Communications Source and Channel Coding with examples

By:

Peter Grant

Communications Source and Channel Coding with examples

By:
Peter Grant

Online:
< http://cnx.org/content/col10601/1.3/ >

C O N N E X I O N S

Rice University, Houston, Texas

Collection structure revised: May 7, 2009
PDF generated: February 5, 2011
For copyright and attribution information for the modules contained in this collection, see p. 43.

Table of Contents

1 Huffman source coder 1
2 Block FECC coding 7
3 Block code performance 13
4 Convolutional FECC Encoder 25
5 Viterbi Decoder 31
6 Turbo Coding 37
Index 42
Attributions 43

Chapter 1

Huffman source coder[1]

1.1 Source coding

Huffman coding deploys variable length coding and then allocates the longer codewords to less frequently occurring symbols and shorter codewords to more regularly occurring symbols. By using this technique it can minimize the overall transmission rate as the regularly occurring symbols are allocated the shorter codewords.

1.1.1 Simple source coding

Symbol	Probability
A	0.10
B	0.18
C	0.40
D	0.05
E	0.06
F	0.10
G	0.07
H	0.04

Table 1.1: 8-symbol signal to be encoded

We have to start with knowledge of the probabilities of occurrence of all the symbols in the alphabet. The table above shows an example of an 8-symbol alphabet, A...H, with the associated probabilities for each of the eight individual symbols.

[1] This content is available online at <http://cnx.org/content/m18172/1.4/>.

The entropy of this source is:-

$$H =$$
$$-0.10\log_2(0.10) - 0.18\log_2(0.18) - 0.40\log_2(0.40)$$
$$-0.05\log_2(0.05) - 0.06\log_2(0.06) - 0.10\log_2(0.1000)$$
$$-0.07\log_2(0.07) - 0.04\log_2(0.04) = 2.5524 \quad \text{bits/symbol}$$

Figure 1.1: Source encoder entropy calculation

Figure 1.1 shows that the entropy of this source data is 2.5524 bits/symbol.

Symbol	Code
A	000
B	001
C	010
D	011
E	100
F	101
G	110
H	111

Table 1.2: Simple fixed length (3-bit) encoder

This shows the application of very simple coding where, as there are 8 symbols, we adopt a 3-bit code. Figure 1.1 shows that the entropy of such a source is 2.5524 bit/symbol and, with the fixed 3 bit/symbol

length allocated codewords, the efficiency of this simple coder would be only 2.5524/3.0 = 85.08%, which is a rather poor result.

1.1.2 Huffman coding

This is a variable length coding technique which involves two processes, reduction and splitting.

1.1.2.1 Reduction

We start by listing the symbols in descending order of probability, with the most probable symbol, C, at the top and the least probable symbol, H, at the foot, see left hand side of Figure 1.2. Next we reduce the two least probable symbols into a single symbol which has the combined probability of these two symbols summed together. Thus symbols H and D are combined into a single (i.e. reduced) symbol with probability 0.04 + 0.05 = 0.09.

Now the symbols have to be reordered again in descending order of probability. As the probability of the new H+D combined symbol (0.09) is no longer the smallest value it then moves up the reordered list as shown in the second left column in Figure 1.2.

This process is progressively repeated as shown in Figure 1.2 until all symbols are combined into a single symbol whose probability must equal 1.00.

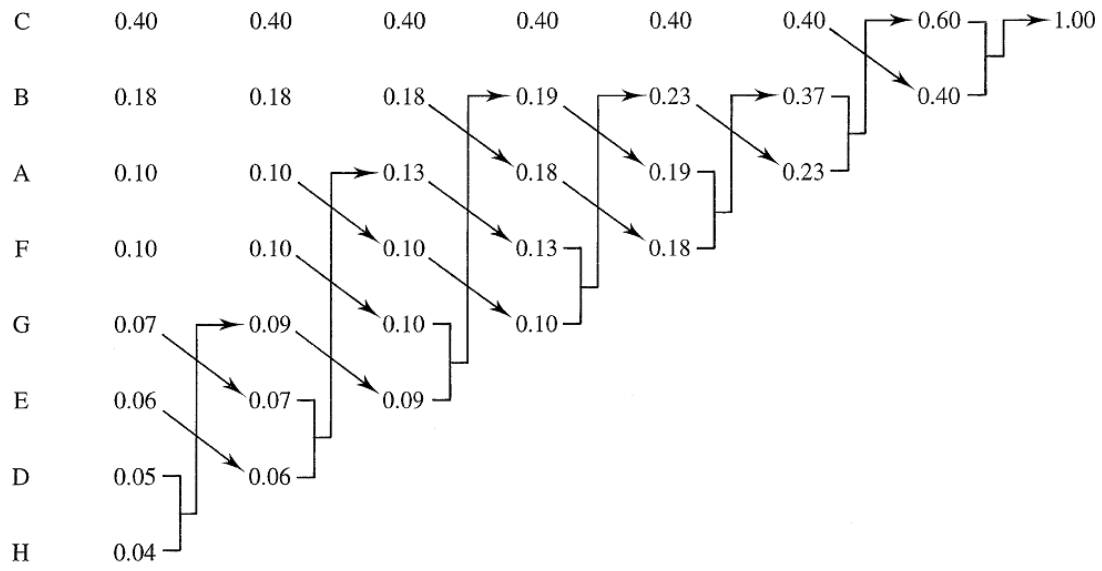

Figure 1.2: Huffman coder reduction process

1.1.2.2 Splitting

The variable length codewords for each transmitted symbol are now derived by working backwards (from the right) through the tree structure created in Figure 1.2, by assigning a 0 to the upper branch of each combining operation and a 1 to the lower branch.

The final "combined symbol" of probability 1.00 is thus split into two parts of probability 0.60 with assigned digit of 0 and another part with probability 0.40 with assigned digit of 1. This latter part with probability 0.40 and assigned digit of 1 actually represents symbol C, Figure 1.3.

The "combined symbol" with probability 0.60 (and allocated first digit of 0) is now split into two further parts with probability 0.37 with an additional or second assigned digit of 0 (i.e. its code is now 00) and another part with the remaining probability 0.23 where the additional assigned digit is 1 and associated code will now be 01.

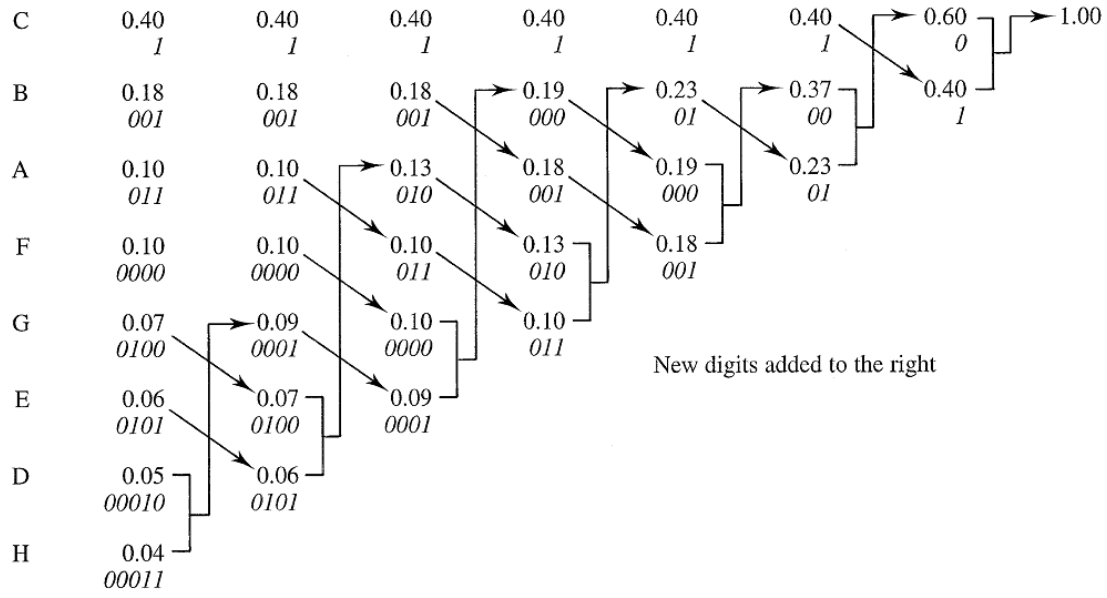

Figure 1.3: Huffman coder splitting process to generate the variable length codewords and allocate these depending on symbol probabilities.

This process is repeated by adding each new digit after the splitting operation to the right of the previous one. Note how this allocates short codes to the more probable symbols and longer codes to the less probable symbols, which are transmitted less often.

Symbol	Code
A	011
B	001
C	1
D	00010
E	0101
F	0000
G	0100
H	00011

Table 1.3: Huffmann coded variable length symbols

1.1.3 Code efficiency

Figure 1.3 summarises the codewords now allocated to each of the transmitted symbols A...H and also calculates the average length of this source coder as 2.61 bits/symbol. Note the considerable reduction from the fixed length of 3 in the simple 3-bit coder in earlier table.

The average length of a codeword for this code is:-

$$L = 0.10\times3 + 0.18\times3 + 0.40\times1 + 0.05\times5 + 0.06\times4 + 0.10\times4 + 0.07\times4 + 0.04\times5$$
$$= 2.61 \text{ binary digits/symbol}$$

Symbol	Code
A	011
B	001
C	1
D	00010
E	0101
F	0000
G	0100
H	00011

Figure 1.4: Summary of allocated codewords for each symbol, A ...H, and calculation of average length of transmitted codeword.

Now recall from Figure 1.1 that the entropy of the source data was 2.5524 bits/symbol and the simple fixed length 3-bit code in the earlier table, with a length of 3.00 which gave an efficiency of only 85.08%.

