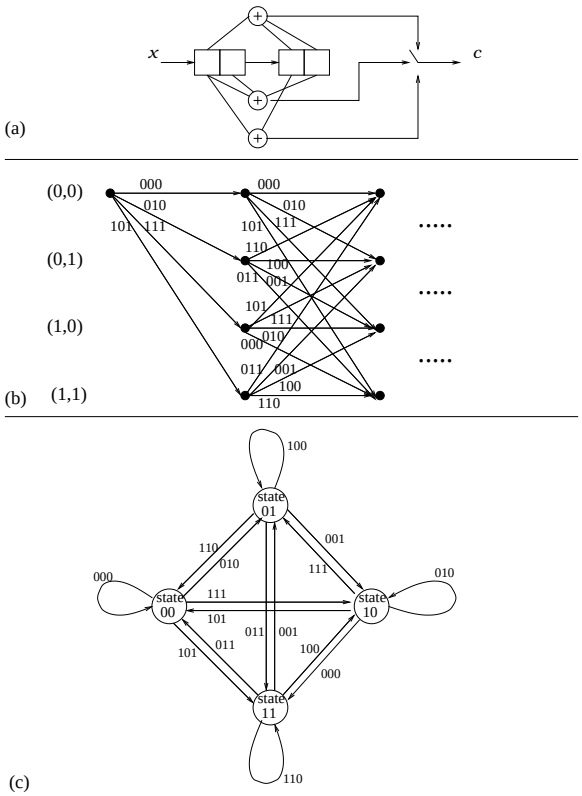
ECE8771
Information Theory & Coding for Digital Communications

Villanova University
ECE Department

Prof. Kevin M. Buckley

Lecture Set 3
Convolutional Codes



Contents

List of Figures

8 Convolutional Codes

In this Section of the Course we continue our discussion on channel coding. In the previous Section we considered block encoding as an approach to error control. Compared to direct transmission of binary information, block encoding of binary information bits into codewords incorporates redundancy (e.g. in the form of parity bits for binary systematic codes) that is used to reduce information bit errors. In hard-decision decoding, the parity bits are effectively used to detect and correct errors. In soft-decision decoding, the fact that the minimum distance d_{min} of codewords is greater than one effectively reduces the probability of codeword detection error, and subsequently bit errors compared to uncoded communications. Compared to hard-decision decoding, soft-decision decoding provides reduced information bit error rate with an additional implementation cost.

We now consider a different approach to channel coding termed *convolutional coding*. Unlike block codes, convolutional codes operate on a continuous stream of information bits to, in the binary case, form a continuous stream of coded bits. Redundancy is embedded into the coded bits resulting in a *code rate* R_c which is less than one. R_c is the ratio of the convolutional encoder input bit rate to the output bit rate. As an initial justification, you can look at the potential advantage of convolutional encoding to be the potential of generating very long codewords. We have noted previously, in discussions on channel capacity and block codes, the advantage of long codewords.

Decoding convolutional codes represents a new challenge. Up to this point in the Course, we have considered memoryless digital communications schemes. Decoding schemes have involved processing a received sample or a block of coded samples to detect a bit or codeword. Convolutional encoders have memory. As we will see, this means that the optimum (i.e. ML or MAP) decoder must use all of the ongoing stream of received data to detect each received symbol. At first, this may appear to be a significant or even unmanageable complication. However, we will learn about efficient, effective convolutional code decoding algorithms, the most widely used one being the Viterbi algorithm.

Below, we first describe convolutional codes and develop a number of useful mathematical descriptions of them. We will then consider codeword distance characteristics. Next we will see how to efficiently decode convolutional codes, using both hard-decision and soft-decision approaches. Then we will look at performance. We will close this Section by considering selected practical topics. In the next Section of the Course we will cover several advanced channel coding techniques involving both block and convolutional coders.

8.1 Linear Convolutional Codes & Their Descriptions

Generally, we consider encoders which generate an output stream of codeword symbols from an input stream of data symbols. The resulting code is termed a tree code. A trellis code, which can be represented with a trellis, is a tree code with a finite-state encoder whose states depend only on a finite number of past input symbols. A trellis code that adheres to certain linearity properties is termed a convolutional code.

We assume that the input and output data streams are binary so that operations are in the binary Galois field GF(2), at least for the binary convolutional coding schemes covered throughout this Section. Consider encoders with a finite-dimensional shift register of length Kk, that stores K frames of k input bits. Figure 73 illustrates this. k input information bits are shifted into this register at a time (and the k bits on the last frame are shifted out). As shown, each time k inputs are shifted in, $n > k$ linear algebraic operators (i.e. modulo-2 adders) generate n output bits. We will refer to this k-input, n-output operation as a *stage* of the encoder. At each stage, k new bites are shifted in and n output bits are generated, so the code rate is defined as $R_c = \frac{k}{n} < 1$. K is termed the *constraint length*. As information bits stream in, the encoder generates an output *code sequence* which is infinitely or indefinitely long. The code associated with this encoding structure is called a convolutional code. For reasons established later, this convolutional encoder structure is called the *canonical nonrecursive form*.

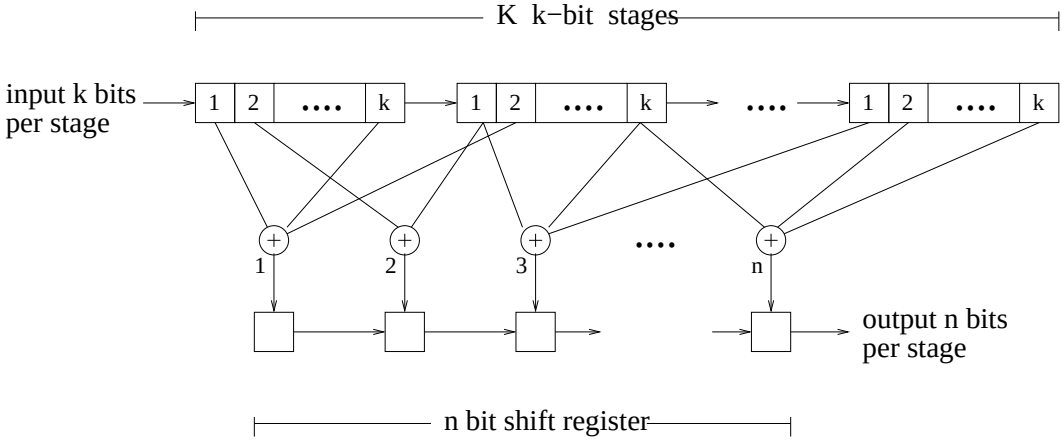


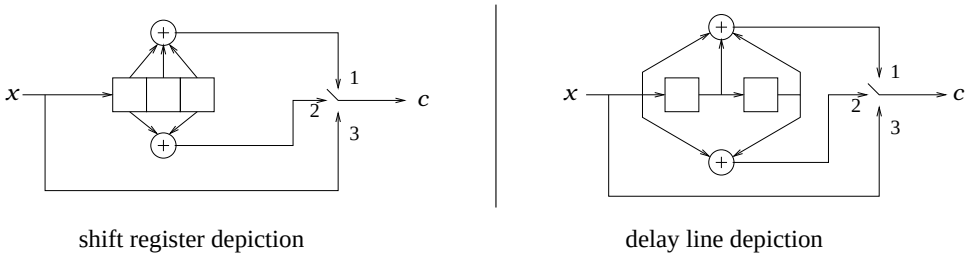
Figure 73: Block diagram of a general convolutional encoder.

Figure 74 shows two equivalent diagram for a $K = 3$ constraint length, $R_c = \frac{1}{2}$ rate convolutional encoder, where $\mathbf{x}$ and $\mathbf{c}$ are the information and code sequences respectively.

A convolutional code is linear in that, given any two code sequences $\mathbf{C}_1$ and $\mathbf{C}_2$, and any two binary constants α_1 and α_2 (binary since we are operating in GF(2)), then $\mathbf{C} = \alpha_1\mathbf{C}_1 + \alpha_2\mathbf{C}_2$ is also a code sequence. The input/output computation is a convolution. That is, for each algebraic operator, the input is folded and shifted against the operator binary (zero or one) weights, the input is multiplied by the weights and the results modulo 2 added. Thus the term convolutional code.

Block Diagram Representation

A *block diagram* description of a binary convolutional encoder is composed of an input shift register, modulo 2 multipliers and modulo 2 adders. Figures 73 & 74 are two illustrations. Each algebraic operator, consisting of a modulo 2 adder and corresponding binary "multipliers", is described by a *generator*. This generator indicates which stored information bits are used in the algebraic operation. The generators are usually given in octal form. The generators for the encoder shown in Figure 74 are $\mathbf{g}_1 = [1\ 1\ 1] = (7)$, $\mathbf{g}_2 = [1\ 0\ 1] = (5)$ and $\mathbf{g}_3 = [1\ 0\ 0] = (3)$.



shift register depiction delay line depiction

Figure 74: A constraint length $K = 3$, $k = 1$, $n = 3$, rate $R_c = \frac{1}{3}$ convolutional encoder.

Example 8.1 - Figure 75 illustrates a rate $R_c = \frac{1}{2}$, constraint length $K = 5$ convolutional encoder, with generators $\mathbf{g}_1 = (37)$ and $\mathbf{g}_2 = (21)$. This particular code is often used as a constituent code in a turbo encoder.

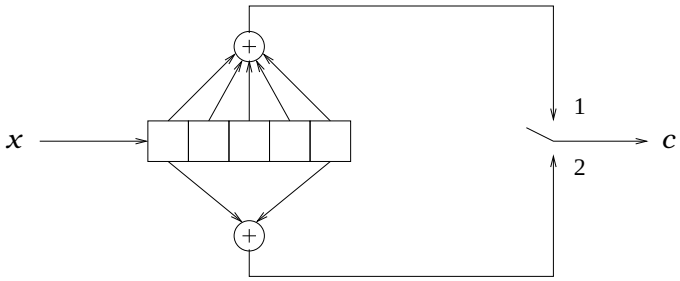


Figure 75: A constraint length $K = 5$, rate $R_c = \frac{1}{2}$ convolutional encoder with generators $\mathbf{g}_1 = (37)$ and $\mathbf{g}_2 = (21)$.

Problem: Determine the output code sequence for input sequence $\mathbf{X} = [0\ 1\ 0\ 1\ 0\ 1]$. Assume, as is standard, that initial conditions are zero.

Solution: $\mathbf{C} = [0\ 0\ ,\ 1\ 1\ ,\ 1\ 0\ ,\ 0\ 1\ ,\ 0\ 0\ ,\ 1\ 0\ ,\ 0\ 0\ ,\ 0\ 1\ ,\ 1\ 0\ ,\ 1\ 1]$.

Tree Diagram Representation

The block diagram representation of a convolutional encoder directly illustrates how the encoder can be implemented. For a convolutional code, a *tree diagram* is an alternative representation that more conveniently displays the output, suggests a basic suboptimum (tree search) approach to decoder implementation, and can be used as an intermediate step towards developing the very useful *trellis diagram* representation. A tree diagram is an expanding (tree branch) structure. Each path through the tree describes an input/output sequence pair. So the tree structure describes the input sequence to output sequence mapping.

Example 8.2 - Consider the convolutional code encoder shown in Figure 76(a). This code has rate $R_c = \frac{1}{2}$ and constraint length $K = 4$, with generators $\mathbf{g}_1 = (10)$ and $\mathbf{g}_2 = (15)$. This particular convolutional encoder is a *systematic encoder* because the information bits are transmitted directly, through the #1 generator.

Problem: Draw the tree diagram for the first 4 input bits.

Solution: Figure 76(b) shows the corresponding tree diagram. Note that the tree diagram assumes that the initial conditions are zero. Each input sequence corresponds to a path through the tree. The corresponding output code sequence is read off the tree path. Notice that, once the tree is built, it is a simple matter to trace an input path through the tree and read off the corresponding output sequence. Finally note that, after the initial conditions effects have dissipated, the structure repeats itself as it expands.

