$= X^4 + \alpha^3 X^3 + X^2 + \alpha X + \alpha^3$$

where α *is the primitive element of* $GF(8)$. *Clearly the code can correct up to* $t = (n - k)/2 = 2$ *erroneous symbols.*

[4] $X + \alpha^i$ is the minimal polynomial of α^i.

MATLAB Experiment 4.1

MATLAB has a function `rspoly` built in the Communications Toolbox that can produce the generator polynomial for RS codes over *GF*(2^m).

```
>> n = 7; k = 3; m = 3;                    % code parameters
>> g = rspoly(n,k,m)                       % produce generator
g =
  3  1  0  3  0
```

The result corresponds to the generator polynomial obtained in Example 4.1.

4.1.2.2 Encoding Using Generator Polynomial

The (n, k) RS code can be encoded just as binary BCH code. The only difference is that the multiplications and additions must now be performed over *GF*(2^m). For systematic encoding of a message polynomial $m(X)$ with k coefficients being the k message symbols over *GF*(2^m), the parity-check polynomial $\mu(X)$ is the remainder of the shifted message polynomial $X^{n-k}m(X)$ divided by the generator polynomial $g(X)$, that is:

$$\mu(X) = X^{n-k}m(X) \bmod g(X) \tag{4.8}$$

The resulting code polynomial is:

$$c(X) = \mu(X) + X^{n-k}m(X) \tag{4.9}$$

Replacing XOR and AND by the *GF*(2^m) adder and multiplier, respectively, the binary BCH encoding circuit in Figure 3.11 can readily be used for systematic RS encoding (the circuit is duplicated in Figure 4.2).

Example 4.2

Continue with the (7,3) RS code. Let the message polynomial be $m(X) = \alpha^2 + \alpha^3 X + X^2$. *The parity-check polynomial is calculated as:*

$$\begin{aligned}\mu(X) &= X^4 m(X) \bmod g(X) \\ &= \alpha^2 X^4 + \alpha^3 X^5 + X^6 \bmod \alpha^3 + \alpha X + X^2 + \alpha^3 X^3 + X^4 \\ &= \alpha^2 + X + \alpha^4 X^2 + \alpha^4 X^3\end{aligned}$$

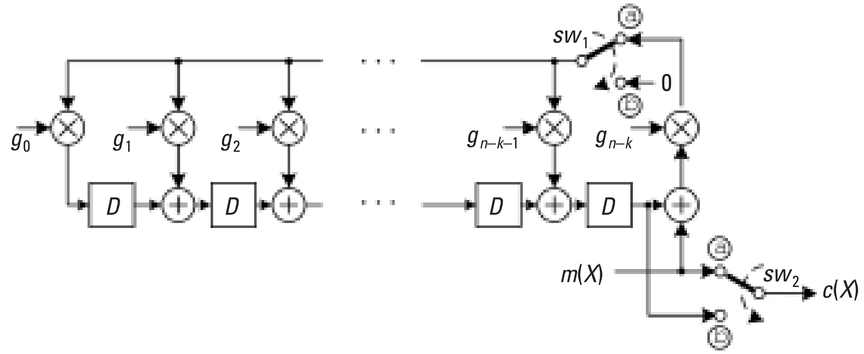


Figure 4.2 Systematic RS encoder.

The complete code polynomial therefore is:

$$c(X) = \mu(X) + X^4 m(X)$$
$$= \alpha^2 + X + \alpha^4 X^2 + \alpha^4 X^3 + \alpha^2 X^4 + \alpha^3 X^5 + X^6$$

MATLAB Experiment 4.2

We can use the MATLAB function `encode` to perform RS encoding if the generator polynomial is known, or simply use `rsenco` and let MATLAB figure out the appropriate generator polynomial.

```
>> n = 7; k = 3;                   % code parameter
>> m = [2 3 0];                    % message poly. as in example 4.2
>> c = rsenco(m,n,k,’power’)   % ‘power’: power representation
c =
  2  0  4  4  2  3  0
```

The codeword is $\boldsymbol{c} = (\alpha^2\ 1\ \alpha^4\ \alpha^4\ \alpha^2\ \alpha^3 1)$.

4.1.2.3 Encoding in the Frequency Domain

RS code can also be generated in the so-called frequency domain. To explain this, we first need to define the Galois field Fourier transform (GFFT). Let α be a primitive element in $GF(q)$. The GFFT of an n-symbol vector $\boldsymbol{c} = (c_0\ c_1$

… c_{n-1}) over $GF(q)$, denoted by $\boldsymbol{C} = \mathcal{F}(\boldsymbol{c})$, is the vector $\boldsymbol{C} = (C_0\ C_1 \ldots C_{n-1})$ with the elements being computed as:

$$C_i = \sum\nolimits_{j=0}^{n-1} c_j \alpha^{ij} \qquad (i = 0,1,0,\cdots,n-1) \qquad (4.10)$$

The inverse GFFT of $\boldsymbol{C}$, denoted by $\boldsymbol{c} = \mathcal{F}^{-1}(\boldsymbol{C})$, is obtained as:

$$c_j = \frac{1}{n \bmod \lambda} \sum\nolimits_{i=0}^{n-1} C_i \alpha^{-ij} \quad (j = 0,1,2,\cdots,n-1) \qquad (4.11)$$

where λ is the characteristic of $GF(q)$. (The characteristic of a Galois field was defined in Chapter 2.) The GFFT is analogous to the discrete Fourier transform (DFT) in signal processing. Therefore, C_i ($i = 0, 1, 0, \ldots, n - 1$) can be viewed as components in the frequency domain.

Interestingly, if we take the GFFT of an RS codeword $\boldsymbol{c}$ over $GF(2^m)$, we find that $\boldsymbol{C}$ contains $2t = n - k$ consecutive zeros. In fact, it can be shown that a polynomial has as roots $2t$ consecutive powers of α: α^1, α^1, …, α^{2t} if and only if the GFFT of its coefficient vector has $2t$ consecutive zeros at locations 1, 2, …, $2t$ [5, p. 272]. RS code contains $2t$ roots of consecutive powers of α. It is therefore possible to perform RS encoding by treating the message as a vector in the frequency domain, inserting necessary zeros and inverse transforming the sequence as follows:

$$\boldsymbol{c} = \mathcal{F}^{-1}(\boldsymbol{C}) \qquad (4.12)$$

where $\boldsymbol{C} = \left(m_0\ \underbrace{0\ 0 \cdots 0}_{2t\ 0\text{'s}}\ m_1 \cdots m_{k-1} \right)$.

RS encoding based on (4.12) is called frequency-domain RS encoding (Figure 4.3).

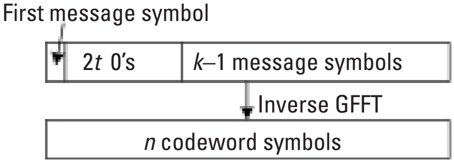


Figure 4.3 RS encoding in the frequency domain.

Example 4.3

Assume the same message polynomial as in Example 4.2, that is, $m(X) = \alpha^2 + \alpha^3 X + X^2$*. Based on (4.11), we have:*
$\boldsymbol{C} = (\alpha^2 0\ 0\ 0\ 0\ \alpha^3 1)$*, and*

$$c_0 = \alpha^2(\alpha^0)^0 + 0(\alpha^0)^1 + 0(\alpha^0)^2 + 0(\alpha^0)^3 + 0(\alpha^0)^4 + \alpha^3(\alpha^0)^5 + 1(\alpha^0)^6 = \alpha^4$$

$$c_1 = \alpha^2(\alpha^{-1})^0 + 0(\alpha^{-1})^1 + 0(\alpha^{-1})^2 + 0(\alpha^{-1})^3 + 0(\alpha^{-1})^4 + \alpha^3(\alpha^{-1})^5 + 1(\alpha^{-1})^6 = 1$$

$$c_2 = \alpha^2(\alpha^{-2})^0 + 0(\alpha^{-2})^1 + 0(\alpha^{-2})^2 + 0(\alpha^{-2})^3 + 0(\alpha^{-2})^4 + \alpha^3(\alpha^{-2})^5 + 1(\alpha^{-2})^6 = 1$$

$$c_3 = \alpha^2(\alpha^{-3})^0 + 0(\alpha^{-3})^1 + 0(\alpha^{-3})^2 + 0(\alpha^{-3})^3 + 0(\alpha^{-3})^4 + \alpha^3(\alpha^{-3})^5 + 1(\alpha^{-3})^6 = \alpha^3$$

$$c_4 = \alpha^2(\alpha^{-4})^0 + 0(\alpha^{-4})^1 + 0(\alpha^{-4})^2 + 0(\alpha^{-4})^3 + 0(\alpha^{-4})^4 + \alpha^3(\alpha^{-4})^5 + 1(\alpha^{-4})^6 = \alpha^2$$

$$c_5 = \alpha^2(\alpha^{-5})^0 + 0(\alpha^{-5})^1 + 0(\alpha^{-5})^2 + 0(\alpha^{-5})^3 + 0(\alpha^{-5})^4 + \alpha^3(\alpha^{-5})^5 + 1(\alpha^{-5})^6 = \alpha^4$$

$$c_6 = \alpha^2(\alpha^{-6})^0 + 0(\alpha^{-6})^1 + 0(\alpha^{-6})^2 + 0(\alpha^{-6})^3 + 0(\alpha^{-6})^4 + \alpha^3(\alpha^{-6})^5 + 1(\alpha^{-6})^6 = \alpha^3$$

So, the codeword $\boldsymbol{c} = (\alpha^4\ 11\ \alpha^3\ \alpha^2\ \alpha^4\ \alpha^3)$*. Obviously, the code is not in systematic form.*

Notice that the number of multiplications needed in (4.11) is the same as that required by (4.8). So why bother to encode RS in the frequency domain? The answer is that it is sometimes useful when the number of message symbols is not fixed. Frequency-domain RS encoding can adapt to different message lengths *k* simply by setting the appropriate zeros in ***C***, which requires no hardware change.

4.1.2.4 Error Probability Performance

By design, an RS code can correct up to $t = (n - k)/2$ errors. Figure 4.4 shows the performance of an $m = 8$ (i.e., $n = 255$) RS code in an AWGN channel.

Singleton proved that the best achievable random error correction capability of a block code is $[(n - k)/2]$ [7], and such a block code is called the *maximum distance separable* (MDS) code. RS codes are exactly MDS codes.

For RS codes over $GF(2^m)$, it has been shown that BER is upper bounded by [8, p. 72]:

$$P_B \le \frac{2^{m-1}}{2^m - 1} \sum_{i=t+1}^{n} \left(\frac{i+t}{n} \right) \binom{n}{i} P_S^i (1 - P_S)^{n-i} \tag{4.13}$$

where P_S is the symbol error rate (SER) before decoding. Given the BSC channel with a crossover probability of p_X, P_S is calculated as:

$$P_S = 1 - (1 - p_X)^m \tag{4.14}$$

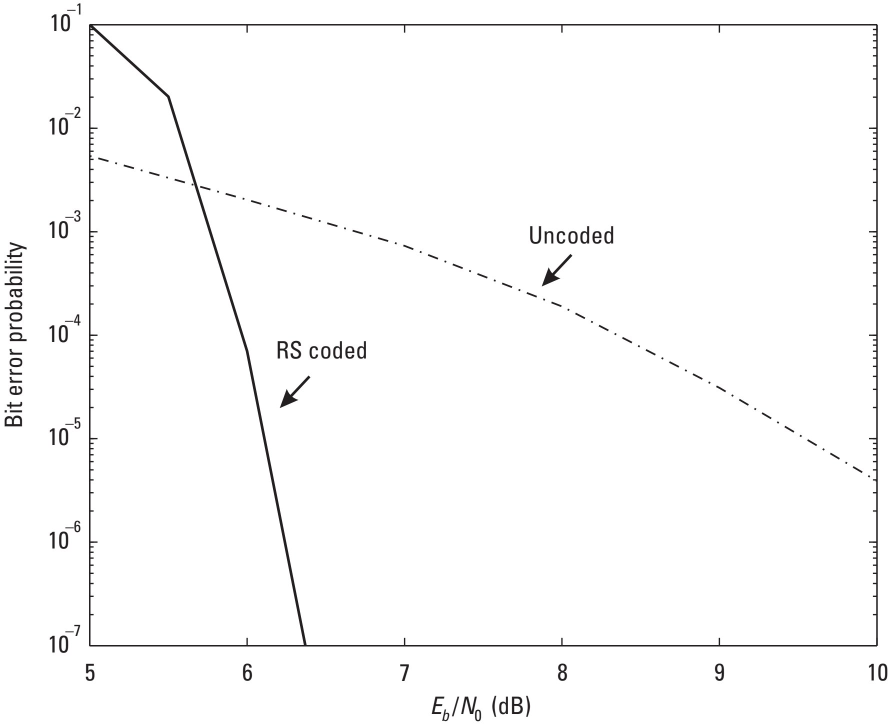


Figure 4.4 RS code performance.

Following a derivation similar to that of (3.29), the upper bound of the probability of a word error can be obtained as:

$$P_W \leq \sum_{i=t+1}^{n} \binom{n}{i} P_S^i (1 - P_S^i)^{n-i} \tag{4.15}$$

RS codes demonstrate strong burst error correcting capability, too. A burst error of length l is a sequence of errors confined to l consecutive symbols of which the first and last are nonzeros. It has been proven that the maximum burst error correcting capability of an (n, k) linear block codes, b, satisfies the *Rieger bound* as follows [9]:

$$b \leq \lfloor (n-k)/2 \rfloor \tag{4.16}$$

For RS codes, (4.16) holds with equality. Therefore, RS codes are also the most powerful block codes for burst error correction. One of the reasons for this strong burst error correction capability is that RS codes are nonbinary, and their error correction is based on a symbol, that is, on a binary tuple, regardless of whether one or all of the bits are in error.

MATLAB Experiment 4.3

We have included in this book's companion DVD two m-files: `rssim.m*` simulating the BER of the example (7,3) RS code, and `rstheory.m*` computing the upper bound in (4.13). Run the simulation and compare the result with the performance of the (7,4) Hamming code simulated in MATLAB Experiment 3.7. The simulation involves RS decoding, which is yet to be discussed. For the time being let us put aside the decoding issue and concentrate on the performance.

4.2 Decoding of RS Codes

4.2.1 General Remarks

RS decoding can be accomplished by going through the same decoding process as outlined in Section 4.1. Most recently, however, several breakthroughs have been achieved in this area, the most notable being the list decoding of RS codes able to correct errors *beyond* half the minimum distance of the code.

The algorithm is based on interpolation and factorization of polynomials over $GF(2^m)$ and its extensions. This advanced topic falls outside the scope of this book. Readers willing to explore further are referred to [10] and [11].

For the conventional RS decoding given earlier, the only thing left untouched is some efficient algorithms for finding the error location polynomial $\sigma(X)$ in (4.3). We now present two such methods that are widely used in practice. Note that the methods are not specific to RS codes; they apply to nonbinary BCH codes in general.

MATLAB Experiment 4.4

As one of the core decoding functions, `rsdeco` is a function that MATLAB has specifically designed to decode RS codes. In MATLAB Experiment 4.3 we used it to decode the example RS code. Readers are encouraged to take a look at the m-file to find out how the function is used.

4.2.2 Determining the Error Location Polynomial

4.2.2.1 Berlekamp-Massey Algorithm

If we compare Newton's identity in (3.72):

$$S_i = -\sum\nolimits_{j=1}^{v} \sigma_j S_{i-j} \quad (i = v+1, v+2, \cdots, 2t \text{ and } v \le t \text{ is the number of errors})$$

with the input/output relation of a classical autoregressive filter:

$$y_i = -\sum\nolimits_{j=1}^{N-1} a_j y_{i-j} + x_i$$

we can see that Newton's identity represents just an autoregressive filter that outputs the syndrome sequence S_{v+1}, S_{v+1}, …, S_{2t} under zero input. Because an autoregressive filter can be implemented as a linear feedback shift register (LFSR), the task of determining the error location polynomial can be transformed into the task of constructing an LFSR that takes the syndrome sequence as its output. One comment on this is that the LFSR may not be unique, and we want the *shortest* such LFSR (i.e., the smallest number of errors) in order to conform to the maximum-likelihood decoding principle (see Section 3.3.3).

The *Berlekamp-Massey* (*BM*) *algorithm* synthesizes such an LFSR iteratively [12, 13]. Starting with an LFSR that produces S_1, the LFSR is checked to see if it can also produce S_2. If it can, then the LFSR is not touched. Otherwise, the LFSR is modified such that it can also generate S_2. Next the LFSR is examined to see if it can also produce S_3. Again, if it can, the LFSR remains unchanged. Otherwise, the LFSR is updated so that it can also produce S_2. The procedure is carried out for $2t$ times. At the end, the LFSR is able to produce all of the $2t$ syndrome components. Because the algorithm guarantees that the resulting LFSR will be the shortest, the LFSR is the desired error location polynomial.

The BM algorithm defines a supporting polynomial $B(X)$ to assist in the updating of $\sigma(X)$. Denoting L as the length of the LFSR, a summary of the algorithm follows. The superscript (j) indicates the jth iteration. For example, $\sigma^{(j)}(X)$ means the error location polynomial at the jth iteration.

Berlekamp-Massey Algorithm

Initialization:

Set:

$$\sigma^{(0)}(X) = 1,\ B^{(0)}(X) = 1,\ L^{(0)} = 0, \text{ and } j = 1$$

Normal Operation: the jth iteration:

1. Compute the LFSR output:

$$\tilde{S}_j = -\sum_{i=1}^{L^{(j-1)}} \sigma_i^{(j-1)} S_{j-i}$$

2. Calculate the discrepancy[5]:

$$\Delta_j = S_j - \tilde{S}_j$$

3. Assign value to the variable δ:

$$\delta = \begin{cases} 1 & \Delta_j \neq 0 \text{ and } 2L^{(j-1)} \leq j-1 \\ 0 & \text{otherwise} \end{cases}$$

[5] Some of the literature skips the first step and computes the discrepancy directly as $\Delta_j = \sum_{i=0}^{L(j-1)} \sigma_i^{(j-1)} S_{j-i}$. This is justified by $\Delta_j = S_j - \tilde{S}_j = S_j - \sum_{i=0}^{L(j-1)} \sigma_i^{(j-1)} S_{j-i} = \sum_{i=0}^{L(j-1)} \sigma_i^{(j-1)} S_{j-i}$, in which the last equality holds due to $\sigma_0^{(j-1)} = 1$.

4. Update:

$$\begin{bmatrix} \sigma^{(j)}(X) \\ B^{(j)}(X) \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_j X \\ \Delta_j^{-1}\delta & (1-\delta)X \end{bmatrix} \cdot \begin{bmatrix} \sigma^{(j-1)}(X) \\ B^{(j-1)}(X) \end{bmatrix} \tag{4.17}$$

$$L^{(j)} = \delta(k - L^{(j-1)}) + (1 - \delta)L^{(j-1)}$$

5. If $j = 2t$, stop. Otherwise, $j = j + 1$ and return to step 1.

Note that $L^{(j)}$ is the degree of $\sigma^{(j)}(X)$. Figure 4.5 is a flowchart of the BM algorithm.

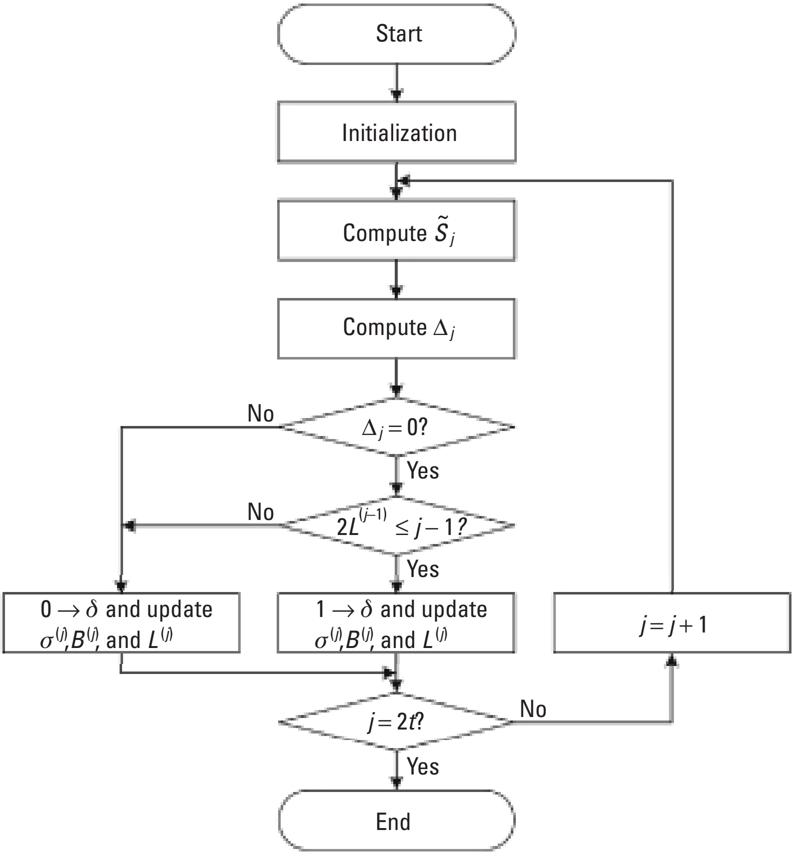

Start
Initialization
Compute $\tilde{S}_j$
Compute Δ_j
No
$\Delta_j = 0?$
Yes
No
$2L^{(j-1)} \le j-1?$
Yes
$0 \rightarrow \delta$ and update $\sigma^{(j)}, B^{(j)}$, and $L^{(j)}$
$1 \rightarrow \delta$ and update $\sigma^{(j)}, B^{(j)}$, and $L^{(j)}$
$j = j + 1$
$j = 2t?$
No
Yes
End

Figure 4.5 Flowchart of the Berlekamp-Massey algorithm.

Example 4.4

Again, we use the (7,3) RS code. Suppose that the code polynomial $c(X) = X^6 + \alpha^3X^5 + \alpha^5X^4 + \alpha^3X^3 + \alpha^6X^2 + \alpha^5X + 1$ *was transmitted. The received polynomial* $r(X) = X^6 + \alpha^3X^5 + X^4 + \alpha^3X^3 + \alpha^3X^2 + \alpha^5X + 1$ *contains two errors. The error polynomial is* $e(X) = \alpha^4X^4 + X^2$*. Compute the syndrome as follows:*

$$S_1 = r(\alpha) = \alpha^4,\ S_2 = r(\alpha^2) = 1,\ S_3 = r(\alpha^3) = 1,\ S_4 = r(\alpha^4) = \alpha^5$$

Now we use the BM algorithm to find the error location polynomial for the received polynomial:

Initialization: $\sigma^{(0)}(X) = 1, B^{(0)}(X) = 1, L^{(0)} = 0$, and $j = 1$

$j = 1$: $\tilde{S}_1 = 0 \Rightarrow \Delta_1 = \alpha^4, 2L^{(0)} = 0 = j - 1 \Rightarrow \delta = 1$

$$\begin{bmatrix} \sigma^{(1)}(X) \\ B^{(1)}(X) \end{bmatrix} = \begin{bmatrix} 1 & -\alpha^4X \\ \alpha^{-4} & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 - \alpha^4X \\ \alpha^{-4} \end{bmatrix}$$

$$L^{(1)} = \left(j - L^{(0)}\right) = 1$$

$j = 2$: $\tilde{S}_2 = \sigma_1^{(1)}S_1 = \alpha \Rightarrow \Delta_2 = \alpha^3, 2L^{(1)} = 2 > j - 1 \Rightarrow \delta = 0$

$$\begin{bmatrix} \sigma^{(2)}(X) \\ B^{(2)}(X) \end{bmatrix} = \begin{bmatrix} 1 & -\alpha^3X \\ 0 & X \end{bmatrix} \cdot \begin{bmatrix} 1 - \alpha^4X \\ \alpha^{-4} \end{bmatrix} = \begin{bmatrix} 1 - \alpha^3X \\ \alpha^{-4}X \end{bmatrix}$$

$$L^{(2)} = L^{(1)} = 1$$

$j = 3$: $\tilde{S}_3 = \sigma_1^{(2)}S_2 = \alpha^3 \Rightarrow \Delta_3 = \alpha, 2L^{(2)} = 2 = j - 1 \Rightarrow \delta = 1$

$$\begin{bmatrix} \sigma^{(3)}(X) \\ B^{(3)}(X) \end{bmatrix} = \begin{bmatrix} 1 & -\alpha X \\ \alpha^{-1} & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 - \alpha^3X \\ \alpha^{-4}X \end{bmatrix} = \begin{bmatrix} 1 - \alpha^3X - \alpha^{-3}X^2 \\ \alpha^{-1} - \alpha^2X \end{bmatrix}$$

$$L^{(3)} = \left(j - L^{(2)}\right) = 2$$

$j = 4$: $\tilde{S}_4 = \sigma_1^{(3)}S_3 + \sigma_2^{(3)}S_2 = \alpha^6 \Rightarrow \Delta_4 = \alpha, 2L^{(3)} = 4 > j - 1 \Rightarrow \delta = 0$

$$\begin{bmatrix} \sigma^{(4)}(X) \\ B^{(4)}(X) \end{bmatrix} = \begin{bmatrix} 1 & -\alpha X \\ 0 & X \end{bmatrix} \cdot \begin{bmatrix} 1 - \alpha^3X - \alpha^{-3}X^2 \\ \alpha^{-1} - \alpha^2X \end{bmatrix} = \begin{bmatrix} 1 - \alpha X - \alpha^6X^2 \\ \alpha^{-1}X - \alpha^2X^2 \end{bmatrix}$$

$$L^{(4)} = L^{(3)} = 2$$

Because $\sigma(X) = \sigma^{(4)}(X) = 1 - \alpha X - \alpha^6 X^2 = (1 - \alpha^2 X)(1 - \alpha^4 X)$*, the two errors are located at positions* 2 *and* 4.

Once the error location polynomial has been found, we can construct the error evaluation polynomial using (4.3). Considering Example 4.4, for instance, the error evaluation polynomial is computed to be:

$$\Omega(X) = S(X)\sigma(X) \bmod X^{2t} = (\alpha^4 + X + X^2 + \alpha^5 X^3)(1 - \alpha X - \alpha^6 X^2) \bmod X^4 = \alpha^4 + \alpha^4 X$$

MATLAB Experiment 4.5

The m-file `bmdemo.m*` in this book's DVD finds the error location and error evaluation polynomials for the above example using the BM algorithm.

```
>> bmdemo
sigma =
  0   1   6
omega =
        4   4
```

Comment: Extending the program to other RS codes should be straightforward.

The Berlekamp-Massey algorithm involves the inversion Δ_j^{-1} at each iteration. We know that Galois field inversion is a costly operation. In view of this, some inversion-free variations on the BM algorithm (referred to as iBM) have been proposed [14, 15], leading to simplified and more structured hardware designs.

The basic idea of the iBM algorithm is straightforward. Reexamining (4.17), we find that Δ_j^{-1} is needed to compute $B^{(j)}(X)$ when $\delta = 1$:

$$B^{(j)}(X) = \Delta_j^{-1}\sigma^{(j-1)}(X)$$

If we scale $B^{(j)}(X)$ as follows when $\delta = 1$, the inversion operation is avoided:

$$B^{(j)}(X) = \Delta_j \bullet \left[\Delta_j^{-1}\sigma^{(j-1)}(X)\right] = \sigma^{(j-1)}(X)$$

Notice that the scaled version $B^{(j)}(X)$ will be added to $\sigma^{(j+1)}(X)$ at the next iteration; we therefore need to multiply $\sigma^{(j+1)}(X)$ by Δ_j at the $(j + 1)$th iteration. Summarizing the above, we have the following *inversionless Berlekamp-Massey algorithm*.

Inversionless Berlekamp-Massey Algorithm

Initialization:

Set

$$\sigma^{(0)}(X) = 1,\ B^{(0)}(X) = X,\ \theta^{(0)} = 1,\ L^{(0)} = 0 \text{ and } j = 1$$

Normal Operation: the *j*th iteration:

1. Calculate the discrepancy between the syndrome and the LFSR output:

$$\Delta_j = \sum\nolimits_{i=0}^{L^{(j-1)}} \sigma_i^{(j-1)} S_{j-i}$$

2. Assign value to the variable δ:

$$\delta = \begin{cases} 1 & \Delta_j \neq 0 \text{ and } 2L^{(j-1)} \leq j-1 \\ 0 & \text{otherwise} \end{cases}$$

3. Update:

$$\begin{bmatrix} \sigma^{(j)}(X) \\ B^{(j)}(X) \end{bmatrix} = \begin{bmatrix} \theta^{(j-1)} & -\Delta_j X \\ \delta & (1-\delta)X \end{bmatrix} \cdot \begin{bmatrix} \sigma^{(j-1)}(X) \\ B^{(j-1)}(X) \end{bmatrix} \tag{4.18}$$

$$\theta^{(j)} = \delta\Delta_j + (1-\delta)\ \theta^{(j-1)} \text{ and}$$
$$L^{(j)} = \delta(k - L^{(j-1)}) + (1-\delta)L^{(j-1)}$$

4. If $j = 2t$, stop. Otherwise, $j = j + 1$ and go to step 1.

The variable $\theta^{(j-1)}$ in (4.18) controls the multiplication of $\sigma^{(j)}(X)$ by Δ_{j-1}. The new error location polynomial $\sigma(X)$ resulting from the inversionless algorithm has the same roots as the error location polynomial obtained from the original BM algorithm. Because it is the roots of $\sigma(X)$, not $\sigma(X)$ itself, that specify the error positions, the new algorithm should not change the decoding outcome.

Example 4.5

We now apply the inversionless BM algorithm to the previous example.

Initialization: $\sigma^{(0)}(X)=1, B^{(0)}(X)=1, \theta^{(0)}=1, L^{(0)}=0$, and $j=1$

$j=1$: $\Delta_1=\sigma_0^{(0)}S_1=\alpha^4, 2L^{(0)}=0=j-1\Rightarrow\delta=1$

$$\begin{bmatrix}\sigma^{(1)}(X)\\ B^{(1)}(X)\end{bmatrix}=\begin{bmatrix}1 & -\alpha^4X\\ 1 & 0\end{bmatrix}\cdot\begin{bmatrix}1\\ 1\end{bmatrix}=\begin{bmatrix}1-\alpha^4X\\ 1\end{bmatrix}$$

$$\theta^{(1)}=\Delta=\alpha^4$$

$$L^{(1)}=(j-L^{(0)})=1$$

$j=2$: $\Delta_2=\sigma_0^{(1)}S_2+\sigma_1^{(1)}S_1=\alpha^3, 2L^{(1)}=2>j-1\Rightarrow\delta=0$

$$\begin{bmatrix}\sigma^{(2)}(X)\\ B^{(2)}(X)\end{bmatrix}=\begin{bmatrix}\alpha^4 & -\alpha^3X\\ 0 & X\end{bmatrix}\cdot\begin{bmatrix}1-\alpha^4X\\ 1\end{bmatrix}=\begin{bmatrix}\alpha^4-X\\ X\end{bmatrix}$$

$$\theta^{(2)}=\theta^{(1)}=\alpha^4$$

$$L^{(2)}=L^{(1)}=1$$

$j=3$: $\Delta_3=\sigma_0^{(2)}S_3+\sigma_1^{(2)}S_2=\alpha^5, 2L^{(2)}=2=j-1\Rightarrow\delta=1$

$$\begin{bmatrix}\sigma^{(3)}(X)\\ B^{(3)}(X)\end{bmatrix}=\begin{bmatrix}\alpha^4 & -\alpha^5X\\ 1 & 0\end{bmatrix}\cdot\begin{bmatrix}\alpha^4-X\\ X\end{bmatrix}=\begin{bmatrix}\alpha-\alpha^4X-\alpha^5X^2\\ \alpha^4-X\end{bmatrix}$$

$$\theta^{(3)}=\Delta=\alpha^5$$

$$L^{(3)}=(j-L^{(2)})=2$$

$k=4$: $\Delta_4=\sigma_0^{(3)}S_4+\sigma_1^{(3)}S_3+\sigma_2^{(3)}S_2=\alpha^2, 2L^{(3)}=4>j-1\Rightarrow\delta=0$

$$\begin{bmatrix}\sigma^{(4)}(X)\\ B^{(4)}(X)\end{bmatrix}=\begin{bmatrix}\alpha^5 & -\alpha^2X\\ 0 & X\end{bmatrix}\cdot\begin{bmatrix}\alpha-\alpha^4X-\alpha^5X^2\\ \alpha^4-X\end{bmatrix}=\begin{bmatrix}\alpha^6-X-\alpha^5X^2\\ \alpha^4X-X^2\end{bmatrix}$$

Note that $\sigma(X)-\sigma^{(4)}(X)=\alpha^6-X-\alpha^5X^2$ can be factorized as $\alpha^6(1-\alpha X-\alpha^6X^2)$, where $1-\alpha X-\alpha^6X^2$ is the error location polynomial obtained from the original BM algorithm in the last example. Beyond doubt, $\sigma(X)$ has the same roots.

4.2.2.2 Euclid's Method

Besides the Berlekamp-Massey algorithm, an error location polynomial can also be constructed using Euclid's algorithm [16]. *Euclid's algorithm*, attributed to mathematician Euclid, was originally formulated to find the greatest common divisor (GCD) of two positive integers. To obtain (a, b), the GCD of two integers a and b, the algorithm divides a by b to get a remainder d_1, then divides b by d_1 to get a new remainder d_2, and afterward divides d_1 by $d_2 \cdots$. This simple division of the latest divisor by the latest remainder is repeated until the remainder becomes zero. The latest nonzero remainder is (a, b).

Example 4.6

Find greatest common divisor of 469 and 161.

Step 1. Divide 469 by 161: $469 = 161 \times 2 + 147$

Step 2. Divide 161 by the remainder 147: $161 = 147 \times 1 + 14$

Step 3. Divide the latest divisor 147 by the latest remainder 14: $147 = 14 \times 10 + 7$

Step 4. Divide the latest divisor 14 by the latest remainder 7: $14 = 7 \times 2 + 0$

So (469,161) is 7, which is the latest nonzero remainder.

It has been observed that if $d = (a, b)$, then two numbers f and g exist such that [17]:

$$fa + gb = d \tag{4.19}$$

Given a, b, and d, f and g can be found by executing the above procedure backwards.

Example 4.7

Find f and g such that (469,161) = 469f + 161g.

We perform the procedure in the previous example backwards, starting from step 3 above:

Step a. $7 = 147 - 14 \cdot 10$

Step b. $14 = 161 - 147 \cdot 1$

$\Rightarrow 7 = 147 - (161 - 147 \cdot 1) \cdot 10$

Step c. $147 = (469 - 161 \cdot 2)$

$\Rightarrow 7 = (469 - 161 \cdot 2) - (161 - (469 - 161 \cdot 2) \cdot 1) \cdot 10$

$\Rightarrow 7 = 11 \cdot 469 - 32 \cdot 161$

Therefore, f = 11 and g = –32.

Euclid's algorithm is not limited to integers; it is applicable to polynomials as well. In that case $a = a(X)$ and $b = b(X)$ are two polynomials, and $d = d(X)$, $f = f(X)$, and $g = g(X)$ are also polynomials. The following lists the procedure for Euclid's algorithm when using it for polynomials. The superscript (j) denotes the jth iteration.

Euclid's Algorithm for Polynomial

Initialization:

Set

$f^{(-1)}(X) = g^{(0)}(X) = 1, f^{(0)}(X) = g^{(-1)}(X) = 0,\ d^{(-1)}(X) = a(X),\ d^{(0)}(X) = b(X)$ and $j = 1$

Normal Operation: the jth iteration:

1. $q^{(j)}(X) = \text{Quotient}[d^{(j-2)}(X)/d^{(j-1)}(X)]$
2. $d^{(j)}(X) = d^{(j-2)}(X) - q^{(j)}(X)d^{(j-1)}(X)$
   $f^{(j)}(X) = f^{(j-2)}(X) - q^{(j)}(X)f^{(j-1)}(X)$
   $g^{(j)}(X) = g^{(j-2)}(X) - q^{(j)}(X)g^{(j-1)}(X)$
3. If $d^{(j)}(X) \neq 0$, set $j = j + 1$ and go back to step 1. Otherwise, stop and $d^{(j-1)}(X) = (a, b)$

Note that at each iteration the relationship $f^{(j)}a + g^{(j)}b = d^{(j)}$ is always maintained.

Example 4.8

Let $a(X) = X^3 + 1$, $b(X) = X^2 + 1$. The process of Euclid's algorithm is demonstrated in Table 4.1, and (a,b) is $X + 1$.

Table 4.1
Example Process of Euclid's Algorithm

j	$q^{(j)}(X)$	$d^{(j)}(X)$	$f^{(j)}(X)$	$g^{(j)}(X)$	Note
–1	—	$X^3 + 1$	1	0	Initialization
0	—	$X^2 + 1$	0	1	
1	X	$X + 1$	1	X	
2	$X + 1$	0	$X + 1$	$X^2 + X + 1$	Zero remainder

MATLAB Experiment 4.6

Two MATLAB functions, `gcdnum*` and `gcdpoly*`, are included in the DVD that accompanies this book. They are used to compute the GCD of two integers and of two polynomials, respectively. For example, (469, 161) is found as:

```
>> gcd = gcdnum(469,161)
>> gcd =
     7
```

Recall that the error location polynomial satisfies (4.3). Reformulate (4.3) as:

$$S(X)\sigma(X) - X^{2t}\pi(X) = -\Omega(X) \tag{4.20}$$

where $\pi(X)$ is the quotient polynomial of $S(X)\sigma(X)/X^{2t}$. Let $a = S(X)$, $b = -X^{2t}$. The error location polynomial $\sigma(X)$ and the error evaluation polynomial $\Omega(X)$ can readily be found [together with $\pi(X)$] using the recursive Euclid's algorithm. Notice that by definition $\Omega(X)$ is of degree less than t. The decoding process therefore should terminate as soon as the degree of $\Omega(X)$ becomes less than t, instead of when $\Omega(X) = 0$. *Euclid's method* for constructing the error location polynomial is summarized next in matrix form.

Euclid's Method for RS Decoding

Initialization:

Set

$$R^{(0)}(X) = X^{2t}, T^{(0)}(X) = S(X), \boldsymbol{A}^{(0)}(X) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \text{ and } j = 1$$

Normal Operation: the *j*th iteration:

1. $Q(j)(X) = \text{Quotient } [R^{(j-1)}(X)/T^{(j-1)}(X)]$

$$\boldsymbol{A}^{(j)}(X) = \begin{bmatrix} 0 & 1 \\ 1 & Q^{(j)}(X) \end{bmatrix} \cdot \boldsymbol{A}^{(j-1)}(X)$$

$$\begin{bmatrix} R^{(j)}(X) \\ T^{(j)}(X) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & Q^{(j)}(X) \end{bmatrix} \cdot \begin{bmatrix} R^{(j-1)}(X) \\ T^{(j-1)}(X) \end{bmatrix}$$

2. If the degree of $T^{(j+1)}(X)$ is less than t, stop and compute the following:
   $U = A_{22}^{(j)}(0)$, where $A_{22}^{(j)}(X)$ is the element of $A^{(j)}(X)$ in the second row and second column,
   $\sigma(X) = U^{-1} \cdot A_{22}^{(j)}(X)$
   $\Omega(X) = U^{-1} \cdot T^{(j)}(X)$
   Otherwise, $j = j + 1$ and go back to step 1.

Example 4.9

Let us now construct the error location polynomial from Example 4.4 *using Euclid's method. The syndrome polynomial is* $S(X) = \alpha^4 + X + X^2 + \alpha^5 X^3$.

Initialization: $R^{(0)}(X) = X^4, T^{(0)}(X) = \alpha^4 + X + X^2 + \alpha^5 X^3$

$$\boldsymbol{A}^{(0)}(X) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \text{and } j = 1$$

$j = 1$: $Q^{(1)}(X) = \text{Quotient}\,[R^{(0)}(X)/T^{(0)}(X)] = \alpha^2 X + \alpha^4$

$$\boldsymbol{A}^{(1)}(X) = \begin{bmatrix} 0 & 1 \\ 1 & Q^{(1)}(X) \end{bmatrix} \cdot \boldsymbol{A}^{(0)}(X) = \begin{bmatrix} 0 & 1 \\ 1 & \alpha^2 X + \alpha^4 \end{bmatrix}$$

$$\begin{bmatrix} R^{(1)}(X) \\ T^{(1)}(X) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & Q^{(1)}(X) \end{bmatrix} \cdot \begin{bmatrix} R^{(0)}(X) \\ T^{(0)}(X) \end{bmatrix} = \begin{bmatrix} \alpha^4 + X + X^2 + \alpha^5 X^3 \\ \alpha + \alpha^3 X + \alpha X^2 \end{bmatrix}$$

$j = 2$: $Q^{(2)}(X) = \text{Quotient}\,[R^{(0)}(X)/T^{(0)}(X)] = \alpha^4 X$

$$\boldsymbol{A}^{(2)}(X) = \begin{bmatrix} 0 & 1 \\ 1 & Q^{(2)}(X) \end{bmatrix} \cdot \boldsymbol{A}^{(1)}(X) = \begin{bmatrix} 1 & \alpha^2 X + \alpha^4 \\ \alpha^4 X & 1 + \alpha X + \alpha^6 X^2 \end{bmatrix}$$

$$\begin{bmatrix} R^{(2)}(X) \\ T^{(2)}(X) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & Q^{(2)}(X) \end{bmatrix} \cdot \begin{bmatrix} R^{(1)}(X) \\ T^{(1)}(X) \end{bmatrix} = \begin{bmatrix} \alpha + \alpha^3 X + \alpha X^2 \\ \alpha^4 + \alpha^4 X \end{bmatrix}$$

Note that $U = A_{22}^{(2)}(0) = 1 \Rightarrow \sigma(X) = U^{-1} \cdot A_{22}^{(2)}(X) = 1 + \alpha X + \alpha^6 X^2$, *which is completely identical to what was obtained with the BM algorithm:* $\Omega(X) = U^{-1} \cdot T^{(2)}(X) = \alpha^4 + \alpha^4 X$.

MATLAB Experiment 4.7

The MATLAB m-file `eucdemo.m*` uses Euclid's method to generate the error location and error evaluation polynomials for Example 4.9. The program can easily be modified for RS codes with other parameters.

```
>> eucdemo
sigma =
  0  1  6
omega =
     4  4
```

Compared with the Berlekamp-Massey algorithm, Euclid's method has the advantage of obtaining the error evaluator polynomial $\Omega(X)$ automatically as a by-product of the process. Also, all of the steps in the Euclid's method are identical, which translates into a more structured hardware implementation. However, as far as the number of $GF(2^m)$ operations is concerned, the Berlekamp-Massey algorithm is somewhat more efficient. Both algorithms provide identical error correction performances.

4.2.3 Frequency-Domain Decoding

Before presenting the frequency-domain decoding of RS codes, we have an important GFFT property to introduce. Let $\boldsymbol{x} = (x_0\, x_1\, x_2\, \cdots\, x_{n-1})$ and $\boldsymbol{y} = (y_0\, y_1\, y_2\, \cdots\, y_{n-1})$ be two vectors over $GF(q)$, and $\boldsymbol{X} = (X_0\, X_1\, X_2\, \cdots\, X_{n-1})$ and $\boldsymbol{Y} = (Y_0\, Y_1\, Y_2\, \cdots\, Y_{n-1})$ be the GFFTs of $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively. Then the GFFT of the inner product of $\boldsymbol{x}$ and $\boldsymbol{y}$ is the convolution of $\boldsymbol{X}$ and $\boldsymbol{Y}$ [18, p. 184]:

$$\begin{aligned}\mathcal{F}(\boldsymbol{x}\cdot\boldsymbol{y}) &= \mathcal{F}[(x_0y_0\ x_1y_1 \cdots x_{n-1}y_{n-1})] \\ &= \boldsymbol{X} \circledast \boldsymbol{Y} = (\Phi_0\,\Phi_1 \cdots \Phi_{n-1})\end{aligned} \tag{4.21}$$

where$\circledast$ represents cyclic convolution, and

$$\Phi_i = \frac{1}{n \bmod \lambda}\sum_{j=0}^{n-1} X_j Y_{i-j} \qquad (i = 0,1,2,\cdots,n-1) \tag{4.22}$$

The subscript $i - j$ is computed modulo n, that is, it "wraps around" in a cyclic fashion. The parameter λ is the characteristic of the Galois field

GF(*q*) (see Section 2.2.1 in Chapter 2). We notice that (4.22) is analogous to the convolution property of the DFT in signal processing. In fact, the GFFT and DFT share many properties: the linearity property, similarity property, and so forth.

Now let $\boldsymbol{r} = (r_0\ r_1\ r_2\ \cdots\ r_{n-1})$ be the received word. The frequency-domain RS decoding proceeds with computing the GFFT of $\boldsymbol{r}$, $\boldsymbol{R} = (R_0\ R_1\ R_2 \cdots R_{n-1})$:

$$R_i = r(\alpha^i) = \sum\nolimits_{j=0}^{n-1} r_j \alpha^{ij} \qquad (i = 0,1,2,\cdots,n-1) \tag{4.23}$$

Notice that $\boldsymbol{r} = \boldsymbol{c} + \boldsymbol{e}$. Follow the linearity property of GFFT, we have:

$$\boldsymbol{R} = \boldsymbol{C} + \boldsymbol{E} \tag{4.24}$$

where $\boldsymbol{C} = (C_0\ C_1\ C_2\ \cdots\ C_{n-1})$ and $\boldsymbol{E} = (E_0\ E_1\ E_2\ \cdots\ E_{n-1})$ are the GFFT of $\boldsymbol{c}$ and $\boldsymbol{e}$, respectively.

