Today

1. Encoding data using convolutional codes
   – Encoder state
   – Changing code rate: Puncturing

2. Decoding convolutional codes: Viterbi Algorithm
Convolutional Encoding

- Don’t send message bits, send only parity bits
- Use a **sliding window** to select which message bits may participate in the parity calculations

![Diagram of message bits and sliding window]
Sliding Parity Bit Calculation

\[ K = 4 \]

-3 -2 -1 0 1 2 3 4 5 6 7 8 ..... 

\[ P[0] = 0 \]

Output: 0
Sliding Parity Bit Calculation

\[ K = 4 \]

\[
\begin{array}{cccccccccccc}
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
\end{array}
\]

\[ + \]

\[ P[1] = 1 \]

Output: 01
Sliding Parity Bit Calculation

Output: 010
Sliding Parity Bit Calculation

-3 -2 -1 0 1 2 3 4 5 6 7 8 ..... 

\[ K = 4 \]

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
\end{array}
\]

\[ P[3] = 0 \]

• Output: 0100
Multiple Parity Bits

Output: 11
Multiple Parity Bits

• Output: ….1110
Multiple Parity Bits

- Output: \ldots111001

- $P_1[5] = 0$
- $P_2[5] = 1$
Encoder State

• **Input bit** and **K-1 bits of current state** determine state on next clock cycle
  – Number of states: \(2^{K-1}\)
Constraint Length

- $K$ is the constraint length of the code

- Larger $K$:
  - Greater redundancy
  - Better error correction possibilities (usually, not always)
Transmitting Parity Bits

- Transmit the parity sequences, not the message itself
  - Each message bit is “spread across” K bits of the output parity bit sequence

- If using multiple generators, *interleave* the bits of each generator
  - *e.g.* (two generators):
    
    \[
    p_0[0], p_1[0], p_0[1], p_1[1], p_0[2], p_1[2]
    \]
Transmitting Parity Bits

- **Code rate** is $1 / \# \text{of generators}$
  - e.g., 2 generators $\rightarrow$ rate = $\frac{1}{2}$

- **Engineering tradeoff:**
  - More generators **improves bit-error correction**
    - But **decreases rate of the code** (the number of message bits/s that can be transmitted)
Shift Register View

- One message bit $x[n]$ in, two parity bits out
  - Each timestep, saved message bits are shifted right by one, the incoming bit moves into the left position

The values in the registers define the state of the encoder
Today

1. Encoding data using convolutional codes
   - Encoder state machine
   - Changing code rate: Puncturing

2. Decoding convolutional codes: Viterbi Algorithm
Example: $K = 3$, code rate $\frac{1}{2}$, convolutional code

- There are $2^{K-1}$ states
- States labeled with $(x[n-1], x[n-2])$
- Arcs labeled with $x[n]/p_0[n]p_1[n]$
- Generator: $g_0 = 111$, $g_1 = 101$
- $msg = 101100$
State-Machine View

- Starting state
  - \( P_0[n] = (1 \cdot x[n] + 1 \cdot x[n-1] + 1 \cdot x[n-2]) \mod 2 \)
  - \( P_1[n] = (1 \cdot x[n] + 0 \cdot x[n-1] + 1 \cdot x[n-2]) \mod 2 \)
  - Generators: \( g_0 = 111, g_1 = 101 \)

- \( \text{msg} = 101100 \)
- Transmit:
State-Machine View

- \( P_0[n] = 1*1 + 1*0 + 1*0 \mod 2 \)
- \( P_1[n] = 1*1 + 0*0 + 1*0 \mod 2 \)
- Generators: \( g_0 = 111, g_1 = 101 \)

**msg** = 101100

**Transmit**: 11
State-Machine View

- Starting state
- $P_0[n] = 1*0 + 1*1 + 1*0 \mod 2$
- $P_1[n] = 1*0 + 0*1 + 1*0 \mod 2$
- Generators: $g_0 = 111$, $g_1 = 101$

- $msg = 101100$
- Transmit: $11 \ 10$
State-Machine View

- Starting state: 11

- $P_0[n] = 1*1 + 1*0 + 1*1 \mod 2$
- $P_1[n] = 1*1 + 0*0 + 1*1 \mod 2$
- Generators: $g_0 = 111$, $g_1 = 101$

- msg = 101100
- Transmit: 11 10 00
State-Machine View

- Starting state
  - $P_0[n] = 1*1 + 1*1 + 1*0$
  - $P_1[n] = 1*1 + 0*1 + 1*0$
  - Generators: $g_0 = 111$, $g_1 = 101$

