Additional Experiments for Communication System Design Using DSP Algorithms with Laboratory Experiments for the TMS320C6713 DSK

Steven A. Tretter
Contents

19 Adaptive Equalization for PAM
  19.1 System Description .............................................. 1
  19.2 LMS Adaptive Equalization Theory for PAM ..................... 3
  19.3 Experiments for Adaptive Equalization for PAM ............... 4
    19.3.1 A Handshaking Sequence .................................... 5
    19.3.2 Experiments with Transmitter Output Looped Back to the Same DSK 6
    19.3.3 Experiments with Transmitter Output Connected to Another DSK . 9
  19.4 References ...................................................... 10

20 Continuous-Phase Frequency Shift Keying (FSK) ..................... 11
  20.1 Definition of the FSK Signal ..................................... 11
  20.2 Power Spectral Density for an FSK Signal ....................... 13
  20.3 FSK Demodulation ................................................ 16
    20.3.1 An Exact Frequency Discriminator ............................ 17
      20.3.1.1 Symbol Clock Acquisition and Tracking ................. 19
    20.3.2 A Simple Approximate Frequency Discriminator .............. 21
    20.3.3 The Phase-Locked Loop ....................................... 21
    20.3.4 Optimum Noncoherent Detection by Tone Filters ............. 23
      20.3.4.1 Discrete-Time Implementation ............................ 25
      20.3.4.2 Recursive Implementation of the Tone Filters .......... 27
      20.3.4.3 Simplified Demodulator for Binary FSK ................ 28
      20.3.4.4 Generating a Symbol Clock Timing Signal ............... 28
  20.4 Symbol Error Probabilities for FSK Receivers .................. 35
    20.4.1 Orthogonal Signal Sets ..................................... 35
  20.5 Experiments for Continuous-Phase FSK .......................... 36
    20.5.1 Theoretical FSK Spectra ..................................... 37
    20.5.2 Making FSK Transmitters .................................... 37
      20.5.2.1 Initial Handshaking Sequence ............................ 37
      20.5.2.2 Simulating Random Customer Data ....................... 37
      20.5.2.3 Experimentally Measure the FSK Power Spectral Density . 38
    20.5.3 Making a Receiver Using an Exact Frequency Discriminator .... 38
      20.5.3.1 Running a Bit-Error Rate Test (BERT) ................... 39
    20.5.4 Making a Receiver Using an Approximate Frequency Discriminator 40
20.5.5 Making a Receiver Using a Phase-Locked Loop .......................... 40
20.5.6 Making a Receiver Using Tone Filters ..................................... 40
20.5.6.1 $M = 4$ Tone Filter Receiver ............................................... 40
20.5.6.2 Simplified $M = 2$ Tone Filter Receiver ............................... 41

21 Brief Introduction to Direct Sequence Spread Spectrum Systems ............. 43
21.1 Direct Sequence Spread Spectrum Transmitters ................................ 44
21.1.1 Bipolar Data with Bipolar Spreading ...................................... 44
21.1.1.1 Spectrum of $s(t)$ in the Case of Ideal Binary Random Data and Spreading Codes ......................................................... 46
21.1.1.2 Spectrum of a Maximal Length Bipolar PN Signal .................. 47
21.1.2 Bipolar Data and QPSK Spreading ........................................... 48
21.1.3 Two Different Data Streams Spread on Quadrature Carriers .......... 50
21.2 Rejection of Narrowband Interference ......................................... 50
21.2.1 Bipolar Data Signal with Bipolar Spreading ............................. 50
21.2.2 Bipolar Data Signal with QPSK Spreading ............................... 51
21.3 PN Code Tracking for Bipolar Data and Spreading ........................... 53
21.3.1 Coherent Delay-Locked Loop for Bipolar Data and Spreading ...... 53
21.3.2 Noncoherent Delay-Locked Loop ........................................... 56
21.3.2.1 Phase Adjustment Using a Random Walk Filter .................. 58
21.4 Code Tracking for QPSK Spreading .............................................. 59
21.5 Data Detection at the Receiver for Bipolar Data and Spreading .......... 60
21.5.1 Costas Loop for Carrier Tracking and Demodulation .................. 60
21.5.2 Matched Filtering ................................................................. 63
21.5.3 Symbol Clock Tracking .......................................................... 65
21.6 Data Detection for Bipolar Data and QPSK Spreading ....................... 67
21.7 Data Detection for Two Different Bipolar Data Streams Spread on Quadrature Carriers .............................................................. 69
21.8 Experiments for Spread Spectrum Transmitters and Receivers .......... 70
21.8.1 Experiments for Bipolar Data and Bipolar Spreading .................. 71
21.8.1.1 Making a Transmitter for Bipolar Data and Bipolar Spreading 71
21.8.1.2 Making a Noncoherent Delay-Locked Loop Code Tracker for Bipolar Data and Bipolar Spreading ................................. 72
21.8.1.3 Making a Costas Loop Data Demodulator Acting on the Despread Received Signal ....................................................... 74
21.8.1.4 Making the Matched Filter and Data Symbol Clock Tracker ...... 75
21.8.1.5 Testing Immunity to Sinusoidal Interference .......................... 76
21.8.2 Experiments for Bipolar Data and QPSK Spreading ..................... 76
21.8.2.1 Making a Transmitter for Bipolar Data and QPSK Spreading .... 76
21.8.2.2 Making a Noncoherent DLL for QPSK Code Tracking ............ 76
21.8.2.3 Making a Costas Loop, Matched Filter, Clock Tracker, and Data Detector ................................................................. 77
21.8.2.4 Testing Immunity to Sinusoidal Interference .......................... 77
## 21.8.3 Experiments for Two Different Bipolar Data Streams Spread on Quadrature Carriers

- 21.8.3.1 Making a Transmitter for Two Different Bipolar Data Streams Spread on Quadrature Carriers
- 21.8.3.2 Implementing a Noncoherent DLL Code Tracker
- 21.8.3.3 Making a Data Demodulator

## 21.9 References

## 22 Introduction to Convolutional Codes

- 22.1 The Huffman D-Transform
  - 22.1.1 Two-Sided Transform of a Delayed Sequence
  - 22.1.2 One-Sided Transform of a Delayed Sequence
  - 22.1.3 D-Transform of a Convolution
- 22.2 Transfer Functions and Realizations
  - 22.2.1 Type 1 Direct Form Realization
  - 22.2.2 Type 2 Direct Form Realization
- 22.3 Description of a Convolutional Code by its Generator Matrix
- 22.4 Systematic Form of a Convolutional Code
- 22.5 The Parity Check Matrix and Syndromes
- 22.6 The Code Trellis
- 22.7 Weight Distributions and Error Correction Properties
- 22.8 Elements of Lattice Theory
  - 22.8.1 Definition of a Lattice
  - 22.8.2 Sublattices, Lattice Partitions, and Cosets
- 22.9 Trellis Coded Modulation (TCM)
- 22.10 The Viterbi Decoding Algorithm
  - 22.10.1 Three Channel Models
    - Example 22.18. The Binary Symmetric Channel
    - Example 22.19. Biphase Data Over a White Gaussian Noise Channel
    - Example 22.20. Quadrature Amplitude Modulation Over a White Gaussian Noise Channel
  - 22.10.2 Detailed Explanation of the Viterbi Algorithm
  - 22.10.3 Some Practical Implementation Techniques
- 22.11 The BCJR or Forward-Backward Decoding Algorithm
  - 22.11.1 The Encoder is a Markov Source
  - 22.11.2 Formulas for Computing the Probabilities in (22.107)
    - 22.11.2.1 Computing $\sigma_n(l', l)$
  - 22.11.3 Normalization for Computational Stability
    - 22.11.3.1 A Decomposition of $\Lambda_n(r)$ for Systematic Codes with BPSK on an AWGN Channel to Use with Turbo Codes
    - 22.11.3.2 Memory and Computation Requirements
  - 22.11.4 Summary of the Steps for Applying the BCJR Algorithm
- 22.12 The Max-Log-MAP Algorithm
22.13 The Log-MAP Algorithm ................................................. 128
22.14 Estimating Bit Error Rates ........................................... 129
22.15 Experiments for Convolutional Codes .......................... 130
   22.15.1 Exploring Uncoded BPSK Transmission .......................... 130
   22.15.2 Implementing the LTE Encoder and a Viterbi Decoder for Biphase Transmission Over an Additive White Gaussian Noise Channel ........................................... 131
   22.15.3 Decoding the LTE Code with the BCJR Algorithm for Biphase Transmission Over an Additive White Gaussian Noise Channel ........................................... 133
   22.15.4 Decoding the LTE Code with the log-MAP Algorithm for Biphase Transmission Over an Additive White Gaussian Noise Channel ........................................... 134
   22.15.5 Decoding the LTE Code with the max-log-MAP Algorithm for Biphase Transmission Over an Additive White Gaussian Noise Channel ........................................... 134
   22.15.6 Trellis Coded Modulation Example Using the Ungerboeck 4-State Systematic Code with an 8-Phase Constellation and Viterbi Decoding ........................................... 134
22.16 References ................................................................. 139

23 Turbo Codes ................................................................. 141
   23.1 Introduction ............................................................. 141
   23.2 Capacity Formulas for the AWGN Channel ....................... 142
   23.3 The Turbo Encoder ...................................................... 145
   23.4 Bit Error Probability for Turbo Codes ............................ 148
   23.5 Exact Minimum Bit Error Probability Decoding of Turbo Codes ........................................... 150
   23.6 Iterative Decoding of Turbo Codes .................................. 150
      23.6.1 MAP Decoder 1 Computations .................................. 151
      23.6.2 MAP Decoder 2 Computations .................................. 154
   23.7 Iteration Stopping Methods ......................................... 156
      23.7.1 A Fixed Number of Iterations .................................. 156
      23.7.2 The Cross Entropy Criterion .................................. 156
      23.7.3 The Sign-Change-Ratio (SCR) Criterion ....................... 158
      23.7.4 The Hard-Decision-Aided (HDA) Criterion ................. 158
   23.8 Turbo Code Experiments ............................................. 159
   23.9 Appendix ................................................................. 161
   23.10 References ............................................................... 165

24 Low-Density Parity-Check Codes .................................... 167
   24.1 Introduction ............................................................ 167
   24.2 Definition of Low-Density Parity-Check Codes ................ 167
      24.2.1 Efficient Representation of a Sparse Matrix for Computer Storage ........................................... 169
   24.3 Representing a Parity-Check Code by a Tanner Graph ....... 170
      24.3.1 Cycles in a Tanner Graph ..................................... 171
   24.4 Probabilities for Binary Phase Shift Keying (BPSK) Over an Additive, White, Gaussian Noise Channel ........................................... 172
      24.4.1 Hard Bit Decisions Using a Log-Likelihood Ratio ....... 173
Chapter 19

Adaptive Equalization for PAM

This chapter is an extension of “Chapter 11 Digital Data Transmission by Baseband Pulse Amplitude Modulation.” The PAM transmitter created in that chapter will be used here. The PAM transmitter output will be passed through a channel simulation filter to add intersymbol interference (ISI). An adaptive FIR filter using the least mean-square (LMS) algorithm will be used to eliminate most of the ISI. In the first experiments the distorted transmitter output samples with ISI will be simply looped back to an adaptive equalizer in the same DSK program as the transmitter to avoid having to implement the interpolator of Chapter 12 and the symbol clock tracker of Chapter 11. As an optional experiment, the distorted transmitter output will be sent to a codec output and connected to the codec input in another PC. Then the interpolator and symbol clock tracker will have to be implemented in addition to the adaptive equalizer.

For the experiments in this chapter, initial equalizer training will be accomplished by using a 15-symbol repeating sequence and the LMS algorithm. See Tretter [2, Chapter 11] for a rapid method of computing the initial equalizer tap values using the DFT and IDFT when a known repeating sequence is transmitted. See “Chapter 15 QAM Receiver II” of Communication System Design Using DSP Algorithms for a detailed presentation of adaptive equalization for quadrature amplitude systems. A method for initially adjusting the equalizer called “blind equalization” when the transmitted sequence is unknown is also discussed in Chapter 15 and can be modified for PAM equalizers.

19.1 System Description

A block diagram of a PAM receiver with an adaptive equalizer is shown in Figure 19.2. As in Chapter 11, $T$ is the symbol period. The received signal $r(t)$ is sampled $L$ times per symbol, that is, with sampling period $T/L$ resulting in the sequence $r(nT/L)$. For the experiments of this chapter, $L = 4$ will be used. The symbol clocks in the transmitter and receiver will be slightly different because they are generated by hardware at different locations and, possibly, by relative motion between the transmitter and receiver. Therefore, the receiver must acquire the symbol clock in the received signal. A method for doing this is to pass the received
sequence $r(nT/L)$ through a variable phase interpolator. Implementations of the interpolator are presented in Chapter 12. Let the interpolator output be $\tilde{r}(nT/L) = r[(n + \delta)T/L]$ where $\delta$ is the time shift introduced by the interpolator as a fraction of the sampling period. Then $\tilde{r}(nT/L)$ is applied to a symbol clock tone generator and phase tracking logic which is discussed in Section 12.3 and shown in Figure 12.1. This symbol clock tracking loop locks to the positive zero crossings of the clock tone generator output.

The interpolator output is then down sampled by a factor of $K$ to give the sequence $\tilde{r}(nTK/L)$ which is applied to the adaptive equalizer. $K$ and $L$ are chosen so that $L/K = Q$ is an integer. Thus the adaptive equalizer operates on samples taken with period $TK/L = T/Q$. For the experiments in this chapter, you will use $L = 4$ and $K = 2$ so $Q = 2$ and $TK/L = T/2$. This allows the equalizer to have a frequency response that can compensate for received signals like raised cosine signals whose bandwidth extends somewhat beyond the Nyquist frequency of $\omega_s/2 = \pi/T$. The equalizer is an FIR filter with a delay line with taps spaced $TK/L = T/Q$ which is $T/2$ for our experiments. A block diagram of the equalizer is shown if Figure 19.2. The blocks labeled $z^{-K/L}$ represent delays of $KT/L = T/Q$. It is said to be a fractionally spaced equalizer. The equalizer output is down-sampled by a factor of $Q$ to get the symbol rate samples $\hat{s}(nT)$. The equalizer output is computed just once per symbol, that is, every $T$ seconds and gives an estimate of the transmitted symbol. The intermediate equalizer outputs are not used. However, samples are entered into the equalizer delay line every $KT/L$ seconds.

It turns out that a fractionally spaced equalizer also acts as an interpolator and automatically adjusts for signal time shifts. However, the receiver symbol clock must be locked in frequency to the clock in the received signal. If there is a frequency difference in the clocks, the equalizer will try to compensate for the drifting time reference and will fail when the correct timing falls off the ends of the equalizer.
19.2 LMS Adaptive Equalization Theory for PAM

The equalizer is an FIR filter with \( N \) adjustable tap values \( c_0, c_1, \ldots, c_{N-1} \). The taps are spaced by \( \tau = TK/L = T/Q \) which for the experiments in this chapter will be \( \tau = T/2 \). The taps are updated only at the symbol instants \( nT \) to minimize the mean-square-error between the equalizer output and the ideal transmitted symbol. Samples are entered into the delay line every \( TK/L = T/Q \) seconds but the equalizer output is only computed every \( T \) seconds. The intermediate equalizer outputs are not needed.

The equalizer output at time \( nT \) is

\[
\hat{s}(nT) = \sum_{k=0}^{N-1} c_k \tilde{r} \left( nT - k \frac{T}{Q} \right) = \sum_{k=0}^{N-1} c_k \tilde{r} (nT - k\tau) \quad (19.1)
\]

Suppose the actual transmitted symbol sequence is \( a_n \). In practice, there is usually an initial handshaking procedure between the transmitter and receiver where a know symbol sequence is transmitted for a period of time and the equalizer is adjusted in what is called the *ideal reference* mode. Once the equalizer has converged, reliable estimates of the transmitted ideal symbols can be obtained from the equalizer outputs and the equalizer can be adjusted in what is called the *decision directed* mode. The instantaneous equalizer output error at the symbol times is

\[
e(nT) = a_n - \hat{s}(nT) \quad (19.2)
\]

and the mean-squared output error is

\[
\Lambda = E\{e^2(nT)\} = E\{(a_n - \hat{s}(nT))^2\} = E \left\{ \left[ a_n - \sum_{k=0}^{N-1} c_k \tilde{r} (nT - k\tau) \right]^2 \right\} \quad (19.3)
\]
The tap values that minimize $\Lambda$ can be found by setting the partial derivatives of $\Lambda$ with respect to the tap values to zero. The partial derivative of $\Lambda$ with respect to tap $c_i$ is

$$\frac{\partial \Lambda}{\partial c_i} = 2E \left\{ e(nT) \frac{\partial e(nT)}{\partial c_i} \right\} = -2E \{ e(nT) \tilde{r}(nT - i\tau) \} \quad \text{for } i = 0, \ldots, N - 1 \quad (19.4)$$

Setting the partial derivatives to zero results in a set of $N$ linear equations in $N$ unknowns which is essentially the same as the set for QAM presented in Section 15.1.1 and involves inverting the $N \times N$ correlation matrix for the delay line contents.

Rather than solving the equations directly, an iterative technique for converging to the solution is typically used. The partial derivatives point in the directions of increase in $\Lambda$ with respect to the tap values. Therefore, incrementing a tap value by a small step in the direction opposite to the partial derivative of $\Lambda$ with respect to that tap value will decrease $\Lambda$. The expectation $E\{ e(nT) \tilde{r}(nT - i\tau) \}$ where $\tau = T/Q$ in (19.4) can be approximated by just $e(nT) \tilde{r}(nT - i\tau)$. The factors $e(nT)$ and $\tilde{r}(nT - i\tau)$ can be directly measured in the receiver. In fact, $\tilde{r}(nT - i\tau)$ is just the equalizer input sample sitting at tap $i$. Let $\mu$ be a small positive scale factor. Then the tap update algorithm is

$$c_i(n + 1) = c_i(n) + \mu e(nT) \tilde{r}(nT - i\tau) \quad \text{for } i = 0, \ldots, N - 1 \quad (19.5)$$

The tap updates are performed only at the symbol instants $nT$. This algorithm is called the LMS (least mean-square) algorithm and was popularized by B. Widrow [3]. A block diagram illustrating the tap update algorithm is shown in Figure 19.3. The switch is initially connected to $a_n$ for ideal reference training. It is connected to $\hat{a}_n$ for decision directed adaptation after the equalizer has converged.

The scale factor $\mu$ must be chosen small enough to guarantee stability of the LMS algorithm. Let the eigenvalues of the correlation matrix for the delay line contents be $\{\lambda_i\}$. Then it can be shown that the criterion for stability of tap convergence is

$$0 < \mu < 1/ \max_i \{\lambda_i\} \quad (19.6)$$

See Section 15.1.2 for more details on convergence.

The scale factor $\mu$ determines the speed of convergence and accuracy of the algorithm. The tap values hover about the optimum solution when steady-state is reached and the equalizer output is somewhat noisy. A larger $\mu$ results in faster convergence but a noisier output. Smaller values result in slower convergence but less output noise. An approach in practice is to use a large $\mu$ for initial training and then switch to a smaller one during data detection.

### 19.3 Experiments for Adaptive Equalization for PAM

As usual, set the codec sampling rate to 16,000 Hz. In all the experiments use an $N = 30$ tap $T/2$ spaced equalizer. Thus the equalizer delay line spans 15 symbols. You can use the PAM
transmitter you created in Chapter 11 as a starting point. The transmitter you created there generates four output samples per symbol, so the symbol rate is $f_s = 4000$ symbols/second. Only two-level PAM will be investigated. Internally in your program use the levels 3 and -3. Scale the interpolation filter bank outputs by an appropriate value to use a significant part of the dynamic range of the DAC, convert the samples to integers, and put them in the left output channel. Put a baud sync signal in the right channel. These steps should already be in the program you created for Chapter 11.

### 19.3.1 A Handshaking Sequence

Modify your PAM transmitter to generate the following handshaking sequence. You will need to include an integer variable in your program to count the number of transmitted
symbols to determine when different parts of the handshaking sequence should run.

1. First send 1/2 second of silence by transmitted 2000 symbols with level 0. This will allow a receiver to decide that no signal is present and initiate code to detect signal presence.

2. Next send 1/2 second of symbols that alternate between 3 and -3 for 2000 symbols. This is called a *dotting sequence*. The transmitter output will be a sine wave at 2000 Hz which is half the symbol rate. It will provide a strong signal to allow a receiver to detect the presence of a signal, adjust its AGC, and lock its symbol clock tracking loop to the received symbol clock.

3. Next transmit a two-level symbol sequence that repeats every 15 symbols for 2000 symbols. Notice that the sequence length is the same as the number of symbols spanned by the equalizer delay line. The receiver will use this known sequence for ideal reference equalizer training. Generate the sequence with a 4-stage maximal length feedback shift register with the connection polynomial \( h(D) = 1 + D + D^4 \) as explained in Chapter 9. Set the initial state of the shift register to any non-zero value of your choosing. Let the binary sequence generated be \( b(n) \). Then the rule for generating it is

\[
b(n) = b(n - 1) \oplus b(n - 4)
\]  

(19.7)

where \( \oplus \) is modulo 2 addition. Include a listing of the sequence in your lab report.

Map the logical binary value 0 to symbol level +3 and value 1 to symbol level -3. An equation for this mapping if \( b(n) \) is considered to be a real number is

\[
a(n) = 3 - 6b(n)
\]  

(19.8)

Scale the interpolation filter bank outputs by the same value as before, convert them to integers, and send them to the codec in the left channel along with a baud sync signal in the right channel.

4. As the last step, continually send a two-level pseudo-random symbol sequence based on a 23-stage maximal length feedback shift register generator with the connection polynomial \( h(D) = 1 + D^{18} + D^{23} \) as you did for Chapter 11. This will allow the equalizer to do finer equalizer adjustment and also simulate binary random customer data.

### 19.3.2 Experiments with Transmitter Output Looped Back to the Same DSK

For this experiment simply loop the unscaled transmitter output samples internally back to your receiver equalizer code in the same program as the transmitter. This will significantly simplify the receiver program because you will not have to detect signal presence, not have
to implement an interpolator and symbol clock tracking logic, and can use the transmitter’s symbol counter to determine the handshaking phase. Clock tracking is not an issue because the transmitter and receiver code are running in the same DSP with the same clock. Actually, you will introduce inter-symbol-interference (ISI) in the transmitted signal by filtering the original interpolation filter bank output with a channel simulation filter and loop these filtered samples back to the receiver.

Create a receiver program to perform the following items:

1. Introduce ISI by passing the original transmitter interpolation filter bank output samples through an IIR filter of the form

\[
G(z) = \frac{1.5(1 - b_1)(1 - b_2)}{(1 - b_1 z^{-1})(1 - b_2 z^{-1})} = \frac{c}{1 + d_1 z^{-1} + d_2 z^{-2}}
\]

(19.9)

where \(b_1 = 0.9\), \(b_2 = 0.7\), \(c = 1.5(1 - b_1)(1 - b_2)\), \(d_1 = -(b_1 + b_2)\), and \(d_2 = b_1 b_2\). Scale the filtered samples which occur at a 16 kHz rate appropriately, convert them to integers and send them to the left codec channel along with a baud clock sync signal in the right channel.

Arrange your program so that this channel simulation filter can be included or not. First do not include the filter and observe the nearly ideal eye diagram on the oscilloscope. The two-level eyes will be almost completely open and the transmitted symbols can be determined without error by observing the polarity of the received signal at the symbol instants. You should be able to see the different phases of the handshaking sequence.

Next enable the channel simulation filter and you should see that the eye is completely closed and that the transmitted symbols cannot be determined from this signal.

2. Down sample the output of the channel simulation filter by a factor of two and put the resulting samples into the delay line of a \(T/2\) spaced 30-tap adaptive equalizer. That is, put every other channel filter output into the equalizer delay line. The channel output samples occur at a 16 kHz rate, so the down sampled sequence samples occur at an 8 kHz rate. Do this for the floating point samples without scaling for the codec.

3. Cyclic Equalization

Wait for the silence and dotting phases of the transmitted handshaking sequence to end based on the transmitter’s symbol counter. Then wait 30 more \(T/2\) samples for the equalizer delay line to fill up with samples from the 15-symbol repeating phase. Make a replica in your receiver of the 15-symbol sequence generator. Use it as an ideal reference and update the equalizer taps once per symbol, that is, once every second delay line input sample, using the LMS algorithm specified by (19.5). Try \(\mu = 0.002\). Do not compute the equalizer output between symbol instants, just shift a new \(T/2\) sample into the delay line.

After several cycles of the 15-point symbol sequence, the output of the channel filter will also repeat every 15 symbols. Let one period of the sequence be \(a_0, \ldots, a_{14}\) and
its DFT be $A_0, \ldots, A_{14}$. According to the IDFT formula

$$a_n = \frac{1}{15} \sum_{k=0}^{14} A_k e^{j\frac{2\pi}{15}nk} = \frac{1}{15} \sum_{k=0}^{14} A_k e^{j(k\omega_s/15)nT} \text{ for } n = 0, \ldots, 14 \quad (19.10)$$

where $\omega_s = 2\pi/T = 2\pi \times 4000$ is the radian symbol rate. Thus the repeating sequence probes the channel only at the discrete frequencies $k\times4000/15$ for $k = 0, \ldots, 14$.

At the end of Cyclic Equalization and before Tap Rotation, extract and plot the equalizer tap sequence. Do this only once since the program must be stopped to read the tap values so the system will not run in real time. After extracting the tap values once, reload and restart the program without looking at the tap values at the end of Cyclic Equalization so the program runs in real time.

4. **Tap Rotation**

The periodic sequences in the transmitter and receiver will usually not be in phase with each other. However, the equalizer will automatically set up to optimize for periodic sequence in the receiver. It will automatically shift the received sequence to align it in time with the local ideal reference. The position of the largest equalizer tap indicates the shift required for this alignment. The largest tap may not be near the center of the equalizer delay line and this will not be good when a random data signal with a distributed spectrum is received. A solution to this problem is given next.

At the end of the repeating 15-point sequence phase, determine the location of the tap with the largest magnitude. Then rotate the equalizer tap sequence an integer number of symbols, that is, by a multiple of two positions, to move the largest tap near the center of the delay line. Cyclically rotating an $N$-point sequence $\ell$ positions to the right results in its DFT being multiplied by $\exp(-j2\pi\ell k/N)$ and just adds a linear phase shift to the equalizer frequency response at the probe frequencies. The effect is to delay the 15-point repeating equalizer output sequence by $\ell$ symbols.

Also at the end of the periodic sequence phase, “gear shift” the update scale factor $\mu$ to a smaller value to achieve finer convergence.

5. **Decision Directed Equalization**

By the end of cyclic ideal reference training, the equalizer outputs at the symbol instants should be close to the ideal symbol values and the ideal transmitted symbols can be correctly determined with high probability from the polarity of the equalizer outputs. This is sometimes referred to as *slicing* the equalizer output to the nearest ideal symbol level. From the end of the periodic ideal reference phase onward, use the sliced equalizer outputs as the ideal symbol reference values. This is called *decision directed equalization*.

The equalizer will continue to adapt some after the periodic training phase has ended and a random customer data sequence is transmitted because the spectrum of the transmitted signal becomes distributed over the signal bandwidth rather than discrete
lines at the 15 probe frequencies. The receiver has no idea what random sequence is transmitted and cannot use ideal reference training at this point. However, it can use decision directed training once the eye is open. In practice, the equalizer is continually adapted during data transmission to track small deviations in symbol clock timing and channel changes.

Plot the equalizer coefficients after they have converged with Decision Directed Equalization and compare them with the coefficients at the end of Cyclic Equalization and Tap Rotation.

6. Observing the Equalizer Output

The equalizer output is computed only at the symbol instants, so the question of how to observe it in real-time using the lab equipment arises. You cannot write it to the console or to a file in the PC because the program will not run in real-time then.

A method to view the equalizer output on the oscilloscope is to apply it to an interpolation filter bank similar to the one in the transmitter to generate four samples per symbol and send the resulting samples at the 16 kHz rate to one output channel of the codec. The interpolation filter bank should be based on a raised cosine filter, not a square root of raised cosine filter, so it introduces no inter-symbol-interference. Send a baud sync signal to the other output channel. Make sure to scale the equalizer outputs appropriately to use a large portion of the dynamic range of the codec’s DAC. You can then observe an eye diagram on the oscilloscope and see the equalizer converge in real-time.

Experiment with different values of the equalizer update scale factor $\mu$ to see how it affects the convergence speed and accuracy.

19.3.3 Experiments with Transmitter Output Connected to Another DSK

Now you will make a PAM receiver that works in a different DSK than the one containing the transmitter. It will contain most of the systems that a real world receiver requires. Use the transmitter code you created for the previous section in one DSK. The transmitter should use the handshaking sequence specified there. Connect the line out of the transmitter DSK to the line in of a DSK in another PC where you will make the receiver. Use a 16 kHz sampling rate for the codec in the receiver DSK. Your receiver code should perform the following tasks:

1. The receiver should monitor the 16 kHz input samples to detect the presence of a PAM signal. Devise a method to relatively quickly detect received signal energy, the presence of a 2 kHz dotting tone, or a combination of both criteria. Once a signal is detected, continue to monitor the input energy to determine when the input signal stops.

2. Start a symbol counter when an input signal is detected to determine when different phases of the handshaking sequence are present.
3. As soon as the dotting tone is detected, start your symbol clock tracking loop including a variable phase interpolator, clock tone generator, and phase correction logic. All these components should operate at a 16 kHz sampling rate. Remember that the clock tracking loop locks to the positive zero crossings of the generated clock tone. You can get an initial rough estimate for the correct interpolator phase by finding the position of a positive zero crossing of the clock tone and using it to set the phase. Making a good guess for the initial phase results in quicker loop lock.

4. When the 15-symbol periodic phase starts, down sample the interpolator output by a factor of two, and wait for at least thirty $T/2$ samples to fill the equalizer delay line. Then use ideal reference training to adapt the equalizer as you did in the previous section. Also send the equalizer outputs four times per symbol to the left codec channel and a baud sync signal to the right channel as before. Continue to send the equalizer output to the codec from here on.

5. At the end of the periodic signal phase, perform tap rotation and switch to decision directed updating as before.

6. Plot the equalizer coefficients after they have converged.

7. Connect the baud sync signals from the transmitter and receiver to the oscilloscope and see if they are locked in frequency or drift relative to each other. You will see some clock jitter but they should be essentially locked in frequency. Turn off the interpolator phase updating and see if the two baud sync clocks drift relative to each other.

### 19.4 References


Chapter 20
Continuous-Phase Frequency Shift Keying (FSK)

20.1 Definition of the FSK Signal

Continuous-phase frequency shift keying (FSK) is often used to transmit digital data reliably over wireline and wireless links at low data rates. Simple receivers with low error probability can be built. The block diagram of an M-ary FSK transmitter is shown in Figure 20.1. Binary \((K = 1, M = 2)\) FSK is used in most applications, often to send important control information. The early voice-band telephone line modems used binary FSK to transmit data at 300 bits per second or less and were acoustically coupled to the telephone handset. Teletype machines used these modems. The 3GPP Cellular Text Telephone Modem (CTM) for use by the hearing impaired over regular cellular speech channels uses \(M = 4\) FSK. At the FSK transmitter input, bits from a binary data source with a bit-rate of \(R\) bits per second are grouped into successive blocks of \(K\) bits by the “Serial to Parallel Converter.” Each block is used to select one of \(M = 2^K\) radian frequencies from the set

\[
\Lambda_k = \omega_c + \omega_d [2k - (M - 1)] = 2\pi f_c + 2\pi \omega_d [2k - (M - 1)] \quad \text{for} \quad k = 0, 1, \ldots, M - 1 \tag{20.1}
\]

The frequency \(\omega_c = 2\pi f_c\) is called the carrier frequency. The radian frequencies

\[
\Omega_k = \omega_d [2k - (M - 1)] = 2\pi \omega_d [2k - (M - 1)] \quad \text{for} \quad k = 0, 1, \ldots, M - 1 \tag{20.2}
\]

are the possible frequency deviations from the carrier frequency during each symbol. The deviations range from \(-\omega_d(M - 1)\) to \(\omega_d(M - 1)\) in steps of \(\Delta \omega = 2\omega_d\). Blocks are formed at the rate of \(f_b = R/K\) blocks per second, so each frequency is sent for \(T_b = 1/f_b\) seconds. Let \(\omega_b = 2\pi f_b\). The sinusoid transmitted during a block is called the FSK symbol specified by the block. The symbol rate, \(f_b\), is also called the baud rate.

During the symbol period \(nT_b \leq t < (n + 1)T_b\) the “D/A” box uniquely maps each possible input block to a possible frequency deviation

\[
\Omega(n) = \omega_d [2k_n - (M - 1)] \tag{20.3}
\]
and forms the signal $\Omega(n)p(t - nT_b)$ where $p(t)$ is the unit height pulse of duration $T_b$ defined as

$$p(t) = \begin{cases} 1 & \text{for } 0 \leq t < T_b \\ 0 & \text{elsewhere} \end{cases}$$  \hspace{1cm} (20.4)

Assuming transmission starts at $t = 0$, the complete “D/A” converter output is the staircase signal

$$m(t) = \sum_{n=0}^{\infty} \Omega(n)p(t - nT_b)$$  \hspace{1cm} (20.5)

This baseband signal is applied to an FM modulator with carrier frequency $\omega_c$ and frequency sensitivity $k_\omega = 1$ to generate the FSK signal

$$s(t) = A_c \cos \left( \omega_c t + \int_0^t m(\tau) \, d\tau + \phi_0 \right)$$  \hspace{1cm} (20.6)

where $A_c$ is a positive constant and $\phi_0$ is a random angle representing the initial phase value of the phase of the modulator. The pre-envelope of $s(t)$ is

$$s_+(t) = A_c e^{j\omega_c t} e^{j\int_0^t m(\tau) \, d\tau} e^{j\phi_0}$$  \hspace{1cm} (20.7)

and the complex envelope is

$$x(t) = A_c e^{j\int_0^t m(\tau) \, d\tau} e^{j\phi_0}$$  \hspace{1cm} (20.8)

The phase contributed by the baseband message is

$$\theta_m(t) = \int_0^t m(\tau) \, d\tau = \int_0^t \sum_{n=0}^{\infty} \Omega(n)p(\tau - nT_b) \, d\tau$$

$$= \sum_{n=0}^{\infty} \Omega(n) \int_0^t p(\tau - nT_b) \, d\tau$$  \hspace{1cm} (20.9)
20.2 Power Spectral Density for an FSK Signal

Now consider the case when $iT_b \leq t < (i+1)T_b$. Then

$$\theta_m(t) = \sum_{n=0}^{i-1} \Omega(n)T_b + \Omega(i) \int_{iT_b}^{t} d\tau$$

$$= T_b \omega_d \sum_{n=0}^{i-1} [2k_n - (M-1)] + T_b \omega_d[2k_i - (M-1)] \frac{(t - iT_b)}{T_b}$$

$$= \pi \frac{2\omega_d}{\omega_b} \sum_{n=0}^{i-1} [2k_n - (M-1)] + \pi \frac{2\omega_d}{\omega_b} [2k_i - (M-1)] \frac{(t - iT_b)}{T_b} \quad (20.10)$$

The modulation index for an FSK signal is defined to be

$$h = \frac{2\omega_d}{\omega_b} = \frac{\Delta\omega}{\omega_b} = \frac{\Delta f}{f_b} \quad (20.11)$$

and the phase at the start of the $i$th symbol is

$$\theta_m(iT_b) = \pi \frac{\Delta\omega}{\omega_b} \sum_{n=0}^{i-1} [2k_n - (M-1)] = \pi h \sum_{n=0}^{i-1} [2k_n - (M-1)] \quad (20.12)$$

Therefore,

$$\theta_m(t) = \theta_m(iT_b) + \pi h [2k_i - (M-1)] \frac{(t - iT_b)}{T_b} \quad \text{for} \quad iT_b \leq t < (i+1)T_b \quad (20.13)$$

The phase function $\theta_m(t)$ is continuous and consists of straight line segments whose slopes are proportional to the frequency deviations. Another approach to FSK would be to switch between independent tone oscillators. This switched oscillator approach could cause discontinuities in the phase function which would cause the resulting FSK signal to have a wider bandwidth than continuous phase FSK.

### 20.2 Power Spectral Density for an FSK Signal

Deriving the power spectral density for an FSK signal turns out to be a surprisingly complicated task. Lucky, Salz, and Weldon\(^1\) present the solution for a slightly more generalized form of FSK than described above. The term “power spectrum” will be used for “power spectral density” from here on for simplicity. They allow the pulse $p(t)$ to have an arbitrary shape but still be confined to be zero outside the interval $[0, T_b)$. They use the following definition of the power spectrum, $S_{xx}(\omega)$, of a random process $x(t)$:

$$S_{xx}(\omega) = \lim_{\lambda \to \infty} \frac{1}{\lambda} \mathbb{E}\{ |X_\lambda(\omega)|^2 \} \quad (20.14)$$

where $\mathbb{E}\{\cdot\}$ denotes statistical expectation and

$$X_\lambda(\omega) = \int_0^\lambda x(t)e^{-j\omega t} \, dt \quad (20.15)$$

Only formulas for the power spectrum of the complex envelope will be presented here since the power spectrum for the complete FSK signal can be easily computed as

$$S_{ss}(\omega) = \frac{1}{4}S_{xx}(\omega - \omega_c) + \frac{1}{4}S_{xx}(-\omega - \omega_c) \quad (20.16)$$

The frequency deviation in the complex envelope during the interval $[nT_b, (n+1)T_b)$ is

$$s_n(t - nT_b) = \Omega(n)p(t - nT_b) \quad (20.17)$$

The phase change caused by this frequency deviation during the baud when time is taken relative to the start of the baud is

$$b_n(t) = \Omega(n) \int_0^t p(\tau) \, d\tau \quad \text{for } 0 \leq t < T_b \quad (20.18)$$

The total phase change over a baud is

$$B_n = b_n(T_b) = \Omega(n) \int_0^{T_b} p(\tau) \, d\tau \quad (20.19)$$

The Fourier transform of a typical modulated pulse is

$$F_n(\omega) = \int_0^{T_b} e^{j\Omega(t)}e^{-j\omega t} \, dt \quad (20.20)$$

It is convenient to define the following functions:

1. The characteristic function of $b_n(t)$

$$C(\alpha; t) = \mathbb{E}\{e^{j\alpha b_n(t)}\} \quad (20.21)$$

2. The average transform of a modulated pulse

$$F(\omega) = \mathbb{E}\{F_n(\omega)\} \quad (20.22)$$

3. 

$$G(\omega) = \mathbb{E}\left\{|F_n(\omega)e^{jB_n}|^2\right\} \quad (20.23)$$

4. The average squared magnitude of a pulse transform

$$P(\omega) = \mathbb{E}\{|F_n(\omega)|^2\} \quad (20.24)$$
In terms of these quantities, the power spectrum is

\[
\frac{T_b}{A_c^2} S_{xx}(\omega) = \begin{cases} 
P(\omega) + 2\Re \left[ F(\omega)G(\omega) \frac{e^{-j\omega T_b}}{1 - C(1; T_b)e^{-j\omega T_b}} \right] & \text{for } |C(1; T_b)| < 1 \\
|P(\omega)|^2 + \omega_b \sum_{n=-\infty}^{\infty} |F(\omega + n\omega_b)|^2 \delta(\omega - \gamma - n\omega_b) & \text{for } C(1; T_b) = e^{j\gamma T_b}
\end{cases}
\]

Notice that the spectrum has discrete spectral lines as well as a distributed part when the characteristic function has unity magnitude.

The power spectrum for the case where \( p(t) \) is the rectangular pulse given by (20.4) and the frequency deviations are equally likely reduces to

\[
\frac{T_b}{A_c^2} S_{xx}(\omega) = \begin{cases} 
P(\omega) + 2\Re \left[ \frac{F^2(\omega)}{1 - C(1; T_b)e^{-j\omega T_b}} \right] & \text{for } h = \frac{2\omega_d}{\omega_b} \text{ not an integer} \\
P(\omega) - |F(\omega)|^2 + \omega_b \sum_{n=-\infty}^{\infty} |F(\omega + n\omega_b)|^2 \delta(\omega - \gamma - n\omega_b) & \text{for } h = \text{an integer } k
\end{cases}
\]

where

\[
\gamma = \begin{cases} 
0 & \text{for } k \text{ even} \\
\omega_b/2 & \text{for } k \text{ odd}
\end{cases}
\]

\[
F_n(\omega) = T_b \frac{\sin (\omega - \Omega_k)T_b}{2(\omega - \Omega_k)T_b} \frac{e^{-j(\omega - \Omega_k)T_b/2}}{2}
\]

\[
P(\omega) = \frac{T_b^2}{M} \sum_{k=0}^{M-1} \left[ \frac{\sin (\omega - \Omega_k)T_b}{(\omega - \Omega_k)T_b} \right]^2
\]

\[
F(\omega) = \frac{T_b}{M} \sum_{k=0}^{M-1} \frac{\sin (\omega - \Omega_k)T_b}{(\omega - \Omega_k)T_b} \frac{e^{-j(\omega - \Omega_k)T_b/2}}{2}
\]

and

\[
C(1; T_b) = \frac{2}{M} \sum_{k=1}^{M/2} \cos[\omega_d T_b(2k - 1)] = \frac{\sin(M\pi h)}{M\sin(h)}
\]

Notice that \( F_n(\omega) \) has its peak magnitude at the tone frequency \( \Omega_n = \omega_d[2n - (M - 1)] \) and zeros at multiples of the symbol rate, \( \omega_b \), away from the tone frequency. This is exactly what
would be expected for a burst of duration $T_b$ of a sinusoid at the tone frequency. The term $P(\omega)$ is what would result for the switched oscillator case when the phases of the oscillators are independent random variables uniformly distributed over $[0, 2\pi)$. The remaining terms account for the continuous phase property and give a narrower spectrum than if the the phase were discontinuous. The power spectrum has impulses at the $M$ tone frequencies when $h$ is an integer. However, the impulses at other frequencies disappear because they are multiplied by the nulls of $F(\gamma + n\omega_b)$.

Examples of the power spectral densities for binary continuous phase and switched oscillator FSK are show in the following four subfigures for $h = 0.5, 0.63, 1$ and 1.5. The spectrum for continuous phase FSK with $h = 0.63$ is quite flat for $-\omega_d < \omega < \omega_d$ and small outside this interval. The spectra become more peaked near the origin for smaller values of $h$. They become more and more peaked near $-\omega_d$ and $\omega_d$ as $h$ approaches 1 and include impulses at these frequencies when $h = 1$. The spectra for $M = 4$ continuous phase and switched oscillator FSK are shown in the next four subfigures for $h = 0.5, 0.63, 0.9$, and 1.5. FSK is called “narrow band FSK” for $h < 1$ and “wide band FSK” for $h \geq 1$. When Bell Laboratories designed its telephone line FSK modems, it avoided integer $h$ because the impulses in the spectrum caused cross-talk in the cables. It released the Bell 103 modem in 1962 which used binary FSK with $h = 2/3$ to transmit at 300 bits/second. The international ITU-T V.21 binary FSK modem recommendation uses the same $h$ and data rate. The CTM with $M = 4$ uses a symbol rate of 200 baud with a tone separation of 200 Hz and, thus, has the modulation index $h = 1$.

### 20.3 FSK Demodulation

Continuous phase FSK signals can be demodulated using a variety of methods including a frequency discriminator, a phase-locked loop, and tone filters with envelope detectors. A frequency discriminator works well when signal-to-noise ratio (SNR) is high but performs poorly when the SNR is low or the FSK signal has been distorted by a cell phone speech code, for example. A phase-locked loop performs better at lower SNR but is not good when the FSK signal is present for short time intervals because a narrow-band loop takes a long time to acquire lock. Tone filters with envelope detection is theoretically the optimum noncoherent detection method when the FSK signal is corrupted by additive white Gaussian noise in terms of minimizing the symbol error probability. These demodulation methods are discussed in the following subsections.
20.3 FSK Demodulation

Figure 20.2: Normalized Power Spectral Densities $T_b S_{xx}(\omega)/A_c^2$ for Continuous Phase and Switched Oscillator Binary FSK for Several Values of $h$

### 20.3.1 An Exact Frequency Discriminator

A frequency discriminator using the complex envelope is presented in Chapter 8 and the discussion is repeated here for reference. The complex envelope of the FM signal is

$$x(t) = s_+(t)e^{-j\omega_c t} = A_c e^{\int_0^t m(\tau) d\tau} e^{j\phi_0} = s_I(t) + j s_Q(t) \quad (20.33)$$

The angle of the complex envelope is

$$\tilde{\phi}(t) = \arctan[s_Q(t)/s_I(t)] = \int_0^t m(\tau) d\tau + \phi_0 \quad (20.34)$$
The derivative of this angle is

$$\frac{d}{dt} \tilde{\phi}(t) = \frac{s_I(t) \frac{d}{dt} s_Q(t) - s_Q(t) \frac{d}{dt} s_I(t)}{s_I^2(t) + s_Q^2(t)} = m(t)$$  \hspace{1cm}(20.35)$$

which is the desired message signal.

A block diagram for implementing this discriminator is shown in Figure 20.4. First the pre-envelope is formed and demodulated to get the complex envelope whose real part is the inphase component and imaginary part is the quadrature component. The inphase and quadrature components are both lowpass signals. The frequency response of the differentiators must approximate $j\omega$ over a band centered around $\omega = 0$ out to the cut-off frequency for the I and Q components which will be somewhat greater than the maximum frequency.
deviation $\omega_d(M - 1)$. The differentiator amplitude response should fall to a small value beyond the cut-off frequency because differentiation emphasizes high frequency noise which can cause a significant performance degradation. Also a wide band differentiator can cause large overshoots at the symbol boundaries where the tone frequencies change. If the differentiators are implemented as FIR filters, their amplitude responses will automatically pass through 0 at the origin and excellent designs can be achieved. Notice how the delays through the Hilbert transform filter and differentiation filter are matched by taking signals out of the center taps. The denominator $s^2_I(t) + s^2_Q(t)$ is the squared envelope of the the FSK signal and is just the constant $A^2_c$. Therefore, division by this constant at the discriminator output can be ignored with appropriate scaling of the FSK discriminator output level decision thresholds.

An example of the discriminator output is shown in Figure 20.5 when $f_c = 4000$ Hz, $f_d = 200$ Hz, and $f_b = 400$ Hz, so the modulation index is $h = 1$. The tone frequency deviations alternate between 200 and $-200$ Hz for eight symbols followed by two symbols with $-200$ Hz deviation.

### 20.3.1.1 Symbol Clock Acquisition and Tracking

The discriminator output must be sampled once per symbol at the correct time to estimate the transmitted frequency deviation and, hence, the input data bit sequence. The discriminator output will look like an $M$-level PAM signal with rapid changes at the symbol
boundaries where the frequency deviation has changed. The symbol clock must be acquired and tracked because there will be a phase difference between the transmitter and receiver symbol clocks and the two clocks can also differ slightly in frequency because of hardware differences. There are many ways to generate the symbol clock. When the signal-to-noise ratio is large at the receiver, the sharp transitions in the discriminator output can be detected. A method for doing this is to form the absolute value of the derivative of the discriminator output. This will generate a positive pulse whenever the output level changes. A pulse location can be determined by looking for a positive threshold crossing. Then the symbol can be sampled in its middle by waiting for half the symbol period, $T_b/2$, after the pulse detection before sampling the discriminator output level. The absolute value of the derivative will be very small in the middle of the symbol and a search for the next peak can be started. The derivative will be zero at the symbol boundaries where the levels do not change. Therefore, the search for a new peak should only extend for slightly more than $T_b/2$. If no new peak is found by that time then successive symbol levels are the same and the start of the next symbol should be estimated as the sampling time in the middle of the last symbol plus $T_b/2$. This process can then be repeated for each successive symbol. This approach assumes that

Figure 20.5: Discriminator Output for $h = 1$
the transmitter and receiver symbol clock frequencies are close and it will track small clock frequency differences.

In lower SNR environments, the method for generating a symbol clock signal for PAM signals discussed in Chapter 11 can be used. This involves passing the discriminator output through a bandpass filter with center frequency at \( f_b/2 \), squaring the filter output, and passing the result through a bandpass filter with a center frequency at the symbol rate \( f_b \). The receiver can then lock to the positive zero crossings of the resulting clock signal and sample the discriminator output with an appropriate delay from the zero crossings.

### 20.3.2 A Simple Approximate Frequency Discriminator

A simpler approximate discriminator will be derived in this subsection. Let \( 1/T = f_s \) be the sampling rate. Usually there will be multiple samples per symbol so \( T << T_b \). Using the complex envelope the following product can be formed:

\[
c(nT) = \frac{1}{A^2cT} \text{Im} \left\{ x(nT)x(nT-T) \right\} = \frac{1}{T} \text{Im} \left\{ e^{j\int_0^{nT} m(\tau) d\tau + \phi_0} e^{-j\int_0^{nT-T} m(\tau) d\tau + \phi_0} \right\}
\]

\[
= \frac{1}{T} \text{Im} \left\{ e^{j\int_0^{nT} m(\tau) d\tau} \right\} = \frac{1}{T} \sin \int_{nT-T}^{nT} m(\tau) d\tau
\]

\[
= \frac{1}{T} \sin[Tm(nT-T)] = \frac{1}{T} \sin[m(nT-T)/f_s] \approx m(nT-T)
\] (20.36)

To get the final result, it was assumed that the peak frequency deviation is significantly less than the sampling rate, and the approximation \( \sin x \approx x \) for \( |x| << 1 \) was used. In terms of the inphase and quadrature components

\[
c(nT) = \frac{1}{A^2cT} [s_Q(nT)s_I(nT-T) - s_I(nT)s_Q(nT-T)]
\] (20.37)

and this is the discriminator equation that would be implemented in a DSP.

As another approach, suppose the derivatives in (20.35) are approximated at time \( nT \) by

\[
\frac{d}{dt} s_I(t)|_{t=nT} \approx \frac{s_I(nT) - s_I(nT-T)}{T} \quad \text{and} \quad \frac{d}{dt} s_Q(t)|_{t=nT} \approx \frac{s_Q(nT) - s_Q(nT-T)}{T}
\] (20.38)

Substituting these approximate derivatives into (20.35) gives \( \frac{d}{dt} \tilde{\phi}(t)|_{t=nT} \approx c(nT) \) exactly as in the previous approach.

### 20.3.3 The Phase-Locked Loop

The block diagram of a phase-locked loop (PLL) that can be used to demodulate a continuous phase FSK signal is shown in Figure 20.6. The theory for this PLL is discussed extensively in Chapter 8 and the main points are summarized in this subsection.

First, the received FSK signal is sampled with period \( T \) and passed through a discrete-time Hilbert transform filter to form the pre-envelope \( s_+(nT) = s(nT) + j\tilde{s}(nT) \). Suppose
Continuous-Phase Frequency Shift Keying (FSK)

Figure 20.6: Phase-Locked Loop for FSK Demodulation

there are \( L \) samples per baud so that \( T_b = LT \). Then for \( iT_b \leq nT < (i + 1)T_b \), \( n = iL + \ell \) for some integer \( \ell \) with \( 0 \leq \ell \leq L - 1 \). From (20.6) and (20.10) it follows that the total phase angle of the pre-envelope during baud \( i \) is

\[
\Theta(nT) = \omega_c nT + T_b \sum_{k=0}^{i-1} \Omega(k) + \Omega(i)\ell T + \phi_0 \quad \text{for} \quad 0 \leq \ell \leq L - 1
\]  

(20.39)

The PLL contains a voltage controlled oscillator (VCO) which generates a complex exponential sinusoid at the carrier frequency \( \omega_c \) when its input is zero. The PLL acts to make the VCO total angle \( \phi(nT) = \omega_c nT + \theta_1(nT) \) equal to the angle of the pre-envelope. The multiplier in the Phase Detector box demodulates the pre-envelope using the replica complex exponential carrier generated by the VCO. The phase error between the angles of the pre-envelope and replica carrier is computed by the \( \text{atan2} \) function \( \text{atan2}(y, x) \) where \( y \) is the imaginary part of the multiplier output and \( x \) is its real part. The parameters \( \alpha \) and \( \beta \) in the Loop Filter are positive constants. Typically, \( \beta < \alpha/50 \) to make the loop have a transient response to a phase step without excessive overshoot. The accumulator generating \( \sigma(nT) \) is included so that the loop will track a carrier frequency offset. The parameter \( k_v \) is also a positive constant. The product, \( \alpha k_v T \), controls the tracking speed of the loop. It should be large enough so the loop tracks the input phase changes, but small enough so the loop is stable and not strongly influenced by additive input noise.

The VCO generates its phase angle by the following recursion:

\[
\phi(nT + T) = \phi(nT) + \omega_c T + k_v T y(nT)
\]  

(20.40)
Therefore
\[ k_v y(nT) = \frac{\phi(nT + T) - \phi(nT)}{T} - \omega_c \] (20.41)

During baud \( i \) and assuming the loop is perfectly in lock so that \( \Theta(nT) = \phi(nT) \), substituting \( \Theta(nT) \) given by (20.39) for \( \phi(nT) \) into (20.41) gives
\[ k_v y(nT) = \frac{\Theta(nT + T) - \Theta(nT)}{T} - \omega_c = \Omega(i) \] (20.42)

Therefore, the PLL is an FSK demodulator.

When the loop is in lock and the phase error is small, \( \text{atan2}(x, y) \) can be closely approximated by the imaginary part of the complex multiplier output divided by \( A_c \). The multiplier output is
\[ [s(nT) + j\hat{s}(nT)]e^{-j\phi(nT)} = A_c e^{j[\omega_c nT + \theta_m(nT)]}e^{-j\phi(nT)} = A_c e^{j[\phi_m(nT) - \theta_1(nT)]} \] (20.43)
and its imaginary part is
\[ \hat{s}(nT) \cos \phi(nT) - s(nT) \sin \phi(nT) = A_c [\phi_m(nT) - \theta_1(nT)] \simeq A_c [\phi_m(nT) - \theta_1(nT)] \] (20.44)

where \( A_c = |s(nT) + j\hat{s}(nT)| \). The imaginary part can be divided by the computed \( A_c \) or this scaling can be accomplished by an automatic gain control (AGC) in the receiver or by adjusting the loop parameters. The loop gain in the PLL and, hence, its transient response depend on \( A_c \), if the approximation (20.44) is used, so this normalization by \( A_c \) is important. The \( \text{atan2}(y, x) \) function automatically does the normalization.

An example of the PLL behavior is shown in Figure 20.7 for a binary FSK input signal. The binary data input to the modulator was a PN sequence generated by a 23-stage feedback shift register. The carrier frequency was 4 kHz, the frequency deviation was 200 Hz, and the baud rate was 400 Hz. The output shows a segment where the input alternated between 0 and 1 followed by a string of 1’s.

### 20.3.4 Optimum Noncoherent Detection by Tone Filters

The FM discriminator performs very poorly when the SNR is low or the FSK signal is distorted, for example, by a speech compression codec in a cell phone because differentiation emphasizes noise. The phase-locked loop demodulator performs better than the discriminator at low SNR but can have difficulty locking on to FSK signals that are present in short bursts. A better detector for these cases that uses “tone filters” is described in this section. This approach does not use knowledge of the carrier phase and is called noncoherent detection.

A result in detection theory is that in the presence of additive white Gaussian noise the detection strategy that is optimum in the sense of minimizing the symbol error probability for symbol interval \( N \) is to compute the following statistics for the symbol interval and decide
Figure 20.7: PLL Output with \( 1/T = 16000 \text{ Hz}, k_v = 1, \alpha k_v T = 0.2, \) and \( \beta = \alpha/100 \)

that the frequency that was transmitted corresponds to the largest statistic\(^2\):

\[
I_k(N) = \left[ \int_{NT_b}^{(N+1)T_b} s(t) \cos(\Lambda_k t + \epsilon) \, dt \right]^2 + \left[ \int_{NT_b}^{(N+1)T_b} s(t) \sin(\Lambda_k t + \epsilon) \, dt \right]^2
\]

\[
= \left[ \int_{NT_b}^{(N+1)T_b} s(t) e^{-j(\Lambda_k t + \epsilon)} \, dt \right]^2 = \left[ \int_{NT_b}^{(N+1)T_b} s(t) e^{-j\Lambda_k t} \, dt \right]^2 \text{ for } k = 0, \ldots, M-1 \quad (20.46)
\]

where \( s(t) \) is the noise corrupted received signal and \( \epsilon \) is a conveniently selected phase angle. Notice that the statistics have the same value for every choice of \( \epsilon \). Remember that \( \Lambda_k = \omega_c + \Omega_k = \omega_c + \omega_d [2k - M - 1] \) is the total tone frequency.