The efficiency of the Huffman coded data with its variable length codewords is therefore 2.5524/2.62 = 97.7% which is a much more acceptable result.

If the symbol probabilities all have values $1/(2^n)$ which are integer powers of 2 then Huffmann coding will result in 100% efficiency.

NOTE: This module has been created from lecture notes originated by P M Grant and D G M Cruickshank which are published in I A Glover and P M Grant, "Digital Communications", Pearson Education, 2009, ISBN 978-0-273-71830-7. Powerpoint slides plus end of chapter problem examples/solutions are available for instructor use via password access at http://www.see.ed.ac.uk/~pmg/DIGICOMMS/

Chapter 2

Block FECC coding[1]

2.1 Block FECC coding

2.1.1 Forward error correcting coding (FECC)

Block codes are one example of the forward error correcting coding (FECC) technique where we encode the signal by adding additional bits or digits of redundant data so that the decoder is then able to correct most of the errors which are introduced by transmission through a noisy channel. FECC was invented for deep space probes where the extremely long transmission propagation path loss results in received data with particularly low signal to noise ratio as the modest transmitter power is limited by the solar panel outputs.

As we are adding additional bits to generate each codeword this is a systematic encoder as the information data bits are included directly within the codewords. The additional bits required for the transmission of the redundant information increases the data rate which will consume more bandwidth if we wish to maintain the same throughput, but, if we seek to obtain low error rates, then this trade-off is usually acceptable.

[1] This content is available online at <http://cnx.org/content/m18174/1.3/>.

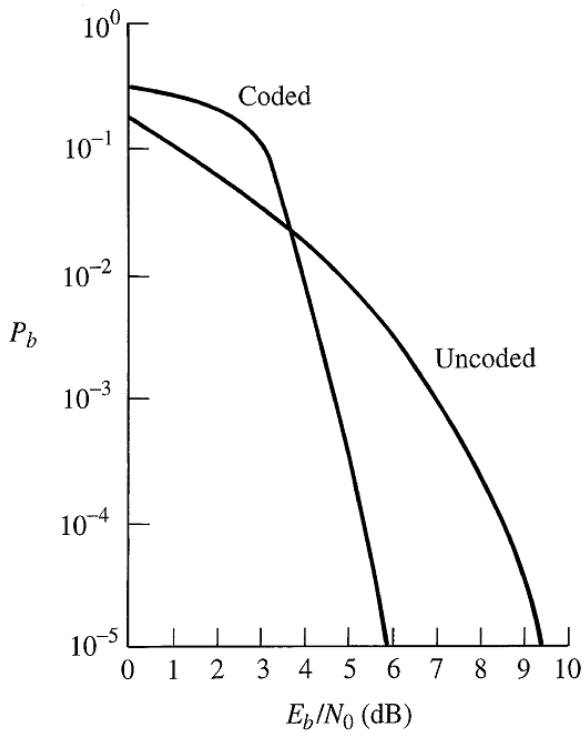

Figure 2.1: Error probability against received noise level for FECC and uncoded data transmissions

Figure 2.1 shows the typical error rate performance for uncoded data compared with FECC data. It plots the bit error probability, P_b , against $\frac{E_b}{N_0}$. $\frac{E_b}{N_0}$ is the measure of the energy per bit to the noise power spectral density ratio and is the normally used signal to noise ratio ratio measure on these error rate plots.

FECC is used widely in compact discs (CD), computer storage and data transmission, all manor of radio links, data modems, video, TV and cellphone transmissions, space communications etc. Note in Figure 2.1 the ability of FECC to achieve a much lower error rate than for the uncoded data transmissions at low bit error probability, P_b .

2.1.2 ASCII coding

In some computer communication systems, information is sent as 7-bit ASCII codes with a parity check bit added on the end. Using even parity the 7-bit all zero ASCII code 0000000 expands into 00000000 while 0000001 codes to 00000011. This (and all other cases) thus has a binary digit difference or Hamming Distance of 2. Figure 2.2 shows that we can use this to detect 1 (or an odd number of) errors.

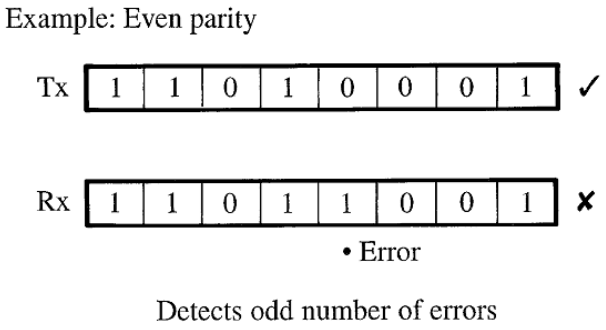

Figure 2.2: ASCII code example where received codeword has single error in the 5th bit position

The block length is then n = 8 and the number of information bits k = 7. This generally assists with error detection but is is insufficiently robust or redundant to achieve an error correction capability as the coding rate is only 7/8.

The minimum distance in binary digits between any two codewords is known as the minimum Hamming Distance, $D_{\min}$, which is 2 for the case of odd or even parity check in ASCII data transmission. We can then calculate the error detecting and correcting power of a code from the minimum distance in bits between error free blocks or codewords, see error correction capability module.

Although we shall look exclusively at coding schemes for binary systems, error correcting and detecting coding is not confined to binary digits. For example the ISBN numbers used on books have a checksum appended to them and these are calculated via modulo 11 arithmetic.

2.1.3 Block code construction

Block codes collect or arrange incoming information carrying data into groups of k binary digits and add coding (i.e. parity) bits to increase the coded block length up to n bits, where n>k.

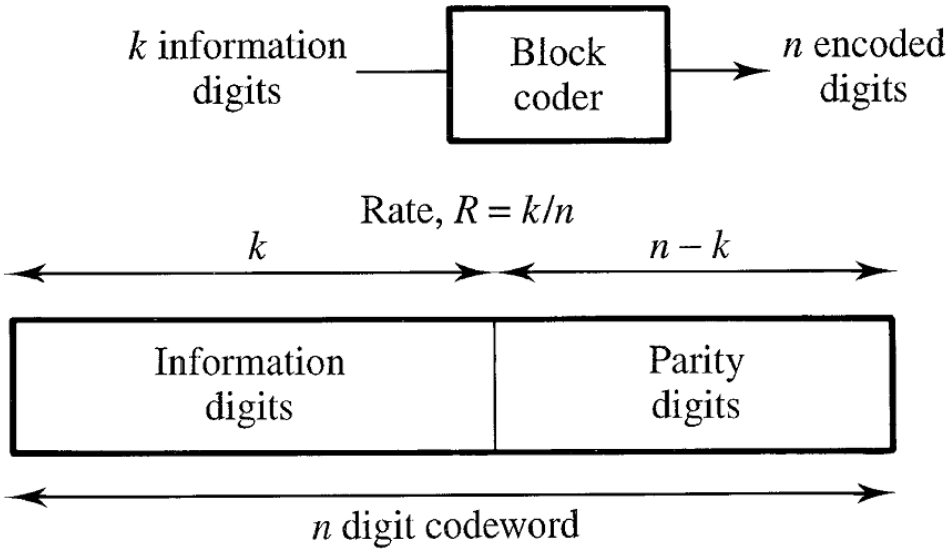

Figure 2.3: Block coder with k information digits and appended parity check bits

The coding rate R is simply the ratio of data or information carrying bits to the overall block length, k/n. The number of parity check (redundant) bits is therefore n – k, . This block code is usually described as a (n, k) code.

2.1.4 Block code example

Suppose we want to code k = 4 information bits into a n = 7 bit codeword, giving a coding rate of 4/7. Code design is performed by using finite field algebra to achieve linear codes. We can achieve this (7, 4) block code using 3 input exclusive or (EX - OR) gates to form the three even parity check bits, P_1, P_2 and P_3 , from the 4 information carrying bits, $I_1 ... I_4$, as shown in Figure 2.4.

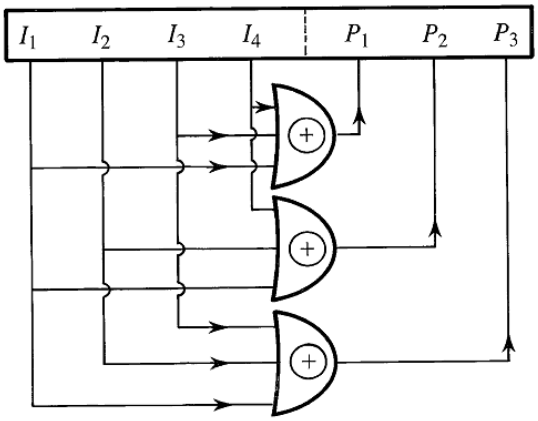

Figure 2.4: Logic gate representation for (n, k) block coder where k = 4 information bits and n = 7 encoded block length (i.e. (7, 4) coder)

This circuitry can be represented by the logic gates in Figure 2.4 or written either as a set of parity check equations or the corrresponding parity check matrix H, as in Figure 2.5.

$$P_1 = 1 \times I_1 \oplus 0 \times I_2 \oplus 1 \times I_3 \oplus 1 \times I_4$$
$$P_2 = 1 \times I_1 \oplus 1 \times I_2 \oplus 0 \times I_3 \oplus 1 \times I_4$$
$$P_3 = 1 \times I_1 \oplus 1 \times I_2 \oplus 1 \times I_3 \oplus 0 \times I_4$$

$$\mathbf{H} = \left[\begin{array}{cccc:ccc} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}\right]$$

Parity check equations — Identity matrix

Figure 2.5: Parity check bit computation and corresponding H matrix representation for (7, 4) block encoder

Remember here that the "cross-in-the-circle" symbol indicates a bitwise exclusive-or ($EX-OR$) operation. This H matrix can also be used to directly generate codewords from the information bits via a closely related G matrix.

