6.02 Spring 2011  
Lecture #10

- convolutional codes
- state & trellis diagrams
- most likely message to have been transmitted

---

**Do We Need Better Channel Coding?**

The graph shows how a rate ½ “rectangular” block code experimentally improves over using no coding at all, especially at higher SNRs (lower overall BER).

But in low SNR environments, there’s considerable room for improvement.

Can we find more effective rate ½ codes?

---

**Convolutional Codes**

- Like the block codes discussed earlier, send parity bits computed from blocks of message bits
  - Unlike block codes, don’t send message bits, only the parity bits!
  - The code rate of a convolutional code tells you how many parity bits are sent for each message bit. We’ll be talking about rate 1/p codes.
  - Use a sliding window to select which message bits are participating in the parity calculations. The width of the window (in bits) is called the code’s constraint length.

```
0101100101100011...
```

Phoning home using a k=15, rate=1/6 convolutional code
Block diagram view

- One often sees convolutional encoders described with a block diagram like the following:

  ![Block Diagram](image)

  - Think of this a “black box”: message in, parity out
    - Input bits arrive one-at-a-time on the wire on the left
    - The box computes the parity bits using the incoming bit and the k-1 previous message bits
    - At the end of the bit time, all the saved message bits are shifted right one location and the incoming bit moves into the left locn.

Parity Bit Equations

- A convolutional code generates sequences of parity bits from sequences of message bits:

  \[
  p_i[n] = \left( \sum_{j=0}^{k-1} g_i[j] x[n-j] \right) \mod 2
  \]

  - \( k \) is the constraint length of the code
    - The larger \( k \) is, the more times a particular message bit is used when calculating parity bits
      - greater redundancy
      - better error correction possibilities

  - \( g_i \) is the k-element generator polynomial for parity bit \( p_i \).
    - Each element \( g_i[n] \) is either 0 or 1
    - More than one parity sequence can be generated from the same message; a common choice is to use 2 generator polynomials

Example: xmit 1011

- We’ll transmit the parity sequences, not the message itself
  - As we’ll see, we can recover the message sequences from the parity sequences
  - Each message bit is “spread across” \( k \) elements of each parity sequence, so the parity sequences are better protection against bit errors than the message sequence itself

- If we’re using multiple generators, construct the transmit sequence by interleaving the bits of the parity sequences:

  \[
  xmit = p_0[0], p_1[0], p_0[1], p_1[1], p_0[2], p_1[2], \ldots
  \]

  - Code rate is \( 1/\text{number of generators} \)
    - 2 generator polynomials → rate = \( \frac{1}{2} \)
    - Engineering tradeoff: using more generator polynomials improves bit-error correction but decreases the number of message bits/sec that can be transmitted
Example

- Using two generator polynomials:
  - \( g_0 = 1, 1, 1, 0, 0, \ldots \) abbreviated as 111 for k=3 code
  - \( g_1 = 1, 1, 0, 0, 0, \ldots \) abbreviated as 110 for k=3 code

- Writing out the equations for the parity sequences:
  - \( p_0[n] = (x[n] + x[n-1] + x[n-2]) \mod 2 \)
  - \( p_1[n] = (x[n] + x[n-1]) \mod 2 \)

- Let \( x[n] = [1,0,1,1,\ldots] \); as usual \( x[n]=0 \) when \( n<0 \):
  - \( p_0[0] = (1 + 0 + 0) \mod 2 = 1 \)
  - \( p_1[0] = (1 + 0) \mod 2 = 1 \)
  - \( p_0[1] = (0 + 1 + 0) \mod 2 = 1 \)
  - \( p_1[1] = (0 + 1) \mod 2 = 1 \)
  - \( p_0[2] = (1 + 0 + 1) \mod 2 = 0 \)
  - \( p_1[2] = (1 + 0) \mod 2 = 1 \)
  - \( p_0[3] = (1 + 1 + 0) \mod 2 = 0 \)
  - \( p_1[3] = (1 + 1) \mod 2 = 0 \)

- Transmit: 1, 1, 1, 1, 0, 1, 0, 0, \ldots

“Good” generator polynomials

Table 1-Generator Polynomials found by Busgang for good rate ½ codes

<table>
<thead>
<tr>
<th>Constraint Length</th>
<th>( G_1 )</th>
<th>( G_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>4</td>
<td>1101</td>
<td>1110</td>
</tr>
<tr>
<td>5</td>
<td>11010</td>
<td>11101</td>
</tr>
<tr>
<td>6</td>
<td>110101</td>
<td>111011</td>
</tr>
<tr>
<td>7</td>
<td>110101</td>
<td>1101011</td>
</tr>
<tr>
<td>8</td>
<td>110111</td>
<td>1110011</td>
</tr>
<tr>
<td>9</td>
<td>1101111</td>
<td>11001101</td>
</tr>
<tr>
<td>10</td>
<td>11011101</td>
<td>111001101</td>
</tr>
</tbody>
</table>

