Charan Langton, Editor

SIGNAL PROCESSING & SIMULATION NEWSLETTER

Introduction to Coding and decoding with Convolutional Codes (*Tutorial 12*)

Convolutional codes are commonly specified by three parameters; (n,k,m).

n = number of output bits
k = number of input bits
m = number of memory registers

The quantity k/n called the code rate, is a measure of the efficiency of the code. Commonly k and n parameters range from 1 to 8, m from 2 to 10 and the code rate from 1/8 to 7/8 except for deep space applications where code rates as low as 1/100 or even longer have been employed.

Often the manufacturers of convolutional code chips specify the code by parameters (n,k,L), The quantity L is called the constraint length of the code and is defined by

Constraint Length, L = $k\ (m\text{-}1)$

The constraint length L represents the number of bits in the encoder memory that affect the generation of the n output bits. The constraint length L is also referred to by the capital letter K, which can be confusing with the lower case k, which represents the number of input bits. In some books K is defined as equal to product the of k and m. Often in commercial spec, the codes are specified by (r, K), where r = the code rate k/n and K is the constraint length. The constraint length K however is equal to L – 1, as defined in this paper. I will be referring to convolutional codes as (n,k,m) and not as (r,K).

Code parameters and the structure of the convolutional code

The convolutional code structure is easy to draw from its parameters. First draw m boxes representing the m memory registers. Then draw n modulo-2 adders to represent the n output bits. Now connect the memory registers to the adders using the generator polynomial as shown in the Fig. 1.

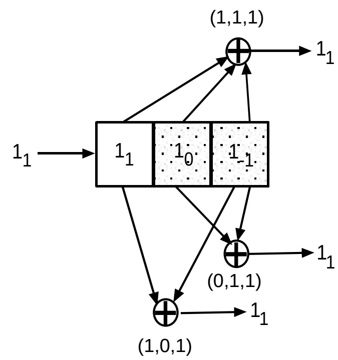


Figure 1 - This (3,1,3) convolutional code has 3 memory registers, 1 input bit and 3 output bits.

This is a rate 1/3 code. Each input bit is coded into 3 output bits. The constraint length of the code is 2. The 3 output bits are produced by the 3 modulo-2 adders by adding up certain bits in the memory registers. The selection of which bits are to be added to produce the output bit is called the generator polynomial (g) for that output bit. For example, the first output bit has a generator polynomial of (1,1,1). The output bit 2 has a generator polynomial of (0,1,1) and the third output bit has a polynomial of (1,0,1). The output bits just the sum of these bits.

$v_1 = \text{mod2}\ (u_1 + u_0 + u_{-1})$
$v_2 = \text{mod2}\ (\ u_0 + u_{-1})$
$v_3 = \text{mod2}\ (u_1 + u_{-1})$

The polynomials give the code its unique error protection quality. One (3,1,4) code can have completely different properties from another one, depending on the polynomials chosen.

How polynomials are selected

There are many choices for polynomials for any *m* order code. They do not all result in output sequences that have good error protection properties. Petersen and Weldon's book contains a complete list of these polynomials. Good polynomials are found from this list usually by computer simulation. A list of good polynomials for rate ½ codes is given below.

Table 1-Generator Polynomials found by Busgang for good rate ½ codes

Constraint Length	$\mathbf{G_1}$	$\mathbf{G_2}$
3	110	111
4	1101	1110
5	11010	11101
6	110101	111011
7	110101	110101
8	110111	1110011
9	110111	111001101
10	110111001	1110011001

States of a code

We have states of mind and so do encoders. We are depressed one day, and perhaps happy the next from the many different states we can be in. Our output depends on our states of mind and tongue-in-cheek we can say that encoders too act this way. What they output depends on what is their state of mind. Our states are complex but encoder states are just a sequence of bits. Sophisticated encoders have long constraint lengths and simple ones have short indicating the number of states they can be in.

The (2,1,4) code in Fig. 2 has a constraint length of 3. The shaded registers below hold these bits. The unshaded register holds the incoming bit. This means that 3 bits or 8 different combinations of these bits can be present in these memory registers. These 8 different combinations determine what output we will get for v_1 and v_2, the coded sequence.

The number of combinations of bits in the shaded registers are called the states of the code and are defined by

Number of states = 2^L

where L = the constraint length of the code and is equal to k (m - 1).

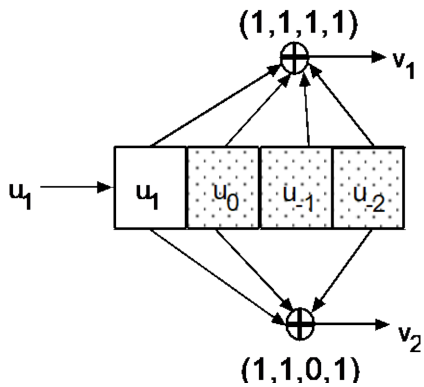


Figure 2 – The states of a code indicate what is in the memory registers

Think of states as sort of an initial condition. The output bit depends on this initial condition, which changes at each time tick.

Let's examine the states of the code (2,1,4) shown above. This code outputs 2 bits for every 1 input bit. It is a rate ½ code. Its constraint length is 3. The total number of states is equal to 8. The eight states of this (2,1,4) code are: 000, 001, 010, 011, 100, 101, 110, 111.

Punctured Codes

For the special case of k = 1, the codes of rates ½, 1/3, ¼, 1/5, 1/7 are sometimes *called mother codes*. We can combine these single bit input codes to produce punctured codes which give us code rates other than 1/n.

By using two rate ½ codes together as shown in Fig. 3 , and then just not transmitting one of the output bits we can convert this rate ½ implementation into a 2/3 rate code. 2 bits come and 3 go out. This concept is called puncturing. On the receive side, dummy bits that do not affect the decoding metric are inserted in the appropriate places before decoding.

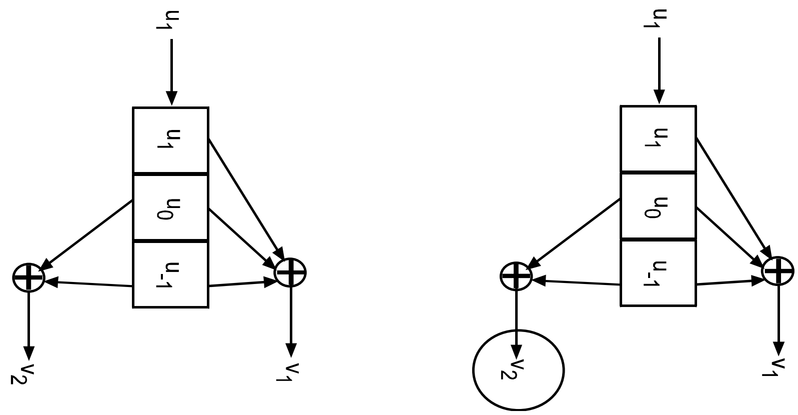


Figure 3 - Two (2,1,3) convolutional codes produce 4 output bits. Bit number 3 is “punctured” so the combination is effectively a (3,2,3) code.

This technique allows us to produce codes of many different rates using just one simple hardware type. Although we can also directly construct a code of rate 2/3 as we shall see later, the advantage of a punctured code is that the rates can be changed dynamically (through software) depending on the channel condition such as rain, etc. A fixed implementation, although easier, does not allow this flexibility.

Structure of a code for *k* > 1

Alternately we can create codes where *k* is more than 1 bit such as the (4,3,3) code. This code takes in 3 bits and outputs 4 bits. The number of registers is 3. The constraint length is 3 x 2 = 6. The code has 64 states. And this code requires polynomials of 9th order. The shaded boxes represent the constraint length.

The procedure for drawing the structure of a (*n*,*k*,*m*) code where *k* is greater than 1 is as follows. First draw *k* sets of *m* boxes. Then draw *n* adders. Now connect *n* adders to the memory registers using the coefficients of the n_{th} (*km*) degree polynomial. What you will get is a structure like the one in Fig. 4 for code (4,3,3).

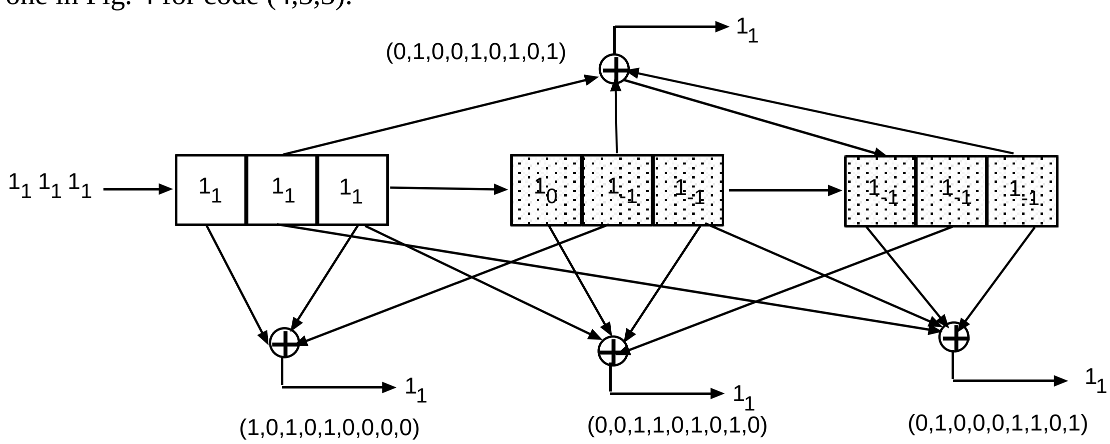


Figure 4 - This (4,3,3) convolutional code has 9 memory registers, 3 input bit and 4 output bits. The shaded registers contain “old” bits representing the current state.

Systematic vs. non-systematic CC

A special form of convolutional code in which the output bits contain an easily recognizable sequence of the input bits is called the systematic form. The systematic version of the above (4,3,3) code is shown below. Of the 4 output bits, 3 are exactly the same as the 3 input bits. The 4th bit is kind of a parity bit produced from a combination of 3 bits using a single polynomial.

