Channel Codes

Chapter 8

Channel Coding

System designers use codes to improve data rate and/or reduce P_e and $\bar{P}_b$. Chapter 2's capacity upper bounds the reliably achievable data rate. More than a half-century after Shannon's capacity introduction [1] of capacity, many fine engineers cumulative effort has produced a repertoire of good codes that with increasing complexity can permit data rates that reliably approach capacity on both the BSC and the AWGN. This chapter instead addresses these codes' use in design; there are many excellent texts that explore the often fascinating and sophisticated mathematics that underly these codes' specifics, see for instance [11], [3], [4]

This chapter introduces, describes, and tabulates these codes. Our present objective is a reference to which the designer can refer and estimate the various coding gains, complexities, and types of codes applicable to support their communication-system designs. Lists of good codes then help designers make prudent choices.

Sections 8.1 and 8.2 respectively study and tabulate binary codes. Section 8.1 also finds binary block codes as a special case of convolutional codes. Section 8.1 also addresses binary codes' Gray Code mapping to multi-level constellations in ways that preserve the gain; Section 8.1 also investigates this mapping's impact on the use of Chapter 7's decoding methods. Section 8.3 progresses to the most powerful binary codes, which have long block lengths and challenge ML decoding complexity. Different types of suboptimal decoding presumption lead to the 3 most powerful forms of coding known as Turbo, LDPC, and Polar codes. These codes allow designs to transmit reliably with rate approaching capacity. Section 8.3 also discusses random interleaving that is used in both Bit-Interleaved Coded Modulation (BICM) in general and in Turbo Codes as well. BICM extends Section 8.1's Gray Code mappers to long-length codes in expanding code applicability to multilevel constellations. Another constellation mapping, called "mapping by set partitioning," found wide past use, but has performance below best codes; this method appears in Appendix B for completeness, Sections B.3-B.6.

Discrete Memoryless Channels (DMCs) often use Chapter 2's powerful maximum-distance separable (MDS) codes, which Section 8.4 explains and tabulates for design, under the title Cyclic Codes. Chapters 1 and 2 define and overview shaping gain; Section 8.5 explains and tabulates constellation-shaping methods that independently adjoin good coding-gain codes to effect best additional shaping gain.

Interleavers amplify coding gain in different ways, to both increase the codes' power in AWGN use as well as to deal with noise bursts or "impulse noise" or its wireless dual time-variant fading. Appendix B addresses deterministic block and convolutional interleaving that can be useful with these non-Gaussian impairments. Section 8.4 addresses codes that improve shaping gain on the AWGN.

8.1 Binary Codes and Encoder Implementation

Binary codes' theory and realization uses extensively Appendix B's finite-field theory, fortunately here simplifying to simple binary arithmetic. [1] This section does not attempt rigor, but rather use of the basic binary convolutional-coding concepts of convolutional codes. Appendix B, Section 7, also describes methods for equivalent binary encoders' representations.

8.1.1 Binary Sequences

Let $GF(2)$, or more compactly $\mathbb{F}_2$ be the finite (binary or modulo-2) field with elements $\{0, 1\}$. A sequence of bits, $\{a_m\}$, in $\mathbb{F}_2$ has modulo-2 D transform

$$a(D) = \sum_{m=-\infty}^{\infty} a_m \cdot D^m \quad . \tag{8.1}$$

Addition is modulo 2, so "exclusive or" for simplicity in binary codes. With finite-field transforms, the variable D is a placeholder, and has no relation to the complex/real sequences (then $a_m \in \mathbb{C}$) Fourier transform (i.e., $D \neq e^{-\jmath\omega T}$). For this reason, $a(D)$ uses a lower case a in the transform notation $a(D)$, rather than the upper case A that instead addresses $a_m \in \mathbb{C}$. When $D = 0$, the sequence simplifies to the block $a(D) = a_0$. The vector sequence $\boldsymbol{a}(D)$ replaces a_m with a vector $\boldsymbol{a}_m$ that has all $a_{m,n} \in \mathbb{F}_2$, equivalently $\boldsymbol{a} \in \mathbb{F}_2^n$.

$F[D]$ denotes[2] the set of all polynomials (finite-length causal sequences) with coefficients in $\mathbb{F}_2$,

$$F[D] \triangleq \left\{ a(D) \ \mid \ a(D) = \sum_{m=0}^{\nu} a_m \cdot D^m \ , \ a_m \in \mathbb{F}_2, \ \nu \in \{0, \ \mathbb{Z}^+\} \right\} \quad . \tag{8.2}$$

The set $F[D]$ can also be described as the set of all polynomials in D with finite Hamming weight. Let $F_r(D)$ be the field formed by taking ratios $a(D)/b(D)$ of any two polynomials in $F[D]$, with $b(D) \neq 0$ and $b_0 = 1$:

$$F_r(D) \triangleq \left\{ c(D) \ \mid \ c(D) = \frac{a(D)}{b(D)} \ , \ a(D) \in F[D], \ b(D) \neq 0, \ b(D) \in F[D] \text{with } b_0 = 1 \right\} \quad . \tag{8.3}$$

This set $F_r(D)$ is the set of **rational polynomial fractions** in D.

> **Definition 8.1.1** *[Delay of a Sequence] The* **delay**, *$del(a)$, of a nonzero sequence $a(D)$, is the smallest time index, m, for which a_m is nonzero. The delay of the zero sequence $a(D) = 0$ is defined to be ∞. This is equivalent to the lowest power of D in $a(D)$.*

A causal sequence starts at time (dimension) $m = 0$, and so all sequences of interest will have delay greater than or equal to 0. For example, the delay of the sequence $1 + D + D^2$ is 0, while the delay of the sequence $D^5 + D^{10}$ is 5.

> **Definition 8.1.2** *[Degree of a Sequence] The* **degree**, *$deg(a)$, of a nonzero sequence, $a(D)$ is the largest time index, m, for which a_m is nonzero. The degree of the zero sequence $a(D) = 0$ is defined to be $-\infty$. This is equivalent to the highest power of D in $a(D)$. A block has degree 0.*

For example, the degree of the sequence $1 + D + D^2$ is 2, while the degree of the sequence $D^5 + D^{10}$ is 10.

> **Definition 8.1.3** *[Length of a Sequence] The* **length**, *$len(a)$, of a nonzero sequence, $a(D)$ is*
>
> $$len(a) \triangleq deg(a) - del(a) + 1 \quad . \tag{8.4}$$
>
> *The length of the zero sequence $a(D) = 0$ is defined to be $len(0) \triangleq 0$. A block has length 1.*

For example, the length of the sequence $1 + D + D^2$ is 3, while the length of the sequence $D^5 + D^{10}$ is 6.

[1] Appendix B's finite-fields more general (non-binary) finite fields find use also in Section 8.4 and 8.6.

[2] The set $F[D]$ can be shown to be a **ring** (see Appendix B.)

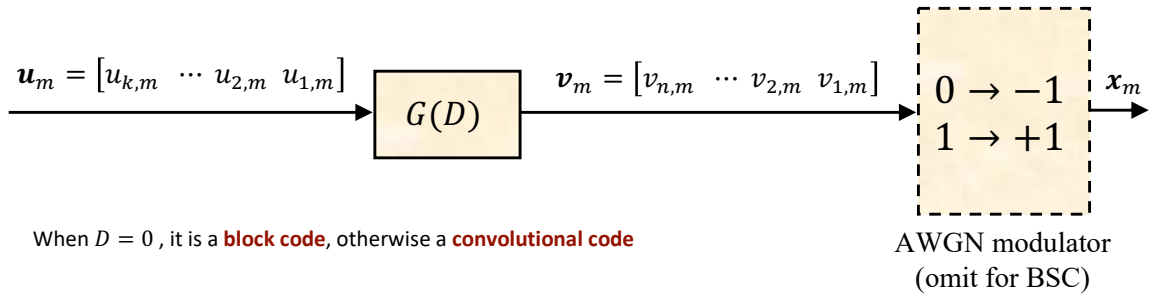

Figure 8.1: General convolutional encoder block diagram.

8.1.2 The Binary Code

Figure 8.1 illustrates the general binary encoder. Coding theorists often use a time index m instead of k; k instead represents the number of encoder input bits ($k = b$ earlier in this text). This chapter then tries to maintain consistency with the convolutional-coding literature, in particular with k input bits at time m producing n output bits. The input message bit-vector is $\boldsymbol{u}(D)$ avoiding also then this texts traditional $m(D)$ message nomenclature with this chapter's binary codes.

Binary codes have the k-dimensional binary-input vector sequence, $\boldsymbol{u}_m = [u_{k,m} \ u_{k-1,m} \ ... \ u_{1,m}]$ that produces an n-dimensional binary-vector output sequence $\boldsymbol{v}_m = [v_{n,m} \ v_{n-1,m} \ ... \ v_{1,m}]$. The output vector may then pass through a modulator that produces the N-dimensional channel input vector $\boldsymbol{x}_m \in \mathbb{C}^N$. The binary code codes relates $\boldsymbol{u}_m$ and $\boldsymbol{v}_m$. The relationship between $\boldsymbol{v}_m$ and $\boldsymbol{x}_m$ for the AWGN is often a simple binary translation to the bipolar (2 PAM) modulated waveform with amplitude $\pm\sqrt{\bar{\mathcal{E}}_{\boldsymbol{x}}}$, as shown. Section 8.1.7 later relaxes this 2PAM input restriction through the use of a constellation mapping. The vector sequence $\boldsymbol{v}(D)$ can apply directly to a BSC, while $\boldsymbol{x}(D)$ is for an AWGN.

The linear binary code has a **generator** $G(D)$:

Definition 8.1.4 *[Linear Binary Code] A linear binary code's* **generator matrix**, *$G(D)$ can be any $k \times n$ matrix with entries in $F_r(D)$ and rank k. The* **binary code**, *$C(G)$, is the set of all n-dimensional vector sequences $\{\boldsymbol{v}(D)\}$ that can be formed by premultiplying $G(D)$ by any k-dimensional vector sequence, $\boldsymbol{u}(D)$, whose component polynomials are causal and binary. That is, the code $C(G)$ is the set of n-dimensional sequences, whose components are in $F_r(D)$, that fall in the subspace spanned by the rows of $G(D)$. Mathematically,*

$$C(G) \triangleq \{\boldsymbol{v}(D) \mid \boldsymbol{v}(D) = \boldsymbol{u}(D) \cdot G(D) \ , \ \boldsymbol{u}(D) \in F_r(D)\} \quad . \tag{8.5}$$

If $G(D) = G(0)$, then it is a **binary block code** *and otherwise a* **convolutional code**. *A binary code's* **code rate** *is $r = k/n = \bar{b}$.*

Two linear binary encoders that correspond to generators $G(D)$ and $G'(D)$ are **equivalent** if they generate the same code, that is $C(G) = C(G')$.

Lemma 8.1.1 *[Equivalence of Binary Codes] Two binary codes are* **equivalent** *if and only if there exists an invertible $k \times k$ matrix, A, of polynomials in $F_r(D)$ such that these two codes generators satisfy $G'(D) = A \cdot G(D)$.*

The rows of $G(D)$ span a k-dimensional subspace of $F_r(D)$. Any $(n-k) \times n$ matrix that spans the orthogonal complement of the rows of $G(D)$ is known as a **parity matrix** for the binary code, that is:

Definition 8.1.5 (Parity Matrix) *An $(n-k) \times n$ matrix of rank $(n-k)$, $H(D)$, is known as the parity matrix for a code if for any codeword $\boldsymbol{v}(D)$, $\boldsymbol{v}(D) \cdot H^*(D) = 0$ (where $*$ denotes transpose in this case).*

An alternative binary-code definition follows then the set of n-dimensional sequences $\boldsymbol{v}(D) \in \{F_r(D)\}^n$ that satisfy

$$C(G) = \{\boldsymbol{v}(D) \mid \boldsymbol{v}(D) \cdot H^*(D) = 0\} \quad . \tag{8.6}$$

Thus, $G(D) \cdot H^*(D) = 0$. When $(n-k) < k$, the parity matrix more compactly describes the code. $H(D)$ also describes a $(n-k) \times n$ **dual binary code** generator. All dual-code codewords are obviously orthogonal to those in the original code, and the two codes together span the n-dimensional space of sequences with coefficients in $\{\mathbb{F}_2\}^n$.

Binary code's codeword separation is again given by Chapter 2's **Hamming distance**, now adding also the **Hamming weight**:

Definition 8.1.6 *[Hamming Distance] The* **Hamming Distance**, *$d_H(\boldsymbol{v}(D), \boldsymbol{v}'(D))$, between two sequences, $v(D)$ and $v'(D)$, is the number of bit positions in which they differ.*

Similarly the **Hamming weight** *$w_H(\boldsymbol{v}(D))$ is defined as the Hamming distance between the codeword and the zero sequence, $w_H(\boldsymbol{v}(D)) \triangleq d_H(\boldsymbol{v}(D), 0)$. Equivalently, the Hamming weight is the number of "ones" in a binary codeword.*

Definition 8.1.7 *[Systematic Encoder] A* **systematic** *convolutional encoder has $v_{n-i}(D) = u_{k-i}(D)$ for $i = 0, ..., k-1$.*

Equivalently, the systematic encoder directly passes all input bits to corresponding output bits, with the remaining $n-k$ output bits as "parity" bits. Proper output indexing may be needed to ensure that a systematic encoder exists.

A common code-complexity measure is the **constraint length**. The constraint-length concept helps transform any generator $G(D)$ whose entries are not in $F[D]$ into an equivalent generator with entries in $F[D]$ by multiplying every row by the denominator polynomials' least common multiple, call it $\phi(D)$ ($A = \phi(D) \cdot I_k$ in Lemma 8.1.1). Such transformation permits definition of a code-complexity measure, the constraint length:

Definition 8.1.8 *[Constraint Length] The* **constraint length**, *ν, of a convolutional encoder $G(D)$ with entries in $F[D]$ is $\log_2$ of the number of encoder states; equivalently it is the number of D flip-flops (or delay elements) in the obvious realization (by means of a FIR filter). Block codes have $\nu = 0$.*

If ν_i $(i = 1, ..., k)$ is the degree or constraint length of $G(D)$'s i^{th} row (that is the maximum degree of the n polynomials in the ith row of $G(D)$), then in $G(D)$'s obvious realization has

$$\nu = \sum_{i=1}^{k} \nu_i \quad . \tag{8.7}$$

A convolutional code's **complexity** measures its implementation over all possible equivalent encoders:

> **Definition 8.1.9** *[Minimal Encoder] A convolutional code's* **complexity** μ *is the minimum constraint length over all equivalent encoders* $\{G(D)\}$ *such that* $C = C(G)$*. An encoder is said to be* **minimal** *if the complexity equals the constraint length,* $\nu = \mu$*.*
>
> *A minimal encoder with feedback has a realization with the number of delay elements equal to the complexity.*

The following is a (brute-force) algorithm to construct a minimal encoder $G_{min}(D)$ for a given generator $G(D)$. This algorithm was first described by Piret[5]:

1. Construct a list of all possible n-vectors whose entries are polynomials of degree ν or less, where ν is the constraint length of $G(D)$. Sort this list in order of nondecreasing degree, so that the zeros vector is first on the list.
2. Delete from this list all n-vectors that are not codeword sequences in $C(G)$. (To test whether a given row of n polynomials is a codeword sequence, it suffices to test whether all $(k+1) \times (k+1)$ determinants vanish when the $k \times n$ matrix $G(D)$ is augmented by that row.)
3. Delete from this remaining list all codewords that are linear combinations of previous codewords.

The final list should have k codewords in it that can be taken as the rows of $G_{min}(D)$ and the constraint length must be μ because the above procedure selected those codewords with the smallest degrees. The initial list of codewords should have $2^{n(\nu+1)}$ codewords. A well-written program would combine steps 2 and 3 above and would stop on step 2 after generating k independent codeword sequences. This combined step need only search somewhere between $2^{n\mu}$ and $2^{n(\mu+1)}$ codewords, because the minimum number of codewords searched would be $2^{n\mu}$ if at least one codeword of degree μ exists, and the maximum would be $2^{n(\mu+1)} - 1$ if no codewords of degree $\mu + 1$ exist. A more efficient method for generating the minimal encoder involves Appendix B's basic encoders. Codes tabulated and/or exemplified here are all minimal.

8.1.3 Examples

EXAMPLE 8.1.1 (4-state Optimal Rate 1/2 Code) Section 2.2's convolutional encoder reappears in more detail in Figure 8.2. There is $k = 1$ input bit and $n = 2$ output bits, so that the code rate is $r = k/n = 1/2$. The input/output relations are

$$v_2(D) = (1 + D + D^2) \cdot u_1(D) \tag{8.8}$$

$$v_1(D) = (1 + D^2) \cdot u_1(D) \tag{8.9}$$

Thus,

$$G(D) = \begin{bmatrix} 1 + D + D^2 & 1 + D^2 \end{bmatrix} \quad . \tag{8.10}$$

The constraint length is $\nu = 2$. The parity matrix is

$$H(D) = \begin{bmatrix} 1 + D^2 & 1 + D + D^2 \end{bmatrix} \quad . \tag{8.11}$$

The systematic encoder

$$G(D) = \begin{bmatrix} 1 & \dfrac{1 + D^2}{1 + D + D^2} \end{bmatrix} \quad , \tag{8.12}$$

generates the same code, even though the mapping from input bits to the codewords differs.

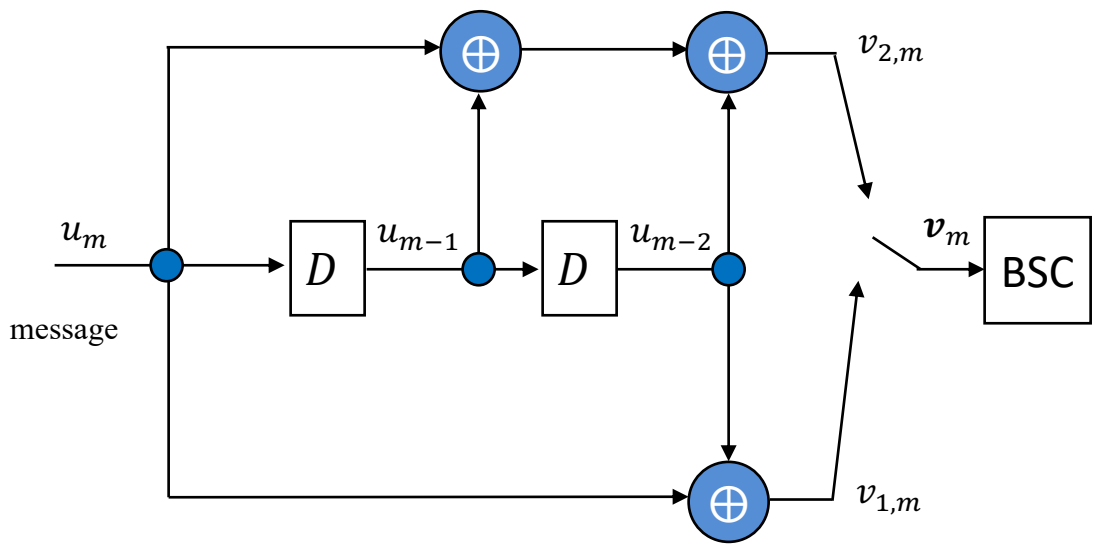

Figure 8.2: 4-state encoder example.

8.1.4 Trellis Diagrams for Convolutional Codes

Trellis diagrams often describe convolutional codes. Figure 8.3's trellis diagram alternately describes Example 8.1.1's code.

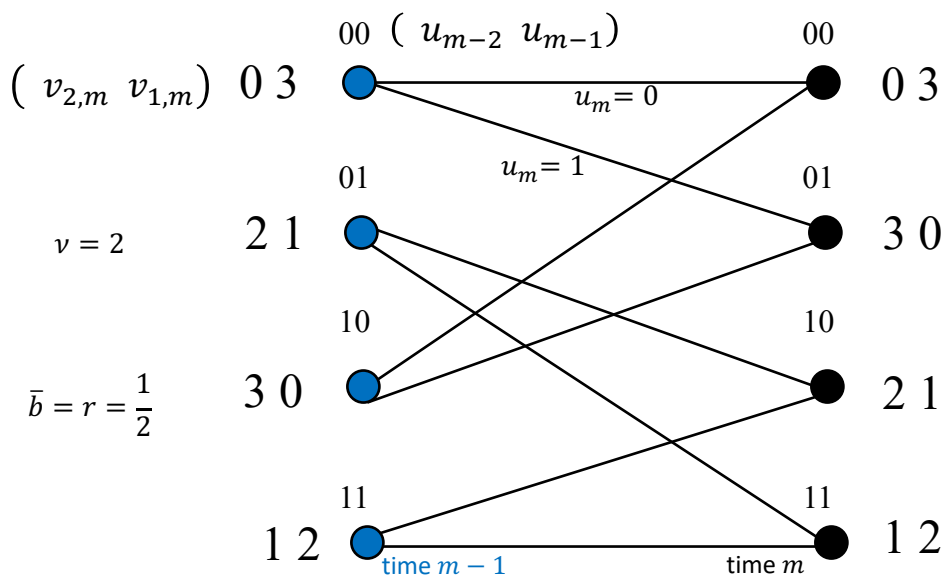

Figure 8.3: Trellis for 4-state encoder example.

The state is the ordered pair $(u_{1,m-2}\ ,\ u_{1,m-1})$. Each states upper emanating trellis branch at time $m-1$ corresponds to $u_m = 0$, while the lower branch corresponds to $m = 1$. The output labelss are mod-4 to simplify the label notation with the most significant bit corresponding to the left-most

bit, $v_{n,m}$. On the left, the mod-4 value written closer to the trellis corresponds to the lowest branch emanating from the state it labels, while the mod-4 value written furthest from the trellis corresponds to the highest branch emanating from the state it labels. The labeling is vice-versa on the right.

Figure 8.10 shows the systematic realization, which has the same paths but would correspond to a different-looking trellis stage.

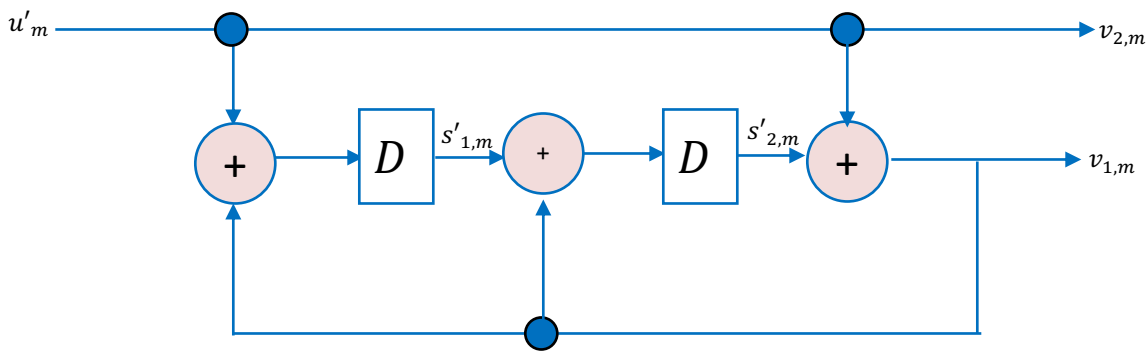

Figure 8.4: Systematic minimal encoder with feedback for 4-state example.

8.1.5 Matlab's base convolutional-code structure programs:

Matlab provides some convenient functions to help in convolutional code use. The root program actually defines a trellis.

poly2trellis.m The poly2trellis creates a data structure that contains trellis connectivity and output labelling. This program is core matlab so no source file is available, but here are the comments. Example 8.1.2's imminent revisit with this function also reveals its trellis labelling versus input-assignment is somewhat confusing.

```
poly2trellis Convert convolutional code polynomial to trellis description.
    TRELLIS = poly2trellis(CONSTRAINTLENGTH, CODEGENERATOR) converts a
    polynomial representation of a feedforward convolutional encoder to a
    trellis structure. For a rate k/n code, the encoder input is a vector
    of length k, and the encoder output is a vector of length n. Therefore,
    - CONSTRAINTLENGTH is a 1-by-k vector specifying the delay for each of
      the k input bit streams.
    - CODEGENERATOR is either; (i) a k-by-n matrix of octal numbers or
      (ii) a k-by-n cell array of polynomial character vectors, or
      (iii) a k-by-n string array, all of which specify
      the n output connections for each of the k inputs.
    TRELLIS = poly2trellis(CONSTRAINTLENGTH, CODEGENERATOR, FEEDBACKCONNECTION)
    is the same as the first syntax, but for a feedback convolutional encoder.
    - FEEDBACKCONNECTION is a 1-by-k vector of octal numbers specifying the
      feedback connection for each of the k inputs.
    A trellis is represented by a structure with the following fields:
      numInputSymbols,  (number of input symbols, 2^k)
      numOutputSymbols, (number of output symbols, 2^n)
      numStates,        (number of states)
      nextStates,       (next state matrix)
      outputs,          (output matrix)
    For more information about trellis structures, type 'help istrellis' in
    MATLAB.
```

plotnextstates.m The plotnextstates.m is from matlab's file exchange, and so therefore source code is available (also provided in Appenedix G). It uses the trellis produced by the core poly2trellis.m program. Again example use appears in Example Example 8.1.2'.

```
>> help plotnextstates
  function plotnextstates(nextStates)
  Utility function to display nextStates matrix for Convolutional Encoder
  with Uncoded Bits and Feedback example.
```

convenc.m This program produces the outputs, again using the trellis provided by the poly2trellis.m function. Fortunately this program does maintain ordering of bits in time m even though that is inconsistent with matlab's left-to-right labelling of smallest to largest significant index, and thus is consistent with this text's labeling of highest time index on left or top.

```
convenc Convolutionally encode binary data.
    CODE = convenc(MSG,TRELLIS) encodes the binary vector MSG using the
    convolutional encoder defined by the MATLAB structure TRELLIS.  See
    POLY2TRELLIS and ISTRELLIS for a valid TRELLIS structure.  The encoder
    starts at the all-zeros state.  Each symbol in MSG consists of
    log2(TRELLIS.numInputSymbols) bits.  MSG may contain one or more symbols.
    CODE is a vector in the same orientation as MSG, and each of its symbols
    consists of log2(TRELLIS.numOutputSymbols) bits.
    CODE = convenc(MSG,TRELLIS,PUNCPAT) is the same as the syntax above,
    except that it specifies a puncture pattern (PUNCPAT) to allow higher
    rate encoding. PUNCPAT must be a vector of 1's and 0's where the 0's
    indicate the punctured bits. PUNCPAT must have a length of at least
    log2(TRELLIS.numOutputSymbols) bits.
    CODE = convenc(...,INIT_STATE) is the same as the syntaxes above,
    except that the encoder registers start at a state specified by
    INIT_STATE. INIT_STATE is an integer between 0 and
    TRELLIS.numStates - 1 and must be the last input parameter.
    [CODE FINAL_STATE] = convenc(...) returns the final state FINAL_STATE of
    the encoder after processing the input message.
    Example:
        t = poly2trellis([3 3],[6 5 1;7 2 5]);
        msg = [1 1 0 1 0 0 1 1];
        [code1 state1] = convenc(msg(1:end/2),t);
        [code2 state2] = convenc(msg(end/2+1:end),t,state1);
        [codeA stateA] = convenc(msg,t);
        % The same result will be returned in [code1 code2] and codeA.
        % The final states state2 and stateA are also equal.
```

distspec.m : This program computes d_{free} and optionally larger distances along with their coefficients N_e or $\{N_i\}$ (SPECT.EVENT) and N_b or $\{N(b,d)\}$ (SPECT.WEIGHT)

```
distspec  Compute the distance spectrum of a convolutional code.
    SPECT = distspec(TRELLIS,N) computes the free distance and the first N
    components of the weight and distance spectra of a linear convolutional
    code.  The output is a structure with three elements:
        SPECT.DFREE  -- the free distance of the code
        SPECT.WEIGHT -- a length N vector that lists the total number of
                        information bit errors in the error events
                        enumerated in SPECT.EVENT
        SPECT.EVENT  -- a length N vector that lists the number of error
                        events for each distance between SPECT.DFREE and
                        SPECT.DFREE+N-1
    SPECT = distspec(TRELLIS) is the same as SPECT = distspec(TRELLIS,1)
```

EXAMPLE 8.1.2 **[4-state Code with Matlab]** Example 8.1.1's code has octal generator $[1+D+D^2 \;\; 1+D^2] = [7 \;\; 5]$. Matlab's code description follows the industry practice of octal representation. What matlab calls the constraint length is actually the generator length, so $\nu+1$, with the number of states ouput as 2^ν.

```
t=poly2trellis(3, [7 5]);
    t.numInputSymbols = 2
    t.numOutputSymbols= 4
    t.numStates = 4
    t.nextStates =
     0     2
     0     2
     1     3
     1     3
    t.outputs =
     0     3
     3     0
     2     1
     1     2
```

For this code it happens that the nextStates is the same in first two rows (states) and rows 3 and 4, and so then consistently the upper path corresponds to input bit 0 and lower path corresponds to input bit 1. If these first two rows were reversed (so 2 0), then the 0 input bit would correspond to the lower path. This happens with many other codes, so the designer

should beware that this is highly inconsistent with most illustration of trellises. The outputs correspond 1-to-1 with the nextStates transition (and corresponding input mapping). The program convenc.m produces encoder outputs

```
>> convenc([0      0       0        0        1        1],t)    =
          0 0     0 0     0 0     0 0       1 1      0 1
```

The spacings here created to help associate input subsymbol bits with output subsymbols. The trellis appears in Figure 8.5.

```
>> plotnextstates(t.nextStates)
>> distspec(t,1)
     dfree: 5
    weight: 1
     event: 1
```

Figure 8.5(a) superimposes the labels matching the colors output from the plotnextsates program. The trellis transitions essentially do not correspond to the typical trellis convention, with for instance state 0 splitting to emanate to states 0 and state 2, instead of the usual state 0 and 1. For this trellis this corresponds to essentially relabelling the states in Figure 8.5(a) to reverse order (essentially grab states 01 and 10 by hand and reverse them (on both sides) to obtain Figure 8.5(b). There is a sequence of commands that can provide this as:

```
numbits=2;
for i=1:4    for j=1:2
nst(i,j)=uint16( bin2dec( fliplr( dec2bin( oct2dec(t.nextStates(i,j)),numbits) ) ) );
end
end
cst=bin2dec(fliplr(dec2base(0:3,2)))=
     0
     2
     1
     3
>> t2=t; t2.nextStates=nst(cst+1,:);
>> t2.nextStates =
     0     1
     2     3
     0     1
     2     3
     >> t2.outputs=t.outputs(cst+1,:);
>> t2.outputs =
     0     3
     2     1
     3     0
     1     2
convenc([0 0 0 0 1 1],t2) =     00     00     00     00     11     01
>> distspec(t2,1)
     dfree: 5
    weight: 1
     event: 1
```

Different trellis, but it has the same response to the same generator as it should. The free distance also remains the same as does N_e. In this case, N_b also remains the same, although this can vary with generator (which did not change here, but will for instance with other generators for the same code).

Another program use is a systematic generator with the following

```
>> t3=poly2trellis(3,[7 5],7) =
  struct with fields:
     numInputSymbols: 2
     numOutputSymbols: 4
           numStates: 4
>> t3.nextStates =
     0     2
     2     0  % note reversal of input-bit to branch
     3     1
     1     3   % note reversal of input-bit to branch
>> t3.outputs =
     0     3
     0     3   % so outputs reversed
```

```
     1     2
     1     2  % so outputs reversed
>> distspec(t3,1) =
    dfree: 5
    weight: 3 % different Nb
     event: 1
```

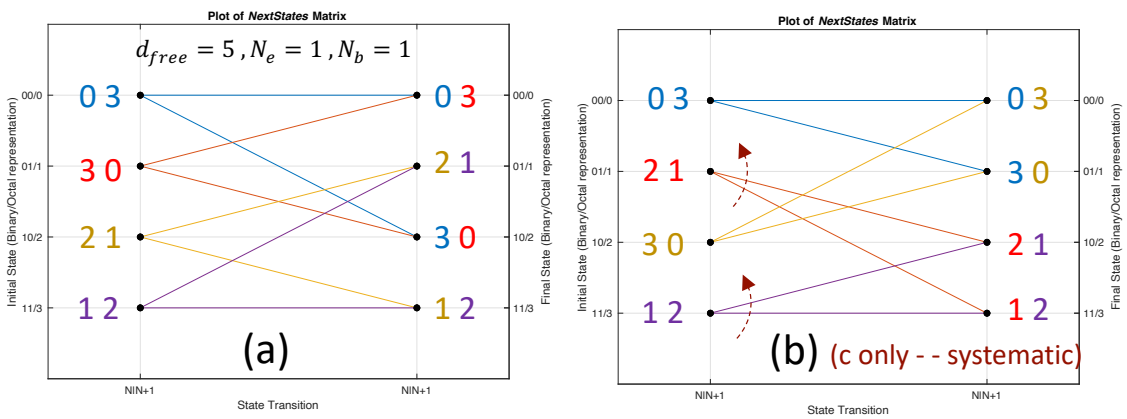

Figure 8.5: Trellis for 4-state code with $G(D) = 1 + D + D^2 \;\; 1 + D^2$ and (c) $[1 \;\; \frac{1+D^2}{1+D+D^2}]$.

8.1.6 Puncturing with convolutional codes

Puncturing removes some encoder-output bits according to a regular pattern (so less than n code output bits). Puncturing follows a periodic pattern over at least one subsymbol of n bits, and usually more than one. Puncturing necessarily increases the code rate r by decreasing its denominator. Consequent to puncturing's reduced dimensionality, the code d_{free} may also decrease. Effectively, the punctured code is a new code. However, implementations may prefer a single binary-code structure and vary rate (with single channel-quality indicator, for instance AWGN SNR) through puncturing.

Puncturing increases code rate. Removal of i bits from n output bits leads to

$$r = \frac{k}{n} =\rightarrow \frac{k}{n-i} \quad . \tag{8.13}$$

Punturing allows use of a single encoder and decoder mechanism across several rates with the deleted bits having also zero contribution to any decoder metric that would otherwise use them. This can lower distance, trivially by up to the i bits deleted so $d_{free} - i \leq d_{free,punc} \leq d_{free}$. Puncturing patters are usually periodic and known to both encoder and decoder. For instance an encoder that punctures an $r = 1/2$ code to $r = 3/4$ could aggregate 3 successive 2-output-bit groups into 6 bits and then delete them as

$$\underbrace{1 \;\; 1}_{\text{group 1}} \quad \underbrace{\not{0} \;\; 1}_{\text{group 2}} \quad \underbrace{1 \;\; \not{0}}_{\text{group 3}} \quad . \tag{8.14}$$

This puncturing alternately deletes the first and second bits of groups 2 and 3. Often the puncturing pattern is just given as [1 1 0 1 1 0] without the slashes with 0 indicating a puncture position and 1 indicating retained encoder-output bits. To puncture from $r = 1/2$ to $r = 2/3$, 4 successive groups are used as

$$\underbrace{1 \;\; 1}_{\text{group 1}} \quad \underbrace{1 \;\; \not{0}}_{\text{group 2}} \quad \underbrace{1 \;\; 1}_{\text{group 3}} \quad \underbrace{\not{0} \;\; 1}_{\text{group 4}} \quad . \tag{8.15}$$

The puncturing spaces the punctured bits from one another within the pattern as well as ensures that deletions do not always the same position within the defined group. Ultimately, puncturing finds the

least common multiple of the original encoder's n, so $j \cdot n$, for which it is possible realize the desired fraction $k/(n-i) = (j \cdot \cdot k)/(j \cdot n - i)$. For the first example above, $j = 3$, for the second $j = 4$. The puncturing patterns depend on the base code and some will be better than others. This is often found by trial and error. As turbo codes will later show, random bit interleaving and puncturing can often be best, consistently with random coding somewhat.

For the first example, the puncturing retains linearity for the new code that has 3 input bits and 4 output bits as

$$G_{punc}(D) = \begin{bmatrix} G(D) & 0 & 0 \\ 0 & G(D) & 0 \\ 0 & 0 & G(D) \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} . \tag{8.16}$$

8.1.6.1 The 64-state rate-1/2 code

One very popular code for use with puncturing is Figure 8.6's 64-state rate 1/2 "mother" convolutional code, which Chapter 4 illustrated for Wi-Fi.

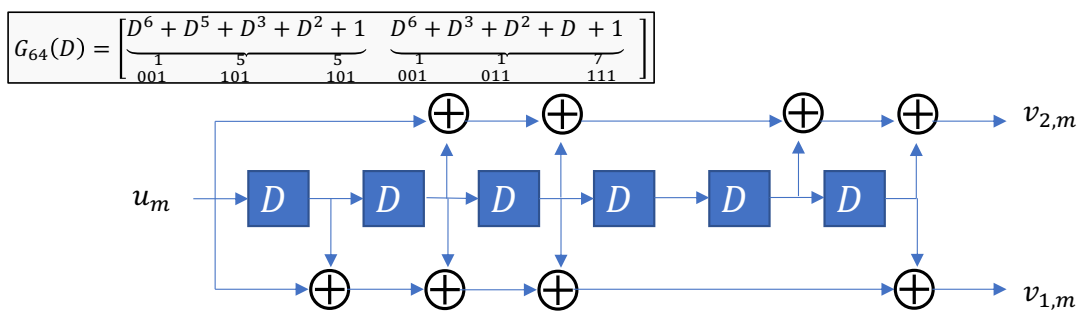

Figure 8.6: 64-state rate-1/2 Wi-Fi Convolutional Code encoder and generator.

EXAMPLE 8.1.3 *[Wi-Fi Convolutional-Code Puncturing]* Figure 8.6's heavily used Wi-Fi mother convolutional code has the generator shown. The generator-matrix entries appear above binary, and especially octal notation that convolutional-code tabulation often uses to simplify specification. This code has $d_{free} = 10$. This code's output bits map to SQ QAM constellations according to a Gray Code as Section 8.3 later shows. Figure 8.7 illustrates the puncturing patterns that accommodate 2 additional higher code rates. Figure 8.7 groups the 2-bit output vectors in correspondence with 6-bit input sets. Puncturing then occurs across these 12 bits for rates $r > 1/2$. The higher rates calculate as $r = 2/3$ and $r = 3/4$, with respective lower d_{free}'s of 7 and 6.

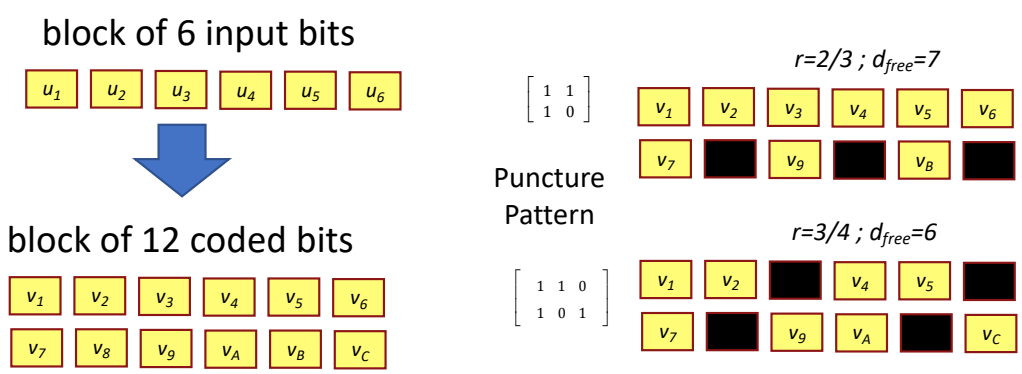

Figure 8.7: Wi-Fi Puncturing Scheme.

Puncturing spreads the punctured bits as evenly as possible over successive subsymboll groups. Section 8.3 has more puncturing examples.

8.1.7 BICM and Gray Code Mapping to PAM and SQ QAM

For 2PAM on an AWGN with levels $\pm\sqrt{\bar{\mathcal{E}}_{\boldsymbol{x}}}$, the minimum squared distance d^2_{min} relates when $r \geq 1/2$ to the binary code's free distance d_{free} as

$$d^2_{min} = 4 \cdot d_{free} \cdot \bar{\mathcal{E}}_{\boldsymbol{x}} \quad . \tag{8.17}$$

Coding theorists often use (8.17) to quote a **binary-coding gain**

$$\gamma_{bc} = r \cdot d_{free} \quad , \tag{8.18}$$

recognizing that with constant power $P_x = \mathcal{E}_{\boldsymbol{x}}/T$ that $1/T$ must increase to accommodate $1/r$ more dimensions, and thus $\bar{\mathcal{E}}_{\boldsymbol{x}}$ must reduce by r to support the consequent bandwidth expansion with constant P_x. This is a good relative measure for binary-code comparison with $|C| = 2$; however, Section 8.2 will show that it however will violate Chapter 1's fair comparison criteria. (8.17) holds for binary codes where d_{free} is the dominant contributor to error probability; very powerful codes instead necessarily have several distances' terms contribute to P_e. Furthermore, low-rate $r < 1/2$ codes often have large d_{free} making the ratio between d_{free} and $d_{free} + i$ for small integer i relatively close to unity. Thus binary-code coding gain has a limited range of meaningful use where d_{free} concerns dominate error probability. Binary codes with gain exceeding 6 to 7 dB and $r \geq 1/2$ also necessarily have this issue. So as long as binary codes with bipolar (2PAM) constellations on the AWGN are compared, (8.18) is a good metric; not necessarily otherwise. Good binary codes may have a large binary-coding gain, but it does not necessarily determine their performance (nor consequently gap Γ).