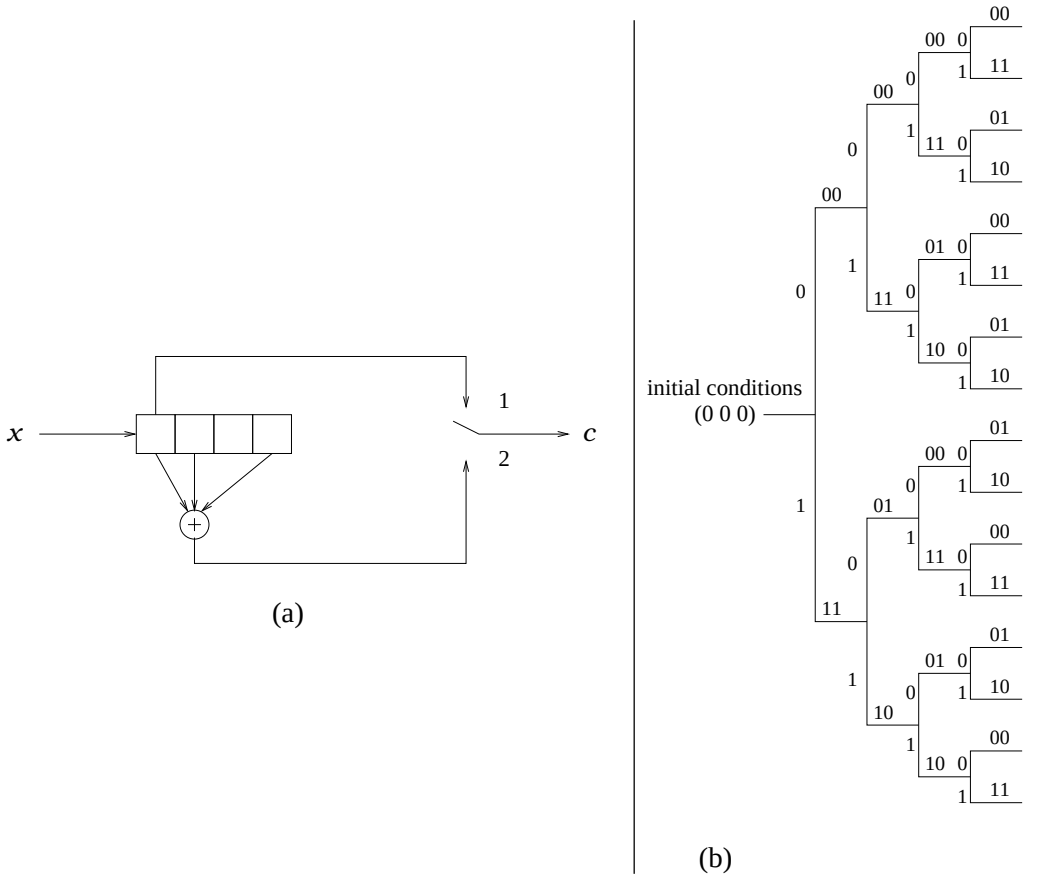


Figure 76: The block and tree diagrams for a $R_c = \frac{1}{2}$, $K = 4$ convolutional encoder, with generators $\mathbf{g}_1 = (10)$ and $\mathbf{g}_2 = (15)$, (a) the block diagram; (b) the tree diagram.

Trellis Diagram Representation

The trellis diagram representation can be viewed as a collapsed tree diagram representation that takes advantage of the repetitive structure of the tree. Equivalently, the trellis diagram can be viewed as a natural representation of the convolutional encoder block diagram, which is a finite state machine. The trellis diagram is a particularly useful representation for optimum and suboptimum decoder development.

A trellis diagram consists of *states, stages, branches* and *paths*. These components are described as follows:

- Each stage represents a time instant (i.e. k information bits into the encoder).
- The states represent the possible inputs stored in the past memory of the encoder.
- A branch, from one state at a previous stage to another state at the next stage given a particular new input, represents both: the transition of the previous state to the next state given the particular new input; and the output generated for that previous state and new input.
- A path through the trellis is a sequence of branches that represents an input information sequence and provides the corresponding output code sequence.

After an initial set of stages that depends on initial conditions, the trellis structure repeats itself, stage to stage.

> *Example 8.3* - Consider a rate $R_c = \frac{1}{2}$, constraint length $K = 3$ convolutional code, with generators $\mathbf{g}_1 = (7)$ and $\mathbf{g}_2 = (5)$. This convolutional encoder is shown in Figure 77. For the trellis, each stage has four states – (0 0), (0 1), (1 0), (1 1) – corresponding to the four possible states of the two past-input registers. Each of these states, plus the current input (in the current-input register), dictates both a state at the next stage and the encoder outputs at that stage (at that time).
>
> Consider stage 0. Assuming that the initial conditions are zero, only the (00) state is possible. This state can go to either state (00) or state (10) at stage 1, respectively, if a 0 or a 1 bit is entered at time 1. The two branches corresponding to these transitions are shown, labeled with the corresponding encoder outputs[1]. State (00) at stage 1 can again go to either state (00) or state (10) at stage 2, respectively, if a 0 or a 1 bit is entered at time 2. The output labels are as before. But now state (10) is considered, which can go to either state (01) or state (11) at stage 2, respectively, if a 0 or a 1 bit is entered at time 2. The output labels for these two transitions are also shown. Starting at stage 2, all states are possible, and the subsequent trellis stages are identical. That is,

[1]To understand how states & branches are labeled, consider the branch out of state (00) at stage $j = 0$. In going to the next stage, in this case stage $j = 1$ and state (10), to compute the output codeword, the shift register memory state will be (00) (i.e. as indicated by the previous trellis state), and the new input will be 1 as indicated by the first element of the next state.

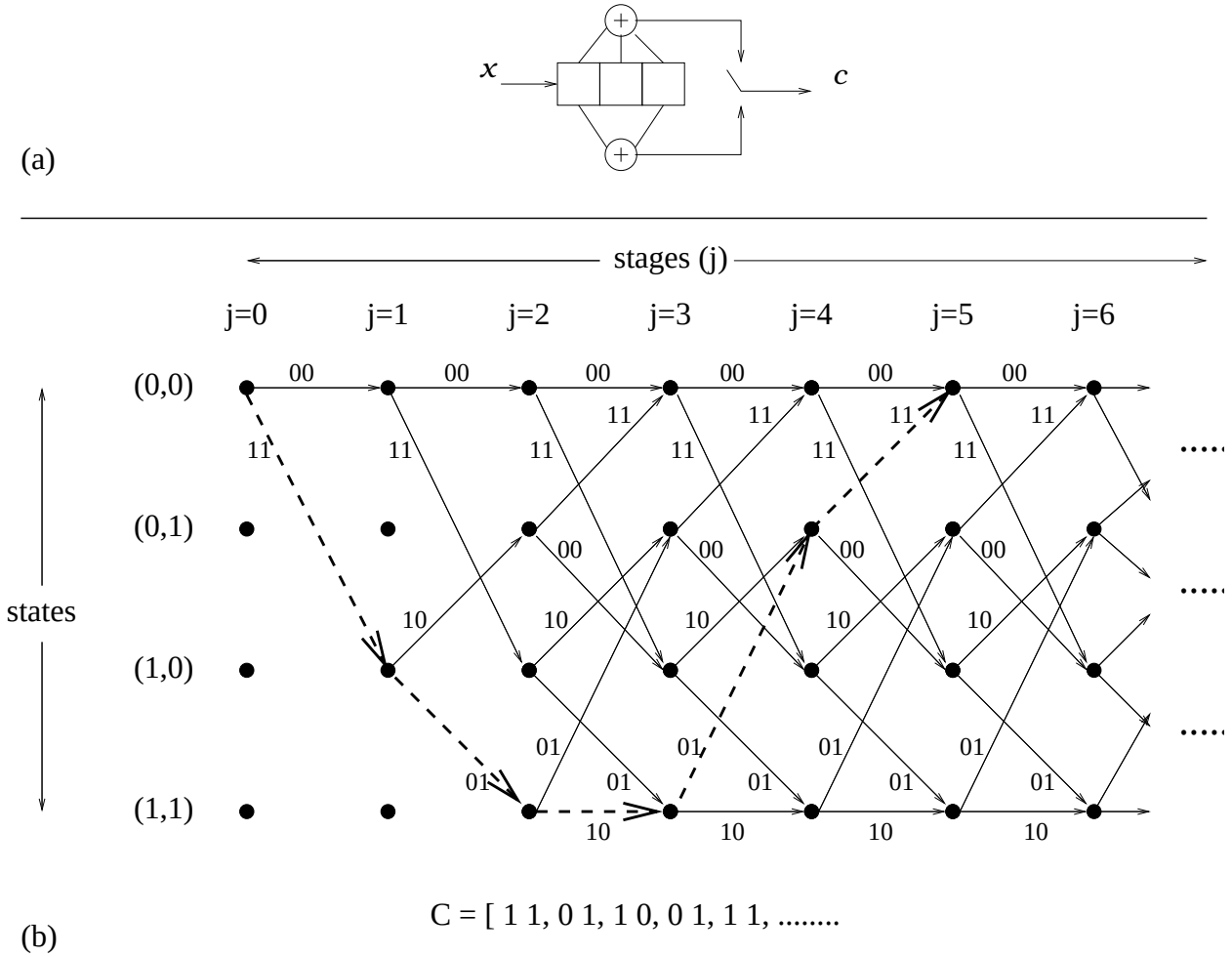


Figure 77: The trellis diagrams for the $R_c = \frac{1}{2}$, $K = 3$ convolutional encoder, with generators $\mathbf{g}_1 = (7)$ and $\mathbf{g}_2 = (5)$. Initial conditions are assumed to be zero.

1. state (00) can go the state (00) or (10), with branch labels (00) or (11), respectively, depending on whether the new information bit is a 0 or 1;
2. state (01) can go the state (00) or (10), with branch labels (11) or (00), respectively, depending on whether the new information bit is a 0 or 1;
3. state (10) can go the state (01) or (11), with branch labels (10) or (01), respectively, depending on whether the new information bit is a 0 or 1; and
4. state (11) can go the state (01) or (11), with branch labels (01) or (10), respectively, depending on whether the new information bit is a 0 or 1.

Problem: From the trellis diagram, determine the trellis path and output code sequence for input $\mathbf{X} = [11100]$.

Solution: See the dashed-branches and the code sequence at the bottom of Figure 77.

State Diagram Representation

A *state diagram* provides a compact description of a convolutional encoder. Compared to the trellis diagram, the state diagram is more compact. That is, a single state diagram represents all stage of the trellis. It is used again and again as we progress through the trellis stages. It is not as convenient as the trellis diagram for visualizing decoding algorithms because paths are not as evident, since for long input streams they will overlap themselves. However, as we will see, this representation is useful for deriving the convolutional code *transfer function*, which in turn is useful characterizing code performance.

Example 8.4 - Consider a rate $R_c = \frac{1}{2}$, constraint length $K = 4$ convolutional code, with generators $\mathbf{g}_1 = (10)$ and $\mathbf{g}_2 = (15)$. This, the code from Example 8.2, is shown again if Figure 78(a). The state diagram is shown in Figure 78(b).

