Chapter 5

Convolutional codes and their decoding

5.1 History

It was in 1955 that Peter Elias introduced the notion of convolutional code [5.5]. The example of an encoder described is illustrated in Figure 5.1. It is a systematic encoder, that is, the coded message contains the message to be transmitted, to which redundant information is added. The message is of infinite length, which at first sight limits the field of application of this type of code. It is however easy to adapt it for packet transmissions thanks to *tail-biting* techniques.

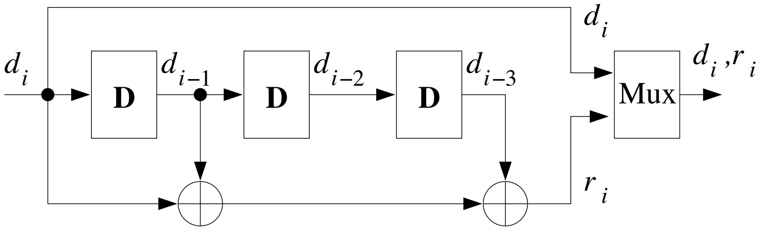


Figure 5.1 – Example of a convolutional encoder.

The encoder presented in Figure 5.1 is designed around a shift register with three memory elements. The redundancy bit at instant i, denoted r_i is constructed with the help of a modulo 2 sum of the information at instant i, d_i and the data present at instants $i-1$ and $i-3$ (d_{i-1} and d_{i-3}). A multiplexer plays the role of a parallel to serial converter and provides the result of the encoding at a rate twice that of the rate at the input. The coding rate of this encoder is

1/2 since, at each instant i, it receives data d_i and delivers two elements at the output: d_i (systematic part) and r_i (redundant part).

It was not until 1957 that the first algorithm capable of decoding such codes appeared. Invented by Wozencraft [5.15], this algorithm, called sequential decoding, was then improved by Fano [5.6] in 1963. Four years later, Viterbi introduced a new algorithm that was particularly interesting when the length of the shift register of the encoder is not too large [5.14]. Indeed, the complexity of the Viterbi algorithm increases exponentially with the size of this register whereas the complexity of the Fano algorithm is almost independent of it.

In 1974, Bahl, Cocke, Jelinek and Raviv presented a new algorithm [5.1] capable of associating a probability with the binary decision. This property is very widely used in the decoding of concatenated codes and more particularly turbocodes, which have brought this algorithm back into favour. It is now referred to in the literature in one of these three ways: BCJR (initials of the inventors), MAP (Maximum *A Posteriori*) or APP (*A Posteriori* Probability). The MAP algorithm is rather complex to implement in its initial version, and it exists in simplified versions, the most common ones being presented in Chapter 7.

In parallel with these advances in decoding algorithms, a number of works have treated the construction of convolutional encoders. The aim of these studies has not been to decrease the complexity of the encoder, since its implantation is trivial. The challenge is to find codes with the highest possible error correction capability. In 1970, Forney wrote a reference paper on the algebra of convolutional codes [5.7]. It showed that a *good* convolutional code is not necessarily systematic and suggested a construction different from that of Figure 5.1. For a short time, that paper took systematic convolutional codes away from the field of research on channel coding.

Figure 5.2 gives an example of a non-systematic convolutional encoder. Unlike the encoder in Figure 5.1, the data are not present at the output of the encoder and are replaced by a modulo 2 sum of the data at instant i, d_i, and of the data present at instants $i-2$ and $i-3$ (d_{i-2} and d_{i-3}). The rate of the encoder remains unchanged at 1/2 since the encoder always provides two elements at the output: $r_i^{(1)}$ and $r_i^{(2)}$, at instant i.

When Berrou *et al.* presented their work on turbocodes [5.4], they rehabilitated systematic convolutional codes by using them in a recursive form. The interest of recursive codes is presented in Sections 5.2 and 5.3. Figure 5.3 gives an example of an encoder for recursive systematic convolutional codes. The original message being transmitted (d_i), the code is therefore truly systematic. A feedback loop appears, the structure of the encoder now being similar to that of pseudo-random sequence generators.

This brief overview has allowed us to present the three most commonly used families of convolutional codes: systematic, non-systematic, and recursive systematic codes. The next two sections tackle the representation and performance

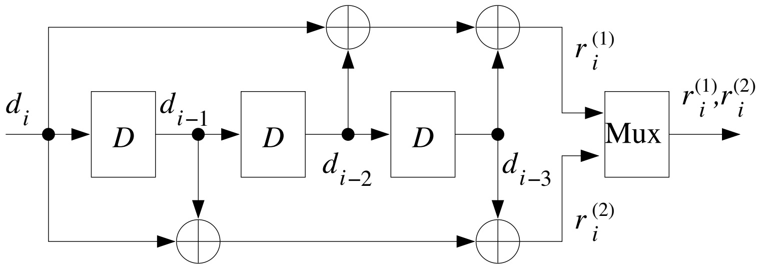


Figure 5.2 – Example of an encoder for non-systematic convolutional codes.

of convolutional codes. They give us the opportunity to compare the properties of these three families. The decoding algorithms most commonly used in current systems are presented in Section 5.4. Finally, Section 5.5 tackles the main *tail-biting* and *puncturing* techniques.

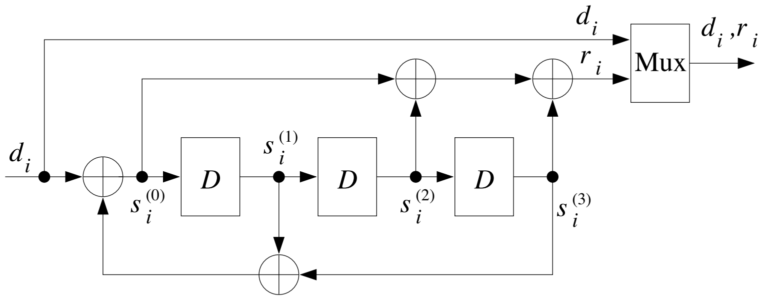


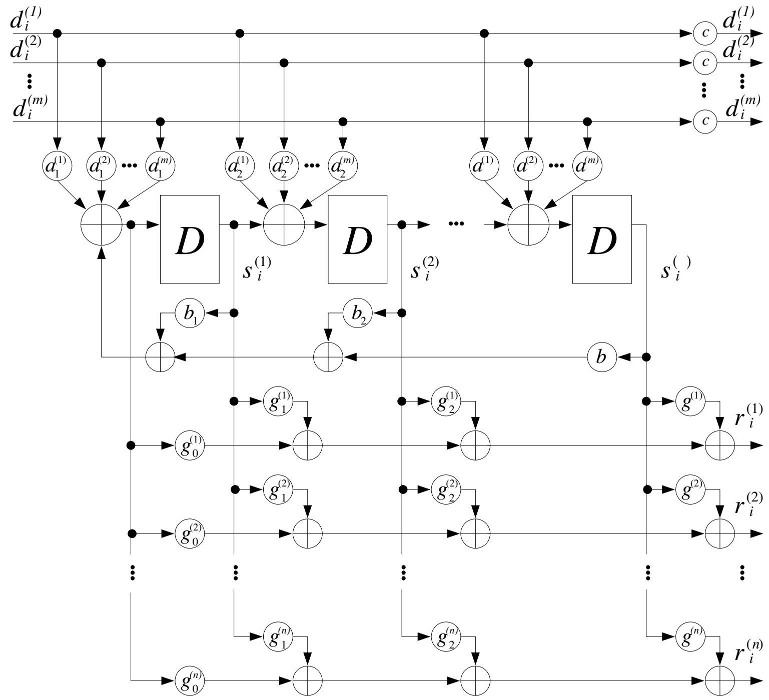
Figure 5.3 – Example of a encoder for recursive systematic convolutional codes.

5.2 Representations of convolutional codes

This chapter makes no claim to tackle the topic of convolutional codes exhaustively. Non-binary or non-linear codes are not treated, nor are encoders with several registers. Only the most commonly used codes, in particular for the construction of turbocodes, are introduced. The reader wishing to go further into the topic can, for example, refer to [5.11].

5.2.1 Generic representation of a convolutional encoder

Figure 5.4 gives a sufficiently general model for us to represent all the convolutional codes studied in this chapter. At each instant i, it receives a vector $\mathbf{d}_i$ of m bits at its input. The code thus generated is a binary code. However,



$d_i^{(1)}$
$d_i^{(2)}$
$d_i^{(m)}$
c
$d_1^{(1)}$ $d_1^{(2)}$ $d_1^{(m)}$
$d_2^{(1)}$ $d_2^{(2)}$ $d_2^{(m)}$
$d^{(1)}$ $d^{(2)}$ $d^{(m)}$
D
$s_i^{(1)}$
$s_i^{(2)}$
$s_i^{(\ )}$
b_1
b_2
b
$g_0^{(1)}$ $g_1^{(1)}$ $g_2^{(1)}$ $g^{(1)}$
$g_0^{(2)}$ $g_1^{(2)}$ $g_2^{(2)}$ $g^{(2)}$
$g_0^{(n)}$ $g_1^{(n)}$ $g_2^{(n)}$ $g^{(n)}$
$r_i^{(1)}$
$r_i^{(2)}$
$r_i^{(n)}$

Figure 5.4 – Generic representation of an encoder for convolutional codes.

to simplify the writing, we will call it an *m-binary* and *double-binary* code if $m = 2$ (as in the example presented in Figure 5.5). When $c = 1$, the code generated is systematic, since $\mathbf{d}_i$ is transmitted at the output of the encoder. The code is thus made up of the systematic part $\mathbf{d}_i$ on m bits and the redundancy $\mathbf{r}_i$ on n bits. The coding rate is then $R = m/(m+n)$. If $c = 0$, the code is non-systematic and the rate becomes $R = m/n$.

The non-systematic part is constructed with the help of a shift register made up of ν flip-flops and of binary adders, in other words, of *XOR* gates. We then define an important characteristic of convolutional codes: the *constraint length*, here equal to $\nu + 1$ (some authors denote it ν, which implies that the register is then made up of $\nu - 1$ flip-flops). The register at instant i is characterized by the ν bits $s_i^{(1)}, s_i^{(2)}, \ldots, s_i^{\nu}$ memorized: they define its *state*, that we can thus code on ν bits and represent in the form of a vector $\mathbf{s}_i = (s_i^{(1)}, s_i^{(2)}, \cdots, s_i^{\nu})$. This type of convolutional encoder thus has 2^{ν} possible state values, that we often denote in natural binary or binary decimal form. Thus the state of an encoder made up of three flip-flops can take $2^3 = 8$ values. If $s_1 = 1$, $s_2 = 1$ and $s_3 = 0$, the encoder is in state 110 in natural binary, that is, 6 in decimal.

Using the coefficients $a_j^{(l)}$, each of the m components of the vector $\mathbf{d}_i$ is selected or not as the term of an addition with the content of a previous flip-flop (except in the case of the first flip-flop) to provide the value to be stored in the following flip-flop. The new content of a flip-flop thus depends on the current input and on the content of the previous flip-flop. The case of the first flip-flop has to be considered differently. If all the b_j coefficients are null, the input is the result of the sum of the only components selected of $\mathbf{d}_i$. In the opposite case, the contents of the flip-flops selected by the non-null b_j coefficients are added to the sum of the components selected of $\mathbf{d}_i$. The code thus generated is recursive. Thus, the succession of states of the register depends on the departure state and on the succession of data at the input. The components of redundancy $\mathbf{r}_i$ are finally produced by summing the content of the flip-flops selected by the coefficients g.
Let us consider some examples.

