MIT 6.02 **DRAFT** *Lecture Notes*
Last update: February 24, 2012
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 7
Convolutional Codes: Construction and Encoding

This chapter introduces a widely used class of codes, called **convolutional codes**, which are used in a variety of systems including today's popular wireless standards (such as 802.11) and in satellite communications. They are also used as a building block in more powerful modern codes, such as turbo codes, which are used in wide-area cellular wireless network standards such as 3G, LTE, and 4G. Convolutional codes are beautiful because they are intuitive, one can understand them in many different ways, and there is a way to decode them so as to recover the *most likely* message from among the set of all possible transmitted messages. This chapter discusses the encoding of convolutional codes; the next one discusses how to decode convolutional codes efficiently.

Like the block codes discussed in the previous chapter, convolutional codes involve the computation of parity bits from message bits and their transmission, and they are also linear codes. Unlike block codes in systematic form, however, the sender does not send the message bits followed by (or interspersed with) the parity bits; in a convolutional code, the sender *sends only the parity bits*. These codes were invented by Peter Elias '44, an MIT EECS faculty member, in the mid-1950s. For several years, it was not known just how powerful these codes are and how best to decode them. The answers to these questions started emerging in the 1960s, with the work of people like Andrew Viterbi '57, G. David Forney (SM '65, Sc.D. '67, and MIT EECS faculty member), Jim Omura SB '63, and many others.

■ 7.1 Convolutional Code Construction

The encoder uses a *sliding window* to calculate $r > 1$ parity bits by combining various subsets of bits in the window. The combining is a simple addition in $\mathbb{F}_2$, as in the previous chapter (i.e., modulo 2 addition, or equivalently, an exclusive-or operation). Unlike a block code, however, the windows overlap and slide by 1, as shown in Figure 7-1. The size of the window, in bits, is called the code's **constraint length**. The longer the constraint length, the larger the number of parity bits that are influenced by any given message bit. Because the parity bits are the only bits sent over the channel, a larger constraint length generally

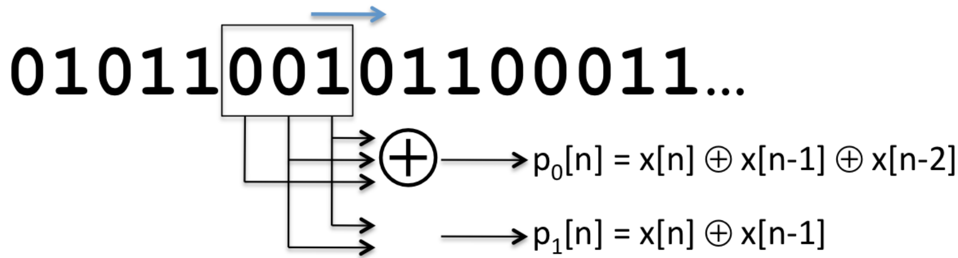

Figure 7-1: An example of a convolutional code with two parity bits per message bit and a constraint length (shown in the rectangular window) of three. I.e., $r = 2, K = 3$.

implies a greater resilience to bit errors. The trade-off, though, is that it will take considerably longer to decode codes of long constraint length (we will see in the next chapter that the complexity of decoding is exponential in the constraint length), so one cannot increase the constraint length arbitrarily and expect fast decoding.

If a convolutional code produces r parity bits per window and slides the window forward by one bit at a time, its rate (when calculated over long messages) is $1/r$. The greater the value of r, the higher the resilience of bit errors, but the trade-off is that a proportionally higher amount of communication bandwidth is devoted to coding overhead. In practice, we would like to pick r and the constraint length to be as small as possible while providing a low enough resulting probability of a bit error.

In 6.02, we will use K (upper case) to refer to the constraint length, a somewhat unfortunate choice because we have used k (lower case) in previous chapters to refer to the number of message bits that get encoded to produce coded bits. Although "L" might be a better way to refer to the constraint length, we'll use K because many papers and documents in the field use K (in fact, many papers use k in lower case, which is especially confusing). Because we will rarely refer to a "block" of size k while talking about convolutional codes, we hope that this notation won't cause confusion.