- msg = 101100
- Transmit: 11 10 00 01
State-Machine View

- Starting state

- $P_0[n] = 1*0 + 1*1 + 1*1$
- $P_1[n] = 1*0 + 0*1 + 1*1$
- Generators: $g_0 = 111, g_1 = 101$

- $msg = 101100$
- Transmit: 11 10 00 01 01
State-Machine View

Starting state

- $P_0[n] = 1*0 + 1*0 + 1*1$
- $P_1[n] = 1*0 + 0*0 + 1*1$
- Generators: $g_0 = 111, g_1 = 101$

- $\text{msg} = 101100$
- Transmit: 11 10 00 01 01 11
Today

1. Encoding data using convolutional codes
   – Encoder state machine
   – Changing code rate: Puncturing

2. Decoding convolutional codes: Viterbi Algorithm
Varying the Code Rate

• How to increase/decrease rate?

• Transmitter and receiver agree on coded bits to **omit**
  – *Puncturing table* indicates which bits to include (1)
    • Contains $p$ columns, $N$ rows

![Diagram showing data, convolutional coder, coded bits, puncturing, and 2/3 and 3/4 codes](diagram.png)
Punctured convolutional codes: example

- Coded bits =

0 0 1 0 1
0 0 1 1 1

- With Puncturing:

\[ P_1 = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \]
Punctured convolutional codes: example

- Coded bits =

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- With Puncturing:

\[ P_1 = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \]

3 out of 4 bits are used

2 out of 4 bits are used
Punctured convolutional codes: example

- Coded bits =

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- With Puncturing:

$$P_1 = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

- Punctured, coded bits:

```
0
0
```
Punctured convolutional codes: example

- Coded bits =

- With Puncturing:

\[ P_1 = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \]

- Punctured, coded bits:
Punctured convolutional codes: example

• Coded bits =

0 0 1 0 1
0 0 1 1 1

• With Puncturing:

\[ P_1 = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \]

• Punctured, coded bits:
Punctured convolutional codes: example

- Coded bits =

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

- With Puncturing:

```
  P_1 = \begin{pmatrix}
    1 & 1 & 1 & 0 \\
    1 & 0 & 0 & 1
  \end{pmatrix}
```

- Punctured, coded bits:

```
  0 0 1 0 1
  0 1 1
```
Punctured convolutional codes: example

• Coded bits =

0 0 1 0 1
0 0 1 1 1

• With Puncturing:

\[ P_1 = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \]

• Punctured, coded bits:

0 0 1 1
0 1 1 1
Punctured convolutional codes: example

- Coded bits =

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

- Punctured, coded bits:

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

- Punctured rate is: $R = \frac{(1/2)}{(5/8)} = \frac{4}{5}$
1. Encoding data using convolutional codes
   – Changing code rate: Puncturing

2. Decoding convolutional codes: Viterbi Algorithm
   – Hard decision decoding
   – Soft decision decoding
**Motivation: The Decoding Problem**

- Received bits: **000101100110**
- Some errors have occurred
- *What’s the 4-bit message?*
- **Most likely: 0111**
  - Message whose codeword is *closest to received bits* in Hamming distance

<table>
<thead>
<tr>
<th>Message</th>
<th>Coded bits</th>
<th>Hamming distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>00000000000000</td>
<td>5</td>
</tr>
<tr>
<td>0001</td>
<td>00000001110111</td>
<td>--</td>
</tr>
<tr>
<td>0010</td>
<td>00001110110000</td>
<td>--</td>
</tr>
<tr>
<td>0011</td>
<td>00001101011100</td>
<td>--</td>
</tr>
<tr>
<td>0100</td>
<td>00111011000000</td>
<td>--</td>
</tr>
<tr>
<td>0101</td>
<td>00111000101100</td>
<td>--</td>
</tr>
<tr>
<td>0110</td>
<td>00110101110000</td>
<td>--</td>
</tr>
<tr>
<td>0111</td>
<td>00110110011100</td>
<td>2</td>
</tr>
<tr>
<td>1000</td>
<td>11011000000000</td>
<td>--</td>
</tr>
<tr>
<td>1001</td>
<td>11011111110111</td>
<td>--</td>
</tr>
<tr>
<td>1010</td>
<td>11100010110000</td>
<td>--</td>
</tr>
<tr>
<td>1011</td>
<td>11100001011100</td>
<td>--</td>
</tr>
<tr>
<td>1100</td>
<td>11010111000000</td>
<td>--</td>
</tr>
<tr>
<td>1101</td>
<td>11010100101100</td>
<td>--</td>
</tr>
<tr>
<td>1110</td>
<td>11011001110000</td>
<td>--</td>
</tr>
<tr>
<td>1111</td>
<td>11011010011100</td>
<td>--</td>
</tr>
</tbody>
</table>
The Trellis

