Convolutional Codes

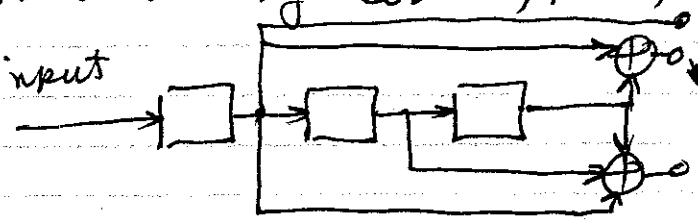
Convolutional Codes are the linear sub-class of Trellis Codes. A Convolutional encoder consists in kK bit shift register and a few Exclusive OR gates. The shift register consists of K stages each with k bits in them. After receiving k input bits and those bits (together with the bits already in the shift register n bits are generated by XORing output of selected elements of the shift register. The rate of the code is

$$R = \frac{k}{n}$$

and K is called the constraint length of the code. The number of state the encoder (and, subsequently, the code) has is:

$$|S| = 2^{(K-1)k}$$

for a binary code, i.e., $k=1$, $|S| = 2^{K-1}$.



$K=3, k=1, n=3$

A code is specified in terms of its generators. For, example for the above $\frac{1}{3}$ code, we have

$$g_1 = [1\ 0\ 0]$$

$$g_2 = [1\ 0\ 1]$$

and $$g_3 = [1\ 1\ 1]$$

These are each an impulse (actually pulse) function.

The outputs are:

$$c^{(1)} = u * g_1$$

$$c^{(2)} = u * g_2$$

$$c^{(3)} = u * g_3$$

where $*$ represents convolution.

We can represent the input stream as

$$u(D) = \sum_{i=0}^{\infty} u_i D_i$$

and the generators will give the following transfer functions: →

$$c^{(1)}(D) = u(D)\, g_1(D) \qquad g_1(D) = 1$$

$$c^{(2)}(D) = u(D)\, g_2(D) \qquad g_2(D) = 1 + D^2$$

$$c^{(3)}(D) = u(D)\, g_3(D) \qquad g_3(D) = 1 + D + D^2$$

The overall output $\underline{c}$ will have the transform

$$C(D) = C^{(1)}(D^3) + D\,C^{(2)}(D^3) + D^2 C^{(3)}(D^3).$$

As an example: assume we pass $\underline{u} = (1\,0\,0\,1\,1\,1)$ through the rate $\frac{1}{3}$, $K = 3$, code:

$$u(D) = 1 + D^3 + D^4 + D^5$$

$$C^{(1)}(D) = 1 + D^3 + D^4 + D^5$$

$$C^{(2)}(D) = (1 + D^3 + D^4 + D^5)(1 + D) = 1 + D^2 + D^3 + D^4 + D^6 + D^7$$

$$C^{(3)}(D) = (1 + D^3 + D^4 + D^5)(1 + D + D^2) = 1 + D + D^2 + D^3 + D^5 + D^7$$

$$C(D) = C^{(1)}(D^3) + D\,C^{(2)}(D^3) + D^2 C^{(3)}(D^3)$$

$$= 1 + D + D^2 + D^5 + D^7 + D^8 + D^9 + D^{10} + D^{11} + D^{12} + D^{13} + D^{15} + D^{17} + D^{19} + D^{22} + D^{23}$$

or

$$\underline{c} = (1\,1\,1\,0\,0\,1\,0\,1\,1\,1\,1\,1\,1\,1\,0\,1\,0\,1\,0\,1\,0\,0\,1\,1)$$

$$g_1 = [100]$$

The second function generator is connected to stages 1 and 3. Hence

$$g_2 = [101]$$

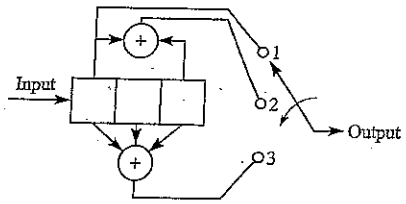


FIGURE 8.1–2
$K = 3, k = 1, n = 3$ convolutional encoder.

