Convolutional Codes

Introduction

In the last lecture we looked at block codes where every block of *k* bits of data is coded into a block of *n* bits of code. Rather than break the data into discrete blocks, convolutional codes produce a more continuous output. The subject is immense, and we will only touch on the simplest types of such codes. However, this will illustrate the basic concepts. And, in fact, the convolutional codes used in digital wireless communication are typically of this simple type.

Encoding

We will consider convolutional codes that produce *n* bits of output for each 1 bit of new data. These therefore have a code rate $R = 1/n$. Rates 1/2 and 1/3, for example, find application in wireless systems. An example rate 1/2 coder is shown in Fig. 29.1.

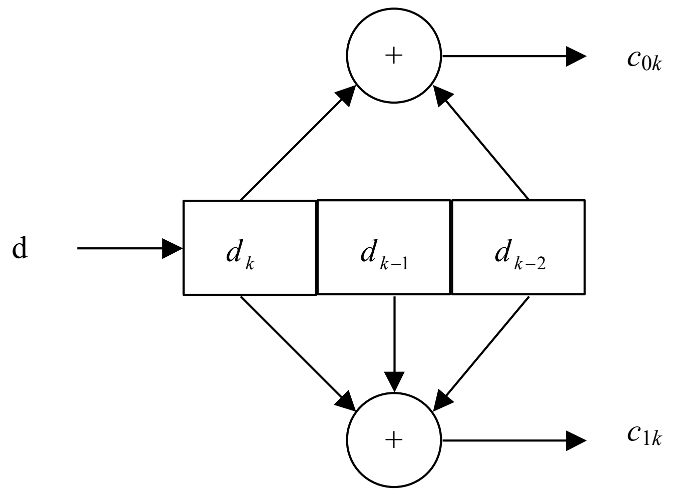

Figure 29.1: An example rate 1/2 convolutional coder. The blocks are shift registers and the circles are modulo 2 adders (xor gates).

The data bits d_k, with *k* denoting time, are fed into a shift register, that is, a binary sequential memory. In this example we remember the last two data bits. The number of available data bits, in this example 3: d_k, d_{k-1}, d_{k-2}, is called the *constraint length* of the coder. The number of bits in memory, 2 in this case: d_{k-1}, d_{k-2}, is called the *memory "M"* of the code. At each step we perform *n* binary operations on these bits to produce *n* code bits. In the example shown, we produce two code bits via the operations

$$\begin{aligned} c_{0k} &= d_k \oplus d_{k-2} \\ c_{1k} &= d_k \oplus d_{k-1} \oplus d_{k-2} \end{aligned} \tag{29.1}$$

The modulo 2 addition operator "$\oplus$" is equivalent to an "exclusive or" (xor) logic operation, i.e., $0\oplus 0=0$, $1\oplus 1=0$, $0\oplus 1=1\oplus 0=1$. We then stack the two code bits into our output for transmission.

Designing a convolutional code of this type involves choosing the memory length, *M*, the number of logic operations per input bit, *n*, and determining the "connections" for the xor operations. How do you figure these things out? That gets you more deeply into the subject than we wish to go.

If the data sequence contains a finite number of bits, say *L*, then $nL+nM$ bits will be output. The nM term comes because the last data bit will go through *M* shifts before all data is cleared out of the coder. During this we shift 0's into the coder. The result is a block code of rate

$$\begin{aligned} R_L &= \frac{L}{nL+ML} \\ &= R\left(1-\frac{M}{L+M}\right) \end{aligned} \tag{29.2}$$

This approaches $R=1/n$ if $L>>M$. So, when used on a finite block of data bits, a convolutional coder of this type is effective a way to implement a certain block code. However, the data stream could be infinite in which case there is not a direct correspondence to block coding.