The statistic, $I_k(N)$, can be computed in several ways. The receiver could implement (20.45) in the obvious way. It could have a set of $M$ oscillators, each generating an inphase sine wave $\cos \Lambda_k t$ and a quadrature sine wave $\sin \Lambda_k t$. Then it would multiply the input, $s(t)$, by the sine waves, integrate the products over each baud, and form the sum of the squares of the inphase and quadrature integrator outputs for each tone frequency. The receiver would then decide that the tone frequency corresponding to the largest statistic was the one that was transmitted for that baud.

The statistics can also be generated using a bank of filters. Let the impulse response of the $k$th tone filter be

$$h_k(t) = \begin{cases} e^{j\Lambda_k t} & \text{for } 0 \leq t < T_b \\ 0 & \text{elsewhere} \end{cases}$$ (20.47)

The output of this filter when the input is $s(t)$ is

$$y_k(t) = \int_{t-T_b}^{t} s(\tau)e^{j\Lambda_k(t-\tau)} d\tau = \int_{t-T_b}^{t} s(\tau)e^{-j\Lambda_k \tau} d\tau e^{j\Lambda_k t}$$ (20.48)

Let $t = (N+1)T_b$ which is at the start of symbol interval $N+1$ or the end of symbol interval $N$. Then

$$|y_k(NT_b + T_b)|^2 = \left| \int_{NT_b}^{(N+1)T_b} s(\tau)e^{-j\Lambda_k \tau} d\tau \right|^2$$ (20.50)

which is the desired statistic $I_k(N)$ given by (20.46).

The frequency response of a tone filter is

$$H_k(\omega) = \int_{0}^{T_b} e^{j\Lambda_k t} e^{-j\omega t} dt = \frac{1 - e^{-j(\omega - \Lambda_k)T_b}}{j(\omega - \Lambda_k)} = e^{-j(\omega - \Lambda_k)T_b/2} T_b \frac{\sin[(\omega - \Lambda_k)T_b/2]}{(\omega - \Lambda_k)T_b/2}$$ (20.51)

The magnitude of this function has a peak at the tone frequency $\Lambda_k$ and zeros spaced at distances that are integer multiples of $\omega_b = 2\pi/T_b$ away from the peak. Thus, the tone filters are bandpass filters with center frequencies equal to the $M$ tone frequencies.

### 20.3.4.1 Discrete-Time Implementation

Discrete-time approximations to the statistics must be used in DSP implementations. The integrals can be approximated by sums. Suppose there are $L$ samples per symbol so that $T_b = LT$. Then the last term on the right of (20.46) can be approximated by

$$I_k(N)/T^2 \simeq D_k(N) = \sum_{\ell=NL}^{(N+1)L-1} s(\ell T)e^{-j\Lambda_k \ell T}$$ (20.52)
A discrete-time approximation to the tone filter is
\[
h_k(nT) = \begin{cases} 
e^{j\Lambda_k nT} & \text{for } n = 0, 1, \ldots, L - 1 \\ 0 & \text{elsewhere} \end{cases} \quad (20.53)
\]
and the output of this filter is
\[
y_k(nT) = \sum_{\ell=n-L+1}^{n} s(\ell T) e^{j\Lambda_k (n-\ell)T} = e^{j\Lambda_k nT} \sum_{\ell=n-L+1}^{n} s(\ell T) e^{-j\Lambda_k \ell T} \quad (20.54)
\]
Notice that each of the \( M \) tone filter impulse responses is convolved with the same set of \( L \) samples \( \{s(\ell T)\}_{\ell=n-L+1}^{n} \). Therefore, an efficient implementation in terms of minimum memory usage should have only one delay line containing these \( L \) samples.

The decision statistics for symbol interval \( N \) are obtained at the end of this interval by letting \( n = (N+1)L - 1 \) in (20.54) to get
\[
|y_k[(N+1)LT - T]|^2 = \left| e^{j\Lambda_k [(N+1)L-1]T} \sum_{\ell=NL}^{(N+1)L-1} s(\ell T) e^{-j\Lambda_k \ell T} \right|^2 = D_k(N) \quad (20.55)
\]
The frequency response for tone filter \( k \) is
\[
H_k(\omega) = \sum_{n=0}^{L-1} e^{j\Lambda_k nT} e^{-j\omega nT} = e^{-j(\omega - \Lambda_k)(L-1)T/2} \frac{\sin[(\omega - \Lambda_k)LT/2]}{\sin[(\omega - \Lambda_k)T/2]} \quad (20.56)
\]
The amplitude response of this filter has a peak of value \( L \) at the tone frequency \( \Lambda_k \) and zeros at frequencies \( \Lambda_k + p\omega_b \) in the interval \( 0 \leq \omega < \omega_s \) where \( p \) is an integer and \( \omega_b = 2\pi/T_b \). It repeats periodically outside this interval as would be expected for the transform of a sampled signal. This tone filter is a bandpass filter centered at the tone frequency \( \Lambda_k \).

The block diagram of a receiver using tone filters is shown in Figure 20.8 for \( M = 4 \). The boxes labelled “Complex BPF” are the tone filters. The solid line at the output of a box is the real part of the output and the dotted line is the imaginary part. The boxes labelled \(| \cdot |\) form the squared complex magnitudes of their inputs. These squared magnitudes are the squared envelopes of the tone filter outputs. The squared envelopes are sampled at the end of each symbol period, the largest is found, and the corresponding frequency deviation is assumed to be the one that was actually transmitted. This decision is then mapped back to the corresponding bit pattern.

If the receiver has locked its local symbol clock frequency to that of the received signal and the phase for sampling at the end of a symbol has been determined, then the convolution sum in (20.54) only has to be computed at the sampling times. In between sampling times, new samples must be shifted into the filter delay line but the output does not have to be computed. In practice the clocks will continually drift and must be tracked. The block diagram indicates that the tone filter outputs are computed for each new input sample. It will be shown below how a signal for clock tracking can be derived from these signals.
20.3.4.2 Recursive Implementation of the Tone Filters

The tone filters can be efficiently implemented recursively when their outputs must be computed at every sampling time. To ensure stability of the recursion, the tone filter impulse responses will be slightly modified to \( g_k(nT) = r^n h_k(nT) \) where \( r \) is slightly less than 1. The z-transform of a modified tone filter impulse response is

\[
G_k(z) = \sum_{n=0}^{L-1} r^n e^{j\Lambda_k nT} z^{-n} = \frac{1 - r^L e^{j\Lambda_k LT} z^{-L}}{1 - r e^{j\Lambda_k T} z^{-1}} \tag{20.57}
\]

The output of this modified tone filter can be computed as

\[
y_k(nT) = s(nT) - r^L e^{j\Lambda_k LT} s(nT - LT) + r e^{j\Lambda_k T} y_k(nT - T) \tag{20.58}
\]

The real part of \( y_k(nT) \) is

\[
v_{k,r}(nT) = \Re\{y_k(nT)\} = s(nT) - r^L \cos(\Lambda_k LT) s(nT - LT) + r \cos(\Lambda_k T) v_{k,r}(nT - T) - r \sin(\Lambda_k T) v_{k,i}(nT - T) \tag{20.59}
\]

and the imaginary part is

\[
v_{k,i}(nT) = \Im\{y_k(nT)\} = -r^L \sin(\Lambda_k LT) s(nT - LT) + r \cos(\Lambda_k T) v_{k,i}(nT - T) + r \sin(\Lambda_k T) v_{k,r}(nT - T) \tag{20.60}
\]

These last two equations are what could actually be implemented in a DSP since additions and multiplications must operate on real quantities in a DSP. This filter structure is sometimes called a “cross-coupled” implementation.
The quantities \( r^L \cos(\Lambda_k LT) \), \( r^L \sin(\Lambda_k LT) \), \( r \cos(\Lambda_k T) \), and \( r \sin(\Lambda_k T) \) can be precomputed. Then, computation of the real and imaginary outputs for the cross-coupled form requires six real multiplications and five real additions for each \( n \). Computation by direct convolution requires \( 2(L - 1) \) real multiplications and \( 2(L - 1) \) real additions for each \( n \) since \( h_k(0) = g_k(0) = 1 \) and this is usually much larger than the computation required for the cross-coupled form.

The signal memory required for the cross-coupled form is an \( L + 1 \) word buffer to store the real values \( \{ s(\ell T) \}_{\ell=-L}^n \) plus two locations to store \( v_{k,r}(nT - T) \) and \( v_{k,i}(nT - T) \). This is just slightly more than required by the direct convolution method.

### 20.3.4.3 Simplified Demodulator for Binary FSK

The demodulator structure can be simplified for binary (\( M = 2 \)) FSK. A block diagram of the simplified demodulator is shown in Figure 20.9. The squared envelopes of the two tone filter outputs are computed as before but now one is subtracted from the other. This difference is passed through a lowpass filter to smooth it and eliminate some noise. The slicer hard limits its input to a positive voltage \( A \) if its input is positive and to a negative voltage \( -A \) if its input is negative. When no noise is present on the transmission channel, a slicer output of \( A \) indicates that the frequency deviation \( \Omega_1 \) was transmitted and an output of \( -A \) indicates that \( \Omega_0 \) was transmitted during the symbol interval.

![Simplified FSK Demodulator Using Tone Filters for \( M = 2 \)](image)

### 20.3.4.4 Generating a Symbol Clock Timing Signal

In a low noise environment and when the system filters are wideband, the symbol clock can be tracked by locking to the sharp transitions in the demodulator output. This will not work in a high noise environment and when the system filters cause gradual transitions. One way to generate a signal for clock tracking in this latter case is to form the sum, \( c(nT) \), of the \( \binom{M}{2} = M(M - 1)/2 \) absolute values of the differences of the pairs of different tone filter output squared envelopes. In equation form

\[
c(nT) = \sum_{0 \leq i < j \leq M-1} |e_i(nT) - e_j(nT)|
\]  

(20.61)
The idea behind this signal is that during each symbol where the tone frequency changes from the one in the previous symbol, the tone filter output for the previous tone will ring down and the tone filter output for the new tone will ring up, so the absolute value of the difference will show a transition. The tone filter envelopes can then be sampled at the peaks of \( c(nT) \), the largest envelope determined, and the result mapped back to a data bit sequence.

The presence of an FSK signal can be detected by monitoring the sum of the \( M \) squared envelopes

\[
\rho(nT) = \sum_{k=0}^{M-1} e_k(nT)
\]

(20.62)

This sum indicates the power received in the tone filter pass bands. Detection of an FSK signal can be declared when \( \rho(nT) \) exceeds a threshold for one or more samples. The termination of an FSK signal can be declared when the sum falls below a threshold. The termination threshold can be set below the detection threshold to allow hysteresis.

When the tone frequency is the same in adjacent symbols, the tone filter output envelopes will not change and \( c(nT) \) will not have a transition between the symbols. A symbol clock tracking algorithm based on \( c(nT) \) would have to “flywheel” through the symbol intervals where \( c(nT) \) has no transitions. A solution to this problem is to pass \( c(nT) \) through a bandpass filter centered at the symbol rate \( f_b \). A simple 2nd order bandpass filter with nulls at 0 and \( f_s/2 \) Hz and a peak near \( f_b \) Hz has the transfer function

\[
H(z) = (1 - r) \frac{1 - z^{-2}}{1 - 2z^{-1}r \cos(2\pi f_b/f_s) + r^2z^{-2}}
\]

(20.63)

where \( f_s \) is the sampling rate and \( r \) is a number close to but slightly less than 1. The closer \( r \) is to 1, the more narrow the filter bandwidth. Let \( c(nT) \) be the filter input, \( y(nT) \) the filter output, and \( v(nT) \) an internal filter signal. Then the filter output can be computed recursively by the equations

\[
v(nT) = (1 - r)c(nT) + 2r \cos(2\pi f_b/f_s)v(nT - T) - r^2v(nT - 2T)
\]

(20.64)

\[
y(nT) = v(nT) - v(nT - 2T)
\]

(20.65)

This filter will “ring” at the symbol clock frequency. The receiver’s clock tracker can lock to the positive zero crossings of this signal. The slope of the filter output \( y(nT) \) is a maximum at the zero crossings. Therefore, the zero crossings can be determined with significantly higher accuracy than the peaks where the slope is zero. The peaks of the tone filter envelopes will occur with some delay from these zero crossings depending on the filter parameters. The tone filter squared envelopes should be sampled with this delay from the zero crossings. The bandpass filter output will continue to oscillate at the symbol rate but will decay exponentially through intervals where the input is constant because of no tone frequency changes. By choosing \( r \) close enough to 1, the output will remain large enough during the intervals with no transitions to still detect the positive zero crossings and allow the clock tracker to automatically flywheel through these intervals.
An $M = 4$ FSK Example Using Tone Filters

Typical signals for an $M = 4$ FSK signal with $f_d = 200$ Hz, $f_c = 4000$ Hz, $f_b = 400$ Hz, and $f_s = 16000$ Hz and tone filter detection are shown in Figures 20.10, 20.11, 20.12, 20.13, and 20.14. The tone frequencies are 3400, 3800, 4200, and 4600 Hz. For the tone filters $r = 0.999$ and for the clock bandpass filter $r = 0.998$.

Figure 20.10 shows a small segment of the FSK signal. The tone frequency for symbols during normalized times 10 to 11 and 12 to 13 is 4600 Hz. The tone frequency during times 11 to 12 and 13 to 14 is 3400 Hz. The varying amplitudes is an illusion created by connecting samples of the signal taken at a 16000 Hz rate with straight lines.

Figure 20.11 shows the squared envelope $e_0(nT)$ at the output of the 3400 Hz tone filter. Notice that the peaks occur at the integer normalized times.

Figure 20.12 shows a segment of the preliminary clock signal $c(nT)$ computed as

$$c(nT) = |e_0(nT) - e_1(nT)| + |e_0(nT) - e_2(nT)| + |e_0(nT) - e_3(nT)|$$
$$+ |e_1(nT) - e_2(nT)| + |e_1(nT) - e_3(nT)| + |e_2(nT) - e_3(nT)|$$  \hspace{1cm} (20.66)
The tone frequencies for symbols 10 through 21 alternate between 3400 and 4600 Hz creating peaks in $c(nT)$ each symbol as the envelopes of the corresponding two tone filters charge up and down. The tone frequency remains constant during symbols 22, 23, and 24, so there is no change in the tone filter output squared envelopes and $c(nT)$ had no transitions. Observe that the peaks occur at the integer normalized times which are exactly where the peaks in $e_0(nT)$ occur in Figure 20.11. One could lock to the peaks in $c(nT)$ and sample the envelopes at the peak times but would have to “flywheel” through intervals when the tone frequency does not change.

Figure 20.13 shows the result when $c(nT)$ is passed through the bandpass filter centered at the symbol clock frequency. This signal oscillates at the symbol clock frequency. Notice that the signal is exponentially damped between normalized times 22 and 30. This corresponds to an interval when $c(nT)$ has no transitions.

Figure 20.14 shows a segment of the preliminary clock signal, $c(nT)$, and the bandpass filtered clock signal, $y(nT)$, superimposed on the same graph. The peaks of $y(nT)$ occur at almost the same times as the peaks in $c(nT)$. The positive zero crossings of $y(nT)$ occur 1/4 of a symbol before the peaks of $c(nT)$. A good clock tracker would lock to these zero crossing and the receiver would then sample the tone filter squared envelopes with a delay of 1/4 of
Continuous-Phase Frequency Shift Keying (FSK)

Figure 20.12: The Preliminary Symbol Clock Tracking Signal \( c(nT) \)

A symbol which corresponds to the peaks of \( c(nT) \). The exact delay necessary depends on the filter parameters.
Figure 20.13: The Signal $c(nT)$ Passed through a 2nd Order Bandpass Filter
Figure 20.14: Superimposed Preliminary and Bandpass Filtered Clock Signals
20.4 Symbol Error Probabilities for FSK Receivers

The problem of computing the symbol error probability for different types of FSK receivers is discussed extensively in Chapter 8 of Lucky, Salz, and Weldon. The problem is very difficult because of the nonlinear natures of the modulator and various receivers. Many of the results are approximations or require evaluation of complicated integrals by numerical integration.

20.4.1 Orthogonal Signal Sets

One case where exact closed form results are know is when the transmitted symbols are orthogonal, they are corrupted by additive white Gaussian noise, and optimum noncoherent detection by tone filters is used. Two continuous-time signals over the interval \([t_1, t_2]\) with complex envelopes \(x_1(t)\) and \(x_2(t)\) are said to be orthogonal if

\[
\rho = \int_{t_1}^{t_2} x_1(t)x_2^*(t) \, dt = 0
\]  

(20.67)

From (20.13) it follows that the complex envelopes of the FSK signal set during symbol period \(i\) where \(iT_b \leq t < (i+1)T_b\) are

\[
x_k(t) = A_c e^{j\theta_m(iT_b)} e^{j\omega_d[2k-(M-1)](t-iT_b)} \quad \text{for } k = 0, \ldots, M - 1
\]

(20.68)

For two distinct integers \(k_1\) and \(k_2\) in this set

\[
\rho = \int_{iT_b}^{(i+1)T_b} x_{k_1}(t)x_{k_2}^*(t) \, dt = A_c^2 \int_{iT_b}^{(i+1)T_b} e^{j2\omega_d(k_1-k_2)(t-iT_b)} \, dt = A_c^2 \int_0^{T_b} e^{j2\omega_d(k_1-k_2)t} \, dt
\]

\[
= A_c^2 \frac{e^{j2\omega_d(k_1-k_2)T_b} - 1}{j2\omega_d(k_1-k_2)}
\]

(20.69)

This integral will be zero if \(2\omega_d(k_1-k_2)T_b = 2\pi \ell\) or \(h(k_1-k_2) = \ell\) where \(\ell\) is an integer. This will be satisfied for all pairs of signals in the FSK signal set when the modulation index, \(h\), is an integer.

An analogous property holds for the discrete-time FSK approximation. Assume there are \(L\) samples per symbol so that \(T_b = LT\). The complex discrete-time envelopes during symbol interval \(i\) where \(iT_b \leq nT < (i+1)T_b\) are

\[
x_k(nT) = A_c e^{j\theta_m(iT_b)} e^{j\omega_d[2k-(M-1)](nT-iLT)} \quad \text{for } k = 0, \ldots, M - 1
\]

(20.70)

Then for two distinct integers \(k_1\) and \(k_2\) the correlation is

\[
\rho = \sum_{n=1}^{(i+1)L-1} x_{k_1}(nT)x_{k_2}^*(nT) = A_c^2 \sum_{n=0}^{L-1} e^{j2\omega_d(k_1-k_2)nT} = A_c^2 \frac{1 - e^{j2\omega_d(k_1-k_2)LT}}{1 - e^{j2\omega_d(k_1-k_2)T}}
\]

(20.71)

The correlation $\rho$ will be zero if $2\omega_d(k_1 - k_2)LT = 2\omega_d(k_1 - k_2)T_b = 2\pi\ell$ where $\ell$ is an integer just as in the continuous-time case. Therefore, all pairs of discrete-time FSK signals in the set will be orthogonal if $h$ is an integer.

The energy transmitted during symbol period $i$ is

$$E = \int_{iT_b}^{(i+1)T_b} s^2(t) \, dt = \frac{1}{2} \int_{iT_b}^{(i+1)T_b} |x_k(t)|^2 \, dt = A_k^2 T_b / 2$$  \hspace{1cm} (20.72)

and the average power transmitted during this interval is $S = E / T_b$. Let the two-sided noise power spectral density be $N_0 / 2$. Then the symbol error probability is

$$P_e = \frac{\exp \left( -\frac{E}{N_0} \right)}{M} \sum_{i=2}^{M} (-1)^i \binom{M}{i} \exp \left( \frac{E}{iN_0} \right)$$ \hspace{1cm} (20.73)

For binary FSK, i.e., $M = 2$, the symbol error probability is

$$P_e = \frac{1}{2} \exp \left( -\frac{E}{2N_0} \right)$$ \hspace{1cm} (20.74)

An upper bound for the symbol error probability for arbitrary $M$ is

$$P_e \leq \frac{M - 1}{2} \exp \left( -\frac{E}{2N_0} \right)$$ \hspace{1cm} (20.75)

There are $k = \log_2 M$ bits per symbol. For orthogonal signal sets, all symbol errors are equally likely, so all bit-error patterns in a block of $k$ transmitted bits assigned to a symbol are equally likely. Based on this observation, Viterbi$^5$ shows that the bit error probability is related to the symbol error probability by the formula

$$P_b = \frac{2^{k-1}}{2^k - 1} P_e$$ \hspace{1cm} (20.76)

### 20.5 Experiments for Continuous-Phase FSK

For these experiments you will explore $M = 2$ and $M = 4$ continuous-phase FSK transmitters and receivers. For all these experiments use the following parameters: carrier frequency $f_c = 4000$ Hz, frequency deviation $f_d = 200$ Hz, symbol rate $f_b = 400$ Hz, sampling frequency $f_s = 16000$ samples per second, and $p(t)$ is the rectangular pulse given by (20.4). Initialize the TMS320C6713 DSK as usual.


20.5 Experiments for Continuous-Phase FSK

20.5.1 Theoretical FSK Spectra

Write a MATLAB program or use any other favorite programming language to compute the power spectral density for an FSK signal with arbitrary $M$, $f_c$, $f_d$, and $f_b$ using (20.27). Then plot the spectra for $M = 2$ and $M = 4$ vs. the normalized frequency $(\omega - \omega_c)/\omega_b$ for the parameters specified for these experiments. Experiment with other parameters also.

20.5.2 Making FSK Transmitters

Write programs for the TMS320C6713 DSK to implement continuous-phase FSK transmitters for $M = 2$ and $M = 4$. Write the output samples to the left codec output channel. You will be using these transmitters as FSK signal sources for your receivers.

20.5.2.1 Initial Handshaking Sequence

To help the receivers detect the presence of an FSK signal and lock to the transmitter’s symbol clock, make your transmitter send the following signal sequence:

1. First send 0.25 seconds of silence, that is, send 0 volts for 0.25 seconds. This will allow your receiver to skip over any initial transient that occurs when the transmitter program is loaded and started.

2. Then for $M = 2$ send 25 symbols alternating each symbol between $f_0 = 3800$ and $f_1 = 4200$ Hz tones. This will allow the receivers to detect the FSK signal and lock on to the symbol clock. For $M = 4$ send 25 symbols alternating each symbol between $f_0 = 3400$ and $f_3 = 4600$ Hz.

3. Suppose the frequencies of the last few symbols of the alternating sequence for $M = 2$ were $\cdots, f_0, f_1, f_0$. Next send an alternating frequency sequence for 10 symbols but with the alternation reversed. That is send $f_0, f_1, f_0, f_1, f_0, f_1, f_0, f_1$. Your receiver can detect this change in the alternations and use it as a timing mark to determine when actual data will start.

For the $M = 4$ transmitter change to alternating between $f_1 = 3800$ and $f_2 = 4200$ Hz for 10 symbols. Again, this change can be used as a timing mark.

20.5.2.2 Simulating Random Customer Data

After the alternations, begin transmitting “customer” data continuously. Simulate this data by using a 23 stage PN sequence generator as discussed in Chapter 9. Use the connection polynomial $h(D) = 1 + D^{18} + D^{23}$ so the data bit sequence, $d(n)$, is generated by the recursion

$$d(n) = d(n - 18) \oplus d(n - 23) \tag{20.77}$$

where “$\oplus$” in the recursion is modulo 2 addition, that is, the exclusive-or operation. Initialize the PN sequence generator shift register to some non-zero state.
For $M = 2$, shift the PN generator once to get a new data bit $d(n)$ which will be a 0 or 1. Map this bit to the tone frequency $\Lambda(n) = \omega_c + \omega_d[2d(n) - 1]$.

For $M = 4$, shift the register twice to get a pair of bits $[d_1(n), d_0(n)]$. Consider this bit pair to be the integer $k(n) = 2d_1(n) + d_0(n)$ which can be 0, 1, 2, or 3. Map this bit pair to the tone frequency $\Lambda(n) = \omega_c + \omega_d[2k(n) - 3]$.

### 20.5.2.3 Experimentally Measure the FSK Power Spectral Density

Measure the power spectral density of the transmitted FSK signals for $M = 2$ and 4 after the initial handshaking sequence when random customer data is being transmitted. If you made the spectrum analyzer for Chapter 4, run it on one station and connect your transmitter to it. You can use a commercial spectrum analyzer if it is available. Otherwise collect an array of transmitted samples, write them to a file on the PC, and use MATLAB’s Signal Processing Toolbox function `pwelch()` . Compare your measured spectra with the theoretical ones you computed.

### 20.5.3 Making a Receiver Using an Exact Frequency Discriminator

Make a receiver using the exact frequency discriminator shown in Figure 20.4 for $M = 2$. Connect the transmitter from another station to your receiver. There are RCA-to-RCA barrel connectors in the cabinet to connect RCA to mini-stereo cables together. First leave your transmitter off and turn on your receiver. When the receiver is running, turn on your transmitter. Your receiver program should do the following:

1. The receiver should detect the absence or presence of an input FSK signal by monitoring the received signal power. The power can be calculated by doing a running average of the squared input samples over several symbols. You can also try a single pole exponential averager. The receiver should assume that no FSK input signal is present when this power is small and sit in a loop checking for the presence of an input signal. When the measured power crosses a threshold, the receiver can start the discriminator and symbol clock tracking algorithm. You should predetermine a good threshold based on your knowledge of the transmitter amplitude and system gains. You can do this experimentally by observing the received power when the transmitter is running.

2. The receiver should continue to monitor the input signal power and detect when the signal is gone and go into a loop looking for the return of a signal.

3. Start your discriminator and symbol clock tracker once an input signal is detected. Monitor the tone frequency alternations and look for the alternation switch. Count for 10 symbols after the switch and begin detecting the tone frequencies resulting from the input customer data.
4. Send the output samples of the discriminator to the left codec channel. Send a signal to the right codec output channel that is a square wave at the symbol clock frequency to use for synching the oscilloscope. You can do this by sending a positive value for 20 samples at the start of a symbol followed by its negative value for the next 20 samples. Observe the result and take a picture of a single trace on the oscilloscope screen to show a typical output of the discriminator. Alternatively, you can capture an array of discriminator output samples with Code Composer or use fprintf(·) to write the array to a PC file and plot the output file with your favorite plotting program.

If you allow the oscilloscope to run freely, you will see multiple traces synchronized with the symbol clock overlapped on the screen. This type of display is called an “eye pattern” in the communications industry. At the end of each symbol you should see two distinct equal and opposite levels and the eye is said to be open. The eye pattern can be used as a diagnostic tool. Noise and system problems cause the eye to be less open. Decision error will occur if the eye is closed.

5. Map the detected tone frequency sequence back into a bit sequence $\hat{d}(n)$.

6. Check that the received bit stream is the same as the transmitted one. Your receiver should have a 24-stage shift register that contains $\hat{d}(n), \hat{d}(n - 1), \ldots, \hat{d}(n - 23)$. You can check for errors by checking that

$$\hat{d}(n) \oplus \hat{d}(n - 18) \oplus \hat{d}(n - 23) = 0$$

for all $n$ except for an initial burst of 1’s when the shift register is filling up. If you initialize the state of the register to the initial state of the transmitter register the result should be all 0’s if you have detected the starting time of the customer data correctly.

20.5.3.1 Running a Bit-Error Rate Test (BERT)

A measure of the quality of a digital transmission scheme is its bit-error rate performance in the presence of additive noise. Once your receiver is working correctly with noiseless received FSK signals, perform a bit-error rate test as follows:

1. Generate zero-mean Gaussian noise samples in the DSP with some variance $\sigma^2$ and add them to the received signal samples. Implement a power meter in your receiver program to measure the power of the FSK input samples, say $P$. Compute the SNR $= 10\log_{10}(P/\sigma^2)$ dB.

2. Your receiver should have a replica of the PN sequence generator in the transmitter. You should synchronize the state of the local PN generator to that of the transmitter so its output sequence will be in phase with the received one.

3. Start your BERT test with a very high SNR so few errors will occur. Check if each bit estimated by the receiver is the same as the transmitted one and count any bit errors
for a number of bits sufficient to give a good estimate of the bit-error probability. The estimated bit-error rate is \( \text{BER} = \frac{\text{the number of errors in the observed sequence of detected bits}}{\text{the number of observed detected bits}} \). The number of observed bits should be at least 10 times the bit-error rate to get an estimate with good accuracy. The variance of this estimator decreases inversely with the number of observed bits.

4. Decrease the SNR in steps of 0.25 dB and measure the bit-error rate for each SNR. Continue decreasing the SNR until you can no longer synchronize the replica PN generator with the transmitter’s generator.

5. Plot the bit-error rate vs. SNR. The bit-error rate should be plotted on a logarithmic scale and SNR in dB on a linear scale. This kind of plot is known as a “waterfall curve” in the communications industry.

20.5.4 Making a Receiver Using an Approximate Frequency Discriminator

Repeat the tasks for the exact frequency discriminator in Section 20.5.3 for the approximate frequency discriminator of Section 20.3.2.

20.5.5 Making a Receiver Using a Phase-Locked Loop

Make a receiver using the phase-locked loop described in Section 20.3.3 for \( M = 2 \). Test your receiver by following the steps in Section 20.5.3. Compare the bit-error rate performance of this receiver with the discriminator receiver.

20.5.6 Making a Receiver Using Tone Filters

20.5.6.1 \( M = 4 \) Tone Filter Receiver

Make a receiver using tone filters for \( M = 4 \) FSK. Implement the method for generating a symbol clock tracking signal described in Section 20.3.4.4. Use the sum of the tone filter output squared envelopes, \( \rho(nT) \) given by (20.62), to detect the presence or absence of a received FSK signal. Test your receiver by following the steps for the exact discriminator in Section 20.5.3. In addition, send the squared envelope at the output of one of the tone filters to one output channel of the codec, send the output of the 2nd order bandpass symbol clock tone generation filter to the other channel to use as a synch signal, observe them on the oscilloscope, and record the display. Also send the preliminary clock tracking signal to the oscilloscope and record the result. Compare your measured bit-error rate curve with the theoretical bit error probability curve given by (20.76). Also compare the bit-error rate performance of this receiver to the others.
20.5.6.2 Simplified $M = 2$ Tone Filter Receiver

Make and test the simplified binary tone filter receiver described in Section 20.3.4.3. Measure and plot the bit-error rate vs. SNR and compare it with the theoretical curve.
Chapter 21

Brief Introduction to Direct Sequence Spread Spectrum Systems

Spread spectrum systems were actively developed starting in the late 1940’s primarily for military applications. The main reason for using spread spectrum systems was their resistance to enemy jamming. Other reasons include low probability of detection when the spread power spectral density is down at ambient noise levels, significant difficulty for a hostile listener to detect the transmitted data, and low interference with other transmissions in the same band. Two basic types of spread spectrum systems have been manufactured: (1) frequency hopping systems and (2) direct sequence spread spectrum. In frequency hopping systems the carrier is pseudo-randomly jumped over a set of frequencies. Frequency hopping will not be discussed in this chapter. In direct sequence spread spectrum, a narrow band signal like BPSK has its spectrum spread by multiplying it by a significantly more rapidly varying pseudo-random bipolar signal. The receiver multiplies the received signal by a replica of the spreading signal to collapse the spectrum back to its original width.

More recently, direct sequence spread spectrum has been used for ranging. The Global Positioning System (GPS) uses it for medium precision public location determination and for high precision military location determination. NASA has used it in its Tracking and Data Relay Satellite System (TDRSS). Some cellular phone systems use direct sequence spread spectrum for multiple access. In these cellular systems, different users all transmit spread signals in the same band but use different spreading codes. This is called code division multiple access (CDMA).

In this chapter several types of spread spectrum transmission formats are presented. Methods for implementing receivers are discussed. In particular methods for synchronizing the replica spreading signal generated by the receiver with the spreading signal of the received signal are explained, demodulation of the despread received signal using a Costas loop is reviewed, and a method for tracking the data bit clock using the output of a matched filter is explained.
21.1 Direct Sequence Spread Spectrum Transmitters

Three different types of direct sequence spread spectrum transmitters are presented in this section for: (1) bipolar data and bipolar spreading, (2) bipolar data and quadrature phase shift spreading, and (3) two different bipolar data streams spread on quadrature carriers.

21.1.1 Bipolar Data with Bipolar Spreading

![Diagram of Bipolar Data and Bipolar Spreading]

The block diagram of a spread spectrum transmitter for the case where the original data signal is bipolar and the spreading signal is also bipolar is shown in Figure 21.1. A binary data source emits a sequence, \( d_0(n) \), of logical bits having the values 0 or 1 with frequency \( f_b \) bits per second (bps) or, equivalently, every \( T_b = 1/f_b \) seconds. These logical levels are converted to a sequence of analog levels, \( d(n) \), with a logical 0 mapped to 1 volt and a logical 1 mapped to -1 volt. This can be represented mathematically by the equation

\[
d(n) = 1 - 2d_0(n)
\]  

(21.1)

The sequence \( d(n) \) is said to be bipolar or bi-phase. We will assume that the output of the “Bipolar Mapper” remains constant during each data bit period. Let \( p_d(t) \) be a rectangular pulse with duration \( T_b \) and height 1. That is,

\[
p_b(t) = \begin{cases} 
1 & \text{for } 0 \leq t < T_b \\
0 & \text{elsewhere}
\end{cases}
\]  

(21.2)

Then the output of the Bipolar Mapper can be written as

\[
\rho(t) = \sum_{n=-\infty}^{\infty} d(n)p_d(t - nT_b)
\]  

(21.3)
The “Spreading Code Generator” produces a sequence, \( b_0(k) \), of pseudo-random logical bits at a rate \( f_c \) bps which is significantly larger than the data bit rate \( f_b \). Let \( T_c = 1/f_c \). In this chapter we will use the PN sequences studied in Chapter 9 as the spreading codes. We will also assume that \( L = f_c/f_b \) is an integer. The spreading code is mapped to a bipolar sequence \( b(k) = 1 - 2b_0(k) \) and converted into a continuous-time sequence by the rule

\[
c(t) = \sum_{k=-\infty}^{\infty} b(k)p_c(t - kT_c)
\]

(21.4)

where \( p_c(t) \) is the rectangular pulse

\[
p_c(t) = \begin{cases} 1 & \text{for } 0 \leq t < T_c \\ 0 & \text{elsewhere} \end{cases}
\]

(21.5)

The bandwidth of the data signal \( \rho(t) \) is spread by multiplying it by the faster varying bipolar signal \( c(t) \) to give the baseband bipolar signal \( \tilde{s}(t) = \rho(t)c(t) \). Starting at a data bit boundary, the values of \( \tilde{s}(t) \) remain constant over intervals of duration \( T_c \). Each of these sub-intervals of a data bit is called a chip. Thus there are \( L \) chips per bit.

Finally, the spread bipolar baseband signal is multiplied by the local oscillator signal \( A \cos \omega_0 t \) to generate the passband signal \( s(t) = \tilde{s}(t)A \cos \omega_0 t \). Its spectrum is centered around the carrier frequency \( \omega_0 \). The signal \( \tilde{s}(t) \) is 1 or \(-1\) and changes the carrier phase by 0 or 180 degrees. The transmitted signal is said to be binary phase shift keyed (BPSK).

![Figure 21.2: Efficient Transmitter for Bipolar Data and Spreading](image)

A simpler version of the spread spectrum transmitter is shown in Figure 21.2. Here the spreading is performed at the baseband logical signal level. The data sequence \( d_0(n) \) is exclusive-ored (XOR-ed) with the rapidly varying code sequence \( b_0(k) \) to “chop up” each data bit into \( L \) chips. XOR-ing \( d_0(n) \) with \( b_0(k) \) leaves \( d_0(n) \) unchanged when \( b_0(k) = 0 \) and complements \( d_0(n) \) when \( b_0(k) = 1 \). In the bipolar domain this maps to multiplying \( d(n) \) by 1 when \( b_0(k) = 0 \) and by \(-1\) when \( b_0(k) = 1 \). The resulting chips are then converted to a continuous-time signal by a bipolar mapper operating at the chip rate and the result BPSK modulates the carrier.
21.1.1.1 Spectrum of $s(t)$ in the Case of Ideal Binary Random Data and Spreading Codes

Consider the case where there are an integer number, $L$, of chips per data bits and the chips are aligned with the start of the data bits. Suppose the bipolar data bits are independent and equally likely to be $1$ or $-1$ and the bipolar spreading code values are also independent and equally likely to be $1$ or $-1$. Also assume the data and code sequences are statistically independent. Then it can be shown that the baseband spread signal $\tilde{s}(t) = \rho(t)c(t)$ is a binary random signal with bit period equal to the chip duration $T_c$. It is shown in most texts on communications that the random signal $\tilde{s}(t + \Theta)$ where $\Theta$ is a random variable uniformly distributed over $[0, T_c)$ has the triangular shaped autocorrelation function

$$R_{\tilde{s}}(\tau) = \mathbb{E}\{\tilde{s}(t + \tau + \Theta)\tilde{s}(t + \Theta)\} = \begin{cases} 1 - \frac{|	au|}{T_c} & \text{for } |	au| \leq T_c \\ 0 & \text{elsewhere} \end{cases} \quad (21.6)$$

The power spectral density for this signal is the Fourier transform of its autocorrelation function and is

$$S_{\tilde{s}}(\omega) = T_c \left(\frac{\sin \omega T_c/2}{\omega T_c/2}\right)^2 \quad (21.7)$$

This function has a peak value of $T_c$ for $\omega = 0$ and regularly spaced zeros at integer multiples of $2\pi/T_c = \omega_c$.

The original unspread bipolar data signal, $\rho(t)$, has a similar autocorrelation and power spectral density except with $T_c$ replaced by $T_b$. So

$$S_{\rho}(\omega) = T_b \left(\frac{\sin \omega T_b/2}{\omega T_b/2}\right)^2 \quad (21.8)$$

This function is $T_b$ at the origin and has regularly space zeros at integer multiples of $2\pi/T_b = \omega_b$. Thus the main lobe of $S_{\tilde{s}}(\omega)$ is $\omega_c/\omega_b = L$ times wider than the main lobe of $S_{\rho}(\omega)$.

If $\tilde{s}(t + \Theta)$ is translated up to passband by multiplying by the carrier $A \cos(\omega_0 t + \Phi)$ where $\Phi$ is a random variable uniformly distributed over $[0, 2\pi)$ and independent of the other random variables, the power spectral density of the transmitted signal $s(t + \Theta)$ can be shown to be

$$S_s(\omega) = \frac{A^2}{4} S_{\tilde{s}}(\omega - \omega_0) + \frac{A^2}{4} S_{\tilde{s}}(\omega + \omega_0) \quad (21.9)$$

Now consider the case where the data and spreading code bipolar signals are independent random process with code chips randomly phased relative to the data bits. Then the autocorrelation function of the baseband spread signal is

$$R_{\tilde{s}}(\tau) = R_{\rho}(\tau)R_{c}(\tau) \quad (21.10)$$

and its power spectral density is the frequency domain convolution

$$S_{\tilde{s}}(\omega) = \frac{1}{2\pi} \int_{-\infty}^{\infty} S_c(\lambda)S_{\rho}(\omega - \lambda) d\lambda \quad (21.11)$$
21.1 Direct Sequence Spread Spectrum Transmitters

The spreading code spectrum $S_c(\omega)$ is wideband with its main lobe extending from $-2\pi/T_c$ to $2\pi/T_c$. The data spectrum $S_\rho(\omega)$ is narrow band with its main lobe extending from $-2\pi/T_\rho$ to $2\pi/T_\rho$. The main contribution to $S_\tilde{s}(\omega)$ in the convolution is an average of $S_c(\lambda)$ over the interval $\omega - \omega_b < \lambda < \omega + \omega_b$ which corresponds to the narrow main lobe of $S_\rho(\omega - \lambda)$. Thus the convolution broadens the spreading code spectrum main lobe somewhat but its bandwidth is still close to that of the spreading code when $\omega_b << \omega_c$ and it has a shape close to $S_c(\omega)$.

21.1.1.2 Spectrum of a Maximal Length Bipolar PN Signal

Now suppose that the spreading binary code, $b_0(k)$, is a maximal length PN sequence of length $N$ as discussed in Chapter 9. Then the bipolar spreading signal $c(t)$ repeats with period $T_{pn} = NT_c$ or fundamental frequency $\omega_{pn} = 2\pi/T_{pn} = 2\pi f_c/N$ and has a Fourier series expansion of the form

$$c(t) = \sum_{n=-\infty}^{\infty} c_n e^{jn\omega_{pn}t} \quad (21.12)$$

Thus the spectrum of $c(t)$ consists of discrete lines at frequencies $n\omega_{pn}$ with powers $|c_n|^2$. The values of $c_n$ depend on $N$ and the particular PN sequence. However, a simple formula for $|c_n|^2$ can be found that only depends on $N$. The proof uses the following identity:

$$\sigma(t) = \sum_{n=-\infty}^{\infty} |c_n|^2 e^{jn\omega_{pn}t} = \frac{1}{T_{pn}} \int_{T_{pn}}^{0} c(t + \tau)\overline{c(\tau)} d\tau \quad (21.13)$$

which is the periodic autocorrelation function for $c(t)$. This correlation function repeats with period $T_{pn}$. One period extending over the interval $-T_{pn}/2 < t < T_{pn}/2$ has the offset triangular shape

$$\sigma(t) = \begin{cases} 
1 - \frac{N + 1}{N} \frac{|t|}{T_c} & \text{for } |t| < T_c \\
-1/N & \text{for } T_c \leq |t| < T_{pn}/2 = NT_c/2 
\end{cases} \quad (21.14)$$

This function is 1 for $t = 0$ and decreases linearly to $-1/N$ as $|t|$ increases to $T_c$ and remains at $-1/N$ for the remainder of the interval. The Fourier series coefficients for the periodic autocorrelation function are

$$|c_n|^2 = \begin{cases} 
\frac{N + 1}{N^2} \left[ \frac{\sin(n\pi/N)}{n\pi/N} \right]^2 & \text{for } n \neq 0 \\
\frac{1}{N^2} & \text{for } n = 0 
\end{cases} \quad (21.15)$$

There is a slight DC value of $1/N^2$ since one period of the PN sequence has one more 1 than 0.
21.1.2 Bipolar Data and QPSK Spreading

In Section 21.1.1 the carrier phase shifts only took the values 0 or 180 degrees. In this section the carrier phase shifts will be allowed to have the four values 45, 135, -135, and -45 degrees. This will be called “quadrature phase shift keying” (QPSK) spreading. It will be shown in Section 21.2 that QPSK spreading has better rejection of narrowband interference than BPSK spreading. Now two different bipolar spreading codes, \( b_1(k) \) and \( b_2(k) \) are required with the corresponding BPSK spreading signals

\[
c_1(t) = \sum_{k=-\infty}^{\infty} b_1(k)p_c(t-kT_c) \quad \text{and} \quad c_2(t) = \sum_{k=-\infty}^{\infty} b_2(k)p_c(t-kT_c) \quad (21.16)
\]

Since \( b_1(k) \) and \( b_2(k) \) can only have the values 1 and -1,

\[
B(k) = b_1(k) + jb_2(k) = \sqrt{2}e^{j\beta(k)}; \quad \beta(k) = \arctan[b_2(k)/b_1(k)] \quad (21.17)
\]

The four values of \( \beta(k) \) are shown in the following table.

<table>
<thead>
<tr>
<th>( b_1(k) )</th>
<th>( b_2(k) )</th>
<th>( \beta(k) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>( \pi/4 )</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>( 3\pi/4 )</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>( -3\pi/4 )</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>( -\pi/4 )</td>
</tr>
</tbody>
</table>

The complex QPSK spreading signal is

\[
C(t) = c_1(t) + jc_2(t) = \sum_{k=-\infty}^{\infty} [b_1(k) + jb_2(k)]p_c(t-kT_c) \quad (21.18)
\]

\[
= \sqrt{2} \sum_{k=-\infty}^{\infty} e^{j\beta(k)}p_c(t-kT_c) = \sqrt{2} e^{j\sum_{k=-\infty}^{\infty} \beta(k)p_c(t-kT_c)} \quad (21.19)
\]

\[
= \sqrt{2} e^{jB(t)} \quad (21.20)
\]

where the angle of the spreading signal is

\[
B(t) = \sum_{k=-\infty}^{\infty} \beta(k)p_c(t-kT_c) \quad (21.21)
\]

As before, let \( \rho(t) \) be the bipolar data signal. Then the complex envelope of the QPSK spread signal is

\[
\tilde{s}(t) = A\rho(t)[c_1(t) + jc_2(t)] = A\rho(t) \sum_{k=-\infty}^{\infty} [b_1(k) + jb_2(k)]p_c(t-kT_c) \quad (21.22)
\]

\[
= A\rho(t)\sqrt{2} e^{j\sum_{k=-\infty}^{\infty} \beta(k)p_c(t-kT_c)} \quad (21.23)
\]
One form of the pre-envelope of the spread signal is

\[ s_+(t) = \tilde{s}(t)e^{j\omega_0 t} = \rho(t)\sqrt{2}Ae^{j[\omega_0 t + \sum_{k=-\infty}^{\infty} \beta(k)p_{c}(t-kT_{c})]} \]  

(21.24)

and the transmitted spread signal is

\[ s(t) = \Re\{s_+(t)\} = \rho(t)\sqrt{2}A\cos \left[ \omega_0 t + \sum_{k=-\infty}^{\infty} \beta(k)p_{c}(t-kT_{c}) \right] \]  

(21.25)

These last two equations illustrate the QPSK nature of the spreading operation. A form of \( s(t) \) more convenient for implementation is

\[ s(t) = A\Re \left\{ \rho(t)e^{j\omega_0 t} \sum_{k=-\infty}^{\infty} [b^{(1)}(k) + jb^{(2)}(k)]p_{c}(t-kT_{c}) \right\} \nonumber \]

\[ = \left\{ \rho(t) \sum_{k=-\infty}^{\infty} b_1(k)p(t-kT_{c}) \right\} A\cos \omega_0 t 
- \left\{ \rho(t) \sum_{k=-\infty}^{\infty} b_2(k)p(t-kT_{c}) \right\} A\sin \omega_0 t \]  

(21.26)

\[ = \rho(t)c_1(t)A\cos \omega_0 t - \rho(t)c_2(t)A\sin \omega_0 t \]  

(21.27)

The first line of (21.26) is \( \rho(t) \) BPSK spread by \( c_1(t) \) and translated to passband by the carrier \( A\cos \omega_0 t \). The bottom line is \( \rho(t) \) BPSK spread by \( c_2(t) \) and translated to passband by the quadrature carrier \( A\sin \omega_0 t \). This is illustrated in Figure 21.3.
21.1.3 Two Different Data Streams Spread on Quadrature Carriers

Another variation that has been implemented is the transmission of two different spread data streams on quadrature carriers. Let $\rho_1(t)$ and $\rho_2(t)$ be two distinct bipolar data signals. Let $c_1(t)$ and $c_2(t)$ be two different bipolar spreading signals that have very small cross-correlation. Then the transmitted signal is

$$s(t) = \rho_1(t)c_1(t)A\cos\omega_0t - \rho_2(t)c_2(t)A\sin\omega_0t$$

(21.28)

This signal is a QPSK signal that carries two different data streams. It is a special type of quadrature amplitude modulated (QAM) signal presented in Chapter 13. If a perfect carrier reference were available, its performance against noise with an optimum receiver would be the same as for a single bipolar data signal with bipolar spreading since the signals on each quadrature carrier can be demodulated into separate bipolar signals with bipolar spreading.


21.2 Rejection of Narrowband Interference

Spread spectrum systems were initially designed and built to combat hostile narrowband jamming. In this section the effects of jamming by a sinusoid at the carrier frequency when the transmitted signal is a bipolar data signal spread by BPSK or QPSK. Jamming with a sinusoid is often called “tone” jamming. Interference from other noise sources will be ignored.

21.2.1 Bipolar Data Signal with Bipolar Spreading

As before, the BPSK data signal will be denoted by $\rho(t)$. Suppose the desired received spread spectrum signal is $s(t) = \rho(t)c(t)A\cos(\omega_0t + \theta)$ and the jamming signal is the sinusoid $J(t) = B\cos(\omega_0t + \alpha)$ where $\theta$ is a random variable uniformly distributed over $[0, 2\pi]$ and $\alpha$ is a constant angle chosen by the jammer. Then the actual received signal is

$$r(t) = s(t) + J(t) = \rho(t)c(t)A\cos(\omega_0t + \theta) + B\cos(\omega_0t + \alpha)$$

(21.29)

The desired signal power is

$$P_s = E\{s^2(t)\} = \rho^2(t)c^2(t)A^2E\{\cos^2(\omega_0t + \theta)\} = A^2E\{0.5 + 0.5\cos(2\omega_0t + 2\theta)\} = 0.5A^2$$

(21.30)

The time averaged jamming power is $P_J = 0.5B^2$. Therefore, the signal-to-noise ratio at the input of the receiver is

$$\text{SNR}_{IN} = \frac{P_s}{P_J} = \frac{A^2}{B^2}$$

(21.31)
We will assume an ideal situation where the receiver knows the carrier frequency, carrier phase, and spreading signal exactly. The receiver then despreads the received signal by first forming the product

\[
\begin{align*}
r_0(t) &= r(t)c(t) \cos(\omega_0 t + \theta) \\
&= \rho(t)c(t)A \cos(\omega_0 t + \theta) \times c(t) \cos(\omega_0 t + \theta) + B \cos(\omega_0 t + \alpha) \times c(t) \cos(\omega_0 t + \theta) \\
&= \rho(t) \frac{A}{2} [1 + \cos(2\omega_0 t + 2\theta)] + \frac{B}{2} c(t) [\cos(\alpha - \theta) + \cos(2\omega_0 t + \alpha + \theta)]
\end{align*}
\]

This product is passed through a lowpass filter with a cutoff frequency \( \omega_b \) that passes the main lobe of the spectrum of the data signal, \( \rho(t) \), and eliminates all the spectral components around \( 2\omega_0 \). Let \( H\{x(t)\} \) represent the output of the lowpass filter to an arbitrary input signal \( x(t) \). Then the output of the lowpass filter to input \( r_0(t) \) is

\[
r_1(t) = \rho(t) \frac{A}{2} + \frac{B}{2} \cos(\alpha - \theta) H\{c(t)\}
\]

The product operation collapses the spectrum of the desired signal and spreads the spectrum of the jamming signal. The power of \( H\{c(t)\} \) can be closely approximated by the ratio of the widths of the main lobes of the spectra of \( \rho(t) \) and \( c(t) \), that is, by \( \omega_b/\omega_c \). The desired signal power at the output of the lowpass filter is \( P_{so} = A^2/4 \) and the jamming power is \( P_{jo} = B^2 \cos^2(\alpha - \theta)(\omega_b/\omega_c)/4 \). Therefore, the output signal-to-noise ratio is

\[
\text{SNR}_{\text{out}} = \frac{P_{so}}{P_{jo}} = \frac{A^2}{B^2 \cos^2(\alpha - \theta)} \left( \frac{\omega_c}{\omega_b} \right)
\]

The worst case, that is, the lowest output SNR occurs when \( \alpha = \theta \). This happens when the jammer makes the phase of its sinusoid exactly the same as that of the desired spread signal’s carrier phase. The worst case output SNR is

\[
\text{SNR}_{\text{out}} = \frac{A^2}{B^2} \left( \frac{\omega_c}{\omega_b} \right) = \text{SNR}_{\text{IN}} \left( \frac{\omega_c}{\omega_b} \right)
\]

The bandwidth ratio \( (\omega_c/\omega_b) \) is called the processing gain.

### 21.2.2 Bipolar Data Signal with QPSK Spreading

According to (21.25) the QPSK spread signal has the form \( s(t) = \rho(t) \sqrt{2} A \cos[\omega_0 t + B(t)] \) where \( B(t) \) is given by (21.21). Again let the jamming signal be \( J(t) = B \cos(\omega_0 t + \alpha) \) so that the received signal is

\[
r(t) = s(t) + J(t) = \rho(t) \sqrt{2} A \cos[\omega_0 t + B(t)] + B \cos(\omega_0 t + \alpha)
\]

The pre-envelope for the received signal is

\[
r_+(t) = r(t) + j\dot{r}(t) = \rho(t) \sqrt{2} A e^{j[\omega_0 t + B(t)]} + Be^{j(\omega_0 t + \alpha)}
\]
The power of the signal component of \( r_+ (t) \) is
\[
P_s = |\rho (t) \sqrt{2} A e^{j[\omega_0 t + B(t)]}|^2 = 2A^2
\]  
(21.38)
and the power of the jamming component is
\[
P_J = |B e^{j(\omega_0 t + \alpha)}|^2 = B^2
\]  
(21.39)
so the input SNR is
\[
P_{IN} = \frac{P_s}{P_J} = \frac{2A^2}{B^2}
\]  
(21.40)
Multiplying \( r_+ (t) \) by \( e^{-j[\omega_0 t + B(t)]} \) to despread it gives
\[
\tilde{r}(t) = \sqrt{2} A \rho (t) + B e^{j(\alpha - B(t))}
\]  
(21.41)
The receiver then computes
\[
r_0(t) = \Re \{\tilde{r}(t)\} = \sqrt{2} A \rho (t) + B \cos[\alpha - B(t)]
\]  
(21.42)
and passes this through a lowpass filter with cutoff frequency \( \omega_b \) as in the BPSK spreading case. It will be assumed that \( \rho (t) \) passes through this filter with little distortion. The jamming component \( B \cos[\alpha - B(t)] \) is now spread with the main lobe of its spectrum having its first null at \( \omega_c \). Let the output of this lowpass filter be
\[
r_1(t) = \sqrt{2} A \rho (t) + B H \{\cos[\alpha - B(t)]\}
\]  
(21.43)
The power of the signal component of \( r_1(t) \) is \( P_{so} = 2A^2 \). The power of the filtered jamming component is approximately \( P_{Jo} = B^2 (\omega_b / \omega_c) / 2 \). Thus, the output SNR is
\[
\text{SNR}_{out} = \frac{P_{so}}{P_{Jo}} = \frac{4A^2}{B^2} \left( \frac{\omega_c}{\omega_b} \right) = 2 \text{SNR}_{IN} \left( \frac{\omega_c}{\omega_b} \right)
\]  
(21.44)
Thus QPSK spreading provides twice the improvement in the output SNR as BPSK spreading for sinusoidal jamming. An intuitive explanation for this result is that with QPSK spreading the jammer cannot make its tone phase the same as the transmitted signal carrier phase which is jumping over four values.

Similar results hold for more complicated narrow band signals than pure sine waves. Despreading at the receiver collapses the desired signal spectrum to a narrow bandwidth and spreads the bandwidth of the narrow band interferer to a wide band. Then a lowpass filter that passes the narrow band desired signal eliminates much of the power of the spread interferer. If another direct sequence spread spectrum signal using a different spreading code interferes with the received signal, the despreading operation collapses the desired signal spectrum but leaves the other direct sequence spread signal as a wideband signal that remains as wideband noise and its power is reduced by the processing gain. This is the basic idea behind multiple communicators sharing the same bandwidth using different spreading codes and is called \textit{code division multiple access (CDMA)}. 
21.3 PN Code Tracking for Bipolar Data and Spreading

The first function that must be performed at a spread spectrum receiver is to synchronize its local replica of the spreading code with that of the received signal. Initially there can be a large phase offset between the two spreading codes and the local code must be aligned within a fraction of a chip with the code of the received signal. This is called code acquisition. There is no easy and quick method for acquisition. Basically, the receiver must repeatedly shift its code by a fraction of a chip, perhaps 1/2 or 1/4 of a chip, and test when the maximum correlation is achieved. In some systems a short spreading code is used at the start of transmission to allow the receiver to synchronize quickly. There is an extensive literature on code acquisitions schemes. See for example Holmes [Holmes] and Ziemer and Peterson [Ziemer]. Code acquisition will not be explored more here.