$$g_3 = [111]$$

$$\begin{aligned} \boldsymbol{c}^{(1)} &= \boldsymbol{u} \star \boldsymbol{g}_1 \\ \boldsymbol{c}^{(2)} &= \boldsymbol{u} \star \boldsymbol{g}_2 \\ \boldsymbol{c}^{(3)} &= \boldsymbol{u} \star \boldsymbol{g}_3 \end{aligned} \tag{8.1–1}$$

where $\star$ denotes the convolution operation. The corresponding code sequence $\boldsymbol{c}$ is the result of interleaving $\boldsymbol{c}^{(1)}$, $\boldsymbol{c}^{(2)}$, and $\boldsymbol{c}^{(3)}$ as

$$\boldsymbol{c} = \left(c_1^{(1)}, c_1^{(2)}, c_1^{(3)}, c_2^{(1)}, c_2^{(2)}, c_2^{(3)}, \ldots\right) \tag{8.1–2}$$

The convolutional operation is equivalent to multiplication in the transform domain. We define the D transform[†] of $\boldsymbol{u}$ as

$$u(D) = \sum_{i=0}^{\infty} u_i D^i \tag{8.1–3}$$

and the transfer function for the three impulse responses $\boldsymbol{g}_1$, $\boldsymbol{g}_2$, and $\boldsymbol{g}_3$ as

$$\begin{aligned} g_1(D) &= 1 \\ g_2(D) &= 1 + D^2 \\ g_3(D) &= 1 + D + D^2 \end{aligned} \tag{8.1–4}$$

The output transforms are then given by

$$\begin{aligned} c^{(1)}(D) &= u(D)g_1(D) \\ c^{(2)}(D) &= u(D)g_2(D) \\ c^{(3)}(D) &= u(D)g_3(D) \end{aligned} \tag{8.1–5}$$

and the transform of the encoder output $\boldsymbol{c}$ is given by

$$c(D) = c^{(1)}(D^3) + Dc^{(2)}(D^3) + D^2c^{(3)}(D^3) \tag{8.1–6}$$

EXAMPLE 8.1–1. Let the sequence $\boldsymbol{u} = (100111)$ be the input sequence to the convolutional encoder shown in Figure 8.1–2. We have

$$u(D) = 1 + D^3 + D^4 + D^5$$

and

$$c^{(1)}(D) = (1 + D^3 + D^4 + D^5)(1) = 1 + D^3 + D^4 + D^5$$

$$c^{(2)}(D) = (1 + D^3 + D^4 + D^5)(1 + D^2) = 1 + D^2 + D^3 + D^4 + D^6 + D^7$$

$$c^{(3)}(D) = (1 + D^3 + D^4 + D^5)(1 + D + D^2) = 1 + D + D^2 + D^3 + D^5 + D^7$$

and

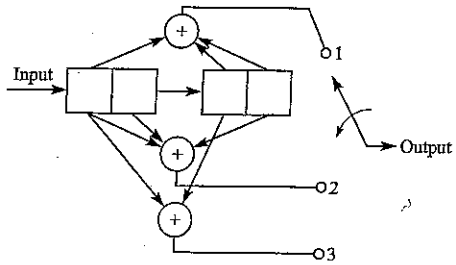
$$\begin{aligned} c(D) &= c^{(1)}(D^3) + Dc^{(2)}(D^3) + D^2c^{(3)}(D^3) \\ &= 1 + D + D^2 + D^5 + D^7 + D^8 + D^9 + D^{10} + D^{11} + D^{12} + D^{13} + D^{15} \\ &\quad + D^{17} + D^{19} + D^{22} + D^{23} \end{aligned}$$

corresponding to the code sequence

$$\boldsymbol{c} = (11100101111111010101010011)$$

Now consider $K=2$, $\frac{2}{3}$ Code

if we use the realization with multiplexed inputs

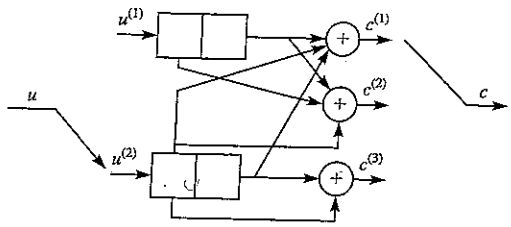


we have

$$g_1 = [1\ 0\ 1\ 1] \qquad g_2 = [1\ 1\ 0\ 1]$$

$$g_3 = [1\ 0\ 1\ 0]$$

However, we can also use:



