

Abstract

This project presents a study on the Design of high performance turbo encoder and iterative turbo decoder using different algorithms. In data transmission, turbo coding helps achieve near Shannon limit performance. MAP, LOG-MAP have been studied and their design considerations have been presented. Turbo coding is an advanced error correction technique widely used in the communications industry. Turbo encoders and decoders are key elements in today's communication systems to achieve the best possible data reception with the fewest possible errors. The basis of turbo coding is to introduce redundancy in the data to be transmitted through a channel. The redundant data helps to recover original data from the received data. The design problem of generating the turbo code and decoding the code iteratively using MAP detectors has been considered. This paper has proposed a design of a TURBO Decoder using MAP Algorithm. Turbo Codes is implemented on DSK TMS320C6424, INTEL atom processor and WiCOMM_T.

Key Words: Iterative Decoding, Turbo Codes, Redundancy, Forward Error Correction.

INTRODUCTION

1.1 Basics

In information theory and coding theory with applications in computer science and telecommunication, error detection and correction or error control are techniques that enable reliable delivery of digital data over unreliable communication channels. Many communication channels are subject to channel noise, and thus errors may be introduced during transmission from the source to a receiver. Error detection techniques allow detecting such errors, while error correction enables reconstruction of the original data. The near Shannon limit error correction performance of Turbo codes and parallel concatenated convolutional codes have raised a lot of interest in the research community to find practical decoding algorithms for implementation of these codes. The demand of turbo codes for wireless communication systems has been increasing since they were first introduced by Berrou et. al. in the early 1990's. Various systems such as 3GPP, HSDPA and WiMAX have already adopted turbo codes in their standards due to their large coding gain. In it has also been shown that turbo codes can be applied to other wireless communication systems used for satellite and deep space applications.'

1.2 SHANNON–HARTLEY THEOREM

The field of Information Theory, of which Error Control Coding is a part, is founded upon a paper by Claude Shannon in 1948. Shannon calculated a theoretical maximum rate at which data could be transmitted over a channel perturbed by additive white Gaussian noise (AWGN) with an arbitrarily low bit error rate. This maximum data rate, the capacity of the channel, was shown to be a function of the average received signal power, W , the average noise power N , and the bandwidth of the system. This function, known as the Shannon-Hartley Capacity Theorem, can be stated as: $C = W \log_2 (1 + S/N)$ bits/sec If W is in Hz, then the capacity, C is in bits/s. Shannon stated that it is theoretically possible to transmit data over such a channel at any rate $R \leq C$ with an arbitrarily small error probability

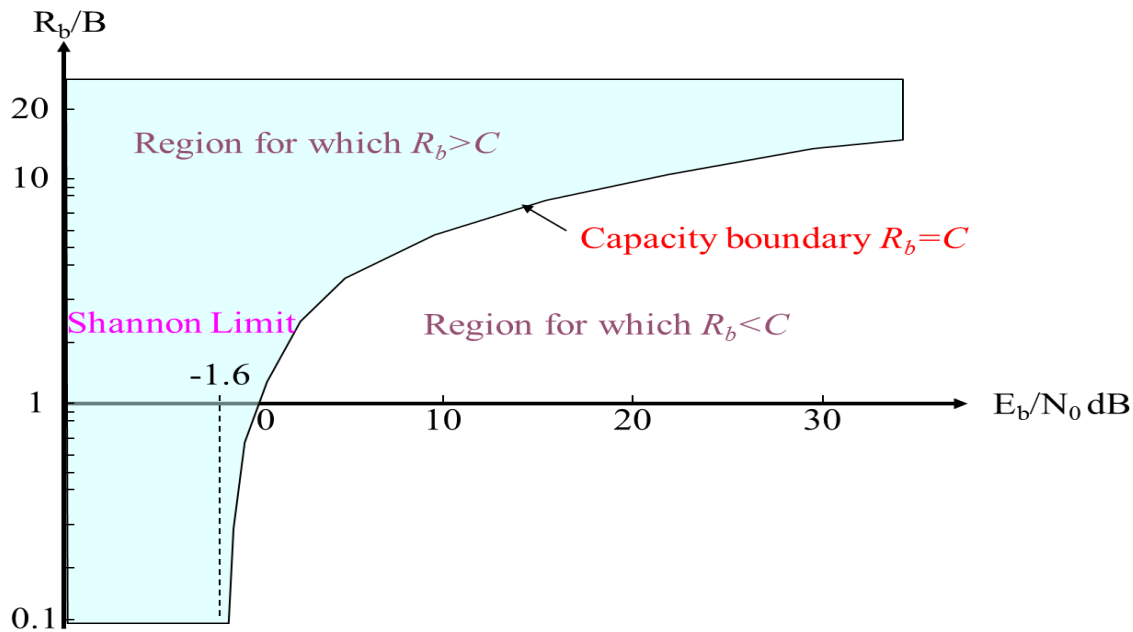


Fig 1: Representation of Shannon limit

1.3 CODING IN WIRELESS COMMUNICATIONS

Coding theory is the study of the properties of codes and their fitness for a specific application and used for data compression, error correction and more recently also for network coding. Codes are studied by various scientific disciplines, such as information theory, electrical engineering, mathematics, and computer science for the purpose of designing efficient and reliable data transmission methods. This typically involves the removal of redundancy and the correction (or detection) of errors in the transmitted data. Most digital communication techniques rely on error correcting coding to achieve an acceptable performance under poor carrier to noise conditions. Basically coding in wireless communications are of two types:

1.5 Source coding: In computer science and information theory, „data compression“, „source coding“, or „bit-rate reduction“ involves encoding information using fewer bits than the original representation. Compression can be either lossy or lossless. The lossless compression reduces bits by identifying and eliminating statistical redundancy. No information is lost in lossless compression. Lossy compression reduces bits by identifying unnecessary information and

removing it. The process of reducing the size of a data file is popularly referred to as data compression, although its formal name is source coding (coding done at the source of the data before it is stored or transmitted).

1.5 Channel coding: The channel coding also called as forward corrections codes (FEC). The purpose of channel coding is to find codes which transmit quickly, contain many valid code words and can correct or at least detect many errors. Channel coding is referred to the processes done in both transmitter and receiver of a digital communications system. While not mutually exclusive, performance in these areas is a trade off. So, different codes are optimal for different applications. The needed properties of this code mainly depend on the probability of errors happening during transmission. Channel coding is distinguished from source coding, i.e., digitizing of analog message signals and data compression.

Types of FEC Codes: 1. Linear block codes. 2. Convolutional codes.

1.5. 1. Linear Block Codes: With Block Codes a block of data has error detecting and correcting bits added to it. One of the simplest error correcting block code is the Hamming Code, where parity bits are added to the data. By adding the error correcting bits to data, transmission errors can be corrected. However since more data has to be squeezed into the same channel bandwidth the more errors will occur. Linear block codes have the property of linearity, i.e. the sum of any two code words is also a code word, and they are applied to the source bits in blocks, hence the name, linear block codes. There are block codes that are not linear, but it is difficult to prove that a code is a good one without this property. Linear block codes are summarized by their symbol alphabets (e.g., binary or ternary) and parameters (n, m, d_{min}) where n is the length of the codeword, in symbols, m is the number of source symbols that will be used for encoding at once, d_{min} is the minimum hamming distance for the code. Block codes submit k bits in their inputs and forwards n bits in their output. These codes are frequently known as (n,k) codes. Apparently, whatever coding scheme is, it has added $n-k$ bits to the coded block. Block codes are used primarily to correct or detect errors in data transmission. Commonly used block codes are Reed–Solomon codes, BCH codes, Golay codes and Hamming codes

1.5. 2. Convolutional Codes: Despite of block codes which are memory less, convolutional codes are coding algorithms with memory. Since, their coding rate (R) is higher than its counterpart in block codes they are more frequently used coding method in practice. Every convolutional code

uses m units of memory, therefore a convolutional code represents with (n,k,m) . In Convolutional coding the input bits are passed through a shift register of length K . N output bits are generated by modulo 2 adding selected bits held in different stages of the shift register. For each new data bit N output bits are produced. The output bits are influenced by K data bits, so that the information is spread in time. The channel code is used to protect data sent over it for storage or retrieval even in the presence of noise (errors). In practical communication systems, convolutional codes tend to be one of the more widely used channel codes. These codes are used primarily for real-time error correction and can convert an entire data stream into one single codeword. The Viterbi algorithm provided the basis for the main decoding strategy of convolutional codes. The encoded bits depend not only on the current informational k input bits but also on past input bits.

Turbo Codes

2.1 Theory Of Turbo Codes

In 1993 Berrou, Glavieux and Thitimajshima² proposed “a new class of convolution codes called Turbo codes whose performance in terms of Bit Error Rate (BER) are close to the Shannon limit”.

Turbo codes are a class of forward error correction codes that provide a high performance in error correction, close to the channel capacity. The information sequence is encoded twice, with an interleaver between the two encoders serving to make the two encoded data sequences approximately statistically independent of each other. Often half rate Recursive Systematic Convolutional (RSC) encoders are used, with each RSC encoder producing a systematic output which is equivalent to the original information sequence, as well as a stream of parity information. The two parity sequences can then be punctured before being transmitted along with the original information sequence to the decoder. This puncturing of the parity information allows a wide range of coding rates to be realised, and often half the parity information from each encoder is sent. Along with the original data sequence this results in an overall coding rate of $1/2$. This feature allows trading off data rate by robustness of the transmitted data against noise in the channel. When the noise in the channel is significant all the encoded bits are transmitted, conversely, when the noise in the channel decreases, some encoded bits can be removed, provided that the decoder is able to recover the transmitted data, and thus increasing the transmitted data rate. At the decoder two RSC decoders are used. Special decoding algorithms must be used which accept soft inputs and give soft outputs for the decoded sequence. These soft inputs and outputs provide not only an indication of whether a particular bit was a 0 or a 1, but also a likelihood ratio which gives the probability that the bit has been correctly decoded. The turbo decoder operates iteratively. In the first iteration the first RSC decoder provides a soft output giving an estimation of the original data sequence based on the soft channel inputs alone. It also provides an extrinsic output. The extrinsic output for a given bit is based not on the channel input for that bit, but on the information for surrounding bits and the constraints imposed by the code being used. This extrinsic output from the first decoder is used by the second RSC decoder as a-priori information, and this information together with the channel inputs are used by the second RSC decoder to give its soft output and extrinsic information. In the second iteration

the extrinsic information from the second decoder in the first iteration is used as the a-priori information for the first decoder, and using this a-priori information the decoder can hopefully decode more bits correctly than it did in the first iteration. This cycle continues, with at each iteration both RSC decoders producing a soft output and extrinsic Information based on the channel inputs and a-priori information obtained from the extrinsic information provided by the previous decoder. After each iteration the Bit Error Rate (BER) in the decoded sequence drops, but the improvements obtained with each iteration falls as the number iterations increases so that for complexity reasons usually only between 4 and 12 iterations are used.



Fig 2: Communication block diagram

2.2 TURBO ENCODER

A turbo encoder is the parallel concatenation of recursive systematic convolutional (RSC) codes, separated by an interleaver, as shown in Fig. 2. The data flow d_k goes into the first elementary RSC encoder, and after interleaving, it feeds a second elementary RSC encoder. The input stream is also systematically transmitted as X_k , and the redundancies produced by encoders 1 and 2 are transmitted as Y_{1k} and Y_{2k} . For turbo codes, the main reason of using RSC encoders as constituent encoders instead of the traditional non-recursive non-systematic convolutional codes is to use their recursive nature and not the fact that they are systematic.

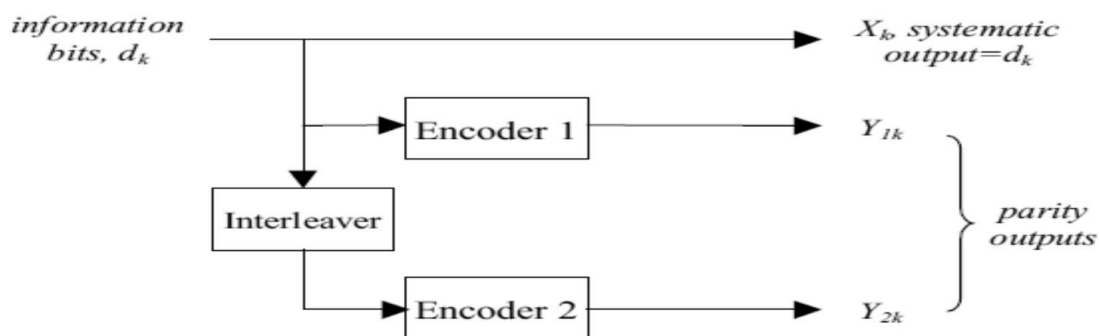
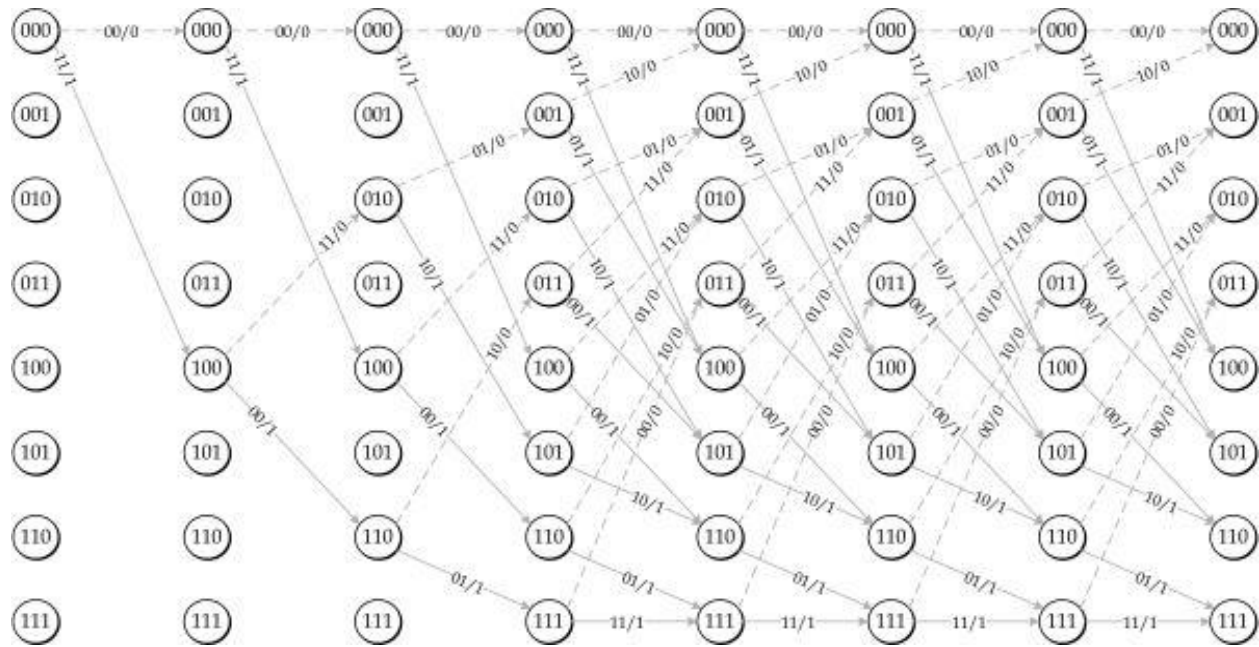


Fig 2: Turbo Encoder(Source – Google Images)

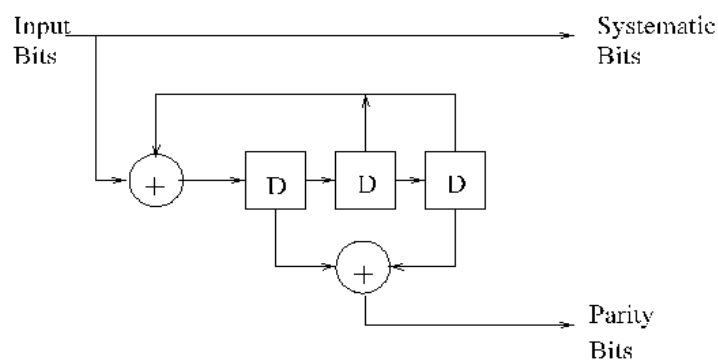
The interleaver is an important design parameter in a turbo code. It takes a particular stream at its input and produces a different sequence as output. Its main purpose at the encoder side is to increase the free distance of the turbo code, hence improving its error-correction performance.

Based on the equations of the turbo encoder we get the Trellis Diagram as shown in Fig 2



2.2.1 Recursive systematic Convolution Encoder:

The RSC Encoder used in the turbo encoders for standard LTE proposed by 3GPP is as shown in fig 2



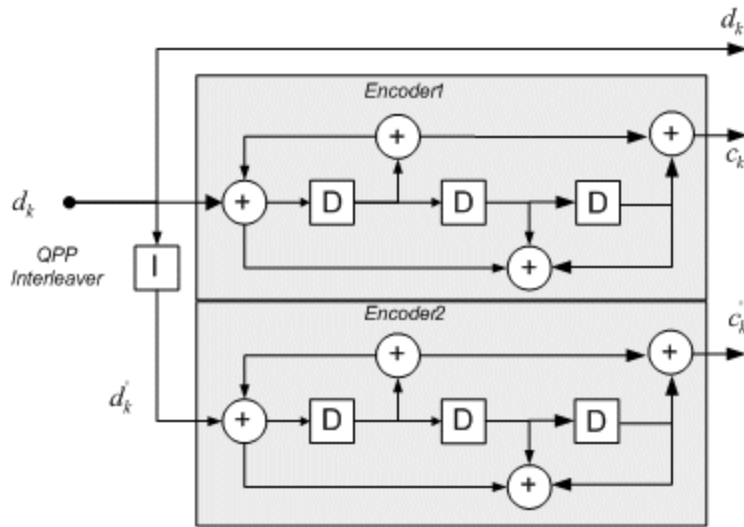


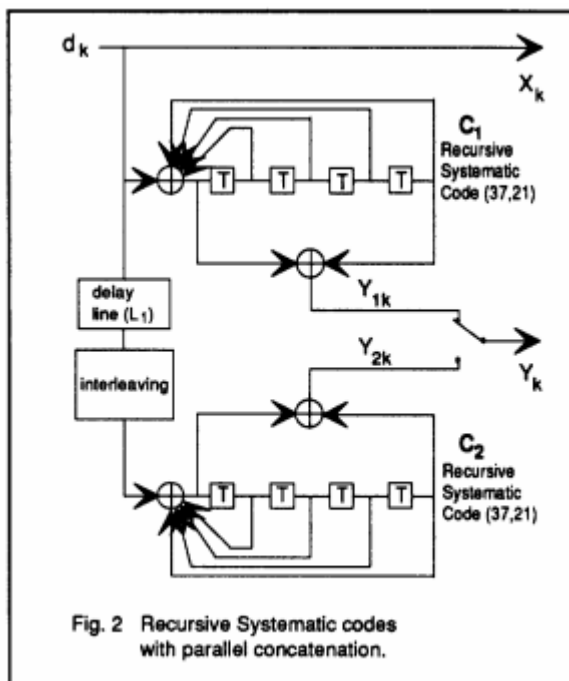
Fig 3: RSC Encoder (Source – Google Images)

The transfer function of the 8-state constituent code for the RSC encoder is shown in equation (a)

$$G(D) = \begin{bmatrix} 1, \frac{g_1(D)}{g_0(D)} \\ g_0(D) \end{bmatrix} \dots\dots\dots (a)$$

Where $g_0(D) = 1 + D^2 + D^3$,

$g_1(D) = 1 + D + D^3$.



2.2.2 INTERLEAVER:

The interleaving technique is commonly used in communication systems to overcome correlated channel noise such as burst error or fading. Essential to turbo coding are the interleaver (and de-interleaver) blocks that rest between the different encoder and decoder pairs. There are different types of interleavers, such as the block, pseudo-random, simple, and odd-even interleavers. They differ in the way they shuffle the input symbols. As the name implies, the function of the Interleaver is to take each incoming block of N data bits and rearrange them in a pseudorandom fashion prior to encoding by the second encoder. Permuting the input bits effectively randomizes the noise across the channel and minimizes burst errors. Although there are many different types of interleavers that one can use, good interleavers are known to break low weight input sequences and increase free Hamming distance of the code or reduce the number of codewords with small distances in the code distance spectrum.

In this work, the interleaver follows a very simple permutation pattern and hence may not be the most effective for high-performance systems: inputs are written into a matrix by row, and the outputs are fed column wise. Thus, the de-interleavers used to reconstruct the original order are easily constructed since they are identical structures.