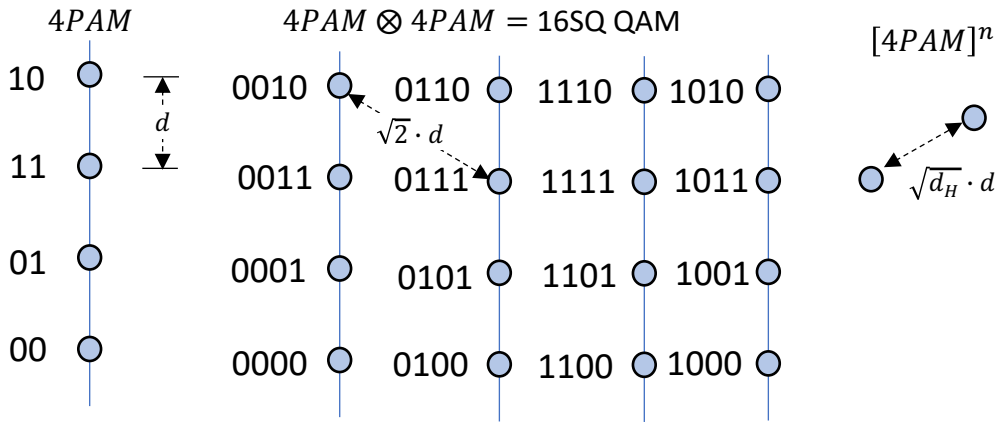

Figure 8.8: Gray Mapping of PAM and/or SQ QAM.

8.1.7.1 Multilevel Distance Map:

Chapter 2's BICM reuses binary codes with multilevel ($|C| > 2$) constellations with Figure 8.8's Gray Code mapping that ensures no two adjacent SQ QAM (or PAM) constellation points' labels have more that 1-bit hamming distance difference, Figure 8.8 illustrates a Gray Code for 4PAM, and also notes the extension through a Cartesian Product operation to any number of dimensions, with SQ QAM being often most of interest. Figure 8.8 illustrates that each step in another dimension that leads to one more bit difference, up to $d_H \leq n$ leads to a minimum distance increase by $\sqrt{d_H} \cdot d$ where d is the one-dimensional point spacing in the original PAM constellation. It appears with such Gray Code mapping, then (8.17) continues to hold if the bits individual à posterioi probabilities are independent, requiring Section 8.3's bit interleaving. Thus, the Gray code allows multilevel constellation's squared distance to scale also with free distance as

$$d_{min}^2 = d_{free} \cdot \bar{\mathcal{E}}_{\boldsymbol{x}} \quad . \tag{8.19}$$

For non-square QAM constellations, it is not generally possible to have a Gray mapping, but good designers can select the mapping to have a minimum number of points that differ by more than 1 bit in label.

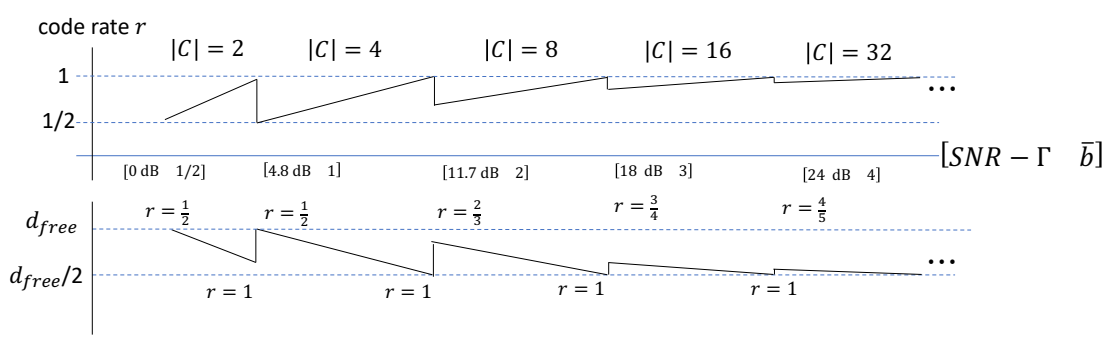

Figure 8.9: Rate-distance parametrization for binary codes with multilevel constellations.

Constellation-Expansion Care: Section 8.2 introduces the binary-coding theorist's fallacy, which generalizes the implied concepts here. Figure 8.9 illustrates PAM's (and SQ QAM's) discontinuous

transition for integer $\bar{b}$. At low $1/2 \leq \bar{b} \leq 1$, the coding gain $\gamma \approx \gamma_f$ because shaping effects are negligible at low bits/dimension. The discontinuous constellation jumps correspond to the applied binary code's (with Gray Mapping) application as the SNR grows and allows larger data rate. On the left of the discontinuity, the code rate can approach $r = 1$ because the SNR is relatively large with respect to the constellation and less d_{free} is needed; however the right side, the SNR is low for the new constellation and so the distance needs to increase to offset this effect. As the SNR continues to grow, the applied code continues to benefit from larger SNR for the fixed constellation size until the next discontinuity. With a sufficiently large N (and implied sufficiently large n), small gap codes approach capacity-achieving codes and the granularity of puncturing choice as well as corresponding d_{free} offset that compensates for the larger constellation is better.

This "code-rate-adaption-through-puncturing" essentially creates a good binary code with a nearly constant gap. It is thus an example of a code (really code class with appropriate puncturing) with constant gap. Figure 8.9 assumes a rate 1/2 code has good free distance (like the previous Wi-Fi code) that decreases with rate increase (punctured bits). Each continuous (line) region corresponds to a single constellation, just the rate r is changing. This means that the bandwidth narrows approaching the discontinuities from the left, and jumps (increasing Q-function argument because less symbol rate is required with a larger constellation to carry the same data rate) to the right of each discontinuity. As in the upcoming Section 8.2's coding-theorist's fallacy, such bandwidth increase may make some channel assumptions not likely to be true. In practice often the sampling/subsymbol rate is fixed and increasing r linearly does not increase energy/dimension as $\bar{\mathcal{E}}_{\boldsymbol{x}} \to \bar{\mathcal{E}}_{\boldsymbol{x}}/r$, instead requiring a larger constellation with required energy growing exponentially roughly as 6 dB/bit-dimension. This is another example of when $\bar{b} \geq 1$, it is better to take a factor increase in bandwidth rather than energy, somewhat diluting further the convolutional code's gain. Another way to vary the code-correction capability instead r and puncturing (or together with it) would use iterative decoding by passing the **output** bits' soft-information from the main binary code's decoder to the Gray-Code demapper, which accounts for this à prior information as in Chapter 7, Section 5. This increases the overall iterative-decoded systems' power, essentially offsetting Figure 8.9's discontinuous energy increases. That additional soft information is more helpful to overall decoding just to the right of the discontinuity when the intra-consellation-point distance decreases. Because of the Gray Code's simplicity, only 1 or 2 iterative passes are necessary.

The simple Gray Mapping may cause a more complex decoder than another mapping known as mapping by set partitioning (introduced by Ungerboeck [19] in Trellis Coded Modulation). TCM also directly addresses the 6 dB/bit-dimension scaling and often gets a little better coding gain for the same number of states as convolutional coding. It therefore finds use in some systems. Wireless systems also exhibit fading effects where the TCM unfortunately loses performance relative to good convolutional codes. Appendix B addresses the mapping by set partitioning and some related codes, but it is not further pursued in this text's primary chapters. The convolutional codes however readily become components of Section 8.3's more powerful turbo codes while TCM may have issues in such concatenated systems. TCM thus appears in Appendix B for completeness, but finds increasingly less use.

8.1.8 Error probabilities and Performance

Section 7.2 generally investigated sequential decoders that use the trellis description of codes, which specializes to binary convolutional codes. Binary codes typically find use on either an AWGN or a BSC. For the AWGN when d_{min} or d_{free} dominate, the error probabilities are:

$$\bar{P}_e \approx \bar{N}_e \cdot Q\left(\frac{d_{min}}{2\sigma}\right) = \bar{N}_e \cdot Q\left(\sqrt{d_{free} \cdot \text{SNR}}\right) \tag{8.20}$$

$$\bar{P}_b \approx \frac{N_b}{b} \cdot Q\left(\sqrt{d_{free} \cdot \text{SNR}}\right) \quad . \tag{8.21}$$

For the BSC, the equivalent relations are

$$\bar{P}_e \approx \bar{N}_e \cdot [4p(1-p)]^{\lceil \frac{d}{2} \rceil} \tag{8.22}$$

$$\bar{P}_b \approx \frac{N_b}{b} \cdot [4p(1-p)]^{\lceil \frac{d}{2} \rceil} \quad . \tag{8.23}$$

When other larger distances contribute, the analysis applies by taking sums of the corresponding terms for the first few smallest distances in the formulas above.

8.2 Binary Coding Tables

This section characterizes implementation complexity and tabulates convolutional and block codes in successive subsections.

8.2.1 Convolutional-Code Tabulation

There are two basic convolutional-code implementations in practice: feedback-free implementations and systematic implementations (possibly with feedback). This subsection illustrates these implementations' $G(D)$ and $H(D)$. Any code may have a large number of implementations that differ in the exact mapping of input bit sequences to the output codewords. With feedback it is always possible to design a systematic encoder that directly passes $b = k$ bits to the output. Systematic (convolutional) encoders are not always possible without feedback.

Code tables specify the direct realization (without feedback) as in Subsection 8.2.1.3. For instance the rate 1/2 code $G(D) = [1 + D + D^2 \quad 1 + D^2]$ is an example with largest minimum-distance when $r = 1/2$ and $\nu = 4$ states. Section 8.1's direct realization is obvious. This encoder's conversion uses instead the input sequence

$$u'(D) = \frac{1}{1 + D + D^2} \cdot u(D) \quad , \tag{8.24}$$

which leaves $u'(D)$ in one-to-one correspondence with $u(D)$. Clearly $u'(D)$ can take on all possible causal sequences, just as can $u(D)$ and the two inputs correspond to different input sequences that map to the same output code sequences. Thus, the codewords are

$$\boldsymbol{v}'(D) = u'(D) \cdot G(D) = u(D) \cdot \frac{1}{1 + D + D^2} G(D) = u(D) \cdot \begin{bmatrix} 1 & \frac{1 + D^2}{1 + D + D^2} \end{bmatrix} \quad . \tag{8.25}$$

(8.25)'s new generator creates the same code (the same set of codewords). This new generator, or encoder, is systematic; but it requires feedback. Any systematic encoder has form

$$G_{sys} = [I_k \quad h(D)] \quad , \tag{8.26}$$

and $h(D)$ defines the parity matrix through

$$H_{sys} = [h^*(D) \quad I_{n-k}] \quad . \tag{8.27}$$

Thus for rate $r = \frac{n-1}{n}$ codes, the parity matrix easily converts to (8.27)'s form by simply dividing all elements by the last. Then (8.26) defines the systematic encoder's realization. Thus for the 4-state example $h(D) = \frac{1+D^2}{1+D+D^2}$, and Figure 8.10 shows the systematic encoder with two D memory elements and feedback.

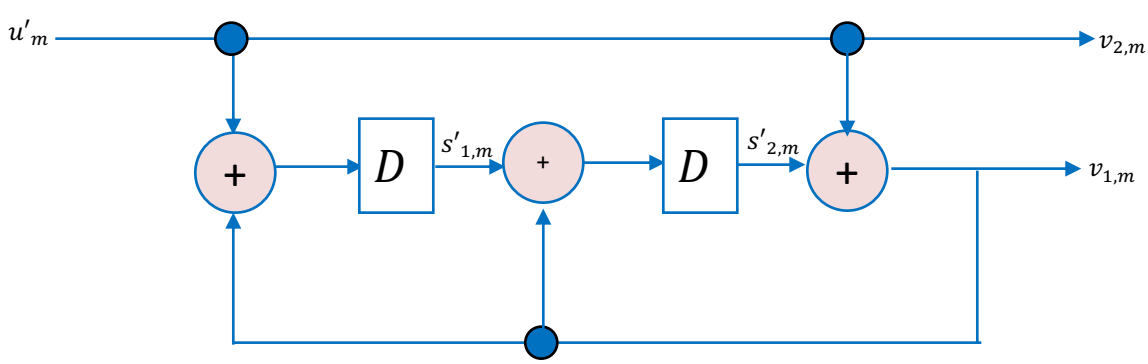

Figure 8.10: Systematic encoder of the 4-state convolutional code.

More generally for a convolutional code, the systematic realization finds the inverse of the first k columns of

$$G(D) = [G_{1:k}(D) \quad G_{k+1:n}(D)] \tag{8.28}$$

and premultiplies $G(D)$ by $G_{1:k}^{-1}$ to get

$$G_{sys}(D) = G_{1:k}^{-1}(D) \cdot G(D) = \left[I \quad G_{1:k}^{-1}(D) \cdot G_{k+1:n}(D)\right] \quad . \tag{8.29}$$

The inverse uses modulo-two arithmetic and of course the first k columns must form an invertible matrix for this conversion. This chapter's tabulated codes guarantee such invertibility. Premultiplication by the inverse does not change the codewords because this amounts to replacing rows of the generator by linear combinations of its rows in an invertible manner. Because the codeword set spans all possible row combinations of the rows of $G(D)$, this set is simply reindexed in terms of mappings to all possible input sequences by (8.29)'s matrix premultiplication.

Definition 8.2.1 *[Catastrophic encoder] A* **catastrophic encoder** *is one for which at least one codeword with finite Hamming weight corresponds to an input of infinite Hamming weight.*

Since the set of all possible codewords is also the set of all possible error events, a catastrophic encoder permits a finite number of decoding errors to correspond to an infinite number of input bit errors, clearly a "catastrophic" event. Catastrophic implementations are thus avoided. There are many tests for catastrophic codes, but one suffices here:

Lemma 8.2.1 *[Catastrophic Encoder Test] An encoder is non-catastrophic if and only if the greatest common divisor of the determinants of all the $k \times k$ submatrices of $G(D)$ is a nonnegative power of D – that is D^δ for $\delta \geq 0$.*

The proof is in Appendix B, but heuristically this prevents "pole cancelling" (infinite string) of an bit input sequence.

A non-catastrophic encoder always exists for any code, and all this section's tabulated encoders are non catastrophic. Trivially, a systematic encoder can never be catastrophic.

It is also possible to find any code's minimal encoder, which will be non catastrophic, and then to convert that into a systematic encoder through the method above. That systematic encoder will also be minimal and thus non catastrophic. For this reason, convolutional codes always have implementation with systematic encoders and minimal complexity.

8.2.1.1 Binary Codes' Coding Gain with the AWGN

Chapter 1's coding gain compares distance-to-energy for two systems, a coded system and an uncoded system. This performance measure is fair; from Chapter 1, two systems' fair comparisons should evaluate the 3 parameters $\bar{b}$, $\bar{\mathcal{E}}_{\boldsymbol{x}}$, and $\bar{P}_e$. Any two of the 3 can be fixed and the other compared. This requires cautious interpretation with convolutional codes. Convolutional coding gain fixes $\bar{b}$ and $\bar{\mathcal{E}}_{\boldsymbol{x}}$ and looks at performance in terms of d_{min}, which presumably directly relates to $\bar{P}_e$.

On the AWGN Channel, the free distance and minimum distance relate by

$$d_{min}^2 = d_{free} \cdot 4 \cdot \bar{\mathcal{E}}_{\boldsymbol{x}} \quad , \tag{8.30}$$

where the 4 comes from $\sqrt{\bar{\mathcal{E}}_{\boldsymbol{x}}} - (-\sqrt{\bar{\mathcal{E}}_{\boldsymbol{x}}}) = 2\sqrt{\bar{\mathcal{E}}_{\boldsymbol{x}}}$, so $d_{min}^2 = 4 \cdot \bar{\mathcal{E}}_{\boldsymbol{x}}$ for each AWGN dimension where subsymbols differ. So the Q-function argument for P_e calculation is

$$\frac{d_{free} \cdot \bar{\mathcal{E}}_{\boldsymbol{x}}}{1/r \cdot \sigma^2} = \underbrace{r \cdot d_{free}}_{\gamma} \cdot SNR \quad . \tag{8.31}$$

The denominator's $1/r$ factor recognizes the coded system needs either (1) a factor $1/r$ higher sampling rate $1/T' = W$, $W_{coded} = W_{uncoded}/r$, at fixed subsymbol period to get n dimensions where the compared uncoded system with $k < n$ uses fewer dimensions, or similarly (2) a factor $1/r$ longer subsymbol period $T_{coded} = T_{uncoded}/r$. With either of these (not both), the coded and uncoded system will then have the same $\bar{\mathcal{E}}_{\boldsymbol{x}}$ and $\bar{b}$. Thus, $\bar{\mathcal{E}}_{\boldsymbol{x}}$ effectively reduces by the factor r as energy spreads over more dimensions. $r = 1$ for the uncoded system. The **binary-code coding gain** is

$$\gamma \triangleq 10 * log10(r \cdot d_{free}) \quad . \tag{8.32}$$

This is then Chapter 1's coding gain γ and a fair comparison.

Specifically, the coded system has with subsymbol rate $1/T$ and corresponding (brickwall) bandwidth $W = 1/2T$ (corresponding to no intersymbol interference as in Chapter 3's simplest Nyquist pulse shape of $\varphi(t) = \frac{1}{\sqrt{T}} \cdot \text{sinc}(t/T)$. Then coded systems bits per $n = N = 2WT$ dimensions is

$$\bar{b}_{coded} = \frac{b}{N} = \frac{b}{2WT'} = \frac{b}{n} = \frac{r \cdot b}{k} = r \cdot \bar{b}_{uncoded} \quad , \tag{8.33}$$

so for equal $\bar{b}$, the coded system needs to increase the uncoded system's WT' to a coded system's WT'/r Similarly, the uncoded system's WT' to a coded system's WT'/r then causes the energy per dimension of the two systems to be equal:

$$\bar{\mathcal{E}}_{x,coded} = \frac{\mathcal{E}_{\boldsymbol{x}}}{N} = \frac{r \cdot \mathcal{E}_{\boldsymbol{x}}}{k} = r \cdot \bar{\mathcal{E}}_{x,uncoded} \ , \tag{8.34}$$

The energy per bit is

$$\mathcal{E}_b = \frac{\bar{\mathcal{E}}_{\boldsymbol{x}}}{\bar{b}} \tag{8.35}$$

is then also the same for coded and uncoded.

The Binary-Coding Theorist's Fallacy: When $W \to W/r$, the underlying assumption is that the "clock" rate, equivalently the number of dimensions n, can be freely increased on the AWGN. While theoretically true, this is often not possible in practice where systems have finite bandwidth. Finite bandwidth corresponds to the filtered AWGNs as studied in earlier chapters. For instance, if intersymbol interference is an issue, then increasing $1/T'$ worsens the ISI. In such systems, only the symbol period $T \to T/r$ can increase. What this means then is with ISI, the data rate reduces. **Full Stop** - the data rate reduces! Thus, to use the code with ISI, the data rate is lower, which is exactly the opposite of the original designer's intent. When there is no ISI, the coding gain $\gamma = r \cdot d_{free}$ has meaning. With ISI, it simply means that the uncoded system's data rate was too high, and the designer will lower that data rate to achieve accepaible performance. This is not what Chapter 4's water-filling teaches, which is that the rate can be increased to capacity reliably. This then causes the design to shift to multilevel constellations where energy grows exponentially with $\bar{b}$. Good designs then can use BICM (and Gray Code mapping) with longer symbol periods. BICM's interleaving (see Section 8.3.1) then prevents the exponential energy growth by creating a set of parallel independent approximately AWGN channels.

While this expansion of the binary code's coding gain to multilevel constellations then seems straightfoward, Subsection 8.1.7, and Figure 8.9 specifically, call attention to the discontinuity in r that necessarily must accompany the BICM multi-level constellation design's use. The coding rate r decreases (perhaps through puncturing or perhaps through use of a lower r code) each time the constellation size jumps by $\bar{b} \to \bar{b} + 1$. Thus, the coding theorist's fallacy occurs if the designer is not cognizant of this discontinuous design procedure that accompanies use of the formula $\gamma = r \cdot d_{free}$ in good design.

This section's tables list coding gain for convolutional codes, even though it implies a bandwidth increase. The codeword/sequence error probability again is

$$P_e \approx N_e \cdot Q\left[\sqrt{\gamma \cdot \text{SNR}_{uncoded}}\right] \quad . \tag{8.36}$$

A more accurate expression is

$$P_e \leq \sum_{d=d_{free}}^{\infty} N_d \cdot Q\left[\sqrt{\gamma \cdot \text{SNR}_{uncoded}}\right] \quad , \tag{8.37}$$

where N_d is the number of error events of weight d.

8.2.1.2 BSC Error Probability

The BSC has simplest approximation (using tighter symmetric B-Bound of Section 1.1) to error probability again as

$$P_e \approx \frac{N_e}{2} \cdot [4p(1-p)]^{\left\lceil \frac{d_{free}}{2} \right\rceil} \quad . \tag{8.38}$$

Similar to AWGN, the distance spectrum can be accommodated through

$$P_e \leq \sum_{d=d_{free}}^{\infty} \frac{N_d}{2} \cdot \left[\sqrt{4p(1-p)}\right]^d \quad , \tag{8.39}$$

where again N_d is the number of error events of weight d.

8.2.1.3 Convolutional Code Tables

This subsection tabulates several popular convolutional codes. These tables arise from a very exhaustive set found by R. Johannesson K. Sigangirov [6], but in which there appeared some inconsistencies. Using Chapter 7's Ginis dmin-calculation program, the tables here update some detailed parameters. The generators are specified in octal with an entry of 155 for $g_{11}(D)$ in Table 8.1 for the 64-state code corresponding to $g_{11}(D) = \underbrace{D^6}_{1} + \underbrace{D^5 + D^3}_{5} + \underbrace{D^2 + 1}_{5}$. This format appears also in Figure 8.6 as an example.

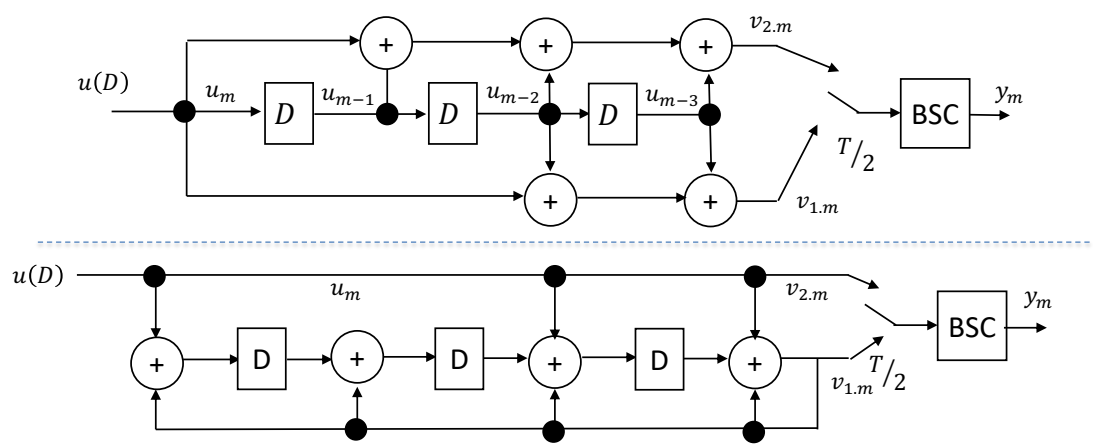

Figure 8.11: 8-State $r = 1/2$ equivalent encoders, with and without feedback, from Table 8.1

EXAMPLE 8.2.1 *[Rate 1/2 example]* The $\nu = 8$-state $r = 1/2$ code in the tables has generator [17 13]. The generator polynomial is thus $G(D) = [D^3 + D^2 + D + 1 \;\; D^3 + D + 1]$. Both feedback-free and systematic-with-feedback encoders appear in Figure 8.11. The quantity L_D is the length of the minimum-distance error event.

2^ν	$g_{11}(D)$	$g_{12}(D)$	d_{free}	γ	(dB)	N_e	N_1	N_2	N_b	L_D
4	7	5	5	2.5	3.98	1	2	4	1	3
8	17	13	6	3	4.77	1	3	5	2	5
16	23	35	7	3.5	5.44	2	3	4	4	8
(2G) 16	31	33	7	3.5	5.44	2	4	6	4	7
32	77	51	8	4	6.02	2	3	8	4	8
64	163	135	10	5	6.99	12	0	53	46	16
(802.11a) 64	155	117	10	5	6.99	11	0	38	36	16
(802.11b) 64	133	175	9	4.5	6.53	1	6	11	3	9
128	323	275	10	5	6.99	1	6	13	6	14
256	457	755	12	6	7.78	10	9	30	40	18
(3G) 256	657	435	12	6	7.78	11	0	50	33	16
512	1337	1475	12	6	7.78	1	8	8	2	11
1024	2457	2355	14	7	8.45	19	0	80	82	22
2048	6133	5745	14	7	8.45	1	10	25	4	19
4096	17663	11271	15	7.5	8.75	2	10	29	6	18
8192	26651	36477	16	8	9.0	5	15	21	26	28
16384	46253	77361	17	8.5	9.29	3	16	44	17	27
32768	114727	176121	18	9	9.54	5	15	45	26	37
65536	330747	207225	19	9.5	9.78	9	16	48	55	33
131072	507517	654315	20	10	10	6	31	58	30	27

Table 8.1: Rate 1/2 Maximum Free Distance Codes

2^ν	$g_{11}(D)$	$g_{12}(D)$	$g_{13}(D)$	d_{free}	γ	(dB)	N_e	N_1	N_2	N_b	L_D
4	5	7	7	8	2.67	4.26	2	0	5	3	4
8	15	13	17	10	3.33	5.23	3	0	2	6	6
16	25	33	37	12	4	6.02	5	0	3	12	8
32	71	65	57	13	4.33	6.37	1	3	6	1	6
64	171	165	133	15	5	6.99	3	3	6	7	11
128	365	353	227	16	5.33	7.27	1	5	2	2	10
256	561	325	747	17	5.67	7.53	1	2	6	1	9
512	1735	1063	1257	20	6.67	8.24	7	0	19	19	15
1024	3645	2133	3347	21	7	8.45	4	1	4	12	16
2048	6531	5615	7523	22	7.33	8.65	3	0	9	8	15
4096	13471	15275	10637	24	8.00	9.03	2	8	10	6	17
8192	32671	27643	22617	26	8.67	9.38	7	0	23	19	20
16384	47371	51255	74263	27	9	9.54	6	4	6	22	24
32768	151711	167263	134337	28	9.33	9.70	1	6	5	2	17
65536	166255	321143	227277	28	10	10	1	9	20	4	22

Table 8.2: Rate 1/3 Maximum Free Distance Codes

2^ν	$g_{11}(D)$	$g_{12}(D)$	$g_{13}(D)$	$g_{14}(D)$	d_{free}	γ	(dB)	N_e	N_1	N_2	N_b	L_D
4	7	7	7	5	10	2.5	3.98	1	1	1	2	4
8	17	15	13	13	13	3.25	5.12	2	1	0	4	6
16	37	35	33	25	16	4	6.02	4	0	2	8	7
32	73	65	57	47	18	4.5	6.53	3	0	5	6	8
64	163	147	135	135	20	5	6.99	10	0	0	37	16
128	367	323	275	271	22	5.5	7.40	1	4	3	2	9
256	751	575	633	627	24	6.0	7.78	1	3	4	2	10
512	0671	1755	1353	1047	26	6.5	8.13	3	0	4	6	12
1024	3321	2365	3643	2277	28	7.0	8.45	4	0	5	9	16
2048	7221	7745	5223	6277	30	7.5	8.75	4	0	4	9	15
4096	15531	17435	05133	17627	32	8	9.03	4	3	6	13	17
8192	23551	25075	26713	37467	34	8.5	9.29	1	0	11	3	18
16384	66371	50575	56533	51447	37	9.25	9.66	3	5	6	7	19
32768	176151	123175	135233	156627	39	9.75	9.89	5	7	10	17	21
65536	247631	264335	235433	311727	41	10.25	10.1	3	7	7	7	20

Table 8.3: Rate 1/4 Maximum Free Distance Codes

2^ν	$h_3(D)$	$h_2(D)$	$h_1(D)$	d_{free}	γ	(dB)	N_e	N_1	N_2
8	17	15	13	4	2.667	4.26	1	5	24
16	05	23	27	5	3.33	5.22	7	23	59
32	53	51	65	6	4.00	6.02	9	19	80
64	121	113	137	6	4.0	6.02	1	17	47
128	271	257	265	7	4.67	6.69	6	26	105
256	563	601	475	8	5.33	7.27	8	40	157
512	1405	1631	1333	8	5.33	7.27	1	20	75
1024	1347	3641	3415	9	6	7.78	9	45	166
2048	5575	7377	4033	10	6.67	8.24	29	0	473
4096	14107	13125	11203	10	6.67	8.24	4	34	127
8192	29535	31637	27773	11	7.33	8.65	9	72	222
16384	66147	41265	57443	12	8	9.02	58	0	847
32768	100033	167575	155377	12	8	9.02	25	0	462
65536	353431	300007	267063	12	8	9.02	7	26	120

Table 8.4: Rate 2/3 Maximum Free Distance Codes

2^ν	$h_4(D)$	$h_3(D)$	$h_2(D)$	$h_1(D)$	d_{free}	γ	(dB)	N_e	N_1	N_2
32	45	71	63	51	5	3.75	5.74	9	47	218
64	151	121	177	111	6	4.5	6.53	30	112	640

Table 8.5: Rate 3/4 Maximum Free Distance Codes

EXAMPLE 8.2.2 (Rate 2/3 example) The 16-state rate 2/3 code in the tables has a parity matrix given by [05 23 27]. The parity polynomial is thus $H(D) = [D^2+1 \;\; D^4+D^2+D \;\; D^4+D^2+D+1]$. Equivalently, $H_{sys}(D) = [\frac{D^2+1}{D^4+D^2+D+1} \;\; \frac{D^4+D^2+D}{D^4+D^2+D+1} \;\; 1]$ A systematic generator is then

$$G(D) = \begin{bmatrix} 1 & 0 & \frac{D^2+1}{D^4+D^2+D+1} \\ 0 & 1 & \frac{D^4+D^2+D}{D^4+D^2+D+1} \end{bmatrix} . \tag{8.40}$$

The systematic-with-feedback encoder appears in Figure 8.12.

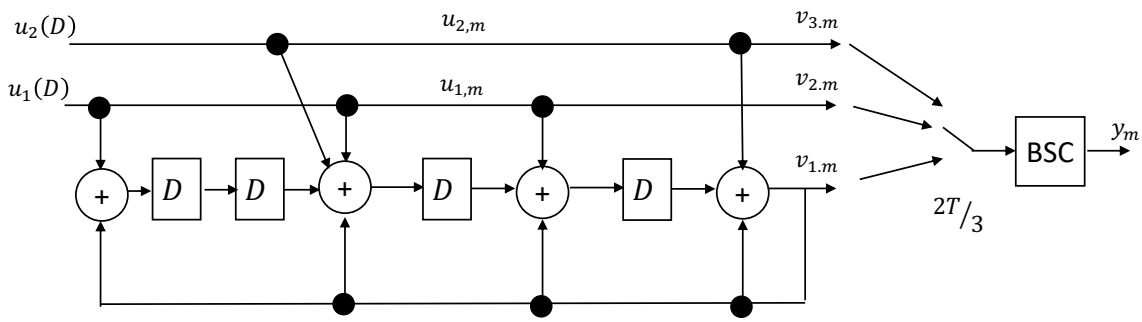

Figure 8.12: 16-State rate-2/3 equivalent encoders from Tables

8.2.1.4 Complexity

Code complexity measures the number of adds and compares that a decoder executes per symbol period by Chapter 7's Viterbi ML decoder. This number is always

$$N_D = 2^{\nu}\left(2^k + 2^k - 1\right) \tag{8.41}$$

for convolutional codes. This is a good relative complexity measure, but not necessarily an accurate count of an implemenation's instruction count or gate count.

8.2.1.5 The Code's Cyclic Prefix - Tail Biting

Block codes send independent codewords. However, convolutional codes' state depends on the last ν input $\boldsymbol{u}_m$ values, resembling an FIR channel's dependency on the last ν input samples; however, the encoder now has the memory, not the channel. Thus it is possible to use a "guard period" without Chapter 4's bandwidth penalty. Figure 8.13 illustrates this concept. The guard period (including a cyclic prefix as possibility) can have any value. Whatever that value, the trellis finishes in the same state as it starts. With the cyclic value (like all zeros), there is an information rate loss of ν/N symbols. However, unlike the cyclic prefix for DMT/OFDM, the known values can "prefeed" the encoder without any bandwidth loss; thus these values can be information (unlike Chapter 4's DMT/OFDM cyclic prefix). Thus, there is no information-rate reduction by forcing the common state. Coding theorists call this cyclic prefix "tail biting."

Chapter 7's sequential decoders then need only consider possible codeword packets that end in the same state as they began, which can improve decoder performance.

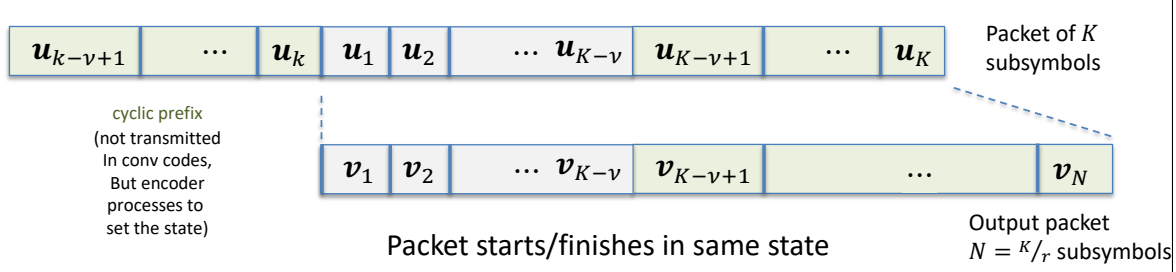

Figure 8.13: Illustration of Tail Biting.

8.2.2 Block Code Tabulation

8.2.2.1 Hamming Codes

Hamming Codes are among the simplest and yet still reasonably helpful binary block codes. Hamming codes have p parity bits and $n = 2^p - 1$, so then $k = 2^p - p - 1$. The rate is

$$r = \frac{2^p - p - 1}{2^p - 1} \quad , \tag{8.42}$$

which tends to $r = 1$ for large block length n. They can be constructed by the following steps

1. Enumerate indices $i = 1, ..., 2^p - 1$ as binary p-digit values to contstruct H.
2. For all indices with a single 1 element, create a parity bit - this isolated bit appears in only 1 syndrome element of $\boldsymbol{v} \cdot H^t = 0$; $H = [h \;\; I_p]$.
3. Rearrange the bits so it is systematic, which simplifies slightly finding $G = [I_k \;\; h^t]$.

No two rows can add to zero in $GF(2)$, but 3 rows clearly can add to zero so $d_{free} = 3$ for all Hamming codes. Thus, they correct one error. Rearrangement of H's columns can cause any syndrome computed to be the index of the single bit believed in error.

Figure 8.14 provides an example for $p = 3$ parity bits.

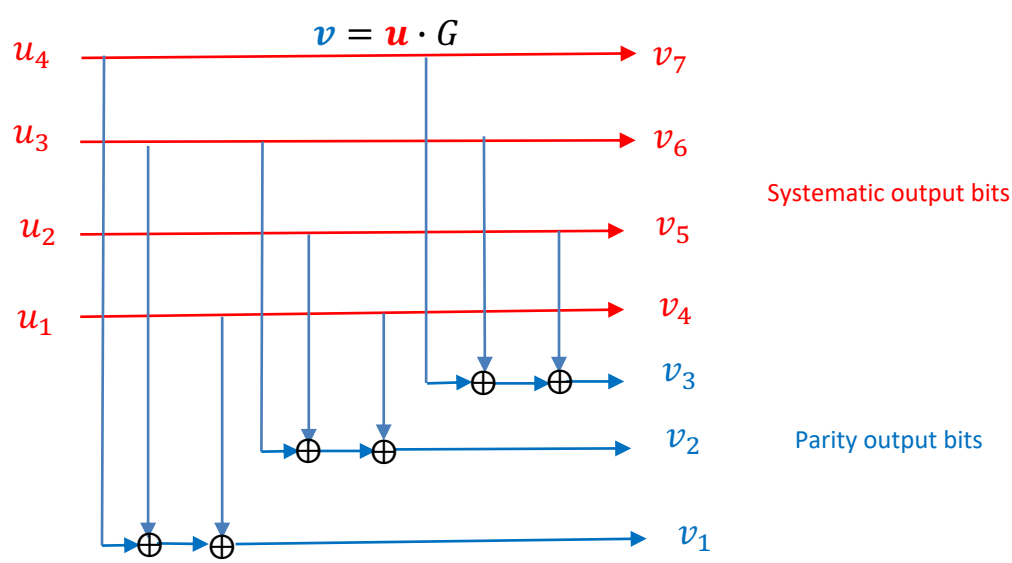

Figure 8.14: Hamming [7,3] Encoder Circuit.

The Generator is

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \quad , \tag{8.43}$$

while the corresponding parity matrix is

$$H = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \quad . \tag{8.44}$$

These can be generated in matlab with the following commands:

```
>> p=3;
>> [H,G]=hammgen(p);
>> G(end:-1:1,end:-1:1) % =
     1     0     0     0     1     0     1
     0     1     0     0     1     1     1
     0     0     1     0     1     1     0
     0     0     0     1     0     1     1
>> H(end:-1:1, end:-1:1)
     1     1     1     0     1     0     0
     0     1     1     1     0     1     0
     1     1     0     1     0     0     1
```

These recognize that permutations of rows or columns (here an index reversal relative to this text's convention) does not change the code's properties.

Expanded Hamming Codes Expanded Hamming codes add one extra overall parity bit that sums all the rest. They form by

1. add a column of zeros to the H matrix, and then
2. add a row of all ones to the H matrix.

For instance

$$H_{exp} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} . \tag{8.45}$$

To find the systematic version with original Hamming parity matrix H:

```
>> Hext=[H , zeros(3,1) ;(ones(1,8))]  % =
     1     0     0     1     0     1     1     0
     0     1     0     1     1     1     0     0
     0     0     1     0     1     1     1     0
     1     1     1     1     1     1     1     1
>> Hprime=inv(gf(Hext(1:4,1:4)))*gf(Hext) % =
   1   0   0   0   1   1   0   1
   0   1   0   0   0   1   1   1
   0   0   1   0   1   1   1   0
   0   0   0   1   1   0   1   1
>> Hsys=[Hprime(1:4,5:8) Hprime(1:4,1:4)]
   1   1   0   1   1   0   0   0
   0   1   1   1   0   1   0   0
   1   1   1   0   0   0   1   0
   1   0   1   1   0   0   0   1
```

The systematic G then follows directly.

8.2.2.2 Hadamard Codes

Hadamard codes are low rate, linear, and intended for use with very low SNR (or high $\bar{P}_b$ systems. They resemble binary orthogonal signal constellations in that all codewords are mutually orthogonal in $GF(2)$. They have $n = 2^k$ and thus $p = 2^k - k$ parity bits. The code rate is

$$r = \frac{k}{2^k} \tag{8.46}$$

and tends to zero as n becomes large. Thus, Hadamard Codes make more sense for low rate designs than would Hamming Codes. The Hadamard generator can be constructed by enumerating in binary the numbers $0...2^k - 1$ as column vectors. For example for $n = 16$ and $k = 4$:

```
n=16;
k=log2(n);
Gtemp=dec2bin(0:2^k-1)';
  G=zeros(k,n);
  for i=1:k for j=1:n
      G(i,j)=bin2dec(Gtemp(i,j));
  end ; end
>> G % =
     0     0     0     0     0     0     0     0     1     1     1     1     1     1     1     1
     0     0     0     0     1     1     1     1     0     0     0     0     1     1     1     1
     0     0     1     1     0     0     1     1     0     0     1     1     0     0     1     1
     0     1     0     1     0     1     0     1     0     1     0     1     0     1     0     1

% 4 x 16 (n=16, k=4, dfree is 8)

gf(G)*gf(G)' % =
   0   0   0   0
   0   0   0   0
   0   0   0   0
   0   0   0   0
```

Note the generator matrix rows are all orthogonal to each other, and indeed in $GF(2)$ even to themselves! A systematic generator follows by rearranging column indices so that those columns with a single 1 in them appear as an identity in the last k positions, $G_{sys} = [I\ h^t]$. The remaining $n-k$ columns then form h^t and $H_{sys} = [h\ I]$.

Augmented Hadamard Codes: The Augmented Hadamard Code has slightly higher rate with $n = 2^m$, $k = m+1$, and $d_{free} = n/2$. The Augmented Hadamard Code can be found from the "next size up" Hadamard code by retaining only the columns that have 1 at the top. For the example above:

```
GA=G(:,8:16) =
     0     1     1     1     1     1     1     1     1
     1     0     0     0     0     1     1     1     1
     1     0     0     1     1     0     0     1     1
     1     0     1     0     1     0     1     0     1
% 4 x 8   (n=8, k=4, m=3), dfree is now 4

>> gf(GA)*gf(GA)' % =
   0   0   0   0
   0   1   1   1
   0   1   1   1
   0   1   1   1
```

The codewords are no longer orthogonal, but it remains a slightly better rate code with the same free distance. The Augmented Golay Code can be found to be the dual of the Expanded Hamming Code with $n/2$.