Armed with this notation, we can describe the encoding process succinctly. The encoder looks at K bits at a time and produces r parity bits according to carefully chosen functions that operate over various subsets of the K bits.[1] One example is shown in Figure 7-1, which shows a scheme with $K = 3$ and $r = 2$ (the rate of this code, $1/r = 1/2$). The encoder spits out r bits, which are sent sequentially, slides the window by 1 to the right, and then repeats the process. That's essentially it.

At the transmitter, the two princial remaining details that we must describe are:

1. What are good parity functions and how can we represent them conveniently?
2. How can we implement the encoder efficiently?

The rest of this chapter will discuss these issues, and also explain why these codes are called "convolutional".

[1]By convention, we will assume that each message has $K - 1$ "0" bits padded in front, so that the initial conditions work out properly.

7.2 Parity Equations

The example in Figure 7-1 shows one example of a set of *parity equations*, which govern the way in which parity bits are produced from the sequence of message bits, X. In this example, the equations are as follows (all additions are in $\mathbb{F}_2$)):

$$\begin{aligned} p_0[n] &= x[n] + x[n-1] + x[n-2] \\ p_1[n] &= x[n] + x[n-1] \end{aligned} \tag{7.1}$$

The rate of this code is $1/2$.

An example of parity equations for a rate $1/3$ code is

$$\begin{aligned} p_0[n] &= x[n] + x[n-1] + x[n-2] \\ p_1[n] &= x[n] + x[n-1] \\ p_2[n] &= x[n] + x[n-2] \end{aligned} \tag{7.2}$$

In general, one can view each parity equation as being produced by combining the message bits, X, and a **generator polynomial**, g. In the first example above, the generator polynomial coefficients are $(1,1,1)$ and $(1,1,0)$, while in the second, they are $(1,1,1),(1,1,0)$, and $(1,0,1)$.

We denote by g_i the K-element generator polynomial for parity bit p_i. We can then write $p_i[n]$ as follows:

$$p_i[n] = (\sum_{j=0}^{k-1} g_i[j]x[n-j]) \bmod 2. \tag{7.3}$$

The form of the above equation is a *convolution* of g and x—hence the term "convolutional code". The number of generator polynomials is equal to the number of generated parity bits, r, in each sliding window. The rate of the code is $1/r$ if the encoder slides the window one bit at a time.

7.2.1 An Example

Let's consider the two generator polynomials of Equations 7.1 (Figure 7-1). Here, the generator polynomials are

$$\begin{aligned} g_0 &= 1,1,1 \\ g_1 &= 1,1,0 \end{aligned} \tag{7.4}$$

If the message sequence, $X = [1,0,1,1,\ldots]$ (as usual, $x[n] = 0 \ \forall n < 0$), then the parity

bits from Equations 7.1 work out to be

$$\begin{aligned}
p_0[0] &= (1+0+0) = 1 \\
p_1[0] &= (1+0) = 1 \\
p_0[1] &= (0+1+0) = 1 \\
p_1[1] &= (0+1) = 1 \\
p_0[2] &= (1+0+1) = 0 \\
p_1[2] &= (1+0) = 1 \\
p_0[3] &= (1+1+0) = 0 \\
p_1[3] &= (1+1) = 0.
\end{aligned} \quad (7.5)$$

Therefore, the bits transmitted over the channel are $[1,1,1,1,0,0,0,0,\ldots]$.

There are several generator polynomials, but understanding how to construct good ones is outside the scope of 6.02. Some examples (found by J. Busgang) are shown in Table 7-1.

Constraint length	g_0	g_1
3	110	111
4	1101	1110
5	11010	11101
6	110101	111011
7	110101	110101
8	110111	1110011
9	110111	111001101
10	110111001	1110011001

Table 7-1: Examples of generator polynomials for rate $1/2$ convolutional codes with different constraint lengths.

7.3 Two Views of the Convolutional Encoder

We now describe two views of the convolutional encoder, which we will find useful in better understanding convolutional codes and in implementing the encoding and decoding procedures. The first view is in terms of **shift registers**, where one can construct the mechanism using shift registers that are connected together. This view is useful in developing hardware encoders. The second is in terms of a **state machine**, which corresponds to a view of the encoder as a set of states with well-defined transitions between them. The state machine view will turn out to be extremely useful in figuring out how to decode a set of parity bits to reconstruct the original message bits.

