*MIT 6.02* **DRAFT** *Lecture Notes*
*Fall 2010 (Last update: October 4, 2010)*
*Comments, questions or bug reports?*
*Please contact 6.02-staff@mit.edu*

# LECTURE 8
# Convolutional Coding

This lecture introduces a powerful and 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. 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 *mathematically most likely* message from among the set of all possible transmitted messages. This lecture discusses the encoding; the next one discusses how to decode convolutional codes efficiently.

## ■ 8.1 Overview

Convolutional codes are a bit like the block codes discussed in the previous lecture in that they involve the transmission of parity bits that are computed from message bits. 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*.

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 lectures (i.e., modulo 2 addition, or equivalently, an exclusive-or operation). Unlike a block code, the windows overlap and slide by 1, as shown in Figure 8-1. 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 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, so one can't increase the constraint length arbitrarily and expect fast decoding.

If a convolutional code that 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

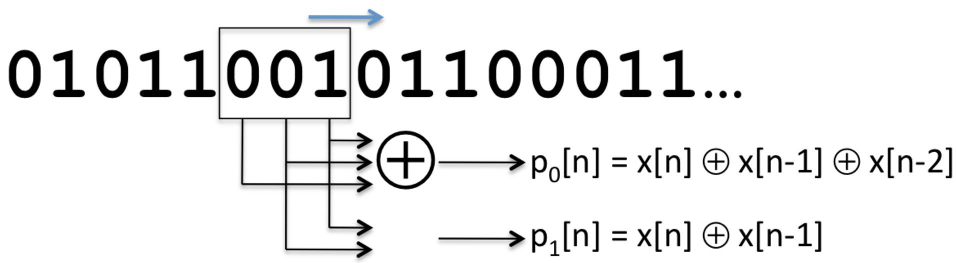


**Figure 8-1: An example of a convolutional code with two parity bits per message bit ($r = 2$) and constraint length (shown in the rectangular window) $K = 3$.**

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 lectures 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, most 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 8-1, 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 only remaining details that we have to worry about now 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 lecture will discuss these issues, and also explain why these codes are called "convolutional".

## ■ 8.2 Parity Equations

The example in Figure 8-1 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 $\mathbb{F}_2$)):

$$\begin{aligned} p_0[n] &= x[n] + x[n-1] + x[n-2] \\ p_1[n] &= x[n] + x[n-1] \end{aligned} \tag{8.1}$$

[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.

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

$$\begin{aligned}
p_0[n] &= x[n] + x[n-1] + x[n-2] \\
p_1[n] &= x[n] + x[n-1] \\
p_2[n] &= x[n] + x[n-2]
\end{aligned} \tag{8.2}$$

In general, one can view each parity equation as being produced by composing 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 $p_i$ as follows:

$$p_i[n] = (\sum_{j=0}^{k-1} g_i[j]x[n-j]) \bmod 2. \tag{8.3}$$

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.

## 8.2.1 An Example

Let's consider the two generator polynomials of Equations 8.1 (Figure 8-1). Here, the generator polynomials are

$$\begin{aligned}
g_0 &= 1,1,1 \\
g_1 &= 1,1,0
\end{aligned} \tag{8.4}$$

If the message sequence, $X = [1,0,1,1,\ldots]$ (as usual, $x[n] = 0 \ \forall n < 0$), then the parity bits from Equations 8.1 work out to be

$$\begin{aligned}
p_0[0] &= (1+0+0) = 1 \\
p_1[0] &= (1+0) = 1 \\
p_0[1] &= (0+1+0) = 1 \\
p_1[1] &= (0+1) = 1 \\
p_0[2] &= (1+0+1) = 0 \\
p_1[2] &= (1+0) = 1 \\
p_0[3] &= (1+1+0) = 0 \\
p_1[3] &= (1+1) = 0.
\end{aligned} \tag{8.5}$$

Therefore, the parity bits sent 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 8-1.

| Constraint length | $G_1$ | $G_2$ |
|---|---|---|
| 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 8-1: Examples of generator polynomials for rate $1/2$ convolutional codes with different constraint lengths.**