— The encoder represented in Figure 5.1 is systematic binary, therefore $m = 1$ and $c = 1$. Moreover, all the $a_j^{(l)}$ coefficients are null except $a_1^{(1)} = 1$. This encoder is not recursive since all the coefficients b_j are null. The redundancy (or parity) bit is defined by $g_0^{(1)} = 1$, $g_1^{(1)} = 1$, $g_2^{(1)} = 0$ and $g_3^{(1)} = 1$.

— In the case of the non-systematic non-recursive binary (here called "classical") encoder in Figure 5.2, $m = 1$, $c = 0$; among the $a_j^{(l)}$, only $a_1^{(1)} = 1$ is non-null and $b_j = 0 \quad \forall j$. Two parity bits come from the encoder and are defined by $g_0^{(1)} = 1$, $g_1^{(1)} = 1$, $g_2^{(1)} = 0$, $g_3^{(1)} = 1$ and $g_0^{(2)} = 1$, $g_1^{(2)} = 0$, $g_2^{(2)} = 1$, $g_3^{(2)} = 1$.

— Figure 5.3 presents a recursive systematic binary encoder ($m = 1$, $c = 1$ and $a_1^{(1)} = 1$). The coefficients of the recursivity loop are then $b_1 = 1$, $b_2 = 0$, $b_3 = 1$ and those of the redundancy are $g_0^{(1)} = 1$, $g_1^{(1)} = 0$, $g_2^{(1)} = 1$, $g_3^{(1)} = 1$.

— Figure 5.5 represents a recursive systematic double-binary encoder. The only coefficients that differ from the previous case are the $a_j^{(l)}$: coefficients $a_1^{(1)}$, $a_1^{(2)}$, $a_2^{(2)}$ and $a_3^{(2)}$ are equal to 1, the other $a_j^{(l)}$ are null.

To define an encoder, it is not however necessary to make a graphic representation since knowledge of the parameters presented in Figure 5.4 is sufficient. A condensed representation of these parameters is known as generator polynomials. This notation is presented in the following paragraph.

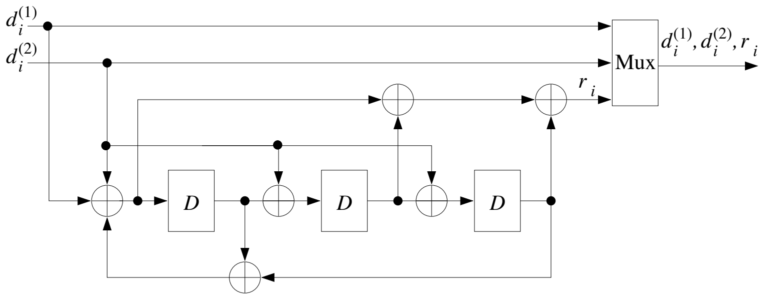


Figure 5.5 – Example of a recursive systematic double-binary convolutional encoder.

5.2.2 Polynomial representation

Let us first consider a classical (non-systematic and non-recursive) binary code: $c = 0$, all the coefficients b_j are null and $m = 1$. The knowledge of $\left[g_j^{(k)}\right]_{j=0..\nu}^{k=1..n}$ then suffices to describe the code. The coefficients $\left[g_j^{(k)}\right]_{j=0..\nu}$ therefore define n generator polynomials $G^{(k)}$ in D (*Delay*) algebra:

$$G^{(k)}(D) = \sum_{j=0...\nu} g_j^{(k)} D^j \tag{5.1}$$

Let us take the case of the encoder defined in Figure 5.2. The outputs $r_i^{(1)}$ and $r_i^{(2)}$ are expressed as functions of the successive data d as follows:

$$r_i^{(1)} = d_i + d_{i-2} + d_{i-3} \tag{5.2}$$

which can also be written, via the transform in D:

$$r^{(1)}(D) = G^{(1)}(D) \; d(D) \tag{5.3}$$

with $G^{(1)}(D) = 1 + D^2 + D^3$, the first generator polynomial of the code and $d(D)$ the transform in D of the message to be encoded. Likewise, the second generator polynomial is $G^2(D) = 1 + D + D^3$.

These generator polynomials can also be resumed by the series of their coefficients, (1011) and (1101) respectively, generally denoted in octal representation, $(13)_{octal}$ and $(15)_{octal}$ respectively. In the case of a non-recursive systematic code , like the example in Figure 5.1, the generator polynomials are expressed according to the same principle. In this example, the encoder has generator polynomials $G^{(1)}(D) = 1$ and $G^{(2)}(D) = 1 + D + D^3$.

To define the generator polynomials of a recursive systematic code is not straightforward. Let us consider the example of Figure 5.3. The first generator polynomial is trivial since the code is systematic. To identify the second, we must note that

$$s_i^{(0)} = d_i + s_i^{(1)} + s_i^{(3)} = d_i + s_{i-1}^{(0)} + s_{i-3}^{(0)} \tag{5.4}$$

and that:

$$r_i = s_i^{(0)} + s_i^{(2)} + s_i^{(3)} = s_i^{(0)} + s_{i-2}^{(0)} + s_{i-3}^{(0)} \tag{5.5}$$

which is equivalent to:

$$\begin{aligned} d_i &= s_i^{(0)} + s_{i-1}^{(0)} + s_{i-3}^{(0)} \\ r_i &= s_i^{(0)} + s_{i-2}^{(0)} + s_{i-3}^{(0)} \end{aligned} \tag{5.6}$$

This result can be reformulated by introducing the transform in D:

$$\begin{aligned} d(D) &= G^{(2)}(D)s(D) \\ r(D) &= G^{(1)}(D)s(D) \end{aligned} \tag{5.7}$$

where $G^{(1)}(D)$ and $G^{(2)}(D)$ are the generator polynomials of the code shown in Figure 5.2, which leads to

$$\begin{aligned} s(D) &= \tfrac{d(D)}{G^{(2)}(D)} \\ \\ r(D) &= \tfrac{G^{(1)}(D)}{G^{(2)}(D)}d(D) \end{aligned} \tag{5.8}$$

Thus, a recursive systematic code can easily be derived from a non-systematic non-recursive code. The codes generated by such encoders can be represented graphically according to three models: the tree, the trellis and the state machine.

5.2.3 Tree of a code

The first graphic representation of a code and certainly the least pertinent for the rest of the chapter is the tree representation. It enables all the sequences of possible states to be presented. The root is associated with the initial state of the encoder. From the root, we derive all the possible successive states as a function of input d_i of the encoder. The branch linking a father state to a son state is labelled with the value of the outputs of the encoder during the associated transition. This principle is iterated for each of the strata and so forth. The tree diagram associated with the systematic encoder of Figure 5.1 is illustrated in Figure 5.6. This type of diagram will not be used in what follows and the only use that is made of it concerns a sequential decoding algorithm (Fano's algorithm) not treated in this book.

5.2.4 Trellis of a code

The most common representation of a convolutional code is the trellis diagram. It is of major importance both for defining the properties of a code and for decoding it, as we shall see in the next part of Chapter 5.

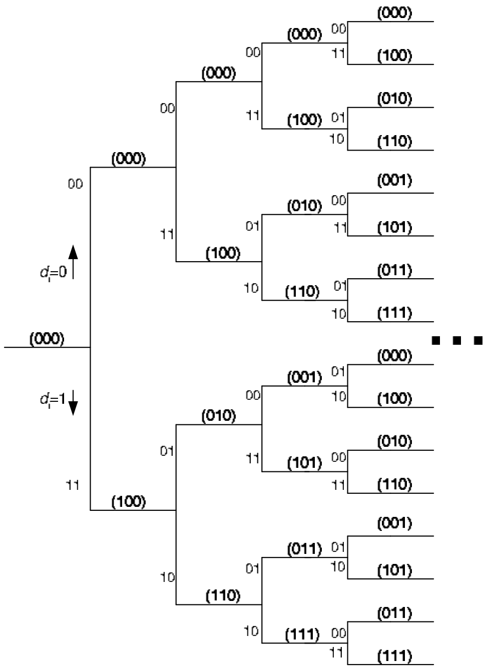


Figure 5.6 – Tree diagram of the code of polynomials $[1, 1+D+D^3]$. The binary pairs indicate the outputs of the encoder and the values in brackets are the future states.

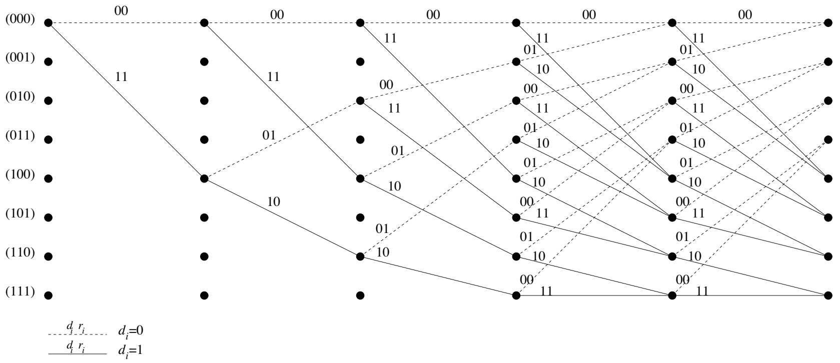


Figure 5.7 – Trellis diagram of a code with generator polynomials $[1, 1+D+D^3]$.

At an instant i, the state of a convolutional encoder can take 2^ν values. Each of these possible values is represented by a node. With each instant i is associated a states-nodes column and according to input d_i, the encoder transits from a state $\mathbf{s}_i$ to $\mathbf{s}_{i+1}$ while delivering the coded bits. This transition between

two states is represented by an arc between the two associated nodes and labelled with the outputs of the encoder. In the case of a binary code, the transition on an input at 0 (resp. 1) is represented by a dotted (resp. solid) line. The succession of $\mathbf{s}_i$ states up to instant t is represented by the different paths between the initial state and the different possible states at instant t.
Let us show this with the example of the systematic encoder of Figure 5.1. Hypothesizing that the initial state $\mathbf{s}_0$ is state (000) :

- If $d_1 = 0$ then the following state, $\mathbf{s}_1$, is also (000). The transition is represented by a dotted line and labelled in this first case 00, the value of the encoder outputs;
- If $d_1 = 1$, then the following state, $\mathbf{s}_1$, is (100). The transition is represented by a solid line and here labelled 11.
- We must next envisage the four possible transitions: $\mathbf{s}_1 = (000)$ if $d_2 = 0$ or $d_2 = 1$ and $\mathbf{s}_1 = (100)$ if $d_2 = 0$ or $d_2 = 1$.

Iterating this construction, we reach the representation, in Figure 5.7, of all the possible successions of states from the initial state to instant 5, without the unlimited increase of the tree diagram.

A complete section of the trellis suffices to characterize the code. The trellis section of the previous code is thus shown in Figure 5.8(a). Likewise, the encoders presented in Figures 5.2 and 5.3 are associated with trellis sections, illustrated in Figures 5.8(b) and 5.8(c) respectively.