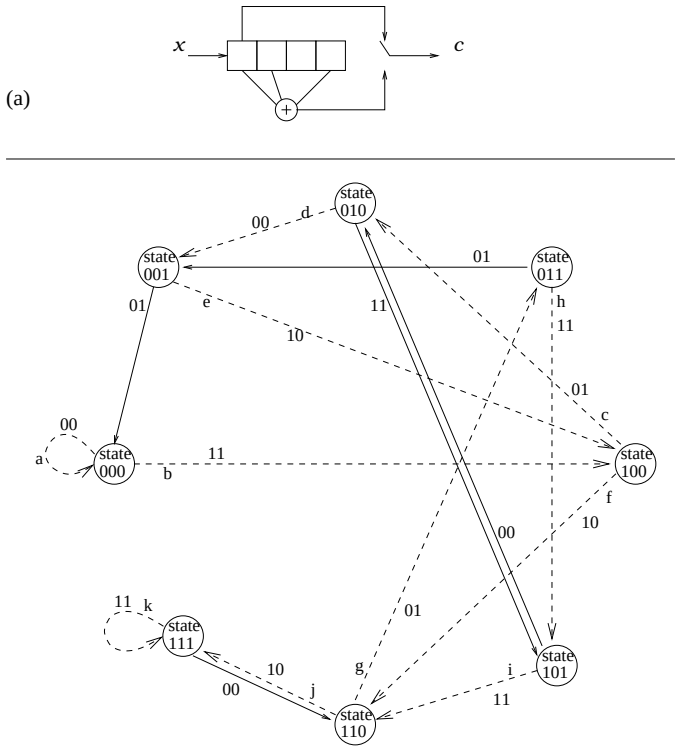


Figure 78: State diagram for the $R_c = \frac{1}{2}$, $K = 4$ convolutional code, with generators $\mathbf{g}_1 = (10)$, $\mathbf{g}_2 = (15)$.

Problem: For input sequence $\mathbf{X} = [0\ 1\ 0\ 0\ 1\ 1\ 0\ 1\ 1\ 1\ 1]$, show the path through the state diagram, and determine the output code sequence. (Initial conditions are assumed to be zero.)

Solution: The path is indicated with dashed-arrow lines, which are labeled in order alphabetically. The resulting code sequence is shown at the bottom.

Example 8.5 - Consider the rate $R_c = \frac{2}{3}$, constraint length $K = 2$ convolutional code with generators $\mathbf{g}_1 = (13)$, $\mathbf{g}_2 = (15)$ and $\mathbf{g}_3 = (12)$.

Problem: Draw the block diagram, the trellis, and the state diagram.

Solution: See Figure 79.

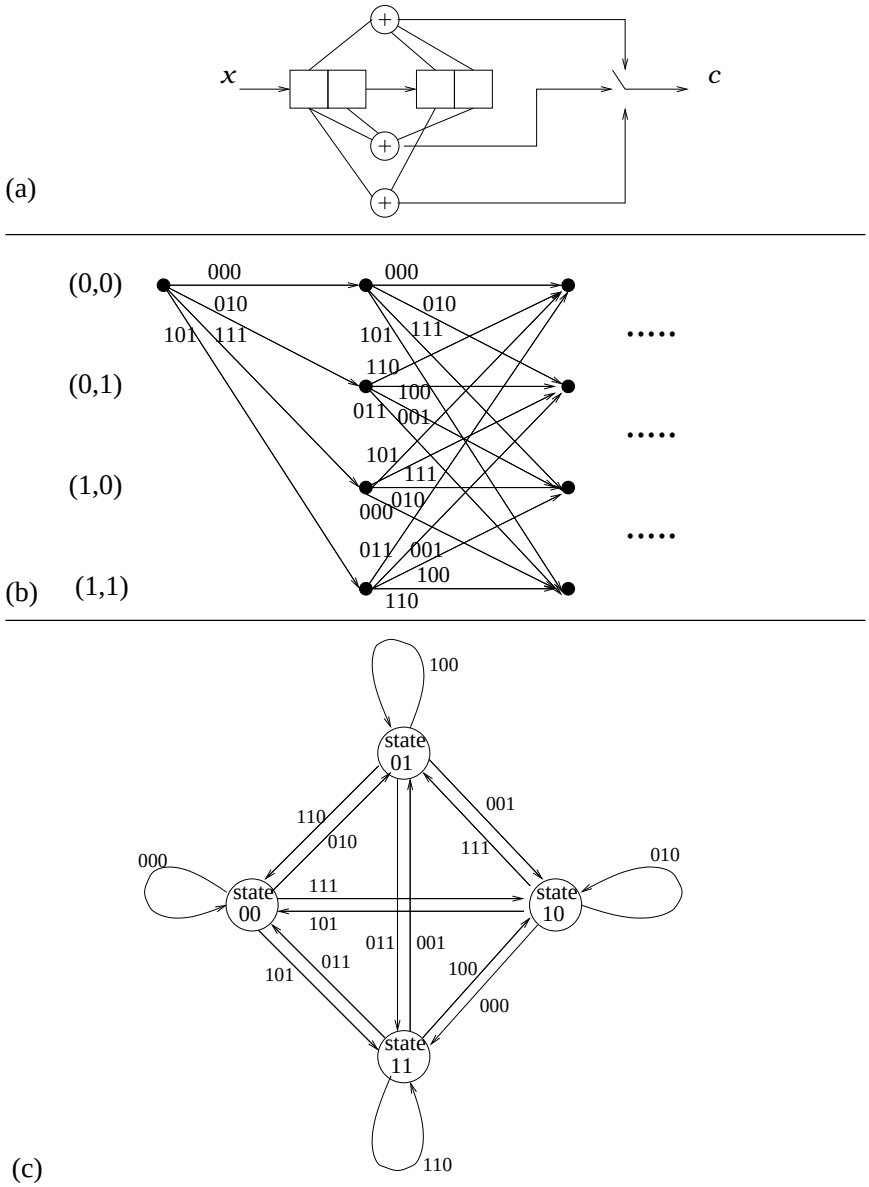


Figure 79: The block diagram, the trellis, and the state diagram for the $R_c = \frac{2}{3}$, constraint length $K = 2$ convolutional code in Example 8.5.

8.2 Transfer Function Representation & Distance Properties

As we have observed previously for linear block codes, and as we will again see below for convolutional codes, a very convenient and surprisingly powerful property of a linear code is that the weights of the codewords characterize the distance characteristics between all codewords. For block codes this means that we can use the Hamming distance of codewords from $\mathbf{C}_1$, the all-zero codeword, to determine the performance of the code.

Convolutional codes generate codewords which are infinitely long (if the input is infinitely long) or at least very long (if a long but finite length input is used). As noted earlier, we will refer to these as *code sequences*. The minimum code sequence weight d_{min} is not as useful as a related measure called the *minimum free distance* and denoted d_{free}. Minimum free distance is a minimum Hamming distance of a section of a code sequence from a section of the all-zero code sequence $\mathbf{C}_1$. Consider again the trellis in Figure 77, and recall that paths through the trellis represent code sequences. The upper most path, through all (00) states, represents $\mathbf{C}_1$. We can see from the Figure that paths (i.e. code sequences) can diverge from the $\mathbf{C}_1$ path and merge back with $\mathbf{C}_1$ at a later stage. For example, the code sequence derived in Example 8.3, $\mathbf{C} = [11, 01, 10, 01, 11, \cdots]$, diverges from the $\mathbf{C}_1$ path at stage 0 and merges at stage 5. The Hamming distance of this code sequence section from $\mathbf{C}_1$, over this duration of divergence, is $d = 7$. The minimum free distance d_{free} of a convolutional code is the minimum Hamming distance among all diverged/merged paths from $\mathbf{C}_1$ in the trellis (i.e. of a code sequence section). We will see that d_{free} is closely related to code performance.

The Code Transfer Function & State Equations

To identify the Hamming distances from all diverged/merged code sequence sections from $\mathbf{C}_1$, let Z be a variable which, for each trellis branch, is assigned a power that represents the number of nonzero elements (i.e. the Hamming weight) of the branch's output assignment[2]. For example, the branch output assignments (000), (101) and (111) correspond to $Z^0 = 1$, Z^2 and Z^3, respectively. The convolutional code *transfer function* $T(Z)$ is a function that represents the weights of paths that diverge from the all-zero path, from point of divergence to point of merging back. It is derived from these state diagram branch assignments as illustrated in the following example.

> *Example 8.6* - Consider the rate $R_c = \frac{1}{2}$, constraint length $K = 3$ convolutional code from Example 8.3: Its state diagram shown in Figure 80(a).
>
> *Problem:* Determine the code transfer function $T(Z)$.
>
> *Solution:* Considering the state diagram in Figure 80(a), note that the states are labeled with letters, and the (0 0) state is labeled with both **a** and **e**. Also note that, as described above, the branches between states are labeled with powers of Z, with the powers indicating the Hamming weight of the code sequence section associated with that branch.
>
> Recall that for free distance we are interested in the Hamming weight of code sequence sections which deviate and then return the all-zeros code sequence. For these path sections that deviate from the all zero code sequence, each section

[2]The variable Z is related to Hamming weight or distance.

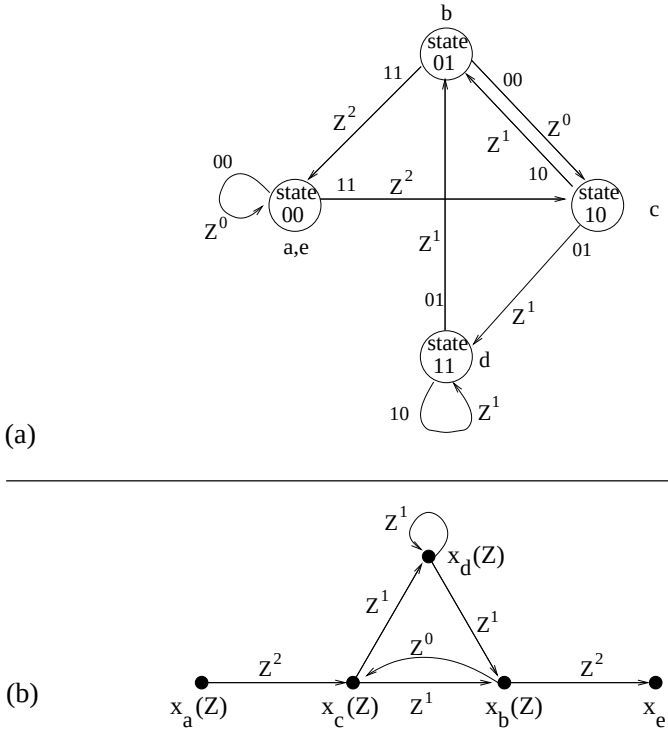


Figure 80: The state diagram for the $R_c = \frac{1}{2}$, constraint length $K = 3$ convolutional encoder in Example 8.3.

starts and ends at the (0 0) state. So as to represent code sequence sections which deviate from the all zero code sequence, we therefore form a modified state diagram which has a beginning node **a** (the (0 0) state) and ends at node **e** (again, the (0 0) state). We label the node $x_i(Z)$; $i = a, b, c, d, e$. The branches are labeled with their Hamming weight indicators. Figure 80(b) shows this modified state diagram for the code considered in this example.

Now, $T(Z)$ is simply the transfer function from node $x_a(Z)$ to node $x_e(Z)$. We generate the following node equations

$$\begin{aligned} x_b(Z) &= Z\ x_c(Z) + Z\ x_d(Z) \\ x_c(Z) &= Z^2\ x_a(Z) + x_b(Z) \\ x_d(Z) &= Z\ x_c(Z) + Z\ x_d(Z) \\ x_e(Z) &= Z^2\ x_b(Z)\ . \end{aligned} \tag{1}$$

Then, solving for $T(Z) = \frac{x_e(Z)}{x_a(Z)}$ by substitution, we get

$$T(Z) = \frac{x_e(Z)}{x_a(Z)} = \frac{Z^5}{1-2Z}\ . \tag{2}$$

The derivation of the transfer function $T(Z) = \frac{x_a(Z)}{x_e(Z)}$ in Example 8.6 was informal. Specifically, the reduction of the linear node equations in Eq (1) to an "input/output" expression was by substitution. In this next example we repeat the derivation in Example 8.6 using a structured approach which can be applied for any convolutional code.