According to (3.64), the syndrome of $\boldsymbol{r}$ is computed as:

$$S_i = r(\alpha^i) = \sum\nolimits_{j=0}^{n-1} r_j \alpha^{ij} \qquad (i = 1,2,\cdots,2t) \tag{4.25}$$

Comparing (4.25) with (4.23), we can conclude:

$$S_i = R_i \qquad (i = 1,2,\cdots,2t) \tag{4.26}$$

Considering that an RS code in the frequency domain contains $2t$ zeros (i.e., $C_i = 0$ for $i = 1, 2, \cdots 2t$), we have:

$$S_i = E_i \qquad (i = 1,2,\cdots,2t) \tag{4.27}$$

This equation tells us that the $2t$ elements of error pattern in frequency-domain, E_i $(i = 1, 2, \cdots, 2t)$, are the $2t$ components of the syndrome. As such, all we need to do next for the decoding is to find the errors E_i for $i = 0, 2t + 1, 2t + 2, \cdots, n - 1$.

Suppose that $\nu \le t$ errors have occurred. Then the corresponding error polynomial $e(X)$ is:

$$e(X) = e_{j_1} X^{j_1} + e_{j_2} X^{j_2} + \cdots + e_{j_\nu} X^{j_\nu} \tag{4.28}$$

On the other hand, following (3.70), the error location polynomial is formed as:

$$\sigma(X) = \sigma_0 + \sigma_1 X + \cdots + \sigma_\nu X^\nu = (1 - \alpha^{j_1} X)\ (1 - \alpha^{j_2} X) \cdots (1 - \alpha^{j_\nu} X)$$

where $j_1, j_2, \cdots j_v$ are the locations of the v errors, and

$$\sigma(\alpha^{-j_i}) = 0 \qquad (i = 1, 2, \cdots, v) \tag{4.29}$$

Let $\Lambda = (\Lambda_0\ \Lambda_1\ \cdots\ \Lambda_v)$ be the inverse GFFT of the vector $\sigma = (\sigma_0\ \sigma_1\ \cdots\ \sigma_v)$, where:

$$\Lambda_k = \left(\frac{1}{n \bmod \lambda}\right)\sum_{i=0}^{v} \sigma_i \alpha^{-ik} = \left(\frac{1}{n \bmod \lambda}\right)\sigma(\alpha^{-k}) \qquad (k = 0, 1, 2, \cdots, v) \tag{4.30}$$

It follows from (4.29) and (4.30) that:

$$\Lambda_{j_i} = \left(\frac{1}{n \bmod \lambda}\right)\sigma(\alpha^{-j_i}) = 0 \qquad (i = 1, 2, \cdots, v) \tag{4.31}$$

which indicates that $\Lambda_k = 0$ wherever $e_k \neq 0$. In other words, either Λ_k or e_k, or maybe both, are zero. As an immediate consequence, $\Lambda_k\, e_k = 0$ for $k = 1, 2, \cdots, v - 1$. By expanding Λ into $\Lambda = (\Lambda_0\ \Lambda_1\ \cdots\ \Lambda_{v-1}\Lambda_v\Lambda_{v+1}\ \cdots\ \Lambda_{n-1})$ with $\Lambda_k \triangleq 0$ for $k \geq v$, the following exists:

$$\boldsymbol{\Lambda} \cdot \boldsymbol{e} = \Lambda_0 e_0 + \Lambda_1 e_1 + \cdots + \Lambda_{n-1} e_{n-1} = 0 \tag{4.32}$$

Taking the GFFT of both sides of (4.32), we obtain:

$$\boldsymbol{\sigma} \circledast \boldsymbol{E} = 0 \text{ or } \sum_{j=0}^{n-1} \sigma_j E_{i-j} = 0 \qquad (i = 0, 1, 2, \cdots, n-1) \tag{4.33}$$

Since the degree of $\sigma(X)$ is v, (4.33) shrinks to:

$$\sum_{j=0}^{v} \sigma_j E_{i-j} = 0 \qquad (i = 0, 1, 2, \cdots, n-1) \tag{4.34}$$

Considering $\sigma_0 = 1$, (4.34) can be reformulated as:

$$E_i = -(\sigma_1 E_{i-1} + \sigma_2 E_{i-2} + \cdots + \sigma_v E_{i-v}) \qquad (i = 0, 1, 2, \cdots, n-1) \tag{4.35}$$

Based on the known errors $E_1, E_2, \cdots, E_{2t}$ [see (4.27)], we readily find the errors E_i for $i = 0, 2t + 1, 2t + 2, \cdots, n - 1$, by recursively computing (4.35). Setting $i = v$ in (4.35) yields:

$$E_v = -(\sigma_1 E_{v-1} + \sigma_2 E_{v-2} + \cdots + \sigma_v E_0)$$

Solving the equation for E_0, we obtain:

$$E_0 = -(1/\sigma_v)(E_v + \sigma_1 E_{v-1} + \cdots + \sigma_{v-1}E_1) \tag{4.36}$$

Up to now we have found solutions to all $2t$ elements of $\boldsymbol{E}$.

With $\boldsymbol{E}$ ready, the error vector $\boldsymbol{e}$ in the time domain is obtained by taking the inverse GFFT of $\boldsymbol{E}$:

$$\boldsymbol{e} = \mathcal{F}^{-1}(\boldsymbol{E}) \tag{4.37}$$

The error location polynomial is found as before (using either the Berlekamp-Massey algorithm or the Euclid's method).

RS decoding in the frequency domain offers certain advantages, since we only need to find errors E_i for $i = 0, 2t + 1, 2t + 2, \cdots, n - 1$ instead of for the whole codeword.

Example 4.10

Suppose that $\boldsymbol{c} = (\alpha^4\ 1\ 1\ \alpha^3\ \alpha^2\ \alpha^4\ \alpha^3)$, *the codeword in Example 4.3, has been transmitted, and the corresponding received word* $\boldsymbol{r} = (\alpha^4\ \alpha\ 1\ \alpha^3\ \alpha^2\ \alpha^6\ \alpha^3)$ *contains two errors. Now we decode it in the frequency domain.*

Based on (4.10), we calculate the GFFT of $\boldsymbol{r}$:

$$R_0 = \alpha^4(\alpha^0)^0 + \alpha(\alpha^0)^1 + 1(\alpha^0)^2 + \alpha^3(\alpha^0)^3 + \alpha^2(\alpha^0)^4$$
$$+\alpha^6(\alpha^0)^5 + \alpha^3(\alpha^0)^6 = \alpha^2$$

$$R_1 = \alpha^4(\alpha^1)^0 + \alpha(\alpha^1)^1 + 1(\alpha^1)^2 + \alpha^3(\alpha^1)^3 + \alpha^2(\alpha^1)^4$$
$$+\alpha^6(\alpha^1)^5 + \alpha^3(\alpha^1)^6 = \alpha^2$$

$$R_2 = \alpha^4(\alpha^2)^0 + \alpha(\alpha^2)^1 + 1(\alpha^2)^2 + \alpha^3(\alpha^2)^3 + \alpha^2(\alpha^2)^4$$
$$+\alpha^6(\alpha^2)^5 + \alpha^3(\alpha^2)^6 = \alpha$$

$$R_3 = \alpha^4(\alpha^3)^0 + \alpha(\alpha^3)^1 + 1(\alpha^3)^2 + \alpha^3(\alpha^3)^3 + \alpha^2(\alpha^3)^4$$
$$+\alpha^6(\alpha^3)^5 + \alpha^3(\alpha^3)^6 = \alpha^3$$

$$R_4 = \alpha^4(\alpha^4)^0 + \alpha(\alpha^4)^1 + 1(\alpha^4)^2 + \alpha^3(\alpha^4)^3 + \alpha^2(\alpha^4)^4$$
$$+\alpha^6(\alpha^4)^5 + \alpha^3(\alpha^4)^6 = \alpha^6$$
$$R_5 = \alpha^4(\alpha^5)^0 + \alpha(\alpha^5)^1 + 1(\alpha^5)^2 + \alpha^3(\alpha^5)^3 + \alpha^2(\alpha^5)^4$$
$$+\alpha^6(\alpha^5)^5 + \alpha^3(\alpha^5)^6 = 0$$
$$R_6 = \alpha^4(\alpha^6)^0 + \alpha(\alpha^6)^1 + 1(\alpha^6)^2 + \alpha^3(\alpha^6)^3 + \alpha^2(\alpha^6)^4$$
$$+\alpha^6(\alpha^6)^5 + \alpha^3(\alpha^6)^6 = \alpha$$

The syndrome polynomial is obtained as:

$$S(X) = \alpha^2 + \alpha X + \alpha^3 X^2 + \alpha^6 X^3$$

Using the BM algorithm or the Euclid's method, we get:

$$\sigma(X) = 1 - \alpha^6 X - \alpha^6 X^2$$

Based on (4.35), E_5 and E_6 are computed:

$$E_5 = -(\sigma_1 E_4 + \sigma_2 E_3) = -(-\alpha^6\alpha^6 - \alpha^6\alpha^3) = \alpha^3, \textit{ and}$$
$$E_6 = -(\sigma_1 E_5 + \sigma_2 E_4) = -(-\alpha^6\alpha^3 - \alpha^6\alpha^6) = \alpha^3$$

E_0 is obtained to be:

$$E_0 = (1/\sigma_2)\,(E_2 + \sigma_1 E_1) = \alpha^{-6}(\alpha - \alpha^2\alpha^6) = 0$$

Finally the decoded word is:

$$\tilde{\boldsymbol{C}} = \boldsymbol{R} + \boldsymbol{E} = (\alpha^2\ 0\ 0\ 0\ 0\ \alpha^3\ 1)$$

which is identical to the frequency-domain codeword obtained in Example 4.3.

4.2.4 Error and Erasure Decoding

Up until now we have limited RS decoding to error-only decoding. However, an RS code can correct any combination of ν errors and ζ erasures, provided:

$$\nu + \zeta / 2 \leq t \tag{4.38}$$

To correct both errors and erasures, we replace the erasures in $r(X)$ with some arbitrary elements from $GF(2^m)$, say, zeros, and form a new received polynomial $r'(X)$. By doing so, we are then able to apply the previously developed decoding algorithm to it.

Suppose that the received polynomial $r(X)$ contains ν errors at the locations $i_1, i_2, \cdots, i_\nu$ and ζ erasures at the locations $l_1, l_2, \cdots, l_\zeta$. Similar to the error location polynomial, we define an *erasure location polynomial* as follows:

$$\Gamma(X) \triangleq (1-\rho_0 X)(1-\rho_1 X)\cdots(1-\rho_\zeta X) \tag{4.39}$$

where $\rho_k = \alpha^{l_k}$ represents the erasure location l_k (just as $\beta_k = \alpha^{i_k}$ represents the error location i_k).

Then, as we just said, we replace the erasures in $r(X)$ with zeros and form $r'(X)$. Notice that the total number of errors in $r'(X)$, in the worst case, can be $\nu + \zeta$. This occurs when the true symbols corresponding to the erasures are all nonzeros. The error location polynomial for $r'(X)$ can thus be expressed as:

$$\begin{aligned}\Phi(X) &= \Phi_0 + \Phi_1 X + \Phi_2 X^2 + \cdots + \Phi_{\nu+\zeta} X^{\nu+\zeta} \\ &= \underbrace{(1+\beta_0 X)(1-\beta_1 X)\cdots(1-\beta_\nu X)}_{\sigma(X)}\underbrace{(1+\rho_0 X)(1-\rho_1 X)\cdots(1-\rho_\zeta X)}_{\Gamma(X)}\end{aligned} \tag{4.40}$$

where $\sigma(X) = (1-\beta_0 X)\,(1-\beta_1 X)\cdots(1-\beta_\nu X)$ is the error location polynomial corresponding to the ν true errors.

Afterwards, we construct the key equation as follows:

$$\Xi(X) = \Phi(X)S(X) \bmod X^{2t} = \sigma(X)\Gamma(X)S(X) \bmod X^{2t} \tag{4.41}$$

where the syndrome $S(X)$ is:

$$S_k = r'(\alpha^k) = \sum\nolimits_{i=1}^{\upsilon} e_{j_i}\beta_i^k + \sum\nolimits_{i=1}^{\zeta} w_{l_i}\rho_i^k \qquad (k = 1, 2, \cdots, 2, t) \tag{4.42}$$

where e_{j_i} is the magnitude of the original error at location j_i, and w_{l_i} is the magnitude of the error at erasure location l_i. Letting

$$\Theta(X) = \Gamma(X)S(X) \bmod X^{2t} \tag{4.43}$$

Equation (4.41) becomes:

$$\Xi(X) = \sigma(X)\Theta(X) \bmod X^{2t} \tag{4.44}$$

The key equation in (4.44) involves erasures and is called the *Berlekamp-Forney key equation*. The most important observation is that (4.44) has exactly the same form as the Forney key equation in (4.3) except the syndrome $S(X)$ is replaced by $\Theta(X)$. As a result, any decoding algorithm presented earlier (e.g., Berlekamp-Massey, Euclid's) can be used to solve (4.44) for $\sigma(X)$ and $\Xi(X)$ [5, p. 268]. If the BM algorithm is used, then the coefficients of $\Theta(X)$, Θ_0, Θ_1, $\cdots$, Θ_{2t-1}, are used in place of the components of the syndrome $S(X)$, S_1, S_2, $\cdots$, S_{2t}. If instead Euclid's method is employed, then $a = X^{2t}$ and $b = \Theta(X)$. Once $\sigma(X)$ and $\Xi(X)$ are found, the error/erasure magnitude can be calculated by using the following modified Forney algorithm:

$$\begin{aligned} e_{j_i} &= -\Xi(X)/\Phi'(X)\big|_{X=\beta_i^{-1}} \qquad (i = 1, 2, \cdots, \nu) \\ w_{j_i} &= -\Xi(X)/\Phi'(X)\big|_{X=\rho_i^{-1}} \qquad (i = 1, 2, \cdots, \zeta) \end{aligned} \tag{4.45}$$

Example 4.11

We use the (7,3) example RS code to show how error and erasure decoding is performed. Suppose that the code polynomial $c(X) = X^6 + \alpha^3X^5 + \alpha^5X^4 + \alpha^3X^3 + \alpha^6X^2 + \alpha^5X + 1$ *was transmitted. The received polynomial* $r(X) = X^6 + \alpha^3X^5 + X^4 + \alpha^3X^3 + \alpha^6X^2 + \times X + 1$ *contains one error (the term* X^4*) and one erasure (marked with ×).*

First let us replace the erasures with zeros. The new received polynomial becomes $r'(X) = X^6 + \alpha^3X^5 + X^4 + \alpha^3X^3 + 1$*. The erasure location polynomial is calculated as:*

$$\Gamma(X) = (1 - \alpha X)$$

The syndrome is found to be:

$$S_1 = r'(\alpha) = \alpha^5,\ S_2 = r'(\alpha^2) = \alpha^4,\ S_3 = r'(\alpha^3) = \alpha^4,\ S_4 = r'(\alpha^4) = 1$$

and the syndrome polynomial is therefore:

$$S(X) = \alpha^5 + \alpha^4 X + \alpha^4 X^2 + X^3$$

With both $\Gamma(X)$ and $S(X)$ available, we now compute $\Theta(X)$:

$$\Theta(X) = \Gamma(X) S(X) mod\ X^{2t} = \alpha^5 + \alpha^3 X + X^2 + \alpha^4 X^3$$

We can also find:

$$\sigma(X) = 1 - \alpha^4 X$$

whose root $X = \alpha^{-4}$ indicates an error at $i_1 = 4$.
The error evaluation polynomial is computed next:

$$\Xi(X) = \sigma(X)\Theta(X)\, mod\ X^{2t} = \alpha^5 + \alpha^5 X$$

So the error magnitude is:

$$e_4 = -\Xi(X)/\Phi'(X)\Big|_{X=\alpha^{-4}} = \alpha^4$$

and the erasure magnitudes are:

$$e_1 = -\Xi(X)/\Phi'(X)\Big|_{X=\alpha^{-1}} = \alpha^5$$

where

$$\Phi(X) = \sigma(X)\Gamma(X) = 1 + \alpha^2 X + \alpha^5 X^2$$

The decoded polynomial:

$$\tilde{c}(X) = X^6 + \alpha^3 X^5 + \alpha^5 X^4 + \alpha^3 X^3 + \alpha^6 X^2 + \alpha^5 X + 1 = c(X)$$

4.3 RS Decoder: From Algorithm to Architecture

Design of a high-performance RS decoder is a challenging task and has been an active research topic for years. Performance of an RS decoder is largely determined by three key factors: (1) the decoding algorithm, (2) the architecture to which the algorithm is mapped, and (3) the Galois field arithmetic units. Since the third part was discussed in Chapter 2, in the next section we focus on the architecture design.

4.3.1 Syndrome Computation Circuit

As in (3.64), the syndrome of an RS code is computed as:

$$S_i = r(\alpha^i) = r_0 + r_1\alpha^i + r_2(\alpha^i)^2 + \cdots + r_{n-1}(\alpha^i)^{n-1} \qquad (i = 1, 2, \cdots, 2t) \tag{4.46}$$

Using Horner's rule, (4.46) can be reformulated as:

$$S_i = (((r_{n-1}\alpha^i + r_{n-2})\alpha^i + r_{n-3})\alpha^i + \cdots) + r_0 \quad (i = 1, 2, \cdots, 2t) \tag{4.47}$$

Equation (4.47) results in the recursive syndrome computation circuit shown in Figure 4.6.

4.3.2 Architectures for Berlekamp-Massey Algorithm

As a key step in RS decoding, the Berlekamp-Massey algorithm is probably the most difficult part to implement in an RS decoder. A high-level architecture for realization of the algorithm is illustrated in Figure 4.7 [19, Chap. 5]. The circuit consists of a control unit and three register lines to hold $B(X)$, $\sigma(X)$, $S(X)$ respectively. The control unit generates δ and other necessary control signals. All registers are designed to be large enough to accommodate the largest possible degrees of their respective polynomials. Short-degree polynomials are stored with the registers filled out with zeros. The discrepancy Δ_k is recursively calculated as:

$$\begin{aligned}\Delta_k &= \sum\nolimits_{i=0}^{L^{(k-1)}} \sigma_i^{(k-1)} S_{k-i} \\ &= \left(\left(\left(\sigma_0^{(k-1)} S_k + \sigma_1^{(k-1)} S_{k-1}\right) + \sigma_2^{(k-1)} S_{k-2}\right) + \cdots\right) + \sigma_{L^{(k-1)}}^{(k-1)} S_{k-L^{(k-1)}}\end{aligned}$$

Table 4.2 summarizes the operations.

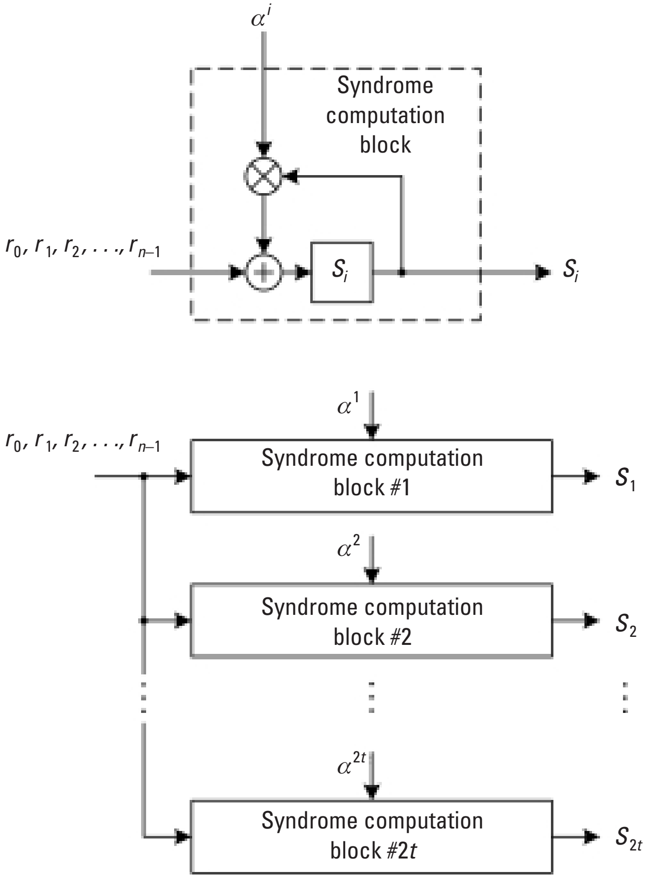

α^i
Syndrome computation block
$r_0, r_1, r_2, \ldots, r_{n-1}$
S_i
S_i
α^1
$r_0, r_1, r_2, \ldots, r_{n-1}$
Syndrome computation block #1
S_1
α^2
Syndrome computation block #2
S_2
α^{2t}
Syndrome computation block #2t
S_{2t}

Figure 4.6 Syndrome computation circuit for RS codes.

The preceding implementation of the Berlekamp-Massey algorithm is rudimentary. Its speed is bottlenecked by computation of the discrepancy followed by updating of the error location polynomial. Some authors have proposed a reformulated inversionless architecture called riBM [20], which pipelines the error location polynomial update and the discrepancy computation so that the discrepancy Δ_{j+1} can be computed and become available one iteration earlier (i.e., at iteration j instead of $j + 1$). To explain the idea, a discrepancy polynomial should be defined:

$$\Delta^{(j)}(X) \triangleq \sigma^{(j)}(X) \cdot S(X) = \Delta_0^{(j)} + \Delta_1^{(j)} X + \Delta_2^{(j)} X^2 + \cdots + \Delta_k^{(j)} X^k + \cdots + \Delta_{2t}^{(j)} X^{2t}$$

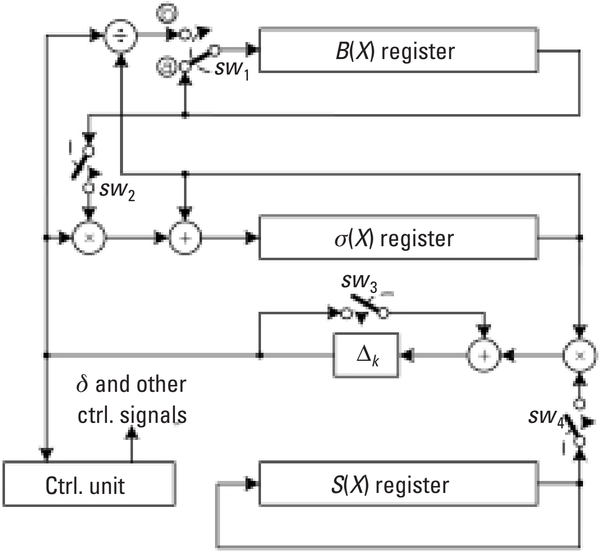


Figure 4.7 Circuit structure for implementing the Berlekamp-Massey algorithm.

where the superscript (j) signifies the jth iteration. By computing $\sigma^{(j)}(X)\cdot S(X)$ directly, we obtain the coefficient $\Delta_j^{(j)}$ as:

$$\Delta_j^{(j)} = \sum_{i=0}^{L^{(j)}} \sigma_i^{(j)} S_{k-i}$$

which equals exactly the discrepancy Δ_{j+1}.

Next let us see if $\Delta^{(j)}(X)$ can be obtained at the jth iteration. Based on (4.18), we have:

$$\Delta^{(j)}(X) = \sigma^{(j)}(X)\cdot S(X) = \left[\theta^{(j-1)}\sigma^{(j-1)}(X) - \Delta_j \cdot X \cdot B^{(j-1)}(X)\right]\cdot S(X)$$

$$= \theta^{(j-1)} \underbrace{\sigma^{(j-1)}(X)S(X)}_{\Delta^{(j-1)}(X)} - \Delta_{j-1}^{(j-1)} \cdot X \cdot \underbrace{B^{(j-1)}(X)S(X)}_{\Psi^{(j-1)}(X)}$$

$$= \theta^{(j-1)}\Delta^{(j-1)}(X) - \Delta_{j-1}^{(j-1)} \cdot X \cdot \Psi^{(j-1)}(X)$$

It is clear that $\Delta^{(j)}(X)$ can indeed be ready at iteration j, and so is the discrepancy Δ_{j+1}. The process is illustrated in Figure 4.8.

Table 4.2
Operation of a Berlekamp-Massey Circuit

	Registers			Switches				
Step	***S*(*X*)**	***B*(*X*)**	**σ(*X*)**	***sw*$_1$**	***sw*$_2$**	***sw*$_3$**	***sw*$_4$**	**Note**
0	Load with S_1, $S_2, \cdots, S_{2t}$	Load with 1	Load with 1	ⓐ	—	—	—	Initialization
1	Shift	—	Shift	ⓐ		Close	Close	Compute Δ
2	—	Shift	Shift/ update (4.17)	ⓐ	Close	Open	Open	Update $\sigma(X)$
3	—	Shift	—	ⓐ	—	—	—	$\delta = 0$
		Shift/ update (4.17)		ⓑ				$\delta = 1$

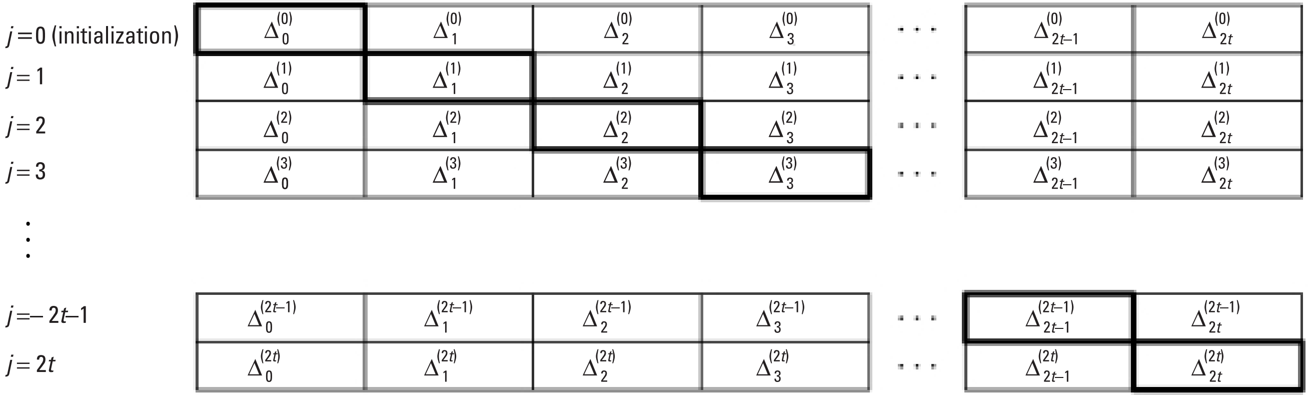
The polynomial $\Psi^{(j)}(X)$ may be computed in a similar way to $B^{(j)}$ [see (4.18)] as:

$$\begin{aligned}\Psi^{(j)}(X) = B^{(j)}(X)S(X) &= \left[\delta\sigma^{(j-1)}(X) + (1-\delta)XB^{(j-1)}(X)\right]S(X)\\ &= \delta\sigma^{(j-1)}(X)S(X) + (1-\delta)XB^{(j-1)}(X)S(X)\\ &= \delta\Delta^{(j-1)}(X) + (1-\delta)X\Psi^{(j-1)}(X)\end{aligned}$$

Putting riBM in matrix form, we have:

$$\begin{bmatrix}\Delta^{(j)}(X)\\ \Psi^{(j)}(X)\end{bmatrix} = \begin{bmatrix}\theta^{(j-1)} & -\Delta_{j-1}^{(j-1)}X\\ \delta & (1-\delta)X\end{bmatrix}\cdot\begin{bmatrix}\Delta^{(j-1)}(X)\\ \Psi^{(j-1)}(X)\end{bmatrix} \tag{4.48}$$

where $\Delta^{(0)}(X) = S(X)$. The approach may be improved even further. For more details, readers are referred to [20].



$j=0$ (initialization)	$\Delta_0^{(0)}$	$\Delta_1^{(0)}$	$\Delta_2^{(0)}$	$\Delta_3^{(0)}$	⋯	$\Delta_{2t-1}^{(0)}$	$\Delta_{2t}^{(0)}$
$j=1$	$\Delta_0^{(1)}$	$\Delta_1^{(1)}$	$\Delta_2^{(1)}$	$\Delta_3^{(1)}$	⋯	$\Delta_{2t-1}^{(1)}$	$\Delta_{2t}^{(1)}$
$j=2$	$\Delta_0^{(2)}$	$\Delta_1^{(2)}$	$\Delta_2^{(2)}$	$\Delta_3^{(2)}$	⋯	$\Delta_{2t-1}^{(2)}$	$\Delta_{2t}^{(2)}$
$j=3$	$\Delta_0^{(3)}$	$\Delta_1^{(3)}$	$\Delta_2^{(3)}$	$\Delta_3^{(3)}$	⋯	$\Delta_{2t-1}^{(3)}$	$\Delta_{2t}^{(3)}$
⋮							
$j=-2t-1$	$\Delta_0^{(2t-1)}$	$\Delta_1^{(2t-1)}$	$\Delta_2^{(2t-1)}$	$\Delta_3^{(2t-1)}$	⋯	$\Delta_{2t-1}^{(2t-1)}$	$\Delta_{2t}^{(2t-1)}$
$j=2t$	$\Delta_0^{(2t)}$	$\Delta_1^{(2t)}$	$\Delta_2^{(2t)}$	$\Delta_3^{(2t)}$	⋯	$\Delta_{2t-1}^{(2t)}$	$\Delta_{2t}^{(2t)}$

Figure 4.8 The riBM process.

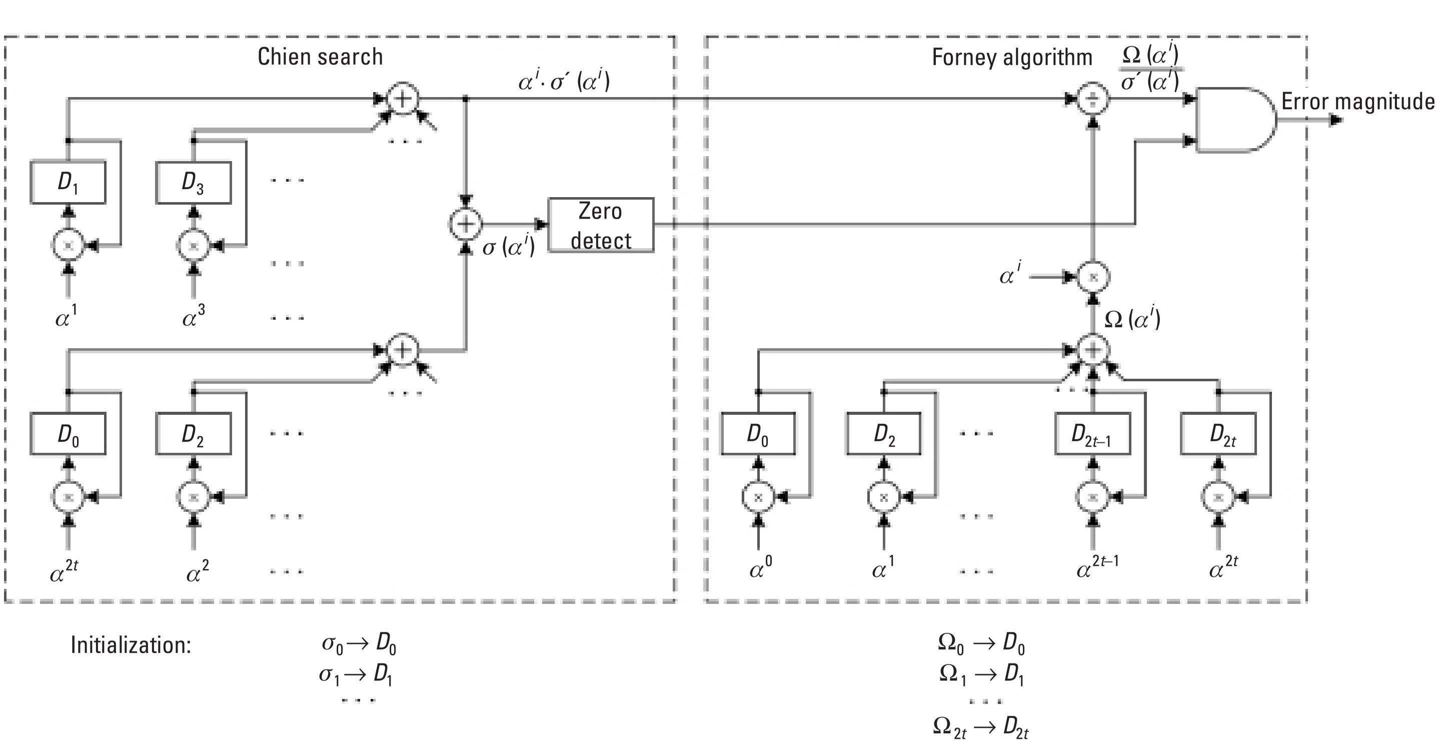


Figure 4.9 Circuit combining the Chien search and Forney algorithm.

4.3.3 Circuit for Chien Search and Forney's Algorithm

We discussed the Chien search in Chapter 3. Notice that the following summation of the Chien search output generates the quantity $\sigma'(X)|_{X=\alpha^i}$, which is required in Forney's algorithm:

$$\begin{aligned}
&\sigma_1\alpha^i + \sigma_3\alpha^{3i} + \cdots + \sigma_{2t-1}\alpha^{(2t-1)i} \\
&= \alpha^i\left(\sigma_1 + \sigma_3\alpha^{2i} + \cdots + \sigma_{2t-1}\alpha^{(2t-2)i}\right) \\
&= \alpha^i\left(\sigma_1 + \sigma_3X^{2i} + \cdots + \sigma_{2t-1}X^{(2t-2)i}\right)\Big|_{X=\alpha^i} \\
&= \alpha^i \cdot \sigma'(X)\Big|_{X=\alpha^i}
\end{aligned}$$

The architecture for a Chien search and the error magnitude computation are shown in Figure 4.9.

4.4 Standardized RS Codes

RS codes have been standardized for a variety of applications. After the famous *Voyage* mission, which employed the (255,223) RS code over $GF(2^8)$, the CCSDS (Consultative Committee for Space Data Systems) officially recommended the code for space communications in the Space Communications Protocol Specifications. The Galois field is generated based on the primitive polynomial:

$$p(X) = 1 + X + X^2 + X^7 + X^8$$

The generator polynomial of the code is specified as:

$$g(X) = \prod_{j=112}^{143}\left(X - \alpha^{11j}\right)$$

The motivation behind the selection of these polynomials was to minimize the encoder hardware.

In commercial applications, RS code was first introduced in compact disc (CD) digital audio systems. The RS code was standardized as the cross-interleaved Reed-Solomon code (CIRC). CIRC consists of two RS codes, the (28, 24) and (32, 28) RS codes, both of which are shortened from the (255, 251) RS code over $GF(2^8)$. From what we learned about code shortening in Chapter 3, we know that shortened codes have the same error correction capability as the original code, which is $t = 2$. The generator polynomial is constructed as follows:

$$g(X)=\prod\nolimits_{j=1}^{4}(X-\alpha^{j})=\alpha^{10}+\alpha^{81}X+\alpha^{251}X^{2}+\alpha^{76}X^{3}+X^{4}$$

The two codes are concatenated together. The coding rate of CIRC is (24/28)(28/32) = 3/4. An interleaver is placed between the two codes to enhance the burst error correcting capability of the code (as will be explained in the next chapter), because defects on the surface of CDs cause errors in bursts. As a result, CIRC is able to correct error bursts up to 4,000 bits, allowing for reproduction of high-fidelity sound [21].

The RS code that has been most widely adopted is perhaps the (255, 239) RS code. The code is stipulated in many different standards such as ANSI T1.413 and ITU G.992 (for ADSL), IEEE 802.16 (for WiMAX) and ITU OTN G.709 (for optical transmission networks), and so forth. The (204,188) RS code used in ETSI DVB-T for digital broadcasting is also a shortened version of this code. The primitive polynomial of the Galois field $GF(2^8)$ and the code generator polynomial for the code are specified as:

$$\varphi(X) = 1 + X^2 + X^3 + X^4 + X^8 \text{ and}$$
$$g(X) = (X + \alpha^0)\,(X + \alpha^1)\ldots(X + \alpha^{15}), \text{ respectively}$$

Note RS code is such an abundant topic that a chapter of this scale can only accommodate the basics. Readers are certainly encouraged to go beyond this. In particular, we would like to bring to the attention of readers two more decoding algorithms. The first one is the Welch-Berlekamp algorithm, which could lead to faster and simpler decoder design [22]. The other is the soft decoding of RS codes. Exemplified by list decoding, this type of RS decoding is able to decode beyond the design distance of the code (amazingly) [23 and the references therein].

This chapter marks the end of our introduction to linear block codes. The next chapter will explore another major class of error control codes, namely convolutional codes.

Problems

4.1 Write a MATLAB program to verify that the Rieger bound holds with equality for RS codes. [You may use the (7,3) RS code as example.]

4.2 Sketch a block diagram of an RS decoder in the frequency domain.

4.3 In MATLAB program a decoder for the (255,239) RS code.

4.4 For the same RS code in our examples, decode the received polynomial $r(X) = X^6 + \alpha^3 X^5 + \alpha^5 X^4 + \alpha^3 X^3 + \alpha^2 X^2 + \alpha^5 X + 1$ using both the BM algorithm and Euclid's method.

4.5 How should we modify the decoding algorithms for nonnarrow-sense RS codes?

References

[1] Reed, I. S., and G. Solomon, "Polynomial Codes over Certain Finite Fields," *SIAM J. Applied Math.*, Vol. 8, 1960, pp. 300–304.

[2] Peterson, W. W., "Encoding and Error-Correction Procedures for the Bode-Chaudhuri Codes," *IRE Trans. Info. Theory,* Vol. IT-6, September 1960, pp. 459–470.

[3] Gorenstein, D., and N. Zierler, "A Class of Error Correcting Codes in p^m Symbols," *J. SIAM*, Vol. 9, June 1961, pp. 207–214.

[4] Forney, G. D., "On Decoding BCH Codes," *IEEE Trans. Inform. Theory*, Vol. IT-11, October 1965, pp. 549–557.

[5] Moon, T. K., *Error Control Coding—Mathematical Methods and Algorithms*, New York: John Wiley & Sons, 2005.

[6] Lin, S., and D. J. Castello, *Error Control Coding—Fundamentals and Application*, 2nd ed., Upper Saddle River, NJ: Prentice-Hall, 2004.

[7] Singleton, R. C., "Maximum Distance Q-nary Codes," *IEEE Trans. Inform. Theory*, Vol. IT-10, 1964, pp. 116–118.

[8] Morelos-Zaragoza, R. H., *The Art of Error Correcting Coding*, New York: John Wiley & Sons, 2002.

[9] Rieger, S. H., "Codes for the Correction of 'Clustered' Errors," *IRE Trans. Inform. Theory*, Vol. IT-6, 1960, pp. 16–21.

[10] Sudan, M., "Decoding of Reed-Solomon Codes Beyond the Error Correction Bound," *J. Complexity*, Vol. 12, 1997, pp. 180–193.

[11] Guruswami, V., and M. Sudan, "Improved Decoding of Reed-Solomon and Algebraic-Geometric Codes," *IEEE Trans. Inform. Theory*, Vol. 45, September 1999, pp. 1755–1764.

[12] Berlekamp, E. R., "Nonbinary BCH Decoding," *Intl. Symposium Inform. Theory*, San Remo, Italy, 1967.

[13] Massey, J. L., "Shift Register Synthesis and BCH Decoding," *IEEE Trans. Inform. Theory*, Vol. IT-15, No. 1, January 1969, pp. 122–127.

[14] Xu, Y., "Implementation of Berlekamp-Massey Algorithm Without Inversion," *IEE Proc.*, Pt. I, Vol. 138, No. 3, June 1991, pp. 138–140.

[15] Reed, I. S., M. T. Shih, and T. K. Truong, "VLSI Design of Inversion-Free Berlekamp-Massey Algorithm," *IEE Proc.*, Part E, Vol. 138, No. 5, September 1991, pp. 295–298.

[16] Sugiyama, Y., et al., "A Method for Solving Key Equation for Decoding Goppa Codes," *Inform. Control*, Vol. 27, 1975, pp. 87–99.

[17] Niven, I., H. S. Zuckerman, and H. L. Montgomery, *An Introduction to the Theory of Numbers*, 5th ed., New York: John Wiley & Sons, 1991.

[18] Clark, G., and J. Cain, *Error-Correcting Codes for Digital Communications*, New York: Plenum Press, 1981.

[19] Wicker, S. B., and V. K. Bhargava, (eds.), *Reed-Solomon Codes and Their Applications*, New York: Wiley-IEEE Press, 1999.

[20] Sarwate, D. V., and N. R. Shanbhag, "High-Speed Architectures for Reed-Solomon Decoders," *IEEE Trans. VLSI Syst.*, Vol. 9, No. 5, October 2001, pp. 641–655.

[21] Wicker, S. B., *Error Control Systems for Communication and Storage*, Englewood Cliffs, NJ: Prentice-Hall, 1995.

[22] Welch, L. R., and E. R. Berlekamp, *Error Correction for Algebraic Block Codes,* U.S. Patent 4633470, December 1986.

[23] Vardy, A., "Recent Advances in Algebraic Decoding of Reed-Solomon Codes," http://ccc.ustc.edu.cn/abstract/vard.ps.

5

Convolutional Codes

Convolutional codes were first introduced by Elias in 1955 [1]. Since then they have gained vast popularity in practical applications. The codes are not only equal (or sometimes even superior) to block codes in performance but also relatively simpler to decode.

Starting with the basic concept, this chapter discusses the fundamental aspects of convolutional codes. Implementation issues frequently arising in practice are also addressed in detail. The focus is on binary code.

5.1 Fundamentals of Convolutional Codes

5.1.1 Code Generation and Representations

5.1.1.1 Codes with Memory

As we mentioned in Chapter 1, convolutional code contains memory, that is, a convolutional encoding process is dependent on both the current and the previous message inputs. Because of this, a convolutional code is specified by three parameters: the codeword length n, the message length k, and the constraint length υ defined as the number of previous messages involved, M, plus 1. So, an (n, k, υ) convolutional code involves not only the current message but also $\upsilon - 1$ previous ones. The parameter M refers to the *memory depth* of the code.

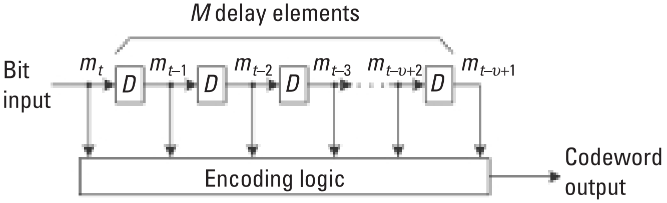


Figure 5.1 Structure of binary convolutional encoder.

For an $(n, 1, \upsilon)$ binary convolutional code, the message input to the encoder is a binary sequence. Upon receiving an input bit m_t at time t, the encoder (Figure 5.1) produces an n-bit codeword $\boldsymbol{c}_t = (c_t^{(1)} c_t^{(2)} \cdots c_t^{(n)})$ as follows:

$$c_t^{(j)} = \sum_{i=0}^{\upsilon-1} g_i^{(j)} \oplus m_{t-i} \tag{5.1}$$

where $j = 1, 2, \cdots, n.g_i^{(j)} \in \{0,1\}$ are the coefficients. These coefficients constitute the encoding logic of the encoder.

Example 5.1

A simple (2,1,3) binary code is defined in (5.2) and its corresponding encoder is shown in Figure 5.2:

$$\begin{cases} c_t^{(1)} = m_t \oplus m_{t-1} \oplus m_{t-2} \\ c_t^{(2)} = m_t \oplus m_{t-2} \end{cases} \tag{5.2}$$

Generation of the codeword $\boldsymbol{c}_t = (c_t^{(1)} c_t^{(2)})$ depends not only on the current input bit m_t but also on two previous ones: m_{t-1} and m_{t-2}. The constraint length v is therefore 3. The coding rate is $k/n = 1/2$.

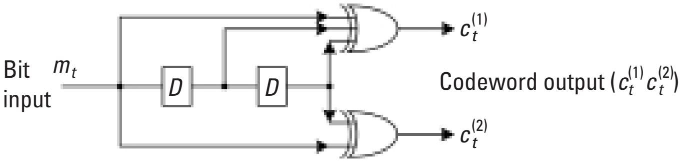


Figure 5.2 A (2,1,3) convolutional encoder.

Notice that the coefficient set $g_i^{(j)}$ in (5.1) completely defines a convolutional encoder. It is therefore referred to as a code generator. For instance, the generators of the example (2,1,3) code are (111) and (101). Often, the two binary vectors are abbreviated in octal as 7 and 5. Alternatively, they may also be expressed in the transform domain as a generator polynomial: $1 + D + D^2$ and $1 + D^2$, where D is the delay operator.

Example 5.2

Apply an input sequence of $\vec{m} = 1\,1\,0\,1\,0\,0$ to the (2,1,3) convolutional encoder. Using (5.2), it is encoded to be $\vec{c} = 11\,01\,01\,00\,10\,11$ (assume that the encoder is reset to the all-zero state initially).

MATLAB Experiment 5.1

The following MATLAB commands produce the same coded output as in Example 5.2. The MATLAB function `convenc` convolutionally encodes a binary data sequence.

```
>> g = [7 5];                        % code generator
>> m = [1 1 0 1 0 0];                % message
>> trellis = poly2trellis(3,g);      % constraint length = 3
>> c = convenc(m,trellis)            % encoding
c =
      1  1  0  1  0  1  0  0  1  0  1  1
```

Comment: Put aside the third line for now because we will explain it in the next section.