Then we have:

$$g_1^{(1)} = [0\ 1] \qquad g_2^{(1)} = [1\ 1]$$
$$g_1^{(2)} = [1\ 1] \qquad g_2^{(2)} = [1\ 0]$$
$$g_1^{(3)} = [0\ 0] \qquad g_2^{(3)} = [1\ 1]$$

and the transfer functions are

$$g_1^{(1)}(D) = D \qquad g_2^{(1)}(D) = 1 + D$$
$$g_1^{(2)}(D) = 1 + D \qquad g_2^{(2)}(D) = 1$$
$$g_1^{(3)}(D) = 0 \qquad g_2^{(3)}(D) = 1 + D$$

From the transfer functions and the D transform of the input sequences we obtain the D transform of the three output sequences as

$$c^{(1)}(D) = u^{(1)}(D)g_1^{(1)}(D) + u^{(2)}(D)g_2^{(1)}(D)$$
$$c^{(2)}(D) = u^{(1)}(D)g_1^{(2)}(D) + u^{(2)}(D)g_2^{(2)}(D)$$
$$c^{(3)}(D) = u^{(1)}(D)g_1^{(3)}(D) + u^{(2)}(D)g_2^{(3)}(D)$$

and finally

$$c(D) = c^{(1)}(D^3) + Dc^{(2)}(D^3) + D^2c^{(3)}(D^3)$$

Equation 8.1–9 can be written in a more compact way by defining

$$\boldsymbol{u}(D) = [u^{(1)}(D) \quad u^{(2)}(D)]$$

and

$$\boldsymbol{G}(D) = \begin{bmatrix} g_1^{(1)}(D) & g_1^{(2)}(D) & g_1^{(3)}(D) \\ g_2^{(1)}(D) & g_2^{(2)}(D) & g_2^{(3)}(D) \end{bmatrix}$$

By these definitions Equation 8.1–9 can be written as

$$\boldsymbol{c}(D) = \boldsymbol{u}(D)\boldsymbol{G}(D)$$

where

$$\boldsymbol{c}(D) = [c^{(1)}(D) \quad c^{(2)}(D) \quad c^{(3)}(D)]$$

So for the $K=2$, $k=2$, $n=3$, we have

$$G(D) = \begin{bmatrix} D & 1+D & 0 \\ 1+D & 1 & 1+D \end{bmatrix}$$

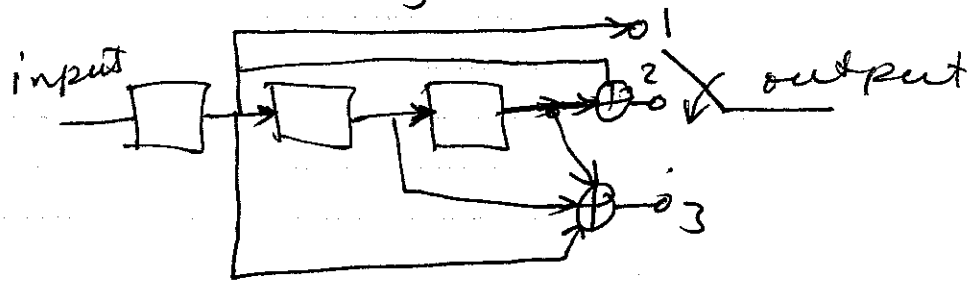
For the $K=3$, $k=1$, $n=3$, we have:

$$G(D) = [1 \quad D^2+1 \quad D^2+D+1]$$

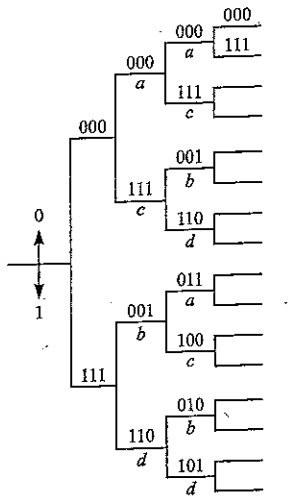
Representation of Convolution codes

- Tree
- Trellis
- State-Diagram.