Fine code synchronization must be obtained once course code acquisition has been achieved. In addition the code phase alignment must be continually maintained since transmit and receive clocks will be slightly different and Doppler effects will be present if the transmitter and receiver are moving relative to each other. Two tracking methods will be discussed in this section, the coherent delay-locked loop and the noncoherent delay-locked loop.

People making these tracking systems have found that gain imbalances in the early and late branches of these tracking loops cause unwanted bias in the local code phase. A system called a $\tau$-dither loop can be used to eliminate the bias at the expense of slower lock time. This will not be discussed here. See the references for discussions about the $\tau$-dither loop.

21.3.1 Coherent Delay-Locked Loop for Bipolar Data and Spreading

A receiver using a coherent delay-locked loop (DLL) is shown in Figure 21.4. It will be assumed that the transmitted signal uses bipolar data and spreading. Interference from noise and jamming will be ignored in explaining how the receiver works. Then the received IF signal has the form

$$r(t) = \rho(t - \tau_0)c(t - \tau_0)A\cos\omega_0(t - \tau_0)$$  \hspace{1cm} (21.45)

where $\rho(t)$ is the bipolar data signal, $c(t)$ is the bipolar spreading signal, $\tau_0$ is the bulk propagation delay, and $\omega_0$ is the IF carrier frequency. The receiver first locks its local oscillator to the carrier frequency. The received signal is squared to form

$$r^2(t) = \rho^2(t - \tau_0)c^2(t - \tau_0)A^2\cos^2\omega_0(t - \tau_0) = A^2[1 + \cos 2\omega_0(t - \tau_0)]/2$$  \hspace{1cm} (21.46)

The bandpass filter, $B(\omega)$ passes the component at $2\omega_0$ and rejects the DC component. A phase-locked loop (PLL) with nominal frequency $2\omega_0$ generates a very stable sinusoid at frequency $2\omega_0$ and the this signal is applied to a frequency divider to supply a sinusoid at
frequency $\omega_0$. Then $r(t)$ is multiplied by the reference carrier and passed through a lowpass filter, $D(\omega)$, to eliminate the spectral components around $2\omega_0$.

Suppose the scaling is such that the output of $D(\omega)$ is

$$r_1(t) = \rho(t - \tau_0)c(t - \tau_0)$$  \hfill (21.47)

Let $\tau$ be the coarse estimate of the signal delay determined during the acquisition phase. The code tracking loop contains a device that generates a replica of the code used by the transmitter. The phase of the code generated by this device can be varied by the “PN Code Phase Controller” which adjusts $\tau$. The output of the code generator passes through two
delay elements, each of which delays the output by $\Delta$. Typically $\Delta$ is set to $T_c/2$, that is, half the chip duration. Other values larger and smaller can be used. The first output of the code generator is $c(t - \tau + \Delta)$ and is called the *early* PN reference signal. The signal, $c(t - \tau)$ out of the first delay element is called the *inphase* reference and when $\tau$ is properly adjusted it is exactly in phase with the code in the received signal. The signal, $c(t - \tau - \Delta)$, out of the second delay is called the *late* reference signal.

The output of the early multiplier is
\[
v_1(t) = \rho(t - \tau_0)c(t - \tau_0)c(t - \tau + \Delta)
\]
and the output of the late multiplier is
\[
v_2(t) = \rho(t - \tau_0)c(t - \tau_0)c(t - \tau - \Delta)
\]
Let the output of the lowpass filter $L(\omega)$ to an arbitrary input $x(t)$ be denoted by $L\{x(t)\}$. Then the error signal $e(t)$ is
\[
e(t) = L\{v_1(t) - v_2(t)\} = L\{v_1(t)\} - L\{v_2(t)\}
\]
Now suppose that the data signal $\rho(t)$ is 1 for all time. The autocorrelation function for the spreading signal $c(t)$ was defined in (21.13) to be
\[
\sigma(t) = \frac{1}{T_{pn}} \int_{0}^{T_{pn}} c(t + \tau)\overline{c(\tau)} \, d\tau
\]
The lowpass filter $L(\omega)$ is designed to closely approximate computing the periodic autocorrelation function, so
\[
L\{v_1(t)\} = L\{c(t - \tau_0)c(t - \tau + \Delta)\} = \sigma(\tau - \tau_0 - \Delta)
\]
and
\[
L\{v_2(t)\} = L\{c(t - \tau_0)c(t - \tau - \Delta)\} = \sigma(\tau - \tau_0 + \Delta)
\]
Let $\epsilon = \tau - \tau_0$ which is the error between the code phase assumed by the DLL and the code phase in the received signal. Then
\[
e(t, \epsilon) \simeq \sigma(\epsilon - \Delta) - \sigma(\epsilon + \Delta)
\]
This function is shown in Figure 21.5 for $\Delta = T_c/2$ and was derived using (21.14). This is often called an *s-curve* in the literature. Notice that for $|\epsilon| < T_c/2$ this function provides an estimate of the code phase error that is linearly proportional to the error. For $T_c/2 < |\epsilon| < 3T_c/2$ it still gives a value that has the same polarity as the phase error. The “PN Code Phase Controller” samples $e(t)$ periodically and bumps phase of the local “PN Code Generator” forward or backward based on the samples to drive the phase error to zero. Of course, this assumes that the phase error was adjusted to be less than one chip during the acquisition phase. Detailed analysis of the behavior of this code tracking loop can be found in the literature.
Once the code tracking loop locks up, the baseband received signal is multiplied by the local inphase reference code to despread it. The despread signal is then demodulated by the data detector. Details about data detection are explained in the next section.

This coherent DLL is rarely used because it has two significant problems:

1. The first problem is that when the data signal, \( \rho(t) \), is a random bipolar signal it becomes part of the products in the early correlation (21.52) and late correlation (21.53). This randomizes the correlations and makes their values zero on average. This problem can be solved with some complexity by delaying the correlator outputs and multiplying them by appropriately delayed estimates of the data levels. This is called decision directed operation.

2. The second major problem is that the carrier recovery loop operates on the received signal before it is despread. Therefore, this loop cannot benefit from the rejection of narrowband interference resulting from despreading.

### 21.3.2 Noncoherent Delay-Locked Loop

The noncoherent delay-locked loop shown in Figure 21.6 solves the problems of the coherent delay-locked loop. No coherent carrier reference is generated by this DLL. The received signal is first multiplied by early and late local reference code signals. In addition, the received signal is multiplied by a local oscillator signal \( \cos \omega_{LO}(t) \) to translate the spectrum down to the passband of the bandpass filter \( B(\omega) \). This translation can be avoided if a bandpass filter with center frequency \( \omega_0 \) is available, but it can be economical to use a commercially available bandpass filter with center frequency \( \omega_0 - \omega_{LO} \). The bandpass filter should pass at least the main lobe of the despread signal spectrum. The outputs of the bandpass filters are
squared, subtracted, and passed through a narrow band lowpass filter \( L(\omega) \). Narrow band lowpass filtering the squared output of a bandpass filter gives a measure of the input signal power in the bandpass filter’s passband. If the reference code is not in synchronism with the code of the received signal, the product of the reference code and the received signal will remain spread and the output power of the bandpass filter will be small. The output power will be a maximum when the reference code has the correct phase. However, in order to generate a phase error signal, the method used in the coherent DLL is employed. An early reference signal \( c(t - \tau + \Delta) \), an inphase reference \( c(t - \tau) \), and a late reference \( c(t - \tau - \Delta) \) are generated. The variable \( \tau \) is the best estimate of the code phase in the received signal determined during the acquisition process. It can be shown that the output of the lowpass filter \( L(\omega) \) has a shape somewhat similar to the s-curve for the coherent DLL and gives an estimate of the phase error. The error signal is sampled and is used to increment the phase
of the PN Code Phase Generator to drive the error signal to zero.

No coherent carrier reference is generated by this system. The effect of the data signal on the correlations is eliminated by the squaring operations. Also, multiplication by the early and late reference signals spreads any narrow band interference in the input signal and reduces its effect according to the processing gain.

The recovered inphase spreading code is used to despread the received signal and the resulting data signal is demodulated in a conventional way. This is discussed in Section 21.5.

### 21.3.2.1 Phase Adjustment Using a Random Walk Filter

When the code tracking loop is implemented using discrete-time signal processing, the code phase can only be incremented by discrete values that are integer multiples of the sampling period. Also the adjustment process during tracking should introduce only a very small amount of jitter in the reference code phase. It should just make small, slow changes in the phase. A random walk filter similar to the one presented in Chapter 14, pages 237–238 can be used to accomplish this function. A simple implementation of a random walk filter is shown in Figure 21.7. The error signal \( e(t) \) is sampled with period \( T_s \) and the samples are hard limited. The hard limiting can be represented mathematically by forming \( \text{sign}[e(nT_s)] \) which will have the values 1 when \( e(nT_s) > 0 \), -1 when \( e(nT_s) < 0 \), and 0 when \( e(nT_s) = 0 \). The hard limited samples are applied to an up/down counter. This counter sums its inputs. Let the counter value be \( \zeta(n) \). Then

\[
\zeta(n) = \zeta(n-1) + \text{sign}[e(nT_s)]
\]  

(21.55)

At each sampling instant, the count can be incremented by 1, decremented by 1, or stay the same. If the error signal has a bias in one direction, the count will increase or decrease in the bias direction. The count is applied to a threshold detector. Let the threshold be the positive integer \( T_{\text{max}} \). As long as the count stays between \( -T_{\text{max}} \) and \( T_{\text{max}} \) no code phase adjustments are made. When the count exceeds \( T_{\text{max}} \) this is a strong indication that the phase of the local reference code is greater than that in the received signal, that is \( \tau > \tau_0 \). Then the threshold detector sends a signal to the code phase controller to reduce \( \tau \) and sends a signal to the up/down counter to reset the count to 0. The phase controller responds by decrementing \( \tau \) by one sample period. When the count falls below \( -T_{\text{max}} \) this is a strong indication that the error is negative and that \( \tau \) should be increased. The threshold detector sends a signal to the phase controller to increase \( \tau \) and a signal to the up/down counter.
to reset the count to 0. The threshold determines the responsiveness of the DLL. A small threshold will allow more frequent phase updates and quicker loop response but with more jitter and susceptibility to noise. A larger threshold will result in less frequent updates with a slower loop response and more immunity to noise.

A C code segment to implement the random walk filter is shown below. The phase error $e(nT_s)$ is represented by `CodePhaseError`, the count $\zeta(n)$ by `walk_count`, the local code delay $\tau$ by `channel_delay`, and the threshold $T_{\text{max}}$ by `WalkLimit`.

```c
/* Do a random walk filter */
if(CodePhaseError > 0) ++walk_count;
else if(CodePhaseError < 0) --walk_count;

if(walk_count > WalkLimit){
    walk_count = 0;
    channel_delay = channel_delay - 1;
} else if (walk_count < -WalkLimit){
    walk_count = 0;
    channel_delay = channel_delay + 1;
}
```

### 21.4 Code Tracking for QPSK Spreading

The noncoherent DLL discussed in Section 21.3.2 can be extended to track the code in signals with QPSK spreading. A noncoherent QPSK code tracker is shown in Figure 21.8. It is assumed that the spreading signals $c_1(t)$ and $c_2(t)$ are uncorrelated. Therefore, the tracker has early/late correlators and power detectors for each of the spreading signals and the two give separate results because of the spreading signals are uncorrelated. The early/late power differences are combined to give an error signal. The “DLL LO” in Figure 21.6 will be ignored in this section.

The received signal $r(t)$ is first multiplied by the locally generated early and late PN reference signals $c_1(t - \tau + \Delta)$, $c_1(t - \tau - \Delta)$, $c_2(t - \tau + \Delta)$, and $c_2(t - \tau - \Delta)$. As before, the variable $\tau$ is the local estimate of the required delay of the reference codes and $\Delta$ is in the order of a chip width, for example, $T_c/2$ is a common choice. The center frequency of the bandpass filter $B(\omega)$ is the carrier frequency and it passes a large portion of the power in the unspread data signal. Thus the output power of the bandpass filters is the largest when the local codes are aligned with the code in the input signal. When the phase is off by more than a chip, the signals at the input to the bandpass filters remain spread and their output powers are small. The power reduction is in the order of the processing gain. When the delay estimate is correct, the powers out of the early and late branches for code 1 and code 2 will be the same and their differences will be close to zero. As the phase error deviates from the correct value, the early and late powers will become unbalanced and the average
of their difference will generate an s-curve. The lowpass filter $L(\omega)$ should have a narrow bandwidth so that it essentially does the averaging. The local code delay, $\tau$, is adjusted by the “PN Code Phase Controller” like for the BPSK noncoherent DLL.

The synchronized inphase code reference signals, $c_1(t - \tau)$ and $c_2(t - \tau)$ are used to despread the received signal and the result is passed to a BPSK or QPSK demodulator depending on whether the unspread data signal is a BPSK or QPSK signal.

See Holmes [2, pp. 482–486] for an analysis of the performance of this tracking loop in the presence of additive Gaussian noise.

## 21.5 Data Detection at the Receiver for Bipolar Data and Spreading

A discrete-time implementation of the data receiver is shown in Figure 21.9. First the received signal, $r(t)$, is despread by multiplying by the recovered PN reference $c(t - \tau)$. The resulting signal, $q(t)$, is then sampled with period $T_s$ and passed through an FIR bandpass filter, $H(\omega)$, and an FIR bandpass Hilbert transform filter, $G(\omega)$. Both filters are centered at the carrier frequency, $\omega_0$, and have bandwidths that pass the modulated data signal. Both should also have the same number of taps so that their delays are matched. Since the two filter outputs are band limited typically to a bandwidth much less than the carrier frequency, they can be downsampling by a significant factor to reduce the computational complexity. If the outputs are downsampling by the factor $K$, the sampling period is increased to $T = KT_s$ or the sampling rate, $1/T$, is reduced by the factor $K$. The two filters operate on $T_s$ spaced samples and samples must be shifted into the filter delay lines every $T_s$ seconds. Actually, the same sequence of samples is shifted into both delay lines, so only one delay line is required with the two sets of filter taps operating on the same delay line. The output samples are only required every $K$ input samples, so the filters and downsampling can be efficiently implemented by only computing the outputs every $K$ input samples. Remember that the inputs still must be shifted into the delay line at the original fast rate $1/T_s$.

### 21.5.1 Costas Loop for Carrier Tracking and Demodulation

The downsampling outputs, $a(nT)$ and $\hat{a}(nT)$, are demodulated by a discrete-time Costas loop which is the same as the one presented in Chapter 6 for demodulating DSBSC AM signals. This is not surprising since the despread signal is a DSBSC AM signal with a bipolar message rather than a continuous amplitude message. It is convenient to form the complex signal $\tilde{a}(nT) = a(nT) + j\hat{a}(nT)$ which we called the pre-envelope before. The solid lines in Figure 21.9 are the real parts of complex signals and the dotted lines are the imaginary parts. Ignoring additive noise and assuming the filters do not distort the data signal, the pre-envelope has the form

\[
a_+(nT) = A\rho(nT)e^{j[\omega_0nT + \beta(nT)]}
\] (21.56)
21.5 Data Detection at the Receiver for Bipolar Data and Spreading

Figure 21.8: A Noncoherent DLL for QPSK Code Tracking
The angle $\beta(nT)$ accounts for the constant carrier phase shift caused by propagation delay, a linearly varying component caused by frequency offsets, and other slow phase perturbations. The Costas loop forms the estimate $\theta(nT)$ of the received carrier phase and multiplies the pre-envelope by $\exp[-j\theta(nT)] = \cos\theta(nT) - j \sin\theta(nT)$ to demodulate it. The angle $\theta(nT)$ has the form

$$\theta(nT) = \omega_0 nT + \gamma(nT)$$  \hspace{1cm} (21.57)

where $\omega_0 nT$ is the phase of a sinusoid with the carrier frequency $\omega_0$ and $\gamma(nT)$ is the deviation from this nominal carrier phase ramp. The output of the complex multiplier is

$$a_+(nT)e^{-j\theta(nT)} = A\rho(nT)e^{j[\beta(nT) - \gamma(nT)]}$$  \hspace{1cm} (21.58)

The angle $\psi(nT) = \beta(nT) - \gamma(nT)$ is the error between the received carrier and locally generated carrier reference phases. The real part of the multiplier output is

$$\Re\{a_+(nT)e^{-j\theta(nT)}\} = A\rho(nT)\cos[\beta(nT) - \gamma(nT)] = a(nT)\cos\theta(nT) + \dot{a}(nT)\sin\theta(nT)$$  \hspace{1cm} (21.59)
and the imaginary part is
\[ \Im \{ a_+(nT)e^{-j\theta(nT)} \} = A\rho(nT) \sin[\beta(nT) - \gamma(nT)] \quad (21.61) \]
\[ = \bar{a}(nT) \cos \theta(nT) - a(nT) \sin \theta(nT) \quad (21.62) \]

The values of \( A, \rho(nT), \) and \( \beta(nT) \) are not known by the receiver, so it cannot use (21.59) and (21.61) for computation. However, the receiver knows \( a(nT), \bar{a}(nT), \) and \( \theta(nT) \) and uses (21.60) and (21.62) for its computations. The product of the real and imaginary parts
\[ e(nT) = A^2\rho^2(nT) \cos[\beta(nT) - \gamma(nT)] \sin[\beta(nT) - \gamma(nT)] \quad (21.63) \]
When the magnitude of the phase error, \( |\beta(nT) - \gamma(nT)| \), is less than \( \pi/2 \), \( e(nT) \) has the same polarity as the phase error and can be used to adjust the phase of the local oscillator.

The formula for adjusting the local oscillator phase each sampling instance is
\[ \theta((n+1)T) = \theta(nT) + \omega_0 T + \alpha e(nT) + w(nT) \quad (21.64) \]
where
\[ w(nT) = \beta e(nT) + w((n-1)T) \quad (21.65) \]
The parameters \( \alpha \) and \( \beta \) are small positive constants with \( \beta < \alpha/100 \) for a transient response without large oscillations. The term \( \omega_0 T = \omega_0 KT_s \) is the nominal phase change between samples for the carrier frequency \( \omega_0 \). The term \( \alpha e(nT) \) increments the LO phase by a small fraction of the phase error estimate. The signal \( w(nT) \) is a scaled accumulation of the error signal \( e(nT) \) and allows the loop to track frequency offsets with zero error. If the frequency offset is \( \Delta \omega \), \( w(nT) \) converges to \( T\Delta \omega \) when the loop reaches steady state and is in lock effectively retuning the nominal frequency of the LO to \( \omega_0 + \Delta \omega \).

The loop is said to be in lock when the phase error \( \psi(nT) = \beta(nT) - \gamma(nT) \) has converged to essentially zero. Then the real part of the complex multiplier output is the demodulated data signal \( x(nT) = A\rho(nT) \) and the imaginary part is zero. The energy of the imaginary part can be monitored to determine if the loop is in lock or not.

Details about the transfer function and frequency response to the Costas loop are presented in Chapter 6.

### 21.5.2 Matched Filtering

Suppose a known signal \( s(t) \) or \( -s(t) \) is transmitted during the time interval \([0, T_b] \) with equal likelihood and that the received signal over this time interval is either \( r(t) = s(t) + n(t) \) or \( r(t) = -s(t) + n(t) \) where \( n(t) \) is white Gaussian noise. In the case of rectangular pulses, \( s(t) \) is constant over the symbol period and can be chosen as 1 with no loss of generality. It is shown in many books on communications that the receiver that minimizes the error probability computes the statistic
\[ I = \int_{0}^{T_b} r(t)s(t) \, dt \quad (21.66) \]
and decides \( s(t) \) was transmitted if \( I > 0 \) and decides \(-s(t)\) was transmitted if \( I \leq 0 \). See, for example, Holmes [2, pp. 212–216].

This statistic can also be generated with a matched filter. The impulse response of the matched filter is \( g(t) = s(T_b - t) \) which is nonzero only over the interval \([0, T_b]\). The output of this filter is

\[
m(t) = \int_{t-T_b}^{t} r(\tau) g(t-\tau) \, d\tau = \int_{t-T_b}^{t} r(\tau) s(T_b - t + \tau) \, d\tau
\]  

(21.67)

At the end of the signal interval \( t = T_b \) and the filter output is

\[
m(T_b) = \int_{0}^{T_b} r(\tau) s(\tau) \, d\tau = I
\]

(21.68)

which is the desired statistic.

In a typical wideband binary digital communication system, the input data bits arrive every \( T_b \) seconds and a logical 0 is mapped to \( s(t - nT_b) \) and a logical 1 to \(-s(t - nT_b)\) over the \( n \)-th bit period \([nT_b, (n + 1)T_b]\). Thus the receiver can use a matched filter and sample its output at the end of each bit period to form the decision statistics. Integrals are replaced by sums for a discrete-time implementation.

A simulated example of the matched filter output with no additive noise is shown in Figure 21.10. The simulation used 100 chips/bit and 40 samples/chip resulting in 4000 samples/bit. Rectangular pulses of amplitude 1 were used. The outputs of the receive bandpass and bandpass Hilbert transform filters were down sampled by a factor of 50 resulting in 80 samples/bit in the data demodulator. There is an initial transient during the first few bits while the PN code tracker, Costas carrier tracking loop, and bit synchronizer converge to the correct values. Let \( x(nT) \) be the data signal output of the Costas loop as shown in Figure 21.9. The discrete-time matched filter theoretically computes the sum

\[
m(nT) = \sum_{k=n-79}^{n} x(kT)
\]

(21.69)

More generally, let the number of samples per data bit be \( N_b \). Then the matched filter computes

\[
m(nT) = \sum_{k=n-(N_b-1)}^{n} x(kT)
\]

(21.70)

The z-transform of the filter impulse response is

\[
Q(z) = \sum_{n=0}^{N_b-1} 1 \times z^{-n} = \frac{1 - z^{-N_b}}{1 - z^{-1}}
\]

(21.71)

Thus this filter is equivalent to an IIR filter and can be very efficiently implemented by the following recursion:

\[
m(nT) = x(nT) - x((n - N_b)T) + m((n - 1)T)
\]

(21.72)
In other words, the new sum at the next time instant is obtained by adding the new input sample to the previous sum and subtracting the input $N_b$ samples in the past from it. This method was used to implement the matched filter in the simulation program.

Another approach is to wait for bit synch to be achieved and then calculate the matched filter sum in (21.70) only at the end of each bit, that is, at times $nN_bT$. The two methods have essentially the same computational complexity. We will not use this method because our bit tracking algorithm will use $m(nT)$ for all $n$.

![Figure 21.10: Sample Matched Filter Output Starting at Time 0](image)

### 21.5.3 Symbol Clock Tracking

The receiver samples the output of the matched filter at the end of each data bit interval, which is also the beginning of the next bit interval, and makes a decision on what was transmitted. These decisions nominally occur at the symbol clock frequency. There can be slight differences in the symbol clock frequency at the transmitter and receiver caused by hardware differences and Doppler shifts if the two are moving relative to each other. Thus, the receiver must continually adjust its symbol clock to the correct phase and frequency. The maximum likelihood estimate for symbol clock synchronization when the transmitted signal is corrupted by additive white Gaussian noise is presented in many books on communications. For example, see Holmes [2, Chapter 12]. The solution involves the matched filter output. The complete maximum likelihood estimation formula is moderately complex computationally, so a good suboptimal method will be used in this chapter.

The suboptimal solution is suggested by looking at the matched filter output in Figure 21.10. Notice that the matched filter output has a maximum amplitude at the end of each bit.
interval. When two successive bits differ, the filter output makes a straight-line transition from one peak amplitude to its negative. This line crosses zero in the middle of the bit interval. The receiver can lock to these zero crossings. When two successive bits are the same, the filter output stays essentially constant at the corresponding peak amplitude and the receiver symbol clock must “flywheel” through these intervals. The receiver can sample the matched filter output one-half of a bit period after the zero crossings, which should be close to the end of the bits, and make decisions on the transmitted bits corresponding to the polarities of the filter output samples. An initial estimate of the correct symbol clock phase can be determined by locating the first zero crossing of the matched filter output. The receiver can then look for filter output zero crossings every $T_b$ seconds after that and sample the output $T_b/2$ seconds later to estimate the transmitted bits.

Now suppose the initial zero crossing is detected at time $\tau_0$. Let the analog bit decision at time $t_{n-1} = \tau_0 + (n - 0.5)T_b + \epsilon_{n-1}$ be $a_{n-1} = \text{sign}[m(t_{n-1})]$ and the next bit decision at time $t_n = \tau_0 + (n + 0.5)T_b + \epsilon_n$ be $a_n = \text{sign}[m(t_n)]$. The terms $\epsilon_{n-1}$ and $\epsilon_n$ account for the changes to the sampling times made by the symbol clock tracker. The matched filter output at the nominal zero crossing time $t_{0,n} = \tau_0 + nT_b + \epsilon_n$ essentially halfway between these two bits is $m(t_{0,n})$ which should be close to zero if there was a bit transition. There are several cases to consider. Each case requires a delay of half a bit from the mid-bit sample time, that is, zero crossing sample time, since $a_n$ must be known.

1. $a_{n-1} = -1$ and $a_n = 1$ (Positive zero crossing slope)
   (a) $m(t_{0,n}) > 0$
   Looking at Figure 21.10 it can be seen that the nominal sampling time for the zero crossing is too late and the time to the next zero crossing measurement should be reduced slightly.
   (b) $m(t_{0,n}) < 0$
   The sampling time is too early and the time to the next zero crossing measurement should be increased slightly.

2. $a_{n-1} = 1$ and $a_n = -1$ (Negative zero crossing slope)
   The clock increments should be the negatives of the positive slope case.

3. $a_{n-1} = a_n$ (No bit transition)
   In this case the matched filter output stays nearly constant with large magnitude for two bits and there is no zero crossing. The time for the next zero crossing sample should be the current zero crossing time plus the bit duration $T_b$.

These three cases can be described by the following formula for computing the next zero crossing sampling time from the current one. The parameter $\gamma$ is a small positive constant:

$$t_{0,n+1} = t_{0,n} + T_b + \gamma(a_{n-1} - a_n)m(t_{0,n})$$ (21.73)

Making $\gamma$ smaller results in clock tracking with less phase jitter and more immunity to noise but slower convergence. Making it larger gives faster convergence but more susceptibility to noise.
21.6 Data Detection for Bipolar Data and QPSK Spreading

The clock phase increments should be small so noise effects are averaged over many bits and do not cause significant clock jitter. The clock increments \( \gamma (a_{n-1} - a_n) m(t_{0,n}) \) can be passed through a random walk filter like the one described in Section 21.3.2.1 to accomplish this smoothing.

### 21.6 Data Detection for Bipolar Data and QPSK Spreading

A method for demodulating a bipolar data signal spread by QPSK is described in this section. Additive noise will be ignored and it is assumed that perfect acquisition of the spreading codes has been achieved, possibly, by the noncoherent DLL described in Section 21.4. Suppose the received signal is

\[
r(t) = \rho(t - \tau) [c_1(t - \tau) \cos(\omega_0 t + \beta) - c_2(t - \tau) \sin(\omega_0 t + \beta)]
\]

where \( \tau \) is the propagation delay, \( \omega_0 \) is the carrier frequency (or possibly an IF frequency), and \( \beta \) is a carrier phase offset. The Hilbert transform of \( r(t) \) is

\[
\dot{r}(t) = \rho(t - \tau) [c_1(t - \tau) \sin(\omega_0 t + \beta) + c_2(t - \tau) \cos(\omega_0 t + \beta)]
\]

so the pre-envelope is

\[
r_+(t) = r(t) + j\dot{r}(t) = \rho(t - \tau) [c_1(t - \tau) + jc_2(t - \tau)] e^{j(\omega_0 t + \beta)}
\]

The receiver synthesizes the following local despreading signal:

\[
c(t) = [c_1(t - \tau) - jc_2(t - \tau)] e^{-j(\omega_0 t + \gamma)}
\]

The real and imaginary parts of \( c(t) \) are

\[
c_I(t) = \Re\{c(t)\} = c_1(t - \tau) \cos(\omega_0 t + \gamma) - c_2(t - \tau) \sin(\omega_0 t + \gamma)
\]

and

\[
c_Q(t) = \Im\{c(t)\} = -c_1(t - \tau) \sin(\omega_0 t + \gamma) - c_2(t - \tau) \cos(\omega_0 t + \gamma)
\]

The receiver despreads and demodulates \( r_+(t) \) to baseband by forming

\[
a(t) = r_+(t)c(t) = \left\{ \rho(t - \tau) [c_1(t - \tau) + jc_2(t - \tau)] e^{j(\omega_0 t + \beta)} \right\} \\
\quad \times \left\{ [c_1(t - \tau) - jc_2(t - \tau)] e^{-j(\omega_0 t + \gamma)} \right\} \\
\quad = 2\rho(t - \tau) e^{j(\beta - \gamma)}
\]

The real and imaginary parts of \( a(t) \) are

\[
a_I(t) = r(t)c_1(t - \tau) + \dot{r}(t)c_2(t - \tau) = 2\rho(t - \tau) \cos(\beta - \gamma)
\]

\[
a_Q(t) = r(t)\dot{c}_1(t - \tau) + \dot{r}(t)\dot{c}_2(t - \tau) = 2\rho(t - \tau) \sin(\beta - \gamma)
\]
and

\[ a_Q(t) = \hat{r}(t)c_1(t - \tau) - r(t)c_2(t - \tau) = 2\rho(t - \tau)\sin(\beta - \gamma) \quad (21.82) \]

Notice that \( \beta - \gamma \) is the phase error between the received and local carriers. Now a Costas loop can be used to drive the phase error to zero. Since \( \rho^2(t - \tau) = 1 \), a phase error estimate is computed by forming

\[ e(t) = a_I(t)a_Q(t) = 2\sin(2(\beta - \gamma)) \quad (21.83) \]

When \(|\beta - \gamma < \pi/2|\), \( e(t) \) has the same polarity as the phase error. When the phase error is small, \( \sin 2(\beta - \gamma) \approx 2(\beta - \gamma) \) and gives a phase error estimate linearly proportional to the phase error. This phase error estimate is used to drive a VCO to push the phase error to zero as described in previous descriptions of a Costas loop.

A discrete-time implementation of the receiver is shown in Figure 21.11. The received signal, perhaps demodulated to IF, is sampled with period \( T_s \) or frequency \( f_s = 1/T_s \) that satisfies the Nyquist sampling requirement to give the sequence \( r(nT_s) \). This sequence is passed through an FIR Hilbert transform filter that operates on the \( T_s \) spaced samples resulting in the signal \( \hat{r}(nT_s) \). The pre-envelope samples are \( r_+(nT_s) = r(nT_s) + j\hat{r}(nT_s) \). The pre-envelope and input sequence are both down-sampled by a factor of \( K \). The sampling period for this down-sampled pre-envelope is \( T = KT_s \) and the sampling rate is \( 1/T = f_s/K \). The down-sampled pre-envelope is despread and translated to baseband by multiplying by the local despreading reference signal to get

\[
a(nT) = r_+(nT)c(nT) = a_I(nT) + ja_Q(nT) = 2\rho(nT - \tau)e^{j(\beta - \gamma)} \quad (21.84)
\]
where

\[ a_I(nT) = r(nT)c_1(nT - \tau) + \hat{r}(nT)c_2(nT - \tau) = 2\rho(nT - \tau)\cos(\beta - \gamma) \quad (21.85) \]

and

\[ a_Q(nT) = \hat{r}(nT)c_1(nT - \tau) - r(t)c_2(nT - \tau) = 2\rho(nT - \tau)\sin(\beta - \gamma) \quad (21.86) \]

The down-sampling is justified because \( a(nT) \) is a lowpass signal with a bandwidth of the baseband bipolar data signal \( \rho(nT) \). The Hilbert transform filter delay line must have \( T_s \) spacing and samples \( r(nT_s) \) must be shifted into the delay line at the fast sampling rate \( f_s \). However, the convolution sum to compute the output of the filter only has to be computed every \( K \) input samples when an output sample is required. The local carrier phase estimate, \( \theta(nT) \) is updated as described for the Costas loop in Section 21.5.1. The phase estimate and spreading codes from the DLL are used by the “PN Reference Generator” to create samples of the inphase and quadrature despreading reference sequences.

When the Costas loop has achieved lock, \( a_I(nT) = 2\rho(nT - \tau) \) and this signal is passed through a matched filter, symbol clock recovery system, and bit detector just like those described for the BPSK spreading case.

### 21.7 Data Detection for Two Different Bipolar Data Streams Spread on Quadrature Carriers

Code synchronization for the QPSK data and QPSK spreading case can be achieved using the noncoherent DLL described in Section 21.4. In this section it will be assumed that the local codes are perfectly synchronized with those in the received signal. Also, additive noise will be ignored. Suppose the received signal has the form

\[ r(t) = \rho_1(t - \tau)c_1(t - \tau)\cos(\omega_0t + \beta) - \rho_2(t - \tau)c_2(t - \tau)\sin(\omega_0t + \beta) \quad (21.87) \]

The Hilbert transform of \( r(t) \) is

\[ \hat{r}(t) = \rho_1(t - \tau)c_1(t - \tau)\sin(\omega_0t + \beta) + \rho_2(t - \tau)c_2(t - \tau)\cos(\omega_0t + \beta) \quad (21.88) \]

so the pre-envelope is

\[ r_+(t) = r(t) + \hat{r}(t) = [\rho_1(t - \tau)c_1(t - \tau) + j\rho_2(t - \tau)c_2(t - \tau)]e^{j(\omega_0t + \beta)} \quad (21.89) \]

If the local carrier oscillator phase is \( \gamma \), the receiver can first form the following product to find \( \rho_1(t - \tau) \):

\[ a_I(t) = r_+(t)c_1(t - \tau)e^{-j(\omega_0t + \gamma)} = \rho_1(t - \tau)e^{j(\beta - \gamma)} + j\rho_2(t - \tau)c_2(t - \tau)c_1(t - \tau)e^{j(\beta - \gamma)} \quad (21.90) \]
The second term on the right contains the product of the two different codes, so it remains spread and acts as a wideband noise signal. Passing $a_I(t)$ through a lowpass filter, $L(\omega)$, that just passes the lowpass data signal, $\rho_1(t-\tau)e^{j(\beta-\gamma)}$, gives

$$a_{IL}(t) = \rho_1(t-\tau)e^{j(\beta-\gamma)} + jn_I(t) \quad (21.91)$$

where

$$n_I(t) = L\{\rho_2(t-\tau)c_2(t-\tau)c_1(t-\tau)\}e^{j(\beta-\gamma)} \quad (21.92)$$

The filtered noise power is reduced from its original power by the processing gain. The signal $a_{IL}(t)$ can be passed through a Costas loop to make the phase error, $\beta - \gamma$, zero and create $\rho_1(t-\tau)$. Notice that when $\beta = \gamma$, $n_I(t)$ becomes real so

$$\alpha_I(t) = \Re\{a_{IL}(t)\} = \rho_1(t-\tau) \quad (21.93)$$

and the crosstalk from the quadrature data, $\rho_2(t-\tau)$, is zero. Then $\alpha_I(t)$ can be processed by a matched filter, symbol clock tracker, and data detector to estimate the original data bits.

The quadrature data signal can be detected in a similar fashion. First the receiver can form the following product:

$$a_Q(t) = r_+(t)c_2(t-\tau)e^{-j(\omega_0t+\gamma)} = \rho_1(t-\tau)c_1(t-\tau)c_2(t-\tau)e^{j(\beta-\gamma)} + j\rho_2(t-\tau)e^{j(\beta-\gamma)} \quad (21.94)$$

Passing this through the lowpass filter, $L(\omega)$ gives

$$a_{QL}(t) = n_Q(t) + j\rho_2(t-\tau)e^{j(\beta-\gamma)} \quad (21.95)$$

where

$$n_Q(t) = L\{\rho_1(t-\tau)c_1(t-\tau)c_2(t-\tau)\}e^{j(\beta-\gamma)} \quad (21.96)$$

The signal, $-ja_{QL}(t) = \rho_2(t-\tau)e^{j(\beta-\gamma)} - jn_Q(t)$ can be processed with a Costas loop or the estimate of $\gamma$ from the inphase processing can be used. When $\beta = \gamma$, the quadrature noise $n_Q(t)$ is strictly real and

$$\alpha_Q(t) = \Im\{a_{QL}(t)\} = \rho_2(t-\tau) \quad (21.97)$$

This signal can be processed with a matched filter and data detector. The symbol clock from the inphase processing can be used.

### 21.8 Experiments for Spread Spectrum Transmitters and Receivers

Set the AIC sampling rate for all the experiments in this chapter to 16,000 Hz. Use 100 chips per data bit and 40 samples per chip so that the number of samples per data bit is $100 \times 40 = 4000$ and the data bit rate is $16000/4000 = 4$ data bits per second. Of course, the sampling rates and data rates would be much higher in a real system and many functions
might be implemented with special purpose hardware that can operate at high frequencies. However, we need to use rates that can be implemented with the DSK. Use a transmitter carrier frequency $f_0 = 4000$ Hz. In an actual system there will be a propagation delay between the transmitter and receiver. Use a circular buffer in the transmitters to simulate up to a six chip propagation delay.

### 21.8.1 Experiments for Bipolar Data and Bipolar Spreading

#### 21.8.1.1 Making a Transmitter for Bipolar Data and Bipolar Spreading

Write a C program for the C6713 DSK to implement a spread spectrum transmitter to spread a bipolar data signal with a bipolar PN signal. Block diagrams for possible transmitter implementations are shown in Figure 21.1 and Figure 21.2.

Your transmitter should first send an initial handshake sequence to allow the receiver to detect the presence of a signal and roughly synchronize its local code generator. The handshake signal should have the following stages:

1. First send 2 seconds of silence, that is, transmit zeros for two seconds. This corresponds to 32,000 zeros for a sampling rate of 16 kHZ. This will make sure there are no glitches when your program starts running. It will also give you time to start the receiver. You can increase this time if necessary.

2. Next transmit 80 pure carrier samples, that is $A \cos 2\pi f_0 n T_s$ for $n = 0, \ldots, 79$ where $f_0 = 4000$ and $T_s = 1/16000$. This is 20 cycles of the carrier. The receiver will use this to detect the presence of a signal.

3. Then send 32 samples of the negative of the carrier, that is, $-A \cos 2\pi f_0 n T_s$. The receiver will use this 180 degree phase shift as a timing marker and start its code generator 32 samples after detecting the phase shift.

4. Finally, begin sending the spread spectrum signal.

Do the following for the spread spectrum phase:

1. Simulate the binary random data source by using a 23-stage maximal length feedback shift register as presented in Chapter 9. Use the connection polynomial $h_0(D) = 1 + D^5 + D^{23}$. What is the period of the generated PN sequence?

2. Use a 23-stage maximal length feedback shift register to generate the spreading code bits. Use the connection polynomial $h_1(D) = 1 + D^5 + D^{11} + D^{17} + D^{23}$. Initialize the shift register state to a value of your choosing. The replica shift register in the receiver should use this initial state.

3. Use a 16 kHz codec sampling rate. Generate a sequence of transmit samples using 40 samples per chip and 100 chips per data bit. Thus the spread spectrum processing gain is 100.
4. Use a carrier frequency of \( f_0 = 4000 \text{ Hz} \). Scale the transmit samples to use a significant portion of the DAC dynamic range.

5. Write your spread signal samples into a circular buffer that can simulate up to a six chip propagation delay from the transmitter to the receive. Write transmit samples with a several chip delay to the codec.

6. Compute and plot the theoretical power spectral density for the bipolar data signal modulated by the carrier but unspread. Assume that the data signal is an ideal binary random sequence.

7. Compute and plot the theoretical power spectral density for the spread and modulated signal. Assume that the data and spreading sequences are ideal binary sequences that are independent.

8. Disable spreading in your C program by setting the bipolar spreading sequence to 1 all the time. Run your transmitter and measure and record the transmitted signal spectrum with the oscilloscope. Set the scope parameters to see reasonable spectrum details around the 4 kHz carrier frequency.

9. Turn on spreading and measure and record the spectrum for the spread signal.

10. Compare the theoretical and measured spectra.

### 21.8.1.2 Making a Noncoherent Delay-Locked Loop Code Tracker for Bipolar Data and Bipolar Spreading

One of the first functions the receiver must perform is to synchronize its locally generated spreading signal with that in the received signal. Create a C program to implement the noncoherent delay-locked loop presented in Section 21.3.2 and shown in Figure 21.6. Assume that the receiver has already translated the signal down to an IF frequency of 4 kHz which is the carrier frequency you used in your transmitter. The “DLL LO” and associated multipliers further frequency translate the IF signal, \( r(t) \), perhaps because a manufacturer may wish to use commercially available bandpass filters with a certain center frequency. This step is often ignored in texts and can be ignored if bandpass filters at the IF frequency are available. Perform the following tasks to implement and test the noncoherent DLL:

1. Use your transmitter as the signal source for your receiver. You can loop the signal back from the line out to the line in using cables, loop the samples back internally from your transmitter to your receiver code, or send the line out on your PC to another PC and make the receiver on this other PC.

2. The receiver should monitor the incoming samples to determine when a signal is present and then begin its handshake processing. The handshake phase should do the following:
(a) Detection of the presence of a handshake signal can be accomplished by measuring the received signal power. Pass the received samples through a bandpass filter with a center frequency of 4 kHz. Square the filter output samples and pass the squared signal through a lowpass filter to average them and get an estimate of the received signal power. Decide a signal is present once the lowpass filter output remains above an appropriately set threshold.

Use your engineering judgement to determine the cutoff frequency for the lowpass filter. If the cutoff frequency is very low, the filter output will take a long time to reach steady-state but will be smooth. The output will reach steady-state more quickly with a larger cutoff frequency but the output will have more ripples. An input signal should be detected before the 180 degree phase shift occurs in the received handshake signal.

Continue to monitor the received signal power until a signal is detected.

(b) Once a signal is detected begin looking for the 180 degree phase shift in the received 4 kHz sine wave. Use your ingenuity to create an algorithm to detect the phase shift. As a hint, notice that a 4 kHz sine wave sampled at 16 kHz generates a sequence that repeats every 4 samples. Let \( r(n) \) be the sampled received signal. Then \( |r(n) - r(n - 4)|^2 \) will be zero when these samples are both taken from the 4 kHz sine wave or its negative. The squared difference will have positive values when \( n \) and \( n - 4 \) overlap the carrier and its negative. You could average the squared difference over a couple of samples. This glitch can be used as the timing mark.

(c) Count for 32 samples after detecting the phase shift and then start the local code generator and DLL.

3. Use a DLL LO frequency of \( \omega_{LO} = 2 \text{ kHz} \). The bandpass filter \( B(\omega) \) should pass signal components in the vicinity of \( 4 - 2 = 2 \text{ kHz} \). Design this filter as an elliptic IIR bandpass filter with a passband of 1996 to 2004 Hz. This passes the main lobe of the despread signal spectrum. Try stop bands that are at least 100 dB down extending from 0 to 1980 Hz and 2020 to 8000 Hz.

4. Implement a replica of the PN spreading code generator for your receiver. This should generate samples of the spreading signal at the 16 kHz sampling rate. Put these bipolar signal samples into a buffer so you can match the bulk propagation delay, form the early and late PN reference signals, and make adjustments to the reference signal phase. Use 1/2 chip early and late delays. That is, make \( \Delta \) one-half of a chip period. This will be 20 samples since we are using 40 samples per chip.

5. Square the outputs of the early and late bandpass filters and form their difference. Pass the difference signal through the lowpass filter, \( L(\omega) \). The lowpass filter should have a very narrow bandwidth so it significantly smooths the difference signal. The loop will take longer to achieve lock with a smaller lowpass filter bandwidth but will be more immune to jitter and noise. It will lock faster with a larger bandwidth but
will have more jitter and be more susceptible to noise. This is a standard trade-off for feedback systems.

6. Figure out a method for making the “PN Code Phase Controller.” Since you are implementing the DLL using sampled signals and have stored samples of the local spreading reference signal in a buffer, you will only be able to change the reference signal phase by one or more complete samples without implementing a computationally intensive interpolation algorithm. Since you are using 40 samples per chip, you will be able to adjust the phase to within 1/40 of the chip period which should be quite sufficient. Based on the phase error signal, \( e(t) \), at the output of the lowpass filter, determine in what direction you should increment the phase to drive the phase error to zero. The next item describes a method for adjusting the phase using a random walk filter.

7. Now implement the random walk filter discussed in Section 21.3.2.1 and shown in Figure 21.7 to smoothly adjust the reference code phase with little jitter. You will have to experiment to determine a reasonable count threshold. The threshold determines how fast the DLL achieves lock and its immunity to noise. As usual, a tradeoff between speed of convergence and noise immunity must always be made. One approach that has been used is to start with values that give rapid convergence and then “gear shift” to values that give a small bandwidth and greater noise immunity.

8. Now that you have constructed all the components of the DLL, connect them together and close the loop. Use your transmitter to generate the input to the DLL and test that it acquires code synchronization.

(a) In testing that the DLL is working, initially make the bipolar data bits all 1 so that just the spreading signal is transmitted. Observe the signal \( q(t) \) in Figure 21.6. It should be a 4 kHz sine wave when the loop is in lock. Also observe the signal \( e(t) \). It should hover around zero when the loop is in lock.

(b) Now turn on the random data bits in the transmitter and check that the DLL achieves lock.

(c) Experiment with different count thresholds in the random walk filter. Observe and record the loop behavior for different thresholds.

### 21.8.1.3 Making a Costas Loop Data Demodulator Acting on the Despread Received Signal

Once your noncoherent DLL is working, begin making the data demodulator and detector. Write C code to implement the system shown in Figure 21.9 and described in Section 21.5 and Section 21.5.1. Your program should perform the following operations:

1. Assume that received signal samples, \( r(nT_s) \) taken at a 16 kHz rate are available. Despread this signal by multiplying it by samples of the inphase PN reference signal to form \( q(nT_s) = r(nT_s)c(nT_s - \tau) \).
2. Pass the despread received signal through an FIR bandpass filter, $H(\omega)$, to get $a(nT_s)$. This filter should have a center frequency of 4 kHz and pass at least the main lobe of the BPSK data signal spectrum. Use the Remez FIR filter design program to design a 63-tap bandpass filter with a lower stopband of 0 to 3000 Hz, a passband of 3996 to 4004 Hz, and an upper passband of 5000 to 8000 Hz. The purpose of this filter is to eliminate noise outside of the BPSK signal band.

3. Pass the despread received signal through an FIR bandpass Hilbert transform filter, $G(\omega)$, to get $\hat{a}(nT_s)$. Use the Remez FIR filter design program to design a 63-tap Hilbert transform filter with a lower stopband of 0 to 3000 Hz, a passband of 3996 to 4004 Hz, and an upper passband of 5000 to 8000 Hz. This filter and the bandpass filter have the same length, so their delays are automatically matched.

4. The pre-envelope $a(nT_s) + j\hat{a}(nT_s)$ has a one-sided bandpass spectrum essentially 8 Hz wide extending from 3996 to 4004 Hz, so it is highly over-sampled with the 16 kHz rate. Therefore, it can be down-sampled to a significantly lower sampling rate. Down-sample both $a(nT_s)$ and $\hat{a}(nT_s)$ by a factor of $K = 50$, that is, select every 50th sample from the original sequences. The resulting sampling rate will be $16000/50 = 320$ Hz and there will be $4000/50 = 80$ samples per data bit. The filter outputs are only needed every 50th input sample. Therefore, only perform the convolutions when an output is required. The convolutions operate on samples taken at the 16 kHz rate, so you must shift samples into the filter delay lines at the 16 kHz rate. However, only perform the convolutions every 50th input sample.

5. Implement the Costas loop as shown in Figure 21.9 and described in Section 21.5.1. The Costas loop is discussed in more detail in Chapter 6. When the loop has locked to the received carrier, the loop output signal $x(nT)$ will be the baseband transmitted bipolar data signal. To test that the Costas loop is working, you may want to transmit a simple known periodic data sequence that you can easily see at the loop output.

### 21.8.1.4 Making the Matched Filter and Data Symbol Clock Tracker

Create C code to pass the Costas loop output, $x(t)$, through the matched filter. This filter is discussed in Section 21.5.2. Use (21.72) to efficiently compute the matched filter output. After down-sampling by the factor $K = 50$ you will have 80 samples per data bit, so $N_b = 80$. You will need a buffer to store the samples $x((n - N_b)T)$ through $x(nT)$. The matched filter output should look similar to Figure 21.10.

Implement the data symbol clock tracker presented in Section 21.5.3. Sample the matched filter output at the end of each data bit period and convert the samples to logical levels 0 and 1 to recover the transmitted bits. Check that there are no errors in the recovered bit stream. Remember that the transmitted bits generated by the maximal length feedback shift register satisfy the homogeneous difference equation

$$d_0(n) + d_0(n - 5) + d_0(n - 23) = 0 \quad (21.98)$$
If your recovered bit stream has no errors, it should satisfy this equation except for a brief initial transient if the initial states of the transmit and receive shift registers are different.

21.8.1.5 Testing Immunity to Sinusoidal Interference

Add an interfering sine wave of the form \( I(t) = B \cos \omega_0 t \) to the transmitter output before putting samples into the buffer that simulates propagation delay. As before, \( \omega_0 \) is the carrier frequency and this sine wave will be inphase with the carrier which causes the worst case interference. Start with a small value of \( B \) that causes no bit errors in the detected bit stream. Increase \( B \) until bit errors occur and record the value.

Disable spreading in the transmitter so the unspread BPSK signal \( s(t) = \rho(t)A \cos \omega_0 t \) is transmitted. You can do this by simply making \( c(t) = 1 \) for all \( t \). Disable despreading in the receiver. You can do this by making the inphase spreading code reference signal, \( c(t - \tau) \), equal to 1 for all \( t \). Continue to use your Costas loop, matched filter, and bit clock tracking system to recover the transmitted data bits. Again, add an interfering sine wave \( I(t) = B \cos \omega_0 t \) to the transmitted signal. Increase \( B \) from a small value until bit errors occur. Compare this value to the one you got when spreading was enabled. Compute the ratio of \( B \) with spreading that just causes bit errors to the value without spreading that just causes bit errors and compare it to the processing gain.

21.8.2 Experiments for Bipolar Data and QPSK Spreading

21.8.2.1 Making a Transmitter for Bipolar Data and QPSK Spreading

Implement a transmitter for spreading a bipolar data signal with a QPSK carrier as presented in Section 21.1.2 and shown in Figure 21.3. Use a maximal length feedback shift register with the primitive connection polynomial \( h_0(D) = 1 + D^5 + D^{23} \) to simulate the binary random customer data. Use a maximal length feedback shift register with the primitive connection polynomial \( h_1(D) = 1 + D^5 + D^{11} + D^{17} + D^{23} \) to generate the bipolar spreading signal \( c_1(t) \) and the primitive connection polynomial \( h_2(D) = 1 + D + D^4 + D^5 + D^{23} \) to generate the bipolar spreading signal \( c_2(t) \). Create the required C programs and do the following tasks for your transmitter:

1. First send the same handshake sequence presented in Section 21.8.1.1.

2. Do items 1 through 10 of Section 21.8.1.1 for the QPSK spread spectrum phase except use the two spreading codes for Item 2 and set \( c_1(t) = c_2(t) = 1 \) in Item 8 to disable the spreading.

21.8.2.2 Making a Noncoherent DLL for QPSK Code Tracking

Implement the noncoherent DLL for QPSK code tracking presented in Section 21.4 and shown in Figure 21.8. Use the steps of Section 21.8.1.2 with the appropriate modifications to implement and test your QPSK code tracker.
21.8 Experiments for Spread Spectrum Transmitters and Receivers

21.8.2.3 Making a Costas Loop, Matched Filter, Clock Tracker, and Data Detector

Make and test the receiver for BPSK data with QPSK spreading discussed in Section 21.6 and shown in Figure 21.11. Use the methods of Section 21.8.1.1 with the appropriate modifications.

21.8.2.4 Testing Immunity to Sinusoidal Interference

Test the immunity of your system to interfering sinusoids at the carrier frequency with various phases and compare the results with your system using BPSK spreading.

21.8.3 Experiments for Two Different Bipolar Data Streams Spread on Quadrature Carriers

21.8.3.1 Making a Transmitter for Two Different Bipolar Data Streams Spread on Quadrature Carriers

Make a transmitter to send two different bipolar spread data streams on quadrature carriers. The transmitted signal is given by (21.28) in Section 21.1.3. This transmitted signal has QPSK chips. Continue to use a 16 kHz sampling rate, 4 kHz carrier frequency, 40 samples per chip, and 100 chips per data symbol in each quadrature. Use feedback shift registers to simulate the two data streams and two spreading codes. Use the connection polynomial $h_1(D) = 1 + D^5 + D^{23}$ for $\rho_1(t)$, $h_Q(D) = 1 + D^5 + D^{11} + D^{17} + D^{23}$ for $\rho_2(t)$, $h_1(D) = 1 + D + D^4 + D^5 + D^{23}$ for $c_1(t)$, and $h_2(D) = 1 + D + D^4 + D^5 + D^8 + D^9 + D^{11} + D^{17} + D^{23}$ for $c_2(t)$. Test your transmitter as you did for the previous transmitters with the appropriate modifications.

21.8.3.2 Implementing a Noncoherent DLL Code Tracker

Make and test a noncoherent QPSK DLL for code tracking. The DLL of Section 21.4 should work for this signal also.

21.8.3.3 Making a Data Demodulator

Make and test a receiver for recovering the two transmitted quadrature data streams. Equations for implementing this receiver are presented in Section 21.7.
21.9 References


Chapter 22

Introduction to Convolutional Codes

The this chapter introduces the basic definitions, notation, and concepts for binary convolutional codes and trellis codes. Some references for detailed treatments of these codes are Johannesson and Zigangirov [6], Schlegel [9], Viterbi and Omura [14], and Vucetic [15]. First, a tool for analyzing linear sequential circuits called the Huffman $D$-transform is presented. Realizations for linear time-invariant sequential circuits that can be used as encoders are derived using $D$-transform methods. Then convolutional codes are described in terms of their generator matrices, generator polynomials, and check matrices. Conversion from non-systematic to systematic form is discussed. Convenient representations of code sequences in terms of a trellis diagram or a state transition diagram are introduced and the error correction properties of binary convolutional codes in terms of the weight distributions of the code sequences are briefly discussed. Next, the method of combining binary convolutional codes with QAM modulation, known as trellis coded modulation (TCM), is presented. Then, a brief introduction to the Viterbi decoding algorithm is presented. This algorithm finds the maximum likelihood estimate of the transmitted trellis sequence. Finally, the BCJR or forward-backward algorithm is discussed. This algorithm finds the maximum a posteriori (MAP) transmitted bit at each time instance but does not guarantee that the decoded bit stream is a trellis codeword. It is used extensively in decoding turbo codes. Turbo codes are used in current broadband wireless systems and will not be discussed in this chapter.

Experiments for implementing and testing these encoders and decoding methods using the TMS320C6713 DSK are provided.

22.1 The Huffman D-Transform

The two-sided $D$-transform or Huffman transform of a sequence $f(n)$ is defined to be the power series

$$F(D) = \sum_{n=-\infty}^{\infty} f(n)D^n$$

(22.1)

When dealing with binary convolutional codes, the values of $f(n)$ can be one of two values, 0 or 1. In the digital signal processing field, it is customary to replace $D$ by $z^{-1}$ and call
the series the Z-transform. These transforms allow signals to be compactly represented by rational functions of $D$ and linear time-invariant systems to be analyzed by simple algebraic means in the transform domain.

**EXAMPLE 22.1** Unit Step Function

The unit step function is defined to be

$$u(n) = \begin{cases} 
1 & \text{for } n \geq 0 \\
0 & \text{for } n < 0
\end{cases} \quad (22.2)$$

Then

$$U(D) = \sum_{n=0}^{\infty} D^n = \frac{1}{1 - D} \quad \text{for } |D| < 1 \quad (22.3)$$

This is the sum of a geometric series with ratio $D$. The time sequence can be obtained from its $D$ transform by expanding the transform into a power series by a variety of methods. In this case, the closed form for the sum of a geometric series is evident. The denominator, $1 - D$, can also be divided into the numerator, 1, to obtain the power series coefficients.