This is an example of a systematic code, where the data is included directly within the codeword. Convolutional FECC, see later module, is an example of a non-systematic coder where we do not explicitly include the information carrying data within the transmissions, although the transmitted coded data is derived from the information data.

NOTE: This module has been created from lecture notes originated by P M Grant and D G M Cruickshank which are published in I A Glover and P M Grant, "Digital Communications", Pearson Education, 2009, ISBN 978-0-273-71830-7. Powerpoint slides plus end of chapter problem examples/solutions are available for instructor use via password access at http://www.see.ed.ac.uk/~pmg/DIGICOMMS/

Chapter 3

Block code performance[1]

3.1 Block code error correction capability

3.1.1 Hamming distance

Consider two distinct five digit codewords C1 = 00000 and C2 = 00011. These have a binary digit difference (or Hamming distance) of 2 in the last two digits. The minimum distance in binary digits between any two codewords is known as the minimum Hamming distance, $D_{\min}$

For block codes the minimum Hamming distance or the smallest difference between the digits for any two codewords in the complete code set, $D_{\min}$, is the property which controls the error correction performance. We can thus calculate the error detecting and correcting power of a code from the minimum distance in bits between the codewords.

Thus for a code with a minimum distance $D_{\min} = 3$ then this code can be used to correct:

[1] This content is available online at <http://cnx.org/content/m18175/1.7/>.

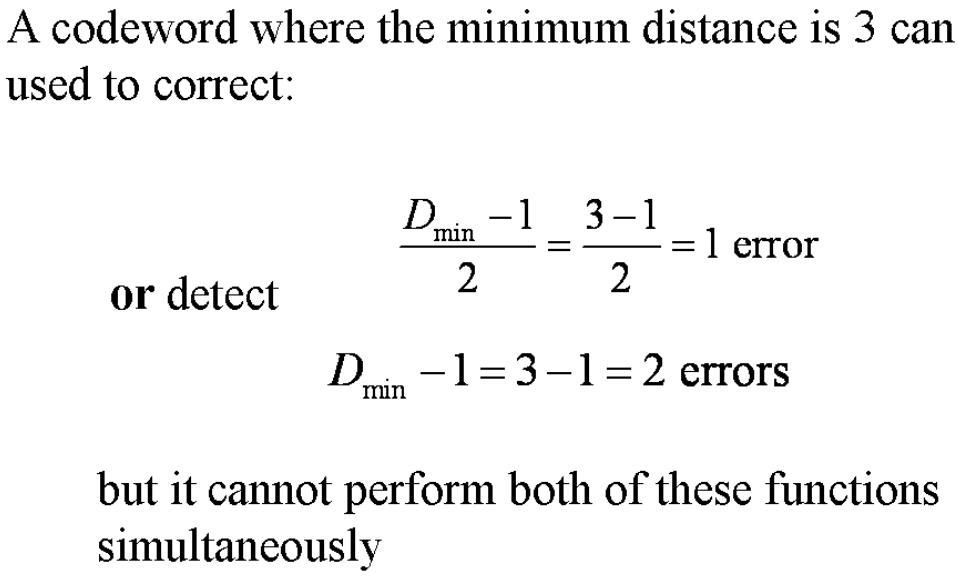

Figure 3.1: Relationship between Dmin and error detection OR correction capability (but not both simultaneously)

Note in the earlier example of two five digit codewords C1 = 00000 and C2 = 00011 which had a Hamming distance of 2 there is only one codeword (e.g. A = 00001 or B = 00010) which lies inbetween these two codewords. Now if there was an error result (e.g. A = 00001) we cannot tell whether it came from C1 or C2 so we can thus only used this to detect that an error has occurred.

If the two five digit codewords had been C1 = 00000 and C2 = 00111, which have a Hamming distance of 3, there are then two words which lie inbetween these codewords (e.g. A = 00001 and B = 00011) and these can thus be used EITHER to detect two errors without any correction capability OR if detection is not required they can used to correct a single error (e.g. C1 = 00000 distorted into A = 00001 or C2 = 00111 distorted into B = 00011), Figure 3.1.

When performing the correction operation we require to insert the decision boundary as shown in Example 1 below. If in this example we had wished to perform detection only as shown in the lower part of Figure 3.1 then we would ignore whether the received code A = 00001 resulted from a single error from a C1 transmission or a double error from a C2 code transmission and only identify it as a detected error.

Example 1 – error correction

C1 = 00000

A = 00001

B = 00011

C2 = 00111

This explains further the detailed operation of the equations in Figure 3.1 where detection only operation does not require the decision boundary to aid identification of the origination of the error.

3.1.2 Block error probability and correction capability

If we have an error correcting code which can correct R' errors, than the probability of a codeword not being correctable is the probability of having more than R' errors in n digits. The probability of having more than R' errors is given in Figure 3.2. We can this calculate this probability by summing all the induvidual error probabilities up to and including R' errors in the block.

More than *R'* errors in *n* binary digits?

For a code which can correct *R'* errors then, the probability of an uncorrectable error, is the probability of having more than *R'* errors in *n* digits as given by:

$$P(>R'\text{errors}) = 1 - \sum_{j=0}^{R'} P(j \text{ errors})$$

Figure 3.2: Correction of more than R' errors in an n digit block

---

The probability of j errors occurring in an n digit codeword is given in Figure 3.3. P_e is the probability of error in a single binary digit and n is the block length. Figure 3.3 also shows how to calculate the nCj term representing all the possible number of ways or error positions that j errors can occur within a block of length n binary digits.

The probability of j errors in n digit codeword is:

$$P(j \text{ errors}) = (P_e)^j (1 - P_e)^{n-j} \times {}^nC_j$$

P_e is the probability of error in a single binary digit and n is the block length. ${}^nC_j$ is the number of ways of choosing j error digit positions within a block of length n binary digits:

$${}^nC_j = \frac{n!}{j!(n-j)!}$$

where ! denotes the factorial operation.

Figure 3.3: Probability of j errors occurring an an n-digit codeword

Example 2: If we have an error correcting code which can correct 3 errors within a block length n of 10, what is the probability that the code cannot correct a received block if the per digit error probability is P_e = 0.01?

Solution: The code cannot correct the received block if there are more than 3 errors. Thus:

P > 3 errors = 1 - P(0 errors) - P(1 error) - P(2 errors) - P(3 errors).

Figure 3.4 shows the component parts of this calculation.

$$P(0\text{ errors}) = 0.01^{0}0.99^{10}\,\frac{10!}{0!10!} = 0.9043821$$

$$P(1\text{ error}) = 0.01^{1}0.99^{9}\,\frac{10!}{1!9!} = 0.0913517$$

$$P(2\text{ errors}) = 0.01^{2}0.99^{8}\,\frac{10!}{2!8!} = 0.0041523$$

$$P(3\text{ errors}) = 0.01^{3}0.99^{7}\,\frac{10!}{3!7!} = 0.0001118$$

Figure 3.4: Calculation of zero, 1, 2 and 3 errors in a 10-digit codeword with a per-digit P_e of 0.01

Thus the probability that the code cannot correct a received block is then:
1 - 0.9043821 - 0.0913517 - 0.0041523 - 0.0001118 = 0.0000021.

This illustrates that the very low overall error remaining after correction of three errors is much less than the original probability of error in a single bit, $P_e = 0.01$. Note also the need for high precision arithmetic (it may be an eight digit calculator is not good enough to calculate the answer to more than 1 significant figure). Note also in Figure 3.4 the much lower probability of t + 1 errors occuring, compared to t errors, as is implied in FECC.

3.1.3 Group codes

Group codes are a special kind of block codes. They comprise a set of codewords, C1 ... CN, which contain the all zeros codeword (e.g. 00000) and exhibit a special property called closure. This property means that if any two valid codewords are subject to a bit wise EX – OR operation then they will produce another valid codeword in the set.

The closure property means that to find the minimum Hamming distance, see below, all that is required is to compare all the remaining codewords in the set with the all zeros codeword instead of comparing all the possible pairs of codewords.

The saving gets bigger the longer the codeword. For example a code set with 100 codewords will require 100 comparisons for a Group code design, compared with 100+99+98+...+2+1, for a non-group code!

In Group codes the $D_{\min}$ calculation is further simplified into calculating the minimum codeword weight or minimum number of 1 digits in a codeword in the set.

3.1.4 Nearest neighbour decoding

Nearest neighbour decoding assumes that the codeword nearest in Hamming distance to the received word is what was transmitted, as shown in Example 1 above. This inherently contains the assumption that the probability of a small number of t errors is greater than the probability of the larger number of t+1 errors, i.e that P_e is small.

A nearest neighbour decoding table for a (n, k) = (5, 2) i.e. a 5-digit group code is shown in Figure 3.5. Recall that for an n = 5 bit codeword there are $2^5 = 32$ unique patterns generated by all the possible combinations of the 5 digits.