Example 8.7 - Solving the node equations using linear algebra:

Let x_a and x_e denote, respectively, the "input" and "output" nodes. Let $\underline{x} = [x_b, x_c, x_d]^T$ be the vector of internal nodes. Then the node (i.e. state) and output equations can be written as

$$\underline{x} \;=\; \underline{A}\;\underline{x} \;+\; \underline{b}\;x_a \quad , \tag{3}$$

and

$$x_e \;=\; \underline{c}^T\;\underline{x} \quad , \tag{4}$$

where

$$\underline{A} \;=\; \begin{bmatrix} 0 & Z & Z \\ 1 & 0 & 0 \\ 0 & Z & Z \end{bmatrix} \;, \qquad \underline{b} \;=\; \begin{bmatrix} 0 \\ Z^2 \\ 0 \end{bmatrix} \;, \qquad \underline{c} \;=\; \begin{bmatrix} Z^2 \\ 0 \\ 0 \end{bmatrix} \;. \tag{5}$$

Noting from Eq (3) that $\underline{x} \;=\; (\underline{I}_3 - \underline{A})^{-1}\;\underline{b}\;x_a$, we have that

$$x_e \;=\; \underline{c}^T\;(\underline{I}_3 - \underline{A})^{-1}\underline{b}\;x_a \quad , \tag{6}$$

or

$$T(Z) \;=\; \frac{x_e}{x_a} \;=\; \underline{c}^T\;(\underline{I}_3 - \underline{A})^{-1}\;\underline{b} \quad . \tag{7}$$

Try completing this derivation on your own. Note that because of the sparse construction of $\underline{b}$ and $\underline{c}$ for this example, only the (1,2) element of $(\underline{I}_3 - \underline{A})^{-1}$ is required. This element will be $Z/(1-2Z)$.

Eq (7) is a general expression for deriving the transfer function of a convolutional code in terms of its state equations generated from its state diagram.

The Code Segment Hamming Distances

The Hamming distances of all of the diverged/merged paths sections can be derived from $T(Z)$. Specifically, if $T(Z)$ is expressed as a power series in Z, the powers of Z represent the Hamming distances and the coefficients associated with them identify the number of unique paths that have that distance.

Example 8.8 - In Example 8.7 we derived the following transfer function for the rate $R_c = \frac{1}{2}$ code of Example 8.6:

$$T(Z) \;=\; \frac{Z^5}{1-2Z} \quad . \tag{8}$$

By long division or through use of the geometric series equation, we also have that

$$T(Z) \;=\; Z^5 \sum_{d=0}^{\infty} (2Z)^d \;=\; Z^5 + 2Z^6 + 4Z^7 + 8Z^8 + \cdots \quad . \tag{9}$$

So, the minimum Hamming distance of any diverged/merged path section is $d = 5$. There is one unique path segment with this Hamming distance from $\mathbf{C}_1$. This can be readily seen from either the state or trellis diagrams. Note also that there are two unique path sections of distance $d = 6$, etc..

The minimum Hamming distance is the minimum free distance d_{free}.

Equivalent Encoders

Recall that the generator matrix for a block code is not unique. That is, given a generator matrix, we can interchange rows or perform elementary operations of the rows and end up with at generator matrix which generates the same set of codewords. So we say that these generator matrices, each of which is an encoder, are *equivalent* in that they generate the same code. For example, there are a number of generator matrices (encoders) for the Hamming $(7, 4)$ code, one of which has a systematic form.

This idea of equivalence also applies for convolutional codes. Below we will first develop the procedure for generating equivalent encoders (i.e. transforming one encoder to another for the same code). Then, as examples, we will consider:

1. transforming an encoder into systematic form; and
2. transforming an encoder from a *catastrophic* form.

Systematic encoders are required for some systems – for example for Turbo encoders. Catastrophic encoders should always be avoided.

Consider generators $\mathbf{g}_i = [g_{i,0}, g_{i,1}, \cdots, g_{i,kK-1}];\ i = 1, 2, \cdots, n$ associated with an encoder for a convolutional code with rate $R_c = \frac{k}{n}$ and constraint length K. As used earlier for cyclic block code representation, let D be the delay operator (in z-transform notation, $D = z^{-1}$). Consider the generator polynomials

$$\mathbf{g}_i(D) \;=\; g_{i,0} + g_{i,1}D + g_{i,2}D^2 + \cdots + g_{i,kK-1}D^{kK-1} \qquad i = 1, 2, \cdots, n \quad . \tag{10}$$

Similarly, consider operator (transform) representation of the input

$$\mathbf{x}(D) \;=\; \sum_k \; x_k \; D^k \tag{11}$$

and the generator output code sequences

$$\mathbf{c}_i(D) \;=\; \sum_k \; c_{i,k} \; D^k \quad i = 1, 2, \cdots, n \quad . \tag{12}$$

Note that

$$\mathbf{c}_i(D) \;=\; \mathbf{g}_i(D)\,\mathbf{x}(D) \quad i = 1, 2, \cdots, n \quad . \tag{13}$$

(That is, convolution of the coefficients corresponds to a product of the polynomials.)

Consider an rational function $\mathbf{a}(D)$ in GF(2). We can think of preprocessing the input as

$$\mathbf{x}^1(D) \;=\; \frac{1}{\mathbf{a}(D)}\,\mathbf{x}(D) \quad . \tag{14}$$

Now, defining $\mathbf{g}_i^1(D) = \mathbf{g}_i(D)\,\mathbf{a}(D)$, we have, from Eq (13),

$$\mathbf{c}_i(D) \;=\; \mathbf{g}_i^1(D)\,\mathbf{x}^1(D) \quad i = 1, 2, \cdots, n \quad . \tag{15}$$

So, the set of generators $\mathbf{g}_i^1;\ i = 1, 2, \cdots, n$ generate the same code sequences as the generators $\mathbf{g}_i;\ i = 1, 2, \cdots, n$, it's just that different inputs generate them (i.e. the mapping from the inputs to the same set of output code sequences is different). Thus we can use any invertible rational function $\mathbf{a}(D)$ in GF(2) to generate an equivalent encoder.

The *generator matrix* of the encoder is defined as

$$\mathbf{G}(D) \;=\; [\mathbf{g}_1(D), \mathbf{g}_2(D), \cdots, \mathbf{g}_n(D)] \quad . \tag{16}$$

Let $\mathbf{a}(D)$ be a rational function in GF(2). Then an equivalent generator matrix (i.e. for an equivalent encoder) is

$$\mathbf{G}^1(D) \;=\; \mathbf{G}(D)\,\mathbf{a}(D) \quad . \tag{17}$$

Example 8.9 - Consider the rate $R_c = \frac{1}{2}$, constraint length $K = 2$ convolutional code from Examples 8.3, 8.6, 8.7 and 8.8.

Problem: Starting with the encoder we have already considered, which has generators $\mathbf{g}_1 = (7)$ and $\mathbf{g}_2 = (5)$, transform it into an encoder which is in systematic form. the original encoder structure is called the *standard or (controller) canonical nonrecursive form.*

Solution: The generator matrix for the initial encoder is

$$\mathbf{G}(p) \;=\; [1 + D + D^2\ ,\ \ 1 + D^2] \quad . \tag{18}$$

Let

$$\mathbf{a}(D) \;=\; \frac{1}{1 \;+\; D \;+\; D^2} \quad . \tag{19}$$

Then the equivalent encoder in systematic form has generator matrix

$$\mathbf{G}^1(D) \;=\; \mathbf{G}(D)\,\mathbf{a}(D) \;=\; \left[\, 1\ ,\ \frac{1 \;+\; D^2}{1 \;+\; D \;+\; D^2} \right] \quad . \tag{20}$$

Figure 81 shows the two encoders.

Using the operator based generator matrix representation developed above, we can study characteristics of a catastrophic encoder. We will do this with the following example.

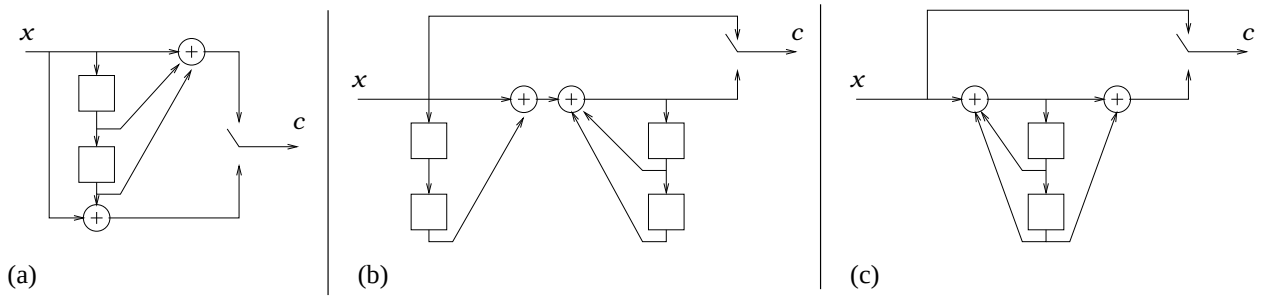


Figure 81: Equivalent encoders in: (a) canonical nonrecursive form; b) systematic recursive form (direct form I IIR); and c) systematic recursive form (direct form II IIR).

Example 8.10 - Consider again the convolutional code of the previous example. The generator matrix for the controller canonical nonrecursive form is

$$\mathbf{G}(D) \;=\; [1 + D + D^2 \;,\;\; 1 + D^2] \quad . \tag{21}$$

Consider a function $\mathbf{a}(D) = 1 + D$, and the new generator matrix

$$\mathbf{G}^2(D) \;=\; \mathbf{G}(D)\; \mathbf{a}(D) \;=\; [1 + D^3 \;,\;\; 1 + D + D^2 + D^3] \quad . \tag{22}$$

Now consider input $\mathbf{x}(D) = \frac{1}{1+D} = 1 + D + D^2 + D^3 + D^4 + \cdots$. This is an input with infinite Hamming weight. The encoder output is

$$\mathbf{c}_1(D) \;=\; \mathbf{g}_1^2(D)\; \mathbf{x}(D) \;=\; 1 + D + D^2 \tag{23}$$

interleaved with

$$\mathbf{c}_2(D) \;=\; \mathbf{g}_2^2(D)\; \mathbf{x}(D) \;=\; 1 + D^2 \quad . \tag{24}$$

So the code sequence is $[11,\ 10,\ 11,\ 00,\ 00, \cdots]$. The code sequence has finite weight, $d = 5$, which, being less than infinity, is catastrophic. This is because a few code sequence bit errors at the receiver can result in an infinite number of information bit errors. For example, if the 5 nonzero bits in the above code sequence are received in error (as zeros), the $\mathbf{C}_1$ code sequence will be received, and the information bits will be decoded as all zeros, even though the actual information bits are all ones.

8.3 Decoding Convolutional Codes

In the introduction to Section 8 of this Course, we noted that decoding convolutional codes represents a new challenge. This is because convolutional encoders have memory, so that the optimum (i.e. ML or MAP) decoder must simultaneously use all of the ongoing stream of received data to detect all of the received symbols. As a result, the optimum decoding problem is either:

1. a *sequence estimation* problem (as opposed to the symbol or bit estimation problem that we have considered thus far), where the entire sequence is estimated at once using all of the data; or

2. a *symbol-by-symbol* estimation problem, in which all of the received data is used to estimate each symbol.

In this Subsection we develop optimum algorithms for decoding convolutional codes. We begin, in Subsection 8.3.1, with optimum estimation of a sequence of discrete symbols based directly on sampled matched filter outputs. This is called soft-decision sequence estimation. We will consider the maximum likelihood (ML) criterion and develop the resulting soft-decision ML sequence estimator (soft-decision MLSE). This result easily generalizes to the maximum a posterior (MAP) sequence estimator. In Subsection 8.3.2 we apply MLSE to hard-decision decoding of convolutional codes. Then, in Subsection 8.3.3, we describe an efficient computational algorithm, the Viterbi algorithm, as applied to the convolutional code hard-decision and soft-decision decoding MLSE problems.