The one-sided $D$-transform is sometimes used to automatically include initial conditions stored in the sequential circuit. Let $f(n)$ be a sequence that may or may not be zero for $n < 0$. Its one-sided transform is defined to be

$$F_{\pm}(D) = \sum_{n=0}^{\infty} f(n)D^n \quad (22.4)$$

That is, the sum always begins at time $n = 0$.

**22.1.1 Two-Sided Transform of a Delayed Sequence**

Suppose $f(n)$ has the transform $F(D)$. Let $f(n)$ delayed by $L$ samples be $g(n) = f(n - L)$. The delay $L$ can be any positive or negative integer. Then $G(D) = D^L F(D)$.

**Proof:**

$$G(D) = \sum_{n=-\infty}^{\infty} g(n)D^n = \sum_{n=-\infty}^{\infty} f(n - L)D^n \quad (22.5)$$

Replacing $n - L$ by $m$ gives

$$G(D) = \sum_{m=-\infty}^{\infty} f(m)D^{m+L} = D^L F(D) \quad (22.6)$$
22.1 The Huffman D-Transform

22.1.2 One-Sided Transform of a Delayed Sequence

Let \( f(n) \) have the one-sided transform \( F_+(D) \) and let \( L \) be an integer greater than or equal to 0. The delayed sequence \( g(n) = f(n - L) \) has the one-sided transform

\[
G_+(D) = D^L F_+(D) + \sum_{n=0}^{L-1} f(n - L)D^n
\]  

(22.7)

The sum depends on the initial sequence values \( f(-L), f(1 - L), \ldots, f(-1) \).

Proof:

\[
G_+(D) = \sum_{n=0}^{\infty} g(n)D^n = \sum_{n=0}^{\infty} f(n - L)D^n
\]

\[
= \sum_{n=0}^{L-1} f(n - L)D^n + \sum_{n=L}^{\infty} f(n - L)D^n
\]

(22.8)

Making the substitution \( m = n - L \) in the last summation on the right gives

\[
G_+(D) = \sum_{n=0}^{L-1} f(n - L)D^n + \sum_{m=0}^{\infty} f(m)D^{m+L}
\]

\[
= \sum_{n=0}^{L-1} f(n - L)D^n + D^L F_+(D)
\]

(22.9)

EXAMPLE 22.2

Consider a system whose input \( x(n) \) and output \( y(n) \) are related by the first-order difference equation \( y(n) = x(n) + y(n - 1) \). Taking the one-sided transform of both sides gives

\[
Y_+(n) = X_+(n) + [y(-1) + DY_+(D)]
\]

(22.10)

So

\[
Y_+(D) = \frac{X_+(D)}{1 - D} + \frac{y(-1)}{1 - D}
\]

(22.11)

The first term on the right is the output when the initial condition is \( y(-1) = 0 \). Let the zero initial condition solution be denoted by \( y_0(n) \). The second term is the observed output caused by the initial condition when the input is identically 0. Therefore,

\[
y(n) = y_0(n) + y(-1) \quad \text{for} \quad n \geq 0
\]

(22.12)
22.1.3 *D*-Transform of a Convolution

The convolution of two sequences \( f(n) \) and \( g(n) \) is defined to be the sequence

\[
\begin{align*}
    h(n) &= \sum_{k=-\infty}^{\infty} f(k)g(n-k) = \sum_{k=-\infty}^{\infty} g(k)f(n-k) \\
\end{align*}
\] (22.13)

The *D*-transform of the convolution is

\[
    H(D) = F(D)G(D) \quad (22.14)
\]

**Proof:**

\[
\begin{align*}
    H(D) &= \sum_{n=-\infty}^{\infty} \left[ \sum_{k=-\infty}^{\infty} f(k)g(n-k)D^n \right] \\
    &= \sum_{k=-\infty}^{\infty} f(k) \sum_{n=-\infty}^{\infty} g(n-k)D^n \\
    &= \sum_{k=-\infty}^{\infty} f(k)G(D)D^k = G(D) \sum_{k=-\infty}^{\infty} f(k)D^k \\
    &= F(D)G(D) \quad (22.15)
\end{align*}
\]

The last summation on the right is just the *D*-transform of the delayed sequence \( g(n-k) \), so using the delay property (22.5) gives

\[
\begin{align*}
    H(D) &= \sum_{k=-\infty}^{\infty} f(k)G(D)D^k = G(D) \sum_{k=-\infty}^{\infty} f(k)D^k \\
    &= F(D)G(D) \quad (22.16)
\end{align*}
\]

22.2 Transfer Functions and Realizations

The output \( y(n) \) of a linear, time-invariant, sequential circuit is the convolution of its input \( x(n) \) with its unit pulse response \( h(n) \), that is

\[
    y(n) = \sum_{k=-\infty}^{\infty} h(k)x(n-k) \quad (22.17)
\]

For binary circuits, the signal values can only be 0 or 1 and modulo 2 addition is used in the convolution formula. The *D*-transform of the output is

\[
    Y(D) = X(D)H(D) \quad (22.18)
\]

The *transfer function* for the circuit is defined to be the ratio

\[
    H(D) = Y(D)/X(D) \quad (22.19)
\]
and is the transform of the circuit’s unit pulse response. Initial conditions are all assumed to be 0 when computing transfer functions.

**EXAMPLE 22.3  Transfer Function of a Delay Element**

Suppose the input and output of a circuit are related by the equation \( y(n) = x(n-1) \). Then \( Y(D) = DX(D) \) and the transfer function is \( H(D) = Y(D)/X(D) = D \). Therefore, the symbol \( D \) is often used to represent a one unit delay element.

Linear, time-invariant sequential circuits with a finite number of storage elements have transfer functions that are the ratio of two polynomials in \( D \). The ratio of two polynomials is said to be a **rational function**. We will assume that the rational transfer function has the form

\[
H(D) = \frac{A(D)}{B(D)} = \frac{a_0 + a_1D + \cdots + a_MD^M}{1 + b_1D + \cdots + b_ND^N}
\]

(22.20)

The numerator order \( M \) can be less than, greater than, or equal to the denominator order \( N \). When the denominator is \( B(D) = 1 \), the circuit is called a **finite duration impulse response** (FIR) system. When \( B(D) \) is not 1, the circuit is called a **recursive** or **infinite duration impulse response** (IIR) system.

A rational transfer function can be realized by many different circuits. Two common realizations will be presented in the following sections.

### 22.2.1 Type 1 Direct Form Realization

The rational transfer function can be decomposed into the cascade of the denominator and numerator portions as shown in Figure 22.1. The transform of the intermediate signal \( V(D) \)

\[
X(D) \rightarrow \frac{1}{B(D)} \rightarrow V(D) \rightarrow A(D) \rightarrow Y(D)
\]

Figure 22.1: Representing \( H(D) \) as a Cascade

is related to the input \( X(D) \) by the equation \( V(D) = X(D)/B(D) \). Therefore,

\[
X(D) = B(D)V(D) = (1 + b_1D + \cdots + b_ND^N)V(D)
\]

\[
= V(D) + b_1V(D)D + \cdots + b_NV(D)D^N
\]

(22.21)

or

\[
V(D) = X(D) - b_1V(D)D - \cdots - b_NV(D)D^N
\]

(22.22)

In the time domain, this is equivalent to the following difference equation:

\[
v(n) = x(n) - b_1v(n-1) - \cdots - b_Nv(n-N)
\]

(22.23)
Similarly,
\[ Y(D) = A(D)V(D) = a_0 V(D) + a_1 V(D)D + \cdots + a_M V(D)D^M \]  
(22.24)

or
\[ y(n) = a_0 v(n) + a_1 v(n - 1) + \cdots + a_M v(n - M) \]  
(22.25)

These two equations describe what we will call a \textit{type 1 direct form} realization. The nomenclature is not standard and this realization is sometimes called the \textit{controller canonical form}. A block diagram for this realization is shown in Figure 22.2 with \( M = N \). This incurs no loss of generality because the appropriate higher order coefficients can be set to zero when \( M \) and \( N \) differ. It is called a direct form because the transfer function coefficients appear explicitly in the equations or block diagram.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{type1_direct_form.png}
\caption{Type 1 Direct Form Realization}
\end{figure}

### 22.2.2 Type 2 Direct Form Realization

A second realization known as the \textit{type 2 direct form} or \textit{observer canonical form} will now be developed. We will let \( M = N \) for simplicity. We know that
\[ H(D) = \frac{A(D)}{B(D)} = \frac{Y(D)}{X(D)} \]  
(22.26)

Cross multiplying gives
\[ Y(D)B(D) = X(D)A(D) \]  
(22.27)

or
\[ Y(D)(1 + b_1 D + \cdots + b_N D^N) = X(D)(a_0 + a_1 D + \cdots + a_N D^N) \]  
(22.28)

This can be rearranged into the form
\[ Y(D) = a_0 X(D) + \sum_{k=1}^{N} [a_k X(D) - b_k Y(D)] D^k \]  
(22.29)

The block diagram for the realization suggested by this equation is shown in Figure 22.3
22.3 Description of a Convolutional Code by its Generator Matrix

An \((N, K)\) convolutional encoder is a device that accepts successive input blocks of \(K\) bits from a data source, adds redundancy to the input blocks, and outputs blocks of \(N\) coded bits where \(N > K\) in such a way that some errors introduced in the output stream when transmitted over a channel can be corrected. The current output block is a function of the present and some past input blocks. The output may depend on a finite or infinite number of past inputs. When the current output block only depends on the current input block, the code is called a \emph{block code} and is a special case of a convolutional code with no memory. The \emph{code rate} is defined as the ratio \(R = K/N\). Generally, the lower the code rate, the higher the error correction capability.

To make the notation concrete, we will envision that the serial data input bit stream is separated into blocks of \(K\) bits \(x_1(n), \ldots, x_K(n)\) at block time \(n\). These can be combined into the row vector

\[
x(n) = [x_1(n) \ x_2(n) \ \cdots \ x_K(n)]
\]

Similarly, the encoder output blocks will be represented at block time \(n\) by the row vector

\[
y(n) = [y_1(n) \ y_2(n) \ \cdots \ y_N(n)]
\]

Linear, time-invariant codes form an important class of convolutional codes. For these codes, the output code streams are sums of convolutions of the input data streams with the impulse responses of a set of filters known as the \emph{generators} for the code. The impulse response samples can only have the values 0 and 1 and sums in the convolutions are computed using modulo 2 addition. That is, the sum of two bits is their exclusive-or. In the coding literature, this is also called GF(2) arithmetic where GF stands for Galois field. If \(g_{i,j}(D)\) is the transfer function from the \(i\)th input stream to the \(j\)th output stream, the Huffman

Figure 22.3: Type 2 Direct Form Realization
transform of the encoder output can be expressed as

\[ Y(D) = X(D) \begin{bmatrix} g_{1,1}(D) & g_{1,2}(D) & \cdots & g_{1,N}(D) \\ g_{2,1}(D) & g_{2,2}(D) & \cdots & g_{2,N}(D) \\ \vdots & \vdots & \ddots & \vdots \\ g_{K,1}(D) & g_{K,2}(D) & \cdots & g_{K,N}(D) \end{bmatrix} \]

\[ = X(D)G(D) \quad (22.32) \]

When transforms are added, the coefficients of like powers of \( D \) are combined using GF(2) arithmetic. In general, these systems are multi-input and multi-output systems. The question of how to realize these systems most efficiently has been studied extensively by system theorists. For example, the question of how to realize a system with the minimum number of state variables has been answered 6.

The matrix \( G(D) \) is called the generator matrix for the code. When the components of \( G(D) \) are polynomials with maximum degree \( M \), the convolutional encoder has finite memory and the output depends only on the present and past \( M \) inputs. The components can also be rational functions of \( D \) and then the encoder has infinite memory. The rank of \( G(D) \) must be \( K \) so that unique input sequences result in unique output sequences.

The contents of each delay element in an encoder realization can be selected as a state variable and a vector consisting of the set of state variables is called the state of the encoder. When a realization for an encoder has \( L \) delay elements, the state vector can take on \( 2^L \) values. The output for a time-invariant sequential circuit is a function of the current state and inputs. The next state is also a function of the current state and inputs. These functions may be linear or nonlinear but do not depend on the time index \( n \).

**EXAMPLE 22.4** The Ungerboeck 4-State Code

The Ungerboeck 4-state convolutional encoder [11, 12] shown in Figure 22.4 has \( K = 1 \) input stream and \( N = 2 \) output streams. The input-output relationship is

\[ \begin{bmatrix} y_1(D) \\ y_2(D) \end{bmatrix} = x_1(D)[D \ 1 + D^2] \quad (22.33) \]

![Figure 22.4: The Ungerboeck 4-State Convolutional Encoder](image-url)
Thus the generator matrix is
\[
G(D) = \begin{bmatrix} g_{1,1}(D) & g_{1,2}(D) \end{bmatrix} = \begin{bmatrix} D & 1 + D^2 \end{bmatrix}
\] (22.34)

The time domain formulas for the outputs are
\[
y_1(n) = x_1(n - 1) \quad \text{and} \quad y_2(n) = x_1(n) + x_1(n - 2)
\] (22.35)

### 22.4 Systematic Form of a Convolutional Code

A systematic code is one in which the $K$ input data streams appear explicitly in $K$ of the $N$ encoder output streams. The remaining $N - K$ output streams are called check symbols. A convolutional code with a $G(D)$ matrix of rank $K$ but arbitrary otherwise is not necessarily systematic. However, a non-systematic code can always be converted into an equivalent systematic one. Since $G(D)$ has rank $K$, it must contain $K$ linearly independent columns. These $K$ columns can be moved to the right-hand side of $G(D)$ by reordering the encoder output streams. Assuming this has been done, the generator matrix can be partitioned as follows:

\[
G(D) = \begin{bmatrix} A(D)_{K \times (N-K)} & B(D)_{K \times K} \end{bmatrix}
\]

\[
= B(D)B^{-1}(D) \begin{bmatrix} A(D) & B(D) \end{bmatrix}
\]

\[
= B(D) \begin{bmatrix} B^{-1}(D)A(D) & I_{K \times K} \end{bmatrix}
\] (22.36)

The encoder output is
\[
Y(D) = X(D)G(D) = X(D)B(D) [B^{-1}(D)A(D) : I] = \hat{X}(D)\hat{G}(D)
\] (22.37)

where
\[
\hat{G}(D) = [B^{-1}(D)A(D) : I] = [P(D) : I]
\] (22.38)

and
\[
\hat{X}(D) = X(D)B(D)
\] (22.39)

Since $B(D)$ is invertible, there is a unique $\hat{X}(D)$ for each $X(D)$ and vice versa. The set of codewords generated by an encoder with the generator matrix $\hat{G}(D)$ is the same as by one with the generator matrix $G(D)$. However, input data sequences are mapped to codewords differently. Since the set of codewords is the same for both codes, the probability of making an error between two sequences is the same in both cases. However, the bit error probability in the decoded outputs may be different.
The output of an encoder with input $\hat{X}(D)$ and generator matrix $\hat{G}(D)$ is

$$Y(D) = [\hat{X}(D)P(D) : \hat{X}(D)]$$  \hspace{1cm} (22.40)

Notice that the input sequence appears on the right-hand side of the encoded sequence, so the code is systematic. A block diagram for an encoder of this form is shown in Figure 22.5. There is nothing special about putting the input streams on the right. The initial generator matrix columns could have been rearranged to put them in any $K$ columns.

**EXAMPLE 22.5** Systematic Form for Ungerboeck 4-State Code

The $(2,1)$ 4-state Ungerboeck non-systematic code has the generator matrix

$$G(D) = [D \hspace{0.5cm} 1 + D^2]$$  \hspace{1cm} (22.41)
22.4 Systematic Form of a Convolutional Code

Let \( A(D) = D \) and \( B(D) = 1 + D^2 \), so \( B^{-1}(D) = 1/(1 + D^2) \). The generator matrix for the equivalent systematic code is

\[
\hat{G}(D) = \begin{bmatrix}
D & 1 \\
1 + D^2 & 1
\end{bmatrix}
\]  

(22.42)

A block diagram for this encoder using the type 2 direct form is shown in Figure 22.6.

**EXAMPLE 22.6**  The LTE Turbo Code Constituent Encoder

![Figure 22.7: The 3GPP LTE Turbo Code Constituent Systematic Encoder](image)

The systematic encoder shown in Figure 22.7 is used in phones using 4G LTE technology [1]. You have one if your phone uses 4G technology. This is a type 1 direct form implementation. The switch at the left side is connected to \( x(n) \) during data transmission. After transmission of a data block is complete, say at time \( N \), the switch is moved to the position shown in the figure and the encoder is clocked three more times. This sets the final state of the encoder to zero. That is, the contents of the three \( D \) elements all become 0. The final three pairs of bits transmitted, sometimes called the *tail bits*, are \( y_2(N+1) = z(N+1), y_1(N+1), y_2(N+2) = z(N+2), y_1(N+2), y_2(N+3) = z(N+3), y_1(N+3) \). Setting the final state of the encoder helps some decoding algorithms.

The steps for computing the encoder output are:

1. \( y_2(n) = x(n) \) is the systematic output bit.

2. Compute the intermediate variable \( q_0(n) = x(n) + q_2(n) + q_3(n) \).
   
   Note: This step can be skipped and the right-hand side of the next step used instead.

3. Compute \( y_1(n) = q_0(n) + q_1(n) + q_3(n) = x(n) + q_1(n) + q_2(n) \).

4. Update the state: \( q_3(n) = q_2(n), q_2(n) = q_1(n), \) and \( q_1(n) = q_0(n) \).
The generator matrix for this encoder is
\[ G(D) = \begin{bmatrix} 1 + D + D^3 \\ 1 + D^2 + D^3 \end{bmatrix} 1 \] (22.43)

### 22.5 The Parity Check Matrix and Syndromes

Consider an (N,K) systematic convolutional code with generator matrix \( G(D) = [P(D) : I_{K \times K}] \). The check symbols in the output code block are
\[ [Y_1(D), \ldots, Y_{N-K}(D)] = X(D)P(D) \] (22.44)
and the information symbols are
\[ [Y_{N-K+1}(D), \ldots, Y_N(D)] = X(D) \] (22.45)

Thus the check symbols can also be expressed as
\[ [Y_1(D), \ldots, Y_{N-K}(D)] = [Y_{N-K+1}(D), \ldots, Y_N(D)]P(D) \] (22.46)
so
\[ [Y_1(D), \ldots, Y_{N-K}(D)] - [Y_{N-K+1}(D), \ldots, Y_N(D)]P(D) = [0, \ldots, 0]_{1 \times (N-K)} \] (22.47)

or, equivalently,
\[ [Y_1(D) \cdots Y_N(D)] = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ \cdots & \cdots & \cdots & \cdots \\ -P(D) \end{bmatrix} = [0 0 \cdots 0] \] (22.48)

Note that with GF(2) arithmetic the minus sign in front of \( P \) can be left out. More compactly,
\[ Y(D)H^t(D) = 0 \] (22.49)

where the superscript \( t \) stands for transpose and the \( N - K \times K \) parity check matrix is defined to be
\[ H(D) = [I_{(N-K)\times(N-K)} : -P^t(D)] \] (22.50)
The parity check equation has a very simple interpretation. The check symbols are recomputed from the information symbols found in the last \( K \) positions of the code block and subtracted from the check symbols contained in the first \( N - K \) positions in the block. If the observed block contains no error, the difference must be 0. Every codeword satisfies the check equation and every vector that satisfies the check equation is a codeword.

Suppose a codeword \( Y(D) \) is transmitted and \( R(D) = Y(D) + E(D) \) is received where \( E(D) \) is the channel error sequence. The syndrome for the received sequence is defined to be

\[
S(D) = [S_1(D), \ldots, S_{N-K}(D)] = R(D)H^t(D)
\]

Substituting for \( R(D) \) we find that

\[
S(D) = [Y(D) + E(D)]H^t(D) = Y(D)H^t(D) + E(D)H^t(D)
\]

Observe that the syndrome depends only on the channel error pattern. It does not depend on the transmitted codeword at all.

Now consider a new error pattern \( E'(D) = E(D) + Z(D) \) where \( Z(D) \) is any codeword. The syndrome for this new error pattern is

\[
S'(D) = E'(D)H^t(D) = E(D)H^t(D) = S(D)
\]

Thus, there are many error patterns that have the same syndrome. This set is said to form an equivalence class of error patterns.

**EXAMPLE 22.7  Check Matrix for Ungerboeck 4-State Code**

From the systematic form of the generator matrix for the Ungerboeck 4-state code given by (22.42), it can be seen that \( P(D) = D/(1 + D^2) \). Therefore, a check matrix for this code is

\[
H(D) = \begin{bmatrix} 1 & D \\ 1 + D^2 & 1 \end{bmatrix}
\]

To verify that this is a check matrix, let \( Y(D) \) be any codeword. Then

\[
Y(D)H^t(D) = X(D) \begin{bmatrix} D \\ 1 + D^2 \end{bmatrix} \begin{bmatrix} 1 & D \\ 1 + D^2 & 1 \end{bmatrix} = 2X(D) \begin{bmatrix} D \\ 1 + D^2 \end{bmatrix} = 0
\]

using GF(2) arithmetic for coefficients. Also

\[
(1 + D^2)Y(D)H^t(D) = Y(D) \begin{bmatrix} 1 + D^2 \\ D \end{bmatrix} = 0
\]
Therefore, another check matrix for this code is

\[ \hat{H}(D) = [1 + D^2 \quad D] \] (22.57)

In general, \( H(D) \) can always be multiplied by a polynomial to clear the fractions resulting in a check matrix whose entries are polynomials.

**EXAMPLE 22.8** Check Matrix for the LTE 8-State Code

A check matrix for the LTE turbo code constituent code is

\[ H(D) = \begin{bmatrix}
1 & 1 + D + D^3 \\
1 + D^2 + D^3
\end{bmatrix} \] (22.58)

so that the check equation in the transform domain is

\[ Y(D)H^t(D) = Y_1(D) + Y_2(D) \frac{1 + D + D^3}{1 + D^2 + D^3} = 0 \] (22.59)

or

\[ Y_1(D)(1 + D^2 + D^3) + Y_2(D)(1 + D + D^3) = 0 \] (22.60)

The time-domain check equation is

\[ y_1(n) + y_1(n - 2) + y_1(n - 3) + y_2(n) + y_2(n - 1) + y_2(n - 3) = 0 \] (22.61)

### 22.6 The Code Trellis

The *state* of a sequential circuit is a set of circuit variables such that the current output and next state can be computed from the current input and current state. The contents of the delay elements in a realization of a circuit can be used as a set of state variables. For example, the variables \( q_1(n) \) and \( q_2(n) \) shown in Figure 22.6 can be used as the state variables for the Ungerboeck systematic 4-state code.

The behavior of the circuit can be described by a *trellis diagram*. In a trellis diagram, the states are displayed as a vertical array of nodes which is repeated horizontally for each time instant. The state transitions are shown as lines connecting the nodes in adjacent arrays with circuit outputs labeling the lines. As an example, a section of the trellis diagram for the Ungerboeck systematic 4-state code is shown in Figure 22.8. This encoder has a single input. The transitions caused by a 0 input are shown as solid lines and those caused by a 1 input as dotted lines. In general, the inputs direct the encoder along a path in the trellis.

The trellis diagram can be condensed into a *state transition diagram*. Each state is shown as a node and the possible transitions between states are shown as curved lines with arrows.
The encoder outputs are shown next to the lines. An example for the Ungerboeck systematic 4-state code is shown in Figure 22.9. The numbers in the circles represent the states \([q_1(n), q_2(n)]\) and the numbers next to the lines are the outputs \([y_1(n), y_2(n)]\). Sometimes the input causing a transition is indicated by putting a slash after the outputs and then writing the input value. Since the Ungerboeck example is a systematic code, the output \(y_2(n)\) is always equal to the input \(x_1(n)\) and no extra information is required.

### 22.7 Weight Distributions and Error Correction Properties

The task of a decoder for a convolutional code can be viewed as estimating the path taken through the code trellis by the encoder based on the observed noise corrupted received signal. The Viterbi algorithm [13,14] is one decoding method for decoding convolutional codes. It makes a maximum likelihood estimate of the encoder’s trellis path. Several types of errors
are of interest when analyzing the performance of a decoder. One type called a sequence error occurs when the decoder follows a trellis path that deviates from the encoder’s path. A special type of sequence error is a first-event error which is when the decoder excludes the correct path for the first time at depth $j$ into the trellis. The ultimate goal of a decoder is to form an estimate of the information sequence entering the encoder. Errors in this estimate are called bit errors. The probabilities of these types of errors depend on the code structure, channel characteristics, and decoding algorithm. Exact formulas for these error probabilities are not known but moderately tight upper bounds have been found for some cases.

When the encoder output sequences are transmitted serially over a binary symmetric channel (BSC) or additive white Gaussian noise channel (AWGN) with binary phase-shift keying (BPSK), the error probabilities depend on the Hamming distances between the transmitted code bit sequence along a trellis path and sequences that diverge from the correct path at some point and later remerge with it. The Hamming distance between two binary sequences of the same length is the number of positions in which they differ. The Hamming weight of a sequence is the number of 1’s in it. The number of sequences $n_d$ at distance $d$ from the correct one is also of interest. This set of numbers is called the weight distribution. For linear codes the all 0 sequence can be taken as the transmitted codeword without loss of generality since the weight distribution is the same relative to any codeword.

The weight distribution and additional information can be determined from a modified state transition diagram. An example for the Ungerboeck systematic 4-state code is shown in Figure 22.10. The 0 state is split in two with one part on the left and one on the right. Each branch is labeled with a $D$ whose exponent is the number of 1’s in the code sequence for that branch, an $N$ with an exponent that is the number of 1’s in the information sequence
Weight Distributions and Error Correction Properties

22.7 Weight Distributions and Error Correction Properties

entering the encoder for the branch, and an $L$. Any path from the 0 state on the left to the 0 state on the right is a path that diverges from the all 0's path and remerges with it later for all time. The product of the branch labels along a path has an exponent of $D$ equal to the number of 1’s in the code sequence along the path, an exponent of $L$ equal to the path length in branches, and an exponent of $N$ equal to the number of 1’s in the input data sequence directing the encoder along that path. The sum of the products for all possible paths between the two 0 states is just the transfer function of the enhanced state transition diagram when it is considered to be a flow graph. Viterbi calls this function $T(D, L, N)$.

\[ T(D, L, N) = D^3L^3N^2 \frac{1 - DL + D^3LN^2}{1 - DL(1 + L) + D^2L^3 - D^4L^3N^2} \]  

(22.62)

Dividing the denominator into the numerator, we find that the first few terms in the series expansion of the transfer function are

\[ T(D, L, N) = D^3L^3N^2 + D^4L^5N^2 + D^5L^7N^2 + D^6(L^4N^4 + L^9N^2) + \cdots \]  

(22.63)

Figure 22.10: Opened and Enhanced State Transition Diagram for the Ungerboeck Systematic 4-State Code

EXAMPLE 22.9 Weight Distribution Generating Function for the Ungerboeck Systematic 4-State Encoder

Using the standard techniques for reducing a flow graph, the transfer function from left to right for Figure 22.10 is found to be
This sequence shows that there is one code sequence with Hamming weight 3, with a length of 3 branches, and caused by two 1 inputs. There is one code sequence with Hamming weight 4, length 5, and caused by two 1 inputs. The next term corresponds to a weight 5 path of length 7 caused by two 1 inputs. The last term shows there are two sequences of weight 6 – one of length 4 cause by four 1 inputs and one of length 9 caused by two 1 inputs. You should look for these paths in the enhanced state transition diagram.

Viterbi [13, 14] has shown that when the code bits are transmitted serially over a binary symmetric channel with cross-over probability $p$, the first-event error probability for a Viterbi decoder can be upper bounded in terms of the weight distribution generating function by

$$P_E < T(D, L, N)|_{L=N=1, D=2\sqrt{p(1-p)}}$$

(22.64)

The bit error probability in the decoded information sequence is upper bounded by

$$P_B < \frac{\partial T(D, L, N)}{\partial N}\bigg|_{L=N=1, D=2\sqrt{p(1-p)}}$$

(22.65)

When the code bits are transmitted over an additive white Gaussian noise channel with one-sided noise power spectral density $N_0$ using binary phase shift keying with energy $\epsilon_s$ per transmitted code bit, the first-event error probability is upper bounded by

$$P_E < \text{erfc}\left(\sqrt{\frac{2d\epsilon_s}{N_0}}\exp\left(\frac{d\epsilon_s}{N_0}\right)\right) T(D, L, N)|_{L=N=1, D=\exp(-\epsilon_s/N_0)}$$

(22.66)

and the decoded information sequence bit error probability is upper bounded by

$$P_B < \text{erfc}\left(\sqrt{\frac{2d\epsilon_s}{N_0}}\exp\left(\frac{d\epsilon_s}{N_0}\right)\frac{\partial T(D, L, N)}{\partial N}\right)\bigg|_{L=N=1, D=\exp(-\epsilon_s/N_0)}$$

(22.67)

where

$$\text{erfc}(x) = \int_x^\infty \frac{1}{\sqrt{2\pi}} e^{-t^2/2} \, dt$$

(22.68)

### 22.8 Elements of Lattice Theory

Some definitions and concepts from lattice theory are used in the next section to explain and analyze a technique called trellis coded modulation. The basic elements of lattice theory that will be used there are presented in this section.

#### 22.8.1 Definition of a Lattice

Roughly speaking, a lattice is a regular array of points in an $N$-dimensional space. To define a lattice mathematically, let a basis for the lattice be the following set of $N$ linearly independent vectors over an $M$ dimensional space with $M \geq N$:
\[ x_1 = [x_{1,1}, x_{1,2}, \ldots, x_{1,M}] \]
\[ x_2 = [x_{2,1}, x_{2,2}, \ldots, x_{2,M}] \]
\[ \vdots \]
\[ x_N = [x_{N,1}, x_{N,2}, \ldots, x_{N,M}] \]

For our purposes, we will always let \( N = M \). Let \( i_1, i_2, \ldots, i_N \) be integers. Then, the lattice points are any points with the form
\[ v = i_1 x_1 + i_2 x_2 + \ldots + i_N x_N \quad (22.69) \]

These definitions can also be expressed in matrix form. Let the \textit{generator matrix} for the lattice be the \( N \times M \) matrix
\[ G = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} x_{1,1} & x_{1,2} & \ldots & x_{1,M} \\ x_{2,1} & x_{2,2} & \ldots & x_{2,M} \\ \vdots & \vdots & & \vdots \\ x_{N,1} & x_{N,2} & \ldots & x_{N,M} \end{bmatrix} \quad (22.70) \]
Then any lattice point has the form
\[ v = [i_1, i_2, \ldots, i_N] G \quad (22.71) \]

The sum of any two lattice points is also a lattice point. This can be seen as follows. Let \( v \) be the lattice point defined by (22.69) and let another point be
\[ v' = i'_1 x_1 + i'_2 x_2 + \ldots + i'_N x_N \]
Then
\[ v + v' = (i_1 + i'_1)x_1 + (i_2 + i'_2)x_2 + \ldots + (i_N + i'_N)x_N \]
The coefficients of the basis vectors are integers, so the sum is a lattice point.

The set of lattice points form an algebraic structure known as a commutative group.

\textbf{EXAMPLE 22.10} \ The One-Dimensional Integer Lattice \( \mathbb{Z} \)

This lattice consists of all points on the real line that are integers. Any one-dimensional lattice is simply a scaled version of \( \mathbb{Z} \).

\textbf{EXAMPLE 22.11} \ The Two-Dimensional Integer Lattice \( \mathbb{Z}^2 \)

Figure 22.11 shows the 2-dimensional integer lattice \( \mathbb{Z}^2 \). The lattice consists of all 2-dimensional vectors with integer coordinates. A set of basis vectors for this lattice is
\[ x_1 = [1, 0] \quad \text{and} \quad x_2 = [0, 1] \quad (22.72) \]
and the corresponding generator matrix is

\[
G = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}
\] (22.73)

The generalization of this lattice is the \(N\)-dimensional lattice \(\mathbf{Z}^N\) whose points consist of all the \(N\)-dimensional vectors with integer coordinates.

![Figure 22.11: A Section of the Two-Dimensional Integer Lattice \(\mathbf{Z}^2\)](image)

**EXAMPLE 22.12** The Rotated Integer Lattice \(R\mathbf{Z}^2\)

Figure 22.12 shows the 2-dimensional lattice \(R\mathbf{Z}^2\). This lattice can be obtained by rotating \(\mathbf{Z}^2\) by 45 degrees and scaling it by \(\sqrt{2}\). Thus, \(R\) is a rotation operator. Alternatively, it can be generated by starting with \(\mathbf{Z}^2\) and deleting all the points along the x-axis with odd x-coordinates, moving up one line and deleting all points with even x-coordinates, etc. Equivalently, the lattice consists of all points with integer coordinates for which the sum of the x and y-coordinates is even. A basis for this lattice is

\[
x_1 = [1, 1] \quad \text{and} \quad x_2 = [1, -1]
\] (22.74)
and the corresponding generator matrix is

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

(22.75)

Let \( \mathbf{v} \) be a point in \( \mathbb{Z}^2 \). Then all points in \( R\mathbb{Z}^2 \) have the form \( \mathbf{v}G \). Clearly, the sum of two points of this form is another point of the same form, so \( R\mathbb{Z}^2 \) is a lattice.

**EXAMPLE 22.13** The Scaled Integer Lattice \( 2\mathbb{Z}^2 \)

The lattice \( 2\mathbb{Z}^2 \) shown in Figure 22.13 is simply \( \mathbb{Z}^2 \) scaled by a factor of 2. Notice that \( 2\mathbb{Z}^2 \) is the same as \( R[R\mathbb{Z}^2] = R^2\mathbb{Z}^2 \). In general, \( c\mathbb{Z}^2 \) is \( \mathbb{Z}^2 \) scaled by a factor of \( c \). A basis for \( 2\mathbb{Z}^2 \) is

$$\mathbf{x}_1 = [2, 0] \quad \text{and} \quad \mathbf{x}_2 = [0, 2]$$

(22.76)
22.8.2 Sublattices, Lattice Partitions, and Cosets

A sublattice $\Lambda'$ of a lattice $\Lambda$ is a subset of points from $\Lambda$ that is itself a lattice. That is, the sum of any two points in $\Lambda'$ is also a point in $\Lambda'$. For example, the lattice $R\mathbb{Z}^2$ shown in Figure 22.12 is a sublattice of $\mathbb{Z}^2$. The lattice $2\mathbb{Z}^2$ shown in Figure 22.13 is also a sublattice of $\mathbb{Z}^2$. In addition, $2\mathbb{Z}^2$ is a sublattice of $R\mathbb{Z}^2$.

A lattice is a type of algebraic structure known as a group. A sublattice is an algebraic structure called a subgroup of the original group.

**Definition.** A group consists of the following:

1. A set of elements $\Lambda$

2. A rule, which we will call $+$, that associates with each pair of elements $\lambda_1$ and $\lambda_2$ in $\Lambda$ an element $\lambda_1 + \lambda_2$ in $\Lambda$ and has the following properties:

   (a) **Associative Law:** For all $\lambda_1$, $\lambda_2$, and $\lambda_3$ in $\Lambda$, 
       $$\lambda_1 + (\lambda_2 + \lambda_3) = (\lambda_1 + \lambda_2) + \lambda_3$$

   (b) **Identity Element:** $\Lambda$ contains an identity element $0$ such that $0 + \lambda = \lambda + 0 = \lambda$ for every $\lambda$ in $\Lambda$.

   (c) **Inverses:** For each element $\lambda$ in $\Lambda$ there is an inverse element $-\lambda$ in $\Lambda$ such that $\lambda + (-\lambda) = (-\lambda) + \lambda = 0$. 

Figure 22.13: A Section of the Lattice $2\mathbb{Z}^2$
A translation of a sublattice by an element of the original lattice is called a coset of the sublattice. That is, let \( \Lambda' \) be a sublattice of \( \Lambda \) and let \( \lambda \) be an element of \( \Lambda \). Then the translation \( \lambda + \Lambda' = \{ \lambda + \lambda' \mid \lambda' \in \Lambda' \} \) is a coset of \( \Lambda' \) in \( \Lambda \). The element \( \lambda \) is called a coset representative. The coset generated by \( \lambda \) is sometimes designated by \( \{ \lambda \} \).

According to the definition of a coset, \( \lambda \) could be chosen as an element of the sublattice \( \Lambda' \). The resulting coset is then just the sublattice itself. The identity element \( 0 \) belongs to every lattice, so \( \Lambda' = \{ 0 \} \).

**EXAMPLE 22.14**

We have observed that the lattice \( R \mathbf{Z}^2 \) in Figure 22.12 is a sublattice of \( \mathbf{Z}^2 \). A coset of \( R \mathbf{Z}^2 \) in \( \mathbf{Z}^2 \) is the translation \( (1, 0) + R \mathbf{Z}^2 \), which is Figure 22.12 shifted to the right by one. This is shown in Figure 22.14 where \( \Lambda' = R \mathbf{Z}^2 \) is shown by circles and the coset by solid squares. The sublattice \( \Lambda' = R \mathbf{Z}^2 \) is the coset \( (0, 0) + R \mathbf{Z}^2 \). The lattice points \((0, 0)\) and \((1, 0)\) are one possible pair of coset representatives. The entire set of lattice points in the coset represented by \((0, 0)\) is designated by \( \{(0, 0)\} \) and, similarly, the set of points in the coset represented by \((1, 0)\) is represented by \( \{(1, 0)\} \). Notice that the coset generated by \((1, 0)\) is the only coset of \( R \mathbf{Z}^2 \) distinct from \( R \mathbf{Z}^2 \) and that the union of \( R \mathbf{Z}^2 \) and this coset forms \( \mathbf{Z}^2 \).

**EXAMPLE 22.15**

The distinct cosets of the lattice \( 2 \mathbf{Z}^2 \) shown in Figure 22.13 with respect to \( \mathbf{Z}^2 \) are the translations of \( 2 \mathbf{Z}^2 \) by the four vectors \((0, 0)\), \((1, 0)\), \((0, 1)\), and \((1, 1)\).

It can be shown that the union of all the distinct cosets of a sublattice is the original lattice. (In group theory, the union of the cosets of a subgroup is the original group.) The set of distinct cosets of a sublattice \( \Lambda' \) of a lattice \( \Lambda \) is called a partition of \( \Lambda \) or, in group theory language, a quotient group and is designated by \( \Lambda / \Lambda' \). The number of distinct cosets in the partition (including the sublattice) is called the order of the partition and is designated by \( |\Lambda / \Lambda'| \). Thus the partition of \( \mathbf{Z}^2 \) generated by \( R \mathbf{Z}^2 \) is denoted by \( \mathbf{Z}^2 / R \mathbf{Z}^2 \) and has order 2. Also, \( \mathbf{Z}^2 / 2 \mathbf{Z}^2 \) has order 4.

Let \( \{ c_1, c_2, \ldots, c_L \} \), where \( L = |\Lambda / \Lambda'| \), be a set of distinct coset representatives for the partition \( \Lambda / \Lambda' \). This set of coset representatives is sometimes designated by \( [\Lambda / \Lambda'] \). Therefore

\[
\Lambda = \bigcup_{i=1}^{L} \{ c_i + \Lambda' \} = [\Lambda / \Lambda'] + \Lambda' \quad (22.77)
\]

The quotient group consisting of the set of cosets generated by partitioning \( \Lambda \) by a sublattice \( \Lambda' \) is actually a group. The elements of this group are the cosets \( \{ c_i \} \) for \( i = 1, \ldots, |\Lambda / \Lambda'| \). Any element from the coset generated by \( c_i \) has the form \( c_i + \lambda_1 \), where \( \lambda_1 \in \Lambda' \). Similarly, elements in the coset generated by \( c_j \) have the form \( c_j + \lambda_2 \), where \( \lambda_2 \) is also an element of \( \Lambda' \). Therefore, the sum of an element from coset \( i \) and an element from coset \( j \) has the form \( c_i + c_j + \lambda_1 + \lambda_2 \). However, both \( \lambda_1 \) and \( \lambda_2 \) are elements of \( \Lambda' \), so \( \lambda_1 + \lambda_2 = \lambda' \) is also an element of \( \Lambda' \). Thus \( c_i + c_j + \lambda_1 + \lambda_2 = c_i + c_j + \lambda' \) which is an
element in the coset generated by $c_k = c_i + c_j$. Based on this conclusion, the addition rule, $\boxplus$, for the elements of the quotient group of cosets is
\[
\{c_i\} \boxplus \{c_j\} = \{c_i + c_j\}
\] (22.78)

Since vector addition is associative, it follows that $\boxplus$ is also associative. The coset $\{0\}$ is the identity element. For each coset representative $c_i$, its negative $-c_i$ is an element of some coset $\{c_j\}$, so $\{c_i\} \boxplus \{c_j\} = \{c_i - c_i\} = \{0\}$ and each element has an inverse. Thus the quotient group satisfies all the abstract group axioms.

The elements of the lattice partition $\Lambda/\Lambda'$ can themselves be partitioned with respect to a sublattice $\Lambda''$ of $\Lambda'$. The sublattice $\Lambda''$ generates the partition $\Lambda'/\Lambda''$ of $\Lambda'$ with $|\Lambda'/\Lambda''|$ elements. In terms of the notation presented above, $[\Lambda'/\Lambda'']$ is a set of coset representative for this partition and $\Lambda' = [\Lambda'/\Lambda''] + \Lambda''$. Consequently, the original lattice $\Lambda$ can be represented by
\[
\Lambda = [\Lambda/\Lambda'] + \Lambda' = [\Lambda/\Lambda'] + [\Lambda'/\Lambda''] + \Lambda''
\] (22.79)

Figure 22.14: Partition of $\mathbb{Z}^2$ into $R\mathbb{Z}^2$ and its Coset
In words, any element of the lattice $\Lambda$ can be expressed as a sum of a coset representative from the first level partition plus a representative from the second level partition plus an element of the sublattice $\Lambda''$. The original lattice and its two sublattices can be represented by the *partition chain* $\Lambda/\Lambda'/\Lambda''$. This chain induces the partition of $\Lambda$ given by (22.79). The number of elements in the overall partition is

$$|\Lambda/\Lambda''| = |\Lambda/\Lambda'| \times |\Lambda'/\Lambda''|$$  (22.80)

**EXAMPLE 22.16**

The lattice $\Lambda'' = 2\mathbb{Z}^2$ is a sublattice of $\Lambda' = R\mathbb{Z}^2$ and generates a partition of $\Lambda'$ of order 2. It can be obtained by rotating $R\mathbb{Z}^2$ by 45 degrees and scaling by the square root of 2. The sequence of partitions of $\mathbb{Z}^2$ with respect to $R\mathbb{Z}^2$ and $R\mathbb{Z}^2$ with respect to $2\mathbb{Z}^2$ is induced by the partition chain $\mathbb{Z}^2/R\mathbb{Z}^2/2\mathbb{Z}^2$. Notice that the order of the partition of the first lattice in the chain, $\mathbb{Z}^2$, generated by the last lattice, $2\mathbb{Z}^2$, is the product of the orders of the subpartitions in the chain, that is, $|\mathbb{Z}^2/2\mathbb{Z}^2| = |\mathbb{Z}^2/R\mathbb{Z}^2| \times |R\mathbb{Z}^2/2\mathbb{Z}^2| = 2 \times 2 = 4$.

More generally, a *partition chain* $\Lambda/\Lambda_1/\Lambda_2/\ldots/\Lambda_N$ is a sequence of lattices where each is a sublattice of the one to the left. This chain induces the partition

$$\Lambda = [\Lambda/\Lambda_1] + [\Lambda_1/\Lambda_2] + \cdots + [\Lambda_{N-1}/\Lambda_N] + \Lambda_N$$  (22.81)

The number of elements in the overall partition is

$$|\Lambda/\Lambda_N| = |\Lambda/\Lambda_1| \times |\Lambda_1/\Lambda_2| \times \cdots \times |\Lambda_{N-1}/\Lambda_N|$$  (22.82)

### 22.9 Trellis Coded Modulation (TCM)

By the mid 1970’s people began to express the opinion that there was little to be gained by error control coding for heavily band limited channels like voice band telephone circuits. The envisioned problem was that coding requires adding redundancy which seemed to require transmitting more channel symbols per second for a fixed information rate which causes bandwidth expansion. Then Ungerboeck and Csajka [10] disclosed a major breakthrough at the 1976 IEEE Information Theory Symposium showing how significant coding gains could be achieved without expanding bandwidth by combining convolutional coding and QAM modulation. Their method was to add redundancy by expanding the QAM 2D constellation size beyond the $2^K$ points required for $K$ information bits, partitioning the constellation into a sequence of subsets with increasing minimum squared Euclidean distance between subset points at each partition level, and using the output bits of the convolutional encoder to assign constellation subsets to trellis branches rather than binary encoded bits. This caused no bandwidth expansion at all. Based on capacity curves, they argued that almost all the potential gain could be achieved by simply doubling the constellation size which meant the convolutional encoder should generate just one check bit and be a $(K+1,K)$ encoder. Gains
of more than 3 dB could easily be achieved and the method was very quickly included in commercial wireline modems. The method is now known as *trellis coded modulation* (TCM). Calderbank and Sloane [3] and Forney [4, 5] formalized the set partitioning idea by using the language and theory of lattices. They recognized that the constellation partitioning process was equivalent to coset decompositions of lattices and lattice partition chains.

The block diagram of a typical systematic trellis coded modulation system is shown in Figure 22.15. The serial input data stream is divided into the $K+U$ streams $X_1(D), \ldots, X_{K+U}(D)$. The $K$ input streams $X_1(D), \ldots, X_K(D)$ are applied to a $(K + 1, K)$ systematic convolutional encoder to generate the single check stream $Y_0(D)$. The $K+1$ signals consisting of the check stream and first $K$ information bit streams are used to select one of $2^{K+1}$ cosets of a lattice partition $\Lambda_c/\Lambda'_c$ at each time instant. The actual constellation points typically belong to a translation of the coding lattice $\Lambda_c$ so that the zero lattice point is never transmitted. For example, the coding lattice might be $\mathbb{Z}^2$ with the constellation points selected from $\mathbb{Z}^2 + (0.5, 0.5)$. The coding sublattice $\Lambda'_c$ must be selected so that the partition $\Lambda_c/\Lambda'_c$ has order $2^{K+1}$. The remaining $U$ input bit streams $X_{K+1}(D), \ldots, X_U(D)$ select one of $2^U$ points from the designated coset of the coding sublattice $\Lambda'_c$ at each time instant. These bits are often called the *uncoded bits*.

According to Ungerboeck’s set partitioning method, the encoder output bits $[y_0(n), y_1(n), \ldots, y_K(n)]$ specify a path in the tree corresponding to a $K + 1$ level partition chain. The bits sequentially specify how to split the subset of points at the corresponding level into two sets of points. Thus at level $K + 1$ there are $2^{K+1}$ subsets. The bit $y_0(n)$ determines which of the two partitions of $\Lambda_c$ is chosen at the top level, $y_1(n)$ determines which of the two partitions are chosen at the next level, etc. At each level the minimum distance between points within a coset (or subset, as they are called by Ungerboeck) increases. The minimum distance between points at the top level, that is, in the lattice $\Lambda_c$, will be designated by $d_0$. 

![Figure 22.15: Block Diagram for a Systematic Trellis Coded Modulator](image-url)
at the next level by \(d_1\), etc.

The error probability for a trellis code can be estimated at high signal-to-noise ratios from the minimum Euclidean distance between constellation sequences selected by the encoder along trellis paths that diverge for the first time at a node and later remerge forever. This minimum distance will be called \(d_{\text{min}}\).

**EXAMPLE 22.17** Ungerboeck’s Systematic 4-State Trellis Code

Ungerboeck’s 4-state trellis code is based on the partition chain \(\mathbb{Z}^2/R\mathbb{Z}^2/2\mathbb{Z}^2\). In practice, the transmitted constellation points would be selected from the translated lattice \(\mathbb{Z}^2 + [0.5\ 0.5]\) so that the zero lattice point is not transmitted. The translation will be ignored in the rest of this example. The partition \(\mathbb{Z}^2/2\mathbb{Z}^2\) has order 4 and each sub-partition has order 2.

Consider the systematic version of Ungerboeck’s 4-state encoder shown in Figure 22.6. No uncoded bits are shown for this encoder, so the encoder outputs \(y_1\) and \(y_2\) select subsets of the partition. (Notice that in this example the subscripts for the \(y\)’s start at 1 rather than 0 as in Figure 22.15.) The partition tree is shown in Figure 22.16. This corresponds to the partition equation

\[
\mathbb{Z}^2 = 2\mathbb{Z}^2 + y_1[0\ 1] + y_2[1\ 1] \tag{22.83}
\]

Notice from the tree that

\[
R\mathbb{Z}^2 = 2\mathbb{Z}^2 \cup \{2\mathbb{Z}^2 + [1\ 1]\} \tag{22.84}
\]

and

\[
\mathbb{Z}^2 = R\mathbb{Z}^2 \cup \{R\mathbb{Z}^2 + [0\ 1]\} \tag{22.85}
\]

The minimum squared Euclidean distance (MSED) for this code will now be determined. The trellis diagram for this code is repeated in Figure 22.17 with the all zeros path and the minimum distance path that diverges from all zeros and remerges shown by wider lines. The labels \(y_1\ y_2\) on the branches represent the cosets at the bottom of the partition tree. Notice that the two branches that diverge from any node always have the same \(y_1\) bit. Either \(y_1 = 0\) for both or \(y_1 = 1\) for both. Also, the two branches that converge on each node have the same \(y_1\) value. This is shown explicitly in the trellis diagram but also becomes evident from the encoder block diagram, Figure 22.6. The current \(y_1\) value does not depend on the current input bit, so paths diverging from the current state have the same \(y_1\). The value of \(y_1(n-1)\) for paths converging on a state at time \(n\) is the value of the state variable \(q_1(n)\) which also is not affected by the current input \(x_1(n)\). Therefore, the two paths converging on a state must have had the same \(y_1\) at the previous time instant. The values of \(y_2\) for branches converging on or diverging from a state are always different.

Whenever \(y_1\) is the same for two branches, the constellation points must be from the same first level partition, that is, either from \(R\mathbb{Z}^2\) for \(y_1 = 0\) or from \(R\mathbb{Z}^2 + [0\ 1]\) for \(y_1 = 1\). In either case, the squared distance between the two points must be no less than the minimum in \(R\mathbb{Z}^2\) which is \(d^2_1 = 2\). The two darkened paths in the trellis diagram shown in Figure 22.17 extend over three branches. The segments that diverge from state 00 or remerge with state 00 must each have minimum squared distance 2. The branches of the middle segment...
Figure 22.16: The Partition Tree

have $[y_1 \ y_2] = [0 \ 0]$ or $[1 \ 0]$. Thus, the constellation points on the upper branch belong to $RZ^2$ and on the lower branch to $RZ^2 + [0 \ 1]$. Since the union of these two subsets is $Z^2$, the minimum squared distance between them is the minimum distance in $Z^2$ which is $d_0^2 = 1$. Therefore, the minimum squared distance between the two paths is

$$d_f^2 = 2d_1^2 + d_0^2 = 5 \quad (22.86)$$

The distance between paths that actually diverge for more than one branch before remerging is called the minimum free distance. Each branch in the trellis represents a coset selection. The particular point in the coset is selected by the uncoded input bits. Thus, each branch really represents a set of parallel transitions which are paths that diverge and remerge one node later. The minimum distance, $d_p = 2$, within each coset of the coding sublattice $2Z^2$ has to be considered when finding the minimum distance for the trellis code. The minimum squared Euclidean distance for the trellis code is

$$d_{\text{min}}^2 = \min\{d_p^2, d_f^2\} = 4 \quad (22.87)$$

Ungerboeck has given a set of rules for selecting good trellis codes. One of his rules is that paths diverging from a state should have the same first level partition bit ($y_0$ in Figure 22.15 or $y_1$ in the previous example) and, similarly, all paths converging on a state should have the same first level partition bit. This ensures a minimum squared distance along diverging and remerging paths of at least $2d_1^2$. For systematic $(K + 1, K)$ codes, this can be insured
by having no input bits add into the input of the delay element on the left or the output of the delay element on the right. This is clearly the case for the systematic form of the Ungerboeck 4-state code shown in Figure 22.6.

Actually, the signal constellation does not have to be a portion of a lattice. Ungerboeck presents an example of using his 4-state code with an 8-phase constellation in [12] to transmit 2 bits per symbol with a 3 dB gain over uncoded transmission using a 4-phase constellation. The 8-phase constellation is partitioned into two 4-phase constellations selected by the encoder output bit \( y_1(n) \) shown in Figure 22.6 with the point in the 4-phase constellation determined by the bit \( y_2(n) \) and an additional uncoded coded bit not shown in the figure. The method to transmit \( K \) bits/symbol using trellis coded modulation is to choose a constellation with \( 2^{K+1} \) points, which is double the number of points required for uncoded transmission. The TCM constellation is then successively partitioned into subsets with half the number of points and increasing minimum distance between points. The sequence of subset partitions is specified by the coded input bits and the point within the final partition is determined by the uncoded input bits. This technique is known as set partitioning.
22.10 The Viterbi Decoding Algorithm

Consider an \((N, K)\) convolutional encoder that starts in the zero state. Suppose \(L\) blocks of \(K\) information bits are encoded and \(L_0\) additional blocks are transmitted to drive the final encoder state back to zero so the total number of transmitted blocks is \(L_T = L + L_0\). The total number of bits output by the encoder is \(N_T = N(L + L_0)\) for the \(LK\) input information bits. There are \(2^{LK}\) possible distinct paths through the trellis from the initial to final zero state. If the input bits are equally likely, the resulting output sequences are also equally likely. The Viterbi decoding algorithm is a method for finding the maximum likelihood estimate of a transmitted trellis sequence given the corresponding received sequence. This rule minimizes the probability of making a sequence error when the transmitted sequences are equally likely, that is, the probability of choosing the wrong sequence. The maximum likelihood estimation rule is derived for three common channels in the following three examples. Then the Viterbi algorithm for efficiently implementing these rules is presented.

22.10.1 Three Channel Models

EXAMPLE 22.18 The Memoryless Binary Symmetric Channel (BSC)

The input to a BSC is a stream of binary digits with values 0 or 1. The output is also a stream of binary digits but each received bit can be complemented, that is, received in error with probability \(p\). They are received correctly with probability \(1 - p\). Errors in different bits are statistically independent. Quantizing the received signals to 0 or 1 is called making hard bit decisions. Let the transmitted bits for a branch at depth \(n\) in the trellis be \(y(n) = [y_1(n), y_2(n), \ldots, y_N(n)]\) and the corresponding received hard bit decisions be \(r(n) = [r_1(n), \ldots, r_N(n)]\). Let the entire transmitted bit sequence be denoted by \(y = [y(0), \ldots, y(L_T - 1)]\) and the received sequence by \(r = [r(0), \ldots, r(L_T - 1)]\). Then the probability of the received sequence given the transmitted sequence is

\[
P(r|y) = \prod_{n=0}^{L_T-1} P(r(n)|y(n)) = \prod_{n=0}^{L_T-1} \prod_{i=1}^{N} P(r_i(n)|y_i(n))
\]

Each factor \(P(r_i(n)|y_i(n))\) is \(1 - p\) if \(r_i(n) = y_i(n)\) and \(p\) if \(r_i(n) \neq y_i(n)\). The Hamming distance, \(d(x, y)\), between two binary sequences \(x\) and \(y\) of equal length is the number of places in which they differ. Therefore,

\[
P(r|y) = p^{d(r, y)}(1 - p)^{N_T - d(r, y)} = (1 - p)^{N_T} \left( \frac{p}{1-p} \right)^{d(r, y)}
\]

Usually the crossover probability \(p\) is much less than the correct transmission probability \(1 - p\) so \(p/(1-p)\) is less than 1. Thus a straightforward method of implementing the maximum likelihood decision rule is:
Choose the trellis sequence $y$ that is closest to the received sequence $r$ in Hamming distance.

Of course, this can be a computationally intensive task because there can be a large number, $2^{LK}$, of trellis sequences to compare to the received sequence.

The total Hamming distance $d(r, y)$ is called the cumulative path metric. The Hamming distance $d(r(n), y(n))$ is called the branch metric at depth $n$. The cumulative metric is the sum of the branch metrics, that is,

$$d(r, y) = \sum_{n=0}^{L_T-1} d(r(n), y(n)) \quad (22.90)$$

This decomposition of the cumulative metric will be an important property allowing for an efficient search for the best trellis path by the Viterbi algorithm.