For example, to code the 4-bit data sequence [1101] using the convolutional coder of Fig. 29.1, we will get $2\cdot 4+2\cdot 2=12$ bits of code since $L=4, M=2$. The following Matlab code gives the output [11 10 10 00 01 11].

```
d = [ 1 1 0 1 ];
d = [ 0 0 d 0 0];
N = length(d);
c = [];
for i=1:N-2
   c0 = xor( d(i), d(i+2) );
   c1 = xor( c0, d(i+1) );
   c = [ c c0 c1 ];
end
c
```

You can consider the two "memory" bits d_{k-1} and d_{k-2} as the "state" of the coder. With the addition of a new bit d_k the output of the coder is defined by the logic operations, and on the next clock cycle we move to a new state. This is represented in the following *state table* for the coder of Fig. 29.1. For different coders you'd have different tables.

state	input	new state	output
00	0	00	00
00	1	10	11
01	0	00	11
01	1	10	00
10	0	01	01
10	1	11	10
11	0	01	10
11	1	11	01

Notice that for any state, the two possible outputs are separated by Hamming distance 2. For example, for state 10, the possible outputs are 01 and 10. A single bit error in either of these states cannot transform it into the other state. This gives you leverage when trying to decode in the presence of bit errors, as we'll see.

Decoding

The basic idea for decoding a convolutional code is to find the input data stream that gives a code output as close as possible, in the Hamming distance sense, to the received bits. For example, suppose we are using the coder of Fig. 29.1 and we received the following bits [11 10 10 00 01 11]. We know that the coder starts off in the state 00. What data bit gives code output 11 for state 00? From the table we see that it must have been 1 and that this would move us to the new state 10. Then we ask, what data gives us code output 10 for state 10? From the table we see that it must have been 1, and so on. In this manner we reconstruct the data bits [1101].

A graphical way to represent this is by a *trellis diagram* as shown in Fig. 29.2. We list the possible states of the coder, 00, 01, 10, and 11, at left from top to bottom. For each time step *k*, we draw a column of four circles corresponding to the four possible coder states. We start at state 00 and end up at state 00. In going from time step *k* to $k+1$ we draw a line between the two coder states we find ourselves in. In the Fig. 29.1 coder, there are four states and each state can branch to one of two states. In this manner we can map out all possible sequences of state transitions. The four data bits specify the paths through the trellis diagram. Therefore, there are $2^4 = 16$ paths.

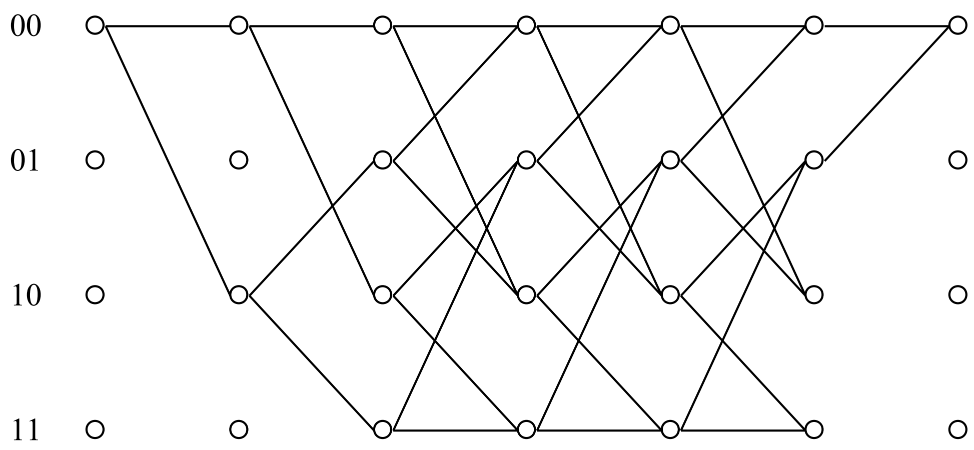

Figure 29.2: Trellis diagram for coder of Fig. 29.1 and 4 bits of data. All possible paths specified by the state table are shown.

For example, starting at state 00, the data sequence 1101 takes us along the path shown in Fig. 29.3 to get the output code [11 10 10 00 01 11]. Decoding can be thought of as find the path through the trellis that corresponds most closely to the received bits. If there are no bit errors in the received bits then there is one and only one answer.