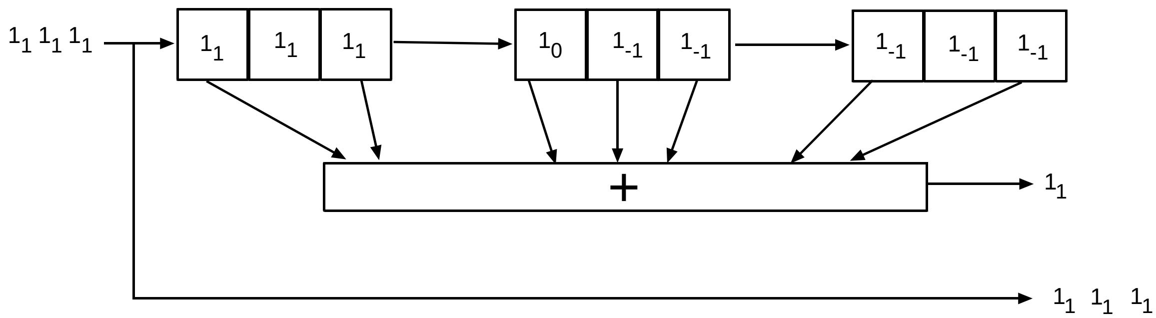


Figure 5 - The systematic version of the (4,3,3) convolutional code. It has the same number of memory registers, and 3 input bits and 4 output bits. The output bits consist of the original 3 bits and a 4th "parity" bit.

Systematic codes are often preferred over the non-systematic codes because they allow quick look. They also require less hardware for encoding. Another important property of systematic codes is that they are non "catastrophic", which means that errors can not propagate catastrophically. All these properties make them very desirable. Systematic codes are also used in Trellis Coded Modulation (TCM). The error protection properties of systematic codes however are the same as those non-systematic codes.

Coding an incoming sequence

The output sequence v, can be computed by convolving the input sequence u with the impulse response g. we can express this as

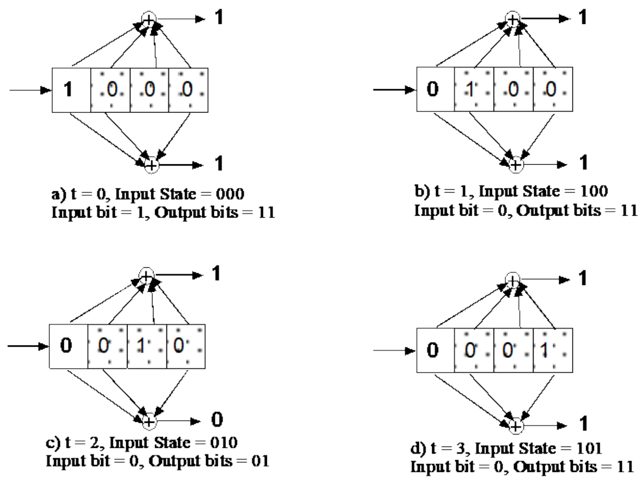
$$v = u * g$$

or in a more generic form

$$v_l^j = \sum_{i=0}^{m} u_{l-i}\, g_i^j$$

where v_l^j is the output bit l from the encoder j, and u_{l-i} is the input bit, and g_i^j is the i_{th} term in the polynomial j.

Let's encode a two bit sequence of 10 with the (2,1,4) code and see how the process works.



(There is a mistake in the figure in that 'Input bit =1' , in the bottom two figures, was replaced with 'Input bit = 0')

Figure 6 - A sequence consisting of a solo 1 bit as it goes through the encoder. The single bit produces 8 bit of output.

First we will pass a single bit 1 through this encoder as shown in Fig 6.
a) At time t = 0, we see that the initial state of the encoder is all zeros (the bits in the right most L register positions). The input bit 1 causes two bits 11 to be output. How did we compute that? By a mod2 sum of all bits in the registers for the first bit and a mod2 sum of three bits for second output bit per the polynomial coefficients.

b) At t = 1, the input bit 1 moves forward one register. The input register is now empty and is filled with a flush bit of 0. The encoder is now in state 100. The output bits are now again 11 by the same math.

c) As the input bit 1 moves forward again. Now the encoder state is 010 and another flush bit is moved into the input register. The output bits are now 10.

d) At time 3, the input bit moves to the last register and the input state is 001. The output bits are now 11. At time 4, the input bit 1 has passed completely thorough the encoder and the encoder has been flushed to an all zero state, ready for the next sequence.

Note that a single bit has produced an 8-bit output although nominally the code rate is ½. This shows that for small sequences the overhead is much higher than the nominal rate, which only applies to long sequences.

If we did the same thing with a 0 bit, we would get an 8 bit all zero sequence.

What we just produced is called the impulse response of this encoder. The 1 bit has a response of

11 11 10 11 which is called the impulse response. The 0-bit similarly has an impulse response of

00 00 00 00 (not shown but this is obvious)

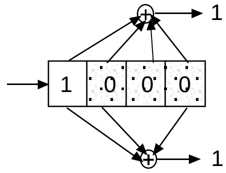
Convolving the input sequence with the code polynomials produced these two output sequences, which is why these codes are called convolutional codes. From the principle of linear superposition, we can now produce a coded sequence from the above two impulse responses as follows.

Let's say that we have a input sequence of 1011 and we want to know what the coded sequence would be. We can calculate the output by just adding the shifted versions of the individual impulse responses.

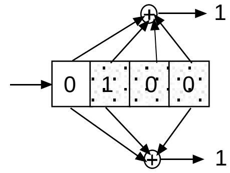
Input Bit	Its impulse response
1	11 11 10 11
0	00 00 00 00
1	11 11 10 11
1	11 11 10 11
Add to obtain response	
1011	11 11 01 11 01 01 11

We obtained the response to sequence 1011 by adding the shifted version of the responses for 1, and 0.

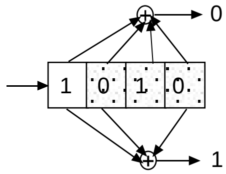
In Figure 7, we manually put the sequence 1011 through the encoder to verify the above and amazingly enough; we will get the same answer. This shows that that the convolution model is correct.



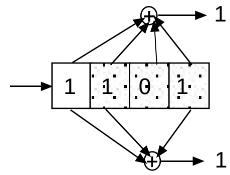
a) t = 0, Input State = 000
Input bit = 1, Output bits = 11



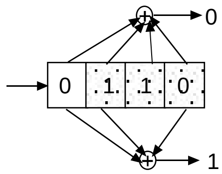
b) t = 1, Input State = 100
Input bit = 0, Output bits = 11



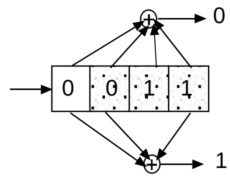
c) t = 2, Input State = 010
Input bit = 1, Output bits = 01



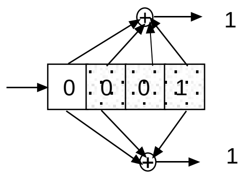
d) t = 3, Input State = 101
Input bit = 1, Output bits = 11



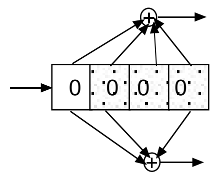
e) t = 4, Input State = 110,
Input bit = 0 (tail bit),Output bits = 01



f) t = 5, Input State = 011
Input bit = 0 (tail bit), Output bits = 01



g) t = 6, Input State = 001
Input bit = 0 (tail bit), Output bits = 11



h) t = 4,
Flushed and reset to Input State = 000

The result of the above encoding at each time interval is shown in Table 2.

Table 2 - Output Bits and the Encoder Bits through the (2,1,4 Code)
Input bits: 1011000

Time	Input Bit	Output Bits	Encoder Bits
0	1	11	000
1	0	11	100
2	1	01	010
3	1	11	101
4	0	01	110
5	0	01	011
6	0	11	001

The coded sequence is 11 11 01 11 01 01 11.

The encoder design

The two methods in the previous section show mathematically what happens in an encoder. The encoding hardware is much simpler because the encoder does no math. The encoder for convolutional code uses a table look up to do the encoding. The look up table consists of four items.

1. Input bit
2. The State of the encoder, which is one of the 8 possible states for the example (2,14) code
3. The output bits. For the code (2,1,4), since 2 bits are output, the choices are 00, 01, 10, 11.
4. The output state, which will be the input state for the next bit.

For the code (2,1,4) given the polynomials of Figure 7, I created the following look up table in an Excel worksheet.

Table 3 – Look-up Table for the encoder of code (2,1,4)

Input Bit	Input State				Output Bits			Output State		
I_1	s_1	s_2	s_3		O_1	O_2		s_1	s_2	s_3
0	0	0	0		0	0		0	0	0
1	0	0	0		1	1		1	0	0
0	0	0	1		1	1		0	0	0
1	0	0	1		0	0		1	0	0
0	0	1	0		1	0		0	0	1
1	0	1	0		0	1		1	0	1
0	0	1	1		0	1		0	0	1
1	0	1	1		1	0		1	0	1
0	1	0	0		1	1		0	1	0
1	1	0	0		0	0		1	1	0
0	1	0	1		0	0		0	1	0
1	1	0	1		1	1		1	1	0
0	1	1	0		0	1		0	1	1
1	1	1	0		1	0		1	1	1
0	1	1	1		1	0		0	1	1
1	1	1	1		0	1		1	1	1

This look up table uniquely describes the code (2,1,4). It is different for each code depending on the parameters and the polynomials used.

Graphically, there are three ways in which we can look at the encoder to gain better understanding of its operation. These are

1. State Diagram
2. Tree Diagram
3. Trellis Diagram.

State Diagram

A state diagram for the (2,1,4) code is shown in Fig 8. Each circle represents a state. At any one time, the encoder resides in one of these states. The lines to and from it show state transitions that are possible as bits arrive. Only two events can happen at each time, arrival of a 1 bit or arrival of a 0 bit. Each of these two events allows the encoder to jump into a different state. The state diagram does not have time as a dimension and hence it tends to be not intuitive.