The bits input to the turbo code internal interleaver are denoted by c_0, c_1, \dots, c_{K-1} , where K is the number of input bits. The bits output from the turbo code internal interleaver are denoted by $c'_0, c'_1, \dots, c'_{K-1}$. The relationship between the input and output bits is as shown in equation (b):

$$c'_i = c_{\Pi(i)}, i=0, 1, \dots, (K-1) \dots \dots (b)$$

where the relationship between the output index i and the input index $\Pi(i)$ satisfies the following quadratic form shown in equation (c):

$$\Pi(i) = (f_1 \cdot i + f_2 \cdot i^2) \bmod K \dots \dots (c)$$

The parameters f_1 and f_2 depend on the block size K and are summarized in Table 5.1.3-3.

i	K	f_1	f_2	i	K	f_1	f_2	i	K	f_1	f_2	i	K	f_1	f_2
1	40	3	10	48	416	25	52	95	1120	67	140	142	3200	111	240
2	48	7	12	49	424	51	106	96	1152	35	72	143	3264	443	204
3	56	19	42	50	432	47	72	97	1184	19	74	144	3328	51	104
4	64	7	16	51	440	91	110	98	1216	39	76	145	3392	51	212
5	72	7	18	52	448	29	168	99	1248	19	78	146	3456	451	192
6	80	11	20	53	456	29	114	100	1280	199	240	147	3520	257	220
7	88	5	22	54	464	247	58	101	1312	21	82	148	3584	57	336
8	96	11	24	55	472	29	118	102	1344	211	252	149	3648	313	228
9	104	7	26	56	480	89	180	103	1376	21	86	150	3712	271	232
10	112	41	84	57	488	91	122	104	1408	43	88	151	3776	179	236
11	120	103	90	58	496	157	62	105	1440	149	60	152	3840	331	120
12	128	15	32	59	504	55	84	106	1472	45	92	153	3904	363	244
13	136	9	34	60	512	31	64	107	1504	49	846	154	3968	375	248
14	144	17	108	61	528	17	66	108	1536	71	48	155	4032	127	168
15	152	9	38	62	544	35	68	109	1568	13	28	156	4096	31	64
16	160	21	120	63	560	227	420	110	1600	17	80	157	4160	33	130
17	168	101	84	64	576	65	96	111	1632	25	102	158	4224	43	264
18	176	21	44	65	592	19	74	112	1664	183	104	159	4288	33	134
19	184	57	46	66	608	37	76	113	1696	55	954	160	4352	477	408
20	192	23	48	67	624	41	234	114	1728	127	96	161	4416	35	138
21	200	13	50	68	640	39	80	115	1760	27	110	162	4480	233	280
22	208	27	52	69	656	185	82	116	1792	29	112	163	4544	357	142
23	216	11	36	70	672	43	252	117	1824	29	114	164	4608	337	480
24	224	27	56	71	688	21	86	118	1856	57	116	165	4672	37	146
25	232	85	58	72	704	155	44	119	1888	45	354	166	4736	71	444
26	240	29	60	73	720	79	120	120	1920	31	120	167	4800	71	120
27	248	33	62	74	736	139	92	121	1952	59	610	168	4864	37	152
28	256	15	32	75	752	23	94	122	1984	185	124	169	4928	39	462
29	264	17	198	76	768	217	48	123	2016	113	420	170	4992	127	234

30	272	33	68	77	784	25	98	124	2048	31	64	171	5056	39	158
31	280	103	210	78	800	17	80	125	2112	17	66	172	5120	39	80
32	288	19	36	79	816	127	102	126	2176	171	136	173	5184	31	96
33	296	19	74	80	832	25	52	127	2240	209	420	174	5248	113	902
34	304	37	76	81	848	239	106	128	2304	253	216	175	5312	41	166
35	312	19	78	82	864	17	48	129	2368	367	444	176	5376	251	336
36	320	21	120	83	880	137	110	130	2432	265	456	177	5440	43	170
37	328	21	82	84	896	215	112	131	2496	181	468	178	5504	21	86
38	336	115	84	85	912	29	114	132	2560	39	80	179	5568	43	174
39	344	193	86	86	928	15	58	133	2624	27	164	180	5632	45	176
40	352	21	44	87	944	147	118	134	2688	127	504	181	5696	45	178
41	360	133	90	88	960	29	60	135	2752	143	172	182	5760	161	120
42	368	81	46	89	976	59	122	136	2816	43	88	183	5824	89	182
43	376	45	94	90	992	65	124	137	2880	29	300	184	5888	323	184
44	384	23	48	91	1008	55	84	138	2944	45	92	185	5952	47	186
45	392	243	98	92	1024	31	64	139	3008	157	188	186	6016	23	94
46	400	151	40	93	1056	17	66	140	3072	47	96	187	6080	47	190
47	408	155	102	94	1088	171	204	141	3136	13	28	188	6144	263	480

Table 5.1.3-3: Turbo code internal interleaver parameters.

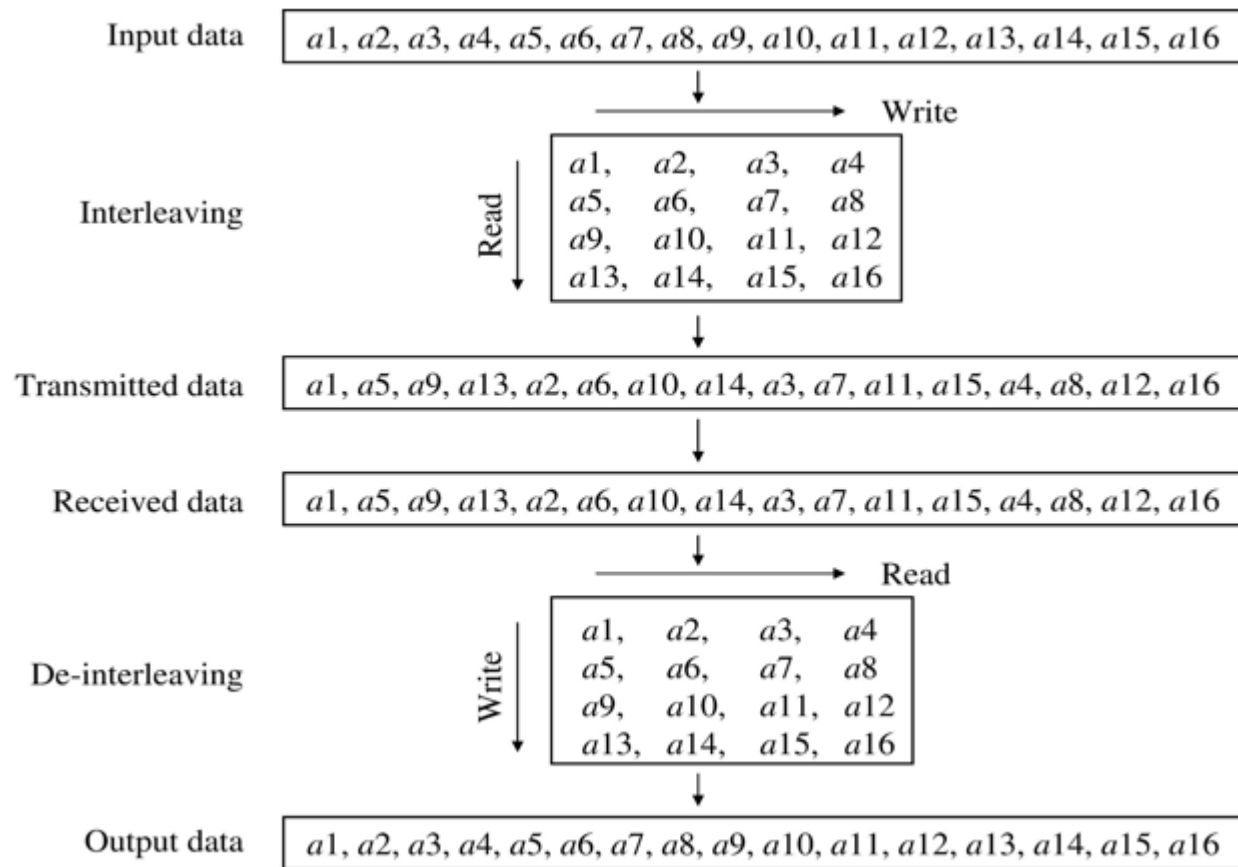


Fig 3: Concepts of interleaver

2.2.3 Trellis state

The name *trellis* was coined because a state diagram of the technique, when drawn on paper, closely resembles the trellis lattice used in rose gardens. The scheme is basically a convolutional code of rates $(r, r+1)$. Ungerboeck's unique contribution is to apply the parity check on a per symbol basis instead of the older technique of applying it to the bit stream then modulating the bits. The key idea he termed Mapping by Set Partitions. This idea was to group the symbols in a tree like fashion then separate them into two limbs of equal size. At each limb of the tree, the symbols were further apart. Although hard to visualize in multi-dimensions, a simple one dimension example illustrates the basic procedure. Suppose the symbols are located at $[1, 2, 3, 4, \dots]$. Then take all odd symbols and place them in one group, and the even symbols in the second

group. This is not quite accurate because Ungerboeck was looking at the two dimensional problem, but the principle is the same, take every other one for each group and repeat the procedure for each tree limb. He next described a method of assigning the encoded bit stream onto the symbols in a very systematic procedure. Once this procedure was fully described, his next step was to program the algorithms into a computer and let the computer search for the best codes. The results were astonishing. Even the most simple code (4 state) produced error rates nearly one one-thousandth of an equivalent uncoded system. For two years Ungerboeck kept these results private and only conveyed them to close colleagues. Finally, in 1982, Ungerboeck published a paper describing the principles of trellis modulation.

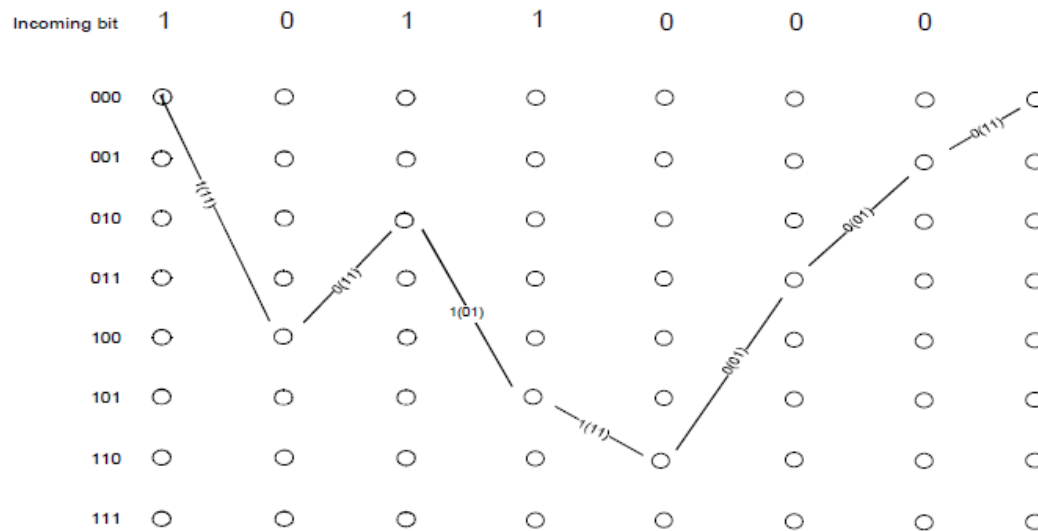
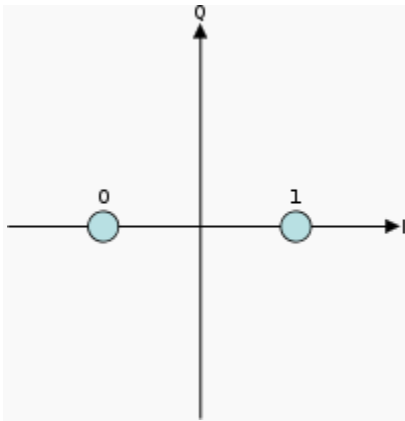


Figure 10a – Encoded Sequence, Input bits 1011000, Outbit 1101111010111

Binary phase-shift keying (BPSK)



Constellation diagram example for BPSK.

BPSK (also sometimes called PRK, phase reversal keying, or 2PSK) is the simplest form of phase shift keying (PSK). It uses two phases which are separated by 180° and so can also be termed 2-PSK. It does not particularly matter exactly where the constellation points are positioned, and in this figure they are shown on the real axis, at 0° and 180° . This modulation is the most robust of all the PSKs since it takes the highest level of noise or distortion to make the demodulator reach an incorrect decision. It is, however, only able to modulate at 1 bit/symbol (as seen in the figure) and so is unsuitable for high data-rate applications.

In the presence of an arbitrary phase-shift introduced by the communications channel, the demodulator is unable to tell which constellation point is which. As a result, the data is often differentially encoded prior to modulation.

BPSK is functionally equivalent to 2-QAM modulation.

Implementation[edit]

The general form for BPSK follows the equation:

$$s_n(t) = \sqrt{\frac{2E_b}{T_b}} \cos(2\pi f_c t + \pi(1 - n)), n = 0, 1.$$

This yields two phases, 0 and π . In the specific form, binary data is often conveyed with the following signals:

$$s_0(t) = \sqrt{\frac{2E_b}{T_b}} \cos(2\pi f_c t + \pi) = -\sqrt{\frac{2E_b}{T_b}} \cos(2\pi f_c t) \quad \text{for binary "0"}$$

$$s_1(t) = \sqrt{\frac{2E_b}{T_b}} \cos(2\pi f_c t) \quad \text{for binary "1"}$$

where f_c is the frequency of the carrier-wave.

Hence, the signal-space can be represented by the single basis function

$$\phi(t) = \sqrt{\frac{2}{T_b}} \cos(2\pi f_c t)$$

where 1 is represented by $\sqrt{E_b}\phi(t)$ and 0 is represented by $-\sqrt{E_b}\phi(t)$. This assignment is, of course, arbitrary. This use of this basis function is shown at the end of the next section in a signal timing diagram. The topmost signal is a BPSK-modulated cosine wave that the BPSK modulator would produce. The bit-stream that causes this output is shown above the signal (the other parts of this figure are relevant only to QPSK).

The bit error rate (BER) of BPSK in AWGN can be calculated as:^[5]

$$P_b = Q\left(\sqrt{\frac{2E_b}{N_0}}\right) \quad \text{or} \quad P_b = \frac{1}{2} \operatorname{erfc}\left(\sqrt{\frac{E_b}{N_0}}\right)$$

Since there is only one bit per symbol, this is also the symbol error rate

Additive white Gaussian noise

Additive white Gaussian noise (AWGN) is a basic noise model used in Information theory to mimic the effect of many random processes that occur in nature. The modifiers denote specific characteristics:

- 'Additive' because it is added to any noise that might be intrinsic to the information system.
- 'White' refers to idea that it has uniform power across the frequency band for the information system. It is an analogy to the color white which has uniform emissions at all frequencies in the visible spectrum.

- 'Gaussian' because it has a normal distribution in the time domain with an average time domain value of zero.

Wideband noise comes from many natural sources, such as the thermal vibrations of atoms in conductors (referred to as thermal noise or Johnson-Nyquist noise), shot noise, black body radiation from the earth and other warm objects, and from celestial sources such as the Sun. The central limit theorem of probability theory indicates that the summation of many random processes will tend to have distribution called Gaussian or Normal.

AWGN is often used as a channel model in which the only impairment to communication is a linear addition of wideband or white noise with a constant spectral density (expressed as watts per hertz of bandwidth) and a Gaussian distribution of amplitude. The model does not account for fading, frequency selectivity, interference, nonlinearity or dispersion. However, it produces simple and tractable mathematical models which are useful for gaining insight into the underlying behavior of a system before these other phenomena are considered.

The AWGN channel is a good model for many satellite and deep space communication links. It is not a good model for most terrestrial links because of multipath, terrain blocking, interference, etc. However, for terrestrial path modeling, AWGN is commonly used to simulate background noise of the channel under study, in addition to multipath, terrain blocking, interference, ground clutter and self interference that modern radio systems encounter in terrestrial operation.

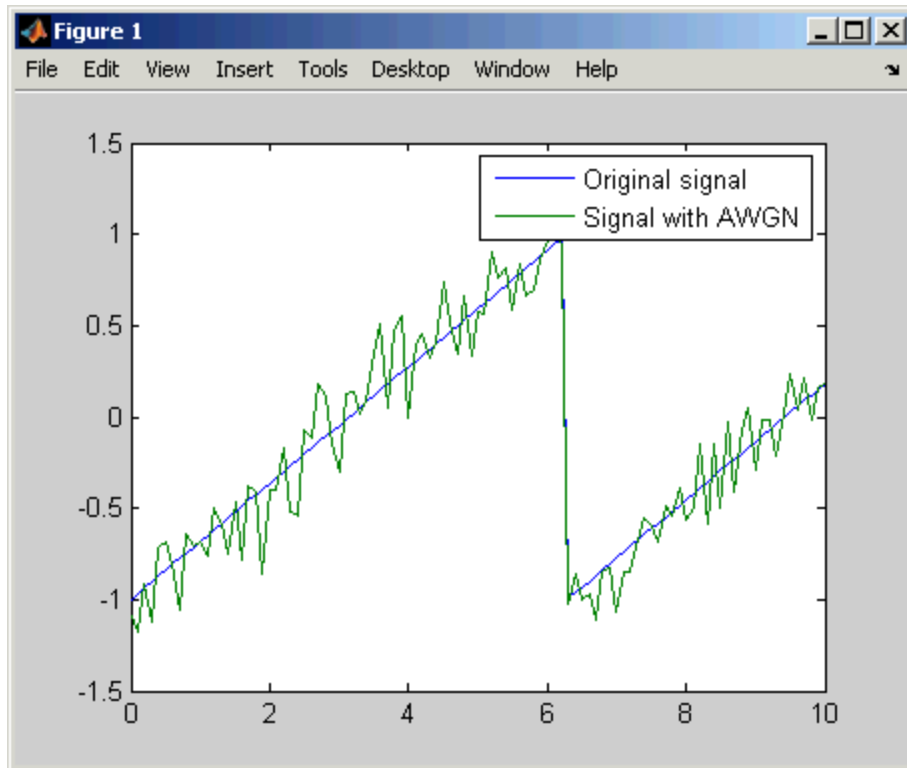


Fig: AWGN noise

TURBO DECODER:

Decoding Algorithms

The two main types of decoder are Maximum A Posteriori (MAP) and the Soft Output Viterbi Algorithm⁴ (SOVA). MAP looks for the most likely symbol received, SOVA looks for the most likely sequence. Both MAP and SOVA perform similarly at high E_b/N_0 . At low E_b/N_0 MAP has a distinct advantage, gained at the cost of added complexity. MAP was first proposed by Bahl⁵ et al and was selected by Berrou² et al as the optimal decoder for turbo codes.

MAP looks for the most probable value for each received bit by calculating the conditional probability of the transition from the previous bit, given the probability of the received bit. The focus on transitions, or state changes within the trellis, makes LLR a very suitable probability measure for use in MAP.

SOVA is very similar to the standard Viterbi algorithm used in hard demodulators. It uses a trellis to establish a surviving path but, unlike its hard counterpart, compares this with the sequences that were used to establish the non-surviving paths. Where surviving and non-

surviving paths overlap the likelihood of that section being on the correct path is reinforced. Where there are differences, the likelihood of that section of the path is reduced. At the output of each decoding stage the values of the bit sequence are scaled by a channel reliability factor, calculated from the likely output sequence, to reduce the probability of over-optimistic soft outputs. The sequence and its associated confidence factors are then presented to the interleaver for further iterations. After the prescribed number of iterations, the SOVA decoder will output the sequence with the maximum likelihood.

In 1974 Bahl, Cocke, Jelinek and Raviv published the decoding algorithm based on a posteriori probabilities later on known as the BCJR, Maximum a Posteriori (MAP) or forward-backward algorithm. The procedure can be applied to block or convolutional codes but, as it is more complex than the Viterbi algorithm, during about 20 years it was not used in practical implementations. The situation was dramatically changed with the advent of turbo codes in 1993. Their inventors, Berrou, Glavieux and Thithimajshima, used a modified version of the BCJR algorithm, which has reborn vigorously that way. There are several simplified versions of the MAP algorithm, namely the log-MAP and the max-log-MAP algorithms. The purpose of this tutorial text is to clearly show, without intermediate calculations, how all these algorithms work and are applied to turbo decoding. A complete worked out example is presented to illustrate the procedures.