7.3.1 Shift-Register View

Figure 7-2 shows the same encoder as Figure 7-1 and Equations (7.1) in the form of a block diagram made up of shift registers. The $x[n-i]$ values (here there are two) are referred to

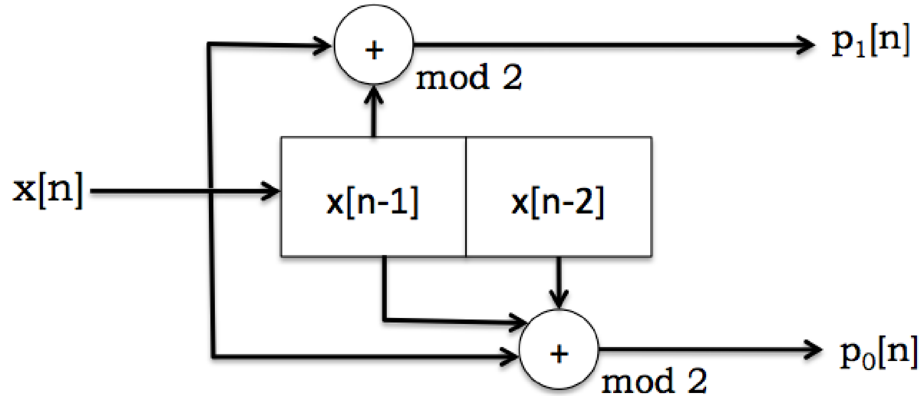

Figure 7-2: Block diagram view of convolutional coding with shift registers.

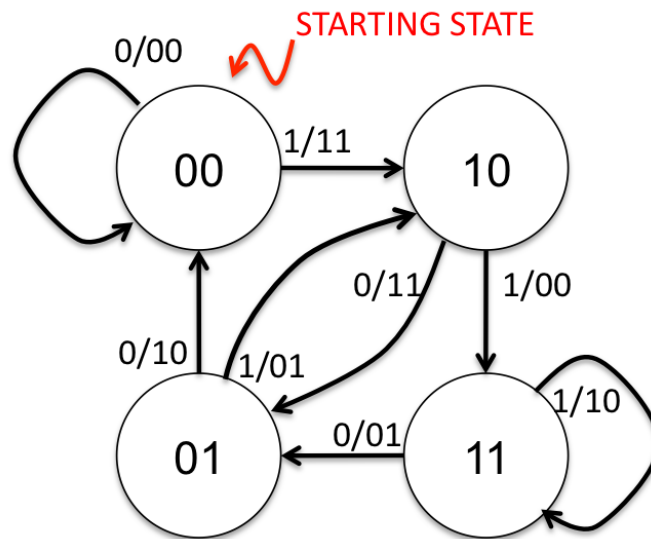

Figure 7-3: State-machine view of convolutional coding.

as the *state* of the encoder. This block diagram takes message bits in one bit at a time, and spits out parity bits (two per input bit, in this case).

Input message bits, $x[n]$, arrive from the left. The block diagram calculates the parity bits using the incoming bits and the state of the encoder (the $k-1$ previous bits; two in this example). After the r parity bits are produced, the state of the encoder shifts by 1, with $x[n]$ taking the place of $x[n-1]$, $x[n-1]$ taking the place of $x[n-2]$, and so on, with $x[n-K+1]$ being discarded. This block diagram is directly amenable to a hardware implementation using shift registers.

■ 7.3.2 State-Machine View

Another useful view of convolutional codes is as a state machine, which is shown in Figure 7-3 for the same example that we have used throughout this chapter (Figure 7-1).

An important point to note: the state machine for a convolutional code is *identical* for

all codes with a given constraint length, K, and the number of states is always 2^{K-1}. Only the p_i labels change depending on the number of generator polynomials and the values of their coefficients. Each state is labeled with $x[n-1]x[n-2]\ldots x[n-K+1]$. Each arc is labeled with $x[n]/p_0p_1\ldots$. In this example, if the message is 101100, the transmitted bits are 11 11 01 00 01 10.