Take $K=3$, $\frac{1}{3}$ Code

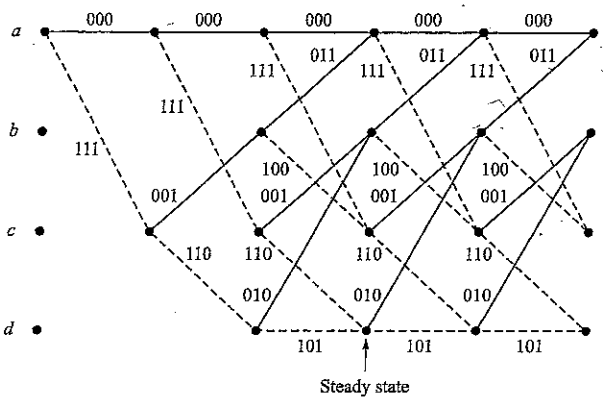


We can represent the operation of this encoder using a tree

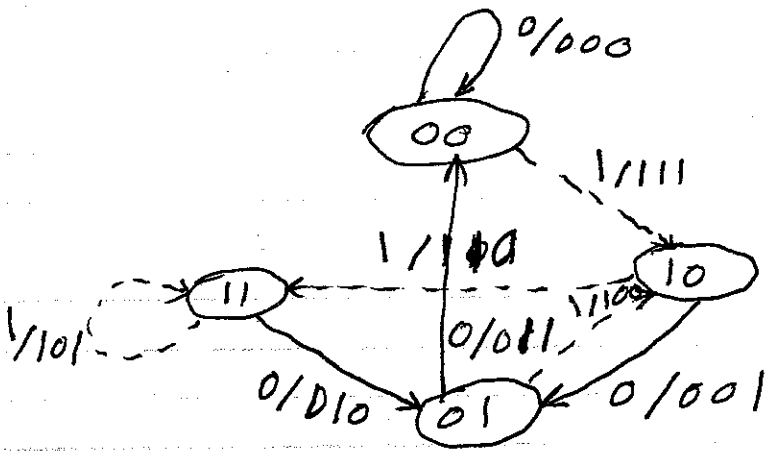


However, we observe that while the tree can span into an infinite number of (terminal) nodes, the number of state ~~~~ is

four (4) and the possibilities for the output sequence are 8. So, a more suitable data structure a trelli's can be used:



yet another way to visualize the change of states is to use a state diagram



A convolutional code, as seen from the above, is a finite state machine. Its operation is explained by two functions $f_c(\cdot,\cdot)$ and $f_s(\cdot,\cdot)$

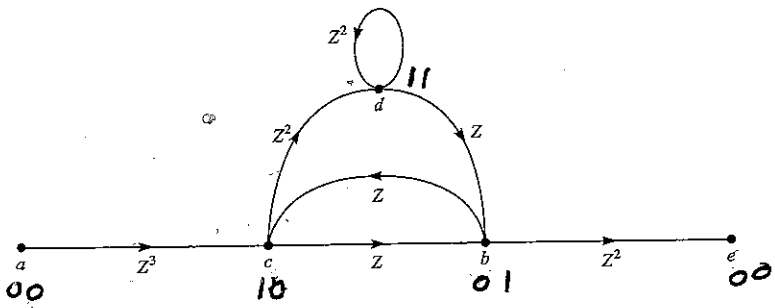
$$c_i = f_c(u_i, \sigma_{i-1})$$

$$\sigma_i = f_s(u_i, \sigma_{i-1})$$

$\sigma_i \in \sum$ set of states

Transfer function of a convolutional code:

Draw a graph with $2^{K-1}+1$ nodes: one per state plus another one repeating the first state. Connect the states if there is transition between them and label each connecting chord with Z^i where i is the weight of the output corresponding to the transition.