The purpose of the BCJR algorithm

Let us consider a block or convolutional encoder described by a trellis, and a sequence

$x = x_1 x_2 \dots x_N$ of N n -bit codewords or symbols at its output, where x_k is the symbol generated by the encoder at time k . The corresponding information or message input bit, u_k , can take on the values -1 or $+1$ with an a priori probability $P(u_k)$, from which we can define the so-called log-likelihood ratio (LLR)

$$L(u_k) = \ln \frac{P(u_k = +1)}{P(u_k = -1)}$$

This log-likelihood ratio is zero with equally likely input bits. Suppose the coded sequence x is transmitted over a memoryless additive white gaussian noise (AWGN) channel and received as

the sequence of nN real numbers $y = y_1 y_2 \dots y_N$, as shown in Fig. 1. This sequence is delivered to the decoder and used by the BCJR [1], or any other, algorithm in order to estimate the original bit sequence u_k . for which the algorithm computes the a posteriori log-likelihood ratio $L(u_k | y)$, a real number defined by the ratio

$$L(u_k | y) = \ln \frac{P(u_k = +1 | y)}{P(u_k = -1 | y)}$$

The numerator and denominator of Eq. (2) contain a posteriori conditional probabilities, that is, probabilities computed after we know y . The positive or negative sign of $L(u_k | y)$ is an indication of which bit, +1 or -1, was coded at time k . Its magnitude is a measure of the confidence we have in the preceding decision: the more the $L(u_k | y)$ magnitude is far away from the zero threshold decision the more we trust in the bit estimation we have made. This soft information provided by $L(u_k | y)$ can then be transferred to another decoding block, if there is one, or simply converted into a bit value through a hard decision: as it was said before, if $L(u_k | y) < 0$ the decoder will estimate bit $u_k = -1$ was sent. Likewise, it will estimate $u_k = +1$ if $L(u_k | y) > 0$.

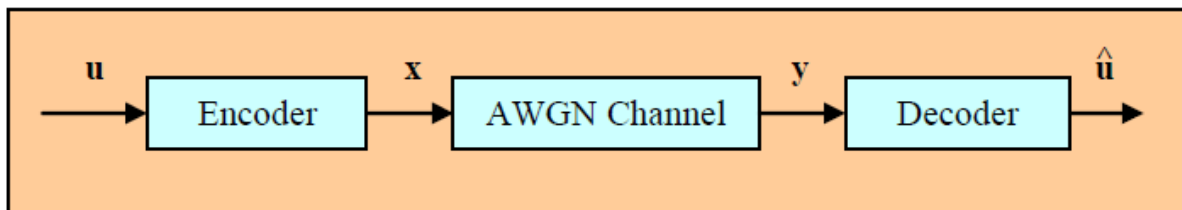


Fig. 1 A simplified block diagram of the system under consideration

It is convenient to work with trellises. Let us admit we have a rate $1/2$, $n = 2$, convolutional code, with $M = 4$ states $S = \{0, 1, 2, 3\}$ and a trellis section like the one presented in Fig. 2. Here a dashed line means that branch was generated by a +1 message bit and a solid line means the opposite. Each branch is labeled with the associated two bit codeword x_k , where, for simplicity, 0 and 1 correspond to -1 and +1, respectively.

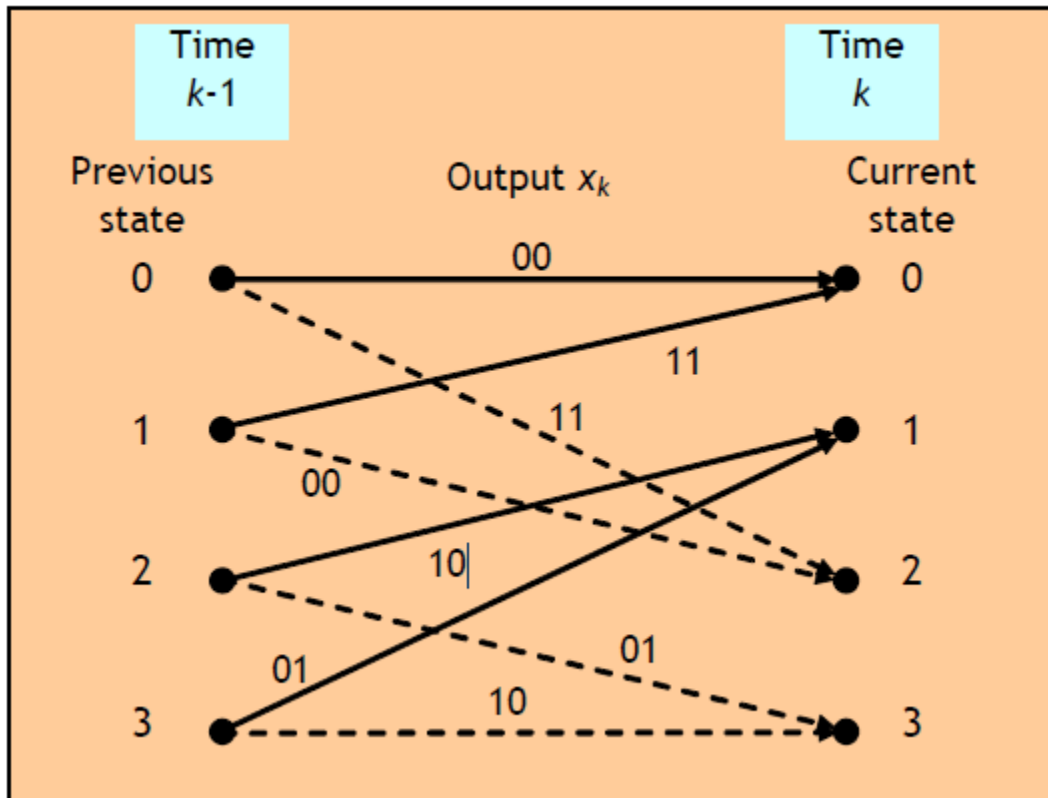


Fig. 2 The convolutional code trellis used in the example

Let us suppose we are at time k . The corresponding state is $S_k = s$, the previous state is $S_{k-1} = s'$ and the decoder received symbol is y_k . Before this time $k-1$ symbols have already been received and after it $N-k$ symbols will be received. That is, the complete sequence y can be divided into three subsequences, one representing the past, another the present and another one the future:

$$\mathbf{y} = \underbrace{y_1 y_2 \dots y_{k-1}}_{\mathbf{y}_{<k}} y_k \underbrace{y_{k+1} \dots y_N}_{\mathbf{y}_{>k}} = \mathbf{y}_{<k} \mathbf{y}_k \mathbf{y}_{>k}$$

It is shown that the a posteriori LLR $L(u_k | y)$ is given by the expression

$$L(u_k | y) = \ln \frac{\sum_{R_1} P(s', s, y)}{\sum_{R_0} P(s', s, y)} = \ln \frac{\sum_{R_1} \alpha_{k-1}(s') \gamma_k(s', s) \beta_k(s)}{\sum_{R_0} \alpha_{k-1}(s') \gamma_k(s', s) \beta_k(s)} \quad (4)$$

$P(s', s, y)$ represents the joint probability of receiving the N-bit sequence y and being in state s' at time $k-1$ and in state s at the current time k . In the numerator $R1$ means the summation is computed over all the state transitions from s' to s that are due to message bits $u_k = +1$ (i. e. , dashed branches). Likewise, in the denominator $R0$ is the set of all branches originated by message bits $u_k = -1$. The variables α , γ and β represent probabilities to be defined later.

2 The joint probability $P(s', s, y)$

This probability can be computed as the product of three other probabilities,

$$P(s', s, y) = \alpha_{k-1}(s') \gamma_k(s', s) \beta_k(s), \quad (5)$$

They are defined as:

$$\alpha_{k-1}(s') = P(s', < k) \quad (6)$$

$$\gamma_k(s', s) = P(y_k, s | s') \quad (7)$$

$$\beta_k(s) = P(y > k | s) \quad (8)$$

At time k the probabilities α , γ and β are associated with the past, the present and the future of sequence y , respectively. Let us see how to compute them by starting with γ .

2.1 Calculation of γ

The probability $\gamma_k(s', s) = P(y_k, s | s')$ is the conditional probability that the received symbol is y_k at time k and the current state is $S_k = s$, knowing that the state from which the connecting branch came was $S_{k-1} = s'$. It turns out that γ is given by $\gamma_k(s', s) = P(y_k | x_k) P(u_k)$

In the special case of an AWGN channel the previous expression becomes

$$\gamma_k(s', s) = C_k e^{u_k L(u_k)/2} \exp\left(\frac{L_c}{2} \sum_{l=1}^n x_{kl} y_{kl}\right), \quad (9)$$

where C_k is a quantity that will cancel out when we compute the conditional LLR $L(u_k | y)$, as it appears both in the numerator and the denominator of Eq. (4), and L_c , the channel reliability measure, or value, is equal to

$$L_c = 4a \frac{E_c}{N_0} = 4aR_c \frac{E_b}{N_0}$$

$N_0/2$ is the noise bilateral power spectral density, a is the fading amplitude (for nonfading channels it is $a = 1$), E_c and E_b are the transmitted energy per coded bit and message bit, respectively, and R_c is the code rate.

2.2 Recursive calculation of α and β

The probabilities α and β can (and should) be computed recursively. The respective recursive formulas are:

$$\alpha_k(s) = \sum_{s'} \alpha_{k-1}(s') \gamma_k(s', s) \quad \text{Initial conditions: } \alpha_0(s) = \begin{cases} 1 & s = 0 \\ 0 & s \neq 0 \end{cases}$$

$$\beta_{k-1}(s') = \sum_s \beta_k(s) \gamma_k(s', s) \quad \text{Initial conditions: } \beta_N(s) = \begin{cases} 1 & s = 0 \\ 0 & s \neq 0 \end{cases}$$

(10) (11)

Please note that:

- In both cases we need the same quantity, $\gamma_k(s', s)$. It will have to be computed first.
- In the $\alpha_k(s)$ case the summation is over all previous states $S_{k-1} = s'$ linked to state s by converging branches, while in the $\beta_{k-1}(s')$ case the summation is over all next states $S_k = s$ reached with the branches stemming from s' . These summations contain only two elements with binary codes.
- The probability α is being computed as the sequence y is received. That is, when computing α we go forward from the beginning to the end of the trellis.
- The probability β can only be computed after we have received the whole sequence y . That is, when computing β we come backward from the end to the beginning of the trellis.

- We will see that α and β are associated with the encoder states and that γ is associated with the branches or transitions between states.
- The initial values $\alpha_0(s)$ and $\beta_N(s)$ mean the trellis is terminated, that is, begins and ends in the all-zero state. Therefore it will be necessary to add some tail bits to the message in order that the trellis path is forced to return back to the initial state.

It is self-evident now why the BCJR algorithm is also known as the “forward-backward algorithm”.

3 Combatting numerical instability

Numerical problems associated with the BCJR algorithm are well known. In fact, the iterative nature of some computations may lead to undesired overflow or underflow situations. To circumvent them normalization countermeasures should be taken. Therefore, instead of using α and β directly from recursive equations (10) and (11) into Eq. (5) those probabilities should previously be normalized by the sum of all α and β at each time, respectively. The same applies to the joint probability $P(s', s, y)$, as follows. Define the auxiliary unnormalized variables α' and β' at each time step k :

$$\alpha'_k(s) = \sum_{s'} \alpha_{k-1}(s') \gamma_k(s', s)$$

$$\beta'_{k-1}(s') = \sum_s \beta_k(s) \gamma_k(s', s)$$

After all M values of α' and β' have been computed sum them all: $\sum_s \alpha'_k(s)$ and $\sum_{s'} \beta'_{k-1}(s')$.

Then normalize α and β dividing them by these summations:

$$\alpha_k(s) = \frac{\alpha'_k(s)}{\sum_s \alpha'_k(s)}$$

$$\beta_{k-1}(s') = \frac{\beta'_{k-1}(s')}{\sum_{s'} \beta'_{k-1}(s')}$$

Likewise, after all $2M$ products $\alpha_{k-1}(s')\gamma_k(s',s)\beta_k(s)$ – over all the trellis branches have been computed at time k their sum

$$\begin{aligned}\Sigma_{Pk} &= \sum_{R_o, R_1} \alpha_{k-1}(s')\gamma_k(s',s)\beta_k(s) = \\ &= \sum_{R_o} \alpha_{k-1}(s')\gamma_k(s',s)\beta_k(s) + \sum_{R_1} \alpha_{k-1}(s')\gamma_k(s',s)\beta_k(s)\end{aligned}$$

will normalize $P(s',s,y)$:

$$P_{norm}(s',s,y) = \frac{P(s',s,y)}{\Sigma_{Pk}}$$

This way we guarantee all α , β and $P_{norm}(s',s,y)$ always sum to 1 at each time step k . None of these normalization summations affect the final log-likelihood ratio $L(u_k|y)$ as all them appear both in its numerator and denominator:

$$L(u_k|y) = \ln \frac{\sum_{R_1} P(s',s,y)}{\sum_{R_0} P(s',s,y)} = \ln \frac{\sum_{R_1} P_{norm}(s',s,y)}{\sum_{R_0} P_{norm}(s',s,y)}$$

4 Trellis-aided calculation of α and β

Fig. 3 shows the trellis of Fig. 2 but now with new labels. We recall that in our convention a dashed line results from a +1 input bit while a solid line results from a -1 input bit.

Let us do the following as computations are made:

- Label each trellis branch with the value of $\gamma_k(s', s)$ computed according to Eq. (9).
- In each state node write the value of $\alpha_k(s)$ computed according to Eqs. (10) or (12) from the initial conditions $\alpha_0(s)$.

In each state node and below $\alpha_k(s)$ write the value of $\beta_k(s)$ computed according to Eqs. (11) or (13) from the initial conditions $\beta_N(s)$

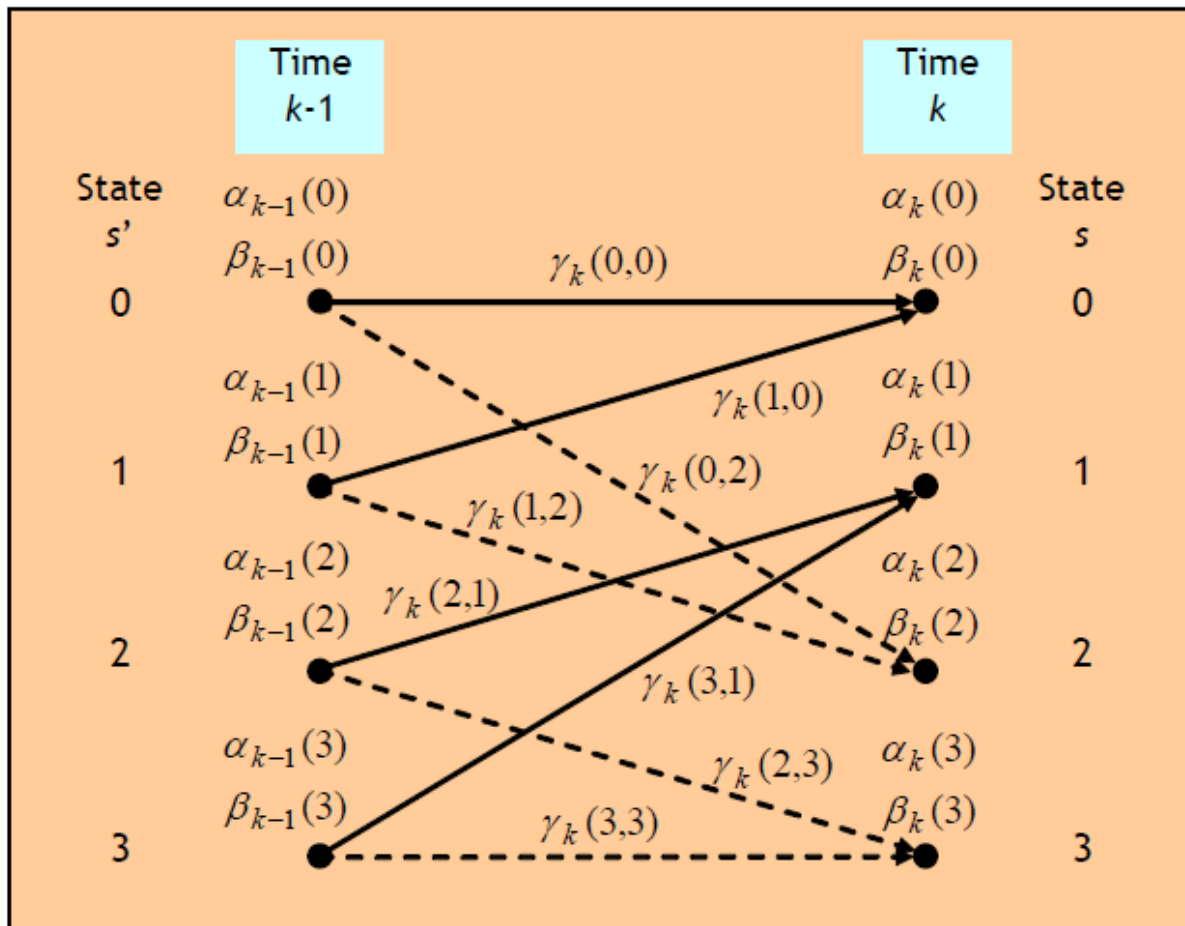


Fig. 3 α , β and γ as trellis labels

4.1 Calculation of α

Let us suppose then that we know $\alpha_{k-1}(s')$. The probability $\alpha_k(s)$ (without normalization) is obtained by summing the products of $\alpha_{k-1}(s')$ and $\gamma_k(s', s)$ associated with the branches that converge into s . For example, we see in Fig. 3 that at time k two branches arrive at state $S_k = 2$,

one coming from state 0 and the other one coming from state 1, as Fig. 4a shows. After all $M \alpha_k(s)$ have been computed as explained they should be divided by their sum.

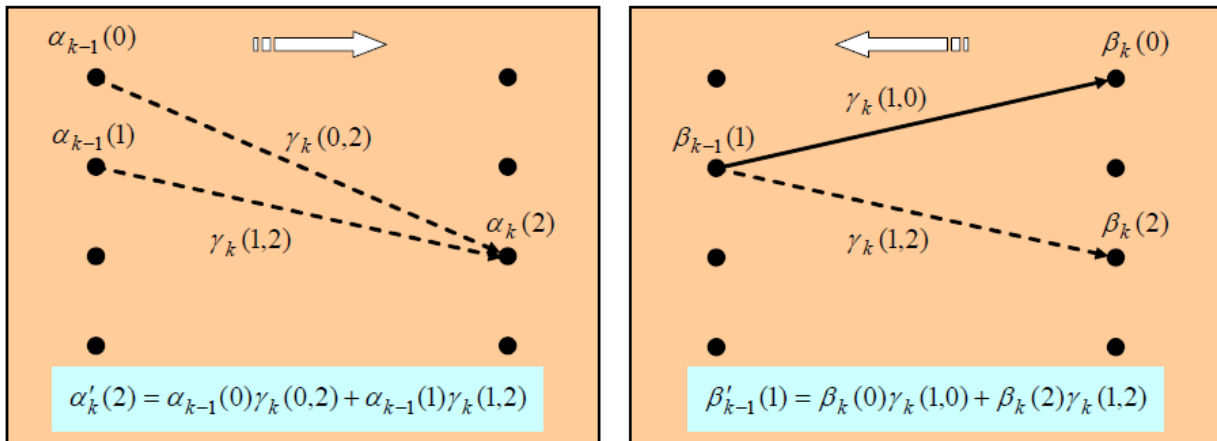


Fig. 4 Trellis-aided recursive computation of α and β . Expressions are not normalized.

The procedure is repeated until we reach the end of the received sequence and have computed $\alpha_N(0)$ (remember our trellis terminates at the all-zero state)

4.2 Calculation of β

The quantity β can be calculated recursively only after the complete sequence y has been received. Knowing $\beta_k(s)$ the value of $\beta_{k-1}(s')$ is computed in a similar fashion as $\alpha_k(s)$ was: we look for the branches that leave state $S_{k-1} = s'$, sum the corresponding products $\gamma_k(s) \beta_k(s)$ and divide by the sum $\sum_{s'} \beta'_{k-1}(s')$. For example, in Fig. 3 we see that two branches leave the state $S_{k-1} = 1$, one directed to state $S_k = 0$ and the other directed to state $S_k = 2$, as Fig. 4b shows. The procedure is repeated until the calculation of $\beta_0(0)$.