**EXAMPLE 22.19** Biphase Data Over an Additive White Gaussian Noise Channel

In this case the code bits, $y_i(n)$, which can be 0 or 1 are converted to the analog levels $\tilde{y}_i(n) = (-1)^{y_i(n)}$ which can be 1 or $-1$ for a code bit duration of $T$ seconds. This transmission method is known as binary phase shift keying (BPSK). Let $q(t)$ be a signal pulse that exists for $0 \leq t < T$ and is zero elsewhere. Then the transmitted signal is

$$s(t) = \sum_{n=0}^{L_T-1} \sum_{i=1}^{N} \tilde{y}_i(n) q[t - (i - 1)T - nNT] \quad (22.91)$$

and the received signal is $r(t) = s(t) + n(t)$ where $n(t)$ is white Gaussian noise with two-sided power spectral density $N_0/2$. The energy in each transmitted code bit is

$$E_s = \int_0^T q^2(t) \, dt \quad (22.92)$$

The received analog signal is passed through a matched filter which forms

$$r_i(n) = \frac{1}{E_s} \int_{nNT + (i-1)T}^{nNT + iT} r(t)q[t - nNT - (i - 1)T] \, dt$$

$$= \frac{1}{E_s} \int_0^T \tilde{y}_i(n) q^2(t) \, dt + \frac{1}{E_s} \int_0^T n[t + nNT + (i - 1)T]q(t) \, dt$$

$$= \tilde{y}_i(n) + z_i(n) \quad (22.93)$$

where the noise component in the matched filter output is

$$z_i(n) = \frac{1}{E_s} \int_0^T n[t + nNT + (i - 1)T]q(t) \, dt \quad (22.94)$$
It can be shown that the noise term, \( z_i(n) \), is a zero mean, Gaussian, random variable with variance \( \sigma^2 = N_0/(2E_s) \). The different noise terms are statistically independent. Hard bit decisions are made by deciding \( y_i(n) = 0 \) was transmitted if \( r_i(n) \geq 0 \) and \( y_i(n) = 1 \) was transmitted if \( r_i(n) < 0 \). The probability of making a decision error can be shown to be

\[
p_b = \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \, dx = Q \left( \sqrt{\frac{2E_s}{N_0}} \right) \tag{22.95}
\]

Let the transmitted analog sequence for a trellis path branch at depth \( n \) be \( \tilde{y}(n) = [\tilde{y}_1(n), \ldots, \tilde{y}_K(n)] \) and the corresponding received branch symbols be \( r(n) = [r_1(n), \ldots, r_N(n)] \). As in the previous example, let the entire transmitted analog sequence be denoted by \( \tilde{y} = [\tilde{y}(0), \ldots, \tilde{y}(L_T - 1)] \) and the received sequence by \( r = [r(0), \ldots, r(L_T - 1)] \). Then the probability density function for the received sequence given the transmitted biphase sequence is

\[
f(r|y) = \prod_{n=0}^{L_T-1} f(r(n)|y(n)) = \prod_{n=0}^{L_T-1} \prod_{i=1}^{N} f(r_i(n)|y_i(n)) = \frac{1}{(2\pi\sigma^2)^{NT}} \exp \left( -\frac{1}{2\sigma^2} \sum_{n=0}^{L_T-1} \sum_{i=1}^{N} (r_i(n) - \tilde{y}_i(n))^2 \right) \tag{22.96}
\]

Maximizing this probability with respect to the transmitted trellis sequence is equivalent to choosing the trellis sequence that minimizes

\[
\sum_{n=0}^{L_T-1} \sum_{i=1}^{N} (r_i(n) - \tilde{y}_i(n))^2 = \sum_{n=0}^{L_T-1} \sum_{i=1}^{N} r_i^2(n) - \sum_{n=0}^{L_T-1} \sum_{i=1}^{N} r_i(n)\tilde{y}_i(n) + \sum_{n=0}^{L_T-1} \sum_{i=1}^{N} \tilde{y}_i^2(n) \tag{22.97}
\]

This sum is the squared Euclidean distance between \( r \) and \( y \), so the maximum likelihood estimate for \( y \) is the trellis sequence that is closest to \( r \) in Euclidean distance. The sum over \( r_i^2(n) \) does not depend on the trellis sequence. Since \( \tilde{y}_i^2(n) = 1 \), the sum on the far right is \( L_TN \) and does not depend on the chosen trellis sequence. Therefore, the maximum likelihood decision rule can be also be stated as:

Choose the trellis path that maximizes the correlation

\[
\Gamma(r, \tilde{y}) = \sum_{n=0}^{L_T-1} \sum_{i=1}^{N} r_i(n)\tilde{y}_i(n) \tag{22.98}
\]

The quantity \( \Gamma(r, \tilde{y}) \) is called the cumulative path metric. The partial correlation

\[
\gamma(r(n), \tilde{y}(n)) = \sum_{i=1}^{N} r_i(n)\tilde{y}_i(n) \tag{22.99}
\]
is called the branch metric at depth \( n \). Thus the cumulative metric can again be decomposed into the sum of the branch metrics, that is,

\[
\Gamma(r, \tilde{y}) = \sum_{n=0}^{L_T-1} \gamma(r(n), \tilde{y}(n))
\] (22.100)

**EXAMPLE 22.20** Quadrature Amplitude Modulation (QAM) Over an Additive White Gaussian Noise Channel

Two dimensional (2D) constellation points, \( u(n) \), are selected by the \( N \) code bits on the trellis path branches. These symbols will be considered to be complex numbers. They are transmitted over an additive, white, Gaussian noise channel by quadrature amplitude modulation as discussed in Chapter 13. The received demodulated and equalized symbols have the form

\[
r(n) = u(n) + v(n) \quad \text{for} \quad n = 0, \ldots, L_T - 1
\] (22.101)

where \( v(n) \) is a white, complex, zero-mean Gaussian noise sequence with variance \( \sigma^2 = E\{|v(n)|^2\} \) and \( r(n) \) is a sequence of complex numbers ideally close to the transmitted symbols. Let the entire sequence of transmitted constellation points be denoted by \( u = [u(0), \ldots, u(L_T - 1)] \) and the entire received sequence by \( r = [r(0), \ldots, r(L_T - 1)] \). The conditional probability density function for the received sequence given a possible trellis sequence is

\[
f(r(0), \ldots, r(L_T - 1) | u(0), \ldots, u(L_T - 1)) = \frac{1}{(\sigma^2 \pi)^{L_T/2}} e^{-\sum_{n=0}^{L_T-1} |r(n) - u(n)|^2 / \sigma^2}
\] (22.102)

A maximum likelihood receiver selects the sequence \( u(n) \) that maximizes this pdf. It is maximized if \( u(n) \) is chosen to minimize

\[
\Gamma(r, u) = \sum_{n=0}^{L_T-1} |r(n) - u(n)|^2
\] (22.103)

The function \( \Gamma(r, u) \) is the squared Euclidean distance between the received sequence \( r(n) \) and hypothesized transmitted trellis sequence \( u(n) \). \( \Gamma(r, u) \) is called the cumulative sequence metric for the selected trellis path. An individual term, \( |r(n) - u(n)|^2 \), in the sum is called a branch metric. Stated in another way, a maximum likelihood decoder should select the trellis sequence that is closest to the received sequence in Euclidean distance. Once again, the cumulative metric is the sum of the branch metrics.
22.10.2 Detailed Explanation of the Viterbi Algorithm

From here on it will be assumed that the encoder starts in state 0 at time \( n = 0 \) and is forced to end in state 0 at trellis depth \( L_T \) by appending an appropriate string of dummy information bits to the input data stream. “Time” will be used interchangeably with “depth”. For example, if the encoder is realized using the type 1 direct form shown in Figure 22.2 with \( M \) delay elements, the last \( M \) inputs should equal the feedback signal to the adder on the left to clear the delay line to the zero state.

The maximum likelihood path through the trellis can be found by iteratively progressing from the beginning to the end of the trellis. Let the encoder have \( S \) states labeled 0, \ldots, \( S - 1 \), and let the state at time \( n \) be \( s(n) \). The trellis is forced to have the boundary conditions \( s(0) = s(L_T) = 0 \). Suppose that somehow we know the best trellis paths and corresponding sums of branch metrics to each of the states at a specific time \( n \). The sums of branch metrics will be called the *cumulative metrics* to the states at time \( n \). A trellis path consists of the sequence of states traversed and the branch bits, or, equivalently, transmitted symbols associated with the branches connecting the states. By “best trellis path to a state” we mean the path from state 0 at time 0 to a state at time \( n \) that has the “best” sum of branch metrics along any path ending in the state at time \( n \). “Best” means the minimum sum for the BSC and QAM examples and the maximum branch correlation sum for the biphase data example.

The Viterbi algorithm will now be presented using the QAM example with trellis coded modulation. It can easily be modified for other examples. Hamming distance should be minimized for the BSC example and the correlation should be maximized for the biphase example. Let the sequence of constellation points along the best path to state \( i \) at time \( n \) be \( u^*(k; s(n) = i) \) for \( k = 0, \ldots, n \) and let the cumulative metric for the best path to state \( i \) at time \( n \) be

\[
\Gamma^*_i(n) = \sum_{k=0}^{n-1} |r(k) - u^*(k; s(n) = i)|^2
\]  

(22.104)

The best paths are called the *survivors* to the states. Any path from the beginning to the end of the trellis that passes through a particular state at time \( n \) must follow the survivor to that state up to time \( n \) since the survivor has the minimum metric up to that state and the branch metrics for paths leading out of the state get added to the cumulative metric of the survivor. No other path through that state can have a smaller overall cumulative metric. At time \( n \) we can fill in a data record consisting of (1) an \( S \)-dimensional array where the \( i \)th element is the cumulative metric of the survivor to state \( i \) at time \( n \), (2) an \( S \times L_T \) dimensional array where element \( (i, n) \) is the state at time \( n - 1 \) where the survivor to state \( i \) at time \( n \) came from, that is, pointers to the previous state along the survivor to state \( i \) at time \( n \), and (3) an \( S \times L_T \) dimensional array where element \( (i, n) \) contains the subset point selected for the surviving branch connecting to state \( i \) at time \( n \). Item (3) must be saved with a systematic encoder because from knowledge of two successive states in the trellis only the constellation subset can be determined. The point within the subset is selected by the uncoded bits and this information must be saved. For a traditional binary convolutional code, there are no parallel transitions and item (3) is not required. The array in (2) is used.
to find the surviving path to the zero state at the end of the trellis by *tracing* the path back from state 0 at the end of the trellis.

Next the survivors and their cumulative metrics to the states at time \( n + 1 \) can be found using the known results at time \( n \). When the convolutional encoder has \( K \) inputs, \( 2^K \) paths diverge from and converge on each state. The \( 2^K \) candidates for the survivor to a state \( i \) at time \( n + 1 \) must be extensions of the survivors to states at time \( n \) that have branches connecting to state \( i \) at time \( n + 1 \). Let \( C_i \) be the set of \( 2^K \) states converging on state \( i \). A branch converging on state \( i \) from a state \( j \in C_i \) has a subset of \( 2^U \) constellation points, where \( U \) is the number of uncoded input bits, assigned to it which we will call \( U_{j,i} \). Since the cumulative metric at time \( n + 1 \) is the sum of the cumulative metric of the survivor to state \( j \) at time \( n \) and the branch metric for the branch connecting to state \( i \) at time \( n + 1 \), the best choice for the constellation point along this path must be the point \( u_{j,i}^*(n) \in U_{j,i} \) that is closest to the received point \( r(n) \) in Euclidean distance. The process of selecting this subset point is often called *slicing* the received point to the subset. After performing slicing, the \( 2^K \) cumulative metrics for the survivor candidates to state \( i \) at time \( n + 1 \) are

\[
\Gamma_{j,i}(n + 1) = |r(n) - u_{j,i}^*(n)|^2 + \Gamma_j^*(n) \quad \text{for } j \in C_i
\]

The Viterbi decoder then selects the \( j \in C_i \) which gives the minimum \( \Gamma_{j,i}(n + 1) \) and records this new survivor metric and optimum previous state in the decoding data record for time \( n + 1 \). The cumulative metric for the survivor to state \( i \) at time \( n + 1 \) is

\[
\Gamma_i^*(n + 1) = \min_{j \in C_i} \{|r(n) - u_{j,i}^*(n)|^2 + \Gamma_j^*(n)\}
\]

This survivor selection must be performed for each state at time \( n + 1 \).

The process of survivor selection can be started at time \( n = 0 \) when the encoder is known to be in state 0 and extended stage-by-stage into the trellis until time \( L_T \) when the encoder is again known to be in state 0. At depth \( L_T \) all trellis paths converge to state 0 and a single survivor to this state is selected. This survivor to state 0 at depth \( L_T \) is the maximum likelihood estimate of the transmitted sequence. Since the stored data record for each state at a given time contains a pointer back to the previous best state connecting to it, the path can be *traced back* from the survivor to state 0 at depth \( L_T \) to the beginning of the trellis to find this maximum likelihood path. The use of pointers eliminates the computational burden of swapping entire paths from the beginning of the trellis up to the current depth as new survivors are found at each iteration.

### 22.10.3 Some Practical Implementation Techniques

In some cases, the trellis length, \( L_T \), is essentially unlimited. For example, the trellis code in a telephone line modem runs from the time the modem initially connects until the call is terminated. This can be several hours so \( L_T \) is very large and the storage for the decoding data records becomes unreasonably large. The solution is to limit the storage to \( I \) stages back from the current time where \( I \) is several times the memory length of the encoder. That
is, the records are stored from time $n$ back to time $n - I$, typically, in a circular buffer. Simulations have shown that as paths are traced back from the states at time $n$, they all merge into a common tail when there is not excessive noise. A simulation example for the Ungerboeck 4-state code using an 8-PSK constellation described in Section 22.15.6 is shown in Figure 22.18 when there is no noise. Another example when the noise variance in each quadrature is 0.15 is shown in Figure 22.19. In this second example the surviving paths do not merge until ten branches back. Little is to be gained by saving records back beyond the point where the paths merge into the common tail. No good design equations exist for choosing $I$. The best approach seems to be by simulations. As $I$ is increased, the coding gain initially increases rapidly but then exponentially converges to a maximum value. The decoder at each time instant $n$ should find the state with minimum cumulative metric and trace the path from that state back $I$ stages to the end of the data record storage and output the constellation point stored there. A decoder of this type introduces a decoding delay of $I$ trellis branches. It may seem that the trace back can be started from any state since it is assumed all the paths merge into a common tail. However, simulations have shown that when $I$ is not very large, starting the trace back from the state with minimum metric can speed up the convergence of the cumulative metrics.

![Figure 22.18: Surviving Paths for the Ungerboeck 4-State Code with an 8-Phase Constellation and No Noise](image-url)

Another problem is that the cumulative metrics continue to grow and can eventually overflow the computer word length. In making metric comparisons to find the surviving paths, only the relative sizes of the metrics are important. Therefore, the overflow problem can be solved by periodically resetting the metrics by subtracting the minimum metric from all the metrics at that stage. In terms of implementation logic, it is easiest to do the resetting at each iteration.
22.11 The BCJR or Forward-Backward Decoding Algorithm

In this section the transmitted trellis sequence will be required to have finite length, start in state 0, and end in state 0 at depth $L_T$. Only an $(N, 1)$ convolutional code will be considered. That is, one information bit, $x(n)$, is input to the encoder and $N$ encoded bits, $y(n) = [y_1(n), \ldots, y_N(n)]$ are output at each iteration. It will be assumed that the code bits are transmitted with biphase modulation over a white Gaussian noise channel as described in the example on page 109 and the received signal is matched filtered resulting in received branch vectors $r(n) = [r_1(n), \ldots, r_N(n)]$. For simplicity, it will be assume that a binary 0 is transmitted as +1 and a binary 1 as −1. The entire received sequence will be denoted by $r = [r(0), \ldots, r(L_T - 1)]$. The notation for a segment of this vector extending from branch $p$ to branch $q$ that will be used is $r_p^q = [r(p), \ldots, r(q)]$.

The rule that minimizes the error probability in estimating the value of an input data bit $x(n)$ given the entire received sequence is to select the value that maximizes the a posteriori probability $P(x(n) = i | r)$ for $i = 0$ or 1. It is known as the maximum a posteriori or MAP rule. An equivalent rule is to compute the log likelihood ratio (LLR)

$$\Lambda_n(r) = \log \frac{P(x(n) = 1 | r)}{P(x(n) = 0 | r)} \quad (22.107)$$

and decide

$$\hat{x}(n) = \begin{cases} 
1 & \text{if } \Lambda_n(r) \geq 0 \\
0 & \text{if } \Lambda_n(r) < 0 
\end{cases} \quad (22.108)$$
The LLR’s are said to provide soft information about the bits. They give a measure of the reliability of each received symbol. The LLR’s quantized to 0 or 1 are said to be hard bit decisions and do not give information about the reliability of the decisions. The MAP rule minimizes the error probability for each individual bit, but it does not guarantee that the entire decoded word corresponds to the most likely entire trellis sequence. The Viterbi algorithm guarantees the decoded sequence corresponds to the most likely trellis sequence but does not minimize the bit error probability. Both approaches give good results in practice.

The conditional bit probabilities can be found from the LLR’s. From (22.107) it follows that

\[ P(x(n) = 1|\mathbf{r}) = P(x(n) = 0|\mathbf{r})e^{\Lambda_n(\mathbf{r})} \]  

Also \( P(x(n) = 0|\mathbf{r}) = 1 - P(x(n) = 1|\mathbf{r}) \). Substituting this identity into (22.109) and solving for \( P(x(n) = 1|\mathbf{r}) \) gives

\[ P(x(n) = 1|\mathbf{r}) = \frac{e^{\Lambda_n(\mathbf{r})}}{1 + e^{\Lambda_n(\mathbf{r})}} \]  

and

\[ P(x(n) = 0|\mathbf{r}) = 1 - P(x(n) = 1|\mathbf{r}) = \frac{1}{1 + e^{\Lambda_n(\mathbf{r})}} \]  

The Bahl, Cocke, Jelinek, Raviv (BCJR) algorithm [2] is a method for computing the MAP estimates of each transmitted data bit of a trellis code based on the entire received symbol sequence. As you will see, it requires a forward iteration from the beginning of the trellis up to the bit for the branch being estimated and a backward iteration from the end of the trellis back to the branch being estimated, so it is also known as the forward-backward algorithm. The algorithm generates soft information about each bit which can be quantized to hard decisions. The BCJR algorithm is extensively being used in current broadband wireless systems like 4G LTE in decoding an extension of convolutional codes know as Turbo Codes. The decoding algorithm for Turbo Codes uses the soft bit information. Turbo codes will not be discussed in this chapter.

### 22.11.1 The Encoder is a Markov Source

The convolutional encoder is a discrete-time finite-state Markov source. The state of the encoder at any time is a vector consisting of the contents of the delay elements. Let the number of encoder delay elements be \( n_d \). Then the number of states is \( M_s = 2^{n_d} \). The state vectors can have decimal values 0 through \( M_s - 1 \). Let \( S_n \) be the encoder state at time \( n \). An input bit, \( x(n-1) \), at time \( n - 1 \) cases the encoder to transition from state \( S_{n-1} \) to state \( S_n \) at time \( n \) and generates an output vector \( y(n-1) \). This is represented by a branch in the trellis diagram.

Let the state transition probabilities for the encoder be

\[ p_n(l|l') = P(S_n = l|S_{n-1} = l') \quad \text{for} \quad 0 \leq l, l' \leq M_s - 1 \]  

For a convolutional code with a single bit input at each time, only two branches leave each node in the trellis. The transition probabilities are 0.5 if states \( l \) and \( l' \) are connected and 0 otherwise, assuming the inputs are equally likely to be 0 or 1.
The probabilities of the encoder outputs for each branch are
\[
q_n(y(n)|l', l) = P(y(n)|S_n = l', S_{n+1} = l) \quad \text{for} \quad 0 \leq l, l' \leq M_s - 1
\] (22.113)
and since the biphase output \(\tilde{y}(n)\) is uniquely determined by \(y(n)\)
\[
q_n(\tilde{y}(n)|l', l) = P(y(n)|S_n = l', S_{n+1} = l) \quad \text{for} \quad 0 \leq l, l' \leq M_s - 1
\] (22.114)

**EXAMPLE 22.21** Transition and Output Probabilities for the Ungerboeck 4-State Code

Consider the trellis shown for the Ungerboeck 4-state systematic code shown in Figure 22.6. Let the states be numbered by the decimal values of \((q_1, q_2) = 2q_1 + q_2\). The transition probabilities are
\[
\begin{align*}
p_n(0|0) &= p_n(1|0) = 0.5 \\
p_n(2|1) &= p_n(3|1) = 0.5 \\
p_n(0|2) &= p_n(1|2) = 0.5 \\
p_n(2|3) &= p_n(3|3) = 0.5
\end{align*}
\]
All other transition probabilities are 0.

The output probabilities are
\[
\begin{align*}
q_n(00|0, 0) &= q_n(01|0, 1) = 1 \\
q_n(10|1, 2) &= q_n(11|1, 3) = 1 \\
q_n(01|2, 0) &= q_n(00|2, 1) = 1 \\
q_n(11|3, 2) &= q_n(10|3, 3) = 1
\end{align*}
\]
All other branch probabilities are 0.

**22.11.2 Formulas for Computing the Probabilities in (22.107)**

Formulas for computing the bit \textit{a posteriori} probabilities (APP’s) in (22.107) will now be developed. Probability mass functions will be used for density functions somewhat carelessly. This can be justified by assuming the continuous amplitude received signals are finely quantized to discrete values with an analog-to-digital converter. Let the set \(B_n^0\) be the set of all branches, that is, all transitions in the trellis from a state \(l'\) at time \(n\) to a state \(l\) at time \(n + 1\), that can be caused by the input \(x(n) = 0\). For example, for the Ungerboeck 4-state code, \(B_n^0 = \{(0, 0), (1, 3), (2, 0), (3, 2)\}\). Then
\[
P(x(n) = 0|r) = \sum_{(l', l) \in B_n^0} P(S_n = l', S_{n+1} = l|r)
\]
\[
= \sum_{(l', l) \in B_n^0} \frac{P(S_n = l', S_{n+1} = l, r)}{P(r)}
\] (22.115)
Let the set $B^1_n$ be the set of all transitions in the trellis from a state $l'$ at time $n$ to a state $l$ at time $n+1$, that can be caused by the input $x(n) = 1$. For example, for the Ungerboeck 4-state code, $B^1_n = \{(0,1), (1,2), (2,1), (3,3)\}$. Then

$$P(x(n) = 1|r) = \sum_{(l',l) \in B^1_n} P(S_n = l', S_{n+1} = l|r)$$

$$= \sum_{(l',l) \in B^1_n} \frac{P(S_n = l', S_{n+1} = l, r)}{P(r)}$$

(22.116)

To simplify the notation, let

$$\sigma_n(l', l) = P(S_n = l', S_{n+1} = l, r) \quad \text{for} \quad 0 \leq l', l \leq M_s - 1$$

(22.117)

Thus, the LLR given by (22.107) can be written as

$$\Lambda_n(r) = \log \frac{\sum_{(l',l) \in B^1_n} \sigma_n(l', l)}{\sum_{(l',l) \in B^0_n} \sigma_n(l', l)}$$

(22.118)

### 22.11.2.1 Computing $\sigma_n(l', l)$

A formula for computing $\sigma_n(l', l)$ will now be derived involving the following three functions:

$$\alpha_n(l) = P(S_n = l, r_0^{n-1})$$

(22.119)

$$\beta_n(l) = P(r_n^{L_T-1}|S_n = l)$$

(22.120)

$$\gamma^i_n(l', l) = P(x(n) = i, S_{n+1} = l, r(n)|S_n = l') \quad \text{for} \quad i = 0, 1$$

(22.121)

Also let

$$\gamma_n(l', l) = \sum_{i=0}^1 \gamma^i_n(l', l)$$

(22.122)

The derivation uses the fact that the entire received sequence, $r$, is the concatenation of $r_0^{n-1}$ leading from the beginning of the trellis to a trellis state $S_n$, $r(n)$ leading from trellis state $S_n$ to state $S_{n+1}$, and $r_{n+1}^{L_T-1}$ leading from trellis state $S_{n+1}$ to the end of the trellis. Therefore

$$\sigma_n(l', l) = P(r_0^{n-1}, r(n), r_{n+1}^{L_T-1}, S_n = l', S_{n+1} = l)$$

$$= P(r_{n+1}^{L_T-1}|r_0^{n-1}, r(n), S_n = l', S_{n+1} = l) \times P(r_0^{n-1}, r(n), S_n = l', S_{n+1} = l)$$

(22.123)

The “future” received sequence $r_{n+1}^{L_T-1}$ if the encoder is in state $S_{n+1}$ at time $n+1$ does not depend on the past outputs or states. It only depends on future inputs and channel noise. Therefore,

$$P(r_{n+1}^{L_T-1}|r_0^{n-1}, r(n), S_n = l', S_{n+1} = l) = P(r_{n+1}^{L_T-1}|S_{n+1} = l) = \beta_{n+1}(l)$$

(22.124)
22.11 The BCJR or Forward-Backward Decoding Algorithm

So

\[
\sigma_n(l', l) = \beta_{n+1}(l) P \left( r^n_{0-1}, r(n), S_n = l', S_{n+1} = l \right)
\]

\[= \beta_{n+1}(l) P \left( r(n), S_{n+1} = l | r^n_{0-1}, S_n = l' \right) \times \]

\[P \left( r^n_{0-1}, S_n = l' \right) \]

\[= \beta_{n+1}(l) P \left( r(n), S_{n+1} = l | r^n_{0-1}, S_n = l' \right) \alpha_n(l') \quad (22.125)\]

Given the encoder is in state \( S_n = l' \) at time \( n \), the branch output, \( r(n) \), and next state, \( S_{n+1} \) are independent of the past received sequence, \( r^n_{0-1} \). Therefore

\[\sigma_n(l', l) = \beta_{n+1}(l) P \left( r(n), S_{n+1} = l | S_n = l' \right) \alpha_n(l') \quad (22.126)\]

Also

\[P \left( r(n), S_{n+1} = l | S_n = l' \right) = \sum_{i=0}^{1} P \left( x_n = i, r(n), S_{n+1} = l | S_n = l' \right) = \sum_{i=0}^{1} \gamma^i_n(l', l) \quad (22.127)\]

Consequently,

\[\sigma_n(l', l) = \alpha_n(l') \beta_{n+1}(l) \sum_{i=0}^{1} \gamma^i_n(l', l) \quad (22.128)\]

Notice that

\[P \left( x_n = 0, r(n), S_{n+1} = l | S_n = l' \right) = \gamma^0_n(l', l) = 0 \quad \text{for} \quad (l', l) \in B^n_1 \quad (22.129)\]

and

\[P \left( x_n = 1, r(n), S_{n+1} = l | S_n = l' \right) = \gamma^n_1(l', l) = 0 \quad \text{for} \quad (l', l) \in B^n_0 \quad (22.130)\]

Using these properties, it follows that the LLR, \( \Lambda_n(r) \), is

\[\Lambda_n(r) = \log \frac{\sum_{(l', l) \in B^n_1} \alpha_n(l') \beta_{n+1}(l) \gamma^1_n(l', l)}{\sum_{(l', l) \in B^n_0} \alpha_n(l') \beta_{n+1}(l) \gamma^0_n(l', l)} \quad (22.131)\]

The Forward Iteration to Find \( \alpha_n(l') \)

\[\alpha_n(l) = P \left( S_n = l, r^n_{0-1} \right) \]

\[= \sum_{l'=0}^{M_{n-1}} P \left( S_{n-1} = l', S_n = l, r^n_{0-1} \right) \]

\[= \sum_{l'=0}^{M_{n-1}} P \left( S_{n-1} = l', S_n = l, r^n_{0-2}, r(n-1) \right) \]

\[= \sum_{l'=0}^{M_{n-1}} P \left( S_{n-1} = l', r^n_{0-2} \right) P \left( S_n = l, r(n-1) | S_{n-1} = l', r^n_{0-2} \right) \quad (22.132)\]
Given the encoder is in state $S_{n-1} = l'$ at time $n-1$, the next state $S_n = l$ and received branch $r(n-1)$ for the transition from $l'$ to $l$ are independent of the received sequence $r_{n-2}^n$ leading up to state $l'$. Therefore $P(S_n = l, r(n-1)|S_{n-1} = l', r_{0}^{n-2}) = P(S_n = l, r(n-1)|S_{n-1} = l')$ and so

$$\alpha_n(l) = \sum_{l'=0}^{M_s-1} \alpha_{n-1}(l')P(S_n = l, r(n-1)|S_{n-1} = l')$$

$$= \sum_{l'=0}^{M_s-1} \alpha_{n-1}(l') \sum_{i=0}^{1} P(x(n-1) = i, S_n = l, r(n-1)|S_{n-1} = l')$$

$$= \sum_{l'=0}^{M_s-1} \alpha_{n-1}(l') \sum_{i=0}^{1} \gamma_{n-1}^i(l', l) = \sum_{l'=0}^{M_s-1} \alpha_{n-1}(l') \gamma_{n-1}(l', l) \quad (22.133)$$

For an $(N,1)$ code, only two trellis branches connect to a state $l$ at time $n$, one from the input $x(n-1) = 0$ and one from $x(n-1) = 1$. Let the previous state at time $n-1$ corresponding to $x(n-1) = 0$ be $l'_0,l$ and the state at time $n-1$ corresponding to $x(n-1) = 1$ be $l'_1,l$. Then $\gamma_{n-1}(l', l)$ is nonzero only for these two previous states and (22.133) simplifies to

$$\alpha_n(l) = \alpha_{n-1}(l'_0,l)\gamma_{n-1}^0(l'_0,l) + \alpha_{n-1}(l'_1,l)\gamma_{n-1}^1(l'_1,l) \quad \text{for} \quad l = 0, \ldots, M_s - 1 \quad (22.134)$$

Assuming the $\gamma$'s are known, the $\alpha$'s can be computed for all the states iteratively starting from time $n = 0$ up to time $L_T - 1$ using (22.134). Since the encoder is started in state 0, the initial values for $\alpha$ are $\alpha_0(0) = 1$ and $\alpha_0(l) = 0$ for $l = 1, \ldots, M_s - 1$. 
The Backward Iteration to Find $\beta_n(l)$

\[
\beta_n(l) = P(r_{n+1}^{L_T-1}|S_n = l) = \left( \sum_{l'=0}^{M_S-1} P(S_{n+1} = l', r_{n+1}^{L_T-1}|S_n = l) \right) \sum_{l'=0}^{M_S-1} P(S_{n+1} = l', S_n = l, r_{n+1}^{L_T-1}) P(s_n = l)
\]

\[
= \sum_{l'=0}^{M_S-1} \frac{P(S_{n+1} = l', S_n = l, r(n), r_{n+1}^{L_T-1})}{P(s_n = l)} = \sum_{l'=0}^{M_S-1} \frac{P(r_{n+1}^{L_T-1}|S_{n+1} = l', S_n = l, r(n)) P(S_{n+1} = l', S_n = l, r(n))}{P(s_n = l)}
\]

\[
= \sum_{l'=0}^{M_S-1} \frac{P(r_{n+1}^{L_T-1}|S_{n+1} = l')} P(S_{n+1} = l', S_n = l, r(n)) P(s_n = l)
\]

\[
= \sum_{l'=0}^{M_S-1} \sum_{l'=0}^{M_S-1} \beta_{n+1}(l') P(S_{n+1} = l', r(n)|S_n = l)
\]

\[
= \sum_{l'=0}^{M_S-1} \sum_{l'=0}^{M_S-1} \beta_{n+1}(l') \sum_{i=0}^{1} P(x(n) = i, S_{n+1} = l', r(n)|S_n = l) = \beta_n(l) = \sum_{l'=0}^{M_S-1} \beta_{n+1}(l') \sum_{i=0}^{1} \gamma_i^n(l, l') \quad \text{for } n = L_T - 1, \ldots, 1, 0 \quad l = 0, \ldots, M_s - 1
\]  (22.135)

For an (N,1) code, only two branches go out of a state $l$ at time $n$, one for $x(n) = 0$ and one for $x(n) = 1$. Let the state at time $n + 1$ caused by $x(n) = 0$ be $l_{0,l}'$ and the state caused by $x(n) = 1$ be $l_{1,l}$. Then all the terms in the sum in (22.135) over $l'$ are 0 except for $l' = l_{0,l}'$ and $l' = l_{1,l}'$. Therefore, (22.135) reduces to

\[
\beta_n(l) = \beta_{n+1}(l_{0,l}') \gamma_i^n(0, l_{0,l}') + \beta_{n+1}(l_{1,l}') \gamma_i^n(1, l_{1,l}') \quad \text{for } n = L_T - 1, \ldots, 1, 0 \quad l = 0, \ldots, M_s - 1
\]  (22.136)

Assuming the $\gamma$'s are known, the $\beta$'s can be computed for all the states iteratively starting from depth $n = L_T - 1$ down to depth 0 using (22.136). Since the encoder is forced to end in state 0, the initial values for $\beta$ are $\beta_{L_T}(0) = 1$ and $\beta_{L_T}(l) = 0$ for $l = 1, \ldots, M_s - 1$.

Computation of $\gamma_i^n(l', l)$

When $(l', l) \notin B_i^n, \gamma_i^n(l', l) = 0$. Using the rules for factoring probabilities gives

\[
\gamma_i^n(l', l) = P(x(n) = i, S_{n+1} = l, r(n)|S_n = l')
\]

\[
= P(r(n)|S_n = l', S_{n+1} = l, x(n) = i) P(x(n) = i|S_n = l', S_{n+1} = l)
\quad \times P(S_{n+1} = l|S_n = l')
\]  (22.137)

The last factor is

\[
P(S_{n+1} = l|S_n = l') = \begin{cases} P(x(n) = i) & \text{for } (l', l) \in B_i^n, i = 0, 1 \\ 0 & \text{otherwise} \end{cases}
\]  (22.138)
In most cases it is assumed that the input bits are equally likely, so that $P(x(n) = i) = 1/2$. The factor, $P(x(n) = i|S_n = l', S_{n+1} = l)$ is 1 for $(l', l) \in B^i_n$ and zero otherwise.

Now $(l', l)$ will be required to be in $B^i_n$. The conditioning event \{\(S_n = l', S_{n+1} = l, x(n) = i\)\} in the first factor specifies an encoder output for that trellis branch. Let the biphase output vector for that branch be $\tilde{y}(n; l', l)$ and the $k$-th component be $y_k(n; l', l)$. Then

$$P(r(n)|S_n = l', S_{n+1} = l, x(n) = i) = \frac{1}{(2\pi\sigma^2)^{N/2}} \exp \left\{ -\frac{1}{2\sigma^2} ||r(n) - \tilde{y}(n; l', l)||^2 \right\}$$

$$= \frac{1}{(2\pi\sigma^2)^{N/2}} \exp \left\{ -\frac{1}{2\sigma^2} \sum_{k=1}^{N} [r_k(n) - y_k(n; l', l)]^2 \right\} \quad (22.139)$$

Since $\gamma^i_n(l', l)$ is in both the numerator and denominator of the likelihood function, it can be scaled by any convenient number. Combining all the properties stated above, $\gamma^i_n(l', l)$ can be computed as

$$\gamma^i_n(l', l) = \begin{cases} \hspace{1cm} P(x(n) = i) \exp \left\{ -\frac{1}{2\sigma^2} \sum_{k=1}^{N} [r_k(n) - y_k(n; l', l)]^2 \right\} & \text{for } (l', l) \in B^i_n \quad (22.140) \\
0 & \text{otherwise} \end{cases}$$

When the inputs are equally likely, which is the usual assumption, the factor $P(x(n) = i)$ can also be removed since it is always 1/2.

### 22.11.3 Normalization for Computational Stability

Propagating $\alpha_n(l)$ forward and $\beta_n(l)$ backward using the formulas above involves computing products and sums of small numbers. These numbers get significantly smaller as the iterations progress and large numerical errors can occur because of the loss of numerical precision even with floating-point arithmetic. Therefore, normalization at each iteration is used in practical implementations of the BCJR algorithm. One normalization is to replace $\alpha_n(l)$ and $\beta_n(l)$ by scaled versions $\alpha'_n(l)$ and $\beta'_n(l)$ so that

$$\sum_{l=0}^{M_S-1} \alpha'_n(l) = 1 \quad \text{and} \quad \sum_{l=0}^{M_S-1} \beta'_n(l) = 1 \quad (22.141)$$

The formula for propagating $\alpha'_n(l)$ becomes

$$\alpha'_n(l) = \frac{\sum_{l'=0}^{M_S-1} \alpha'_{n-1}(l') \gamma_{n-1}(l', l)}{\sum_{l''=0}^{M_S-1} \sum_{l'=0}^{M_S-1} \alpha'_{n-1}(l') \gamma_{n-1}(l', l'')} = A_{n-1} \sum_{l'=0}^{M_S-1} \alpha'_{n-1}(l') \gamma_{n-1}(l', l) \quad (22.142)$$
22.11 The BCJR or Forward-Backward Decoding Algorithm

where

\[ A_{n-1} = \left( \sum_{l'=0}^{M_{S}-1} \sum_{l''=0}^{M_{S}-1} \alpha'_{n-1}(l') \gamma_{n-1}(l', l'') \right)^{-1} \]  \hspace{1cm} (22.143)

The numerator in (22.142) can be computed first for \( l = 0, \ldots, M_{S} - 1 \) and then summed over \( l \) to get the normalizing denominator.

The formula for propagation \( \beta'_n(l) \) becomes

\[ \beta'_n(l) = \frac{\sum_{l'=0}^{M_{S}-1} \beta'_{n+1}(l') \gamma_n(l, l')}{\sum_{l'=0}^{M_{S}-1} \sum_{l''=0}^{M_{S}-1} \beta'_{n+1}(l') \gamma_n(l'', l')} = B_n \sum_{l'=0}^{M_{S}-1} \beta'_{n+1}(l') \gamma_n(l, l') \]  \hspace{1cm} (22.144)

where

\[ B_n = \left( \sum_{l'=0}^{M_{S}-1} \sum_{l''=0}^{M_{S}-1} \beta'_{n+1}(l') \gamma_n(l'', l') \right)^{-1} \]  \hspace{1cm} (22.145)

Again, the numerator for \( \beta'_n(l) \) can be computed for each \( l \) and then summed to form the normalizing denominator. The unnormalized and normalized quantities are related by

\[ \alpha'_n(l) = \left[ \prod_{i=0}^{n-1} A_i \right] \alpha_n(l) \quad \text{and} \quad \beta'_n(l) = \left[ \prod_{i=l}^{L_{T}-1} B_n \right] \beta_n(l) \]  \hspace{1cm} (22.146)

The same scale factors in the numerator and denominator cancel when the likelihood ratio is formed leaving the LLR the same for the normalized and unnormalized versions when the arithmetic is perfect. However, the normalized results are more accurate when the computations are performed using an actual finite word length computer. Of course, the normalization increases the required amount of computation.

22.11.3.1 A Decomposition of \( \Lambda_n(r) \) for Systematic Codes with BPSK on an AWGN Channel to Use with Turbo Codes

Now suppose the code is a systematic code with binary input sequence \( x(n) \) and the first bit in the encoder output vectors is \( y_1(n; \ell', \ell) = x(n) \). A binary variable \( z \) that can have value 0 or 1 is BPSK modulated to \( \tilde{z} = (-1)^z \) and can have analog values +1 or −1. A tilde above a binary variable will be used to denote its BPSK value in the future. The BPSK encoder outputs are transmitted over an additive, white, Gaussian noise (AWGN) channel and the receiver observes the outputs of a matched filter as explained in Example 22.19 on page 109.

Using the systematic property, \( \gamma_n(l, l') \) can be expanded as

\[ \gamma_n^i(l', l) = P(x(n) = i) \, e^{-\frac{1}{2\sigma^2}[r_1(n)-\tilde{z}]^2} \, e^{-\frac{1}{2\sigma^2} \sum_{k=2}^{N} [r_k(n) - \tilde{s}_k(n,l',l)]^2} \, ; \quad (l', l) \in B_n^i, \, i = 0 \, \text{or} \, 1 \]  \hspace{1cm} (22.147)
Substituting (22.147) into the bit LLR equation gives

\[ \Lambda_n(r) = \log \left\{ \frac{P(x(n) = 1)}{P(x(n) = 0)} \times \frac{e^{-\frac{1}{\sigma^2} (r_1(n)+1)^2}}{e^{-\frac{1}{\sigma^2} (r_1(n)-1)^2}} \right\} \]

\[ = \log \left[ \frac{P(x(n) = 1)}{P(x(n) = 0)} \right] - \frac{2}{\sigma^2} r_1(n) \]

\[ = \log \frac{\sum_{(l',l) \in B^n_1} \alpha_n(l') \beta_{n+1}(l)e^{-\frac{1}{2\sigma^2} \sum_{k=2}^{N} r_k(n)R_k(n',l')}}{\sum_{(l',l) \in B^n_0} \alpha_n(l') \beta_{n+1}(l)e^{-\frac{1}{2\sigma^2} \sum_{k=2}^{N} r_k(n)R_k(n',l')}} \tag{22.148} \]

where the facts that \( \hat{y}_k^2(n,l') = 1 \) and the \( r_k(n) \) terms are the same in the numerators and denominators were used to cancel extra factors. The first term in the sum on the right is the log likelihood ratio for the input bit \textit{a priori} probabilities. Let this be denoted by

\[ \lambda_p(x(n)) = \log \frac{P(x(n) = 1)}{P(x(n) = 0)} \tag{22.149} \]

It is a measure of the information known to the decoder about the input data bits prior to any received signal observations. The second term is

\[ \lambda_e(r_1(n)) = \log \frac{p(r_1(n)|x(n) = 1)}{p(r_1(n)|x(n) = 0)} = -\frac{2}{\sigma^2} r_1(n) \tag{22.150} \]

It provides information learned from the receiver systematic bit input \( r_1(n) \). (Note that the BPSK mapping used in this chapter for a bit \( x \) is \( y = (-1)^x \) so a binary 0 maps to an analog +1 and a binary 1 maps to an analog −1. This is the opposite of many other articles and accounts for the minus sign.) Let the third term be denoted by

\[ \lambda_e(r(n)) = \log \frac{\sum_{(l',l) \in B^n_1} \alpha_n(l') \beta_{n+1}(l)e^{-\frac{1}{2\sigma^2} \sum_{k=2}^{N} r_k(n)R_k(n',l')}}{\sum_{(l',l) \in B^n_0} \alpha_n(l') \beta_{n+1}(l)e^{-\frac{1}{2\sigma^2} \sum_{k=2}^{N} r_k(n)R_k(n',l')}} \tag{22.151} \]

This term provides information obtained from the code structure and received check symbols and is called the \textit{extrinsic} information.

### 22.11.3.2 Memory and Computation Requirements

Memory required by the BCJR algorithm includes:

1. Memory for \( \gamma_n^{\dagger}(l',l) \). The trellis has \( 2 \times M_s \times L_T \) branches neglecting the differences at the start and end of the trellis and \( \gamma_n^{\dagger}(l',l) \) is required for each branch.
22.11 The BCJR or Forward-Backward Decoding Algorithm

2. $M_S \times L_T$ values of $\alpha_n(l)$.

3. $M_S \times L_T$ values of $\beta_n(l)$.

4. Tables defining the trellis including the pair of next states and pair of previous states for each current state, and the output bits for each branch. Some of these may be recomputed each iteration rather than stored.

The computation requirements include:

1. $2 \times M_s \times L_T$ products and $M_s \times L_T$ additions to propagate $\alpha_n(l)$ forward across the entire trellis and the same to propagate $\beta_n(l)$ backward.

2. Computing the numerator of the likelihood ratio for one $n$ requires $2 \times M_s$ products to compute $\alpha_n(l')\beta_{n+1}(l)\gamma_n^0(l',l)$ and $(M_s - 1)$ additions to sum the products. Computing the denominator requires the same amount of computation. This has to be performed for each of the $L_T$ transmitted bits.

3. Computing the LLR’s requires $L_T$ divisions and logarithms.

22.11.4 Summary of the Steps for Applying the BCJR Algorithm

The steps required to compute the bit LLR by the BCJR algorithm are outlined in this section.

1. **Compute the branch metrics.**
   
   For all branches in the trellis, that is, for $n = 0, \ldots, L_T - 1$, and $(l', l) \in B^i_n$ with $i = 0$ and 1, compute and store
   
   $$
   \gamma_n^i(l', l) = P(x(n) = i, S_{n+1} = l, r(n)|S_n = l')
   $$
   
   $$
   = P(x(n) = i) \exp \left\{-\frac{1}{2\sigma^2} \sum_{k=1}^{N} [r_k(n) - \tilde{y}_k(n; l', l)]^2 \right\} \quad (22.152)
   $$
   
   where $r_k(n)$ is the $k$-th component of the received branch sequence and $\tilde{y}_k(n; l', l)$ is the $k$-th component of the transmitted biphase sequence for branch $(l', l)$.

2. **Perform the forward recursion.**

   (a) Initialize $\alpha_0'(l)$

   $\alpha_0'(0) = \alpha_0(0) = 1$ and $\alpha_0'(l) = \alpha_0(1) = 0$ for $l = 1, \ldots, M_s - 1$

   (b) For $n = 1, \ldots, L_T - 1$ and $l = 0, \ldots, M_s - 1$, calculate and store $\alpha_n'(l)$. First compute

   $$
   \alpha_n(l) = \alpha_n'(l') \gamma_{n-1}^0(l_{0, l}, l') + \alpha_n'(l') \gamma_{n-1}^1(l_{1, l}, l) \text{ for } l = 0, \ldots, M_s - 1 \quad (22.153)
   $$
and then normalize to
\[ \alpha_n'(l) = \frac{\alpha_n(l)}{\sum_{m=0}^{M_s-1} \alpha_n(m)} \quad \text{for } l = 0, \ldots, M_s - 1 \] (22.154)

where for state \( S_n = l \) at time \( n \), \( S_{n-1} = l'_{0,l} \) is the state at time \( n - 1 \) connected to \( S_n = l \) by the input \( x(n-1) = 0 \), and \( S_{n-1} = l'_{1,l} \) is the state at time \( n - 1 \) connected to \( S_n = l \) by \( x(n-1) = 1 \).

3. Perform the backward recursion.

(a) Initialize \( \beta_{LT}'(l) \)
\[ \beta_{LT}'(0) = \beta_{LT}(0) = 1 \text{ and } \beta_{LT}'(l) = \beta_{LT}(l) = 0 \text{ for } l = 1, \ldots, M_s - 1 \]

(b) For \( n = L_T - 1, \ldots, 0 \) and \( l = 0, \ldots, M_s - 1 \), calculate and store \( \beta_n(l) \). First compute
\[ \beta_n(l) = \beta_{n+1}(l'_{0,l}) \gamma_n^0(l,l'_{0,l}) + \beta_{n+1}(l'_{1,l}) \gamma_n^1(l,l'_{1,l}) \quad \text{for } l = 0, \ldots, M_s - 1 \] (22.155)

and then normalize to
\[ \beta_n'(l) = \frac{\beta_n(l)}{\sum_{m=0}^{M_s-1} \beta_n(m)} \quad \text{for } l = 0, \ldots, M_s - 1 \] (22.156)

where for state \( S_n = l \) at time \( n \), \( S_{n+1} = l'_{0,l} \) is the state at time \( n + 1 \) connected to \( S_n = l \) by the input \( x(n) = 0 \), and \( S_{n+1} = l'_{1,l} \) is the state at time \( n + 1 \) connected to \( S_n = l \) by \( x(n) = 1 \).

4. Compute the Bit Log Likelihood Ratios

For \( n = 0, \ldots, L_T - 1 \) compute
\[ \Lambda_n(r) = \log \frac{\sum_{(l',l)\in B_n^1} \alpha_n'(l') \beta_{n+1}'(l) \gamma_n^1(l',l)}{\sum_{(l',l)\in B_n^0} \alpha_n'(l') \beta_{n+1}'(l) \gamma_n^0(l',l)} \] (22.157)

where \( S_n = l' \) and \( S_{n+1} = l \).

22.12 The Max-Log-MAP Algorithm

The BCJR decoding algorithm requires a large amount of memory and a large number of computations involving sums of products, exponentiations to compute \( \gamma_n(l',l) \), and quotients.
22.12 The Max-Log-MAP Algorithm

The algorithm can be approximated and computational requirements significantly reduced by using the logarithms of $\alpha_n(l)$, $\beta_n(l)$, and $\gamma^i_n(l',l)$. Let these logarithms be denoted by

$$a_n(l) = \log \alpha_n(l) \quad \text{or} \quad \alpha_n(l) = e^{a_n(l)} \quad (22.158)$$
$$b_n(l) = \log \beta_n(l) \quad \text{or} \quad \beta_n(l) = e^{b_n(l)} \quad (22.159)$$
$$g^i_n(l',l) = \log \gamma^i_n(l',l) \quad \text{or} \quad \gamma^i_n(l',l) = e^{g^i_n(l',l)} \quad (22.160)$$

Substituting the logs into (22.133) gives the following formula for propagating $a_{n-1}(l) = \log \alpha_{n-1}(l)$:

$$a_n(l) = \log \sum_{l'=0}^{M_s-1} \sum_{i=0}^1 \alpha_{n-1}(l') \gamma^i_{n-1}(l',l) = \log \sum_{l'=0}^{M_s-1} \sum_{i=0}^1 e^{a_{n-1}(l')} + g^i_{n-1}(l',l) \quad (22.161)$$

with the initial conditions

$$a_0(0) = 0 \quad \text{and} \quad a_0(l) = -\infty \quad \text{for} \quad l = 1, \ldots, M_s - 1 \quad (22.162)$$

The formula for propagating $a_n(l)$ can be approximated and simplified by using the following approximation:

$$\log (e^{\delta_1} + e^{\delta_2} + \cdots + e^{\delta_L}) \approx \max_{i=1, \ldots, L} \delta_i \quad (22.163)$$

Using this approximation gives

$$a_n(l) \approx \max_{l'=0, \ldots, M_s-1} \left\{ a_{n-1}(l') + g^i_{n-1}(l',l) \right\} \quad \text{for} \quad l = 0, \ldots, M_s - 1 \quad (22.164)$$

This formula has numerical complexity similar to computing the cumulative metrics for the surviving paths to states in the Viterbi algorithm with $a_{n-1}(l')$ analogous to the cumulative metric for the surviving path to state $l$ at time $n-1$ and $g_{n-1}(l',l)$ analogous to the branch metric for the branch connecting states $l'$ and $l$.

Remember that for an $(N,1)$ code, only two trellis branches connect to a state $l$ at time $n$, one from the input $x(n-1) = 0$ and one from $x(n-1) = 1$. Let the previous state at time $n-1$ corresponding to $x(n-1) = 0$ be $l'_{0,i}$ and the state at time $n-1$ corresponding to $x(n-1) = 1$ be $l'_{1,i}$. Then $\gamma_{n-1}(l',l)$ is nonzero only for these two previous states and $g^i_{n-1}(l',l) = -\infty$ except for these two previous states. Therefore, (22.164) simplifies to

$$a_n(l) \approx \max_{i=0,1} \left\{ a_{n-1}(l'_{0,i}) + g^0_{n-1}(l'_{0,i},l), a_{n-1}(l'_{1,i}) + g^1_{n-1}(l'_{1,i},l) \right\} \quad \text{for} \quad n = 0, \ldots, L_T - 1 \quad l = 0, \ldots, M_s - 1 \quad (22.165)$$

Similarly, substituting the logs into (22.135) gives the following formula for propagating $b_n(l) = \log \beta_n(l)$:

$$b_n(l) = \log \sum_{l'=0}^{M_s-1} \sum_{i=0}^1 \beta_{n+1}(l') \gamma^i_n(l,l') = \log \sum_{l'=0}^{M_s-1} \sum_{i=0}^1 e^{b_{n+1}(l')} + g^i_n(l,l') \quad (22.166)$$
with the initial conditions
\[ b_{LT}(0) = 0 \text{ and } b_{LT}(l) = -\infty \text{ for } l = 1, \ldots, M_s - 1 \] (22.167)

Using the max-log approximation gives
\[ b_n(l) \approx \max_{l' = 0, \ldots, M_s - 1} \{ b_{n+1}(l') + g_n^i(l', l) \} \text{ for } l = 0, \ldots, M_s - 1 \] (22.168)

For an \((N,1)\) code, only two branches go out of a state \(l\) at time \(n\), one for \(x(n) = 0\) and one for \(x(n) = 1\). Let the state at time \(n + 1\) caused by \(x(n) = 0\) be \(l'_0, l\) and the state caused by \(x(n) = 1\) be \(l'_1, l\). Then all the terms in the maximum over \(l'\) in (22.168) are \(-\infty\) except for \(l' = l'_0, l\) and \(l' = l'_1, l\). Therefore, (22.168) reduces to
\[ b_n(l) \approx \max \{ b_{n+1}(l'_0, l) + g_n^0(l'_0, l), b_{n+1}(l'_1, l) + g_n^1(l'_1, l) \} \text{ for } n = L_T - 1, \ldots, 0 \] for \(l = 0, \ldots, M_s - 1\) (22.169)

Substituting the logarithms into the log likelihood ratio (22.131) gives
\[ \Lambda_n(r) = \log \frac{\sum_{(l', l) \in B_1^n} e^{a_n(l') + b_{n+1}(l') + g_n^i(l', l)}}{\sum_{(l', l) \in B_0^n} e^{a_n(l') + b_{n+1}(l') + g_n^0(l', l)}} \] (22.170)

which can be approximated by
\[ \Lambda_n(r) \approx \max_{(l', l) \in B_1^n} \{ a_n(l') + b_{n+1}(l') + g_n^1(l', l) \} - \max_{(l', l) \in B_0^n} \{ a_n(l') + b_{n+1}(l') + g_n^0(l', l) \} \] (22.171)

The max-log-MAP algorithm is suboptimal but performs close to the exact MAP algorithm at medium to high SNR’s.

### 22.13 The Log-MAP Algorithm

The approximation to the log of a sum of exponentials used in the Max-Log-MAP algorithm can be replaced by an exact computation of this quantity by using a recursion based on an exact formula for the log of the sum of two exponentials known as the Jacobian logarithm [7]. The resulting algorithm is called the log-MAP algorithm and it has exactly the same performance as the BCJR MAP algorithm. The formula is
\[ \log (e^{\delta_1} + e^{\delta_2}) = \max(\delta_1, \delta_2) + \log (1 + e^{-|\delta_2 - \delta_1|}) = \max(\delta_1, \delta_2) + f_c(|\delta_2 - \delta_1|) \] (22.172)
where
\[ f_c(|\delta_2 - \delta_1|) = \log (1 + e^{-|\delta_2 - \delta_1|}) \] (22.173)
is a correction to the max-log approximation. It has been suggested that \( f_c(x) \) for \( x = |\delta_2 - \delta_1| \) can be closely approximated by a look-up table with eight values of \( x \) between 0 and 5 [7]. Suppose it is desired to compute

\[
q_L = \log \sum_{k=1}^{L} e^{\delta_k}
\]  

(22.174)

Let the log of a partial sum for \( i \leq L \) be

\[
q_i = \log \sum_{k=1}^{i} e^{\delta_k} \quad \text{or} \quad \sum_{k=1}^{i} e^{\delta_k} = e^{q_i}
\]  

(22.175)

Then

\[
q_i = \log \left( e^{\delta_i} + \sum_{k=1}^{i-1} e^{\delta_k} \right) = \log \left( e^{\delta_i} + e^{q_{i-1}} \right)
\]

\[
= \max(\delta_i, q_{i-1}) + f_c(|\delta_i - q_{i-1}|)
\]  

(22.176)

Notice that \( q_1 = \log e^{\delta_1} = \delta_1 \). Thus \( \delta_i \) can be computed recursively for \( i = 2, \ldots, L \) using (22.176).

The numerator and denominator in the LLR in the form of (22.170) can be computed exactly using the recursion of (22.176) with \( q_n = a_n(l') + b_{n+1}(l) + g_n^l(l', l) \). The computational complexity is somewhat more than for the max-log-MAP algorithm because \( f_c(\cdot) \) must be computed or read from a table.