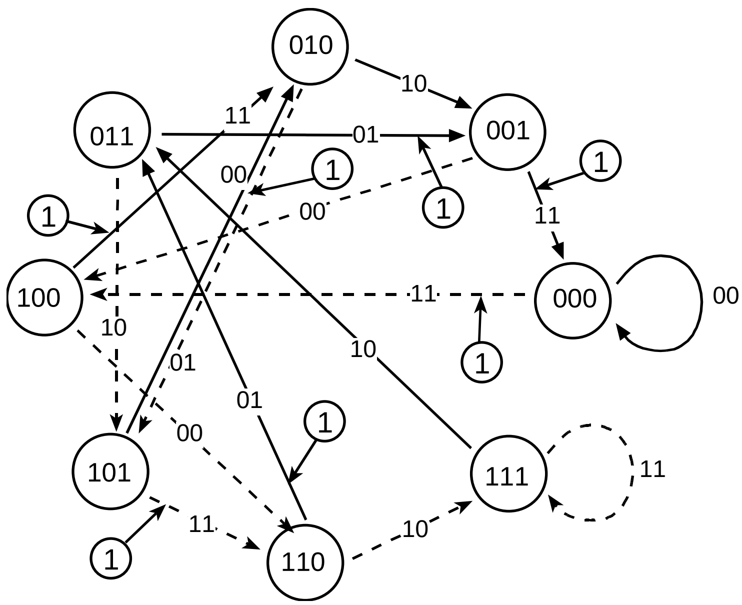


Figure 8 – State diagram of (2,1,4) code

Compare the above State diagram to the encoder lookup table. The state diagram contains the same information that is in the table lookup but it is a graphic representation. The solid lines indicate the arrival of a 0 and the dashed lines indicate the arrival of a 1. The output bits for each case are shown on the line and the arrow indicates the state transition.

Once again let's go back to the idea of states. Imagine you are on an island. How many ways can you leave this island? You can take a boat or you can swim. You have just these two possibilities to leave this small island. You take the boat to the mainland. Now how many ways can you move around? You can take boat but now you can also drive to other destinations. Where you are (your state) determines how many ways you can travel (your output).

Some encoder states allow outputs of 11 and 00 and some allow 01 and 10. No state allows all four options.

How do we encode the sequence 1011 using the state diagram?
(1) Let's start at state 000. The arrival of a 1 bit outputs 11 and puts us in state 100.
(2) The arrival of the next 0 bit outputs 11 and put us in state 010.
(3) The arrival of the next 1 bit outputs 01 and puts us in state 101.
(4) The last bit 1 takes us to state 110 and outputs 11. So now we have 11 11 01 11.
But this is not the end. We have to take the encoder back to all zero state.

(5) From state 110, go to state 011 outputting 01.
(6) From state 011 we go to state 001 outputting 01 and then
(7) To state 00 with a final output of 11.

The final answer is: 11 11 01 11 01 01 11

This is the same answer as what we got by adding up the individual impulse responses for bits 1011000.

Tree Diagram

Figure 9 shows the Tree diagram for the code (2,14). The tree diagram attempts to show the passage of time as we go deeper into the tree branches. It is somewhat better than a state diagram but still not the preferred approach for representing convolutional codes.

Here instead of jumping from one state to another, we go down branches of the tree depending on whether a 1 or 0 is received.

The first branch in Fig 9 indicates the arrival of a 0 or a 1 bit. The starting state is assumed to be 000. If a 0 is received, we go up and if a 1 is received, then we go downwards. In figure 9, the solid lines show the arrival of a 0 bit and the shaded lines the arrival of a 1 bit. The first 2 bits show the output bits and the number inside the parenthesis is the output state.

Let's code the sequence 1011 as before. At branch 1, we go down. The output is 11 and we are now in state 100 (According to figure 9 and not 111). Now we get a 0 bit, so we go up. The output bits are 11 and the state is now 011.

The next incoming bit is 1. We go downwards and get an output of 01 and now the output state is 101.

The next incoming bit is 1, so we go downwards again and get output bits 11. From this point, in response to a 0 bit input, we get an output of 01 and an output state of 011.

What if the sequence were longer, then what? We have run out of tree branches. The tree diagram now repeats. In fact we need to flush the encoder so our sequence is actually 1011 000, with the last 3 bits being the flush bits.

We now jump to point 2 in the tree and go upwards for three branches. Now we have the output to the complete sequence and it is

11 11 01 11 01 01 11

Perhaps you are not surprised that this is also the same answer as the one we got from the state diagram.

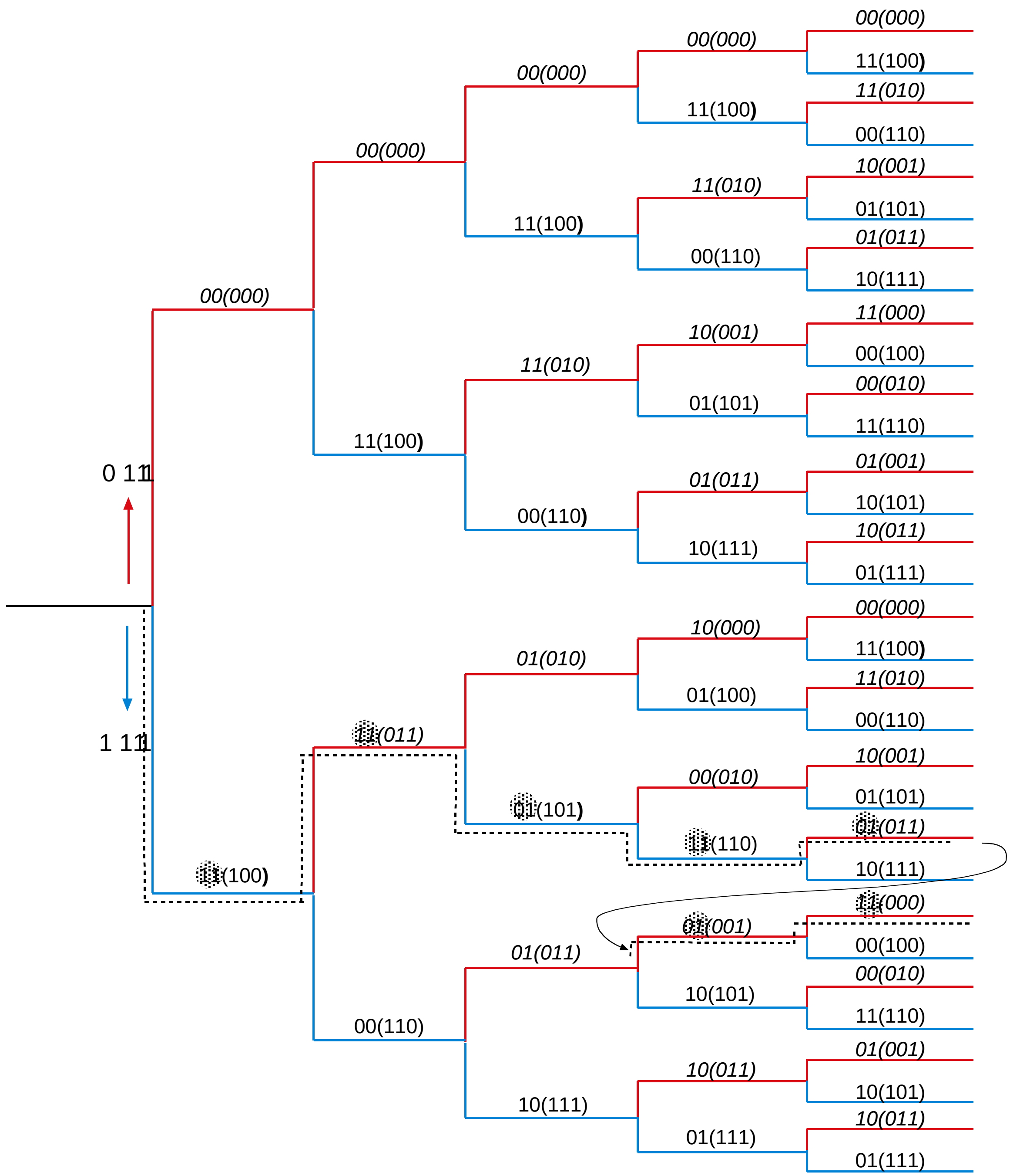


Figure 9 – Tree diagram of (2,1,4) code

Trellis Diagram

Trellis diagrams are messy but generally preferred over both the tree and the state diagrams because they represent linear time sequencing of events. The x-axis is discrete time and all possible states are shown on the y-axis. We move horizontally through the trellis with the passage of time. Each transition means new bits have arrived.

The trellis diagram is drawn by lining up all the possible states (2^L) in the vertical axis. Then we connect each state to the next state by the allowable codewords for that state. There are only two choices possible at each state. These are determined by the arrival of either a 0 or a 1 bit. The arrows show the input bit and the output bits are shown in parentheses. The arrows going upwards represent a 0 bit and going downwards represent a 1 bit. The trellis diagram is unique to each code, same as both the state and tree diagrams are. We can draw the trellis for as many periods as we want. Each period repeats the possible transitions.

We always begin at state 000. Starting from here, the trellis expands and in L bits becomes fully populated such that all transitions are possible. The transitions then repeat from this point on.