8.2.2.3 Matlab encode.m and decode.m functions

Matlab has an encode function for linear (or cyclic) codes that uses:

```
codeword = encode(inbits, n,k, 'linear', G)
```

The word "cyclic" replaces "linear" for cyclic codes (which are also linear as in Section 8.4) but can be more simply encoded (and decoded) than the general linear code. For a Hamming Code, "hamming" can replace "linear" and no G is needed. Similarly there is the corresponding decode function:

```
msgbits = decode(y, n,k, 'linear', G)
```

8.2.2.4 Golay Codes

to be added

8.2.2.5 Product Codes

8.3 Longer-Length Binary Codes

While Section 8.2's binary codes improve performance significantly, they do not result in designs that approach transmission at capacity reliably. More powerful codes, including when mapped through BICM and a Gray Code to higher-level QAM constellations, need longer block lengths. Such long block lengths ultimately lead to impractical ML decoder implementation. Yet, there are more powerful codes that use some degree of "controlled randomness" to perform better. These encourage good, but suboptimal, decoders. Figure 8.15 illustrates such controlled randomness in the form known as a concatenated code. An example is Turbo Codes that interleave two (or more) relatively simple convolutional encoders' output streams so that errors occurring in one code's ML decoder are unlikely to correspond to errors in the other code's ML decoder. As Figure 8.15 also sketches, Section 7.3's Iterative decoding allows the two decoders to exchange (suboptimally) decoding information, specifically log likelihood ratios computed for each information bit. Figure 8.15's decoder-information exchange improves the overall performance with respect to either short-length code. Good iterative decoders approach the overall ML decoder's performance with less complexity.

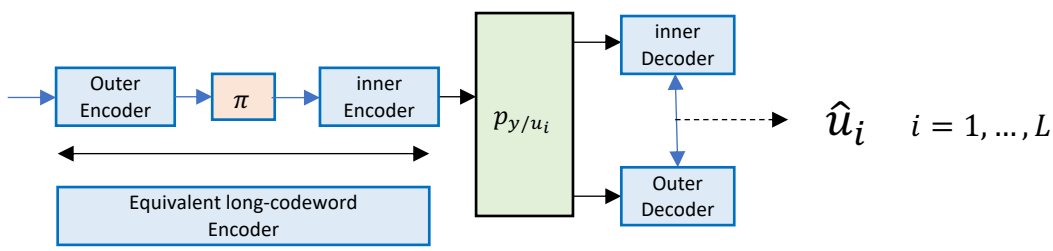

Figure 8.15: Basic Code Concatenation.

Interleaving specfies an order $m' = \pi(m)$ for a group of L subsymbols, where L is the interleaver period. Typically, L exceeds the block, or survivor, length for either of the concatenated codes. Interleaving uses the same order functionality as Chapter 5's order function π, just not for user reordering in the concatenated-code context; to avoid confusion the name interleave characterizes this single user application.

There are two somewhat distinct uses of interleaving:

Iterative Decoding Separation of individual codeword bits from one another for BICM or concatenated binary codes, such as Section 8.3.2's Turbo Codes, so that different codes' can assist one another. In BICM, one of those codes is the gray mapping to constellation points; more generally iterative decoding with interleaving exploits soft information. This interleaving is Subsection 8.3.1's random interleaving.

Mitigation of burst/Implusive channel events This form of interleaving handles events outside the stationary channel model that change too quickly for the system to react. It reorders subsymbol level (like bytes) in a block or convolutional/triangular fashion to spread nonstationary events' effects over many codewords, introducing more significant processing delay. The concatenation of a "outer" hard-decoded code with high rate $r = K/N$ (subsymbols/subsymbols ratio) to take an inner system perhaps operating at as yet insufficiently low P_e and/or $\bar{P}_b$ (perhaps 10^{-3} or 10^{-4}) to a very low, nearly 0, error probability. This concatenation and interleaving essentially achieves Shannon's promise of $P_e \to 0$ and where even small coding gain in dB substantially reduces P_e, essentially with a steep drop in P_e with very little further data-rate loss (but with latency increase). Section 8.6 discusses this type of interleaving and its use with Hard-Soft code concatenation (which can even include retransmission of blocks/packets in which an error has been detected).

Subsection 8.3.1 reviews random interleaving, which can be used with Turbo Codes and/or with bit-interleaved coded modulation (BICM). BICM exploits random interleaving's creation of parallel

independent "bit channels," as in Chapter 2's BICM introduction. With Gray code constellation-bit mapping, essentially the relation $r \cdot d_{free} \cdot SNR = \gamma \cdot SNR$ expands to characterize multilevel transmission. This reduces the Turbo Code's N_d values significantly, lowering the achieved $\bar{P}_b$. This also helps BICM de-correlate the effect of a large noise sample that affects multiple bits within a single multi-level SQ QAM constellation. Subsection 8.3.2 then proceeds to tabulate some well-known and heavily used Turbo Codes. Subsection 8.3.3 progresses to block binary codes (to which BICM also applies) that create long block length by essentially randomly picking the code's parity matrix entries. These are called "Low Density Parity Check" (LDPC) codes. Section 8.3.3 tablulates various well-known/used LDPC coding schemes. LDPC decoders use Section 7.4's implementable constraint message passing in iterative decoding that again approximates ML performance. Both Turbo and LDPC codes find heavy use with BICM in high-performance communication systems. A third type of long-block-length code are Section 8.3.4's Polar Codes, which are essentially the finite-field equivalents of Chapter 5's GDFE in that they are canonical (can have arbitrarily small P_e at rates very close to capacity for given AWGN channel and SNR) with successive decoding - so Polar codes are not designed for iterative decoding, but rather successive decoding. Unfortunately, the Polar Codes gains do not (as yet) expand to multilevel QAM systems because their intended successive decoding complicates BICM's interleaving. This chapter does not pursue another long-standing approach to good codes (not quite as good as those here, but will relatively simple decoders) known as mapping-by-set-partitioning, which instead Appendix B discusses.

8.3.1 Random Binary Interleaving

Random interleaving ideally dispurses each concatenated encoders' regular error event patterns, making failure of both decoders less likely if they share decoding information. The associated interleaving rule $\pi(k)$ spans a longer period L than codeword length $N << L$ for block codes and $L_{error-event} << L$ for convolutional codes. Section 8.3's Turbo Codes use random interleaving. The **uniform random interleaver** is an abstraction that averages over many statistical possibilities for an interleaver with large period L. For any pattern of l positions (the positions may be viewed as the location of 1's), random interleaving may place those positions into

$$L_l = \binom{L}{l} \tag{8.47}$$

possibile patterns. An implementable deterministic interleaver could only approximate such an effect. However, over an interleaver ensemble (essentially making the interleaver cyclostationary over the interleaver "period" L), the uniform random interleaver can be hypothesized. The uniform random interleaver helps ensemble-analyze interleaved coding systems' performance.

8.3.1.1 BICM Revisit

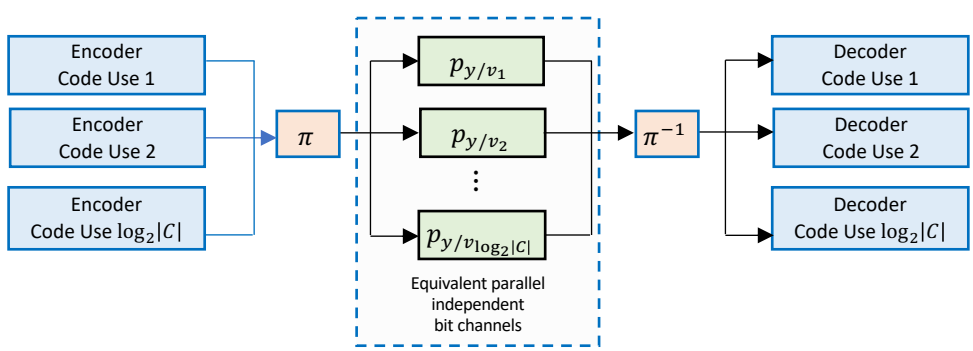

Figure 8.16: BICM's creation of parallel independent bit channels.

BICM uses random interleaving to represent a Gray-Coded PAM/QAM constellation by parallel independent bit channels as in Figure 8.16. The ML detector for individual bits maximizes

$$\hat{u}_i = \arg \max_{u_i} p_{\boldsymbol{y}/u_i} \tag{8.48}$$

which will average the influence of adjacent same-subsymbol bits $u_{j\neq i}$ in its construction. In Chapter 7's iterative decoding methods, the corresponding soft information (e.g., log-likelihood ratio LLR) also depends on adjacent bits from the same PAM/QAM subsymbol. When that subsymbol is multilevel ($|C| > 2$) and these bits are from the same codeword, they will instead be co-dependent. Thus the presence of a single large noise sample can influence in a dependent way multiple bits. Random interleaving largely causes these adjacent same-subsymbol bits to correspond to different codewords. This recaptures the independence that occurs when $|C| = 2$ with binary codes' use on an AWGN.

8.3.1.2 Concatenated code interleaving use

Random interleaving also creates, in separate use from any BICM, a long block length for the concatenated codes when viewed as a single code. Codes selected randomly, as long as the block length is very long, can achieve capacity (as in Section 2.3). Random interleaving approximates this code-design randomness with implementable decoder complexity, but presumes Chapter 7's iterative decoding's use to combine the two codes' decoder results. The number of ways in which decoders make specific errors essentially reduces because it is unlikely that a specific error pattern for one code's low-distance error event also just happens to touch the right places to cause the other decoder also to err. The individual interleaved codes' d_{min} value becomes less important with respect to the code neighbor/distance spectrum of $\{N_{e,i}\ ,\ d_i\}$ In this way $P_e \approx \sum_{i=0}^{\infty} N_{e,i} \cdot Q\left(\frac{d_i}{2\sigma}\right)$, which often has larger $N_{e,i>0}$ that contribute in combination more significantly to error probability than the single term $N_e = N_{e,0}$ term, reduces over the ensemble of all significant error-probability-contributing terms. Subsection 8.3.2 analyzes turbo codes with presumed uniform random interleaver.

An ideal random uniform interleaver then translates any pattern of l ones to have probability, L_l^{-1}. This subsection lists 4 random interleaver types that approximate uniform random interleaving, and each may find use under different application details.

8.3.1.3 Berrou- Glavieux Block Interleavers

The Berrou- Glavieux interleaver has period $L = K \cdot J = 2^i \cdot 2^j$, which is a power of 2 and the product of parameters K and J. It uses eight prime numbers:

m	1	2	3	4	5	6	7	8
p_m	17	37	19	29	41	23	13	7

The time index is $k = 0, ..., L-1$. Recall that $(\cdot)_M$ means the quantity in brackets modulo M, i.e., the part left over after subtracting the largest contained integer multiple of M. For each time sample, this interleaver uses parameters $r_0 = (k)_J$, $c_0 = (k - r_0)/J$, and $m = (r_0 + c_0)_8$. For any single choice of the eight primes, these two time-index k quantities are:

$$r(k) = (p_{m+1} \cdot (c_0 + 1) - 1)_K \tag{8.49}$$

$$c(k) = ((K/2 + 1) \cdot (r_0 + c_0))_J \quad . \tag{8.50}$$

Then, the order function is

$$\pi(k) = c(k) + J \cdot r(k) \quad . \tag{8.51}$$

This interleaver has a long period for reasonable values of K and J and causes a robust positioning randomness of one code's error events with respect to other. An event that results from exceeding free/minimum distance in one constituent code is very unlikely to also be in just the right places after de-interleaving to exceed the free/minimum distance of another code. This reduces the number of

free/minimum distance and larger events. At lower SNR when many single-decoder's error events are more likely, this error-event position redistribution dominates $\bar{P}_b$ reduction. As SNR increases with Gray-code mapping to multilevel SQ QAM (or PAM) constellations ($|C| > 2$), this disbursements applies also to the $n \geq d/2$ events occurring at the bit levels corresponding to adjacent constellation values. The former error-event redistribution was Berrou's insight in developing Turbo Codes (see Section 8.3.2.2).

8.3.1.4 JPL (Jet Propulsion Laboratory) Block Interleaver

The JPL interleaver expands to any $K \in \mathbb{Z}^+$ and even $J \in \mathbb{Z}^+$, such that $L = K \cdot J$ with primes:

m	1	2	3	4	5	6	7	8
p_m	31	37	43	47	53	59	61	67

Again the time index is $k = 0, ..., L-1$. Defining $r_0 = \left(\frac{i-m}{2} - c_0\right)_J$, $c_0 = \left(\frac{i-m}{2}\right)_J$, and $m = (r_0)_8$, and

$$r(k) = (19 \cdot r_0)_{\frac{K}{2}} \tag{8.52}$$

$$c(k) = (p_{m+1} \cdot c_0 + 21 \cdot (k)_2)_J \quad , \tag{8.53}$$

the JPL interleaver order function is

$$\pi(k) = 2 \cdot r(k) \cdot K \cdot c(k) - (k)_2 + 1 \quad . \tag{8.54}$$

The same insights as the earlier interleaver apply. Best interleaver choice may depend more on second-order effects that deviate from the pure AWGN channel model like a wireless fading channel. An adaptive (as far as this author knows until corrected) interleaver that uses machine learning to adapt interleavers to channel specifics awaits invention.

8.3.1.5 Pseudorandom Interleavers

Pseudorandom interleavers use pseudorandom binary sequences (PRBS). Such sequences in turn use maximum-length polynomials (See Appendix B). A PRBS circuit is a rate one convolutional code with constant 0 input and with feedback based on a maximum-length polynomial $P(D)$ with implementation $G(D) = \frac{1}{p(D)}$. The degree-ν polynomial is chosen so that with binary arithmetic, it has no nontrivial factors (i.e., it is "prime") and has other properties not discussed here. Table 8.6 lists maximum-length polynomials[3]. These polynomials find use in synchronization and "scrambling" of input bit streams to appear "white." Chapter 6 illustrates choice of a polynomial and circuit implementation, so this chapter does not repeat it here. Such circuits (if initialized with a nonzero initial condition) generate a periodic sequence of period $L \leq 2^\nu - 1$ that thus necessarily must include every nonzero binary pattern of length ν bits at the maximum L value exactly once per period.

[3]See [11], which has a far more complete listing of all the possible polynomials for each ν (there are many). The polynomials in x in that reference (as well as most coding books) corresponds to an advance, not a delay, which is why an expert reader might note a reversal of Table 8.6's polynomials.

ν	$P(D)$
2	$1 + D + D^2$
3	$1 + D^2 + D^3$
4	$1 + D^3 + D^4$
5	$1 + D^3 + D^5$
6	$1 + D^5 + D^6$
7	$1 + D^6 + D^7$
8	$1 + D^4 + D^5 + D^6 + D^8$
9	$1 + D^5 + D^9$
10	$1 + D^7 + D^{10}$
11	$1 + D^9 + D^{11}$
12	$1 + D^6 + D^8 + D^{11} + D^{12}$
13	$1 + D^9 + D^{10} + D^{12} + D^{13}$
14	$1 + D^9 + D^{11} + D^{13} + D^{14}$
15	$1 + D + D^{15}$
16	$1 + D + D^3 + D^{12} + D^{16}$
17	$1 + D^3 + D^{17}$
18	$1 + D^7 + D^{18}$
23	$1 + D^5 + D^{23}$
24	$1 + D^{17} + D^{22} + D^{23} + D^{24}$
31	$1 + D^{28} + D^{31}$

Table 8.6: A list of maximum-length (primitive) polynomials.

The pseudorandom interleaver uses the PRBS to specify the output position of each input bit. Each successive PRBS-circuit output bit specifies an address $\pi(k)$ for the specific message-bit's position. The interleaver discards any address that exceeds the interleaver period (when $L < 2^{\nu-1}$) and recyles the PRBS circuit. The de-interleaver regenerates the same sequence and then successively restores the position to $\pi^{-1}(\pi(k)) = k$.

8.3.1.6 S-Random Interleavers

An S-random interleaver spaces its input's adjacent symbols further than $S \in \mathbb{Z}^+$ (a positive integer) after interleaving. This approximates also the random re-distribution of the uniform random interleaver. The S-random interleaver first choses an $S \leq \sqrt{\frac{L}{2}}$ and executes Figure 8.17's algorithm. This algorithm converges if the condition on S is met and the second integer S' is ignored. Usually designers run the algorithm several times increasing S' until they find the largest such value for which the algorithm converges.

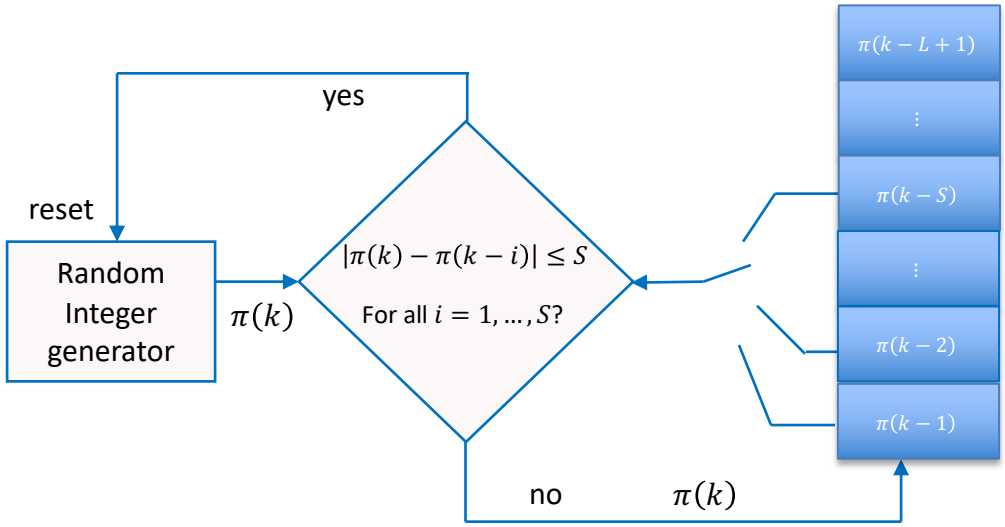

Figure 8.17: S-random interleaver illustrated.

The S-Random Interleaver's correlation between Chapter 7's iterative-decoding intrinsic and extrinsic information decays exponentially with the difference between interleaver indices (and has the exception value zero for time difference of 0).

8.3.2 Turbo Codes

Turbo Codes are parallel or serial concatenations of simple good convolutional codes with significant interleaving, first discovered by Claude Berrou[4] in 1993[16]. The name "Turbo" follows the iterative decoder's resemblance to a turbo-charged engine, reusing iteratively a resource (information for the code instead of exhaust-fuel for the engine) Turbo codes use Interleaving to a reduce two concatenated codes' nearest neighbor counts. Turbo Code design targets a data rate that is less than capacity, but perhaps just slightly so. The Turbo Code's interleaver is in addition to any interleaver used for BICM with the Turbo Code when $|C| > 2$. Subsection 8.3.2.1 investigates Turbo Codes' concatenation, while Subsection 8.3.2.2 further discusses code puncturing to implementing the desired binary code-rate r. Error-probability approximations appear in Subsections 8.3.2.3 and 8.3.2.4 for parallel and serial concatenations respectively. Subsections 8.3.2.5 and 8.3.2.6 enumerate various convolutional codes for turbo-code use.

8.3.2.1 Turbo-Code rate definition

Definition 8.3.1 *[Turbo Code] A* **Turbo Code** *is a parallel or serial concatenation of two binary convolutional codes with uniform random interleaving (or an approximation to it) to distribute the two codes' error events with respect to one another. Turbo codes are designed with the expectation that the receiver uses iterative (i.e., "turbo") decoding.*

[4]Claude Berrou, (1951-) is a French mathematician and professor of telecommunications at the École nationale supérieure des télécommunications de Bretagne (ENST Bretagne). Berrou initiated the belief that indeed capacity-achieving codes could be implemented practically.

parallel concatenations: Two systematic convolutional codes' parallel concatenation has rate (where the information bits are sent only once, along with all corresponding parity bits) is[5]

$$\frac{1}{r} = \frac{1}{r_1} + \frac{1}{r_2} - 1 \quad . \tag{8.55}$$

The subtracted term -1 term is eliminated if the two binary codes are not systematic.

A few examples illustrate the possibilities:

EXAMPLE 8.3.1 *[basic rate 1/3]* If two convolutional codes both have rate $r_1 = r_2 = .5$, then $r = 1/3$.

EXAMPLE 8.3.2 *[rate 1/6]* If two different systematic encoders both have rates $\bar{b}_1 = 1/3$ and $\bar{b}_2 = 1/4$, then $\bar{b} = 1/6$.

EXAMPLE 8.3.3 *[higher rates]* Higher rate turbo codes can be constructed from higher-rate convolutional codes. If two systematic encoders both have rate $\bar{b}_1 = \bar{b}_2 = 3/4$, then $\bar{b} = 3/5$. Simarly, if two encoders both have rate $\bar{b}_1 = \bar{b}_2 = .8$, then $\bar{b} = 2/3$.

Convolutional codes with high r and good distance properties are rate or very complex, so Subsection 8.3.2.2's puncturing becomes the preferred alternative to implement Subsection 8.3.2.5's high-rate turbo codes.

Serial concatenations: Serially concatenated turbo codes have rate

$$r = r_1 \cdot r_2 \quad . \tag{8.56}$$

A serial turbo code constructed from two rate 1/2 codes would have rate 1/4. Similarly a serial turbo code from two rate 2/3 codes has rate 4/9 (or less than 1/2). Clearly serial concatenation requires very high rates for the two used codes if the concatenation is to be high rate. The puncturing of the Subsection 8.3.2.2 again reduces serial concatenation's rate loss.

8.3.2.2 Puncturing

Subsection 8.1.6 introduced puncturing. Figure 8.18 illustrates puncturing for an $r = 1/3$ turbo code that results from parallel concatenation of two $r = 1/2$ convolutional codes. Puncturing restores the turbo code to $r = 1/2$ by alternately deleting one of the two parity bits. Some performance loss with respect to rate-1/3 turbo coding might be expected, but of course at an increase in data rate. Often, the resultant higher-rate code is still a very good code at the new higher data rate.

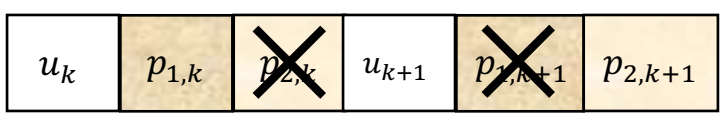

Figure 8.18: Puncturing of rate 1/3 turbo (or convolutional) code to rate 1/2.

Figure 8.19 achieves a yet higher data rate: A frame of 12 bits, 4 information and 8 parity, retains only 2 of the parity bits to increase the data rate to 2/3. This puncturing alternately deletes 1 of the 2 parity bits, or both bits, generated at each (sub-) symbol period.

[5] More generally, if $1/r_i$ is the number of output bits per input bit for code i, then the total number of output bits/input bits is $\sum_i r_i^{-1}$. When systematic, the input bits need not be repeated so the subtracted term (-1) becomes "subtract the number of codes less one".

Figure 8.19: Puncturing of rate 1/3 turbo (or convolutional) code to rate 2/3.

Generally, a systematic code with k information bits per symbol and $n-k$ parity bits per symbol can puncture to a higher rate q/p by accepting kq input bits and deleting $nq-kp$ of the parity bits,

$$\frac{kq}{nq-(nq-kp)} = \frac{q}{p} \quad . \tag{8.57}$$

The deleted Turbo-Code bits may not be the same for each successive symbol to "distribute" the loss of parity, so j successive symbols may be used, as in Figure 8.19. In this most general case, puncturing can be described by the $nqj \times kpj$ singular permutation/generator matrix G_{punc}. Figure 8.18's puncturing has $j=2$, $k=1$, $n=3$, $q=1$ and $p=2$

$$G_{punc} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad , \tag{8.58}$$

accepting 6 bits and outputting 4. The transmitter generates the punctured codewords by multiplying the original encoder's output code sequence by G_{punc}, or $\boldsymbol{v}_{punc}(D) = \boldsymbol{v}(D) \cdot G_{punc}$ where $\boldsymbol{v}(D)$ may be written in terms of "stacking" j successive symbols. Puncturing can avoid excessive constellation expansion when applying turbo codes to multilevel Gray-coded PAM/QAM transmission systems.

8.3.2.3 Bit-error-probability analysis: parallel concatenation

Turbo Codes' error-probability analysis presumes Section 8.3.2.2's random interleaver. This subsection begins with Chapter 2's 4-state rate-1/2 code. A realization of this code has $r = 1/2$ systematic generator

$$G_1(D) = \begin{bmatrix} 1 & \dfrac{1+D+D^2}{1+D^2} \end{bmatrix} \quad . \tag{8.59}$$

The corresponding parity matrix $H(D)$ reverses these two columns (or effectively "flips" the encoder's two output bits' order)[6] two columns here reversed in order with respect this well-known 4-state code's nominal generator.

Clearly the output-bit reordering changes no code-distance property (although the number of input bit errors and consequent encoder mappings differ). A parallel concatenated Turbo Code has uses two instances of this same code. For either instance, any input bit stream with weight $w = 1$ (or only one 1 in the stream so $u(D) = D^m$ where m is some integer) leads to an infinite-weight string of output 1's[7], which means that for this encoder the probability of a single input-bit error occuring is essentially zero. Two-bit-error input-bit error events are thus of more interest. A length-2 input sequence $1+D^2$ produces a single code instance's output sequence $v(D) = \begin{bmatrix} 1+D^2 & 1+D+D^2 \end{bmatrix}$, which is also the minimum-distance ($d_{free} = 5$) output-bit error-event. The $r = 1/3$ parallel-concatenation Turbo Code's decoder has an error if this two-bit input error event occurs for one code instance and

[6]The generator matrix differs from another choide of $G(D) = \begin{bmatrix} 1 & \frac{1+D^2}{1+D+D^2} \end{bmatrix}$ with corresponding $H(D) = \begin{bmatrix} \frac{1+D^2}{1+D+D^2} & 1 \end{bmatrix}$ that often represents this code. This second form appears later in this section's analysis.

[7]An infinite number of channel errors must occur for only 1 input bit error.

then also after de-interleaving on the other. The second code's error then corresponds to $D^m \cdot (1 + D^2)$ for any of $m = 0, ..., L - 1$ within the period L of the interleaver. For the uniform random interleaver, this simultaneous event occurs with probability

$$\binom{L}{2}^{-1} = \frac{2}{L \cdot (L-1)} \quad , \tag{8.60}$$

for each particular value of m. It thus has probability $L \cdot \frac{2}{L \cdot (L-1)} = \frac{2}{L-1}$ that it could occur for any of the L values $m = 0, ..., L - 1$. Furthermore, the rate-1/3 concatenated code has a $d_{free} = 8 = 2 + 3 + 3$ (and corresponds to essentially $v_{turbo}(D) = \left[1 + D^2 \;\; 1 + D + D^2 \;\; \pi(1 + D + D^2)\right]$ where π has been used loosely to denote that reordering of the 3 parity bits from the second code's interleaved use occurs in 3 different positions. The overall concatenated coding gain is $8/3 = 2.67 = 4.26$ dB. The original "mother" code had gain $5/2 = 2/5 = 3.97$ dB, and thus the Turbo Code's gain improves by only .3 dB [8].

distance	in-ev code 1	in-ev code 2	$b \cdot N(b,d)$ $b = 2$	d_{free} construct	$\frac{4 \cdot N(2,d)}{100}$	$\frac{4 \cdot N(2,d)}{1000}$	$\frac{4 \cdot N(2,d)}{10000}$
8	$1 + D^2$	$D^m \cdot (1 + D^2)$	2	2+3+3	.04	.004	.004
9	$1 + D^2$	$D^m \cdot (1 + D^4)$	4	2+3+4	.08	.08	.008
	$1 + D^4$	$D^m \cdot (1 + D^2)$		2+4+3			
10	$1 + D^2$	$D^m \cdot (1 + D^6)$	6	2+3+5	.12	.012	.0012
	$1 + D^4$	$D^m \cdot (1 + D^4)$		2+4+4			
	$1 + D^6$	$D^m \cdot (1 + D^2)$		2+5+3			
11	$1 + D^2$	$D^m \cdot (1 + D^8)$	8	2+3+6	.16	.016	.0016
	$1 + D^4$	$D^m \cdot (1 + D^6)$		2+4+5			
	$1 + D^6$	$D^m \cdot (1 + D^4)$		2+5+4			
	$1 + D^8$	$D^m \cdot (1 + D^2)$		2+6+3			
12	$1 + D^2$	$D^m \cdot (1 + D^{10})$	10	2+3+7	.20	.02	.002
	$1 + D^4$	$D^m \cdot (1 + D^8)$		2+4+6			
	$1 + D^6$	$D^m \cdot (1 + D^6)$		2+5+5			
	$1 + D^8$	$D^m \cdot (1 + D^4)$		2+6+4			
	$1 + D^{10}$	$D^m \cdot (1 + D^2)$		2+7+3			
t	$1 + D^2$	$D^m \cdot (1 + D^{2(t-8)})$	2(t-7)		.04(t-7)	.004(t-7)	.0004(t-7)
	$\vdots$	$\vdots$		$\vdots$			
	$1 + D^{2(t-8)}$	$D^m \cdot (1 + D^2)$					

Table 8.7: $G_1(D) = (1 + D + D^2)/(1 + D^2)$'s $b = 2$ input-bit-error and corresponding codeword-error events.

However, the turbo code's nearest neighbor coefficient N_e is smaller. Because two input-bit errors occur when the noise's error-event projection exceeds half the code's minimum distance, then this Turbo Code's $d_{free} = 8$ error-event probability is

$$\bar{P}_b(d_{free}) \approx \frac{2}{L-1} \cdot b \cdot \overline{N}(b, d_{free}) \cdot Q(\sqrt{d_{free} \cdot \text{SNR}}) \tag{8.61}$$

$$\approx \frac{4}{L-1} \cdot 1 \cdot Q(\sqrt{8 \cdot \text{SNR}}) \quad . \tag{8.62}$$

There are no 3-input-bit error events for the mother convolutional code but there are 4-input-bit error events and other yet-higher even numbers of input bit errors. Table 8.7 from [17] lists these error-event combinations and shows $\bar{N}_b$ for interleaver depths $L = 101,\ 1001$, and 10001. Again, puncturing has less overall performance loss in concatentated codes than in the individual codes because no one distance

[8] Assuming as is usual in convolutional codes that extra dimensions can be added without ISI as in method 1 of Section 8.1.

dominates performance. Table 8.8 repeats some measured results for Table 8.7's code, also from [17]. Table 8.8 lists measured error coefficients for periods of 1,001 and 10,001. The simulation results are very close to Table 8.7's analysis, except when the error-event length becomes large relative to the period then some accurracy is lost. This is because Table 8.7 essentially ignored the finite period in enumerating error-event pairs.

distance	$\bar{N}_b$ for $L = 10^2$	$\bar{N}_b$ for $L = 10^3$	$\bar{N}_b$ for $L = 10^4$
8	3.89×10^{-2}	3.99×10^{-3}	3.99×10^{-4}
9	7.66×10^{-2}	7.96×10^{-3}	7.99×10^{-4}
10	.1136	1.1918×10^{-2}	1.1991×10^{-3}
11	.1508	1.5861×10^{-2}	1.5985×10^{-3}
12	.1986	1.9887×10^{-2}	1.9987×10^{-3}
13	.2756	2.4188×10^{-2}	2.4017×10^{-3}
14	.4079	2.9048×10^{-2}	2.8102×10^{-3}
15	.6292	3.4846×10^{-2}	3.2281×10^{-3}
16	1.197	6.5768×10^{-2}	6.0575×10^{-3}

Table 8.8: 4-state convolutional code as a Turbo Code's base code

Analysis now progresses to the same code with reversed output-bit order so that

$$G_2(D) = \begin{bmatrix} 1 & \dfrac{1+D^2}{1+D+D^2} \end{bmatrix} \quad . \tag{8.63}$$

This encoder maps a weight 5 codeword error event of $\begin{bmatrix} 1+D+D^2 & 1+D^2 \end{bmatrix}$ to 3-input-bit errors $(1+D+D^2)$. With concatenation, then this error event corresponds to $d_{free} = 7 = 3+2+2$ indicating that the concatenated code does not have the same codewords and indeed has a lower minimum distance. Nonetheless, this minimum distance requires 3 input-bit erred positions of errors to coincide for the two codes, which has probability (with uniform random interleaving)

$$L \cdot \binom{L}{3}^{-1} = \frac{6}{(L-1)(L-2)} \quad . \tag{8.64}$$

Thus, 3-input-bit error events have very small probability. From Table 8.9 [17], the two input error event with smallest codeword weight is $1+D^3$ and corresponds to an output distance of $d = 10$. Repeating Table 8.7 now in Table 8.9 shows (there are no odd-number free distances for 2-input-bit-error events).

distance	in-ev code 1	in-ev code 2	$2\bar{N}_b(d)$	d_{free} construct	$\frac{4 \cdot a(d,2)}{100}$
10	$1+D^3$	$D^m \cdot (1+D^3)$	2	2+4+4	.04
12	$1+D^3$	$D^m \cdot (1+D^6)$	4	2+4+6	.08
	$1+D^6$	$D^m \cdot (1+D^3)$		2+6+4	
14	$1+D^3$	$D^m \cdot (1+D^9)$	6	2+4+8	.12
	$1+D^6$	$D^m \cdot (1+D^6)$		2+6+6	
	$1+D^9$	$D^m \cdot (1+D^3)$		2+8+4	
16	$1+D^3$	$D^m \cdot (1+D^{12})$	8	2+4+10	.16
	$1+D^6$	$D^m \cdot (1+D^9)$		2+6+8	
	$1+D^9$	$D^m \cdot (1+D^6)$		2+8+6	
	$1+D^{12}$	$D^m \cdot (1+D^3)$		2+10+4	
t	$1+D^3$	$D^m \cdot (1+D^{1.5(t-8)})$	t-8		.02(t-8)
	$\vdots$	$\vdots$		$\vdots$	
	$1+D^{1.5(t-8)}$	$D^m \cdot (1+D^3)$			

Table 8.9: $G_2(D) = (1+D^2)/(1+D+D^2)$'s 2-input-bit error corresponding codeword-error events.

The second concatenated turbo-code implentation with $G_2(D)$ is slightly better because the distances are larger, and the error coefficients are the same. The improvement is roughly $10 \cdot \log_{10}(10/8) = 1$ dB. However, at high SNRs, the dominant free-distance-7 error event will eventually make asymptotic performance worse. So, up to some SNR, the second code is better and then it is worse after that SNR. The eventual high-SNR minimum-distance dominance causes different error events' contribution to dominate. This leads to a flattening of the $\bar{P}_b$ curve known as turbo codes' "error floor." In both concatenated codes, the $r = 1/3$ turbo code's larger free distance is only a slight improvement. The main improvement is the $\bar{P}_b$ reduction caused by division by roughly the interleaver period $L-1$. Such an improvement is the **interleaver gain.** Over the range of normal code use, a factor of 10 usually corresponds to 1 dB improvement, so this code's interleaver gain is then

$$\gamma_{parallel} = \log_{10}((L-1)/2) \tag{8.65}$$

at least up to a few dB (at which point the approximation of 1 dB per factor of 10 is no longer valid) and d_{free} again dominates.

In a more general situation of a rate $1/n$ code

$$G_{sys}(D) = \begin{pmatrix} 1 & \frac{g_1(D)}{g_0(D)} & \dots & \frac{g_{n-1}(D)}{g_0(D)} \end{pmatrix} \quad , \tag{8.66}$$

any weight-one input information sequence $u(D) = D^m$ produces an infinite-weight output codeword(or large finite weight if the code is terminated in a finite packet length). Thus, again only weight 2 (or higher) input errors are of interest since any $g_0(D)$ must divide $1+D^j$ for some sufficiently large j. Since input-bit weight 3 errors have (8.64)'s coefficient factor of $\frac{6}{(L-1)(L-2)}$, then they typically have much lower contribution to $\bar{P}_b$ unless the SNR is sufficiently high and the inter-codeword distances for these errors is smaller.

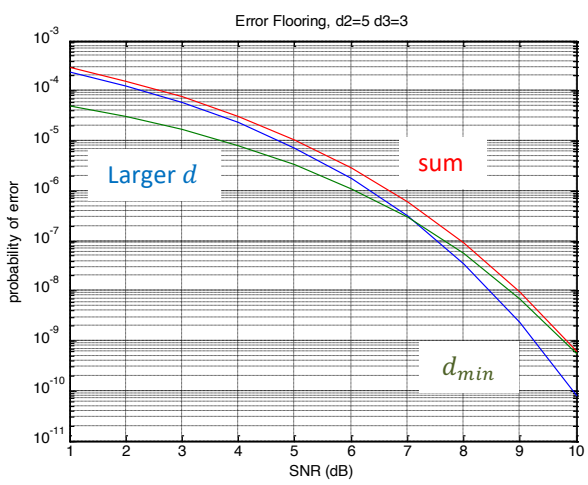

Figure 8.20: Simplified two-term basic error flooring in turbo codes.

Error Flooring: Error flooring occurs when the SNR is sufficiently high that the smaller nearest-neighbor coefficient for error events with 2 errors (or more) is overwhelmed by smaller d_i terms' exponential decrease with SNR of the Q-function at high SNR: At such high SNR, d_{min} error events dominate. Figure 8.20 depicts this effect generically for two types of error events for which one has a $d_2 = 5$ (output distance corresponding to 2-input bit errors), but has a larger coefficient of .01, while the second has a smaller $d_3 = 3$ (output distance corresponding to 3 input bit errors), but a coefficient of .001. At smaller SNR, the corresponding $d_{free} = 5$ Q-function's smaller coefficient and smaller distance contribution to error probability is negligible, but eventually as SNR increases, the exponential decay of the Q-function terms with larger distance causes cause them instead to be negligible. In the middle of Figure 8.20 where the overall error probability (the upper curve, which is sum of the two curves) deviates temporarily from

a pure exponential decay and "curves less or flattens" temporarily until resuming the exponential decay again for the lower distance portion. The smaller error-coefficient term with higher distance dominates at low SNR and provides higher total error probability. Fiugure 8.21 generically exaggerates this effect, but it will be evident to various degrees in different turbo-coding error-probability plots later.

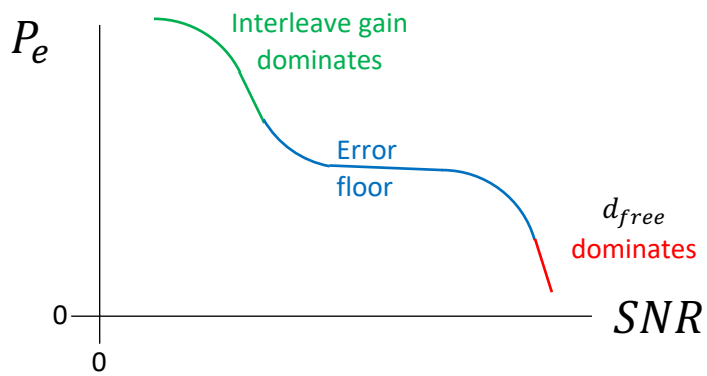

Figure 8.21: Generic depiction of Turbo Codes' multi-term error flooring..

Turbo-code design chooses parameters so that this floor occurs below the designs target error probability. Over the corresponding $\bar{P}_b$ range before the floor, if 2-input-bit error events dominate, then

$$\bar{P}_{b,turbo} \approx \frac{2\bar{N}_b \cdot (d_2)}{L} \cdot Q\left(\frac{d_{min}}{2\sigma}\right) \quad , \tag{8.67}$$

$\bar{P}_b$ thus reduces by the factor $L/2$. More generally, rate $r = k/n$ codes have more complex analysis because a 2-input-bit error event that"cancells all the denominators" is no longer the only error-event of interest. Turbo-code designs, to simplify analysis, typically puncture $r = 1/n$ codes.

Figure 8.22 illustrates Chapter 7's iterative-decoding error probability for $L = 100$, $L = 1000$, and $L = 10000$ for Table 8.10's 4-state turbo code (with encoder G_1) with puncturing that has $r = 1/2$. The convolutional "mother" code itself has gain of 4.7 dB, and the turbo interleaving adds another $\gamma_{parallel} = 3.7$ dB for $L = 10000$, so then roughly 8.4 dB total gain. Figure 8.22's coding gain at $P_e = 10^{-6}$ is about 8 dB, slightly less because of the higher-distance terms' contributions are still not negligible.

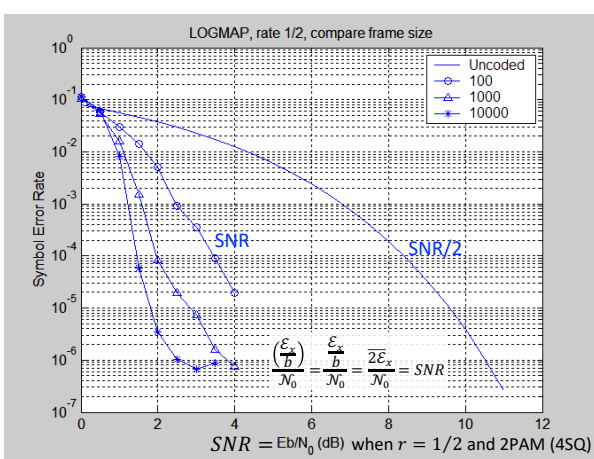

Figure 8.22: Error probability for the $r = 1/2$ 4-state turbo code, with puncturing to $r = 1/2$ ($C = 1/2$ when SNR=0 dB). When $r = 1$ (uncoded), the horizontal access is $SNR/2$.

Figure 8.23 illustrates iterative decoding's convergence for Figure 8.22's (same) code for $L = 1000$.

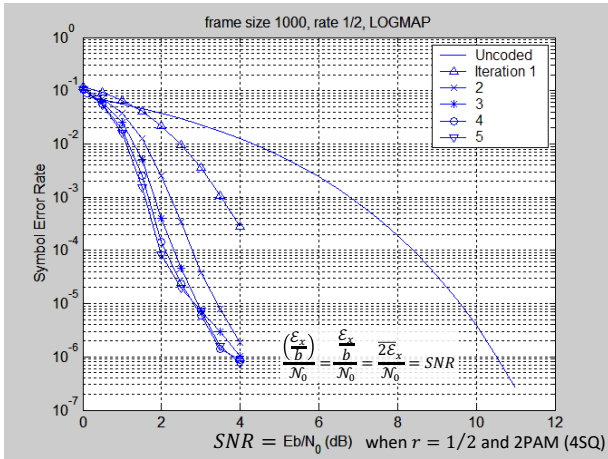

Figure 8.23: APP P_e convergence for the $r = 1/2$ 4-state turbo code, with puncturing to $r = 1/2$.