5.1.1.2 Systematic Convolutional Codes

The preceding example code is apparently nonsystematic. Some convolutional codes, however, are in the systematic form. For example, the following encoding

$$\begin{cases} c_t^{(1)} = m_t \\ c_t^{(2)} = m_t \oplus m_{t-2} \end{cases}$$

produces a systematic code with the first bit in a codeword being the message bit itself. Systematic convolutional codes have a very important feature (or, we should say, advantage), that is, they cannot be catastrophic, as explained later in Section 5.4.

5.1.1.3 Code Representations

Convolutional codes may be represented in several ways. First, from a circuit point of view, a convolutional encoder is a state machine with $2^{\upsilon-1}$ states, where the state is the content of the encoder memory. The encoder may then be represented by a *state transition diagram*.

Example 5.3

> *The (2,1,3) code encoder has $2^{3-1} = 4$ states. They are S(00), S(10), S(01), and S(11). A truth table (Table 5.1) specifies the relationship among the input, output, and state transition of the encoder. Based on this table, the state transition diagram is shown in Figure 5.3.*

If we display all of the possible state transitions in a branching structure, a tree diagram for the code is formed.

Example 5.4

> *The code tree for the example (2,1,3) code is constructed partially in Figure 5.4. The tree is followed upward if the input bit is a 0, or downward if a 1. The bold line corresponds to the encoding process in Example 5.2 and*

Table 5.1
Truth Table of the (2,1,3) Convolutional Encoder

Current State	Message Bit	Codeword	New State
S(00)	0	00	*S*(00)
S(00)	1	11	*S*(10)
S(10)	0	10	*S*(01)
S(10)	1	01	*S*(11)
S(01)	0	11	*S*(00)
S(01)	1	00	*S*(10)
S(11)	0	01	*S*(01)
S(11)	1	10	*S*(11)

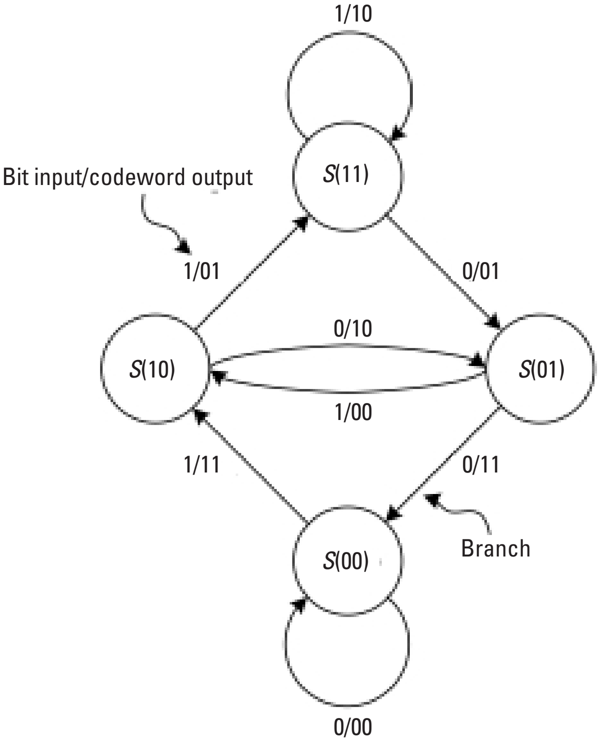


Figure 5.3 State transition diagram.

is called the encoding path. The path is obtained as follows: Starting from the root S(00), the first message bit is a 1, so the path moves downward and produces the first codeword "11." The second bit is also a 1, so the path moves down again and produces the second codeword "01," and so forth. The procedure is repeated until the last message bit is encoded. The coded sequence along the encoding path (from left to right) is 11 01 01 00 ⋯, which agrees with what we had in the previous example.

A third graphical representation is the trellis diagram. The trellis diagram is perhaps the most frequently used representation in convolutional codes.

Example 5.5

Figure 5.5 shows the trellis of our example code. The branches in the diagram represent state transitions. The upper branch coming out of a state corresponds to an input message bit of 0, and the lower branch to a bit of

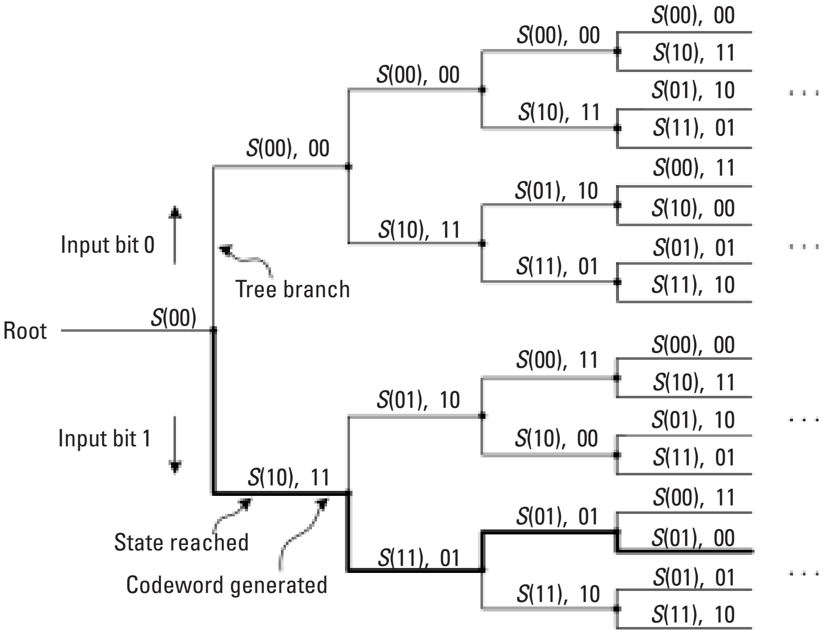


Figure 5.4 Code tree diagram.

1 (which is in line with the convention used in the tree diagram). Each trellis node represents a particular state S at a particular time t and can be uniquely denoted by (S, t).

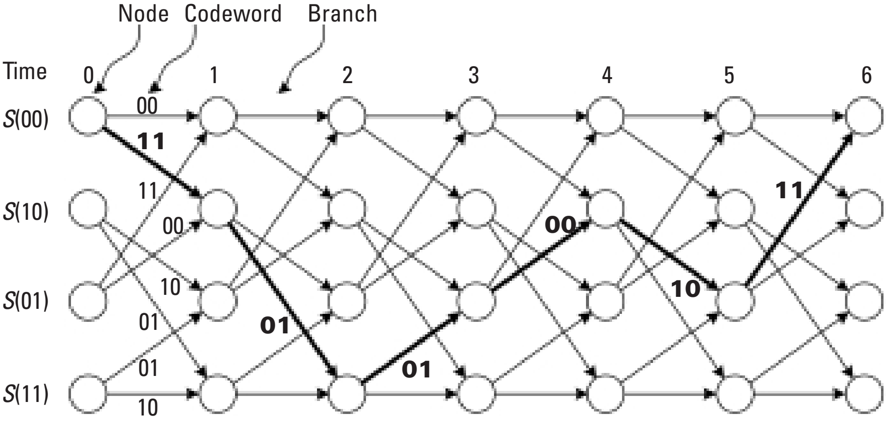


Figure 5.5 Code trellis diagram.

The highlighted encoding path in the figure corresponds to the encoding process under the input sequence 1 1 0 1 0 0 [with the initial state being S(00)]. The process evolves as (S(00),0)→(S(10),1)→(S(11),2)→(S(01), 3)→(S(10),4)→(S(01),5)→(S(00),6). The coded sequence is simply the chain of the codewords on all branches along the encoding path and it is 11 01 01 00 10 11.

The trellis diagram is the temporal repetitions of the radix-2 structure shown in Figure 5.6(a). (It is called radix-2 because there are two branches leaving a node and two branches entering a node.) Rearranging the trellis nodes, we can decompose the radix-2 structure into two independent butterfly-like substructures; see Figure 5.6(b).

The observation that there exist two branches entering a trellis node and two branches leaving the node for the example code applies to any binary convolutional code.

The astute reader may ask at this point what these diagrams are for. The answer is that they exhibit different aspects of convolutional codes. The state transition diagram demonstrates the repetitive nature of a convolutional encoding process, the tree diagram illustrates the time evolution, and the trellis diagram shows both.

From the diagrams just discussed it is easy to see that convolutional encoding is in fact a process in which the encoder traverses the code tree or the code trellis along a particular path (i.e., the encoding path) directed by the message bit sequence. Conversely, convolutional decoding is a process in which the decoder searches for the path that the encoder has traversed (i.e., the decoding path).

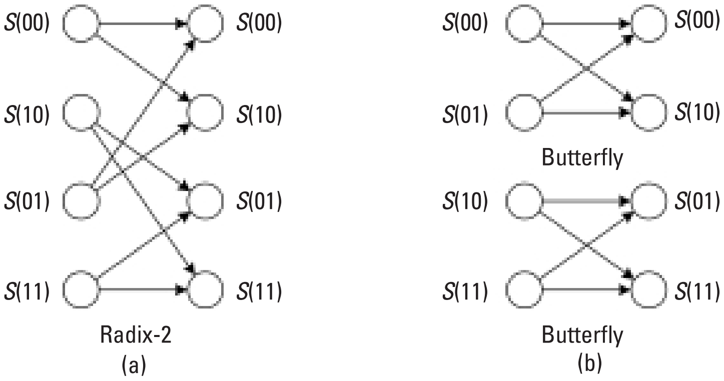


Figure 5.6 (a) Radix-2 structure and (b) butterfly substructure.

MATLAB Experiment 5.2

We now go back to the third line of the script in MATLAB Experiment 5.1. What the command does is to convert the (2,1,3) code generator to the trellis description. Let us check it out:

```
>> trellis = poly2trellis(3,[7 5]);
>> trellis.numInputSymbols
ans =
       2
```

Comment: The input to the encoder assumes two possible values: either bit 0 or bit 1. Therefore, the answer is 2.

```
>> trellis.numOutputSymbols
ans =
       4
```

Comment: There are four possible codewords in total: "00," "01," "10," and "11."

```
>> trellis.numStates
ans =
       4
```

Comment: The total number of states is 4.

```
>> trellis.nextStates
ans =
       0 2
       0 2
       1 3
       1 3
```

Comment: The (i, j)-th element a_{ij} in the matrix `trellis.nextStates` indicates that there exists a trellis branch starting from the current state $(i - 1)$ and ending at the next state a_{ij} under the input $(j - 1)$. For instance, the (1, 2)-th element 2 signifies a branch from state 0 ($S(00)$) to state 2 ($S(10)$) under input bit 1.

```
>> trellis.outputs
ans =
       0 3
       3 0
       2 1
       1 2
```

Comment: Each element in `trellis.outputs` is the codeword associated with the branch correspondingly specified in `nextStates`. For instance, the element at location (1,2), 3, indicates that the branch $S(00) \rightarrow S(10)$ produces codeword "11."

5.1.2 Additional Matters

5.1.2.1 Distance Properties

The minimum distance of a convolutional code, defined as the smallest Hamming distance between all possible code sequences of the code, is called the *free distance*, d_{free}. Similar to the block code case, the distance determines the error correcting capability of a convolutional code. Because convolutional codes are linear and their distance properties are code sequence independent, the free distance may further be defined as the smallest Hamming distance between the nonzero code sequences and the all-zero sequence, or as the minimum Hamming weight of the nonzero code sequences:

$$d_{\text{free}} = \min_{\vec{c}_A \neq \vec{c}_B} d_{\text{H}}(\vec{c}_A, \vec{c}_B) = \min_{\vec{c} \neq \vec{0}} d_{\text{H}}(\vec{c}, \vec{0}) = \min_{\vec{c} \neq \vec{0}} w(\vec{c}) \tag{5.3}$$

where $\vec{c}_A$ and $\vec{c}_B$ are two different code sequences.

Example 5.6

> *Figure 5.7 shows the trellis diagram for the example code. The number on each branch is the Hamming weight of the codeword associated with the branch. Starting from the leftmost side of the trellis, we see that no path enters state S(00) other than the all-zero path until time 3 and it is S(00)→S(10)→S(01)→S(00). The free distance of the code is the cumulative Hamming weight of the path, and is calculated to be 2 + 1 + 2 = 5.*

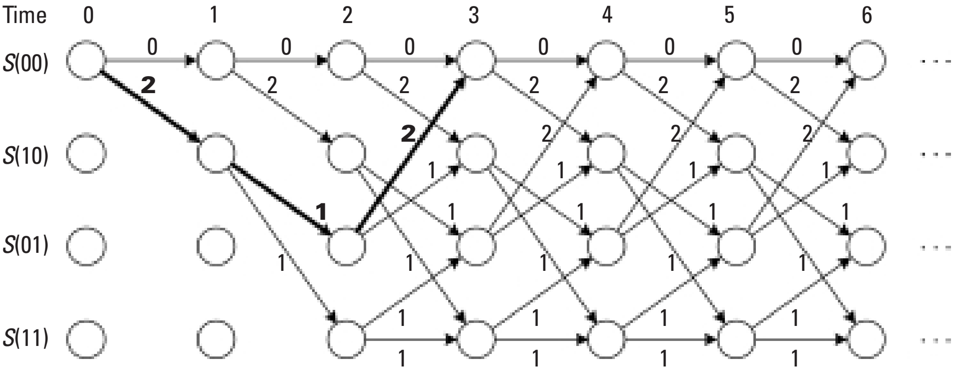


Figure 5.7 Finding free distance in a trellis diagram.

Although using visual examination to find the free distance appears conceptually quite straightforward, the process itself can be rather tedious. The so-called *modified generating function* (sometimes also called the transfer function) provides an analytical solution to the problem.[1] For a given code, the modified generating function is obtained by first modifying the state transition diagram of the code as follows:

1. Replace the "bit input/codeword output" on each branch with the branch gain X^i whose exponent i is the Hamming weight of the branch (i.e., the number of nonzero bits in the codeword of the branch).
2. Split the initial state into a starting state and an ending state, and discard the self-loop of the initial state.[2]

Then solve the simultaneous state equations corresponding to the modified transition diagram.

Example 5.7

Following the above-stated rules, the state transition diagram for the (2,1,3) code in Figure 5.3 is modified as shown in Figure 5.8. The initial state S(00) is split into a starting state S(00)$_s$ and an ending state S(00)$_e$. Based on the diagram, we can establish a set of state equations as follows:

$$\begin{cases} X(10) = X^2 \cdot X(00)_s + 1 \cdot X(01) \\ X(01) = X \cdot X(10) + X \cdot X(11) \\ X(11) = X \cdot X(10) + X \cdot X(11) \\ X(00)_e = X^2 \cdot X(01) \end{cases} \tag{5.4}$$

where X(00)$_s$, X(10), X(01), X(11), and X(00)$_e$ are five temporary variables holding the gains of the branches arriving at state S(10), S(01), S(11), and S(00)$_e$, respectively.

The modified generating function is defined in (5.5), and obtained in (5.6) by solving (5.4):

[1] This method is suitable for convolutional codes with a small constraint length. Unfortunately, no solution yet exists for large constraint lengths.

[2] This is done because circulation of the loop only produces a trivial all-zero sequence and does not contribute to the distance property of the code.

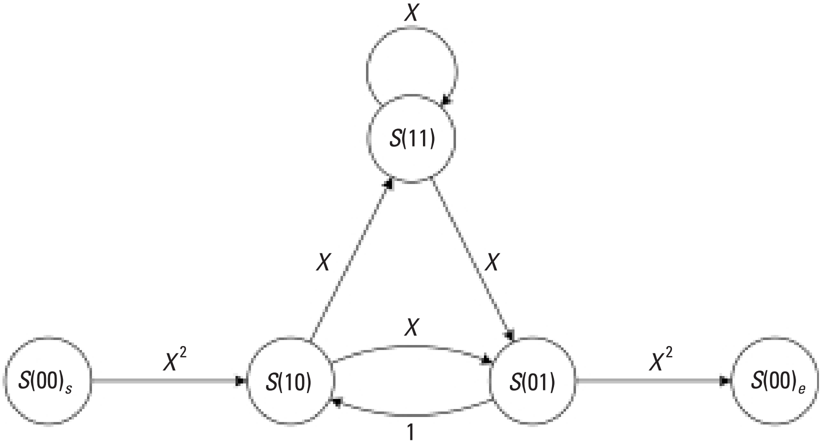


Figure 5.8 Modified state diagram.

$$T(X) \triangleq X(00)_s \,/\, X(00)_e \tag{5.5}$$

$$\begin{aligned} &= X^5/(1-2X) \\ &= 1\cdot X^5 + 2\cdot X^6 + 4\cdot X^7 + 8\cdot X^8 + \cdots + 2^i \cdot X^{i+5} + \cdots \end{aligned} \tag{5.6}$$

It follows from (5.5) that the modified generating function T(X) contains the gains of the paths that depart from the S(00) [i.e., S(00)$_s$] initially and come back to S(00) [i.e., S(00)$_e$] sometime later. The first term on the right-hand side of (5.6) indicates that there is one shortest such path with a path gain of 5, or a Hamming weight of 5. By definition, the free distance of the code is 5.

MATLAB Experiment 5.3

The MATLAB function `dfree`* on this book's companion DVD computes the free distance of a convolutional code.

```
>> g = [7 5];                          % code generator
>> trellis = poly2trellis(3,g);        % convert to trellis
>> df = dfree(trellis)                 % get free distance
df =
       5
```

Recall that a block code can correct up to $t = \lfloor (d_{\min} - 1)/2 \rfloor$ errors, where $\lfloor x \rfloor$ denotes the greatest integer not greater than x, and $d_{\min}$ is the minimum distance. Similarly a convolutional code can also correct up to t errors, and t is computed as:

$$t = \lfloor (d_{\text{free}} - 1)/2 \rfloor \tag{5.7}$$

This error correcting capability is obtained when the errors are separated by at least the constraint length of the code (i.e., random error).

5.1.2.2 Code Termination

To encode a message sequence, we need a starting point. This starting point is the encoder initial state and is usually set to be the all-zero state. However, at the end of the encoding process, the encoder state is usually unknown. On the other hand, an optimum decoder needs to know the encoder final state to decode (see Section 5.2). As such, it is necessary to force the encoder to a known state when the encoding process comes to an end (this is what code termination is all about). Several methods exist for doing this. Among them the zero-tailing and tail-biting methods are the two most popular ones.

The zero-tailing method (Figure 5.9) appends a "tail" of M zeros (M is the memory depth of the encoder) to the message sequence, so that at the end the encoder memory contains only zeros and the encoder is at the all-zero state. We have in fact done this implicitly. Going back to Example 5.2, the actual message bits of the encoder input sequence in the example are 1 1 0 1, and two zeros are purposely added at the end of the sequence in order to let the encoder (with the memory depth $M = 2$) go back to the all-zero state. Also in Section 5.1.2.1, we split state $S(00)$ into a starting state and an ending state, implying that the ending state is also $S(00)$. The M zeros are often called *flushing bits*, meaning that they are used to clear the encoder memory.

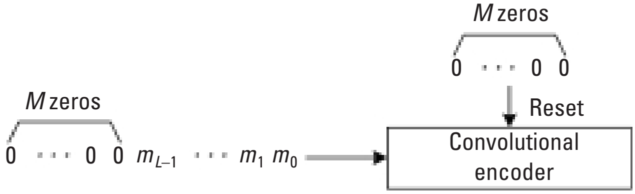


Figure 5.9 Zero-tailing method.

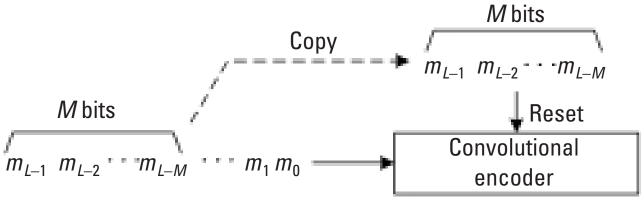

M bits
Copy
m_{L-1} m_{L-2} · · · m_{L-M}
Reset
M bits
m_{L-1} m_{L-2} · · · m_{L-M} · · · m_1 m_0
Convolutional encoder

Figure 5.10 Tail-biting method.

The method is simple to implement, but, due to addition of the extra bits, the effective coding rate is reduced to $R \cdot l/(l + M)$, where l is the length of the actual message sequence, and R is the original coding rate.

The tail-biting method (Figure 5.10) solves the problem by initially setting the encoder to a state that is identical to its final state rather than to the all-zero state. The encoding process then proceeds as follows: First, use the last M bits of the message sequence to initialize the encoder, and then input the message sequence to the encoder to perform the encoding as usual. At the end of encoding, the same M bits terminate the encoder at the same initial state. Because no extra bits are involved, the coding rate is retained. The price to pay is the increase of decoder complexity, because all the decoder knows is merely that the initial state and the final state are identical and the decoder has to find out what the actual state is. Because this common state can be any one of $2^{\upsilon-1}$ possible states (where υ is the code constraint length), in the worst case an optimum decoder needs to run $2^{\upsilon-1}$ decoding trials to search for the decoding path with identical initial and final states. The disadvantage triggered exploration of suboptimum decoding algorithms in exchange for reduction of the decoder complexity [2].

5.2 Decoding of Convolutional Codes

As previously stated, convolutional decoding is the process of searching for the path that an encoder has traversed. Three main convolutional decoding schemes exist: sequential decoding, majority-logic decoding, and Viterbi decoding. Sequential decoding, as the first practical decoding technique for convolutional codes, was introduced in [3]. The most notable achievements in this area are the Fano algorithm and the stack algorithm. The threshold-based majority-logic decoding scheme appeared some time later [4]. In 1967 Viterbi published a new decoding method, known today as the Viterbi algorithm [5]. The algorithm is optimal in the maximum-likelihood sense [6], and has quickly become the most widely used convolutional decoding algorithm in practice for its reduced computational complexity and satisfactory performance. This section discusses sequential decoding and Viterbi decoding because they are the most popular ones in applications. For majority-logic decoding, readers are referred to [4].

5.2.1 Optimum Convolutional Decoding and Viterbi Algorithm

5.2.1.1 Maximum-Likelihood Decoding of Convolutional Codes

Maximum-likelihood decoding of convolutional codes can be expressed as follows:

$$P\left(\vec{r} \mid \vec{c}^{*}\right) = \max_{\{c\}} P(\vec{r} \mid \vec{c}) \tag{5.8}$$

where $\vec{c} = c_1, c_2 \cdots, c_L$ is the coded bit sequence, $\vec{r} = r_1, r_2, \cdots, r_L$ is the received bit sequence, and L is the sequence length $\vec{c}^{*}$ is the decoded sequence. Literally what (5.8) says is that among all possible code sequences, if code sequence $\vec{c}^{*}$ is transmitted, the received sequence is most likely to be $\vec{r}$.

For a memoryless channel, the likelihood function $P(\vec{r} \mid \vec{c})$ can be expressed as:

$$P\left(\vec{r} \mid \vec{c}\right) = P\left(r_1 \mid c_1\right) \cdot P\left(r_2 \mid c_2\right) \cdots P\left(r_L \mid c_L\right) = \prod\nolimits_{i=1}^{L} P(r_i \mid c_i) \tag{5.9}$$

Recall that in a BSC:

$$P\left(r_i \mid c_i\right) = \begin{cases} 1 - p_X & r_i = c_i \\ p_X & r_i \neq c_i \end{cases} \tag{5.10}$$

where p_X is the crossover probability of the channel. Substituting (5.10) into (5.9) the likelihood function for the BSC is obtained:

$$P\left(\vec{r} \mid \vec{c}\right) = p_X^{d_{\mathrm{H}}} (1 - p_X)^{L - d_{\mathrm{H}}} \tag{5.11}$$

where d_{H} is the Hamming distance between the received sequence and the code sequence (i.e., the number of differing bits). In (5.11) $P(\vec{r} \mid \vec{c})$ is a monotonically decreasing function of d_{H}. Maximizing (5.11) is therefore equivalent to choosing a code sequence whose Hamming distance to the received sequence is minimized.

For an AWGN channel with a noise power spectral density (PSD) of N_0, the likelihood function is:

$$P(\vec{r} \mid \vec{c}) = \sqrt{\frac{1}{\pi N_0}} e^{-\frac{d_{\mathrm{E}}^2}{N_0}} \tag{5.12}$$

where d_{E}^2 is the squared *Euclidean distance* between the received sequence and the code sequence and is defined as:

$$d_{\mathrm{E}} = \sqrt{\sum_{i=1}^{L} \left| r_i - c_i \right|^2} \tag{5.13}$$

Similar to the BSC case, it is easy to show that maximization of (5.12) is equivalent to minimization of d_{E}^2.

ML decoding of convolutional codes is straightforward at first glance, but a closer examination reveals that it is impractical for real applications because all 2^L possible code sequences must be examined one by one. For an L as small as 100, there are a total of $2^{100} > 10^{30}$ sequences existing. Even with today's most powerful computer, the task takes weeks to finish. Therefore, an efficient algorithm is necessary. The Viterbi algorithm has a complexity linearly proportional to the sequence length L and, furthermore, it is maximum likelihood based.

5.2.1.2 The Viterbi Algorithm

If we arbitrarily choose a node (S,t) in the trellis diagram of a code and look at all of the paths going into it, we will find that there always exists a path that has a smaller distance between the received sequence and the code sequence than all other paths.[3] By the definition of maximum likelihood, the path is an optimum path, at least for now.[4] This path is called the local survivor path, or simply the *survivor*. Viterbi noticed that the paths that are not optimal now can never be optimal in the future. This observation led to the famous Viterbi algorithm in which only one path, the survivor path, is retained for each trellis node during the entire decoding process. Because there are $2^{\upsilon-1}$

[3] Be it the Hamming distance or the squared Euclidian distance.

[4] Occasionally there may exist more than one such paths for the same node. Choosing any one of them as the local optimum path does not affect the final decoding result.

states, the total number of path examinations for the entire decoding process is $2^{v-1} \times L$, where L is the sequence length. We see that computation of the Viterbi algorithm increases *linearly* with sequence length L.

Two metrics are used in the Viterbi algorithm: a *branch metric* and a *path metric*. Let $\boldsymbol{c} = (c^{(1)}\, c^{(2)} \cdots c^{(n)})$ be the transmitted codeword and $\boldsymbol{r} = (r^{(1)}\, r^{(2)} \cdots r^{(n)})$ be the received vector corresponding to the codeword. The branch metric BM is then defined as follows:

$$BM(\boldsymbol{r},\boldsymbol{c}) = \begin{cases} d_{\mathrm{H}}(\boldsymbol{r},\boldsymbol{c}) & \text{BSC} \\ \sum_{i=1}^{n} \left| r^{(i)} - c^{(i)} \right|^2 & \text{AWGN Channel} \end{cases} \tag{5.14}$$

where $\boldsymbol{r}$ is a binary word for the BSC, or a real-valued vector for the AWGN. The branch metric measures how close the received vector is to the codeword.

Another metric, the path metric PM of state S, is the accumulation of branch metrics on the path from the beginning of the trellis up to the current decoding point:

$$PM_{(S,t)} = \sum_{\{\text{branch}\}} BM(\boldsymbol{r},\boldsymbol{c}) \tag{5.15}$$

where {branch} denotes all branches on the path. The path metric can be calculated recursively as follows:

$$PM_{(S,t+1)} = PM_{(S',t)} + BM_{(S',t)\to(S,t+1)} \tag{5.16}$$

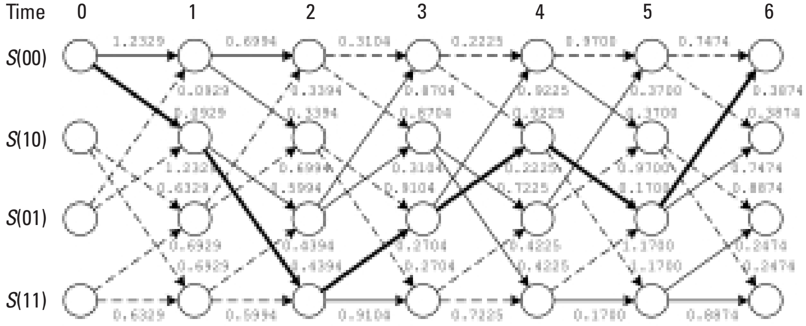
That is, the path metric of the next node $(S, t+1)$ is the path metric of the current node (S', t) plus the branch metric corresponding to branch $(S', t) \to (S, t+1)$.

The Viterbi algorithm is best illustrated by using an example.

Example 5.8

Suppose that the code sequence $\vec{c} = 110101001011$ has been transmitted (corresponding to the message sequence $\vec{m} = 1\,1\,0\,1\,0\,0$ in Example 5.1). The noisy received sequence $\vec{r}$ is 0.8 0.77 0.55 0.63 0.2 0.52 0.25 0.4 0.9 0.4 0.43 0.75.

The Viterbi decoding of the above code is divided into three procedures. The first is to compute the branch metrics for all branches in the trellis diagram using (5.14). The resulting metrics are shown in Figure 5.11. Take the branch S(00)→S(00) at time 3 as an example. Because the codeword



Path metric

0	1.2329	1.9323	1.5627	1.7252	2.0952	1.5826
∞	0.0929	1.5723	1.0027	1.0252	2.0952	1.9426
∞	∞	0.6923	0.8027	1.7252	1.1952	1.8426
∞	∞	0.5323	1.4427	1.4252	1.5952	2.4826

Decode output 1 1 0 1 0 0

Figure 5.11 Viterbi decoding process.

associated with the branch is (00) and the received vector is (0.2 0.52), the BM is computed as $|0-0.2|^2+|0-0.52|^2=0.3104$.

Next, for every node in the trellis diagram, the path metrics are calculated for all arriving paths. Afterwards the path with minimum PM is taken as the survivor of the node and, at the same time, all others are discarded. Use node (S(00),3) as an example. The two paths coming into the node are from (S(00),2) and (S(01),2). Their path metrics are calculated as:

1.9323 [path metric of node (*S*(00),2)] + 0.3104 (branch metric) = 2.2427 and

0.6923 [path metric of node (*S*(01),2)] + 0.8704 (branch metric) = 1.5627,

respectively. The path with the metric value 1.5627 becomes the survivor path (the solid line in the figure), and the other is given up (the dashed line). Because we reset the encoder to state S(00) at the beginning, the initial path metric of S(00) should be set to 0 and all others to ∞.

If the code is properly terminated at state S(00), the last procedure in the Viterbi algorithm selects the local path ending at (S(00),6), that is, S(00)→S(10)→S(11)→S(01)→S(010)→S(01)→S(00), as the global optimum decoding path (shown in bold in the figure). Based on Table 5.1 the *path is mapped to decode sequence as follows: S(00)→S(10) maps to a message bit of 1, S(10)→S(11) again to 1, S(11)→S(01) to 0, and so forth. The decoding sequence is then obtained as 1 1 0 1 0 0, which is exactly the original message bit sequence.*

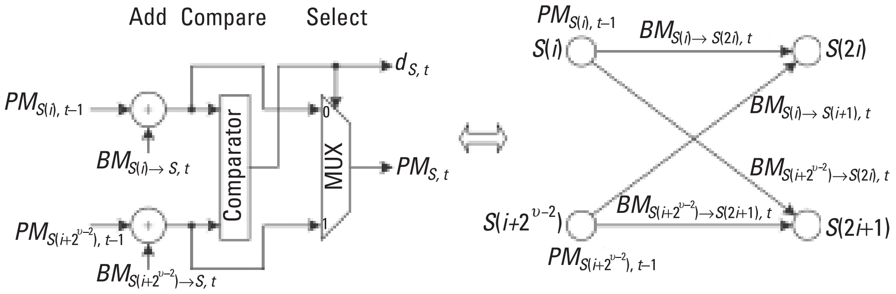
The Viterbi decoding used in the preceding example is considered to be soft-decision decoding because the received sequence is real valued. On the other hand, if the channel is a BSC (the received sequence is binary), the decoding becomes hard-decision decoding.

Notice that the path metric update in the Viterbi algorithm involves add [(5.16)], compare (to find the minimum path metric), and select (choose the path with minimum metric as the survivor). The operations form the so-called add-compare-select (ACS). ACS can be expressed mathematically:

$$PM_{(S,t)} = \min\Big\{\Big[PM_{S(i),t-1} + BM_{S(i)\to S,t}\Big],\Big[PM_{S(i+2^{\upsilon-2}),t-1} + BM_{S(i+2^{\upsilon-2})\to S,t}\Big]\Big]\Big\} \tag{5.17}$$

where $S = S(2i)$ or $S(2i + 1)$. One note about (5.17) is that the state is numbered in decimal notation. For our example code, we have $S(0)\leftrightarrow S(00)$, $S(1)\leftrightarrow S(10)$, $S(2)\leftrightarrow S(01)$ and $S(3)\leftrightarrow S(11)$. The implementation of (5.17) is sketched in Figure 5.12. The output of the MUX, $PM_{S,t}$, is the minimum path metric. Signal $d_{s,t}$ picks the survivor.

To summarize, the Viterbi algorithm decodes as follows:



Note: Subscript $S = S(2i)$ or $S(2i+1)$

ACS

Butterfly

Figure 5.12 ACS architecture.

Viterbi Algorithm

Initialization:

Set: time ← 0, initial state *PM* ← 0, all other *PM* ← ∞.

Normal Operation:

1. Increase time by 1.
2. *BM* computation: Compute *BM* for each branch.
3. *PM* update: Perform ACS for each current state, and store the survivor path together with its *PM*; discard the other(s).
4. If the end of the trellis is reached, map the global optimum path to the decode sequence and output. Otherwise go back to step 1.

Note that in contrast to the previous example, *BM* computation is now distributed over the entire decoding process. Figure 5.13 is the flowchart for the Viterbi algorithm.

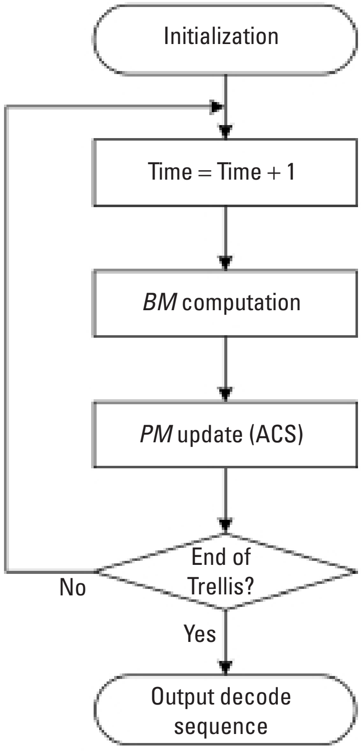


Figure 5.13 Flowchart for the Viterbi algorithm.

MATLAB Experiment 5.4

The function `vitdec` in MATLAB Communications Toolbox decodes binary convolutional codes using the Viterbi algorithm. We now apply it to Example 5.8. The function only accepts quantized input if used as soft-decision decoding. Therefore, we quantize $\vec{r}$ to 256 levels so that it will be close enough to its original value.

```
>> trellis = poly2trellis(3, [7 5]);
>> % received sequence
>> r = [0.8 0.77 0.55 0.63 0.2 0.52 0.25 0.4 0.9 0.4 0.43...
      0.75];
>> % quantization, rq is the quantized version of vector r
>> [x rq] = quantiz(r,[0:1/256:1-1/256],[0:255]);
>> % decoding, ‘soft’ for soft-decoding
>> decoded = vitdec(rq,trellis,3,’term’,’soft’,8)
decoded =
      1  1  0  1  0  0
```

With the frame-by-frame Viterbi decoding above, the decoded data sequence will not be available until the end of the received sequence (i.e., one frame). For real-time applications this may not be acceptable, especially when the sequence is long. To solve the problem, the *sliding-window decoding* method is used instead.

MATLAB Experiment 5.5

Let us run the m-file `survpath.m`* on the book's DVD to trace the survivor paths in the decoding of the (2,1,3) convolutional code.

```
>> survpath;         % run the m-file
>> state             % check the state transition history
state =
      0  0  0  1  1  0  0  0
      0  2  2  3  2  3  3  3
      0  0  0  0  1  1  1  0
      0  2  2  3  3  3  3  2
>> decoded(1:3)      % first 3 bits of the decoded sequence
ans =
      0  1  1
>> m(1:3)            % first 3 bits of the message sequence
ans =
      0  1  1
```

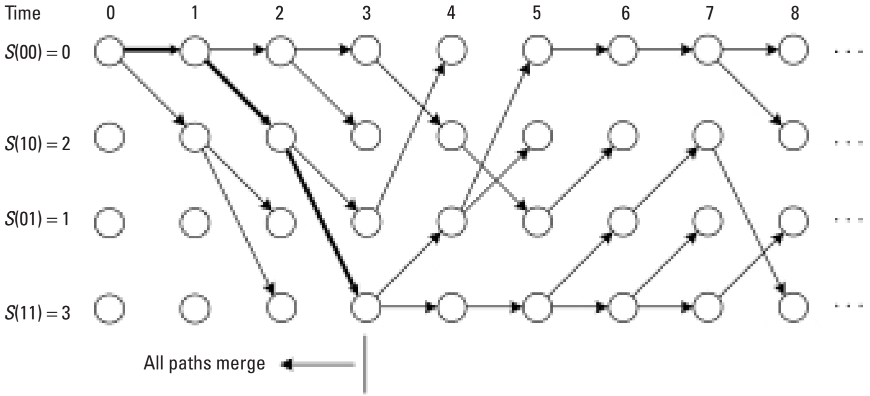


Figure 5.14 Path merge.

The state matrix contains the history of all survivor paths in the past eight decoding steps. The element $a_{i,j}$ represents a branch coming from state $a_{i,j}$ to node (i, j). The corresponding trellis diagram is shown in Figure 5.14. Suppose that the decoder is now at time 8. Looking back at all survivor paths in the trellis diagram, it is easy to find that those paths join together starting from time 3 backward. Moreover, the decoder output of this merged path is 0 1 1, which is exactly the first 3 bits of the message. So, we see that the first 3 bits have been successfully decoded *before* the end of the received sequence, which is 12 bits long.

What has happened in the experiment is not coincidental. Rather it is an inherent feature of the Viterbi algorithm: Survivor paths tend to agree on one common path some distance away from the current decoding point *backward*, and the decode output corresponding to the common path tends to be error free. The sliding-window decoding method makes use of this feature and decodes the survivor paths that have merged prior to a window rather than waiting for the end of the entire sequence. Keep in mind that the common path does not depend on any particular survivor path because all paths should have merged before the window.[5] Once

[5] In practice, we usually take as the decoding path either the path with the smallest metric or the path ending at the all-zero state. The latter is probably easier to implement.

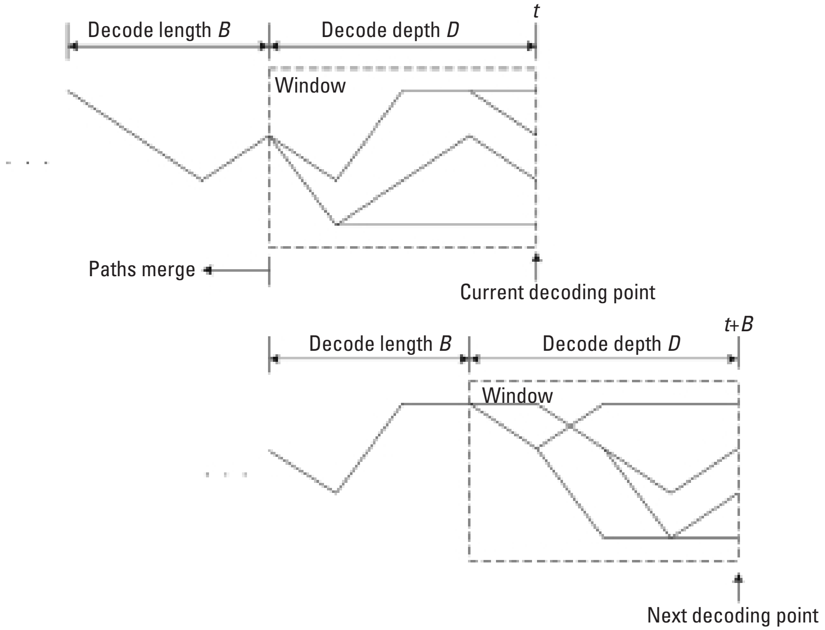


Figure 5.15 Sliding-window decoding.

the current decoding finishes, the window slides forward by the decode length B (Figure 5.15).

The width of the sliding window is called the decoding depth. The possibility that survivors will merge goes higher as the decoding depth increases, and so does the decoding latency. Clearly this is a trade-off we need to make. The decoding error caused by insufficient decoding depth is called the *truncation error*. Forney [7] found that with a decoding depth equal or greater than roughly 5 times the constraint length v, the truncation error is virtually negligible.

A special case of the sliding-window decoding is that when B is set to 1, the decoder output becomes a continuous bit stream.

5.2.1.4 Performance of Viterbi Decoding

We illustrate in Figure 5.16 the simulated BER of rate-1/2 convolutional codes with different constraint lengths decoded using the Viterbi algorithm.

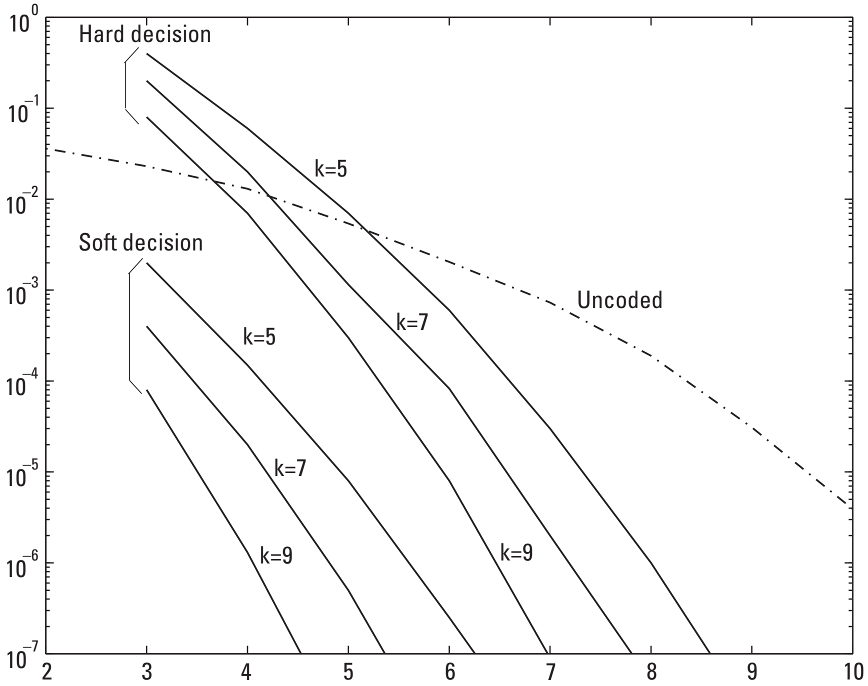


Figure 5.16 Performance of Viterbi decoding. (*After:* [8].)

Exact closed-form BER expression does not exist for Viterbi decoding. However, an upper bound can be derived based on the information contained in the extended generating function of a given code.

Example 5.9

The extended generating function of a convolutional code is similar to the modified generating function except that the former is based on the extended state diagram. The extended state diagram for the (2,1,3) code is illustrated in Figure 5.17. Compared with the modified state diagram in Example 5.7, two more variables are introduced: Y^i and Z^i. The exponent of Y^i is the Hamming weight of the associated encoder input. The exponent of Z^i counts the number of branches through which a path has passed. For instance, the input bit associated with branch $S(00)_s$→$S(10)$ is 1; a Y is then assigned to the branch. Furthermore, the path $S(00)_s$→$S(10)$ passes through

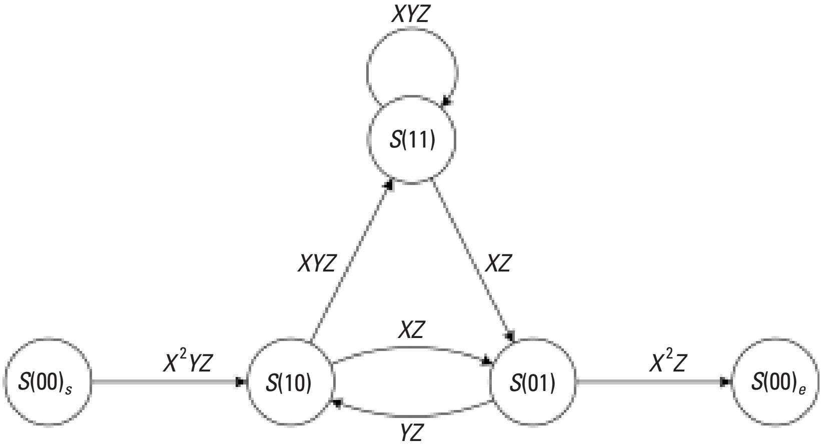


Figure 5.17 Extended state diagram.

one branch, so a Z is added. The label on the branch S(00)$_s$→S(10) becomes X^2YZ. The state equations are obtained as follows:

$$\begin{cases} X(10) = X^2YZ \cdot X(00)_s + YZ \cdot X(01) \\ X(01) = XZ \cdot X(10) + XZ \cdot X(11) \\ X(11) = XYZ \cdot X(10) + XYZ \cdot X(11) \\ X(00)_e = X^2Z \cdot X(01) \end{cases} \tag{5.18}$$

The extended generating function is defined in (5.19), and the result is obtained in (5.20) by solving the state equations in (5.18):

$$\begin{aligned} T(X,Y,Z) &\triangleq X(00)_s / X(00)_e \\ &= X^5YZ^3 / [1 - XYZ(1+Z)] \end{aligned} \tag{5.19}$$

$$= X^5YZ^3 + X^6Y^2Z^4(1+Z) + X^7Y^3Z^5(1+Z)^2 + \cdots \tag{5.20}$$

The extended generating function contains more information than the modified generating function. Take the first term in (5.20) as an example. The term X^5YZ^3 indicates that there is one path with the Hamming weight 5 (seen from X^5) that is 3 branches long (seen from Z^3) and is generated *by an input of weight 1. One more example: The second term $X^6Y^2Z^4$ $(1 + Z)$ tells us that there are two paths with weight 6. One is 4 branches long, and the other 5 branches long. Both are generated by an input of weight 2.*

With the extended generating function available, a loose BER bound P_B for Viterbi decoding can be derived [9, 10]:

$$P_B \approx \begin{cases} \left.\dfrac{\partial T(X,Y,Z)}{\partial Y}\right|_{X=2\sqrt{p_x(1-p_x)},Y=1,Z=1} & \text{for BSC} \\ \left.\dfrac{\partial T(X,Y,Z)}{\partial Y}\right|_{X=e^{-RE_b/N_0},Y=1,Z=1} & \text{for AWGN} \end{cases} \tag{5.21}$$

where the BSC has the crossover probability p_x, the AWGN channel has the noise PSD N_0, E_b is the energy per information bit, and R is the coding rate = $1/n$. Note that, if $k \neq 1$, the right-hand side of both (5.21) must be divided by k.