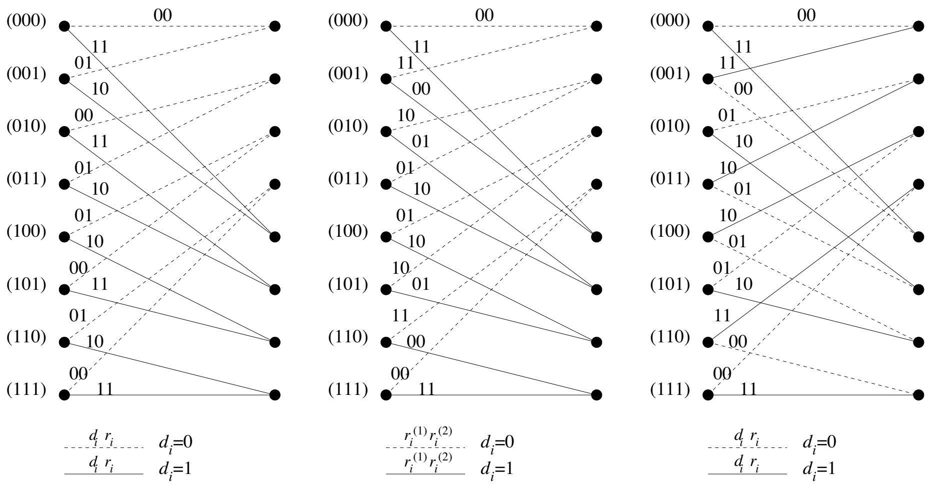


Figure 5.8 – Trellis sections of the codes with generator polynomials $[1, 1 + D + D^3]$ (a), $[1 + D^2 + D^3, 1 + D + D^3]$ (b) and $[1, (1 + D^2 + D^3)/(1 + D + D^3)]$ (c)

Such a representation shows the basic pattern of these trellises: the butterfly, so called because of its shape. Each of the sections of Figures 5.8 is thus made up of 4 butterflies (the transitions of states 0 and 1 towards states 0 and 4 make up one). The butterfly structure of the three trellises illustrated is identical but the sequences coded differ. It should be noted in particular that all the transitions arriving at the same node of a trellis of a non-recursive code are due to the same value at the input of the encoder. Thus, among the two non-recursive examples treated (Figures 5.8(a) and 5.8(b)), a transition associated with a 0 at the input necessarily arrives at one of the states between 0 and 3 and a transition with a 1 arrives at one of the states between 4 and 7. It is different in the case of a recursive code (like the one presented in Figure 5.8(c)): each state allows one incident transition associated with an input with a 0, and another one associated with 1. We shall see the consequences of this in Section 5.3.

5.2.5 State machine of a code

To represent the different transitions between the states of an encoder, there is a final representation, that of a state machine. The convention for defining the transition branches is identical to that used in the previous section. Only 2^ν nodes are represented, independently of instant i, the previous representation being translated into a trellis section. The encoders of Figures 5.1, 5.2 and 5.3 thus allow a representation in the form of a state machine illustrated in Figures 5.9, 5.10 and 5.11 respectively.

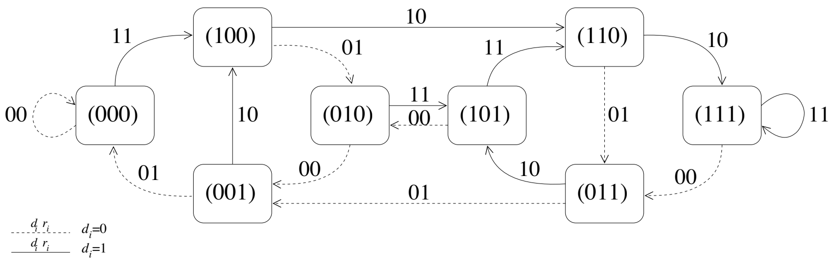


Figure 5.9 – State machine for a code with generator polynomials $[1, 1 + D + D^3]$.

This representation is particularly useful for determining the transfer function and the distance spectrum of a convolutional code (see Section 5.3). It is now possible to see a notable difference between a state machine of a recursive code and that of a non-recursive code: the existence and the number of cycles[1] on an all-zero sequence at the input.

In the case of two non-recursive state machines, there is a single cycle on a null sequence at the input: the loop on state 0.

[1] A cycle is a succession of states such that the initial state is also the final state.

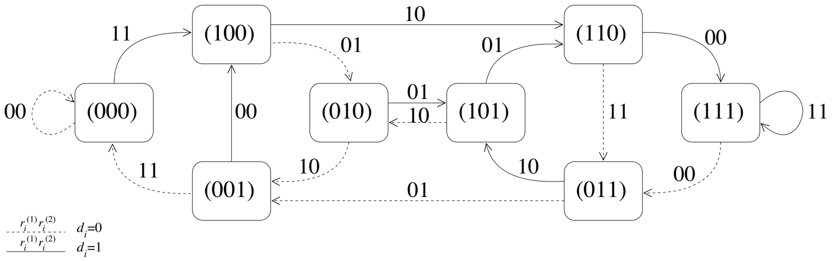


Figure 5.10 – State machine for a code with generator polynomials $[1 + D^2 + D^3, 1 + D + D^3]$.

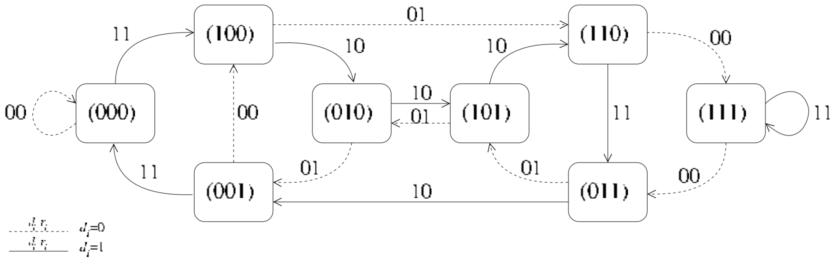


Figure 5.11 – State machine for a code with generator polynomials $[1, (1+D^2+D^3)/(1+D + D^3)]$.

However, the recursive state machine allows another cycle on a null sequence at the input: state 4 $\rightarrow$ state 6 $\rightarrow$ state 7 $\rightarrow$ state 3 $\rightarrow$ state 5 $\rightarrow$ state 2 $\rightarrow$ state 1 $\rightarrow$ state 4.

Moreover, this cycle is linked to the loop on state 0 by two transitions associated with inputs at 1 (transitions 0 $\rightarrow$ 4 and 1 $\rightarrow$ 0). There therefore exists an infinite number of input sequences with Hamming weight [2] equal to 2 producing a cycle on state 0. This weight 2 is the minimum weight of any sequence that makes the recursive encoder leave state 0 and return to zero. Because of the linearity of the code (see Chapter 1), this value of 2 is also the smallest distance that can separate two sequences with different inputs that make the encoder leave the same state and return to the same state.

In the case of non-recursive codes, the Hamming weight of the input sequences allowing a cycle on state 0 can only be 1 (state 0 $\rightarrow$ state 4 $\rightarrow$ state 2 $\rightarrow$ state 1 $\rightarrow$ state 0). This distinction is essential for understanding the interest of recursive codes used alone (see Section 5.3) or in a turbocode structure (see Chapter 7).

[2] The Hamming weight of a binary sequence is equal to the number of bits equal to 1.

5.3 Code distances and performance

5.3.1 Choosing a good code

As a code is exploited for its correcting capacities, we have to be able to estimate these in order to make a judicious choice of one code rather than another, according to the application targeted. Among the bad possible choices, *catastrophic codes* are such that a finite number of errors at the input of the decoder can produce an infinite number of errors at the output of the decoder, which explains their name. One main property of these codes is that there exists at least one input sequence of infinite weight that generates a coded sequence of finite weight: systematic codes therefore cannot be catastrophic. These codes can be identified very simply if they have a rate of the form $R = 1/N$. We can then show that the code is catastrophic if the largest common divisor (L.C.D.) of its generator polynomials is different from unity. Thus, the code with generator polynomials $G^{(1)}(D) = 1 + D + D^2 + D^3$ and $G^{(2)}(D) = 1 + D^3$ is catastrophic since the L.C.D. is $1 + D$.

However, choosing a convolutional code cannot be limited to the question "Is it catastrophic?". By exploiting the graphic representations introduced above, the properties and performance of codes can be compared.

5.3.2 *RTZ* sequences

Since convolutional codes are linear, to determine the distances between the different coded sequences amounts to determining the distances between the non-null coded sequences and the "all zero" sequence. Therefore, it suffices to calculate the Hamming weight of all the coded sequences that leave from state 0 and that return to it. These sequences are called *Return To Zero* (RTZ) sequences. The smallest Hamming weight thus obtained is called the *free distance* of the code. The minimum Hamming distance of a convolutional code is equal to its free distance from a certain length of coded sequence. In addition, the number of RTZ sequences that have the same weight is called the multiplicity of this weight.

Let us consider the codes that have been used as examples so far. Each RTZ sequence of minimum weight is shown in bold in Figures 5.12, 5.13 and 5.14.

The non-recursive systematic code has an RTZ sequence with minimum Hamming weight equal to 4. The free distance of this code is therefore equal to 4. On the other hand, as the classical code and the recursive systematic code each possess two RTZ sequences with minimum weight 6, their free distance is therefore 6. The correction capacity of non-recursive non-systematic and recursive systematic codes is therefore better than that of the non-recursive systematic code.

It is interesting, in addition, to compare the weights of the sequences at the input associated with the RTZ sequences with minimum weight. In the case of

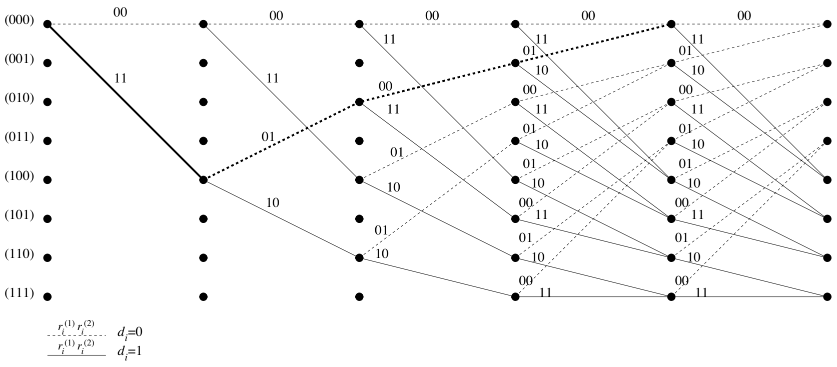


Figure 5.12 – RTZ sequence (in bold) defining the free distance of the code with generator polynomials $[1, 1 + D + D^3]$.

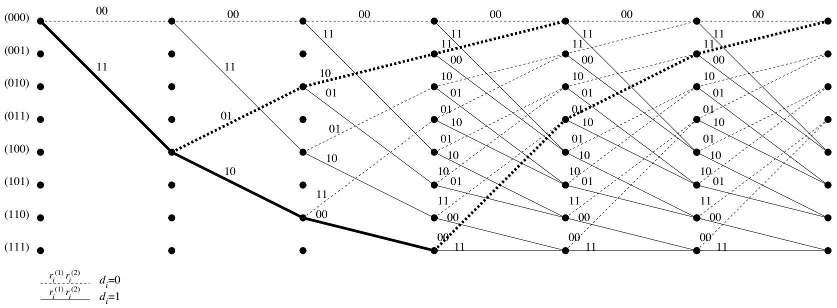


Figure 5.13 – RTZ sequences (in bold) defining the free distance of the code with generator polynomials $[1 + D^2 + D^3, 1 + D + D^3]$.