P_e (and thus $\bar{P}_b$) converge within just a few iterations. Figure 8.24 compares the error probabilities for Chapter 7's SOVA and APP.

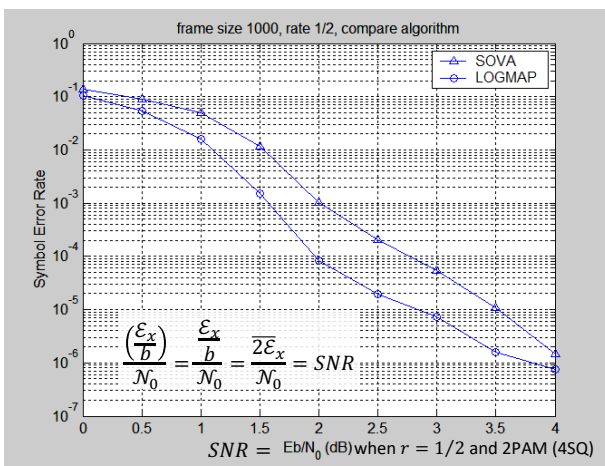

Figure 8.24: SOVA vs APP P_e comparison for the $r = 1/2$ (punctured) turbo code.

While the difference is larger at very low SNR, the difference becomes small in the range of operation of $P_e = 10^{-6}$. Figure 8.25 illustrates the additional gain of not puncturing and using rate 1/3 instead, which is about .7 dB at $\bar{P}_b = 10^{-6}$ for this example. Actually, slightly larger because this rate 1/3 code also sees a larger d distribution generally than the punctured rate 1/2 code. This code's gain is small, as is typical with turbo-code puncturing; little coding gain is lost. This is because the puncturing does not significantly affect the interleave gain.

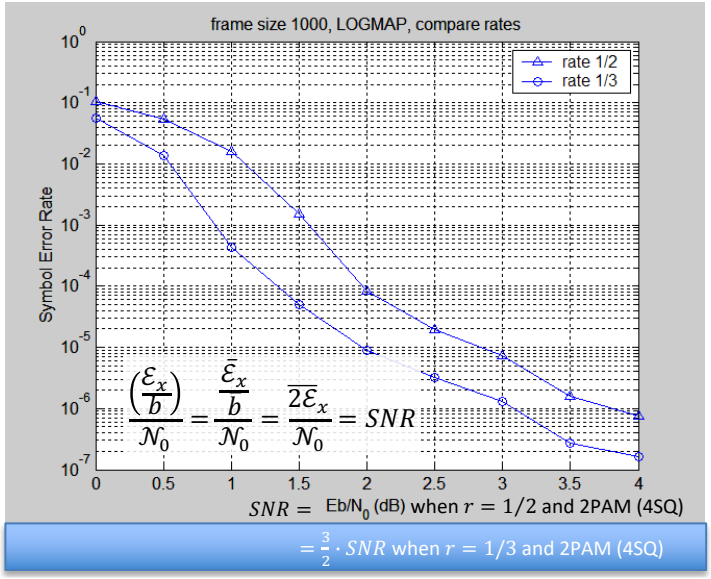

Figure 8.25: Comparison of rate 1/3 and rate 1/2 with puncturing for 4-state Turbo code.

8.3.2.4 Bit-error probability analysis for serial concatenation

Serial code concatenation has different analysis than parallel concatenation. This analysis retains the basic concept that an error event with several bit errors that are less likely to occur, with random interleaving, into the exact positions that cause errors in two codes simultaneously. However, re-encoding inner-code output-parity bits into new additional outer-code parity bits changes the error-probability calculation. Consequently, serial turbo-code concatenation often uses two different codes and the error-probability must also accommodates this difference.

First, the serially concatenated codes' error coefficients simply reflect that the probability that $\left\lceil \frac{d_{free}^{out}}{2} \right\rceil$ bits after depth-L uniform random interleaving again fall in "all the wrong places in the outer code (out)" is

$$\left(\begin{array}{c} L \\ \left\lceil \frac{d_{free}^{out}}{2} \right\rceil \end{array} \right) \quad . \tag{8.68}$$

Serial concantenated-code analysis approximates this expression by

$$c \cdot L^{-\left(\left\lceil \frac{d_{free}^{out}}{2} \right\rceil\right)} \quad , \tag{8.69}$$

with constant c independent of d_{free} and L. (8.69)'s exponential dependence on the codeword (or encoder output) free distance, rather than on the number of input bit errors, distinguishes serial concatenation from parallel concatenation in Equations (8.60) and (8.64). (8.69)'s coefficient multiplies the error coefficent for the outer Turbo Code for both P_e and $\bar{P}_b$. expressions. This following analysis assumes that the inner encoder is systematic so that at least two input-bit errors are necessary to cause finite output distance and thus to have non-zero error-event probability. This inner distance for 2-bit errors is called d_2^{in} and similarly a 3-input-bit error distance is called d_3^{in}. If the inner code's decoder experiences the 2-bit error event, then that inner decoder's error probability has Q-function argument $\sqrt{d_2^{in} \cdot \text{SNR}}$. These 2 input bits (which are also part of the inner-code's output distance in a systematic realization) necessarily must contribute to the outer-code's output error event, so the outer decoder's tolerance for errors reduces to[9]

$$\left\lceil \frac{d_{free}^{out} - 3}{2} \right\rceil \quad . \tag{8.70}$$

[9]The use of the greatest integer function in (8.70) allows the argument to be reduced by 3 when outer free distance is odd and by only 2 when outer free distance is 2 in agreement with the reality that codes with odd distances essentially get one more position of possible error before experiencing an error event.

Input-bit errors corresponding to those remaining bits' outer-code correction ability otherwise increase the combined codes' distance to $\lceil \frac{d_{free}^{out}-3}{2} \rceil \cdot d_2^{in}$. Thus, iteratively decoded serial concatenation's overall effect on the Q-function argument is

$$\left\{ \left\lceil \frac{d_{free}^{out}-3}{2} \right\rceil \cdot d_2^{in} + d_w^{in} \right\} \cdot \text{SNR} \quad , \tag{8.71}$$

where $d_w^{in} = d_2^{in}$ for situations in which the outer code has even d_{free}^{out} and $d_w^{in} = d_3^{in}$ for odd d_{free}^{out}. The SNR in (8.71) lowers as the product of the two code rates, which is $r = r_{in} \cdot r_{out}$. Thus, the outer code's additional error-correcting power (multiplying a single-bit difference by a term like $1 + D^m$ when $d_2^{in} = 2$ and correspondingly more generally) applies to all the extra outer-code bits that must additionally be in error and thus multiplies d_2^{in} but does nog multiply the common bits. However those common bits must be erred and whence (8.71)'s last additive term. The total-bit-error-counting quantity[10]

$$N_b(d) = \sum_{i=1}^{\infty} i \cdot N(i,d) \tag{8.72}$$

has special use in serial concatenation: with it, the overall $\bar{P}_b$ expression is then

$$\bar{P}_b \approx L \left(\begin{array}{c} L \\ \left\lceil \frac{d_{free}^{out}}{2} \right\rceil \end{array} \right)^{-1} \cdot \frac{\bar{N}_b(d_{free}^{in}) \cdot \bar{N}_b(d_{free}^{out})}{b} \cdot Q \left(\sqrt{\left\{ \left\lceil \frac{d_{free}^{out}-3}{2} \right\rceil \cdot d_2^{in} + d_w^{in} \right\} \cdot \text{SNR}} \right) \quad , \tag{8.73}$$

This analysis does not require the outer encoder to be systematic, nor even use feedback. The interleaving gain thus is

$$\gamma_{serial} = \log_{10} \left(\frac{L!}{\left\lceil \frac{d_{free}^{out}}{2} \right\rceil !} \right) \text{ dB} \tag{8.74}$$

over a range of operation of 10^{-4} to 10^{-7}. This factor can be large for $d_{free}^{out} > 2$, but (8.73)'s product of $\bar{N}_b(d)$ terms reduces γ_{serial}'s effect. Serial concatenation's error flooring follows the same basic principle as parallel concatenation, namely that eventually as SNR increases, minimum distance dominates P_e and $\bar{P}_b$, and thus dominates an event with larger free distance than d_2^{in} or d_w^{in}, but larger error coefficient has negligible effect on P_e and $\bar{P}_b$ with sufficiently large SNR.

8.3.2.5 Parallel Concatenation Coding Tables for $r = 1/n$ or $r = (n-1)/n$ with no puncturing

This subsection's tables originate with some exhaustive search reports by Divsalar[11] [18]. These Turbo Codes appear to be the best or among the best known for the various complexities and parameters listed. The rate $1/n$ codes are for AWGN channels with unrestricted bandwidth. The rate $(n-1)/n$ codes, apart from shaping gain, also find best use under Gray Mapping to multilevel constellations with bandwidth-limited AWGN channels. These codes sufficiently approach capacity so that Section 8.6's final outer-hard concatenation effectively achieve capacity. Yet better codes ultimately found elsewhere probably have only small incremental improvement. These codes' optimization did not consider puncturing. Subsection 8.3.2.6 addresses puncturing for channels with higher SNR's such when $1/2 < r < 1$, and possible use in multilevel subsymbol transmission.

This section's earlier $r = 1/2$ 4-state convolutional code has a systematic realization

$$[1 \;\; (1+D^2)/(1+D+D^2)] \quad ,$$

[10] The quantity $N(b,d)$ is the number of code error events with distance d and b erred input bits.

[11] Dariusch Divsalar, (1947-) an American communication theorist and senior scientist at Jet Propulsion Laboratories.

and is Table 8.10's first code with the presumption of division of all terms by $g_0(D)$. Table 8.10's left-most values of d_2, d_3, and d_{free} are for the mother convolutional code[12]. The here-added right-most values are for the (parallel) turbo code and thus can be used directly in P_e expressions like (8.67).

2^ν	$g_0(D)$	$g_1(D)$	d_2	d_3	d_{free}	$d_{2,cat}$	$d_{3,cat}$	$d_{free,cat}$
4	7	5	5	6	5	8	10	8
8	17	13	6	7	6	10	11	10
16	23	35	7	7	7	12	11	11

Table 8.10: Rate 1/2 constituent (mother) parallel convolutional codes
(This table's codes are posed by this text's author as Divsalar did not provide $r = 1/2$ codes presumably because they're relatively easy to find.)

The best $r = 1/3$ Divsalar codes are in Table 8.11.

2^ν	$g_0(D)$	$g_1(D)$	$g_2(D)$	d_2	d_3	d_{free}	$d_{2,cat}$	$d_{3,cat}$	$d_{free,cat}$
2	3	2	1	4	∞	4	6	∞	6
4	7	5	3	8	7	7	14	11	11
8	13	17	15	14	10	10	26	17	17
16	23	33	37	22	12	10	42	21	21

Table 8.11: Rate 1/3 constituent (mother) parallel convolutional codes.

The best rate 1/4 codes found by Divsilar (again with overall turbo code values found by this author) appear in Table 8.12:

2^ν	$g_0(D)$	$g_1(D)$	$g_2(D)$	$g_3(D)$	d_2	d_3	d_{free}	$d_{2,cat}$	$d_{3,cat}$	$d_{free,cat}$
8	13	17	15	11	20	12	12	38	21	21
16	23	35	27	37	32	16	14	62	31	31

Table 8.12: Rate 1/4 constituent parallel convolutional codes.

The best rate 2/3 codes found by Divsilar (again with overall turbo code values found by this author) appear in Table 8.13:

2^ν	$h_0(D)$	$h_1(D)$	$h_2(D)$	d_2	d_3	d_{free}	$d_{2,cat}$	$d_{3,cat}$	$d_{free,cat}$
4	7	3	5	4	3	3	6	3	3
8	13	15	17	5	4	4	8	5	5
16	23	35	27	8	5	5	14	7	7
16	45	43	61	12	6	6	22	9	9

Table 8.13: Rate 2/3 constituent (mother) parallel convolutional codes.

The best rate 3/4 codes found by Divsilar (again with overall turbo code values found by this author) appear in Table 8.14:

2^ν	$h_0(D)$	$h_1(D)$	$h_2(D)$	$h_3(D)$	d_2	d_3	d_{free} d_{free}	$d_{2,cat}$	$d_{3,cat}$	$d_{free,cat}$
4	7	5	3	1	3	3	3	4	3	3
8	13	15	17	11	4	4	4	6	5	5
16	23	35	33	25	5	4	4	8	5	5

Table 8.14: Rate 3/4 constituent (mother) parallel convolutional codes.

[12]These values can follow with some effort from Matlab's distspec.m program that enumerates a list of distances and their occurrence, along with total number of input information-bit errors corresponding for all the associated number of error events. The use requires determining apportionment of the error events to different numbers of info-bit errors for situations where $N_e > 1$, which can lead to a few choices. Fortunately, the references here appear to have resolved between those choices with a more exhaustive search.

8.3.2.6 Parallel and Serial Turbo Code Tables with puncturing for base rate 1/2

When BICM is used with $|C| > 2$, code rates $1/2 < r < 1$ become of interest. Deaneshgaran, Laddomada and Mondin [17] (DLM) have searched over all rate 1/2 codes and puncturing patterns for different rates to find best parallel and serial concatenations for up to 32 states. This subsection follows their work, which is based on a common underlying $r = 1/2$ convolutional mother code's puncturing. The common structure for all codes (at some fixed specified number of states) then can simplify implementation through puncturing of rate-1/2 codes by mother-code-puncturing design, rather than the more complex interleaved-streams' puncturing design. Table 8.15 enumerates rate-1/2 mother systematic constituent encoder parameters while Table 8.16 lists these codes puncturing patterns for increasingly high rate codes. The highest achievable rate is $r_{turbo} = 4/5$ for the constituent punctured rate $r_{mother} = 8/9$.

2^ν	$\left[1 \ \frac{g_1}{g_0}\right]$	d	$N_{e,d}$	$N_{b,d}$	d_2	d_3
4	$[\,1 \ \frac{5}{7}\,]$	5	1	3	6	5
(d_2)		6	2	6		
		7	4	14		
		8	8	32		
		9	16	72		
4	$[\,1 \ \frac{7}{5}\,]$	5	1	2	5	∞
(SNR)		6	2	6		
		7	4	14		
		8	8	32		
		9	16	72		
8	$[\,1 \ \frac{15}{13}\,]$	6	2	6	8	6
(d_2)		8	10	40		
		10	49	245		
		12	241	1446		
		14	1185	8295		
8	$[\,1 \ \frac{17}{13}\,]$	6	1	4	8	7
(d_2)		7	3	9		
		8	5	20		
		9	11	51		
		10	25	124		
8	$[\,1 \ \frac{15}{17}\,]$	6	1	2	6	∞
(SNR)		7	3	12		
		8	5	20		
		9	11	48		
		10	25	126		
16	$[\,1 \ \frac{33}{31}\,]$	7	2	8	12	7
(d_2)		8	4	16		
		9	6	26		
		10	15	76		
		11	37	201		
16	$[\,1 \ \frac{21}{37}\,]$	6	1	2	6	∞
		7	1	5		
		8	3	10		
		9	5	25		
		10	12	56		

2^ν	$\left[1 \ \frac{g_1}{g_0}\right]$	d	$N_{e,d}$	$N_{b,d}$	d_2	d_3
16	$[\,1 \ \frac{27}{31}\,]$	7	2	8	12	7
(d_2)		8	3	12		
		9	4	16		
		10	16	84		
		11	37	213		
16	$[\,1 \ \frac{37}{23}\,]$	6	1	4	12	8
(d_2)		8	6	23		
		10	34	171		
		12	174	1055		
		14	930	6570		
16	$[\,1 \ \frac{33}{23}\,]$	7	2	8	12	7
(d_2)		8	4	16		
		9	6	26		
		10	15	76		
		11	37	201		
16	$[\,1 \ \frac{35}{23}\,]$	7	2	8	12	7
(d_2)		8	3	12		
		9	4	16		
		10	16	84		
		11	37	213		
16	$[\,1 \ \frac{23}{35}\,]$	7	2	6	7	∞
(SNR)		8	3	12		
		9	4	20		
		10	16	76		
		11	137	194		
32	$[\,1 \ \frac{71}{53}\,]$	8	3	12	12	∞
(d_3)		10	16	84		
		12	68	406		
		14	860	6516		
		16	3812	30620		
32	$[\,1 \ \frac{67}{51}\,]$	8	2	7	20	8
(SNR d_2)		10	20	110		
		12	68	406		
		14	469	3364		
		16	2560	20864		

Table 8.15: Best 4/8/16/32-state $r = 1/2$ constituent (mother) convolutional codes with puncturing. $N_{e,d} = N_{d-d_{free}}$ and $N_{b,d} = \sum_b N(b,d)$ in this table.

In Table 8.15, an indication of SNR means among all encoders with the same d_2, the one with minimum SNR to get $\bar{P}_b = 10^{-6}$ was selected. The entry d_3 means instead that the code with maximum d_3 was selected. The entry d_2 is a code with largest d_2.

Parallel concatenations with the same code: Table 8.16 lists the best parallel-concatenation codes (presuming the same code is used twice). The codes are rate $(n-1)/n$ after puncturing, but before turbo-code use. The puncturing pattern is octal and corresponds to the mother code rate-1/2 output

bit pairs (info, parity) are enumerated from left to right in increasing time and the pucturing pattern is pressed on top with 0 meaning puncture that parity bit. For instance 5352 means 101 011 101 010 so and corresponds to (letting i_k be an information bit and p_k be the corresponding parity bit)

$$(i_1, p_1, i_2, p_2, i_3, p_3, i_4, p_4, i_5, p_5, i_6, p_6) \rightarrow (i_1, i_2, i_3, p_3, i_4, i_5, i_6) \quad . \tag{8.75}$$

These codes all presume systematic implementation without repeat of input bits by the turbo code. As earlier, the overall $d_{free,cat}$ for performance analysis of the concatenated system is found as $d_{free,cat} = \min_{i=2,3} 2 \cdot d_i - i$.

$n-1$	4 states	8 states	16 states	32 states
2	$[\,1\ \frac{5}{7}\,]$ 13 (3,1,3) $d_2 = 4$, $d_3 = 3$	$[\,1\ \frac{15}{13}\,]$ 13 (4,3,10) $d_2 = 5$, $d_3 = 4$	$[\,1\ \frac{37}{23}\,]$ 13 (4,2,6) $d_2 = 7$, $d_3 = 4$	$[\,1\ \frac{67}{51}\,]$ 13 (5,2,7) $d_2 = 9$, $d_3 = 5$
3	$[\,1\ \frac{5}{7}\,]$ 56 (3,4,10) $d_2 = 3$, $d_3 = 3$	$[\,1\ \frac{15}{13}\,]$ 53 (3,2,5) $d_2 = 3$, $d_3 = 3$	$[\,1\ \frac{37}{23}\,]$ 53 (3,1,3) $d_2 = 4$, $d_3 = 3$	$[\,1\ \frac{67}{51}\,]$ 53 (4,2,7) $d_2 = 7$, $d_3 = 4$
4	$[\,1\ \frac{5}{7}\,]$ 253 (2,1,2) $d_2 = 2$, $d_3 = 3$	$[\,1\ \frac{15}{13}\,]$ 253 (3,9,24) $d_2 = 3$, $d_3 = 3$	$[\,1\ \frac{37}{23}\,]$ 253 (3,3,9) $d_2 = 4$, $d_3 = 3$	$[\,1\ \frac{67}{51}\,]$ 253 (3,1,3) $d_2 = 5$, $d_3 = 3$
5	$[\,1\ \frac{5}{7}\,]$ 1253 (2,2,4) $d_2 = 2$, $d_3 = 3$	$[\,1\ \frac{17}{13}\,]$ 1253 (3,15,40) $d_2 = 3$, $d_3 = 3$	$[\,1\ \frac{27}{31}\,]$ 1272 (3,2,6) $d_2 = 4$, $d_3 = 3$	$[\,1\ \frac{71}{53}\,]$ 1272 (4,108,406) $d_2 = 4$, $d_3 = \infty$
6	$[\,1\ \frac{5}{7}\,]$ 5352 (2,22,44) $d_2 = 2$, $d_3 = 3$	$[\,1\ \frac{17}{13}\,]$ 5253 (2,1,2) $d_2 = 2$, $d_3 = 3$	$[\,1\ \frac{27}{31}\,]$ 5253 (3,12,33) $d_2 = 3$, $d_3 = 3$	$[\,1\ \frac{71}{53}\,]$ 5253 (3,3,6) $d_2 = 3$, $d_3 = \infty$
7	$[\,1\ \frac{5}{7}\,]$ 25253 (2,7,14) $d_2 = 2$, $d_3 = 3$	$[\,1\ \frac{15}{17}\,]$ 25253 (2,7,14) $d_2 = 2$, $d_3 = \infty$	$[\,1\ \frac{33}{23}\,]$ 25253 (2,1,2) $d_2 = 2$, $d_3 = 3$	$[\,1\ \frac{67}{51}\,]$ 25253 (2,1,2) $d_2 = 3$, $d_3 = 3$
8	$[\,1\ \frac{5}{7}\,]$ 125253 (2,9,18) $d_2 = 2$, $d_3 = 3$	$[\,1\ \frac{15}{13}\,]$ 125253 (2,4,8) $d_2 = 2$, $d_3 = 3$	$[\,1\ \frac{37}{23}\,]$ 125253 (2,1,2) $d_2 = 2$, $d_3 = 3$	$[\,1\ \frac{67}{51}\,]$ 125253 (3,17,49) $d_2 = 3$, $d_3 = 3$

Table 8.16: Best puncturing patterns for given high-rate parallel turbo codes. The triplets listed are $(d_i, N_{d-d_{free}}, \sum_b N(b,d))$.

Serial concatenations - inner codes: Table 8.17 lists best known inner codes with puncturing patterns generated from rate 1/2 mother codes. Often the puncturing can delete information bits (meaning the parity bit carries better information under puncturing than the information bit itself). Puncturing applies to the inner code output and the rate is for the resultant punctured inner code and is $(n-1)/n$.

Serial concatenations - outer codes: Table 8.18 lists best known outer codes with puncturing patterns generated from rate 1/2 mother codes. Often the puncturing can delete information bits (meaning the parity bit carries better information under puncturing than the information bit itself). Puncturing is applied to the outer code output and the rate is for the resultant punctured outer code and is $(n-1)/n$.

A serial turbo-code design would choose one code from Table 8.17 and one from 8.18, using (8.73) to evaluate the performance and the overall code rate is $r = r_{in} \cdot r_{out}$.

$n-1$	4 states	8 states	16 states	32 states
2	$[\ 1\ \frac{5}{7}\ ]$ 7 (3,1,3) $d_2 = 5$, $d_3 = 3$	$[\ 1\ \frac{15}{13}\ ]$ 7 (4,3,10) $d_2 = 7$, $d_3 = 4$	$[\ 1\ \frac{27}{31}\ ]$ 7 (4,1,5) $d_2 = 11$, $d_3 = 5$	$[\ 1\ \frac{67}{51}\ ]$ 7 (5,2,7) $d_2 = 19$, $d_3 = 6$
3	$[\ 1\ \frac{5}{7}\ ]$ 27 (2,1,4) $d_2 = 4$, $d_3 = 3$	$[\ 1\ \frac{15}{13}\ ]$ 27 (3,2,9) $d_2 = 6$, $d_3 = 3$	$[\ 1\ \frac{27}{31}\ ]$ 27 (3,1,5) $d_2 = 10$, $d_3 = 4$	$[\ 1\ \frac{67}{51}\ ]$ 65 (4,7,51) $d_2 = 18$, $d_3 = 5$
4	$[\ 1\ \frac{5}{7}\ ]$ 67 (2,4,13) $d_2 = 4$, $d_3 = 3$	$[\ 1\ \frac{15}{13}\ ]$ 127 (3,9,52) $d_2 = 6$, $d_3 = 3$	$[\ 1\ \frac{27}{31}\ ]$ 351 (4,16,176) $d_2 = 10$, $d_3 = 4$	$[\ 1\ \frac{67}{51}\ ]$ 325 (4,22,191) $d_2 = 18$, $d_3 = 5$
5	$[\ 1\ \frac{5}{7}\ ]$ 527 (2,6,26) $d_2 = 4$, $d_3 = 2$	$[\ 1\ \frac{15}{13}\ ]$ 527 (2,1,6) $d_2 = 6$, $d_3 = 3$	$[\ 1\ \frac{27}{31}\ ]$ 635 (3,8,43) $d_2 = 10$, $d_3 = 4$	$[\ 1\ \frac{67}{51}\ ]$ 1525 (3,3,21) $d_2 = 18$, $d_3 = 5$
6	$[\ 1\ \frac{5}{7}\ ]$ 3525 (2,84,2693) $d_2 = 4$, $d_3 = 2$	$[\ 1\ \frac{17}{13}\ ]$ 2527 (2,4,20) $d_2 = 6$, $d_3 = 4$	$[\ 1\ \frac{37}{23}\ ]$ 6525 (2,4,22) $d_2 = 10$, $d_3 = 5$	$[\ 1\ \frac{67}{51}\ ]$ 6525 (3,4,34) $d_2 = 18$, $d_3 = 5$
7	$[\ 1\ \frac{5}{7}\ ]$ 12527 (2,15,74) $d_2 = 4$, $d_3 = 2$	$[\ 1\ \frac{17}{13}\ ]$ 12527 (2,7,42) $d_2 = 6$, $d_3 = 4$	$[\ 1\ \frac{33}{23}\ ]$ 32525 (2,6,51) $d_2 = 10$, $d_3 = 4$	$[\ 1\ \frac{67}{51}\ ]$ 32525 (3,14,135) $d_2 = 18$, $d_3 = 5$
8	$[\ 1\ \frac{5}{7}\ ]$ 72525 (2,153,5216) $d_2 = 4$, $d_3 = 2$	$[\ 1\ \frac{15}{13}\ ]$ 52527 (2,4,32) $d_2 = 6$, $d_3 = 3$	$[\ 1\ \frac{33}{23}\ ]$ 72525 (2,6,42) $d_2 = 10$, $d_3 = 4$	$[\ 1\ \frac{67}{51}\ ]$ 152525 (3,23,299) $d_2 = 18$, $d_3 = 5$

Table 8.17: Best puncturing patterns for given high-rate serial (inner code) turbo codes.

$n-1$	4 states	8 states	16 states	32 states
2	$[\,1\ \frac{7}{5}\,]$ 15 (3,1,2) $d_2 = 3$, $d_3 = \infty$	$[\,1\ \frac{15}{13}\,]$ 13 (4,3,10) $d_2 = 5$, $d_3 = 4$	$[\,1\ \frac{33}{31}\,]$ 13 (5,7,25) $d_2 = 6$, $d_3 = 5$	$[\,1\ \frac{71}{53}\,]$ 13 (6,15,60) $d_2 = 6$, $d_3 = \infty$
3	$[\,1\ \frac{5}{7}\,]$ 56 (3,4,10) $d_2 = 2$, $d_3 = 3$	$[\,1\ \frac{15}{17}\,]$ 33 (4,29,126) $d_2 = 4$, $d_3 = \infty$	$[\,1\ \frac{23}{35}\,]$ 17 (4,29,150) $d_2 = 4$, $d_3 = \infty$	$[\,1\ \frac{71}{53}\,]$ 36 (4,1,4) $d_2 = 8$, $d_3 = \infty$
4	$[\,1\ \frac{5}{7}\,]$ 253 (2,1,2) $d_2 = 2$, $d_3 = 3$	$[\,1\ \frac{15}{17}\,]$ 136 (3,5,16) $d_2 = 3$, $d_3 = \infty$	$[\,1\ \frac{27}{31}\,]$ 351 (4,16,176) $d_2 = 10$, $d_3 = 4$	$[\,1\ \frac{71}{53}\,]$ 133 (4,28,192) $d_2 = 8$, $d_3 = \infty$
5	$[\,1\ \frac{5}{7}\,]$ 1253 (2,2,4) $d_2 = 2$, $d_3 = 3$	$[\,1\ \frac{17}{13}\,]$ 1253 (3,15,40) $d_2 = 3$, $d_3 = 3$	$[\,1\ \frac{33}{31}\,]$ 653 (4,98,436) $d_2 = 4$, $d_3 = 4$	$[\,1\ \frac{71}{53}\,]$ 1272 (4,108,406) $d_2 = 4$, $d_3 = \infty$
6	$[\,1\ \frac{7}{5}\,]$ 3247 (2,4,8) $d_2 = 2$, $d_3 = \infty$	$[\,1\ \frac{17}{13}\,]$ 5253 (2,1,2) $d_2 = 2$, $d_3 = 3$	$[\,1\ \frac{27}{31}\,]$ 3352 (3,7,24) $d_2 = 4$, $d_3 = 3$	$[\,1\ \frac{71}{53}\,]$ 5253 (3,3,6) $d_2 = 3$, $d_3 = \infty$
7	$[\,1\ \frac{7}{5}\,]$ 15247 (2,6,12) $d_2 = 2$, $d_3 = \infty$	$[\,1\ \frac{15}{17}\,]$ 15652 (2,2,4) $d_2 = 2$, $d_3 = \infty$	$[\,1\ \frac{23}{35}\,]$ 13632 (3,13,52) $d_2 = 3$, $d_3 = \infty$	$[\,1\ \frac{71}{53}\,]$ 13172 (3,4,18) $d_2 = 5$, $d_3 = \infty$
8	$[\,1\ \frac{7}{5}\,]$ 65247 (2,9,18) $d_2 = 2$, $d_3 = \infty$	$[\,1\ \frac{15}{17}\,]$ 65256 (2,3,6) $d_2 = 2$, $d_3 = \infty$	$[\,1\ \frac{33}{31}\,]$ 123255 (3,21,75) $d_2 = 4$, $d_3 = 3$	$[\,1\ \frac{67}{51}\,]$ 124672 (3,11,36) $d_2 = 5$, $d_3 = 3$

Table 8.18: Best puncturing patterns for given high-rate serial (outer code) turbo codes.

INSERT THE PUNCTURED DESIGN EXAMPLE HERE, using basic minimal encoder to circumvent the poly2trellis state-exponentiation bug.

8.3.3 Low-Density Parity Check Codes

Low-Density Parity Check (LDPC) codes were first studied by Robert Gallager of MIT [21] in the early 1960's, as a theoretical exercise. Later, with computing advances, they have become practical to implement. LDPC are linear block codes that essentially implement Chapter 2's random-code construction for designs at rates that approach capacity. LDPC block codes typically have long block length $100 < N < 10000$ bits and a parity matrix $H(D) = H(0) = H$ that has a number of 1's that grows linearly N, rather than as the square of the block length as might be expected if 0's and 1's were selected with equal probability for the H matrix entries. LDPC codes' design avoids Definition 8.3.2's "cycles of 4" so that Chapter 7's constraint-based iterative decoding works effectively.

Definition 8.3.2 *[cycle of 4] A* **cycle of 4** *occurs in a code when for any 1's in the (i,j) and (i,k) positions in any row i of H, there is at least one other row $i' \neq i$ that also has 1's in the jth and k^{th} positions.*

Definition 8.3.3 *[Regular Parity Matrix] A* **regular LDPC code** *has a regular parity matrix has exactly t_r 1's in every row and exactly t_c 1's in every column. Irregular linear binary LDPC block codes do not meet this condition.*

Regular LDPC codes' design chooses the $(n-k) \times n$ parity matrix H to have exactly t_c 1's in each column and exactly t_r 1's in each row, so that consequently

$$(n-k) \cdot t_r = n \cdot t_c \tag{8.76}$$

or equivalently

$$r = 1 - \frac{t_c}{t_r} \quad . \tag{8.77}$$

Essentially, if uniformly distributed integers from 1 to n were selected, t_r at a time, they would represent the positions of the 1's in a row of H. Successive rows could be generated by selecting successively groups of t_r integers. If a row is obtained that is either linearly dependent on previous rows, or forms a 4-cycle, then the row is discarded and the process continued. Elimination of all possible 4-cycle situations upper bounds the codeword length n for a given k and t_c according to[13]:

$$n \leq \underbrace{\binom{(n-k)}{2}}_{\text{choose 2 rows}} \Big/ \underbrace{\binom{t_c}{2}}_{\text{choose 2 cols}} = \frac{(n-k)\cdot(n-k-1)}{t_c \cdot (t_c - 1)} \quad . \tag{8.78}$$

(8.78)'s satisfaction causes the LDPC design to have long n for high-rate codes. On average, codes having such a parity matrix can have high coding gain.

Regular LDPC possible gains: Table 8.19 arises from some average-over-all-LDPC codes originating in Richardson and Urbanke [22]'s Table 2 .

(t_c, t_r)	b	γ_s offset	deviation from $\mathcal{C}_{\lvert C\rvert=2}$
(3,6)	.5	.184 dB	1.1 dB
(4,8)	.5	.184 dB	1.6 dB
(5,10)	.5	.184 dB	2.0 dB
(3,5)	.4	.051 dB	1.3 dB
(4,6)	1/3	.033 dB	1.4 dB

Table 8.19: Regular Low-Density Parity Code Average Performance.

Table 8.19 converts [22]'s results into a shaping offset from capacity, and then lists deviation from capacity reduced by this"shaping gain" that requires non-binary continuous random-code subsymbol distribution to attain. Codes with these gains or better require significant design effort and usually are irregular. Table 8.19's SQ QAM (PAM) constellation loss $\gamma_{s,offset}$ models some lower $r = \bar{b}$ constellation-restriction losses with respect to $\mathcal{C}$ in dB. Larger $\bar{b}$ values of $\gamma_s = \gamma_{s,offset}$ appear in Section 8.5's Table 8.21 for large n. Then

$$\gamma_{s,offset} = SNR_s - SNR \geq 0 \quad . \tag{8.79}$$

Table 8.19 and Table 8.21 permit piece-wise linear fits (in dB):

$$\gamma_{s,offset} = \begin{cases} 0.1 \cdot \bar{b} \text{ dB} & 0 \leq \bar{b} \leq 0.33 \\ 0.27 \cdot \bar{b} - .057 \text{ dB} & 0.33 \leq \bar{b} \leq 0.4 \\ 1.33 \cdot \bar{b} - .48 \text{ dB} & 0.4 \leq \bar{b} \leq 0.5 \\ 0.2 \cdot \bar{b} + .084 \text{ dB} & 0.5 \leq \bar{b} \leq 1 \\ 1 \cdot \bar{b} - .72 \text{ dB} & 1 \leq \bar{b} \leq 2 \\ 0.2 \cdot \bar{b} + .85 \text{ dB} & 2 \leq \bar{b} \leq 3 \\ 0.17 \cdot \bar{b} + .83 \text{ dB} & 3 \leq \bar{b} \leq 4 \\ 1.53 \text{ dB} & \bar{b} \geq 4 \end{cases} \quad . \tag{8.80}$$

Exceptionally Good LDPC Codes: Chung, Forney, Richardson and Urbanke [23] show that for the binary-input AWGN channel, a rate 1/2 code can be constructed that is within 0.0045 dB of the Shannon capacity (again reduced by the shaping gain of .184 dB. This code did not have a constant number of 1's per row or column (i.e., was not regular), and it appears such non-uniform distribution of 1's is necessary to approach capacity capacity very closely. One irregular code later described as an example for 5G cellular systems appears to perform in practice within 1 dB of the (shaping-gain reduced) capacity.

[13] If these 3 are known then of course t_r follows from (8.76)

Design Comment: This text instead provides some simpler generic LDPC designs that are better than Turbo Codes but fall about 1 dB short of best LDPC-alone designs, which typically use optimized irregular LDPC codes with gaps less than 1 dB from capacity. This text instead augments the desginer's choices with simpler way to achieve the extra 1-2 dB: which is to use Section 8.4's cyclic hard codes outside an inner regular or nearly regular LDPC code, as Section 8.6 describes. While simpler to understand and design, this will lead to additional decoding latency as Section 8.6 describes. As data rates increase relative to propagation times (which is a constant at best at speed of light), delays appear relatively small. For instance a delay of 4000 bits at 4 Mbps is 1ms, while that same block size at 4 Gbps is only $1\mu s$. The latter delay likely allows additional outer codes because other system design delays then dominate latency concern. At 4 Tbps, the delay is 1ns well below the channel propagation delay at 3.3 ns/meter. If this is the case, the simpler LDPC code may be reasonable. However, the highly complex irregular LDPC codes may find good application at data rates where additional delay is not acceptable, more likely at lower data rates that may eventually become obsolete.

There are various approaches to generation of the H matrix. Gallager originally used a recursive procedure in which smaller LDPC matrices are inserted into larger H matrices. However, this textbook considers codes that appear to be well suited for multi-level transmission, as well as for low-rate transmission, for which the construction follows the random procedure.

8.3.3.1 Generic LDPC Design

Cite multiple quasi-cyclic references here. This subsection's binary LDPC codes arise from a simple cyclic-H design methodology. Some software is provided to following the procedure, accepting input parameters t_c and t_r (and thus indirectly code rate). The codes presume use of Chapter 7's check-and-variable node iterative decoding. These codes apply to multilevel constellations when $\bar{b} > 1$ through BICM; shaping gain of 1.53 dB at large $\bar{b}$ (and less at smaller $\bar{b} > 1$) is independent, see Section 8.5.

Quasi-cyclic LDPC codes use a cyclic Vandermonde H matrix that depends on a $p \times p$ (p is a prime integer) circular shift matrix

$$J = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} . \tag{8.81}$$

The consequent parity matrix is ALMOST

$$H = \begin{bmatrix} I & I & ... & I & I \\ I & J & J^2 & ... & J^{t_r-1} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ I & J^{\tilde{t}_c-1} & J^{2(\tilde{t}_c-1)} & ... & J^{(t_r-1)(\tilde{t}_c-1)} \end{bmatrix} . \tag{8.82}$$

However, H's construction also deletes linearly dependent rows going from top to bottom and testing along the way. The codeword length (row length) remains constant at n bits. Further the number of ones per row t_r remains equal to the input design parameter leading to eventually $n - k < t_c \cdot p$ input bits. However row deletion can cause t_c to vary with column index so that t_c value now represents the maximum number of ones in any column. The number of deleted rows is always $m = t_c - 1$ so then

$$\tilde{t}_c \triangleq (t_c - 1) \cdot m + t_c \cdot \left(\frac{n-m}{n} \right) \tag{8.83}$$

is the average number of 1's per column, and the code is consequently irregular. The rate is

$$r = 1 - \frac{\tilde{t}_c}{t_r} . \tag{8.84}$$

Table 8.20 lists how many (m) rows of (8.82)'s parity matrix are linearly dependent and thus removed. This number is always $m = t_c - 1$. Consequently, Some columns have instead $t_c - 1$ ones in them, while $\tilde{t}_c$ is the average. The H construction averts all 4-cycles.

Narrowing the gap to capacity: Longer block length typically yields higher gain with this construction. Table 8.20 illustrates the generic LDPC codes and parameters when used for $r \geq .5$. The coding gain $r \cdot d_{free}$ is hard to find, and thus absent; however this coding gain is less helpful than the proximity to capacity because the error performance is not dominated by d_{min} (or equivalently d_{free}) for these codes' $\bar{P}_b$ results. Instead, Table 8.20 lists an effective coding gain that compares the SNR required to obtain measured $\bar{P}_b = 10^{-7}$ with 14.2 dB for uncoded $r = 1$ transmission at the same rate. This table lists also SNR gap Γ to true capacity at $\bar{P}_b = 10^{-7}$ while using (8.80) to adjust parameters reported publicly [25] for these codes. This gap is not constant for LDPC codes when $\bar{b} \leq 1/2$. Thus, with BICM and larger constellations with a shaping code, it is possible to reduce the gap to capacity further than the amount listed (by up to about .7 dB for the lowest rate code listed and 1dB for the highest rate listed with the very best shaping code at 1.53 dB limit, but in practice perhaps .5 dB). To further reduce proximity to capacity, Section 8.6's outer hard code could be used for the last dB or so with small additional rate loss.

(n,k)	m	p	t_c	t_r	r	$\gamma_{s,offset}$ (8.80)	Γ at 10^{-7}	$\gamma_{f,eff}$ at 10^{-7}
(276,209)	2	23	3	12	.7572	0.69 dB	4.5 dB	5.1 dB
(529,462)	2	23	3	23	.8733	0.93 dB	3.9 dB	5.7 dB
(1369,1260)	2	37	3	37	.9204	1.01 dB	3.3 dB	6.3 dB
(2209,2024)	3	47	4	47	.9163	1.01 dB	2.8 dB	6.8 dB
(4489,4158)	4	67	5	67	.9263	1.03 dB	2.5 dB	7.1 dB
(7921,7392)	5	89	6	89	.9332	1.04 dB	2.3 dB	7.3 dB

Table 8.20: Generic LDPC code parameters.

