Convolutional Codes

- 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

Convolutional Codes (cont’d.)

- 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 \( \Rightarrow \text{rate} = \frac{1}{2} \)
  - 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 \( n<0 \):
  - \( p_0[0] = (1 + 0 + 0) \mod 2 = 1, \quad p_1[0] = (1 + 0) \mod 2 = 1 \)
  - \( p_0[1] = (0 + 1 + 0) \mod 2 = 1, \quad p_1[1] = (0 + 1) \mod 2 = 1 \)
  - \( p_0[2] = (1 + 0 + 1) \mod 2 = 0, \quad p_1[2] = (1 + 0) \mod 2 = 1 \)
  - \( p_0[3] = (1 + 1 + 0) \mod 2 = 0, \quad p_1[3] = (1 + 1) \mod 2 = 0 \)

- Transmit: 1, 1, 1, 0, 1, 0, 0, ...

“Good” generator polynomials

Table 1-Generator Polynomials found by Busgang for good rate 1/2 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>111011</td>
</tr>
<tr>
<td>8</td>
<td>110111</td>
<td>11100111</td>
</tr>
<tr>
<td>9</td>
<td>1101111</td>
<td>111001101</td>
</tr>
<tr>
<td>10</td>
<td>110111001</td>
<td>1110011001</td>
</tr>
</tbody>
</table>

www.complexoreal.com

Block diagram view

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

```
\[
\begin{array}{c}
+ \mod 2 \\
\downarrow \\
x[n-1] & x[n-2] \\
\downarrow \\
+ \mod 2 \\
\downarrow \\
p_1[n] \\
\end{array}
\begin{array}{c}
p_0[n] \\
\end{array}
\]
```

- 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.

Example reprised

```
\begin{array}{c}
1 \\
\downarrow \\
0 & 0 \\
\downarrow \\
1 & 1 \\
\downarrow \\
1 & 1 \\
\end{array}
\begin{array}{c}
1 \\
\downarrow \\
0 & 1 \\
\downarrow \\
1 & 0 \\
\downarrow \\
1 & 0 \\
\end{array}
```

Processing \( x[0] \)

Processing \( x[1] \)

Processing \( x[2] \)

Processing \( x[3] \)
State Machine View

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

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, \( \text{prob(more errors)} < \text{prob(fewer errors)} \)
    - 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 \( p_i \)

“Most likely”

Putting aside convolutional codes for a minute...

- Suppose BER = .001 and you receive 1101001
- Is it more likely the transmitter sent 1100111 or 1100001?
- \( p(\text{rcvd 1101001 | xmit 1100111}) = p(\text{ok},\text{ok},\text{ok},\text{err},\text{err},\text{err},\text{ok}) \)
  - \( = (.999)(.999)(.999)(.001)(.001)(.001)(.999) \)
  - \( = 9.9e-10 \)
- \( p(\text{rcvd 1101001 | xmit 1100001}) = p(\text{ok},\text{ok},\text{ok},\text{err},\text{ok},\text{ok},\text{ok}) \)
  - \( = (.999)(.999)(.999)(.001)(.999)(.999)(.999) \)
  - \( = 9.9e-4 \)
- Which is the most likely transmit sequence?
  - Doh! The sequence that corresponds to the event with the greatest probability of occurrence, i.e., xmit = 1100001.
Example

- Using k=3, rate 1/2 code from earlier slides
- Received: 111011000110
- 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>111011000110</td>
<td>7</td>
</tr>
<tr>
<td>0001</td>
<td>000001111110</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>000011111100</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>000111111010</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>001111110000</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>001111110111</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>001110100100</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>001100010011</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>111110000000</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>111110111110</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>111101111000</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>111101000110</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>110001100000</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>110001011110</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>110010011000</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>110010100110</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Most likely: 1011

*Msg padded with 2 zeroes before xmit

Code performance

Source: Butman, Deutsch, Miller, “Performance of Concatenated Codes for Deep Space Missions”