This state-machine view is an elegant way to explain what the transmitter does, and also what the receiver ought to do to decode the message, as we now explain. The transmitter begins in the initial state (labeled "STARTING STATE" in Figure 7-3) and processes the message one bit at a time. For each message bit, it makes the state transition from the current state to the new one depending on the value of the input bit, and sends the parity bits that are on the corresponding arc.

The receiver, of course, does not have direct knowledge of the transmitter's state transitions. It only sees the received sequence of parity bits, with possible bit errors. Its task is to determine the **best possible sequence of transmitter states that could have produced the parity bit sequence**. This task is the essence of the decoding process, which we introduce next, and study in more detail in the next chapter.

■ 7.4 The Decoding Problem

As mentioned above, the receiver should determine the "best possible" sequence of transmitter states. There are many ways of defining "best", but one that is especially appealing is the *most likely* sequence of states (i.e., message bits) that must have been traversed (sent) by the transmitter. A decoder that is able to infer the most likely sequence the *maximum-likelihood* (ML) decoder for the convolutional code.

In Section 6.2, we established that the ML decoder for "hard decoding", in which the distance between the received word and each valid codeword is the Hamming distance, may be found by computing the valid codeword with smallest Hamming distance, and returning the message that would have generated that codeword. The same idea holds for convolutional codes. (Note that this property holds whether the code is either block or convolutional, and whether it is linear or not.)

A simple numerical example may be useful. Suppose that bit errors are independent and identically distributed with an error probability of 0.001 (i.e., the channel is a BSC with $\varepsilon = 0.001$), and that the receiver digitizes a sequence of analog samples into the bits 1101001. Is the sender more likely to have sent 1100111 or 1100001? The first has a Hamming distance of 3, and the probability of receiving that sequence is $(0.999)^4(0.001)^3 = 9.9 \times 10^{-10}$. The second choice has a Hamming distance of 1 and a probability of $(0.999)^6(0.001)^1 = 9.9 \times 10^{-4}$, which is *six orders of magnitude higher* and is overwhelmingly more likely.

Thus, the most likely sequence of parity bits that was transmitted must be the one with the smallest Hamming distance from the sequence of parity bits received. Given a choice of possible transmitted messages, the decoder should pick the one with the smallest such Hamming distance. For example, see Figure 7-4, which shows a convolutional code with $K = 3$ and rate $1/2$. If the receiver gets 111011000110, then some errors have occurred, because no valid transmitted sequence matches the received one. The last column in the example shows d, the Hamming distance to all the possible transmitted sequences, with

Msg	Xmit*	Rcvd	d
0000	000000000000	111011000110	7
0001	000000111110		8
0010	000011111000		8
0011	000011010110		4
0100	001111100000		6
0101	001111011110		5
0110	001101001000		7
0111	001100100110		6
1000	111110000000		4
1001	111110111110		5
1010	111101111000		7
1011	111101000110		2 (Most likely: 1011)
1100	110001100000		5
1101	110001011110		4
1110	110010011000		6
1111	110010100110		3

Figure 7-4: When the probability of bit error is less than 1/2, maximum-likelihood decoding boils down to finding the message whose parity bit sequence, when transmitted, has the smallest Hamming distance to the received sequence. Ties may be broken arbitrarily. Unfortunately, for an N-bit transmit sequence, there are 2^N possibilities, which makes it hugely intractable to simply go through in sequence because of the sheer number. For instance, when $N = 256$ bits (a really small packet), the number of possibilities rivals the number of atoms in the universe!

the smallest one circled. To determine the most-likely 4-bit message that led to the parity sequence received, the receiver could look for the message whose transmitted parity bits have smallest Hamming distance from the received bits. (If there are ties for the smallest, we can break them arbitrarily, because all these possibilities have the same resulting post-coded BER.)

Determining the nearest valid codeword to a received word is easier said than done for convolutional codes. For block codes, we found that comparing against each valid codeword would take time exponential in k, the number of valid codewords for an (n, k) block code. We then showed how syndrome decoding takes advantage of the linearity property to devise an efficient polynomial-time decoder for block codes, whose time complexity was roughly $O(n^t)$, where t is the number of errors that the linear block code can correct.