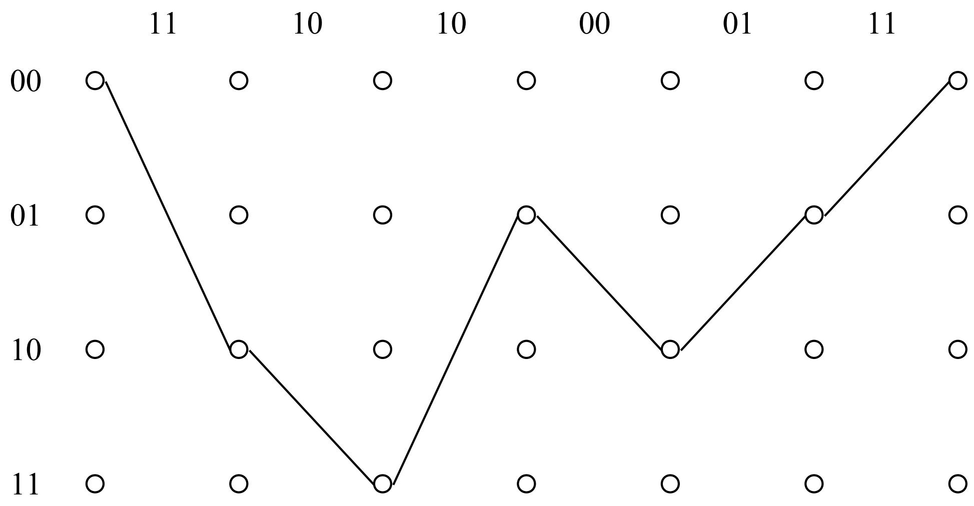

Figure 29.3: Example path through the trellis diagram corresponding to data bits [1101]. Output bits of code are shown at top.

On the other hand, if some of the received bits are in error then we have to examine different possible paths. As an example, suppose the transmitted code [11 10 10 00 01 11] gets corrupted and we receive [11 10 10 01 01 11]. We start at state 00 and note that code bits 11 correspond to a transition to state 10. We continue in this manner until we get to the point labeled "error" in Fig. 29.4.

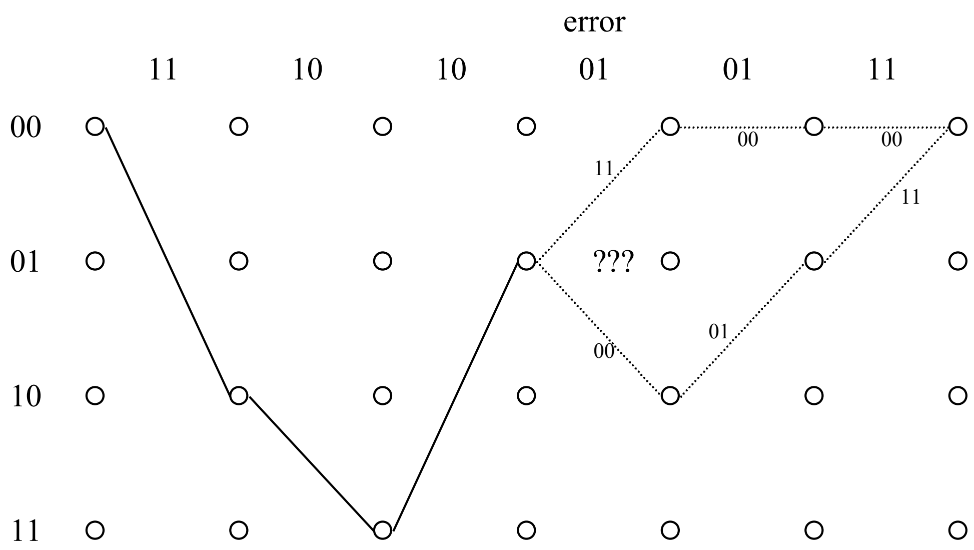

Figure 29.4: When there is an error in the received bits, we have to examine the possible paths through the trellis and choose the one that best matches the received data.

At this point we are in state 01. From the state table we see that no transition from state 01 gives output 01. So the received bits 01 must be in error. They could have originally been 11, which would imply a transition to state 00, or they could have been 00, which would imply a transition to state 10. We can follow both of these possibilities to the final state 00. The dotted paths in Fig. 29.4 show this. At each transition along the dotted paths we show the code bits that would have been output. The upper dotted path would have produced output $[11\ 00\ 00]$ while the lower path would have produced $[00\ 01\ 11]$. Comparing to the received bits $[01\ 01\ 11]$ it is clear that the lower path has a smaller Hamming distance from received data, so we therefore assume that is correct path (which is correct) and so recover the correct data bits from the state table.

References

1. McEliece, R. J., *The Theory of Information and Coding*, Cambridge University Press, 1977. ISBN 0-521-30223-4.
2. Burr, A., *Modulation and Coding for Wireless Communications*, Prentice-Hall, 2001, ISBN 0-201-39857-5.