Figure 8.26 illustrates the generic LDPC codes gap to capacity at $\bar{P}_b = 10^{-7}$:

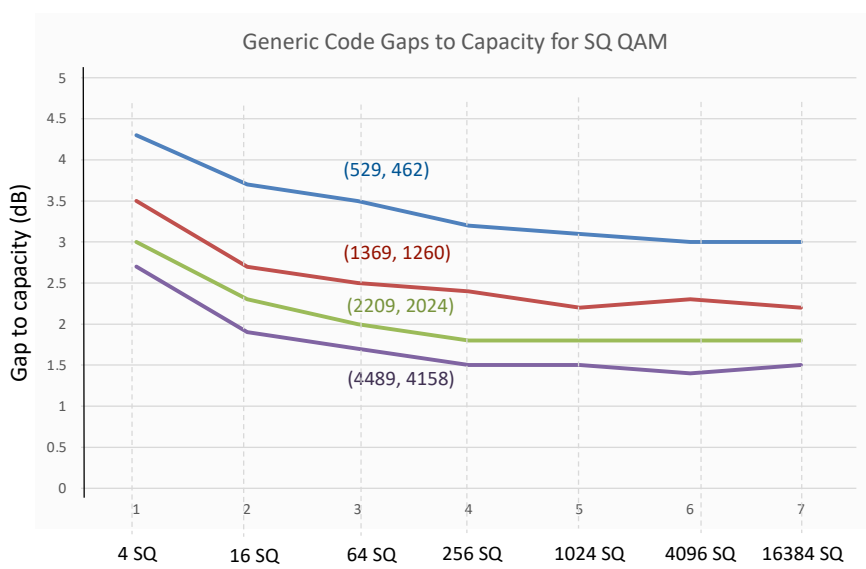

Figure 8.26: Illustration of nearly constant gap to capacity for integer $\bar{b}$ for 4 example LDPC codes at $\bar{P}_b = 10^{-7}$.

8.3.3.2 Feeding the Decoder

To obtain the initial intrinsic probability from an AWGN received channel output value, the mapping allows each bit's value of the encoder output to be identified with one or more points in the constellation. This set of points is called either $X_{v_m=1}$ or $X_{v_m=0}$ The probability is obtained by the following sum

$$p(v_k = 1) = \sum_{x \in X_{v_k=1}} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(y-x)^2} \cdot p_x \quad . \tag{8.85}$$

A similar sum is computed for $v_k = 0$. Then, LDPC decoding is initialized with the prior given by

$$\text{LLR}_k = \ln \frac{p(v_k = 1)}{p(v_k = 0)} \quad . \tag{8.86}$$

8.3.3.3 Generic LDPC code construction

Former EE379 student Chien-Hsin Lee[14] provides the base software routines here to construct the generic LDPC codes, both parity and systematic generator matrices. Lee also provided as well encoder and decoder functions below that are useful to evaluate actual error probability for the codes.

The programs have comments and call information enumerated here, while complete listings are in Appendix G, Section 8.

get_h_matrix.m : This program generates the generic LDPC code parity matrix according to specified t_r and t_c parameters.

```
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  function [H_no_dep H] = get_h_matrix(p,tr,tc,first_1);
  Generate LDPC H Matrix Uses IBM's Method As Per Cioffi's Class Note
  Example: to Generate (529,462) code, p=23, rw=23, cw=3, first_1=2
           H = get_h_matrix(23,23,3,2),
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  Definition of input variables
  p      :  Prime number of the size of base matrix of size p-by-p
  tr     :  Row weight = # of base matrices (or 1's) /row, equivalent to K
  tc     :  Col weight = # of base matrices (or 1's) per column,eq to J
  first_1:  Set to 2 in generic LDPC code, so right shift by first_1-1
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  Definition of output variables
  H_no_dep  : the parity check matrix with no dependent rows
  H         : without removing the dependent rows
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  EE379, Chien-Hsin Lee, first version 06/2006, edits by J. Cioffi since
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

systematic.m A second routine provides a systematic encoder for a block code's input H matrix. The full program listing is in Appendix G.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  function [H_sys, G_sys] = systematic(H);
  This routine removes any dependent rows from an H parity matrix.
  I then finds the systematic G and H matrices
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  Definition of input variables
  H      : parity-check matrix
```

---

[14] Now Sr. Director Business Development, TSMC, Taiwan - formerly General Manager of Rambus Memory and Chip.

```
%%%
```

```
  EE379, 06/2006, Chien-Hsin Lee, subsequent edits J. Cioffi
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% This is the main LDPC simulation bench
%% This sets up to run the (529,462) code
%% This will take about 10 hrs to run
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% EE379, Chien-Hsin Lee, 06/2006 ; consequent edits J. Cioffi
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Example uses below to complete tables.

```
H = get_h_matrix(23,23,3,2)
>> [H_sys, G_sys] = systematic(H);
>> size(G_sys) =   462   529

> H = get_h_matrix(23,12,3,2);
>>  [H_sys, G_sys] = systematic(H);
>> size(G_sys) =   209   276

>> H = get_h_matrix(47,47,4,2);
>> [H_sys, G_sys] = systematic(H);
size(G_sys)  =        2024        2209