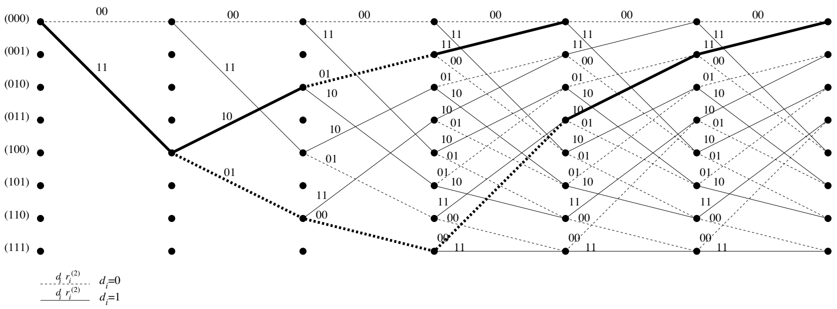


Figure 5.14 – RTZ sequences (in bold) defining the free distance of the code with generator polynomials $[1, (1 + D^2 + D^3)/(1 + D + D^3)]$.

the non-recursive systematic code, the only sequence of this type has a weight equal to 1, which means that if the RTZ sequence is decided instead of the transmitted "all zero" sequence, only one bit is erroneous. In the case of the classical code, one sequence at the input has a weight of 1 and another a weight of 3: one or three bits are therefore wrong if such an RTZ sequence is decoded. In the case of the recursive systematic code, the RTZ sequences with minimum weight have an input weight of 3.

Knowledge of the minimum Hamming distance and of the input weight associated with it is not sufficient to closely evaluate the error probability at the output of the decoder of a simple convolutional code. It is necessary to compute the distances, beyond the minimum Hamming distance, and their weight in order to make this evaluation. This computation is called the *distance spectrum*.

5.3.3 Transfer function and distance spectrum

The error correction capability of a code depends on all the RTZ sequences, which we will consider in the increasing order of their weight. Rather than computing them by reading the graphs, it is possible to establish the transfer function of the code. The latter is obtained from the state transition diagram in which the initial state (000) is cut into two states a_e and a_s, which are no other than the initial state and the arrival state of any RTZ sequence.

Let us illustrate the computation of the transfer function with the example of the systematic code of Figure 5.1, whose state transition diagram is again represented in Figure 5.15.

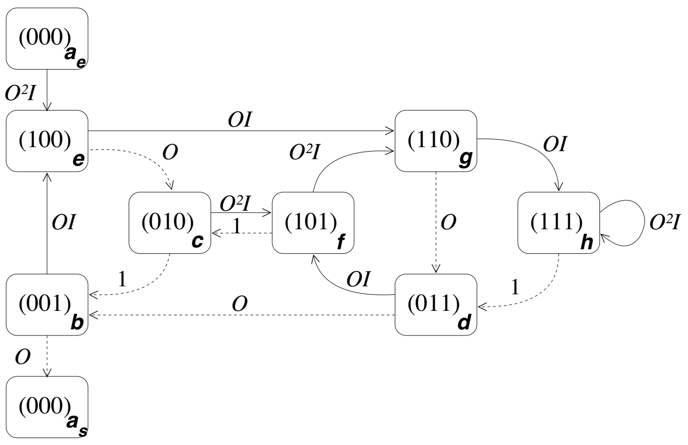


Figure 5.15 – Machine state of the code $[1, 1 + D + D^3]$, modified for the computation of the associated transfer function.

Each transition has a label O^iI^j, where i is the weight of the sequence coded and j that of the sequence at the input of the encoder. In our example, j can take the value 0 or 1 according to the level of the bit at the input of the encoder at each transition and i varies between 0 and 2, since 4 coded symbols are possible (00, 01, 10, 11), with weights between 0 and 2.

The transfer function of the code $T(O, I)$ is then defined by:

$$T(O, I) = \frac{a_s}{a_e} \tag{5.9}$$

To establish this function, we have to solve the system of equations coming from the relations between the 9 states (ae, b, c... h and as):

$$\begin{aligned}
&b = c + Od \\
&c = Oe + f \\
&d = h + Dg \\
&e = O^2 I a_e + OIb \\
&f = O^2 I c + OId \\
&g = OIe + O^2 If \\
&h = O^2 Ih + OIg \\
&a_s = Ob
\end{aligned} \tag{5.10}$$

Using a formal computation tool, it is easy to arrive at the following result:

$$T(O, I) = \frac{-I^4O^{12} + (3I^4 + I^3)O^{10} + (-3I^4 - 3I^3)O^8 + (I^4 + 2I^3)O^6 + IO^4}{I^4O^{10} + (-3I^4 - I^3)O^8 + (3I^4 + 4I^3)O^6 + (-I^4 - 3I^3)O^4 - 3IO^2 + 1}$$

$T(O, I)$ can then be developed as a series:

$$\begin{aligned}
T(O, I) = \;& IO^4 \\
&+ (I^4 + 2I^3 + 3I^2)O^6 \\
&+ (4I^5 + 6I^4 + 6I^3)O^8 \\
&+ (I^8 + 5I^7 + 21I^6 + 24I^5 + 17I^4 + I^3)O^{10} \\
&+ (7I^9 + 30I^8 + 77I^7 + 73I^6 + 42I^5 + 3I^4)O^{12} \\
&+ \cdots
\end{aligned} \tag{5.11}$$

This enables us to observe that an RTZ sequence with weight 4 is produced by an input sequence with weight 1, that the RTZ sequences with weight 6 are produced by a sequence with weight 4, by two sequences with weight 3 and by three sequences with weight 2, etc.

In the case of the classical code mentioned above, the transfer function is:

$$\begin{aligned}
T(O, I) = \;& (I^3 + I)O^6 \\
&+ (2I^6 + 5I^4 + 3I^2)O^8 \\
&+ (4I^9 + 16I^7 + 21I^5 + 8I^3)O^{10} \\
&+ (8I^{12} + 44I^{10} + 90I^8 + 77I^6 + 22I^4)O^{12} \\
&+ (16I^{15} + 112I^{13} + 312I^{11} + 420I^9 + 265I^7 + 60I^5)O^{14} \\
&+ \cdots
\end{aligned} \tag{5.12}$$

Likewise, the recursive systematic code already studied has as its transfer function:

$$\begin{aligned} T(O,I) = \; & 2I^3O^6 \\ & +(I^6+8I^4+I^2)O^8 \\ & +(8I^7+33I^5+8I^3)O^{10} \\ & +(I^{10}+47I^8+145I^6+47I^4+I^2)O^{12} \\ & +(14I^{11}+254I^9+649I^7+254I^5+14I^3)O^{14} \\ & +\cdots \end{aligned} \tag{5.13}$$

Comparing the transfer functions from the point of view of the monomial with the smallest degree allows us to appreciate the error correction capability at very high signal to noise ratio (asymptotic behaviour). Thus, the non-recursive systematic code is weaker than its rivals since it has a lower minimum distance. A classical code and its equivalent recursive systematic code have the same free distance, but their monomials of minimal degree differ. The first is in $(I^3+I)O^6$ and the second in $2I^3O^6$. This means that with the classical code an input sequence with weight 3 and another with weight 1 produce an RTZ sequence with weight 6 whereas with the recursive systematic code two sequences with weight 3 produce an RTZ sequence with weight 6. Thus, if an RTZ sequence with minimum weight is introduced by the noise, the classical code will introduce one or three errors, whereas its recursive systematic code will introduce three or three other errors. In conclusion, the probability of a binary error on such a sequence is lower with a classical code than with a recursive systematic code, which explains that the former will be slightly better at high signal to noise ratio. Things are generally different when the codes are punctured (see Section 5.5) in order to have higher rates [5.13].

To compare the performance of codes with low signal to noise ratio, we must consider all the monomials. Let us take the example of the monomial in O^{12} for the non-recursive systematic code, the classical code and the recursive systematic code, respectively:

$$\begin{aligned} &(7I^9+30I^8+77I^7+73I^6+42I^5+3I^4)O^{12} \\ &(8I^{12}+44I^{10}+90I^8+77I^6+22I^4)O^{12} \\ &(I^{10}+47I^8+145I^6+47I^4+I^2)O^{12} \end{aligned}$$

If 12 errors are introduced by the noise on the channel, 232 RTZ sequences are "available" as errors for the first code, 241 for the second and 241 again for the third. It is therefore (a little) less probable that an RTZ sequence will appear if the code used is the non-recursive systematic code. Moreover, the error expectancy per RTZ sequence of the three codes is 6.47, 7.49 and 6.00, respectively: the recursive systematic code therefore introduces, on average, fewer decoding errors than the classical code on RTZ sequences with 12 errors on the frame coded. This is also true for higher degree monomials. Recursive and non-recursive systematic codes are therefore more efficient at low signal to

d	6	8	10	12	14	...
$\omega(d)$	6	40	245	1446	8295	...

Table 5.1 – First terms of the spectrum of the recursive systematic code with generator polynomials $[1, (1 + D^2 + D^3)/(1 + D + D^3)]$.

noise ratio than the classical code. Moreover, we find the monomials I^2O^{8+4c}, where c is an integer, in the transfer function of the recursive code. The infinite number of monomials of this type is due to the existence of the cycle on a null input sequence different from the loop on state 0. Moreover, such a code does not provide any monomials of the form IO^c, unlike non-recursive codes. These conclusions concur with those drawn from the study of state machines in Section 5.2.

This notion of transfer function is therefore efficient for studying the performance of a convolutional code. A derived version is moreover essential for the classification of codes according to their performance. This is the distance spectrum $\omega(d)$ whose definition is as follows:

$$(\frac{\partial T(O,I)}{\partial I})_{I=1} = \sum_{d=d_f}^{\infty} \omega(d)O^d \tag{5.14}$$

For example, the first terms of the spectrum of the recursive systematic code, obtained from (5.13), are presented in Table 5.1. This spectrum is essential for estimating the performance of codes in terms of calculating their error probability, as illustrated in the vast literature on this subject [5.9].

The codes used in the above examples have a rate of 1/2. By increasing the number of redundancy bits n the rate becomes lower. In this case, the powers of O associated with the branches of the state machines will be higher than or equal to those of the figures above. This leads to higher transfer functions with powers of O, that is, to RTZ sequences with a greater Hamming weight. The codes with lower rates therefore have a higher error correction capability.