- Vertically, lists encoder states
- Horizontally, tracks time steps
- Branches connect states in successive time steps

Trellis:

- States: 00, 01, 10, 11
- Time: x[n-1], x[n-2]
- Branches connect states in successive time steps
The Trellis: Sender’s View

- At the sender, transmitted bits trace a unique, single path of branches through the trellis – e.g. transmitted data bits 1 0 1 1

- Recover transmitted bits ↔ Recover path
Viterbi algorithm

- Andrew Viterbi (USC)
- **Want**: Most likely sent bit sequence
- Calculates *most likely path* through *trellis*

1. **Hard Decision Viterbi algorithm**: Have *possibly-corrupted* encoded *bits*, after reception

2. **Soft Decision Viterbi algorithm**: Have *possibly-corrupted likelihoods* of each bit, after reception
   - e.g.: “this bit is 90% likely to be a 1.”
Viterbi algorithm: Summary

• **Branch metrics** score **likelihood of each trellis branch**

• At any given time there are $2^{K-1}$ **most likely messages** we’re tracking (one for each state)
  – One message $\leftrightarrow$ one trellis path
  – **Path metrics** score **likelihood of each trellis path**

• **Most likely message** is the one that produces the **smallest** path metric
Today

1. Encoding data using convolutional codes
   – Changing code rate: Puncturing

2. Decoding convolutional codes: Viterbi Algorithm
   – Hard decision decoding
   – Soft decision decoding
Hard-decision branch metric

- Hard decisions $\rightarrow$ input is bits

- **Label every branch** of trellis with branch metrics
  - **Hard Decision Branch metric**: Hamming Distance between received and transmitted bits

![Trellis Diagram]

- Received: 00
  - States: 00, 01, 10, 11
  - Branch metrics:
    - 0/00 $\rightarrow$ 0
    - 1/00 $\rightarrow$ 0
    - 0/01 $\rightarrow$ 1
    - 1/01 $\rightarrow$ 1
    - 1/10 $\rightarrow$ 1
    - 0/11 $\rightarrow$ 2
    - 1/11 $\rightarrow$ 2

42
Hard-decision branch metric

• Suppose we know encoder is in state 00, receive bits: 00
Hard-decision path metric

- **Hard-decision path metric:** Sum Hamming distance between *sent* and *received bits* along path

- Encoder is initially in *state 00*, *receive bits*: 00

Received: 00

```
  0 0  0/00 → 0
  0 1  1/11 → 2
  1 0
  1 1
```
Hard-decision path metric

- Right now, each state has a unique predecessor state

- Path metric: Total bit errors along path ending at state
  - Path metric of predecessor + branch metric

Received: 00 11

```
0 0 0/00 \rightarrow 0
0 1 1/11 \rightarrow 2
1 0 0/10 \rightarrow 1
1 1 1/01 \rightarrow 1
```
Hard-decision path metric

- Each state has **two predecessor states, two predecessor paths** (which to use?)

- **Winning** branch has **lower** path metric (**fewer** bit errors): **Prune** **losing** branch

```
Received: 00 11 01
```

![Diagram showing state transitions and path metrics with arrows indicating transitions and path metrics for each state transition.]
Hard-decision path metric

- Prune losing branch for each state in trellis
Pruning non-surviving branches

- **Survivor path** begins at each state, traces unique path back to **beginning** of trellis
  - **Correct path** is one of **four** survivor paths

- Some branches are not part of any survivor: **prune them**

```
Received: 00 11 01
```

![Trellis diagram](image-url)
Making bit decisions

- When **only one branch remains** at a stage, the Viterbi algorithm **decides** that branch’s **input bits**: 

  ```
  Received:   00   11   01  
  Decide:     0     
  ```
End of received data

- Trace back the survivor with **minimal path metric**

- Later stages *don’t get benefit* of future error correction, had data not ended