### 22.14 Estimating Bit Error Rates

In the experiments for this chapter you will be asked to measure the bit error rates at the outputs of the decoders as a function of the signal-to-noise ratio (SNR) of the inputs to the decoders and plot the results. These curves are often referred to as *waterfall* curves. Suppose the actual input to an encoder is the binary stream \( x(n) \) and the output of the decoder is \( \hat{x}(n) \). Then the error sequence is the binary sequence \( e(n) = \hat{x}(n) - x(n) \) mod 2 and \( \hat{x}(n) = x(n) + e(n) \) mod 2. The sequence \( e(n) \) is 1 in every position where an error occurs and is 0 in the positions that are correct. Suppose the encoder input is known and the decoder output is observed for times \( 0, \ldots, L - 1 \) and compared with the known transmitted sequence to form the error sequence \( e(n) \). Then an estimate of the bit error probability is

\[
\hat{p} = \frac{1}{L} \sum_{k=0}^{L-1} e(n) = \frac{\text{number of measured bit errors}}{\text{number of bits observed}}
\]  

(22.177)

where now the sum uses real arithmetic. As and approximation it will be assumed that \( e(n) \) is a sequence of independent binary random variables with \( P\{e(n) = 1\} = p \) and \( P\{e(n) = 0\} = 1 - p \). This is an approximation because the errors in the decoded data
sequence may occur in bursts and not be independent in practice. The sum in (22.177) has a binomial distribution with mean \( Lp \) and variance \( Lp(1 - p) \). Therefore

\[
E\{\hat{p}\} = p \quad \text{and} \quad \text{var} \ \hat{p} = p(1 - p)/L
\]

The question now is how large to make \( L \) to get a reasonably accurate estimate of the error rate. As \( L \) becomes large, the variance of the estimate converges to zero and the estimate converges to the true value. Suppose we desire to have the standard deviation to be less than the mean, \( p \), by a factor of \( c \), that is, \( \sqrt{p(1 - p)/L} < p/c \) which means \( L \) must satisfy

\[
L > c^2 \frac{1 - p}{p}
\]

or for small error probability \( p \)

\[
L > \frac{c^2}{p}
\]

As an example suppose we wish to estimate the error probability with \( p = 10^{-2} \) and \( c = 10 \). This requires an observation length of \( L > 10,000 \) bits.

An approach to estimating the bit error rate is to continue to transmit bits, count the bit errors, and compute the error rate using (22.177) until it converges to a value with some desired precision.

### 22.15 Experiments for Convolutional Codes

Since no real-time signals will be generated and transmitted in these experiments, use any programming tools you like to perform them. You can use any C compiler on a PC or in Code Composer Studio with the DSK. MATLAB is probably the most convenient programming system but will run slower than a compiled C program. Use a 23-stage maximal-length feedback shift register with connection polynomial \( 1 + D^{18} + D^{23} \) or \( 1 + D^{5} + D^{23} \) as discussed in Chapter 9 to simulate random binary input data for all these experiments.

#### 22.15.1 Exploring Uncoded BPSK Transmission

First, you will explore uncoded BPSK transmission to compare it with coded transmission later. Perform the following tasks:

1. Plot the theoretical bit error probability, \( p_b \), given by (22.95) as a function of \( E_s/N_0 \).

   In this case there are no redundant code bits and \( E_s \) is the transmitted energy per data bit which will be denoted by \( E_b \), so \( E_s = E_b \). Plot \( p_b \) on a vertical log scale vs. \( \gamma = 10\log_{10}(E_b/N_0) \) dB on a horizontal linear scale.

2. Write a program to perform an experimental bit error rate test. Use one of the 23-stage maximal length feedback shift registers to simulate the random binary input data.

   (a) Show that the noise variance, \( \sigma^2 \), in terms of \( \gamma \) should be \( \sigma^2 = 0.5 \times 10^{-\gamma/10} \).

   (b) Convert the shift register logical outputs to BPSK levels and add simulated Gaussian noise samples with variance \( \sigma^2 \) to simulate the received symbols. Transmit a block of bits.
(c) Estimate the transmitted bits by quantizing a positive received value to a logical 0 and a negative one to a logical 1.

(d) Count the number of errors between the transmitted and received estimated bits and divide the number of errors by the block length to estimate the bit error rate. Run your simulation for a set of $E_s/N_0$ sufficient to obtain a reasonable waterfall curve. Be sure to transmit enough bits to get a stable estimate of the bit error rate at each value of $E_s/N_0$. Plot your estimated bit error rates vs. $\gamma$.

(e) Compare your experimentally measured bit error rate curve with the theoretical one by plotting them on the same graph.

22.15.2 Implementing the LTE Encoder and a Viterbi Decoder for Biphase Transmission Over an Additive White Gaussian Noise Channel

In this experiment you will implement the LTE encoder shown in Figure 22.7, transmit its output over an additive white Gaussian noise channel using biphase modulation, and perform error rate tests for a Viterbi decoder. Perform the following tasks for this experiment:

1. Write a program to implement the LTE encoder shown in Figure 22.7 on page 89. Use one of the 23-stage maximal length feedback shift registers to simulate the random binary input data. You can use the LTE encoder equations directly. As an efficient alternative, set up a $2 \times 8$ array, $\text{NextState}[2][8]$, that gives the next state resulting from the current input and state. Also set up a $2 \times 8$ array, $\text{CheckBit}[2][8]$, that gives the check bit $y_1(n)$ for the current input and state. Let the state be the decimal number $k(n) = q_1(n) \times 4 + q_2(n) \times 2 + q_3(n)$ which can have the values 0 through 7. Then $\text{NextState}[i][j]$ is the next state for input $i = x(n)$ and current state $j = k(n)$. The value of $\text{CheckBit}[i][j]$ is the check bit, $y_1(n)$, for input $i = x(n)$ and current state $j = k(n)$.

2. Test that your encoder is working properly by storing a reasonable length record of encoder output bits and verifying that the check equation (22.61) is satisfied.

3. Save the present and last 32 input data bits in a 33 word circular buffer. This will be used in making error rate tests at the decoder output.

4. Convert the logical encoder branch output bits to biphase symbols with values 1 or -1 if you have not already done this.

5. Generate zero mean Gaussian noise samples with variance $\sigma^2$ and add them to the transmitted biphase symbols. Now each input data bit $x(n)$ is encoded into the two code bits $y_1(n)$ and $y_2(n)$. Therefore, the transmitted energy per input data bit is $E_b = 2E_s$ and the matched filter output noise variance for each code bit is $\sigma^2 = 2E_s/N_0 = E_b/N_0$. In dB, $\gamma = 10\log_{10}(E_b/N_0)$, so $\sigma^2 = 10^{-\gamma/10}$. Comparing the
performance of different coding schemes on the basis of $E_b/N_0$ rather than $E_s/N_0$ is a fair way of comparing them.

6. Assume that the encoder starts in state 0 but that the input sequence is arbitrarily long. Implement a Viterbi decoder that has a decoding delay of $I$ branches. Include provisions for $I$ up to 32 trellis sections in the past. That is, the decoder should find the current state with the best metric and trace back the surviving path to the state $I$ trellis stages back. The surviving trellis states and associated data should be stored in a length 33 word circular buffer. You will need to store the following:

(a) The cumulative metrics for the surviving paths to each of the current eight states in the current trellis section. Actually, you will need two eight element arrays for this. One array should hold the cumulative metrics for the survivors to the states one time back. The second array should be used to store the new computed cumulative metrics for each of the eight states at the current time. In the next iteration the two arrays should be swapped. What was the current array becomes the array one in the past and the new cumulative survivor metrics can be written over the ones in the other array which are no longer needed. This way, the previous metrics are not destroyed until they completely used to form the new metrics.

(b) For each current state, the value of the previous state connected to it by the surviving path branch to the current state. These will be called pointers to the best previous states. They should be stored for 32 trellis sections in the past.

(c) The input bits associated with the surviving branches between states in adjacent trellis sections from the present section back 32 sections

7. Perform error rate tests vs. $E_b/N_0$ from low to reasonably high values to get waterfall curves. Transmit enough input data bits at each $E_b/N_0$ to get an accurate estimate of the error rate in the decoded data bit stream. Plot error rate on a log scale and $\gamma = E_b/N_0$ in dB on a linear scale. To see the effect of decoding delay, run tests for $I = 2, 8, 16, \text{ and } 32$.

Here is more detail on how to perform an error rate test. Suppose $N_1$ data bits are transmitted for the test and a decoding delay $I$ is used. Then at each time $n$ greater than or equal to $I$, find the state with the best cumulative metric, trace the surviving path back from it $I$ stages, and compare the data bit on the path branch $I$ stages back and the corresponding input data bit. Increment a bit error counter if the two differ. After the decoder reaches trellis depth $N_1$, the number of decoded bits is $N_1 - I$, so estimate the bit error rate as $P_e = (\text{bit error counter final value})/(N_1 - I)$.

8. Plot the theoretical bit error rate curve and your experimentally measured ones on the same graph to compare them.

9. Choose an $E_b/N_0$ where few or no decoding errors occur and run the encoder and decoder to time $n = 200$. Superimpose plots of the surviving paths to each of the
states at time 200 starting at time $200 - 32 = 168$. Let the vertical axis be the integer state values $0, 1, \ldots, 7$ and the horizontal axis be times $168, \ldots, 200$. Connect the states in each individual surviving path by straight lines. You should find that the paths all merge into a common tail some time back.

10. Repeat the previous item for a low $E_b/N_0$ where decoding errors occur even with decoding delay $I = 32$. Now the paths should not merge into a common tail by 32 steps back.

### 22.15.3 Decoding the LTE Code with the BCJR Algorithm for Biphase Transmission Over an Additive White Gaussian Noise Channel

Again, use a 23-stage maximal length shift register generator to simulate the input binary random data. Do the following for this exercise:

1. Encode a block of 1000 data bits. Save the block to use for error checking at the decoder output.

2. After the block of data bits has been encoded, move the switch at the encoder input to the position shown in Figure 22.7 and clock the encoder three times to send the final encoder state to 0.

3. Convert the code bits to biphase symbols and add Gaussian noise with variance $\sigma^2 = 10^{-\gamma/10}$ where $\gamma = 10 \log_{10}(E_b/N_0)$.

4. Decode the received noisy block using the exact BCJR algorithm summarized in Section 22.11.4. Quantize the LLR’s to the appropriate logical values, 0 or 1. To help implement the forward and backward recursions, you may want to create a table that shows the two branches connected to each current state from depth one in the past, and a table that shows the two branches connected to each current state from depth one in the future. You may also want to perform the normalization discussed in Section 22.11.3 if you find that numerical accuracy is a problem.

5. Count the data bit errors in the decoded block. Repeat sending blocks until a stable estimate of the bit error rate with a specific noise variance $\sigma^2$ is obtained.

6. Repeat the error rate test for values of $E_b/N_0$ from low to high and plot the error rate vs. $\gamma$ curve.

7. Compare your error rate curve with the one for the Viterbi decoder with decoding delay $I = 32$. Also plot the theoretical error rate curve on the same graph.
22.15.4 Decoding the LTE Code with the log-MAP Algorithm for Biphase Transmission Over an Additive White Gaussian Noise Channel

Repeat the steps in Section 22.15.3 but now use the log-MAP algorithm of Section 22.13. Compare your bit error rate vs. $E_b/N_0$ curves with those from Section 22.15.3. They should be the same.

22.15.5 Decoding the LTE Code with the max-log-MAP Algorithm for Biphase Transmission Over an Additive White Gaussian Noise Channel

Repeat the steps in Section 22.15.3 but now use the max-log-MAP approximation of Section 22.12. Compare the bit error rate curves with those for the exact BCJR algorithm. Also plot the theoretical error rate curve on the same graph.

22.15.6 Trellis Coded Modulation Example Using the Ungerboeck 4-State Systematic Code with an 8-Phase Constellation and Viterbi Decoding

The Ungerboeck systematic 4-state code shown in Figure 22.6 coupled to the 8-phase constellation (8PSK) shown in Figure 22.20 will be used as a simple introduction to trellis coded modulation (TCM). You should add a second input bit, $x_2(n)$, to the encoder that is passed directly to the output without being connected to the encoding circuit shown in Figure 22.6. The input $x_2(n)$ is called an uncoded bit. Designate the resulting additional output bit of the encoder as $y_3(n) = x_2(n)$. Assume that the received constellation points are corrupted by additive, complex, white Gaussian noise as explained in Example 22.20 on page 111. The performance of the TCM system will be compared to the performance of an uncoded system using a 4-phase (QPSK) constellation that consists of four points on a circle separated by 90 degrees like the set of open and filled circles in Figure 22.20. They can be rotated to any convenient angle. The encoder output at each stage includes the two input bits and the redundant check bit $y_1(n)$ which doubles the required constellation size from 4 for uncoded transmission to 8 for TCM.

The minimum distance between points in the 8PSK constellation is $d_0 = 2 \sin(\pi/8)$. The encoder outputs specify a sequence of partitions of the 8PSK constellation. The check bit, $y_1(n)$, selects the four circles (unfilled and black) if it is 0 and the four squares if it is 1. The minimum distance between points in one of these QPSK constellations is $d_1 = \sqrt{2}$. The bit, $y_2(n)$, selects a pair of points 180 degrees apart from the QPSK constellation selected by $y_1(n)$ and the distance between the pair is $d_2 = 2$. The uncoded bit, $y_3(n)$, selects one of the points in the subset selected by $y_1(n)$ and $y_2(n)$ and this is the transmitted constellation point.
The trellis diagram for 4-state Ungerboeck encoder in Figure 22.17 on page 107 is not quite complete for the encoder of this example. The two bits $y_1y_2$ on the branches in that diagram select one of the biphase subsets of the 8PSK constellation. The uncoded bit $y_3$ selects the point in the biphase subset. Therefore, each branch in Figure 22.17 must be split into two parallel branches, one for $y_3 = 0$ and one for $y_3 = 1$. Each branch in the augmented trellis will now be labelled with the three bits $y_1y_2y_3$.

Each of the two branches that diverge from a state have the same $y_1$ bit and the $y_2$ bits are different. Therefore, the constellation points for the two branches are selected from two different biphase constellations selected from the same QPSK constellation. Thus points for the two branches can be closest points in the QPSK constellation which are $d_1 = \sqrt{2}$ apart. Similarly, the two branches that converge on a state have the same $y_1$ and different $y_2$, so the minimum distance between points on these two branches is $d_1 = \sqrt{2}$. If branches leaving different states have different $y_1$ bits, the constellation points are selected from the two different QPSK subsets and the minimum distance between these points is $d_0 = 2 \sin \frac{\pi}{8}$.

Figure 22.20: Eight-Phase Constellation for the Ungerboeck 4-State Code
The distance between parallel branches is \( d_2 = 2 \).

The thick lines in Figure 22.17 show two paths that diverge from state 0 for more than one branch and then remerge to state 0 and have the minimum squared Euclidean distance. From the facts in the previous paragraph, this squared distance is \( d_f^2 = (\sqrt{2})^2 + (2\sin(\pi/8))^2 + (\sqrt{2})^2 = 4 + (2\sin(\pi/8))^2 \). The minimum squared distance between parallel branches is \( d_p^2 = 4 \). So, the minimum squared distance \( d_{\min}^2 = \min(d_p^2, d_f^2) = 4 \).

Suppose a constellation, \( A = \{a_i\} \), for TCM has power \( P_A = E\{|a_i|^2\} \) and the code has the minimum squared Euclidean distance \( d_{\min}^2 \), and a constellation \( B = \{b_i\} \) with minimum squared distance \( d_B^2 \) and power \( P_B = E\{|b_i|^2\} \) is used for uncoded transmission. The coding gain is defined to be

\[
\gamma = 10 \log_{10} \frac{d_{\min}^2/P_A}{d_B^2/P_B} \text{ dB} \tag{22.180}
\]

Normalization by the powers scales the constellations so the minimum distances in each are compared when the constellations have the same power. In this example, the constellation points lie on a circle of radius one so \( P_A = P_B = 1 \) and the coding gain is \( \gamma = 10 \log_{10} 4/2 = 3 \text{ dB} \).

Perform the following exercises for this encoder and decoder:

1. As a baseline for transmission using uncoded QPSK, do a bit error rate test vs. SNR using a constellation consisting of the four circles in Figure 22.20. Use a 23-stage maximal length shift register generator to simulate random binary data. Map successive pairs of bits to \( y_2 \) and \( y_3 \) shown in the figure and always let \( y_1 = 0 \) to select constellation points. Add complex, zero mean, white, Gaussian noise samples, \( v(n) = v_I(n) + jv_Q(n) \), to the transmitted constellation points. The noise components, \( v_I(n) \) and \( v_Q(n) \), are real, independent, zero mean, Gaussian, random variables with variances \( \sigma^2/2 \), so the power of the complex additive noise is \( \sigma^2 = E\{|v(n)|^2\} = E\{v_I^2(n)\} + E\{v_Q^2(n)\} \). The signal power is 1, so \( \text{SNR} = 1/\sigma^2 \).

2. Use the approach described in the next paragraph and example to derive a theoretical formula for and plot the probability of making a bit error at the receiver vs. SNR = \(-10 \log_{10} \sigma^2 \) dB for the QPSK constellation consisting of the four circles in Figure 22.20. Plot the error rate on a vertical log scale and SNR on a horizontal linear scale. Compare your measured bit error rate curve and the theoretical curve by plotting them on the same graph.

To simplify the derivation, rotate the QPSK constellation \( \pi/8 \) radians clockwise so the points lie at 45 degrees in the quadrants. The point with the bit label (010) should lie at 45 degrees in quadrant I. Assume noise with the same properties as before. Then the decision regions are just the four quadrants. That is, decode each noisy received symbol to the ideal constellation point in the same quadrant. Assume that the rotated point (010) is transmitted and derive formulas for \( P_C = \) the probability that a correct decision is made, \( P_{II} = \) the probability that the receiver decides on point (001), \( P_{III} = \) the probability that the receiver decides on point (011), and \( P_{IV} = \) the probability that
the receiver decides on point (000). Express the probabilities in terms of the function

$$Q(x) = \int_{x}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-\frac{t^2}{2}} dt$$  \hspace{1cm} (22.181)$$

Use the facts that a symbol error from (010) to (001) causes two bit errors, from (010) to (011) causes one bit error, and from (010) to (000) causes one bit error to find an exact expression for the bit error probability given (010) is transmitted. Only consider errors in the systematic bits, \(y_2\) and \(y_3\), since \(y_1\) carries no input data information. The error patterns are the same when the other points are transmitted, so the result does not depend on the transmitted point and is the unconditional bit error probability.

The probability of making a symbol error is \(P_S = P_{II} + P_{III} + P_{IV} = 1 - P_C\). Plot the symbol-error probability and compare it to the bit error probability.

**EXAMPLE 22.22**  Approach to Calculating the Probabilities

An equation for \(P_C\) will be derived in this example to help you see how to derive the error probabilities. Let the transmitted constellation point at time \(n\) be \(c = (1+j)/\sqrt{2}\) and the additive noise be \(v = v_I + jv_Q\). Then the received signal is

$$r = c + v = \left[ \frac{1}{\sqrt{2}} + v_I \right] + j \left[ \frac{1}{\sqrt{2}} + v_Q \right]$$ \hspace{1cm} (22.182)$$

A correct decision is made if \(r\) is in quadrant I. Using the assumption that \(n_I\) and \(n_Q\) are independent, zero-mean, Gaussian random variables each with variance \(\sigma^2/2\), the probability of a correct decision is

$$P_C = P \left( \frac{1}{\sqrt{2}} + v_I > 0 \cap \frac{1}{\sqrt{2}} + v_Q > 0 \right) = P \left( \frac{1}{\sqrt{2}} + v_I > 0 \right) P \left( \frac{1}{\sqrt{2}} + v_Q > 0 \right)$$

$$= P \left( v_I > -\frac{1}{\sqrt{2}} \right) P \left( v_Q > -\frac{1}{\sqrt{2}} \right)$$

$$= P \left( \frac{v_I}{\sigma/\sqrt{2}} > -\frac{1}{\sigma} \right) P \left( \frac{v_Q}{\sigma/\sqrt{2}} > -\frac{1}{\sigma} \right)$$ \hspace{1cm} (22.183)$$

Dividing \(v_I\) or \(v_Q\) by their standard deviation \(\sigma/\sqrt{2}\) gives random variables with a variance of 1 so

$$P_C = \left[ \int_{-1/\sigma}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-\frac{t^2}{2}} dt \right]^2 = \left[ 1 - Q \left( \frac{1}{\sigma} \right) \right]^2$$ \hspace{1cm} (22.184)$$

The last formula was obtained by using the symmetry of the Gaussian probability density function. Notice that \(1/\sigma = \sqrt{\text{SNR}}\).
3. Draw a complete section of the trellis showing the parallel paths labelled with $y_1y_2y_3$.

4. Implement the encoder. Use a 23-stage maximal length shift register generator to simulate the input data. Map the encoder binary outputs $(y_1, y_2, y_3)$ to constellation points and add simulated complex white Gaussian noise with variance $\sigma^2$ to the constellation points. Plot 1000 noisy constellation points in the complex plane using a dot for each point to observe the clouds around the ideal constellation points. Do this for a high, medium, and low SNR.

5. Implement a Viterbi decoder for infinite length sequences with a decoding delay of up to 32 branches. Use the squared Euclidean distance metric discussed in Example 22.20 on page 111. Some hints for implementing the decoder are:

   (a) The two states at time $n - 1$ connected to a state $[q_1(n), q_2(n)]$ at time $n$ are $[q_1(n - 1), q_2(n - 1)] = [q_2(n) + x(n - 1), q_1(n)]$ for $x(n - 1) = 0$ and 1.

   (b) As a first step, find the ideal constellation point in each of the four biphase subsets closest to the received point $r(n)$ in Euclidean distance and record the squared distances. In other words, first select the best branch for each of the four pairs of parallel branches.

   (c) Store the trellis history in a circular buffer and use pointers back to the best previous states to specify the surviving paths. Find the current state with the best metric and trace the surviving path back from this state for the selected coding delay.

   (d) Periodically subtract the smallest metric from all the metrics at the current trellis depth to keep the metrics bounded.

6. Perform bit error rate tests with your Viterbi decoder at various SNR to get a waterfall curve. Do this for several decoding delays. What delay achieves essentially all of the coding gain? Compare these curves with those for uncoded QPSK transmission.
22.16 References


Chapter 23

Turbo Codes

23.1 Introduction

Claude Shannon proved in 1948 that for transmission at any fixed rate below channel capacity, the error probability could be made arbitrarily small by using a sufficiently long error correcting code. His proof involved selecting codes at random and did not provide any computationally efficient decoding algorithm. That led to an intense search for codes that approached capacity and could be reasonably implemented. See Costello and Forney [4] for a very interesting history of “The Road to Capacity.” Cyclic block codes with a beautiful mathematical theory based on Galois fields were invented and closed the gap to capacity somewhat. Convolutional codes using soft decision decoding by the sequential decoding algorithm and then by the Viterbi algorithm were invented and closed the gap even more. However, the performance of these codes in terms of $E_b/N_0$ was still a couple of dB above the capacity limit. It seems that the more structured a code is, the less powerful it is. Then in May 1993 at the IEEE International Conference on Communications in Geneva Switzerland, Berrou, Glavieux, and Thitimajshima [2] presented a paper disclosing what they called turbo codes that achieved a bit error rate of $10^{-5}$ at a few tenths of a dB in $E_b/N_0$ from the capacity bound. Coding experts at the conference initially thought the results were wrong, but subsequent investigations showed they were correct. Turbo codes are now included in international standards like LTE 4G which is used in current cell phones. It is interesting that Berrou was not an information and coding theorist and had not completed a doctoral degree. However, he was a physicist and professor of integrated circuit design at the French Ecole Nationale Supérieure des Télécommunications. Their initial goal was to implement a SOVA decoder using MOS transistors. They were not coding experts and brainwashed by the vast traditional theory of error correcting codes and did not know that what they had accomplished “could not be done!”

Basically, Berrou, Glavieux, and Thitimajshima created codes with long block length by encoding a long information block twice – once with a simple low state recursive systematic convolutional encoder and a second time with an identical encoder but with the information bits permuted in order by an interleaver. The interleaver seems to introduce the randomness
suggested by Shannon’s theorem. The decoders for the two encoders use a soft decision algorithm like the BCJR algorithm and pass information back and forth between themselves iterating over a received codeword block a fixed number of times or until a convergence criterion is met. Several decades ago the computations required by the BCJR algorithm could not be implemented with low cost, small, and power efficient hardware. VLSI technology has now advanced to the point where it is very feasible and economical to implement the BCJR algorithm for convolutional codes with a moderate number of states and large block length.

Details for the turbo encoder are presented in Section 23.3 and for the decoder in Section 23.6. Before discussing the turbo encoder and decoder, formulas for capacity are presented in Section 23.1 to provide theoretical background. Some bounds on bit error rate are discussed in Section 23.4.

### 23.2 Capacity Formulas for the AWGN Channel

In Chapter 11 you learned how a sequence of numbers, \( a_n \), can be transmitted over an ideal lowpass channel with cutoff frequency \( W \) Hz at the rate of \( 2W \) numbers per second by sending the signal \( s(t) = \sum_{n=-\infty}^{\infty} a_n h(t - nT_s) \) where \( T_s = 1/(2W) \), \( h(t) = \sin(2\pi W t)/(2\pi W t) \), and \( H(\omega) = 1/(2W) \) for \( |\omega| < 2\pi W \) and 0 elsewhere. The numbers, \( a_n \), will also be referred to as levels or symbols.

Now suppose white Gaussian noise with two-sided power spectral density \( N_0/2 \) is added to the signal by the channel. This channel is called an additive, white, Gaussian noise (AWGN) channel in information theory. Claude Shannon showed that the capacity of this channel is

\[
C = W \log_2 \left( 1 + \frac{S}{N_0 W} \right) \text{ bits per second} \tag{23.1}
\]

where \( S \) is the transmitted signal power and \( N_0 W \) is the noise power in the signal band. It should be pointed out that the input signal \( s(t) \) is assumed to have a continuous probability density function for the AWGN channel and this capacity formula. In fact, the input signal must have a zero mean Gaussian density function to achieve this capacity. This suggests that input values should be used with non-equal probability to achieve the highest transmission rates on the AWGN channel. Shannon’s noisy channel coding theorem states that data can be sent over this channel at any rate less than the channel capacity \( C \) with arbitrarily small error probability by using a sufficiently long error correcting code. Transmission at rates higher than \( C \) always has an error probability bounded away from zero.

Let \( a_n \) be a sequence of symbols transmitted at the rate of \( 2W \) symbols/second and let the symbol duration be \( T_s = 1/(2W) \). Let each symbol carry \( R \) information bits so the information rate is \( \mathcal{R} = R2W \) bps. The quantity \( \eta = \mathcal{R}/W = 2R \) bits/second/Hz is called the **spectral efficiency**. If the energy used to transmit an information bit is \( E_b \), then each transmitted symbol has energy \( E_s = RE_b \). The transmitted signal power is

\[
S = \frac{E_s}{T_s} = E_s2W = 2W RE_b \tag{23.2}
\]
Substituting this formula for $S$ into (23.1), the channel capacity formula becomes

$$C = W \log_2 \left( 1 + 2R \frac{E_b}{N_0} \right) \text{ information bits/second}$$

or

$$\frac{C}{W} = \log_2 \left( 1 + 2R \frac{E_b}{N_0} \right) \text{ information bits/second/Hz}$$

To transmit at a rate not exceeding capacity, the information rate must satisfy

$$\frac{R}{W} = \frac{R2W}{W} = 2R < \log_2 \left( 1 + 2R \frac{E_b}{N_0} \right)$$

or

$$\frac{E_b}{N_0} > \frac{2^{2R} - 1}{2R} = \frac{2^\eta - 1}{\eta}$$

Some examples of the lower bound on $\frac{E_b}{N_0}$ are:

1 bit/symbol: $R = 1$, $\frac{E_b}{N_0} > 1.5$ or 1.7609 dB

1/2 bit/symbol: $R = 1/2$, $\frac{E_b}{N_0} > 1$ or 0 dB

1/3 bit/symbol: $R = 1/3$, $\frac{E_b}{N_0} > \frac{2^{2/3}1}{2/3}$ or -0.54974 dB

Another special case is when $R$ approaches zero. Then the channel bandwidth $W = \frac{R}{(2R)}$ approaches $\infty$ and

$$\lim_{R \to 0} \frac{2^{2R} - 1}{2R} = \ln 2 \text{ or } -1.5913 \text{ dB}$$

When the channel input symbols are constrained to be selected from a constellation with $M$ points like $M$-ary PAM or QAM, the capacity at a SNR value is less than that given above when the input is unconstrained. At high SNR’s the capacity converges to, but can never exceed, $\log_2 M$. At low SNR’s the capacities for $M$-ary and unconstrained inputs become almost identical. Let $\{a_1, \ldots, a_M\}$ be the points in an $M$-ary PAM constellation and let the points be used with probability $P_i = P(a_i)$. Assume the output of the receiver’s matched filter for a particular transmitted symbol has the form $y = a + z$ where $a$ is an element of the PAM constellation and $z$ is the noise component of the output. Let the probability density function for noise component of the output of the matched filter at the receiver be

$$f(z) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{z^2}{2\sigma^2}}$$
Then the capacity in bits per channel symbol is

$$C = \max_{\{P_i\}} \sum_{i=1}^{M} P_i \int_{-\infty}^{\infty} f(y|a_i) \log_2 \frac{f(y|a_i)}{\sum_{k=1}^{M} P_i f(y|a_k)} dy$$  \hspace{1cm} (23.9)$$

Assuming the PAM constellation is symmetric about the origin, it is shown in Information Theory texts that the capacity is maximized for equally likely inputs. See for example, Gallager [5]. Then the capacity formula becomes

$$C = \log_2 M - \frac{1}{M} \sum_{i=1}^{M} \int_{-\infty}^{\infty} f(y|a_i) \log_2 \frac{\sum_{k=1}^{M} f(y|a_k)}{f(y|a_i)} dy$$  \hspace{1cm} (23.10)$$

Using the fact that

$$f(y|a_i) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(y-a_i)^2}{2\sigma^2}}$$  \hspace{1cm} (23.11)$$

the capacity becomes

$$C = \log_2 M - \frac{1}{M} \sum_{i=1}^{M} \int_{-\infty}^{\infty} \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(y-a_i)^2}{2\sigma^2}} \log_2 \left( \sum_{k=1}^{M} e^{-\frac{(y-a_k)^2}{2\sigma^2} + \frac{(y-a_i)^2}{2\sigma^2}} \right) dy$$

$$= \log_2 M - \frac{1}{M} \sum_{i=1}^{M} \int_{-\infty}^{\infty} \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{y^2}{2\sigma^2}} \log_2 \left( \sum_{k=1}^{M} e^{-\frac{(y-a_k)^2}{2\sigma^2}} \right) dy$$

$$= \log_2 M - \int_{-\infty}^{\infty} \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{y^2}{2\sigma^2}} \frac{1}{M} \sum_{i=1}^{M} \log_2 \left( \sum_{k=1}^{M} e^{-\frac{(y-a_k)(y-a_i)}{2\sigma^2}} \right) dy$$  \hspace{1cm} (23.12)$$

Now let $M = 2$ with $a_1 = 1$ and $a_2 = -1$. Then the capacity formula can be reduced to

$$C = 1 - \int_{-\infty}^{\infty} \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{y^2}{2\sigma^2}} \frac{1}{2} \left[ \log_2 \left( 1 + e^{-\frac{2y^2}{\sigma^2}} \right) + \log_2 \left( 1 + e^{-\frac{2y^2}{\sigma^2}} \right) \right] dy$$  \hspace{1cm} (23.13)$$

For biphase signaling as described in Section 22.19, $\sigma^2 = N_0/(2E_s)$ where $N_0/2$ is the two-sided noise power spectral density and $E_s = 1$ is the symbol power so that the signal-to-noise ratio is $\text{SNR} = E_s/\sigma^2 = \sigma^{-2}$. MATLAB code for evaluating this integral using the Gauss-Hermite quadrature method is given in the appendix to this chapter. Results are plotted in Figure 23.1. Capacity for the unconstrained case is also shown on the same graph. For $C = 0.5 \text{ bits/symbol}$ the unconstrained case gives $\text{SNR} = 0 \text{ dB}$ while for the biphase case $\text{SNR} = 0.1900 \text{ dB}$, so $0.19 \text{ dB}$ is lost by using biphase modulation.
23.3 The Turbo Encoder

The block diagram for a typical turbo encoder is shown in Figure 23.2. A block of $K$ input information bits is represented by the vector $\mathbf{x}$. The input bits are sent directly to the output vector $\mathbf{y}_0$ so the turbo code is systematic. The input bits are also applied to the input of the recursive, systematic encoder represented by the “RSC Encoder 1” block which generates the $K$-dimensional output check bit vector $\mathbf{y}_1$. This encoder typically has rate $R = 1/2$, so each individual input bit generates one output check bit. The order of the input bits is permuted by the “Interleaver” resulting in the $K$-dimensional vector $\tilde{\mathbf{x}}$. The permuted input block $\tilde{\mathbf{x}}$ is then applied to “RSC Encoder 2” which is the same as RSC Encoder 1 resulting in the second check bit vector $\mathbf{y}_2$. For each input information bit $x(n)$, three output bits $y_0(n)$, $y_1(n)$, and $y_2(n)$ are generated. Thus each codeword has length $N = 3K$ and the code has rate $R = 1/3$. The code bits are biphase modulated and serially transmitted over the channel.

Higher code rates can be achieved by not sending some of the check bits. This is called \textit{puncturing} the code. For example, a rate $1/2$ code can be obtained by always transmitting...
the information bit and just one of the check bits alternately taken from Encoder 1 and Encoder 2. The puncturing rule is commonly represented by a puncturing matrix \( P \) with three rows and the number of columns equal to the period of the puncturing process. Each row is for one of the encoder output streams. An element in row \( i \) that is 1 indicates that the bit from \( y_i \) should be transmitted and a 0 indicates it should be punctured. For example, a common puncturing matrix for a rate 1/2 code is

\[
P = \begin{bmatrix}
1 & 1 \\
1 & 0 \\
0 & 1 
\end{bmatrix}
\]  

The constituent encoders are recursive systematic convolutional (RSC) encoders implemented by Type 1 direct forms as described in Section 22.2.1 and shown in Figure 22.2. One reason the Type 1 direct form is used is that the final encoder state can easily be forced to zero as explained in Example 22.6. Recursive encoders must be used and the reason for this is partly explained in Section 23.4.

**EXAMPLE 23.1** The LTE Rate 1/3 Turbo Encoder

The LTE 4G standard [1, Section 5.1.3.2] includes the turbo encoder shown in Figure 23.3. The constituent encoders are rate 1/2 RSC encoders. These are discussed in detail in Example 22.6 on page 89. The encoders are started in the zero state at time \( n = 0 \). At time \( n \) the coded output bits are transmitted serially in order \( y_0(n), y_1(n), y_2(n) \).

The encoders are driven back to state zero after all the information bits \( x(0), \ldots, x(K-1) \) have been encoded. The bits transmitted during this process are called *tail bits*. The 1st Constituent Encoder is driven to state zero by connecting its switch as shown in Figure 23.3 to the feedback bit \( z_1(n) \) and the encoder is clocked three times. The 2nd Constituent encoder is disabled for these three clocks. Then the 2nd Constituent encoder is driven to
23.3 The Turbo Encoder

state zero by connecting its switch to its feedback bit $z_2(n)$ and clocking it three times. The 1st Constituent Encoder is disabled while the 2nd is driven to state zero. The transmitted tail bits are:

$y_0(K), y_1(K), y_0(K + 1), y_1(K + 1), y_0(K + 2), y_1(K + 2),$

$\tilde{y}_0(K), y_2(K), \tilde{y}_0(K + 1), y_2(K + 1), \tilde{y}_0(K + 2), y_2(K + 2)$

The index for the outputs from the 2nd encoder start at $K$ rather than $K + 3$ because it is disabled while the 1st encoder is clocked three times.

The interleaver permutes the order of the input bits according to a rule of the form

$$\Pi(n) = (f_1 n + f_2 n^2) \mod K$$  \hfill (23.15)

so that

$$\tilde{x}(n) = x(\Pi(n)) \text{ for } n = 0, \ldots, K - 1$$  \hfill (23.16)

This is called a quadratic permutation polynomial (QPP) interleaver [10]. The LTE standard has options for 188 different block lengths with $K$ ranging from 40 to 6144. The values for $f_1$ and $f_2$ are given in Table 5.1.3-3 of the standard. Two examples are: (1) $K = 1008$, $f_1 = 55$, $f_2 = 84$; and (2) $K = 6144$, $f_1 = 263$, $f_2 = 480$. The coefficients were chosen to give a permutation resulting in a code with a good distance spectrum.
23.4 Bit Error Probability for Turbo Codes

The bit error rate performance of turbo codes has been studied extensively. See, for example, [6], [8], [9], and [11]. The bounds in this section are for the case where the code bits are transmitted using biphase signaling as described in Section 22.19. The turbo code is assumed to be an \((N, K)\) block code, so the code rate is \(R = K/N\) information bits per code bit and there are \(2^K\) codewords. Suppose the codewords are labeled from 1 to \(2^K\), the Hamming weight of the \(K\)-bit information vector associated with codeword \(i\) is \(w_i\), and the Hamming weight of codeword \(i\) is \(d_i\). Then an upper bound on the bit error probability is

\[
P_b \leq \frac{2^K}{K} \sum_{i=1}^{2^K} \frac{w_i}{K} Q \left( \sqrt{\frac{2d_iRE_b}{N_0}} \right) \tag{23.17}
\]

Let \(N_d\) be the number of codewords of Hamming weight \(d\), \(W_d\) be the sum of the weights of all information vectors generating codewords of weight \(d\), and \(\bar{w}_d = W_d/N_d\). The quantity \(\bar{w}_d\) is the average weight of information vectors generating codewords of weight \(d\). Also let \(d_{\text{free}}\) be the minimum Hamming distance between codewords. It is called the free distance of the code. Then the bit error rate bound can be written as

\[
P_b \leq \sum_{d=d_{\text{free}}}^{N} \frac{W_d}{K} Q \left( \sqrt{\frac{2dRE_b}{N_0}} \right) = \sum_{d=d_{\text{free}}}^{N} \frac{N_d\bar{w}_d}{K} Q \left( \sqrt{\frac{2dRE_b}{N_0}} \right) \tag{23.18}
\]

When the SNR, \(E_b/N_0\), becomes large, the term for \(d = d_{\text{free}}\) in (23.18) becomes significantly larger than the rest of the terms and \(P_b\) approaches

\[
P_b \approx \frac{N_{\text{free}}\bar{w}_{\text{free}}}{K} Q \left( \sqrt{\frac{2d_{\text{free}}RE_b}{N_0}} \right) \tag{23.19}
\]

where \(N_{\text{free}}\) is the number of codewords of weight \(d_{\text{free}}\), and \(\bar{w}_{\text{free}}\) is the average weight of information sequences causing codewords of weight \(d_{\text{free}}\).

Typical bit error rate curves obtained by simulation of a \(K = 2^{16} = 65536\), rate \(R = 1/2\) turbo code are illustrated in Figure 23.4. The curves are based on Figure 14.1 of Moon [6, p. 583]. The transfer function from the rate 1/2 constituent encoder input to check symbol was \((1+D^4)/(1+D+D^2+D^3+D^4)\). Puncturing was used increase the code rate from 1/3 to 1/2. Starting around \(E_b/N_0 = 0.5\) dB for the 18 iteration curve, the bit error rate rapidly falls from about \(6 \times 10^{-2}\) to \(3 \times 10^{-6}\). This is called the waterfall region. The performance is about 0.5 dB above the Shannon limit for biphase transmission when \(P_b \approx 3 \times 10^{-6}\). Then the curve flattens out for higher \(E_b/N_0\) and this is called the error floor region. At high \(E_b/N_0\) by far the most likely errors are to the codewords nearest the transmitted ones. These are at distance \(d_{\text{free}}\) from the transmitted one and the \(d_{\text{free}}\) bit error rate curve approaches the \(d_{\text{free}}\) bound of (23.19). The large improvement in turbo code performance over other known types of codes in the waterfall region has been explained by the fact that well designed turbo codes have many fewer low weight codewords associated with low weight information vectors than other codes.
Recursive encoders are required for turbo codes. See [11, Section 4.3] for a detailed theoretical analysis and explanation for this requirement. It was shown in Section 22.4 that a non-recursive non-systematic convolutional code can be turned into an equivalent systematic recursive convolutional code. The total set of codewords for each type is the same, but the association of input sequences with codewords is different. For example, weight one information sequences produce finite weight codewords with a non-recursive encoder. If the encoder is initially in the zero state, the code words will have weight equal to 1 for the systematic output + (the number of nonzero coefficients in the polynomial transfer functions from the inputs to the constituent encoders to the check streams) and this weight is relatively small. On the other hand, weight 1 information sequences will cause infinite weight codewords for recursive encoders because of the feedback resulting from the denominator polynomials in the transfer functions if the input is extended with 0’s forever. If the denominator polynomial has degree $L$ and cannot be factored into the product of polynomials with binary coefficients, the check sequence from the constituent encoder will repeat with period $2^L - 1$.
or one of its factors. If the period is \( P \) and the transfer function is \( a(D)/b(D) \), then it can be shown that \( b(D) \) divides \( D^P + 1 \). Let \( q(D) = (D^P + 1)/b(d) \). Then when the encoder is initially in state zero, weight 2 information sequences with transform \( D^I(D^P + 1) \) for some integer \( I \) generate the finite weight check sequences \( D^I(D^P + 1)a(D)/b(D) = D^Iq(D)a(D) \).

Decoding errors are most likely to be made to the nearest code words in Hamming distance on the AWGN channel with biphase modulation. It has been shown that \( N_d\tilde{w}_d \) is smaller for turbo codes with recursive encoders than with non-recursive encoders at smaller values of \( d \). It has also been shown that this “spectral thinning” improves as the length of the interleaver increases. A good interleaver tends to make the inputs to the two constituent encoders appear to be uncorrelated sequences. This makes the code seem more random. If a low weight input to one encoder generates a low weight codeword, the interleaved input to the other encoder is likely to generate a higher weight codeword.

### 23.5 Exact Minimum Bit Error Probability Decoding of Turbo Codes

In this section it will be argued that exact decoding to minimize the bit error probability for a turbo code transmitted over an AWGN channel with biphase modulation is too computationally intensive to be practical except for very short codes. It will be assumed that information bits are independent and equally likely so that the codewords are equally likely. The decision rule that minimizes the probability of error for deciding on the value of a particular transmitted bit \( x(n) \) is to form the log likelihood function

\[
\Lambda_n(r) = \log \frac{P(x(n) = 1 | r)}{P(x(n) = 0 | r)} = \log \frac{\sum_{\{x|\text{x(n)=1}\}} P(x | r)}{\sum_{\{x|\text{x(n)=0}\}} P(x | r) - \sum_{\{x|\text{x(n)=0}\}} P(x | r) \cdot \sum_{\{x|\text{x(n)=0}\}} P(x \cdot r | x)} (23.20)
\]

and decide

\[
x(n) = \begin{cases} 1 & \text{if } \Lambda_n(r) \geq 0 \\ 0 & \text{if } \Lambda_n(r) < 0 \end{cases} (23.21)
\]

When \( K \) information bits are transmitted, calculating this LLR requires summing over all \( M = 2^K \) codewords. The number of information bits per codeword can be several hundred to several thousand so \( M \) can be a very large number. Making the required computations in real-time is well beyond the capability of today’s hardware except for very short codes.

### 23.6 Iterative Decoding of Turbo Codes

Berrou, et. al., discovered a suboptimal iterative decoding scheme that astonished the coding community by achieving bit error probabilities less than \( 10^{-5} \) at a few tenths of a dB from the Shannon limit and can be implemented economically with today’s technology. A block diagram of a turbo decoder using the MAP algorithm and LLR’s is shown in Figure 23.5. Basically, the turbo decoder uses two MAP decoders arranged in series for the two constituent
23.6 Iterative Decoding of Turbo Codes

Encoders with the output of the first decoder sent to the second one and the output of the second one fed back to the first one. Appropriate interleaving and deinterleaving is included between the decoders. Actually, only the extrinsic information discussed in Section 22.11.3.1 is passed between the decoders. The decoding of a block is repeated (iterated) a fixed number of times or until a stopping criterion is met. The implementation that will be discussed in detail in this section assumes the code bits are transmitted by biphase modulation on an AWGN channel and the outputs of a matched filter are observed at the receiver as described in Section 22.19.

![MAP Turbo Decoder for the Rate 1/3 Turbo Encoder](image)

**Figure 23.5: MAP Turbo Decoder for the Rate 1/3 Turbo Encoder**

### 23.6.1 MAP Decoder 1 Computations

The inputs to MAP Decoder 1 are the received systematic stream vector $r_0$, the received check stream vector $r_1$ from the 1st Constituent Encoder, and the LLR vector $\lambda_{1p}$ which is the deinterleaved version of the extrinsic LLR vector $\lambda_{2e}$ generated by MAP Decoder 2. The extrinsic information $\lambda_{1p}$ is used as the *a priori* component of the LLR for MAP Decoder.
1. The extrinsic information from Decoder 2 for the first iteration is set to \( \lambda_{1p} = 0 \) which indicates equally likely input bit values. Suppose \( \lambda_{1p}(n) \) is the actual LLR for \( x(n) \), that is,

\[
\lambda_{1p}(n) = \log \frac{P(x(n) = 1)}{P(x(n) = 0)}
\]  
(23.22)

As before, let a logical bit value \( v \) be mapped to the biphase value \( \tilde{v} = (-1)^v \) so a logical 0 gets mapped to 1 and a logical 1 gets mapped to \(-1\). This is the opposite of many other articles and accounts for the minus sign in some following equations. Then

\[
P(x(n) = 1) = \frac{e^{\lambda_{1p}(n)}}{1 + e^{\lambda_{1p}(n)}} \quad \text{and} \quad P(x(n) = 0) = \frac{1}{1 + e^{\lambda_{1p}(n)}} = \frac{e^{-\lambda_{1p}(n)}}{1 + e^{-\lambda_{1p}(n)}}
\]  
(23.23)

which can be reduced to the single equation

\[
P(x(n) = v) = \frac{e^{-\tilde{v}\lambda_{1p}(n)}}{1 + e^{-\tilde{v}\lambda_{1p}(n)}} \quad \text{for} \quad v = 0 \text{ and } 1
\]  
(23.24)

Map Decoder 1 uses (23.24) to compute the “a priori” probabilities for the entire block and then computes for the entire block

\[
\gamma_{n,1}^i(l', l) = \begin{cases} 
    P(x(n) = i) e^{-\frac{[r_0(n) - (-1)^i]^2 + [r_1(n) - \tilde{y}_1(n; l', l)]^2}{2\sigma^2}} & (l', l) \in B_n^i, \ i \in (0, 1) \\
    0 & (l', l) \notin B_n^i
\end{cases}
\]  
(23.25)

Once all the values of \( \gamma_{n,1}^i(l', l) \) have been computed, Decoder 1 can compute \( \alpha_{n,1}(l) \) and \( \beta_{n,1}(l) \) using the following iterations which were derived in Section 22.11.2. Let the state at time \( n - 1 \) transitioning to state \( \ell \) at time \( n \) for \( x(n - 1) = 0 \) be \( l_{0,l} \) and the state at time \( n - 1 \) transitioning to state \( \ell \) at time \( n \) for \( x(n - 1) = 1 \) be \( l_{1,l} \). Then

\[
\alpha_{n,1}(l) = \sum_{l'=0}^{M_s-1} \alpha_{n-1,1}'(l') \sum_{i=0}^{1} \gamma_{n-1,1}^i(l', l) = \alpha_{n-1,1}'(l_{0,l}) \gamma_{n-1,1}^0(l_{0,l}, l) + \alpha_{n-1,1}'(l_{1,l}) \gamma_{n-1,1}^1(l_{1,l}, l)
\]  
(23.26)

where \( L_T \) is the total number of trellis branches including tail branches. To preserve numerical accuracy, \( \alpha_{n,1}(l) \) should be normalized as explained in Section 22.11.3 to

\[
\alpha_{n,1}'(l) = \frac{\alpha_{n,1}(l)}{M_s-1} = \sum_{l''=0}^{M_s-1} \frac{\alpha_{n-1,1}'(l'') \sum_{i=0}^{1} \gamma_{n-1,1}^i(l', l) \sum_{i=0}^{1} \gamma_{n-1,1}^i(l', l'')}{\sum_{l''=0}^{M_s-1} \alpha_{n,1}(l'')} \sum_{l'=0}^{M_s-1} \sum_{l'=0}^{M_s-1} \alpha_{n-1,1}'(l') \sum_{i=0}^{1} \gamma_{n-1,1}^i(l', l'')}
\]  
(23.27)
Assuming the encoder is started in state 0 at time \( n = 0 \), the initial conditions are:
\[
\alpha_{0,1}(0) = 1 \quad \text{and} \quad \alpha_{0,1}(l) = 0 \quad \text{for} \quad l = 1, \ldots, M_s - 1
\] (23.28)

Given the state at time \( n \) is \( \ell \), let the state at time \( n + 1 \) caused by \( x(n) = 0 \) be \( l_0' \) and the state caused by \( x(n) = 1 \) be \( l_1' \). Then
\[
\beta_{n,1}(l) = \sum_{l' = 0}^{M_s-1} \beta_{n+1,1}(l') \gamma_{n,1}^i(l, l')
\]
\[
= \beta_{n+1,1}(l_0') \gamma_{n,1}^0(l, l_0') + \beta_{n+1,1}(l_1') \gamma_{n,1}^1(l, l_1')
\]
for \( n = L_T - 1, \ldots, 0; \quad l = 0, \ldots, M_s - 1 \) (23.29)

To preserve numerical accuracy, \( \beta_{n,1}(l) \) should be normalized to
\[
\beta_{n,1}'(l) = \frac{\beta_{n,1}(l)}{\sum_{l'' = 0}^{M_s-1} \beta_{n,1}(l'')} \sum_{l'' = 0}^{M_s-1} \beta_{n,1}(l'') \sum_{l = 0}^{M_s-1} \beta_{n+1,1}(l') \gamma_{n,1}^i(l', l')
\]
(23.30)

Assuming the encoder ends in state 0 at depth \( n = L_T \), the initial conditions are:
\[
\beta_{L_T,1}(0) = 1 \quad \text{and} \quad \beta_{L_T,1}(l) = 0 \quad \text{for} \quad l = 1, \ldots, M_s - 1
\] (23.31)

Once \( \alpha_{n,1}, \beta_{n,1}, \) and \( \gamma_{n,1} \) have been calculated, Decoder 1 can compute the complete LLR for \( x(n) \) as
\[
\Lambda_{n,1}(r_0, r_1) = \log \left| \sum_{(l', l) \in B_1} \alpha_{n,1}(l') \beta_{n+1,1}(l') \gamma_{n,1}^i(l', l) \right|\]
\[
= \log \left| \sum_{(l', l) \in B_1} \alpha_{n,1}(l') \beta_{n+1,1}(l') e^{-\frac{1}{2\sigma^2} [r_1(n) - \tilde{y}_1(n', l)]^2} \right|
\] (23.32)

Computing the complete LLR is useful if bit decisions are desired in Decoder 1.

As explained in Section 22.11.3.1, \( \Lambda_{n}(r_0, r_1) \) can be decomposed into the following sum:
\[
\Lambda_{n,1}(r_0, r_1) = \log \left| \frac{P(x(n) = 1)}{P(x(n) = 0)} \right| - \frac{2}{\sigma^2} r_0(n) + \\
\log \left| \sum_{(l', l) \in B_1} \alpha_{n,1}(l') \beta_{n+1,1}(l') e^{-\frac{1}{2\sigma^2} [r_1(n) - \tilde{y}_1(n', l)]^2} \right|
\]
\[
= \lambda_{1p}(n) + \lambda_{1s}(r_0(n)) + \lambda_{1e}(r_1(n))
\] (23.33)
where the facts that \( \tilde{y}^2_1(n; l', l) = 1 \) and the \( r_1(n) \) terms are the same in the numerators and denominators were used to cancel extra factors. The first term in the sum on the right,

\[
\lambda_{1p}(n) = \log \frac{P(x(n) = 1)}{P(x(n) = 0)}
\]

(23.34)
is the \textit{a priori} input bit LLR which is actually the deinterleaved extrinsic LLR passed by Decoder 2 to Decode 1 for turbo decoding. The second term,

\[
\lambda_{1s}(r_0(n)) = \log \frac{p(r_0(n) | x(n) = 1)}{p(r_0(n) | x(n) = 0)} = - \frac{2}{\sigma^2} r_0(n)
\]

(23.35)
is the systematic input, \( r_0(n) \), LLR. The third term,

\[
\lambda_{1e}(r_1(n)) = \log \frac{\sum_{(l', l) \in B^n_0} \alpha'_{n, 1}(l') \beta'_{n+1, 1}(l) e^{-\frac{1}{2 \sigma^2} [r_1(n) - \tilde{y}_1(n, l')]^2}}{\sum_{(l', l) \in B^n_0} \alpha'_{n, 1}(l') \beta'_{n+1, 1}(l) e^{-\frac{1}{2 \sigma^2} [r_1(n) - \tilde{y}_1(n, l')]^2}}
\]

(23.36)

\[
= \log \frac{\sum_{(l', l) \in B^n_0} \alpha'_{n, 1}(l') \beta'_{n+1, 1}(l) e^{-\frac{1}{2 \sigma^2} r_1(n) \tilde{y}_1(n, l')}}{\sum_{(l', l) \in B^n_0} \alpha'_{n, 1}(l') \beta'_{n+1, 1}(l) e^{-\frac{1}{2 \sigma^2} r_1(n) \tilde{y}_1(n, l')}}
\]

(23.37)
is the \textit{extrinsic} LLR for Decoder 1.

Since Decoder 1 has already computed \( \Lambda_n(r_0, r_1) \), its extrinsic LLR can be computed as

\[
\lambda_{1e}(r_1(n)) = \Lambda_{n, 1}(r_0, r_1) - \lambda_{1p}(n) - \left( - \frac{2}{\sigma^2} r_0(n) \right)
\]

(23.38)

This extrinsic LLR sequence is interleaved and passed to Decoder 2 as \( \lambda_{2p} = \hat{\lambda}_{1e} \).