In Subsection 8.4 we will consider the performance of both soft and hard-decision MLSE algorithms for convolutional code decoding. In Subsections 8.5 and 8.6 we will overview alternative MLSE like suboptimum decoding algorithms (sequential decoding and the stack algorithm). In Subsection 8.7 we discuss puncturing, concatenated coding, interleaving and iterative decoding (e.g. Turbo coding/decoding). Later, in Section 9, to support the introduction of turbo coding, we describe two alternative decoding schemes which have become important in recent years (an algorithm based on Symbol-by-Symbol MAP and a soft output Viterbi Algorithm (SOVA)).

8.3.1 Soft Decision Maximum Likelihood Sequence Estimation (MLSE)

Consider the problem of estimating a code sequence (or equivalently the information bits it represents). For a rate $R_c = \frac{k}{n}$ convolutional code, at each point in time j there are n code bits to be estimated, represented by the vector $\mathbf{C}_j$. Here we will call these vectors *codeword vectors*. Coding stage (i.e. time) runs from $j = 1$ to $j = N$, where $N \leq \infty$. Being discrete, each $\mathbf{C}_j$ is taken from the finite number of vectors. In general, the $\mathbf{C}_j;\ j = 1, 2, \cdots, N$ are correlated over time (i.e. the process generating the $\mathbf{C}_j$ has memory).

We are given a set of noisy observations represented by the observation vectors $\mathbf{r}_j;\ j = 1, 2, \cdots, N$ which are vectors of sampled outputs of receiver matched filters. We assume the noise is zero-mean AWGN independent of the parameters $\mathbf{C}_j$. Depending on the modulation scheme and channel, the observation at time j, $\mathbf{r}_j$, is some function of all of the $\mathbf{C}_j;\ j = 1, 2, \cdots, N$. The $\mathbf{C}_j$ could, for example, be transmitted through a channel with

memory. However here we only consider the memoryless channel problem where

$$\mathbf{r}_j \;=\; \sqrt{\mathcal{E}_c}\; f(\mathbf{C}_j) \;+\; \mathbf{n}_j \;; \qquad j = 1, 2, \cdots, N \tag{25}$$

where $\mathcal{E}_c$ is the energy per code sequence bit, $f(\mathbf{C}_j)$ is a vector function of $\mathbf{C}_j$ that depends on the modulation scheme, and $\mathbf{n}_j$ is the AWGN vector. The dimension of the $\mathbf{r}_j$ vectors will depend on the modulation scheme and n, the dimension of the $\mathbf{C}_j$. We assume that $E\{\mathbf{n}_j \mathbf{n}_j^H\} = \sigma_n^2 \mathbf{I}$.

The joint PDF of the N observation vectors $\mathbf{r}_j;\; j = 1, 2, \cdots, N$, conditioned the parameter vectors $\mathbf{C}_j;\; j = 1, 2, \cdots, N$ is (assuming complex valued data and parameters)

$$p(\mathbf{r}_1, \mathbf{r}_2, \cdots, \mathbf{r}_N \;/\; \mathbf{C}_1, \mathbf{C}_2, \cdots, \mathbf{C}_N) \;=\; \prod_{j=1}^{N} \frac{1}{(\pi\sigma_n^2)^{p/2}}\; e^{-|\mathbf{r}_j - \sqrt{\mathcal{E}_c}\; f(\mathbf{C}_j)|^2/\sigma_n^2} \quad , \tag{26}$$

where p is the dimension of the $\mathbf{r}$ vectors. The reasons this conditional joint PDF factors into the product of the conditional PDF's of the individual observation vectors are that: 1) the noise is uncorrelated across time; and 2) the PDF is conditioned on the codeword vectors so that the fact that codeword sequence is correlated is irrelevant.

Note that at any time j that the n generators provide encoder output, there are 2^k possible binary inputs to the binary convolutional encoder. Thus, over time $j = 1, 2, \cdots, N$ there are 2^{kN} possible input bit sequences an correspondingly 2^{kN} possible encoder output sequences. With this in mind, consider the MLSE problem:

$$\max_{\mathbf{C}_1^{(i_1)}, \mathbf{C}_2^{(i_2)}, \cdots, \mathbf{C}_N^{(i_N)}} \qquad p(\mathbf{r}_1, \mathbf{r}_2, \cdots, \mathbf{r}_N \;/\; \mathbf{C}_1^{(i_1)}, \mathbf{C}_2^{(i_2)}, \cdots, \mathbf{C}_N^{(i_N)}) \quad , \tag{27}$$

where the maximization is over all possible combinations of codeword vectors, $\mathbf{C}_j^{(i_j)}; j = 1, 2, \cdots, N;\; i_j = 1, 2, \cdots, 2^k$. Since the $\mathbf{C}_j;\; j = 1, 2, \cdots, N$ are not independent, this problem can not be solved separately for each $\mathbf{C}_j, \mathbf{r}_j$ pair. Instead, the conditional joint PDF of the $\mathbf{r}_1, \mathbf{r}_2, \cdots, \mathbf{r}_N$ must be evaluated for all 2^{kN} possible $\mathbf{C}_j^{(i_j)}; j = 1, 2, \cdots, N;$ $i_j = 1, 2, \cdots, 2^k$. Letting $\underline{\mathbf{C}}_N^{(i)};\; i = 1, 2, \cdots, 2^{kN}$ denote the 2^{kN} possible or hypothesized codeword sequences of length N and $\underline{\mathbf{r}}_N = [\mathbf{r}_1, \mathbf{r}_2, \cdots, \mathbf{r}_N]$, Eq(27) can be more compactly written as

$$\max_{\underline{\mathbf{C}}_N^{(i)}} \qquad p(\underline{\mathbf{r}}_N \;/\; \underline{\mathbf{C}}_N^{(i)}) \quad . \tag{28}$$

Starting from Eq(26), the negative log likelihood function for Eq(28) is (after discarding terms that do not effect the optimization problem)

$$\begin{aligned} L_N^{(i)} \;=\; L(\underline{\mathbf{C}}_N^{(i)}) \;&=\; \sum_{j=1}^{N} \;|\; \mathbf{r}_j - \sqrt{\mathcal{E}_c}\; f(\mathbf{C}_j^{(i)})|^2 \\ &=\; L_{N-1}^{(i)} \;+\; |\; \mathbf{r}_j - \sqrt{\mathcal{E}_c}\; f(\mathbf{C}_N^{(i)})|^2 \;. \end{aligned} \tag{29}$$

With the first line of Eq(29), the MLSE problem can now be written as

$$\min_{\underline{\mathbf{C}}_N^{(i)}} \qquad L_N^{(i)} \;= \sum_{j=1}^{N} \; l_j^{(i)} \quad , \tag{30}$$

where $L_N^{(i)}$ is the *cost* of the i^{th} hypothesized codeword sequence, and

$$l_j^{(i)} \;=\; |\, \mathbf{r}_j - \sqrt{\mathcal{E}_c}\; f(\mathbf{C}_j^{(i)})|^2 \tag{31}$$

is the *incremental cost* of the j^{th} codeword of the i^{th} hypothesized sequence.

The second line of Eq(29), i.e.

$$L_N^{(i)} \;=\; L_{N-1}^{(i)} \;+\; l_N^{(i)} \quad , \tag{32}$$

shows that the MLSE cost can be computed in a time *recursive form*. We will see that this is a key to the Viterbi algorithm, which is an efficient MLSE algorithm.

As noted earlier, for the convolutional code decoding problem considered here, the observation $\mathbf{r}_j$ received at stage j depends on the modulation scheme. For a rate $R_c = \frac{k}{n}$ binary encoder and an M'-dimensional modulation scheme, $\mathbf{r}_j$ will be $\frac{n}{\log_2(M')}$-dimensional. For binary PSK, which is a 1-dimensional modulation scheme, $\mathbf{r}_j$ is n-dimensional and real-valued. For QPSK, which is 2-dimensional, $\mathbf{r}_j$ is $\frac{n}{2}$-dimensional and complex valued, which is equivalent to $\mathbf{r}_j$ being n-dimensional and real-valued. This is the same as for block coding. In fact, the receiver preprocessor, i.e. the matched-filter/sampler (and threshold device for hard-decision decoding), is the same for block and convolutional decoding.

As an example, we will specifically formulate the convolutional code MLSE soft-decoding cost function for binary PSK. For binary PSK, $\mathbf{C}_j = [c_{j1}, c_{j2}, \cdots, c_{jn}]$ is the encoder output at stage j, and

$$f(\mathbf{C}_j) \;=\; [f(c_{j1}), f(c_{j1}), \cdots, f(c_{jn})] \quad , \tag{33}$$

where $f(c_{jm}) = (2c_{jm} - 1)$, i.e.

$$f(c_{jm}) \;=\; \begin{cases} 1 & c_{jm} = 1 \\ -1 & c_{jm} = 0 \end{cases} \quad . \tag{34}$$

Then, after discarding terms which do not effect the minimization (i.e. constant terms common to all costs), the incremental cost $l_j^{(i)}$ in Eq(32) for the j^{th} stage of the i^{th} hypothesized codeword sequence reduces to

$$l_j^{(i)} \;=\; -\sum_{m=1}^{n} r_{j,m}(2c_{j,m}^{(i)} - 1) \quad , \tag{35}$$

where $r_{j,m}$ is the m^{th} element of $\mathbf{r}_j$ and $c_{j,m}^{(i)}$ is the m^{th} element of $\mathbf{C}_j^{(i)}$.

8.3.2 Hard-Decision Maximum Likelihood Sequence Estimation (MLSE)

As with hard-decision decoding for block codes, with hard-decision convolutional code decoding the sampled matched filter outputs are first detected, codeword vector bit by codeword vector bit. For SNR per information bit $\gamma_b = \frac{\mathcal{E}_b}{N_0}$, the SNR per codeword vector bit is $\gamma_b R_c$, where $R_c = \frac{k}{n}$ is the convolutional code rate. The probabilities of codeword vector bit errors for ML codeword bit detection in AWGN channels are, for for coherent PSK, coherent FSK and noncoherent FSK respectively,

$$\rho \;=\; Q(\sqrt{2\gamma_b R_c}) \quad , \tag{36}$$

$$\rho \;=\; Q(\sqrt{\gamma_b R_c}) \tag{37}$$

and

$$\rho \;=\; \frac{1}{2}\, e^{-\gamma_b R_c/2} \quad . \tag{38}$$

Let the noisy detected codeword vector at stage j be denoted $\mathbf{Y}_j = [y_{j,1}, y_{j,2}, \cdots, y_{j,n}]$. Given $\underline{\mathbf{Y}}_N = [\mathbf{Y}_1, \mathbf{Y}_2, \cdots, \mathbf{Y}_N]$, the ML codeword vector sequence hard-decision detection problem is

$$\max_{\underline{\mathbf{C}}_N^{(i)}} \qquad P(\underline{\mathbf{Y}}_N / \underline{\mathbf{C}}_N^{(i)}) \;=\; \prod_{j=1}^{N} \prod_{m=1}^{n} \left[(1-\rho)\; \delta(y_{j,m} - c_{j,m}^{(i)}) \;+\; \rho\; \delta(y_{j,m} - \overline{c}_{j,m}^{(i)}) \right] \quad , \tag{39}$$

where $\overline{c}_{j,m}^{(i)}$ is the converse of $c_{j,m}^{(i)}$. Assume $\rho < 0.5$. One decoding algorithm is to directly compare each $\underline{\mathbf{C}}_N^{(i)}$ to $\underline{\mathbf{Y}}_N$ and pick as $\hat{\underline{\mathbf{C}}}_N$ the $\underline{\mathbf{C}}_N^{(i)}$ with minimum Hamming distance from $\underline{\mathbf{Y}}_N$. Therefore, an equivalent ML problem statement is:

$$\min_{\underline{\mathbf{C}}_N^{(i)}} \qquad w_h(\underline{\mathbf{Y}}_N - \underline{\mathbf{C}}_N^{(i)}) \quad , \tag{40}$$

where $w_h(\underline{\mathbf{Y}}_N - \underline{\mathbf{C}}_N^{(i)})$ is the Hamming weight of $\underline{\mathbf{Y}}_N - \underline{\mathbf{C}}_N^{(i)}$.