Example 5.10

Let us now calculate the BER bounds for the (2,1,3) code in both the BSC and AWGN channel. Substituting (5.20) into (5.21), we get:

$$P_B \approx \left.\frac{\partial T(X,Y,Z)}{\partial Y}\right|_{Y=1,Z=1} = X^5 + 4\cdot X^6 + 12\cdot X^7 + \cdots \tag{5.22}$$

where $X = 2\sqrt{p_x(1-p_x)}$ if the channel is BSC, or $X = e^{-RE_b/N_0}$ if AWGN. Because p_x and e^{-RB_b/N_0} are both <<1 in practice, the right-hand side of (5.22) is dominated by the first term. By ignoring the higher-order terms in (5.22) the BER bounds are obtained for both the BSC and AWGN channels as follows:

$$P_B \approx \begin{cases} \left.X^5\right|_{X=2\sqrt{p_x(1-p_x)}} = 32\cdot\left[p_x(1-p_x)\right]^{2.5} & \text{for BSC} \\ \left.X^5\right|_{X=e^{-RE_b/N_0}} = e^{-2.5E_b/N_0} & \text{for AWGN} \end{cases} \tag{5.23}$$

Note that the above bound computation has assumed BPSK signaling. The crossover probability p_x is related to E_b/N_0 as:

$$p_x = Q\left(\sqrt{2RE_b / N_0}\right) = Q\left(\sqrt{E_b / N_0}\right) \tag{5.24}$$

where $Q(x) = \int_x^{\infty} \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt$ *is the Q-function.*

MATLAB Experiment 5.6

The m-files `convsim1.m*` and `convsim2.m*` numerically simulate BER for the example code in the BSC and AWGN channels, respectively. Run the programs and obtain the error probability curves.

Example 5.10 has shown that the BER bound of the (2,1,3) code is calculated as $P_B \approx X^5 + 4X^6 + 12X^7 + \cdots \approx X^5$. The following expression is the general BER bound expression for any convolutional code:

$$P_{\mathrm{B}} \approx b_{d_{\mathrm{free}}} X^{d_{\mathrm{free}}} + b_{d_{\mathrm{free}}+1} X^{d_{\mathrm{free}}+1} + b_{d_{\mathrm{free}}+2} X^{d_{\mathrm{free}}+2} + \cdots \approx b_{d_{\mathrm{free}}} X^{d_{\mathrm{free}}} \tag{5.25}$$

where $b_{d_{\mathrm{free}}}, b_{d_{\mathrm{free}}+1}, b_{d_{\mathrm{free}}+2}, \cdots$ are the coefficients, and d_{free} is the free distance.

Besides BER, coding gain is also frequently used as another performance measure. For hard-decision decoding, $X = 2\sqrt{p_x(1 - p_x)}$ and

$$p_x = Q\left(\sqrt{2R\frac{E_{\mathrm{b}}}{N_0}}\right) < \frac{1}{2} e^{-\frac{RE_{\mathrm{b}}}{N_0}} \tag{5.26}$$

Substituting the above X expression and (5.26) into (5.25) results in the following for small p_x:

$$\begin{aligned} P_{\mathrm{B}} &\approx b_{d_{\mathrm{free}}} \left[2\sqrt{p_x\left(1 - p_x\right)}\right]^{d_{\mathrm{free}}} \\ &\approx 2^{d_{\mathrm{free}}} \cdot b_{d_{\mathrm{free}}} \cdot p_x^{\frac{d_{\mathrm{free}}}{2}} < 2^{d_{\mathrm{free}}-1} \cdot b_{d_{\mathrm{free}}} \cdot e^{-\frac{Rd_{\mathrm{free}}E_{\mathrm{b}}}{2N_0}} \end{aligned} \tag{5.27}$$

On the other hand, the uncoded performance is bounded by

$$P_B = Q\left(\sqrt{2E_b / N_0}\right) < \frac{1}{2} e^{-\frac{E_b}{N_0}} \tag{5.28}$$

Comparing the exponents in (5.27) and (5.28), it is easy to see that coding has yielded an advantage of $Rd_{\text{free}} / 2$ over the uncoded system in terms of E_b/N_0. So, the coding gain is:

$$K_{\text{BSC}} \approx Rd_{\text{free}} / 2 \tag{5.29}$$

Similarly, for soft-decision decoding, the coding gain is found to be:

$$K_{\text{AWGN}} \approx Rd_{\text{free}} \tag{5.30}$$

From (5.29) and (5.30) we can see that soft-decision decoding is roughly 3 dB better than hard-decision decoding. The numerical simulation results presented in Figure 5.16 confirm our observation.

5.2.2 Sequential Decoding

Although the complexity of the Viterbi algorithm increases linearly with the sequence length, it is still an exponential function of the constraint length υ (see the previous section). As a result, the algorithm is only good for relatively short constraint length ($\upsilon < 10$). Notice that in a decoding process there always exist some paths that appear better than the others. By "better" we mean that these paths are more likely to be the correct path. Sequential decoding is such a decoding approach that concentrates only on better paths and consequently eases its computational burden. Although the technique is suboptimal, it is attractive in decoding long constraint length codes (for υ as large as 50) since its computational complexity is independent of υ. Several algorithms have been developed. The *stack algorithm* [11] and the *Fano algorithm* [12] are the most recognized two among them.

5.2.2.1 Fano Metric

Recall that in the Viterbi algorithm the Hamming distance metric or the squared Euclidian distance metric is used for decoding. Analogously the stack algorithm and the Fano algorithm use the *Fano metric* to perform sequential decoding. Consider a code tree branch with codeword $\boldsymbol{c} = (c_1 c_2 \cdots c_n)$. The Fano metric of the branch is defined as:

$$\lambda = \sum_{i=1}^{n} \underbrace{\log_2 \left[P(r_i \mid c_i) / P(r_i)\right] - R}_{\lambda_i} \tag{5.31}$$

where $\boldsymbol{r} = (r_1 r_2 \cdots r_n)$ is the received vector corresponding to $\boldsymbol{c}$, $P(r_i|c_i)$ denotes the probability of r_i conditional on c_i, R is the coding rate, and $P(r_i)$ can be computed using (5.32):

$$P(r_i) = \sum\nolimits_{\{c_i\}} P(c_i)P(r_i \,|\, c_i) = (1/2)\cdot P(r_i \,|\, c_i = 0) + (1/2)\cdot P(r_i \,|\, c_i = 1) \tag{5.32}$$

where $c_i \in \{0, 1\}$, and we assume that code bits 0 and 1 are transmitted with equal probability 1/2: $P(c_i = 0) = P(c_i = 1) = 1/2$.

For BSC with a crossover probability of p_x, by definition the following exist:

$$\begin{aligned} P(r_i = 0 \,|\, c_i = 1) &= P(r_i = 1 \,|\, c_i = 0) = p_x \\ P(r_i = 0 \,|\, c_i = 0) &= P(r_i = 1 \,|\, c_i = 1) = 1 - p_x \end{aligned} \tag{5.33}$$

By inserting (5.32) and (5.33) into (5.31), we obtain the Fano metric expression for BSC:

$$\lambda_i = \begin{cases} \log_2(1 - p_x) + (1 - R) & r_i = c_i \\ \log_2(p_x) + (1 - R) & r_i \neq c_i \end{cases} \tag{5.34}$$

The Fano metric of a path starting from the tree root to the current node is the summation of the Fano metrics of all branches on that path:

$$\Lambda = \sum\nolimits_{\{\text{branch}\}} \lambda \tag{5.35}$$

Similar to the path metric calculation in the Viterbi algorithm, Λ can also be computed recursively as follows:

$$\Lambda_S = \Lambda_C + \lambda_{C \to S} \tag{5.36}$$

where the subscript C represents the current node, S represents the successor node, and $C \to S$ represents the transition from C to S.

The branch Fano metric of an $R = 1/2$ code in a BSC with a p_x of 0.1 is 0.3480 for $r_i = c_i$ and –2.8219 for $r_i \neq c_i$. From this we can imagine that a correct path will have a moderately increasing metric and an incorrect path will have a sharply decreasing metric. This is, in fact, a property that the metric was purposely designed to have, and it is this property that the two sequential decoding algorithms use to distinguish between the correct path and the incorrect path.

MATLAB Experiment 5.7

A simple MATLAB routine `fanom`* is provided to compute the Fano metric.

```
>> r = [0 0]; c = [0 0]; px = 0.1; R = 0.5;
>>                            % crossover 0.1, coding rate 0.5
>> fanom(r,c,px,R)
ans =
     0.6960
```

Now we are ready to present the stack algorithm and the Fano algorithm. The stack algorithm is presented first for its conceptual simplicity.

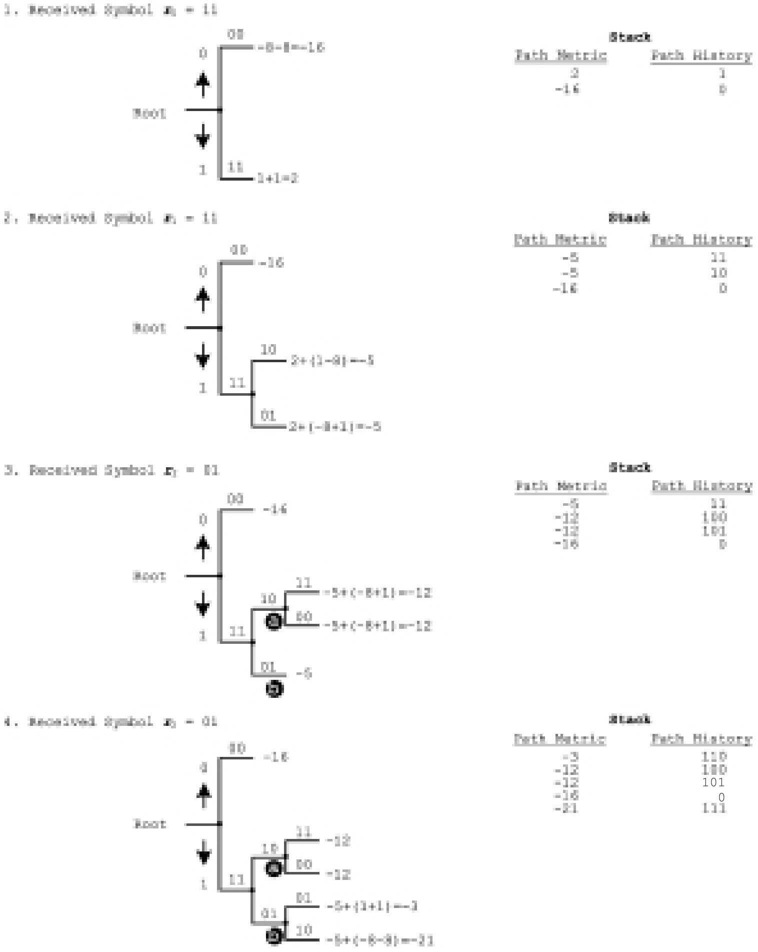
5.2.2.2 Stack Algorithm

Starting from the tree root of a given code, at each decoding step, the stack algorithm calculates the branch metric λ for each current branch, and computes the path metric Λ for each current node. The path metrics, together with the corresponding paths, are stored in a stack in the descending order; that is, the path with the largest metric is placed at the top of the stack, and the path with the smallest metric at the bottom. The top path is then chosen as the best path and is extended with its successor branches. The procedure repeats itself until the top path reaches the end of the code tree. Then the top path is taken as the final decoding path. By keeping extending the best path, the algorithm maximizes the probability that the resultant path is the correct path.

Example 5.11

Consider the (2,1,3) example code and the message sequence $\vec{m} = 1,1,0,1,0,0$. The code sequence $\vec{c}$ is 11, 01, 01, 00, 10, 11. Assume that the channel is a BSC with $p_X = 0.1$ and the received sequence $\vec{r}$ is 11, 11, 01, 00, 10, 11 with one error in its third position.

The process of decoding $\vec{r}$ using the stack algorithm is shown in Figure 5.18. For the sake of simplicity, the branch metrics calculated are normalized as $0.3480 \rightarrow 1$ for $r_i = c_i$ and $-2.8219 \rightarrow -8$ for $r_i \neq c_i$. Notice that the second decoding step ends up with two equally good paths with the same metric value –5. We arbitrarily choose to extend node a. However, step 3 proves it an incorrect decision. So, at the fourth step, we come back to



Stack (step 1)

Path Metric	Path History
2	1
-16	0

Stack (step 2)

Path Metric	Path History
-5	11
-5	10
-16	0

Stack (step 3)

Path Metric	Path History
-5	11
-12	100
-12	101
-16	0

Stack (step 4)

Path Metric	Path History
-3	110
-12	100
-12	101
-16	0
-21	111

Figure 5.18 Stack decoding process.

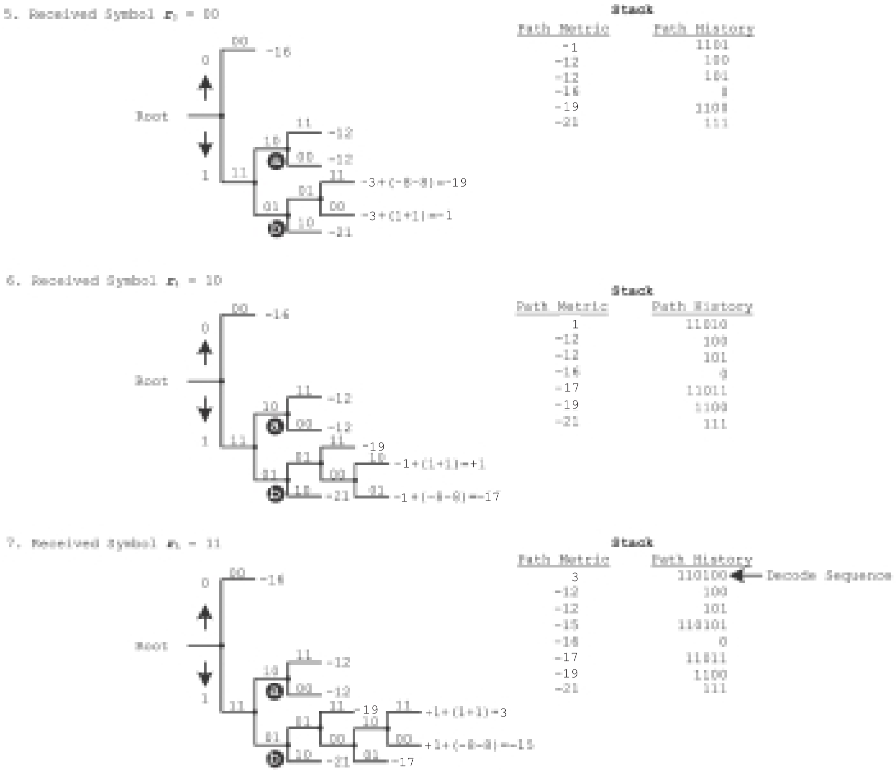
extend node b. The received symbol $\mathbf{r}_2$ *is therefore used twice in the decoding process, once at step 3 and the other time at step 4.*

The stack algorithm can be summarized as follows.

Stack Algorithm

Initialization:

Load the stack with the root of the code tree and set its path metric to zero.



Stack

Path Metric	Path History
-1	1101
-12	100
-12	101
-16	0
-19	1100
-21	111

Stack

Path Metric	Path History
1	11010
-12	100
-12	101
-16	0
-17	11011
-19	1100
-21	111

Stack

Path Metric	Path History
3	110100 ← Decode Sequence
-12	100
-12	101
-15	110101
-16	0
-17	11011
-19	1100
-21	111

Figure 5.18 (*Continued*)

Normal Operation:

1. Extend the best path (i.e., the path at the top of stack) with its 2^k successor branches.
2. Compute the metrics of the extended paths.
3. Store all of the paths and their path metrics in the stack from top to bottom in the order of decreasing metric values.
4. If the top path has reached the end of the code tree, stop and output it as the decoded sequence. Otherwise, return to 1.

The flowchart for the stack algorithm is shown in Figure 5.19.

The amount of computation needed in the algorithm varies with the channel condition. If the channel SNR is high, little computation is needed.

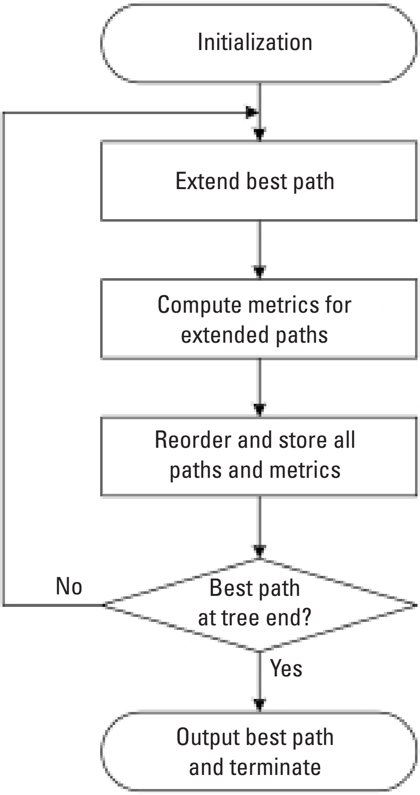

Initialization
Extend best path
Compute metrics for extended paths
Reorder and store all paths and metrics
No
Best path at tree end?
Yes
Output best path and terminate

Figure 5.19 Flowchart for the stack algorithm.

Otherwise, more computation is expected. Recall that in the example the received symbol r_2 was used twice, once at decoding step 3 and the other time at step 4. Consequently, the stack algorithm needs extra storage (in addition to the stack storage) to store the received symbols for possible reuse in the future.

MATLAB Experiment 5.8

The companion DVD contains a program `stackdemo.m*` that simulates the stack decoder for the previous example.

```
>> r = [1 1; 1 1; 0 1; 0 0; 1 0; 11]; % received sequence
>> px = 0.1;                           % channel crossover prob.
>> stackdemo
decoded =
       1  1  0  1  0  0
```

Step through the program. What difference did you notice between the simulation and Example 5.11?

5.2.2.3 Fano Algorithm

The Fano algorithm works differently than the stack algorithm. The key operation here is the continuous comparison of the path metric with a changing threshold T. When the path metric exceeds T, the decoder regards the path as the correct path and moves forward. If the metric falls below the threshold, the algorithm goes backward and searches for a better path. The threshold is tightened whenever possible so that the algorithm will only keep moving forward on a path whose metric is increasing.

The algorithm relies on the information of the following three paths to carry out the decoding: the current path and its metric Λ_C, the immediate predecessor and its metric Λ_P, and one of the successors and its metric Λ_S. Figure 5.20 shows a flowchart for the algorithm. Although at first glance the chart seems rather complicated, it basically describes the following three procedures.

Fano Algorithm

Initialization:

Start from the root of the code tree and reset both the initial threshold and the path metric to 0.

Normal Operation:

1. At the current tree node C, the algorithm tries to move forward by doing the following: First, the metrics of all 2^k possible successor branches are computed and the branch with the maximum branch metric is selected to extend the current best path. Then Λ_S of the extended path is computed according to (5.36). If $\Lambda_S \geq T$, the algorithm advances to the successor node S, at which point T is increased by the largest possible multiple of a fixed increment Δ without exceeding the updated metric, if this is the first time visit to this node; that is, $T \leftarrow T + K\Delta$ (where K is the largest possible integer). Otherwise, T remains unchanged. The algorithm repeats until the end of the tree is reached. This procedure corresponds to the main loop 1 in the flowchart.

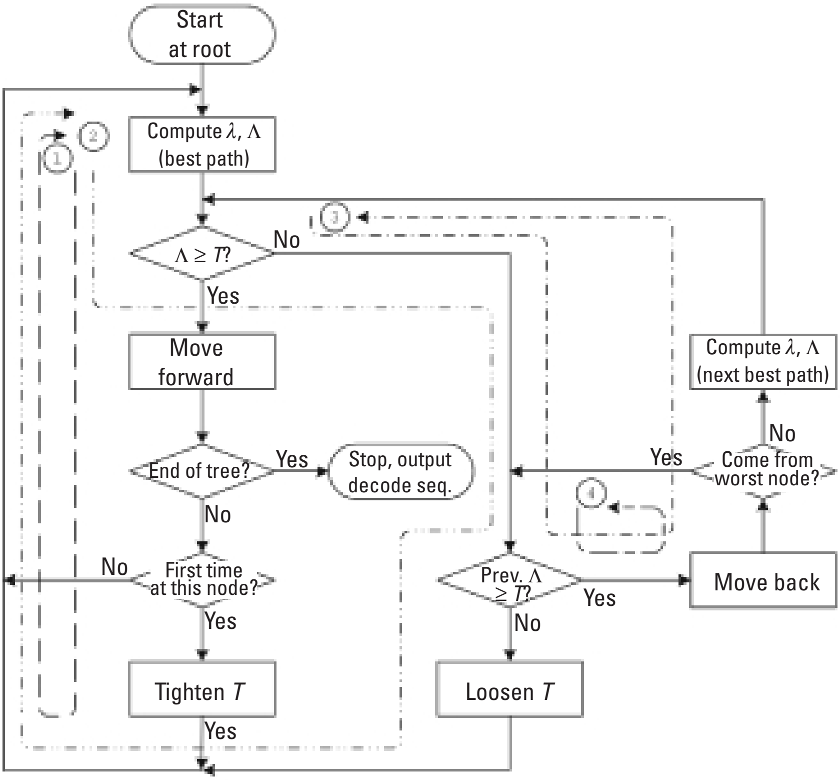


Figure 5.20 Flowchart for the Fano algorithm.

2. If at some point in step 1 it turns out that $\Lambda_S < T$, and moreover $\Lambda_P < T$, the algorithm lowers T by Δ (i.e., $T \leftarrow T - \Delta$), and attempts to go forward from node C by repeating step 1 but with the lowered T. This procedure corresponds to loop 2.
3. The last procedure considers the case where $\Lambda_S < T$ and $\Lambda_P \geq T$. The algorithm first moves backward to node P, and then tries to move forward from node P via the next best path if branch $P \rightarrow C$ is *not* the last remaining successor for P (i.e., the worst node). Otherwise, it moves one more step back (because in this case all possible successor branches of node P have been examined and no other choice is left). This is loop 3 or loop 4.

Example 5.12

The process of Fano decoding for the received sequence in Example 5.11 is shown in Table 5.2 and Figure 5.21. The input sequence records the input bits along the selected path. The value of Δ *is set to 4. Notice that in the last three sections of the code tree each node is only extended by the branch corresponding to the input bit 0; the branch corresponding to bit 1 is eliminated.*

Table 5.2
Fano Decoding Process

#	Node	Input Sequence	Λ_P	Λ_C	Λ_S	T	Scenario	Action	Succ.
1	*r*	—	$-\infty$	0	2	0	$\Lambda_S \ge T$	f	1
2	*a*	1	0	2	−5	0*	$\Lambda_S < T$ and $\Lambda_P \ge T$	b	
3	*r*	—	$-\infty$	0	−16	0	$\Lambda_S < T$ and $\Lambda_P < T$	l	2
4	*r*	—	$-\infty$	0	2	−4	$\Lambda_S \ge T$	f	1
5	*a*	1	0	2	−5	−4	$\Lambda_S < T$ and $\Lambda_P \ge T$	b	
6	*r*	—	$-\infty$	0	−16	−4	$\Lambda_S < T$ and $\Lambda_P < T$	l	2
7	*r*	—	$-\infty$	0	2	−8	$\Lambda_S \ge T$	f	1
8	*a*	1	0	2	−5	−8	$\Lambda_S \ge T$	f	
9	*b*	10	2	−5	−12	−8*	$\Lambda_S < T$ and $\Lambda_P \ge T$	b	
10	*a*	1	0	2	−5	−8	$\Lambda_S \ge T$	f	
11	*e*	11	2	−5	−3	−8*	$\Lambda_S \ge T$	f	1
12	*d*	110	−5	−3	−1	−4*	$\Lambda_S \ge T$	f	1
13	*e*	1101	−3	−1	1	−4*	$\Lambda_S \ge T$	f	1
14	*j*	11010	−1	1	3	0*	$\Lambda_S \ge T$	f	1
15	*g*	110100	1	3	—	4*	—	—	—

↑ Decode Sequence

Note: f: move forward, b: move backward, l: lower threshold; 1: best successor, 2: second best successor; *: threshold tightened (the actual number may not change); succ.: successor.

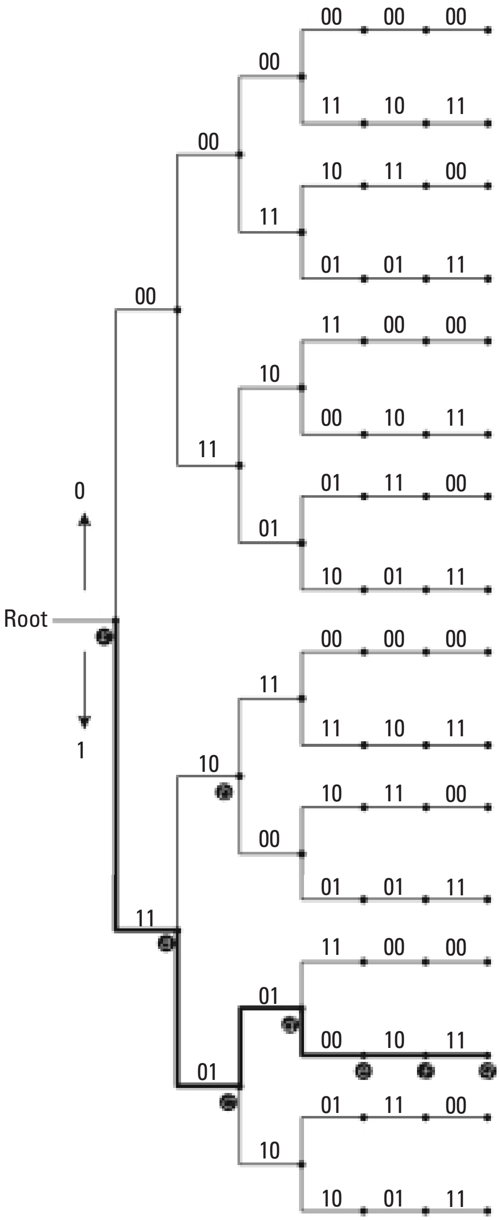


Figure 5.21 A Fano decoding process.

The simplification is made because the last two bits of the decoded sequence are the flushing bits and can only be 0 0.

Similar to the stack algorithm case, the amount of computation required in the Fano algorithm goes up or down as the channel SNR rises or drops. Although the number of computations in the Fano algorithm usually exceeds that in the stack algorithm, the Fano algorithm is nevertheless preferred in hardware implementation because it does not need stack storage.

5.3 Designing Viterbi Decoders

This section addresses some key issues relating to the implementation of the Viterbi algorithm.

5.3.1 Typical Design Issues

5.3.1.1 Design Partition

A typical Viterbi decoder is partitioned into three data path units: a branch metric generation unit, a path metric update unit, and a global survivor decoding unit. The branch metric unit takes the received symbol in and computes the corresponding branch metrics. Given the branch metrics by the branch metric unit, the path metric update unit performs the ACS operation, stores updated path metrics in its path metric memory, and produces a decision signal indicating which paths are the survivors. The survivor decoding unit updates survivor path history based on the decision signal, and at the end of the decoding process finds the global optimum path in the survivor paths. A control unit coordinates the operations. Figure 5.22 shows a block diagram for a typical Viterbi decoder.

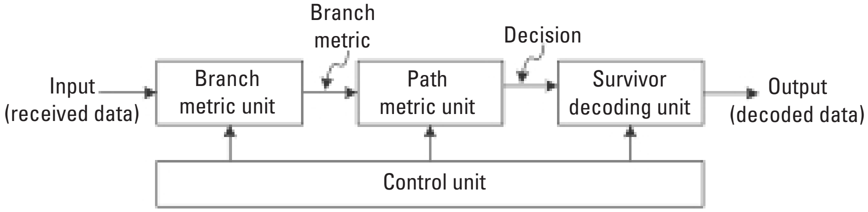


Figure 5.22 Viterbi decoder block diagram.

5.3.1.2 Signal Quantization

The soft-decision Viterbi decoding in Example 5.8 tackles real numbers with infinite precision. However, actual decoders have to quantize the received signal with a finite word length at the expense of some performance loss. Needless to say, higher precision incurs less performance loss, but at the same time means more implementation cost. Observations made in the past have found that a 3- to 5-bit quantization usually gives a satisfactory trade-off. Study shows that a 3-bit quantization for a rate-1/2 code incurs a coding gain loss as small as 0.14 dB [13].

MATLAB Experiment 5.9

Repeat MATLAB Experiment 5.5 with a quantization of 8 levels (3 bits), 16 levels (4 bits), and 32 levels (5 bits). The results are plotted in Figure 5.23. The difference between the 8-level quantization and the unquan-

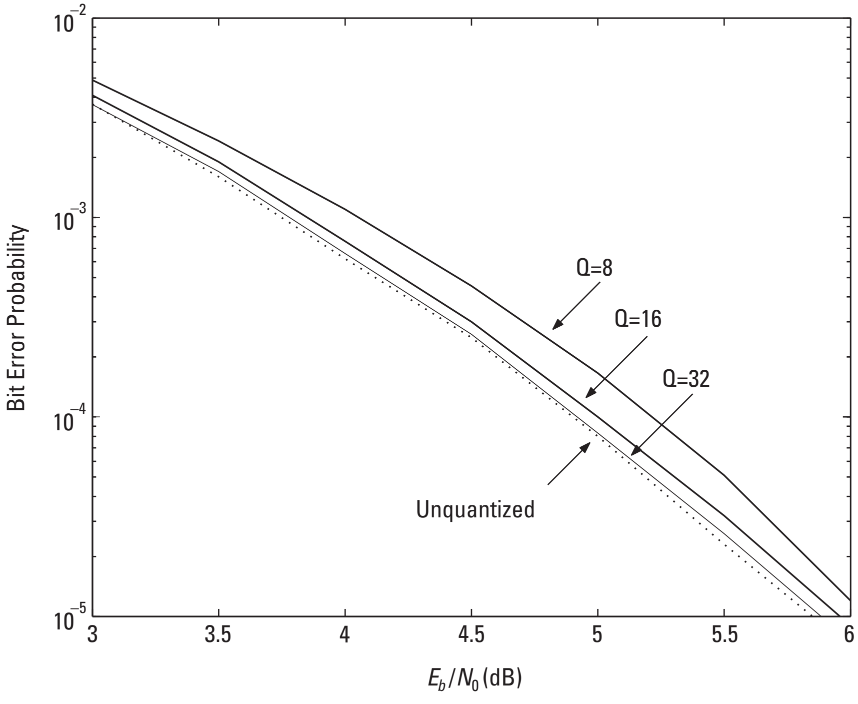


Figure 5.23 Performance of Viterbi decoding with different quantization levels.

tized is indeed only a fraction of decibels. (Note that your result may vary a bit because the experiment uses random numbers.)

5.3.1.3 Metric Normalization

The path metric in the Viterbi algorithm is a continuous accumulation of branch metrics and, therefore, is unbounded. Overflow occurs when it exceeds the word length and leads to an erroneous decoding result. So some sort of normalization must be in place to prevent such overflow.

The most straightforward solution is based on the following property existing in the Viterbi algorithm:

> Property 1: *The outcome of Viterbi decoding depends only on the differences of metrics.*

The method avoids metric overflow by subtracting a constant from the path metrics when the smallest metric exceeds the constant. Since the subtraction is made to *all* path metrics, the metric difference is unaffected, and so is the decoding result. The method is conceptually simple, but it involves extra computations such as comparisons and subtractions.

An alternative solution [14] favored in practical designs utilizes another property of the Viterbi algorithm as follows:

> Property 2: *The path metric differences are bounded by a fixed quantity of*
>
> $$\sigma_{\max} = \kappa \cdot M \tag{5.37}$$
>
> where κ is the maximum branch metric and M is the memory depth of the code.

The method computes the path metrics modulo U so that their magnitudes will always fall within $[0, U)$, and U is computed as:

$$U = 2\sigma_{\max} \tag{5.38}$$

This translates to the following minimum number of bits B to be used to represent the path metric:

$$B = \lceil \log_2(2\sigma_{\max}) \rceil \tag{5.39}$$

Table 5.3
Initial Path Metrics for Example 5.13

Path Metric	Value in Decimal	Value in Binary
PM_1	19	10011
PM_2	13	01101
PM_3	15	01111

where $\lceil x \rceil$ means the smallest integer not smaller than x. The trick is that, by using two's-complement arithmetic in path metric comparisons, a correct metric comparison (and ultimately correct decoding) can be guaranteed regardless of overflow. We use an example to illustrate the technique.

Example 5.13

Given the three path metrics shown in Table 5.3, in which PM_1 is the largest, followed by PM_3 and then PM_2. Normally at this point we need 5 bits to represent the metrics to avoid overflow. However, based on the method above, we need only 4 bits (because the maximum path metric difference $\sigma_{max} = PM_1 - PM_2 = 6$ and $\lceil \log_2(2 \times 6) \rceil = 4$). As a result, PM_1 is truncated to $(0011)_2$ and overflow occurs; the other two are unaffected (see Table 5.4).

Using two's-complement arithmetic to compare the above three binary numbers, we have $PM_2^* = -4 < PM_3^* = -1 < PM_1^* = 3$. So the comparison result is still correct even though overflow has occurred.

The maximum path metric difference is usually unknown to us in practice, and often is determined through extensive numerical simulations.

Table 5.4
Truncated Path Metrics for Example 5.13

Truncated Metric	Value in Decimal	Value in Binary	Overflow
PM_1*	3	0011	Yes
PM_2*	13	1101	No
PM_3*	15	1111	No

5.3.1.4 Survivor Path Management

The Viterbi algorithm requires that $2^{\upsilon-1}$ survivor paths be kept in storage for final decoding use. One essential task in a Viterbi decoder is to properly manage the storage so that decoding can be performed effectively. Basically two methods are frequently used in practice: the *register-exchange method* and the *traceback method*. Of the two methods, the former can run at much faster than the latter but is more power consuming. They are now introduced through two examples.

Example 5.14

This example shows how the register-exchange method works when it is applied to Example 5.8.

The method uses a set of 4 by 6 registers to store the survivor paths, corresponding to the 4 × 6 nodes in the decoding trellis. The complete decoding process is illustrated in Figure 5.24. Take node (S(01),4) as an example. At decoding step 4, of the two paths going into node (S(01),4), the path from node (S(10),3) is selected by ACS as the survivor for node (S(01),4). The register associated with node (S(01),4) copies the register of node (S(10),3) (whose content is "S(00)S(10)S(01)") and appends "S(10)" to form the survivor path for node (S(01),4) (i.e., "S(00)S(10)S(01)S(10)"). The path is then saved in the register. When the decoding comes to the end, the content of the register for node (S(00),6) is filled with the decoding path.[6] The decoding sequence is obtained by mapping the decoding path to the encoder input based on Table 5.1:

$$\begin{array}{ccccccccccccc} S(00) & \rightarrow & S(10) & \rightarrow & S(11) & \rightarrow & S(01) & \rightarrow & S(10) & \rightarrow & S(01) & \rightarrow & S(00) \\ & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & \\ & 1 & & 1 & & 0 & & 1 & & 0 & & 0 & \end{array}$$

The method bears the name register-exchange because the registers keep copying each other in the decoding process.

Example 5.15

Redecode the last example but with the traceback method.

Figure 5.25 illustrates the traceback decoding process. The method sets two successive procedures. The first procedure tracks the survivor history and stores it in a so-called traceback memory (also 4 × 6 in size). For example, at

[6] Because of the added flushing bits, the final decoding path must terminate at $S(00)$. See Section 5.1.2.2.

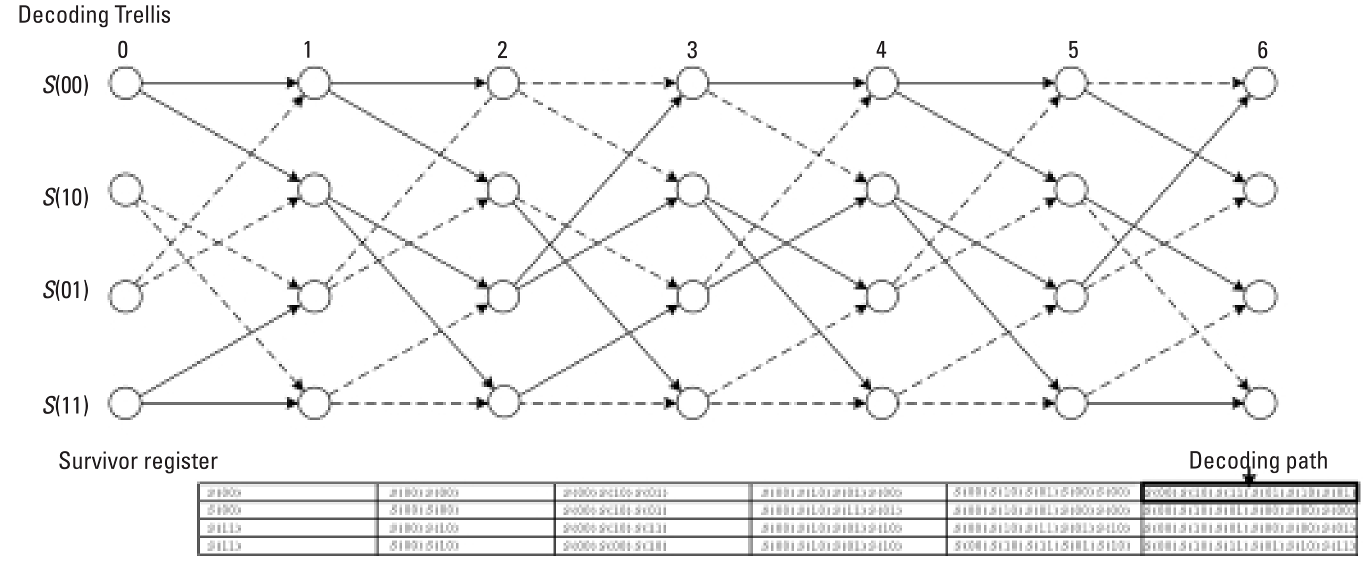
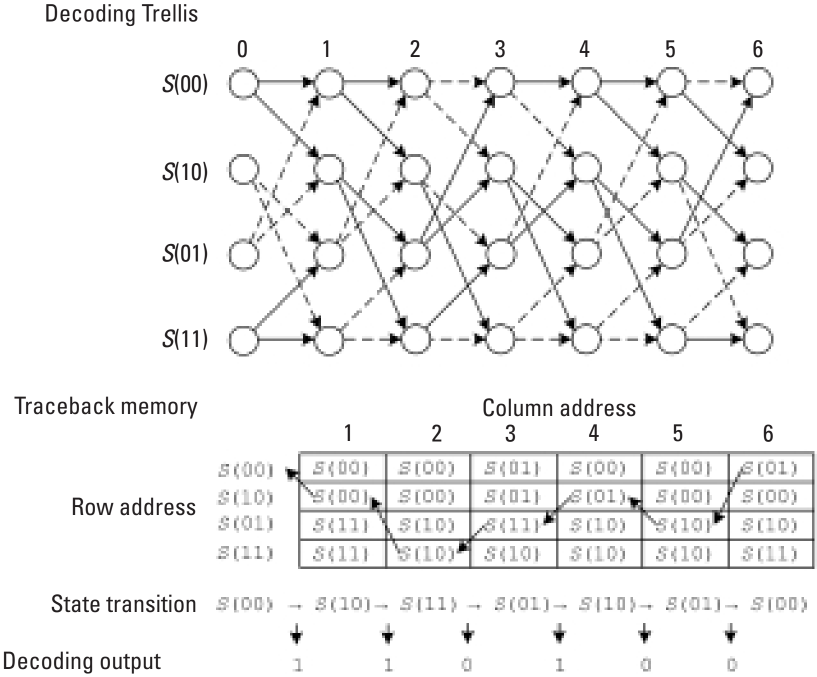


Figure 5.24 Register-exchange decoding process.



	1	2	3	4	5	6
S(00)	S(00)	S(00)	S(01)	S(00)	S(00)	S(01)
S(10)	S(00)	S(00)	S(01)	S(01)	S(00)	S(00)
S(01)	S(11)	S(10)	S(11)	S(10)	S(10)	S(10)
S(11)	S(11)	S(10)	S(10)	S(10)	S(10)	S(11)

State transition S(00) → S(10) → S(11) → S(01) → S(10) → S(01) → S(00)

Decoding output 1 1 0 1 0 0

Figure 5.25 Traceback decoding process.

time 1, the survivor path to node (S(01),1) is from node (S(11),0). S(11) is then stored in the traceback memory location (S(01),1). Next at time 2, the survivor path to node (S(01),2) is from node (S(10),1). S(10) is then stored in the location (S(01),2), and so forth.

The second procedure finds the optimum decoding path by tracing back the survivor history. Starting from the traceback memory location (S(00),6), we see an "S(01)" in that location. Traceback then goes to memory location (S(01),5). Next in (S(01),5) we see that an "S(10)," traceback moves to location (S(10),4), and so forth. The sequence S(00)→S(01)→S(10)→ ··· forms the optimum path (in reverse order). Notice that while the first procedure is carried out forward, the second procedure is backward.

Similar to the register-exchange, the final decoding sequence is also obtained by mapping the optimum path to the encoder input bits.

The preceding examples are illustrative only. Some minor modifications could simplify the implementation complexity effectively. For instance, instead of keeping the sequence of states in the register exchange, we may store the sequence of the information bits, and at the end the register contains the decoded output directly. Also, in traceback, we can use the ACS decisions rather than the states as the traceback pointers so that the space of the traceback storage can be reduced. The details are left as a problem for the reader.

One more remark is that the two methods require a total of $N \times L$ independent storages in general, where N is the number of states, L is either the sequence length for frame-by-frame decoding, or L equals to the decode length plus the decoding depth for sliding-window decoding.

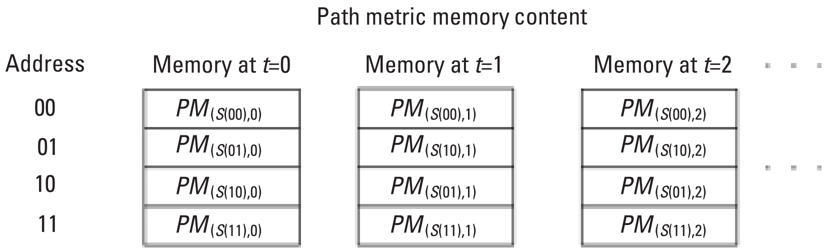
5.3.1.5 Path Metric Memory Management

ACS uses path metrics at time t to update path metrics at time $t + 1$. A naïve design would then use two path metric memories, one for storing the old metrics and the other for the new metrics. A more efficient way is to use the in-place computation and save one memory [15].

To enable the strategy, ACS should be performed on the butterfly substructure. Take a closer look at the butterfly substructure in Figure 5.6, we find that the old *PM* of, for instance, states $S(00)$ and $S(01)$ are used solely for updating the *PM* of states $S(00)$ and $S(10)$ and are no longer needed after the update. This tells us that the new metrics can write over the old metrics. The address to store the updated *PM* is one bit right-rotation of the address in which *PM* was originally placed:

$$\text{New Address} = \text{Right cyclic shift of } (a_{n-1}a_{n-2}\cdots a_2a_1a_0) = a_0a_{n-1}a_{n-2}\cdots a_2a_1 \tag{5.40}$$

where $a_{n-1}\ a_{n-2}\ \cdots\ a_2a_1a_0$ is the address of *PM* before updating. Figure 5.26 shows an example for four states. The initial path metrics $PM_{(S(00),0)}$, $PM_{(S(01),0)}$, $PM_{(S(10),0)}$, and $PM_{(S(11),0)}$, are placed in addresses 00, 01, 10, and 11, respectively. The address for $PM_{(S(00),1)}$, at time $t = 1$ is still 00 because



Path metric memory content

Address	Memory at t=0	Memory at t=1	Memory at t=2	. . .
00	$PM_{(S(00),0)}$	$PM_{(S(00),1)}$	$PM_{(S(00),2)}$	
01	$PM_{(S(01),0)}$	$PM_{(S(10),1)}$	$PM_{(S(10),2)}$	. . .
10	$PM_{(S(10),0)}$	$PM_{(S(01),1)}$	$PM_{(S(01),2)}$	
11	$PM_{(S(11),0)}$	$PM_{(S(11),1)}$	$PM_{(S(11),2)}$	

Figure 5.26 Path metric memory management.

one bit right-rotation of 00 is again 00. The address for $PM_{(S(01),1)}$, at $t = 1$ becomes 10 because one bit right-rotation of 01 is 10, and so forth.