State Machine View

- Example: \( k=3 \), rate ½ convolutional code
- States labeled with \( x[n-1] \) \( x[n-2] \)
- Arcs labeled with \( x[n] / p_0p_1 \)
- \( msg=101100 \); \( xmit = 11 11 01 00 01 10 \)

State Machines & Trellises

- Example: \( k=3 \), rate ½ convolutional code
  - \( G_0 = 111 \): \( p_0 = 1^*x[n] \oplus 1^*x[n-1] \oplus 1^*x[n-2] \)
  - \( G_1 = 110 \): \( p_1 = 1^*x[n] \oplus 1^*x[n-1] \oplus 0^*x[n-2] \)
- States labeled with \( x[n-1] / x[n-2] \)
- Arcs labeled with \( x[n] / p_0p_1 \)

Addition mod 2 aka XOR
Trellis View @ Transmitter

Using Convolutional Codes

- **Transmitter**
  - Beginning at starting state, processes message bit-by-bit
  - For each message bit: makes a state transition, sends \( p_i \)
  - Pad message with \( k-1 \) zeros to ensure return to starting state

- **Receiver**
  - Doesn’t have direct knowledge of transmitter’s state transitions; only knows (possibly corrupted) received \( p_i \)
  - Must find most likely sequence of transmitter states that could have generated the received \( p_i \)
  - If BER is small, prob(more errors) < prob(fewer errors)
    - Most likely message sequence is the one that generated the sequence of parity bits with the smallest Hamming distance from the actual received \( p_i \), i.e., where we minimize the number of bit errors that explains how the transmit sequence was corrupted to produce the received \( p_i \)

---

**Example**

- Using \( k=3 \), rate \( \frac{1}{2} \) code from earlier slides
- Received: \( 11101000110 \)
- Some errors have occurred...
- What’s the 4-bit message?
- Look for message whose xmit bits are closest to rcvd bits

<table>
<thead>
<tr>
<th>Msg</th>
<th>Xmit*</th>
<th>Rcvd</th>
<th>( d )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>000000000000</td>
<td>000000000000</td>
<td>7</td>
</tr>
<tr>
<td>0001</td>
<td>000000111110</td>
<td>000000111110</td>
<td>8</td>
</tr>
<tr>
<td>0010</td>
<td>000011111000</td>
<td>000011111100</td>
<td>8</td>
</tr>
<tr>
<td>0011</td>
<td>000011010110</td>
<td>000011010110</td>
<td>4</td>
</tr>
<tr>
<td>0100</td>
<td>001111100000</td>
<td>001111100000</td>
<td>6</td>
</tr>
<tr>
<td>0101</td>
<td>001111101110</td>
<td>001111101110</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>001101001000</td>
<td>001101001000</td>
<td>7</td>
</tr>
<tr>
<td>0111</td>
<td>001100100110</td>
<td>001100100110</td>
<td>6</td>
</tr>
<tr>
<td>1000</td>
<td>111110000000</td>
<td>111110000000</td>
<td>4</td>
</tr>
<tr>
<td>1001</td>
<td>111110111110</td>
<td>111110111110</td>
<td>5</td>
</tr>
<tr>
<td>1010</td>
<td>111101111000</td>
<td>111101111000</td>
<td>7</td>
</tr>
<tr>
<td>1011</td>
<td>111101001110</td>
<td>111101000110</td>
<td>2</td>
</tr>
<tr>
<td>1100</td>
<td>111101000110</td>
<td>111101000110</td>
<td>4</td>
</tr>
<tr>
<td>1101</td>
<td>111101011110</td>
<td>111101010110</td>
<td>5</td>
</tr>
<tr>
<td>1110</td>
<td>111101010110</td>
<td>111101010110</td>
<td>6</td>
</tr>
<tr>
<td>1111</td>
<td>111101010011</td>
<td>111101010011</td>
<td>3</td>
</tr>
</tbody>
</table>

*Msg padded with 2 zeroes before xmit

Most likely: 1011

---

Finding the Most-likely Path

Given the received parity bits, the receiver must find the most-likely sequence of transmitter states, i.e., the path through the trellis that minimizes the Hamming distance between the received parity bits and the parity bits the transmitter would have sent had it followed that state sequence.