We can write the equations:

$$X_c = Z^3 X_a + Z X_b$$

$$X_b = Z X_c + Z X_d$$

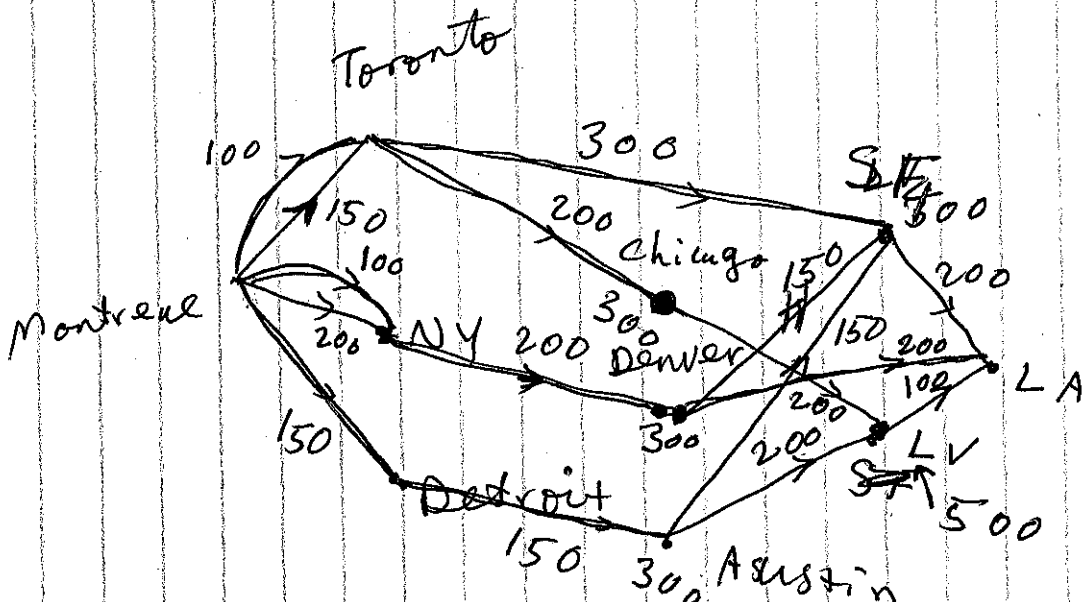
$$X_d = Z^2 X_c + Z^2 X_d$$

$$X_e = Z^2 X_b$$

Then we find

$$T(Z) = \frac{X_e}{X_a} = \frac{Z^6}{1 - 2Z^2}$$

$$= Z^6 + 2Z^8 + 4Z^{10} + 8Z^{12} + \cdots = \sum_{d=6}^{\infty} a_d Z^d$$



Montreal - NY - Denver - LA

Maximum-Likelihood decoding:

Assume, we have received a stream of channel outputs $\underline{r} = (r_{11}, r_{12}, \ldots, r_{jm}, \ldots, r_{Nk})$ and we would like to decide which codeword (which path through the trellis) has been chosen by the transmitter.

If transmitter had (at the output of encoder and input of modulator) codeword $c^{(i)}$ where i ranges from 1 to 2^{kN}, it will send

$$\sqrt{E}\,(2c_{jm}^{(i)} - 1) \quad \text{where} \quad c_{jm}^{(i)} \in \{0, 1\}$$

So, the jm-th component of $\underline{r}$ is

$$r_{jm} = \sqrt{E}\,(2c_{jm}^{(i)} - 1) + n_{jm} \qquad \begin{matrix} j = 1, \ldots, N \\ m = 1, \ldots, k \end{matrix}$$

where n_{jm} is the noise component.

In ML-decoding, we decide in favour of i-th path when

$$p(\underline{r} \mid \underline{c}^{(i)}) = \prod_{j=1}^{N} \prod_{m=1}^{k} p(r_{jm} \mid c_{jm}^{(i)})$$

is larger than $p(\underline{r} \mid \underline{c}^{(i')})$ for all $i' \neq i$.