>> H = get_h_matrix(89,89,6,2);
>> [H_sys, G_sys] = systematic(H);
size(G_sys)
```

8.3.3.4 5G LDPC Codes

See Lecture 10, late slides. To be added at later date.

Please see [28] and [29].

8.3.4 Polar Codes

8.4 Cyclic Codes

8.5 Shaping Codes

Shaping codes for the AWGN improve shaping gain up to a maximum of 1.53 dB (see also Problems **??** and **??**). Following a review of this limit, this section proceeds to the 3 most popular shaping-code methods.

The shaping gain of a code is again

$$\gamma_s \triangleq \frac{\frac{V_{\tilde{\boldsymbol{x}}}^{2/N}}{\bar{\mathcal{E}}_{\tilde{\boldsymbol{x}}}}}{\frac{V_{\boldsymbol{x}}^{2/N}}{\bar{\mathcal{E}}_{\boldsymbol{x}}}} \quad . \tag{8.87}$$

Presuming the usual Z^N-lattice cubic reference, best shaping gain occurs for a spatially uniform distribution of constellation points within a hypersheric volume[15] . For an even number of dimensions $N = 2n$, an N-dimensional sphere of radius r is known to have volume

$$V(\text{sphere}) = \frac{\pi^n \cdot r^{2n}}{n!} \quad , \tag{8.88}$$

and energy

$$\bar{\mathcal{E}}(\text{sphere}) = \frac{1}{2}\left[\frac{r^2}{n+1}\right] \quad . \tag{8.89}$$

The asymptotic equipartition analysis of Chapter 2 suggests that if the number of dimensions goes to infinity, so that $n \to \infty$, then there is no better situation for overall coding gain and thus for shaping than the uniform distribution of equali-size regions throughout the volume. For the AWGN, the marginal one- (or two-) dimensional input distributions are all Gaussian at capacity and correspond to a uniform distribution over an infinite number of dimensions.[16] Since the coding gain is independent of region shape, then the shaping gain must be maximum when the overall gain is maximum, which is thus the uniform-over-hypershere/Gaussian case. Thus, the asymptotic situation provides an upper bound on shaping gain. The uncoded reference used throughout this text is again

$$\bar{\mathcal{E}}_{\boldsymbol{x}} = \frac{1}{12}\left(2^{2\bar{b}} - 1\right) \tag{8.90}$$

$$V_{\boldsymbol{x}}^{2/N} = 2^{2\bar{b}} \cdot 1 \quad . \tag{8.91}$$

Then, algebraic subsitution of (8.88) -(8.91) into (8.87) leads to the **shaping-gain bound**

$$\gamma_s \leq \frac{\pi r^2 (n!)^{-\frac{1}{n}} / \frac{r^2}{2(n+1)}}{12\left(1 - 2^{-2\bar{b}}\right)} = \frac{\pi}{6}\frac{(n!)^{-\frac{1}{n}} \cdot (n+1)}{1 - 2^{-2\bar{b}}} \quad . \tag{8.92}$$

Table 8.21 evaluates the formula in (8.92) for $b \to \infty$ while Table 8.22 evaulates this formula for $n \to \infty$. With $\bar{b}$ infinite, and then taking limits as $n \to \infty$ for the asymmptotic best case using Stirling's approximation (see Problems **??** and **??**),

$$\lim_{n\to\infty} \gamma_s \leq \lim_{n\to\infty} \frac{\pi}{6} \cdot \frac{n+1}{(n!)^{\frac{1}{n}}} = \frac{\pi}{6} \lim_{n\to\infty} \frac{n+1}{\frac{n}{e}} = \frac{\pi \cdot e}{6} = 1.53 \text{ dB} \quad . \tag{8.93}$$

Equation (8.92) provides a shaping-gain bound for any finite b and n, and corresponds to uniform density of the 2^b points with a $2n$-dimensional sphere. Equation (8.93) is the overall asymptotically attainable bound on shaping gain.

[15] Inspection of (8.87)'s numerator observes that for any given radius, which is directly proportional to the square root of energy $\sqrt{\bar{\mathcal{E}}_{\tilde{\boldsymbol{x}}}}$, a sphere has largest volume and thus largest number of points if they can be uniformly situated spatially.

[16] A simple Gaussian proof sketch: A point component in any dimension can be obtained by linear projection, which has an infinite number of terms, thus satisfying the central limit theorem in infinite dimensions. Thus all the components' (or marginal) distributions are Gaussian. QED.

$\bar{b}$	$\gamma_s(n \to \infty)$
∞	1.53 dB
4	1.52 dB
3	1.46 dB
2	1.25 dB
1	0.28 dB
0	0 dB

Table 8.21: Shaping gain limits for an infinite number of dimensions.

$n(N)$	$\gamma_s(\bar{b} \to \infty)$
1 (2)	0.20 dB
2 (4)	0.46 dB
3 (6)	0.62 dB
4 (8)	0.73 dB
12 (24)	1.10 dB
16 (32)	1.17 dB
24 (48)	1.26 dB
32 (64)	1.31 dB
64 (128)	1.40 dB
100 (200)	1.44 dB
500 (64)	1.50 dB
1000 (2000)	1.52 dB
10000 (20000)	1.53 dB

Table 8.22: Shaping gain limits for an infinite number of constellation points.

8.5.1 Non-Equiprobable Signaling and Shell Codes

Shell constellations were introduced by former EE379 student Paul Fortier [26]. They are conceptually straightforward and partition any constellation into shells (or rings). Each "shell" is a group of constellation points contained on a shell (or circle in 2D) as in Figure 8.27. Shell (SH)constellations have all points within a group (or shell) at constant energy. These shell groups are indexed by $i = 1, ..., G$, and each has energy $\mathcal{E}_i$ and contains M_i points. The probability that a point from group i is selected by an encoder is p_i. For the example of Figure 8.27, there are 5 shell groups with two-dimensional energies (for $d = 2$) $\mathcal{E}_1 = 2$, $\mathcal{E}_2 = 10$, $\mathcal{E}_3 = 18$, $\mathcal{E}_4 = 26$, $\mathcal{E}_5 = 34$. Each contains $M_1 = 4$, $M_2 = 8$, $M_3 = 4$, $M_4 = 8$, and $M_5 = 8$ points respectively. The shells have respective probabilities $p_1 = 1/8$, $p_2 = 1/4$, $p_3 = 1/8$, $p_4 = 1/4$ and $p_5 = 1/8$ if each point in the constellation is equally likely to occur. The average energy of this constellation can be easily computed as

$$\mathcal{E}_{\boldsymbol{x}} = \sum_{i=1}^{G} p_i \cdot \mathcal{E}_i = \left[\frac{2}{8} + \frac{10}{4} + \frac{18}{8} + \frac{26}{4} + \frac{34}{4}\right] = 20 \quad , \tag{8.94}$$

with consequent shaping gain

$$\gamma_s = \frac{(1/6)31 \cdot d^2}{20} = \frac{31}{30} = 0.14 \text{ dB} \quad . \tag{8.95}$$

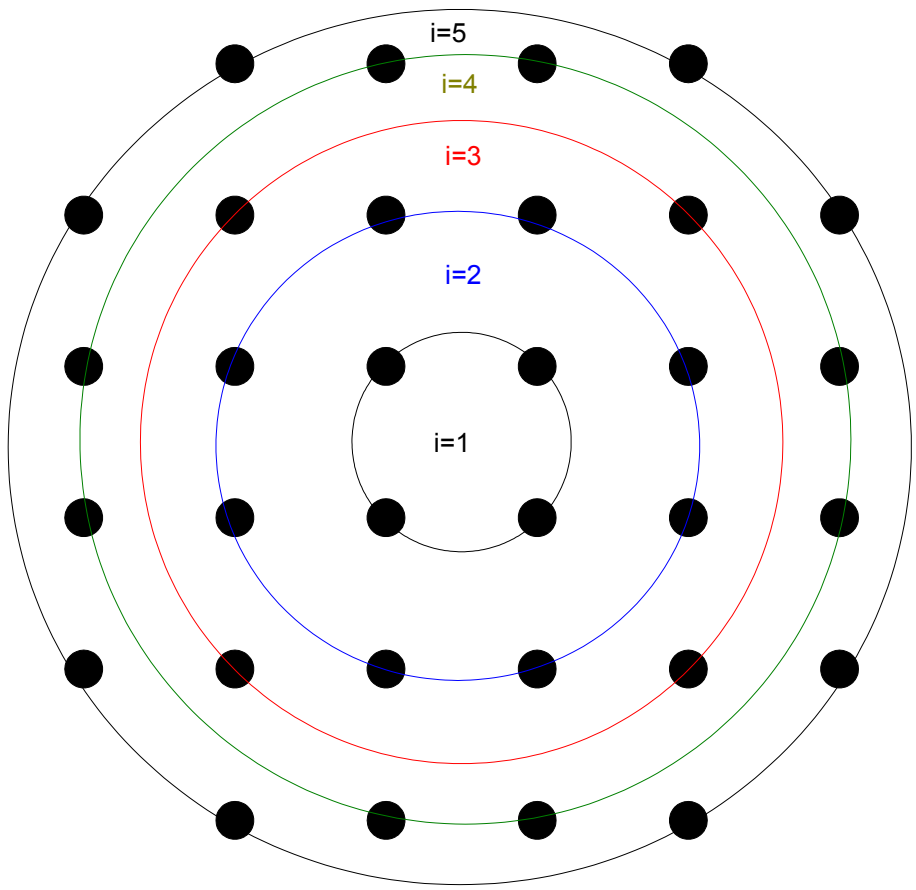

Figure 8.27: Shell Construction Illustration for 32SH (=32CR).

The rectangular-lattice constellation (a coset of $2Z^2$) in 2D that uses the least energy for 32 points is 32CR=32SH, because it corresponds to using the shells with the least successive energies first.

128CR does not have such a least-energy property. A 128SH constellation is more complicated. The 128-point shell constellation 128SH constellation uses 17 shells of energies (number of points) (for $d = 2$) of 2 (4), 10 (8), 18 (4), 26 (8), 34 (8), 50 (12), 58 (8), 74 (8), 82 (8), 90 (8), 98 (4) , 106 (8) , 123 (8), 130 (16), 146 (8), 162 (4), and 170 (4 of 16 possible). The 128SH uses the points $(\pm 9, \pm 9)$ that do not appear in 128CR. The shaping gain of 128SH is .17 dB and is slightly higher than the .14 dB of 128CR. Problem 8.10 discusses a 64SH constellation (which is clearly not equal to 64SQ).

The number of bits per dimension can approach the entropy per dimension with well-designed input buffering (and possibly long delay) as in Chapter 8. While a constellation may have a uniform density over N dimensions, the marginal distribution for one- or two-dimensional constituent subsymbols need not be uniform. Sometimes this is used in shaping to effect large-dimensional γ_s with unequal probabilities in a smaller number of dimensions. The entropy per symbol of any constellation with point probabilities p_{ij} for the jth point in the ith group upper bounds the bits per one-or-two-dimensional, or any finite-dimensional, symbol

$$b \le H = \underbrace{\sum_{i=1}^{G}}_{\text{shells}} \underbrace{\sum_{j=1}^{M_i}}_{\text{pts in shell}} p_{ij} \log_2 \left(\frac{1}{p_{ij}} \right) \quad . \tag{8.96}$$

Equation (8.96) holds even when the groups are not shells. When all points are equally likely, b is simply computed in the usual fashion as

$$b = H = \sum_{i=1}^{G} \sum_{j=1}^{M_i} \frac{1}{M} \log_2 (M) = \log_2(M) = \log_2 \left(\sum_i M_i \right) \quad . \tag{8.97}$$

When each point within a group has equal probability of occuring (but the group probability is not necessarily the same for each group as in the 32CR and 128SH examples above), $p_{ij} = \frac{pi}{M_i}$, the same H can be written

$$H = \sum_{i=1}^{G} \frac{p_i}{M_i} \sum_{j=1}^{M_i} \log_2(\frac{M_i}{p_i}) = \sum_{i=1}^{G} p_i \log_2(\frac{M_i}{p_i}) \quad , \tag{8.98}$$

which if $b_i \triangleq \log_2(M_i)$, then becomes

$$H = \sum_{i=1}^{G} p_i \log_2 \left(\frac{1}{p_i} \right) + \sum_{i=1}^{G} p_i \cdot b_i = H(\boldsymbol{p}) + \sum_{i=1}^{G} p_i \cdot b_i \quad , \tag{8.99}$$

where $H(\boldsymbol{p})$ is the entropy of the group probability distribution. good shaping code design would cause $b = H$, as is evident later. Equation (8.99) supports the intuition that the over all data rate is the sum of the data rates for each group plus the entropy of the group probability distribution.

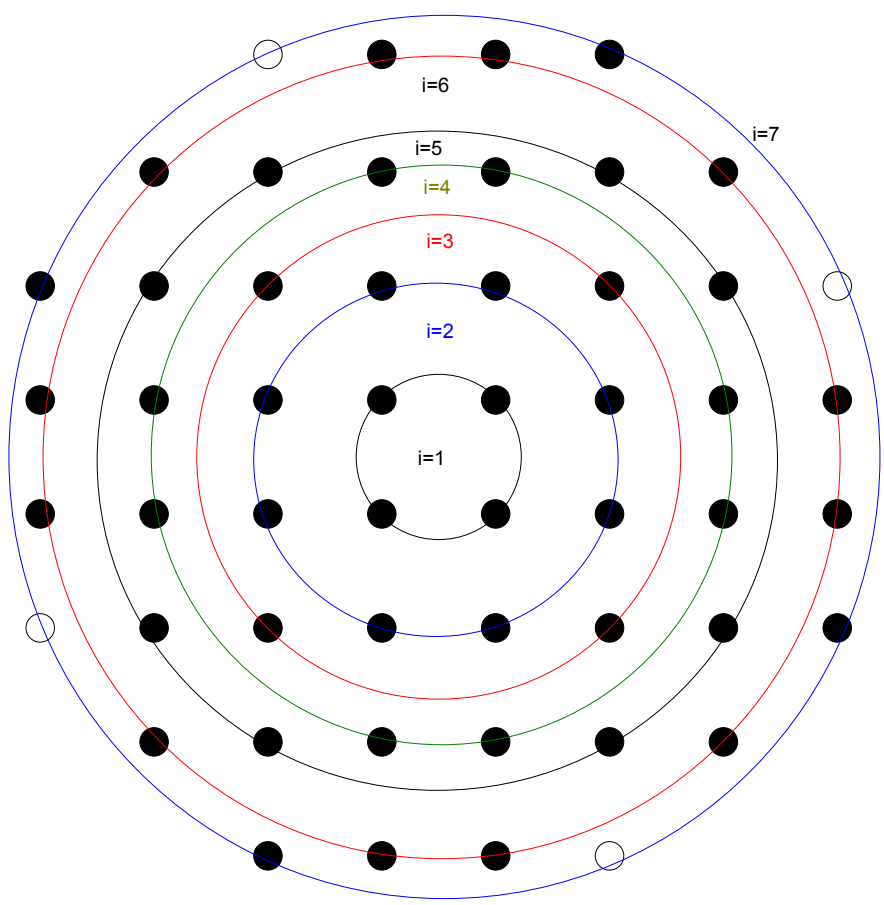

Figure 8.28: 48SH Constellation (with extra 4 "empty" points for 52SH).

The one-dimensional slices of the constellation should favor points with smaller energy; that is smaller energies have a higher probability of occurrence. For instance in 32CR, a one-dimensional slice has $p_{\pm 1} = \frac{12}{32}$, $p_{\pm 3} = \frac{12}{32}$, but $p_{\pm 5} = \frac{8}{32}$. As the dimensionality $N \to \infty$, the enumeration of shells can be very tedious, but otherwise follows the two-dimensional examples above. Furthermore as $N \to \infty$ the one-dimensional distribution of amplitudes (for large numbers of points) approaches Gaussian[17] with very large amplitudes in any dimension having low probability of occurrence. Such low probability of occurrence reduces energy of that dimension, thus increases shaping gain. The non-uniform distribution in a single (or two) dimensions suggests that a succession of two-dimensional constellations viewed as a shaping-code codeword might have extra points (that is more than $2^{2\bar{b}}$, as also in trellis codes) with points in outer shells (or groups) having lower probability than an equally likely $2^{-2\bar{b}}$ in one dimension. For instance, a 32CR constellation might be used with $\bar{b} = 2 < 2.5$, but with points in the outer shells at much lower probability of occurence than points in the inner shell when viewed over a succession of $N/2$ 2D-constellations as one large shaping codeword. Then, the two-dimensional "M" in equation (8.99) is greater than $2^{2\bar{b}}$, but Equation (8.99) still holds with b directly computed even though $p_{ij} \neq \frac{1}{M}$, but $p_{ij} = p_i \cdot \frac{1}{M_i}$. The number of points in two dimensions is not the $\left(\frac{N}{2}\right)^{th}$ root of the true N-dimensional M. Indeed,

$$b = H(\boldsymbol{p}) + \sum_{i=1}^{G} p_i \cdot \log_2(M_i) \tag{8.100}$$

is a good equation to use since it avoids the confusion on M applying to a larger number of dimensions than 2.

Such was the observation of Calderbank and Ozarow in their **Nonequiprobable Signalling (NES)**[27]. In NES, A 2D constellation is expanded and viewed as a part of a larger multi-dimensional constellation. INES chooses outer groups/shells less frequently than inner groups (or less frequently than even $\frac{M_i}{2^{2\bar{b}}}$).

A simple example of nNES was the 48CR constellation used for 4D constellations earlier in this chapter. In that constellation, the 32 inner two-dimensional points had probability of occurence 3/4 while the 16 outer points had probability only 1/4. While this constellation corresponds also to a 2D shell constellation (see Figure 8.28), the probabilities of the various groups are not simply computed by dividing the number of points in each group by 48. Furthermore,the constellation is not a 4D shell, but does have a higher shaping gain than even the two-dimensional 128SH. In one dimension, the probabilities of the various levels are

$$p_{\pm 1} = \frac{12}{32} \cdot \frac{3}{4} + \frac{4}{16} \cdot \frac{1}{4} = \frac{11}{32} \tag{8.101}$$

$$p_{\pm 3} = \frac{12}{32} \cdot \frac{3}{4} + \frac{2}{16} \cdot \frac{1}{4} = \frac{10}{32} \tag{8.102}$$

$$p_{\pm 5} = \frac{8}{32} \cdot \frac{3}{4} + \frac{4}{16} \cdot \frac{1}{4} = \frac{8}{32} \tag{8.103}$$

$$p_{\pm 7} = 0 + \frac{6}{16} \cdot \frac{1}{4} = \frac{3}{32} \quad . \tag{8.104}$$

Equations (8.101 -(8.103) illustrate that large points in one dimension have low probability of occurrence. This 48CR constellation can be used to transmit 5.5 bits in two dimensions, emphasizing that $2^{5.5} < 48$ so that the points are not equally likely. An astute reader might note that 4 of the points in the outer most 2D shell in Figure 8.28 could have been included with no increase in energy to carry an additional $\frac{1}{4} \cdot \frac{4}{16} \cdot 1 = \frac{1}{16}$ bit in two dimensions. The energy of 48CR (and of thus of 52CR) is

$$\mathcal{E}_{\boldsymbol{x}} = \frac{3}{4} 20 + \frac{1}{4} \left(\frac{50 \cdot 12 + 58 \cdot 4}{16} \right) = 28 \quad , \tag{8.105}$$

leading to a shaping gain of for $b = 5.5$ of

$$\gamma_s = \frac{(1/6) \cdot 2^{5.5} \cdot 4}{28} = .23 \text{ dB} \quad , \tag{8.106}$$

[17] The result if viewed in terms of 2D constellations is that the 2D complex amplitudes approach a complex Gaussian distribution.

and for 52CR's $b = 5.625$ of .42 dB. The higher shaping gain of the 52SH simply suggests that sometimes best spherical approximation occurs with a number of points that essentially covers all the points interior to a hypersphere and is not usually a nice power of 2.

The 48CR/52CR example is not a shell constellation in 4D, but does provide good shaping gain relative to 2D because of the different probabilities (3/4 and 1/4) of inner and outer points. Essentially, 48CR/52CR is a 4D shaping method, but uses 2D constellations with unequal probabilities. The four-dimensional probability of any of the 4D 48SH/52SH points is still uniform and is $2^{-4\bar{b}}$. That is, only the 2D component constellations have non-uniform probabilities. Again, for large number of constellation points and infinite dimensionality, the uniform distribution over a large number of dimensions with an average energy constraint leads to Gaussian marginal distributions in each of the dimensions.

For the 48CR/52CR or any 4D inner/outer design with an odd number of bits per symbol, Table **??** provided a nonlinear binary code with 3 bits in and 4 bits out that essentially maps input bits into two partitions of the inner group or the outer group. The left-inner and right-inner partitioning of Table **??** is convenient for implementation, but the essence of that nonlinear code is the mapping of inputs into a description of "inner" group (call it "0") and "outer" group (call it "1"). This general description would like as few 1's (outers) per codeword as possible, with the codewords being (0,0), (1,0), and (0,1) but never (1,1). For even numbers of bits in four dimensions, Table **??** was not necessary because there was no attempt to provide shaping gain – in Section 8.6, the interest was coding gain and the use of inner/outer constellations there was exclusively to handle the "half-bit" constellations, coincidentally providing shaping gain. However, the concept of the 3 codewords (0,0), (1,0), (0,1) remains if there are extra points in any 2D constellation beyond $2^{2\bar{b}}$. The set of 2D constellation points would be divided into two groups, inner and outer, where the size of the outer group M_1 is most easily exactly[18] 1/2 the size of the inner group M_0 if the probabilities of the groups are to remain 3/4 and 1/4 respecively. This type of view of course begs the question of generalization of the concept.

Calderbank and Ozarow essentially generalized this "low maximum number of 1's" concept in an elegant theory that is simplified here for design. This text calls these "NE" (Non-Equal probability) codes. NE codes require the partitioning of the constellation into equal numbers of points in each group. For a binary (nonlinear) code that spans n dimensions[19] where no more than m ones occur, the NE shaping code rate is easily

$$2\bar{b} = \frac{1}{n}\log_2\left\{\sum_{j=0}^{m}\binom{n}{j}\right\} \quad , \tag{8.107}$$

and the probability of a 0 in any position is given by

$$p_0 = \frac{\sum_{j=0}^{m}\binom{n}{j}\cdot(n-j)}{n\cdot\sum_{j=0}^{m}\binom{n}{j}} \quad . \tag{8.108}$$

The probability of a 1 is then easily $p_1 = 1 - p_0$. Such an NE code is labelled $B_{n,m}$ for "binary" code with n positions and no more than m of these positions containing 1's. The 1's correspond to the outer group of points in a 2D constellation, while the 0's correspond to inner points in the same constellation. Such a binary code then specifies which points are inner or outer. The numerator of the code rate in (8.107) is unlikely to be an integer, so an efficient implementation of the mapping into codewords may have to span several successive codewords. Alternatively, the least integer in the numerator can instead be chosen. For instance, a $B_{12,3}$ code has 299 codewords and thus rate $\frac{\log_2(299)}{12} = .6853$ and might likely be rounded down to $\frac{8}{12}$ so that 8 bits enter a non-linear memory (look-up table like that of Table **??**) while 12 bits leave. In such a code in 12 successive 2D symbols, no more than 3 symbols would be outer points. The shaping gain of such a code is easily and exactly computed from the p_0 in (8.108) as

$$\gamma_s = \frac{\frac{1}{6}\left(2^{2\bar{b}-1}\right)\cdot d^2}{p_0\cdot\mathcal{E}_0 + (1-p_0)\cdot\mathcal{E}_1} \leq 1.53 \text{ dB} \; . \tag{8.109}$$

[18] Please note we've re-indexed the groups starting with 0 in this case.

[19] these dimensions correspond to 2D sub-constellation points in a $2n$-dimensional shaping-code symbol or codeword.

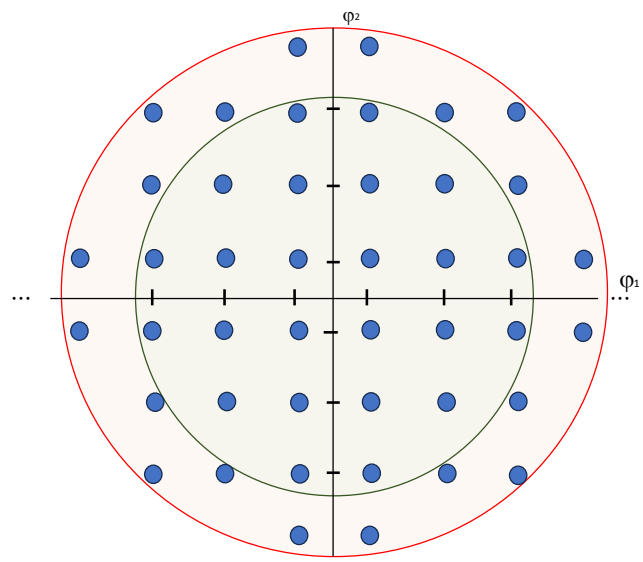

Figure 8.29: 44SH Constellation.

Calderbank and Ozarow provide a number of contellation-size independent approximations for various large block lengths and large numbers of constellation points in their work, but this author has found that direct calculation of the exact gain is probably more covenient, even though it requires knowledge of the exact constellation and inner-outer energies.

EXAMPLE 8.5.1 *[8D 40CR and 44SH constellation]* The 8D 40CR constellation of Figure **??** is also a 40SH constellation in 2D, but is not an 8D shell constellation. The 4 points $(\pm 5, \pm 5)$ were previously not used, so they are added here to form 44SH, which is divided into two equal groups of 22 points as shown in Figure 8.29. The nonlinear $B_{4,1}$ code rate is

$$2\bar{b} = (1/4) \cdot \log_2(1+4) = \frac{log_2(5)}{4} = .5805 \quad . \tag{8.110}$$

The probability of an inner point with straightforward use of the $B_{4,1}$ NE is computed as

$$p_0 = \frac{1 \cdot 4 + 4 \cdot 3}{4 \cdot (1+4)} = \frac{4}{5} \quad , \tag{8.111}$$

and thus the outer point has probability 1/5. The overall shaping-code rate is then

$$\log_2(22) + \mathcal{H}(.2) = 4.4594 + .7219 = 5.1814 \text{ bits per 2D symbol} \quad . \tag{8.112}$$

The shaping gain requires the two energies, so

$$\mathcal{E}_0 = \frac{4 \cdot 2 + 8 \cdot 10 + 4 \cdot 18 + 6 \cdot 26}{22} = \frac{316}{22} = 14.3636 \tag{8.113}$$

$$\mathcal{E}_1 = \frac{2 \cdot 26 + 8 \cdot 34 + 12 \cdot 50}{22} = \frac{872}{22} = 39.6364 \quad , \tag{8.114}$$

and

$$\gamma_s = \frac{(1/6)(2^{5.1814} - 1) \cdot 4}{.8 \cdot 14.3636 + .2 \cdot 39.6364} = \frac{23.5243}{19.4182} = 1.2115 = .83 \text{ dB} \quad . \tag{8.115}$$

This compares favorably with the original use in trellis coding where the shaping gain was only .15 dB. Table 8.22 has a bound for 8 dimensional shaping that is .7 dB, an apparent contradiction with the gain in this example. However, those bounds are for infinite numbers of points in the constellation and do not consider the nuances in detail of specific constellations, so .8dB (which is still less than the maximum of 1.53 dB, which always holds for any number of points) is not unreasonable, especially when one considers that a code rate of .5805 would effectively need to be implemented over a large number of dimensions anyway (thus, the example could not truly be called "8-dimensional" because the bit-fractioning process itself can add shaping gain as in Section 8.6's 40CR example earlier.) Unfortunately, realization constraints will reduce this shaping gain. For instance, an implementation might choose nonlinear code rate of $\bar{b} = .5625 = 9/16$. Since there are 5 possible codewords ([0000], [1000], [0100], [0010], [0001]) per symbol, 4 successive codewords would have $5^4 = 625$ possibilities of which $2^9 = 512$ with the least energy would be used, reducing code rate to 9/16. Of the 625 possibilities, 500 will have a 0 in any particular position and $125 > 113$ would have a 1 in that same particular position. There are 256 codewords with four 1's in the orginal code. 113 of these can be deleted. The probability that a 1 is in the same particular position after deletion would then be .2(113/256)=.0883.

$$\log_2(22) + \mathcal{H}(.0883) = 4.4594 + .4307 = 4.89\text{bits per 2D symbol} \quad . \tag{8.116}$$

The shaping gain would then be a lower value of

$$\gamma_s = \frac{(1/6)(2^{4.89} - 1) \cdot 4}{.9117 \cdot 14.3636 + .0883 \cdot 39.6364} = \frac{19.102}{16.5947} = 1.15 = .60 \text{ dB} \quad , \tag{8.117}$$

and effectively corresponds to a 32-dimensional realization and is below the bound for both 8 and 32 dimensions. The original 8D trellis coded system by coincidence had .15 dB of shaping gain simply because the 40CR constellation was closer to a circle than a square. Interestingly enough, with no increase in energy, that constellation could have transmitted another 4 points and thus $(1/8)(1/2)(1) = 1/16$ bit/2D. The shaping gain would thus increase by $(2^{5.25+.0625} - 1)/(2^{5.25+.0625} - 1) = 1.0455 = = .19$ dB, to become then .15+.19 = .34 dB. Clearly the earlier system with trellis code is easier to implement in terms of look-up tables, but NE codes do have a gain as this example illustrates.

The above example illustrates that at least when groups are chosen in cocentric fashion that NE provides good shaping gain in effectively a few dimensions with reasonable complexity. Calderbank and Ozarow found some limitation to the use of binary codes. So for large number of constellation points, they found shaping gains for binary $B_{n,m}$ codes to be limited to about .9 dB for up to $n = 20$. An obvious extension is to use more groups, but then the code is no longer binary. NE codes with 4 cocentric groups make use of 2 binary codes for the each of the bits in a 4-ary label to the group, so for a concatenation of two $B_{m,k}$ codes

$$0 \rightarrow 00 \tag{8.118}$$
$$1 \rightarrow 01 \tag{8.119}$$
$$2 \rightarrow 10 \tag{8.120}$$
$$3 \rightarrow 11 \tag{8.121}$$

The probabilities of the 4 groups are then

$$p_{00} \rightarrow p_1 \cdot p_2 \tag{8.122}$$
$$p_{01} \rightarrow (1 - p_1) \cdot p_2 \tag{8.123}$$
$$p_{10} \rightarrow (1 - p_2) \cdot p_1 \tag{8.124}$$
$$p_{11} \rightarrow (1 - p_1) \cdot (1 - p_2) \quad . \tag{8.125}$$

Energy and shaping gain then follow exactly as in general with 4 groups contributing to all calculations. An extra .2 dB for sizes up to $n = 20$ for a total shaping gain of 1.1 dB was found by Calderbank and Ozarow for 4 groups.

Essentially, the designer needs to compute exact gain for particular constellations, but generally Calderbank and Ozarow's NE codes will provide good shaping gain for reasonable constellation sizes.

8.5.2 Voronoi/Block Shaping

Forney's **Voronoi constellations** have shaping boundaries that approximate a hypersphere in a finite number of dimensions. These boundaries are scaled versions of the decision (or "Voronoi") regions of dense lattices. The closeness of the approximation depends on the lattice selected to create the constellation boundary – very good lattices for coding will have decision regions that approach hyperspheres, and thus can also provide good boundaries for constellation design[20]. This subsection describes a procedure for encoding and decoding with Voronoi shaping regions and also enumerates the possible shaping gains for some reasonable and common lattice choices. This subsection prepares the reader for the generalization to trellis shaping in Subsection 8.5.3.

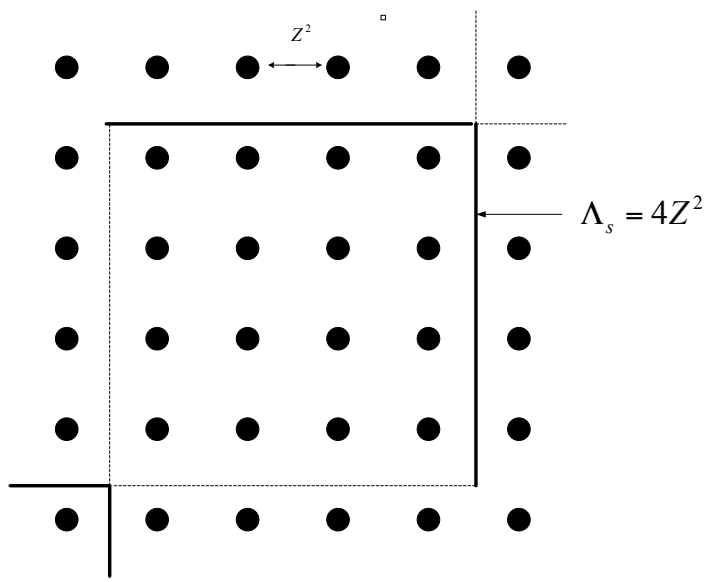

Figure 8.30: Tesselation of two-dimensional space by shaping lattice $4Z^2$ with constellation based on $Z^2 + (\frac{1}{2}, \frac{1}{2})$

Since nearly all the codes in this chapter (and the next) are based on (cosets of) origin-centered Z^N lattices (usually coded or uncoded sequences of one- or two-dimensional sub-symbols), the objective will be to cirumscribe such points by the origin-centered larger Voronoi shaping region of some reasonable scaled-by-power-of-two lattice. The Voronoi shaping regions of such a scaled lattice will **tesselate** the N-dimensional space into mutually disjoint regions. Tesselation means that the union of all the Voronoi regions is the entire space, and so there are no interstial gaps between the Voronois shaping regions. Points on Voronoi-region boundaries need to be assigned in a consistent way so that each such region is closed on some of the faces of the Voronoi region and is open on the others. Each such Voronoi shaping region should essentially "look the same" in terms of the center of the region relative to which boundaries are open and closed. A two-dimensional example appears in Figure 8.30. Figure 8.30 shows a Voronoi region of $4Z^2$, which is a square of side 4. The closed boundaries (solid lines) are the upper and right-most faces, while the open boundaries (dashed lines) are the lower and left-most faces. Such two-dimensional squares with closed-open boundaries precisely tesselate two-dimensional space. Each contains 16 points from $Z^2 + (\frac{1}{2}, \frac{1}{2})$. In general, the number of included points is the ratio of the volume

[20]There are limiting definitions of lattice sequences that do approach hyperspheres and thus capacity, but the practical interest in such systems is for a smaller number of dimensions like 4, 8, 16, 24, or 32.

of the Voronois shaping region to the volume of the underlying constellation lattice's decision region, $(4^2/1^2) = 16$ in Figure 8.30.

In general, one-half the boundaries should be closed and one-half should be open. The closed boundaries should be a connected set of points (that is any point on the closed part of the boundary can be connected to any other point on the closecd part of the boundary by passing through other boundary points that are included in the closed part of the boundary). Thus, the use of top and bottom for the closed boundaries and left/right for the open boundaries is excluded from consideration. Constellation points on a closed face of the shaping lattice's Voronoi region are included, while those on an open face are not included if such a situation arises (it does not arise in Figure 8.30 because of the coset offset of $(\frac{1}{2}, \frac{1}{2})$). In some cases including the rest of this section, it may be easier to add coset offsets as the last step of encoding, and so situations of points on boundaries may occur prior to such a last coset-offset step.

While the concept is easily pictured in two-dimensions, the encoding of points in multiple dimensions nominally might require a large table-look-up operation as well as a difficult search to find which points are in/out of the Voronoi region of the selected shaping lattice of size $M \cdot V(Z^N)$. Figure 8.31 provides a simplified implementation that uses a maximum-likelihood search in the encoder. The ML search (say implemented by Viterbi algorithm, as for instance for the lattices studied earlier in Section 8.5 of this chapter) finds that point in the shaping lattice $\boldsymbol{x}_s$ that is closest to an input point $\boldsymbol{x}$ in Z^N. Any such closest point has an error that is within the Voronoi/decision region. Thus, the error is transmitted, since it must be within the desired shaping region. As long as the input constellation points $\boldsymbol{x}$ to such a search are reasonably and carefully chosen, then the output errors $\boldsymbol{e}$ will be in a one-to-one correspondence with those input points. Thus as in Figure 8.32, decoding of $\boldsymbol{e}$ by any true ML decoder for $\tilde{\boldsymbol{x}} - \boldsymbol{c}$ in a receiver can be inverted to the corresponding input.

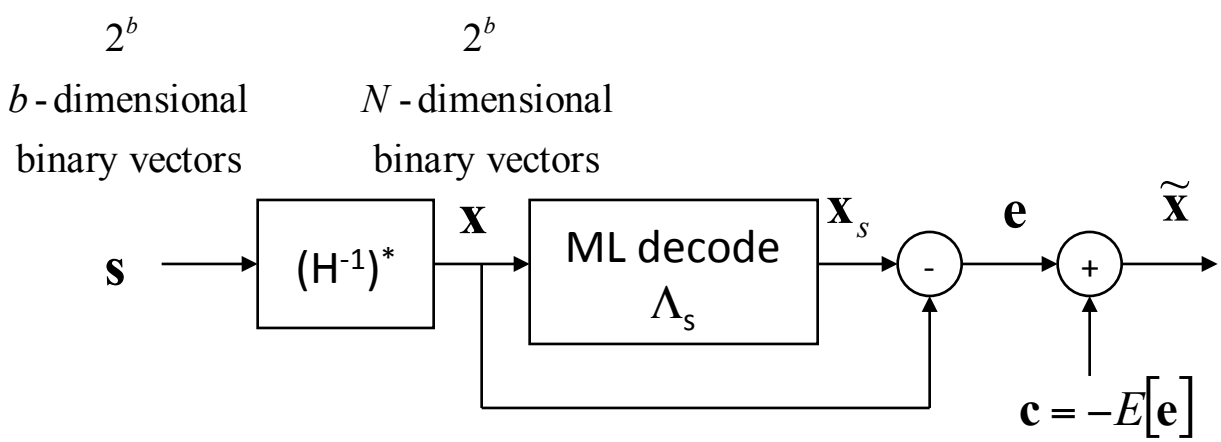

Figure 8.31: Voronoi encoder for a shaping code.

Since the Voronoi regions tesselate the complete space, a square region when $2^{2\bar{b}}$ is even and a rectangular region of sides $2^{2\lfloor \frac{\bar{b}}{2} \rfloor}$ and $2^{2\lfloor \frac{\bar{b}}{2} \rfloor + 1}$ when $2^{2\bar{b}}$ is odd, will suffice for $\boldsymbol{x}$ in two dimensions. For more general M, care must be exercised to ensure that two input vectors do not correspond to the same error vector $\boldsymbol{e}$ at the ML search output. Such care is the subject of Subsection 8.5.2.1 below. With such care, the ML decoder invertibly maps easily generated inputs $\boldsymbol{x}$ into "error" vectors for transmission in the Voronoi shaping region. Thus, the shaping gain will be that of the Voronoi shaping region, presumably good because the shaping lattice is well chosen. The volume of the shaping lattice must be at least M times the volume of the underlying code constellations, typically $V(Z^N) = 1$, making $V(\Lambda_s) = M$, where M is the number of points in the N-dimensional constellation. The number of these points may exceed 2^b in situations where a code is already in use. However, the shaping is independent of such coding. Thus, convenience of notation for the rest of this section presumes that $b = \log_2(M)$ where M is the number of points in the N-dimensional constellation. So as far as shape-encoding is concerned, M is simply the total number of points in the constellation including any earlier-introduced redundancy

for coding. The final coset addition for centering (making the mean value zero) the constellation is also shown in Figure 8.31. The added vector $\boldsymbol{c}$ should be the negative of the mean value of the points in the constellation corresponding to the values of $\boldsymbol{e}$.

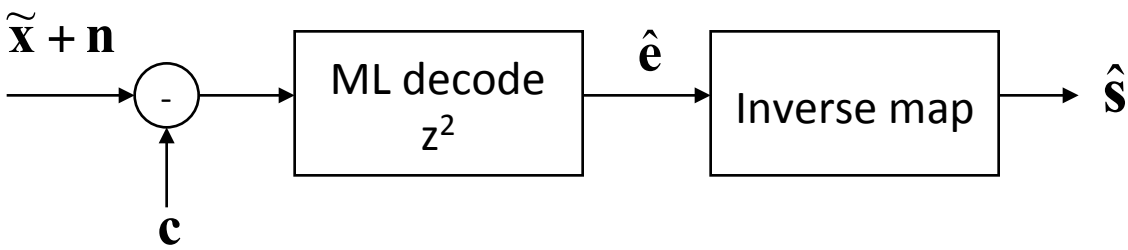

Figure 8.32: Decoding of AWGN with shaping inversion included.

EXAMPLE 8.5.2 (R_2D_2 simple shaping) Figure 8.33 illustrates the shaping of 3 bits in a rectangular pattern on the grid of Z^2 into the shape of R_2D_2. The points with squares and circles are the original 8 points $\boldsymbol{x}$ in a rectangular pattern that is simple to create. By tesselating the two-dimensional space with the diamond R_2D_2 shapes of volume 8 (with right-sided boundaries included and left-sided boundaries not included), it is clear the original points do not fit the shape. However, by taking the error of any point with respect to the center of the closest region, the points map as shown as the dark circles $\boldsymbol{e}$. Four points remain in the original positions, but the other 4 move as a result of the ML search and consequent transmission of the corresponding error vector. A point on a boundary has error vector with respect to the center of the Voronoi region that has that point on the closed boundary. There is no shaping gain in this example, but the concept of the mapping into Voronoi regions is clear.

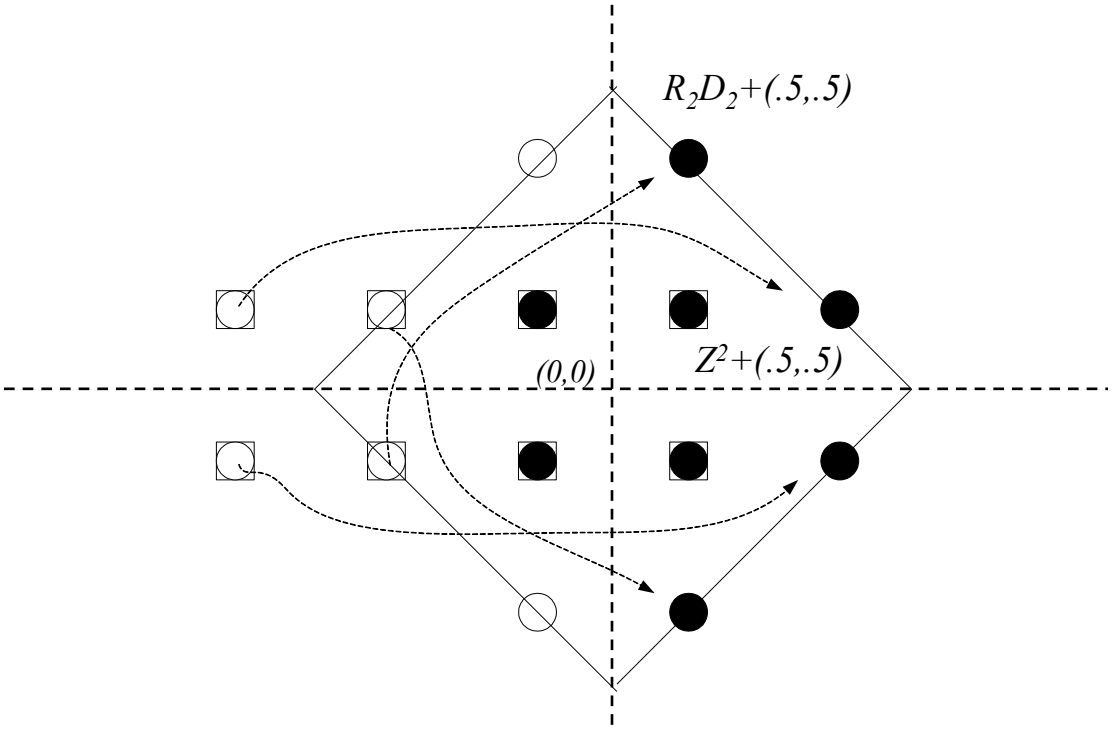

Figure 8.33: Shaping 3 input bits in Z^2 into an R_2D_2 boundary.

The 8 points in the shaped constellation are darkened. Those points have characteristics

$\boldsymbol{e}$ point	$\bar{\mathcal{E}}_{\boldsymbol{e}}(i)$	$m_{\boldsymbol{e}}(i)$
$\frac{1}{2}\,[\pm 1,\ \pm 1]$	$\frac{1}{4}$	$[0,\ 0]$
$\frac{1}{2}\,[3,\ \pm 1]$	$\frac{5}{4}$	$[\frac{3}{2},\ 0]$
$\frac{1}{2}\,[1,\ \pm 3]$	$\frac{5}{4}$	$[\frac{1}{2},\ 0]$
average	$\frac{3}{4}$	$[\frac{1}{2},\ 0]$

Since the constellation after shaping has non-zero mean, energy can be reduced by subtracting that non-zero mean from all points, so $\boldsymbol{c} = -[\frac{1}{2},\ 0]$. Then the energy is

$$\mathcal{E}_{\boldsymbol{x}} = \frac{3}{4} - \frac{1}{2}\left\{\left(\frac{1}{2}\right)^2 + 0^2\right\} = \frac{3}{4} - \frac{1}{8} = \frac{5}{8} \quad , \tag{8.126}$$

which is the same energy as an 8SQ constellation and so has no shaping advantage. The shaping gain with respect to the common reference Z^2 is also $\frac{(8-1)}{12} \cdot \frac{8}{5} = \frac{14}{15}$ or about -0.1 dB. Thus R_2D_2, while easy for illustration , is not a particularly good lattice for shaping.

8.5.2.1 Encoder Simplification for Voronoi Shaping

A key concept in Voronoi-encoding simplification, cleverly constructed by Forney, makes use of the observation in Appendix **??** that (scaled) binary lattices tesselate Z^N. Essentially every point in Z^N is in some coset of the binary shaping lattice Λ_s if $Z^N/2^m\Lambda_S/2^mZ^N$ where $m \geq 1$. In Appendix **??**, $m = 1$. In such an $m = 1$ case, the shape of Λ_s may be desirable, but its volume may not equal M. If constellations have a power of two number of points, then the factor 2^m represents the additional enlargement (scaling) necessary of the desired Voronoi shaping region to cover the M points in N dimensions. This subsection first describes encoder implementation with $m = 1$, and then proceeds to $m > 1$.

Appendix **??** notes that every point in Z_N can be constructed by some point in the partitioning (and shaping) binary lattice Λ_s plus some coset offset. A binary code with linear binary block encoder G_s and corresponding binary-block parity matrix H_s defines a binary lattice Λ_s as in Appendix **??**. The order of the partition is

$$\mid Z^N/\Lambda_s \mid = 2^b = M \quad . \tag{8.127}$$

Then, as in Appendix **??**, the distinct 2^b cosets of Λ_s whose union is Z^N are in a one-to-one relationship with the 2^b distinct syndrome vectors of the binary code (such syndrome vectors being generated by taking any of the 2^b binary b-row-vectors and postmultiplying by the parity matrix H_s^*). Since these binary-code syndrome vectors are easily generated as any or all of the 2^b binary b vectors, then they are equivalent to an easily generated input as in Figure 8.34. The transformation by the inverse-transpose parity matrix generates a set of distinct N-vector coset leaders that when added to any and all points in Λ_s generate Z_N. These coset vectors now in Z_N represent the distinct error vectors with respect to the closest points in Λ_s modulo 2, but are not necessarily the error vectors of minimum Euclidean norm. These inputs $\boldsymbol{x}$ are now assured of each mapping in a one-to-one relationship with one of the 2^b distinct syndromes/error-vectors $\boldsymbol{e}$. The ML search simply re-maps them into the Voronoi shaping region of minimum total energy.

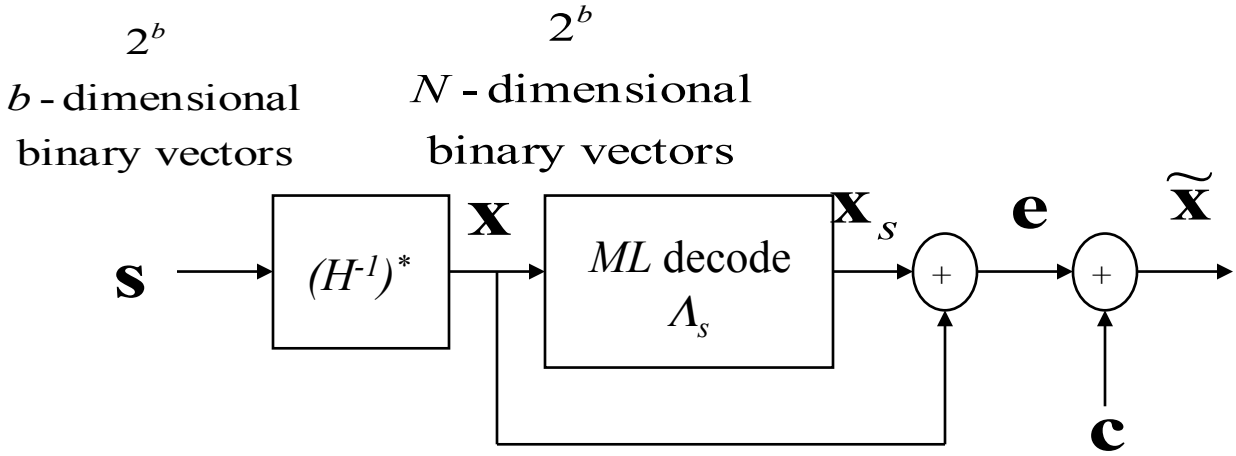

Figure 8.34: Voronoi encoder for shaping code with syndrome generation of input vectors.

The channel with any receiver decoding is eventually viewed as hard coding to one of 2^b possible N-dimensional $\boldsymbol{e}$ values. The original binary syndrome vectors (viewed as the b information bits here) is recovered in the receiver by multiplying $\hat{\boldsymbol{x}}$ by H^* after executing a modulo-2 operation. The modulo-2 operation precedes the syndrome re-generation because the vectors $\boldsymbol{e}$ are equivalent modulo 2 to the original $\boldsymbol{x}$ vectors, and then the binary arithmetic of the parity matrix can be directly applied. The decoder is shown in Figure 8.35 for $m = 1$.

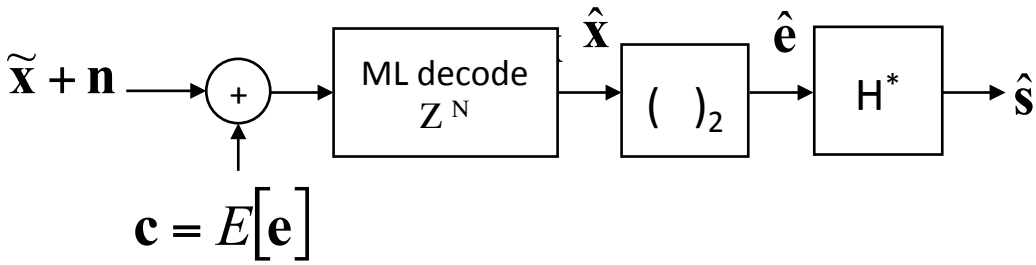

Figure 8.35: Voronoi decoder for shaping code with syndrome generation of input vectors.

The situation with $m > 1$ corresponds to $(m-1)N + k$ bits. The coset-indicator outputs in Figure 8.36 are

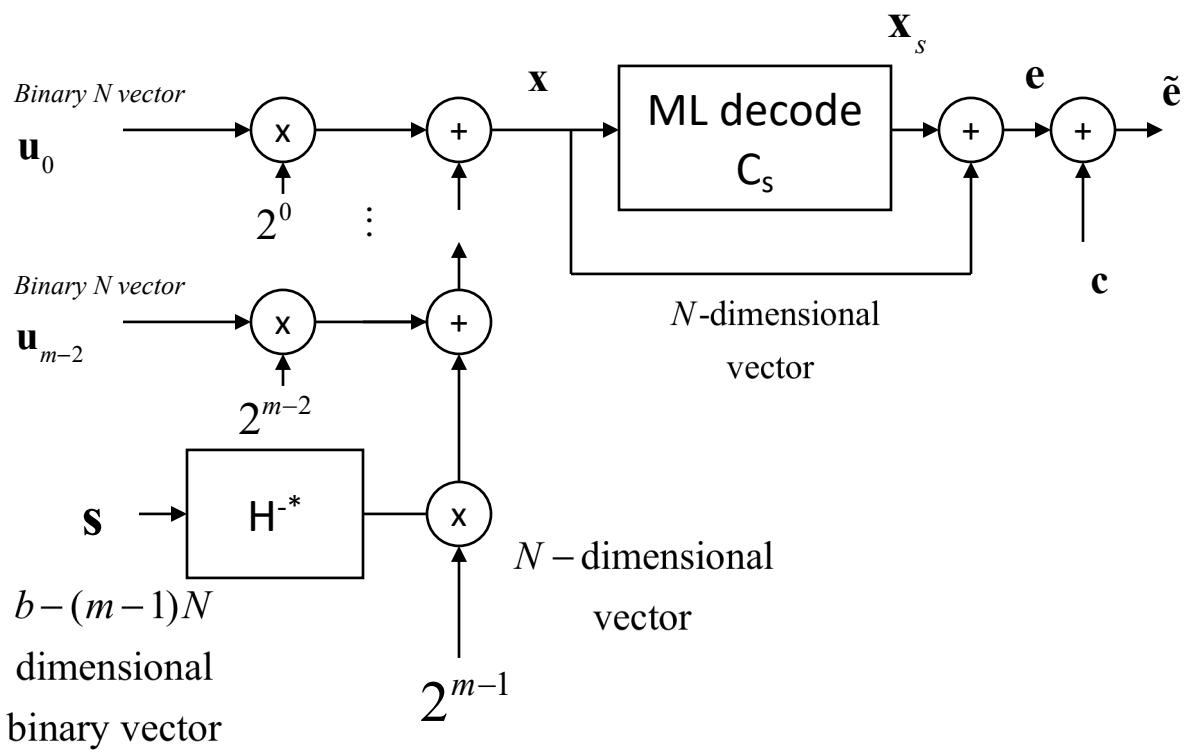

Figure 8.36: Voronoi encoder for shaping code with syndrome generation of input vectors with scaling and $m > 2$.

multiplied by 2^{m-1} and added to a binary N-vector $\boldsymbol{u}_{m-2}$ of N bits and multiplying it by 2^{m-2} and then adding it in turn to another vector of N bits $\boldsymbol{u}_{m-1}$ multiplied by 2^{m-2} ... and adding it to the last binary vector of N bits unscaled. Essentially , the scaling by decreasing powers of 2 on the integer vector $\boldsymbol{x}$'s components allows the contributions at each stage to represent the "least significant contributions" with respect to values scaled with a larger power of 2. For decoding in Figure 8.37,

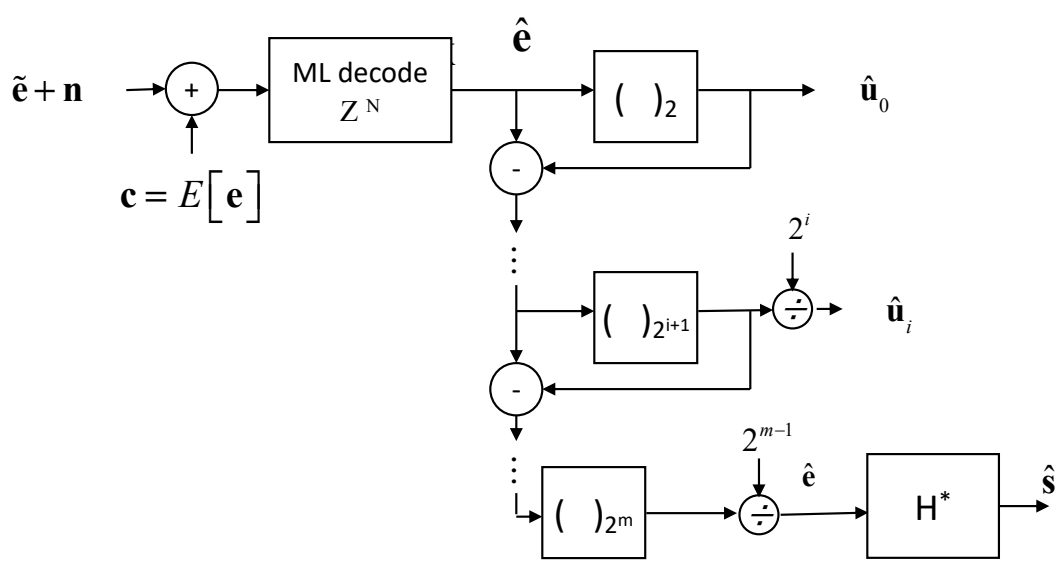

Figure 8.37: Voronoi decoder for shaping code with syndrome generation of input vectors with $m > 2$.

a modulo-2 operation recovers the upper binary N-vector $\hat{\boldsymbol{u}}_0$ from the ML decoder output integer, while after removal of those decided bits with integer subtraction, the remaining N-dimensional vector can be processed modulo 2^2, then its components divided by 2^1. Continuing through stage index i, the next remaining N-dimensional vector is produced by an operation of modulo 2^{i+1} and the components divided by $2^{i=1}$ to produce $\hat{\boldsymbol{u}}_i$, and so forth until the last remaining N-dimensional vector is modulo 2^m and devided by 2^{m-1} prior to multiplication by the parity transpose for deciding the syndrome bits of the encoder.

A drawback of the encoder is that only integer multiples (that is $m-1$) of N bits are allowed to be added to the k-dimensional syndrome vector of bits, severely limiting bit rates when N is large. If some intermediate number of bits corresponds to the desired data rate, then this number of bits will fall between two values for m. If some fractional value of $m = 1 + \frac{p}{p+q} > 1$ of N-bit blocks is desired, then p successive block-packets are used with $m-1=p$, followed by q packets with $m-1=q$. The shaping gain is assumed to be the same for both designs, although a slight deviation related to the number of points might alter the actual shaping gain. The decoder is altered correspondingly.

EXAMPLE 8.5.3 (Continuing R_2D_2 shaping example) For the earlier 8-point example, the generator matrix is $G = [1\ 1]$ and the parity matrix is also $H = [1\ 1]$. An acceptable left inverse is $H^{-*} = [1\ 0]$. However, this H is for D_2, so that R_2D_2 has $m = 2$. Thus, a single bit $\boldsymbol{s}$ times $[1\ 0]$ is scaled by $2^1 = 2$ before adding it to the 4 possible binary 2-vector values of $\boldsymbol{u}_0$ in Figure 8.38. The constellation produced is not exactly the same as in Figure 8.30, but is equivalent.

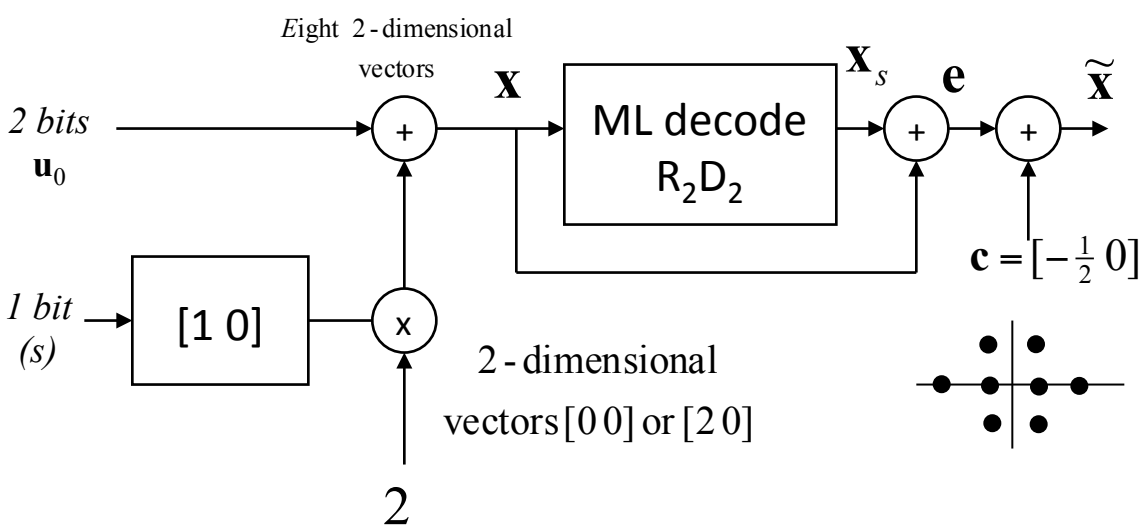

Figure 8.38: Voronoi encoder for example.

The decoder in Figure 8.39 adds back the mean $[\frac{1}{2}\ 0]$ coset offset and does ML slicing for the offset of the constellation shown. The 8 points at the output then processed modulo 2 to produce the bits u_2 and u_1. These bits are then subtracted from the ML slicing output and will produce either the points $[0\ 0]$ when $u_3 = 0$ or will produce one of the points $[-2\ 0]$ or $[0\ -2]$ when $u_3 = 1$. The encoder multiplication by 2 is inverted by first processing these two points modulo 4 (that is by modulo-2^{m+1}, and then dividing by 2. The final step is adding the two dimensions (since $H = [1\ 1]$) to produce 0+0=0 when $u_3 = 0$ and 1+0=1 when $u_3 = 1$.

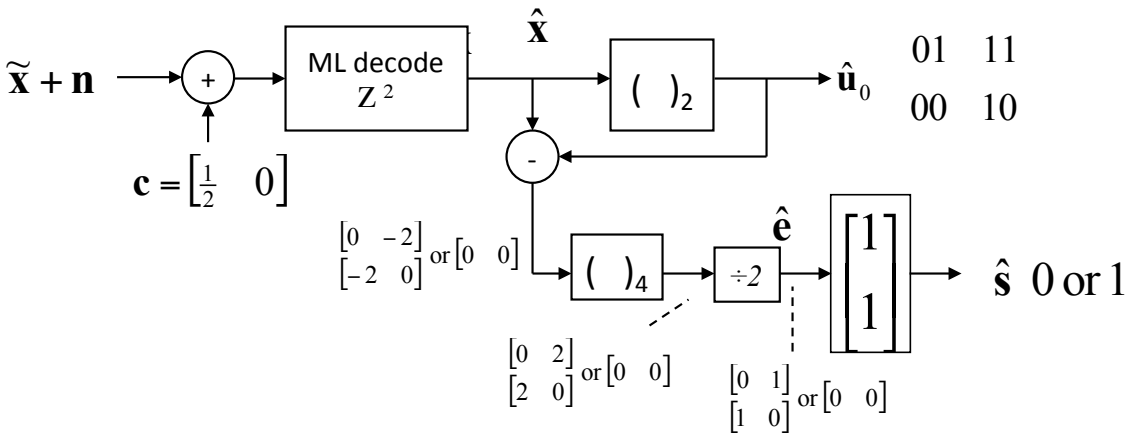

Figure 8.39: Voronoi decoder for the R_2D_2 shaping code example.

8.5.2.2 Voronoi Shaping Code Design

The shaping gain of a Voronoi shaping code is precisely computed only if b is known as

$$\gamma_s = \frac{2^{2\bar{b}}}{12\mathcal{E}_{\Lambda_s}(1 - 2^{-2\bar{b}})} \quad , \tag{8.128}$$

but for large values of b, the asymptotic gain can be listed and approximated by the continuous approximation that replaces the term $2^{2\bar{b}}/\mathcal{E}_{\Lambda_s}$ by

$$2^{2\bar{b}}/\mathcal{E}_{\Lambda_s} \approx \frac{\int_{V(\Lambda_s)} \|\boldsymbol{x}\|^2 dV}{\int_{V(\Lambda_s)} dV} \quad . \tag{8.129}$$