5.3.2 Design for High Performance

5.3.2.1 From ACS to CAS

High-speed implementation of the Viterbi algorithm is a challenging task. One bottleneck is ACS operation. Due to the sequential execution of the add, compare, and select operations, the ACS method is very time consuming. Moreover, ACS is feedback in nature and its speed is hard to raise via conventional pipelining and/or parallelization.

Many possible solutions have been suggested in the past. One proposal is to replace ACS with CAS, that is, compare-add-select. By breaking the boundary of the ACS iteration, the CAS method performs the compare and add operations in parallel before select (Figure 5.27). A 34% increase in speed has been achieved at the expense of a fairly modest silicon area increase [16].

Readers will probably find that in Figure 5.27 the formation of CSA is fairly straightforward, but the transformation from CSA to CAS appears a little more difficult to understand. A functional table (Table 5.5) is provided to help verify that CSA and CAS are functionally equivalent. Under the same input conditions, the outputs of both CSA and CAS are indeed identical.

5.3.2.2 Scarce-State-Transition Decoding

Low-power design at an algorithmic level has been focusing on reducing circuit switchings, because dynamic power constitutes the largest power consumption in complementary metal-oxide-semiconductor (CMOS). The scarce-state-transition (SST) Viterbi decoding uses a simple predecoding to minimize the state transitions, thereby reducing the circuit on/off switching activities [17]. Shown in Figure 5.28, such a decoder consists of an SST processing module and a conventional Viterbi decoder. The re-encoder in the SST module is simply a duplicate of the original encoder. The predecoder is designed so that when there are no channel errors, the input to the conventional decoder is an all-zero sequence and, thus, no state transition takes place. Consequently, when occasional errors occur the state transitions will be limited. The technique is able to effectively cut down power dissipation [18]. Study has also shown that an SST Viterbi decoder requires a shorter decoding depth than its conventional counterpart and therefore can be used for area reduction as well [19].

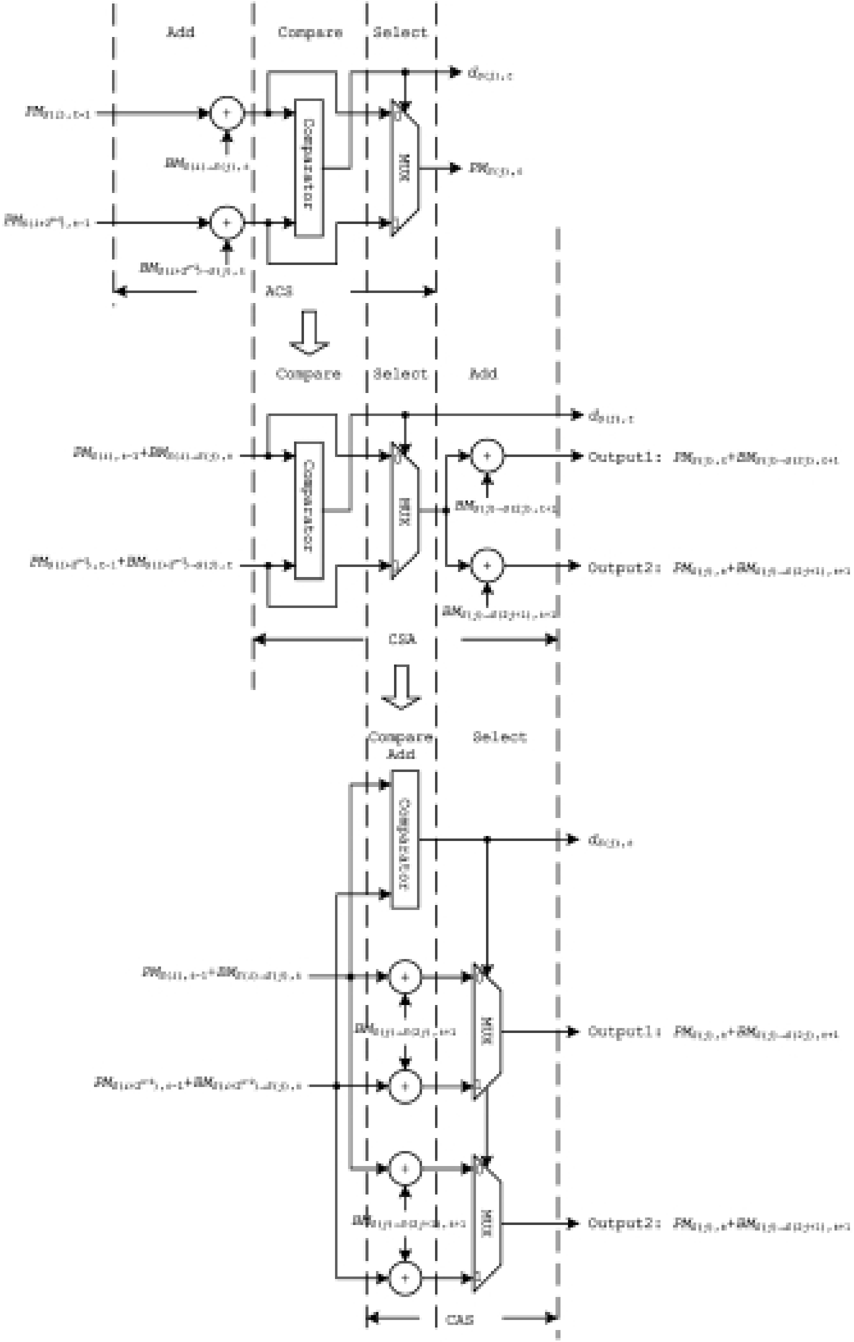


Figure 5.27 From ACS to CAS.

Table 5.5
Functional Table of CSA and CAS

Condition		$\boldsymbol{PM_{1,t-1}+BM_{1,t}<PM_{2,t-1}+BM_{2,t}}$	$\boldsymbol{PM_{1,t-1}+BM_{1,t}>PM_{2,t-1}+BM_{2,t}}$
CSA	Decision $d_{S,t}$	0	1
	MUX output	$PM_{1,t-1}+BM_{1,t}$	$PM_{2,t-1}+BM_{2,t}$
	Output 1	$PM_{1,t-1}+BM_{1,t}+BM_{1,t+1}$	$PM_{2,t-1}+BM_{2,t}+BM_{1,t+1}$
	Output 2	$PM_{1,t-1}+BM_{1,t}+BM_{2,t+1}$	$PM_{2,t-1}+BM_{2,t}+BM_{2,t+1}$
CAS	Decision $d_{S,t}$	0	1
	Output 1	$PM_{1,t-1}+BM_{1,t}+BM_{1,t+1}$	$PM_{2,t-1}+BM_{2,t}+BM_{1,t+1}$
	Output 2	$PM_{1,t-1}+BM_{1,t}+BM_{2,t+1}$	$PM_{2,t-1}+BM_{2,t}+BM_{2,t+1}$

Example 5.16

Now let us design an SST Viterbi decoder for the (2,1,3) example code.[7]

Given a binary message sequence of $m_0, m_1, \cdots, m_{t-1}, m_t, \cdots$, *the codeword at time t,* $\boldsymbol{c}_t = (c_t^{(1)}c_t^{(2)})$, *is produced by:*

$$c_t^{(1)} = m_t \oplus m_{t-1} \oplus m_{t-2} \quad \text{and} \quad c_t^{(2)} = m_t \oplus m_{t-2} \tag{5.41}$$

Let $\boldsymbol{e}_t = (e_t^{(1)}e_t^{(2)})$ *be the channel error. The received symbol* $\boldsymbol{r}_t = (r_t^{(1)}r_t^{(2)})$ *is the codeword corrupted by the channel error and can be expressed as:*

$$r_t^{(1)} = c_t^{(1)} \oplus e_t^{(1)} \text{ and } r_t^{(2)} = c_t^{(2)} \oplus e_t^{(2)} \tag{5.42}$$

On the other hand, the output of the reencoder $\overline{\boldsymbol{r}}_t = (\overline{r}_t^{(1)}\overline{r}_t^{(2)})$ *is:*

$$\overline{r}_t^{(1)} = q_t \oplus q_{t-1} \oplus q_{t-2} \text{ and } \overline{r}_t^{(2)} = q_t \oplus q_{t-2} \tag{5.43}$$

As we have said, we want $\boldsymbol{p}_t$ *to equal (00) when* $\boldsymbol{e}_t$ *is (00). This gives the following:*

$$\begin{aligned} p_t^{(1)} &= r_{t-\tau_1}^{(1)} \oplus \overline{r}_t^{(1)} = r_{t-\tau_1}^{(1)} \oplus q_t \oplus q_{t-1} \oplus q_{t-2} = 0 \\ p_t^{(2)} &= r_{t-\tau_1}^{(2)} \oplus \overline{r}_t^{(2)} = r_{t-\tau_1}^{(2)} \oplus q_t \oplus q_{t-2} = 0 \end{aligned} \tag{5.44}$$

[7] For the sake of simplicity, hard-decision decoding is used. However, the underlying principle applies to soft-decision decoding as well. Refer to [18] for details.

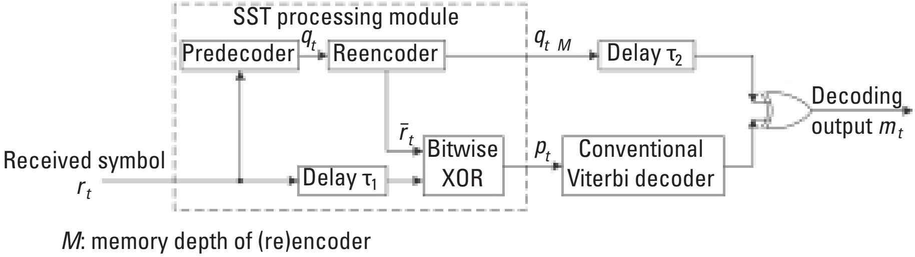


Figure 5.28 SST Viterbi decoder.

By XOR'ing $p_t^{(1)}$ and $p_t^{(2)}$, the following equation is obtained:

$$p_t^{(1)} \oplus p_t^{(2)} = r_{t-\tau_1}^{(1)} \oplus r_{t-\tau_1}^{(2)} \oplus q_{t-1} = 0 \tag{5.45}$$

or, equivalently:

$$q_{t-1} = r_{t-\tau_1}^{(1)} \oplus r_{t-\tau_1}^{(2)} \tag{5.46}$$

This equation is the logic of the predecoder.[8] *Setting $\tau_1 = 1$, the complete schematic of the SST Viterbi decoder for the (2,1,3) code is depicted in Figure 5.29.*

Next we prove that the SST decoder decodes correctly. From (5.41), (5.42), and (5.46), we have:

$$q_t = r_t^{(1)} \oplus r_t^{(2)} = m_{t-1} \oplus e_t^{(1)} \oplus e_t^{(2)} = m_{t-1} \oplus e_t' \tag{5.47}$$

where $e_t' = e_t^{(1)} \oplus e_t^{(2)}$. The output of the predecoder q_t is reencoded as:

$$\overline{r}_t^{(1)} = q_t \oplus q_{t-1} \oplus q_{t-2} = c_{t-1}^{(1)} \oplus x_e^{(1)} \text{ and } \overline{r}_t^{(2)} = q_t \oplus q_{t-2} = c_{t-1}^{(2)} \oplus x_e^{(2)} \tag{5.48}$$

where $x_e^{(1)} = e_t' \oplus e_{t-1}' \oplus e_{t-2}'$ and $x_e^{(2)} = e_t' \oplus e_{t-2}'$. The input to the conventional Viterbi decoder is found to be:

$$p_t^{(1)} = r_{t-1}^{(1)} \oplus \overline{r}_t^{(1)} = x_e^{(1)} \oplus e_{t-1}^{(1)} \text{ and } p_t^{(2)} = r_{t-1}^{(2)} \oplus \overline{r}_t^{(2)} = x_e^{(2)} \oplus e_{t-1}^{(2)} \tag{5.49}$$

[8] We have taken an ad hoc approach in designing the predecoder. In general the predecoder is the inverse of the encoder. Refer to [20] for more details.

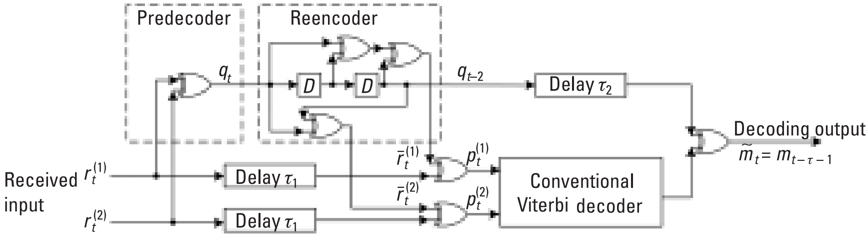


Figure 5.29 The (2,1,3) code SST Viterbi decoder.

By treating $\boldsymbol{x}_e = \left(x_e^{(1)} x_e^{(2)}\right)$ as data symbol, $\boldsymbol{p}_t = \left(p_t^{(1)} p_t^{(2)}\right)$ can be viewed as $\boldsymbol{x}_e$ corrupted by the channel error $\boldsymbol{e}_t = \left(e_{t-1}^{(1)}\ e_{t-1}^{(2)}\right)$. Then $\boldsymbol{p}_t$ is sent to the conventional Viterbi decoder for decoding. If it is decoded correctly, the conventional decoder outputs $e'_{t-\tau}$,[9] where τ is the latency of the conventional decoder. The final decoded output of the SST decoder is simply:

$$\tilde{m}_t = q_{t-2-\tau_2} \oplus e'_{t-\tau} = m_{t-\tau-1} \tag{5.50}$$

where $2 + \tau_2 = \tau$. The delay 2 in $q_{t-2-\tau_2}$ accounts for the encoder memory depth.

5.4 Good Convolutional Codes

Although the majority of block codes are obtained by algebraic manipulations, convolutional codes are only found through computer searches.[10] Given a coding rate and a code constraint length, the computer first constructs all possible codes. Only the codes with no catastrophic error propagation are considered further. Then the performance bounds for the remaining codes are computed and compared, and finally the codes with the best performance are selected as good convolutional codes.

Note that different decoding algorithms may pose different additional criteria for good codes. For example, a good code for Viterbi decoding should have a free distance d_{free} as large as possible, simply because the BER performance of Viterbi decoding decreases exponentially as d_{free} increases (refer to Section 5.2.1.4).

[9] Recall that x_e is the codeword of e'_t.

[10] At present there is no known method for analytically finding good convolutional codes.

5.4.1 Catastrophic Error Propagation

Catastrophic error propagation is a problem associated with some convolutional codes. Let us look at a simple example.

Example 5.17

> *Consider a (2,1,3) code with code generators of (110) and (101). Assume that the message sequence is an all-one sequence: $\vec{m} = 1,1,1,1,\cdots$. The code sequence would then be $\vec{c} = 1,1,0,1,0,0,\cdots$. Now suppose that the channel noise turns the three nonzero bits in the code sequence into zeros. Then the received sequence becomes an all-zero sequence $\vec{r} = 0,0,0,0,\cdots$. Because an all-zero received sequence could legitimately translate to an all-zero message sequence with no channel error, any decoder will inevitably take the all-zero sequence as its valid decode output, thus producing a catastrophic result.*

Catastrophic error propagation occurs only in nonsystematic codes; systematic codes [for example, the rate-1/2 code with the generators (100) and (111) is inherently error-propagation free] [8, 21–23].

5.4.2 Some Known Good Convolutional Codes

Convolutional codes are specified by their generators. For example, $g_1 = (111)$ and $g_2 = (101)$, or 7 and 5 in octal, specify the (2,1,3) code that has been used throughout this chapter as the example code.

The set of good convolutional codes listed in Tables 5.6 through 5.9 is the work of Odenwalder, Larsen, and Daut [24–26]. The codes are "good" in the sense that they have the largest free distance and therefore suitable to Viterbi decoding. Note that the first nonzero LSB of each octal generator in the tables, when converted into binary, corresponds to the lowest power in the generator polynomial. For instance, $(13)_8 \rightarrow (001011)_2 \rightarrow 1 + D^2 + D^3$.

5.5 Punctured Convolutional Codes

The convolutional codes that we have discussed so far are rate-1/*n* codes. Although the codes provide powerful error correcting capabilities, their coding rate is sometimes too low for bandwidth-constrained applications. One approach to raise the code rate is to increase the value of *k* in an (*n*, *k*, *v*) code to *k* > 1. The problem with this is that the resultant code will be a nonbinary code. The decoding complexity of nonbinary codes rises geometrically with an increase in *k*.

Table 5.6
Rate-1/2 Convolutional Codes

Constraint Length	Generator	Free Distance
3	(5,7)	5
4	(15,17)	6
5	(23,35)	7
6	(53,75)	8
7	(133,171)	10
8	(247,371)	10
9	(561,753)	12
10	(1167,1545)	12

An alternative solution proposed by Cain and Geist [27] is the so-called code puncturing, in which a rate-$(n - 1)/n$ code with $n > 3$ is obtained by systematically deleting some bits in a "base" rate-$1/n$ code. The complexity increase of this approach is marginal.

Example 5.18

The rate-1/2 convolutional encoder in Figure 5.2 produces two code bits, $c^{(1)}$ and $c^{(2)}$, for each incoming information bit. Eliminating one code bit every other codeword in the code sequence generates the following sequence:

Table 5.7
Rate-1/3 Convolutional Codes

Constraint Length	Generator	Free Distance
3	(5,7,7)	8
4	(13,15,17)	10
5	(25,33,37)	12
6	(47,53,75)	13
7	(133,145,175)	15
8	(225,331,367)	16
9	(557,663,711)	18
10	(1117,1365,1633)	20

Table 5.8
Rate-1/4 Convolutional Codes

Constraint Length	Generator	Free Distance
3	(5,7,7,7)	10
4	(13,15,15,17)	13
5	(25,27,33,37)	16
6	(53,67,71,75)	18
7	(133,135,147,163)	20
8	(235,275,313,357)	22
9	(463,535,733,745)	24
10	(1117,1365,1633,1653)	27

$$c_0^{(1)}c_0^{(2)},c_0^{(1)}\times,\cdots,c_t^{(1)}c_t^{(2)},c_{t+1}^{(1)}\times,\cdots$$

where × represents the deleted code bit. As a result, only three code bits are generated per two information bits. The code rate therefore becomes 2/3. A puncturing matrix P specifies how the output bits from the base encoder are deleted. A 1 in the matrix indicates that the output bit from the base encoder is transmitted, and a 0 indicates that the bit is deleted. The puncturing matrix for this example is:

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

Table 5.9
Rate-1/5 Convolutional Codes

Constraint Length	Generator	Free Distance
3	(5,5,7,7,7)	13
4	(13,15,15,17,17)	16
5	(25,27,33,35,37)	20
6	(57,65,71,73,75)	22
7	(131,135,135,147,175)	25
8	(233,257,271,323,357)	28

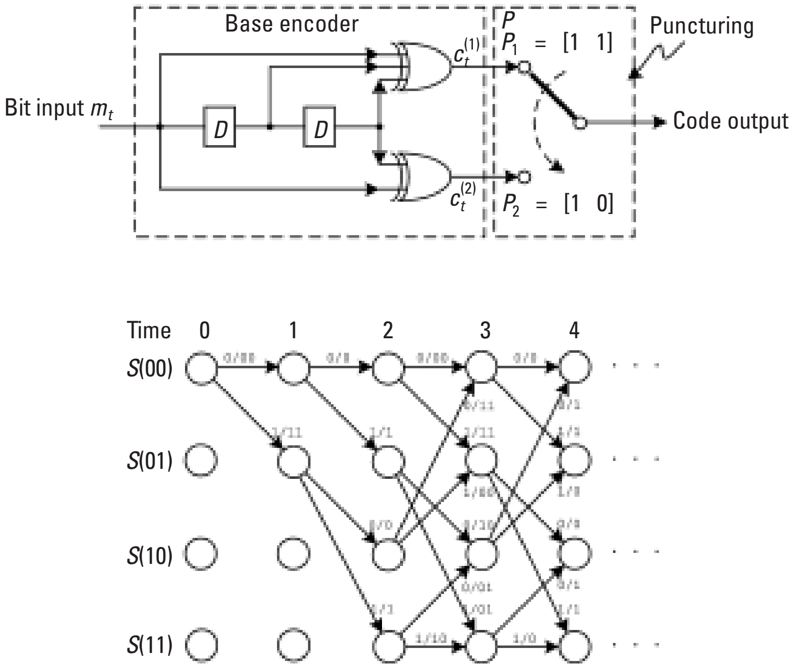


Figure 5.30 Punctured convolutional encoder and trellis.

Figure 5.30 depicts the punctured encoder and the associated trellis diagram.

The rate-2/3 code, if constructed in the traditional way, would have four branches leaving and entering a state, which greatly increases the decoding complexity.

Apart from the computational savings, punctured codes also make it possible to adaptively use codes of different rates in a communications system according to the channel condition with only one encoder and one decoder. (We will see how this is done in the next subsection). Therefore, in this sense, the punctured codes of the same base are collectively called *rate-compatible punctured convolutional* (RCPC) codes [28].

MATLAB Experiment 5.10

Two MATLAB functions, `punc`* and `depunc`*, are provided in the accompanying DVD for code puncturing and depuncturing.

```
>> p = [1 1; 1 0 ]; % puncture matrix
>> c = [1 1 0 1 0 1 0 0 1 0 1 1];    % original code sequence
>> c1 = punc(c,p)                    % puncture
c1 =
       1  1  0  0  1  0  1  0  1
>> r = [-1 -1 1 1 -1 1 -1 1 -1];     % received sequence
>> r1 = depunc(r,p)                  % depuncture
r1 =
       -1  -1  1  0  1  -1  1  0  -1  1  -1  0
```

5.5.1.1 Decoding of Punctured Codes

Viterbi decoding of punctured codes involves an additional procedure known as code depuncturing. Code depuncturing inserts nulls into the received sequence at the positions of the originally deleted bits. A null is a dummy symbol that is equidistant from all codewords. As we have seen, the number of bits per codeword changes periodically after code puncturing. (For instance, in the previous example the codeword contains one bit at time 1, 3, 5, ⋯ and two bits at 2, 4, 6, ⋯.) Without depuncturing, the number of bits per codeword must be tracked for each trellis branch in order for Viterbi decoding to perform properly. Code depuncturing makes up the deleted bits in the received sequence, so that decoding can be carried out just as if the code were the base code. This is particularly helpful in RCPC codes because we need only one decoder for all codes with different rates, as long as they are correctly depunctured. Figure 5.31 illustrates a decoder for punctured code.

Example 5.19

> *The code sequence in Example 5.2 $\vec{c} = 11, 01, 01, 00, 10, 11$ is punctured to $\vec{c} = 11, 0, 01, 0, 10, 1$ to increase the coding rate from 1/2 to 2/3. Let us assume that BPSK is employed. So, what is actually transmitted is $-1\ -1\ +1\ +1\ -1\ +1\ -1\ +1\ -1$. The corresponding received sequence is $-1\ -1\ +1\ -1\ -1\ +1\ -1\ +1\ -1$, which has one error in the fourth posi-*

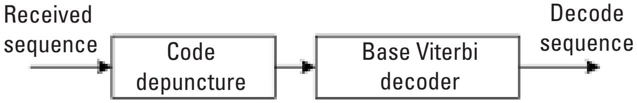


Figure 5.31 Decoder for punctured code.

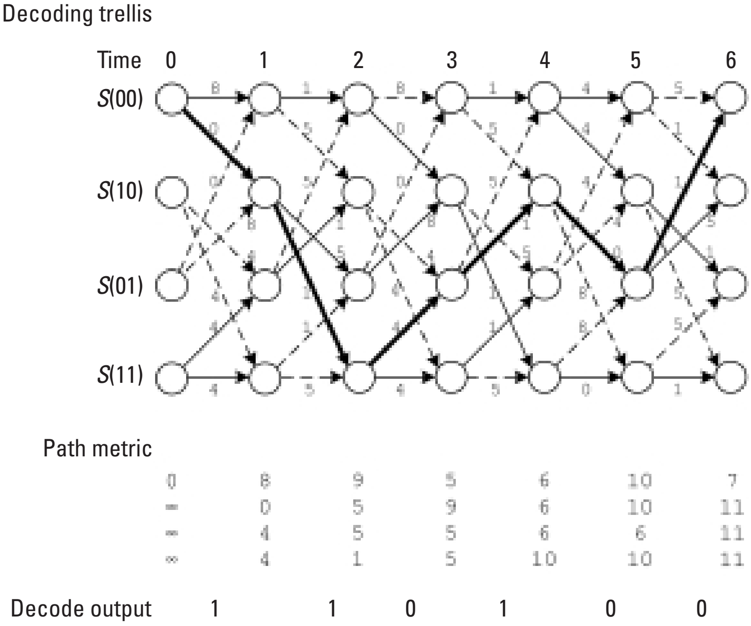


Figure 5.32 Punctured code decoding process.

tion. To decode the sequence, we first depuncture it by inserting zeros (nulls) into it. The depunctured sequence then is –1 –1 +1 0 –1 –1 +1 0 –1 +1 –1 0. The Viterbi decoder takes the depunctured sequence and performs decoding on it as usual. Figure 5.32 shows the decoding process.

Table 5.10
Rate-2/3 Punctured Code

Constraint Length	Generator	Free Distance
3	(7,5),7	3
4	(15,13),15	4
5	(31,33),31	5
6	(73,41),73	6
7	(163,135),163	6
8	(337,251),337	8
9	(661,473),661	8

Table 5.11
Rate-3/4 Punctured Code

Constraint Length	Generator	Free Distance
3	(5,7),5,7	3
4	(15,17),15,17	4
5	(35,37),37,37	4
6	(61,53),53,53	5
7	(135,163),163,163	6
8	(205,307),307,307	6
9	(515,737),737,737	6

MATLAB Experiment 5.11

Use MATLAB to decode Example 5.19:

```
>> trellis = poly2trellis(3,[7 5]);
>> r = [-1 -1 1 1 -1 1 -1 1 -1];        % received sequence
>> rd = [-1 -1 1 0 -1 -1 1 0 -1 1 -1 0]; % depunctured seq.
>> decoded = vitdec(rd,trellis,3,’term’,’unquant’)  % decode
decoded =
       1  1  0  1  0  0
```

The above-received sequence is successfully decoded without difficulty. In general, however, the error correcting capabilities of punctured codes are

Table 5.12
Rate-4/5 Punctured Code

Constraint Length	Generator	Free Distance
4	(17,11),11,11,13	3
5	(37,35),25,37,23	4
6	(61,53),47,47,53	4
7	(151,123),153,151,123	5
8	(337,251)237,237,235	5
9	(765,463),765,765,473	5

Table 5.13
Rate-5/6 Punctured Code

Constraint Length	Generator	Free Distance
4	(17,15),13,15,15,13	3
5	(37,23),23,23,25,25	4
6	(75,53),75,75,75,75	4
7	(145,127)133,127,145,133	4
8	(251,237),235,235,251,251	5
9	(765,473),765,473,463,457	5

somewhat weakened. This is because as a coding rate goes higher, the redundancy in the codes is reduced and, hence, so is the code's error correcting capabilities. Consequently, the decoding depth must be increased in order to maintain performance.

5.5.1.2 Popular Punctured Codes

The rate-2/3 punctured code discussed earlier is simply a punctured version of the known good rate-1/2 code, code (7,5). However a good rate-1/*n* code does not necessarily yield a good punctured code. Similar to the rate-1/*n* codes, good punctured codes are also discovered by computer search. The known good punctured codes are tabulated in Tables 5.10 through 5.13, together with their respective free distances [29, 30].

A punctured code is specified by the set of generators of its base code followed by the generators used at successive times. For example, the code in Example 5.18 is specified by (7,5),7, which means that the first message bit is encoded using generators (111) and (101), and the second message bit is encoded using (111) only. Using the (5,7),5,7 code as another example, the first message bit is encoded using (101) and (111), the second is encoded using (101), and the third using (111).

Note This chapter has covered almost every important aspect of convolutional codes except one: recursive convolutional codes. This particular type of convolutional code is of special significance because it is the basis of another powerful error correcting code—the newly developed turbo codes. So it will be discussed in the next chapter where we present turbo codes.

Problems

5.1 Use a MATLAB simulation to find the relationship between the BER and decoding depth. Validate the "five times constraint length" rule suggested in [7].

5.2 Errors at the output of a Viterbi decoder occur in bursts. Explain why and determine the minimum length of the error bursts.

5.3 For the sake of implementation simplicity, some Viterbi decoder designs simply replace the squared Euclid's distance $|r^{(i)} - c^{(i)}|^2$ in (5.14) with a linear distance $|r^{(i)} - c^{(i)}|$ at the expense of some performance degradation. Taking the (2,1,3) code as an example, evaluate the degradation through simulation.

5.4 Use simulation to verify that errors at the output of a Viterbi decoder tend to occur in bursts.

5.5 For the (2,1,3) convolutional code, assume a received sequence $\vec{r}$, and decode on $\vec{r}$ using the Viterbi algorithm, the stack algorithm, and the Fano algorithm.

References

[1] Elias, P., "Coding for Noisy Channels," *IRE Conv. Rec.*, Part 4, 1955, pp. 37–47.

[2] Ma, H. H., and J. K. Wolf, "On Tail Biting Convolutional Codes," *IEEE Trans. Commun.*, Vol. COM-34, No. 2, February 1986, pp. 104–111.

[3] Wozencraft, J. M., "Sequential Decoding for Reliable Communications," *IRE Nat. Conv. Rec.*, Vol. 5, Part 2, 1957, pp. 11–25.

[4] Massey, J. L., *Threshold Decoding*, Cambridge, MA: The MIT Press, 1963.

[5] Viterbi, A. J., "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm," *IEEE Trans. Inform. Theory*, Vol. IT-13, April 1967, pp. 260–269.

[6] Forney, G. D., Jr., "Convolutional Code II: Maximum Likelihood Decoding," *Inform. Control*, Vol. 25, July 1974, pp. 222–266.

[7] Forney, G. D., Jr., "The Viterbi Algorithm," *Proc. IEEE*, Vol. 61, No. 3, March 1973, pp. 268–278.

[8] Lin, S., and D. J. Costello, *Error Control Coding—Fundamentals and Applications*, 2nd ed., Englewood Cliffs, NJ: Prentice-Hall, 2004.

[9] Viterbi, A. J., "Convolutional Codes and Their Performance in Communication Systems," *IEEE Trans. Inform. Theory*, Vol. IT-17, October 1971, pp. 751–772.

[10] Viterbi, A. J., and J. K. Omura, *Principles of Digital Communications and Coding*, New York: McGraw-Hill, 1979.

[11] Jelenik, F., "A Fast Sequential Decoding Algorithm Using a Stack," *IBM J. Res. Develop.*, Vol. 13, November 1969, pp. 675–685.

[12] Fano, R. M., "A Heuristic Discussion of Probabilistic Decoding," *IEEE Trans. Inform. Theory*, Vol. IT-9, April 1963, pp. 64–74.

[13] Onyszchuk, I. M., et al., "Quantization Loss in Convolutional Decoding," *IEEE Trans. Commun.*, Vol. 41, No. 2, February 1993, pp. 261–265.

[14] Hekstra, A. P., "An Alternative to Metric Rescaling in Viterbi Decoders," *IEEE Trans. Commun.*, Vol. 37, No. 11, November 1989, pp. 1220–1222.

[15] Rader, C. M., "Memory Management in a Viterbi Decoder," *IEEE Trans. Commun.*, Vol. 29, No. 9, September 1981, pp. 1399–1401.

[16] Lee, I., and Sonntag, J. L., "A New Architecture for the Fast Viterbi Algorithm," *IEEE Trans. Commun.*, Vol. 51, No. 10, October 2003, pp. 1624–1628.

[17] Ishitani, T., et al., "A Scarce-State-Transition Viterbi-Decoder VLSI for Bit Error Correction," *IEEE J. Solid-State Circuits*, Vol. SC-22, No. 4, August 1987, pp. 575–582.

[18] Seki, K., et al., "Very Low Power Consumption Viterbi Decoder LSIC Employing the SST (Scarce State Transition) Scheme for Multimedia Mobile Communications," *IEE Electronics Lett.*, Vol. 30, No. 8, April 1994, pp. 637–639.

[19] Kubota, S., S. Kato, and T. Ishitani, "Novel Viterbi Decoder VLSI Implementation and Its Performance," *IEEE Trans. Commun.*, Vol. 41, No. 8, August 1993, pp. 1170–1178.

[20] Massey, J. L., and M. K. Sain, "Inverse of Linear Sequential Circuits," *IEEE Trans. Comput.*, Vol. C-17, April 1968, pp. 330–337.

[21] Clark, G. C., and J. B. Cain, *Error-Correction Coding for Digital Communications*, New York: Plenum Press, 1981.

[22] Wicker, S. B., *Error Control Systems for Digital Communication and Storage*, Englewood Cliffs, NJ: Prentice-Hall, 1994.

[23] Lee, L. H. C., *Convolutional Coding: Fundamentals and Applications*, Norwood, MA: Artech House, 1997.

[24] Odenwalder, J. P., "Optimal Decoding of Convolutional Codes," Ph.D. Dissertation, University of California at Los Angeles, 1970.

[25] Larsen, K. J., "Short Convolutional Codes with Maximal Free Distances for Distances for Rates 1/2, 1/3 and 1/4," *IEEE Trans. Inform. Theory*. Vol. IT-19, 1973, pp. 371–372.

[26] Daut, D., J. Modestino, and L. Wismer, "New Short Constraint Length Convolutional Code Construction for Selected Rational Rates," *IEEE Trans. Inform. Theory*, Vol. IT-28, No. 5, September 1982, pp. 793–799.

[27] Cain, J., Jr., and J. Geist, "Punctured Convolutional Codes of Rate $(n - 1)/n$ and Simplified Maximum Likelihood Decoding," *IEEE Trans. Inform. Theory*, Vol. IT-25, No. 1, January 1979, pp. 97–100.

[28] Hagenauer, J., "Rate Compatible Punctured Convolutional Codes and Their Applications," *IEEE Trans. Commun.*, Vol. 36, No. 4, April 1988, pp. 389–400.

[29] Lee, P., "Construction of Rate $(n - 1)/n$ Punctured Convolutional Codes with Minimum Required SNR Criterion," *IEEE Trans. Commun.*, Vol. 36, No. 10, October 1988, pp. 1171–1174.

[30] Wells, R. B., *Applied Information Theory and Coding for Engineers*, Upper Saddle River, NJ: Prentice-Hall, 1998.

Selected Bibliography

Lou, H. L., "Implementing the Viterbi Algorithm," *IEEE Sig. Proc. Mag.*, September 1995, pp. 42–52.

Wozencraft, J. M., and B. Reiffen, *Sequential Decoding*, Cambridge, MA: The MIT Press, 1961.

6

Modern Codes

During the past half-century, coding theorists have been busy searching for better error correcting codes. A breakthrough was made in 1993 when Berrou, Glavieux, and Thitimajshima published their research on what they called turbo codes [1], leading coding research into a new era. Turbo codes leapt past the classical block codes and convolutional codes by their near Shannon capacity performance. Following this exciting finding, 3 years later, MacKay rediscovered the low-density parity code (LDPC) that had been invented by Gallager thirtysomething years before [2–5]. LDPC codes push the limit even further [6].

This chapter surveys the modern advances in the area of error correcting codes.

6.1 Turbo Codes

6.1.1 Code Concatenation

6.1.1.1 Serial Concatenated Codes

Although, as Shannon indicated, we can always improve coding performance by increasing code length, this approach does not work in practice because decoding complexity rises exponentially with code length. In 1966 Forney proposed a simple and yet elegant strategy called code concatenation [7]. The basic idea is to cascade two or more relatively simple codes in a serial manner

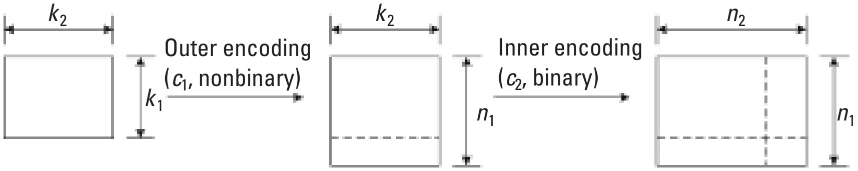


Figure 6.1 Concatenating codes in a serial manner.

such that the resulting composite code is equivalent to a much longer code in performance but much lower in decoding complexity.

Illustrated in Figure 6.1, such a strategy arranges a message block of k_1k_2 bits into a $k_1 \times k_2$ two-dimensional array. The k_1 message symbols (of k_2 bits each) are first encoded into a nonbinary code $C_1(n_1, k_1)$ (the outer code). The codeword is stored in a buffer. A second binary code $C_2(n_2, k_2)$ (the inner code) reads in the k_2-bit-long words from the buffer, one by one, to produce n_1 codewords of code C_2 (each of which is n_2 bits long). The result is a concatenated (n_1n_2, k_1k_2) code C. If the minimum distances of the two codes are d_1 and d_2, respectively, the minimum distance of C is at least d_1d_2 [7]. Due to its simplicity and effectiveness, the technique has been widely adopted. The most popular concatenated code in practical applications is perhaps the code that pairs RS code (as the outer code) with convolutional code (as the inner code). Whereas Viterbi decoding output tends to produce errors in bursts (see Problem 5.2 in Chapter 5), RS code is able to correct burst errors effectively. The combination thus makes perfect sense.

A typical serial concatenated coding system is illustrated in Figure 6.2. The buffers in the encoder and the decoder are called the interleaver and

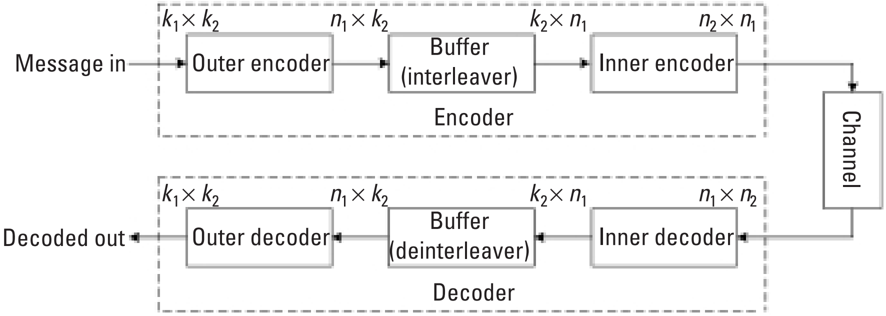


Figure 6.2 Serial concatenated coding system.

deinterleaver, respectively. Their role in code concatenation goes beyond simple data buffering and will be explained in the next subsection.

Example 6.1

> *The simple two-dimensional code introduced in Problem 3.1 of Chapter 3 is in fact a concatenated code. The values of k_1 and k_2 are 5 and 4, respectively. The outer code is a $(k_1 + 1, k_1)$ block code with codewords over GF(2^4), and the inner code is a $(k_2 + 1, k_2)$ binary block code. An example encoding process is shown in Figure 6.3. Although neither the outer code nor the inner code alone can correct any errors, the concatenated code is able to correct up to one error, as shown previously.*

Figure 6.4 presents the performance of a (255,223) RS code + rate-1/2 convolutional code with the generator (155,117). The BER of the individual code is also plotted for comparison purposes.

MATLAB Experiment 6.1

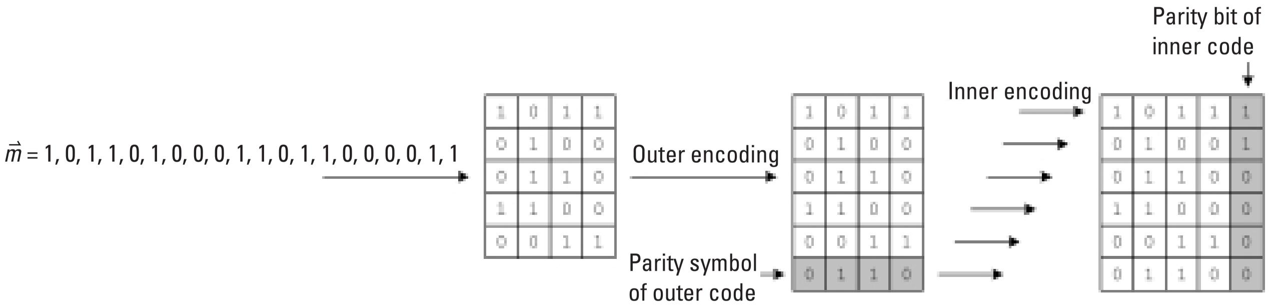
The program `concat.m*` on the companion DVD simulates the performance of concatenation of the (2,1,3) convolutional code and the (7,4) RS code. Compare the error probability with that of the individual codes, which were simulated in previous chapters.

6.1.1.2 Role of Interleaving

The interleaver/deinterleaver between the outer and the inner code serves two purposes. First, the interleaver provides buffering between the outer code and the inner code. Second, more importantly, interleaving spreads out burst errors so that error bursts are randomized and become more correctable. As shown in Figure 6.5, data sequence $\vec{d} = d_1, d_2, \cdots, d_{24}$ is first read into the interleaver row by row, and then read out column by column as:

$$\begin{array}{l} d_1,\ d_5,\ d_9,\ d_{13},\ d_{17},\ d_{21},\ d_2,\ d_6,\ d_{10},\ d_{14},\ d_{18},\ d_{22},\ \hookleftarrow \\ \hookrightarrow d_3,\ d_7,\ d_{11},\ d_{15},\ d_{19},\ d_{23},\ d_4,\ d_8,\ d_{12},\ d_{16},\ d_{20},\ d_{24} \end{array}$$

Assume that an error burst of length 4 occurs at positions 3, 4, 5, and 6, and the data affected are d_9, d_{13}, d_{17}, and d_{21}. The deinterleaver puts the permuted sequence back into the original order. These errors are spread out and are no longer contiguous.


$\vec{m}$ = 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1
Outer encoding
Parity symbol of outer code
Inner encoding
Parity bit of inner code

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

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

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

Figure 6.3 Encoding process of concatenated code.

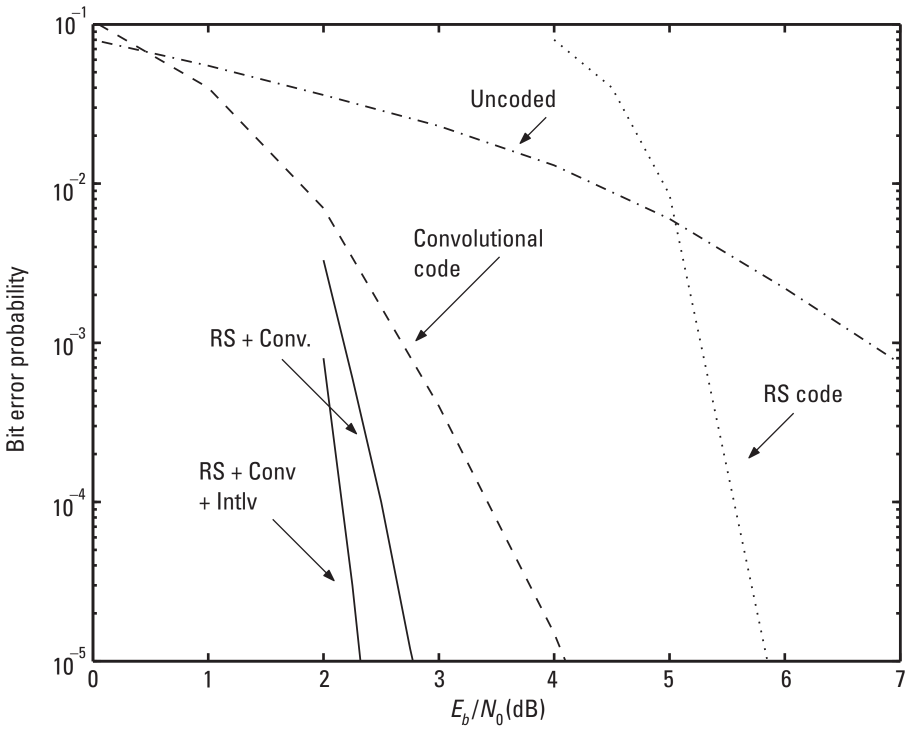


Figure 6.4 Performance of concatenated code. (*After:* [8].)

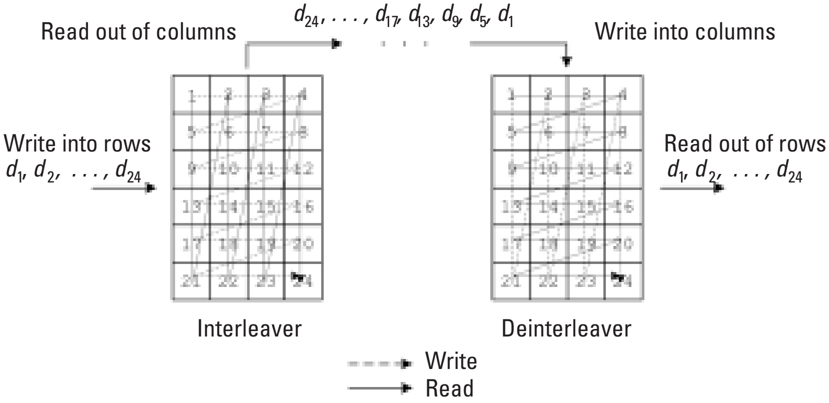


Figure 6.5 Operation of block interleaver.