For convolutional codes, syndrome decoding in the form we described is impossible because n is *infinite* (or at least as long as the number of parity streams times the length of the entire message times, which could be arbitrarily long)! The straightforward approach

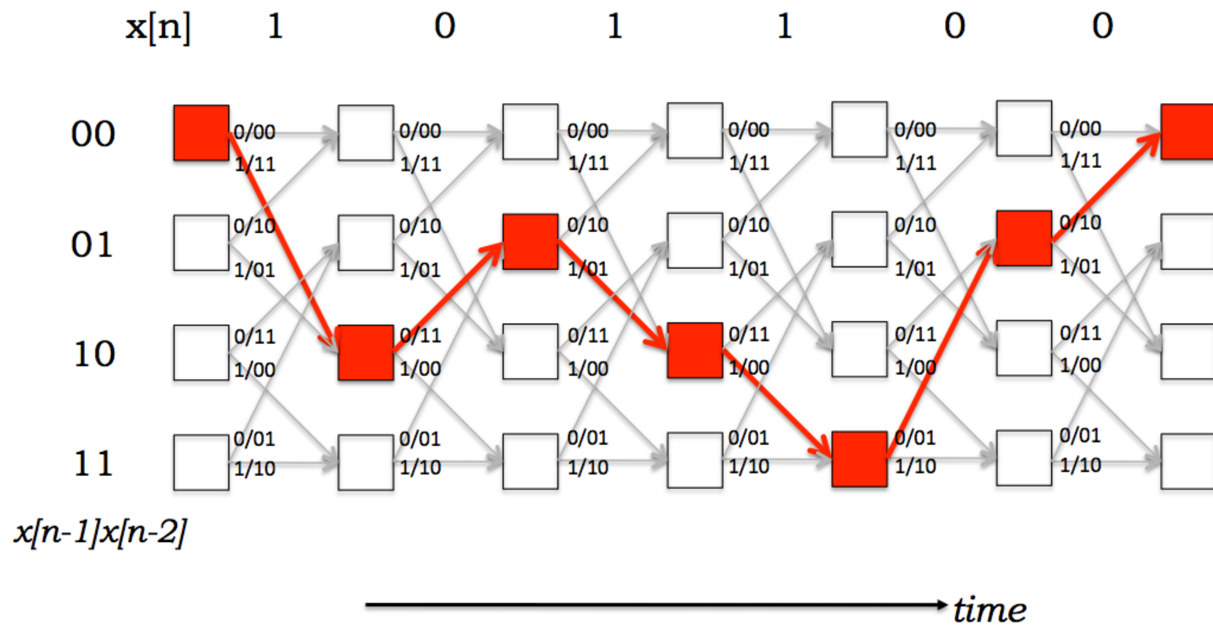

Figure 7-5: The trellis is a convenient way of viewing the decoding task and understanding the time evolution of the state machine.

of simply going through the list of possible transmit sequences and comparing Hamming distances is horribly intractable. We need a better plan for the receiver to navigate this unbelievable large space of possibilities and quickly determine the valid message with smallest Hamming distance. We will study a powerful and widely applicable method for solving this problem, called *Viterbi decoding*, in the next chapter. This decoding method uses a special structure called the **trellis**, which we describe next.

■ 7.5 The Trellis

The trellis is a structure derived from the state machine that will allow us to develop an efficient way to decode convolutional codes. The state machine view shows what happens at each instant when the sender has a message bit to process, but doesn't show how the system evolves in time. The *trellis* is a structure that makes the time evolution explicit. An example is shown in Figure 7-5. Each column of the trellis has the set of states; each state in a column is connected to two states in the next column—the same two states in the state diagram. The top link from each state in a column of the trellis shows what gets transmitted on a "0", while the bottom shows what gets transmitted on a "1". The picture shows the links between states that are traversed in the trellis given the message 101100.

We can now think about what the decoder needs to do in terms of this trellis. It gets a sequence of parity bits, and needs to determine the best path through the trellis—that is, the sequence of states in the trellis that can explain the observed, and possibly corrupted, sequence of received parity bits.

The Viterbi decoder finds a **maximum-likelihood path** through the trellis. We will study it in the next chapter.

Problems and exercises on convolutional coding are at the end of the next chapter, after we discuss the decoding process.