Table 8.23 enumerates the shaping gain for some well-known lattices.

2^ν	code rate	γ_s (dB)	$H(D)$	$H(D)^{-*}$
2	1/2	.59	$[1+D\ 1]$	$[\ 0\ 1]$
4	1/2	.97	$[1+D+D^2\ 1+D^2]$	$[D\ 1+D]$
8	1/2	1.05	$[1+D^2+D^3\ 1+D+D^3]$	$[1+D\ D]$
8	$D_8^\perp$	.47	see (8.130)	see (8.131)
8	E_8	.65	see (8.132)	see see (8.133)
16	Λ_{16}	.86	not provided	not provided
24	Λ_{24}	1.03	not provided	not provided

Table 8.23: Table of Voronoi Shaping Codes - needs update to remove the trellis shaping codes

The gains for the Barnes-Wall Lattice Λ_{16} and the Leech Lattice Λ_{24} are provided to get an idea of the increase in shaping gain with N. The ML-search for these structures may be excessive for the gain provided and trellis shaping in Subsection 8.5.3 provides an easier alternative. For the dual of the D_8 lattice

$$H_{D_8^\perp} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{8.130}$$

with transmitter inverse

$$H_{D_8^\perp}^{-*} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{bmatrix} . \tag{8.131}$$

For the implementation of the self-dual E_8 shaping

$$H_{E_8} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix} \tag{8.132}$$

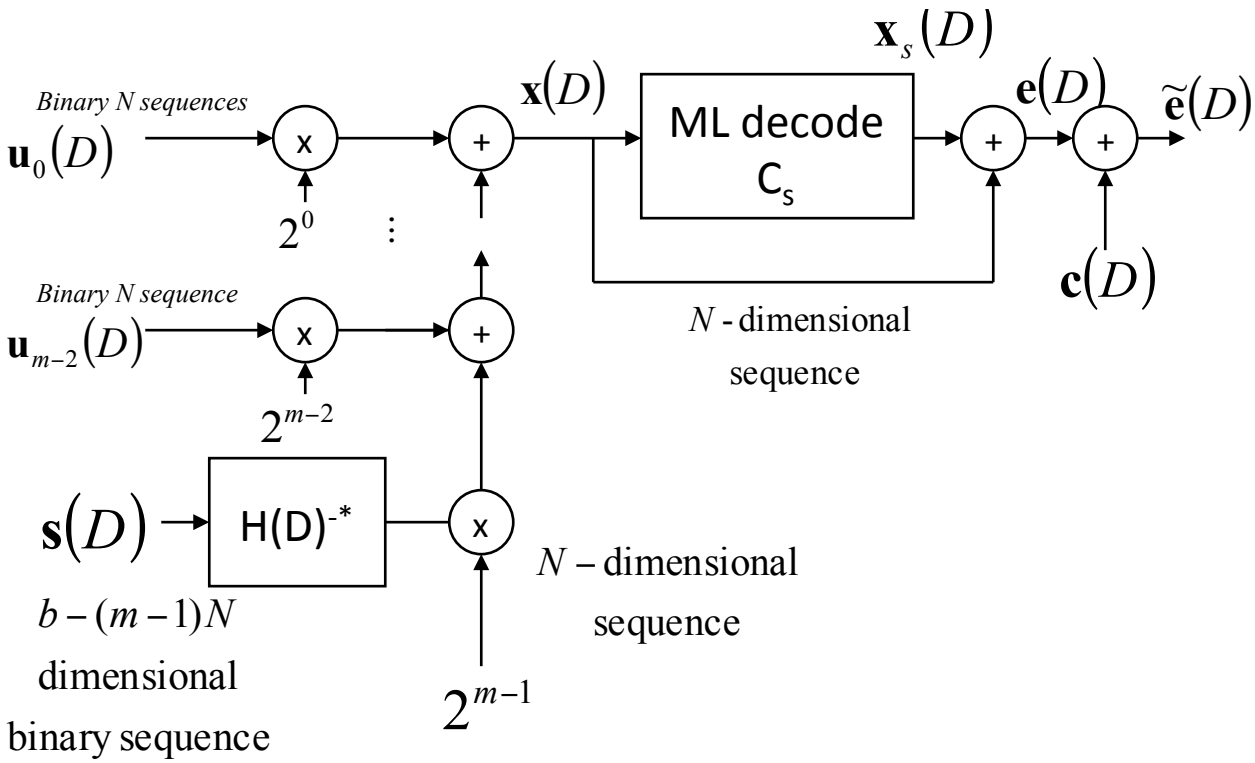

Figure 8.40: Trellis-shaping encoder for shaping code with syndrome generation of input vectors with scaling and $m \geq 2$.

and

$$H_{E_8}^{-*} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{8.133}$$

8.5.3 Trellis Shaping

Voronoi shaping extends naturally to the use of trellis codes or lattice-sequences to define a shaping region from which constellation points must be selected. This concept was introduced by Forney and called Trellis Shaping. This method can achieve shaping gains of over 1 dB with relatively low complexity. The ML search becomes a semi-infinite trellis search (Viterbi algorithm) and thus implies infinite delay. Thus, the search is terminated after an appropriate number of stages. There may thus be a small probability of sending the wrong sequence and thus of the receiver detecting correctly that wrong sequence. By using feed-back-free parity realization in the receiver for recovery of the "syndrome" sequence, any such errors with finite small non-zero probability are limited to a finite run (of course the transmit shaping code needs to be non-catastrophic also).

Calculation of the exact transmit energy basically is achieved via computer simulation in trellis shaping as it appears no known method for computing these gains exists (even for infinite number of points in the constellation). Figure 8.40 shows the encoder diagram, which works for $N \geq 2$ and $m \geq 2$ (the cases of smaller numbers of bits and dimensions is considered later in this subsection). When $N = 2$, then odd integer numbers of bits can be handled directly and an even integer number of bits requires the upper most encoding path choose only the sequences (0,0) or (1,1) instead of all 4 binary 2-tuples. For $N > 2$, then the granularity to integer bits is achieved in a similar manner as Voronoi shaping codes (as long as m is large enough). As with Voronoi codes, duals of good codes (lattices) tend to be best because the redundancy introduced is least (that is, a potentially desired event is H^{-*}

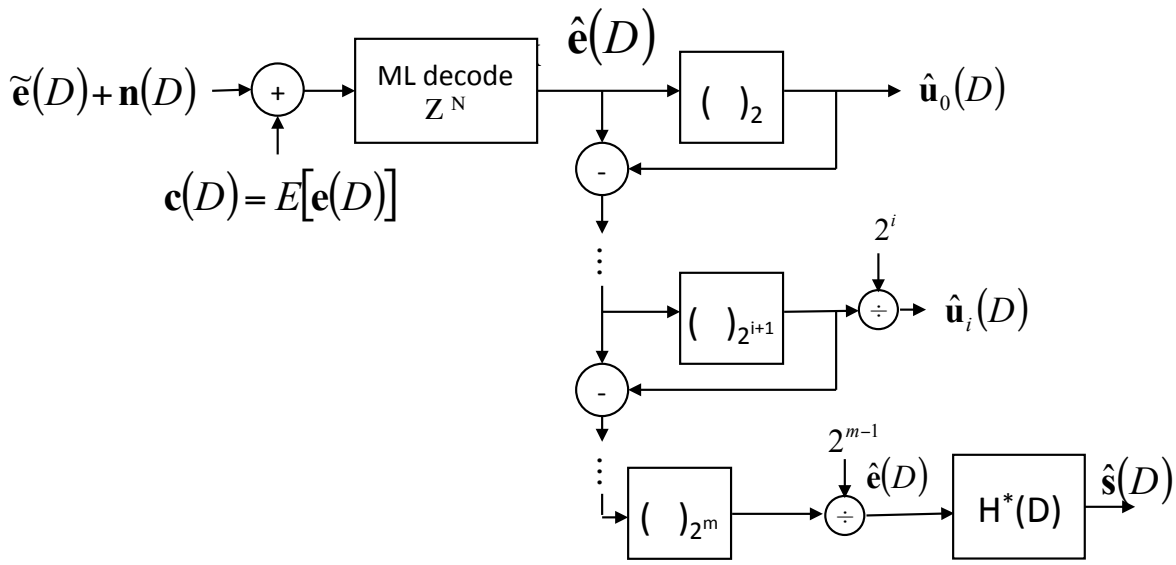

Figure 8.41: Trellis-shaping decoder for shaping code with syndrome generation of input vectors with scaling and $m \geq 2$.

is $(b-[m-1]N) \times (b-[m-1]N+1)$ so that only a doubling in constellation size is necessary; however, E_8 is a self dual and quadruples constellation size and has good shaping gain.)

Figure 8.41 illustrates the corresponding receiver. Again, the parallels with the Voronoi situation are clear. The sequences are processed N dimensions at a time. The $H(D)$ matrix should be chosen to be feed-back free to avoid error propagation.

Table 8.24 lists the two-dimensional trellis-shaping codes and their simulated gains (along with the delay in the ML search and b). These gains are all for 7 bits/symbol and a 256-point square constellation for $\boldsymbol{x}$ in Figure 8.40. The gains do not increase substantially for higher numbers of bits per dimension, but will reduce for lower numbers of bits per dimension. Feed-back-free parity matrices have been provided, but there are many generalized left inverses. In the last two codes, inverses with feedback were used (because they're easy to determine). Feedback in the transmitter is acceptable (but not in the receiver). However, the zeros in an inverse with 0 feedback reduces the redundancy in the two-dimensional constellation (fewer points can be selected in any two-dimensional symbol because of the zeros).

The drawback of the encoder is that only multiples (that is $m-1$) of $N=2$ bits is somewhat less restrictive with trellis-shaping codes than with Voronoi codes. However, the same method for fraction m values as in the Voronoi section can be used. Four and eight dimensional trellis shaping codes were investigated by Forney, but the shaping gains are less (although constellation expansion is less also). One-dimensional trellis shaping is capable of higher gains, but typically quadruples the number of points in one dimension, which may be unacceptable from an implementation standpoint. The additional gain is about .2 dB, but requires more states (and ML decoders than run at double the speed). Thus, the two-dimensional shaping codes seem preferred when trellis shaping may be used.

2^ν	γ_s (dB)	$H(D)$	$H(D)^{-*}$	delay
2	0.59	$[1+D \;\; 1]$	$[1 \;\; D]$	8
4	0.97	$[1+D^2 \;\; 1+D+D^2]$	$[D \;\; 1+D]$	26
8	1.05	$\begin{bmatrix} 1+D^2+D^3 & 1+D+D^3 \end{bmatrix}$	$\begin{bmatrix} 1+D & D \end{bmatrix}$	34
8	1.06	$\begin{bmatrix} 1+D^3 & 0 & D \\ 0 & 1+D^3 & D^2 \end{bmatrix}$	$\begin{bmatrix} \frac{1}{1+D^3} & 0 & 0 \\ 0 & \frac{1}{1+D^3} & 0 \end{bmatrix}$	42
16	1.14	$\begin{bmatrix} 1+D+D^4 & 0 & D+D^2+D^3 \\ 0 & 1+D+D^4 & D^2 \end{bmatrix}$	$\begin{bmatrix} \frac{1}{1+D+D^4} & 0 & 0 \\ 0 & \frac{1}{1+D+D^4} & 0 \end{bmatrix}$	100?

Table 8.24: Simulated gains and parity matrices for 2-dimensional trellis shaping at $b = 7$.

8.6 Hard-Soft Concatenation and Retransmission

A final step to improve coding and drive $P_e \to 0$ is Figure 8.42's hard-soft concatenation, which is simlar to Section 8.3.1's serial interleaving concatenation, except the outer and inner decoders typically do not exchange information.

Figure 8.42: **Serial Concatenation**

Instead the inner decoder makes a hard decision upon, and outputs, subsymbols to the outer decoder. These output subsymbols in some Galois Field $GF(2^m)$, typically not bits/binary - more typically bytes so $m = 8$, so the outer code's design treats the inner system as a SDMC. The outer code has its own redundancy and corrects errors. This outer code can slightly increase overall-system coding gain by roughly 0.5-1dB if the redundancy ρ is small with Section 8.6.1's deterministic interleaving. The inner system is presumably well designed and operating within 1-2 dB of capacity at a reasonable bit-error rate of $\bar{P}_b < 10^{-4}$. The outer code's design simply reduces furhter P_e. This outer system essentially completes reducing $\Gamma \to 0$ and thus approaches $\bar{b} = \bar{\mathcal{C}}$.

Subsection 8.6.1's deterministic interleaving is crucial to the further improvement. Inner-code decoding failures create long erred codewords, so they contain subsymbol "error bursts." The interleaving distributes these bursts over many outer-code codewords, effectively scaling the minimum distance and more generally the distance spectrum. Typically the outer codes use Section 8.4's block cyclic codes and can both detect errors' presence as well as correct them. Section 8.6.2's design methodology uses interleave depth, estimates consequent delay, and attempts to ensure the outer code reduces $\bar{P}_e$ to an acceptable level. Section 8.6.3 progresses to the inevitable potential of some remaining errors and the subject of retransmission codes that detect these errors, feedback a detected error's associated block-codeword index, and the entire code or a portion of it is then resent. Several potential resendings will further increase delay but eventually the potential of residual remaining error becomes acceptable[21]

8.6.1 Deterministic Interleaving

As in Section 8.3.1, **interleaving** reversibly reorders $L \geq N$ transmitted subsymbols. A receiver de-interleaver reorders the subsymbols. Interleaving disperses error bursts that may occur because the inner code errs in decoding. Nonstationary noise may also cause error bursts. Erred subsymbols caused by inner-code detection errors will typically span the entire incorrect codeword, leading to a significant-length erred-subsymbol burst. If different bursts occur at intervals long with respect L, then the receiver's de-interleaver distributes them more evenly over time (or more generally dimensions). The bursts' redistribution effectively enables realistic modeling for the inner-code-and-channel as memoryless, i.e., modeled by a BSC, DMC, or other channel for which successive subsymbol outputs are independent.

[21] No practical system is ever guaranteed never to make an error.

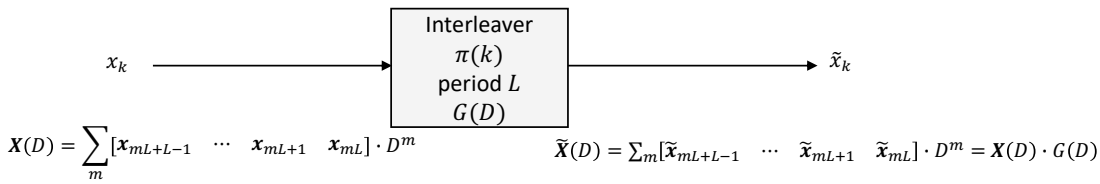

Figure 8.43: **Basic Deterministic Periodic Interleaver.**

Periodic Interleaver: Figure 8.43 generically depicts the deterministic periodic interleaver as accepting subsymbols indexed in time by $k = mL + \ell$, $\ell = 0, ..., L-1$ or in block/packet in m, where L subsymbols occur within each packet and interleaver satisfies $\pi(m \cdot L + \ell) = \pi(m \cdot L)\ \forall\ m \in \mathbb{Z},\ \ell \in \mathbb{Z}$.

Definition 8.6.1 [Interleaver Depth] *The* **depth** $\mathcal{J}$ *of an interleaver is the minimum separation in subsymbol periods at the interleaver output between any two subsymbols that are adjacent at the interleaver input.*

Interleaver depth might best be called "de-interleaver" depth because the de-interlaver disperses channel/inner-decoder error bursts. However, the interleaver and de-interleaver will have the same dispersive properties, which follows from the fact that any interleaver can be written as a permutation matrix containing only one 1 in each row/column, and such matrices have inverse equal to their transpose, said transpose could simply relabel its indices and have the same properties as the original permutaiton matrix. Interleaver depth has significant implication for error bursts entering a receiver's de-interleaver. If a burst of errors has duration less than the depth, then two subsymbols affected by the burst cannot be adjacent after de-interleaving.

Interleaver peiod: Formally, interleaver period is also important:

Definition 8.6.2 [Interleaver Period] *The* **period** L *of an interleaver is the shortest time interval for which the re-ordering algorithm used by the interleaver repeats.*

Interleaver design chooses its period L and depth $\mathcal{J}$. The interleaver repeatedly acts with the same algorithm upon successive blocks of L subsymbols. The desired depth $\mathcal{J}$ and outer-codeword length N_{out} often guide or determine L choice.

Theorem 8.6.1 [Interleaver Minimum Distance Magnification] *For inner channels with an outer hard-decision codeword length N_{out} and interleaver depth $\mathcal{J}$, and for which only one error burst occurs within $(\mathcal{J}-1)\cdot(N_{out}-1)$ subsymbols, the outer code's free distance in subsymbols increases to $\mathcal{J} \cdot d_{free,out}$ for these inner-channel error bursts.*

Proof: The de-interleaver disperses adjacent de-interleaver output subsymbols within an de-interleaver-input burst by at least $\mathcal{J}-1$ subsymbols. A group of $\mathcal{J}$ successive outer-code codewords contain $N_{out} \cdot \mathcal{J}$ subsymbols that (pre-de-interleaver) have $L_{burst} \leq \mathcal{J}$ successive erred subsymbols. This error-subsymbol burst can have a most $N_{out}/\mathcal{J}$ errored subsymbols after de-interleaving because any codeword having more than $N_{out}/\mathcal{J}$

could not satisfy the separation by $\mathcal{J}$ that the de-interleaver imposes within that codeword. Thus the decoder for this codeword need correct $N_{out}/\mathcal{J}$ erred subsymbols of the original L_{burst} errored subsymbols. Therefore $\mathcal{J}$ times more errors can be corrected than is possible without interleaving. The inter-burst spacing must be in general longer than this minimum delay $(\mathcal{J}-1)\cdot(N_{out}-1)$ to prevent different error bursts from placing erred subsymbols in the same group of $N_{out}\cdot\mathcal{J}$ codewords.**QED.**

Description via Generators: Interleaving with a single code does not improve performance for stationary additive white Gaussian noise; it improves channels that exhibit error bursts. Such an error burst occurs when an "inner" decoder incorrectly decodes. Any interleaver relates its input $\boldsymbol{x}_k$ to its output $\tilde{\boldsymbol{x}}_k$ through

$$\tilde{\boldsymbol{x}}_k = \boldsymbol{x}_{\pi(k)} \quad , \tag{8.134}$$

where $\pi(k)$ is a reversible function that describes the mapping of interleaver output indices to interleaver input indices. Deterministic interleavers are perioidic;

$$\pi(k) - L = \pi(k-L) \quad . \tag{8.135}$$

The deterministic interleaver's depth $\mathcal{J}$ follows more precisely using the function π as

$$\mathcal{J} = \min_{k=0,\ldots,L-1} \mid \pi^{-1}(k) - \pi^{-1}(k+1) \mid \quad . \tag{8.136}$$

Interleaver description borrows traditional coding's use of a delay variable D, but now as corresponding to one interleaver period such that

$$D = D_{ss}^{L} \quad . \tag{8.137}$$

Interleaver Model: Figure 8.43 uses the notation $\boldsymbol{X}(D)$ to denote a sequence of interleaver-input subsymbols, where each sequence element spans one interleaver period, with the index $k = m\cdot L+\ell$, $\ell = 0,\ldots,L-1$, yielding to an interleaver block index m,

$$\boldsymbol{X}_m = \left[\boldsymbol{x}_{m\cdot L+(L-1)}\ \boldsymbol{x}_{m\cdot L+(L-2)}\ \ldots\ \boldsymbol{x}_{m\cdot L}\right] \quad , \tag{8.138}$$

where m thus denotes a time index corresponding to a specific block of L successive interleaver-input symbols, and

$$\boldsymbol{X}(D) = \sum_m \boldsymbol{X}_m \cdot D^m \quad . \tag{8.139}$$

Similarly the interleaver output $\tilde{\boldsymbol{X}}(D)$ can be considered an L-dimensional vector sequence of interleaver outputs. Then, interleaving can be modeled as a "rate 1" convolutional/block code over the symbol alphabet with generator and input/output relation

$$\tilde{\boldsymbol{X}}(D) = \boldsymbol{X}(D)\cdot G(D) \quad , \tag{8.140}$$

where $G(D)$ is an $L\times L$ nonsingular generator matrix with the following restrictions:

1. one and only 1 entry in each row/column can be nonzero, and
2. nonzero entries are of the form D^l where l is an integer.

The de-interleaver has generator $G^{-1}(D)$, so that $\boldsymbol{X}(D) = \tilde{\boldsymbol{X}}(D)\cdot G^{-1}(D)$. Further, a **causal interleaver** has the property that all nonzero elements have D^l with $l \geq 0$, and elements above the diagonal must have $l \geq 1$ – the de-interleaver for a causal interleaver is necessarily noncausal, and thus must be instead realized with delay because interleaving necessarily introduces delay from interleaver input to de-interleaver output.

The equivalent relationships in terms of subsymbol delay are similarly

$$\tilde{\boldsymbol{X}}(D_{ss}) = \boldsymbol{X}(D_{ss}) \cdot G(D_{ss}) \quad , \tag{8.141}$$

but the input and output vectors have each entry defined in terms of the subsymbol-period D-transform as

$$x_i(D_{ss}) = \sum_{k=0}^{\infty} x_{i,k} \cdot D_{ss}^k \quad . \tag{8.142}$$

The entries in the vector $\boldsymbol{X}(D_{ss})$ are **not** simply found by substituting $D = D_{ss}^L$ into the entries in the vector $\boldsymbol{X}(D)$. In fact,

$$\boldsymbol{X}(D_{ss}) = \left[D_{ss}^{L-1} \cdot x_{L-1}(D) \big|_{D=D_{ss}^L} \;\; D_{ss}^{L-2} \cdot x_{L-2}(D) \big|_{D=D_{ss}^L} \;\; ... \; x_0(D) \big|_{D=D_{ss}^L} \right] \quad . \tag{8.143}$$

The same relation holds for the interleaver output vector sequence $\tilde{\boldsymbol{X}}(D)$. The subsymbol-spaced generator follows easily by replacing any nonzero power of D, say D^δ in $G(D)$ by substituting

$$D^\delta \rightarrow D_{ss}^{L \cdot \delta + (\text{column number} - \text{row number})_{mod L}}$$

and circularly shifting the entry within the row to the left by (row number − column number) positions.

> **Rule 8.6.1** **[Generator Conversion for Interleavers]** Any non-zero entry in $G(D_{ss})$ be written as a constant coefficient times D_{ss}^r. A new column index i' is formed by $i' = (r+i)_{\text{mod } L}$ where i is the column index of the original non-zero entry (counting from right to left, starting with 0). Further let $r' = \lfloor \frac{r+i}{L} \rfloor$, then place $D^{r'}$ in the same row in the position of column i'. One can think of this as circularly shifting by $r+i$ positions and increasing the power of D for every factor of L in the exponent of D_{ss}^{r+i}.

(An example use of Rule 8.6.1 will occur shortly.)

It is often easier to avoid the rule and simply directly write $G(D)$ based on a description of the interleaving rule (and/or to do the same for $G(D_{ss})$. A few examples will help illustrate the concepts.

EXAMPLE 8.6.1 **[Simple Block Permutation]** A simple block-interleaver example has ordering function

$$\pi(k) = \begin{cases} k+1 & \text{if } k = 0 \text{ mod } L = 3 \\ k-1 & \text{if } k = 1 \text{ mod } L = 3 \\ k & \text{if } k = 2 \text{ mod } L = 3 \end{cases} \tag{8.144}$$

or in tabular form:

$\pi(k)$:	-1	1	0	2	4	3	5
k:	-1	0	1	2	3	4	5

,

which has inverse de-interleaver:

$k' = \pi(k)$:	-1	0	1	2	3	4	5
$\pi^{-1}(k') = k$:	-1	1	0	2	4	3	5

Since $\pi(k+3) = \pi(k)$, then $L = 3$. The depth is $\mathcal{J} = 1$, and thus this simple interleaver illustrates functionality only and is not very useful.

The corresponding generator is:

$$G(D) = G(D_{ss}) = G(0) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{8.145}$$

with de-interleaving inverse

$$G^{-1}(D) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad . \tag{8.146}$$

Rule 8.6.1's two deterministic interleaver descriptions, $G(D)$ and $G(D_{ss})$, are equivalent if $G(D) = G(0)$, i.e., a block interleaver. Subsection 8.6.1.1 studies such block interleavers.

EXAMPLE 8.6.2 [Simple Triangular Interleaving] A second interleaver example is

$$\pi(k) = \begin{cases} k & \text{if } k = 0 \bmod L \\ k-3 & \text{if } k = 1 \bmod L \\ k-6 & \text{if } k = 2 \bmod L \end{cases} \tag{8.147}$$

or in tabular form:

$\pi(k)$:	-7	0	-2	-4	3	1	-1	6	4	2
k:	-1	0	1	2	3	4	5	6	7	8

with inverse:

$\pi^{-1}(k)$:	-1	0	1	2	3	4	5	6	7	8
k:	5	0	4	8	3	7	11	6	10	14

Since $\pi(k+3) = \pi(k)$, then $L = 3$. This interleaver's depth follows directly from (8.147) as $\mathcal{J} = 4$ subsymbols because any $k = q \cdot 3 + r$ with $r = (k)_3$ must differ from $k-3 = (q-1) \cdot 3 + r$ by at least $L = 3$ positions, so then adjacent $k+1$ or $k-1$ will have r increased by 1, leaving $\mathcal{J} = 4 = L - 1$.

This interleaver's generator is (recall a delay, D, corresponds to a delay of one period of $L = 3$ subsymbols)

$$G(D) = \begin{bmatrix} D^2 & 0 & 0 \\ 0 & D^1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ and } G(D_{ss}) = \begin{bmatrix} D_{ss}^6 & 0 & 0 \\ 0 & D_{ss}^3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{8.148}$$

This straightforward diagonal interleaver is clearly a one-to-one (nonsingular) generator and has inverse

$$G^{-1}(D) = \begin{bmatrix} D^{-2} & 0 & 0 \\ 0 & D^{-1} & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ and } G(D_{ss}) = \begin{bmatrix} D_{ss}^{-6} & 0 & 0 \\ 0 & D_{ss}^{-3} & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad . \tag{8.149}$$

This block-level inverse's straightforward causal implementation requires a delay of 2 block periods or 6 subsymbol periods; however Subsection 8.6.1.2's triangular intereleavers can halve this delay.

The modulo-L operation continues with the next example.

EXAMPLE 8.6.3 [A depth-3 Interleaver] An interleaver with period $L = 5$ and depth $\mathcal{J} = 3$ has generator most easily specified with D_{ss}, essentially with $k \to k - 2 \cdot i$ for $(k)_L = i = 0, ..., L-1$ as

$$G(D_{ss}) = \begin{bmatrix} D_{ss}^8 & 0 & 0 & 0 & 0 \\ 0 & D_{ss}^6 & 0 & 0 & 0 \\ 0 & 0 & D_{ss}^4 & 0 & 0 \\ 0 & 0 & 0 & D_{ss}^2 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{8.150}$$

and could be described as delaying each symbol within a period by its index times $(\mathcal{J}-1)$ symbol periods. The inverse is also easily described. Using notation with D corresponding to a block period, then rule 8.6.1 provides[22]

$$G(D) = \begin{bmatrix} 0 & 0 & D^2 & 0 & 0 \\ D & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & D & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} . \tag{8.151}$$

The second description often complicates easy insight into the implementation and the inverse. The interpretation that each symbol is delayed by its index times $\mathcal{J}-1$ symbol periods is not nearly as evident in $G(D)$, although true if under intense examination. This simplified insight illustrates the value of the subsymbol-spaced interpretation, and the convenient $D_{ss}^{(\mathcal{J}-1)i}$ diagonal in (8.150)

8.6.1.1 Block Interleaving Implementation

Deterministic block interleaviers again have $G(D) = G(D_{ss}) = G(0) = G$, such as this $L = 12$ interleaver:

$$G(D) = G = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} . \tag{8.152}$$

The de-interleaver is trivially $G^{-1} = G^*$. Figure 8.44 illustrates a straightforward implementation common block interleaver as write columns (rows) and then read rows (columns). The period is $L = 12$ and $N_{out} = 3$, while the depth is $\mathcal{J} = 4$. Outer encoder's subsymbols arrive serially in codewords that the interleaver writes by column into a read buffer. When the memory is full, the interleaver transfers the 12 subsymbols into a write butter that outputs by row into the channel. The receiver's de-interleaver accepts the channel-output subsymbol rows over one period and then transfers these 12 subsymbols to a read buffer that the de-interleaver outputs. The original subsymbols have delay 24 at the de-interleaver output. Also, 48 memory cells are used. While straightforward in description, Figure 8.44 has twice the delay and memory necessary for the block interleaver. Figure 8.45 illustrates a better, but more difficult to describe, implementation. Figure 8.45's generator matrix illustrates in red color entries that are noncausal (or need delay). Six of these entries (including one that is just barely cause at the upper left) are "above diagonal" while the other six are below.

[22] For instance D_{ss}^8 is in column 4 from right starting at 0, so $(5+8)_{L=5} = 2$, $\lfloor \frac{5+8}{5} \rfloor = 2$, and thus D^2 appears in column 2 from the right, starting with 0; similarly D_{ss}^6 is in column 3 from right starting at 0, so $(3+6)_{L=5} = 4$, $\lfloor \frac{3+6}{5} \rfloor = 1$, and thus D^1 appears in column 4 from the right,

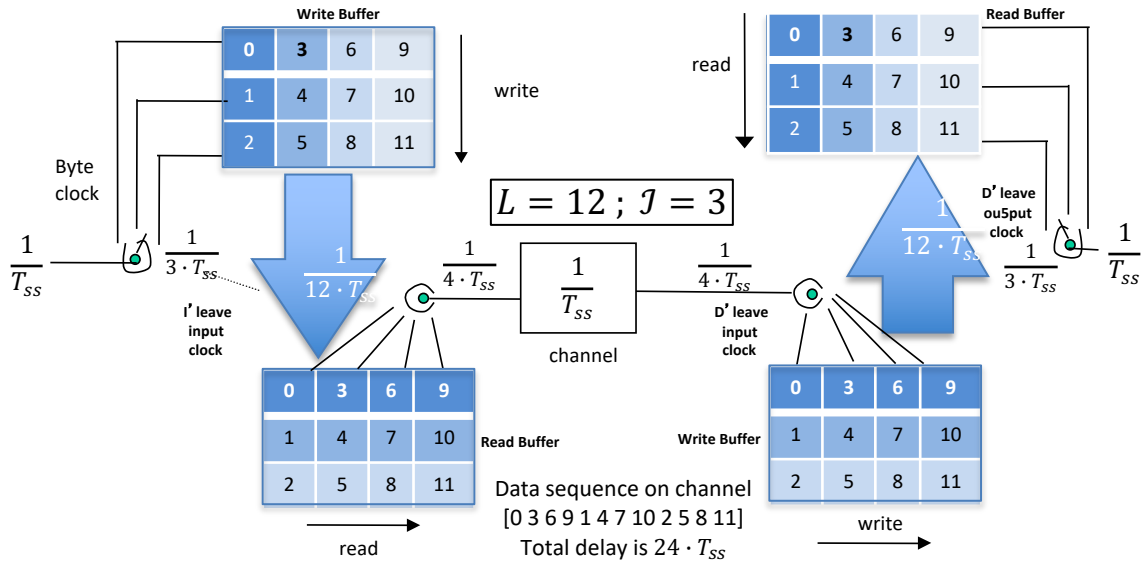

Figure 8.44: **Block interleaving and de-interleaving with $L = 12$ and $\mathcal{J} = 4$.**

A_0		B_0	
0	3	6	9
1	4	7	10
2	5	8	11

write order 0,1,2,3,4,5

write order 6,7,8,9,10,11

6 Memory Cells Needed

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

m	read	write	cell
0	---	----	pass
1	Past 3	Currrent 1	A
2	Past 6	Current 2	B
3	Past 9	Current 3	C
4	Current 1	Current 4	A
5	Current 4	Current 5	A
6	Past 7	Current 6	D
7	Past 10	Current 7	E
8	Current 3	Current 8	C
9	Current 5	Current 9	A
10	Current 8	Current 10	C
11	Past 11	Current 11	F

Figure 8.45: **Lower-Complexity Block interleaving and de-interleaving with $L = 12$ and $\mathcal{J} = 3$.**

Figure 8.45 illustrates the use of 6 memory cells[23] with labels A, B, C, D, E, and F. When a cell releases a subsymbol into the channel, it is immediately available for use. This leads to an average delay of 6 subsymbols in each of transmitter (and the corresponding de-interleaver that is not shown), for a total of 12 subsymbol periods. Also the memory reduces to 12 cells total, so 1/4 the amount in Figure 8.44's row/column block interleaver. This is still not the most efficient implementation for $N_{out} = 3$ and $\mathcal{J} = 3$, which will gain another factor of 2 reduction yet with the triangular interleavers to come. It is

[23] A designer might try to use only 5 memory cells by passing 0 and 11 immediately, but this can lead to adjacent entries not being spaced by the intended depth of 4 subsymbol periods.

also possible to reverse the N_{out} and $\mathcal{J}$ to be 4 and 3 respectively, keeping basically the same structure with the tables becoming 4×3.

The block de-interleaver redistributes a burst of B erred subsymbols distributes roughly evenly over $\mathcal{J}$ codewords. If this is the only burst within the total L received subsymbols, as in Theorem 8.6.1, the subsequent outer-code decoder can correct approximately $\mathcal{J}$ times more erred subsymbols. Larger depth $\mathcal{J}$ means more memory and more delay, but greater outer-code burst-corrective power as long as a second burst does not occur within the same $N_{outer} = K \cdot \mathcal{J}$ subsymbols. The minimum distance of the code is thus essentially multiplied by $\mathcal{J}$ as long as errors are initially in bursts that do not occur very often. Thus interleaving easily improves the performance of concatenation.

For instance, if codewords of 100 subsymbols occur in the inner code with error rate $P_e = 10^{-4}$, then there are on average roughly 10,000 codeword blocks until another error burst occurs. Each of these errors would correspond to worst-case every single subsymbol being incorrect. Thus as long as the depth satisfies $\mathcal{J} < 10000$, the burst would not in general overlap and the full depth distribution of errors to different codewords, increasingly corrective ability, would apply.

8.6.1.2 Convolutional Interleaving

Examples 8.6.2 and 8.6.3 were convolutional interleavers. Generally, convolutional interleavers have $G(D) \neq G(0)$, meaning there is at least one delay element. While more complex in concept, convolutional interleavers can allow a reduction in delay and in memory required for depth-$\mathcal{J}$ implementation.

Triangular Interleavers: Coding theorists have often reserved the name **triangular interleaver** for the special case of a convolutional interleaver where $G(D)$ is a diagonal matrix of increasing or decreasing powers in D proceeding down the diagonal. Example 8.6.2 illustrates such a triangular interleaver. The reason for the names "multiplexed" or "triangular" interleaver follow from Figure 8.46, which illustrates the 3×3 implementation.

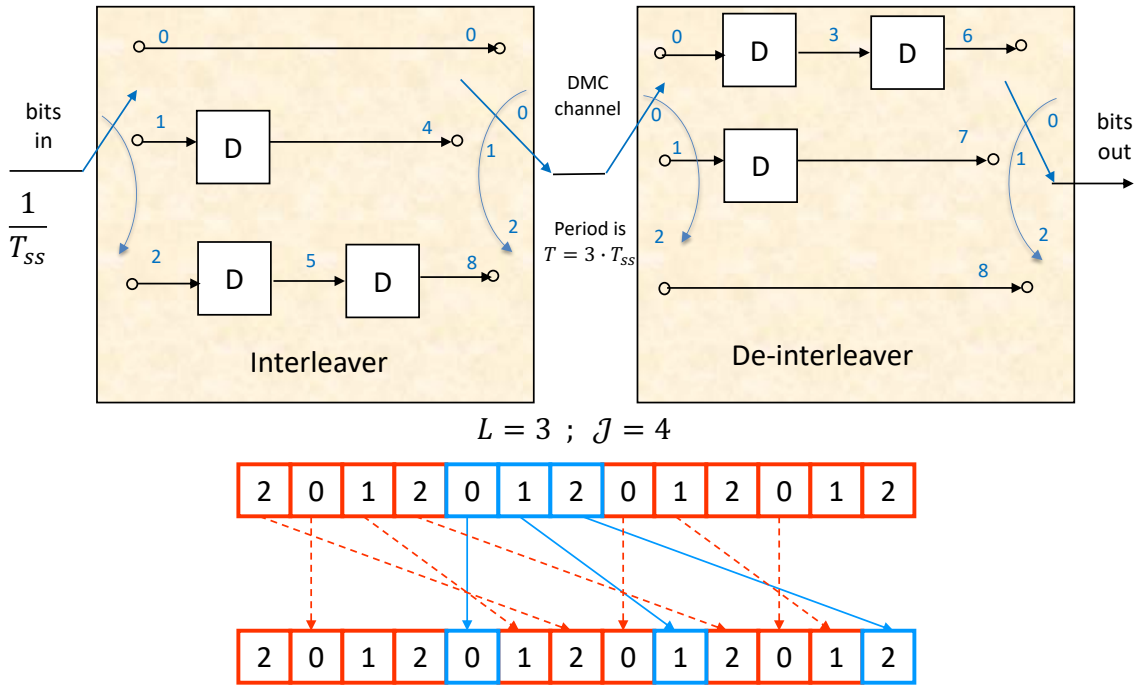

Figure 8.46: **Triangular interleaver illustrated for $L = 3 = \mathcal{J} - 1$.** (Figures 8.44 and 8.45 are equivalent block interleavers with the same depth $\mathcal{J} = 4$ and $N_{out} = 3$.)

The delay elements (or the memory) are organized in a triangular structure in both the interleaver and the de-interleaver. Subymbols enter the triangular interleaver from the left and with successive subsymbols cyclically allocated to each of the 3 possible paths through the interleaver in periodic succession. The input switch and output switch for the interleaver are synchronized so that when in the upper position, the symbol simply passes immediately to the output, but when in the second (middle) position the symbol is stored for release to the output the next time that the switch is in this same position. Finally, the bottom row subsymbols undergo two interleaver periods of delay before reaching the interleaver output switch. The deinterleaver operates in analogous fashion, except that the symbol that was not delayed at the transmitter is now delayed by two periods in the receiver, while the middle symbol that was delayed one period in transmitter sees one additional period of delay in receiver, and the symbol that was delayed twice in transmitter is not delayed at all at receiver. Clearly all subsymbols then undergo two interleaver periods of delay, somehow split between transmitter and receiver. Any subsymbols in the same block of 3 on the input have at least 3 subsymbols from other blocks of inputs in between as illustrated in Figure 8.46. The depth is thus $\mathcal{J} = 4$. However, the total delay is 2 codewords or 6 subsymbols, 1/2 the delay Figure 8.45's best block-interleaver implementation. Further the total memory is 6 cells, 3 each in interleaver and de-interleaver, for a total of 6 total, which is also 1/2 the best block-interleaver implementation.

Classical Triangular Interleavers: A form of convolutional interleaver is the classical triangular interleaver, the depth is restricted to be equal to $\mathcal{J} = L + 1$ and processes input blocks of period L subsymbols. Then, L subsymbols within the mth symbol are numbered from $\boldsymbol{x}_{0,m}$, ..., $\boldsymbol{x}_{\ell,m}$,..., $\boldsymbol{x}_{L-1,m}$ and the triangular interleaver delays each subsymbol ℓ, m within the symbol block by $\ell \cdot L = \ell \cdot (\mathcal{J} - 1)$

subsymbol periods. The generator is thus

$$G(D) = \begin{bmatrix} D^{L-1} & 0 & 0 & 0 \\ 0 & \ddots & 0 & 0 \\ 0 & 0 & D & 0 \\ 0 & 0 & ... & 1 \end{bmatrix} . \tag{8.153}$$

Only the diagonal is nonzero with increasing power of D (where again $D = D_{ss}^{L}$ corresponds to L subsymbol periods of delay).

Figure 8.46's single block-of-subsymbols delay element D holds a subsymbol for 3 subsymbol periods, so $D = D_{ss}^{3}$ and is realized as one storage element. The depth is clearly $\mathcal{J} = L+1$ because the increment in delay between successive subsymbols is $D_{ss}^{\mathcal{J}-1}$.

The classical triangular deinterleaver inverse $G^{-1}(D)$ has negative powers of D in it, so must be realized with delay of $L \cdot (L-1) = (\mathcal{J}-1) \cdot (L-1) = L^2 - L$ symbol periods to be causal. This is equivalent to multiplying $G^{-1}(D)$ by D^{L-1} to obtain

$$G_{causal}^{-1}(D) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \ddots & 0 & 0 \\ 0 & 0 & D^{L-2} & 0 \\ 0 & 0 & ... & D^{L-1} \end{bmatrix} . \tag{8.154}$$

The total delay is $L \cdot (L-1) = (\mathcal{J}-1) \cdot (L-1) = L^2 - L$ symbol periods, which is the theoretical minimum possible. The maximum codeword length to ensure that adjacent interleaved Generally, the classical triangular interleaver requires only at most 1/2 the memory and 1/2 the delay of the classical block interleaver. The classical triangular interleaver carries the restriction (so far) of $\mathcal{J} = L+1$, for any $\mathcal{J}$ and corresponding L.

The triangular interleaver is often said to not require synchronization in that the period can differ from the underlying codeword block length, unlike the block interleaver that essentially accepts entire codewords as input. This is somewhat of a misnomer, in that it requires less synchronization than the block interleaver because the receiver still needs to know the boundaries of codewords for other purposes. The triangular interleaver has an overly restrictive depth of $\mathcal{J} = L+1$ only, but has an attractive and simple triangular implementation.

Generalized triangular interleaving relieves this depth constraint somewhat by essentially replacing each delay element in the triangular implementation by a FIFO (first-in-first-out) queue of M subsymbol periods. However, inputs switch position every subsymbol period. The delay is $M \cdot L \cdot i$ for $i = 0, ..., L-1$ or $(i)_L$. The period remains L subsymbol periods, but the depth increases to $\mathcal{J} = M \cdot L + 1$ symbol periods. Figure 8.47 illustrates the generalized triangular interleaver where the box with an M in it refers to a queue (first-in-first-out) containing M subsymbols and a delay of $M \cdot L$ subsymbols (or M interleaver periods of L subsymbols each). The generalized triangular interleaver is designed for use with block codes of length $N = q \cdot L$ where q is a positive integer. Thus a codeword is divided into q interleaver-period-size groups of size L symbols that are processed by the generalized triangular interleaver. If $q = 1$, one codeword is processed per period, but otherwise $(1/q)^{th}$ of a codeword in general. Often, a symbol is a byte (or "octet" of 8 bits as stated in International Telecommunications Union (ITU) standards).

Thus, the delays for this particular interleaver can be witten as $D_{ss}^{M \cdot L}$. The ith symbol entering the interleaver in Figure 8.47 is delayed by $i \cdot L \cdot M$ block periods. The total delay of any byte is then $\Delta = L \cdot (L-1) \cdot M = (\mathcal{J}-1) \cdot (\mathcal{J}-2)M$ subsymbol periods.

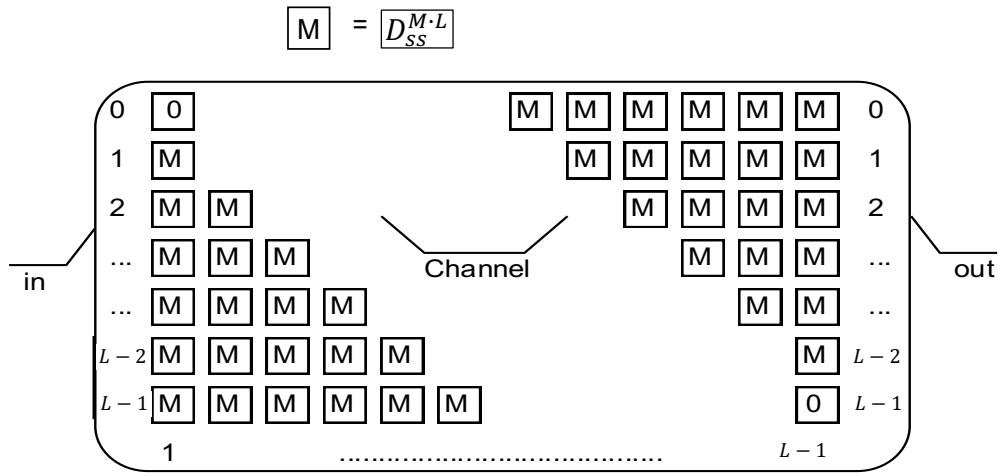

Figure 8.47: **The Generalized Triangular Interleaver.**

The generalized triangular interleaver is typically used with a block code that can correct t subsymbols in error (typically Reed Solomon block code where a symbol is a byte, then t is the number of parity bytes if erasures are used and 1/2 the number of parity bytes if no erasures are used). The block-code codewords may thus be subdivided into q blocks of subsymbols so that $N = q \cdot K$. Then Table 8.25 lists the various parameter relations for the generalized triangular interleaver.

Parameter	Value
Interleaver block length (K)	$K = L$ *subsymbols (equal to or divisor of* N*)*
Interleaving Depth ($\mathcal{J}$)	$\mathcal{J} = M \cdot K + 1$
(De)interleaver memory size	$M \cdot K \cdot K \cdot ({}^{K-1}/_{2})$ subsymbols
Correction capability (block code that corrects t symbol errors) With *q=N/K*)	$\left\lfloor \frac{t}{q} \right\rfloor \cdot (M \cdot K + 1)$ *subsymbols* $\left\lfloor \frac{t}{q} \right\rfloor \cdot (\mathcal{J})$
End-to-end delay	$M \cdot K \cdot (K-1)$ *subsymbols*

Table 8.25: **Table of parameters for Generalized Triangular Interleaver.**

EXAMPLE 8.6.4 **[International Telecommunications Union Generalized Triangular Interleaver]** The ITU uses a Generalized Triangular Interleaver in several transmission-line standards. Table 8.26 works some specific data rates, interleave depths, and consequent delays and memory sizes for the ITU generalized triangular interleaver. More delay can create application problems, but also improves the correcting capability of the code by the depth parameter. The data rates scale with $1/T_{ss}$, which was selected as 50 MHz for this table.

Rate (Mbps)	Interleaver parameters	Interleaver depth ($\mathcal{J}$)	(De)interleaver memory size	Erasure correction	End-to-end delay
50x1024	$K = 72$ $M = 13$	937 blocks of 72 bytes	33228 bytes	3748 bytes 520 ns	9.23 μs
24x1024	$K = 36$ $M = 24$	865 blocks of 36 bytes	15120 bytes	1730 bytes 500 ns	8.75 μs
12x1024	$K = 36$ $M = 12$	433 blocks of 36 bytes	7560 bytes	866 bytes 501 ns	8.75 μs
6x1024	$K = 18$ $M = 24$	433 blocks of 18 bytes	3672 bytes	433 bytes 501 ns	8.5 μs
4x1024	$K = 18$ $M = 16$	289 blocks of 18 bytes	2448 bytes	289 bytes 501 ns	8.5 μs
2x1024	$K = 18$ $M = 8$	145 blocks of 18 bytes	1224 bytes	145 bytes 503 ns	8.5 μs

Table 8.26: **Calculated delays and data rates for a symbol equal to a byte in the ITU Interleaver.**

8.6.1.3 Enlarging the interpretation of triangular interleavers

The delay $\ell \cdot (\mathcal{J} - 1)$ suggests a more general form of triangular interleaving that follows when $\mathcal{J} \leq L+1$. Figure 8.48 illustrates this for $\mathcal{J} = 4$ and $L = 5$. Each subsymbol's delay after it enters the interleaver is again $\Delta_i = i \cdot (\mathcal{J} - 1)\ \forall\ i = 0, ..., L-1$ **subsymbol** periods. The triangular structure remains, but the interleaver-output clocking is irregular as in Figure 8.48 and also in Figure 8.49 for $\mathcal{J} = 3$ and a $L = 5$ subsymbols.

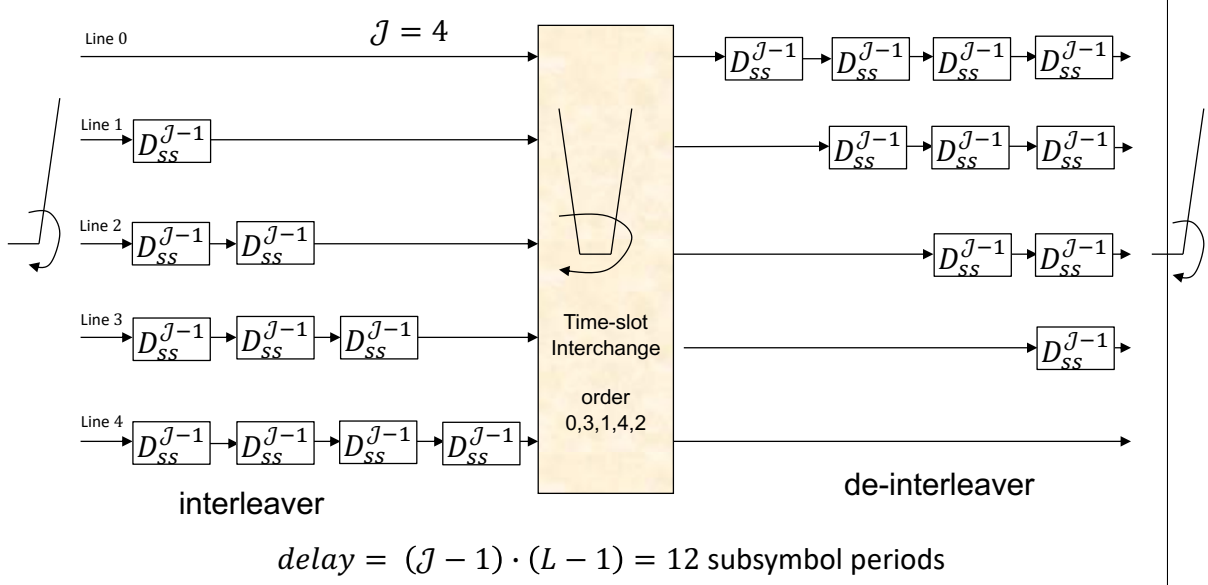

Figure 8.48: **Triangular interleaver for $L = 5$ and $\mathcal{J} = 4$.**

Many authors call the interleavers with $\mathcal{J} < L + 1$ a "convolutional interleaver," using the more general term because Figure 8.48's triangular implementation escaped notice. The triangular implementation follows by noting the generator in form $G(D_{ss})$ remains diagonal with decreasing (increasing) powers of D_{ss}, while the generator $G(D)$ has a more complicated non-diagonal form.

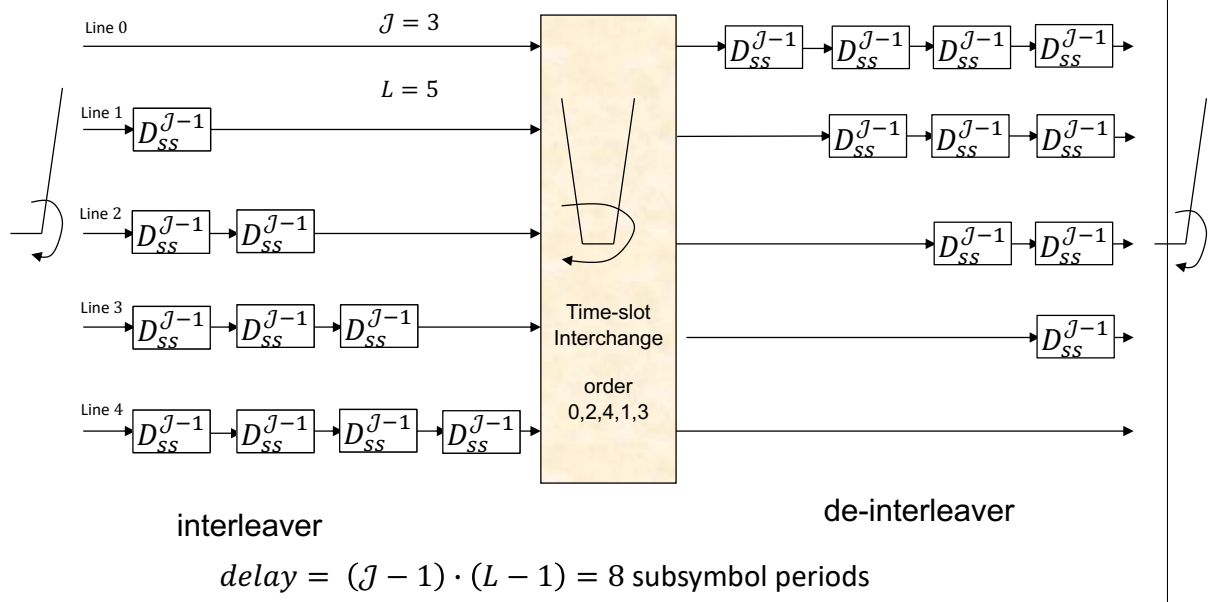

Figure 8.49: **Triangular interleaver for $L = 5$ and $\mathcal{J} = 3$.**

Thus, the first subsymbol ($i = 0$) in a subsymbol block is not delayed at all, while the second symbol is delayed by $\mathcal{J}-1$ **subsymbol** periods, and the last subsymbol is delayed by $(L-1)\cdot(\mathcal{J}-1)$ subsymbol periods. The generator matrix for Figure 8.49's $\mathcal{J} = 3$ example appeared in Example 8.6.3. It had a complicated $G(D)$, but has a simple $G(D_{ss})$. For Figure 8.48's example with $\mathcal{J} = 4$, the generator is

$$G(D) = \begin{bmatrix} 0 & 0 & D & 0 & 0 \\ D & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & D & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{8.155}$$

or

$$G(D_{ss}) = \begin{bmatrix} D_{ss}^{12} & 0 & 0 & 0 & 0 \\ 0 & D_{ss}^{9} & 0 & 0 & 0 \\ 0 & 0 & D_{ss}^{6} & 0 & 0 \\ 0 & 0 & 0 & D_{ss}^{3} & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{8.156}$$

The subsymbol-based diagonal form is simpler and corresponds directly to the implementation. The interior order is different from the case when $\mathcal{J} = L+1$. Figure 8.48's and 8.49's "time-slot interchange" implements this order . The order follows easily: The interleaver's input switch position (and de-interleaver's output switch position) cycles normally through the indices $k = 0, ..., L-1$. The total delay of the kth symbol with respect to the beginning of the period on any line i is $i + i\cdot(\mathcal{J}-1) = i\cdot\mathcal{J}$ symbol periods. After this delay of $i\cdot J$ symbol periods, the symbol must leave the interleaver and thus the interleaver output switch position (and also the de-interleaver input position) must be then be such at time k that

$$(i\cdot\mathcal{J})_L = k \quad , \tag{8.157}$$

where k measures from the beginning of an interleaver period. That is, at time k, the output switch position is a function of k, $i(k)$, such that this equation is satisfied for some index i. When the equation

is solved for time $k = 1$, that particular time-one solution is $i(1) = \Delta$, and $(\Delta\dot{\mathcal{J}})_L = 1$. For all other times k, then the output position is

$$i = (k \cdot \Delta)_L \quad , \tag{8.158}$$

which is easily proved by substituting (8.157) into (8.158) or $((\mathcal{J} \cdot i)_L \cdot \Delta)_L = ((\mathcal{J} \cdot \Delta)_L \cdot i)_L = (1 \cdot i)_L = i$. The switch orders in Figure 8.48 both satisfy this equation for the particular depth. Any depth $\mathcal{J} \leq L + 1$ is possible unless $\mathcal{J}$ and L have common factors. If $\mathcal{J}$ and L have common factors, then (8.157) does not have a unique solution for each value of i, and the interleaver is no longer a 1-to-1 transformation. The delay of this triangular interleaver and de-interleaver (with time-slot interchange) is then always $(\mathcal{J}-1)\cdot(L-1)$ symbol periods. The straightforward implementation's memory requirement is excessive. The memory reduces to the theoretical minimum of $(\mathcal{J}-1)\cdot(L-1)/2$ in each of the interleaver and de-interleaver by a memory-reuse algorithm described momentarily.

The generalized triangular interleaver will follow exactly the same procedure for any depth $\mathcal{J} \leq M \cdot L + 1$. Since M can be any positive integer, then any interleave depth is thus possible (as long as L and $\mathcal{J}$ are co-prime. The time-slot algorithm still follows (8.158). Again, memory in the straightforward conceptual implementation is not minimum, but delay is again at the minimum of $(\mathcal{J} - 1) \cdot (L - 1)$ symbol periods.