4.3 Calculation of $P(s', s, y)$ and $L(u_k/y)$

With all the values of α , β and γ available we are ready to compute the joint probability

$$P_{norm}(s', s, y) = \alpha_{k-1}(s') \gamma_k(s', s) \beta_k(s) / \sum_{P_k}$$

We are left just with the a posteriori LLR $L(u_k/y)$. Well, let us observe the trellis of Fig. 3 again:

we notice that a message bit +1 causes the following state transitions: $0 \rightarrow 2$, $1 \rightarrow 2$, $2 \rightarrow 3$ e $3 \rightarrow 3$. These are the R1 transitions of Eq. (4). The remaining four state transitions, represented by a solid line, are caused, of course, by an input bit -1. These are the R0 transitions. Therefore, the first four transitions are associated with the numerator of Eq. (4) and the remaining ones with the denominator:

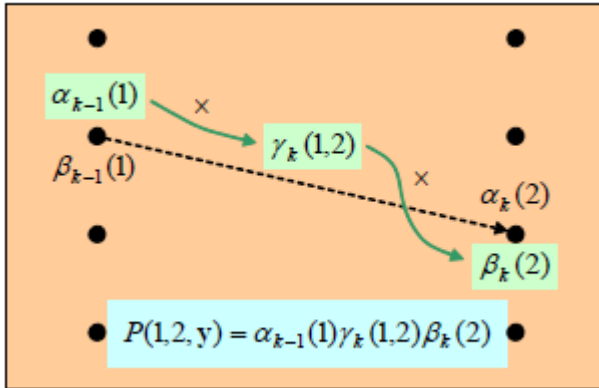


Fig. 5 The unnormalized probability $P(s', s, y)$ as the three-factor product $\alpha\gamma\beta$

$$\begin{aligned}
 L(u_k | y) &= \ln \frac{\sum_{R_1} P(s', s, y)}{\sum_{R_0} P(s', s, y)} = \\
 &= \ln \frac{P(0,2, y) + P(1,2, y) + P(2,3, y) + P(3,3, y)}{P(0,0, y) + P(1,0, y) + P(2,1, y) + P(3,1, y)}
 \end{aligned}$$

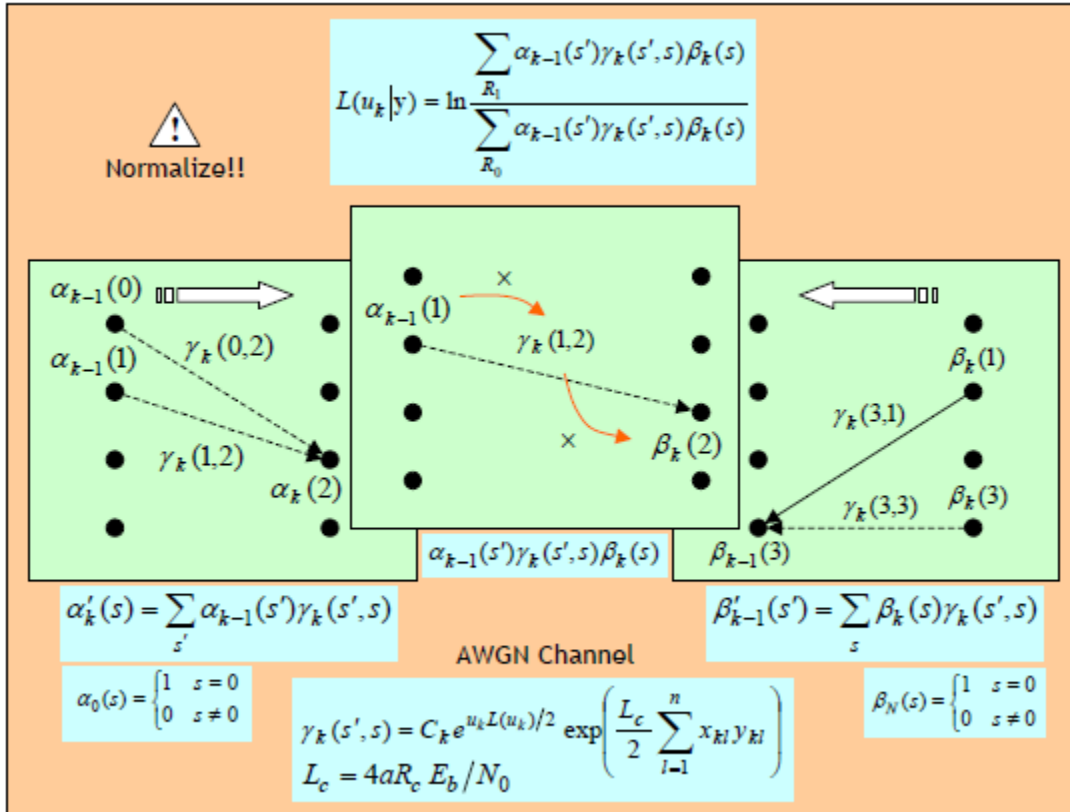


Fig. 6 Summary of expressions used in the BCJR algorithm

5 Simplified versions of the MAP algorithm

The BCJR, or MAP, algorithm suffers from an important disadvantage: it needs to perform many multiplications. In order to reduce this computational complexity several simplifying versions have been proposed, namely the Soft-Output Viterbi Algorithm (SOVA) in 1989 [2], the max-log-MAP algorithm in 1990-1994 [3] [4] and the log-MAP algorithm in 1995 [5]. In this text we will only address the log-MAP and the max-log-MAP algorithms. Both substitute additions for multiplications. Three new variables, A, B e Γ , are defined:

$$\begin{aligned}
\Gamma_k(s', s) &= \ln \gamma_k(s', s) = \\
&= \ln C_k + \frac{u_k L(u_k)}{2} + \frac{L_c}{2} \sum_{l=1}^n x_{kl} y_{kl} \\
A_k(s) &= \ln \alpha_k(s) = \\
&= \max_{s'}^* [A_{k-1}(s') + \Gamma_k(s', s)] & A_0(s) &= \begin{cases} 0 & s = 0 \\ -\infty & s \neq 0 \end{cases} \\
B_{k-1}(s') &= \ln \beta_{k-1}(s') = \\
&= \max_s^* [B_k(s) + \Gamma_k(s', s)] & B_N(s) &= \begin{cases} 0 & s = 0 \\ -\infty & s \neq 0 \end{cases}
\end{aligned}$$

$$\max^*(a, b) = \begin{cases} \max(a, b) + \ln(1 + e^{-|a-b|}) & \text{log-MAP algorithm} \\ \max(a, b) & \text{max-log-MAP algorithm} \end{cases} \quad (15)$$

The element $\ln C_k$ in the expression of $\Gamma_k(s', s)$ will not be used in the $L(u_k|y)$ computation. The LLR is given by the final expression

$$L(u_k|y) = \max_{R_1}^* [A_{k-1}(s') + \Gamma_k(s', s) + B_k(s)] - \max_{R_0}^* [A_{k-1}(s') + \Gamma_k(s', s) + B_k(s)]$$

The log-MAP algorithm uses exact formulas so its performance equals that of the BCJR algorithm although it is simpler, which means it is preferred in implementations. In turn, the max-log-MAP algorithm uses approximations, therefore its performance is slightly worse.

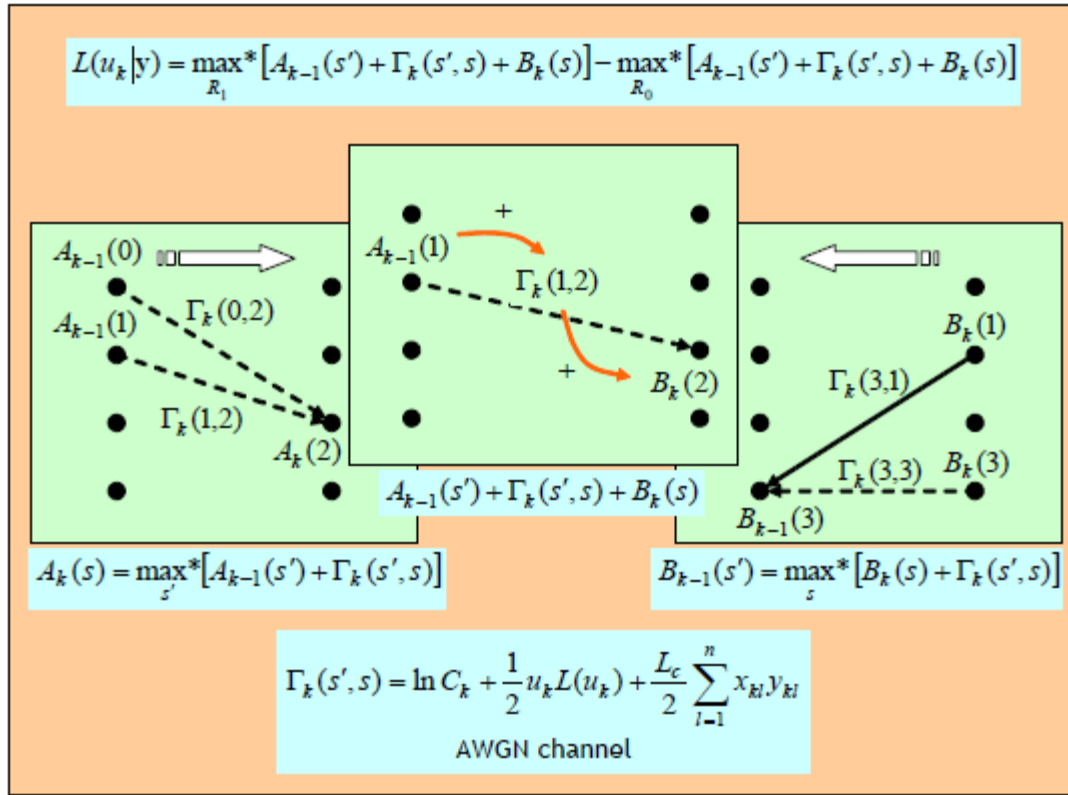


Fig. 7 Summary of expressions used in the simplified MAP algorithms

6 Application of the BCJR algorithm in iterative decoding

Let us consider a rate $1/n$ systematic convolutional encoder in which the first coded bit, x_{k1} , is equal to the information bit u_k . In that case the a posteriori log-likelihood ratio $L(u_k|y)$ can be decomposed into a sum of three elements (see the Appendix for details):

$$L(u_k|y) = L(u_k) + L_c y_{k1} + L_e(u_k).$$

The first two terms in the right hand side are related with the information bit u_k . On the contrary, the third term, $L_e(u_k)$, depends only on the codeword parity bits. That is why $L_e(u_k)$ is called extrinsic information. This extrinsic information is an estimate of the a priori LLR $L(u_k)$. How? It is easy: we provide $L(u_k)$ and $L_c y_{k1}$ as inputs to a MAP (or other) decoder and in turn we get $L(u_k|y)$ at its output. Then, by subtraction, we get the estimate of $L(u_k)$:

$$L_e(u_k) = L(u_k|y) - L(u_k) - L_c y_{k1}$$

This estimate of $L(u_k)$ is presumably a more accurate value of the unknown a priori LLR so it should replace the former value of $L(u_k)$. If we repeat the previous procedure in an iterative way

providing $L(y_k)$ (again) and the new $L(u_k) = L_e(u_k)$ as inputs to another decoder we expect to get a more accurate $L(u_k | y)$ at its output. This fact is explored in turbo decoding, our next subject. The turbo code inventors [6] worked with two parallel concatenated and interleaved rate $1/2$ recursive systematic convolutional codes decoded iteratively with two MAP decoders where P and P^{-1} stand for interleaver and deinterleaver, respectively. P is also a row vector containing the permutation pattern). Both encoders in Fig. 8 are equal and their output parity bits $x_{kp}^{(1)}$ and $x_{kp}^{(2)}$ are often collected alternately through puncturing in order that an overall rate $1/2$ code is obtained. The i -th element of the interleaved sequence, $u_i^{(p)}$, is merely equal to the P_i -th element of the original sequence, $u_i^{(p)} = u_{P_i}$

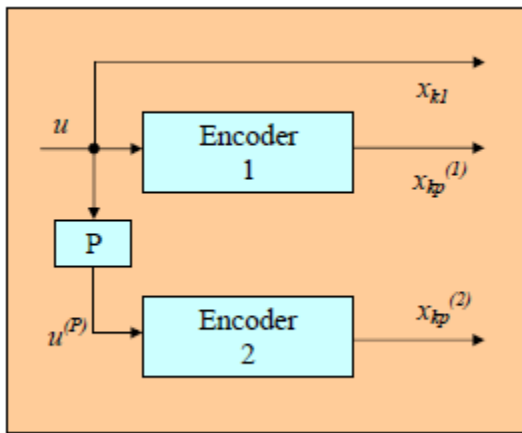


Fig. 8 The turbo encoder

Eq. (17) is the basis for iterative decoding. In the first iteration the a priori LLR $L(u_k)$ is zero if we consider equally likely input bits. The extrinsic information $L_e(u_k)$ that each decoder delivers will be used to update $L(u_k)$ from iteration to iteration and from that decoder to the other. This way the turbo decoder progressively gains more confidence on the ± 1 hard decisions the decision device will have to make in the end of the iterative process. Fig. 9 shows a simplified turbo decoder block diagram that helps illuminate the iterative procedures. We have just seen how to

get an interleaved sequence $u(P)$ from u (we compute $u_i^{(p)} = u_{p_i}$)

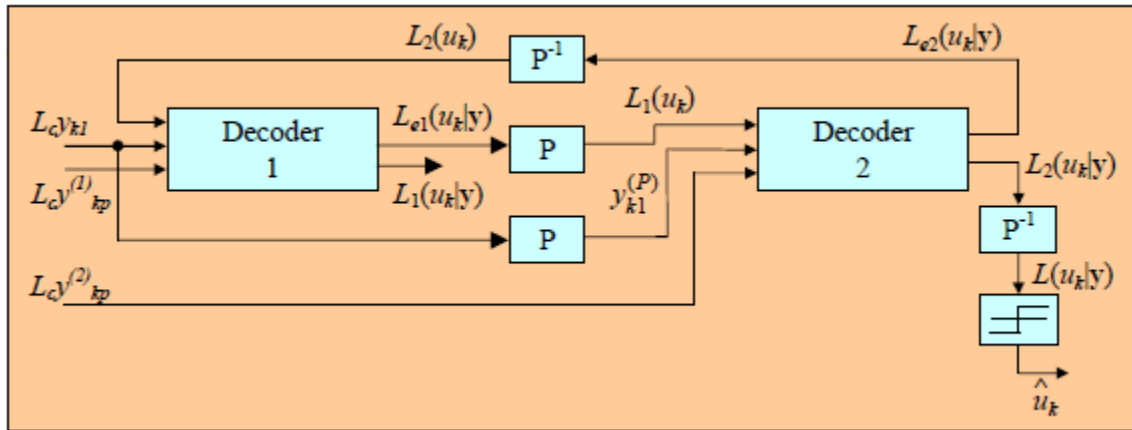


Fig. 9 A simplified block diagram of a turbo decoder

The iterative decoding proceeds as follows:

- in the first iteration we arbitrarily assume $L(u_k) = 0$, then decoder 1 outputs the extrinsic information $L_{e1}(u_k|y)$ on the systematic, or message, bit it gathered from the first parity bit (so note that actually decoder 2 does not need the LLR $L_1(u_k|y)$!);
- After appropriate interleaving the extrinsic information $L_{e1}(u_k|y)$ from decoder 1, computed from Eq. (18), is delivered to decoder 2 as $L_1(u_k)$, a new, more educated guess on $L(u_k)$. Then decoder 2 outputs $L_{e2}(u_k|y)$, its own extrinsic information on the systematic bit based on the other parity bit (note again we keep on “disdaining” the LLR!). After suitable deinterleaving, this information is delivered to decoder 1 as $L_2(u_k)$, a newer, even more educated guess on $L(u_k)$. A new iteration will then begin.
- After a prescribed number of iterations or when a stop criterium is reached the log-likelihood $L_2(u_k|y)$ at the output of decoder 2 is deinterleaved and delivered as $L(u_k|y)$ to the hard decision device, which in turn estimates the information bit based only on the sign of the deinterleaved LLR,

$$\hat{u}_k = \text{sign}[L(u_k|y)] = \text{sign}\{P^{-1}[L_2(u_k|y)]\}.$$

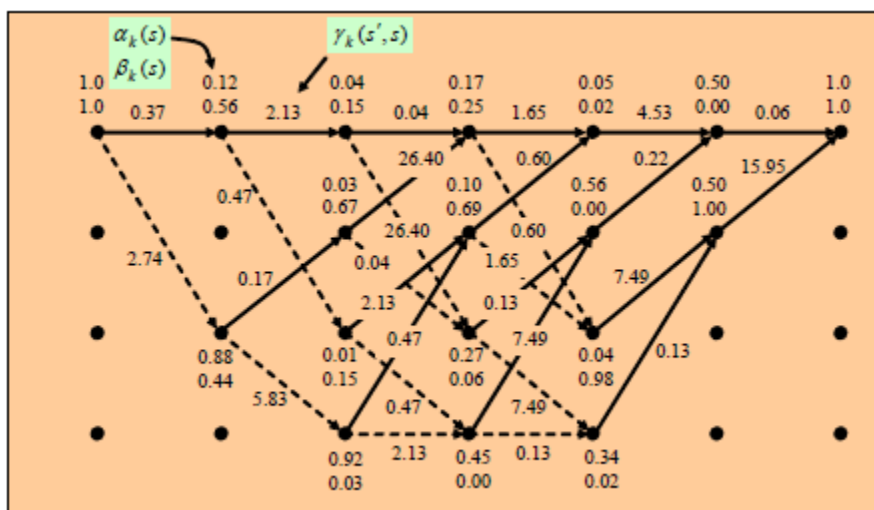


Fig 10 : example for trellis diagram

DESIGN FLOW

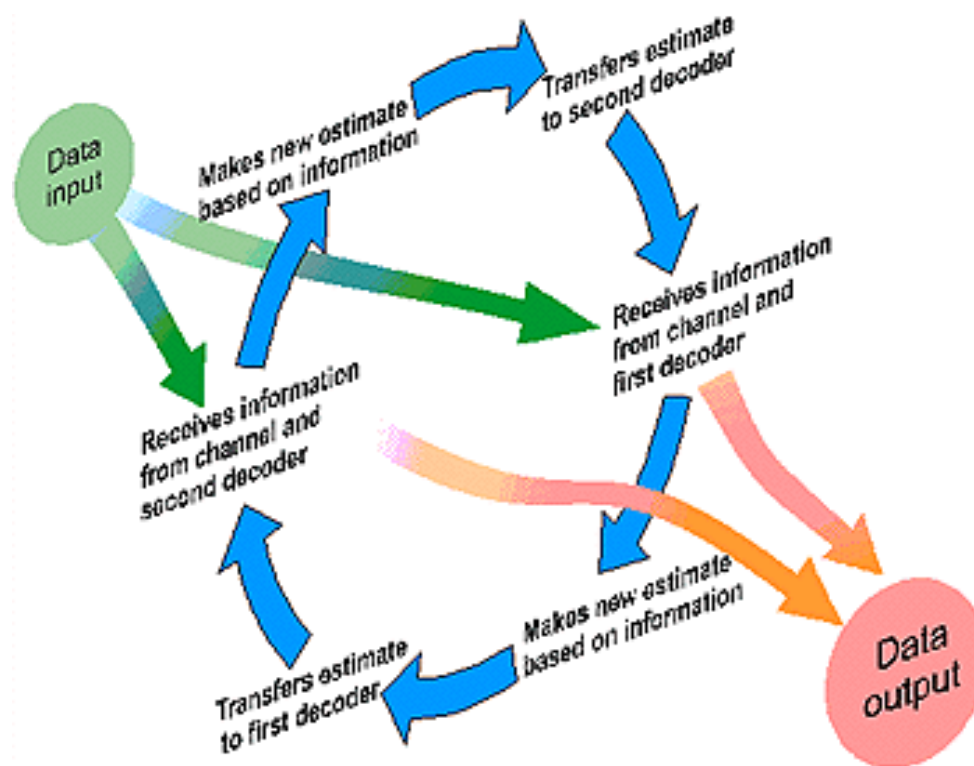


Fig 11 Design Flow diagram

MATLAB

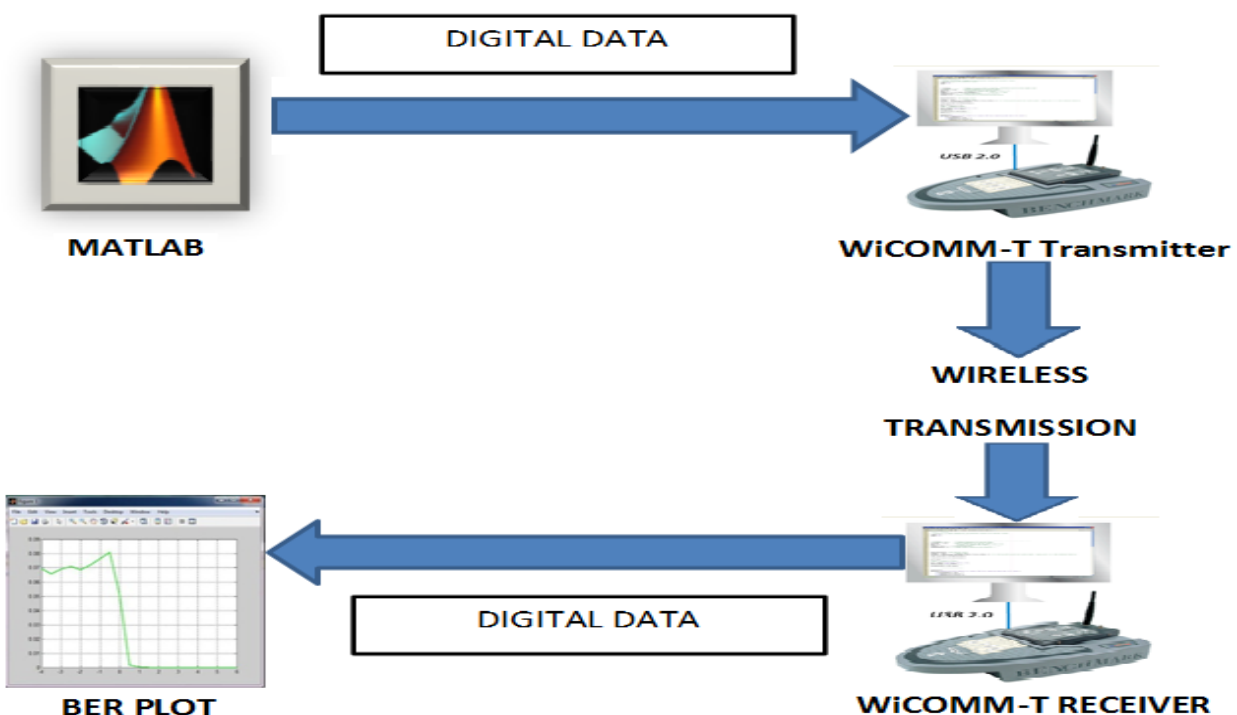
INTRODUCTION

MATLAB (matrix laboratory) is a multi-paradigm numerical computing environment and fourth-generation programming language. Developed by MathWorks, MATLAB allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages, including C, C++, Java, and Fortran.