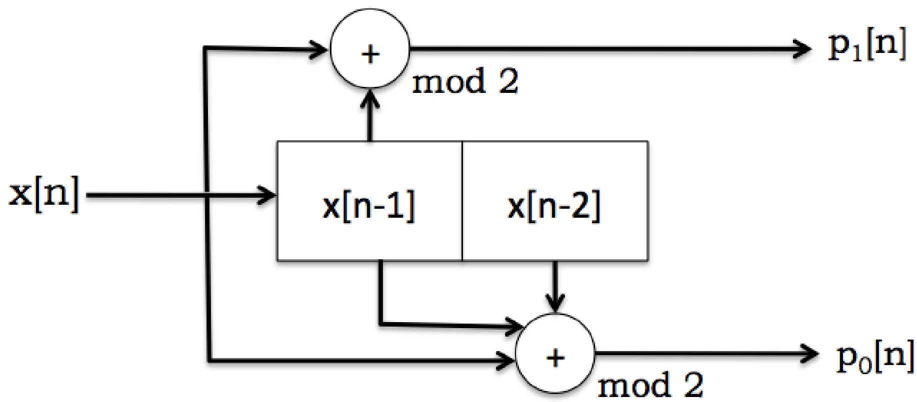


**Figure 8-2: Block diagram view of convolutional coding with shift registers.**

## ■ 8.3 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 a **block diagram**, where one can construct the mechanism using shift registers that are connected together. 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.

### ■ 8.3.1 Block Diagram View

Figure 8-2 shows the same encoder as Figure 8-1 and Equations (8.1) in the form of a block diagram. The $x[n-i]$ values (here there are two) are referred to as the *state* of the encoder. The way to think of this block diagram is as a "black box" that takes message bits in and spits out parity bits.

Input message bits, $x[n]$, arrive on the wire from the left. The box calculates the parity bits using the incoming bits and the state of the encoder (the $k-1$ previous bits; 2 in this example). After the $r$ parity bits are produced, the state of the encoder shifts by 1, with $x[n]$

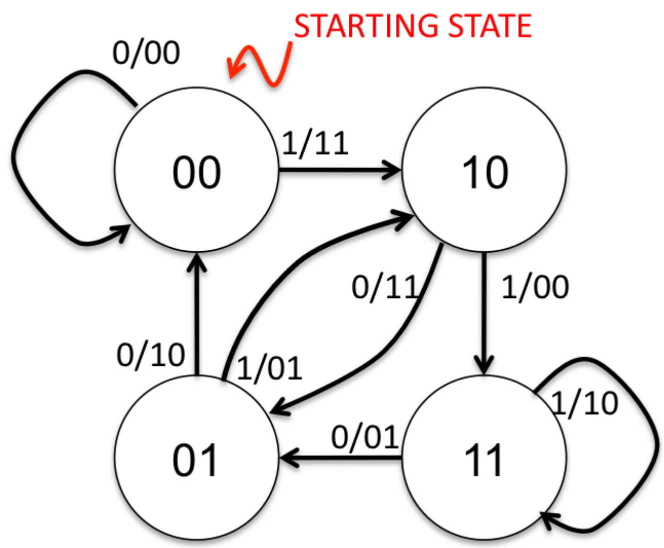


**Figure 8-3: State machine view of convolutional coding.**

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.

## ■ 8.3.2 State Machine View

Another useful view of convolutional codes is as a state machine, which is shown in Figure 8-3 for the same example that we have used throughout this lecture (Figure 8-1).

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 8-3) 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 corruptions. Its task is to determine the **best possible sequence of transmitter states that could have produced the parity bit sequence**. This task is called decoding, which we will introduce next, and then study in more detail in the next lecture.

## ■ 8.4 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 is also called a **maximum likelihood** decoder.

Consider the binary symmetric channel, where bits are received erroneously with probability $p < 1/2$. What should a maximum likelihood decoder do when it receives $r$? We show now that if it decodes $r$ as $c$, the nearest valid codeword with smallest Hamming distance from $r$, then the decoding is a maximum likelihood one.