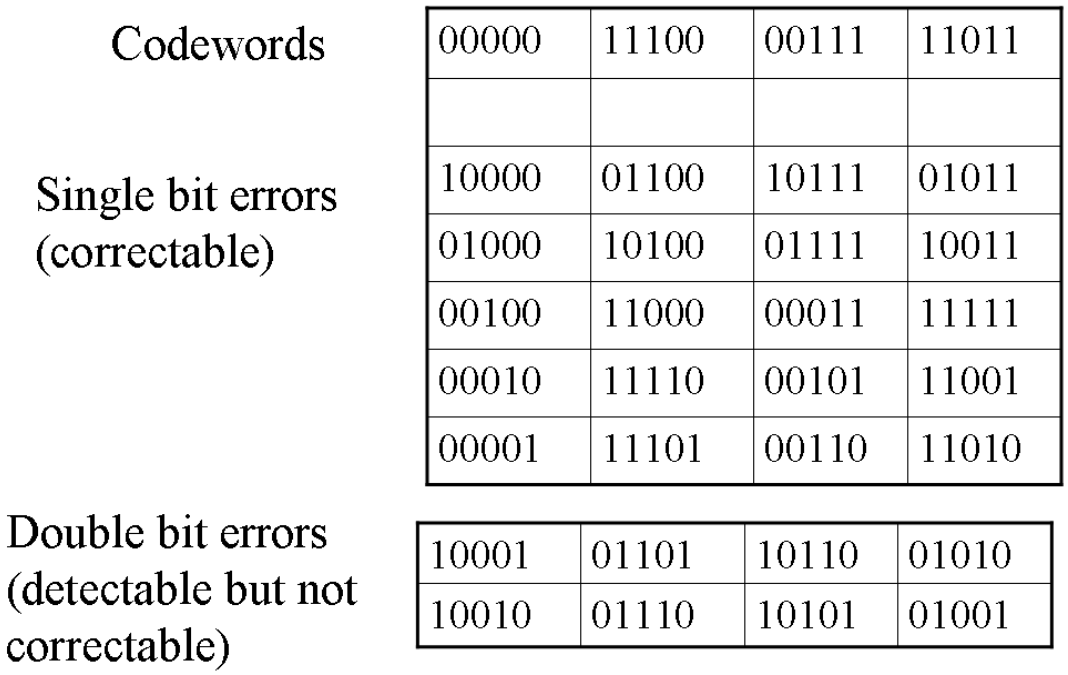

Figure 3.5: Nearest neighbour decoding table for 5 bit code with 4 codewords implying 2 information bits

Figure 3.5 starts by forming a table with the 4 codewords across the top row. All the single error patterns, which each only differ by one bit from each of the transmitted codewords, can be readily and uniquely assigned back to an error free codeword. Thus the next 5 rows represent these single errors in position 1 through 5 in each of the 4 codewords. Now we have a table up to this point with a total of 4 x 6 = 24 unique entries. Therefore this code is capable of correcting all these single errors.

There are also eight remaining codes or table entries as 32 - 24 = 8 and these represent double error patterns which, as can be seen, lie an equal Hamming distance from at least 2 of the initial 4 codewords in the top row. Note for example errors in the first two digits of the 00000 codeword result in us receiving 11000. However data bit pattern is identified here in Figure 3.5 as a single error from codeword 11100 as we assume that 1 error is a much more likely occurence than two errors!

These represent some of the double error patterns, which can thus be detected here, but they cannnot be corrected as all the possible double error patterns do not have a unique representation in Figure 3.5.

3.1.5 Soft decision decoding

Nearest neighbour decoding can also be done on a soft decision basis, with real non-binary numbers from the receiver. The nearest Euclidean distance (nearest to these 5 codewords in terms of a 5-D geometry) is then used and this gives a considerable performance increase over the hard decision decoding described here.

3.1.6 Hamming bound

This defines mathematically the error correcting performance of a block coder. The upper bound on the performance of block codes is given by the Hamming bound, some times called the sphere packing bound. If we are trying to create a code to correct t errors with a block length of n with k information digits, then Figure 3.6 shows the Hamming bound equation.

The upper bound on the performance of block codes is given by the **Hamming Bound**,

$$2^k \leq \frac{2^n}{1 + n + {}^nC_2 + {}^nC_3 + \ldots + {}^nC_t}$$

nCj is the number of ways of choosing j error positions within a block of length n binary digits:

$${}^nC_j = \frac{n!}{j!(n-j)!}$$

Figure 3.6: Hamming bound calculation for (n, k) block code to establish number of terms which can be included in the denominator and hence arrive at the codes error correcting power t

Here the denominator terms, which are represented by the binomial coefficients, represent the number of possible patterns or positions in which 1, 2, ..., t errors can occur in an n-bit codeword.

Note the relationship between the decoding table in Figure 3.5 and the Hamming Bound equation in Figure 3.6. The $2^k = 4$ left hand entry represents the number of transmitted codewords or columns in the table. The numerator $2^n = 32$ represents the total possible number of unique entries in the table. The demoninator represents the number of rows which can be accommodated within the table. Here the first denominator term (1) represents the first row (i.e. the transmitted codewords) and the second term (n) the 5 single error patterns. Subsequent terms then represent all the possible double, triple error patterns, etc. The denominator has to be sized or restricted to t to ensure the inequality and this gives or defines the error correction capability as t.

If the equation in Figure 3.6 is satisfied then the design of suck an (n, k) code is possible with the error correcting power of t. If the equation is not satisfied, then we must be less ambitious by reducing t or k (for the same block length n) or increasing n (while maintaining t and k).

Example 2

Comment on the ability of a (5, 2) code to correct 1 error and the possibility of a (5, 2) code to correct 2 errors?

Solution

For single error: k = 2, n = 5 and t = 1, leads to the case summarized in Figure 3.7.

Can a (5,2) code correct a single error?

$k = 2, n = 5$ and $t = 1$, leads to:

$$2^2 \leq \frac{2^5}{1+5} \text{ or } 4 \leq \frac{32}{6} \text{ or } 4 \leq 5.333$$

Figure 3.7: Calculation to assess whether (5, 2) block code can correct t = 1 error - Answer yes

which is true so such a code design is possible.

However if we try to design a (5, 2) code to correct 2 errors we have k = 2, n = 5 and t = 2, which is summarized in Figure 3.8.

Attempt to design a (5,2) code which corrects 2 errors
we have k=2, n=5 and t=2 and in the Hamming Bound:

$$2^2 \leq \frac{2^5}{1+5+10} \text{ or } 4 \leq \frac{32}{16} \text{ or } 4 \leq 2$$

which is false so such a code cannot be created.

Figure 3.8: Calculation to assess whether (5, 2) block code can correct t = 2 errors - Answer no

This result is false or cannot be satisfied and thus this short code cannot be designed with a t = 2 error correcting power or capability.

This provides further mathematical derivation for the error correcting performance limits of the nearest neighbour decoding table shown previously in Figure 3.5 where we could correct all single error patterns but we could not correct all the possible double error patterns.

A full decoding table is not required to be created as, through checking the Hamming bound, one can identify the required block size and number of parity check bits which are required for a given error correction capability in a block or group coder design.

Figure 3.9 shows the performance of various BLOCK codes, all of rate $\frac{1}{2}$, whose performance progressively improves as the block length increases from 7 to 511, even for the same coding rate of $\frac{1}{2}$.

The power of these forward error correcting codes (FECC) is quantified as the coding gain, i.e. the reduction in the required $\frac{E_b}{N_0}$ ratio or energy required to transmit each bit divided by the spectral noise density, for a given bit error ratio or error probability.

For example in Figure 3.9 the (31, 16) code has a coding gain over the uncoded case of around 1.8 dB at a P_b of 10^{-5} .

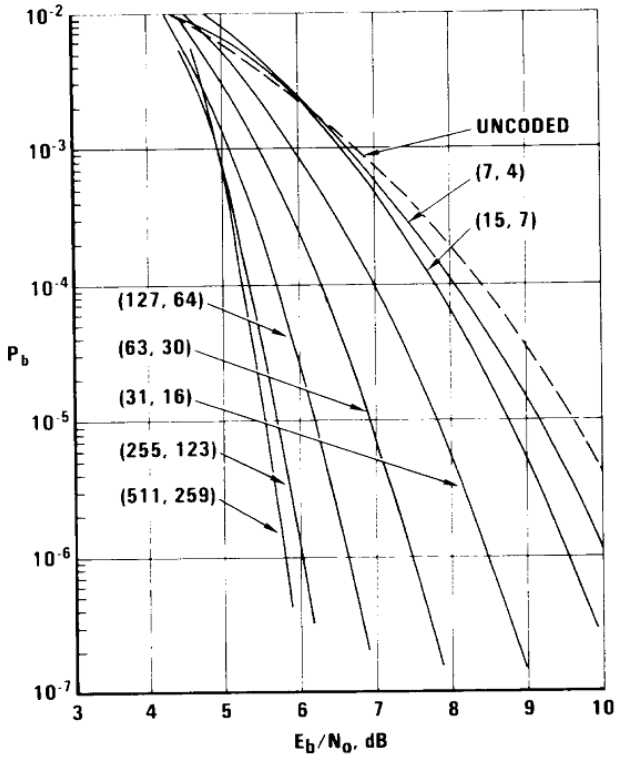

Figure 3.9: Error performance of 1/2 rate block coders with differing block lengths

NOTE: This module has been created from lecture notes originated by P M Grant and D G M Cruickshank which are published in I A Glover and P M Grant, "Digital Communications", Pearson Education, 2009, ISBN 978-0-273-71830-7. Powerpoint slides plus end of chapter problem examples/solutions are available for instructor use via password access at http://www.see.ed.ac.uk/~pmg/DIGICOMMS/

Chapter 4

Convolutional FECC Encoder[1]

4.1 FECC – $\frac{1}{2}$ Rate Convolutional Encoder Example

4.1.1 Convolutional coding

Convolutional codes are another type of forward error correcting coder (FECC) which are quite distinct from block codes. They are simpler to implement for longer codes than block coders and soft decision decoding can be employed easily at the decoder.

Convolutional codes are non-systematic (i.e. the transmitted data bits do not appear directly in the output encoded data stream) and are generated by passing a data sequence through a transversal or finite impulse response (FIR) filter. The coder output can be regarded as the convolution of the input sequence with the impulse response of the coder, hence their name: convolutional codes.

4.1.2 Convolutional encoder

A simple example is shown in Figure 4.1. Here the encoder shift register starts with zeros at all three stored locations (i.e. 0, 0, 0). The input data sequence to be encoded is 1, 1, 0, 1 in this example. The shift register contents thus become, after each data bit arrives and propagates into the shift register: 100, 110, 011, 101. As there are two outputs for overy input bit the above encoder is rate $\frac{1}{2}$.