Although MATLAB is intended primarily for numerical computing, an optional toolbox uses the MuPAD symbolic engine, allowing access to symbolic computing capabilities. An additional package, Simulink, adds graphical multi-domain simulation and Model-Based Design for dynamic and embedded systems.

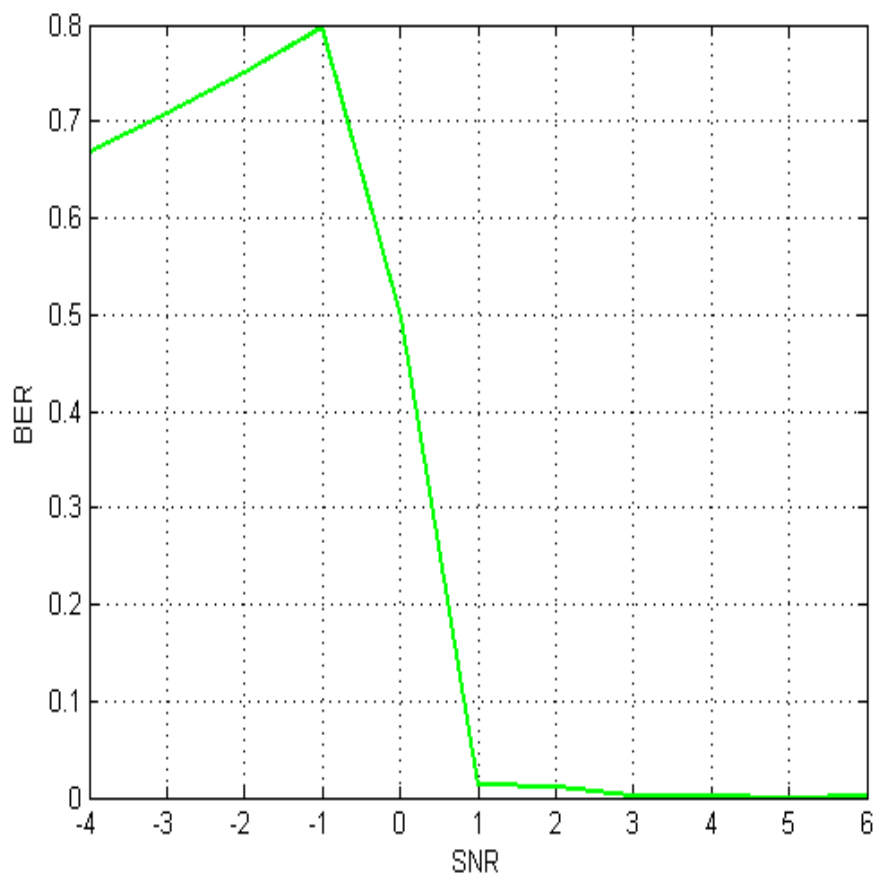
In 2004, MATLAB had around one million users across industry and academia.^[3] MATLAB users come from various backgrounds of engineering, science, and economics. MATLAB is widely used in academic and research institutions as well as industrial enterprises.

APPLICATIONS



Simulations

MATLAB routines for simulating turbo codes are available on the Internet⁷, with the proviso that they are only used for educational purposes. In exploring the performance of turbo codes several simulations were run on a PC. For the purposes of this simulation a punctured turbo code at rate $R=1/2$ was used. The data block length was 400 bits, and a MAP decoder was used in the simulation. The results shown at Figure 4 are the BER vs E_b/N_0 curves for different numbers of iterations from $n=1, 2, 5$ and 10 . A BPSK channel was assumed.



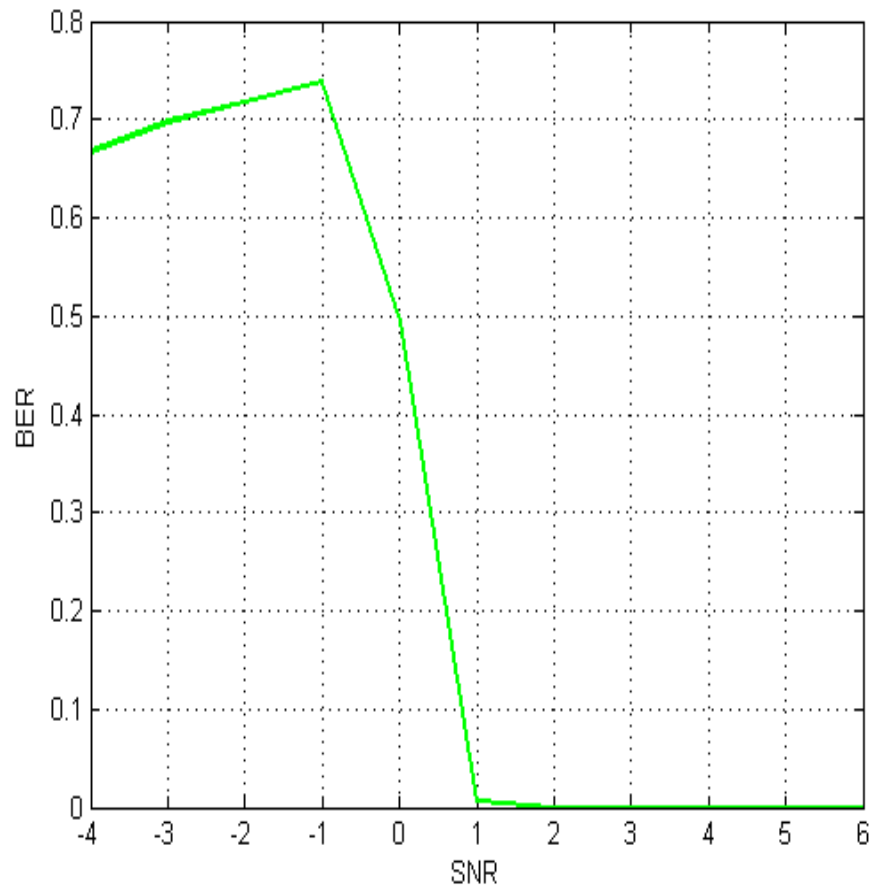


Figure 4 BER for Turbo Code

It can be seen that BERs of the order of 10^{-5} are achievable with $E_b/N_0 > 3$ dB with modest numbers of iterations. A typical coding gain of E_b/N_0 of 7dB, relative to an uncoded channel, was observed at a BER of 10^{-5} . Figure 4 infers that the BER should improve with each iteration, so a series of simulations were run to evaluate the improvement. It can be seen from Figure 5 that the first few iterations yield the most significant improvements in BER for any given E_b/N_0 . Thereafter the results appear to converge onto a BER for each value of E_b/N_0 . It is apparent that there is a tradeoff to be made between the number of iterations, processing power, and E_b/N_0 when seeking a given BER.

A final simulation was run to compare the performance of the MAP and SOVA decoders, particularly at low values of E_b/N_0 . The number of iterations was set at 5, since Figure 4 and

Figure 5 indicated that further iterations would yield marginal improvements. The results, shown at Figure 6, confirm that MAP is about 0.5 dB better than SOVA at low values of E_b/N_0 .

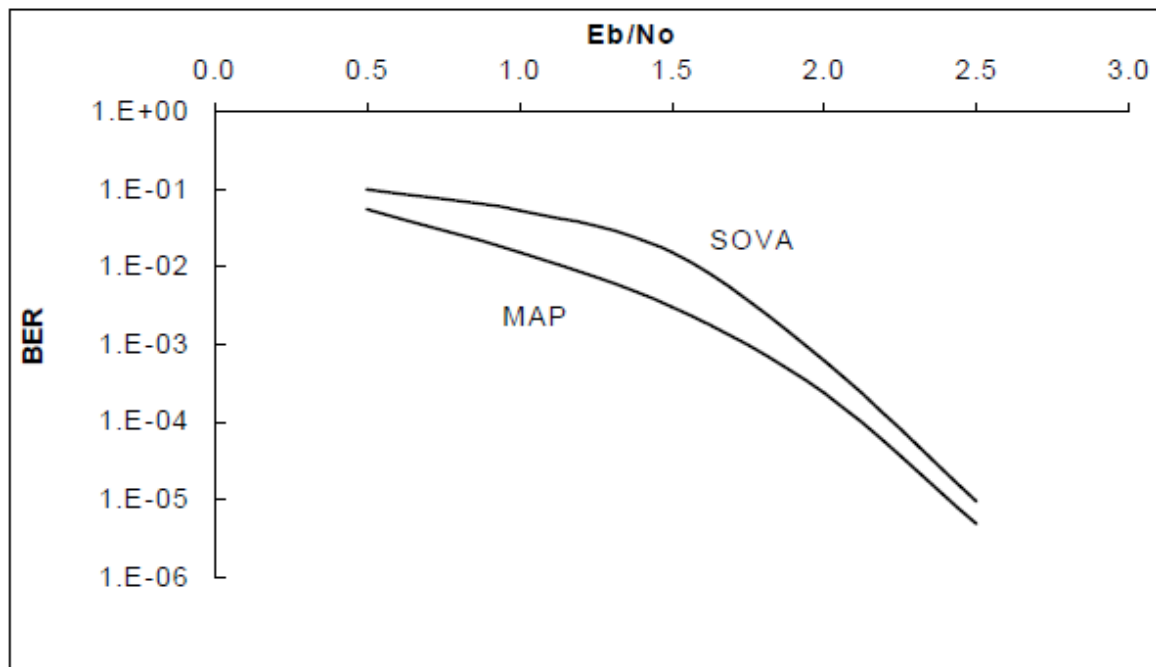


Figure 6 Comparison of MAP and SOVA at Low E_b/N_0

Turbo code rate	$\frac{1}{2}$	$\frac{2}{3}$	$\frac{3}{4}$	$\frac{2}{3}$
Modulation	16 QAM	8 PSK	16 QAM	64 QAM
Spectral Efficiency (bits/Hz)	2	2	3	4
Coding gain at 10^{-6} over uncoded modulation	6.0 dB	5.5 dB	7.8 dB	5.8 dB
Coding gain at 10^{-6} over 64 state TCM	2.4 dB	1.9 dB	2.6 dB	2.2 dB

Figure 7 Coding Gains for AWGN Channels

Code Composer Studio™ - Integrated Development Environment



Code Composer Studio is an integrated development environment (IDE) that supports TI's Microcontroller and Embedded Processors portfolio. Code Composer Studio comprises a suite of tools used to develop and debug embedded applications. It includes an optimizing C/C++ compiler, source code editor, project build environment, debugger, profiler, and many other features. The intuitive IDE provides a single user interface taking you through each step of the application development flow. Familiar tools and interfaces allow users to get started faster than ever before. Code Composer Studio combines the advantages of the Eclipse software framework with advanced embedded debug capabilities from TI resulting in a compelling feature-rich development environment for embedded developers.

Features by Platform

Find out more about the features available for a specific processor family:

- MSP430 Ultra Low Power MCUs
- C2000 Real-time MCUs
- SimpleLink Wireless MCUs
- TM4x MCUs MCUs
- TMS570 & RM4 Safety MCUs

- Sitara (Cortex A & ARM9) Processors
- Multicore DSP and ARM including KeyStone Processors
- For F24x/C24x devices
- For C3x/C4x DSPs

Code Composer Studio supports TI's broad portfolio of embedded processors. If you do not see a link for the family you are interested in above then select the one that is closest in terms of the processor cores used.

IMPLEMENTINS ON CCCS TOOL

We wrote C program based on the simulated MATLAB programs. These programs were tested on simulator of CCS and then implemented on DSK emulator. Same codes were implemented on INTEL atom processor.

DSK TMS320C6713

1.1 Key Features

The C6713 DSK is a low-cost standalone development platform that enables users to evaluate and develop applications for the TI C67xx DSP family. The DSK also serves as a hardware reference design for the TMS320C6713 DSP. Schematics, logic equations and application notes are available to ease hardware development and reduce time to market.

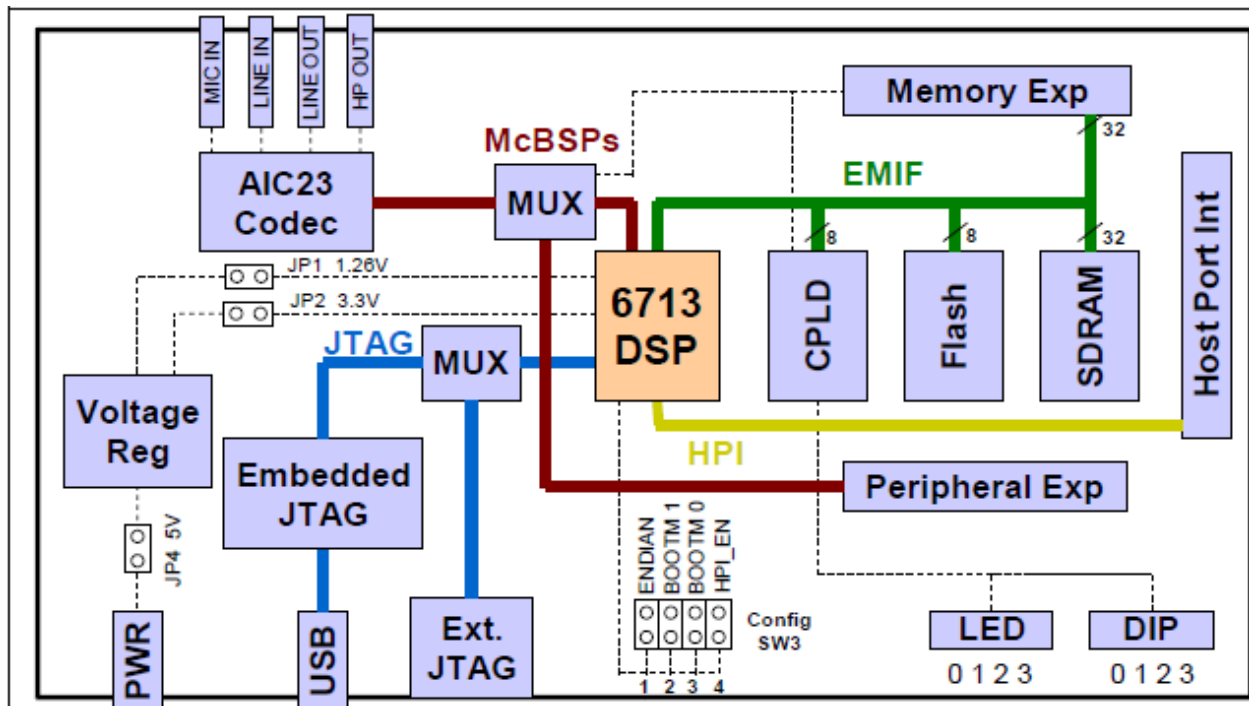


Fig 1 Block diagram of DSK TMS320C6713.

The DSK comes with a full compliment of on-board devices that suit a wide variety of application environments. Key features include:

- A Texas Instruments TMS320C6713 DSP operating at 225 MHz.
- An AIC23 stereo codec

- 16 Mbytes of synchronous DRAM
- 512 Kbytes of non-volatile Flash memory (256 Kbytes usable in default configuration)
- 4 user accessible LEDs and DIP switches
- Software board configuration through registers implemented in CPLD
- Configurable boot options
- Standard expansion connectors for daughter card use
- JTAG emulation through on-board JTAG emulator with USB host interface or external emulator
- Single voltage power supply (+5V)

1.2 Functional Overview of the TMS320C6713 DSK

The DSP on the 6713 DSK interfaces to on-board peripherals through a 32-bit wide EMIF (External Memory InterFace). The SDRAM, Flash and CPLD are all connected to the bus. EMIF signals are also connected daughter card expansion connectors which are used for third party add-in boards. The DSP interfaces to analog audio signals through an on-board AIC23 codec and four 3.5 mm audio jacks (microphone input, line input, line output, and headphone output).

The codec can select the microphone or the line input as the active input. The analog output is driven to both the line out (fixed gain) and headphone (adjustable gain) connectors. McBSP0 is used to send commands to the codec control interface while McBSP1 is used for digital audio data. McBSP0 and McBSP1 can be re-routed to the expansion connectors in software.

A programmable logic device called a CPLD is used to implement glue logic that ties the board components together. The CPLD has a register based user interface that lets the user configure the board by reading and writing to its registers. The DSK includes 4 LEDs and a 4 position DIP switch as a simple way to provide the user with interactive feedback. Both are accessed by reading and writing to the CPLD registers.

An included 5V external power supply is used to power the board. On-board switching voltage regulators provide the +1.26V DSP core voltage and +3.3V I/O supplies. The board is held in

reset until these supplies are within operating specifications. Code Composer communicates with the DSK through an embedded JTAG emulator with a USB host interface. The DSK can also be used with an external emulator through the external JTAG connector.

1.3 Basic Operation

The DSK is designed to work with TI's Code Composer Studio development environment and ships with a version specifically tailored to work with the board. Code Composer communicates with the board through the on-board JTAG emulator. To start, follow the instructions in the Quick Start Guide to install Code Composer. This process will install all of the necessary development tools, documentation and drivers.

After the install is complete, follow these steps to run Code Composer. The DSK must be fully connected to launch the DSK version of Code Composer.

- 1) Connect the included power supply to the DSK.
- 2) Connect the DSK to your PC with a standard USB cable (also included).
- 3) Launch Code Composer from its icon on your desktop.

1.4 Memory Map

The C67xx family of DSPs has a large byte addressable address space. Program code and data can be placed anywhere in the unified address space. Addresses are always 32-bits wide.

The memory map shows the address space of a generic 6713 processor on the left with specific details of how each region is used on the right. By default, the internal memory sits at the beginning of the address space. Portions of the internal memory can be reconfigured in software as L2 cache rather than fixed RAM. The EMIF has 4 separate addressable regions called chip enable spaces (CE0-CE3). The SDRAM occupies CE0 while the Flash and CPLD share CE1. CE2 and CE3 are generally reserved for daughtercards.

Address	C67x Family Memory Type	6713 DSK
0x00000000	Internal Memory	Internal Memory
0x00030000	Reserved Space or Peripheral Regs	Reserved or Peripheral
0x80000000	EMIF CE0	SDRAM
0x90000000	EMIF CE1	Flash
0xA0000000	EMIF CE2	CPLD
0xB0000000	EMIF CE3	Daughter Card

0x90080000

Fig 2 Memory Mapping of DSK TMS320C6713

1.5 Configuration Switch Settings

The DSK has 4 configuration switches that allows users to control the operational state of the DSP when it is released from reset. The configuration switch block is labeled SW3 on the DSK board, next to the reset switch. Configuration switch 1 controls the endianness of the DSP while switches 2 and 3 configure the boot mode that will be used when the DSP starts executing. Configuration switch 4 controls the on-chip multiplexing of HPI and McASP signals brought out to the HPI expansion connector. By default all switches are off which corresponds to EMIF boot (out of 8-bit Flash) in little endian mode and HPI signals on the HPI expansion connector.