<table>
<thead>
<tr>
<th>Received:</th>
<th>00</th>
<th>11</th>
<th>01</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decide:</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

```
0/00 \rightarrow 2
0/00 \rightarrow 1
1/11 \rightarrow 0
1/11 \rightarrow 1
0/10 \rightarrow 2
0/10 \rightarrow 1
1/01 \rightarrow 0
1/01 \rightarrow 1
0/01 \rightarrow 1
0/01 \rightarrow 2
1/10 \rightarrow 0
1/10 \rightarrow 1
```
Terminating the code

- **Sender** transmits **two 0 data bits** at end of data

- **Receiver** uses the following trellis at end:

- **After termination only one trellis survivor path** remains
  - Can **make better bit decisions at end of data** based on this sole survivor
Viterbi with a Punctured Code

• Punctured bits are never transmitted

• Branch metric measures dissimilarity only between received and transmitted unpunctured bits
  – Same path metric, same Viterbi algorithm
  – Lose some error correction capability

Received: 0-

States

\[
\begin{align*}
00 & \rightarrow 0/00 \rightarrow 0 \\
01 & \rightarrow 1/11 \rightarrow 1 \\
10 & \rightarrow 1/00 \rightarrow 0 \\
11 & \rightarrow 0/01 \rightarrow 0 \\
& \rightarrow 1/10 \rightarrow 1 \\
\end{align*}
\]
Today

1. Encoding data using convolutional codes
   – Changing code rate: Puncturing

2. Decoding convolutional codes: Viterbi Algorithm
   – Hard decision decoding
     • Error correcting capability
   – Soft decision decoding
How many bit errors can we correct?

- Think back to the encoder; **linearity property:**
  - Message $m_1 \rightarrow$ Coded bits $c_1$
  - Message $m_2 \rightarrow$ Coded bits $c_2$
  - Message $m_1 \oplus m_2 \rightarrow$ Coded bits $c_1 \oplus c_2$

- So, $d_{\text{min}} = \text{minimum distance between 000...000 codeword and codeword with fewest 1s}$
Calculating $d_{\text{min}}$ for the convolutional code

- Find path with **smallest non-zero path metric** going from **first 00 state** to a **future 00 state**

- Here, $d_{\text{min}} = 4$, so can correct **1 error in 8 bits**:

  \[
  x[n] \begin{array}{cccccccc}
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  \end{array}
  \]

  \[
  \begin{array}{cccccccc}
  00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\
  01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 \\
  10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\
  11 & 11 & 11 & 11 & 11 & 11 & 11 & 11 \\
  \end{array}
  \]

  \[
  x[n-1]x[n-2] \]

  \[
  \begin{array}{cccccccc}
  00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\
  01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 \\
  10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\
  11 & 11 & 11 & 11 & 11 & 11 & 11 & 11 \\
  \end{array}
  \]
Today

1. Encoding data using convolutional codes
   – Changing code rate: Puncturing

2. Decoding convolutional codes: Viterbi Algorithm
   – Hard decision decoding
   – Soft decision decoding
Model for Today

- **Coded bits** are actually **continuously-valued “voltages”** between **0.0 V and 1.0 V**:

  - Strong “1”
  - Weak “1”
  - Weak “0”
  - Strong “0”
On Hard Decisions

- Hard decisions digitize each voltage to “0” or “1” by comparison against \textit{threshold voltage 0.5 V}
  - Lose information about how “good” the bit is
    - Strong “1” (0.99 V) \textit{treated equally to} weak “1” (0.51 V)

- \textbf{Hamming distance} for branch metric computation

- But \textbf{throwing away information} is almost never a good idea when making decisions
  - \textit{Find a better branch metric that retains information about the received voltages?}
Soft-decision decoding

- Idea: **Pass received voltages to decoder before digitizing**
  - **Problem:** Hard branch metric was Hamming distance

- **“Soft” branch metric**
  - **Euclidian distance** between received voltages and voltages of expected bits:

```
(0.0, 1.0)  1.0, 1.0
0.0, 0.0    1.0, 0.0
```

"Soft" metric

Expected parity bits:
(0, 1)
Soft-decision decoding

- **Different** branch metric, hence **different** path metric
- **Same** path metric computation
- **Same** Viterbi algorithm

- **Result:** Choose **path** that minimizes sum of squares of **Euclidean distances** between received, expected voltages
Putting it together: Convolutional coding in Wi-Fi

Data bits

Convolutional encoder

Coded bits

Modulation (BPSK, QPSK, …)

Viterbi Decoder

Coded bits (hard-decision decoding) or Voltage Levels (soft-decision decoding)

Demodulation

Data bits
Thursday Topic: Rateless Codes
Friday Precept: Midterm Review