5.3.4 Performance

The performance of a code is defined by the decoding error probability after transmission on a noisy channel. The previous section allows us to intuitively compare non-recursive non-systematic, non-recursive systematic and recursive systematic codes with the same constraint length. However, to estimate the absolute performance of a code, we must be able to estimate the decoding error probability as a function of the noise, or at least to limit it. The literature, for example [5.9], thus defines many bounds that are not described here and we will limit ourselves to comparing the three categories of convolutional codes. To do this, a transmission on a Gaussian channel of blocks of 53 then 200 bytes

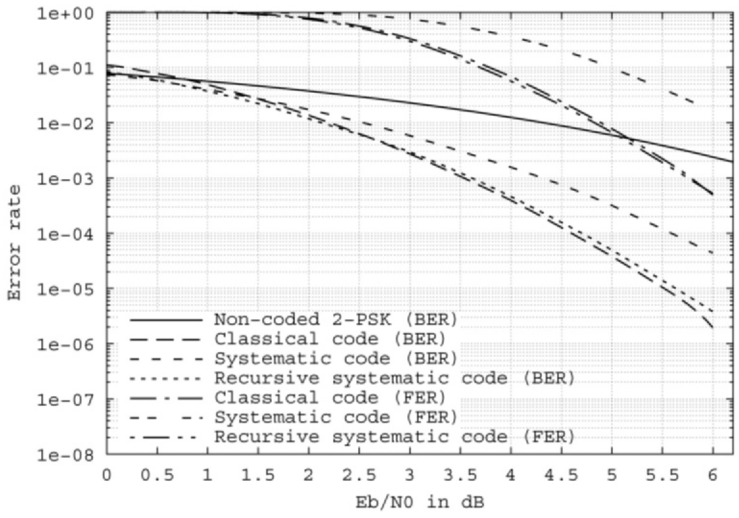


Figure 5.16 – Comparison of simulated performance (Binary Error Rate and Packet Error Rate) of three categories of convolutional codes after transmission of packets of 53 bytes on a Gaussian channel (decoding using the MAP algorithm).

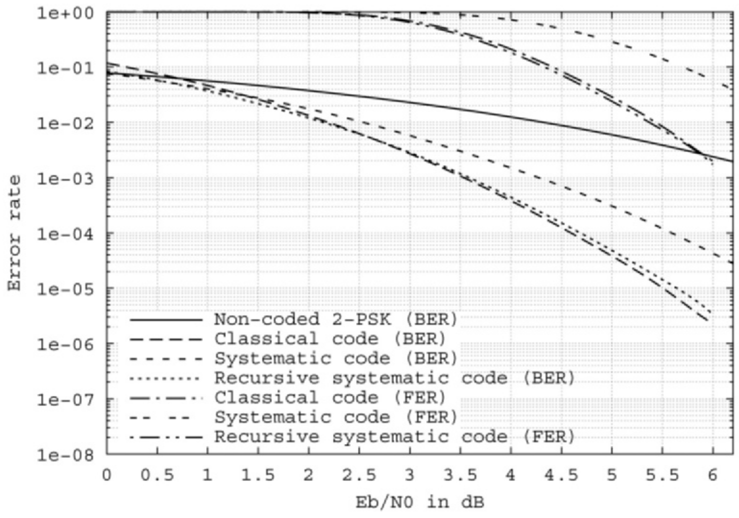


Figure 5.17 – Comparison of simulated performance of three categories of convolutional codes after transmission of blocks of 200 bytes on a Gaussian channel.

coded according to different schemes was simulated (Figures 5.16 and 5.17): classical (non-recursive non-systematic), non-recursive systematic and recursive systematic.

The blocks were constructed following the classical trellis termination technique for non-recursive codes whereas the recursive code is circular tail-biting (see Section 5.5). The decoding algorithm used is the MAP algorithm.

The BER curves are in perfect agreement with the conclusions drawn during the analysis of the free distance of codes and of their transfer function: the systematic code is not as good as the others at high signal to noise ratio and the classical code is then slightly better than the recursive code. At low signal to noise ratios, the hierarchy is different: the recursive code and the systematic code are equivalent and better than the classical code.

Comparing performance as a function of the size of the frame (53 and 200 bytes) shows that the performance hierarchy of the codes is not modified. Moreover, the bit error rates are almost identical. This was predictable as the sizes of the frames are large enough for the transfer functions of the codes not to be affected by edge effects. However, the packet error rate is affected by the length of the blocks since although the bit error probability is constant, the packet error probability increases with size.

The comparisons above only concern codes with 8 states. It is, however, easy to see that the performance of a convolutional code is linked with its capacity to provide information on the succession of data transmitted: the more the code can integrate successive data into its output symbols, the more it improves the quality of protection these data. In other words, the greater the number of states (therefore the size of the register of the encoder), the more efficient a convolutional code is (within its category). Let us compare three recursive systematic codes:

- 4 states $[1, (1 + D^2)/(1 + D + D^2)]$,
- 8 states $[1, (1 + D^2 + D^3)/(1 + D + D^3)]$
- and 16 states $[1, (1 + D + D^2 + D^4)/(1 + D^3 + D^4)]$.

Their performance in terms of BER and PER were simulated on a Gaussian channel and are presented in Figure 5.18.

The higher the number of states of the code, the lower the residual error rates are. For a BER of 10^{-4}, 0.6 dB are thus gained when passing from 4 states to 8 states and 0.5 dB when passing from 8 states to 16 states. This remark is coherent with the qualitative justification of the interest of a large number of states. It would therefore seem logical to choose a convolutional code with a large number of states to ensure the desired protection, especially since such codes offer the possibility of producing redundancy on far more than two components, and therefore of providing even higher protection. Thus, the *Big Viterbi Decoder* project at *NASA*'s *Jet Propulsion Laboratory* used for transmissions with space

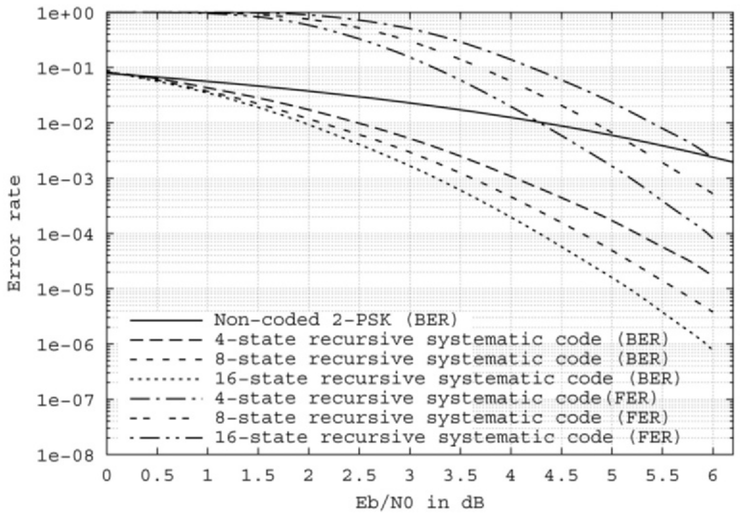


Figure 5.18 – Comparison of simulated performance of recursive systematic convolutional codes with 4, 8 and 16 states after transmitting packets of 53 bytes on a Gaussian channel (decoding according to the MAP algorithm)

probes was designed to process frames encoded with convolutional codes with 2 to 16384 states and rates much lower than 1/2 (16384 states and $R = 1/6$ for the *Cassini* probe to Saturn and the *Mars Pathfinder* probe). Why not use such codes for terrestrial radio-mobile transmissions for the general public? Because the complexity of the decoding would become unacceptable for current terrestrial transmissions using a reasonably-sized terminal operating in real time, and fitting into a pocket.

5.4 Decoding convolutional codes

There exist several algorithms for decoding convolutional codes. The most famous is probably the Viterbi algorithm which relies on the trellis representation of codes [5.14][5.8]. It enables us to find the most probable sequence of states in the trellises from the received symbol sequence.

The original Viterbi algorithm performs *hard output* decoding, that is, it provides a binary estimation of each of the symbols transmitted. It is therefore not directly adapted to iterative systems that require information about the *trustworthiness* of the decisions. Adaptations of the Viterbi algorithm such as those proposed in [5.2], [5.10] or [5.3] have led to versions with weighted output called *Soft-Output Viterbi Algorithm* (SOVA). SOVA algorithms are not described in this book since, for turbo decoding, we prefer another family of algorithms relying on the minimization of the error probability of each symbol transmitted. Thus, the Maximum *A Posteriori* (MAP) algorithm enables the calculation of the exact value of the *a posteriori* probability associated with each symbol transmitted using the received sequence [5.1]. The MAP algorithm and its variants are described in Chapter 7.

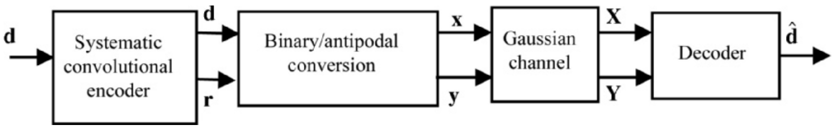


Figure 5.19 – Model of the transmission chain studied.

5.4.1 Model of the transmission chain and notations

Viterbi and MAP algorithms are used in the transmission chain shown in Figure 5.19. The convolutional code considered is systematic and, for each data sequence of information $\mathbf{d} = \mathbf{d}_1^N = \{\mathbf{d}_1, \cdots, \mathbf{d}_N\}$, calculates a redundant sequence $\mathbf{r} = \mathbf{r}_1^N = \{\mathbf{r}_1, \cdots, \mathbf{r}_N\}$. The code is m-binary with rate $R = m/(m+n)$: each vector of data $\mathbf{d}_i$ at the input of the encoder is thus made up of m bits, $\mathbf{d}_i = (d_i^{(1)}, d_i^{(2)}, \cdots, d_i^{(m)})$, and the corresponding redundancy vector at the output is written $\mathbf{r}_i = (r_i^{(1)}, r_i^{(2)}, \cdots, r_i^{(n)})$. The value of $\mathbf{d}_i$ can also be represented by the scalar integer variable $j = \sum_{l=1}^{m} 2^{l-1} d_i^{(l)}$, between 0 and $2^m - 1$, and we can then write $\mathbf{d}_i \equiv j$.

The systematic $\mathbf{d}$ and redundant $\mathbf{r}$ data sequences are transmitted after a binary/antipodal conversion making an antipodal value (-1 or +1) transmitted towards the channel correspond to each value binary (0 or 1) coming from the encoder. $\mathbf{X}$ and $\mathbf{Y}$ represent the noisy systematic and redundant symbol sequences received at the input of the decoder and $\hat{\mathbf{d}}$ the decoded sequence that can denote either a binary sequence in the case of the Viterbi algorithm, or a sequence of weighted decisions associated with the $\mathbf{d}_i$ at the output of the MAP algorithm.

5.4.2 The Viterbi algorithm

The Viterbi algorithm is the most widely used method for the maximum-likelihood (ML) decoding of convolutional codes with low constraint length (typically $\nu \leq 8$). Beyond this limit, its complexity of implementation means that we have to resort to a sequential decoding algorithm, like Fano's [5.6].

ML decoding is based on a search for the codeword $\mathbf{c}$ that is the shortest distance away from the received word. In the case of a channel with binary decisions (binary symmetric channel), ML decoding relies on the Hamming distance, whereas in the case of a Gaussian channel, it relies on the Euclidean distance (see Chapter 2). An exhaustive search for codewords associated with the different paths in the trellis leads to $2^{\nu+k}$ paths being taken into account. In practice, searching for the path with the minimum distance on a working window with a width l lower than k, limits the search to $2^{\nu+l}$paths.