This type interleaver is called a *block interleaver*. It is widely used in practice. In general, a block interleaver deals with a two-dimensional array that can be read and written in different directions. In the preceding example, the data sequence is first written into the interleaver in the x direction and then read out in the y direction.

Another commonly used interleaver, the *convolutional interleaver*, consists of a bank of delay lines with increasing lengths. An input commutator successively connects to the delay lines at each cycle to load the data into the delay lines. An output commutator moves in sync with the input commutator to read the data out of the delay lines. The deinterleaver does just the opposite with the delay lines of decreasing lengths. Figure 6.6 shows an example convolutional interleaver/deinterleaver comprised of three delay lines. The operation of the devices is also detailed in the figure. The convolutional interleaver/deinterleaver can be viewed as a block interleaver split diagonally in half.

Interleaving plays an important role (sometimes a key role) in many applications, including turbo codes (which we will be introducing shortly) and channel fading mitigation. We will have more to say on this later.

MATLAB Experiment 6.2

Two MATLAB functions, `blkintlv*` and `convintlv*`, one for block interleaving and the other for convolutional interleaving, are included on this book's DVD. Readers are encouraged to try them out.

6.1.2 Concatenating Codes in Parallel: Turbo Code

6.1.2.1 Recursive Systematic Convolutional Codes

The basic building block in a turbo code is the so-called *recursive systematic convolutional* (RSC) code. An RSC code is an alternative realization of the nonsystematic rate-$1/n$ convolutional code. The conventional convolutional code is constructed in a feed-forward fashion; that is, the encoder consists of no feedback. In contrast, its RSC equivalence involves feedback in the encoding process. Notice that by using the term *equivalence* we have implied that the recursive code has the same distance property as its nonrecursive counterpart.

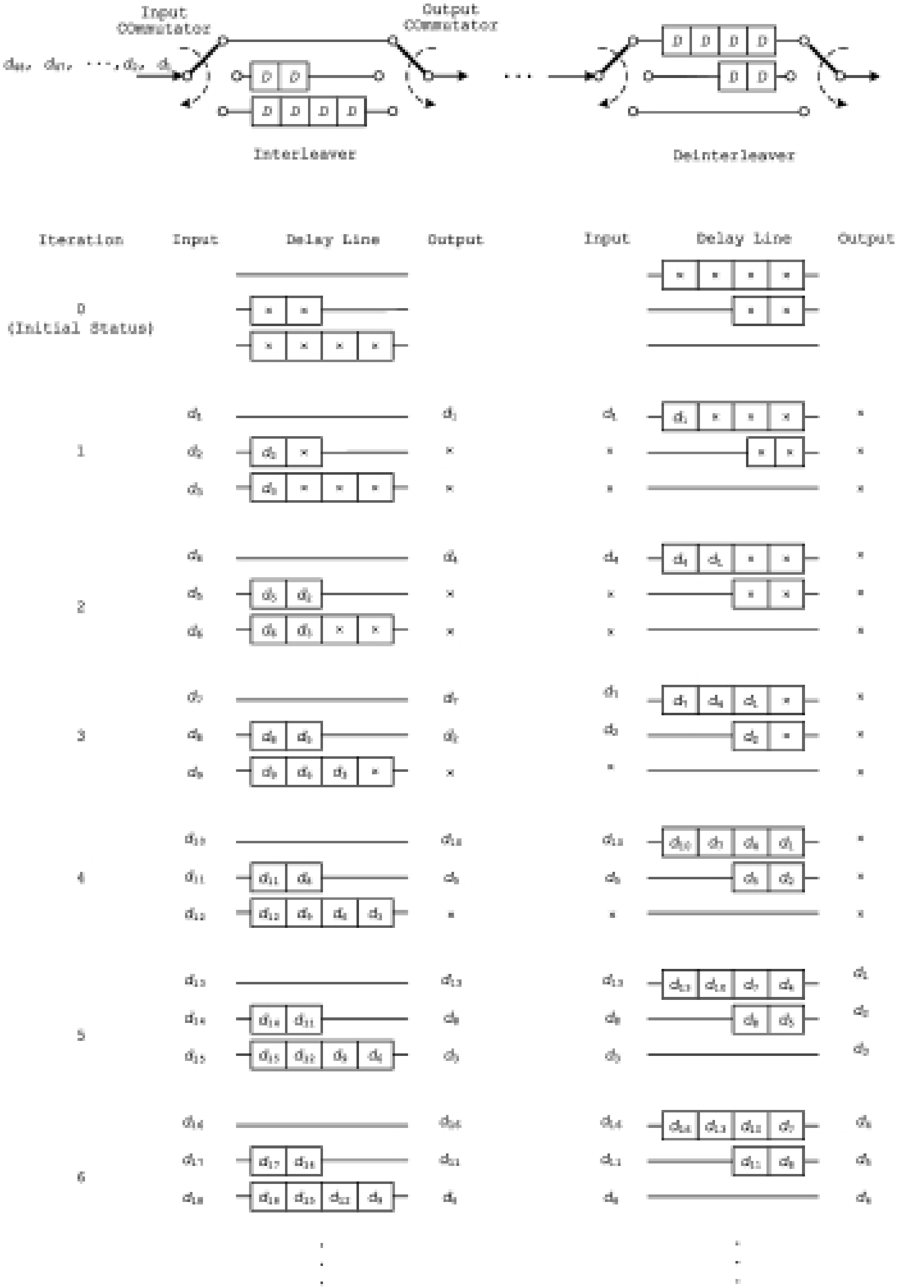
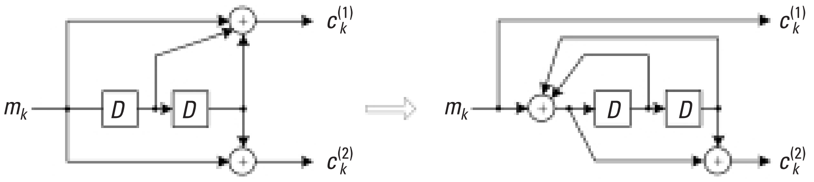


Figure 6.6 Operation of convolutional interleaver/deinterleaver.



Convolutional encoder
(a)

RSC encoder
(b)

Figure 6.7 (a) Convolutional encoder and (b) RSC encoder.

To construct an RSC code, we need to transform the nonsystematic feed-forward code generator into a systematic feedback generator. For a convolutional code with generators $g_1(D)$, $g_2(D)$, …, $g_n(D)$, the generators of the equivalent RSC code are:

$$1, \frac{g_2(D)}{g_1(D)}, \frac{g_2(D)}{g_1(D)}, \ldots, \frac{g_n(D)}{g_1(D)} \tag{6.1}$$

where the denominator represents the feedback involved.

Example 6.2

Figure 6.7 shows the equivalent RSC code of the nonsystematic feed-forward (2,1,3) convolutional encoder introduced in Chapter 5. The original generators are $g_1(D) = 1 + D + D^2$ and $g_2(D) = 1 + D^2$. The generators of the equivalent RSC are $g_1(D) = 1$ and $g_2(D) = (1 + D^2)/(1 + D + D^2)$.

From the circuit schematic in Figure 6.7(b), we can draw the state transition diagram and the trellis diagram of the RSC code as in Figure 6.8. As can be seen, the diagrams are very similar to those for the conventional (2,1,3) code.

For convolutional codes, the code generator is often expressed as an octal number. Similarly, for the example RSC code, we use 1 and 5/7 to represent its generators $g_1(D) = 1$ and $g_2(D) = (1 + D^2)/(1 + D + D^2)$.

6.1.2.2 Turbo Codes

Turbo codes are formed by concatenating *in parallel* two RSC codes separated by an interleaver π.[1] Figure 6.9 is the block diagram of a typical turbo

[1] We may use more than two RSC encoders, but the underlying principle remains the same.

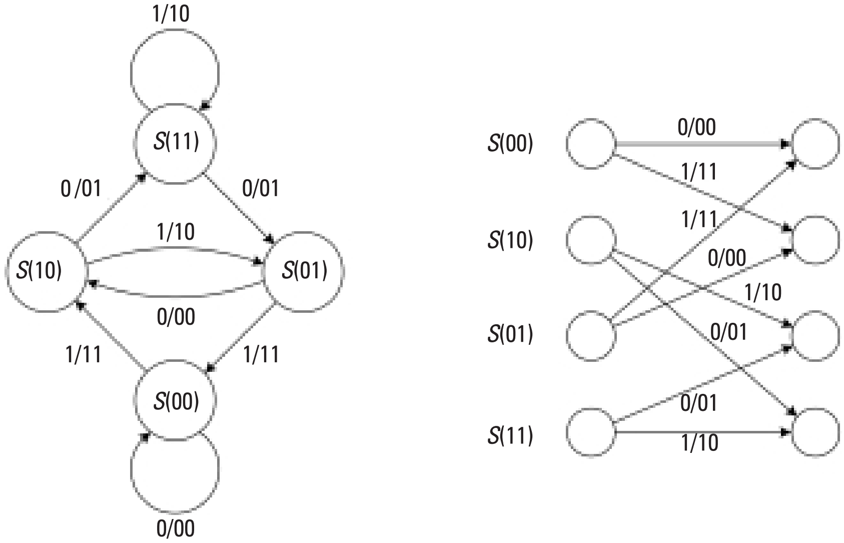


Figure 6.8 State transition diagram and trellis for a (1,5/7) RSC code.

encoder. Apparently, the turbo code is a systematic code. Its coding rate is 1/3; that is, for every input bit, the encoder produces three code bits. One is the message bit itself (we call it the systematic bit), and the other two are the parity bits generated by the two RSC encoders. At the end of encoding, flushing bits are added to force the first RSC encoder to the all-zero state. (RSC II may or may not be terminated to zero.) The two RSC encoders are often referred to as *component encoders.*

The code may also be punctured to obtain a higher coding rate, as depicted in Figure 6.10. Puncturing operates only on the parity sequences; the systematic bits are not touched. The punctured code in the figure raises the rate to 1/2.

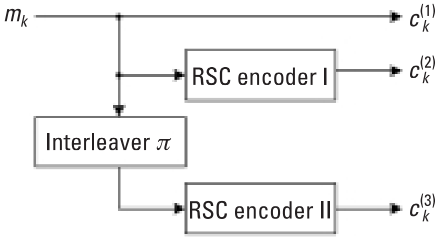


Figure 6.9 Turbo encoder.

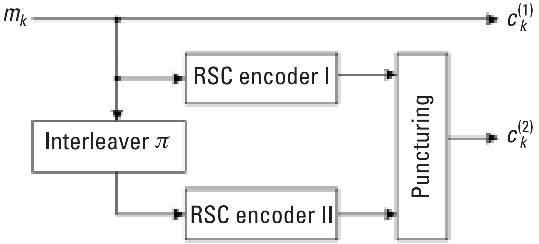


Figure 6.10 Turbo encoder with puncturing.

Example 6.3

Figure 6.11 shows a turbo encoder that uses the preceding example RSC as the component encoder. The interleaver changes the bit positions 1, 2, . . . , 8 to 8, 2, 7, 4, 3, 6, 5, 1. The message bit sequence to encode is $\vec{m}$ = 1, 0, 1, 1, 0, 1, 0, 0 with the leftmost bit entering the encoder first. All registers are reset to zero initially. According to the state transition diagram in the last example, the output of the first RSC encoder is 1, 1, 0, 0, 1, 1, 0, 0. The input sequence to the second encoder is permutated by the interleaver as $\vec{m}'$ = 0, 0, 0, 1, 1, 1, 0, 1. Therefore, the output of the second encoder is 0, 0, 0, 1, 0, 1, 0, 1. The final encoded output sequence then is (110),(010), (100),(101),(010),(111),(000),(001).

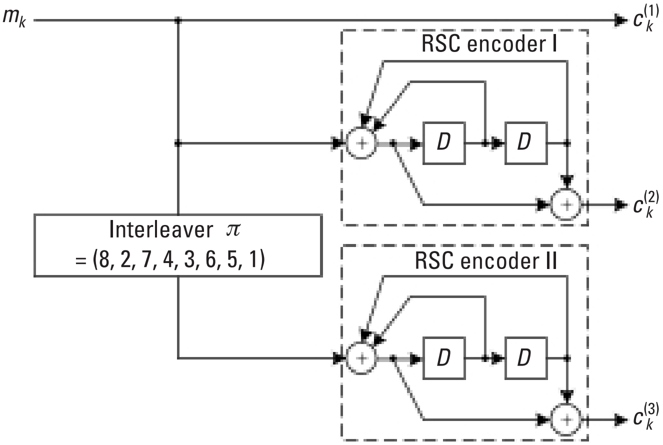


Figure 6.11 Example of a turbo code encoder.

MATLAB Experiment 6.3

We now use the MATLAB routine `tbcenc*` on this book's companion DVD to encode the preceding message sequence into turbo code.

```
>> m = [1,0,1,1,0,1,0,0];          % message sequence
>> intlv = [8 2 7 4 3 6 5 1];      % interleaver permutation
>> c = tbcenc(m,intlv)             % encoding
c =
       1  1  0
       0  1  0
       1  0  0
       1  0  1
       0  1  0
       1  1  1
       0  0  0
       0  0  1
```

6.1.2.3 Notes on Performance of Turbo Codes

At the beginning of the chapter we stated that turbo codes are very powerful error correcting codes. Now we show some typical BER curves in Figure 6.12 and make the following three points:

1. We indeed see that the performance of the turbo code is very impressive. The original turbo code reported in [1] achieved a 10^{-5} BER at only 0.7 dB away from the Shannon limit in an AWGN channel (whereas the convolutional code leaves a 3-dB gap). This superior performance largely benefits from the random-like property of the code due to interleaving. Therefore, interleaving is crucial in turbo codes.
2. An interesting phenomenon is that when the SNR increases to some point, the decrease in the BER suddenly slows down, forming some sort of error floor. This is an inherent drawback of turbo codes, due to the presence of low-weight codewords in the code.
3. The error performance of turbo codes improves as the code block length increases (code length 1,000 versus code length 65,536 in Figure 6.12). So, in order for turbo codes to achieve good performance, the code length is normally chosen to be on the order of several thousand bits. As a result, an interleaver of the same length is needed. This is another major disadvantage of turbo codes, since such a large code length results in significant processing delays.

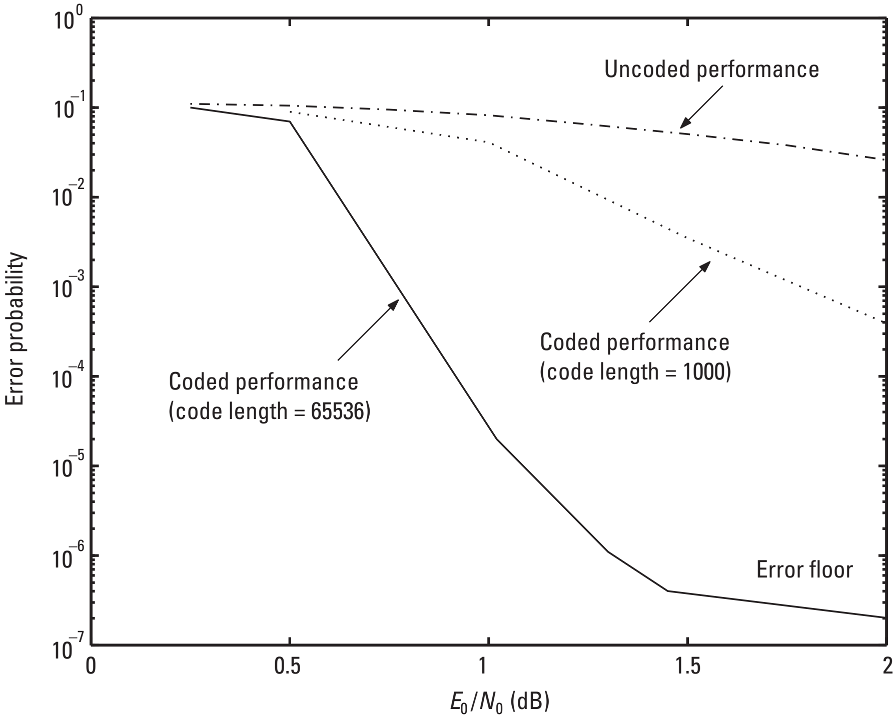


Figure 6.12 Typical performance of turbo code.

MATLAB Experiment 6.4

Run the m-file `tbcsim.m*` to simulate the BER of our turbo code in the example. Readers will observe behavior similar to that of Figure 6.12.

Comment: You may not know what each subroutine does at this point. They will become clear when you finish this section.

6.1.2.4 Interleaver Design for Turbo Codes

Whereas in a conventional serial concatenated code an interleaver is used merely to spread out burst errors, the interleaver in a turbo code plays a far more important role: It permutes the data sequence sent to the second component encoder so as to generate a second parity that is independent from the parity generated by the first component encoder. As we will see, by doing so, the turbo decoder will have two independent (or almost independent, to be more precise) sets of redundancies to exploit in decoding, and the decoding performance is thereby significantly improved.

Traditional block interleavers and convolutional interleavers do not work well in turbo coding. It is important for the interleaver in turbo coding to have a random property and make the interleaved sequence as independent as possible from the original sequence. In the original turbo code [1], a pure random interleaver was adopted. The permutation is based on randomly generated numbers. However, in practice, only a pseudo-random interleaver is feasible, which permutes data sequence according to pseudo-random numbers. We use the following notation:

$$\pi = \pi_1, \pi_2, \cdots, \pi_n$$

to describe the permutation of an interleaver of depth n, where the ith element of the interleaved sequence is the π_ith element of the input sequence. Figure 6.13 illustrates such an interleaver of depth 8. The permutation function is $\pi = 8, 2, 7, 4, 3, 6, 5, 1$.

MATLAB Experiment 6.5

A simple routine, called `rndintlv.m*`, for implementing the random interleaver is included on this book's DVD.

The issue with the random integer approach just discussed is that the resulting interleaver could be good or bad; we have no control over the selection of the random numbers. A remedy is the so-called *S*-random interleaver

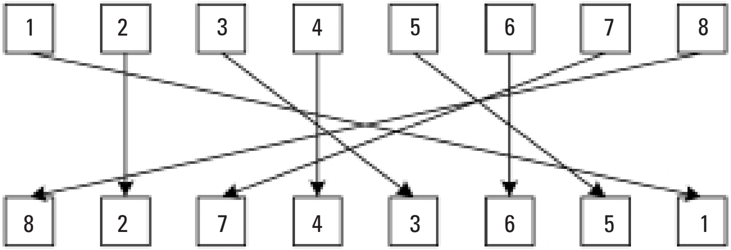


Figure 6.13 Pseudo-random interleaver.

[9]. An *S*-random interleaver attempts to permute the elements to the locations that are at least *S* positions apart. (In other words, it guarantees the contiguous bits in the original sequence to be at least *S* positions away from each other after interleaving.) The interleaver is constructed as follows:

1. Randomly select the first element of the permutation $\pi_1 \in \{1, 2, \ldots, n\}$.
2. For each subsequent element of the permutation, π_i, $i = 2, 3, \ldots, n$, randomly select a number from the set $\{1, 2, \ldots, n\}$ and compare it to *S* previous selections $\pi_1, \pi_2, \ldots, \pi_S$.
3. If the absolute values of the distances between the current selection and all *S* previous selections are larger than *S* (the *S*-random criterion), the selection is accepted. Otherwise, reselect another number until the *S*-random criterion is met.
4. The process stops when every element of the permutation satisfies the *S*-random criterion.

A possible *S*-random interleaver of depth 8 is $\pi = 3, 7, 5, 2, 8, 4, 1, 6$. Notice that every element in the permutation of this design has a distance ≥2 from its two immediate predecessors, whereas the minimum distance in the pseudo-random interleaver is 1.

The *S*-random interleaver offers significant performance improvement over the random interleaver. Because of this, many variations on the *S*-random algorithm have been developed [10, 11].

Another type of interleaver, called an *algebraic interleaver*, admits analytical designs and simple implementations. The simplest such interleaver is the row-column block interleaver introduced earlier. A modification to the block interleaver is the *helical interleaver*, which writes data in row by row and reads data out diagonally [12]. To be more specific, for a two-dimensional array of k_1 rows and k_2 columns, the index for helically reading the bits out is given by:

$$j = (i-1)\cdot(k_2+1)\,\mathrm{mod}(k_1k_2)+1 \qquad (i = 1,2,3,\cdots,k_1k_2) \tag{6.2}$$

For the 6 × 4 array in Figure 6.5, the readout order is 1, 6, 11, 16, 21, 2, 7, 12, 17, 22, 3, 8, 13, 18, 23, 4, 9, 14, 19, 24, 5, 10, 15, 20. The helical interleaver has been reported [12] to outperform the pseudo-random interleaver.

An interleaver can be designed that allows simultaneous flushing of both encoders with one single tail (termed the *simile interleaver*) [13]. Consider an input sequence that can be divided into $M + 1$ subsequences (where M is the number of registers in an RSC encoder). It has been found that the final state of the RSC encoder is the modulo-2 sum of some subsequences. Take as an example the RSC encoder presented earlier. Suppose that an N-bit-long input sequence is $\vec{d} = d_0, d_1, d_2, \ldots$, which can be divided into three subsequences (because there are two registers in the RSC encoder, i.e., $M = 2$) as follows:

$$\vec{d}_{\mathrm{I}} = \{d_k \mid k \bmod(M+1) = 0\} = d_0, d_3, d_6, d_9, d_{12}, \cdots$$

$$\vec{d}_{\mathrm{II}} = \{d_k \mid k \bmod(M+1) = 1\} = d_1, d_4, d_7, d_{10}, d_{13}, \cdots$$

$$\vec{d}_{\mathrm{III}} = \{d_k \mid k \bmod(M+1) = 2\} = d_2, d_5, d_8, d_{11}, d_{14}, \cdots$$

Depending on the value of N mod $(M + 1)$, the final state of the encoder can only be one of the three possibilities shown in Table 6.1, and the order of the bits inside each subsequence does not matter to the final state of the encoder. This implies that we can drive both encoders with one tail and the encoders will end with the same state if the permutation is confined within the subsequences (i.e., the subsequences are interleaved separately).

Most recently, Sun and Takeshita [14] devised an algebraic interleaver called the *quadratic permutation polynomial* (QPP) interleaver. The interleaver provides a number of advantages including excellent performance (it exceeds that of the S-random interleaver), suitability for parallel processing, and low power consumption. For a data sequence of length N, a QPP interleaver of the same size is defined by the following polynomial:

$$\pi_i = (f_1 \cdot i + f_2 \cdot i^2) \bmod N$$

where i is the bit position of the interleaved sequence, π_i is the corresponding bit index before interleaving, and coefficients f_1 and f_2 defining the permutation are conditioned based on the following:

Table 6.1
Possible States of RSC Encoder with Simile Interleaver

***N* mod (*M* + 1)**	**Final State of RSC**	
	RSC I	**RSC II**
0	$\vec{d}_{\mathrm{II}} \oplus \vec{d}_{\mathrm{III}}$	$\vec{d}_{\mathrm{I}} \oplus \vec{d}_{\mathrm{II}}$
1	$\vec{d}_{\mathrm{I}} \oplus \vec{d}_{\mathrm{III}}$	$\vec{d}_{\mathrm{II}} \oplus \vec{d}_{\mathrm{III}}$
2	$\vec{d}_{\mathrm{I}} \oplus \vec{d}_{\mathrm{II}}$	$\vec{d}_{\mathrm{I}} \oplus \vec{d}_{\mathrm{III}}$

1. Coefficient f_1 is relatively prime to data block size N.
2. All prime factors of N are also factors of f_2.

Note that the deinterleaver may not be QPP (i.e., of higher polynomial degree). A detailed explanation of the interleaver is beyond the scope of the book. The readers are referred to [14] for further information.

Last, we have a few words to say about a special interleaver, the *uniform interleaver*. By "special" we mean that the interleaver is an abstract model rather than a real design. When evaluating the performance of turbo codes, an interleaver is preferred that averages all possible interleavers, so that the evaluation outcome will be the average performance rather than a biased result toward a particular interleaver. The interleaver is designed to map a given input word of weight w into all $\binom{N}{w}$ possible permutations with equal probability $1\Big/\binom{N}{w}$ (uniformly distributed).

6.1.3 Iterative Decoding of Turbo Codes

For the sake of simplicity, we limit our discussion on decoding to the turbo code presented in Section 6.1.2. The basic principle applies to all turbo codes.

6.1.3.1 MAP Decoding and Log Likelihood Ratio

Let $\vec{c} = \boldsymbol{c}_0, \boldsymbol{c}_1, \cdots, \boldsymbol{c}_{N-1}$ be a coded sequence produced by the rate-1/2 RSC encoder, and $\vec{r} = \boldsymbol{r}_0, \boldsymbol{r}_1, \cdots, \boldsymbol{r}_{N-1}$ be the noisy received sequence, where the codeword is $\boldsymbol{c}_k = \left(c_k^{(1)} c_k^{(2)}\right)$, with the first bit (i.e., the systematic bit) being the message bit $c_k^{(1)} = m_k$ and the second bit being the parity bit. The corresponding received word is $\boldsymbol{r}_k = \left(r_k^{(1)} r_k^{(2)}\right)$. The coded bit can take on the values +1 or −1.[2] The maximum a posteriori (MAP) decoding is carried out as follows:

$$c_k^{(1)} = \begin{cases} +1, & \text{if } P\left(c_k^{(1)} = +1 \mid \vec{r}\right) \geq P\left(c_k^{(1)} = -1 \mid \vec{r}\right) \\ -1, & \text{if } P\left(c_k^{(1)} = +1 \mid \vec{r}\right) < P\left(c_k^{(1)} = -1 \mid \vec{r}\right) \end{cases} \quad (i = 0, 1, \cdots, N-1) \tag{6.3}$$

[2] Assuming BPSK signaling, 0 → +1, 1 → −1.

We define a quantity called the a posteriori *log likelihood ratio* (LLR) of $c_k^{(1)}$ as follows:

$$L\left(c_k^{(1)}\right) \triangleq \ln\left[\frac{P\left(c_k^{(1)} = +1 \mid \vec{r}\right)}{P\left(c_k^{(1)} = -1 \mid \vec{r}\right)}\right] \tag{6.4}$$

The MAP decoding rule in (6.3) can be alternatively expressed as:

$$c_k^{(1)} = \text{sign}\left[L\left(c_k^{(1)} \mid \vec{r}\right)\right] \tag{6.5}$$

Evidently, the magnitude of the LLR, $\left|L\left(c_k^{(1)} \mid \vec{r}\right)\right|$, measures the likelihood (or reliability, confidence) of $c_k^{(1)} = +1$ (or $c_k^{(1)} = -1$). Now all we need for MAP decoding is to compute the a posteriori LLR. The LLR can be expressed as a function of the probability $P(c_k^{(1)} = +1 | \vec{r})$:

$$L\left(c_k^{(1)}\right) = \ln\left[\frac{P\left(c_k^{(1)} = +1 \mid \vec{r}\right)}{P\left(c_k^{(1)} = -1 \mid \vec{r}\right)}\right] = \ln\left[\frac{P\left(c_k^{(1)} = +1 \mid \vec{r}\right)}{1 - P\left(c_k^{(1)} = +1 \mid \vec{r}\right)}\right] \tag{6.6}$$

and is depicted in Figure 6.14.

6.1.3.2 BCJR Algorithm

The most common algorithms for computing the LLR are the BCJR algorithm and the SOVA algorithm. We first introduce the BCJR algorithm in this section.

Applying Bayes' rule to (6.4), we obtain:

$$\begin{aligned} L\left(c_k^{(1)}\right) &= \ln\left[\frac{P\left(c_k^{(1)} = +1 \mid \vec{r}\right)}{P\left(c_k^{(1)} = -1 \mid \vec{r}\right)}\right] = \ln\left[\frac{P\left(c_k^{(1)} = +1, \vec{r}\right) / P(\vec{r})}{P\left(c_k^{(1)} = -1, \vec{r}\right) / P(\vec{r})}\right] \\ &= \ln\left[\frac{P\left(c_k^{(1)} = +1, \vec{r}\right)}{P\left(c_k^{(1)} = -1, \vec{r}\right)}\right] \end{aligned} \tag{6.7}$$

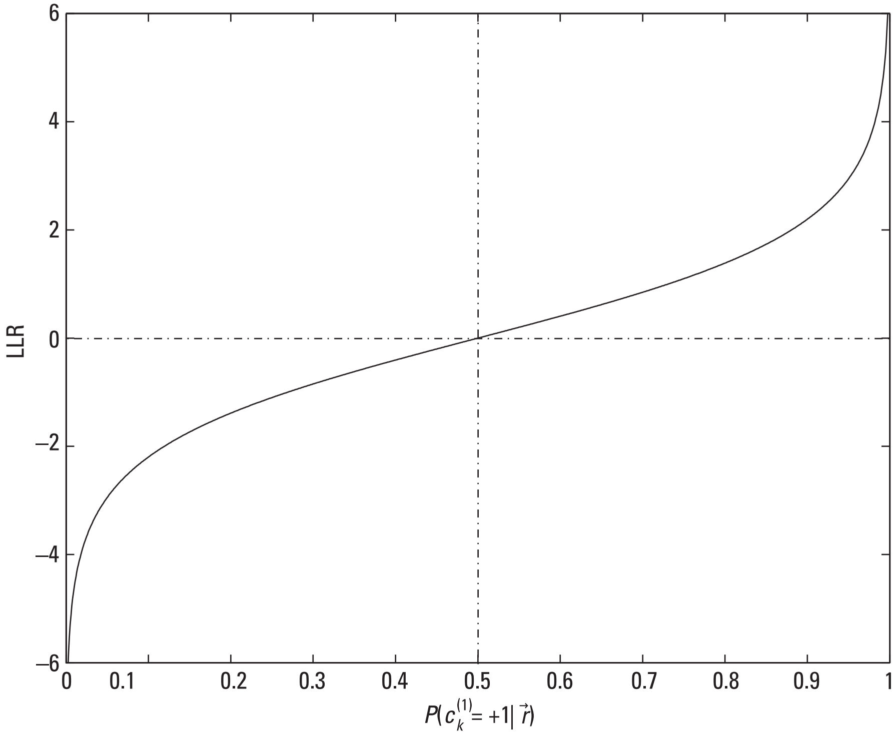


Figure 6.14 LLR versus a posteriori probability.

Now incorporating the code trellis structure (see Example 6.2), we have:

$$\begin{aligned} P\left(c_k^{(1)} = +1, \vec{r}\right) &= \sum\nolimits_{S^+} P(s_{k-1} = S', s_k = S, \vec{r}) \\ P\left(c_k^{(1)} = -1, \vec{r}\right) &= \sum\nolimits_{S^-} P(s_{k-1} = S', s_k = S, \vec{r}) \end{aligned} \tag{6.8}$$

where s_{k-1} and s_k denote the encoder states at time $k - 1$ and k, respectively; the subscript S^+ means the sum is computed over the set of all transitions from state S' to state S due to message bit $m_k = +1$; and S^- is the set of all state transitions caused by $m_k = -1$. Then (6.7) can be reformulated as follows:

$$L\left(c_k^{(1)}\right) = \ln\left[\frac{P\left(c_k^{(1)} = +1, \vec{r}\right)}{P\left(c_k^{(1)} = -1, \vec{r}\right)}\right] = \ln\left[\frac{\sum_{S^+} P(s_{k-1} = S', s_k = S, \vec{r})}{\sum_{S^-} P(s_{k-1} = S', s_k = S, \vec{r})}\right] \tag{6.9}$$

Let us break the received sequence $\vec{r}$ into three pieces, one containing the past, another the present (i.e., time k), and the third the future:

$$\vec{r} = \underbrace{\boldsymbol{r}_0, \boldsymbol{r}_1, \cdots, \boldsymbol{r}_{k-1}}_{\vec{r}(0:k-1)} \boldsymbol{r}_k, \underbrace{\boldsymbol{r}_{k+1}, \cdots, \boldsymbol{r}_{N-1}}_{\vec{r}(k+1:N-1)} \tag{6.10}$$

We then have:

$$\begin{aligned} P(s_{k-1} = S', s_k = S, \vec{r}) &= P[s_{k-1} = S', s_k = S, \vec{r}(0:k-1), \boldsymbol{r}_k, \vec{r}(k+1:N-1)] \\ &= P\Big[\vec{r}\big(k+1:N-1\big) \,|\, s_{k-1} = S', s_k = S, \vec{r}(0:k-1), \boldsymbol{r}_k\Big] \\ &\quad \cdot P\Big[s_{k-1} = S', s_k = S, \vec{r}(0:k-1), \boldsymbol{r}_k\Big] \end{aligned} \tag{6.11}$$

Recall that the future coded output of a convolutional encoder is dependent on the current state s_k, but not on the previous states s_{k-1}, s_{k-2}, … nor on the current and past inputs (i.e., it is a Markov process). A memoryless channel (e.g., AWGN) retains this characteristic; therefore, the received sequence will bear the same Markovity. Consequently (6.11) can be simplified to:

$$\begin{aligned} &P(s_{k-1} = S', s_k = S, \vec{r}) \\ &= P\Big[\vec{r}(k+1:N-1) \,|\, s_{k-1} = S', s_k = S, \vec{r}(0:k-1), \boldsymbol{r}_k\Big] \\ &\quad \cdot P\Big[s_{k-1} = S', s_k = S, \vec{r}(0:k-1), \boldsymbol{r}_k\Big] \\ &= P\Big[\vec{r}(k+1:N-1) \,|\, s_k = S\Big] \cdot P\Big[s_{k-1} = S', s_k = S, \vec{r}(0:k-1), \boldsymbol{r}_k\Big] \\ &= P\Big[\vec{r}(k+1:N-1) \,|\, s_k = S\Big] \cdot P\Big[s_k = S, \boldsymbol{r}_k \,|\, s_{k-1} = S', \vec{r}(0:k-1)\Big] \\ &\quad \cdot P\Big[s_{k-1} = S', \vec{r}(0:k-1)\Big] \\ &= \underbrace{P\Big[\vec{r}(k+1:N-1) \,|\, s_k = S\Big]}_{\beta_k(S)} \cdot \underbrace{P\Big[s_k = S, \boldsymbol{r}_k \,|\, s_{k-1} = S'\Big]}_{\gamma_k(S',S)} \\ &\quad \cdot \underbrace{P[s_{k-1} = S', \vec{r}(0:k-1)]}_{\alpha_{k-1(S')}} \\ &= \alpha_{k-1}(S') \gamma_k(S',S) \beta_k(S) \end{aligned} \tag{6.12}$$

Equation (6.7) then becomes:

$$L\left(c_k^{(1)}\right) = \ln \frac{\sum_{S^+} P\left(s_{k-1} = S', s_k = S, \vec{r}\right)}{\sum_{S^-} P\left(s_{k-1} = S', s_k = S, \vec{r}\right)} = \ln \frac{\sum_{S^+} \alpha_{k-1}(S')\gamma_k(S',S)\beta_k(S)}{\sum_{S^-} \alpha_{k-1}(S')\gamma_k(S',S)\beta_k(S)} \tag{6.13}$$

Now we state the BCJR algorithm for computing the three probabilities $\alpha_{k-1}(S')$, $\gamma_k(S', S)$, and $\beta_k(S)$ for the rate-1/2 component code of the turbo code. First $\gamma_k(S', S)$ is computed as follows:

$$\gamma_k(S',S) = Z_k \cdot \exp\left(\frac{L_c}{2}\sum_{i=1}^{2} c_k^{(i)} r_k^{(i)}\right) \cdot \exp\left[\frac{L^{\text{ext}}(c_k^{(1)}) \cdot r_k^{(i)}}{2}\right] \tag{6.14}$$

where Z_k is some constant, and can be ignored since it appears both in the numerator and in the denominator of (6.13) and cancels out.[3] (For a proof of (6.14), the reader is referred to [15, Chap. 14, Section 14.3].) The $L^{\text{ext}}(c_k^{(1)})$ term is the LLR of the bit $c_k^{(1)}$, but *not* the one we are computing; rather, it is supplied by an external source (the superscript "ext" means it is extrinsic).[4]

Given the coding rate $R = 1/2$, the quantity L_c is calculated to be:

$$L_c = 4R\frac{E_{\text{b}}}{N_0} = 2\frac{E_{\text{b}}}{N_0} \tag{6.15}$$

The value of $\alpha_k(S)$ is computed recursively:

$$\alpha_k(S) = \sum_{\text{all } S'} \gamma_k(S',S)\alpha_{k-1}(S') \tag{6.16}$$

where the subscript "all S'" means the summation is over all states at time $k - 1$ linked to state S at time k. Assuming that encoding starts with the all-zero state, the initial condition for α_0 is:

$$\alpha_0 = \begin{cases} 1 & \text{for all-zero state} \\ 0 & \text{for nonzero states} \end{cases} \tag{6.17}$$

[3] Readers could imagine that, for a rate-$1/n$ code, the sum is over $i = 1$ to $i = n$, that is, $\sum_{i=1}^{n} c_k^{(i)} r_k^{(i)}$.

[4] Obviously the extrinsic LLR must be less accurate; otherwise, there is no point in computing the LLR again here.

Finally,

$$\beta_{k-1}(S') = \sum\nolimits_{\text{all } S} \gamma_k(S', S)\beta_k(S) \tag{6.18}$$

is also computed recursively, where the subscript "all S" means all states at time k with branches originating from state S' at time $k - 1$. The boundary condition is:

$$\beta_N = \begin{cases} 1 & \text{for all-zero state} \\ 0 & \text{for nonzero states} \end{cases} \tag{6.19}$$

Equations (6.14), (6.16), and (6.18) constitute the BCJR algorithm. We make the following observations about the algorithm:

1. Probability γ has to be computed first, because it is needed in computing both α and β.
2. Probability α is computed as $\vec{r}$ is being received; in other words, it is computed forward from the beginning of the trellis to the end.
3. Probability β can only be computed after we have received the entire sequence $\vec{r}$; that is, it is computed backward from the end of the trellis to the beginning.
4. The terms α and β are associated with the encoder states and γ is associated with the state transitions.
5. Equation (6.19) implies that the encoder is terminated at the all-zero state. Therefore, flushing bits are required.

The BCJR algorithm is also known as the *forward-backward algorithm*; the reason is evident. Also, α, β, and γ are referred to as the forward metric, backward metric, and transition metric, respectively.

As with the path metric in the Viterbi algorithm, the forward metric and the backward metric, too, face potential numerical overflow issues. To circumvent the problem, α and β may undergo the following normalization:

$$\bar{\alpha}_k(S) = \alpha_k(S) \Big/ \sum\nolimits_{\text{all } S} \alpha_k(S) \tag{6.20}$$

$$\bar{\beta}_{k-1}(S') = \beta_{k-1}(S') \Big/ \sum\nolimits_{\text{all } S'} \beta_{k-1}(S') \tag{6.21}$$

so that

$$\sum\nolimits_{\text{all } S} \bar{\alpha}_k(S) = 1 \text{ and } \sum\nolimits_{\text{all } S'} \bar{\beta}_{k-1}(S') = 1$$

Notice that the normalization does not affect computation of $L(c_k^{(1)})$ because it is applied to both the numerator and the denominator and cancels out. That is to say, the same LLR can be obtained as:

$$L\left(c_k^{(1)}\right) = \ln\left[\sum\nolimits_{S^+} \bar{\alpha}_{k-1}(S')\gamma_k(S',S)\bar{\beta}_k(S) \Big/ \sum\nolimits_{S^-} \bar{\alpha}_{k-1}(S')\gamma_k(S',S)\bar{\beta}_k(S)\right] \tag{6.22}$$

Computation of the three metrics is best performed with the aid of the trellis diagram. We now give an example.

Example 6.4

Consider the rate-1/2 RSC code used in the turbo code above. Assume an all-zero message sequence $\vec{m}$ = 0,0,0,0,0,0,0,0. The coded sequence $\vec{c}$ is also an all-zero sequence $\vec{c}$ = (+1 +1), (+1 +1), (+1 +1), (+1 +1), (+1 +1), (+1 +1), (+1 +1), (+1 +1) using BPSK signaling. The coded sequence, after going through an AWGN channel with $E_b/N_0 = 0.5$, arrives at the receiver as:

$$\vec{r} = (-0.3\ 0.7), (-0.8\ 0.7), (0.2\ 0.5), (-0.1\ 0.6), (0.2\ 0.6)$$
$$(0.6\ 0.3), (0.5\ 0.9), (-0.2\ 0.6)$$

According to (6.15), $E_b/N_0 = 0.5 \Rightarrow L_c = 1$. Because we do not have the extrinsic LLR, we simply assume it equals zero. The transition metric is first computed using (6.14), which is exemplified in the leftmost figure of Figure 6.15:

$$\gamma_2(2,1) = \exp\left\{\frac{1}{2}\left[(+1)r_2^{(1)} + (+1)r_2^{(2)}\right]\right\}$$
$$= \exp\left\{\frac{1}{2}[(+1)(-0.8) + (+1)(0.7)]\right\} = 0.9512$$

Note that we number the states as 0→(00), 1→(10), 2→(01) and 3→(11), Next, using (6.16) and (6.18), the other two metrics are calculated. As an example,

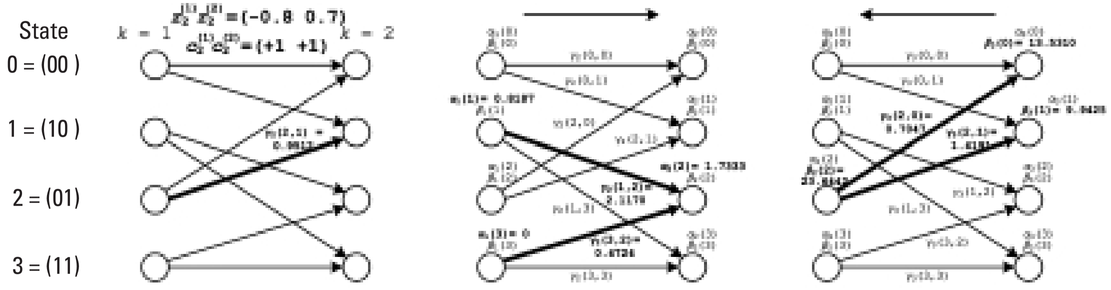


Figure 6.15 Trellis-assisted computation of BCJR metrics.

$$\begin{aligned}\alpha_2(2) &= \alpha_1(1)\ \gamma_2(1,2) + \alpha_1(3)\ \gamma_2(3,2)\\ &= 0.8187 \times 2.1170 + 0 \times 0.4724 = 1.7333\\ \beta_2(2) &= \beta_2(0)\ \gamma_3(2,0) + \beta_3(1)\ \gamma_3(2,1)\\ &= 13.5310 \times 0.7047 + 9.9425 \times 1.4191 = 23.6442\end{aligned}$$

The complete computation process is illustrated in Figure 6.16 where $\tilde{c}_k$ is the decoded bit. Note that the received sequence contains four errors if hard-decision decoded by thresholding $\vec{r}$ at 0, and the MAP decoding is able to correct three of the four errors.

MATLAB Experiment 6.6

A function `bcjr*` is provided to compute α, β, and γ, as well as the LLR L. We now use it to solve Example 6.4.

```
>> % received sequence
>> r = [-0.3 0.7 -0.8 0.7 0.2 0.5 -0.1 0.6 0.2 0.6 0.6...
0.3 0.5 0.9 -0.2 0.6];
>> % bcjr
>> [L,a,b,g] = bcjr(r,0,1);  % a: alpha, b: beta, g: gamma,
>>                           % L: LLR, Lext = 0, Lc = 1
```

We get the same results as in the example. For instance,

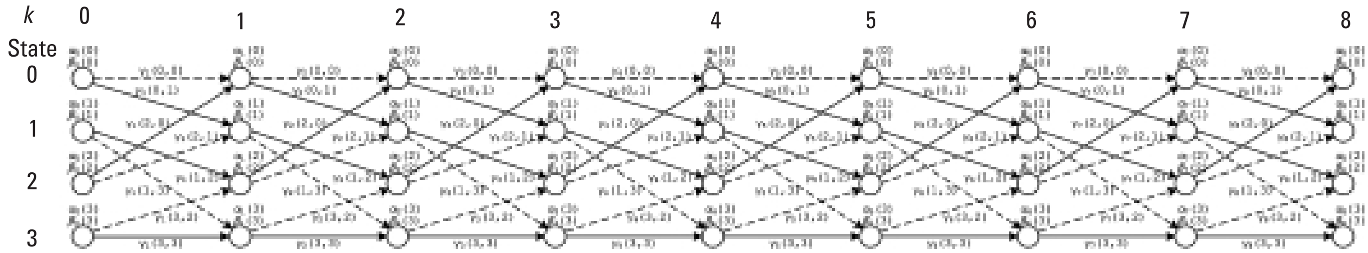
```
>> L
L =
    0.1799 -0.5850 0.2627 0.0092 0.2870 0.8020 0.7513 0.6719
```

MATLAB Experiment 6.7

Suppose that we have somehow obtained the extrinsic $L^{\text{ext}}(c_k^{(1)})$ ($k = 1, 2, \ldots, 8$) to be 0.59, 0.48, 0.12, 0.25, 0.25, 0.31, 0.66, 1.02. Recalculate the four quantities α, β, γ, and L.

```
>> Lext = [0.59 0.48 0.12 0.25 0.25 0.31 0.66 1.02];
>> L = bcjr(r,Lext,1);
>> L
L =
     1.0896 0.4932 0.9152 0.8842 1.1632 1.6394 1.7918 1.7329
```

We see that all bits are correctly decoded. In this new experiment, the extrinsic LLR information has helped the decoding. Now the natural question is this: How do we get it? In fact, this is the central point of turbo decoding and is presented next.