Memory reduction to the theoretical minimum Table 8.27 provides an example for Figure 8.49 with $\mathcal{J} = 3$. It illustrates memory-cell reuse in the triangular interleaver (now generalized). Time is indexed over 3 successive interleaving periods, with no prime for first period, single prime for second period, and double prime for the two subsymbol intervals shown in the third period. A subsymbol is indexed by the period in which it occurred as B0, B1, B2, B3, or B4 with primes also used. Line 0's subsymbols (B0) is always immediately passed at time 0 and therefore never uses any memory, and thus does not appear in the table. Hyphens indicate "idle" lines.

Table 2 for J=3 and L=5															
L/t	0	1	2	3	4	0'	1'	2'	3'	4'	0''	1''	2''	3''	4''
0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
1	-	B1	B1	-	-	-	B1'	B1'	-	-	-	B1''	B1''	-	-
2	-	-	B2	B2	B2	B2	-	B2'	B2'	B2'	B2'	-	B2''	B2''	B2''
3	-	-	-	B3	B3	B3	B3	B3	B3 B3'	B3'	B3'	B3'	B3'	B3'' B3'	B3''
4	-	-	-	-	B4	B4	B4	B4	B4	B4 B4'	B4 B4'	B4 B4'	B4'	B4'	B4'' B4'
CELL1	-	B1	B1	B3	B3	B3	B3	B3	B3	B4'	B4'	B4'	B4'	B4'	B4'
CELL2	-	-	B2	B2	B2	B2	B1'	B1'	B3'	B3'	B3'	B3'	B3'	B3'	B4''
CELL3	-	-	-	-	B4	B4	B4	B4	B4	B4	B4	B4	B2''	B2''	B2''
CELL4								B2'	B2'	B2'	B2'	B1''	B1''	B3''	B3''

Table 8.27: **Minimum-memory scheduling: interleaver Figure 8.48 with $L = 5$ and $\mathcal{J} = 3$.**

After time 1 of the third period, the interleaver is in steady state and uses all the minimum of $4 = \frac{(\mathcal{J}-1)\cdot(L-1)}{2}$ memory cells. Each memory cell progressively stores the subsymbols from line 1, then line 4, then line 3, and line 2 before rotating back to line 1 again. The process is regular and repeats on the different memory CELLs offset in time by one period with respect to one another.

An easy way to determine a schedule for the use of the minimum number of memory cells is to realize that the same cell that is read on any subsymbol period of any period must also be written with the next available subsymbol of input with minimum RAM. At design time for a particular specified depth and period, a set of

$$\text{minimum number of cells} = \frac{(\mathcal{J}-1)(L-1)}{2} \tag{8.159}$$

"fake" RAM cells can be created in computer software, each with a time that is set to "alarm" exactly $k\cdot(\mathcal{J}-1)$ symbol periods later where k is the interleaver-input subsymbol-clock index. At each subsequent time period in steady state, one and only one cell's timer will alarm, and that cell should be read and then written and the timer reset to the value of $k \cdot (\mathcal{J}-1)$. Schedules of "which subsymbol when" will then occur for each storage cell that can be stored and used in later operation. This schedule will repeat over an interval for each cell no longer than

$$S = \text{cell schedule length} \leq \sum_{i=1}^{L-1} i \cdot (\mathcal{J}-1) = \frac{1}{2}(\mathcal{J}-1)\cdot L \cdot (L-1) = \frac{\Delta}{2}\cdot L \tag{8.160}$$

symbol periods for each cell. Equality occurs in (8.160) when

$$\frac{m \cdot L}{\mathcal{J}-1} \notin Z \text{ for any } m < \frac{\Delta}{2} - 1 \quad . \tag{8.161}$$

When equality occurs then any integer number of periods less than S cannot be divided by the interleaver-output spacing between formerly adjacent interleaver input subsymbols $\mathcal{J}-1$. In such a case, all cells go through the same length S schedule, just delayed with respect to one another. When the condition produces an integer, then different cells can have different schedules. The number of schedules for different mutually exclusive cell groups with the same schedules within each group, but different from group to group, is the number of values of m for which (8.161) is satisified that are not integer multiples of previous values of m that solve the condition in (8.161). but if there are s such schedules of distinct lengths S_i, then

$$\sum_{i=1}^{s} S_i = \frac{\Delta}{2}\cdot L \quad . \tag{8.162}$$

See problem 8.17 for an interesting development of the above equations. In the minimum-memory implementation described above, the relationship to the triangular structure in the memory connection is still inherent, but it evolves in time to prevent idle memory cells.

Time variation of depth accompanying a data rate change This triangular-interleaver interpretation in Figures 8.46, 8.48, and 8.49 (with or without minimum memory requirement) allows graceful change in operation of the interleaver depth between values that maintain L and $\mathcal{J}$ co-prime and are both lower and or equal to the upper bound[24]of $\mathcal{J} \leq M \cdot L + 1$. The overall delay must be held constant in time to do so. This description calls the new depth $\mathcal{J}'$ and the old depth $\mathcal{J}$. There will be a point in time, call it time 0 at the beginning of a period where all new subsymbols entering the interleaver will observe depth $\mathcal{J}'$ while all subsymbols already in the interleaver (or de-interleaver) will observe the old depth $\mathcal{J}$. The corresponding subsymbol periods will similarly be denoted by T_{ss} and T'_{ss}. (At a constant delay through interleaving and de-interleaver, the subsymbol rate necessarily must change, which implies a data-rate change.) To maintain the same delay in absolute time, and noting that impulse protection for such a delay remains the same, if the codeword length $N = L$ and correction capability remain the same (just the depth changes to accommodate the same length burst of noise/errors), then

$$T'_{ss} = \frac{\mathcal{J}-1}{\mathcal{J}'-1} T_{ss} \quad , \tag{8.163}$$

[24]The constraint can always be met with the generalized triangular interleaver by using a sufficiently large M and dummy subsymbols to maintain the co-prime dpeth and period.

another advantage of using the subsymbol-clock (rather than interleaver period) notation and interpretation. Since subsymbols through the interleaver/de-interleaver combination must undergo exact the same delay whether before or after the depth change, a subsymbol exiting the interleaver or de-interleaver will be "clocked out" at exactly the same point in absolute time. The time-slot-interchange positions also are at the same time. However, there are two symbol clocks in which to interpret absolute time. A master implementation clock is determined by the greatest common multiple, GCM, of $\mathcal{J}-1$ and $\mathcal{J}'-1$ as

$$\frac{1}{\delta T} = \frac{GCM}{\mathcal{J}-1}\frac{1}{T} = \frac{GCM}{\mathcal{J}'-1}\cdot\frac{1}{T'} \quad . \tag{8.164}$$

Essentially the higher speed clock will always touch in absolute time all write/read instants of the interleaver for either the old or the new depth. New subsymbols (subsymbols) enter the interleaver as shown (the interleaver has the same isosceles triangle structure in concept at any depth) in terms of L clock cycles of the new symbol clock with symbol period $T = \frac{GCM}{\mathcal{J}'-1}\frac{1}{\delta T}$ – and any memory element with a new subsymbol after time zero in it will pass subsymbols $k\cdot(\mathcal{J}'-1)T' = k\cdot GCM\cdot\delta T$ time slots later, where k is the index within the constant period of the interleaver. Old subsymbols within the interleaver exit on the $i(k) = k\cdot\Delta$ line at time instants $kT = k\cdot\frac{GCM}{\mathcal{J}-1}\delta T$, while new subsymbols within the interleaver exit on the $i(k) = k\cdot\Delta'$ line at time instants $kT' = k\cdot\frac{GCM}{\mathcal{J}'-1}\delta T$. All these time instants correspond to integer multiples of the high-rate clock. If at any time, both clocks are active then a read operation must be executed before the write operation on that clock cycle occurs (along with all the shifts in the proper order to not drop data within the delay elements shown if indeed the implementation was directly in the isosceles-triangular structure).

When no old subsymbols exist within the structure any longer (which is exactly the delay of the interleaver/de-interleaver combination), then the higher rate clock can be dropped and the new clock and new depth then used until any other depth change occurs in subsequent operation.

The time-variable structure uses the triangular structure to facilitate the explanation. This structure uses the constant $L\cdot(L-1)/2$ locations of memory independent of depth. However, the lower-memory requirement of the cell structure image discussed earlier can again be used even in the depth-variation case. The interleaver situation with larger depth will require a larger number of cells, so the number of cells increases with an increase in depth and decreases with a decrease in depth. For an increase in depth, $(\mathcal{J}'-\mathcal{J})\cdot\frac{L-1}{2}$ more cells are needed in the interleaver (and the same number less for a decrease in depth). These cells can be allocated (or removed) as they are first necessary (or unnecessary).

Just as in the constant-depth case, an off-line algorithm (software) can be executed to determine the order for each new subsymbol position of cell use by simply watching timers set according to the clock $\frac{1}{\delta T}$ as they sound. Any timer sounding at a time that is only a read can be reserved for potential future use. Any new subsymbol to be written can use the cells reserved for use (either because it was vacated earlier by an isolated read at the old clock instants or because it is an additional cell needed). After $\frac{1}{2}\cdot(J-1)\cdot L\cdot(L-1)$ subsymbol periods, the interleaver algorithm (and de-interleaver algorithms) will have completely switched to the new depth. With such minimum-memory-cell use thus established, the depth change can then begin and follow the consequent pattern.

8.6.2 Design with Cyclic Outer Codes

8.6.3 Retransmission Methods

Exercises - Chapter 8

8.1 *Extending our convolutional encoder and code to include a pass-through bit* (20 pts)

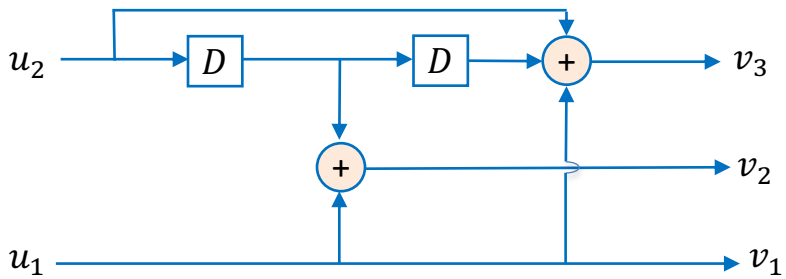

Figure 8.50: A convolutional encoder for Problem 8.1

note: Because u_1 does not affect the state, there are consequently multiple paths between states. That is, there are two different outputs that both have the same origin state and the same destination state but differ in the value of u_1. Stack these vertically with the upper value for $u_1 = 0$ and the lower value for $u_1 = 1$.

a. (2 pts) Determine Figure 8.50's code rate r, along with k and n, and as well the constraint length ν.

b. (2 pts) Determine the encoder's generator matrix $G(D)$ and a corresponding parity matrix $H(D)$.

c. (3 pts) Insert Part b's $G(D)$ into matlab's poly2trellis.m program and show that program's trellis.nextStates and trellis.outputs. Draw trellis diagram using octal labels that conforms to matlab's poly2trellis convention. Use of the plotnextstates.m program is acceptable. Also, branch labels do not arise from this program, so paste them on it. For instance $\begin{smallmatrix}0\\7\end{smallmatrix}$ might be a label for a parallel transition with the upper value corresponding to $u_1 = 0$ and the lower value corresponding to $u_1 = 1$.

d. (3 pts) Find this code's d_{free}. (You may do this by hand or use one of the programs in Chapter 7.) How many input bit errors occur with a minimum-distance error event? How many types of minimum-distance error events are there?

e. (4 pts) If the input is $[u_2\ u_1] = 10\ 00\ 00$, find the corresponding encoder output, assuming the encoder was in state 0 initially at time $m = 0$. Recall that matlab will reverse the order of the bits in each input and accepts essentially $[u_1\ u_2]$ as inputs to convenc.m. Use your trellis in Part c and show the 4 branches in 4 successive stages. How does this path relate to the all zeros path? What is the distance between them? Relative to other paths through this trellis compared to the all zeros path, is this path more or less likely to be confused than most?

f. (3 pts) If the output corresponding to Part e incurs 1 error in bit v_3 at time $m = 2$, find the MLSD output. If the decoded output, even with only 1 bit error, does not match the input, explain why.

g. (3 pts)Repeat f using a sufficient number of extra inputs to force return to state 0. Explain how this caused the error to be now corrected. How could the cost of these extra input bits in terms of bandwidth be reduced? (Incidentally, the current BCJR_BSC.m program does handle parallel transitions: an extra credit project for enthusiastic student.)

8.2 *Systematic encoders, or encoding with feedback. (20 pts)*
Figuure 8.51 illustrates a convolutional code's minimal encoder realization.

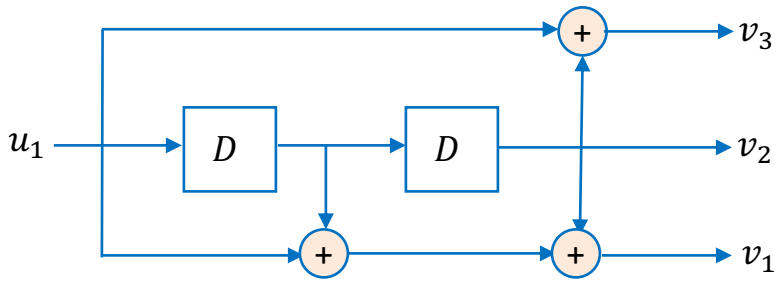

Figure 8.51: Another convolutional encoder.

a. (1 pt) Is this encoder systematic? How many states are there? What is the rate r?

b. (2 pts) Find Figure 8.51's generator matrix $G_{min}(D)$ and a corresponding parity matrix $H(D)$.

c. (3 pts) Find a systematic encoder $G_{sys}(D)$ that does not increase the number of states and draw a diagram of it. This systematic encoder will involve feedback. Attempt to describe this systematic encoder with matlab's poly2trellis.m command? Does what it produces make sense in this instance? (e.g., is the bug evident?) Why or Why not?

d. (2 pts) Plot the trellis with superimposed subsymbol code outputs superimposed. Also, find d_{free} by inspection of trellis, as well as check it with distspec.m.. [Hint: Be careful to understand the relation between nextStates and outputs in matlab's trellis description.]

e. (1 pt) Find an invertible translation of $G_{sys}(D) = G_{min}(D) \cdot A(D)$.

f. (2 pts) Encode the sequence [01 10 11 10] using both encoders. Compare the two outputs - Why do they differ if same code?

g. (4 pts) Extend the input sequence by 2 subsymbols to ensure it returns to state 0, and MAP decode the systematic encoder with 2 output-bit errors at $m = 1$ and $m = 3$ respectively on bit v_2. Show the soft information. Use $p = 0.1$. Comment on the different soft information for the two decoders and what it might mean for MAP decoders with respect to encoder choice - hint: consider your answer in Part e

h. (2 pts) Group pairs of subsymbols and alternately puncture bits v_1 and v_3 from the successive subsymbol outputs. What rate is new code rate r? What size is the new generator? What is ν? What is d_{free}?

i. (3 pts) Repeat part g with the punctured code, recalling that subsymbols are larger (so perhaps more zeroed inputs necessary). Provide soft information, but in this case neither encoder structure seems to have clearly better soft information, so don't expect to see that in the soft information.

8.3 *Convolutional Code Analysis. (10 pts)*
A convolutional code is described by the trellis shown in Figure 8.52. The state label is u_{k-1}. The upper state is 0 and the lower state is 1. Other labellings follow the conventions in the text. There are 2 outputs; v_2 and v_1.

a. (2 pt) Determine the rate and the constraint length of the convolutional encoder corresponding to this trellis. *Hint*: Can you see from the trellis that the encoder has only one input u?

b. (2 pts) Determine the generator matrix $G(D)$ for the code.

c. (2 pts) Draw a circuit that realizes $G(D)$.

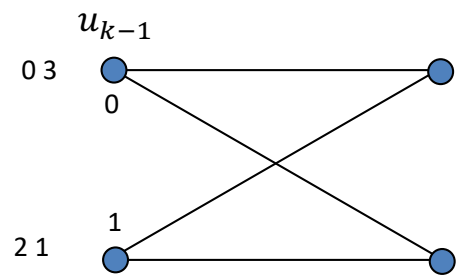

Figure 8.52: Trellis corresponding to a convolutional encoder.

d. (1 pt) Find d_{free}.

e. (3 pts) How many codewords are there, each of weight 3? How many input bit errors occur in an output error event of length l?

8.4 *No Free Lunch Code (18 pts).*

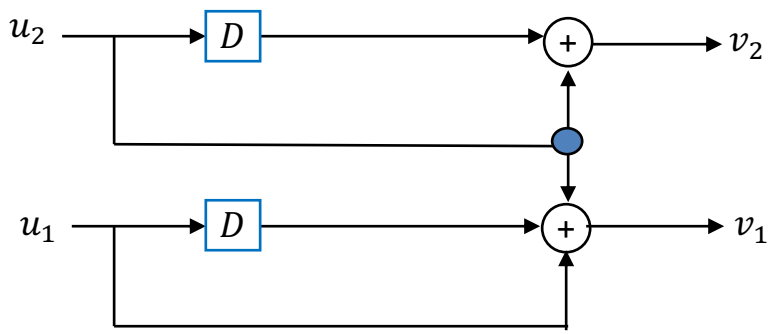

Figure 8.53: A rate 1 convolutional code.

a. (2 pts) Find this code's parameters k, n, r, and ν.

b. (2 pts) Draw the trellis for this code with output labels for an AWGN channel with 2PAM (± 1) on each of the outputs.

c. (3 pts) By inspection of your trellis determine d_{free} and d_{min}. What is this code's coding gain for use on the AWGN in dB if we use the simple coding gain formula of $r \cdot d_{free}$?

d. (2 pts) Find a finite-length channel output, which when decoded corresponds to an infinite number of input 1 values. (Hint: the all zeros sequence is finite length.)

e. (2 pts) Would you expect to see this coding gain realized? Why or Why not?

f. (2 pts) Design an $r = 2/3$ code that does not change the relationships of u_1 and u_2 to v_2 and v_1, without adding any more delay elements (or states).

g. (2 pts) What is coding gain for your new code? Would you expect to realize this gain?

h. (3 pts) Compare the trellis to Part b's trellis and explain why the redundancy helped?

8.5 *Exploring bandwidth power trade-off in system coding design (6 pts)*
Assume we are using a binary transmission system on an AWGN channel with $SNR = 8dB$.

a. (1 pt) Determine the P_e for uncoded binary PAM on this channel.

b. (2 pts) Suppose the design permits subsymbol-rate doubling (and hence use of twice as muchbandwidth) at fixed power. With these constraints, a new design transmits $2PAM$ subsymbols in a convolutional code at the same data rate R as in part (a). Use the tables provided in Section 8.2 to find the code which provides the largest free distance (among those on the tables) of all the codes which do not reduce the rate r by more than a factor of two, and which require no more than six delay elements for implementation. Give $G(D)$ for the code.

c. (2 pts) What is the coding gain of this code? Use ing soft decoding, approximate the P_e of the coded system. How much did coding reduce our P_e?

d. (1 pt) So even when $\bar{b} < 1$, which should the designer take - a linear factor increase in power or a linear factor increase in bandwidth (assuming no ISI)?

8.6 *Convolutional Code Design.*

A baseband PAM ISI channel with Gaussian noise is converted to an AWGN channel, by using a ZF-DFE, operating at a symbol rate $1/T = 10$ kHz. It is calculated that $SNR_{ZF-DFE} = 7.5$ dB.

a. (2 pts) For the AWGN channel produced by the DFE, what is the capacity $\overline{C}$ and C?

b. (1 pt) Find the P_e for uncoded 2-PAM transmission on this channel.

c. (5 pts) The P_e produced by uncoded PAM (part (b)) is intolerable for data transmission. We would like to use convolutional coding to reduce the P_e. However, since we're using a DFE that has been designed for a specific symbol rate, we cannot increase the symbol rate (ie. bandwidth). Nor are we allowed to increase the transmitted signal power. Further, we have a limit on the decoder complexity for the convolutional code, which is represented mathematically as,

$$\frac{1}{k} \cdot [2^{\nu}(2^k + 2^k - 1) + 2^n] \quad \leq \quad 320$$

where k, n, ν have the usual meaning for a convolutional code. Under these constraints, design a convolutional encoder, using the convolutional coding tables of Section 8.2, so as to achieve the highest data rate R for $P_e < 10^{-6}$. What is this maximum R?
Hint: Use the soft decoding approximation to calculate P_e and remember to account for N_e.

d. (2 pts) Draw the systematic encoder and modulator for the convolutional code designed in part (c).

8.7 *Concatenated Convolutional Code Design - 19 pts*

A baseband AWGN has SNR = 8 dB for binary uncoded transmission with symbol rate equal to clock rate of 1 Hz. (Recall that the clock rate of an AWGN channel can be varied when codes are used.)

a. What is the capacity C of the AWGN channel in bits/sec? (1 pt)

b. Design a convolutional code with $r = 3/4$ bits per dimension that achieves the lowest probability of error using one of the codes listed earlier in this chapter, assuming a soft MLSD decoder. This coded system should have the same data rate as the uncoded system. (4 pts) ("Design" means for you to provide an acceptable generator matrix $G(D)$ and to compute the corresponding probability of symbol error.)

c. Model the system in part b as a binary symmetric channel (BSC) and assume that N_b for the code you selected is 2. Provide the value $p =$? (1 pt)

d. For the BSC in part c, Design a convolutional code of rate $r = 2/3$ that achieves the lowest probability of symbol error using codes listed in Section 8.2 (4 pts)

e. What is the combined data rate R of the "concatenation" of the codes you designed in parts b and d? (2 pt)

f. Return to the uncoded transmission system and now design an $r = 1/2$ convolutional code with soft decoding that has lowest probability of error based on codes in Chapter 7. This coded system should have the same information rate, R, as the concatenated code above in part e. (Hint - think carefully about what clock rate you would use.) (4 pts)

g. Compare the system in part f to the system in part d. What does this comparison tell you about concatenation of codes? (3 pts)

8.8 *Convolutional Code design for Satellite Data Transmission (11 pts).*

A digital satellite AWGN channel with $SNR = 17.5$ dB uses two dimensional symbols (QAM) with an uncoded symbol rate of 35 MHz to send 140 Mbps digital signals.

a. (2 pts) How many information bits per 2-D symbol are required to send the HDTV signals? What 2-D constellation would typically be used to send this many bits (uncoded)? If ideal bit interleaving is used and retaining the same constellation but allowing the symbol rate to increase, what happens to SNR if a binary code with rate r is used to encode the information bits?

b. (1 pt) What is the P_e obtained by uncoded transmission using the constellation found in the previous part?

c. (2 pts) What γ is needed for a convolutional code to achieve $P_e \leq 10^{-6}$?

d. (3 pts) From the convolutional-coding tables, find the code with no more than 8 states and least bandwidth expansion that will achieve $P_e \leq 10^{-6}$. Find the associated P_e, parity matrix, and systematic encoder matrix. What is the bandwidth expansion factor?

e. (3 pts) Draw the encoder circuit and the Gray-labeled constellation.

Note: A satellite channel typically has no ISI. Therefore, no DFE is required, and so, there is no problem of error propagation.

8.9 *Lost Charger (10 pts)*

Your PDA nominally uses (uncoded) 16 QAM to transfer data at 10 Mbps to a local hub over a wireless, stationary AWGN channel probability of symbol error 10^{-6}. In transferring data, 100 mW of transmit power uses 500 mW of total PDA battery power, and you may assume that the ratio of 5:1 consumed/transmitted power holds over a range of a factor 4 above or below 100 mW. Unfortunately, you've lost your battery charger and must transfer a 10 Mbyte file (your exam) sometime within the next 3 hours to the internet for grading, or get an F otherwise, at the same sufficiently low probability of error at the fixed symbol rate of nominal use. Your battery has only 1.5 Joules of energy left (1 Watt = Joule/second). You may assume that the binary logic for a convolutional code, as well as any coset and point-selection hardware, uses negligible energy in a trellis encoder.

a. Design a 2-dimensional trellis encoded system that allows you to transmit the file, showing the encoder, constellation, and labeling at the same symbol rate. (5 pts)

b. The hub receiver is a separate device that would have consumed 0.1 Joules of energy for nominal uncoded transmission, assuming $N_D = 1$. How much energy will it consume for your design in part a? (2 pts)

c. Suppose the symbol rate could be changed. What is the minimum energy that any code could use to transmit this (10 Mbyte) file? Equivalently what is the largest size file that could be transferred with your battery (1.5 Joules)? (3 pts)

8.10 *SH Diagrams - 11 pts*

Section 8.5 describes A 128 shell constellation for $d = 2$.

a. Draw this constellation showing all 17 shells. (4 pts)

b. How many shells does a 64SH constellation have? (2 pts)

c. Draw a 64SH constellation and show the shells (3 pts)

d. What is the shaping gain of 64SH? (2 pts)

8.11 *Hexagonal Shells - 9 pts*

Consider hexagonal packing and shaping gains in this problem.

a. For a constellation based on the A_2 hexagonal lattice, how many shells are necessary for a 24-point constellation? (4 pts)

b. What is the average energy of the 24SH constellation based on $A-2$? (3 pts)

c. What is the total coding gain of this constellation? (1 pt)

d. For equal numbers of constellation points, do rectangular or hexagonal points have more shells? (1 pt)

8.12 *Turbo Design and Decoding - 8 pts*

A baseband AWGN channel has SNR=7 dB with $\frac{\mathcal{N}_0}{2} = -60$ dBm/Hz with uncoded symbol rate 20 MHz with fixed transmit power. Transmitter implementation constraints limit code application to use of only a two-level binary constellation. The objective will be a turbo-code design that has $\bar{P}_b \leq 10^{-6}$.

a. What is the transmit power P_x for this channel? What is $\bar{\mathcal{E}}_{\boldsymbol{x}}$ for uncoded transmission. (2 pts)

b. What is this channel's capacity $\bar{\mathcal{C}}$ in bits/dimension? (1 pt)

c. What coding gain γ will ensure transmission at $P_e \leq 10^{-6}$? (1 pt)

d. Generate 1003 random bits using matlab's randi.m facility, and force the last 3[25] to be zeros. (Hint - subtract 1 from random integers 1,2; to get ± 1 from these bits, subtract 1/2 and take the sign.)

   The function randi.m's use can pick any seed because it depends on matlab's internal code, but use rng(7) for this problem where necessary for consistency. Add noise at the appropriate level for the uncoded case and count the number of bit errors made with with a binary-AWGN decoder. Run this 100000 times to estimate bit-error probability and compare to theory. (2 pts)

e. Prepare for a turbo design by selecting an $r = 1/2$ constituent-mother convolutional code with no more than 8 states. Encode the same bits from Part d and add noise at the appropriate level for fair comparison to Part d and count the number of bit errors made with an ML decoder. (3 pts)

f. Design an $r = 1/3$ parallel turbo code using Part e's selected mother convolutional code for this channel. Again use the same input and count the number of errors with noise added at the appropriate level for fair comparison. Your random interleaver may use matlab's randperm.m command. How does this compare to the rate 1/2 convolutional code by itself? Is it more than a factor or 1000/2 better? Why? (2 pts)

8.13 *Constraints and BICM - 20 pts*

[25] The extra 3 bits ultimately will ensure later that an encoder returns to state zero.

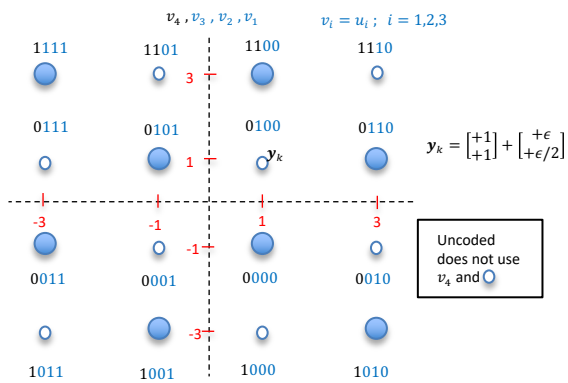

Figure 8.54: Constellations for Problem 8.13.

SQ QAM transmission uses Figure 8.54's constellation on an AWGN. Uncoded transmission uses only the larger blue symbol values. A coded design uses this text's tabluated best $r = 3/4$ binary convolutional code with and without BICM-ID and the constellation C. Coded use includes all 16 vectors as possible subsymbol values. The received value

$$\boldsymbol{y}_k = \begin{bmatrix} +1 \\ +1 \end{bmatrix} + \begin{bmatrix} \epsilon \\ \epsilon/2 \end{bmatrix}$$

at time k also appears where $\epsilon > 0$ is small. The uncoded SNR is 14.5 dB with $\bar{\mathcal{E}}_x = 1$. The gray-coded (for 16 points) constellation values appear on the figure.

a. For the uncoded design, what is the ML decision? What is the symbol error probability for the specific $\boldsymbol{y}_k$ value as $\epsilon \to 0$? What are the ML-detector's 3 individual uncoded bits' error probabilities as $\epsilon \to 0$? The answers may assume that any constellation point further than 3 distance units from $\boldsymbol{y}$ has negligible probability of corresponding to the correct transmitted message. (2 pts)

b. For a sequence of uncoded transmissions and ML detection, what is the average **symbol-error** probability P_e and the corresponding bit-error rate $\bar{P}_b$ in terms of SNR? Evaluate for $SNR = 14.5$ dB (2 pts)

c. Find the 3 uncoded bits' MAP decision for Part a's specific $\boldsymbol{y}_k$ value with $\epsilon \to 0$? What are the corresponding $\bar{P}_b$'s for this detector? Comment on ML-sequence versus MAP for this case. (2 pts)

d. Repeat Part b for a 32-state $r = 3/4$ convolutional-code's use with ideal random interleaving (but not yet iterative decoding), and list your systematic generator matrix[26] $G_{sys}(D)$. Run matlab poly2trellis on this encoder and comment. For the P_e and $\bar{P}_b$ **calculation** (so no simulations here, just analyze with formula), include the 3 smallest-distances' increasing nearest neighbor effects[27]. (4 pts)

e. Let $\epsilon = 0.1$. Compute the $LLR_i(\boldsymbol{y}_k)$ for each **uncoded** input bit and interpret in terms of your answer for Part c's MAP decisions. Only the 4 nearest points need enter the likelihood calculation as the other 4 are negligible. What can you say about this small ϵ relative to $\epsilon = 0$? (4 pts)

f. For an APP decoder with no interleaving, how many possible γ_k values are there for an APP decoder? How many transitions are there with common γ_k values? Compute the 4 γ_k values that correspond to transitions with branch values closest to Figure 8.54's received $\boldsymbol{y}_k$ value. (3 pts)

[26]Hint: The text's usual systematic generator form of $[I \vdots h^t(D)]$ can circularly shift all columns by one position to the right so that the indexing $v_i = u_i \; \forall i = 1, .2, 3$ consistently matches Figure 8.54's indexing - reindexing bits does not change the code nor average bit-error rate $\bar{P}_b$ over all bits. Specifically, $G_{sys}(D)$ can have equivalent reindexed form $[h^t(D) \vdots I]$.

[27]As matlab's distspec.m command fails on its own generated trellis, so you'll need to use information in Section 8.2 and the constellation mapping to proceed.

g. For $\epsilon = .1$ at specific time k, a previous APP iteration for surrounding times $k' \neq k$ (for which we have nothing but its output extrinsic values) provides $\hat{p}_3 = Pr\{v_3 = u_3 = 1\} = .9$, $\hat{p}_2 = .99$, and $\hat{p}_1 = .1$, after de-interleaving. How does this alter the values into the convolutional code's APP decoder LLR_i, $i = 1, 2, 3$? (3 pts)

8.14 *LDPC Use - 14 pts*

A generic (529, 462) LDPC code (see Subsection 8.3.3.1) is used with a 64SQ QAM constellation with subsymbol rate $1/T = 1$ MHz.

a. At $\bar{P}_b = 10^{-7}$, what is the gap to capacity of this code with respect to uncoded at this same bit-error rate? What is the corresponding coding gain expected at this $\bar{P}_b$? (2 pts)

b. What is the coded systems' data rate? (1 pt)

c. Without printing it all, find the parity matrix H for this code (use matlab) and show the lower right 10×10 values of H. What happens if this generic-LDPC-code parity matrix H is input to the ldpcEncoderConfig? Find an equivalent code (different input mapping) that averts this issue, call it say $H_{reverse}$ and list its upper left 10×10 entries and comment on sparse matrices. (3 pts)

d. Encode 462 random PRBS10 bits and show the encoder output **bit** values for positions 132 to 143. How many 64SQ QAM subsymbols correspond to this, and how many known dummy bits would be inserted into the last 64 SQ QAM subsymbol? (1 pt)

An AWGN has $SNR = \bar{\mathcal{E}}_{\boldsymbol{x}}/\sigma^2 = 21$ dB at subsymbol rate $1/T = 1$ MHz.

e. Compute $\bar{P}_b$. Estimate P_e from $\bar{P}_b$. Note the LDPC coding-gain/gap tables in Section 8.3 are actually for gain in terms of bit-error rate. (2 pts)

f. Run 10000 independent simulations of this channel with its decoder and compare your $\bar{P}_b$ to your computed value. Explain any differences. (5 pts)

8.15 *Subsymbol- versus Symbol-Level Deterministic Interleaving - 12 pts*

A generalized triangular interleaver designed for bytes (symbols) has period $L = 7$ and depth $J = 5$.

a. Find this interleaver's generator matrices $G(D_{ss})$ and $G(D)$. (2 pts)

b. Find the inverse generator matrix $G^{-1}(D)$ for the de-interleaver. (2 pts)

c. Draw the subsymbol-based interleaver $G(D_{ss})$ and corresponding de-interleaver $G^{-1}(D_{ss})$. How many bytes of memory are used? (2 pts)

d. Draw the period-based interleave $G(D)$ and corresponding de-interleaver $G^{-1}(D)$. (Hint: you may show an ordering or simply connect elements appropriately.) (4 pts) Does this change the number of memory bytes used?

e. What are the respective end-to-end delays through the interleaver and de-interleave systems of the last two parts? (1 pt)

f. Assuming the byte-level code associated with this interleaver corrects dfree bytes in error - what is the number of bytes that can be corrected from burst disturbances with the interleaver (if the burst is 35 samples or less). (1 pt)

8.16 *Wireless Hard-Soft Interleaving Challenge - xx pts*

A wireless channel uses a 16 QAM modulator to transmit data with Gray Coding of bits to symbol values. The symbol clock is 4 MHz. In addition to the channel being an AWGN, each channel use has gain that can vary with symbol period. When in a nominal state, the channel has SNR=18 dB. In a fading state, the SNR is either 21 dB with probability 0.999 or -9 dB with probability .001. Each of the gains is independent of the others. Initially, the probability that the symbol is in either state is independent of all other symbols.

a. What is the data rate? (1 pt)

b. What is the symbol-error probability P_e in the nominal state? (1 pt)

c. What is the average symbol-error probability $\langle P_e \rangle$ now including the fading state? (Hint, recall for M-ary QAM, $P_e < \frac{M-1}{M}$ because random selection would be a better detector.) (2 pts)

d. The channel's fades last for 25 μs. How does the fade duration affect the number of bit errors in a burst for the uncoded system? (1 pt)

At this point on this AWGN, the design choices of $|C|$ and $1/T$ are left to the designer (you), but that the consequent uncoded AWGN inner system can only produce hard decisions. The target bit error rate is $\bar{P}_b \leq 10^{-6}$.

e. Use the best 4-state $d = 5$ convolutional code as an outer code with hard decoding and design an interleaving scheme of minimal end-to-end delay for this code that ensures that the probability of symbol error that is less than 10^{-6} without loss of data rate with respect to Part a. (6 pts)

8.17 *Schedules – 11 pts*

A given triangular interleaver has depth $J = 3$ and period $L = 4$.

a. What is the minimum number of memory cells? (1 pt)

b. Create a scheduling diagram similar to those in Tables **??** and 8.27 for this triangular interleaver? (5 pts)

c. How many different length schedules are there in your answer for part a? (1 pt)

d. Compute S and compare to the sum of the lengths of the different-length schedules. (1 pt)

e. Explain the condition in Section **??** that $\frac{mL}{J-1}$ should not be an integer if the number of periods is smaller than half the delay minus one. (3 pts)

Bibliography

[1] Claude E. Shannon *A Mathematical Theory of Communication* Bell Systems Technical Journal, 1948. Vol. 27, No. 3, pp. 628-634.. See also *Communication in the Presence of Noise* Proceedings of the IRE, 1949 Vol. 37, No. 1, pp. 10-21.

[2] Thomas M. Cover and Joy A. Thomas *"Elements of Information Theory"*. Wiley, New York, 1991.

[3] Shu Lin and Daniel Constello *"Error Control Coding: Fundamentals and Applications"*. Prentice Hall, Englewood Cliffs, NJ 1983.

[4] Simeon Ball *"A course in Algebraic Error-Correcting Codes"*. Berkhauser, Cham Switzerland, 2020.

[5] Philippe Piret *"Convolutional Codes"*. MIT Press, Cambridge MA, 1988.

[6] R. Johannesson and K. Sigangirov *"Fundamentals of Convolutional Coding"*. IEEE Press, 1999.

[7] Thomas Kailath, Ali H. Sayed, and Babak Hassibi. *"Linear Estimation"*. Prentice Hall, USA, 2000.

[8] G. David Forney, Jr. *"On the role of MMSE estimation in approaching the information-theoretic limits of linear Gaussian channels: Shannon meets Wiener"*. Proceedings Allerton Conference, pp.430-439 October 2003

[9] Abbas El Gamal and Young-Han Kim *"Network Information Theory"*. Cambridge University Press, UK 2011

[10] Sumanth Jagannathan *INTERFERENCE AND OUTAGE OPTIMIZATION IN MULTI-USER MULTI-CARRIER COMMUNICATION SYSTEMS* Ph.D. Dissertation, Stanford University. 2008

[11] Stephen B. Wicker *"Error Control Systems for Digital Communication and Storage"* Prentice Hall, New Jersey 1995

[12] N. McKeown, A. Mikkittikul, V. Anantharam, and J. Walrand *"Achieving 100% Throughput in an Input-Queued Switch"* Proceedings IEEE Infocom 1996, San Francisco, CA, pp. 296-302

[13] K. Seong, R. Narsimhan, and J.M. Cioffi *"Queue Proportional Scheduling in Gaussian Broadcast Channels"* Proceedings IEEE International Conference on Communications 2006, Isltanbul, TURKEY , pp. 1647-1652

[14] G. Caire and G. Taricco, and E. Biglieri *"Bit-Interleaved Coded Modulation"* IEEE Transactions on Information Theory May 1998, May , Vol 44, No. 3, pp. 927- 946

[15] Joel Gorham Smith *"The Information Capacity of Amplitude and Variance-Constrained Scalar Gaussian Channels"* Information and Control April 1971, Vol 18, No. 3, pp. 203 - 219

[16] Claude Berrou and Alain Glavieux *"High-Rate Recursive Convolutional Codes for Concatenated Channel Codes"* IEEE Transactions on Communications October 1996, Vol 44, No. 10, pp. 1261-1271

[17] Fred Daneshgaran, Massimiliano Laddomada, and Marina Mondin *"Near Optimum Error Correction Coding and Decoding: Turbo-Codes"* IEEE Transactions on Communications November 2004, Vol 52, No. 11, pp. 1846-1850

[18] D. Divsalar and F. Pollara *"On the Design of Turbo-Codes"* Jet Propulsion Laboratory Progress Report 42-123 November 15, 1995, pp. 5-121

[19] G. Unberboeck *"Trellis-Coded Modulation with Redundant Signal Sets: Parts 1 and 2"* IEEE Communications Magazine February 1987, Vol. 25, No. 2, pp. 5-21.

[20] Fred Daneshgaran and Massimiliano Laddomada and Marina Mondin *"High-Rate Recursive Convoluiional Codes for Concatenated Channel Codes"* IEEE Transactions on Information Theory November 2004, Vol. 52, No. 11, pp. 1846-1850.

[21] Robert Gallager *"Low-Density Parity-Check Codes"* IEEE Transactions on Information Theory. January 1962, Vol. 10, No. 1, pp. 21-28.

[22] Tom Richardson and Rüdiger L. Urbanke *"The Capacity of Low-Density Parity-Check Codes Under Message-Passing Decoding"* IEEE Transactions on Information Theory. February 2001, Vol. 47, No. 2, pp. 599–618

[23] Sae-Young Chung and G. David Forney, Jr. and Tom Richardson and R udiger L. Urbanke *"On the Design of Low-Density Parity-Check Codes within 0.0045 dB of the Shannon Limit"* IEEE Communications Letters. February 2001, Vol. 5, No. 2, pp. 58-60.

[24] Gottfried Ungerboeck *"Channel Coding with Multilevel/Phase Signals"* IEEE Transactions on Information Theory. January 1982, Vol. 28, No. 1, pp. 55-67.

[25] Evangelos Eletherious and Sedat Öelcer *"LDPC codes standards contributions ITU CF-060, 061, RN-072, 073."* International Telecommunications Union. January and May, 2001

[26] Paul Fortier and Antonio Ruiz and John M. Cioffi *"Multidimensional Signal Sets Through the Shell Construction for Parallel Channels"* IEEE Transactions on Communications. Vol. 36, No. 3, March 1992, pp. 500-512.

[27] A, Robert Calderbank and Lawrence H. Ozarow *"Nonequiprobable Signaling on the Gaussian Channel"* IEEE Transactions on Information Theory. Vol. 36, No. 4, July1992, pp. 726-740.

[28] Tom Richardson and Shrinivas Kudekar *"Design of Low-Density Parity Check Codes for 5G New Radio"* IEEE Communications Magazine. Vol. 56, No. 3, March 2018, pp. 28-34.

[29] Dennis Hui and Sara Sandberg and Yufei Blankenship and Mattias Andersson and Leefke Grosjean *"Channel Coding in 5G New Radio"* IEEE Vehicular Technology Magazine. Vol. 13, No. 4, December 2018, pp. 260-69.

[30] Richard E. Blahut *"Theory and Practice of Error Control Codes"* Addison-Wesley, Reading, MA. 1983.

[31] John Gill *"Error Control Codes"* Class Notes, Stanford EE387 Class. 2020.

Index