The Viterbi algorithm enables a notable reduction in the complexity of the computation. It is based on the idea that, among the set of paths of the trellis that converge in a node at a given instant, only the most probable path can be retained for the following search steps. Let us denote $\mathbf{s}_i = (s_i^{(1)}, s_i^{(2)}, \ldots, s_i^{(\nu)})$ the state of the encoder at instant i and $T(i, \mathbf{s}_{i-1}, \mathbf{s}_i)$ the branch of the trellis corresponding to the emission of data $\mathbf{d}_i$ and associated with the transition between nodes $\mathbf{s}_{i-1}$ and $\mathbf{s}_i$. Applying the Viterbi algorithm involves performing the set of operations described below.

- *At each instant i, for i ranging from 1 to k:*
  - Calculate for each branch of a *branch metric*, $d\left(T(i, \mathbf{s}_{i-1}, \mathbf{s}_i)\right)$. For a binary output channel, this metric is defined as the Hamming distance between the symbol carried by the branch of the trellis and the received symbol, $d\left(T(i, \mathbf{s}_{i-1}, \mathbf{s}_i)\right) = d_H\left(T(i, \mathbf{s}_{i-1}, \mathbf{s}_i)\right)$.
    For a Gaussian channel, the metric is equal to the square of the Euclidean distance between the branch considered and the observation at the input of the decoder (see also Section 1.3):

$$\begin{aligned} d\left(T(i, \mathbf{s}_{i-1}, \mathbf{s}_i)\right) &= \left\|\mathbf{X}_i - \mathbf{x}_i\right\|^2 + \left\|\mathbf{Y}_i - \mathbf{y}_i\right\|^2 \\ &= \sum_{j=1}^{m} \left(x_i^{(j)} - X_i^{(j)}\right)^2 + \sum_{j=1}^{n} \left(y_i^{(j)} - Y_i^{(j)}\right)^2 \end{aligned}$$

  - Calculate the *accumulated metric* associated with each branch $T(i, \mathbf{s}_{i-1}, \mathbf{s}_i)$ defined by:

$$\lambda(T(i, \mathbf{s}_{i-1}, \mathbf{s}_i)) = \mu(i - 1, \mathbf{s}_{i-1}) + d(T(i, \mathbf{s}_{i-1}, \mathbf{s}_i))$$

    where $\mu(i-1, \mathbf{s}_{i-1})$ is the accumulated metric associated with node $\mathbf{s}_{i-1}$.

  - For each node $\mathbf{s}_i$, select the branch of the trellis corresponding to the minimum accumulated metric and memorize this branch in memory (in practice, it is the value of d_i associated with the branch that is stored). The path in the trellis made up of the branches successively memorized at the instants between 0 and i is the *survivor path* arriving in $\mathbf{s}_i$. If the two paths that converge in $\mathbf{s}_i$ have identical accumulated metrics, the survivor is then chosen arbitrarily between these two paths.

- Calculate the accumulated metric associated with each node $\mathbf{s}_i$, $\mu(i, \mathbf{s}_i)$. It is equal to the accumulated metric associated with the survivor path arriving in $\mathbf{s}_i$:

$$\mu(i, \mathbf{s}_i) = \min_{\mathbf{s}_{i-1}} \left(\lambda\left(T(i, \mathbf{s}_{i-1}, \mathbf{s}_i)\right)\right).$$

- *Initialization (instant i =0):*

The initialization values of the metrics μ when commencing the algorithm depend on the initial state $\mathbf{s}$ of the encoder: $\mu(0, \mathbf{s}) = +\infty$ if $\mathbf{s} \neq \mathbf{s}_0$ and $\mu(0, \mathbf{s}_0) = 0$. If this state is not known, all the metrics are initialized to the same value, typically 0. In this case, the decoding of the beginning of the frame is less efficient since the accumulated metrics associated with each branch at instant 1 depend only on the branch itself. The past cannot be taken into account since it is not known: $\lambda(T(1, \mathbf{s}_0, \mathbf{s}_1)) = d(T(1, \mathbf{s}_0, \mathbf{s}_1))$.

- *Calculating the decisions (instant i=k):*

At instant k, if the final state of the encoder is known, the maximum likelihood path is the survivor path coming from the node corresponding to the final state of the encoder. The decoded sequence is given by the series of values of d_i, i ranging from 1 to k, stored in the memory associated with the maximum likelihood path. This operation is called *trellis traceback.*

If the final state is not known, the maximum likelihood path is the survivor path coming from the node with minimum accumulated metric. In this case, the problem is similar to the one mentioned for initialization: the decoding of the end of the frame is less efficient.

When the sequence transmitted is long, or even infinite in length, it is not possible to wait for the whole transmitted binary sequence to be received to begin the decoding operation. To limit the decoding latency and the size of memory necessary to memorize the survivor paths, the trellis must be truncated. Observing the algorithm unfold, we can note that by tracing back sufficiently in time from instant i, the survivor paths coming from the different nodes of the trellis nearly always converge towards a same path. In practice, memorizing the survivors can therefore be limited to a time interval of duration l. It is then sufficient to do a trellis traceback at each instant i over a length l in order to take the decision on the data d_{i-l}. To decrease the complexity, the survivors are sometimes memorized on an interval higher than l (for example l+3). The number of trellis traceback operations is then decreased (divided by 3 in our example) but each of the tracebacks provides several decisions (d_{i-l}, d_{i-l-1} and d_{i-l-2}, in our example).

The higher the code memory and coding rate are, the greater the value of l must be. We observe that, for a systematic code, values of l corresponding to the production by the encoder of a number of redundancy symbols equal to 5 times the constraint length of the code are sufficient. As an example, for coding rate $R = 1/2$, we typically take l equal to $5(\nu + 1)$.

From the point of view of complexity, the Viterbi algorithm requires the calculation of $2^{\nu+1}$ accumulated metrics at each instant i and its complexity varies linearly with the length of sequence k or of decoding window l.

Example of applying the Viterbi algorithm

Let us illustrate the different steps of the Viterbi algorithm described above by applying it to decode the binary recursive systematic convolutional code (7,5) with 4 states and encoding rate $R = 1/2$ $(m = n = 1)$. The structure of the encoder and the trellis are shown in Figure 5.20.

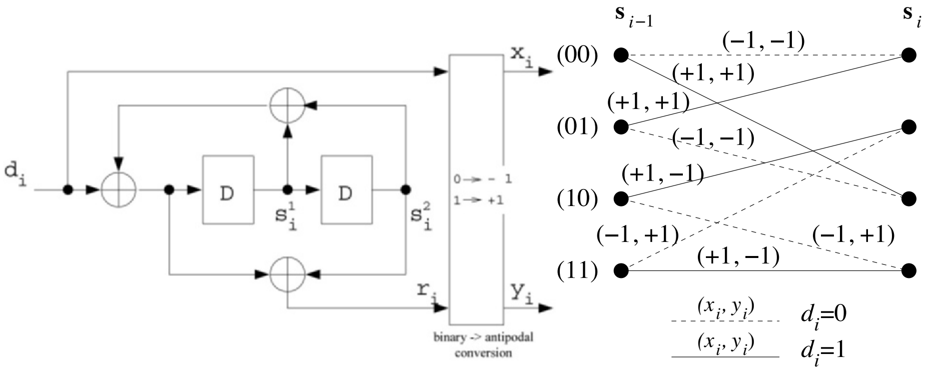


Figure 5.20 – Structure of the recursive systematic convolutional code (7,5) and associated trellis.

Calculate the branch metrics at instant i:

$$
\begin{aligned}
d(T(i,\mathbf{0},\mathbf{0})) = d(T(i,\mathbf{1},\mathbf{2})) &= (X_i + 1)^2 + (Y_i + 1)^2 \\
d(T(i,\mathbf{0},\mathbf{2})) = d(T(i,\mathbf{1},\mathbf{0})) &= (X_i - 1)^2 + (Y_i - 1)^2 \\
d(T(i,\mathbf{2},\mathbf{1})) = d(T(i,\mathbf{3},\mathbf{3})) &= (X_i - 1)^2 + (Y_i + 1)^2 \\
d(T(i,\mathbf{2},\mathbf{3})) = d(T(i,\mathbf{3},\mathbf{1})) &= (X_i + 1)^2 + (Y_i - 1)^2
\end{aligned}
$$

Calculate the accumulated metrics of the branches at instant i:

$$
\begin{aligned}
\lambda(T(i,\mathbf{0},\mathbf{0})) &= \mu(i-1,\mathbf{0}) + d(T(i,\mathbf{0},\mathbf{0})) = \mu(i-1,\mathbf{0}) + (X_i+1)^2 + (X_i+1)^2 \\
\lambda(T(i,\mathbf{1},\mathbf{2})) &= \mu(i-1,\mathbf{1}) + d(T(i,\mathbf{1},\mathbf{2})) = \mu(i-1,\mathbf{1}) + (X_i+1)^2 + (X_i+1)^2 \\
\lambda(T(i,\mathbf{0},\mathbf{2})) &= \mu(i-1,\mathbf{0}) + d(T(i,\mathbf{0},\mathbf{2})) = \mu(i-1,\mathbf{0}) + (X_i-1)^2 + (X_i-1)^2 \\
\lambda(T(i,\mathbf{1},\mathbf{0})) &= \mu(i-1,\mathbf{1}) + d(T(i,\mathbf{1},\mathbf{0})) = \mu(i-1,\mathbf{1}) + (X_i-1)^2 + (X_i-1)^2 \\
\lambda(T(i,\mathbf{2},\mathbf{1})) &= \mu(i-1,\mathbf{2}) + d(T(i,\mathbf{2},\mathbf{1})) = \mu(i-1,\mathbf{2}) + (X_i-1)^2 + (X_i+1)^2 \\
\lambda(T(i,\mathbf{3},\mathbf{3})) &= \mu(i-1,\mathbf{3}) + d(T(i,\mathbf{3},\mathbf{3})) = \mu(i-1,\mathbf{3}) + (X_i-1)^2 + (X_i+1)^2 \\
\lambda(T(i,\mathbf{2},\mathbf{3})) &= \mu(i-1,\mathbf{2}) + d(T(i,\mathbf{2},\mathbf{3})) = \mu(i-1,\mathbf{2}) + (X_i+1)^2 + (X_i-1)^2 \\
\lambda(T(i,\mathbf{3},\mathbf{1})) &= \mu(i-1,\mathbf{3}) + d(T(i,\mathbf{3},\mathbf{1})) = \mu(i-1,\mathbf{3}) + (X_i+1)^2 + (X_i-1)^2
\end{aligned}
$$

Calculate the accumulated metrics of the nodes at instant i:

$$\begin{aligned}\mu(i,\mathbf{0}) &= \min\left(\lambda(T(i,\mathbf{0},\mathbf{0}),\lambda(T(i,\mathbf{1},\mathbf{0})\right)\\ \mu(i,\mathbf{1}) &= \min\left(\lambda(T(i,\mathbf{3},\mathbf{1}),\lambda(T(i,\mathbf{2},\mathbf{1})\right)\\ \mu(i,\mathbf{2}) &= \min\left(\lambda(T(i,\mathbf{0},\mathbf{2}),\lambda(T(i,\mathbf{1},\mathbf{2})\right)\\ \mu(i,\mathbf{3}) &= \min\left(\lambda(T(i,\mathbf{3},\mathbf{3}),\lambda(T(i,\mathbf{2},\mathbf{3})\right)\end{aligned}$$

For each of the four nodes of the trellis, the value of d_i corresponding to the transition of minimum accumulated metric λ is stored in memory.