As is the case for soft-decision decoding of convolutional codes, the MLSE cost can be written in recursive form. Now let $L_N^{(i)}$ be the cost for the i^{th} hypothesized codeword sequence, and let

$$l_j^{(i)} \;=\; w_h(\mathbf{Y}_j - \mathbf{C}_j^{(i)}) \tag{41}$$

be the incremental cost of the j^{th} codeword of the i^{th} hypothesized sequence. The hard-decision MLSE convolutional decoding problem can be written as

$$\min_{\underline{\mathbf{C}}_N^{(i)}} \qquad L_N^{(i)} \;=\; L_{N-1}^{(i)} \;+\; l_N^{(i)} \quad . \tag{42}$$

8.3.3 The Viterbi Algorithm for MLSE

In the previous two Subsections we developed recursive cost equations for both the soft and the hard-decision MLSE convolutional code decoders. Although these provide a computationally efficient form for recursively computing MLSE cost as new data arrives, they do not address the principal computational problem associated with MLSE. The main problem is that the number of hypothesized codeword sequences grows exponentially with time. That is, at stage N there are 2^{kN} hypothesized codeword sequences whose costs must be calculated (recursively, from the $2^{k(N-1)}$ previous costs) and compared.

As noted earlier, the trellis diagram provides a useful visualization of the hypothesized encoder output sequences (and therefore of the encoder binary input information sequence). Figure 77(b), reproduced below, is the trellis for the rate $R_c = \frac{1}{2}$, constraint length $K = 3$ encoder with generators $\mathbf{g}_1 = (7)$ and $\mathbf{g}_1 = (5)$.

Considering Figure 82, the paths through the trellis represent the hypothesized code sequences and corresponding input information sequences. That is, each path through the trellis corresponds to a hypothesis. As before, in the figure we label each branch at stage j with the codeword vector bits at that stage for any codeword vector sequence which goes through that branch. Also note that the states indicate the encoder input bits. The first bit in any state label at stage j is the encoder input bit at stage j for any path going through that state. The decoding problem is then to determine the path through the trellis corresponding to the minimum MLSE cost, and to then trace through this path and read off the corresponding input sequence (i.e. the estimated information bits) using the state labels.

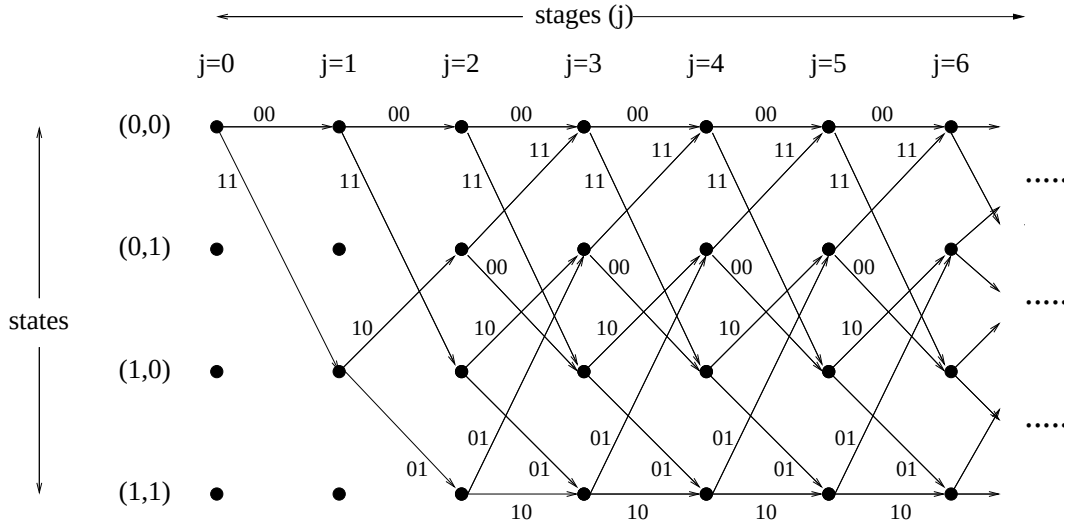


Figure 82: The trellis diagrams for the $R_c = \frac{1}{2}$, $K = 3$ convolutional encoder, with generators $\mathbf{g}_1 = (7)$ and $\mathbf{g}_2 = (5)$. Initial conditions are assumed to be zero.

Recall that the MLSE costs can be written in recursive form. In trellis diagram terminology, for example for soft-decision decoding and coherent binary PSK modulation, we have at stage j a path cost

$$L_N^{(i)} \;=\; L_{j-1}^{(i)} \;+\; l_j^{(i)} \quad , \tag{43}$$

with branch cost at stage j

$$l_j^{(i)} \;=\; -\;\sum_{m=1}^{n}\; r_{j,m}(2c_{j,m}^{(i)}-1) \quad . \tag{44}$$

So, because of the recursive nature of the MLSE costs, these costs can be thought of as path costs which are the sum of branch costs. It is important to note that:

> *a branch cost is computed using only that branch's output vector label and the received data at stage j. Each path through a branch incurs the same branch cost.*

With the trellis representation only, there is still no reduction in computation. In this respect, by itself, the trellis representation gives us nothing. But as a visualization of hypothesized codeword vector sequences, it provides a particularly easy description of an approach to reduce computation. Since the number of paths increases exponentially with stage j, what we need, if possible, is a way to *prune (discard) paths at each stage without sacrificing optimality*. In other words, we would like to prune any path at stage j that is not optimum and can not grow to be optimum at any future stage.

Consider two distinct paths into a given state at stage j. Call these paths Path 1 and Path 2, and assume that at stage j the cost of Path 1 is less then that of Path 2. Let these two paths grow into the future along the same but arbitrary sequence of branches. Since the branch costs are the same for both paths, both paths incur the same additional cost, and so Path 1 is still better than Path 2. This is true regardless of the branches that are transversed subsequent to stage j. This means that at stage j, Path 2 can never grow into an optimum path. So it can be pruned at stage j. This is the key point in the development of the *Viterbi algorithm*. Note that it requires that all paths through a branch incur the same incremental cost. This condition is referred to as the *Markov condition*.

The Viterbi Algorithm

The reason bit sequence costs for both the soft-decision and hard-decision MLSE can be expressed as iterative updates is that each data joint probability density functions, conditioned on the transmitted sequence, is of the form (shown for soft decision decoding)

$$p(\mathbf{r}_1,\mathbf{r}_2,\cdots,\mathbf{r}_N) \;=\; p(\mathbf{r}_N/\mathbf{r}_{N-1})\;p(\mathbf{r}_{N-1}/\mathbf{r}_{N-2})\;\cdots\;p(\mathbf{r}_2/\mathbf{r}_1)\;p(\mathbf{r}_1) \quad . \tag{45}$$

That is, the received data, conditioned on the transmitted bit sequence, is a Markov process. Additionally, as the trellis diagram illustrates, each conditional PDF depends on a finite dimensional state (representing input bits). In terms of the trellis diagram representation of a finite state Markov process, the Viterbi algorithm is an optimum pruning algorithm. Simply put, Viterbi says keep only the incoming path corresponding to the lowest cost. At stage $j-1$, extend all *survivor* paths (i.e. paths not pruned at stage $j-1$) to stage j by adding the branch costs to the stage $j-1$ path costs. Then, for each state at stage j, prune all but the one lowest cost path.

After the beginning stages effected by initial conditions, each of the M paths (one for each of the M stages) will be extended to at most M new states (some states do not connect). So, at each stage, at most M^2 branch costs need be computed to extend the paths. Then, to implement the pruning, a comparison of at most M path costs is required for each of the M states.

Example 8.11 - Consider again the rate $R_c = \frac{1}{2}$, constraint length $K = 3$ convolutional code, with decoder generators $\mathbf{g}_1 = (7)$ and $\mathbf{g}_2 = (5)$. The trellis diagram is shown in Figure 83. Assume BPSK with coherent soft-decision decoding.

Problem: Assume $N = 5$ stages, and let the received data be

$$\underline{r}_5 = [\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3, \mathbf{r}_4, \mathbf{r}_5] = \begin{bmatrix} 1 & 1 & -1 & -1 & 1 \\ 1 & -1 & -1 & 1 & -1 \end{bmatrix} . \quad (46)$$

Using the Viterbi algorithm, determine the soft-decision MLSE.

Solution: The $j-th$ branch cost for the $i-th$ code sequence is

$$l_j^{(i)} = -r_{j,1}\,(2\,c_{j,1}^{(i)} - 1) - r_{j,2}\,(2\,c_{j,2}^{(i)} - 1) . \quad (47)$$

Each branch in the trellis diagram in Figure 83 is labeled with both the codeword vector (the 2-dimensional binary vector) and the branch cost (the italic number). Recall that a path (sequence) cost up to a given stage (time) is the sum of its branch (incremental) costs. Starting with stage 1, for each state compute all path costs into the state, and discard all but the lowest cost path. In Figure 83, each state is labeled with this lowest cost, and each discarded path is marked with an "x". We've done this for the first five stages. At stage 5, the states are labeled with best-path costs: -4 for state (0,0); -6 for state (0,1); -4 for state (1,0); and -10 for state (1,1). The dashed path into each stage 5 state is the minimum cost path into that state. Since at stage 5, state (1,1) has the lowest path cost, the lowest cost path into that state, shown as the thicker dashed path, represents the MLSE. The MLSE, identified from the sequence of states transversed by this path, is then

$$\hat{\mathbf{X}} = [1\ 0\ 1\ 1\ 1] . \quad (48)$$

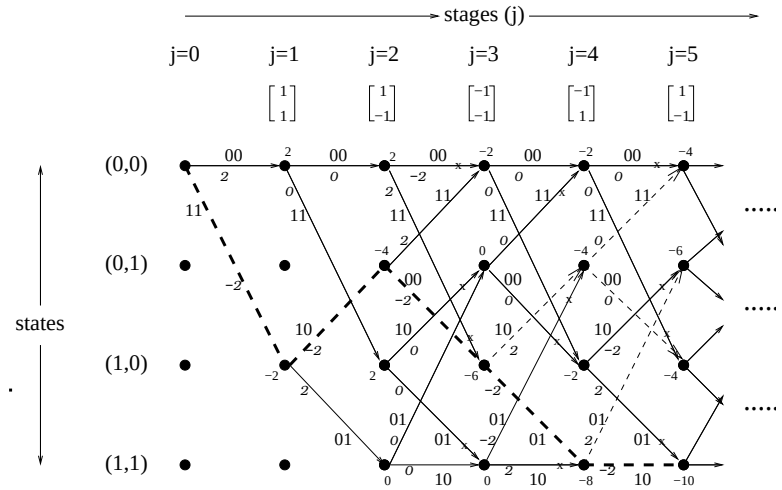


Figure 83: The trellis diagrams for the $R_c = \frac{1}{2}$, $K = 3$ convolutional encoder, with generators $\mathbf{g}_1 = (7)$ and $\mathbf{g}_2 = (5)$. Initial conditions are assumed to be zero.