Table 1: Configuration Switch Settings

Switch 1	Switch 2	Switch 3	Switch 4	Configuration Description
Off				Little endian (default)
On				Big endian
	Off	Off		EMIF boot from 8-bit Flash (default)
	Off	On		HPI/Emulation boot
	On	Off		32-bit EMIF boot
	On	On		16-bit EMIF boot
			Off	HPI enabled on HPI pins (default)
			On	McASP1 enabled on HPI pins

1.6 Power Supply

The DSK operates from a single +5V external power supply connected to the main power input (J5). Internally, the +5V input is converted into +1.26V and +3.3V using separate voltage regulators. The +1.26V supply is used for the DSP core while the +3.3V supply is used for the DSP's I/O buffers and all other chips on the board. The power connector is a 2.5mm barrel-type plug. There are three power test points on the DSK at JP1, JP2 and JP4. All I/O current passes through JP2 while all core current passes through JP1. All system current passes through JP4. Normally these jumpers are closed. To measure the current passing through remove the jumpers and connect the pins with a current measuring device such as a multimeter or current probe.

It is possible to provide the daughter card with +12V and -12V when the external power connector (J6) is used.



Over the past two decades, a major transition has occurred from analog modulation to digital modulation techniques in communication systems. Moreover, while communication systems were initially established as voice networks, they now have to accommodate computer data as well as multimedia content. And, as more and more users join the communication network the need for efficient use of available bandwidth in the RF spectrum becomes even more important. Especially given that every service provider has to pack as much data as possible – into the allotted RF bandwidth – to meet establishment expenses and recurring spectrum license fees. Digital modulation techniques provide more information carrying capacity, better quality communication, data security and RF spectrum sharing to accommodate more services when compared to analog modulation. All modern communication systems and gadgets today use various forms of digital modulation techniques, such as PSK, MSK, QAM as well as multiplexing techniques such as TDMA and CDMA, to pack more in the available RF bandwidth. Digital modulations are often expressed in terms of I (in-phase) and Q (quadrature) signals, and all modern digital communication systems and gadgets have three main blocks namely:

- **Data processing block**
- **ADC and DAC conversion block**
- **RF block**

All the blocks process the I and Q signals simultaneously, leading to the use of complex algebra and analysis in their design and implementation. This standardized and uniform approach in building digital communication systems or gadgets results in lower cost of development and manufacturing. The focus is also turned towards adding more features and bringing in more techniques within the available RF spectrum.

The Benchmark WiCOMM-T – the ultimate Wireless Digital Communication Training Platform – is the actual implementation of modern digital communication systems with direct interface to MATLAB through the Hi-Speed USB port of a PC.



Fig1: IF Loop back

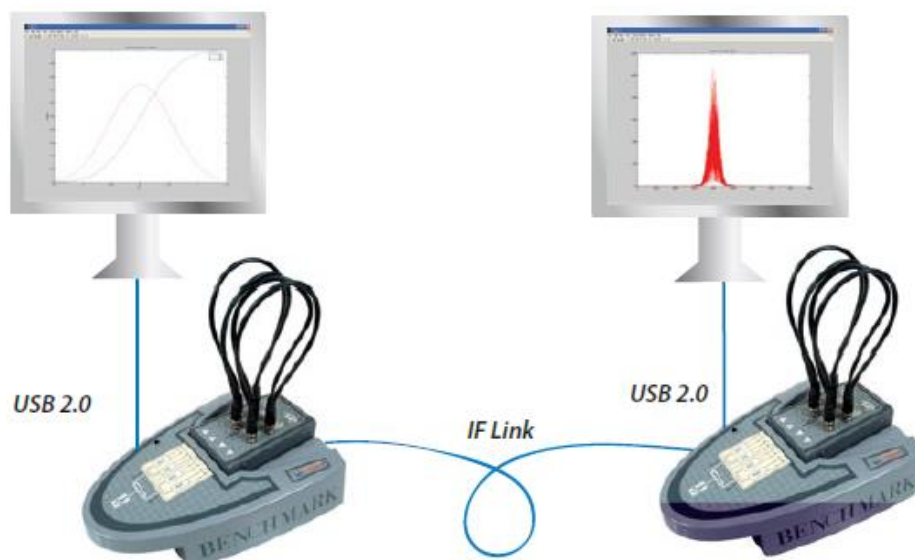


Fig 2: IF module

The WiCOMM-T is based on the curriculum developed in the Intel Wireless Laboratory of IIT Madras.

The WiCOMM-T provides maximum flexibility in learning complete digital communication system concepts, which includes digital modulation techniques, Baseband Equalization, Filtering concepts, and the basics of CDMA, GSM etc. MATLAB codes of all suggested experiment topics are available to users as reference. A MATLAB interface to the Platform also allows users to try out other topics on their own.

Features

- Typical implementation of modern communication systems
- Interface with MATLAB
- Gives the ability to generate required signal and pass it through the transmitter and receiver
- providing a real life wireless digital communication system
- Comprehensive manual – describes wide range of experiments
- Loop back options at Baseband and at IF

Intel® Atom™ Processor E6xx Series

1.0 Introduction



Fig 1 Intel atom processor

The Intel® Atom™ Processor E6xx Series is the next-generation Intel® architecture (IA) CPU for the small form factor ultra low power embedded segments based on a new architecture partitioning. The new architecture partitioning integrates the 3D graphics engine, memory controller and other blocks with the IA CPU core. Please refer to subsequent chapters for a detailed description of functionality.

The processor departs from the proprietary chipset interfaces used by other IA CPUs to an open-standard, industry-proven PCI Express* v1.0 interface. This allows it to be paired with customer-defined IOH, ASIC, FPGA and off-the-shelf discrete components. This provides utmost flexibility in IO solutions. This is important for deeply embedded applications, in which IOs differ from one application to another, unlike traditional PCl like applications. Figure 1 shows an example system block diagram.

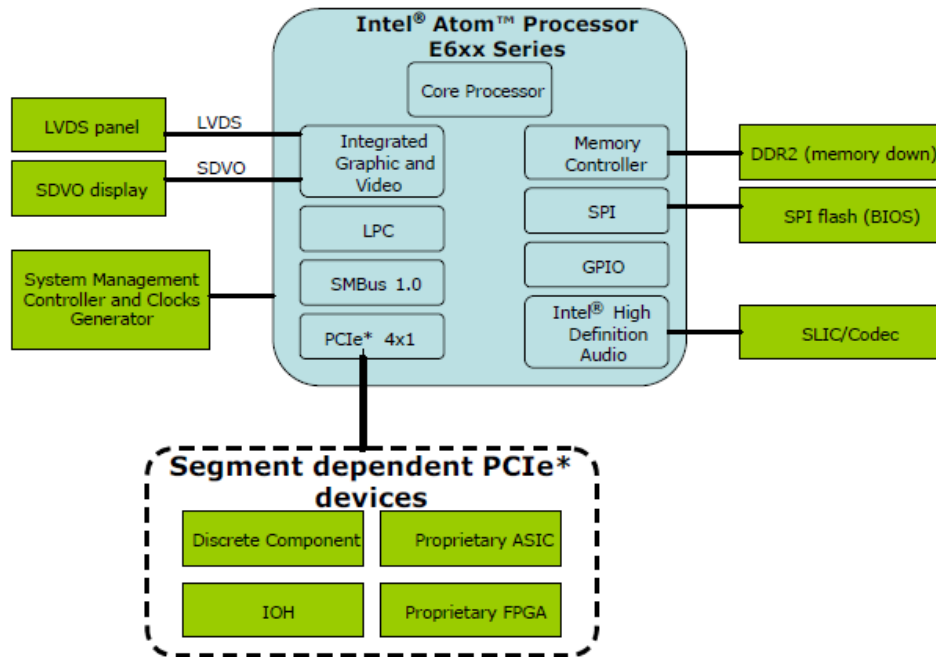


Fig 2: System Block Diagram Example

1.2 Components of the Intel® Atom™ Processor E6xx Series

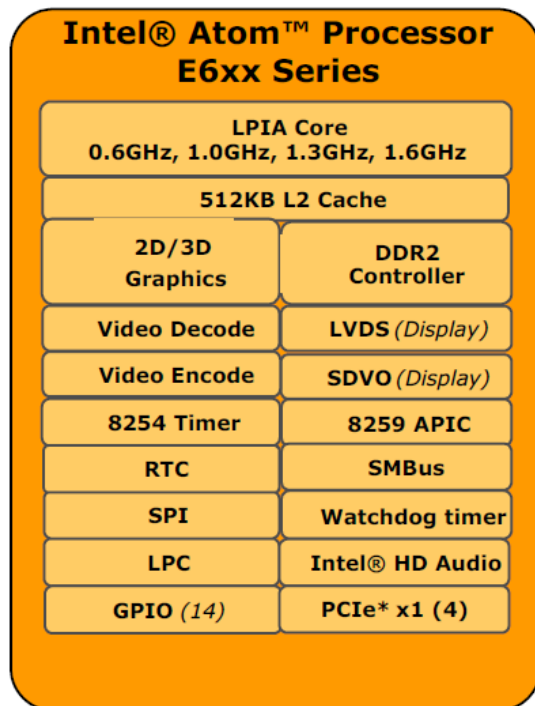


Fig 3 Components of Intel atom processor

1.2.1 Low-Power Intel® Architecture Core

- 600 MHz (Ultra Low Power SKU), 1.0 GHz (Entry SKU), 1.3 GHz (Mainstream SKU) and 1.6 GHz (Premium SKU)
- Macro-operation execution support
- 2-wide instruction decode and in-order execution
- On die, 32 kB 4-way L1 Instruction Cache and 24 kB 6-way L1 Data Cache
- On die, 512 kB, 8-way L2 cache
- L2 Dynamic Cache Sizing
- 32-bit physical address, 48-bit linear address size support
- Support for IA 32-bit architecture
- Supports Intel® Virtualization Technology^δ (Intel® VT-x^δ)
- Supports Intel® Hyper-Threading Technology^α — two threads
- Advanced power management features including Enhanced Intel SpeedStep® Technology^θ
- Deep Power Down Technology (C6)
- Intel® Streaming SIMD Extension 2 and 3 (Intel® SSE2 and Intel® SSE3) and Supplemental Streaming SIMD Extensions 3 (SSSE3) support

1.2.2 System Memory Controller

- Single-channel DDR2 memory controller
- 32-bit data bus
- Supports DDR2 800 MT/s data rates
- Supports 1 or 2 ranks
- Supports x8 or x16 DRAM chips
- One rank - two x16 or four x8 DRAM chips
- Two ranks - two x16 DRAM chips per rank, or four x8 DRAM chips per rank
- Supports up to 2 GB of extended memory
- Supports total memory size of 128 MB, 256 MB, 512 MB, 1 GB and 2 GB
- Supports 256 Mb, 512 Mb, 1 Gb and 2 Gb chip densities for the x8 DRAM
- Supports 512 Mb, 1 Gb and 2 Gb chip densities for the x16 DRAM
- Aggressive power management to reduce power consumption, including shallow

self-refresh and a new deep self-refresh support

- Proactive page closing policies to close unused pages
- Supports partial writes through data mask pins
- Supports only soldered-down DRAM configurations. The memory controller does not support SODIMM or any type of DIMMs.

1.2.3 Graphics

The Intel® Atom™ Processor E6xx Series provides integrated 2D/3D graphic engine that performs pixel shading and vertex shading within a single hardware accelerator. The processing of pixels is deferred until they are determined to be visible, which minimizes access to memory and improves render performance.

1.2.4 Video Decode

The Intel® Atom™ Processor E6xx Series supports MPEG2, MPEG4, VC1, WMV9, H.264 (main, baseline at L3 and high-profile level 4.0/4.1), and DivX*.

1.2.5 Video Encode

The Intel® Atom™ Processor E6xx Series supports MPEG4, H.264 (baseline at L3), and VGA.

1.2.6 Display Interfaces

The Intel® Atom™ Processor E6xx Series supports LVDS and Serial DVO display ports permitting simultaneous independent operation of two displays.

1.2.6.1 LVDS Interface

The Intel® Atom™ Processor E6xx Series supports a Low-Voltage Differential Signaling interface that allows the Graphics and Video adaptor to communicate directly to an onboard flat-panel display. The LVDS interface supports pixel color depths of 18 and 24 bits, maximum resolution up to 1280x768 @ 60 Hz. Minimum pixel clock is 19.75 MHz. Maximum pixel clock rate up to 80 MHz. The processor does provides LVDS backlight control related signal in order to support LVDS panel backlight adjustment. 1.2.6.2 Serial DVO (SDVO) Display Interface Digital display channel capable of driving SDVO adapters that provide interfaces to a variety of external display technologies (e.g., DVI, TV-Out, analog CRT). Maximum resolution up to

1280x1024 @ 85 Hz. Maximum pixel clock rate up to 160 MHz. SDVO lane reversal is not supported.

1.2.7 PCI Express*

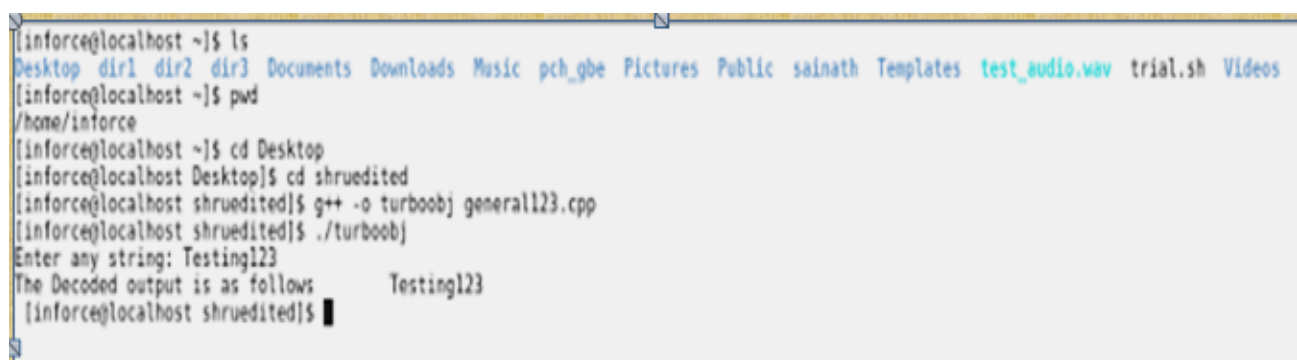
The Intel® Atom™ Processor E6xx Series has four x1 lane PCI Express* (PCIe*) root ports supporting the *PCI Express* Base Specification*, Rev. 1.0a. The processor does not support the “ganging” of PCIe* ports. The four x1 PCIe* ports operate as four independent PCIe* controllers. Each root port supports up to 2.5 Gb/s bandwidth in each direction per lane. The PCIe* ports may be used to attach discrete I/O components or a custom I/O Hub for increased I/O expansion.

1.2.8 LPC Interface

The Intel® Atom™ Processor E6xx Series implements an LPC interface as described in the LPC1.1 Specification. The LPC interface has three PCI-based clock outputs that may be provided to different I/O devices such as legacy I/O chip. The LPC_CLKOUT signals support a total of six loads (two loads per clock pair) with no external buffering.

1.3 Implementation

INTEL works on LINUX platform. We used certain LINUX commands to emulate the processor. The C program simulated on CCS Tool is implemented on INTEL Processor and the decoded result is shown below. We have given “Testing123” as input and we got the input decoded without any errors.



```
[inforce@localhost ~]$ ls
Desktop dir1 dir2 dir3 Documents Downloads Music pch_gbe Pictures Public sainath Templates test_audio.wav trial.sh Videos
[inforce@localhost ~]$ pwd
/home/inforce
[inforce@localhost ~]$ cd Desktop
[inforce@localhost Desktop]$ cd shrudited
[inforce@localhost shrudited]$ g++ -o turboobj general123.cpp
[inforce@localhost shrudited]$ ./turboobj
Enter any string: Testing123
The Decoded output is as follows      Testing123
[inforce@localhost shrudited]$
```

Fig 4 result obtained on Intel atom processor.

Practical applications using turbo code

Telecommunications:

- Turbo codes are used extensively in 3G and 4G mobile telephony standards; e.g., in HSPA, EV-DO and LTE.
- MediaFLO, terrestrial mobile television system from Qualcomm.
- The interaction channel of satellite communication systems, such as DVB-RCS^[3] and DVB-RCS2.
- New NASA missions such as Mars Reconnaissance Orbiter now use turbo codes, as an alternative to RS-Viterbi codes.
- Turbo coding such as block turbo coding and convolutional turbo coding are used in IEEE 802.16 (WiMAX), a wireless metropolitan network standard.

CONCLUSION

Turbo codes are a class of convolution code which exhibit the properties of large block codes through the use of recursive coders. Coder performance is heavily dependent on the design of the interleaver, which must ensure adequate weight for at least one of the codes. Soft decoders are used with turbo codes to allow the a posteriori probability to be passed between decoder iterations. The MAP decoder is generally preferred because it offers the same performance as SOVA at 0.5dB lower value of E_b/N_0 . This performance edge is achieved at the cost of increased complexity.

A half rate turbo coded BPSK channel can offer coding gains of 7dB over an uncoded channel at a BER of 10^{-5} . The coding gain depends on the number of iterations; typically 5 to 10 iterations generate most of the improvement. Turbo code performance has been simulated for a number of high order constellations, including 8PSK, 16 QAM and 64QAM, and the importance of code mapping within the constellation has been recognised. The reasons for the observed error floor are not yet fully understood, but indications are that it is linked in some way to interleaver design.

Research is now beginning to be directed towards applying turbo codes to resolve real communications problems. There are many potential applications for turbo codes, particularly in the field of radio communications. With suitable chipsets becoming available the first products are beginning to be marketed. The superior performance offered by turbo codes ensures that they have a good future in information systems.

Development

Manufacturers are looking seriously at the advantages of turbo codes over the well established Viterbi and Reed-Solomon (RS) codes. The initial emphasis has been on producing chipsets to allow manufacturers to implement turbo codes in their hardware. High-speed electronics has aided the development of chips with sufficient processing power to implement turbo decoders, currently at data rates of a few tens of Mbit/s. AHA and Comatlas have produced turbo code chipsets at 36 and 40 Mbit/s respectively, while Small World Comms have an FPGA for turbo decoding up to 90 Mbit/s.

These enabling technologies have helped to bring the first generation of turbo code based equipment to the marketplace. Comtech have launched a satellite modem that uses a rate $R = 3/4$ turbo code claiming significant bandwidth and BER improvements over modems using Viterbi and RS codes. Other major satellite modem manufacturers such as Comsat Labs are introducing turbo codes into their product. Alantro are developing turbo code firmware for such diverse roles as satellite links and hard disk drives.

The mobile phone industry is looking at turbo codes to provide error correction for third generation handsets. There have been many recent papers on the subject of implementing turbo codes in CDMA systems for UMTS. Turbo codes can operate at reduced power levels offering improved safety and extended battery life, both of which are important to the mobile phone user.

REFERENCES

- [1] Intuitive Guide to Principles of communications www.complextoreal.com.
- [2] Hybrid Log-MAP Algorithm for Turbo Decoding Over AWGN Channel by Li Li Lim and David Wee Gin Lim
- [3] 3GPP2/3GPP Coding Turbo code and Turbo Interleaver Summary by Jungsub Byun
- [4] Fundamentals of Turbo Codes by Bernard Sklar.
- [5] Turbo Coding Based High Performance Technique to Minimise Privacy and Safety Issues in RFID by M Vijaya Deepti, M N V S S kumar L Ganesh and S Deva Prasad.
- [6] Design of parallel concatenated convolutional codes by Sergio Benedetto.
- [7] L.R. Bahl and J. Cocke and F. Jelinek and J. Raviv, "Optimal Decoding of Linear Codes for Minimising Symbol Error Rate," *IEEE Transactions on Information Theory*, vol. 20, pp. 284–287, March 1974.
- [8] J. Hagenauer and P. Hoeher, "A viterbi algorithm with soft-decision outputs and its applications," in *IEEE Globecom*, pp. 1680–1686, 1989.