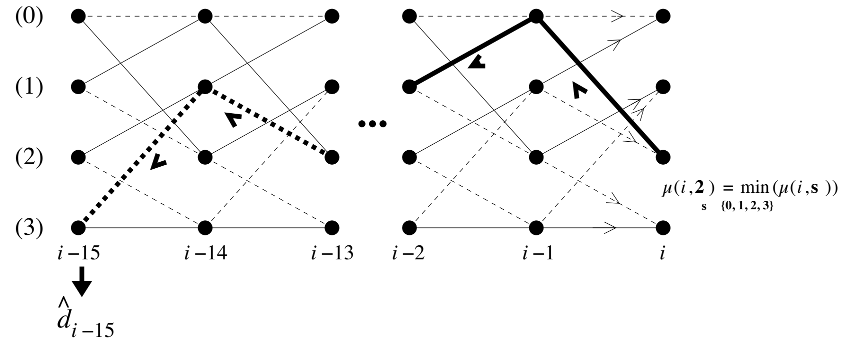

(0)
(1)
(2)
(3)
$i-15$
$i-14$
$i-13$
$i-2$
$i-1$
i
$\hat{d}_{i-15}$
$\mu(i,\mathbf{2}) = \min_{\mathbf{s}\ \{0,1,2,3\}}(\mu(i,\mathbf{s}))$

Figure 5.21 – Survivor path traceback operation (in bold) in the trellis from instant i and determining the binary decision at instant $i-15$.

After selecting the node with minimum accumulated metric, denoted $\mathbf{s}$ (in the example of Figure 5.21, $\mathbf{s} = \mathbf{3}$), we trace back in the trellis along the survivor path to a depth $l = 15$. At instant $i-15$, the binary decision $\hat{d}_{i-15}$ is equal to the value of d_{i-15} stored in the memory associated with the survivor path.

The aim of applying the Viterbi algorithm with weighted inputs is to search for codeword $\mathbf{c}$ that is the shortest Euclidean distance between two codewords. Equivalently (see Chapter 1), this also means looking for the codeword that maximizes the scalar product $\langle\mathbf{x},\mathbf{X}\rangle + \langle\mathbf{y},\mathbf{Y}\rangle = \sum\limits_{i=1}^{k}\left(\sum\limits_{l=1}^{m} x_i^{(l)}X_i^{(l)} + \sum\limits_{l=1}^{n} y_i^{(l)}Y_i^{(l)}\right)$. In this case, applying the Viterbi algorithm uses branch metrics of the form $d\left(T(i,\mathbf{s}_{i-1},\mathbf{s}_i)\right) = \sum\limits_{l=1}^{m} x_i^{(l)}X_i^{(l)} + \sum\limits_{l=1}^{n} y_i^{(l)}Y_i^{(l)}$ and the survivor path then corresponds to the path with maximum accumulated metric.

Figure 5.22 provides the performance of the two variants, with hard and weighted inputs, of a decoder using the Viterbi algorithm for the code (7,5) RSC for a transmission on a channel with additive white Gaussian noise. In practice, we observe a gain of around 2 dB when we substitute weighted input decoding for hard input decoding.

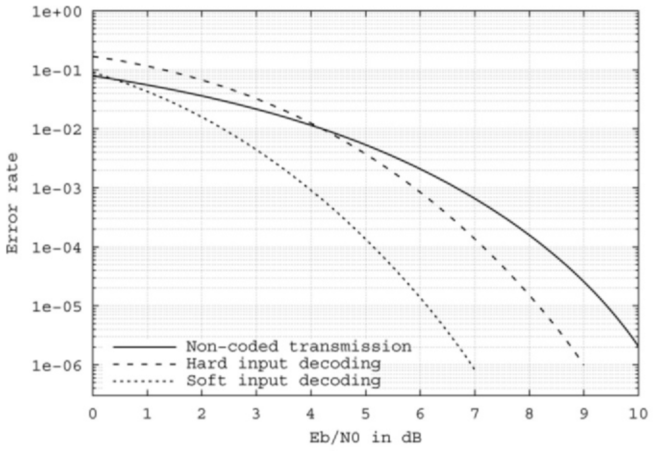


Figure 5.22 – Example of correction performance of the Viterbi algorithm with hard inputs and with weighted inputs on a Gaussian channel. Recursive systematic convolutional code (RSC) with generator polynomials 7 (recursivity) and 5 (redundancy). Coding rate $R = 1/2$.

5.4.3 The Maximum *A Posteriori* algorithm or MAP algorithm

The Viterbi algorithm determines the codeword closest to the received word. However, it does not necessarily minimize the error probability of the bits or symbols transmitted. The MAP algorithm enables us to calculate the *a posteriori* probability of each bit or each symbol transmitted and, at each instant, the corresponding decoder selects the most probable bit or symbol. This algorithm was published in 1974 by Bahl, Cocke, Jelinek and Raviv [5.1]. The impact of this decoding method remained little known until the discovery of turbocodes since it does not provide any notable improvement in performance compared to the Viterbi algorithm for decoding convolutional codes and it turned out to be more complex to implement. However, the situation changed in 1993 as decoding turbocodes uses elementary decoders with weighted or soft outputs and the MAP algorithm, unlike the Viterbi algorithm, enables us to associate a weighting naturally with each decision. The MAP algorithm is presented in Chapter 7.

5.5 Convolutional block codes

Convolutional codes are naturally adapted to transmission applications where the message transmitted is of infinite length. However, most telecommunications systems use independent frame transmissions. Paragraph 5.4.2 showed the importance of knowing the initial and final states of the encoder during the decoding of a frame. In order to know these states, the technique used is usually called trellis termination. This generally involves forcing the initial and final states to values known by the decoder (in general zero).

5.5.1 Trellis termination

Classical trellis termination

As the encoder is constructed around a register, it is easy to initialize it by using reset inputs before beginning the encoding of a frame. This operation has no consequence on the coding rate. But termination at the end of a frame is not so simple.

When the k bits of the frame have been coded, the register of the encoder is in any of the 2^ν possible states. The aim of termination is to lead the encoder towards state zero by following one of the paths in the trellis so that the decoding algorithm can use this knowledge of the final state. In the case of non-recursive codes, the final state is forced to zero by injecting ν zero bits at the end of the frame. It is as if the coded frame were of length $k+\nu$ with $d_{k+1} = d_{k+2} = \ldots = d_{k+\nu} = 0$. The encoding rate is slightly decreased by the transmission of the *termination bits.* However, taking into account the size of the frames generally transmitted, this degradation in rate is very often negligible. In the case of recursive codes, it is also possible to inject a zero at the input of the register. Figure 5.23 shows a simple way to solve this question.

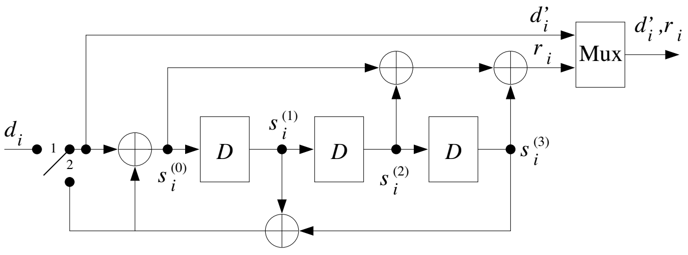


Figure 5.23 – Example of an encoder of recursive systematic convolutional codes allowing termination at state 0 in ν instants.

After initializing the register to zero, switch I is kept in position 1 and data d_1 to d_k are coded. At the end of this encoding operation, instants k to $k+\nu$, switch I is placed in position 2 and d_i takes the value coming from the feedback of the register, that is, a value that forces one register input to zero. Indeed, $S_i^{(0)}$ is the result of a modulo-2 sum of two identical members. As for the encoder, it continues to produce the associated redundancies r_i.

This classical termination has one main drawback: the protection of the data is not independent of their position in the frame. In particular, this can lead to edge effects in the construction of a turbocode (see Chapter 7).

Tail-biting

A technique was introduced in the 70s and 80s [5.12] to terminate the trellis of convolutional codes without edge effects: tail-biting. This involves making the decoding trellis circular, that is, ensuring that the departure and the final states of the encoder are identical. This state is then called the circulation state. This technique is trivial for non-recursive codes as the circulation state is merely the last ν bits of the sequence to encode. As for RSC codes, tail-biting requires operations that are described in the following. The trellis of such a code, called circular recursive systematic codes (CRSC), is shown in Figure 5.24.

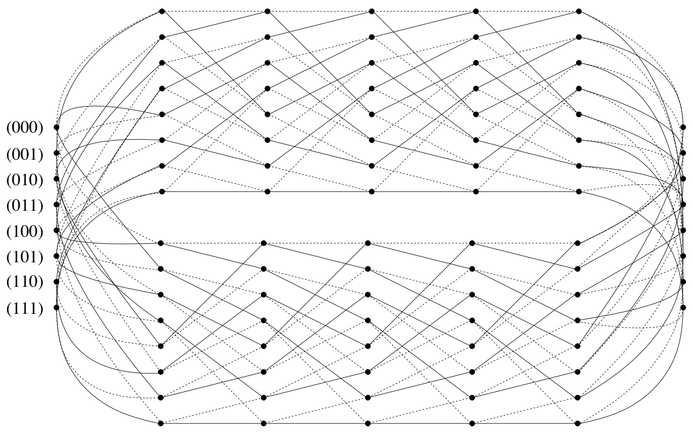


Figure 5.24 – Trellis of a CRSC code with 8 states.

It is possible to establish a relation between state $\mathbf{s}_{i+1}$ of the encoder at an instant $i+1$, its state $\mathbf{s}_i$ and the input data $\mathbf{d}_i$ at the previous time:

$$\mathbf{s}_{i+1} = \mathbf{A}\mathbf{s}_i + \mathbf{B}\mathbf{d}_i \tag{5.15}$$

where $\mathbf{A}$ is the state matrix and $\mathbf{B}$ the input matrix. In the case of the recursive systematic code of generator polynomials $[1,\ (1{+}D^2 + D^3)/(\ 1{+}D + D^3)]$ mentioned above, these matrices are

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \text{ and } \mathbf{B} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}.$$

If the encoder is initialized to state 0 ($\mathbf{s}_0 = 0$), the final state $\mathbf{s}_k^0$ obtained at the end of a frame of length k is:

$$\mathbf{s}_k^0 = \sum_{j=1}^{k} \mathbf{A}^{j-1}\mathbf{B}\mathbf{d}_{k-j} \tag{5.16}$$

When it is initialized in any state $\mathbf{s}_c$, the final state $\mathbf{s}'_k$ is expressed as follows:

$$\mathbf{s}'_k = \mathbf{A}^k\mathbf{s}_c + \sum_{j=1}^{k} \mathbf{A}^{j-1}\mathbf{B}\mathbf{d}_{k-j} \tag{5.17}$$

For this state $\mathbf{s}'_k$ to be equal to the departure state $\mathbf{s}_c$ and for the latter therefore to become the circulation state, it is necessary and sufficient that:

$$\left(\mathbf{I} - \mathbf{A}^k\right)\mathbf{s}_c = \sum_{j=1}^{k} \mathbf{A}^{j-1}\mathbf{B}\mathbf{d}_{k-j} \tag{5.18}$$

where $\mathbf{I}$ is the identity matrix of dimension $\nu \times \nu$.
Thus, by introducing state $\mathbf{s}_k^0$ of the encoder after initialization to 0

$$\mathbf{s}_c = \left(\mathbf{I} - \mathbf{A}^k\right)^{-1}\mathbf{s}_k^0 \tag{5.19}$$

This is only possible on condition that the matrix $(\mathbf{I} - \mathbf{A}^k)$ is *invertible* : this is the condition for the existence of the circulation state.