The first output is obtained after arrival of a new data bit into the shift register when the switch is in the upper position, the second with the switch in the lower position. Thus, in this example, the switch will generate, through the exclusive OR gates, from the four input data bits: 1, 1, 0, 1, the corresponding four output digit pairs: 11, 10, 11, 01

[1] This content is available online at <http://cnx.org/content/m18176/1.3/>.

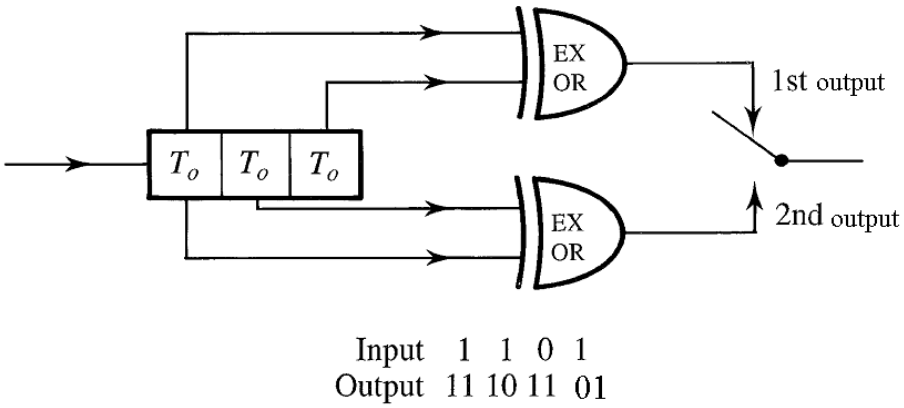

Figure 4.1: $\frac{1}{2}$ rate convolutional encoder

This particular encoder has 3 stages in "the filter" and therefore we say that the constraint length n = 3. The very latest encoders available commercially typically have constraint lengths up to n = 9.

We can consider the coder outputs from the exclusive OR gates as being generated by two polynomials:

$$P_1(\mathrm{x}) = 1 + x^2 \tag{4.1}$$

$$P_2(\mathrm{x}) = 1 + x \tag{4.2}$$

These are often expressed in octal notation, in our example:

$$P_1 = 5_o(101) \tag{4.3}$$

$$P_2 = 6_o(110) \tag{4.4}$$

This encoder may also be regarded as a state machine. The next state is determined by the next input bit or value combined with the previous two input bits or values which were stored in the shift register, (i.e. the previous state).

4.1.3 Tree state diagram

We can regard this as a Mealy state machine with four states corresponding to all the possible combinations of the first two stages in the shift register.

The tree diagram for this state machine is now shown in Figure 4.2, again starting from the all zeros state or condition. The encoder starts in state A holding two zeros (00) within the first two stages of the shift register. (We ignore the final stored digit as it is lost when a new data bit propagates into the shift register.) If the next input bit is a zero (0) we follow the upper path to state B where the stored data is updated to 00. If the next input bit is a one (1) we follow the lower path to progress to the corresponding state C where the stored data is now 10.

The convention is to enter the updated new stored state values below the state letter (B/C). Now returning to Figure 4.1 and the exclusive OR gate connections one can derive the output data bits generated within the encoder. For state B these are 00 and for state C these are 11. These outputs are entered alongside the state in Figure 4.2. States B/C correspond to the arrival of the first new data bit to be encoded, while D/E/F/G correspond to the second data bit and H/I/J/K/h/i/j/k the third data bit.

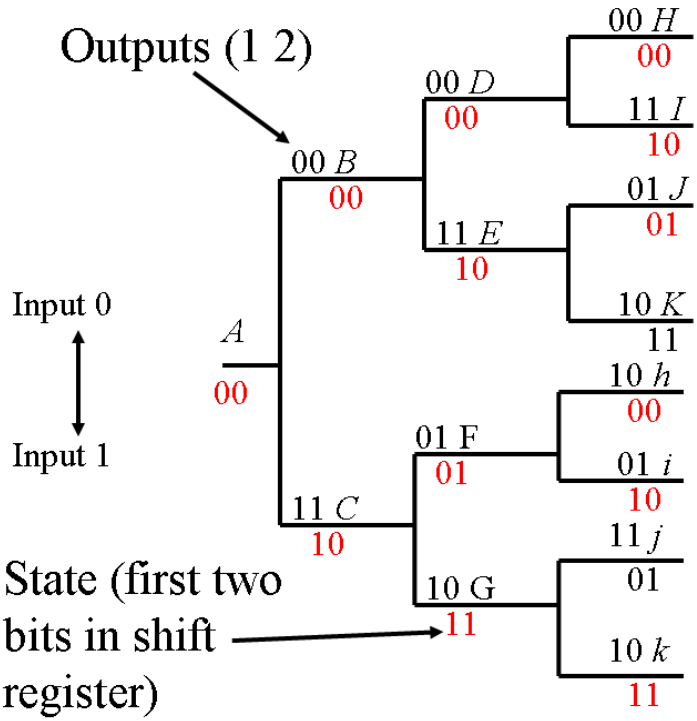

Figure 4.2: Encoded data tree diagram for the encoder of Figure 1

The tree diagram in Figure 4.2 tends to suggest that there are eight states in the last layer of the tree and that this will continue to grow. However some states in the last layer (i.e. the stored data in the encoder) are equivalent as indicated by the same letter on the tree (for example H and h).

These pairs of states may be assumed to be equivalent because they have the same internal state for the

first two stages of the shift register and therefore will behave exactly the same way to the receipt of a new (0 or 1) input data bit.

4.1.4 Trellis state diagram

Thus the tree can be folded into a trellis, as shown in , which is derived from the tree diagram of Figure 4.2 and Figure 4.1 encoder. As the constraint length is n = 3 we have $2^{(3-1)} = 4$ unique states: 00, 01, 10, 11 in Figure 4.2. In Figure 4.3 the states are shown as 00x to denote the third bit, x, which is lost or discarded following the arrival of a new data bit.

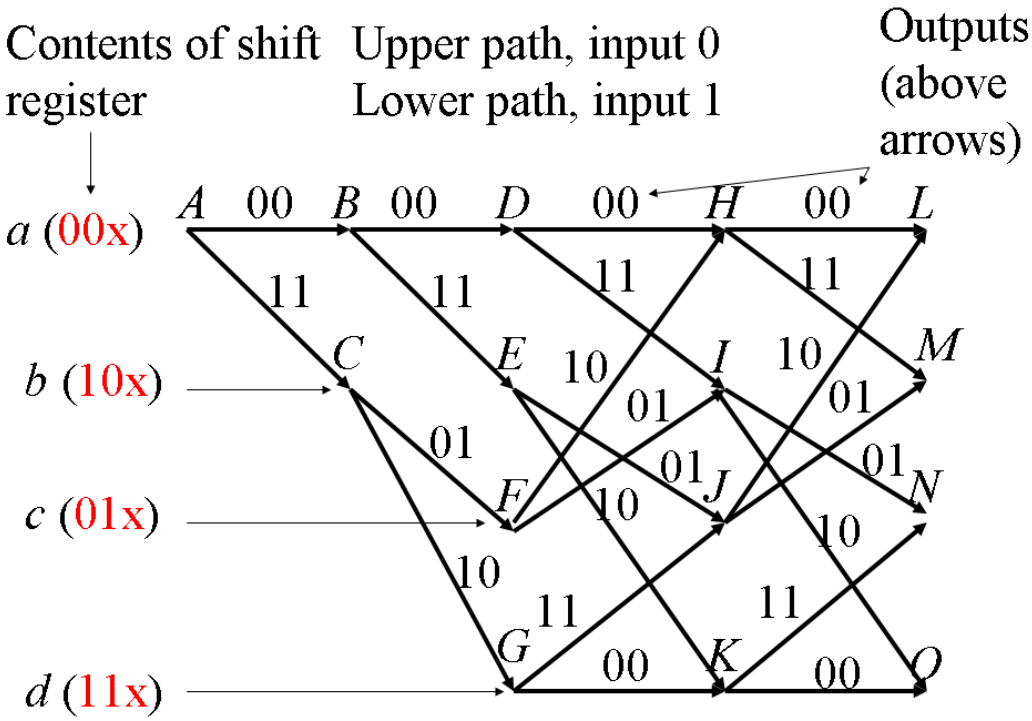

Figure 4.3: Trellis Diagram corresponding to the Tree Diagram of Figure 2

Note in Figure 4.3 the horozontal arrangement of states A, B, D, H and L. The same applies to states C, E, I and M etc. The horizontal direction corresponds to time (the whole diagram in Figure 4.3 now corresponds to encoding 4 input data bits). Here we have dropped the state information from Figure 4.2 as the same states are all represented at the same horizontal level in Figure 4.3. The vertical direction here corresponds to the stored state values a, b, c, d in the encoder shift register.

States along the time axis are thus equivalent, for example H is equivalent to L and C is equivalent to E etc. In fact all the states in a horizontal line are equivalent. Thus we can identify only four states in this coder: a, b, c and d and the related shift register stored values 00, 10, 01, 11 are shown in the left hand side of Figure 4.3.

From any point, e.g. E, if the next input bit is a zero (0) we follow the upper path to state J where the stored data is updated to 01 and the output will be 01. If the next input bit is a one (1) we follow the lower path from E to progress to the next state K where the stored data is now 11 and the output will be 10 as indicated alongside the trellis path.