$\gamma_k(0,0)$	1.2214	0.9512	1.4191	1.2840	1.4918	1.5683	2.0138	1.2214
$\gamma_k(0,1)$	0.8187	1.0513	0.7047	0.7788	0.6703	0.6376	0.4966	0.8187
$\gamma_k(2,0)$	0.8187	1.0513	0.7047	0.7788	0.6703	0.6376	0.4966	0.8187
$\gamma_k(2,1)$	1.2214	0.9512	1.4191	1.2840	1.4918	1.5683	2.0138	1.2214
$\gamma_k(1,2)$	1.6487	2.1170	1.1618	1.4191	1.2214	0.8607	1.2214	1.4918
$\gamma_k(1,3)$	0.6065	0.4724	0.8607	0.7047	0.8187	1.1618	0.8187	0.6703
$\gamma_k(3,2)$	0.6065	0.4724	0.8607	0.7047	0.8187	1.1618	0.8187	0.6703
$\gamma_k(3,3)$	1.6487	2.1170	1.1618	1.4191	1.2214	0.8607	1.2214	1.4918

$\alpha_k(0)$	1	1.2214	1.1618	2.8781	5.1064	11.4786	23.9126	58.6278	108.1783
$\alpha_k(1)$	0	0.8187	1.2840	3.2783	4.5782	11.9973	21.8826	54.3444	102.5575
$\alpha_k(2)$	0	0	1.7333	1.8247	5.7476	9.2893	21.0899	44.6681	111.0224
$\alpha_k(3)$	0	0	0.3867	1.5545	4.5161	9.2644	21.9128	44.6803	103.0833
$\beta_k(0)$	108.1783	48.2579	26.2877	13.5310	7.3778	4.4951	2.4396	1.2214	1
$\beta_k(1)$	114.4067	60.1369	22.1906	9.9425	4.8804	1.3008	1.0000	0	0
$\beta_k(2)$	112.9616	48.6598	23.6442	12.1682	4.9538	3.1366	0.6065	0.8187	0
$\beta_k(3)$	122.4254	56.3538	21.3439	9.3565	4.1334	1.2816	0.6703	0	0

Σ_{u+}	58.8423	38.7036	61.1544	54.3368	61.7983	74.6875	73.5041	71.6071
Σ_{u-}	49.2360	69.4747	47.0239	53.8415	46.3808	33.4908	34.6742	36.5712
$\ln(\Sigma_{u+}/\Sigma_{u-})$	0.1799	-0.5850	0.2627	0.0092	0.2870	0.8020	0.7513	0.6719
$\hat{u}_k^{(1)}$	0	1	0	0	0	0	0	0

Figure 6.16 A BCJR decoding process.

6.1.3.3 Turbo Principle and Iterative Decoding of Turbo Codes

Let us now leave LLR for awhile, and discuss the iterative decoding concept for turbo codes.

In their pioneering work, Berrou et al. [1] adopted MAP decoding to decode their code. However, as we have seen, without the extrinsic LLR, the performance of MAP is compromised. The turbo code inventors then turned to a decoding approach called *iterative decoding*, which starts with a rough estimate of the code and iteratively improves it toward true optimum decoding. The trick is that the turbo decoder employs two component decoders. At each decoding iteration, the two component decoders produce the extrinsic LLRs for each other and exchange them with each other. Each component decoder uses the incoming extrinsic information as a priori probability to refine its decoding outcome, and at the same time generates more accurate extrinsic information for the other decoder. This ping-pong–like decoding principle is similar to a turbo engine in which compressed air is fed back from the compressor to the main engine cylinder in order to gain more power, and therefore is referred to as the *turbo principle* (Figure 6.17).

Figure 6.18 shows the structure of the turbo decoder, where D_I and D_II are the two component decoders. Before the decoding starts, the received sequence $\vec{r}$ is demultiplexed into three subsequences, $\vec{r}^{\,(1)}$, $\vec{r}^{\,(2)}$ and $\vec{r}^{\,(3)}$ in correspondence with the three code bit sequences $\vec{c}^{\,(1)}$, $\vec{c}^{\,(2)}$ and $\vec{c}^{\,(3)}$ respectively (see Figure 6.9). The turbo decoder works as follows. First, $\vec{r}^{\,(1)}$ and $\vec{r}^{\,(2)}$ are fed to the component decoder D_I. D_I performs decoding and at the same time produces extrinsic LLR information (which is a posteriori to D_I itself). This information is interleaved and passed on to D_II. Next D_II uses it as a priori for its decoding and produces more accurate extrinsic information for D_1. Afterwards, the extrinsic information given by D_II is deinterleaved and sent back to D_I for a new iteration of decoding. The decoding proceeds

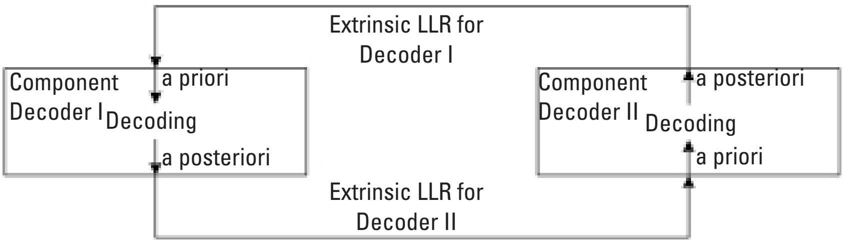


Figure 6.17 Turbo decoding principle.

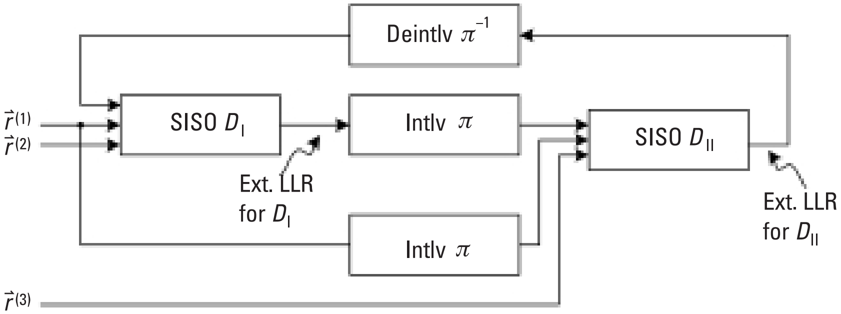


Figure 6.18 Structure of turbo decoder.

as $D_I \rightarrow D_{II} \rightarrow D_I \rightarrow D_{II} \rightarrow \cdots$ until the process has converged or until a predetermined iteration count is reached. As the decoding proceeds, the decoding result is iteratively improved. Notice that the component decoder accepts soft LLR from and generates soft LLR for its companion; therefore, it is called the *soft-in-soft-out* (SISO) decoder.

Now we relate the LLR to the turbo decoding. It can be shown [15, 16] that the LLR can be decomposed into the sum of three parts, a quantity related to channel condition χ_k, an a priori probability L_k^{apri} and an a posteriori probability L_k^{apos}, that is:

$$L\left(c_k^{(1)}\right) = \chi_k + L_k^{\text{apri}} + L_k^{\text{apos}} \tag{6.23}$$

where L_k^{apri} is the very extrinsic LLR provided by the other component decoder, and L_k^{apos} represents the extrinsic LLR that this component decoder generates for its companion. Use of the LLR in turbo decoding process can be illustrated as in Figure 6.19. The subscript I or II indicates the LLR computed by decoder I or decoder II.

The value of L_k^{apos} is obtained based on (6.23):

$$L_k^{\text{apos}} = L\left(c_k^{(1)}\right) - \chi_k - L_k^{\text{apri}} \tag{6.24}$$

where χ_k is computed as:

$$\chi_k = L_c r_k^{(1)} \tag{6.25}$$

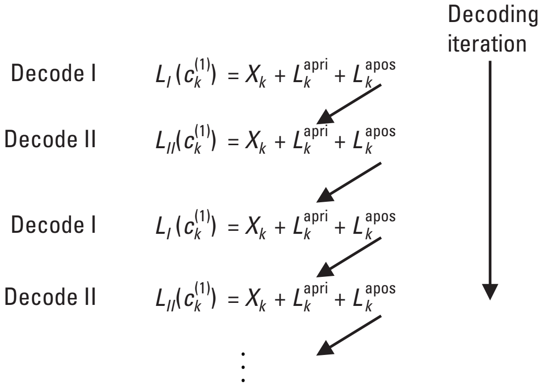

Decoding iteration
Decode I
$L_I(c_k^{(1)}) = X_k + L_k^{\text{apri}} + L_k^{\text{apos}}$
Decode II
$L_{II}(c_k^{(1)}) = X_k + L_k^{\text{apri}} + L_k^{\text{apos}}$
Decode I
$L_I(c_k^{(1)}) = X_k + L_k^{\text{apri}} + L_k^{\text{apos}}$
Decode II
$L_{II}(c_k^{(1)}) = X_k + L_k^{\text{apri}} + L_k^{\text{apos}}$
⋮

Figure 6.19 Use of LLR in turbo decoding.

We now can summarize the turbo decoding procedure:

Turbo Decoding

Initialization:

Compute $\vec{\chi} = \chi_1, \chi_{2,} \ldots, \chi_N$ [use (6.25)].

Decoding Iteration:

1. Decoder I: Computes $L_{\text{I}}(c_k^{(1)})$ (using BCJR[5]) for $k = 1, 2, \ldots, N$. Computes L_k^{apos} based on (6.24) and outputs it. If it is the first iteration, set $L_k^{\text{apri}} = 0$; otherwise, L_k^{apri} is the deinterleaved extrinsic from decoder II.
2. Decoder II: Computes $L_{\text{II}}(c_k^{(1)})$ (using BCJR or SOVA) for $k = 1, 2, \ldots, N$. Computes L_k^{apos} based on (6.24) and output. The term L_k^{apri} is the interleaved extrinsic from decoder I.
3. If decoding converges, or reaches a predetermined number of decoding iterations, stop and output the hard decision [use (6.5)]; otherwise, go back to step 1.

To maximize the benefit of the information exchange, the two component decoders must operate as independently as possible. Otherwise, the

[5] We will see shortly that SOVA can also be used.

extrinsic information produced by the two decoders will be very much correlated and the information exchange loses its sense. This further explains why a random-like interleaver is so important in turbo codes.

Note that the turbo principle is not limited to turbo decoding; it can serve as effective solutions to many other tasks in digital communications as well, for example, channel equalization (turbo equalization), multiple-in-multiple-out (MIMO) processing (turbo MIMO), and multiuser detection (turbo multiuser detection).

Example 6.5

Now we use BCJR to decode our example turbo code (the code is the same as Example 6.3 except the interleaver here is 8,2,5,4,7,6,3,1). The code is an all-zero sequence, and $L_c = 1.7825$. The received sequence is:

$r = (-0\ 3\ \ 0.7\ \ 0.6),(-0.8\ \ 0.7-0.8),\ (0.2\ \ 0.5\ \ 0.9),\ (-0.1\ \ 0.6\ \ 0.6),$
$(0.2\ \ 0.6\ \ 0.5),\ (0.6\ \ 0.3\ \ 0.2),\ (0.5\ \ 0.9\ \ 0.6),\ (-0.2\ 0.6\ 0.5)$

We run five decoding iterations:
Iteration 1: Decoder I works on noninterleaved (or de-interleaved) sequence:

k	1	2	3	4	5	6	7	8
$L(c_k^{(1)})$	0.20	−0.59	0.46	0.27	0.66	1.731	1.68	1.54
L_k^{apri}	0	0	0	0	0	0	0	0
L_k^{apos}	0.74	0.87	0.10	0.45	0.31	0.66	0.79	1.89

Iteration 2: Decoder II works on interleaved sequence:

k	1	2	3	4	5	6	7	8
$L(c_k^{(1)})$	3.39	−0.47	−0.11	0.25	1.95	2.24	1.28	1.76
L_k^{apri}	1.89	0.87	0.31	0.45	0.79	0.66	0.10	0.74
L_k^{apos}	1.86	0.09	−0.78	−0.02	0.27	0.51	0.81	1.56

At the fifth iteration, the LLR is:

k	1	2	3	4	5	6	7	8
$L(c_k^{(1)})$	5.97	4.75	5.78	5.10	4.72	5.73	6.11	6.40

The decoded outcome is indeed an all-zero sequence.

MATLAB Experiment 6.8

Example 6.5 has been implemented in the m-file `tbcdemo.m*`. Readers should step through the program and consult this section and the previous section to enhance their understanding of the topic. Readers may also try a different number of iterations to get an idea of how the decoding improves with more iterations.

```
>> tbcdemo
L =
     5.9734 4.7521 5.7785 5.0993 4.7202 5.7251 6.1074 6.3966
```

MATLAB Experiment 6.9

If we plot the eight LLRs at different iterations, we end up with Figure 6.20. Clearly the reliability (confidence) of the decoded outcome rises as the iteration increases.

6.1.3.4 Variations on MAP Decoding

The BCJR algorithm suffers from a large amount of multiplications. To reduce this computational burden, two simplified MAP algorithms are often adopted in practice: the *max-log-MAP algorithm* and the *log-MAP algorithm* [17]. Both algorithms substitute additions for multiplications. For the first algorithm, the price paid is about a 0.35-dB performance loss [18].

Take the natural logarithm on the three probabilities $\alpha_k(S)$, $\beta_k(S)$, and $\gamma_k(S',S)$:

$$A_k(S) \triangleq \ln[\alpha_k(S)] \tag{6.26}$$

$$B_k(S) \triangleq \ln[\beta_k(S)] \tag{6.27}$$

and

$$\Gamma_k(S',S) \triangleq \ln[\gamma_k(S',S)] \tag{6.28}$$

Substituting (6.16) and (6.18) into (6.26) and (6.27), respectively, yields:

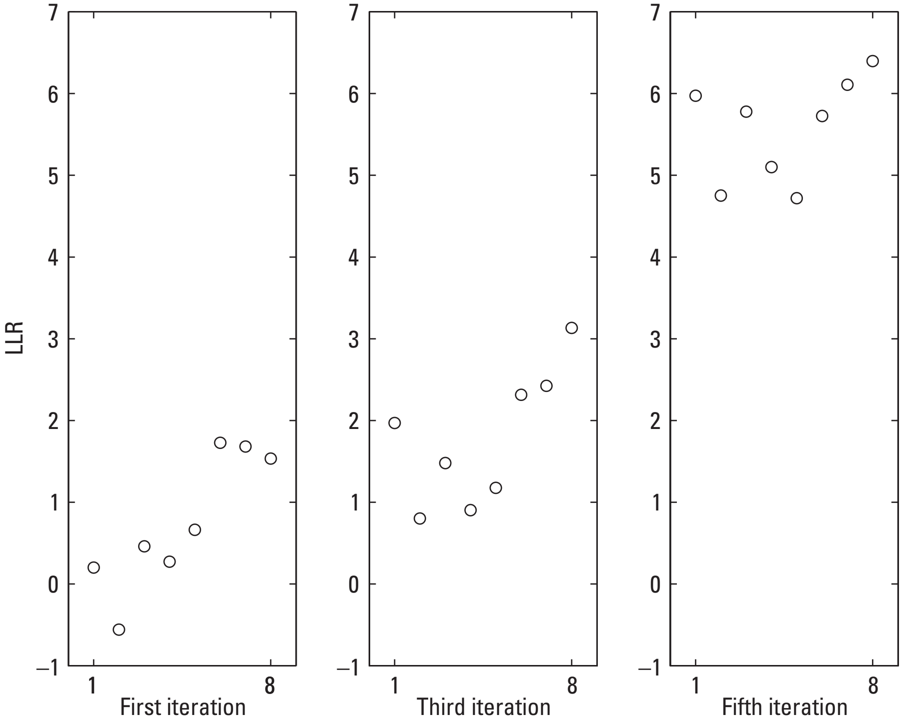


Figure 6.20 Decoding reliability versus decoding iteration.

$$\begin{aligned} A_k(S) &= \ln[\alpha_k(S)] = \ln\left[\sum\nolimits_{\text{all } S'} \gamma_k(S',S)\alpha_{k-1}(S')\right] \\ &= \ln\left(\sum\nolimits_{\text{all } S'} e^{\Gamma_k(S',S)} \cdot e^{A_{k-1}(S')}\right) \qquad (6.29\text{a}) \\ &= \ln\left(\sum\nolimits_{\text{all } S'} e^{\Gamma_k(S',S)+A_{k-1}(S')}\right) \end{aligned}$$

$$\begin{aligned} B_k(S) &= \ln[\beta_k(S)] = \ln\left[\sum\nolimits_{\text{all } S} \gamma_k(S',S)\beta_k(S)\right] = \ln\left(\sum\nolimits_{\text{all } S} e^{\Gamma_k(S',S)} \cdot e^{B_k(S)}\right) \\ &= \ln\left(\sum\nolimits_{\text{all } S} e^{\Gamma_k(S',S)+B_k(S)}\right) \qquad (6.29\text{b}) \end{aligned}$$

Also, substituting (6.14) into (6.28), we obtain:

$$\Gamma_k(S',S) = \frac{L_c}{2}\sum\nolimits_{t=1}^{2} c_k^{(i)} r_k^{(i)} + \frac{L^{\text{ext}}(c_k^{(1)}) \cdot r_k^{(1)}}{2} \qquad (6.30)$$

Finally, following (6.13) and (6.30), we have:

$$\begin{aligned} L\left(c_k^{(1)}\right) &= \ln\left[\sum\nolimits_{S^+}\alpha_{k-1}(S')\gamma_k(S',S)\beta_k(S)\Big/\sum\nolimits_{S^-}\alpha_{k-1}(S')\gamma_k(S',S)\beta_k(S)\right] \\ &= \ln\Big\{\sum\nolimits_{S^+}\exp[A_{k-1}(S)+B_k(S)+\Gamma_k(S',S)]\Big/ \\ &\quad \sum\nolimits_{S^-}\exp[A_{k-1}(S)+B_k(S)+\Gamma_k(S',S)]\Big\} \end{aligned} \tag{6.31}$$

Equations (6.29) through (6.31) tell us that we can perform MAP decoding in the log domain.

The max-log-MAP algorithm uses the following approximation to avoid summation of exponentials in computing $A_k(s)$ $B_k(s)$ and $L(c_k^{(1)})$:

$$\ln\left(\sum\nolimits_i e^{x_i}\right) \approx \max(x_i) \tag{6.32}$$

Then (6.29) and (6.31) become:

$$A_k(S) \approx \max_{S'}[\Gamma_k(S',S)+A_{k-1}(S')] \tag{6.33}$$

$$B_k(S) \approx \max_{S}[\Gamma_k(S',S)+B_k(S)] \tag{6.34}$$

and

$$\begin{aligned} L\left(c_k^{(1)}\right) \approx & \max_{S^+}[A_{k-1}(S')+B_k(S)+\Gamma_k(S',S)] \\ & -\max_{S^-}[A_{k-1}(S')+B_k(S)+\Gamma_k(S',S)] \end{aligned} \tag{6.35}$$

As mentioned, the max-log-MAP algorithm suffers some performance degradation when compared to exact decoding. The problem can be solved by the log-MAP algorithm, which uses the Jacobian logarithm:

$$\ln(e^x+e^y) = \underbrace{\max(x,y)+\ln\left(1+e^{-|x-y|}\right)}_{\max^*(x,y)} = \max^*(x,y) \tag{6.36}$$

where $\ln(1 + e^{-|x-y|})$ is called the offset.

Applying (6.36) to (6.29) and (6.32), the log-MAP algorithm computes $A_k(s)$ $B_k(s)$ and $L(c_k^{(1)})$ recursively as follows:

$$\ln\left(\sum_i e^{x_i}\right) = \max{}^*\left(\cdots\left(x_5, \max{}^*\left(x_4, \max{}^*\left(x_3, \max{}^*\left(x_1, x_2\right)\right)\right)\right)\cdots\right) \quad (6.37)$$

If the term $\ln(1 + e^{-|x-y|})$ is implemented in an LUT, the computational complexity of log-MAP is only marginally higher than that of max-log-MAP, but the decoding is *exact* and no loss is incurred [15, p. 610].

MATLAB Experiment 6.10

The decoding of Example 6.5 using log-MAP and max-log-MAP algorithms is simulated in the files `logmap.m*` and `maxlogmap.m*`, respectively.

6.1.3.5 SOVA Decoding Algorithm

An alternative to MAP-type decoding is the *soft-output Viterbi algorithm* (SOVA). SOVA is extended from the conventional Viterbi algorithm. It differs from its predecessor in that it takes into account the a priori of its input and additionally outputs the soft reliabilities of the decoded bits (i.e., the LLR). Therefore, SOVA is also a SISO algorithm.

Recall that, in the conventional Viterbi algorithm, we choose as the survivor the path with the minimum path metric, which is the sum of the squared Euclidean distance d_{E}^2 [see (5.13) and (5.16) in Chapter 5]. This maximizes the probability [see (5.12)]:

$$p(\vec{r}\,|\,\vec{c}) = \sqrt{\frac{1}{\pi N_0}}e^{-\frac{d_{\mathrm{E}}^2}{N_0}}$$

With SOVA, we maximize the probability $p(\vec{r}\,|\vec{c})P(\vec{c})$, where $P(\vec{c})$ is the a priori probability of the code sequence $\vec{c}$. This leads to the following SOVA metric for the current trellis node (S,t) [8]:

$$SM_{(S,t)} = \sum_{\vec{c}}\left[\sum_{i=1}^{2} L_c c_t^{(i)} r_t^{(i)} + L^{\mathrm{ext}}\left(c_t^{(1)}\right) r_t^{(1)}\right] \quad (6.38)$$

where $L^{\text{ext}}(c_t^{(1)})$ is the extrinsic LLR of the systematic bit $c_t^{(1)}$. The outer sum is carried out over the path that gives the sequence $\vec{c}$. As seen from this equation, the SOVA metric incorporates the channel reliability (the first term), and the extrinsic probability as a priori supplied by an external source (the second term). Similar to the path metric in the conventional Viterbi algorithm, the SOVA metric can also be calculated recursively as:

$$SM_{(S,t)} = SM_{(S',t-1)} + \sum\nolimits_{t=1}^{2} L_c c_t^{(i)} r_t^{(i)} + L^{\text{ext}}\left(c_t^{(1)}\right) r_t^{(1)} \tag{6.39}$$

where $(S', t-1)$ is the preceding trellis node on the path.

SOVA decoding operates just like the conventional Viterbi algorithm: At every decoding step, for each state, we compare the SOVA metrics of two incoming paths and select as the survivor the path with the *maximum* metric. At the end, the global optimum path is our decoding path.

The extra task in SOVA is to compute the soft LLR output. It can be shown [19] that the probability of a path y coming to trellis node (S,t) is related to its metric $SM_{(S,t)}^{y}$ as follows:

$$P_y = \exp\left(SM_{(S,t)}^{y}/2\right) \tag{6.40}$$

Notice that there are two paths coming into trellis node (S,t). Denote the two paths as x and y and assume, without loss of generality, that y is chosen as the survivor path. The probability that y is the true survivor can be computed as follows:

$$\begin{aligned} P(y) &= \frac{P_y}{P_x + P_y} = \exp\left(\frac{SM_{(S,t)}^{y}}{2}\right) \Bigg/ \left[\exp\left(\frac{SM_{(S,t)}^{x}}{2}\right) + \exp\left(\frac{SM_{(S,t)}^{y}}{2}\right)\right] \\ &= \exp\left(\frac{SM_{(S,t)}^{y} - SM_{(S,t)}^{y}}{2}\right) \Bigg/ \left[1 + \exp\left(\frac{SM_{(S,t)}^{y} - SM_{(S,t)}^{x}}{2}\right)\right] \\ &= \exp\left(\Delta_t^{xy}\right) \Big/ \left[1 + \exp\left(\Delta_t^{xy}\right)\right] \end{aligned} \tag{6.41}$$

where Δt^{xy} is the difference of the metrics:

$$\Delta_t^{xy} \triangleq \frac{1}{2}\left|SM_t^{x} - SM_y^{t}\right| \tag{6.42}$$

The LLR of path y as the survivor, by definition, is:

$$
\begin{aligned}
L(y) &= \ln \frac{P(y)}{1-P(y)} = \ln \frac{\exp\left(\Delta_t^{xy}\right) / \left[1+\exp\left(\Delta_t^{xy}\right)\right]}{1-\exp\left(\Delta_t^{xy}\right) / \left[1+\exp\left(\Delta_t^{xy}\right)\right]} \\
&= \Delta_t^{xy}
\end{aligned}
\tag{6.43}
$$

This LLR is just for the path. What we are ultimately interested in is the LLR for each decoded bit along the path. This LLR can be computed as [8, p. 562]:

$$
L\left(c_k^{(1)}\right) = \begin{cases} \min\left[\Delta_{t-1}^{xy}, L\left(c_k^{(1)}\right)\right] & \text{if } c_k^{(1)} \neq d_k^{(1)} \\ L\left(c_k^{(1)}\right) & \text{if } c_k^{(1)} = d_k^{(1)} \end{cases} \qquad (k = 1, 2, 3, \cdots, t-1)
\tag{6.44}
$$

where $c_k^{(1)}$ is the systematic bit on path y at decoding step k, and $d_k^{(1)}$ is the systematic bit associated with path x at that step. Note that $L(c_k^{(1)})$ must be updated at each decoding step.

The computation is graphically illustrated in Figure 6.21. The binary digit on each transition branch is the systematic bit of the associated codeword. Paths x and y merge at state 0 at time 6, and y is the correct path and is chosen as the survivor. The LLR is computed according to (6.44). For example, $L(c_5^{(1)})$ equals $[\Delta_6^{xy}, L(c_5^{(1)})]$ since the systematic bit associated with the transition $(S(0),4) \rightarrow (S(0),5)$ (on path y) is 0, whereas the bit associated with the transition $(S(1),4) \rightarrow (S(2),5)$ (on path x) is 1.

Soft-Output Viterbi Algorithm

Initialization:

Set: time ← 0, initial state $SM \leftarrow 0$, all other $SM \leftarrow -\infty$, $L\left(c_k^{(1)}\right) \leftarrow \infty$.

Normal Operation:

1. Increase time by 1.
2. Compute SOVA metric according to (6.39) for every path.

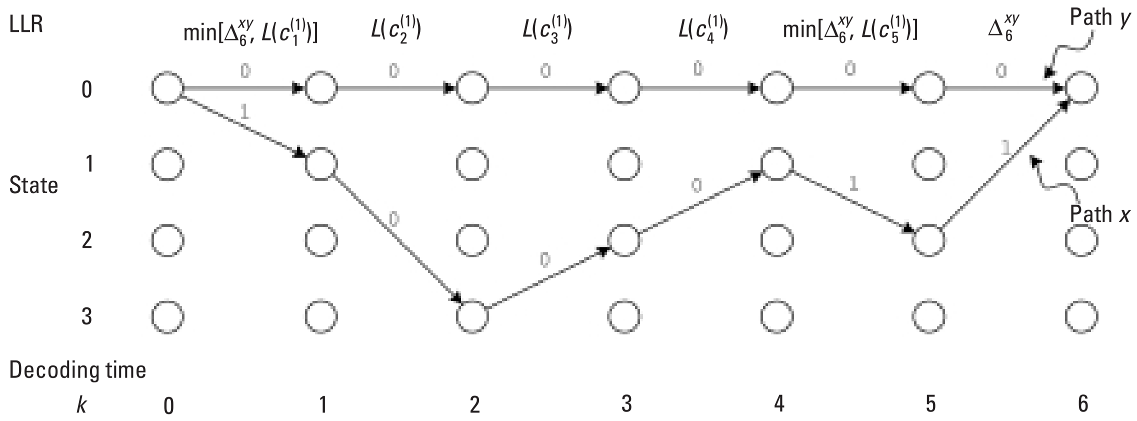


Figure 6.21 Computation of LLR in SOVA.

3. Compare SOVA metrics for each state, store the path with the maximum metric as well as the metric itself.
4. Update the LLR for each systematic bit along the survivor path using (6.44).

Decoding and Output:

At the end of the trellis, choose as the global optimum path the path ending at the initial state (if the encoder is terminated) or the survivor with maximum SOVA metric (if the encoder is not terminated). Output the bit sequence and the associated LLRs.

When applied in turbo decoding, the LLR output generated by one component SOVA decoder is used as the a priori (i.e., the extrinsic LLR) in the other component decoder for computing the SOVA metrics [see (6.38)].

MATLAB Experiment 6.11

This book's DVD includes a SOVA decoding program called `sovademo.m*`. Step through the program to gain more insight into the algorithm.

Note that, in general, SOVA is inferior to BCJR in terms of bit error performance. The main advantage of SOVA is that it has much lower implementation complexity.

SOVA can also be used in the sliding-window fashion, similar to the conventional window-sliding Viterbi decoding [15].

6.1.3.6 Criterion for Stopping Decoding Iterations

Normally turbo decoding is run for a fixed number of iterations. The fixed iteration count is set to handle the worst channel noise. However, in most cases, codewords do not experience that level of noise corruption. Consequently, decoding can finish with fewer iterations. The stopping criterion for iterative decoding is to determine if the decoding has converged so that it can be terminated earlier without compromising its error probability performance. Many such criteria have been proposed. One simple criterion is to count the number of sign changes between the current L_k^{apos} and L_{k-1}^{apos} computed at last iteration. If the number is below a certain ratio (normally 0.005 to 0.003) of the sequence length, decoding stops. This criterion is called the sign-change ratio (SCR) criterion [20]. Another one is to compare the sign of $L(c_k^{(1)})$ with the sign of $L(c_{k-1}^{(1)})$. When they are the same, decoding stops [called the hard-decision-aided (HDA) criterion] [20].

6.1.4 Implementing MAP

In practical implementations, log-MAP is most preferred for its reduced complexity and satisfactory performance. A log-MAP core consists of three processing units: the transition metric calculator (TMC), which computes Γ; the add-compare-select-offset (ACSO) unit, which calculates A and B; and the LLR calculator (LLRC), which computes $L(c_k^{(1)})$. Next we briefly discuss how to implement them.

6.1.4.1 Transition Metric Calculator

Codeword $\boldsymbol{c} = (c^{(1)}c^{(2)})$ has four combinations, that is, (00), (01), (10) and (11). Substituting them into (6.30), we arrive at the four values shown in Table 6.2 for the transition metric. Based on the table, the TMC circuit is easily designed as in Figure 6.22.

6.1.4.2 ACSO

Notice that computation of A and B share a similar recursion architecture [see (6.29)]. Therefore, we focus our discussion on A.

Write out (6.29) for our example turbo code as follows:

$$\begin{aligned} A_k(S) &= \ln\left(\sum\nolimits_{\text{all } S'} e^{\Gamma_k(S',S)+A_{k-1}(S')}\right) \\ &= \ln\left(e^{\Gamma_k(S_1',S)+A_{k-1}(S_1')} + e^{\Gamma_k(S_2',S)+A_{k-1}(S_2')}\right) \end{aligned}$$

Applying the Jacobian logarithm, we have:

$$A_k(S) = \max{}^*\left(\underbrace{\Gamma_k(S_1',S)+A_{k-1}(S_1')}_{x}, \underbrace{\Gamma_k(S_2',S)+A_{k-1}(S_2')}_{y}\right)$$

Table 6.2

Possible Transition Metrics

c_k	(00)	(01)	(10)	(11)
Γ_k	0	$L_c r^{(2)}$	$L_c r^{(1)} + L^{\text{ext}}(c_k^{(1)})$	$L_c r^{(1)} + L_c r^{(2)} + L^{\text{ext}}(c_k^{(1)})$

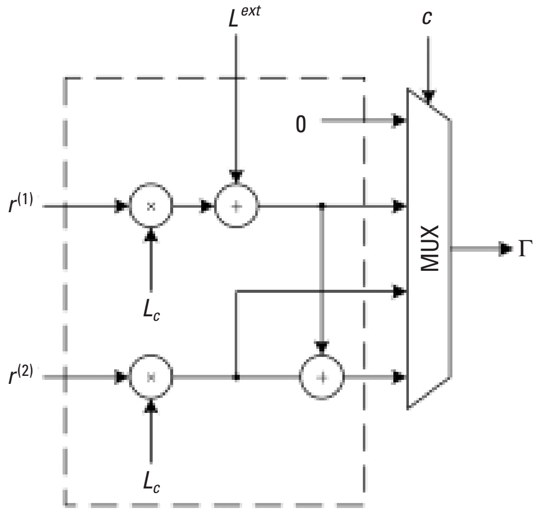


Figure 6.22 Transition metric calculator.

It is clear now that recursive calculation of A involves ACS and correction of $e^{-|x-y|}$ (i.e., the offset). This ACSO operation can be implemented in the structure shown in Figure 6.23. The circuit in the box implements the max* operation. The LUT stores the offset, which is computed off-line, and $x-y$ is used as the access address to the LUT.

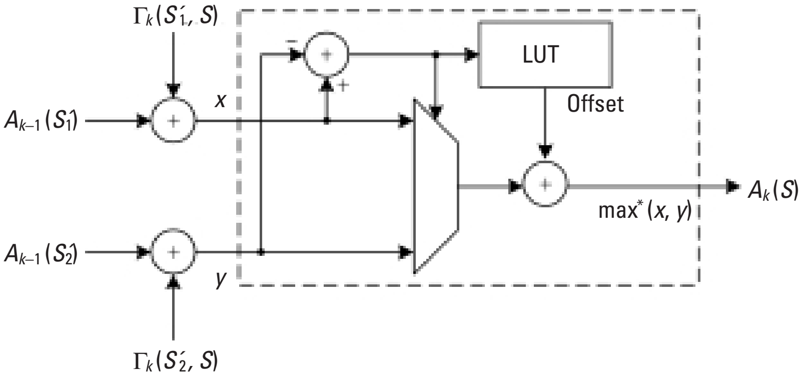


Figure 6.23 Architecture of ASCO.

6.1.4.3 LLR Calculator

We observe:

$$L\left(c_k^{(1)}\right) = \ln \frac{\sum_{S^+} \exp[A_{k-1}(S') + B_k(S) + \Gamma_k(S',S)]}{\sum_{S^-} \exp[A_{k-1}(S') + B_k(S) + \Gamma_k(S',S)]}$$

$$= \underbrace{\ln\left\{\sum_{S^+} \exp[A_{k-1}(S') + B_k(S) + \Gamma_k(S',S)]\right\}}_{\text{LLR+}}$$

$$- \underbrace{\ln\left\{\sum_{S^-} \exp[A_{k-1}(S') + B_k(S) + \Gamma_k(S',S)]\right\}}_{\text{LLR}-}$$

For the example turbo code, *S*+ contains four transition branches (refer to the trellis in Example 6.2): S(0)→S(0), S(1)→S(3), S(2)→S(1), and S(3)→S(2). Therefore, computation of LLR+ takes the form:

$$\text{LLR+} = \ln\{e^a + e^b + e^c + e^d\}$$

It follows from (6.36) that:

$$\ln\left\{e^a + e^b + e^c + e^d\right\} = \ln\left\{e^{\ln\left(e^a + e^b\right)} + e^{\ln\left(e^c + e^d\right)}\right\}$$

$$= \max{}^*\left(\max{}^*(a,b), \max{}^*(c,d)\right)$$

Therefore, computation of LLR+ can readily be realized in a circuit with a tree structure. LLR– can be obtained in exactly the same way. The schematic of an LLR calculator is shown in Figure 6.24.

6.2 Low-Density Parity-Check Codes

The low-density parity-check (LDPC) code, also known as the Gallager code, was invented as early as 1962 [2, 3]. Despite its near ideal performance, the code was forgotten for almost 30 years, largely because its decoding complexity exceeded the capabilities of the hardware for that time. It was Mackay and Neal who rediscovered the code in the mid-1990s [4, 5] and invoked intensive research on it. Most recently, the capability of the code has been pushed to only 0.0045 dB away from the Shannon limit [6], making it the best per-

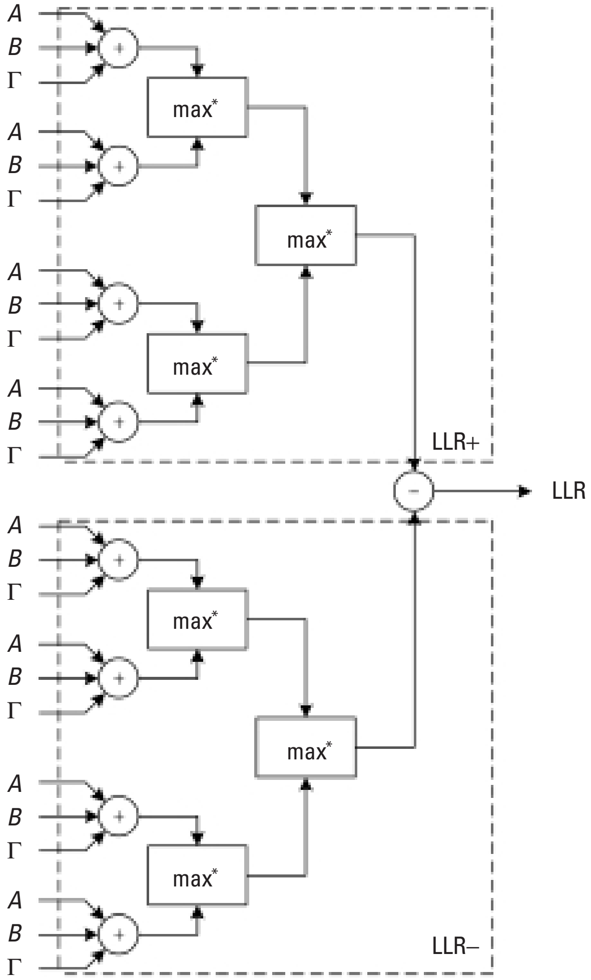


Figure 6.24 LLR calculator.

forming code known so far. On the commercial side, an additional advantage is that the LDPC code is not patented (unlike turbo code, which is patent protected in Europe and America). The disadvantages of the code include its higher encoding complexity and longer latency than turbo code. LDPC code has been adopted in several standards including IEEE 802.16 (WiMAX), IEEE 802.3an (10GBase-T Ethernet), and DVB-S2 (satellite transmission of digital television).

6.2.1 Codes with Sparse Parity-Check Matrix

6.2.1.1 LDPC Codes: Definition and Properties

An LDPC code is an (n,k) linear block code whose parity-check matrix $\boldsymbol{H}$ contains only a few 1's in comparison to 0's (i.e., sparse matrix). For an $m \times n$ parity-check matrix (where $m = n - k$), we define two parameters: the column weight J equal to the number of nonzero elements in a column and the row weight K equal to the number of nonzero elements in a row. So, for LDPC codes, $J \ll n$ and $K \ll m$. However, we want to point out that the generator matrix of LDPC code $\boldsymbol{G}$ is in general not sparse.

Example 6.6

The following parity-check matrix represents an LDPC code. Notice that the number of 1's is far less than the number of 0's in the matrix.

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

In fact, this is the parity-check matrix of the LDPC code proposed by Gallager in [2].

An LDPC code is regular if J is constant for every column and K is also constant for every row. On the other hand, if $\boldsymbol{H}$ is low in density but J and

K are not constant, the code is irregular. The example Gallager code above is obviously a regular LDPC code since it contains three 1's per column and four 1's per row for all columns and rows. An (*n*,*k*) regular LDPC code sometimes may be denoted as *C*(*n*,*J*,*K*), where *n* is the code length. For instance, the example code is denoted by *C*(20,3,4). Also for a regular LDPC code, the coding rate is calculated as follows provided all rows are linearly independent[6]:

$$R = 1 - J/K \tag{6.45}$$

The equation is easy to derive. The total number of 1's counted row by row, *mK*, must equal that counted column by column, *nJ*, that is, *mK* = *nJ*. Substituting this expression into the definition of the coding rate, $R \triangleq k/n = (n-m)/n$, yields (6.45).

In addition to near-capacity performance, LDPC codes possess other advantages over turbo codes as summarized in the following [21]:

1. Suitable for parallel implementation;
2. More amenable to high coding rate;
3. Lower error floor;
4. Superior burst error correcting capability;
5. One single LDPC code can be good for different channels.

Among the disadvantages are:

1. High encoder complexity;
2. Interconnect in decoder is large and irregular;
3. May perform worse than turbo codes when code length is short.

6.2.1.2 Tanner Graph

A *Tanner graph* is a bipartite graph introduced to graphically represent LDPC codes [22]. It consists of nodes and edges. The nodes are grouped into two sets. One set consists of *n* bit nodes (or variable nodes), and the other of *m* check nodes (or parity nodes). The creation of such a graph is straightforward: Check node *i* is connected to bit node *j* if $h_{i,j}$ of the parity matrix $\boldsymbol{H}$ is a 1. From this we can deduce that there are totally *mK* (or *nJ*) edges in a Tanner graph for a regular code. Apparently the Tanner graph has a one-to-one correspondence with the parity-check matrix. The Tanner graph of the *C*(20,3,4) Gallager code is shown Figure 6.25.

[6] The actual coding rate may be somewhat higher.

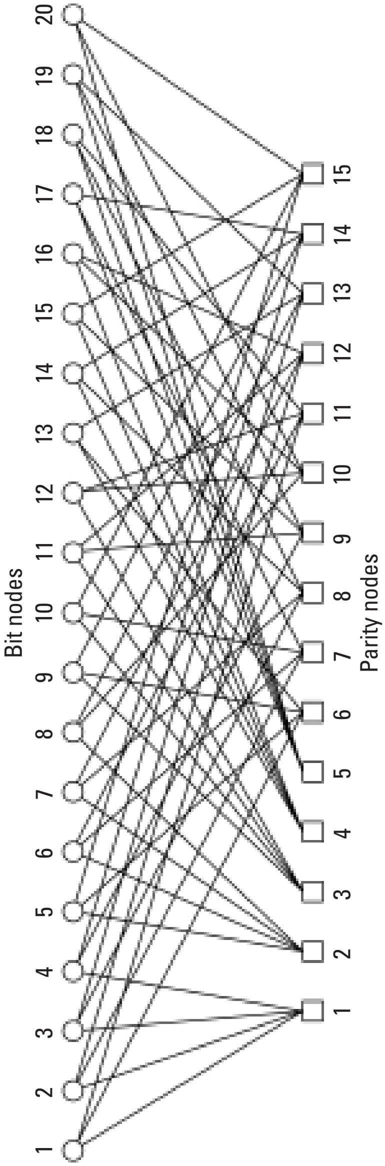


Figure 6.25 Tanner graph of (20,3,4) Gallager code.

As we shall see shortly, decoding of LDPC codes operates by iteratively passing messages between the two sets of nodes in the Tanner graph.

Certain terminology is associated with a Tanner graph. The *degree* of a node is defined as the number of edges connecting to it. A path comprising l edges in a Tanner graph that closes back on itself is called a *cycle* of length l. The minimum cycle length in a Tanner graph is referred to as the *girth* of the graph. The shortest possible cycle in a Tanner graph is a length-4 cycle (see Figure 6.26).

6.2.1.3 Criteria for Good LDPC Codes

A good LDPC code should possess a large minimum distance $d_{\min}$ and no short cycles in its Tanner graph. The first requirement is obvious as LDPC is a block code. The second one comes from the fact that cycles hurt the performance of the message-passing decoding algorithm because they invalidate the assumption of independence under which the algorithm is derived.

The requirements impose some constraints on the parity-check matrix as follows:

1. The matrix is low in both column weight and row weight; that is, $J \ll n$ and $K \ll m$.
2. The overlapping of 1's per column and per row should be at most equal to one. This is necessary to avoid short cycles in the Tanner graph.
3. $J \geq 3$ and $K > J$ to ensure good distance property.

6.2.1.4 Notes on LDPC Performance

Readers should be aware that the near Shannon performance of LDPC codes exists only for long code lengths (at least a few thousands of bits). The minimum distance of an LDPC code increases with increasing code length; at the same time, the error probability of the code decreases exponentially.

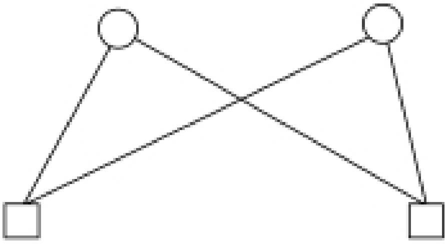


Figure 6.26 Shortest possible cycle length in Tanner graph.

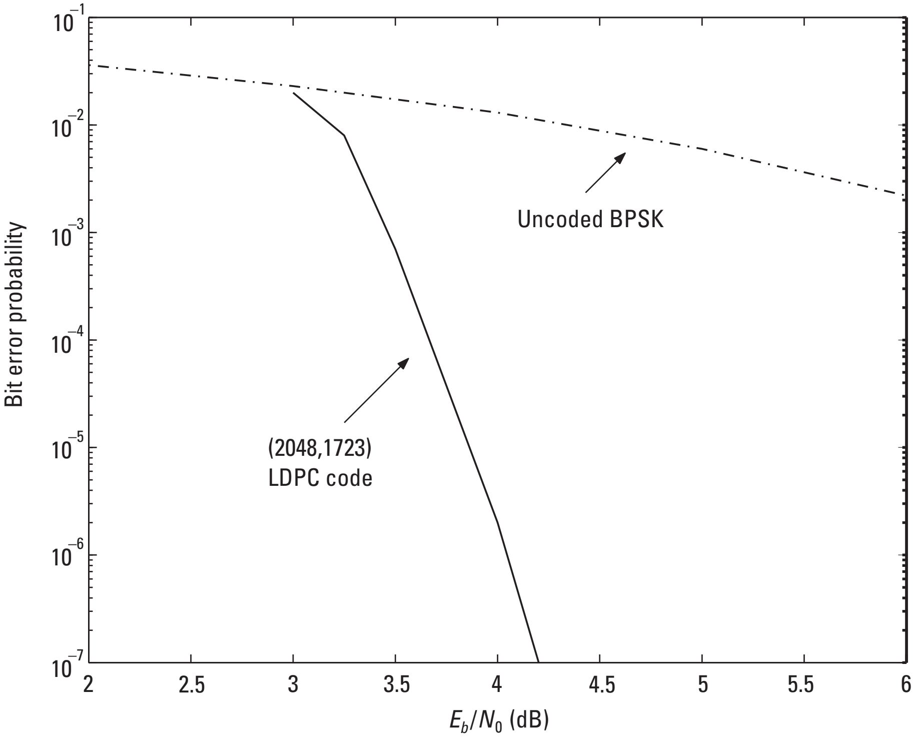


Figure 6.27 Error performance of (2048,1723) LDPC code. (*After:* [24].)