8.4 Performance of Convolutional Code Decoders

To describe upper bounds for soft and hard-decision convolutional code decoder performance, we need to first revisit the transfer function representation of a convolutional code encoder. Recall that with Z being a Hamming distance operator, the transfer function, $T(Z)$, written as a power series, is od the form

$$\begin{aligned} T(Z) &= \sum_{d=d_{free}}^{\infty} a_d \, Z^d \\ &= a_{d_{free}} Z^{d_{free}} + a_{d_{free}+1} Z^{d_{free}+1} + a_{d_{free}+2} Z^{d_{free}+2} + \cdots \end{aligned} \tag{49}$$

From this power series, we concluded that: 1) the Hamming weights of code sequence sections which diverge-from/merge-with the all-zero code sequence path are given as the powers d on Z; and 2) the number of unique path sections associated with each Hamming weight d is given as the coefficient a_d associated with the that power of Z. This transfer function was derived from a modified state diagram by labeling each branch with Z raised to the power equal to the number of 1's in the output code label.

Because a convolutional code is linear, we need only evaluate performance (that is, to determine the bit error rate) for the all-zero code sequence $\mathbf{C}_1$. This will indicate the overall performance. Thus we need only to look at the probabilities of trellis path sections that diverge-from/merge-with the all-zero path. The transfer function $T(Z)$ adequately characterizes these path sections, except that it does not indicate the number of information bits associated with it that have value "1". Thus, it does note tell us how many information bit errors are associated with these diverged/merged path sections. An information bit indicator can be incorporated into the transfer function with a simple modification of the state diagram used to generate the transfer function. This modification is to add to the branch labels an indicator of a "1" information bit. The form of this indicator is a $Y^0 = 1$ if the branch is not associated with a new information bit of value "1" and a Y^1 if the branch is associated with a new information bit of value "1". That is, the power on the p counts the number of "1" information bits. The result is the modified transfer function $T(Z, Y)$. To illustrate this, we extend the Examples 8.6, 8.7 and 8.8 used previously to study the transfer function $T(X)$.

Example 8.12 - Consider the rate $R_c = \frac{1}{2}$, constraint length $K = 3$ convolutional code from Examples 8.6-8: Its state diagram shown in Figure 84.

Problem: Determine the code transfer function $T(Z, Y)$.

Solution: $T(Z, Y)$ is simply the transfer function from node $x_a(Z)$ to node $x_e(Z)$. As before, we generate the following node equations

$$\begin{aligned} x_b(Z) &= Z \, x_c(Z) + Z \, x_d(Z) \\ x_c(Z) &= YZ^2 \, x_a(Z) + Y \, x_b(Z) \\ x_d(Z) &= YZ \, x_c(Z) + YZ \, x_d(Z) \\ x_e(Z) &= Z^2 \, x_b(Z) \ . \end{aligned} \tag{50}$$

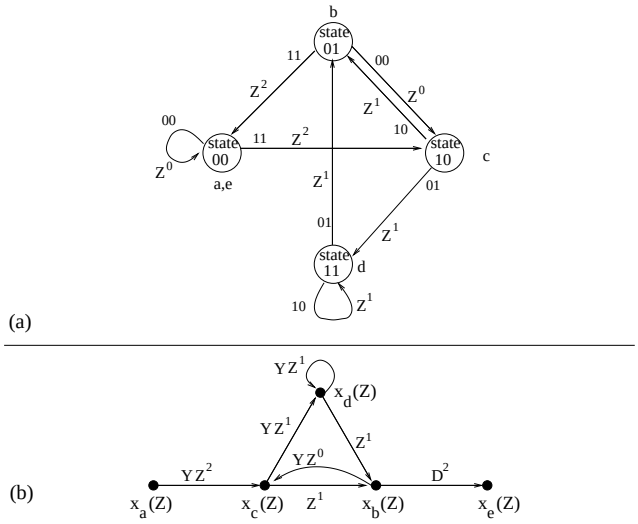


Figure 84: The state diagram for the $R_c = \frac{1}{2}$, constraint length $K = 3$ convolutional encoder in Examples 8.6-8.

Then, solving for $T(Z,Y) = \frac{x_e(Z)}{x_a(Z)}$ by substitution, we get

$$T(Z,Y) \;=\; \frac{x_e(Z)}{x_a(Z)} \;=\; \frac{YZ^5}{1-2YZ} \quad . \tag{51}$$

By long division or through use of the geometric series equation, we also have

$$T(Z,Y) = YZ^5 \sum_{d=0}^{\infty} (2YZ)^d = \sum_{d=d_{free}}^{\infty} Y^{d-4} 2^{d-5} Y^d = YZ^5 + 2Y^2Z^6 + 4Y^3Z^7 + 8Y^4Z^8 + \cdots$$

As noted before, for this convolutional code the minimum Hamming weight of any diverged/merged path section is $d_{free} = 5$. There is one unique path segment with this Hamming distance from $\mathbf{C}_1$, and it corresponds to a single "1" bit (i.e. a single information bit error). There are two unique path sections of distance $d = 6$, both corresponding to two information bit errors.

For Example 8.12, the transfer function $T(Z,Y)$ is be of the form[3]

$$T(Z,Y) \;=\; \sum_{d=d_{free}}^{\infty} a_d \; Y^{f(d)} \; Z^d \quad , \tag{53}$$

where a_d is the number of unique diverged/merged path segments of weight d. $f(d)$ gives the number of bit errors associated with these. For Ex. 8.12, $a_d = 2^{d-5}$ and $f(d) = d - 4$.

[3]A more general form the $T(Z,Y)$ transfer function is

$$T(Z,Y) \;=\; \sum_{d=d_{free}}^{\infty} \left(a1_d \; Y^{f1(d)} \;+\; a2_d \; Y^{f2(d)} \right) \; Z^d \quad . \tag{52}$$

For this more general form, for a given d (i.e. a given Hamming distance of a codeword path segment from the all-zero part), there are $a1_d$ paths with corresponding input sequence segment of Hamming weight $f1(d)$ and $a2_d$ paths with corresponding input sequence segment of Hamming weight $f2(d)$. For some convolution codes there may be an need to further generalize this.

8.4.1 Soft-Decision Decoding Performance

Here we discuss only BPSK and QPSK with coherent reception. We present an upper bound on the bit error rate (BER) for convolutional codes. For a derivation of this bound, see Proakis and Salehi [1] or Lin and Costello [2].

An upper bound on the probability of information bit error, P_b, for coherent detection, BPSK or QPSK, and soft-decision decoding for a rate $\frac{1}{n}$ (i.e. $k = 1$) convolutional code is

$$P_b \; < \; \sum_{d=d_{free}}^{\infty} \; a_d \; Q(\sqrt{2\gamma_b R_c d}) \quad . \tag{54}$$

This is a union bound. It is the sum of the bit error probabilities due to each possible type of segment error.

For $k > 1$ divide this bound by k. This result is from Proakis, 4-th edition. An alternative expression is given by Eqs (8.2-12,13) of the Course Text (see Figure 8.2-1). For a discussion on upper bounds for other modulation schemes, see the Course Text (see pp. 514-515). The derivation of this alternative expression is pretty straightforward, but it requires background from Section 7.2-4 of the Course Text, which we have not covered.

8.4.2 Hard-Decision Decoding Performance

A bound on the probability of bit error, P_b, for coherent detection, BPSK or QPSK, and hard-decision decoding for a rate $\frac{1}{n}$ (i.e. $k = 1$) convolutional code is

$$P_b \; < \; \sum_{d=d_{free}}^{\infty} \; a_d \; f(d) \; P_2(d) \quad , \tag{55}$$

where for d even

$$P_2(d) \; = \; \sum_{k=d/2+1}^{d} \left(\begin{array}{c} d \\ k \end{array} \right) \rho^k (1-\rho)^{n-k} \; + \; \frac{1}{2} \left(\begin{array}{c} d \\ \frac{1}{2}d \end{array} \right) \rho^{d/2} (1-\rho)^{d/2} \quad , \tag{56}$$

where ρ is the codeword bit error probability. For d odd, ignore the 2-nd term in Eq (56).

As with soft-decision decoding, for $k > 1$ divide this bound by k, This result is derived in the Course Text (see pp. 514-515, including Figure 8.2-1) for the Eq (52) form of the transfer function. For the more general form given by Eq (52), you still sum over all types of segment errors, but for each d there is more than one "kind" of segment error (i.e. $a1_d$ with $f1(d)$ information bit errors, and $a2_d$ with $f2(d)$ information bit errors).

8.5 Viterbi Algorithm Implementation Issues

The Viterbi algorithm provides an efficient implementation of convolutional code MLSE decoding. However, there are potential problems. We now address several of these.

8.5.1 Reduced State Sequence Estimation & Per-Survivor Precessing

For a convolutional code, the number of trellis states M is equal to the number of possible bit combinations in the memory of its shift register. That is, $M = 2^{k(K-1)}$. So, for example, for a rate $\frac{2}{n}$ code (i.e. $k = 2$) with constraint length $K = 10$, there are $M = 262,144$ states/stage. With potentially M^2 branch costs to compute and M sets of up to M path costs to update and compare (to prune), it is clear that even with the Viterbi algorithm, MLSE decoding is not practical for codes with large memory depth. On the other hand, for good convolutional codes, d_{free} increases with increasing constraint length K. Because of this, there is interest in computation saving modifications and alternatives to the Viterbi algorithm for convolutional code decoding. In this Subsection we describe a modification to the Viterbi algorithm which trades off optimality for reduced computation. In the next Subsection, on sequential decoding, we consider suboptimum alternatives to Viterbi.

Reduced State Sequence Estimation (RSSE) is a general approach to reducing Viterbi algorithm computational requirements. As the name suggests, the idea is to reduce the number of trellis states. Two approaches have been proposed: one based on clustering groups of states into single states; and one based on truncating the length of the original (full trellis) states. Here we briefly overview the latter of these approaches.

As illustrated in Figure 85(a), denote the M states as $\mathbf{S}_a;\ a = 1, 2, \cdots M$. Each state at stage j represents a possible set of the $k(K-1)$ information bits, $\{x_{jk}, x_{jk-1}, \cdots, x_{(j-(K-1))k+1}\}$ in the encoder memory. Recall that the incremental MLSE cost calculation for a given branch uses the contents of the two states it connects. Specifically, the cost of a branch from state $\mathbf{S}_a$ at stage $j = m - 1$ to a state $\mathbf{S}_b$ at stage m will be some function of information bits, $\{x_{mk}, x_{mk-1}, \cdots, x_{(m-K)k+1}\}$, represented by these two states.

To reduce the number of states, consider truncating state vectors to length $T < k(K-1)$. There are now only 2^T states. Denote these reduced states as $\mathbf{S}'_a;\ a = 1, 2, \cdots, 2^T$. These states are illustrated in Figure 85(b). The trellis of these states will be smaller, with fewer branches, and thus requires less computation. Now each state at stage j represents a possible set of the T information bits, $\{x_{jk}, x_{jk-1}, \cdots, x_{jk-(T-1)}\}$. An incremental MLSE cost at stage $j = m$ still requires information bits $\{x_{mk}, x_{mk-1}, \cdots, x_{(m-K)k+1}\}$ (i.e. it is the same cost). Now, however, the reduced states terminating a branch do not represent the information bits required for the branch's cost. Specifically, a branch from state $\mathbf{S}'_a$ at stage $j = m-1$ to a state $\mathbf{S}'_b$ at stage m will be missing the information bits $\{x_{(m-1)k-T}, x_{(m-1)k-T-1}, \cdots, x_{(m-K)k+1}\}$.