4.1.5 Transition state diagram

We can draw, if desired, the trellis diagram of Figure 4.3 in Figure 4.4 as a state diagram containing only these states with all the corresponding new data bits to be encoded and the corresponding two output bits generated per new input data bit (e.g. 1(10))

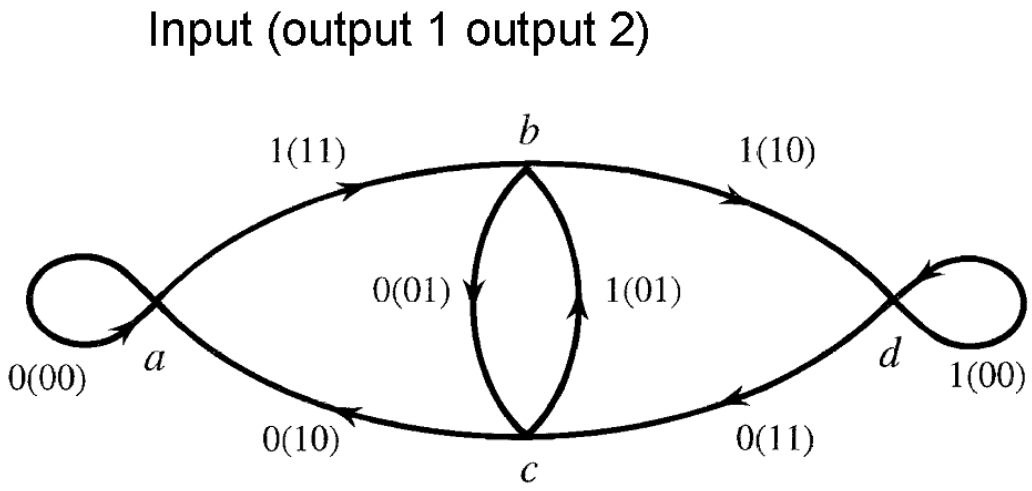

Figure 4.4: State diagram corresponding to the encoder trellis diagram of Figure 3

NOTE: This module has been created from lecture notes originated by P M Grant and D G M Cruickshank which are published in I A Glover and P M Grant, "Digital Communications", Pearson Education, 2009, ISBN 978-0-273-71830-7. Powerpoint slides plus end of chapter problem examples/solutions are available for instructor use via password access at http://www.see.ed.ac.uk/∼pmg/DIGICOMMS/

Chapter 5

Viterbi Decoder[1]

5.1 Viterbi convolutional decoder

A convolutional code is not decoded in short blocks as in a block code. However, to simplify decoding, messages are artificially broken down into very long blocks by periodically flushing the encoder with a string of zeros, as in the example discussed here.

For illustration only this example here uses an unrealistically short block length of 5 data bits with the last two fixed at 0 to flush the encoder (remember that this is very inefficient and, in practice, practical block lengths are very much longer, typically 1,000 to 10,000 bits in length).

Convolutional codes are always decoded using the Viterbi algorithm as this simplifies the decoding operation. The algorithm is based on the nearest neighbour decoding scheme and, like the other algorithms we have looked at, it relies on the assumption that the probability of t errors is much greater than the probability of t+1 errors and it thus selects or chooses and retains only the paths which have fewer errors.

The decoding process is based on the previous decoding trellis. We will use the previous $\frac{1}{2}$ rate encoder example and assume that the received message is: 10 10 00 10 10, representing a total of five (unknown) transmitted data bits each encoded into five bit pairs, i.e. total of ten encoded data bits. We further assume in this simplified example that the last 2 bits of the 5 data inputs were flushing zeros to reset the encoder and decoder.

Starting (after flushing) with the first received bit in position A in the encoder, we know that if a 1 had been input, (lower path) from the encoder figure the output should have been 11 as we moved to state C. If a 0 was input (upper path) we should have received 00 and moved to state B, see upper part of Figure 5.1.

What was actually received was 10, a Hamming distance of 1 from both these possibilities, so we draw that in the lower part of Figure 5.1 onto the first stage of our decoding trellis.

[1] This content is available online at <http://cnx.org/content/m18177/1.3/>.

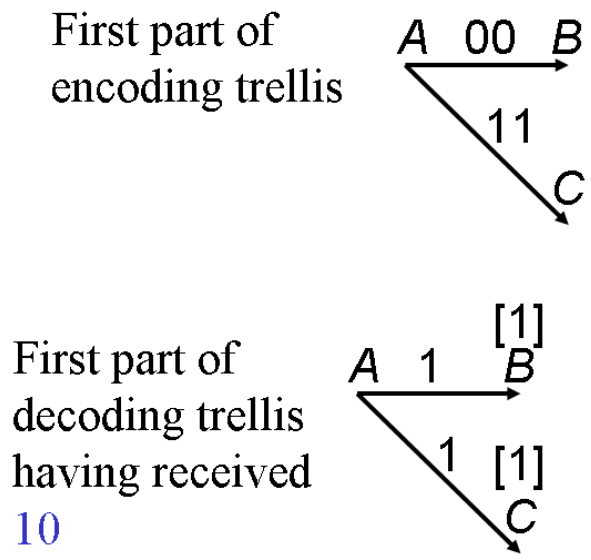

Figure 5.1: First stage of trellis after decoding first two received data bits

Instead of reporting the expected outputs we next annotate the lower part of Figure 5.1 with the separate distances between the received data and the trellis encoder on each path. We then add the cumulative Hamming distance to the states (B, C) in square brackets above the states B and C

Now consider the second pair of received data bits. Consider first state B. As before, we should have received 00 for a 0 input and 11 for a 1 input, see left hand side of Figure 5.2. What we actually received was 10, which is a Hamming distance of 1 from both possibilities so the right hand part of Figure 5.2 is annotated with the individual and cumulative distances to states D and E.

Then consider state C. For a 0 input, (upper part) we should have received 01, but what was actually received was 10, a Hamming distance of 2. For a 1 input (lower path) we should have received 10 and this is exactly what was received, corresponding to a Hamming distance of 0! Again the right part of Figure 5.2 is annotated with the individual distances on the paths and the new cumulative or summed distances to states F and G.

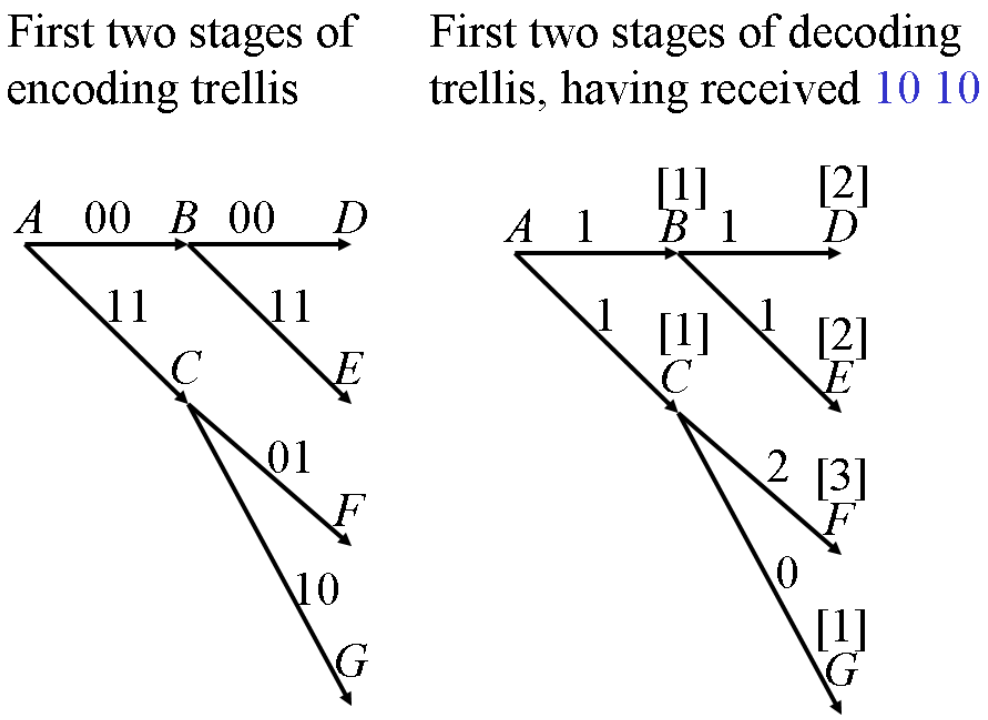

Figure 5.2: First and second stage of the decoding trellis after receiving second pair of data bits

We continue to build our decoding trellis until it is complete after receipt of all ten data bits, as shown in Figure 5.3.

If we have two paths to a state, as in the later states: H, I , J, K, L, M, N, P, we write the smaller (more likely) Hamming distance in square brackets above the state and discard the larger distance (as this is much less likely to represent the correct path). In our example, we assumed the last two bits were 0, so we must expect to finish back in state P, which is the same as the starting state A.

We finally need to find the path from state A to P which gives the lowest overall Hamming distance. We then retrace the path and remember that the upper path from a state represented a 0 transmitted and the lower path represented a 1 transmitted.