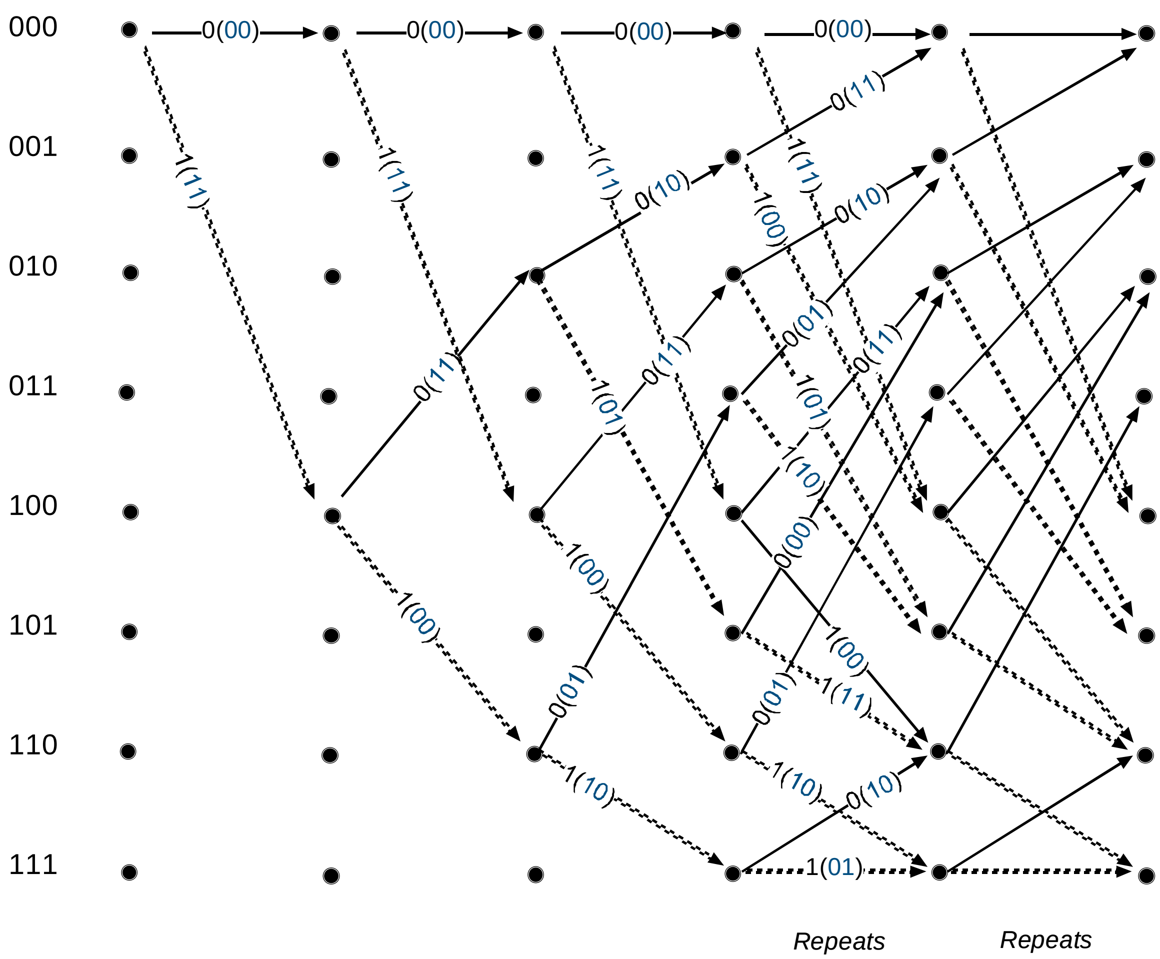


Figure 10 – Trellis diagram of (2,1,4) code

How to encode using the trellis diagram

In Fig 10a, the incoming bits are shown on the top. We can only start at point 1. Coding is easy. We simply go up for a 0 bit and down for a 1 bit. The path taken by the bits of the example sequence (1011000) is shown by the lines. We see that the trellis diagram gives exactly the same output sequence as the other three methods, namely the impulse response, state and the tree diagrams. All of these diagrams look similar but, we should recognize that they are unique to each code.

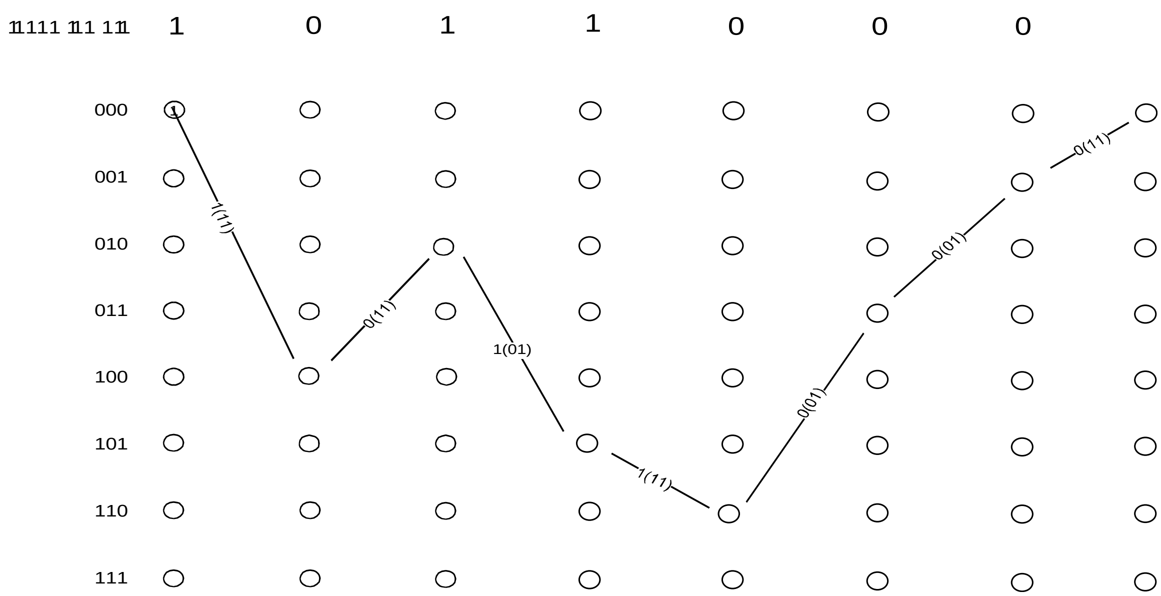


Figure 10a – Encoded Sequence, Input bits 1011000, Outbit 11011111010111

Decoding

There are several different approaches to decoding of convolutional codes. These are grouped in two basic categories.

1.Sequential Decoding
- Fano algorithm

2. Maximum likely-hood decoding
- Viterbi decoding

Both of these methods represent 2 different approaches to the same basic idea behind decoding.

The basic idea behind decoding

Assume that 3 bits were sent via a rate ½ code. We receive 6 bits. (Ignore flush bits for now.) These six bits may or may not have errors. We know from the encoding process that these bits map uniquely. So a 3 bit sequence will have a unique 6 bit output. But due to errors, we can receive any and all possible combinations of the 6 bits.

The permutation of 3 input bits results in eight possible input sequences. Each of these has a unique mapping to a six bit output sequence by the code. These form the set of permissible sequences and the decoder's task is to determine which one was sent.

Table 4 – Bit Agreement used as metric to decide between the received sequence and the 8 possible valid code sequences

Input	Valid Code Sequence	Received Sequence	Bit Agreement
000	000000	111100	2
001	000011	111100	0
010	001111	111100	2
011	001100	111100	4
100	**111110**	111100	**5**
101	**111101**	111100	**5**
110	110001	111100	3
111	110010	111100	3

Let's say we received 111100. It is not one of the 8 possible sequences above. How do we decode it? We can do two things

1. We can compare this received sequence to all permissible sequences and pick the one with the smallest Hamming distance(or bit disagreement)

2. We can do a correlation and pick the sequences with the best correlation.

The first procedure is basically what is behind hard decision decoding and the second the soft-decision decoding. The bit agreements, also the dot product between the received sequence and the codeword, show that we still get an ambiguous answer and do not know what was sent.

As the number of bits increase, the number of calculations required to do decoding in this brute force manner increases such that it is no longer practical to do decoding this way. We need to find a more efficient method that does not examine all options and has a way of resolving ambiguity such as here where we have two possible answers. (Shown in bold in Table 4)

If a message of length *s* bits is received, then the possible number of codewords are 2^s. How can we decode the sequence without checking each and everyone of these 2^s codewords? This is the basic idea behind decoding.

Sequential decoding

Sequential decoding was one of the first methods proposed for decoding a convolutionally coded bit stream. It was first proposed by Wozencraft and later a better version was proposed by Fano.

Sequential decoding is best described by analogy. You are given some directions to a place. The directions consist of some landmarks. But the person who gave the directions has not done a very good job and occasionally you do not recognize a landmark and you end up on a wrong path. But since now you do not see any more landmarks, you feel that you may be on a wrong path. You backtrack to a point where you do recognize a landmark and then take an alternate path until you see the next landmark and ultimately in this fashion you reach your destination. You may back track several times in the process depending on how good the directions were.

In Sequential decoding similarly you are dealing with just one path at a time. You may give up that path at any time and turn back to follow an another path but important thing is that only one path is followed at any one time.

Sequential decoding allows both forwards and backwards movement through the trellis. The decoder keeps track of its decisions, each time it makes an ambiguous decision, it tallies it. If the tally increases faster than some threshold value, decoder gives up that path and retraces the path back to the last fork where the tally was below the threshold.

Let’s do an example. The code is: (2,1,4) for which we did the encoder diagrams in the previous section. Assume that bit sequence 1011 000 was sent. (Remember the last 3 bits are the result of the flush bits. Their output is called tail bits.)

If no errors occurred, we would receive: 11 11 01 11 01 01 11
But let’s say we received instead : **<u>0</u>**1 11 01 11 01 01 11. One error has occurred. The first bit has been received as 0.

Decoding using a sequential decoding algorithm

1. Decision Point 1 The decoder looks at the first two bits, 01. Right away it sees that an error has occurred because the starting two bits can only be 00 or 11. But which of the two bits was received in error, the first or the second? The decoder randomly selects 00 as the starting choice. To correspond to 00, it decodes the input bit as a 0. It puts a count of 1 into its error counter. It is now at point 2.