In conclusion, if the circulation state exists, it can be obtained in two steps. The first involves coding the frame of k bits at the input after initializing the encoder to state zero and keeping the termination state. The second is to simply deduce the circulation state from the previous termination state and from a table (obtained by inverting $\mathbf{I} - \mathbf{A}^k$).

Take for example the recursive systematic code used above. Since it is a binary code, addition and subtraction are equivalent: the circulation state exists if $\mathbf{I} + \mathbf{A}^k$ is invertible. Matrix $\mathbf{A}$ is such that $\mathbf{A}^7$ is equal to $\mathbf{I}$. Thus, if k is a multiple of $7(= 2^\nu - 1)$, $\mathbf{I} + \mathbf{A}^k$ is null and therefore non-invertible: this case should be avoided. Another consequence is that $\mathbf{A}^k = \mathbf{A}^{k \bmod 7}$: it suffices to calculate once and for the 6 state transformation tables associated with the 6 possible values of $(\mathbf{I} - \mathbf{A}^k)^{-1}$, to store them and to read the right table, after calculating $k \mod 7$. The table of the CRSC code with 8 states of generator polynomials $[1, (1 + D^2 + D^3)/(1 + D + D^3)]$ is given in Table 5.2.

A simple method for encoding according to a circular trellis can be summarized in five steps, after checking the existence of the circulation state:

1. Initialize the encoder to state 0

2. Code the frame to obtain the final state $\mathbf{s}_k^0$;

$k \mod 7$ / $\mathbf{s}_k^0$	1	2	3	4	5	6
0	0	0	0	0	0	0
1	6	3	5	4	2	7
2	4	7	3	1	5	6
3	2	4	6	5	7	1
4	7	5	2	6	1	3
5	1	6	7	2	3	4
6	3	2	1	7	4	5
7	5	1	4	3	6	2

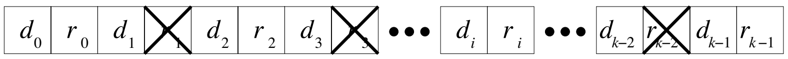
Table 5.2 – Table of the CRSC code with generator polynomials $[1, (1+D^2+D^3)/(1+D+D^3)]$ providing the circulation state as a function of $k \mod 7$ (k being the length of the frame at the input) and of the terminal state $\mathbf{s}_k^0$ obtained after encoding initialized to state 0.

3. Calculate the circulation state $\mathbf{s}_c$ from the tables already calculated and stored;
4. Initialize the encoder to state $\mathbf{s}_c$;
5. Code the frame and transmit the redundancies calculated.

5.5.2 Puncturing

Some applications can only allocate a small space for the redundant part of the codewords. But, by construction, the natural rate of a systematic convolutional code is $m/(m+n)$, where m is the number of input bits $\mathbf{d}_i$ of the encoder and n is the number of output bits. It is therefore maximum when $n = 1$ and becomes $R = m/(m+1)$. High rates can therefore only be obtained with high values of m. Unfortunately, the number of transitions leaving any one node of the trellis is 2^m. In other words, the complexity of the trellis, and therefore of the decoding, increases exponentially with the number of input bits of the encoder. Therefore, this solution is generally not satisfactory. It is often avoided in favour of a technique with a slightly lower error correction capability, but easier to implement: *puncturing*.

The puncturing technique is commonly used to obtain high rates. It involves using an encoder with a low value of m (1 or 2 for example), to keep a reasonable decoding complexity, but transmitting only part of the bits coded. An example is proposed in Figure 5.25. In this example, a 1/2 rate encoder produces outputs d_i and r_i at each instant i. Only 3 bits out of 4 are transmitted, which leads to a global rate of 2/3. The pattern in which the bits are punctured is called the puncturing mask.



d_0 r_0 d_1 r_1 d_2 r_2 d_3 r_3 ••• d_i r_i ••• d_{k-2} r_{k-2} d_{k-1} r_{k-1}

Figure 5.25 – Puncturing a systematic code to obtain a rate 2/3.

In the case of systematic codes, it is generally the redundancy that is punctured. Figure 5.26 shows the trellis of code $[1, (1 + D^2 + D^3)/(1 + D + D^3)]$ resulting from a puncturing operation according to the mask of Figure 5.25. The "X"s mark the bits that are not transmitted and that therefore cannot be used for the decoding.

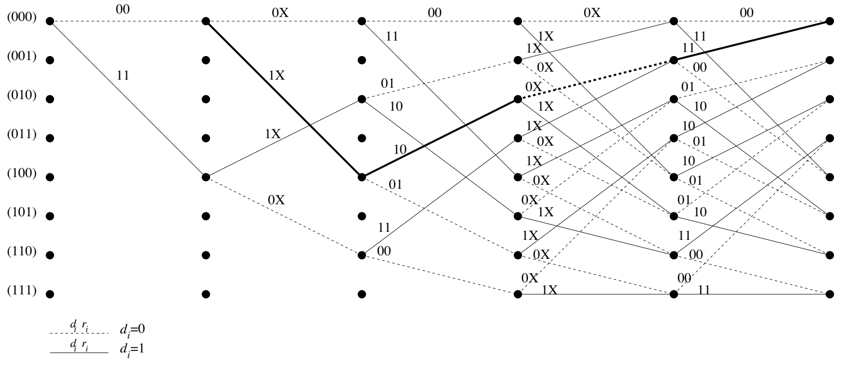


Figure 5.26 – Trellis diagram of the punctured recursive code for a rate 2/3.

The most widely used decoding technique involves taking the decoder of the original code and inserting *neutral* values in the place of the punctured elements. The neutral values are values representing information that is *a priori* not known. In the usual case of a transmission using antipodal signalling (+1 for the logical '1', -1 for the logical '0'), the null value (analogue 0) is taken as the neutral value.

The introduction of puncturing increases the coding rate but, of course, decreases its correction capability. Thus, in the example of Figure 5.26, the free distance of the code is reduced from 6 to 4 (an associated RTZ sequence is shown in the figure). Likewise Figure 5.27, in which we present the error rate curves of code $[1, (1 + D^2 + D^3)/(1 + D + D^3)]$ for rates 1/2, 2/3, 3/4 and 6/7, shows a decrease in error correction capability with the increase in coding rate.

The choice of puncturing mask obviously influences the performance of the code. It is thus possible to favour one part of the frame, transporting sensitive data, by slightly puncturing it to the detriment of another part that is more highly punctured. A regular mask is, however, often chosen as it is simple to implement.

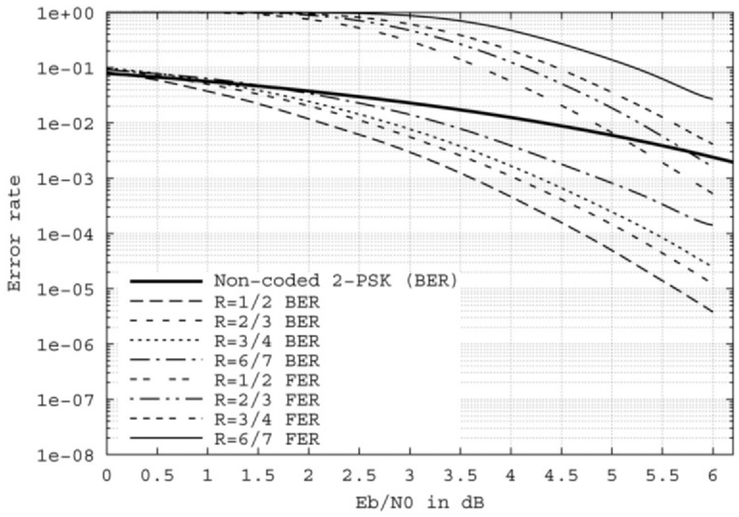


Figure 5.27 – Simulated performance of a CRSC code with 8 punctured states, as a function of its rate.

Puncturing is therefore a flexible technique, and easy to implement. As the encoder and the decoder remain identical whatever the puncturing applied, it is possible to modify the encoding rate at any moment. Some applications use this flexibility to adapt the rate, as they go along, to the channel and/or to the importance of the data transmitted.

Bibliography

[5.1] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv. Optimal decoding of linear codes for minimizing symbol error rate. *IEEE Transactions on Information Theory*, IT-20:284–287, March 1974.

[5.2] G. Battail. Weighting of the symbols decoded by the viterbi algorithm. *Annals of Telecommunications*, 42(1-2):31–38, Jan.-Feb. 1987.

[5.3] C. Berrou, P. Adde, E. Angui, and S. Faudeuil. A low complexity soft-output viterbi decoder architecture. In *Proceedings of IEEE International Conference on Communications (ICC'93)*, pages 737–740, GENEVA, May 1993.

[5.4] C. Berrou, A. Glavieux, and P. Thitimajshima. Near shannon limit error-correcting coding and decoding: turbo-codes. In *Proceedings of IEEE In-*

ternational Conference on Communications (ICC'93), pages 1064–1070, GENEVA, May 1993.

[5.5] P. Elias. Coding for noisy channels. *IRE Convention Records*, 3(4):37–46, 1955.

[5.6] R. M. Fano. A heuristic discussion of probabilistic decoding. *IEEE Transactions on Information Theory*, IT-9:64–74, Apr. 1963.

[5.7] G. D. Forney. Convolutional codes i: Algebraic structure. *IEEE Transactions on Information Theory*, IT-16:720–738, Nov. 1970.

[5.8] G. D. Forney. The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, March 1973.

[5.9] A. Glavieux. *Codage de canal, des bases théoriques aux turbocodes.* Hermès-Science, 2005.

[5.10] J. Hagenauer and P. Hoeher. A viterbi algorithm with soft-decision outputs and its applications. In *Proceedings of IEEE Global Communications Conference (Globecom'89)*, pages 1680–1686, Dallas, Texas, USA, Nov. 1989.

[5.11] R. Johannesson and K. Sh. Zigangirov. *Fundamentals of Convolutional Coding.* Piscataway, IEEE Press, 1999.

[5.12] H. H. Ma and J. K. Wolf. On tail-biting convolutional codes. *IEEE Transactions on Communications*, COM-34:104–111, Feb. 1986.

[5.13] P. Thitimajshima. *Les codes convolutifs récursifs systématiques et leur application à la concaténation parallèle.* Phd thesis, Université de Bretagne Occidentale, Dec. 1993.

[5.14] A. J. Viterbi. Coding for noisy channels. *IEEE Transactions on Information Theory*, IT-13:260–269, Apr. 1967.

[5.15] J. M. Wozencraft. Sequential decoding for reliable communication. *IRE National Convention Records*, 5 part 2(2):11–25, 1957.