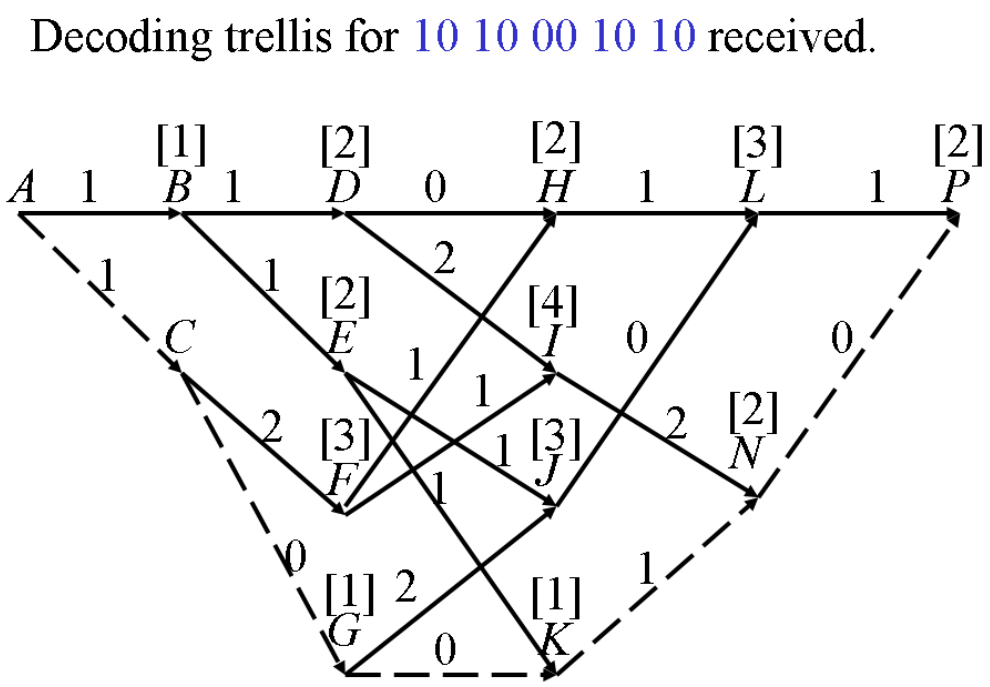

Figure 5.3: Full decoding trellis after receipt of all ten data bits

The reverse decoded data for this example is indicated by the dashed line in Figure 5.3.

Leaving states A, C, G and K always in the lower of the two possible paths implies that a data bit 1 has been received at these states and therefore this translates to 1, 1, 1 as the first three encoded data bits.

The last two bits don't matter in this case as we have assumed they are 0, 0 and we can remove from the docoding trellis all the states that don't support or contribute to this solution.

Note that finishing a block with n-1 zero input data bits is not compulsory. If you make a decision after a delay of approximately five times the constraint length n, this makes little difference in code performance but does limit the memory consumed by the process to a more sensible amount.

Figure 5.4 shows the performance of various BLOCK codes, all of rate $\frac{1}{2}$, whose performance improves as the block length increases, even for the same coding rate of $\frac{1}{2}$.

The power of these forward error correcting codes (FECC) is quantified as the coding gain, i.e. the reduction in the required $\frac{E_b}{N_0}$ ratio or energy required to transmit each bit divided by the spectral noise density, for a given bit error ratio or error probability.

For example in Figure 5.4 the (31, 16) code has a coding gain over the uncoded case of around 1.8 dB at a P_b of 10^{-5} .

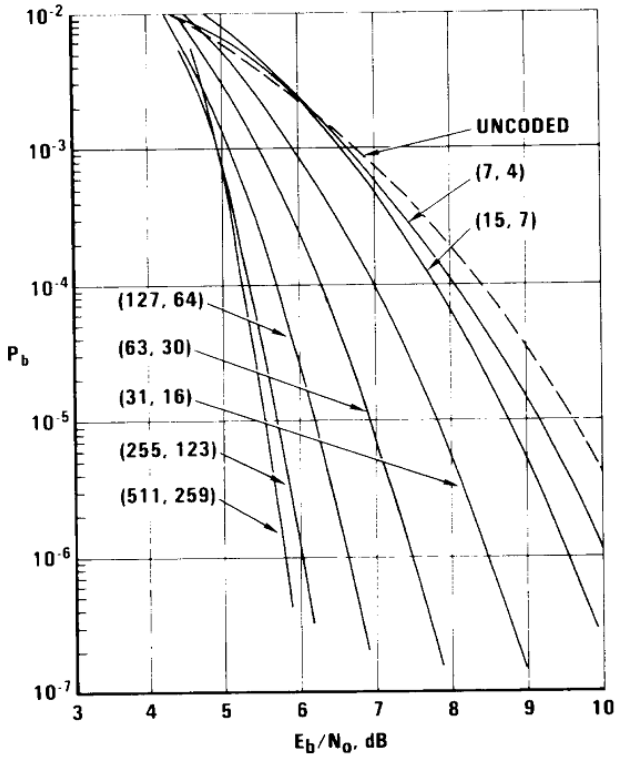

Figure 5.4: Error performance of 1/2 rate block coders with differing block lengths

Figure 5.5 shows for comparison with the block codes of Figure 5.4 the performance of convolutional coders. The convolutional code initially provides very good performance at modest constraint length. A short constraint length of n = v = 3 is already superior to the 511 block length code of Figure 5.4. The additional attraction of the convolutional coder is its further improvement with the increase in constraint length up to n = 7 or 9, as shown in Figure 5.5.

Unfortunately the coding and decoding process gets more complicated with larger block/constraint length. As shown here convolutional codes with Viterbi decoding are generally more powerful than block codes, especially for very low error rates, hence their wider use. Single chip constraint length 9 (512 state) encoder and decoders are now widely available as commercial products from many semiconductor vendors.

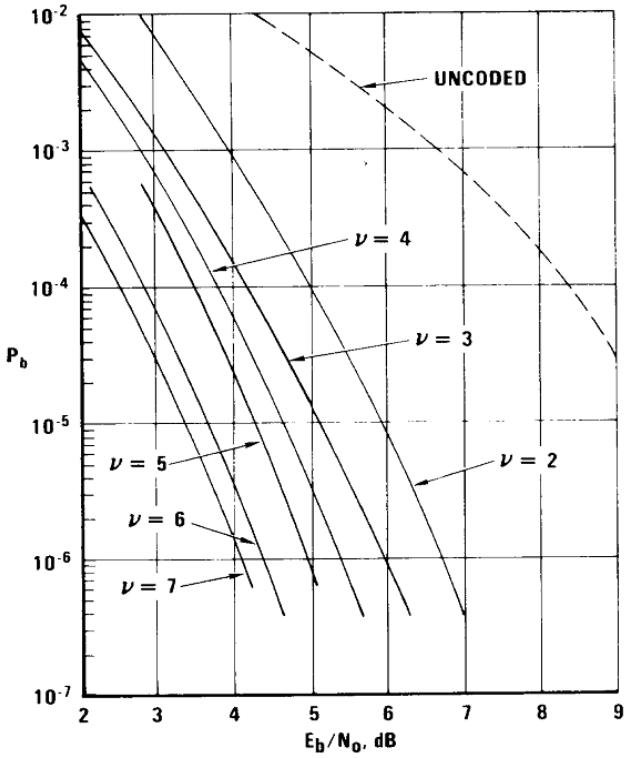

Figure 5.5: Error rate performance of convolutional decoders with differing constraint lengths

WARNING: This module has been created from lecture notes originated by P M Grant and D G M Cruickshank which are published in I A Glover and P M Grant, "Digital Communications", Pearson Education, 2009, ISBN 978-0-273-71830-7. Powerpoint slides plus end of chapter problem examples/solutions are available for instructor use via password access at http://www.see.ed.ac.uk/~pmg/DIGICOMMS/

Chapter 6

Turbo Coding[1]

6.1 Turbo encoding and decoding

6.1.1 Introduction

A paper was published by Claude Berrou and coauthors at the ICC conference in 1993 that rocked or shook the field of forward error correction coding (FECC). This described a method of creating much more powerful block error correcting coding with only the minimum amount of effort. Its main features were two recursive convolutional encoders (RCE) interconnected via an interleaver. The data is fed into the first encoder directly and into the second encoder after interleaving or reordereing of the input data.

6.1.2 Turbo encoding

The important features are the use of two recursive convolutional encoders and the design of the interleaver which gives a block code with the block size equal to the interleaver size, Figure 6.1. Random interleavers tend to work better than row and column interleavers. Note that recursive convolutional encoders were known about well before their use in turbo codes, but the difficulties in driving them into a known state made them less popular than the non-recursive convolutional encoders described in the previous module.

The name turbo decoder came from the turbo charger in an automobile where the exhaust gasses are used to drive a compressor in a feedback loop to increase the input of fuel and hence the vehicles ultimate performance.

[1] This content is available online at <http://cnx.org/content/m18178/1.3/>.

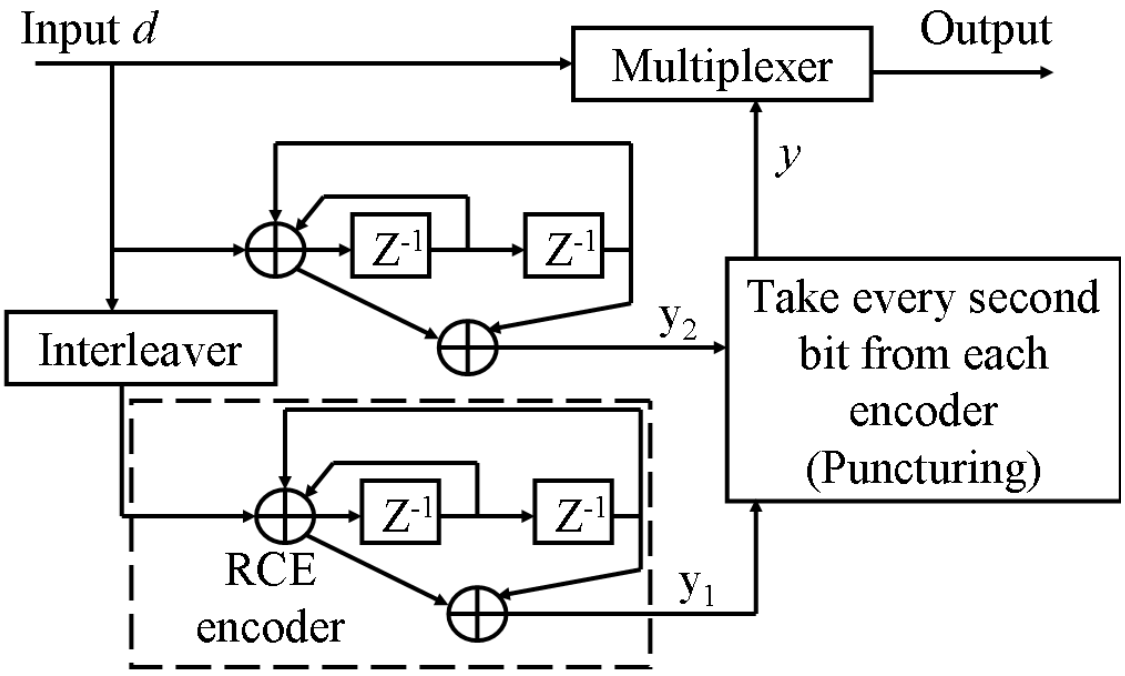

Figure 6.1: Turbo encoder with recursive encoding loops

The desired output rate was initially achieved by puncturing (ignoring every second output) from each of the encoders.