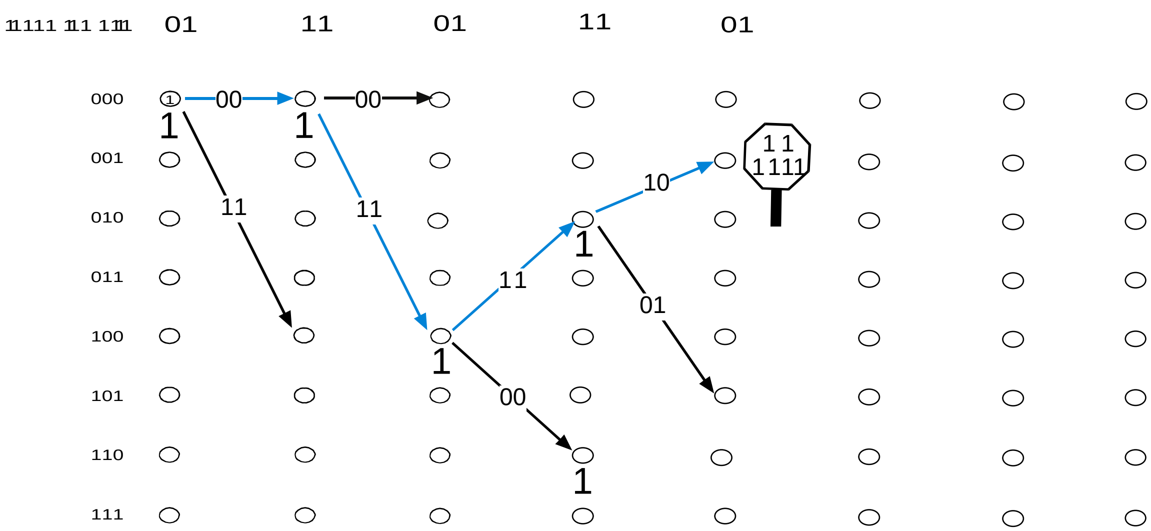


Figure 11a – Sequential decoding path search

2. Decision point 2 The decoder now looks at the next set of bits, which are 11. From here, it makes the decision that a 1 was sent which corresponds exactly with one of the codewords. This puts it at point 3.

3. At Decision point 3, the received bits are 01, but the codeword choices are 11, 00. This is seen as an error and the error count is increased to 2. Since error count is less than the threshold value of 3 (which we have set based on channel statistics) the decoder proceeds ahead. It arbitrarily selects the upper path and proceeds to point 4 making a decision that a 0 was sent.

4. At Decision point 4, it recognizes another error since the received bits are 11 but the codeword choices are 10, 01. The error tally increases to 3 and that tells the decoder to turn back.

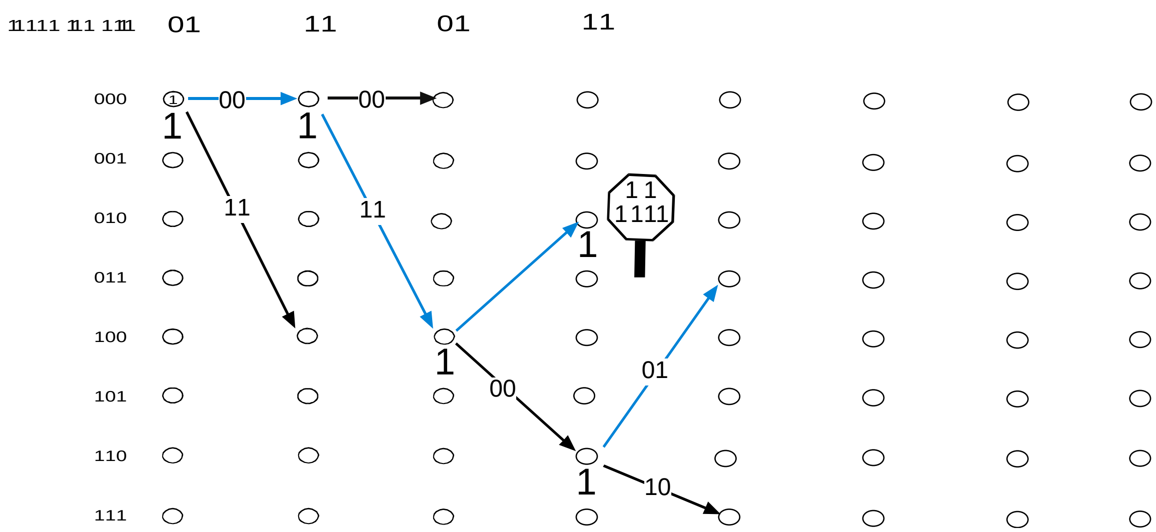


Figure 11b – Sequential decoding path search

5. The decoder goes back to point 3 where the error tally was less than 3 and takes the other choice to point 5. It again encounters an error condition. The received bits are 11 but the codewords possible are 01, 10. The tally has again increased to 3. It turns back again

6. Both possible paths from point 3 have been exhausted. The decoder must go further back than point 3. It goes back to pint 2. But here if it follows to point 2 the error tally immediately goes up to 3. So it must turn back from point 2 back to point 1.

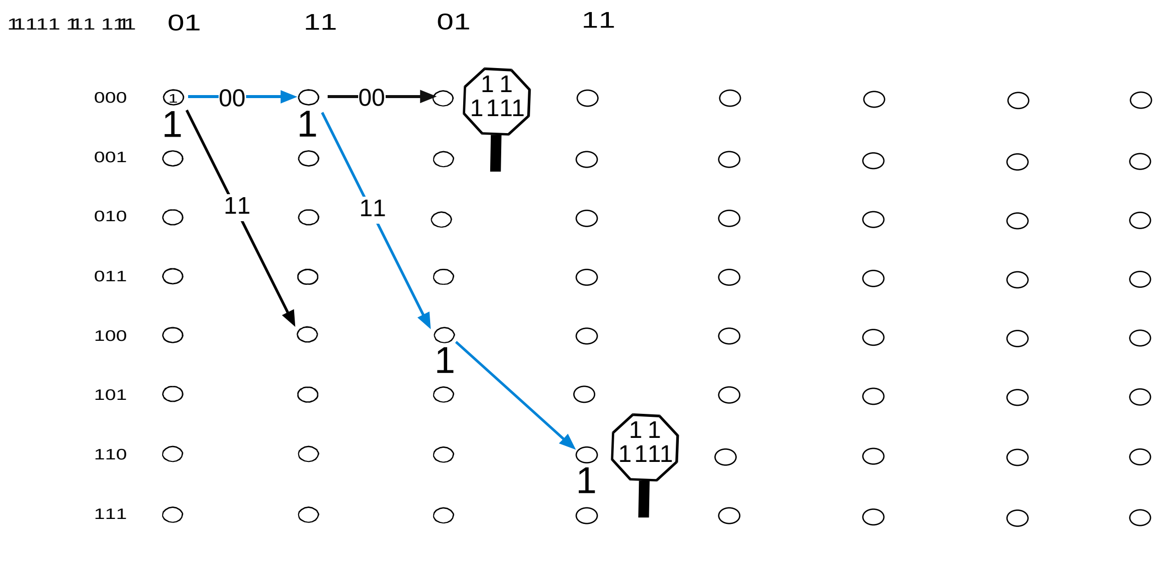


Figure 11c – Sequential decoding path search

From point 1, all choices encountered meet perfectly with the codeword choices and the decoder successfully decodes the message as 1011000

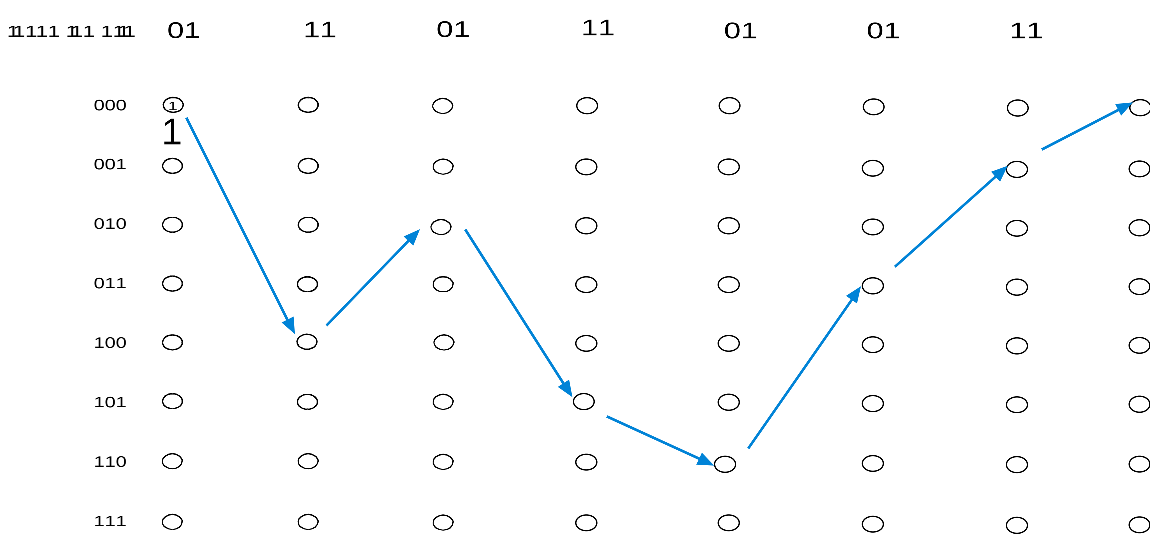


Figure 11e – Sequential decoding path search

The memory requirement of sequential decoding are manageable and so this method is used with long constraint length codes where the S/N is also low. Some NASA planetary mission links have used sequential decoding to advantage.

Maximum Likelihood and Viterbi decoding

Viterbi decoding is the best known implementation of the maximum likely-hood decoding. Here we narrow the options systematically at each time tick. The principal used to reduce the choices is this.

1. The errors occur infrequently. The probability of error is small.
2. The probability of two errors in a row is much smaller than a single error, that is the errors are distributed randomly.

The Viterbi decoder examines an entire received sequence of a given length. The decoder computes a metric for each path and makes a decision based on this metric. All paths are followed until two paths converge on one node. Then the path with the higher metric is kept and the one with lower metric is discarded. The paths selected are called the survivors.

For an N bit sequence, total numbers of possible received sequences are 2^N. Of these only 2^{kL} are valid. The Viterbi algorithm applies the maximum-likelihood principles to limit the comparison to 2 of the power of kL surviving paths instead of checking all paths.