Recall that the states of the trellis represent the information bits for hypothesized paths through the trellis (i.e. hypothesized information sequences). Consider the 2^{km} paths through the reduced trellis up to stage m, denoted $\mathbf{C}^i_j; j = 1, 2, \cdots, 2^{km}$. Consider any survivor (unpruned hypothesized) path into stage m. Figure 85(b) illustrates such a path as the thick solid line. Tracing this path back through the reduced trellis from stage m to stage 0, all of the bits this path represents are available in the (reduced) states that are transversed. The idea behind this truncated state approach to RSSE is that, for each survivor path into

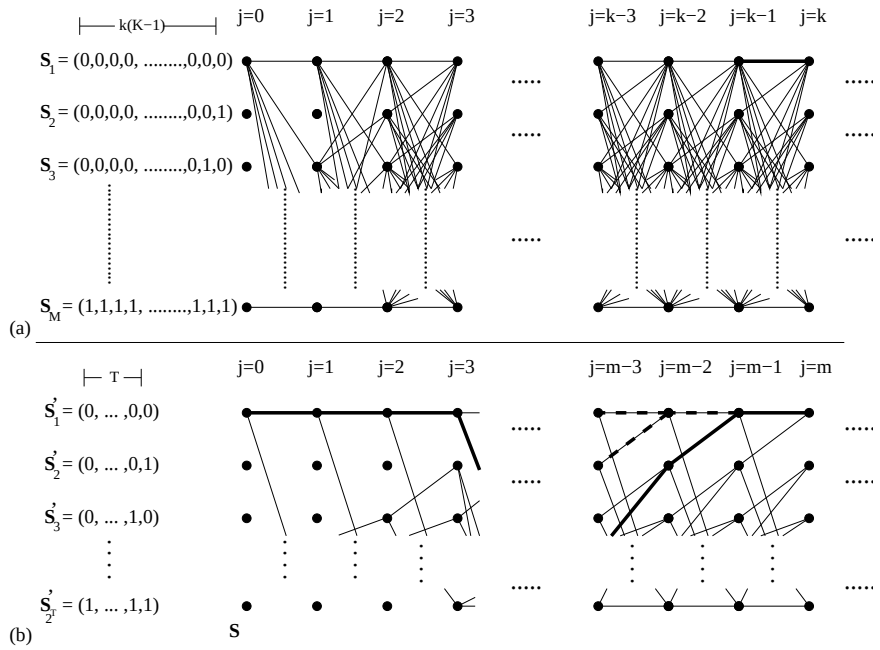


Figure 85: Illustrations of: a) a trellis with a large number of states; and b) a reduced state trellis.

stage m, use previous bits for that path, available in the previous states transversed by that path, in place of the bits not represented by the reduced states at stages m and $m-1$.

Note that different paths through a given branch will now incur different costs. As illustrated in Figure 85(b) using the thick solid and dashed lines, this is because different paths use bits from different previous reduced states. So the Viterbi pruning rule (i.e. keeping only the best path into a state) is no longer optimum. For this approach to RSSE, it can be effective to keep/extend more than one path into a state (i.e. keeping a list of paths at each state). The algorithm that does this is called list Viterbi. A more general approach to pruning and subsequent processing has been termed Per-Survivor Processing (PSP).

8.5.2 Trellis Truncation

In general, given received soft decision data $\mathsf{r}_j; j = 1, 2, \cdots, N$ or hard decision data $\mathsf{Y}_j; j = 1, 2, \cdots, N$, the MLSE estimate $\underline{\hat{\mathbf{X}}}_N$ of the information sequence can not be determined for any stage j until all data up to $j = N$ is processed. For continuous symbol estimation (i.e. $N = \infty$), and even for large finite N, this is impractical. In practice, at any stage m the trellis is "truncated" q stages into the past. This, as illustrated below in Figure 86, involves tracing the best path at stage m back to stage $m-q$ and outputting as the estimated information bits $\hat{x}_{(m-q)k}, \hat{x}_{(m-q)k-1}, \cdots, \hat{x}_{(m-q-1)k+1}$, the bits corresponding to the state at stage m transversed by this best path. A useful rule of thumb, which has been shown empirically to result in negligible performance loss, is $q \geq 5K$, where K is the constraint length.

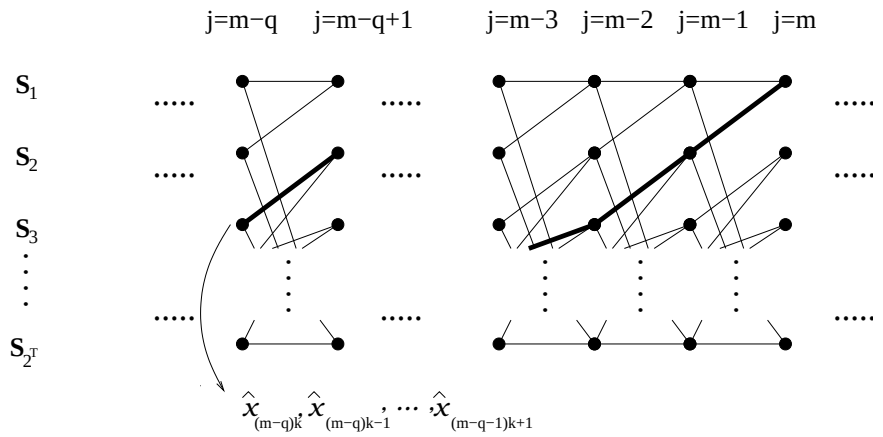


Figure 86: Trellis truncation for practical MLSE.

8.5.3 Cost Normalization

For continuous symbol estimation, the numerical values of the path costs can grow without bound as time progresses. In practice this problem is resolved by periodically (say at every P^{th} stage) subtracting the smallest path cost from all path costs.

8.6 Sequential Decoding

In Subsection 8.5.1 it was pointed out the the Viterbi algorithm is impractical when the number of trellis states becomes too large. RSSE approaches have gained more acceptance in other MLSE applications, including channel equalization and joint equalization and channel code decoding. Another effective approach to convolutional code decoding, which has been widely used since well before the invention of the Viterbi algorithm, is sequential decoding. Sequential decoding constitutes a class of suboptimum, MLSE based, tree path evaluation algorithms. These algorithms are still used today for deep-memory encoder applications, and in some ARQ systems because the can provide erasure information. Here we consider two sequential algorithms: the *stack* and *Fano* algorithms.

8.6.1 The Stack Algorithm

The stack algorithm is best described for decoding a convolutional code in terms of the code's tree diagram. A stack is an ordered list of tree paths and their metrics. Paths are listed in order of decreasing metric (i.e. the best is listed at the top of the stack). Basically, at each iteration, new paths are entered into the stack, the stack is reordered, and the lowest metric paths are deleted. This continues until a best path is identified which transverses the tree from the origin node (i.e. stage $j = 0$) to an end node (at $j = N$ where N is the last stage). Since paths of different lengths will be compared in the stack, the metric of a path must facilitate a fair comparison between paths, independent of length. Massey [3] established the following metric for a rate $R = \frac{k}{n}$ convolutional code:

$$CM^{(i)} \;=\; \sum_{m=1}^{m} \sum_{l=1}^{n} \left\{ \log_2 \left(\frac{p(r_{j,l}/\mathbf{C}_i)}{p(r_{j,l})} \right) \;-\; R \right\} \quad . \tag{57}$$

$CM^{(i)}$ is metric for path i, up to stage m, corresponding to code sequence $\mathbf{C}_i$. $p(r_{j,l}/\mathbf{C}_i)$ is the PDF of received data $r_{j,l}$, the sampled $l-th$ matched filter output for stage j, conditioned on $\mathbf{C}_i$. $p(r_{j,l})$ is the PDF of received data $r_{j,l}$. Note that metrics can be computed recursively, adding an incremental term as a path is extended to a next stage.

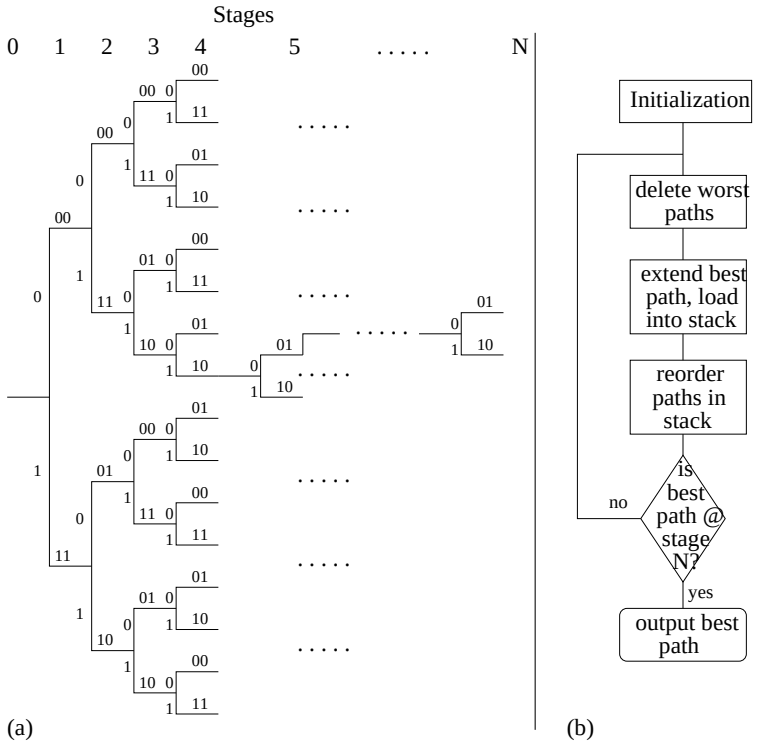


Figure 87: (a) $R = 1/2$ convolutional encoder tree diagram; (b) the stack algorithm.

Figure 87 shows a block diagram of the process. The algorithm is initialized by extending all paths from the origin to the first stage, while loading the path node sequences (i.e. path descriptions) and metrics into the stack. Then, for each iteration of the algorithm, the worst (bottom most) paths in the stack are deleted. This can be accomplished by comparing metrics to a threshold, or by limiting the stack depth. Then, the best path is extended to all possible nodes at the next stage and loaded into the top of the stack. The stack is then reordered. If the best path terminates at the last stage, it is taken as the solution. Otherwise, the iteration is repeated.

The stack algorithm will usually generate the MLSE estimate. Thus its performance is similar to that of the Viterbi algorithm. At high SNR, it usually requires less computation than Viterbi. However, at lower SNR, the stack algorithm will look at a lot of tree nodes (i.e. computing metrics and putting them into the stack), and computation can exceed Viterbi significantly. Memory requirements for the stack algorithm is significant.

8.6.2 The Fano Algorithm

Compared to the stack algorithm, the Fano algorithm works with a single path through the code tree, so very little memory is required. However, typically more paths are examined,

so computational requirements are higher. The same path metric equation is employed.

At each iteration of the algorithm, the current path is extended forward to the best next node (i.e. by computing metrics for all possible next nodes, and selecting the largest). If this extended path is at the terminal stage N, that path is outputted as the estimate. If not, its metric is compared to a threshold T. If the metric exceeds the threshold, then the next iteration begins (i.e. the path is again extended). If the metric does not exceed the threshold, then the path is shortened, another path is considered. In certain situations, the threshold is reduced so as to assure that a solutions is eventually achieved.

As with the stack algorithm, the Fano algorithm usually achieves the MLSE estimate, and required computation increases as the SNR decreases.

8.6.3 The Feedback Decision Decoding Algorithm

This algorithm looks ahead m stages to make a final information bit decision at the current stage. Processing at any stage $j-1$ starts with a single survivor state. All paths from that state up to stage $j+m$ are evaluated and compared. The best path then determines the one state kept at stage j, and the bit estimate at stage j corresponds to that state.

8.7 Techniques for Constructing More Complex Convolutional Codes

We have discussed puncturing, interleaving and concatenated codes previously in the context of block encoding. Issues and techniques are basically the same for convolutional codes.

References

[1] J. Proakis and M. Salehi, *Digital Communications 5-th ed.* McGraw Hill, 2008.

[2] S. Lin and D. J. Costello, *Error Control Coding 2-nd ed.* Pearson Prentice Hall, 2004.

[3] J. Massey, "Variable-length codes and the fano metric," *IEEE Trans. on Information Theory*, vol. IT-18, pp. 196–198, Jan 1972.