Appendix 1: Matlab codes

```
clear all
close all
clc

%Conversion of the input data into continuous stream of binary bits
data = 'turbo12345'; %Input data
ascii = double(data); %Conversion into decimal value
datalen = length(data)
imp = 5-mod(datalen,5);
if (mod(datalen,5) ~= 0)
    dist = datalen+imp;
else
    dist = datalen;
end
ascii = [ascii,zeros(1,imp)];
bin = zeros(dist,8);
for i = 1:dist
    a = ascii(i);
    k = 8;
    while (a>=1)
        bin(i,k) = mod(a,2); %Conversion into binary values
        a = floor(a/2);
        k = k-1;
    end
end
cont = reshape(bin,1,[]); %Reshape to get a single long array of
binary bits
len = length(cont);
encsig = zeros(0,0);

%Consider 40 bits at a time and encode the entire data

for i = 0:ceil(len/40)-1
    in = zeros(1,43);
    for j = 1:40
```

```

        in(j) = cont(40*i+j);
for encoding at a time
    end
    [v0,v1] = sysenc(in);
    [in2] = interleaver(in);
for the second encoder
    [v00,v11] = sysenc(in2);
    [enc] = concat(v0,v1,v11);
40 bits of input data
    encsig = [encsig,enc];
end
len1 = length(encsig);

%Modulate the encoded signal
[modsig] = modulation(encsig,len1);

%Noise generation by box Muller method
snr = 0.5;
noise = awgn(modsig,snr,'measured');

%Decoder stage
len2 = length(noise);
binout = zeros(0,0);
count = zeros(len2/129,4);
for i = 0:(len2/129)-1
    L = zeros(1,129);
    for j = 1:129
        L(j) = noise(129*i+j);
decoding at a time
    end
    [ru,rp1,rp2] = deconcat(L);
    la = zeros(1,43);
will be zeros
    [ru2] = interleaver(ru);
received sytematic input for the second decoder
    lc = 4*snr;

```

%Consider 40 binary bits
 %First encoder
 %Interleaving the input
 %Second encoder
 %Encoded signal for every
 %Final encoded signal
 %snr value
 %Consider 129 bits for
 %Initial intrinsic values
 %Interleaving the

```

    for b = 1:4                                %Performing required
number of iterations
        [lee1] = turbotest(ru,rp1,la,snr);      %First decoder
        [la] = interleaver(lee1);              %Interleaving the
extrinsic values of first decoder for obtaining intrinsic values of second
decoder
        [lee2] = turbotest(ru2,rp2,la,snr);    %Second decoder
        l = lc*ru2+la+lee2;                    %Calculating the output
        [L] = deinterleaver(l);                %Deinterleaving the
output to get the output in the desired order
        out = zeros(1,40);
        for k = 1:40                            %Simple logic to compute
the binary output
            if (L(k)<0)
                out(k) = 0;
            else
                out(k) = 1;
            end
        end
        for k=1:40                                %Calculating the number
of errors
            if out(k)~=cont(40*i+k)
                count(i+1,b) = count(i+1,b)+1;
            end
        end
        if count(i+1,b) == 0                    %If no errors are
present, then step out of the iteration loop
            break;
        end
        [la] = deinterleaver(lee2);            %Deinterleaving the
extrinsic values of second decoder for obtaining intrinsic values of first
decoder
    end
    binout = [binout,out];                      %Storing all the output
binary bits
end

%Conversion of the decoded binary bits back to the data

```

```

binfinal = reshape(binout,[],8);
binstr = num2str(binfinal);
asciiout = bin2dec(binstr);
dataout = char(asciiout);
reshape(dataout,1,[])
count;

```

concatenation function

```

function [out] = concat (in1,in2,in3)
for i=0:42
    out(3.*i+1)=in1(i+1); %combining the 3 outputs from the 2 encoder stages
    out(3.*i+2)=in2(i+1);
    out(3.*i+3)=in3(i+1);
end
end

```

Deconcatenation function

```

function [out1,out2,out3] = deconcat (in)
for i=0:42
    out1(i+1)= in(3.*i+1); %combining the 3 outputs from the 2 encoder stages
    out2(i+1)=in(3.*i+2);
    out3(i+1)=in(3.*i+3);
end
end

```

Interleaver function

```

function [I] = interleaver (u)
f1=3;
f2=10;
i=0:1:39;
x(i+1)= mod(((f1.*i)+(f2.*i.*i)),40);
for i = 0:39
    I(i+1)= u(x(i+1)+1);
end
I(41:1:43)=0;
End

```

Max function

```

function [z]=maxx(x,y)

```

```
z=max(x,y)+log(1+exp(-abs(x-y)));
```

```
end
```

Deinterleaver function

```
function [L] = deinterleaver (l)
```

```
f1=3; %interleaver input frequencies
```

```
f2=10; %interleaver input frequencies
```

```
i=0:1:39;
```

```
x(i+1)= mod(((f1.*i)+(f2.*i.*i)),40);
```

```
for i = 0:39
```

```
    L(x(i+1)+1)= l(i+1); %output of deinterleaver
```

```
end
```

```
end
```

Modulation Function

```
function [out] = modulation (in,l)
```

```
for i=1:l
```

```
    if in(i)==0
```

```
        out(i)=-1;
```

```
    elseif in(i)==1
```

```
        out(i)=1;
```

```
    end
```

```
end
```

```
end
```

```
function[a,b,c,d,e]=myfunc (u,s0,s1,s2)
```

```
a=u;
```

```
b=xor(s1,xor(u,s0));
```

```
c=xor(s1,xor(u,s2));
```

```
d=s0;
```

```
e=s1;
```

Encoder function

```
function [v0,v1] = sysenc (u)
```

```
s0(1) = 0;
```

```
s1(1) = 0;
```

```
s2(1) = 0;
```

```
for n=1:43
```

```
[v0(n),v1(n),s0(n+1),s1(n+1),s2(n+1)] = myfunc(u(n),s0(n),s1(n),s2(n));
```

```
    if n>40
```

```

    if s0(n) == 0
        if s1(n) == 0
            if s2(n) == 1
                v0(n) = 1;
            end
        elseif s1(n) == 1
            if s2(n) == 0
                v0(n) = 1;
            end
            if s2(n) == 1
                v0(n) = 0;
            end
        end
    elseif s0(n) == 1
        if s1(n) == 0
            if s2(n) == 0
                v0(n) = 0;
            end
            elseif s2(n) == 1
                v0(n) = 1;
            end
        elseif s1(n) == 1
            if s2(n) == 0
                v0(n) = 1;
            end
            elseif s2(n) == 1
                v0(n) = 0;
            end
        end
    end
end
end
end

```

Turbo Test function

```

function [leu]=turbotest(ruu,rp,lint,snr)
lc=4*snr;
alp(1,1)=0;
alp(2,1)=-1/2*lint(1)+lc/2*(-ruu(1)-rp(1));
alp(2,5)=1/2*lint(1)+lc/2*(ruu(1)+rp(1));

```

```

alp(3,1)=-1/2*lint(2)+lc/2*(-ruu(2)-rp(2))+alp(2,1);
alp(3,3)=-1/2*lint(2)+lc/2*(-ruu(2)+rp(2))+alp(2,5);
alp(3,5)=1/2*lint(2)+lc/2*(ruu(2)+rp(2))+alp(2,1);
alp(3,7)=1/2*lint(2)+lc/2*(ruu(2)-rp(2))+alp(2,5);

alp(4,1)=-1/2*lint(3)+lc/2*(-ruu(3)-rp(3))+alp(3,1);
alp(4,2)=1/2*lint(3)+lc/2*(ruu(3)-rp(3))+alp(3,3);
alp(4,3)=-1/2*lint(3)+lc/2*(-ruu(3)+rp(3))+alp(3,5);
alp(4,4)=1/2*lint(3)+lc/2*(ruu(3)+rp(3))+alp(3,7);
alp(4,5)=1/2*lint(3)+lc/2*(ruu(3)+rp(3))+alp(3,1);
alp(4,6)=-1/2*lint(3)+lc/2*(-ruu(3)+rp(3))+alp(3,3);
alp(4,7)=1/2*lint(3)+lc/2*(ruu(3)-rp(3))+alp(3,5);
alp(4,8)=-1/2*lint(3)+lc/2*(-ruu(3)-rp(3))+alp(3,7);
k=40;
for i=5:k
alp(i,1)=maxx(-1/2*lint(i-1)+lc/2*(-ruu(i-1)-rp(i-1))+alp(i-1,1),1/2*lint(i-1)+lc/2*(ruu(i-1)+rp(i-1))+alp(i-1,2));
alp(i,2)=maxx(1/2*lint(i-1)+lc/2*(ruu(i-1)-rp(i-1))+alp(i-1,3),-1/2*lint(i-1)+lc/2*(-ruu(i-1)+rp(i-1))+alp(i-1,4));
alp(i,3)=maxx(-1/2*lint(i-1)+lc/2*(-ruu(i-1)+rp(i-1))+alp(i-1,5),1/2*lint(i-1)+lc/2*(ruu(i-1)-rp(i-1))+alp(i-1,6));
alp(i,4)=maxx(1/2*lint(i-1)+lc/2*(ruu(i-1)+rp(i-1))+alp(i-1,7),-1/2*lint(i-1)+lc/2*(-ruu(i-1)-rp(i-1))+alp(i-1,8));
alp(i,5)=maxx(1/2*lint(i-1)+lc/2*(ruu(i-1)+rp(i-1))+alp(i-1,1),-1/2*lint(i-1)+lc/2*(-ruu(i-1)-rp(i-1))+alp(i-1,2));
alp(i,6)=maxx(-1/2*lint(i-1)+lc/2*(-ruu(i-1)+rp(i-1))+alp(i-1,3),1/2*lint(i-1)+lc/2*(ruu(i-1)-rp(i-1))+alp(i-1,4));
alp(i,7)=maxx(1/2*lint(i-1)+lc/2*(ruu(i-1)-rp(i-1))+alp(i-1,5),-1/2*lint(i-1)+lc/2*(-ruu(i-1)+rp(i-1))+alp(i-1,6));
alp(i,8)=maxx(-1/2*lint(i-1)+lc/2*(-ruu(i-1)-rp(i-1))+alp(i-1,7),1/2*lint(i-1)+lc/2*(ruu(i-1)+rp(i-1))+alp(i-1,8));
end
alp(k+1,1)=maxx(lc/2*(-ruu(k)-rp(k))+alp(k,1),lc/2*(ruu(k)+rp(k))+alp(k,2));
alp(k+1,2)=maxx(lc/2*(ruu(k)-rp(k))+alp(k,3),lc/2*(-ruu(k)+rp(k))+alp(k,4));
alp(k+1,3)=maxx(lc/2*(-ruu(k)+rp(k))+alp(k,5),lc/2*(ruu(k)-rp(k))+alp(k,6));
alp(k+1,4)=maxx(lc/2*(ruu(k)+rp(k))+alp(k,7),lc/2*(-ruu(k)-rp(k))+alp(k,8));

```



```

alp(k+2,1)=maxx(lc/2*(-ruu(k+1)-
rp(k+1))+alp(k+1,1),lc/2*(ruu(k+1)+rp(k+1))+alp(k+1,2));
alp(k+2,2)=maxx(lc/2*(ruu(k+1)-rp(k+1))+alp(k+1,3),lc/2*(-
ruu(k+1)+rp(k+1))+alp(k+1,4));

alp(k+3,1)=maxx(lc/2*(-ruu(k+2)-
rp(k+2))+alp(k+2,1),lc/2*(ruu(k+2)+rp(k+2))+alp(k+2,2));

bet(k+3,1)=0;

bet(k+2,1)=lc/2*(-ruu(k+2)-rp(k+2));
bet(k+2,2)=lc/2*(ruu(k+2)+rp(k+2));

bet(k+1,1)=lc/2*(-ruu(k+1)-rp(k+1))+bet(k+2,1);
bet(k+1,2)=lc/2*(ruu(k+1)+rp(k+1))+bet(k+2,1);
bet(k+1,3)=lc/2*(ruu(k+1)-rp(k+1))+bet(k+2,2);
bet(k+1,4)=lc/2*(-ruu(k+1)+rp(k+1))+bet(k+2,2);

bet(k,1)=lc/2*(-ruu(k)-rp(k))+bet(k+1,1);
bet(k,2)=lc/2*(ruu(k)+rp(k))+bet(k+1,1);
bet(k,3)=lc/2*(ruu(k)-rp(k))+bet(k+1,2);
bet(k,4)=lc/2*(-ruu(k)+rp(k))+bet(k+1,2);
bet(k,5)=lc/2*(-ruu(k)+rp(k))+bet(k+1,3);
bet(k,6)=lc/2*(ruu(k)-rp(k))+bet(k+1,3);
bet(k,7)=lc/2*(ruu(k)+rp(k))+bet(k+1,4);
bet(k,8)=lc/2*(-ruu(k)-rp(k))+bet(k+1,4);

for i=k:-1:4
bet(i,1)=maxx(-1/2*lint(i)+lc/2*(-ruu(i)-
rp(i))+bet(i+1,1),1/2*lint(i)+lc/2*(ruu(i)+rp(i))+bet(i+1,5));
bet(i,2)=maxx(-1/2*lint(i)+lc/2*(-ruu(i)-
rp(i))+bet(i+1,5),1/2*lint(i)+lc/2*(ruu(i)+rp(i))+bet(i+1,1));
bet(i,3)=maxx(-1/2*lint(i)+lc/2*(-
ruu(i)+rp(i))+bet(i+1,6),1/2*lint(i)+lc/2*(ruu(i)-rp(i))+bet(i+1,2));
bet(i,4)=maxx(-1/2*lint(i)+lc/2*(-
ruu(i)+rp(i))+bet(i+1,2),1/2*lint(i)+lc/2*(ruu(i)-rp(i))+bet(i+1,6));

```

```

bet(i,5)=maxx(-1/2*lint(i)+lc/2*(-
ruu(i)+rp(i))+bet(i+1,3),1/2*lint(i)+lc/2*(ruu(i)-rp(i))+bet(i+1,7));
bet(i,6)=maxx(-1/2*lint(i)+lc/2*(-
ruu(i)+rp(i))+bet(i+1,7),1/2*lint(i)+lc/2*(ruu(i)-rp(i))+bet(i+1,3));
bet(i,7)=maxx(-1/2*lint(i)+lc/2*(-ruu(i)-
rp(i))+bet(i+1,8),1/2*lint(i)+lc/2*(ruu(i)+rp(i))+bet(i+1,4));
bet(i,8)=maxx(-1/2*lint(i)+lc/2*(-ruu(i)-
rp(i))+bet(i+1,4),1/2*lint(i)+lc/2*(ruu(i)+rp(i))+bet(i+1,8));
end

```

```

bet(3,1)=maxx(-1/2*lint(3)+lc/2*(-ruu(3)-
rp(3))+bet(4,1),1/2*lint(3)+lc/2*(ruu(3)+rp(3))+bet(4,5));
bet(3,3)=maxx(-1/2*lint(3)+lc/2*(-
ruu(3)+rp(3))+bet(4,6),1/2*lint(3)+lc/2*(ruu(3)-rp(3))+bet(4,2));
bet(3,5)=maxx(-1/2*lint(3)+lc/2*(-
ruu(3)+rp(3))+bet(4,3),1/2*lint(3)+lc/2*(ruu(3)-rp(3))+bet(4,7));
bet(3,7)=maxx(-1/2*lint(3)+lc/2*(-ruu(3)-
rp(3))+bet(4,8),1/2*lint(3)+lc/2*(ruu(3)+rp(3))+bet(4,4));

```

```

bet(2,1)=maxx(-1/2*lint(2)+lc/2*(-ruu(2)-
rp(2))+bet(3,1),1/2*lint(2)+lc/2*(ruu(2)+rp(2))+bet(3,5));
bet(2,5)=maxx(-1/2*lint(2)+lc/2*(-
ruu(2)+rp(2))+bet(3,3),1/2*lint(2)+lc/2*(ruu(2)-rp(2))+bet(3,7));

```

```

leu(1)=rp(1)+bet(2,5)-bet(2,1);
leu(2)=maxx(alp(2,1)+bet(3,5)+lc/2*rp(2),alp(2,5)+bet(3,7)-lc/2*rp(2))-
maxx(alp(2,1)+bet(3,1)-lc/2*rp(2),alp(2,5)+bet(3,3)+lc/2*rp(2));
leu(3)=maxx(maxx(alp(3,1)+bet(4,5)+lc/2*rp(3),alp(3,3)+bet(4,2)-
lc/2*rp(3)),maxx(alp(3,5)+bet(4,7)-lc/2*rp(3),alp(3,7)+bet(4,4)+lc/2*rp(3)))-
maxx(maxx(alp(3,1)+bet(4,1)-
lc/2*rp(3),alp(3,3)+bet(4,6)+lc/2*rp(3)),maxx(alp(3,5)+bet(4,3)+lc/2*rp(3),alp(3,7)+bet(4,8)-lc/2*rp(3)));

```

```

for i=4:k

```

```

    p1=alp(i,1)+bet(i+1,5)+lc/2*rp(i);
    p2=alp(i,2)+bet(i+1,1)+lc/2*rp(i);
    q1=alp(i,3)+bet(i+1,2)-lc/2*rp(i);
    q2=alp(i,4)+bet(i+1,6)-lc/2*rp(i);

```

```

r1=alp(i,5)+bet(i+1,7)-lc/2*rp(i);
r2=alp(i,6)+bet(i+1,3)-lc/2*rp(i);
s1=alp(i,7)+bet(i+1,4)+lc/2*rp(i);
s2=alp(i,8)+bet(i+1,8)+lc/2*rp(i);
t1=alp(i,1)+bet(i+1,1)-lc/2*rp(i);
t2=alp(i,2)+bet(i+1,5)-lc/2*rp(i);
u1=alp(i,3)+bet(i+1,6)+lc/2*rp(i);
u2=alp(i,4)+bet(i+1,2)+lc/2*rp(i);
v1=alp(i,5)+bet(i+1,3)+lc/2*rp(i);
v2=alp(i,6)+bet(i+1,7)+lc/2*rp(i);
w1=alp(i,7)+bet(i+1,8)-lc/2*rp(i);
w2=alp(i,8)+bet(i+1,4)-lc/2*rp(i);
leu(i)=maxx((maxx(maxx(p1,p2),maxx(q1,q2))), (maxx(maxx(r1,r2),maxx(s1,s2))))-
maxx((maxx(maxx(t1,t2),maxx(u1,u2))), (maxx(maxx(v1,v2),maxx(w1,w2)))));
end
leu(k+1)=maxx(maxx(alp(k+1,2)+bet(k+2,1)+lc/2*rp(k+1), alp(k+1,3)+bet(k+2,2)-
lc/2*rp(k+1)),maxx(alp(k+1,6)+bet(k+2,3)-
lc/2*rp(k+1), alp(k+1,7)+bet(k+2,4)+lc/2*rp(k+1))) -
maxx(maxx(alp(k+1,1)+bet(k+2,1)-
lc/2*rp(k+1), alp(k+1,4)+bet(k+2,2)+lc/2*rp(k+1)),maxx(alp(k+1,5)+bet(k+2,3)+lc/2*rp(k+1), alp(k+1,8)+bet(k+2,4)-lc/2*rp(k+1)));
leu(k+2)=maxx(alp(k+2,2)+bet(k+3,1)+lc/2*rp(k+2), alp(k+2,3)+bet(k+3,2)-
lc/2*rp(k+2))-maxx(alp(k+2,1)+bet(k+3,1)-
lc/2*rp(k+2), alp(k+2,4)+bet(k+3,2)+lc/2*rp(k+2));
leu(k+3)=rp(k+3)+alp(k+3,2)-alp(k+3,1);
end

```

Appendix 2: CCS tool codes in C language

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
float maxx (float a , float b);
void interleaver(float *in,float *out);
void deinterleaver(float *in,float *out);
void deconcat (float *in,float *out1,float *out2,float *out3);
void modulation(int *in,int *out);
void turbotest(float *leu,float *ruu,float *rp,float *lint,float SNR);
void encoder(int *output);
void boxmuller(int *modout,float *modout11,int snr);
void main(void)
{
    char y[10];
    int i,j,encinput[258],modout[258],dec[5][8],k=0;
    int out=0;
    int finalout[5],d,f;
    float
lc,snr,lee1[43],lee2[43],l[43],L[43],ru[43],ru2[43],rp1[43],rp2[43],noiseoutput[258],
noiseoutput1[129],la[43];

    encoder(encinput);

    modulation(encinput,modout);
    snr = 6;
    lc = 4*snr;
    boxmuller(modout,noiseoutput,snr);
    for (d=0;d<2;d++)
    {
        for(f=0;f<129;f++)
            noiseoutput1[f]=noiseoutput[129*d+f];//modout[129*d+f];//noiseoutput[129*d+f];
        deconcat (noiseoutput1,ru,rp1,rp2);
    }
}
```

```

    for (i=0;i<43;i++)
        la[i] = 0;
    interleaver (ru,ru2);

    for (j=0;j<4;j++)
    {
        turbotest (&lee1[0],&ru[0],&rp1[0],&la[0],snr);
        interleaver (lee1,la);
        turbotest (lee2,ru2,rp2,la,snr);
        for (i=0;i<43;i++)
            l[i] = (lc*ru2[i])+la[i]+lee2[i];
        deinterleaver (l,L);
        deinterleaver (lee2,la);
        for (i=0;i<40;i++)
        {
            if (L[i]<=0)
                L[i]=0;
            else
                L[i]=1;
        }
    }
    k = 0;
    for(i=0;i<5;i++)
        for(j=0;j<8;j++)
            dec[i][j] = L[k++];
    for(i=0;i<5;i++)
    {
        k = 0;
        out = 0;
        for (j=7;j>=0;j--)
            out= out + (dec[i][j] * pow(2,k++));
        finalout[i] = out;
        y[5*d+i] = finalout[i];
    }
}

```