The most common metric used is the Hamming distance metric. This is just the dot product between the received codeword and the allowable codeword. Other metrics are also used and we will talk about these later.

Table 5 – Each branch has a Hamming metric depending on what was received and the valid codewords at that state

Bits Received	**Valid Codeword 1**	**Valid Codeword 2**	**Hamming Metric 1**	**Hamming Metric 2**
00	00	11	2	0
01	10	01	0	2
10	00	11	1	1

These metrics are cumulative so that the path with the largest total metric is the final winner.

But all of this does not make much sense until you see the algorithm in operation.

Let’s decode the received sequence 01 11 01 11 01 01 11 using Viterbi decoding.

1. At t = 0, we have received bit 01. The decoder always starts at state 000. From this point it has two paths available, but neither matches the incoming bits. The decoder computes the branch metric for both of these and will continue simultaneously along both of these branches in contrast to the sequential decoding where a choice is made at every decision point. The metric for both branches is equal to 1, which means that one of the two bits was matched with the incoming bits.

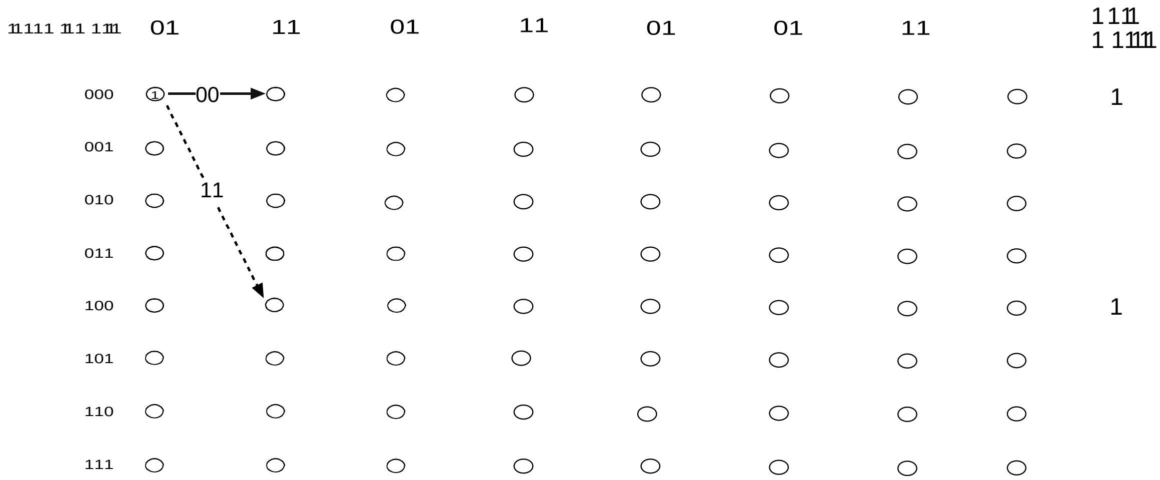


Figure 12a – Viterbi Decoding, Step 1

2. At t = 1, the decoder fans out from these two possible states to four states. The branch metrics for these branches are computed by looking at the agreement with the codeword and the incoming bits, which are 11. The new metric is shown on the right of the trellis.

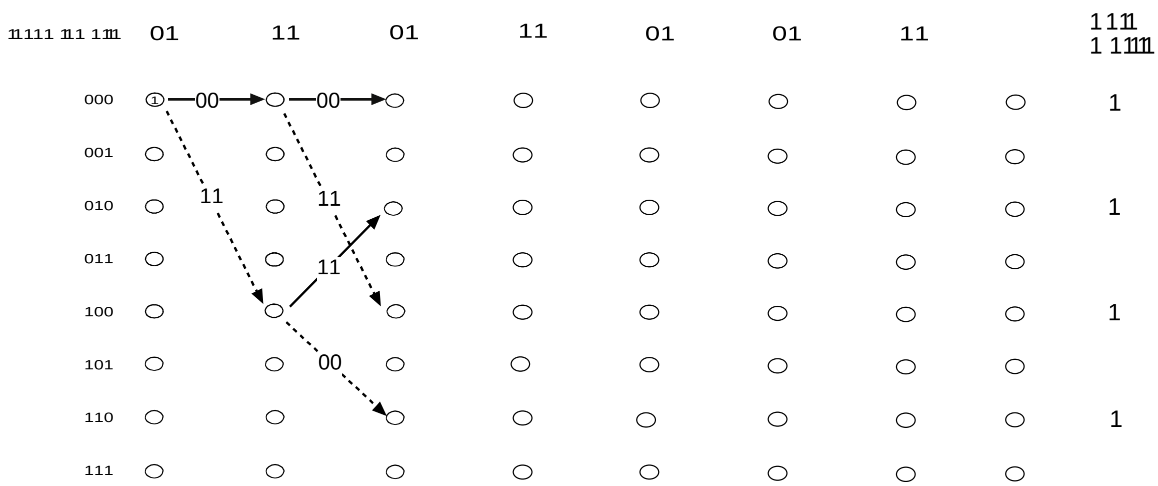


Figure 12b – Viterbi Decoding, Step 2

3. At t = 2, the four states have fanned out to eight to show all possible paths. The path metrics calculated for bits 01 and added to pervious metrics from t = 1.

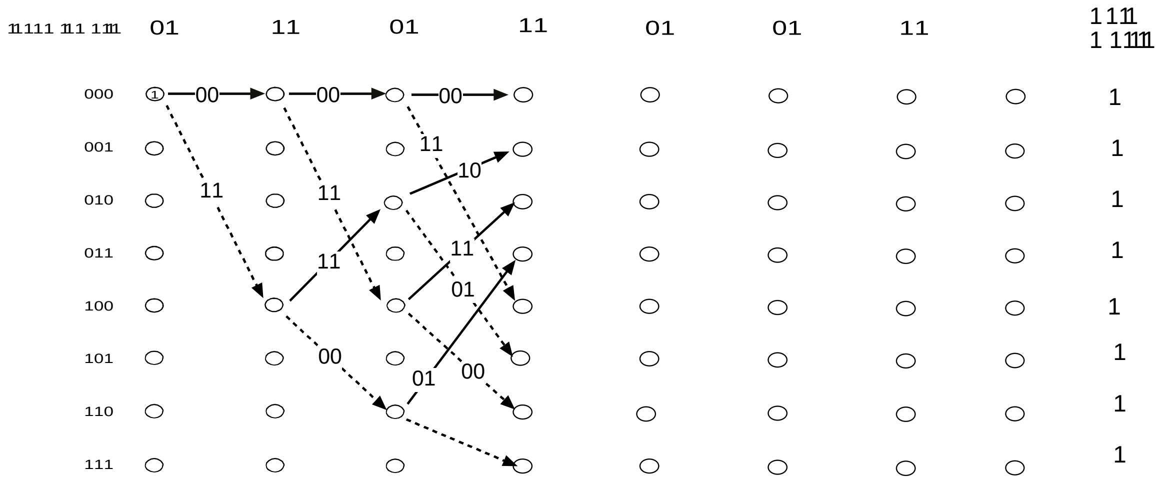


Figure 12c – Viterbi Decoding, Step 3

At t = 4, the trellis is fully populated. Each node has at least one path coming into it. The metrics are as shown in the figure above.

At t = 5, the paths progress forward and now begin to converge on the nodes. Two metrics are given for each of the paths coming into a node. Per the Maximum likelihood principle, at each node we discard the path with the lower metric because it is least likely. This discarding of paths at each node helps to reduce the number of paths that have to be examined and gives the Viterbi method its strength.

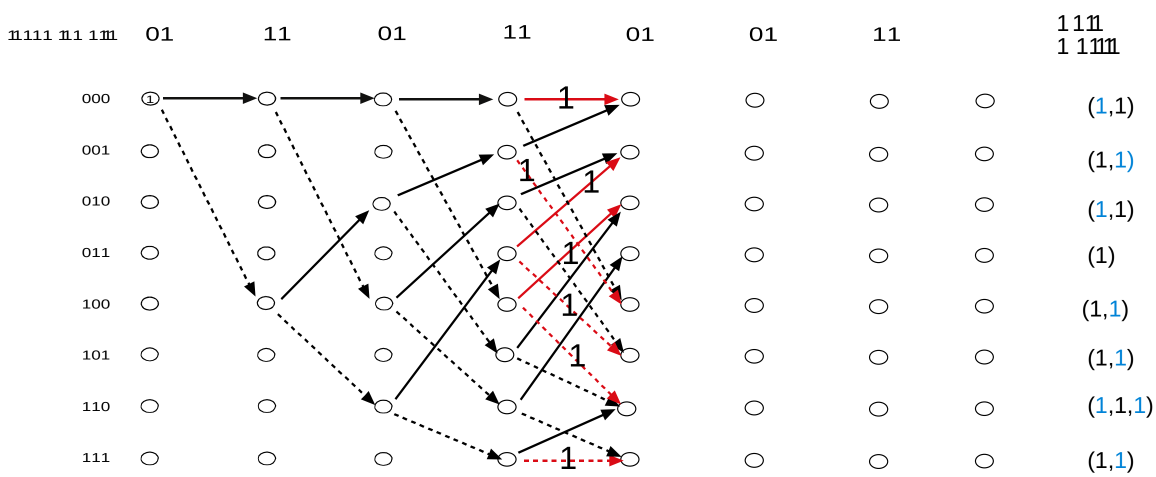


Figure 12d – Viterbi Decoding, Step 4

Now at each node, we have one or more path converging. The metrics for all paths are given on the right. At each node, we keep only the path with the highest metric and discard all others, marked with an X (shown in red). After discarding the paths with the smaller metric, we have the following paths left.
The metric shown is that of the winner path.