### 23.6.2 MAP Decoder 2 Computations

The systematic received stream \( r_0 \) is interleaved to form \( \tilde{r}_0 \) and used as the systematic input for MAP Decoder 2. The received check stream \( r_2 \) from the check stream \( y_2 \) transmitted by Encoder 2 is connected directly to MAP Decoder 2. The extrinsic LLR vector \( \lambda_{1e} \) from Decoder 1 is interleaved to form \( \lambda_{2p} = \hat{\lambda}_{1e} \) and applied to Decoder 2 as the \textit{a priori} LLR. Decoder 2 computes

\[
P_2(\hat{x}(n) = v) = \frac{e^{-\frac{1}{2} \lambda_{2p}(n)}}{1 + e^{-\frac{1}{2} \lambda_{2p}(n)}} \quad \text{for } v = 0, 1 \text{ and } n = 0, \ldots, L_T - 1
\]

(23.39)

and then

\[
\gamma_{n, 2}^i(l', l) = \begin{cases} 
P_2(x(n) = i) e^{-\frac{[\tilde{r}_0(n) - (-1)^i]^2 + [r_2(n) - \tilde{y}_2(n, l', l)]^2}{2\sigma^2}} & (l', l) \in B_n^i, \ i \in (0, 1) \\
0 & (l', l) \notin B_n^i
\end{cases}
\]

(23.40)
23.6 Iterative Decoding of Turbo Codes

Once all the values of $\gamma_{n,2}(l', l)$ have been computed, Decoder 2 can compute $\alpha_{n,2}(l)$ and $\beta_{n,2}(l)$ as follows. Let the state at time $n - 1$ transitioning to state $\ell$ at time $n$ for $x(n-1) = 0$ be $l'_0$ and the state at time $n - 1$ transitioning to state $\ell$ at time $n$ for $x(n-1) = 1$ be $l'_1$. Then

$$\alpha_{n,2}(l) = \sum_{l'=0}^{M_s-1} \alpha'_{n-1,2}(l') \sum_{i=0}^{1} \gamma_{n-1,2}^i(l', l)$$

$$= \alpha'_{n-1,2}(l'_0) \gamma_{n-1,2}^0(l'_0, l) + \alpha'_{n-1,2}(l'_1) \gamma_{n-1,2}^1(l'_1, l)$$

for $n = 0, \ldots, L_T - 1$; $l = 0, \ldots, M_s - 1$ (23.41)

To preserve numerical accuracy, $\alpha_{n,2}(l)$ should be normalized as to

$$\alpha'_{n,2}(l) = \frac{\alpha_{n,2}(l)}{\sum_{l''=0}^{M_s-1} \alpha_{n,2}(l'')} = \frac{\sum_{l'=0}^{M_s-1} \alpha'_{n-1,2}(l') \sum_{i=0}^{1} \gamma_{n-1,2}^i(l', l)}{\sum_{l'=0}^{M_s-1} \sum_{l'=0}^{M_s-1} \alpha'_{n-1,2}(l') \sum_{i=0}^{1} \gamma_{n-1,2}^i(l', l'')}$$

Assuming the encoder is started in state 0 at time $n = 0$, the initial conditions are:

$$\alpha_{0,2}(0) = 1 \text{ and } \alpha_{0,2}(l) = 0 \text{ for } l = 1, \ldots, M_s - 1$$

(23.43)

Given the state at time $n$ is $\ell$, let the state at time $n + 1$ caused by $x(n) = 0$ be $l'_0$ and the state caused by $x(n) = 1$ be $l'_1$. Then

$$\beta_{n,2}(l) = \sum_{l'=0}^{M_s-1} \beta'_{n+1,2}(l') \sum_{i=0}^{1} \gamma_{n,2}^i(l, l')$$

$$= \beta'_{n+1,2}(l'_0) \gamma_{n,2}^0(l, l'_0) + \beta'_{n+1,2}(l'_1) \gamma_{n,2}^1(l, l'_1)$$

for $n = L_T - 1, \ldots, 1, 0$; $l = 0, \ldots, M_s - 1$ (23.44)

To preserve numerical accuracy, $\beta_{n,2}(l)$ should be normalized to

$$\beta'_{n,2}(l) = \frac{\beta_{n,2}(l)}{\sum_{l''=0}^{M_s-1} \beta_{n,2}(l'')} = \frac{\sum_{l'=0}^{M_s-1} \beta'_{n+1,2}(l') \gamma_{n,2}(l, l')} {\sum_{l'=0}^{M_s-1} \sum_{l'=0}^{M_s-1} \beta'_{n+1,2}(l') \gamma_{n,2}(l'', l')}$$

Assuming the encoder ends in state 0 at depth $n = L_T$, the initial conditions are:

$$\beta_{L_T,2}(0) = 1 \text{ and } \beta_{L_T,2}(l) = 0 \text{ for } l = 1, \ldots, M_s - 1$$

(23.46)
Now Decoder 2 can compute its complete LLR for \( x(n) \) as

\[
\Lambda_{n,2}(\mathbf{r}_0, \mathbf{r}_2) = \log \sum_{(l',l) \in B_1^n} \alpha'_{n,2}(l') \beta'_{n+1,2}(l) \gamma_{n,2}^1(l', l) \\
\sum_{(l',l) \in B_0^n} \alpha'_{n,2}(l') \beta'_{n+1,2}(l) \gamma_{n,2}^0(l', l)
\]  

(23.47)

The extrinsic LLR for Decoder 2 can be computed as

\[
\lambda_{2e}(\mathbf{r}_2(n)) = \Lambda_{n,2}(\mathbf{r}_0, \mathbf{r}_2) - \lambda_{2p}(n) - \left( -\frac{2}{\sigma^2} \mathbf{r}_0(n) \right)
\]  

(23.48)

This extrinsic LLR sequence is deinterleaved and passed to Decoder 1 as \( \lambda_{1p} \). Then the Decoder 1 and Decoder 2 processing is repeated until some stopping criterion is reached.

## 23.7 Iteration Stopping Methods

In each iteration Decoder 1 does its calculations and passes its extrinsic LLR to Decoder 2. Then Decoder 2 does its calculations and passes its extrinsic LLR back to Decoder 1. Then the process is repeated. During the initial iterations the LLR’s change significantly. As the iterations progress the LLR’s change more and more slowly and converge towards final values. Several methods that have been proposed for stopping the iterations are discussed below.

### 23.7.1 A Fixed Number of Iterations

The simplest method is to repeat the iterations a fixed number of times. The number of iterations is predetermined by simulations with the specific code and range of expected SNR’s. Figure 23.4 illustrates the behavior as the number of iterations increases. After a certain number of iterations minimal improvements are achieved.

### 23.7.2 The Cross Entropy Criterion

During some code blocks the noise may be smaller than usual and the iterations converge sooner than the fixed maximum number of iterations. When early convergence can be detected the iterations can be stopped to save processing time. One approach that has been suggested is to compare the probability vectors computed from the extrinsic LLR’s at each iteration and stop when they are very similar. At iteration \( m \) let the extrinsic LLR inputs to Decoder 1 from Decoder 2 be \( \lambda_{1p}^{[m]}(n) \) which are the deinterleaved values of \( \lambda_{2e}^{[m-1]}(\mathbf{r}_2(n)) \) so that \( \lambda_{2e}^{[m-1]}(\mathbf{r}_2(n)) = \lambda_{1p}^{[m]}(n) \). Let the extrinsic LLR inputs to Decoder 2 from Decoder 1
be $\tilde{\lambda}_{1e}^{[m]}(r_1(n))$. The corresponding probabilities for Decoder 1 are

$$P_1^{[m]}(x(n) = v) = \frac{e^{-\tilde{\lambda}_{1e}^{[m]}(r_1(n))}}{1 + e^{-\tilde{\lambda}_{1e}^{[m]}(r_1(n))}} \quad \text{for } v = 0, 1 \text{ and } n = 0, \ldots, L_T - 1$$

(23.49)

and for Decoder 2

$$P_2^{[m]}(\tilde{x}(n) = v) = \frac{e^{-\tilde{\lambda}_{1e}^{[m]}(r_1(n))}}{1 + e^{-\tilde{\lambda}_{1e}^{[m]}(r_1(n))}} \quad \text{for } v = 0, 1 \text{ and } n = 0, \ldots, L_T - 1$$

(23.50)

or with deinterleaving

$$P_2^{[m]}(x(n) = v) = \frac{e^{-\tilde{\lambda}_{1e}^{[m]}(r_1(n))}}{1 + e^{-\tilde{\lambda}_{1e}^{[m]}(r_1(n))}} \quad \text{for } v = 0, 1 \text{ and } n = 0, \ldots, L_T - 1$$

(23.51)

The similarity of the probability distributions for Decoder 1 and 2 can be measured by their cross entropy (CE). Let $v = [v_0, v_1, \ldots, v_{L_T-1}]$ be an $L_T$-dimensional vector with binary components $v_n \in \{0, 1\}$ and let $B$ be the set of all the $2^{L_T}$ possible $v$. Let the $L_T$-dimensional probability distributions for Decoders 1 and 2 at iteration $m$ be

$$Q_1(v) = \prod_{n=0}^{L_T-1} P_1^{[m]}(x(n) = v_n) \quad \text{and} \quad Q_2(v) = \prod_{n=0}^{L_T-1} P_2^{[m]}(x(n) = v_n) \quad \text{for } v \in B$$

(23.52)

Notice that $x(n) = v_n$ appears as the argument in both probabilities so interleaving has been taken into account. The cross entropy of these two distributions is

$$T(m) = \sum_{v \in B} Q_2(v) \log \frac{Q_2(v)}{Q_1(v)} = \sum_{v \in B} \sum_{n=0}^{L_T-1} \sum_{v_n=0}^1 P_2^{[m]}(x(n) = v_n) \log \frac{P_2^{[m]}(x(n) = v_n)}{P_1^{[m]}(x(n) = v_n)}$$

(23.53)

It can be shown that the cross entropy is always greater than or equal to zero and is zero if and only if $Q_1(v) = Q_2(v)$. The LLR’s for Decoder 1 at iteration $m$ are

$$\Lambda_{n,1}^{[m]}(r_0, r_1) = \lambda_{1p}^{[m]}(n) + \lambda_{1e}^{[m]}(r_0(n)) + \lambda_{1e}^{[m]}(r_1(n))$$

(23.54)

which after interleaving become

$$\tilde{\Lambda}_{n,1}^{[m]}(r_0, r_1) = \lambda_{2e}^{[m-1]}(r_2(n)) - \frac{2}{\sigma^2} \tilde{r}_0(n) + \tilde{\lambda}_{1e}^{[m]}(r_1(n))$$

(23.55)

The LLR’s for Decoder 2 at iteration $m$ are

$$\Lambda_{n,2}^{[m]}(r_0, r_2) = \tilde{\lambda}_{1e}^{[m]}(r_1(n)) - \frac{2}{\sigma^2} \tilde{r}_0(n) + \lambda_{2e}^{[m]}(r_2(n))$$

(23.56)

The difference of the LLR’s for the two decoders is

$$\Delta_n^{[m]} = \Lambda_{n,2}^{[m]}(r_0, r_2) - \tilde{\Lambda}_{n,1}^{[m]}(r_0, r_1) = \lambda_{2e}^{[m]}(r_2(n)) - \lambda_{2e}^{[m-1]}(r_2(n))$$

(23.57)
As convergence is neared, the LLR’s from iteration \( m - 1 \) to \( m \) change very slightly, so \( \Delta_n[m] \) approaches 0. This implies that the complete LLR’s \( \Lambda_{n,1}[m](r_0, r_1) \) and \( \Lambda_{n,2}[m](r_0, r_2) \) become the same and bit decisions based on Decoder 1 or Decoder 2 become the same.

When the iterations near convergence, it can be shown that the cross entropy can be approximated by

\[
T(m) \approx \sum_{n=0}^{L_T-1} \frac{|\Delta_n[m]|^2}{e^{2e^{\Lambda_{n,1}[m] (r_0, r_1)}}}
\]  

(23.58)

See Shao et. al., [7] and Moon [6, Section 14.3.14] for details.

Simulations have shown that when \( T(m) \) decreases to a value between \( 10^{-2}T(0) \) and \( 10^{-4}T(0) \) the iterations can be stopped with little degradation of bit error rate performance over the fixed number of iterations method.

### 23.7.3 The Sign-Change-Ratio (SCR) Criterion

Further approximations to the cross entropy formula lead to the sign-change-ratio (SCR) criterion. A sign change is said to have occurred in bit position \( n \) between iterations \( m - 1 \) and \( m \) if \( \text{sign} \left[ \Lambda_{2e}^{[m]}(r_2(n)) \right] \neq \text{sign} \left[ \Lambda_{2e}^{[m-1]}(r_2(n)) \right] \). An indicator function for sign changes is

\[
s(n) = 0.5 \left| \text{sign} \left[ \Lambda_{2e}^{[m]}(r_2(n)) \right] - \text{sign} \left[ \Lambda_{2e}^{[m-1]}(r_2(n)) \right] \right| = \begin{cases} 
0 & \text{for no sign change} \\
1 & \text{for sign change}
\end{cases}
\]  

(23.59)

The number total number of sign changes in bit positions between the extrinsic LLR’s for Decoder 2 from iteration \( m - 1 \) to \( m \) is

\[
C(m) = \sum_{n=0}^{L_T-1} s(n)
\]  

(23.60)

Simulations have shown that the iterations can be stopped with little loss in performance when \( C(m)/L_T \) decreases into the range (0.005, 0.03) and performance equal to the CE criterion can be achieved by a threshold in this range. The computations required for the SCR criterion are much less than for the CE criterion for large \( L_T \).

### 23.7.4 The Hard-Decision-Aided (HDA) Criterion

Hard bit decisions are made after the final iteration from the signs of the complete LLR’s \( \Lambda_{n,2}[m](r_0, r_2) \). When convergence has been reached \( \text{sign} \Lambda_{n,2}^{[m-1]}(r_0, r_2) = \text{sign} \Lambda_{n,2}[m](r_0, r_2) \) for \( n = 0, \ldots, L_T - 1, \) that is, the bit decisions no longer change between successive iterations. The HDA rule is to store the hard decisions for successive iterations \( m - 1 \) and \( m \) and stop if the hard decisions do not change.

Simulations have shown that the HDA criterion has fewer iterations at low to medium SNR’s and small to medium interleaver lengths than the CE and SCR criteria with similar
bit error rate performance. However, at high SNR’s the CE and SCR criteria have less iterations.

23.8 Turbo Code Experiments

Perform the following tasks for turbo decoding:

1. Write a C program to implement the LTE rate $R = 1/3$ turbo encoder shown in Figure 23.3 on page 147. Do this for information block lengths of $K = 1008$ and 6144.
   
   (a) Simulate the random input information bits by using a 23-stage maximal length PN sequence generator with the connection polynomial $h(D) = 1 + D^{18} + D^{23}$.
   
   (b) Implement the $K = 1008$ and 6144 LTE interleavers defined on page 147. Prove that the two interleaving rules really work by writing a MATLAB program to generate the permutation of the integers $0, 1, \ldots, K-1$ for both cases and check that each integer appears only once in the permuted sequences. Hint: Use the MATLAB functions \texttt{unique()} and \texttt{length()}.
   
   (c) Use the rules on page 147 to drive the final state of each encoder to zero and generate six tail bits for each.
   
   (d) Save the information blocks to use for error checking after decoding.

2. Biphase modulate the encoder output bits. Assume the channel adds white Gaussian noise and the receiver uses a matched filter to detect the bits as discussed in Section 22.19. The desired noiseless output of the filter has the biphase values $\pm 1$. The noise component of the matched filter outputs is a zero-mean Gaussian random variable with variance $\sigma^2 = N_0/(2E_s) = N_0/(2RE_b)$ where $N_0$ is the two-sided white noise power spectral density, $E_s$ is the biphase symbol energy, $R$ is the code rate, and $E_b$ is the energy per transmitted information bit. The code rate is $R = 1/3$ in this case. In dB $E_b/N_0$ is
   
   $$\rho = 10 \log_{10} \frac{E_b}{N_0} \quad \text{so that} \quad \frac{E_b}{N_0} = 10^{\rho/10}$$

   and

   $$\sigma^2 = 1/(2R \times 10^{\rho/10})$$

   Simulate this noise for a desired $E_b/N_0$ and add it to the block of biphase modulated bit values.

3. Implement MAP turbo decoders for information block lengths of $K = 1008$ and 6144. The decoder programs should have a variable that sets the number of iterations. Make sure the decoders are working properly by checking that there are no decoding errors when $\sigma^2 = 0$, that is, in the case of no added noise.

4. Perform simulations to generate bit error rate vs. $E_b/N_0$ curves like in Figure 23.4 for both values of $K$. 
• Plot bit error probability on a logarithmic scale and $E_b/N_0$ in dB on a linear scale.

• Transmit multiple blocks of information bits and calculate the bit error rate as the observed number of bit errors divided by the total number of transmitted data bits. Transmit enough bits to get a reasonably stable estimate.

• Start with a large $E_b/N_0$ where no errors occur and then reduce it. Make sure to cover the “waterfall” region.

• Generate curves starting with one iteration. Then increase the number of iterations in steps until there is essentially no more improvement.

5. See if the a priori probabilities $P_{2}^{[m]}(x(n) = v)$ approach 0 and 1 as the iteration number $m$ increases.

6. Repeat the previous items for rate $R = 1/2$ codes created by puncturing the $R = 1/3$ code. In this case only one check bit is transmitted for each branch. The check bit is alternately selected from the two constituent encoders.
23.9 Appendix

MATLAB Code for Computing AWGN Channel Capacity

File CBPSK.m

% Compute the capacity for an AWGN channel with BPSK input
clear

N=20;
i=0;
for snrdb = -10:.01:10
    i=i+1;
    sigma = 10^(-snrdb/20);
    [X,W] = hermquad(N);
    s=0;
    for n=1:N
        s=s+W(n)*f(X(n),sigma);
    end
    SNR(i) = snrdb;
    C(i)= 1 - s;
    CU(i) = 0.5* log2(1+sigma^(-2)); % Capacity for unconstrained input
end

plot(SNR,CU,'k--', SNR,C,'k-')
legend('Unconstrained','Biphase','Location','Best')
grid ON
xlabel 'SNR in DB'
ylabel 'Capacity'

File f.m

function [fz] = f(z,sigma)
    fz = (log2(1+exp(-2*(sigma*sqrt(2)*z+1)/sigma^2)) + ...
    log2(1+exp(2*(sigma*sqrt(2)*z -1)/sigma^2)))/(2*sqrt(pi));
end

File hermquad.m

function [X,W] = hermquad(N)
% % [X W] = HERMQUAD(N)
%
Find the Gauss–Hermite abscissae and weights.

Arguments:
N - The number of abscissae and weights to return.

Return Values:
X - A column vector containing the abscissae.
W - A column vector containing the corresponding weights.

Gauss–Hermite quadrature approximates definite integrals of the form
\[ \int_{-\infty}^{\infty} dx W(x) f(x) \]
where
\[ W(x) = \exp(-x^2) \]
with the sum
\[ \sum_{n=1}^{N} w_{n} f(x_{n}) \]
This function returns the set of abscissae and weights
\[ \{x_{n}, w_{n}\}_{n=1}^{N} \]
for performing this calculation given N, the number of abscissae.
These abscissae correspond to the zeros of the Nth Hermite polynomial. It can be shown that such integration is exact when \( f(x) \) is a polynomial of maximum order 2N−1.
The procedure in this calculation is taken more or less directly from

@BOOK{ press-etal-1992a,
  AUTHOR  = {Press, William H. and Flannery, Brian P. and Teukolsky, Saul A. and Vetterling, William T. },
  ISBN    = {0521431085},
  MONTH   = {October},
  PUBLISHER = {{Cambridge University Press}},
  TITLE   = {Numerical Recipes in C : The Art of Scientific Computing},
  YEAR    = {1992} }
% }
%

% precision
EPS = 3.0e-14;

% 1/pi^1/4
PIM4 = 0.7511255444649425;

% maximum number of loops
MAXIT = 10;

% allocate the return values
X = zeros([N 1]);
W = zeros([N 1]);

for i=1:(N+1)/2
    % good guesses at initial values for specific roots
    if i == 1
        z = sqrt(2.0*N+1.0) - 1.85575*((2.0*N+1)^(-0.16667));
    elseif i == 2
        z = z - (1.14 * N^0.426 / z);
    elseif i == 3
        z = 1.86 * z - 0.86 * X(1);
    elseif i == 4
        z = 1.91 * z - 0.91 * X(2);
    else
        z = 2.0*z - X(i-2);
    end

    for iter=1:MAXIT+1
        p1 = PIM4;
        p2 = 0.0;

        for j=1:N
            p3 = p2;
            p2 = p1;
            p1 = z * sqrt(2.0/j) * p2 - sqrt((j-1.0)/j) * p3;
        end

    % the derivative
\[ pp = \sqrt{2.0N} \times p2; \]

\% newton step
\[ z1 = z; \]
\[ z = z1 - p1/pp; \]

\text{if abs}(z-z1) \leq \text{EPS}
\quad \text{break};
\text{end}
\text{end}

\text{if iter} == \text{MAXIT}+1
\quad \text{fprintf}('Too many iterations in hermquad.\n');
\text{end}

\[ X(i) = z; \]
\[ X(N+1-i) = -z; \]
\[ W(i) = 2.0/(pp\times pp); \]
\[ W(N+1-i) = W(i); \]

\text{end}
23.10 References


Turbo Codes
Chapter 24

Low-Density Parity-Check Codes

24.1 Introduction

Low-density parity-check codes (LDPC) were discovered and their properties extensively investigated by Robert Gallager during his doctoral research at MIT and his doctoral dissertation [3] was completed in 1960. His results were subsequently published in an IRE article [4] and an expanded version of the IRE article and his dissertation were published as an MIT Press monograph in 1963 [5]. While he proved that these codes had excellent performance, they were ignored and forgotten for many years because the hardware required to perform the iterative decoding algorithm to achieve near optimum performance was too complex and expensive at that time to be included in most communication devices. As you will see, the decoding algorithm iteratively computes bit probabilities or log likelihood functions and requires a computer with significant arithmetic capabilities.

LDPC codes were rediscovered by David MacKay around 1997. By the random coding arguments frequently used in Information Theory, he proved that LDPC codes can approach the Shannon’s capacity limit exponentially fast as the code block length increases. The codes have been included in several international standards such as IEEE 802.11n and ac WIFI, as optional codes, IEEE 802.16 which is known as WIMAX, IEEE 802.20 for mobile wireless internet access, IEEE 802.3 which defines protocols for wired Ethernet, and DBV-RS2 which stands for “Digital Video Broadcasting - Return Channel via Satellite.” LDPC codes have been chosen for many applications because the complexity of their decoding algorithms and error probability performance have beaten competing turbo codes. Another advantage of LDPC codes is that there are very few patent issues.

24.2 Definition of Low-Density Parity-Check Codes

In the broadest sense, a binary LDPC code is a linear block code with a check matrix $H$ that has a small percentage of 1’s. The check matrix is said to be sparse. Suppose $H$ has $M$ rows and $N$ columns with $N > M$, the rank of $H$ is $M$, and that $x = (x_1, x_2, \ldots, x_N)$ is a binary $N$-tuple. Then $x$ is a codeword if and only if $Hx^T = 0$ where $T$ denotes transpose.
and \( \mathbf{0} \) is a column vector of \( M \) zeros. The set of codewords is the null space of the rows of \( \mathbf{H} \). The 1’s in a row of \( \mathbf{H} \) specify the bits in a codeword that must add to 0 modulo 2 and this sum is called a parity check. Each of the \( M \) parity checks involves a small number of code bits since \( \mathbf{H} \) is sparse. This fact leads to relatively efficient decoding algorithms.

Gallager invented and analyzed what are now known as regular LDPC codes. The parity check matrix for a Gallager LDPC code has a small number, \( j \geq 3 \), of 1’s in each column and a small number, \( k \), of 1’s in each row and \( j < k \). If the code has block length \( N \), which is the number of columns of \( \mathbf{H} \), the code is called an \( (N, j, k) \) code. When the number, \( j \), of 1’s in a column is 2 the codes perform poorly.

The number of 1’s in \( \mathbf{H} \) can be computed as \((\text{number of 1's per row}) \times (\text{number of rows}) = kM \) or as \((\text{number of 1's per column}) \times (\text{number of 1's per column}) = jN \). Thus \( kM = jN \) and \( M/N = j/k \). The number of rows must be \( M = Nj/k \). When \( \mathbf{H} \) has rank \( M \), the number of check symbols in a codeword is \( M \) so the number of information symbols is \( K = N - M \) and the rate of the code is

\[
R = \frac{K}{N} = 1 - \frac{M}{N} = 1 - \frac{j}{k} \tag{24.1}
\]

The number of elements in \( \mathbf{H} \) is \( MN \), so the fraction of 1’s in \( \mathbf{H} \) is \( Mk/(NM) = k/N \). Therefore, when \( k \) is fixed, \( \mathbf{H} \) becomes increasingly sparse as \( N \) increases. Of course, the number or rows must increase with \( N \) since \( M = Nj/k \). The performance of a linear block code in terms of decoding error probability is strongly influenced by the number of low weight codewords. Remember that the number of codewords at distance \( d \) from each codeword is the number of codewords of weight \( d \). If there are few 1’s in each row of the parity check matrix, any codeword must have enough 1’s to match up with the 1’s in each row of \( \mathbf{H} \). Intuitively, this indicates that there are fewer low weight codewords in the LDPC code than in codes where no sparse parity check matrix exists.

Gallager presented a method for constructing parity check matrices for \( (N, j, k) \) LDPC codes. An example of a \( (20, 3, 4) \) is shown in Figure 24.1. This matrix actually has only 13 linearly independent rows, so codewords have \( 20 - 13 = 7 \) information symbols and 13 check symbols. The resulting code has rate \( R = 7/20 = 0.35 \).

Consider the upper five rows of the \( \mathbf{H} \) in Figure 24.1. The number of 1’s in each row is \( k = 4 \). Each successive row contains the four 1’s shifted four elements to the right resulting in a block length of \( N = 4 \times 5 = 20 \). In general, the first row would have a string of \( k \) 1’s starting at the left side followed by all 0’s to the right. Each successive row would have the \( k \) 1’s shifted \( k \) positions to the right from the previous row with 0’s in all the other elements. If the top block has \( M_1 \) rows, the block length must be \( N = M_1k \). Notice that the top block has a single 1 in each column and \( k \) 1’s in each row.

Suppose an \( \mathbf{H} \) with \( j \) 1’s in each column is desired. The columns of the top block of \( M_1 \) rows can be permuted in \( N! \) ways. Each permutation still contains \( k \) 1’s in each row and a single 1 in each column. Therefore, \( j - 1 \) of these permutations can be selected to append to the bottom of the top block to form a matrix with \( j \) 1’s in each column and \( k \) 1’s in each row. The resulting matrix will have \( N = M_1k \) rows and \( M = jM_1 \) rows. The resulting
matrix will not necessarily have full rank $M$. In that case the code will have a somewhat higher rate than the designed value.

The performance of the resulting code depends on the permutations selected and some are better than others. Gallager investigated the average properties of an ensemble of codes formed by randomly selecting the permuted blocks. He proved that performance is very likely to be good if the permutations are selected at random. He showed that the minimum distance between codewords on average will grow linearly with $N$, that is, $d_{\text{min}} \simeq N \delta_{jk}$ [5, pp. 17–18] for some fraction $\delta_{jk}$.

MacKay’s rediscovery of LDPC codes led to a large community of researchers intensely investigating them. Codes with a varying number of 1’s in each column and row were found that have better performance than the uniform codes. These are called non-uniform codes. LDPC codes that perform very close to Shannon’s capacity bound have been found.

### 24.2.1 Efficient Representation of a Sparse Matrix for Computer Storage

A sparse matrix can be efficiently described by lists of the positions of its 1’s. Let the positions of the 1’s in the $m$th row of $H$ be denoted by the set

$$\mathcal{N}_m = \{n : H_{mn} = 1\}$$  \hspace{1cm} (24.2)

Then the $m$th check, that is the $m$ element of $Hx^T$, is

$$z_m = \sum_{n \in \mathcal{N}_m} x_n$$  \hspace{1cm} (24.3)
It will be convenient for future equations to denote the set of elements in \( N_m \) with one of its elements, say \( n \), deleted as \( N_{mn} = N_m \setminus n \). The number of elements in \( N_m \) will be denoted by \( |N_m| \) as usual in set theory.

Let the row indexes of the 1’s in the \( n \)th column of \( H \) be denoted by the set

\[
M_n = \{ m : H_{mn} = 1 \} \tag{24.4}
\]

These are the indexes of the checks bit \( x_n \) participates in. Let \( M_{nm} = M_n \setminus m \) be the set of elements in \( M_n \) with \( m \) deleted.

The sets \( N_m \) and \( M_n \) should be considered as ordered lists with the \( i \)th elements denoted as \( N_m(i) \) and \( M_n(i) \).

### 24.3 Representing a Parity-Check Code by a Tanner Graph

The equation \( Hx^T = z \) where \( H \) is an \( M \times N \) parity check matrix for a binary linear code, \( x \) is an \( N \)-dimensional binary row vector, and \( z \) is an \( M \)-dimensional column vector, can be represented by a graph. The vector \( x \) is a codeword if and only if \( z = 0 \). An example for the simple \( H \) shown in Figure 24.2 is shown in Figure 24.3. In general, the graph has a vertical array of \( N \) nodes corresponding to the \( N \) elements of \( x \) on the left called bit nodes, and a vertical array of \( M \) nodes on the right corresponding to the \( M \) elements of \( z \) called check nodes on the right. A bit node is connected by branches to the \( M \) check nodes it affects. These are specified by the 1’s in the \( i \)th column of \( H \) for bit \( x_i \). The check node for \( z_i \) is connected by branches to the bits that enter into its sum which are specified by the 1’s in the \( i \)th row of \( H \). This type of graph is known as a Tanner graph [16]. A decoding algorithm that iterates between bit nodes and check nodes will be presented in future sections.

Tanner graphs have two types of nodes: bit nodes and check nodes. The nodes in one type only connect to nodes in the other type. That is, bit nodes only connect to check nodes and check nodes only connect to bit nodes. Graphs with this property are called bipartite graphs.

\[
H = \begin{bmatrix}
1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\
1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 
\end{bmatrix}
\]

Figure 24.2: A Simple \( H \) Example
24.3 Representing a Parity-Check Code by a Tanner Graph

24.3.1 Cycles in a Tanner Graph

Starting at a particular bit node, paths can be followed along connecting branches alternating from bit nodes to check nodes. Since there are a finite number of bit nodes, some path must eventually return to the original bit node. Such a path is called a cycle. The number of branches transversed in a cycle is called its length. The length of a shortest cycle contained in a graph is called the girth of a graph. The performance of iterative decoding algorithms that will be presented for LDPC codes depends on the length of the cycles. Cycles with small length degrade the performance.

The Tanner graph shown in Figure 24.3 has cycles of length 4 which is the minimum possible in a Tanner graph. One example is $x_1 \rightarrow z_1 \rightarrow x_7 \rightarrow z_5 \rightarrow x_1$. Whenever two different columns of $H$ each contain a pair of 1’s in the same pair of rows a cycle of length 4 exists. That is, the four 1’s are the corners of a rectangle connecting 1’s in $H$. 

---

Figure 24.3: Tanner Graph for the $H$ in Figure 24.2
24.4 Probabilities for Binary Phase Shift Keying (BPSK) Over an Additive, White, Gaussian Noise Channel

An important transmission model for LDPC codes is binary phase shift keying (BPSK) over an additive, white, Gaussian noise channel. Each binary input bit, \( x_n \), is converted to an analog level \( \hat{x}_n \) which can be \( A \) or \( -A \) with \( A > 0 \) according to the rule

\[
\hat{x}_n = (1 - 2x_n)A = \begin{cases} 
A & \text{for } x_n = 0 \\
-A & \text{for } x_n = 1 
\end{cases}
\]  

(24.5)

This assignment is sometimes reversed. The level, \( \hat{x}_n \), remains constant over the interval \( nT \leq t < (n+1)T \). The variable \( T \) is the bit period. The signal received during the time interval \([nT, (n+1)T)\) is

\[
r(t) = \hat{x}_n + v(t) \quad \text{for } nT \leq t < (n+1)T
\]  

(24.6)

where \( v(t) \) is white, Gaussian noise with two-sided power spectral density \( N_0/2 \) and autocorrelation function \( R(\tau) = N_0/2 \delta(\tau) \). The receiver applies this signal to a matched filter which computes

\[
r_n = \frac{1}{T} \int_{nT}^{(n+1)T} r(t) \, dt = x_n + \frac{1}{T} \int_{nT}^{(n+1)T} v(t) \, dt = \hat{x}_n + v_n
\]  

(24.7)

The term \( v_n \) is a zero-mean, Gaussian random variable with variance

\[
\sigma^2 = E\{v_n^2\} = E \left\{ \frac{1}{T} \int_{nT}^{(n+1)T} v(t) \, dt \cdot \frac{1}{T} \int_{nT}^{(n+1)T} v(s) \, ds \right\}
\]

\[
= \frac{1}{T^2} \int_{nT}^{(n+1)T} \int_{nT}^{(n+1)T} E\{v(t)v(s)\} \, dt \, ds = \frac{1}{T^2} \int_{nT}^{(n+1)T} \int_{nT}^{(n+1)T} R(t-s) \, dt \, ds
\]

\[
= \frac{1}{T^2} \int_{nT}^{(n+1)T} \int_{nT}^{(n+1)T} \frac{N_0}{2} \delta(t-s) \, dt \, ds = \frac{1}{T^2} \int_{nT}^{(n+1)T} \frac{N_0}{2} \, ds = \frac{N_0}{2T}
\]  

(24.8)

The probability density function (pdf) for \( r_n \) given \( \hat{x}_n = A \) is

\[
f(r_n | \hat{x}_n = A) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(r_n-A)^2}{2\sigma^2}}
\]  

(24.9)

and the pdf for \( r_n \) given \( \hat{x}_n = -A \) is

\[
f(r_n | \hat{x}_n = -A) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(r_n+A)^2}{2\sigma^2}}
\]  

(24.10)
The unconditional pdf for \( r_n \) is

\[
f(r_n) = P(\hat{x}_n = A) f(r_n| \hat{x}_n = A) + P(\hat{x}_n = -A) f(r_n| \hat{x}_n = -A) \tag{24.11}
\]

The probability that transmitted bit was \( x_n = 0 \) or, equivalently, \( \hat{x}_n = A \) given the observation \( r_n \) is

\[
P(\hat{x}_n = A| r_n) = \frac{P(\hat{x}_n = A) f(r_n| \hat{x}_n = A)}{f(r_n)} \tag{24.12}
\]

Combining (24.9), (24.10), and (24.11) and cancelling terms gives

\[
P(\hat{x}_n = A| r_n) = \frac{P(\hat{x}_n = A) e^{r_n A/\sigma^2}}{1 + P(\hat{x}_n = -A) e^{-2r_n A/\sigma^2}} \tag{24.13}
\]

and

\[
P(\hat{x}_n = -A| r_n) = 1 - P(\hat{x}_n = A| r_n) = \frac{P(\hat{x}_n = -A) e^{-2r_n A/\sigma^2}}{1 + P(\hat{x}_n = -A) e^{-2r_n A/\sigma^2}} \tag{24.14}
\]

A reasonable model for typical binary data communications is to assume that 1's and 0's occur with equal probability. Then \( P(\hat{x}_n = A) = P(\hat{x}_n = -A) = 0.5 \) and the previous two conditional probabilities reduce to

\[
P(\hat{x}_n = A| r_n) = \frac{1}{1 + e^{-2r_n A/\sigma^2}} \tag{24.15}
\]

and

\[
P(\hat{x}_n = -A| r_n) = \frac{e^{-2r_n A/\sigma^2}}{1 + e^{-2r_n A/\sigma^2}} \tag{24.16}
\]

### 24.4.1 Hard Bit Decisions Using a Log-Likelihood Ratio

Hard decisions can be made for each received bit by deciding \( A \) was transmitted if \( P(\hat{x}_n| r_n) > P(\hat{x}_n = -A| r_n) \) and \(-A\) was transmitted otherwise. This inequality can be tested using the following log-likelihood ratio (LLR):

\[
\lambda(r_n) = \log \frac{P(\hat{x}_n = A| r_n)}{P(\hat{x}_n = -A| r_n)} = \log \frac{P(\hat{x}_n = A)}{P(\hat{x}_n = -A)} + 2r_n \frac{A}{\sigma^2} \tag{24.17}
\]

The hard decision rule in terms of the LLR is: Decide \( A \) was transmitted if \( \lambda(r_n) > 0 \) and decide \(-A\) was transmitted otherwise.

In the case of equally likely bit probabilities the LLR reduces to the simple formula

\[
\lambda(r_n) = 2r_n \frac{A}{\sigma^2} \tag{24.18}
\]
Let $\hat{A}_n$ be the decision for the transmitted $\hat{x}_n$ given the observation $r_n$. Then the hard decision rule reduces to the intuitively obvious strategy

$$
\hat{A}_n = \begin{cases} 
A & \text{if } r_n > 0 \\
-A & \text{if } r_n \leq 0 
\end{cases} \quad (24.19)
$$

Let $\tilde{x}_n$ be the corresponding hard bit decision. Then

$$
\tilde{x}_n = \begin{cases} 
0 & \text{if } r_n > 0 \\
1 & \text{if } r_n \leq 0 
\end{cases} \quad (24.20)
$$

The probability of making a decision error can be shown to be

$$
P_b = \int_{-A/\sigma}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt \quad (24.21)
$$

The hard bit decisions can then be used in a decoding scheme which does not use the reliability information contained in the conditional received bit probabilities or LLR’s. Discarding the reliability information causes a significant degradation in the performance of the code.

### 24.5 Bit Flipping Decoding

Gallager [5, p. 39] suggested a simple decoding scheme applicable to the binary symmetric channel (BSC). The inputs to the BSC are the binary data bits. The channel outputs are the received signals quantized to binary values and any probabilistic reliability information is discarded. These quantized values are called the hard decisions. The decoding scheme is suboptimal and performs well below channel capacity.

The decoder first computes all the $M$ parity checks using the $N$ hard bit decisions. Then for each hard bit decision it counts the number of the $j$ checks on that bit that fail, i.e., are 1. Any bit for which more than a certain fraction of the associated checks fail is complemented. For example, the fraction could be a majority. The checks are then recomputed using the modified hard decisions and the process is repeated until all the parity checks are satisfied, i.e., are 0.

If the BSC crossover bit error probability is small and the number of bits, $j$, in a parity check is small, the number of errors in the bits participating in the check is very most likely to be zero or one. If most of the checks associated with a bit fail, this strongly suggests that the bit is in error. For example, suppose $x_1$ for the code in Figure 24.1 is received in error and there are no other hard decision errors. Then checks $z_1$, $z_6$, and $z_{11}$ would fail, strongly indicating that the hard decision for $x_1$ is incorrect. From $H$ it can be seen that no more than one check on any other bit would fail, so those bits would not be complemented and the single bit error would be corrected.

The bit flipping algorithm can sometimes correct more than one error. Insight into how this works can be seen by expanding the Tanner graph into a tree as shown in Figure 24.4.
for the $H$ of Figure 24.1. This is a code with three checks on each bit and four bits used in each check. Consider starting the tree with the code bit $x_1$. This bit is connected to the three checks $z_1$, $z_6$, and $z_{11}$. Each of these checks is connected to three bits in addition to $x_1$ as shown in the tree. These checks and the bits they are connected to were called Tier 1 by Gallager. Each bit at Tier 1 is connected to two checks one level up as well as the single check one level down for a total of three checks. The two checks one level up are connected to three additional bits forming Tier 2. This expansion can be continued as far as desired. Eventually a previously entered bit will be arrived at corresponding to a loop in the Tanner graph. The process of expanding the Tanner graph into a tree for a particular bit involves generating all the paths from an initial bit node at the root of the tree to check nodes and alternating back and forth between bit and check nodes.

A bit in Tier 1 can possibly be corrected by applying the bit flipping algorithm using the bits and checks at Tier 2. Then the bit at the base of the tree can possibly be corrected by using the corrected bits and checks at Tier 1. In general, the bit flipping algorithm would be started at the highest tier in the tree and proceed tier by tier down to the bit at the root. Gallager [LDPCGallager, p. 40] says “Thus digits and parity-check equations can aid in decoding a digit seemingly unconnected with them.”

![Figure 24.4: Expanding the Tanner Graph into a Tree for the $H$ in Figure 24.1](image-url)
24.6 Three Derivations of the Probability of an Even Number of 1’s in a Random Binary Vector

The formulas derived in this section will be used in following sections. Let \((x_1, \ldots, x_N)\) be a vector of \(N\) independent binary random variables with \(P(x_n = 1) = p_n\) and \(P(x_n = 0) = 1 - p_n = q_n\). It will be shown three different ways below that the probability the vector contains an even number of 1’s is

\[
P_{\text{ev},N} = \frac{1}{2} + \frac{1}{2} \prod_{n=1}^{N} (1 - 2p_n) = \frac{1}{2} + \frac{1}{2} \prod_{n=1}^{N} (q_n - p_n)
\] (24.22)

The probability it contains an odd number of 1’s is

\[
P_{\text{od},N} = 1 - P_{\text{ev}} = \frac{1}{2} - \frac{1}{2} \prod_{n=1}^{N} (1 - 2p_n) = \frac{1}{2} - \frac{1}{2} \prod_{n=1}^{N} (q_n - p_n)
\] (24.23)

A vector with an even number of 1’s is said to have “even parity” and \(\sum_{n=1}^{N} x_n = 0\) where modulo 2 addition is used. Similarly, a vector with an odd number of 1’s is said to have “odd parity” and the modulo 2 sum of its components is 1.

24.6.1 Gallager’s Derivation

The method Gallager [5, p. 41] used to derive the formula for the probability of an even number of 1’s in a binary random vector is presented in this section. Consider the product

\[
f(t) = \prod_{n=1}^{N} (1 - p_n + p_n t) = f_0 + f_1 t + \cdots + f_k t^k + \cdots + f_N t^N
\] (24.24)

What is the coefficient of \(t^k\)? Each contribution is the product of \(p_n t\) terms from \(k\) factors and \(1 - p_n\) terms from the remaining \(N - k\) factors. As one contribution, suppose the \(p_n t\) terms are selected for \(n \in \mathcal{N} = \{n_1, n_2, \ldots, n_k\}\). Let \(\mathcal{N}^c\) be the set of remaining \(N - k\) values of \(n\). Then its contribution to the coefficient of \(t^k\) is

\[
\prod_{n \in \mathcal{N}} p_n \prod_{n \in \mathcal{N}^c} (1 - p_n)
\]

This is the probability of a vector with 1’s in the \(k\) positions given by \(\mathcal{N}\) and 0’s in the remaining positions. There are \(\binom{N}{k}\) ways of selecting the \(k\) positions for the 1’s. The coefficient, \(f_k\), of \(t^k\) is the sum of all these terms, so it is the probability of exactly \(k\) 1’s in the \(N\)-dimensional binary vector. The sum of the coefficients for \(k = 0, \ldots, N\) is the probability of any number of 1’s in the vector and must add up to 1. This is why \(f(1) = 1\).

The polynomial expansion for \(f(-t)\) has the same coefficients as for \(f(t)\) except the coefficients for odd powers of \(t\) are the negatives of those in the expansion for \(f(t)\). The
probability of an even number of 1’s in the vector is the sum of the coefficients of the even powers of $t$. If $f(t)$ and $f(-t)$ are added, the coefficients of the odd powers of $t$ cancel and the coefficients of the even powers become double. Therefore

$$P_{ev,N} = \frac{f(t) + f(-t)}{2} = \prod_{n=1}^{N}(1 - p_n + p_n) + \prod_{n=1}^{N}(1 - p_n - p_n)$$

$$= \frac{1}{2} + \frac{1}{2} \prod_{n=1}^{N}(1 - 2p_n)$$

(24.25)

## 24.6.2 Proof of the Formula by Mathematical Induction

Fan [1, p. 53] proves the formula is correct by mathematical induction. For $N = 1$ the number of 1’s in the one-dimensional vector is even when $x_1 = 0$ so $P_{ev,1} = q_1$. Equation (24.22) reduces to this for $N = 1$. For $N = 2$ the vectors containing an even number of 1’s are $(0, 0)$ and $(1, 1)$. Therefore $P_{ev,2} = q_1q_2 + p_1p_2$ and a little algebra shows (24.22) is equal to this. Now assume $P_{ev,N}$ is given by (24.22). Let the $N$-dimensional vector be increased to an $N + 1$-dimensional vector by appending a component $x_{N+1}$. The new vector will have an even number of 1’s if the original $N$-dimensional vector has an even number of 1’s and $x_{N+1} = 0$ or the original vector has an odd number of 1’s and $x_{N+1} = 1$. Therefore,

$$P_{ev,N+1} = P_{ev,N}q_{N+1} + P_{od,N}p_{N+1}$$

$$= \left(1 + \prod_{n=1}^{N}(q_n - p_n)\right)q_{N+1} + \left(1 - \prod_{n=1}^{N}(q_n - p_n)\right)p_{N+1}$$

$$= \frac{(q_{N+1} + p_{N+1}) + (q_{N+1} - p_{N+1}) \prod_{n=1}^{N}(q_n - p_n)}{2}$$

$$= \frac{1 + \prod_{n=1}^{N+1}(q_n - p_n)}{2}$$

(24.26)

Thus the formula has been proven to be true by induction.

## 24.6.3 Derivation by Propagating Probabilities Through a Trellis

Moon [11,p. 646] shows how computation of $P_{ev,N}$ and $P_{od,N}$ by induction can be viewed as propagating probabilities through a flowgraph that looks like a two-state trellis. The modulo two sum $\sum_{n=1}^{M} x_n$ can have the value 0 for even parity or 1 for odd parity of the binary $k$-tuple $(x_1, \ldots, x_k)$. A trellis for computing these probabilities is shown in Figure 24.5. Any path that starts at the node labeled $P_{ev,0}$ at the upper left of the trellis and ends up at a node in the upper part of the trellis has even parity. The value assigned to a node at depth $k$ in the upper part of the trellis is $P_{ev,k}$ which is the sum of the probabilities of all paths from the start of the trellis to that node. Similarly, paths from the start of the trellis to a bottom node have odd parity. The value assigned to a node at depth $k$ in the bottom part of the trellis is $P_{od,k}$. Horizontal branches connect nodes at depth $k$ to ones at depth
$k + 1$ if $x_{k+1} = 0$ because this leaves the parity of the sum up to depth $k + 1$ unchanged. Diagonal branches connect nodes at depth $k$ to ones at depth $k + 1$ if $x_{k+1} = 1$ because this complements the parity of the sum.

The parity at an upper node at depth $k + 1$ is even if the parity at the upper node at depth $k$ is even and $x_{k+1} = 0$ or if the parity at the bottom node at depth $k$ is odd and $x_{k+1} = 1$. Therefore,

$$P_{ev,k+1} = P_{ev,k}q_{k+1} + P_{od,k}p_{k+1}$$ (24.27)

The parity at a bottom node at depth $k + 1$ is odd if the parity at the upper node at depth $k$ is even and $x_{k+1} = 1$ or if the parity at the bottom node at depth $k$ is odd and $x_{k+1} = 0$. Therefore,

$$P_{od,k+1} = P_{ev,k}p_{k+1} + P_{od,k}q_{k+1}$$ (24.28)

The initial condition is $P_{ev,0} = 1$. Then $P_{ev,1} = q_1$, $P_{od,1} = p_1$ and $P_{ev,1} - P_{od,1} = q_1 - p_1$. Subtracting (24.27) from (24.28) gives

$$P_{ev,k+1} - P_{od,k+1} = P_{ev,k}(q_{k+1} - p_{k+1}) - P_{od,k}(q_{k+1} - p_{k+1}) = (q_{k+1} - p_{k+1})(P_{ev,k} - P_{od,k})$$ (24.29)

By iteration from 1 to $k$, it follows that

$$P_{ev,k} - P_{od,k} = \prod_{n=1}^{k}(q_n - p_n)$$ (24.30)

Replacing $P_{od,k}$ by $1 - P_{ev,k}$ the previous equation can be reduced to

$$P_{ev,k} = \frac{1}{2} + \frac{1}{2}\prod_{n=1}^{k}(q_n - p_n)$$ (24.31)

and subtracting this from 1 gives

$$P_{od,k} = \frac{1}{2} - \frac{1}{2}\prod_{n=1}^{k}(q_n - p_n)$$ (24.32)
24.7 The Log Likelihood Ratio (LLR) for $P_{od,N}$ and $P_{ev,N}$

A formula for the log likelihood ratio (LLR) of the probabilities that a sum of independent binary random variables has odd parity or even parity will be derived in this section in terms of the LLR’s of the individual random variables in the sum. If has been found that computation using LLR’s can be more efficient than by using the probabilities. Let the LLR for the variable $x_n$ be defined as

$$\lambda_n = \log \frac{P(x_n = 1)}{P(x_n = 0)} = \log \frac{p_n}{q_n}$$ (24.33)

The probabilities $p_n$ and $q_n = 1 - p_n$ can be found from the LLR as follows:

$$\frac{p_n}{q_n} = e^{\lambda_n} \quad \text{so} \quad p_n = q_n e^{\lambda_n} = (1 - p_n) e^{\lambda_n}$$ (24.34)

Solving for $p_n$ gives

$$p_n = \frac{e^{\lambda_n}}{1 + e^{\lambda_n}}$$ (24.35)

and

$$q_n = 1 - p_n = \frac{1}{1 + e^{\lambda_n}}$$ (24.36)

The difference of these two probabilities is

$$q_n - p_n = \frac{1 - e^{\lambda_n}}{1 + e^{\lambda_n}} = \frac{e^{-\lambda_n/2} - e^{\lambda_n/2}}{e^{-\lambda_n/2} + e^{\lambda_n/2}} = -\tanh(\frac{\lambda_n}{2})$$ (24.37)

A formula for the inverse hyperbolic tangent that will be used in the formula for the LLR of the parity probability will now be derived. Let $y = \tanh(z/2)$ so that $z = 2 \tanh^{-1} y$. Then

$$y = \tanh(z/2) = \frac{e^{z/2} - e^{-z/2}}{e^{z/2} + e^{-z/2}} = \frac{e^z - 1}{e^z + 1}$$ (24.38)

Cross multiplying and solving for $e^z$ gives

$$ye^z + y = e^z - 1$$
$$e^z(1 - y) = 1 + y$$
$$e^z = \frac{1 + y}{1 - y}$$ (24.39)

Taking the log of both sides gives the following desired formula

$$z = \log \frac{1 + y}{1 - y} = 2 \tanh^{-1} y$$ (24.40)

Dividing (24.23) by (24.22) the LLR for $P_{od,N}$ is found to be

$$\Lambda = \log \frac{P_{od,N}}{P_{ev,N}} = \log \frac{1 - \prod_{n=1}^{N} (q_n - p_n)}{1 + \prod_{n=1}^{N} (q_n - p_n)}$$ (24.41)
Replacing $q_n - p_n$ by (24.37) the LLR becomes

$$
\Lambda = \log \frac{1 - (-1)^N \prod_{n=1}^{N} \tanh(\lambda_n/2)}{1 + (-1)^N \prod_{n=1}^{N} \tanh(\lambda_n/2)} = \log \frac{1 + (-1)^N \prod_{n=1}^{N} \tanh(\lambda_n/2)}{1 - (-1)^N \prod_{n=1}^{N} \tanh(\lambda_n/2)}
$$

(24.42)

Using (24.40) with $y = (-1)^N \prod_{n=1}^{N} \tanh(\lambda_n/2)$ and the fact that $\tanh^{-1}(y)$ is an odd function gives the following desired formula:

$$
\Lambda = (-1)^{N+1} 2 \tanh^{-1} \left( \prod_{n=1}^{N} \tanh(\lambda_n/2) \right)
$$

(24.43)

### 24.7.1 Converting the Product Into a Sum

Computing products took much more time than computing sums quite a few years in the past. Today current DSP’s have high speed integer and floating point hardware multipliers so addition and multiplication take the same time. For historical sake it will now be shown how Gallager replaced the products by sums. Simple approximations to the resulting formula will be presented.

Terms in the product in (24.43) can be represented by the product of a sign and a magnitude as follows:

$$
\tanh(\lambda_n/2) = \text{sgn}(\lambda_n) e^{\log |\tanh(\lambda_n/2)|} = \text{sgn}(\lambda_n) e^{\log \tanh(|\lambda_n|/2)}
$$

(24.44)

Thus

$$
\prod_{n=1}^{N} \tanh(\lambda_n/2) = \left( \prod_{n=1}^{N} \text{sgn}(\lambda_n) \right) \exp \left( \sum_{n=1}^{N} \log \tanh(|\lambda_n|/2) \right)
$$

$$
= \left( \prod_{n=1}^{N} \text{sgn}(\lambda_n) \right) \exp \left( - \sum_{n=1}^{N} - \log \tanh(|\lambda_n|/2) \right)
$$

(24.45)

Gallager [5] realized that the terms in the sum in the exponential can be expressed using the function

$$
\Psi(x) = -\log \left( \tanh \frac{x}{2} \right) = \log \frac{1 + e^{-x}}{1 - e^{-x}} = \log \frac{e^x + 1}{e^x - 1} \quad \text{for} \quad x > 0
$$

(24.46)

This function is plotted in Figure 24.6. It has the curious property of being its own inverse, that is, $\Psi(\Psi(x)) = x$. The following sequence of equations prove this property.

$$
\Psi(\Psi(x)) = \log \frac{e^{\Psi(x)} + 1}{e^{\Psi(x)} - 1} = \log \frac{e^{\log \frac{e^x + 1}{e^x - 1} + 1}}{e^{\log \frac{e^x + 1}{e^x - 1}} - 1} = \log \frac{e^x + 1}{e^x - 1} + 1
$$

$$
= \log \left( \frac{(e^x + 1)(e^x - 1)}{(e^x + 1) + (e^x - 1)} \right) = \log e^x = x
$$

(24.47)
A relationship that will be used below will now be derived. Let \( y = e^{-x} \) in \((24.40)\). Then

\[
2 \tanh^{-1} e^{-x} = \log \frac{1 + e^{-x}}{1 - e^{-x}}
\]

but according to \((24.46)\) this is \(\Psi(x)\) so

\[
\Psi(x) = 2 \tanh^{-1}(e^{-x})
\]

Using the definition for \(\Psi(x)\) given by \((24.46)\) in \((24.45)\) gives

\[
\prod_{n=1}^{N} \tanh(\lambda_n/2) = \left( \prod_{n=1}^{N} \text{sgn}(\lambda_n) \right) \exp \left( -\sum_{n=1}^{N} \Psi(|\lambda_n|) \right)
\]

Substituting \((24.50)\) into \((24.43)\) gives

\[
\Lambda = (-1)^{N+1} 2 \tanh^{-1} \left[ \left( \prod_{n=1}^{N} \text{sgn}(\lambda_n) \right) \exp \left( -\sum_{n=1}^{N} \Psi(|\lambda_n|) \right) \right]
\]

Since \(\tanh^{-1}(x)\) is an odd function

\[
\Lambda = (-1)^{N+1} \left( \prod_{n=1}^{N} \text{sgn}(\lambda_n) \right) 2 \tanh^{-1} \left[ \exp \left( -\sum_{n=1}^{N} \Psi(|\lambda_n|) \right) \right]
\]
Letting \( x = \sum_{n=1}^{N} \Psi(|\lambda_n|) \) in (24.49) the parity LLR can be compactly expressed as

\[
\Lambda = (-1)^{N+1} \left( \prod_{n=1}^{N} \text{sgn}(\lambda_n) \right) \Psi \left( \sum_{n=1}^{N} \Psi(|\lambda_n|) \right)
\]

\[(24.53)\]

### 24.7.2 An Approximation to the Parity LLR

The large values of \( \Psi(|\lambda_n|) \) are the major contributors to the sum in (24.53). If there is a single large value the sum will be large, the other values will not matter, and \( \Psi \left( \sum_{n=1}^{N} \Psi(|\lambda_n|) \right) \) will be small as can be seen from Figure 24.6. The largest value of \( \Psi(|\lambda_n|) \) is caused by the smallest value of \(|\lambda_n|\) and the sum can be approximated by the one term

\[
\Psi \left( \min_{n \in \{1,\ldots,N\}} |\lambda_n| \right)
\]

Thus \( \Lambda \) can be approximated by

\[
\Lambda \approx (-1)^{N+1} \left( \prod_{n=1}^{N} \text{sgn}(\lambda_n) \right) \Psi \left( \Psi \left( \min_{n \in \{1,\ldots,N\}} |\lambda_n| \right) \right)
\]

\[(24.54)\]

but according to (24.47) \( \Psi(x) \) is its own inverse, so the approximation reduces to

\[
\Lambda \approx (-1)^{N+1} \left( \prod_{n=1}^{N} \text{sgn}(\lambda_n) \right) \min_{n \in \{1,\ldots,N\}} |\lambda_n|
\]

\[(24.55)\]

This approximation requires much less computation than the exact formula. Simulations by Fossorier [2] and others have shown this approximation causes only a small degradation in the performance of the LDPC iterative decoding algorithm.

### 24.8 Iterative Decoding Using Probabilities

Consider an arbitrary block code of length \( N \) and codewords \( x \in C \). The decision rule that minimizes the probability of decoding to an incorrect codeword given the received \( n \)-tuple \( r \) is to select the codeword \( x \) that maximizes \( P(x|r) \). For the BPSK over an additive, white, Gaussian noise channel presented in Section 24.4, this rule can be reduced to selecting the codeword \( x \) that minimizes \( \sum_{i=1}^{N} (r_i - \hat{x}_i)^2 \). Performing this search for a long code with \( 2^K \) codewords requires extensive computation and is usually impractical. If the minimum Hamming distance between codewords is \( d_{\text{min}} \), decoding to the wrong codeword will introduce at least \( d_{\text{min}} \) bit errors in the decoded block. Exact formulas for the bit-error probability are difficult to determine and depend on the exact structure of the code. However, the bit-error probability will be small if the probability of decoding to an incorrect codeword is small.