6.1.3 Turbo decoding

Turbo decoding is iterative. The decoding is also soft, the values that flow around the whole decoder are real values and not binary representations (with the exception of the hard decisions taken at the end of the number of iterations you are prepared to perform). They are usually log likelihood ratios (LLRs), the log of the probability that a particular bit was a logic 1 divided by the probability the same bit was a logic 0.

Decoding is accomplished by first demultiplexing the incoming data stream into d, y_1 , y_2. d and y_1 go into the decoder for the first code, Figure 6.2. This gives an estimate of the extrinsic information from the first decoder which is interleaved and past on to the second decoder. The second decoder thus has three inputs, the extrinsic information from the first decoder, the interleaved data d, and the received values for y_2. It produces its extrinsic information and this is deinterleaved and passed back to the first encoder. This process is then repeated or iterated as required until the final solution is obtained from the second decoder interleaver.

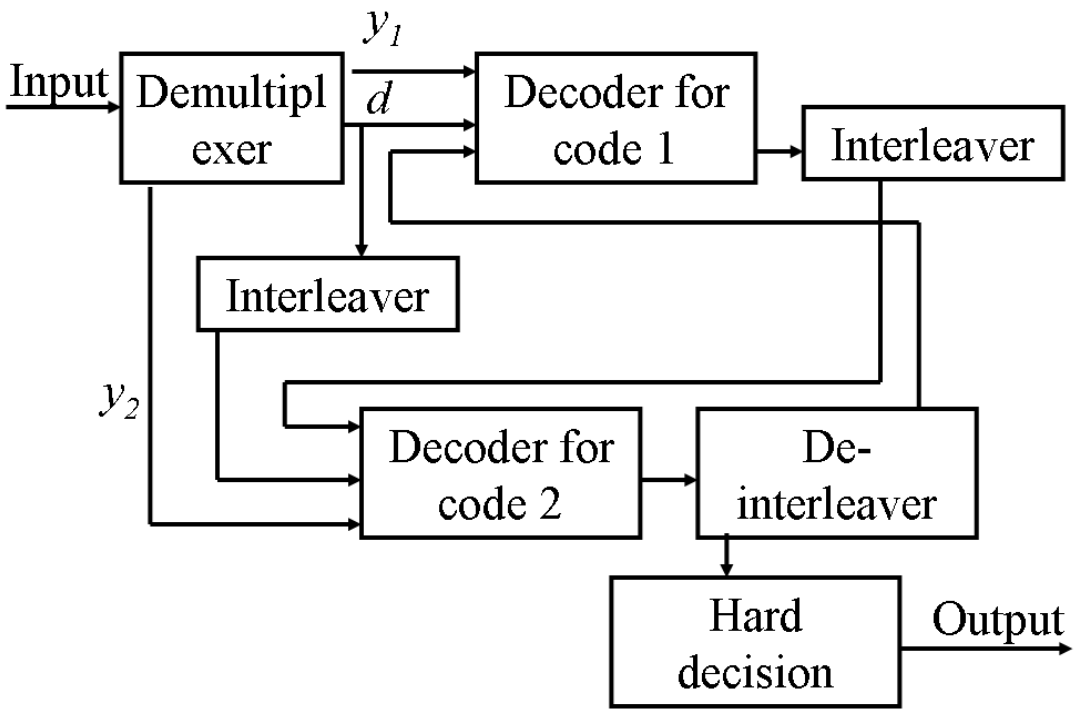

Figure 6.2: Turbo decoder

The decoders themselves generally use soft output Viterbi algorithm (SOVA) to decode the received data. However the preferred turbo decoding method is to use the maximum a-priori (MAP) algorithm but this is too mathematical to discuss here!

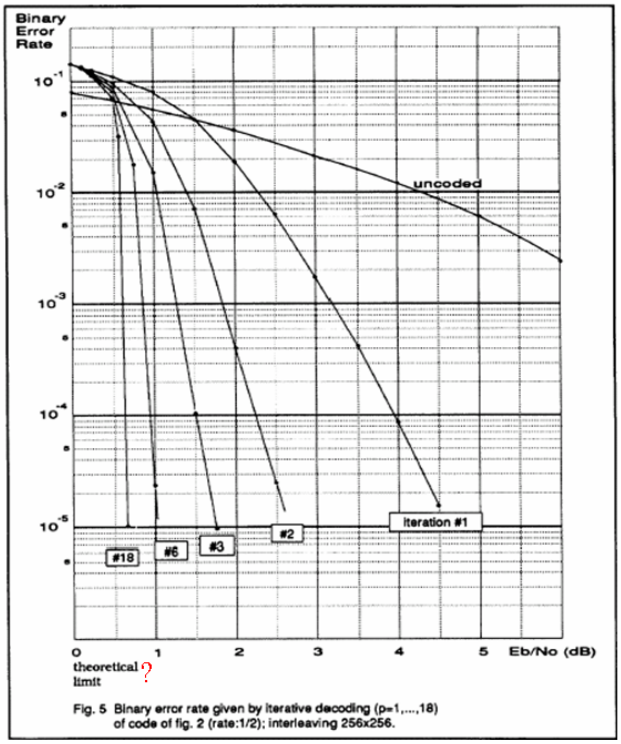

Figure 6.3: Probability of error for turbo decoders with variable number of iterations

6.1.4 Coder performance

Figure 6.3 shows these $\frac{1}{2}$ rate decoders operating at much lower $\frac{E_b}{N_0}$ or SNR values than the convolutional Viterbi decoders of the previous section and, further, as the number of iterations increases to beyond 15, then the performance comes very very close to the theoretical Shannon bound.

This is the attraction that has excited the FECC community, who were unable to achieve this low error rate before 1993! Now that iterative decoding has been introduced for turbo decoders it is also being re-applied in low delay parity check (LDPC) decoders with equal enthusiasm and success.

Turbo Code Example

Figure 6.4

Figure 6.4 includes a turbo decoding example (which as an animated power point slide) will show the black dot noise induced errors being corrected on each subsequent iteration with the black dots being progressively reduced in the upper cartoon.

NOTE: This module has been created from lecture notes originated by P M Grant and D G M Cruickshank which are published in I A Glover and P M Grant, "Digital Communications", Pearson Education, 2009, ISBN 978-0-273-71830-7. Powerpoint slides plus end of chapter problem examples/solutions are available for instructor use via password access at http://www.see.ed.ac.uk/∼pmg/DIGICOMMS/

Index of Keywords and Terms

Keywords are listed by the section with that keyword (page numbers are in parentheses). Keywords do not necessarily appear in the text of the page. They are merely associated with that section. *Ex.* apples, § 1.1 (1) **Terms** are referenced by the page they appear on. *Ex.* apples, 1

B Block codes, § 3(13)
Block coding, § 2(7)

C Convolutional code, § 5(31)
Convolutional coder, § 4(25)

F FECC, § 5(31), § 6(37)
Forward Error Correcting Coder, § 4(25)

H Hamming bound, § 3(13)
huffman coder, § 1(1)

N Nearest neighbour decoding, § 3(13)

P Parity check matrix, § 2(7)

S source coding, § 1(1)

T Turbo Decoders, § 6(37)

V Viterbi decoding, § 5(31)

Attributions

Collection: *Communications Source and Channel Coding with examples*
Edited by: Peter Grant
URL: http://cnx.org/content/col10601/1.3/
License: http://creativecommons.org/licenses/by/2.0/

Module: "Huffman source coder"
By: Peter Grant
URL: http://cnx.org/content/m18172/1.4/
Pages: 1-6
Copyright: Peter Grant
License: http://creativecommons.org/licenses/by/2.0/

Module: "Block FECC coding"
By: Peter Grant
URL: http://cnx.org/content/m18174/1.3/
Pages: 7-12
Copyright: Peter Grant
License: http://creativecommons.org/licenses/by/2.0/

Module: "Block code performance"
By: Peter Grant
URL: http://cnx.org/content/m18175/1.7/
Pages: 13-23
Copyright: Peter Grant
License: http://creativecommons.org/licenses/by/2.0/

Module: "Convolutional FECC Encoder"
By: Peter Grant
URL: http://cnx.org/content/m18176/1.3/
Pages: 25-29
Copyright: Peter Grant
License: http://creativecommons.org/licenses/by/2.0/

Module: "Viterbi Decoder"
By: Peter Grant
URL: http://cnx.org/content/m18177/1.3/
Pages: 31-36
Copyright: Peter Grant
License: http://creativecommons.org/licenses/by/2.0/

Module: "Turbo Coding"
By: Peter Grant
URL: http://cnx.org/content/m18178/1.3/
Pages: 37-41
Copyright: Peter Grant
License: http://creativecommons.org/licenses/by/2.0/

Communications Source and Channel Coding with examples
Huffman variable length source coder is first described then systematic channel coding block coder design is introduced for forward error correction coding (FECC). Nearest neighbour decoding and the Hamming bound is used to define the performance of these block coders. Finally the nonsystematic convolutional coder, Viterbi decoder and turbo recursive coder designs are introduced with examples of the operation of these coders.