Figure 6.27 plots the BER versus E_b/N_0 curve for the (2048,1723) LDPC code. It has also been found that, in general, the BER performances of irregular LDPC codes are better than those of regular LDPC codes by up to 0.5 dB [23].

MATLAB Experiment 6.12

Consider a (12,3,6) LDPC code whose parity-check matrix is as follows:

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

The m-file `ldpcsim.m*` simulates the code.

Comment: The error probability is understandably not as good as Figure 6.27, because this code is obviously too short.

6.2.2 Decoding and Encoding Algorithms

6.2.2.1 General Description

For all of the codes we have discussed so far, their encoding is far less complex than their decoding and has never been an issue. However, the situation is different with LDPC codes. LDPC encoding poses a challenge. As we have mentioned, the generator matrix of an LDPC code is usually not sparse. Due to large size of the matrix, the conventional block encoding method (i.e., multiply message by generator matrix) could require a significant number of computations. For instance, the generator for a rate-1/2 LDPC code with a codeword length of 10,000 and message length of 5,000 is a 5,000×10,000 matrix. Brute-force multiplication of the matrix with the message requires 10^7 XORs even if we assume the density of the matrix to be as low as 0.2. Therefore, an efficient encoding method has been a topic of LDPC code research.

Like turbo decoding, LDPC decoding adopts an iterative approach. The decoding operates alternatively on the bit nodes and the check nodes to find the most likely codeword $\boldsymbol{c}$ that satisfies the condition $\boldsymbol{cH}^{\mathrm{T}} = \mathbf{0}$. Several decoding algorithms exist for LDPC codes. For hard-decision decoding, there is the bit-flipping (BF) algorithm; for soft-decision decoding, there is the sum-product algorithm (SPA), also known as the brief propagation algorithm and Pearl's algorithm. The iterative soft-decision decoding of LDPC code converges to true optimum decoding if the corresponding Tanner graph contains no cycles. Therefore, we want LDPC codes with as few cycles as possible, especially short cycles.

In the following sections, we present the decoding algorithms, followed by the encoding methods.

6.2.2.2 BF Decoding

Gallager [2] proposed the following BF decoding algorithm:

Bit-Flipping Decoding Algorithm

1. Compute each parity check for the received word $\boldsymbol{r}$.
2. For each received bit, count the number of failed parity checks.

3. Flip the bit(s) with the largest number of failed check(s).
4. Repeat steps 1, 2, and 3 until all parity checks are satisfied or until the predetermined number of iterations is reached. In the latter case, if some parity checks still fail, declare decoding failure.

The BF decoding algorithm is attractive for its simplicity (requires only XOR gates and comparators). For fixed values of J and K, the decoding complexity grows linearly with the code length n. To enhance the performance of the algorithm, several variants such as weighted bit flipping (WBF) have been proposed [25].

Example 6.7

Consider the following code[7]*:*

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

The Tanner graph of the code is drawn in Figure 6.28.

Suppose that the codeword is an all-zero vector $\boldsymbol{c} = (0000000)$*, and the received word contains two errors in the second and fourth bits such that* $\boldsymbol{r} = (0101000)$*. The BF decoding process is shown in Figure 6.29, where × denotes a failed parity check and √ a successful parity check. We see that, at the third iteration, the received word is correctly decoded.*

MATLAB Experiment 6.13

The MATLAB program `bf.m*` simulates the BF algorithm. Readers are advised to step through the program for one iteration to see how the BF algorithm is implemented.

[7] This is not quite an LDPC code; we use it merely to illustrate the BF algorithm.

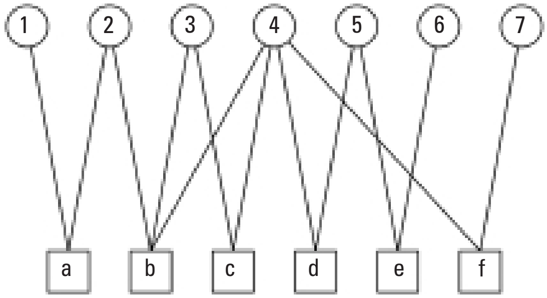


Figure 6.28 Example Tanner graph.

6.2.2.3 Sum-Product Decoding Algorithm

Instead of flipping bits, the sum-product algorithm (SPA) propagates soft probabilities of the bits between bit nodes and check nodes through the Tanner graph, thereby refining the confidence that the parity checks provide about the bits. The exchange of the soft probabilities is termed as *message passing* or *belief propagation*. When no cycles exist in the Tanner graph, SPA computes the exact probabilities [26]; when cycles are present, it computes only approximate solutions. However, even with cycles, the algorithm, given next, can still decode very effectively.

Sum-Product Decoding Algorithm

Initialization:

For bit node j with an edge to check node i:

Set:

$$\begin{cases} p_j^1 = 1/[1+\exp(2r_j/\sigma^2)] = 1/[1+\exp(4r_j/N_0)] \\ p_j^0 = 1 - p_j^1 \end{cases}$$

and

$$\begin{cases} P_{ij}^1 = P(c_j = 1 \mid r_j) = p_j^1 \\ P_{ij}^0 = 1 - P_{ij}^1 \end{cases} \tag{6.46}$$

where r_j is the received bit corrupted by noise, and $\sigma^2 = N_0/2$ is the variance of the AWGN channel.

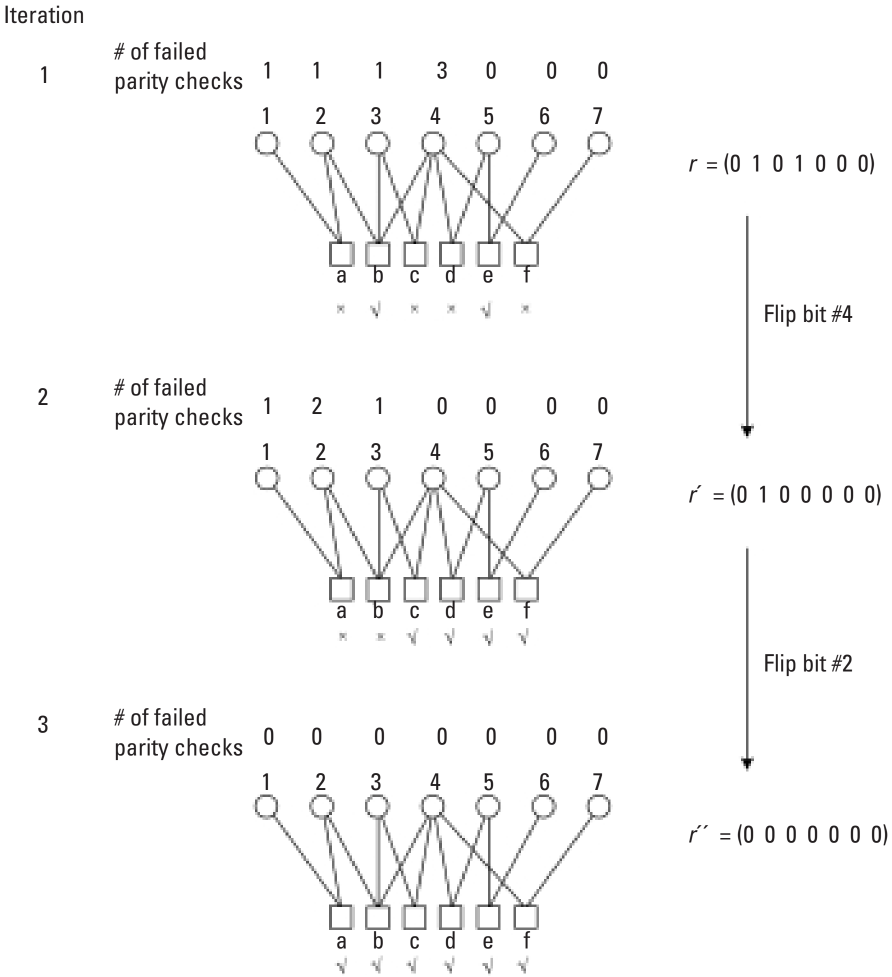


Figure 6.29 BF decoding process.

Message Passing:

1. *From check nodes to bit nodes:* For each check node i with an edge to bit node j, compute

$$\Delta P_{ij} = P_{ij}^{0} - P_{ij}^{1} \tag{6.47}$$

Compute ΔQ_{ij} as the product of $\Delta P_{ij'}$ for all $j' \neq j$, that is:

$$\Delta Q_{ij} = \prod_{j'} \Delta P_{ij'} \qquad (j' = 1,2,\cdots,n \text{ and } j' \neq j) \tag{6.48}$$

Also, compute:

$$\begin{cases} Q_{ij}^{0} = \dfrac{1}{2}(1+\Delta Q_{ij}) \\ Q_{ij}^{1} = \dfrac{1}{2}(1-\Delta Q_{ij}) \end{cases} \qquad (j' = 1,2,\cdots,n \text{ and } j' \neq j) \tag{6.49}$$

2. *From bit nodes to check nodes:* For each bit node j with an edge to check node i:

   Compute P_{ij}^{0} as p_{j}^{0} multiplied by the product of $Q_{i'j}^{0}$ and P_{ij}^{1} as p_{j}^{1} multiplied by the product of $Q_{i'j}^{1}$ over all $i' \neq i$ that is:

$$\begin{cases} P_{ij}^{0} = p_{j}^{0} \prod_{i'} Q_{i'j}^{0} \\ P_{ij}^{1} = p_{j}^{1} \prod_{i'} Q_{i'j}^{1} \end{cases} \qquad (i' = 1,2,\cdots,m \text{ and } i' \neq i) \tag{6.50}$$

   and scale P_{ij}^{0} and P_{ij}^{1} by a same factor so that $P_{ij}^{0} + P_{ij}^{1} = 1$.

   Compute P_{ij}^{0} as p_{j}^{0} times the product of $Q_{i'j}^{0}$ and P_{ij}^{1} as p_{j}^{1} multiplied by the product of $Q_{i'j}^{0}$ over all i, that is:

$$\begin{cases} P_{j}^{0} = p_{j}^{0} \prod_{i} Q_{ij}^{0} \\ P_{j}^{1} = p_{j}^{1} \prod_{i} Q_{ij}^{1} \end{cases} \qquad (i = 1,2,\cdots,m) \tag{6.51}$$

   and scale P_{j}^{0} and P_{j}^{1} by a same factor so that $P_{j}^{0} + P_{j}^{1} = 1$.

3. *Decoding and soft outputs:* For $j = 1,2, \ldots, n$:

$$c_j = \begin{cases} 0 & \text{if} \ln\left(P_{j}^{1} / P_{j}^{0}\right) \geq 0 \\ 1 & \text{otherwise} \end{cases} \tag{6.52}$$

   If $\boldsymbol{cH^T} = 0$, stop and output hard decision $\boldsymbol{c}$ and/or soft likelihood $\ln(P_{j}^{1}/P_{j}^{0})$. Otherwise, go to step 1. In the latter case, if the iterations exceed a preset number, declare a decoding failure.

Notice that the sum-product algorithm performs in the same way as the MAP BCJR algorithm in that it gives the best possible estimate of each bit of the received vector, but not necessarily the best estimate of the whole codeword.

Example 6.8

Consider the (12,3,6) LDPC code in MATLAB Experiment 6.12. Its Tanner graph is drawn in Figure 6.30.

Suppose the codeword $\mathbf{c}$ *= (101000101000) is transmitted through an AWGN channel with* $\sigma^2 = 1$*. The received vector is* $\mathbf{r}$ *= (–0.40 0.80 –0.20 1.10 1.20 –0.20 –1.30 0.70 0.07 1.10 1.30 1.10). Using (15.5), we find the following:*

$P_j^1=$

0.69	0.17	0.60	0.10	0.08	0.60	0.93	0.20	0.47	0.10	0.07	0.10

If we decode $\mathbf{r}$ *simply by thresholding the above probabilities at 0.5, the result is (101001100000), which contains two errors.*

Applying the sum-product algorithm, we have the following decoding process:

Iteration 0 (initialization):

$P_{ij}^1 =$

0.69	0.17		0.10					0.47	0.10		0.10
0.69	0.17	0.60	0.10	0.08			0.20				
0.69		0.60		0.08	0.60				0.10	0.07	
			0.10	0.08	0.60	0.93		0.47		0.07	
	0.17	0.60				0.93	0.20			0.07	0.10
					0.60	0.93	0.20	0.47	0.10		0.10

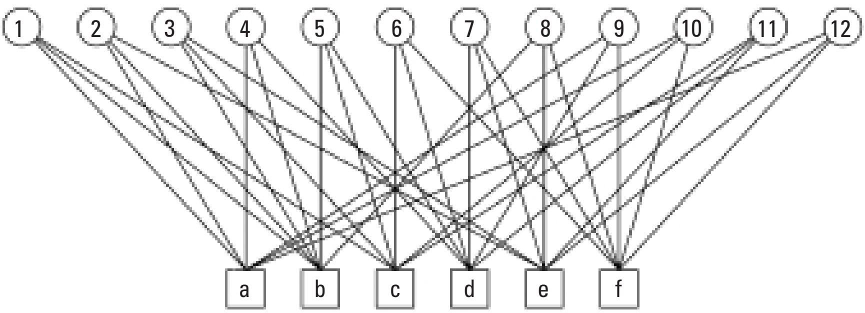


Figure 6.30 Tanner graph of (12,3,6) LPDC code.

$P^0_{ij} =$

0.31	0.83		0.90					0.53	0.90		0.90
0.31	0.83	0.40	0.90	0.92			0.80				
0.31		0.40		0.92	0.40				0.90	0.93	
			0.90	0.92	0.40	0.07		0.53		0.93	
	0.83	0.40				0.07	0.80			0.93	0.90
					0.40	0.07	0.80	0.53	0.90		0.90

Iteration 1:

$\Delta P_{ij} =$

–0.38	0.66		0.80					0.07	0.80		0.80
–0.38	0.66	–0.20	0.80	0.83			0.60				
–0.38		–0.20		0.83	–0.20				0.80	0.86	
			0.80	0.83	–0.20	–0.86		0.07		0.86	
	0.66	–0.20				–0.86	0.60			0.86	0.80
					–0.20	–0.86	0.60	0.07	0.80		0.80

$\Delta Q_{ij} =$

0.02	–0.01		–0.01					–0.13	–0.01		–0.01
–0.06	0.03	–0.10	0.03	0.02			0.03				
0.02		0.04		–0.01	0.04				–0.01	–0.01	
			0.01	0.01	–0.03	–0.01		0.10		0.01	
	0.07	–0.24				–0.05	0.08			0.05	0.06
					–0.02	–0.01	0.01	0.10	0.01		0.01

$Q^0_{ij} =$

0.51	0.49		0.49					0.44	0.49		0.49
0.47	0.52	0.45	0.51	0.51			0.52				
0.51		0.52		0.49	0.52				0.49	0.50	
			0.50	0.50	0.48	0.50		0.55		0.50	
	0.54	0.38				0.47	0.54			0.53	0.53
					0.49	0.50	0.50	0.53	0.50		0.50

$Q_{ij}^{1} =$

0.49	0.51		0.51					0.56	0.51		0.51
0.47	0.48	0.55	0.49	0.49			0.49				
0.49		0.48		0.51	0.48				0.51	0.51	
			0.50	0.50	0.52	0.51		0.45		0.50	
	0.46	0.62				0.53	0.46			0.47	0.47
					0.51	0.50	0.50	0.47	0.50		0.50

$p_{j}^{0} =$

0.31	0.86	0.27	0.90	0.92	0.39	0.06	0.84	0.55	0.90	0.94	0.91

$p_{j}^{1} =$

0.69	0.14	0.73	0.10	0.08	0.61	0.94	0.16	0.45	0.10	0.06	0.09

$P_{ij}^{1} =$

0.70	0.14		0.09					0.39	0.10		0.09
0.67	0.15	0.69	0.10	0.08			0.17				
0.70		0.75		0.08	0.63				0.10	0.06	
			0.10	0.08	0.59	0.94		0.48		0.06	
	0.16	0.63				0.93	0.19			0.07	0.10
					0.59	0.94	0.16	0.48	0.10		0.09

$P_{ij}^{0} =$

0.30	0.86		0.91					0.62	0.90		0.91
0.331	0.85	0.31	0.90	0.92			0.83				
0.30		0.25		0.92	0.37				0.90	0.94	
			0.90	0.92	0.41	0.06		0.50		0.94	
	0.84	0.37				0.07	0.81			0.94	0.90
					0.41	0.06	0.84	0.52	0.90		0.91

$\ln(P_{j}^{1}/P_{j}^{0}) =$

–0.81	1.78	–1.00	2.24	2.44	–0.43	–2.74	1.64	0.21	2.17	2.71	2.31

At the end of the first iteration, the decoded word is (101001100000), which still has two errors. This means we need more decoding iterations.

Finally, at the eighth iteration, we obtain:

$\ln(P_j^1/P_j^0) =$

–0.88	1.84	–1.03	2.24	2.44	0.01	–2.74	1.70	–0.14	2.11	2.70	2.31

The received word is correctly decoded as (101000101000).

MATLAB Experiment 6.14

The MATLAB program `spademo.m*` simulates Example 6.8. Checking the probabilities, we notice that their magnitudes vary greatly.

```
>> spademo
LLR =
 Columns 1 through 8
 -0.8752 1.8355 -1.0286 2.2377 2.4431 0.0095 -2.7386 1.7042
 Columns 9 through 12
 -0.1404 2.1054 2.7035 2.3083
```

6.2.2.4 Sum-Product Algorithm in Log Domain

Several drawbacks are associated with SPA. First, as we saw in the previous experiment, the numerical dynamic range in SPA computations is quite large. This could potentially result in numerical instability. Second, intensive multiplication in the algorithm poses a challenge for implementation. As with MAP, we want to modify the algorithm so that we can use the log-likelihood ratio (LLR) instead of probability [21].

Define the following LLRs:

$$L(c_i) \triangleq \ln[P(c_i = 0 \mid r_i)/P(c_i = 1 \mid r_i)]$$

$$L(P_{ij}) \triangleq \ln\left(P_{ij}^0/P_{ij}^1\right)$$

$$L(Q_{ij}) \triangleq \ln\left(Q_{ij}^0/Q_{ij}^1\right)$$

$$L(P_j) \triangleq \ln\left(P_j^0/P_j^1\right)$$

where ln(·) represents the natural logarithm operation. The log domain SPA can be described as follows:

Sum-Product Algorithm in Log Domain

Initialization:

For bit node j with an edge to check node i:

Set:

$$L(P_{ij}) = L(c_i) = 2r_i/\sigma^2 \tag{6.53}$$

Message Passing:

1. *From check nodes to bit nodes:* For each check node i with an edge to bit node j:

   Update $L(Q_{ij})$ as:

   $$L(Q_{ij}) = \prod_{j'} \alpha_{ij'} \phi\left[\sum_{j'} \phi(\beta_{ij'})\right] \qquad (j' = 1, 2, \cdots, n \text{ and } j' \neq j) \tag{6.54}$$

   where $\alpha_{ij} \triangleq \text{sign}[L(P_{ij})]$ and $\beta_{ij} \triangleq \left|L(P_{ij})\right|$. The ϕ function is defined as:

   $$\phi(x) = -\ln[\tanh(x/2)] = \ln[(e^x + 1)/(e^x - 1)] \tag{6.55}$$

2. *From bit nodes to check nodes:* For each bit node j with an edge to check node i:

   Update $L(P_j)$ as:

   $$L(P_{ij}) = L(c_j) + \sum_{i'} L(Q_{ij}) \qquad (i' = 1, 2, \cdots, m \text{ and } i' \neq i) \tag{6.56}$$

3. *Decoding and soft outputs:* For $j = 1,2,\ldots, n$:

   Update $L(P_j)$ as:

   $$L(P_j) = L(c_j) + \sum_{i} L(P_{ij}) \qquad (i = 1, 2, \cdots, m) \tag{6.57}$$

   Let:

   $$c_j = \begin{cases} 1 & \text{if } L(P_j) < 0 \\ 0 & \text{else} \end{cases} \tag{6.58}$$

If $\boldsymbol{c} \cdot \boldsymbol{H}^{\mathrm{T}} = \mathbf{0}$ or the number of iterations reaches the maximum limit, stop; otherwise, go to step 2.

The most frequently involved computation in log domain SPA is computation of the ϕ function. The function is fairly well behaved, as illustrated in Figure 6.31, and it may be implemented with an LUT.

MATLAB Experiment 6.15

The MATLAB program `logspa.m*` for decoding the previous example using log-SPA is included on this book's DVD. Experiment with the program to gain more insight into the algorithm.

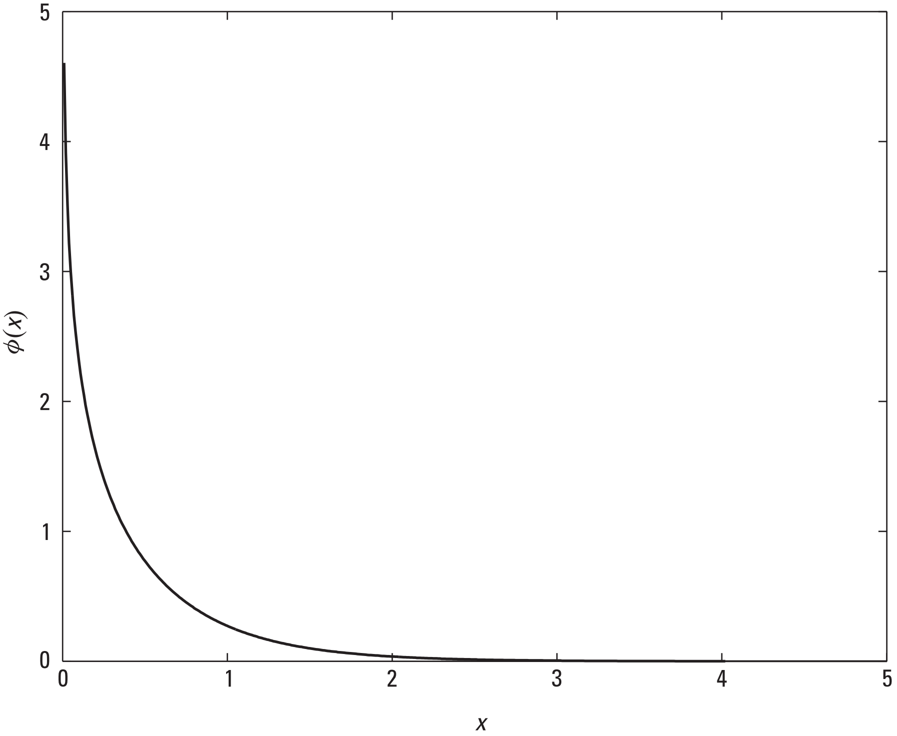


Figure 6.31 The $\phi(x)$ function.

6.2.2.5 Encoding

LDPC encoding can be accomplished using the conventional method presented in Chapter 3: First reformulate the $(n - k) \times n$ parity-check matrix in systematic form: $\boldsymbol{H} = [\boldsymbol{I}_{n-k} | \ \boldsymbol{P}]$ [see (3.8) in Chapter 3], then construct the generator matrix as $\boldsymbol{G} = [\boldsymbol{P}_{k \times (n-k)} | \ \boldsymbol{I}_k]$, and finally encode as $\boldsymbol{c} = \boldsymbol{mG}$. The construction of the generator can be done off-line and does not count toward the encoder's complexity. However, multiplication of $\boldsymbol{m}$ with $\boldsymbol{G}$ has a complexity of $\sim k \times n = Rn \times n \sim n^2$, where R is the coding rate. In other words, the encoder complexity is quadratic in the block length.

Some novel ideas for lower complexity LDPC encoding have been proposed that exploit the sparseness of the parity-check matrix. One proposal is the message-passing encoder [26, 27]. Notice that, for systematic code, encoding is nothing more than finding $n - k$ parity bits. The method treats the codeword with $n - k$ unknown parity bits as if it were a received word with $n - k$ erasures, and uses a decoder to find the erasures. Once the erasures are known, the parity bits are obtained. Hence, encoding is converted to decoding for a binary erasure channel.

Another encoding method proposed in [28] is to transform the parity-check matrix into an approximate lower triangular form by row and column permutations, so that the sparseness of the matrix is preserved. The encoding complexity is on the order of magnitude $n + g^2$, where g is called the "gap" to linear encoding. This gap is actually the number of rows of the parity-check matrix that cannot be brought into triangular form by row and column permutations only.

6.2.3 High-Level Architecture Design for LDPC Decoders

6.2.3.1 Parallel Architecture

We observe that, in the SPA decoding algorithm, computation of the quantity Q_{ij}^1 (or P_{ij}^1) for a node j (or i) does not count on the other nodes. In other words, computation of Q_{ij}^1 (or P_{ij}^1) can be performed independently for all bit nodes (or check nodes). Exploiting this feature, the parallel architecture design directly maps the nodes of the Tanner graph onto processing elements (PE), and the edges of the graph onto a network of interconnects (see Figure 6.32). Each PE corresponding to a check node (we call it CPE) executes step 1 in the message passing of the SPA; each PE corresponding to a bit node (we call it BPE) executes step 2 in the message passing.

The advantage of a fully parallel design is its ability to achieve the highest throughput and no need for large memory to store intermediate messages. However, the architecture (Figure 6.32) suffers from large hardware area and complex interconnection between the PEs (a hard-wired connection is required between every pair of BPEs and CPEs). Also, this irregular interconnection makes it difficult to partition a parallel design into smaller subblocks for repetition.

6.2.3.2 Serial Architecture

The high usage of hardware in the parallel structure can be brought down by using the resource sharing technique. This then leads to a serial design for the decoder (Figure 6.33). Two memory devices are needed in this case to store the messages, one for Q_{ij}^{1} and the other for P_{ij}^{1}.

The serial architecture results in a smaller area and significantly simpler interconnection. It is also more flexible in supporting different parity-check matrices. The main drawbacks include much lower throughput and the need for memory. The control mechanism also becomes rather sophisticated, as compared with a parallel architecture.

6.2.3.3 A Few Words on Architecture Optimization

Besides hardware area and throughput, architecture optimization for LDPC decoders also emphasizes reconfigurability, low-power consumption, and efficient use of memory.

Some cited works on architecture optimization for LDPC decoder design include [29], which focuses on reconfigurable hardware; [30], which reports on an architecture that achieves low power by inducing structural regularity into the decoder; and [31], which covers a high-throughput, memory-efficient parallel architecture.

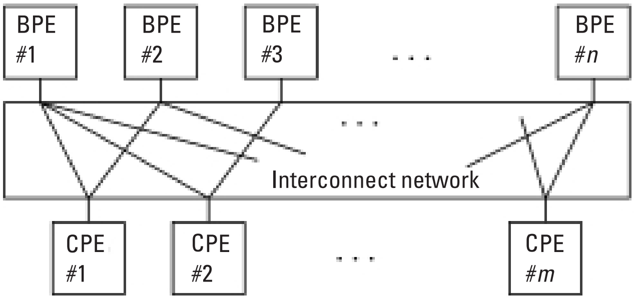


Figure 6.32 Parallel architecture of LDPC decoder.

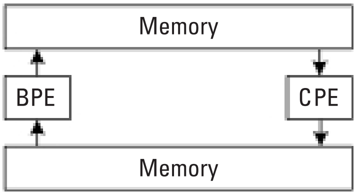


Figure 6.33 Serial architecture of LDPC decoder.

Note Two most important classes of modern error correcting codes are introduced (albeit at a somewhat superficial level). The codes are still under intensive research. We recommend the *IEEE Transactions on Information Theory* journal and other related journals and conferences as sources for the most up-to-date coverage. This chapter serves only as a starting point.

Although the proof of the iterative BCJR decoding is omitted, it is beneficial to go through the mathematical derivations. Reference [16] provides a concise treatment of this. The explanation of turbo code performance including the error floor is quite brief in this book. To gain an insightful understanding of this topic, we suggest Section 16.3 in [8] and Sections 8.6 and 8.7 in [32] for further reading.

For LDPC codes, several other decoding algorithms are worth our attention, such as the min-sum and min-sum-plus-correction-factor algorithms. They are approximations to SPA. These algorithms simplify the decoder design and hence are attractive in practice. Interested readers may want to access [33, 34] for details.

This chapter concludes the book. However, the research on error control coding continues. Breakthroughs are expected at any time. You never know.

Problems

6.1 Use a MATLAB simulation to confirm that SOVA is inferior to MAP decoding in terms of bit error performance, and give the reason why.

6.2 Consider a rate-1/2 turbo code punctured from the rate-1/3 code in Example 6.1. The puncturing matrix is $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. Demonstrate the decoding process of the code.

6.3 Fill the missing decoding iterations in Example 6.5 and compare the results with MATLAB Experiment 6.8.

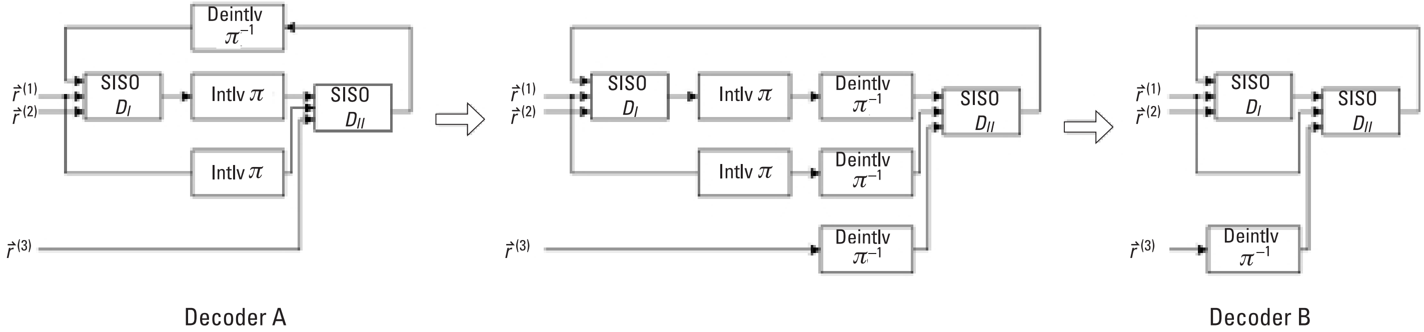


Figure 6.34 Transformation from decoder A to decoder B.

6.4 From the system point of view, decoder B can be obtained from decoder A by block diagram reduction and they do the same thing (as shown in Figure 6.34). Can we use decoder B since it has saved two interleavers [35]? Why or why not?

6.5 Examine the following parity-check matrix and determine if it is a good LDPC code. What is the coding rate of this code?

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

6.6 Derive the generator matrix of the (20,3,4) LDPC code for encoding use.

6.7 Modify SPA for hard-decision decoding. In this case, the message from the check node to the bit node would be the bit that the check node believes to be correct assuming that the bits from the other bit nodes are correct. The message from the bit node to the check node would be the bit that wins the majority vote.

6.8 Fill in the missing decoding iterations in Example 6.8 and check the results with MATLAB Experiment 6.14.

References

[1] Berrou, C., A. Glavieux, and P. Thitimajshima, "Near Shannon Limit Error-Correcting Coding and Decoding: Turbo Codes," *Proc. IEEE Intl. Conf. Commun.*, Geneva, Switzerland, 1993, pp. 1064–1070.

[2] Gallager, R. G., *Low-Density Parity-Check Codes*, Cambridge, MA: The MIT Press, 1963.

[3] Gallager, R. G., "Low-Density Parity-Check Codes," *IRE Trans. Info. Theory*, Vol. IT-8, January 1962, pp. 21–28.

[4] MacKay, D. J., and R. M. Neal, "Near Shannon Limit Performance of Low Density Parity Check Codes," *Electron. Letter*, Vol. 32, No. 18, 1996, pp. 1645–1646.

[5] MacKay, D. J., "Good Error-Correcting Codes Based on Very Sparse Matrices," *IEEE Trans. Info. Theory*, Vol. 45, No. 2, March 1999, pp. 399–431.

[6] Chung, S.-Y., et al., "On the Design of Low-Density Parity-Check Codes Within 0.0045 dB of the Shannon Limit," *IEEE Commun. Letter*, Vol. 5, No. 2, February 2001, pp. 58–60.

[7] Forney, G. D., Jr., *Concatenated Codes,* Cambridge, MA: The MIT Press, 1966.

[8] Lin, S., and D. J. Castello, *Error Control Coding—Fundamentals and Applications*, 2nd ed., Upper Saddle River, NJ: Prentice-Hall, 2004.

[9] Barbelescu, A., and S. Pietrobon, "Interleaver Design for Turbo Codes," *Electron. Letter*, Vol. 30, No. 25, December 1994, pp. 2107–2108.

[10] Dinoi, L., and S. Benedetto, "Design of Fast Prunable S-Random Interleavers," *IEEE Trans. Wireless Commun.*, Vol. 4, No. 5, May 2005, pp. 1–9.

[11] Popovski, P., L. Kocarev, and A. Risreski, "Design of Flexible-Length S-Random Interleaver for Turbo Codes," *IEEE Commun. Letter*, Vol. 8, No. 7, July 2004, pp. 461–463.

[12] Vucetic, B., and J.-H. Yuan, *Turbo Codes*: *Principle and Applications*, Norwell, MA: Kluwer Academic, 2000.

[13] Barbulescu, A., and S. Pietrobon, "Terminating the Trellis of Turbo-Codes in the Same State," *Electron. Letter*, Vol. 31, No. 1, January 1995, pp. 22–23.

[14] Sun, J., and O. Y. Takeshita, "Interleavers for Turbo Codes Using Permutation Polynomial over Integer Rings," *IEEE Trans. Inform. Theory*, Vol. 51, No. 1, January 2005, pp. 101–119.

[15] Moon, T. K., *Error Control Coding—Mathematical Methods and Algorithms*, New York: John Wiley & Sons, 2005.

[16] Ryan, W. E., "A Turbo Code Tutorial," http://citeseer.ist.psu.edu/334930.html.

[17] Viterbi, A. J., "An Intuitive Justification and a Simplified Implementation of the MAP Decoder for Convolutional Codes," *IEEE J. Selected Areas Commun.*, Vol. 16, No. 2, February 1998, pp. 260–264.

[18] Robertson, P., E. Villebrun, and P. Hoher, "A Comparison of Optimal and Sub-Optimal *MAP* Decoding Algorithms Operating in the Log Domain," *Proc. Intl. Conf. Commun.*, Seattle, WA, 1995, pp. 1009–1013.

[19] Hagenauer, J., and P. Hoeher, "A Viterbi Algorithm with Soft-Decision Outputs and Its Application," *Proc. IEEE GlobeCom Conf.*, 1989, pp. 1680–1686.

[20] Shao, R. Y., S. Lin, and M. P. Fossorier, "Two Simple Stopping Criteria for Turbo Decoding," *IEEE Trans. Comm.*, Vol. 47, No. 8, August 1999, pp. 1117–1120.

[21] Ryan, W. E., "An Introduction to LDPC Codes," in *Handbook for Coding and Signal Processing for Recoding Systems*, B. Vasic, (ed.), Boca Raton, FL: CRC Press, 2004.

[22] R. Tanner, "A Recursive Approach to Low Complexity Codes," *IEEE Trans. Info. Theory,* Vol. 27, No. 5, September 1981, pp. 533–547.

[23] Luby, M. G., et al., "Improved Low-Density Parity-Check Codes Using Irregular Graphs," *IEEE Trans. Inform. Theory*, Vol. 47, No. 2, February 2001, pp. 585–598.

[24] Djurdjevic, I., et al., "A Class of Low-Density Parity-Check Codes Constructed Based on Reed-Solomon Codes with Two Information Symbols," *IEEE Commun. Lett.*, Vol. 7, No. 7, July 2003, pp. 317–319.

[25] Kou, Y., S. Lin, and M. P. Fossorier, "Low-Density Parity-Check Codes Based on Finite Geometries: A Rediscovery and New Results," *IEEE Trans. Info. Theory,* Vol. 47, No. 7, November 2001, pp. 2711–2736.

[26] Luby, M., et al., "Practical Erasure Resilient Codes," *Proc. 29th Ann. ACM Symp. Theory of Computing*, 1997, pp. 150–159.

[27] Tanner, R. M., et al., "LDPC Block and Convolutional Codes Based on Circulant Matrices," *IEEE Trans. Inform. Theory*, Vol. 50, No. 12, December 2004, pp. 2966–2984.

[28] Richardson, T. J., and R. L. Urbanke, "Efficient Encoding of Low-Density Parity Check Codes," *IEEE Trans. Inform. Theory*, Vol. 47, No. 2, February 2001, pp. 638–656.

[29] Mansour, M. M., and N. R. Shanbhag, "A 640-Mb/s 2048-Bit Programmable LDPC Decoder Chip," *IEEE J. Solid-State Circuits*, Vol. 41, No. 3, March 2006, pp. 684–698.

[30] Lee, J. K.-S., et al., "A Scalable Architecture of a Structured LDPC Decoder," *Proc. Intl. Symp. Inform. Theory*, 2004, p. 292.

[31] Blanksby, A., and C. J. Howland, "A 690mW 1-Gbit/s 1024-b Rate-1/2 Low-Density Parity-Check Code Decoder," *IEEE J. Solid-State Circuits*, Vol. 37, No. 3, March 2002, pp. 404–412.

[32] Schlegel, C., and L. C. Perez, *Trellis Coding*, Piscataway, NJ: IEEE Press, 1997.

[33] Wiberg, N., "Codes and Decoding on General Graphs," Ph.D. Dissertation, University of Linköping, Sweden, 1996.

[34] Hu, X-Y., et al., "Efficient Implementation of the Sum-Product Algorithm for Decoding LPDC Codes," *Proc. IEEE GlobeCom Conf.*, November 2001, pp. 1036–1036E.

[35] Sklar, B., *Digital Communications—Fundamentals and Applications*, 2nd ed., Upper Saddle River, NJ: Prentice-Hall, 2001.

Selected Bibliography

Boutillon, E., W. J. Gross, and P. G. Gluak, "VLSI Architectures for the MAP Algorithm," *IEEE Trans. Commun.*, Vol. 51, No. 2, February 2003, pp. 175–185.

Pearl, J., *Probabilistic Reasoning in Intelligent Systems*, San Mateo, CA: Morgan Kaufmann, 1988.

Vecetic, B., et al., "Recent Advances in Turbo Code Design and Theory," *IEEE Proc.,* Vol. 95, No. 6, June 2007, pp. 1323–1344.

About the Author

Yuan Jiang is an independent consultant in digital communications and signal processing. During the past 15 years, he has worked for many leading high-tech companies including Nortel Networks, Lucent Technologies, and Cadence Design Systems, where he played key roles in various cutting-edge R&D projects. He also cofounded a start-up specializing in DSP-based intelligent sensors. Mr. Jiang is an inventor with three patents and the author of a dozen technical papers. He has served as an anonymous paper reviewer for the IEEE journal *Communications Letters* and *IEEE Transactions on Vehicular Technology.* Recently, he supervised several master's theses. Mr. Jiang received a B.S. from Zhejiang University, China, and an M.S. from the State University of New York at Buffalo. He was a Ph.D. candidate at SUNY Buffalo.

Index

Add-compare-select (ACS), 170, 197
Additive white Gaussian noise (AWGN) channel, 8
A posteriori probability (APP), 10
A priori probability, 238
Asymptotic coding gain. *See* Coding gain
Automatic repeat request (ARQ), 2

Bahl-Cocke-Jelinek-Raviv (BCJR) algorithm, 229
BCH bound, 100–101
BCH code
  Binary, 97
  Nonbinary, 113–14
Belief-propagation decoding. *See* Sum-product algorithm
Berlekamp-Massey (BM) algorithm, 125
  inversionless BM algorithm (iBM), 129
  reformatted inversionless BM (riBM) architecture, 144
Binary phase shift keying (BPSK), 6
Binary symmetric channel (BSC), 8
Bit-flipping (BF) decoding algorithm, 259–60
Block code, 5
  construction, 45
Bound
  BCH bound, 100
  Rieger bound, 123
  Singleton bound, 122
  union bound, 13
  Viterbi decoding, 177
Branch metric, 168
Butterfly structure, 159

Catastrophic error propagation. *See* Convolutional code
Channel capacity, 14
Channel error
  bursty, 8
  random, 8
Chien search, 108
Coding gain, 13
  asymptotic, 13
Code generator
  matrix, 48
  polynomial, 75
Code puncturing and depuncturing, 203, 206
Concatenated code, 213–14
Constraint length. *See* Convolutional code
Convolutional code, 5
  catastrophic error propagation, 202

Convolutional code (*continued*)
code termination, 164
constraint length, 153
recursive systematic convolutional (RSC) code, 219
Coset, 22, 53
coset leader, 53
Cyclic code, 73
shortened, 91
Cyclic redundancy check (CRC), 93

Decoding depth, 174
Decoding sphere. *See* Hamming sphere
Deinterleaver. *See* Interleaver and deinterleaver
Distance
Euclidean distance, 167
Hamming distance, 10

Elementary symmetric function, 103
Erasure, 59
Erasure decoding
linear binary block code, 59
Reed-Solomon code, 140
Erasure location polynomial, 140
Error correcting capability
block code, 63
Reed-Solomon code, 114, 122–23
Error detecting capability
block code, 61
CRC, 94
Error evaluation polynomial, 116
Error floor. *See* Turbo code
Error location polynomial, 103
modified for IBM, 202
Euclid's algorithm, 131–32
Euclid's decoding of RS codes, 133
Extension field, 25
Extrinsic information, 232

Fano metric, 179
Fano algorithm, 185
Forward-backward algorithm. *See* Bahl-Cocke-Jelinek-Raviv (BCJR) algorithm
Field, 22
Finite field. *See* Galois field
Flushing bit, 164
Forney algorithm, 116
Free distance, 161
Frequency domain RS coding
decoding, 135
encoding, 119–20

Gallager code. *See* Low-density parity-check (LDPC) code
Galois field, 25
characteristic, 25
element representations, 26, 30
extension, 25
ground, 25
prime, 25
primitive element, 25–26
Galois field Fourier transform (GFFT), 119–20
inverse GFFT, 120
Greatest common divisor (GCD), 131
cyclic, 21
finite. *See* Galois field
generating element, 21
group, 19
order, 21
subgroup, 21

Hamming code, 66
construction, 68
decoding, 69
extended, 72
Hamming distance, 10
minimum, 11
Hamming sphere, 12
Hamming weight, 10
Hard-decision decoding, 10

Inner code, 214
Interleaver and deinterleaver, 214–15
algebraic, 226
block, 215, 218
convolutional, 218
helical interleaver, 226
quadratic permutation polynomial (QPP), 227
random, 225
simile, 226–27
S-random, 225–26
uniform, 228

Irreducible polynomial, 28
Iterative decoding
LDPC code, 259
turbo code, 238

Key equation, 116
Berlekamp-Forney, 141

Lagrange's theorem, 21
Log-likelihood ratio (LLR), 229
Log-MAP algorithm, 242, 245
Log-SPA, 268
Low-density parity-check (LDPC) code, 254

Maximum a posteriori (MAP) decoding, 10
Maximum-distance-separable (MDS) code, 122
Maximum likelihood (ML) decoding, 9
block code, 52
convolutional code, 166–67
Max-log-MAP algorithm, 244
Metric normalization
Viterbi decoding, 191
MAP decoding, 233–34
Meggitt decoder, 86–88
Message passing decoding algorithm. *See* Sum-product algorithm (SPA)
Minimum Hamming distance. *See* Hamming distance
Minimum polynomial, 32
Modified generating function, 162
extended, 175–76

Newton's identity, 104
Nonsystematic code, 5

Order of a field element, 26
Outer code, 214

Parity-check
matrix, 49
polynomial, 76
Path metric, 168
Pearl's algorithm. *See* Sum-product algorithm (SPA)
Peterson's algorithm, 106–7
Polynomial
over a prime Galois field, 26
primitive, 28
Polynomial division circuit, 81–82

Q-function, 8

Rate-compatible punctured convolutional (RCPC) code, 205
Reed-Solomon (RS) code, 117
Register-exchange, 193

Shannon's theorem, 14
Single-input single-output (SISO) decoder, 239
Sliding-window decoding
Viterbi algorithm, 173–74
SOVA, 249
Soft-decision decoding, 10
Soft-output Viterbi algorithm (SOVA), 247–49
Stack algorithm, 182–83
Standard array, 52–53
syndrome based, 55–56
State transition diagram, 156–57
Sum-product algorithm (SPA), 261–62
in log domain. *See* Log-SPA
Syndrome, 54
polynomial, 85
Systematic code, 5

Tail-biting, 165
Tanner graph, 255–57
Traceback, 193–95
Tree diagram, 156
Trellis diagram, 157
Truncation error, 174
Turbo code, 220–21
Turbo decoding, 240
component encoder and decoder, 221, 238

Vandermonde matrix, 101
Vector space, 22–23
Viterbi algorithm, 171
soft output. *See* Soft-output Viterbi algorithm (SOVA)

Weight distribution, 62

Zech algorithm, 39
Zech logarithm, 39
Zero-tailing, 164–65