Gallager introduced a suboptimal algorithm for decoding LDPC codes. Rather than computing the probability an entire codeword was transmitted given the received word \( r \),
the algorithm computes in \( m \) iterations the probability that an individual transmitted bit \( x_n \) was 1 (or equivalently 0) conditioned on received symbols out to the \( m \)th tier in the tree obtained from the Tanner graph with \( x_n \) as the root. It is assumed that there are no cycles in the tree. The algorithm starts at the leaves (code bits) in the \( m \)th tier and assigns probabilities \( P(x_i = b | r_i) \) with \( b = 0 \) or 1 to the leaves. Probabilities for code bits in each lower tier are computed iteratively until the tree root \( x_n \) is reached in \( m \) iterations. This is done for each of the code bits.

The probability formulas will now be derived for Tier 1. They can be adapted in an obvious way for the higher tiers and iteration process. Consider the tree root \( x_n \), the \( j \) checks involving \( x_n \), and the \( j(k-1) \) additional bits connected to these checks. These checks and the additional bits form Tier 1. Let \( S_n \) be the event that \( x_n \) and the Tier 1 bits satisfy the \( j \) checks on \( x_n \). Gallager calls the ensemble of vectors of length \( 1 + j(k-1) \) consisting of \( x_n \) and the Tier 1 bits a subcode. Let the desired conditional bit probability be

\[
q_n(b) = P(x_n = b | r, S_n) \quad \text{for} \quad b \in \{0, 1\}
\]  

(24.56)

Using the notation of (24.4)

\[
q_n(b) = P(x_n = b | r, \{z_m = 0, m \in M_n\}) \quad \text{for} \quad b \in \{0, 1\}
\]  

(24.57)

The following slightly different probability will also be used:

\[
q_{mn}(b) = P(x_n = b | r, \{z_{m'} = 0, m' \in M_{nm}\}) \quad \text{for} \quad b \in \{0, 1\}
\]  

(24.58)

The one check, \( z_m \), is not included in the conditions for this probability.

Another set of probabilities that will be required are the probabilities that a check \( z_m \) including bit \( x_n \) is satisfied given \( x_n \) and the received vector \( r \). Let this probability be

\[
r_{mn}(b) = P(z_m = 0 | x_n = b, r) \quad \text{for} \quad b \in \{0, 1\}
\]  

(24.59)

The probabilities \( q_{mn}(b) \) and \( r_{mn}(b) \) are computed only for the positions \((m, n)\) in \( H \) where the 1’s occur. The algorithm to be presented uses probability information about the bits to compute probability information, \( r_{mn}(b) \), for the checks. The check probabilities are then used to compute the bit probabilities \( q_{mn}(b) \). The algorithm alternates between computing the check and bit probabilities until all the checks are satisfied or a predefined number of iterations is reached. If the checks do not become satisfied by the time the iteration limit is reached, the decoder reports that a decoding error has occurred. This is a possible advantage over turbo decoding which is not constrained to decode to valid codewords.

### 24.8.1 Computing the Conditional Bit Probabilities

Consider bit \( x_n \) and the Tier 1 checks and bits in the tree growing from it. It will be assumed that the Tier 1 bits are distinct and independent. Then the \( j \) Tier 1 checks are statistically
independent. The channel will also be assumed to be memoryless. The conditional bit probabilities for \( x_n \) are

\[
q_n(b) = \frac{P(x_n = b \mid r, \{z_m = 0, m \in \mathcal{M}_n\})}{P(\{z_m = 0, m \in \mathcal{M}_n\} \mid r)} \frac{P(x_n = b, \{z_m = 0, m \in \mathcal{M}_n\} \mid x_n = b, r)}{P(\{z_m = 0, m \in \mathcal{M}_n\} \mid r)}
\]

(24.60)

Using the memoryless channel property and the independence of the checks, \( q_n(b) \) can be further expressed as

\[
q_n(b) = \frac{P(x_n = b \mid r_n) \prod_{m \in \mathcal{M}_n} P(z_m = 0 \mid x_n = b, r)}{\sum_{i=0}^{1} P(x_n = i \mid r_n) \prod_{m \in \mathcal{M}_n} P(z_m = 0 \mid x_n = i, r)}
\]

(24.61)

The denominator can be computed by summing the numerator for \( b = 0 \) and \( 1 \), so

\[
q_n(b) = \frac{P(x_n = b \mid r_n) \prod_{m \in \mathcal{M}_n} P(z_m = 0 \mid x_n = b, r)}{\sum_{i=0}^{1} P(x_n = i \mid r_n) \prod_{m \in \mathcal{M}_n} P(z_m = 0 \mid x_n = i, r)}
\]

(24.62)

The denominator is a constant not involving \( b \) and can considered to be a normalizing factor to make \( q_n(b) \) a probability. To simplify the notation define \( \alpha_n \) as

\[
\alpha_n = \frac{1}{P(\{z_m = 0, m \in \mathcal{M}_n\} \mid r)} = \frac{1}{\sum_{i=0}^{1} P(x_n = i \mid r_n) \prod_{m \in \mathcal{M}_n} P(z_m = 0 \mid x_n = i, r)}
\]

(24.63)

Then

\[
q_n(b) = \alpha_n P(x_n = b \mid r_n) \prod_{m \in \mathcal{M}_n} P(z_m = 0 \mid x_n = b, r)
\]

(24.64)

Using the definition of (24.59) \( q_n(b) \) is

\[
q_n(b) = \alpha_n P(x_n = b \mid r_n) \prod_{m \in \mathcal{M}_n} r_{mn}(b) \text{ for } b \in \{0, 1\}
\]

(24.65)

A formula for computing \( r_{mn}(b) \) will be derived shortly.

A bit in Tier 1 is connected to \( j - 1 \) checks in Tier 2 as well as to the one check in Tier 1 connected to it and the tree root bit. The conditional probability for a bit in Tier 1 can be computed from these \( j - 1 \) checks in Tier 2 and the Tier 2 bits connected to these checks. Again it will be assumed that the Tier 2 bits are distinct and independent so the Tier 2 checks are independent given the Tier 1 bit connect to them. Let \( n' \) be the index of a bit in Tier 1 connected to the check \( z_m \) in Tier 1. Let

\[
q_{mn'} = P(x_{n'} = b \mid \text{all checks involving } x_{n'} \text{ except for } z_m \text{ are satisfied, } r) = P(x_{n'} = b \mid \{z_m' = 0, m' \in \mathcal{M}_{n'm}\}, r)
\]

(24.66)
Using the same reasoning as in the derivation of (24.65) it follows that

\[ q_{mn'}(b) = \alpha_{mn'} P(x_{n'} = b \mid r_{n'}) \prod_{m' \in \mathcal{M}_{n'm}} r_{m'n'}(b) \] (24.67)

Once the Tier 1 bit probabilities are computed by (24.67), the probabilities for the root bits can be computed by (24.65).

The \( j - 1 \) terms in the product in (24.67) are computed at the locations \( m' \) down column \( n' \) of \( H \) where there is a 1. Therefore, this has been called the vertical step of the decoding algorithm. There are \( 2^j N \) values of \( q_{mn'}(0) \) and \( q_{mn'}(1) \) that must be computed and each requires \( O(j) \) operations, so this step has complexity \( O(N) \).

### 24.8.2 Computing the Conditional Check Probabilities

Computing \( q_{mn'}(b) \) by (24.67) requires knowing

\[ r_{m'n'}(b) = P(z_{m'} = 0 \mid x_{n'} = b, r) \quad \text{for} \quad b \in \{0, 1\}, m' \in \mathcal{M}_{n'm} \]

These probabilities can be computed using (24.22) and (24.23). In the case where \( b = 0 \)

\[
\begin{align*}
r_{m'n'}(0) & = P(z_{m'} = 0 \mid x_{n'} = 0, r) \\
& = P(\text{even number of 1’s in the bits checked by } z_{m'} \text{ other than } x_{n'} \mid r) \\
& = \frac{1}{2} + \frac{1}{2} \prod_{n \in \mathcal{N}_{m'n'}} \left[ 1 - 2P(x_n = 1 \mid r_n) \right] \\
& = \frac{1}{2} + \frac{1}{2} \prod_{n \in \mathcal{N}_{m'n'}} \left[ P(x_n = 0 \mid r_n) - P(x_n = 1 \mid r_n) \right] \\
& = 1 - r_{m'n'}(0) = \frac{1}{2} - \frac{1}{2} \prod_{n \in \mathcal{N}_{m'n'}} \left[ 1 - 2P(x_n = 1 \mid r_n) \right] \\
& = \frac{1}{2} - \frac{1}{2} \prod_{n \in \mathcal{N}_{m'n'}} \left[ P(x_n = 0 \mid r_n) - P(x_n = 1 \mid r_n) \right]
\end{align*}
\] (24.68)

and

\[
\begin{align*}
r_{m'n'}(1) & = 1 - r_{m'n'}(0) = \frac{1}{2} - \frac{1}{2} \prod_{n \in \mathcal{N}_{m'n'}} \left[ 1 - 2P(x_n = 1 \mid r_n) \right] \\
& = \frac{1}{2} - \frac{1}{2} \prod_{n \in \mathcal{N}_{m'n'}} \left[ P(x_n = 0 \mid r_n) - P(x_n = 1 \mid r_n) \right]
\end{align*}
\] (24.70)

To simplify future notation, let

\[
\delta r_{m'n'} = r_{m'n'}(0) - r_{m'n'}(1) = \prod_{n \in \mathcal{N}_{m'n'}} \left[ P(x_n = 0 \mid r_n) - P(x_n = 1 \mid r_n) \right]
\] (24.72)

and

\[
\delta q_{m'n'} = P(x_n = 0 \mid r_n) - P(x_n = 1 \mid r_n) \quad \text{for} \quad n \in \mathcal{N}_{m'n'}
\] (24.73)
Then
\[ \delta r_{m'n'} = \prod_{n \in N_{m'n'}} \delta q_{m'n} \]
so
\[ r_{m'n'}(0) = (1 + \delta r_{m'n'})/2 \quad \text{and} \quad r_{m'n'}(1) = (1 - \delta r_{m'n'})/2 \]

The terms in the product for \( \delta r_{m'n'} \) are computed at the 1’s in row \( m' \) of \( H \). Therefore, this computation is sometimes called the horizontal step of the algorithm.

24.8.3 Actual Implementation of the Iterative Algorithm

If the graph grown from a root code bit were actually a tree with distinct independent bits at each tier, the probabilities that the root bit is a 0 or 1 given the observed data could be computed exactly by starting at the leaves and iteratively working tier by tier down to the root. However, a tier grown from any bit in the Tanner graph must eventually contain a bit in a previous tier resulting in a cycle and violation of the independence assumptions. However, it has been found by many simulations that the iterative algorithm presented below works well for good codes even with the failure of the independence property.

The algorithm is started by setting all the conditional bit probabilities to \( P(x_n = b \mid r_n) \). Then these bit probabilities are updated by performing the horizontal and vertical steps. The horizontal and vertical steps are repeated using the updated bit probabilities from the previous vertical step. A stopping criterion can be implemented by adjusting the outputs of the vertical step to compute \( q_n(b) \) by (24.62) and making hard decisions on the code bits, deciding \( x_n = 1 \) if \( q_n(1) > 0.5 \) or \( x_n = 0 \) otherwise. The resulting vector can be tested to see if all the parity checks are satisfied, that is, if \( Hx^T = 0 \). The iterations are terminated when all the checks are satisfied. If they are not, another iteration is performed. A maximum number of allowed iterations is set. If all the checks are not satisfied by the maximum number of iterations, a decoding failure is declared.

Pseudo Code for the LDPC Decoding Algorithm Using Probabilities

1. **Input**
   - \( H \), the maximum number of iterations \( L \), and the a posteriori received bit probabilities \( p_n(b) = P(x_n = b \mid r_n) \) for \( b = 0 \) and 1, and \( n = 1, \ldots, N \).

2. **Initialization**
   - Set \( q_{mn}(b) = P(x_n = b \mid r_n) \) for all \( (m, n) \) with \( H_{mn} = 1 \), and \( b = 0, 1 \).

3. **The Horizontal Step**
   - For each \( (m, n) \) with \( H_{mn} = 1 \)
     - (a) Compute \( \delta q_{mn} = q_{mn}(0) - q_{mn}(1) = 1 - 2q_{mn}(1) \)
     - (b) Compute
       \[ \delta r_{mn} = \prod_{n' \in N_{mn}} \delta q_{mn'} \]
(c) Compute
\[ r_{mn}(1) = (1 - \delta r_{mn})/2 \quad \text{and} \quad r_{mn}(0) = (1 + \delta r_{mn})/2 \] (24.77)

4. The Vertical Step

(a) Compute for each \((m, n)\) with \(H_{mn} = 1\)
\[ q_{mn}(0) = \alpha_{mn} p_n(0) \prod_{m' \in \mathcal{M}_{nm}} r_{m'n}(0) \] (24.78)
and
\[ q_{mn}(1) = \alpha_{mn} p_n(1) \prod_{m' \in \mathcal{M}_{nm}} r_{m'n}(1) \] (24.79)

where the normalization constant is
\[ \alpha_{mn} = \frac{1}{p_n(0) \prod_{m' \in \mathcal{M}_{nm}} r_{m'n}(0) + p_n(1) \prod_{m' \in \mathcal{M}_{nm}} r_{m'n}(1)} \] (24.80)

so that \(q_{mn}(0) + q_{mn}(1) = 1\)

(b) Compute the “pseudoposterior” probabilities
\[ q_n(0) = \alpha_n p_n(0) \prod_{m' \in \mathcal{M}_n} r_{m'n}(0) \] (24.81)
and
\[ q_n(1) = \alpha_n p_n(1) \prod_{m' \in \mathcal{M}_n} r_{m'n}(1) \] (24.82)

where the normalization constant is
\[ \alpha_n = \frac{1}{p_n(0) \prod_{m' \in \mathcal{M}_n} r_{m'n}(0) + p_n(1) \prod_{m' \in \mathcal{M}_n} r_{m'n}(1)} \] (24.83)

Notice that the products in this step contain one more term than in the previous step since \(m' \in \mathcal{M}_n\) rather than \(\mathcal{M}_{nm}\).

5. Terminate Iterations or Continue

(a) Make hard bit decisions. Set \(\hat{x}_n = 1\) if \(q_n(1) > 0.5\), otherwise set \(\hat{x}_n = 0\) for \(n = 1, \ldots, N\).

(b) If \(H\hat{x}^T = 0\) stop. Otherwise, if the number of iterations is less than \(L\) loop back to the Horizontal Step.

If the iteration limit is reached and the checks are not satisfied declare a decoding error and stop.
24.9 Iterative Decoding Using Log Likelihood Ratios

The decoding algorithm can be implemented in terms of log likelihood ratios (LLR’s) instead of probabilities. An LLR compactly includes all the probability information about a binary random variable in a single number since the probability of a 0 or 1 can be computed from the LLR. Using LLR’s eliminates the normalization steps of the algorithm using probabilities. Using logs can also reduce dynamic range computational problems. As before suppose the length $N$ binary codeword $x$ is transmitted bit-by-bit and the $n$-tuple $r$ is received. The LLR for bit $x_n$ is

$$\lambda(x_n | r) = \log \frac{P(x_n = 1 | r)}{P(x_n = 0 | r)}$$ (24.84)

According to the rules of conditional probability

$$P(x_n = b | r) = \frac{f(x_n = b, r)}{f(r)} = \frac{f(x_n = b, r, \{r_i; i \neq n\})}{f(r, \{r_i; i \neq n\})}$$

$$= \frac{f(r_n | x_n = b, \{r_i; i \neq n\})f(x_n = b, \{r_i; i \neq n\})}{f(r_n, \{r_i; i \neq n\})}$$ for $b = 0, 1$ (24.85)

The channel output, $r_n$, only depends on the analog input value corresponding to bit $x_n$, so

$$P(x_n = b | r) = \frac{f(r_n | x_n = b)f(x_n = b, \{r_i; i \neq n\})}{f(r_n | \{r_i; i \neq n\})f(\{r_i; i \neq n\})}$$

$$= \frac{f(r_n | x_n = b)P(x_n = b | \{r_i; i \neq n\})}{f(r_n | \{r_i; i \neq n\})}$$ (24.86)

Therefore

$$\lambda(x_n | r) = \log \frac{f(r_n | x_n = 1)}{f(r_n | x_n = 0)} + \log \frac{P(x_n = 1 | \{r_i; i \neq n\})}{P(x_n = 0 | \{r_i; i \neq n\})}$$ (24.87)

The term

$$\lambda_I(r_n | x_n) = \log \frac{f(r_n = 1 | x_n)}{f(r_n = 0 | x_n)}$$ (24.88)

is called the intrinsic information for the bit $x_n$ and provides information about the channel effects. For BPSK signaling over an additive, white, Gaussian noise channel as described in Section 24.4.1, it was shown that $\lambda_I(r_n | x_n) = (-2A/\sigma^2)r_n$. 

End of Pseudo Code

This algorithm has been viewed as passing messages between the nodes of the graph. See Fan [1] for an extensive presentation of this approach. Message in the form of probability vectors $q_{mn}(b)$ are passed from the bit nodes to the check nodes and are combined by the bit nodes using the **Horizontal Step**. The check nodes then pass the messages $r_{mn}(b)$ to the bit nodes which combine them using the **Vertical Step**.
The right-hand term
\[ \lambda_E(x_n \mid \{r_i; i \neq n\}) = \log \frac{P(x_n = 1 \mid \{r_i; i \neq n\})}{P(x_n = 0 \mid \{r_i; i \neq n\})} \] (24.89)
is called the extrinsic information and provides information resulting from the code constraints and other observations.

The extrinsic term will now be expressed in terms of the code constraints. Let the \(m\)th parity check sum on \(x_n\) with \(x_n\) deleted be
\[ z_{mn} = \sum_{i \in N_{mn}} x_i \] (24.90)
so \(x_n + z_{mn} = 0\). This constraint implies \(z_{mn} = 1\) when \(x_n = 1\) and \(z_{mn}\) must have an odd number of 1’s in its sum. Similarly, \(z_{mn}\) must be 0 and have an even number of 1’s in its sum when \(x_n = 0\). Therefore,
\[ \lambda(x_n \mid r) = \lambda_I(r_n \mid x_n) + \log \frac{P(z_{mn} = 1 \forall m \in M_n \mid \{r_i; i \neq n\})}{P(z_{mn} = 0 \forall m \in M_n \mid \{r_i; i \neq n\})} \] (24.91)
It will be assumed that the bits in \(z_{mn}\) are different and independent for different checks so that the checks are independent. Then the probabilities in the numerator and denominator can be expanded into products and
\[ \lambda(x_n \mid r) = \lambda_I(r_n \mid x_n) + \sum_{m \in M_n} \log \frac{P(z_{mn} = 1 \mid \{r_i; i \neq n\})}{P(z_{mn} = 0 \mid \{r_i; i \neq n\})} \] (24.92)
The probability of an odd number of 1’s in a set of \(N\) independent binary random variables was derived in Section 24.6 and is given by (24.23). Accordingly, for \(m \in M_n\)
\[ P(z_{mn} = 1 \mid \{r_i; i \neq n\}) = \frac{1}{2} - \frac{1}{2} \prod_{\ell \in N_{mn}} [P(x_{\ell} = 0 \mid \{r_i; i \neq n\}) - P(x_{\ell} = 1 \mid \{r_i; i \neq n\})] \] (24.93)
\[ = \frac{1}{2} - \frac{1}{2} \prod_{\ell \in N_{mn}} [1 - 2P(x_{\ell} = 1 \mid \{r_i; i \neq n\})] \] (24.94)
The probability of an even number of 1’s in a set of \(N\) independent binary random variables is given by (24.22). Therefore,
\[ P(z_{mn} = 0 \mid \{r_i; i \neq n\}) = \frac{1}{2} + \frac{1}{2} \prod_{\ell \in N_{mn}} [P(x_{\ell} = 0 \mid \{r_i; i \neq n\}) - P(x_{\ell} = 1 \mid \{r_i; i \neq n\})] \] (24.95)
\[ = \frac{1}{2} + \frac{1}{2} \prod_{\ell \in N_{mn}} [1 - 2P(x_{\ell} = 1 \mid \{r_i; i \neq n\})] \] (24.96)
To shorten the equation text let \( p_{\ell}(b) = P(x_{\ell} = b \mid \{r_{i}; i \neq n\}) \) for \( b = 0, 1 \). Then

\[
\lambda(x_n|\mathbf{r}) = \lambda_I(r_n | x_n) + \sum_{m \in M_n} \log \frac{1 - \prod_{\ell \in N_m} (p_{n}(0) - p_{n}(1))}{1 + \prod_{\ell \in N_m} (p_{n}(0) - p_{n}(1))}
\]

(24.97)

\[
= \lambda_I(x_n | r_n) + \lambda_E(x_n \mid \{r_{i}; i \neq n\})
\]

(24.98)

Let the LLR for \( x_{\ell} \) given \( \{r_{i}; i \neq n\} \) be denoted by

\[
\lambda(x_{\ell} \mid \{r_{i}; i \neq n\}) = \log \frac{P(x_{\ell} = 1 \mid \{r_{i}; i \neq n\})}{P(x_{\ell} = 0 \mid \{r_{i}; i \neq n\})}
\]

(24.99)

Using Equation (24.43) in Section 24.7 this can be transformed into

\[
\lambda(x_n|\mathbf{r}) = \lambda_I(r_n | x_n) + \sum_{m \in M_n} (-1)^{|N_m|} 2 \tanh^{-1} \left( \prod_{\ell \in N_m} \frac{\lambda(x_{\ell} \mid \{r_{i}; i \neq n\})}{2} \right)
\]

(24.100)

For \((N,j,k)\) uniform LDPC codes \(|N_m| = k\).

### 24.9.1 Transforming the LLR Product Into a Sum

Gallager’s method of converting the product into a sum can now be used. Hardware multipliers are more complicated than adders. Equations (24.43) and (24.53) in Subsection 24.7.1 can be used to convert the product in (24.100) into a sum so that

\[
\lambda(x_n|\mathbf{r}) = \lambda_I(r_n | x_n) + \sum_{m \in M_n} (-1)^{|N_m|} \Psi \left( \prod_{\ell \in N_m} \frac{\lambda(x_{\ell} \mid \{r_{i}; i \neq n\})}{2} \right)
\]

(24.101)

where

\[
\Psi(x) = -\log \left( \tanh \frac{x}{2} \right) = \log \frac{1 + e^{-x}}{1 - e^{-x}} = \log \frac{e^x + 1}{e^x - 1} \quad \text{for} \quad x > 0
\]

(24.102)

The approximation of (24.55) can be used to simplify \( \lambda(x_n|\mathbf{r}) \) with a small loss in performance. The result is

\[
\lambda(x_n|\mathbf{r}) \approx \lambda_I(r_n | x_n) + \sum_{m \in M_n} (-1)^{|N_m|} \left( \prod_{\ell \in N_m} \text{sgn} \lambda(x_{\ell} \mid \{r_{i}; i \neq n\}) \right) \times \min_{\ell \in N_m} |\lambda_I(x_{\ell} \mid \{r_{i}; i \neq n\})|
\]

(24.103)
24.9 Iterative Decoding Using Log Likelihood Ratios

24.9.2 Details of the Iterative Decoding Algorithm Using LLR’s

Let
\[ \eta_{mn} = (-1)^{|N_m|} 2 \tanh^{-1} \left( \prod_{\ell \in N_{mn}} \tanh \frac{\lambda(\{r_i; i \neq n\})}{2} \right) \] (24.104)

The LLR’s \( \lambda(\{r_i; i \neq n\}) \) in the product can be viewed as messages passed from bit nodes other than bit \( x_n \) to check node \( m \). Then check node \( m \) combines these messages using (24.104) and sends the result as a message \( \eta_{mn} \) to bit node \( n \).

Using (24.104) Equation (24.100) can be compactly written as
\[ \lambda_n = \lambda(x_n | r) = \lambda_I(x_n | r_n) + \sum_{k \in M_n} \eta_{kn} \] (24.105)

Let \( \lambda_n \) with term \( \eta_{mn} \) deleted from the sum be
\[ \lambda_{nm} = \lambda_n - \eta_{mn} = \lambda_I(x_n | r_n) + \sum_{k \in M_{nm}} \eta_{kn} \] (24.106)

where \( M_{nm} \) is \( M_n \) with \( m \) deleted. This can be viewed as bit node \( n \) combining the messages \( \eta_{mn} \) from all the check nodes except check node \( m \) it is connected to according to (24.106) which is then passed to the check node \( m \).

If an iterative decoding algorithm alternating between (24.105) and (24.104) were used, a problem would emerge. The solution is to remove from the message that bit node \( n \) sends to check node \( m \) the message it has already received from check node \( m \). The result \( \lambda_{nm} \) is the extrinsic information passed from the bit nodes to the check nodes. Then the iterations alternate between (24.106) and (24.104).

Pseudo Code for the LDPC Decoding Algorithm Using Log-Likelihood Ratios

1. Input
   \( H \), the maximum number of iterations \( L \), the received vector \( r \), and the intrinsic LLR’s \( \lambda_I(r_n | x_n) = \log[f(r_n = 1 | x_n)/f(r_n = 0 | x_n)] \) for \( n = 1, \ldots, N \)

2. Initialization
   (a) Set \( \eta_{mn}^0 = 0 \) for all \( (m, n) \) with \( H_{mn} = 1 \).
   (b) Set \( \lambda_n^0 = \lambda_{nm}^0 = \lambda_I(r_n | x_n) \) for \( n = 1, \ldots, N \).
   (c) Set loop counter \( i = 1 \).

3. Check Node Update
   For each \( (m, n) \) with \( H_{mn} = 1 \) compute the message \( \eta_{mn}^i \) check node \( m \) sends to bit node \( n \):
   \[ \eta_{mn}^i = (-1)^{|N_m|} 2 \tanh^{-1} \left( \prod_{\ell \in N_{mn}} \tanh \frac{\lambda_{m\ell}^{i-1}}{2} \right) \] (24.107)
3’. Alternative Approximate Check Node Update
For each \((m,n)\) with \(H_{mn} = 1\) compute
\[
\eta\_{mn}^{[i]} = (-1)^{|N_m|} \left( \prod_{\ell \in N_m} \text{sgn} \lambda\_{\ell m}^{[i-1]} \right) \min_{\ell \in N_m} |\lambda\_{\ell m}^{[i-1]}| \tag{24.108}
\]

4. Bit Node Update
For \(n = 1, \ldots, N\) compute
\[
\lambda\_{n}^{[i]} = \lambda_I(r_n \mid x_n) + \sum_{k \in M_n} \eta_{kn}^{[i]} \tag{24.109}
\]

5. Make Hard Bit Decisions
Set \(\hat{x}_n = 1\) if \(\lambda\_{n}^{[i]} > 0\), otherwise set \(\hat{x}_n = 0\).

6. Terminate Iterations or Continue
(a) **Stop** if \(H\hat{x}^T = 0\). Otherwise compute for all \((m,n)\) with \(H_{mn} = 1\)
\[
\lambda_{nm}^{[i]} = \lambda_n^{[i]} - \eta_{mn}^{[i]} = \lambda_I(r_n \mid x_n) + \sum_{k \in M_{nm}} \eta_{kn}^{[i]} \tag{24.110}
\]
Increment \(i\) and loop back to Check Node Update if \(i \leq L\).
(b) if \(i > L\) and \(H\hat{x}^T \neq 0\) declare a decoding failure and stop.

End of Pseudo Code

24.9.3 Interpreting the Iterations as Passing Messages Between Nodes in a Graph

There are many publications presenting the LDPC iterative decoding algorithm in terms of message passing in graphs. One of the earliest presenters of belief propagation in graphs was J. Pearl [12]. Figure 24.7 (a) illustrates the Check Node Update equation (24.107). Check node \(m\) combines the LLR’s, \(\lambda_{\ell m}^{[i-1]}\), from all the bit nodes connected to it except bit \(n\) using (24.107) and sends the message \(\eta_{mn}^{[i]}\) to bit node \(n\). This can be viewed as passing messages from a tier in the tree grown from a bit in the Tanner graph to a bit in the next lower tier.

Figure 24.7 (b) illustrates the Bit Node Update equation (24.109). Bit node \(n\) combines all the messages, \(\eta_{kn}^{[i]}\), from all the check nodes connected to bit node \(n\) except for check node \(m\) and sends the LLR, \(\lambda_{nm}^{[i]}\), to check node \(m\). This is computed for each \(m \in M_n\). This is like using messages from the next tier up in the expansion of the Tanner graph into a tree to be used by bits in this tier to update the checks in this tier.

The term \(\eta_{mn}^{[i]}\) is must be added to \(\lambda_{nm}^{[i]}\) to form the complete LLR for bit \(x_n\) and a hard bit decision is made based on the result to be used in checking it the resulting entire vector of hard decisions is actually a code word.
Figure 24.7: Message Passing for LDPC Code Graphs
24.10 Encoding of LDPC Codes

The codes thus far have been described in terms of a parity check matrix $H$ which has $M = N - K$ rows and $N$ columns with $N > M$. The codewords are the set of all $N$-tuples $x = (x_1, \ldots, x_N)$ that satisfy the parity equation $Hx^T = 0$. It will be assumed that $H$ has full rank so that its rows are linearly independent. The rows of $H$ form a basis for the null space of the space of codewords which has dimension $K$. There must be a set of $K$ linearly independent $N$ component row vectors, $g_1, \ldots, g_K$ that span the code space which can be stacked to form a generator matrix, $G$ of dimension $K \times N$. Each codeword can be specified by a vector of $K$ information bits $u = (u_1, \ldots, u_K)$ and codewords can be computed as $x = uG$.

24.10.1 Classical Systematic Encoding

A code is called systematic if the information bits appear explicitly in a set of known positions in all codewords. Suppose the information bits occupy positions 1 through $K$ in a code word. The remaining $N - K$ positions, $K + 1$ through $N$, contain check bits $c = (c_1, \ldots, c_{N-K})$ selected so the codeword satisfies the parity check equation. Then codewords have the form $x = (u \ c) = (u_1, \ldots, u_K, c_1, \ldots, c_{N-K})$. A generator matrix can be found by reducing the the parity check matrix $H$ by simple row operations and column and/or row exchanges to the standard form

$$H_{syst} = \begin{bmatrix} P_{(N-K)\times K} & I_{N-K} \end{bmatrix} \quad (24.111)$$

Then the generator matrix for the standard systematic code is

$$G = \begin{bmatrix} I_K & P^T \end{bmatrix} \quad (24.112)$$

and the $2^K$ codewords are

$$x = uG = (u \ c) \quad (24.113)$$

for all possible binary information vectors $u$.

EXAMPLE 24.1 Creating a Generator Matrix in Systematic Form

Consider this parity check matrix for an irregular LDPC code.

$$H = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \quad (24.114)$$
This matrix will first be converted to row echelon form. Adding the 1st row to the 4th row gives
\[
H_1 = \begin{bmatrix}
1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
\end{bmatrix}
\] (24.115)

Interchanging the 3rd and 5th rows gives
\[
H_2 = \begin{bmatrix}
1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
\end{bmatrix}
\] (24.116)

Adding the 4th row to the 5th row gives the row-echelon form
\[
H_3 = \begin{bmatrix}
1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\
\end{bmatrix}
\] (24.117)

Next the row-echelon form can be reduced to the reduced row-echelon form by the row operations: add 2nd row to 1st row, add 3rd row to 1st row, add 3rd row to 2nd row, add 4th row to 1st row, add 5th row to 1st row, add 5th row to 2nd row, add 5th row to 4th row, which all results in
\[
H_{rr} = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\
\end{bmatrix}
\] (24.118)

The check matrix $H_{rr}$ has the identity first followed by $P$ so the check bits occupy the first five positions and the information bits the last five positions. The corresponding generator matrix is
\[
G = \begin{bmatrix}
0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
\end{bmatrix}
\] (24.119)

The first five and last five columns of the original $H$ can be swapped if it is desired to have the information bits occupy the first five positions. Then the reduction to standard form
would have the $I$ and $P$ swapped using the corresponding row operations.

The problem with this approach is that the resulting $P$ matrix is usually no longer sparse for longer codes and often has a high density. Therefore, encoding requires significant computation. A more efficient approach to encoding is presented next.

### 24.10.2 Efficient Encoding of LDPC Codes

Consider a linear code with length $N$, $K$ information bits, and $M = N_K$ check bits. It will be assumed that the parity check matrix $H$ has full rank $M$. Richardson and Urbanke [13] presented an efficient method for encoding LDPC codes that takes advantage of the sparseness of the parity check matrix. The method begins by rearranging rows and columns of $H$ to transform it into an approximate lower triangular matrix. This does not change the sparseness of $H$. The resulting form is illustrated in Figure 24.8. The rearranging is performed to maximize the size of the lower triangular matrix $T$ which then has $M - g$ rows and columns. There are $g$ remaining rows at the bottom and $g$ is called the gap. Richardson and Urbanke present methods for achieving a small gap. Let the resulting matrix be $H_1$ as shown in the following equation where $A$ is an $(M - g) \times (N - M)$ matrix, $B$ is an $(M - g) \times g$ matrix, $C$ is a $g \times (N - M)$ matrix, $D$ is a $g \times g$ matrix, $E$ is a $g \times (M - g)$ matrix, and $T$ is an $(M - g) \times (M - g)$ lower triangular matrix with 1’s on the diagonal.

\[
H_1 = \begin{bmatrix} A & B & T \\ C & D & E \end{bmatrix}
\] (24.120)

The next step is to perform row operations that leave the upper $M - g$ rows the same and reduce $E$ to an all 0 matrix. This can be achieved by multiplying $H_1$ from the left by
the $M \times M$ matrix
\[
\begin{bmatrix}
I_{(M-g)} & 0_{(M-g)\times g} \\
-ET^{-1} & I_{g \times g}
\end{bmatrix}
\] (24.121)

The result is
\[
H_2 = \begin{bmatrix} A & B \end{bmatrix} T = \begin{bmatrix} A & B & T \end{bmatrix}
\] (24.122)

Let the systematic code words have the form $x = (u \ p_1 \ p_2)$ where $u$ is the $K$-dimensional information row vector, $p_1$ is a $g$-dimensional row vector of parity bits, and $p_2$ is an $M - g$ dimensional row vector of parity bits. The parity check equation $H_2x^T = 0$ can be separated into the following two equations:
\[
Au^T + Bp_1^T + Tp_2^T = 0
\] (24.123)

and
\[
\tilde{C}u^T + \tilde{D}p_1^T = 0
\] (24.124)

Therefore,
\[
p_1^T = -\tilde{D}^{-1}\tilde{C}u^T = -(\tilde{E}T^{-1}B + D)^{-1}(\tilde{E}T^{-1}A + C)u^T
\] (24.125)

After $p_1$ is computed, $p_2$ can be computed as
\[
p_2^T = -T^{-1}(Au^T + Bp_1^T)
\] (24.126)

The matrices $A$ and $B$ are sparse but $\tilde{D}^{-1}\tilde{C}$ usually will not be sparse, so computing $p_1$ by (24.125) will involve significant computation. Richardson and Urbanke present a computationally efficient method for computing $p_1^T$ by breaking the direct computation into smaller steps involving sparse matrices as shown in the next two paragraphs.

**Steps for efficiently computing $p_1^T$**

1. Compute $Au^T$. This has complexity $O(N)$ since $A$ is sparse.
2. Compute $y^T = T^{-1}[Au^T]$ This equation is equivalent to $[Au^T] = Ty^T$. The vector $y^T$ can be computed efficiently by back substitution since $T$ is lower triangular. This has complexity $O(N)$.
3. Compute $-E[T^{-1}Au^T]$. This is multiplication by the sparse matrix $E$ and has complexity $O(N)$.
4. Compute $Cu^T$. Multiplication by the sparse matrix $C$ has complexity $O(N)$.
5. Compute the sum $[-ET^{-1}Au^T + Cu^T]$. This addition has complexity $O(N)$.
6. Compute $-\tilde{D}^{-1}[-ET^{-1}Au^T + Cu^T]$. The matrix $\tilde{D}^{-1}$ is $g \times g$, so this operation has complexity $O(g^2)$. 

Steps for efficiently computing \( \mathbf{p}_2^T \) after computing \( \mathbf{p}_1^T \)

1. \( \mathbf{A} \mathbf{u}^T \) has already been computed in the steps for computing \( \mathbf{p}_1^T \).
2. Compute \( \mathbf{B} \mathbf{p}_1^T \). Multiplication by the sparse matrix \( \mathbf{B} \) has complexity \( O(N) \).
3. Compute \( \mathbf{A} \mathbf{u}^T + \mathbf{B} \mathbf{p}_1^T \). This addition has complexity \( O(N) \).
4. Compute \( \mathbf{p}_2^T = -\mathbf{T}^{-1}(\mathbf{A} \mathbf{u}^T + \mathbf{B} \mathbf{p}_1^T) \). This equation is equivalent to \( \mathbf{A} \mathbf{u}^T + \mathbf{B} \mathbf{p}_1^T = \mathbf{T} \mathbf{p}_2^T \). The vector \( \mathbf{p}_2^T \) can be efficiently computed by back substitution since \( \mathbf{T} \) is lower triangular. The complexity of this operation is \( O(N) \).

**Example 24.2** Transforming an \( \mathbf{H} \) for Efficient Encoding and a Coding Example

Suppose the original parity check matrix for an LDPC code is

\[
\mathbf{H} = \begin{bmatrix}
1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1
\end{bmatrix}
\]  

(24.127)

Interchanging rows 2 and 3 and columns 6 and 10 gives

\[
\mathbf{H}_1 = \begin{bmatrix}
1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1
\end{bmatrix}
\]  

(24.128)

which has the gap \( g = 2 \). Then

\[
\mathbf{T}^{-1} = \begin{bmatrix}
1 & 0 & 0 \\
1 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]  

(24.129)

and

\[
\begin{bmatrix}
\mathbf{I}_{m-g} & 0 \\
\mathbf{E} \mathbf{T}^{-1} & \mathbf{I}_g
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 & 1
\end{bmatrix}
\]  

(24.130)

So

\[
\tilde{\mathbf{H}} = \begin{bmatrix}
\mathbf{I}_{m-g} & 0 \\
\mathbf{E} \mathbf{T}^{-1} & \mathbf{I}_g
\end{bmatrix} \mathbf{H}_1 = \begin{bmatrix}
1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\
0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0
\end{bmatrix}
\]  

(24.131)
The resulting sub-matrices are
\[
\begin{align*}
A &= \begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \end{bmatrix}, \\
B &= \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}, \\
\tilde{C} &= \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix}, \\
\tilde{D} &= \tilde{D}^{-1} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}
\end{align*}
\] (24.132)

The density of 1’s in $\tilde{H}$ is about the same as in the original $H$. However, this will not be the case as the code length, $N$, grows.

Now suppose $u = (1 \ 1 \ 0 \ 0 \ 1)$, $p_1 = (p_{1,1} \ p_{1,2})$, and $p_2 = (p_{2,1} \ p_{2,2} \ p_{2,3})$ so that the codeword is $x = (u \ p_1 \ p_2)$. For binary codes

\[
p_{1}^{T} = \tilde{D}^{-1} \tilde{C} u^{T} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}
\]

and

\[
p_{2}^{T} = T^{-1}(Au^{T} + Bp_{1}^{T}) = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}
\]

The transmitted codeword is

\[
x = (1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0)
\] (24.135)

As an exercise, you can try the “efficient” encoding algorithm.

24.11 LDPC Codes in IEEE Standard 802.11-2012

The WiFi standard IEEE 802.11-2012 [6] includes QC LDPC codes as an optional feature. “QC” stand for “quasi-cyclic.” The codes can have rates 1/2, 2/3, 3/4, or 5/6. At each rate the length can be 648 = 24 × 27, 1296 = 24 × 54, or 1944 = 24 × 81. The parity check matrices for the codes are defined by matrix prototypes in the standard. Prototype examples for the $N = 648$ codes are shown in Table 24.1. You can find the prototypes for the other two lengths in Annex F of the standard. Each prototype has 24 columns. The entries in the prototypes represent $Z \times Z$ matrices where $Z$ can be 27, 54, or 81. The entry “-” represents a $Z \times Z$ matrix of all zeros. The entry “0” represents a $Z \times Z$ identity matrix. For an integer $i > 0$, the entry “$i$” represents the $Z \times Z$ identity matrix cyclically shifted $i$ positions to the right.
Table 24.1: Matrix prototypes for codeword block length $N = 648$ bits, subblock size is $Z = 27$ bits

<table>
<thead>
<tr>
<th>Coding rate $R$</th>
<th>Matrix Prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R = 1/2$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 - - - 0 0 - - 0 - - 0 1 0 - - - - - - - - - -</td>
</tr>
<tr>
<td></td>
<td>22 0 - - 17 - 0 0 12 - - - - 0 0 - - - - - - - -</td>
</tr>
<tr>
<td></td>
<td>6 - 0 - 10 - - - 24 - 0 - - - 0 0 - - - - - - - -</td>
</tr>
<tr>
<td></td>
<td>2 - - 0 20 - - - 25 0 - - - 0 0 - - - - - - - -</td>
</tr>
<tr>
<td></td>
<td>23 - - - 3 - - - 0 - 9 11 - - - 0 0 - - - - - - - -</td>
</tr>
<tr>
<td></td>
<td>24 - 23 1 17 - 3 - 10 - - - - - - - 0 0 - - - - - -</td>
</tr>
<tr>
<td></td>
<td>25 - - - 8 - - - 7 18 - - 0 - - - - - - - 0 0 - - - -</td>
</tr>
<tr>
<td></td>
<td>13 24 - - 0 - 8 - 6 - - - - - - - - - - - 0 0 - - - -</td>
</tr>
<tr>
<td></td>
<td>7 20 - 16 22 10 - - 23 - - - - - - - - - - - 0 0 - - - -</td>
</tr>
<tr>
<td></td>
<td>11 - - - 19 - - - 13 - 3 17 - - - - - - - - - - 0 0 - - - -</td>
</tr>
<tr>
<td></td>
<td>25 - 8 - 23 18 - 14 9 - - - - - - - - - - - - - - 0 0 - - - -</td>
</tr>
<tr>
<td></td>
<td>3 - - - 16 - - - 2 25 5 - - 1 - - - - - - - - - - - - - -</td>
</tr>
<tr>
<td>$R = 2/3$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>25 26 14 - 20 - 2 - 4 - - 8 - 16 - 18 1 0 - - - - - - - -</td>
</tr>
<tr>
<td></td>
<td>10 9 15 11 - 0 - 1 - - 18 - 8 - 10 - - 0 0 - - - - - - - -</td>
</tr>
<tr>
<td></td>
<td>16 2 20 26 21 - 6 - 1 26 - 7 - - - - - - - 0 0 - - - - - -</td>
</tr>
<tr>
<td></td>
<td>10 13 5 0 - 3 - 7 - - 26 - - 13 - 16 - - - 0 0 - - - - - -</td>
</tr>
<tr>
<td></td>
<td>23 14 24 - 12 - 19 - 17 - - - - 20 - 21 - 0 - - - - 0 0 - - - -</td>
</tr>
<tr>
<td></td>
<td>6 22 9 20 - 25 - 17 - 8 - 14 - 18 - - - - - - - - 0 0 - - - -</td>
</tr>
<tr>
<td></td>
<td>14 23 21 11 20 - 24 - 18 - 19 - - - - - - - - 22 - - - - - - - -</td>
</tr>
<tr>
<td></td>
<td>17 11 11 20 - 21 - 26 - 3 - - 18 - 26 - 1 - - - - - - - - - -</td>
</tr>
<tr>
<td>$R = 3/4$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16 17 22 24 9 3 14 - 4 2 7 - 26 - 2 - 21 - 1 0 - - - - - - - -</td>
</tr>
<tr>
<td></td>
<td>25 12 12 3 3 26 6 21 - 15 22 - 15 - 4 - - 16 - 0 0 - - - - - -</td>
</tr>
<tr>
<td></td>
<td>25 18 26 16 22 23 9 - 0 - 4 - 4 - 8 23 11 - - - - - - - - - - - -</td>
</tr>
<tr>
<td></td>
<td>9 7 0 1 17 - - 7 3 - 3 23 - 16 - - 21 - 0 - - 0 0 - - - - - -</td>
</tr>
<tr>
<td></td>
<td>24 5 26 7 1 - - 15 24 15 - 8 - 13 - 13 - 11 - - - - 0 0 - - - -</td>
</tr>
<tr>
<td></td>
<td>2 2 19 14 24 1 15 19 - 21 - 2 - 24 - 3 - - - - 2 1 - - - - - - - -</td>
</tr>
<tr>
<td>$R = 5/6$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>17 13 8 21 9 3 18 12 10 0 4 15 19 2 5 10 26 19 13 13 1 0 - - - -</td>
</tr>
<tr>
<td></td>
<td>3 12 11 14 11 25 5 18 0 9 2 26 26 10 24 7 14 20 4 2 - 0 0 0 - - - -</td>
</tr>
<tr>
<td></td>
<td>22 16 4 3 10 21 12 5 21 14 19 5 - 8 5 18 11 5 5 15 0 - 0 0 - - - -</td>
</tr>
<tr>
<td></td>
<td>7 7 14 14 4 16 16 24 24 10 1 7 15 6 10 26 8 18 21 14 1 - - - - - - - -</td>
</tr>
</tbody>
</table>
The parity check matrices when viewed in terms of the identity permutation numbers and -’s have an approximately lower triangular (ALT) structure with a gap of $g = 1$ block, so the efficient encoding method of Section 24.10.2 can be used. The matrix prototypes can be partitioned into sub-blocks as

$$
\begin{bmatrix}
1 & 0 & - & - & \ldots & - & - \\
- & 0 & 0 & - & - & \ldots & - \\
\vdots & - & 0 & 0 & - & \ldots & - \\
\vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\
- & - & - & 0 & - & \ldots & - \\
- & - & - & - & 0 & 0 \\
C_P & 1 & - & - & \ldots & - & - & 0 \\
\end{bmatrix}
$$

(24.136)

The subscript $P$ on a matrix will be used to denote that the matrix is represented in terms of the IEEE 802.11-2012 standard’s cyclic permutation notation. The matrix $A_P$ consists of the set of $(M - 1) \times (N - M)$ upper left-hand blocks of $H_P$. $C_P$ is a row vector consisting of the first $K = N - M$ elements of the bottom row of $H_P$. The matrix $B_P$ corresponding to Section 24.10.2 is

$$
B_P = [1 - \ldots - 0 - \ldots -]^T
$$

(24.137)

which is a column vector consisting of the first $M - 1$ elements of column $N - M + 1 = K + 1$. The [0] in $B_P$ occurs in row $M/2 + 1$. The sub-block $D_P$ is the 1 in the bottom row. Sub-block $E_P = [- - \ldots - - 0]$, that is, an $M - 1$ dimensional row vector consisting of elements $N - M + 2 = K + 2$ through $N$ of the bottom row. The matrix $T_P$ is an $(M - 1) \times (M - 1)$ matrix consisting of the upper right-hand $(M - 1) \times (M - 1)$ sub-block of $H_P$. It has the shift number 0 representing the $Z \times Z$ identity matrix down the main diagonal and the diagonal below the main diagonal and -’s representing the all zero $Z \times Z$ matrix everywhere else. It is a lower triangular matrix. The binary matrix $T$ has $Z \times Z$ identity matrices down the main and sub diagonal and zeros everywhere else.

Notice that each of the last $M - 1$ columns of $H_P$ has two 0’s and -’s everywhere else, so these columns of $H$ have two identity blocks and all zero blocks elsewhere. Therefore the sum of all the blocks in any one of these columns is the all zero block. If the last row of $H$ is replaced by the sum of the all the rows, $E$ will be replaced by $M - 1$ blocks of all zeros. Column $N - M + 1$ corresponding to $B_P$ over [1] has a [1] at the top, a [1] at the bottom, a [0] at row $M/2 + 1$, and -’s everywhere else in prototype notation. The sum of all the corresponding binary blocks is the $Z \times Z$ identity matrix corresponding to [0] since the two blocks corresponding to [1] cancel each other. Consequently $\tilde{D} = I_Z$ for each of the codes. The block in position $i$, $1 \leq i \leq N - M$, of the row vector $\tilde{C}$ is the sum of the blocks in column $i$ of $A$ and block $i$ of $C$. In summary, replacing the last row of blocks in $H$ by the
sum of all the $M$ rows of $H$ on a column by column basis transforms $H$ into the form of $H_2$ in (24.122) but in terms of $Z \times Z$ blocks described by Richardson and Urbanke for efficient encoding.

A more formal way of getting to this result is to first verify by multiplication that $T^{-1}$ is an $(M-1) \times (M-1)$ lower triangular block matrix with $I_Z$ along the main diagonal and zero blocks everywhere above the main diagonal. Each element of the bottom row of $T^{-1}$ is $I_Z$. The matrix $E$ consists of $M-2$ all zero blocks followed by $I_Z$ in position $N$. Therefore $ET^{-1} = [I_Z \ldots I_Z]$ which is the bottom row of $T^{-1}$. The matrix for zeroing $E$ given by (24.121) has the bottom row with $M$ columns

\[ Y = [ET^{-1} I_Z] = [I_Z \ldots I_Z I_Z] \]  

(24.138)

Thus the block in position $i$ in the bottom row of $H_2 = YH$ is the sum of all the blocks of $H$ in column $i$.

Let the codewords be row vectors of length $N = 24$ sub row vectors of length $Z$ and have the form $x = (u \ p_1 \ p_2)$ where

\[
\begin{align*}
  u &= (u_1 \ u_2 \ldots \ u_{24-M}); \ u_i \text{ a } Z \text{ component row vector} \\
  p_1 &= \text{ a } Z \text{ component row vector} \\
  p_2 &= (p_{2,1} \ p_{2,2} \ldots \ p_{2,M-1}); \ p_{2,i} \text{ a } Z \text{ component row vector}
\end{align*}
\]

Let row $i$ of $A$ be $a_i$. Then

\[
Au^T = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_{M-1} \end{bmatrix} u^T = \begin{bmatrix} \rho_1 \\ \rho_2 \\ \vdots \\ \rho_{M-1} \end{bmatrix} \]  

(24.139)

where $\rho_i = a_i u^T$. Let $\rho_M = Cu^T$. The bottom row of $H_2 x^T = 0$ is

\[
\begin{align*}
  Cu^T + \tilde{D} p_1^T &= [ET^{-1} I_Z] A u^T + I_Z p_1^T = [I_Z \ldots I_Z] \begin{bmatrix} \rho_1 \\ \rho_2 \\ \vdots \\ \rho_{M-1} \end{bmatrix} + p_1^T \\
  &= \sum_{i=1}^{M} \rho_i + p_1^T = 0
\end{align*}
\]

(24.140)

Therefore,

\[
p_1^T = \sum_{i=1}^{M} \rho_i
\]

(24.141)
The top $M - 1$ block rows of $H_2 x^T = 0$ are $A u^T + B p_1^T + T p_2^T = 0$, so

$$T p_2^T = A u^T + B p_1^T = \begin{bmatrix} \rho_1 \\ \rho_2 \\ \vdots \\ \rho_{M-1} \end{bmatrix} + B p_1^T \tag{24.142}$$

The elements of $p_2^T$ can be found by back substitution since $T$ is lower triangular. The top element, $B_1$, of $B$ is an identity matrix cyclically shifted once to the right. The effect of multiplying $p_1^T$ by this shifted identity from the left is to cyclically shift $p_1^T$ up one position. Let $[p_1^T]^{[1]}$ represent $p_1^T$ shifted cyclically up on position. Then the top element in $p_2^T$ is

$$p_{2,1}^T = \rho_1 + [p_1^T]^{[1]} \tag{24.143}$$

By back substitution

$$p_{2,i}^T = p_{2,i-1}^T + \rho_i \quad \text{for } i = 2, \ldots, M/2 \tag{24.144}$$

$$p_{2,M/2+1}^T = p_{2,M/2}^T + p_1^T + \rho_{M/2+1} \tag{24.145}$$

$$p_{2,i}^T = p_{2,i-1}^T + \rho_i \quad \text{for } i = M/2 + 1, \ldots, M - 1 \tag{24.146}$$

### 24.12 Additional Topics to Explore

LDPC codes have been extensively studied analytically in recent years. One question that is still not exactly answered is how the message passing algorithms behave for specific codes. This question has been partially answered by a method called *density evolution* [14][11]. Approximations to how the probability density functions of the LLR messages passed from the bit nodes to the check nodes evolve are made averaged over an ensemble of codes as the number of iterations increases. The codes are assumed to have lengths approaching infinity and the Tanner graphs are assumed to have no cycles. Threshold values of SNR above which the iterations converge and decode nearly perfectly are found. Optimization of parity check matrix column and row weights to minimize the threshold has been performed using density evolution.

Another major research topic is the design of good LDPC codes. Approaches include random generation of parity check matrices with no small length cycles, use of incidence matrices for finite geometries, and various combinatorial structures. Gallager proposed regular LDPC codes but researchers have found irregular codes that perform better and very close to Shannon’s channel capacity.

The use of LDPC codes has been empirically justified by extensive use in the real world in equipment conforming to a number of international standards.
24.13 Experiments for LDPC Codes

24.13.1 Experiments with a (12,3,6) LDPC Code

Let the parity check matrix for a (12,3,6) code be

\[
H = \begin{bmatrix}
1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
\end{bmatrix}
\]  

Perform the following encoding tasks:

1. Draw the Tanner graph for \( H \).
2. Convert \( H \) to standard form \([ P_{6x6} \ I_6 ]\). You may have to swap rows.
3. Give the generator matrix in the standard form \( G = [ I \ P^T ] \) for the systematic code with the six information bits in the first six codeword positions.
4. Generate all 64 codewords using \( G \) and list them in a table. Find the weight spectrum, that is, the number of codewords with each possible Hamming weight. The Hamming weight of a binary word is the number of 1’s in it.

Notice that columns 6 and 9 of \( H \) are the same so the sum of these two columns is a zero vector. Let \( x = ( 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 ) \). Then the sum of these to columns is \( Hx^T = 0 \) and \( x \) is a codeword of weight two. Since the code is linear, the sum of an error pattern equal to \( x \) and any codeword using modulo 2 addition will be another codeword and the error will be undetectable. Suppose codewords are transmitted over a BSC. Then a single error in position 6 or 9 will result in a word that is Hamming distance 1 from the correct codeword and another codeword, so the error cannot be corrected unambiguously. Either of the two words was equally likely to have been transmitted.

5. Use the method of Section 24.10.2 for an approximately lower diagonal check matrix to generate all 64 codewords.

Perform the following decoding tasks:

1. Assume codewords are transmitted over a BSC and that the all 0 codeword is transmitted. This introduces no loss of generality because the code is linear and all codewords are treated similarly.

   (a) Assume the channel introduces a single error by complementing the first codeword bit. Use the bit flipping algorithm to try to decode the received word.
(b) Assume the channel introduces a single error by complementing bit 6. Try to decode with the bit flipping algorithm. Also try bit flipping decoding if the single error is in bit 9.

(c) Introduce some double errors and see if they can be corrected by the bit flipping algorithm.

2. Assume codewords are transmitted over an AWGN channel using BPSK with analog signal amplitude $A = 1$. Assume the all 0 codeword is transmitted. Generate a noise vector of twelve random numbers each being observations of a Gaussian random variable with zero mean and variance $\sigma^2$. Add this noise vector to the transmitted BPSK word.

(a) Decode the received noisy word using the iterative decoding method using probabilities. Experiment with different values of $\sigma^2$. Repeat the decoding multiple times for the same $\sigma^2$ and new noise vector. Create a plot of the bit error rate versus $1/\sigma^2$.

(b) Decode the received noisy word using the iterative decoding method using LLR’s. Experiment with different values of $\sigma^2$. Repeat the decoding multiple times for the same $\sigma^2$ and new noise vector. Create a plot of the bit error rate versus $1/\sigma^2$.

### 24.13.2 Experiments with the IEEE 802.11 LDPC Codes

If you want to do more experimentation with LDPC codes used extensively in the real world:

1. Create a program to encode the WiFi LDPC code of Section 24.11 using the approximately lower triangular method presented in that section.

2. Create a program for decoding this LDPC code by the iterative method using LLR’s and generate a plot of the decoded bit error rate vs. SNR.
24.14 References