A maximum likelihood decoder maximizes the quantity $P(r|c)$; i.e., it finds $c$ so that the probability that $r$ was received given that $c$ was sent is maximized. Consider any codeword $\tilde{c}$. If $r$ and $\tilde{c}$ differ in $d$ bits (i.e., their Hamming distance is $d$), then $P(r|c) = p^d(1-p)^{N-d}$, where $N$ is the length of the received word (and also the length of each valid codeword). It's more convenient to take the logarithm of this conditional probaility, also termed the *log-likelihood*:[2]

$$\log P(r|\tilde{c}) = d \log p + (N-d)\log(1-p) = d \log \frac{p}{1-p} + N \log(1-p). \tag{8.6}$$

If $p < 1/2$, which is the practical realm of operation, then $\frac{p}{1-p} < 1$ and the log term is negative (otherwise, it's non-negative). As a result, minimizing the log likelihood boils down to minimizing $d$, because the second term on the RHS of Eq. (8.6) is a constant.

A simple numerical example may be useful. Suppose that bit errors are independent and identically distribute with a BER of 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.

Determining the nearest valid codeword to a received word is easier said than done for convolutional codes. For example, see Figure 8-4, 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 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-

[2] The base of the logarithm doesn't matter to us at this stage, but traditionally the log likelihood is defined as the natural logarithm (base $e$).

| *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 |
| 1100 | 110001100000 | | 5 |
| 1101 | 110001011110 | | 4 |
| 1110 | 110010011000 | | 6 |
| 1111 | 110010100110 | | 3 |

Most likely: 1011

**Figure 8-4: 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!**

coded BER.)

The straightforward approach of simply going through the list of possible transmit sequences and comparing Hamming distances is horribly intractable. The reason is that a transmit sequence of $N$ bits has $2^N$ possible strings, a number that is simply too large for even small values of $N$, like 256 bits. 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 lecture. This decoding method uses a special structure called the **trellis**, which we describe next.

## ■ 8.5 The Trellis and Decoding the Message

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

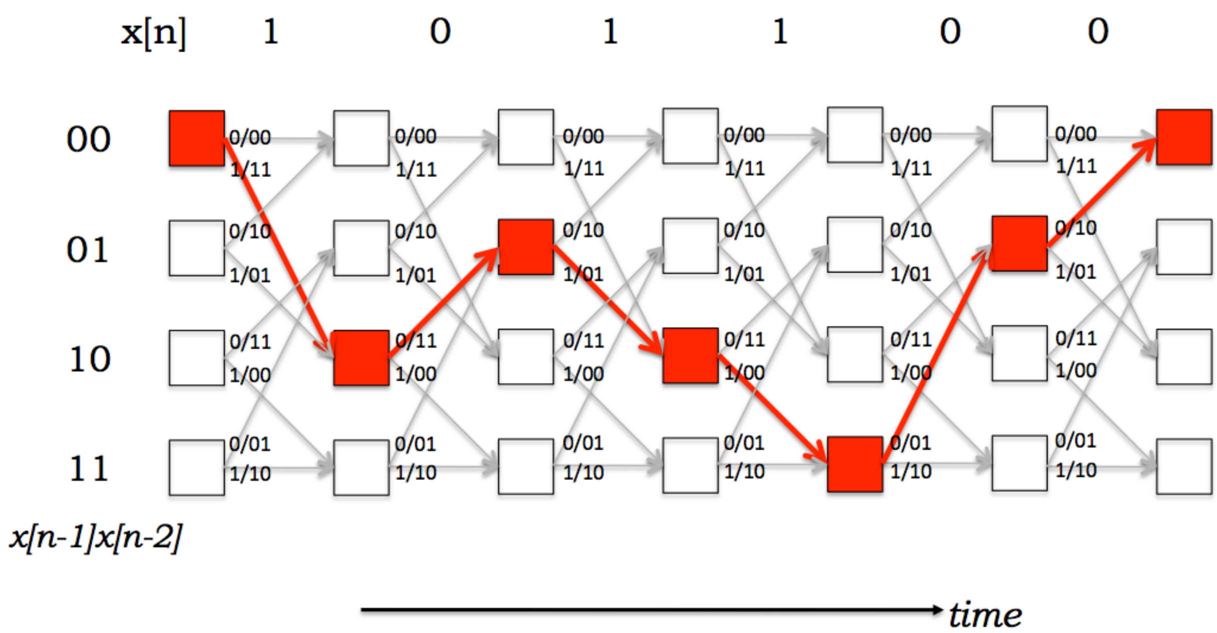


**Figure 8-5: The trellis is a convenient way of viewing the decoding task and understanding the time evolution of the state machine.**

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 8-5. 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 lecture.