```

        for (i=0;i<10;i++)
            printf("%c",y[i]);
    }

```

Deconcat function

```

#include<stdio.h>
#include<stdlib.h>
void deconcat (float *in,float *out1,float *out2,float *out3)
{
    int i;
    for (i=0;i<43;i++)
    {
        out1[i]= in[3*i];
        out2[i]=in[3*i+1];
        out3[i]=in[3*i+2];
    }
}

```

Deinterleaver function

```

#include<stdio.h>
#include<stdlib.h>
void deinterleaver(float *in,float *out)
{
    int x[40],i,f1=3,f2=10;
    for (i=0;i<40;i++)
    {
        x[i] = (((f1*i)+(f2*i*i))%40);
        out[x[i]]= in[i];
    }
    out[40]=in[40];
    out[41]=in[41];
    out[42]=in[42];
}

```

Encoder function

```

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
void enc(int *in, int *sys, int *par);
void encoder(int *output)
{
    unsigned int input[43],inlen,h,f1,f2,inter[43],f,b;
    unsigned int v0[43],v1[43],v00[43],v11[43],x[40],encodedoutput[129];
    char str[10];
    int final[80];
    int no[10],n[10][8];
    int i=0,j,k,m,a=0;

    printf("Enter any string: ");
    scanf("%[^\\n]s",str);

    j=strlen(str);
    for (i=0;i<j;i++)
    for (k=0;k<8;k++)
        n[i][k]=0;

        for(i=0;i<j;i++)

{
    no[i] = str[i];
    m=no[i];
    k=7;
    while (m>0)
    {
        n[i][k] = m%2;
        m = m/2;
        k=k-1;
    }
    n[i][0]=0;
}
}

```

```

k=0;
for(i=0;i<j;i++)
    for(a=0;a<8;a++)
        final[k++] = n[i][a];

if ((8*j)%40!=0)
{
    b = (8*j) + (40-((8*j)%40));
}
else
    b = 8*j;

for (k=8*j;k<b;k++)
{
    final[k]=0;
}

for(f=0;f<(b/40);f++)
{
    for(i=0;i<40;i++)
        input[i]=final[40*f+i];
    input[40]=input[41]=input[42]=0;
    inlen=40;

    h=43;
    f1=3;
    f2=10;
    for (i=0;i<inlen;i++)
        x[i] = (((f1*i)+(f2*i*i))%inlen);
    for( i = 0;i<inlen;i++)
        inter[i]= input[x[i]];
    inter[40]=inter[41]=inter[42]=0;

    enc(&input[0],&v0[0],&v1[0]);
    enc(inter,v00,v11);

    for( i=0;i<h;i++)

```



```

    {
        encodedoutput[3*i]=v0[i];
        encodedoutput[3*i+1]=v1[i];
        encodedoutput[3*i+2]=v11[i];
    }
    for(i=0;i<129;i++)
        output[f*129+i]=encodedoutput[i];
}
}

```

```

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
void enc(int *in, int *sys, int *par)
{
    unsigned int c[44], d[44], e[44],n=0;
    c[0]=0;
    d[0]=0;
    e[0]=0;
    for(n=0;n<43;n++)
    {
        par[n]= d[n]^in[n]^c[n];
        c[n+1]=d[n]^in[n]^e[n];
        d[n+1]=c[n];
        e[n+1]=d[n];

        if(n>39)
        {
            if(c[n]==0)
            {
                if(d[n]==0)
                {
                    if(e[n]==1)
                        in[n]=1;
                }
            }
        }
    }
}

```

```

        else if(d[n]==1)
        {
            if (e[n]==0)
                in[n]=1;
            else if (e[n]==1)
                in[n]=0;
        }
    }
    else if (c[n]==1)
    {
        if (d[n]==0)
        {
            if (e[n]==0)
                in[n]=0;
            else if (e[n]==1)
                in[n]=1;
        }
        else if (d[n]==1)
        {
            if (e[n]==0)
                in[n]=1;
            else if (e[n]==1)
                in[n]=0;
        }
    }
    sys[n]=in[n];
}
}

```

Interleaver function

```

#include<stdio.h>
#include<stdlib.h>
void interleaver(float *in,float *out)
{
    int x[40],i,f1=3,f2=10;

```

```

    for (i=0;i<40;i++)
    {
        x[i] = (((f1*i)+(f2*i*i))%40);
        out[i]= in[x[i]];
    }
    out[40]=in[40];
    out[41]=in[41];
    out[42]=in[42];
}

```

Max function

```

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
float maxx (float x, float y)
{
    float w,b,c,z,d;
    if(x>y)
        z= x-y;
    else
        z= y-x;

    b = exp(-z);
    w = log(1+b);
    c = x+w;
    d = y+w;
    if(x>y)
        return(c);
    else
        return(d);
}

```

Modulation function

```

#include<stdio.h>

```

```

#include<stdlib.h>
void modulation(int *in,int *out)
{
    int i;
    for (i=0;i<258;i++)
    {
        if (in[i]==0)
            out[i]=-1;
        else
            out[i]=1;
    }
}

```

Noise function

```

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>

void boxmuller(int *modout, float *modout11,int snr)
{
    int j;
    float gaussian_noise[258],rn=0,x1=0,x2=0;
    double noise_amp, randnum,noise=0,RATE =0.73;

    noise_amp = sqrt(2.0*RATE*pow(10.0,(snr/10.0)));
    //AWGN -> BOX-MULLER METHOD
    for(j=0;j<258;j++)
    {
        do
        {
            randnum = (double) rand()/(RAND_MAX*.125);
            x1 = 2*randnum - 1;
            randnum = (double) rand()/(RAND_MAX*.125);
            x2 = 2*randnum - 1;
            rn = x1*x1 + x2*x2;
        }
    }
}

```

```

    }
    while(rn >= 1);
    noise = x2 * sqrt( (-2.0*log(rn))/(rn) );
    noise = noise/noise_amp;
    gaussian_noise[j]=noise;
    if(gaussian_noise[j]>.75)
        gaussian_noise[j]=.25;
    if(gaussian_noise[j]<-0.75)
        gaussian_noise[j]=.25;
    modout11[j]= modout[j]+gaussian_noise[j];
}
}

```

Turbo test function

```

#include<stdio.h>
#include<stdlib.h>
float maxx (float a , float b);
void turbotest(float *leu,float *ruu,float *rp,float *lint,float SNR)
{
    float p1,p2,q1,q2,r1,r2,s1,s2,t1,t2,u1,u2,v1,v2,w1,w2;
    float alp[43][8], bet[43][8],lc;
    int k,i;

    alp[0][0]=0;
    alp[1][0]=-0.5*lint[0]+0.5*(-ruu[0]-rp[0]);
    alp[1][4]=0.5*lint[0]+0.5*(ruu[0]+rp[0]);

    alp[2][0]=-0.5*lint[1]+0.5*(-ruu[1]-rp[1])+alp[1][0];
    alp[2][2]=-0.5*lint[1]+0.5*(-ruu[1]+rp[1])+alp[1][4];
    alp[2][4]=0.5*lint[1]+0.5*(ruu[1]+rp[1])+alp[1][0];
    alp[2][6]=0.5*lint[1]+0.5*(ruu[1]-rp[1])+alp[1][4];

    alp[3][0]=-0.5*lint[2]+0.5*(-ruu[2]-rp[2])+alp[2][0];
    alp[3][1]=0.5*lint[2]+0.5*(ruu[2]-rp[2])+alp[2][2];
    alp[3][2]=-0.5*lint[2]+0.5*(-ruu[2]+rp[2])+alp[2][4];
    alp[3][3]=0.5*lint[2]+0.5*(ruu[2]+rp[2])+alp[2][6];
}

```

```

alp[3][4]=0.5*lint[2]+0.5*(ruu[2]+rp[2])+alp[2][0];
alp[3][5]=-0.5*lint[2]+0.5*(-ruu[2]+rp[2])+alp[2][2];
alp[3][6]=0.5*lint[2]+0.5*(ruu[2]-rp[2])+alp[2][4];
alp[3][7]=-0.5*lint[2]+0.5*(-ruu[2]-rp[2])+alp[2][6];

k=40;

for(i=4;i<k;i++)
{
    alp[i][0]=maxx(-0.5*lint[i-1]+0.5*(-ruu[i-1]-rp[i-1])+alp[i-1][0],0.5*lint[i-1]+0.5*(ruu[i-1]+rp[i-1])+alp[i-1][1]);
    alp[i][1]=maxx(0.5*lint[i-1]+0.5*(ruu[i-1]-rp[i-1])+alp[i-1][2],-0.5*lint[i-1]+0.5*(-ruu[i-1]+rp[i-1])+alp[i-1][3]);
    alp[i][2]=maxx(-0.5*lint[i-1]+0.5*(-ruu[i-1]+rp[i-1])+alp[i-1][4],0.5*lint[i-1]+0.5*(ruu[i-1]-rp[i-1])+alp[i-1][5]);
    alp[i][3]=maxx(0.5*lint[i-1]+0.5*(ruu[i-1]+rp[i-1])+alp[i-1][6],-0.5*lint[i-1]+0.5*(-ruu[i-1]-rp[i-1])+alp[i-1][7]);
    alp[i][4]=maxx(0.5*lint[i-1]+0.5*(ruu[i-1]+rp[i-1])+alp[i-1][0],-0.5*lint[i-1]+0.5*(-ruu[i-1]-rp[i-1])+alp[i-1][1]);
    alp[i][5]=maxx(-0.5*lint[i-1]+0.5*(-ruu[i-1]+rp[i-1])+alp[i-1][2],0.5*lint[i-1]+0.5*(ruu[i-1]-rp[i-1])+alp[i-1][3]);
    alp[i][6]=maxx(0.5*lint[i-1]+0.5*(ruu[i-1]-rp[i-1])+alp[i-1][4],-0.5*lint[i-1]+0.5*(-ruu[i-1]+rp[i-1])+alp[i-1][5]);
    alp[i][7]=maxx(-0.5*lint[i-1]+0.5*(-ruu[i-1]-rp[i-1])+alp[i-1][6],0.5*lint[i-1]+0.5*(ruu[i-1]+rp[i-1])+alp[i-1][7]);
}

alp[k][0]=maxx(-0.5*lint[k-1]+0.5*(-ruu[k-1]-rp[k-1])+alp[k-1][0],0.5*lint[k-1]+0.5*(ruu[k-1]+rp[k-1])+alp[k-1][1]);
alp[k][1]=maxx(0.5*lint[k-1]+0.5*(ruu[k-1]-rp[k-1])+alp[k-1][2],-0.5*lint[k-1]+0.5*(-ruu[k-1]+rp[k-1])+alp[k-1][3]);
alp[k][2]=maxx(-0.5*lint[k-1]+0.5*(-ruu[k-1]+rp[k-1])+alp[k-1][4],0.5*lint[k-1]+0.5*(ruu[k-1]-rp[k-1])+alp[k-1][5]);
alp[k][3]=maxx(0.5*lint[k-1]+0.5*(ruu[k-1]+rp[k-1])+alp[k-1][6],-0.5*lint[k-1]+0.5*(-ruu[k-1]-rp[k-1])+alp[k-1][7]);

```

```

    alp[k+1][0]=maxx(-0.5*lint[k]+0.5*(-ruu[k]-
rp[k])+alp[k][0],0.5*lint[k]+0.5*(ruu[k]+rp[k])+alp[k][1]);
    alp[k+1][1]=maxx(0.5*lint[k]+0.5*(ruu[k]-rp[k])+alp[k][2],-0.5*lint[k]+0.5*(-
ruu[k]+rp[k])+alp[k][3]);

    alp[k+2][0]=maxx(-0.5*lint[k+1]+0.5*(-ruu[k+1]-
rp[k+1])+alp[k+1][0],0.5*lint[k+1]+0.5*(ruu[k+1]+rp[k+1])+alp[k+1][1]);

    bet[k+2][0]=0;

    bet[k+1][0]=-0.5*lint[k+1]+0.5*(-ruu[k+1]-rp[k+1]);
    bet[k+1][1]=0.5*lint[k+1]+0.5*(ruu[k+1]+rp[k+1]);

    bet[k][0]=-0.5*lint[k]+0.5*(-ruu[k]-rp[k])+bet[k+1][0];
    bet[k][1]=0.5*lint[k]+0.5*(ruu[k]+rp[k])+bet[k+1][0];
    bet[k][2]=0.5*lint[k]+0.5*(ruu[k]-rp[k])+bet[k+1][1];
    bet[k][3]=-0.5*lint[k]+0.5*(-ruu[k]+rp[k])+bet[k+1][1];

    bet[k-1][0]=-0.5*lint[k-1]+0.5*(-ruu[k-1]-rp[k-1])+bet[k][0];
    bet[k-1][1]=0.5*lint[k-1]+0.5*(ruu[k-1]+rp[k-1])+bet[k][0];
    bet[k-1][2]=0.5*lint[k-1]+0.5*(ruu[k-1]-rp[k-1])+bet[k][1];
    bet[k-1][3]=-0.5*lint[k-1]+0.5*(-ruu[k-1]+rp[k-1])+bet[k][1];
    bet[k-1][4]=-0.5*lint[k-1]+0.5*(-ruu[k-1]+rp[k-1])+bet[k][2];
    bet[k-1][5]=0.5*lint[k-1]+0.5*(ruu[k-1]-rp[k-1])+bet[k][2];
    bet[k-1][6]=0.5*lint[k-1]+0.5*(ruu[k-1]+rp[k-1])+bet[k][3];
    bet[k-1][7]=-0.5*lint[k-1]+0.5*(-ruu[k-1]-rp[k-1])+bet[k][3];

    for(i=k-2;i>=3;i--)
    {
        bet[i][0]=maxx(-0.5*lint[i]+0.5*(-ruu[i]-
rp[i])+bet[i+1][0],0.5*lint[i]+0.5*(ruu[i]+rp[i])+bet[i+1][4]);
        bet[i][1]=maxx(-0.5*lint[i]+0.5*(-ruu[i]-
rp[i])+bet[i+1][4],0.5*lint[i]+0.5*(ruu[i]+rp[i])+bet[i+1][0]);
        bet[i][2]=maxx(-0.5*lint[i]+0.5*(-
ruu[i]+rp[i])+bet[i+1][5],0.5*lint[i]+0.5*(ruu[i]-rp[i])+bet[i+1][1]);
        bet[i][3]=maxx(-0.5*lint[i]+0.5*(-
ruu[i]+rp[i])+bet[i+1][1],0.5*lint[i]+0.5*(ruu[i]-rp[i])+bet[i+1][5]);

```

```

        bet[i][4]=maxx(-0.5*lint[i]+0.5*(-
ruu[i]+rp[i])+bet[i+1][2],0.5*lint[i]+0.5*(ruu[i]-rp[i])+bet[i+1][6]);
        bet[i][5]=maxx(-0.5*lint[i]+0.5*(-
ruu[i]+rp[i])+bet[i+1][6],0.5*lint[i]+0.5*(ruu[i]-rp[i])+bet[i+1][2]);
        bet[i][6]=maxx(-0.5*lint[i]+0.5*(-ruu[i]-
rp[i])+bet[i+1][7],0.5*lint[i]+0.5*(ruu[i]+rp[i])+bet[i+1][3]);
        bet[i][7]=maxx(-0.5*lint[i]+0.5*(-ruu[i]-
rp[i])+bet[i+1][3],0.5*lint[i]+0.5*(ruu[i]+rp[i])+bet[i+1][7]);
    }

    bet[2][0]=maxx(-0.5*lint[2]+0.5*(-ruu[2]-
rp[2])+bet[3][0],0.5*lint[2]+0.5*(ruu[2]+rp[2])+bet[3][4]);
    bet[2][2]=maxx(-0.5*lint[2]+0.5*(-
ruu[2]+rp[2])+bet[3][5],0.5*lint[2]+0.5*(ruu[2]-rp[2])+bet[3][1]);
    bet[2][4]=maxx(-0.5*lint[2]+0.5*(-
ruu[2]+rp[2])+bet[3][2],0.5*lint[2]+0.5*(ruu[2]-rp[2])+bet[3][6]);
    bet[2][6]=maxx(-0.5*lint[2]+0.5*(-ruu[2]-
rp[2])+bet[3][7],0.5*lint[2]+0.5*(ruu[2]+rp[2])+bet[3][3]);

    bet[1][0]=maxx(-0.5*lint[1]+0.5*(-ruu[1]-
rp[1])+bet[2][0],0.5*lint[1]+0.5*(ruu[1]+rp[1])+bet[2][4]);
    bet[1][4]=maxx(-0.5*lint[1]+0.5*(-
ruu[1]+rp[1])+bet[2][2],0.5*lint[1]+0.5*(ruu[1]-rp[1])+bet[2][6]);

    lc=4*(SNR);

    leu[0]=rp[0]+bet[1][4]-bet[1][0];
    leu[1]=maxx(alp[1][0]+bet[2][4]+lc*0.5*rp[1],alp[1][4]+bet[2][6]-
lc*0.5*rp[1])-maxx(alp[1][0]+bet[2][1]-
lc*0.5*rp[1],alp[1][4]+bet[2][2]+lc*0.5*rp[1]);
    leu[2]=maxx(maxx(alp[2][0]+bet[3][4]+lc*0.5*rp[2],alp[2][2]+bet[3][1]-
lc*0.5*rp[2]),maxx(alp[2][4]+bet[3][6]-
lc*0.5*rp[2],alp[2][6]+bet[3][3]+lc*0.5*rp[2]))-maxx(maxx(alp[2][1]+bet[3][1]-
lc*0.5*rp[2],alp[2][2]+bet[3][5]+lc*0.5*rp[2]),maxx(alp[2][4]+bet[3][2]+lc*0.5*rp[2],
alp[2][6]+bet[3][7]-lc*0.5*rp[2]));
    for (i=3;i<k;i++)
    {

```



```

p1=alp[i][0]+bet[i+1][4]+lc*0.5*rp[i];
p2=alp[i][1]+bet[i+1][0]+lc*0.5*rp[i];
q1=alp[i][2]+bet[i+1][1]-lc*0.5*rp[i];
q2=alp[i][3]+bet[i+1][5]-lc*0.5*rp[i];
r1=alp[i][4]+bet[i+1][6]-lc*0.5*rp[i];
r2=alp[i][5]+bet[i+1][2]-lc*0.5*rp[i];
s1=alp[i][6]+bet[i+1][3]+lc*0.5*rp[i];
s2=alp[i][7]+bet[i+1][7]+lc*0.5*rp[i];
t1=alp[i][0]+bet[i+1][0]-lc*0.5*rp[i];
t2=alp[i][1]+bet[i+1][4]-lc*0.5*rp[i];
u1=alp[i][2]+bet[i+1][5]+lc*0.5*rp[i];
u2=alp[i][3]+bet[i+1][1]+lc*0.5*rp[i];
v1=alp[i][4]+bet[i+1][2]+lc*0.5*rp[i];
v2=alp[i][5]+bet[i+1][6]+lc*0.5*rp[i];
w1=alp[i][6]+bet[i+1][7]-lc*0.5*rp[i];
w2=alp[i][7]+bet[i+1][3]-lc*0.5*rp[i];

leu[i]=maxx((maxx(maxx(p1,p2),maxx(q1,q2))),(maxx(maxx(r1,r2),maxx(s1,s2))))-
maxx((maxx(maxx(t1,t2),maxx(u1,u2))),(maxx(maxx(v1,v2),maxx(w1,w2)))));
}

leu[k]=maxx(maxx(alp[k][1]+bet[k+1][0]+lc*0.5*rp[k],alp[k][2]+bet[k+1][1]-
lc*0.5*rp[k]),maxx(alp[k][5]+bet[k+1][2]-
lc*0.5*rp[k],alp[k][6]+bet[k+1][3]+lc*0.5*rp[k]))-maxx(maxx(alp[k][0]+bet[k+1][1]-
lc*0.5*rp[k],alp[k][3]+bet[k+1][1]+lc*0.5*rp[k+1]),maxx(alp[k][4]+bet[k+1][2]+lc*0.5*
rp[k],alp[k][7]+bet[k+1][3]-lc*0.5*rp[k])));

leu[k+1]=maxx(alp[k+1][2]+bet[k+2][0]+lc*0.5*rp[k+1],alp[k+1][2]+bet[k+2][1]-
lc*0.5*rp[k+1])-maxx(alp[k+1][0]+bet[k+2][0]-
lc*0.5*rp[k+1],alp[k+1][3]+bet[k+2][1]+lc*0.5*rp[k+1]);

leu[k+2]=rp[k+2]+alp[k+2][1]-alp[k+2][0];
}

```