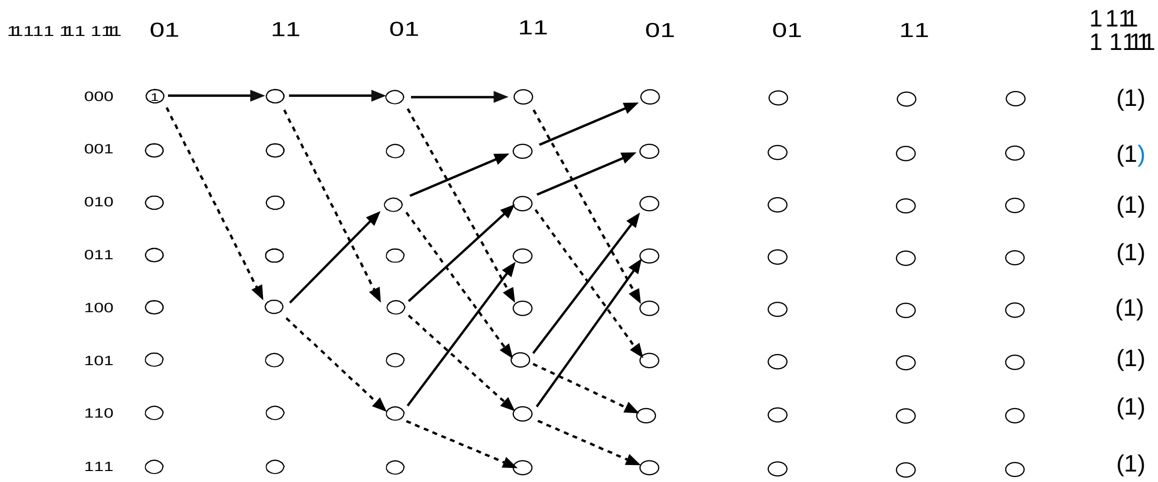


Figure 12e – Viterbi Decoding, Step 4, after discarding

At t = 5, after discarding the paths as shown, we again go forward and compute new metrics. At the next node, again the paths converge and again we discard those with lower metrics.

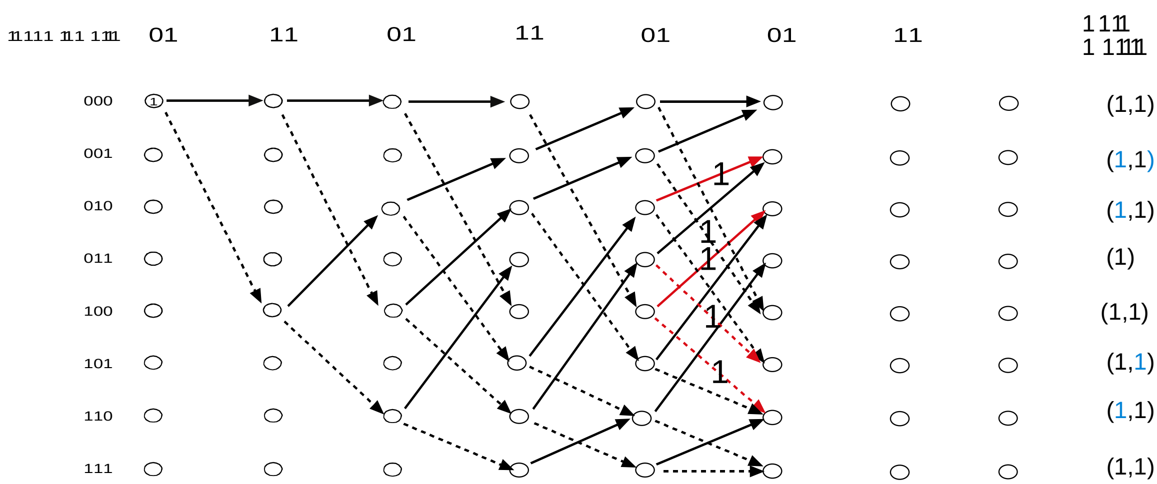


Figure 12f – Viterbi Decoding, Step 5

At t= 6, the received bits are 11. Again the metrics are computed for all paths. We discard all smaller metrics but keep both if they are equal. ()

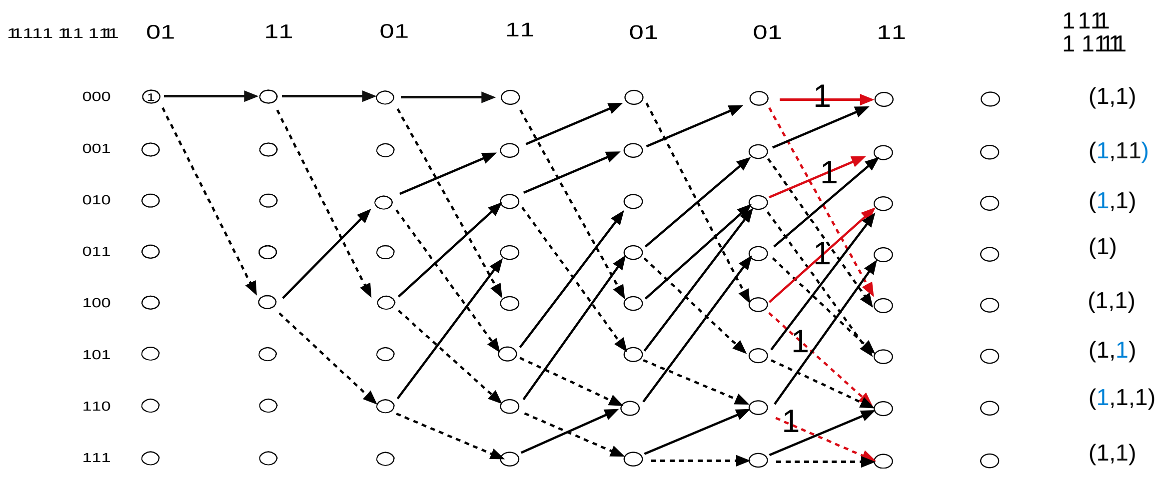


Figure 12g – Viterbi Decoding, Step 6

The 7-step trellis is complete. We now look at the path with the highest metric. We have a winner. The path traced by states 000, 100, 010, 101, 110, 011, 001, 000 and corresponding to bits 1011000 is the decoded sequence.

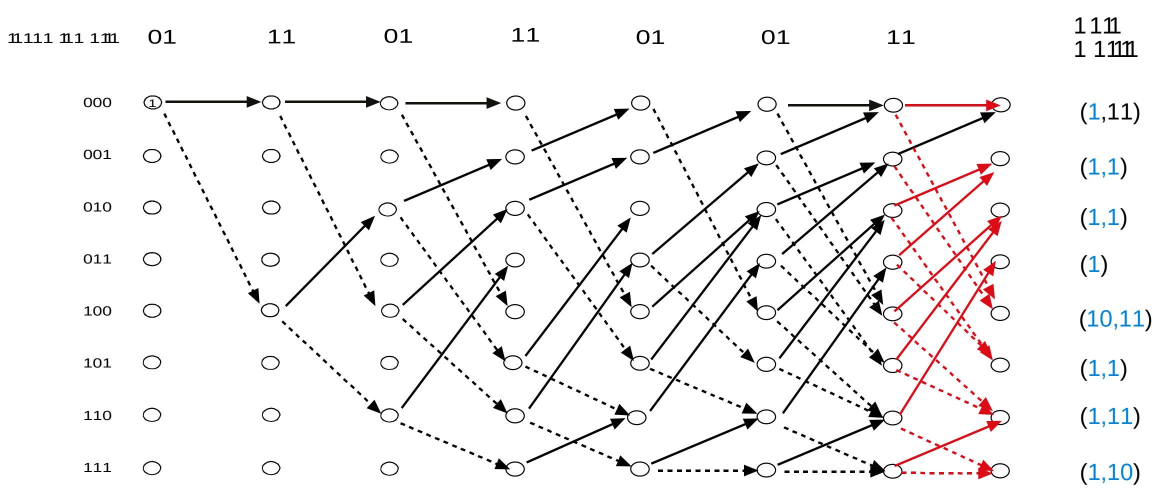
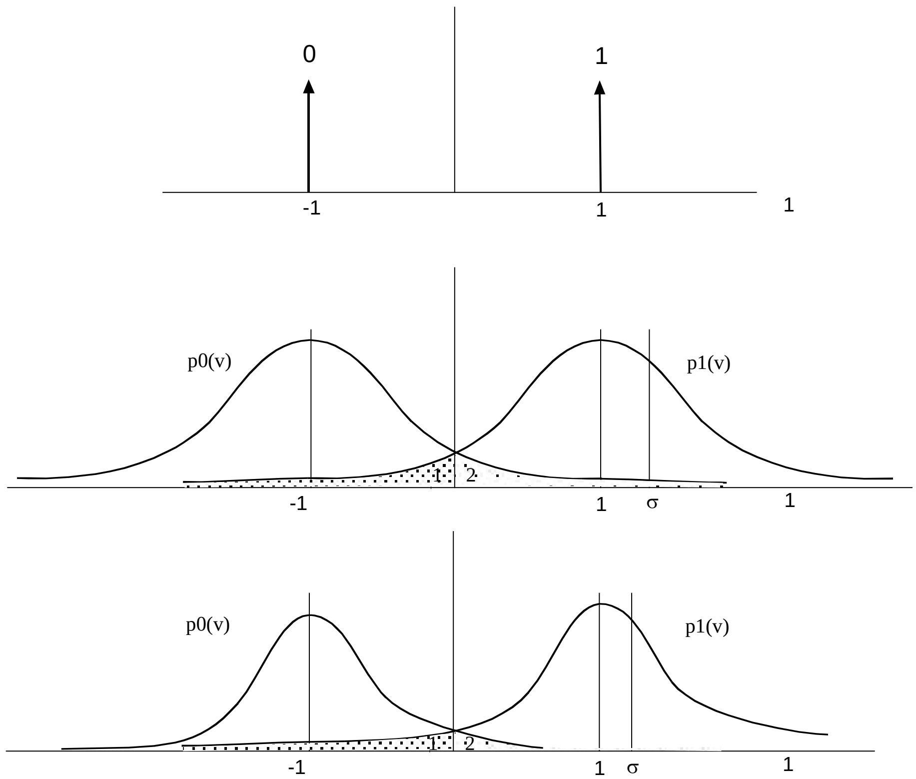


Figure 12h – Viterbi Decoding, Step 7

The length of this trellis was 4bits + m bits. Ideally this should be equal to the length of the message, but by a truncation process, storage requirements can be reduced and decoding need not be delayed until the end of the transmitted sequence. Typical Truncation length for convolutional coders is 128 bits or 5 to 7 times the constraint length.

Viterbi decoding is quite important since it also applies to decoding of block codes. This form of trellis decoding is also used for Trellis-Coded Modulation (TCM).

Soft-decision decoding



The spectrum of two signals used to represent a 0 and 1 bit when corrupted by noise makes the decision process difficult. The signal spreads out and the energy from one signal flows into the other. Fig 13 b, c, shows two different cases for different S/N. If the added noise is small, i.e. its variance is small (since noise power = variance), then the spread will be smaller as we can see in c as compared to b. Intuitively we can see from these pictures that we are less likely to make decoding errors if the S/N is high or the variance of the noise is small.

Figure 13 – a) Two signals representing a 1 and 0 bit. b) Noise of S/N = 2 spreads out the signal, c) Noise of S/N = 4 spread out to spill energy from one decision region to another.

Making a hard decision means that a simple decision threshold which is usually between the two signals is chosen, such that if the received voltage is positive then the signal is decoded as a 1 otherwise a 0. In very simple terms this is also what the Maximum likelihood decoding means.

We can quantify the error that is made with this decision method. The probability that a 0 will be decoded, given that a 1 was sent is a function of the two shaded areas you see in the figure above. The area 1 represent that energy belonging to the signal for 1 that has landed in the opposite decision region and hence erroneously leads to the decoding of bit as a 0 and area 2 which is the energy from the bit 0 that has landed in the region of interest. It subtracts from the voltage received hence it too potentially causing an error in decision.

Given that a 1 was sent, the probability that it will be decoded as 0 is

$$Pe1 = \frac{1}{2}\, erfc(\frac{A - v_t}{\sqrt{2}\sigma})$$

v_t = decision threshold voltage, which is 0 for our case.

σ = Variance of noise or its power.

We can rewrite this equation as a function of S/N

$$Pe1 = \frac{1}{2}\, erfc(\sqrt{\frac{S}{N}})$$

This is the familiar bit error rate equation. We see that it assumes that hard-decision is made. But now what if we did this; instead of having just two regions of decision, we divided the area into 4 regions as shown below.

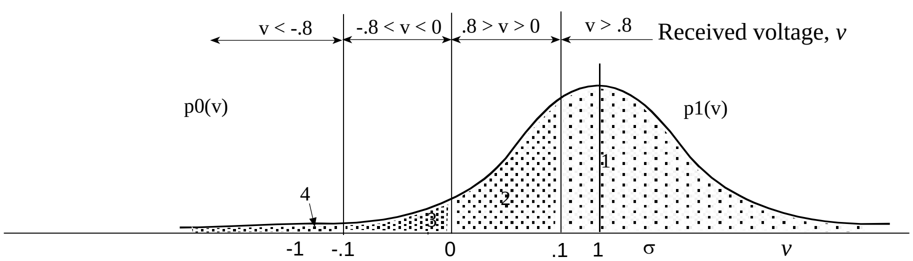


Figure 14 – Creating four regions for decision
The probability that the decision is correct can be computed from the area under the gaussian curve.

I have picked four regions as follows:

Region 1 = if received voltage is greater than .8 v
Region 2 = if received voltage is greater than 0 but less than .8 v
Region 3 = if received voltage is greater than -.8 v but less than 0 v
Region 4 = if received voltage is less than -.8 v

Now we ask? If the received voltage falls in region 3, then what is probability of error that a 1 was sent?

With hard-decision, the answer is easy. It can be calculated from the equation above. How do we calculate similar probabilities for a multi-region space?

We use the Q function that is tabulated in many books. The Q function gives us the area under the tail defined by the distance from the mean to any other value. So Q(2) for a signal the mean of which is 2 would give us the probability of a value that is 4 or greater.

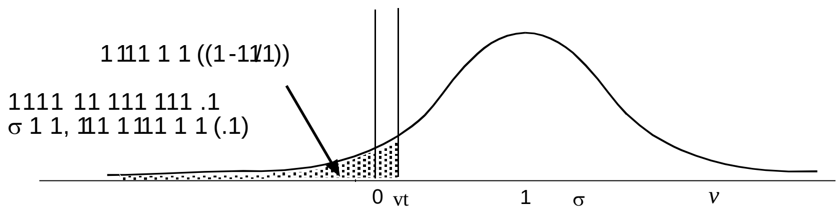


Figure 15 – The Q function is an easy way to determine probabilities of a normally distributed variable

I have assumed a v_t of .8 (this is a standard sort of number for 4-level) but it can be any number. The equations can be written almost by inspection, they are so obvious.

Pe1 (probability that a 1 was sent if the received voltage is in Region 1) = 1 – Q(A- v_t /σ)
Pe4 (probability that a 1 was sent if the received voltage is in Region 4) = Q(2(A+v_t)/ σ)
Pe2 (probability that a 1 was sent if the received voltage is in Region 2) = 1- Pe1-Q(A/ σ)
Pe3 (probability that a 1 was sent if the received voltage is in Region 3) = 1- Pe1- Pe2 – Pe4

I have computed these for S/N = 1 assuming that v_t is .8A and A = 1.0. We have also assumed that both bits 0 and 1 are equiprobable. This is also called the apriori probabilities for bits 0 and 1 and these are nearly always assumed to be equal in communications except in Radar applications, where apriori probabilities are usually not known.

So the area of Region 4 or Pe4 is equal to Q(1.8) which we can look up from the Tables. The area of Region 1 is then equal to 1-Q(.2). The others can be quickly calculated in this manner. We can show the conditional probabilities calculated in a graphically manner.

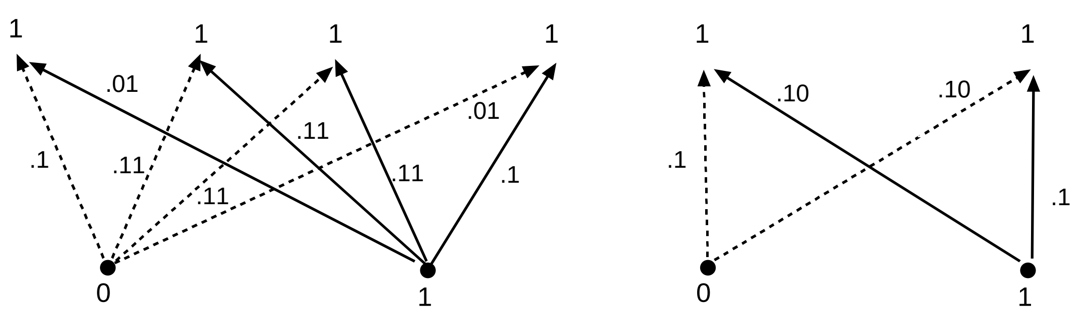


Figure 16 – Conditional probabilities of decoding a 0 or 1 depending on the magnitude of the voltage received, when the voltage is quantized into 4 levels vs. 2.

This process of subdividing the decision space into regions greater than two such as this 4-level example is called soft-decision. These probabilities are also called the transition probabilities. There are 4 different values of voltage for each signal received with which to make a decision. In signals, as in real life, more information means better decisions. Soft-decision improve the sensitivity of the decoding metrics and improves the performance by as much as 3 dB in case of 8-level soft decision.

Now to compute soft decision metric we make a table of the above probabilities.

Sent	v4	v3	v2	v1
1	.03	.12	.25	.60
0	.6	.25	.12	.03

Now take the natural log of each of these and normalize so that one of the values is 0. After some number manipulation, we get

Sent	v4	v3	v2	v1
1	.0	-1	-4	-10
0	.-10	-4	-1	0

In the decoding section, we calculated a Hamming metric by multiplying the received bits with the codewords. We do the same thing now, except, instead of receiving 0 and 1, we get one of these voltages. The decoder looks up the metric for that voltage from a table such as above it has in its memory and makes the following calculation.

Assume voltage pair (v3, v2) are received. The allowed codewords are 01, 10.

Metric for 01 = p(0|v3) + p(1|v2) = -4 + -4 = -8
Metric for 10 = p(1|v3) + p(0|v2) = -1 + -1 = -2

We can tell just by looking at these two that 01 is much more likely than 10. When these metric add, they exaggerate the differences and help the decoding results.

Log-likelyhood ratio

There are other ways to improve decoding performance by playing around with the metrics. One important metric to know about is called the log likelihood metric. This metric takes into account the channel error probability and is defined by

Metric for agreement: = $\dfrac{Log_{10}\ 2(1-p)}{Log_{10}\ 2}$

Metric for disagreement = : $\dfrac{Log_{10}\ 2(p)}{Log_{10}\ 2}$

For p = .1
Metric for agreement is = -20
Metric for disagreement = -1

So if we have received bits 01 and the codeword is 00, the total metric would be –20 + -1 = -21 and the metric for complete agreement would be –40.

Fano algorithm used for sequential decoding uses a slightly different metric and the purpose of all these is to improve the sensitivity.

References:

S. Lin and D. J. Costello, Error Control Coding. Englewood Cliffs, NJ: Prentice Hall, 1982.

A. M. Michelson and A. H. Levesque, Error Control Techniques for Digital Communication. New York: John Wiley
& Sons, 1985.

W. W. Peterson and E. J. Weldon, Jr., Error Correcting Codes, 2nd ed. Cambridge, MA: The MIT Press, 1972.

V. Pless, Introduction to the Theory of Error-Correcting Codes, 3rd ed. New York: John Wiley & Sons, 1998.

C. Schlegel and L. Perez, Trellis Coding. Piscataway, NJ: IEEE Press, 1997

S. B. Wicker, Error Control Systems for Digital Communication and Storage. Englewood Cliffs, NJ: Prentice Hall,
1995.

Charan Langton
Complextoreal.com
Email: charldsp@gmail.com

July 1999 / Some errors corrected as noted here.