Taking logarithm, we need to maximize

$$\log p(\underline{r} \mid \underline{c}) = \sum_{j=1}^{N} \log p(r_j \mid c_j)$$

for a Gaussian channel,

$$p(r_{jm} \mid c_{jm}^{(i)}) = \frac{1}{\sqrt{2\pi}\,\sigma} \exp\left[-\frac{1}{2\sigma^2}\left[r_{jm} - \sqrt{E}\,(2c_{jm}^{(i)} - 1)\right]^2\right]$$

So in this case, we need to minimize:

$$\sum_j \|r_j - c_j\|^2$$

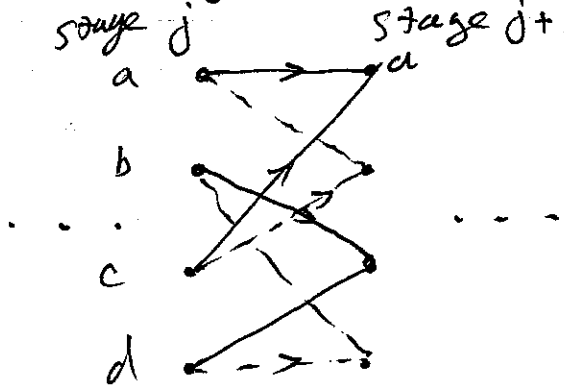
In the case of hard-decision decoding, i.e., deciding on the bit at the output of the channel first and then doing decoding, we minimize

$$\sum_j d_H(y_j, c_j).$$

We can use the Viterbi algorithm to find the path with the minimum path metric: $\sum_j \|r_j - c_j\|^2$ or $\sum_j d_H(y_j - c_j)$.

The advantage of the Viterbi algorithm is that at each stage, we only need to keep $2^{k(K-1)}$ metrics, i.e., one for each state.

Assume, you have ~~four~~ states ($K=3$).



If you have, for ~~each~~ each state, at stage j, a metric that has the minimum up to that point, that is the one that results in the minimum path for the $(j+1)$-st stage for those ~~no~~ state they ~~do~~ lead to. minimum

for example the metric for state a in stage $j+1$, is the minimum of

$$PM_a(j) + \|r_j - c_{aa}\|^2$$

and

$$PM_c(j) + \|r_j - c_{ca}\|^2 .$$

So, at any stage the minimum metric calculated up to that point of the "~~father~~" ~~nodes~~ "parent"

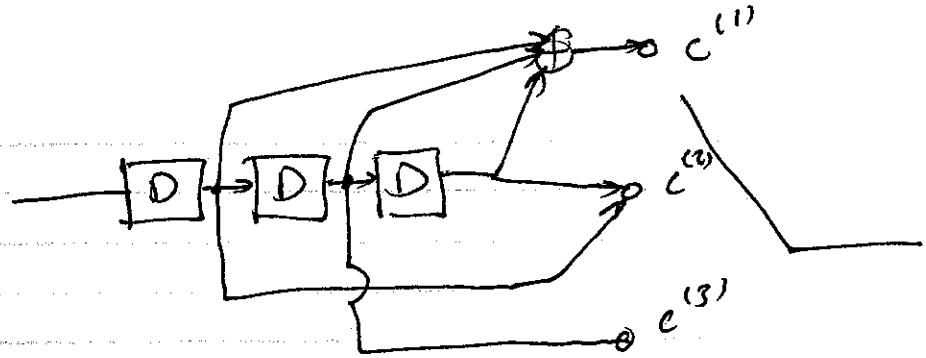
nodes of each node are added to the additional distance leading from each parent to the node in the next stage and the minimum is selected.

There are other ML-decoding schemes such as stack algorithm. These schemes are sequence decoder, i.e., they find the best noisy sequence of bits given a series of received signals.

Later we discuss MAP algorithm, in particular the one called BCJR that finds the best bit u_i at any given time i based on the observation of the whole sequence of noisy symbols.

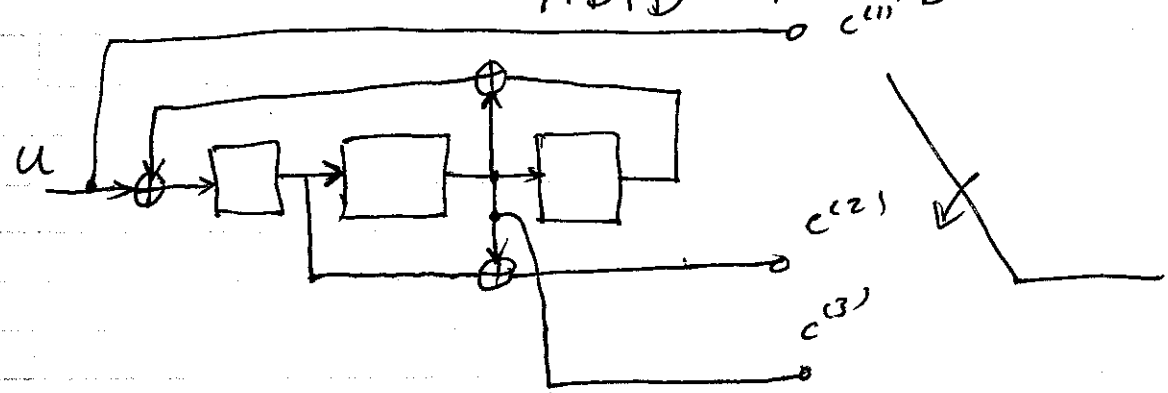
Recursive and Non-recursive codes:

An coder whose outputs are generated using feed-forward connections is non-recursive. But if we have a feedback, we get a recursive code. For example the code $G(D) = [1+D+D^2, 1+D, 1]$ is non-recursive.



But if we divide each transfer function g_i by, say $g_0(D)$, we get,

$$G'(D) = \left[1, \frac{1+D}{1+D+D^2}, \frac{D}{1+D+D^2}\right]$$



which is recursive. It has infinite impulse response (IIR).

This is a systematic recursive code.

These Recursive Systematic Convolutional Code (RSCC) are the building blocks of the Turbo Codes.

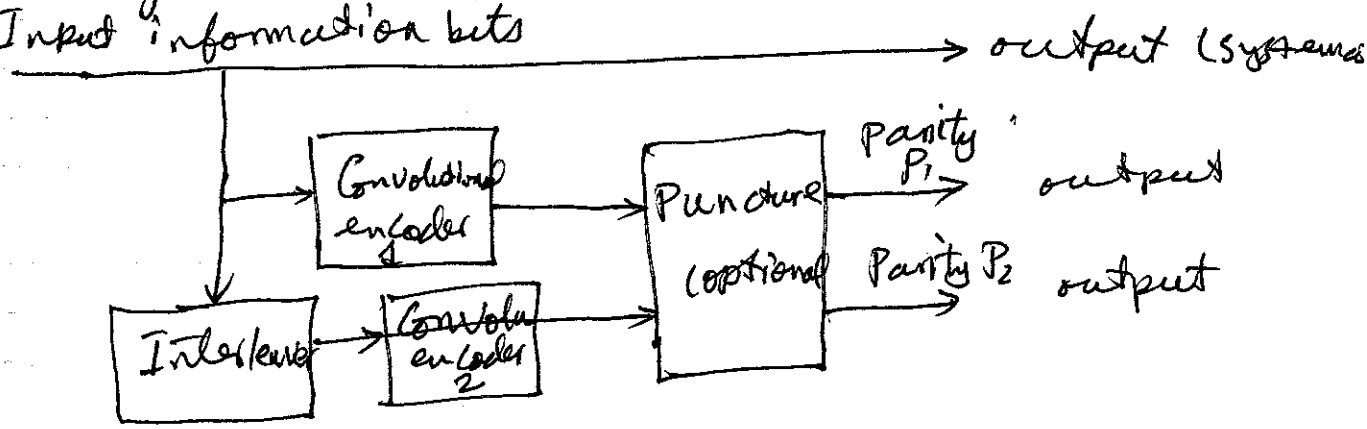
Rate-Compatible Punctured Convolutional Codes Constructed from Rate 1/3, $K = 4$ Code with $P = 8$ $R_c = P/(P + M)$, $M = 1, 2, 4, 6, 8, 10, 12, 14$

Rate	Puncturing Matrix P
$\frac{1}{3}$	$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$
$\frac{4}{11}$	$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$
$\frac{2}{5}$	$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$
$\frac{4}{9}$	$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$
$\frac{1}{2}$	$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$
$\frac{4}{7}$	$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$
$\frac{4}{6}$	$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$
$\frac{4}{5}$	$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$
$\frac{8}{9}$	$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$

Turbo Codes

Turbo Codes were originally formed as a Parallel Concatenation of Convolutional Codes (PCCCs). Of course, later serial concatenation as well as block codes concatenated either serially or in parallel were use.

A typical PCCC, Turbo Code looks like



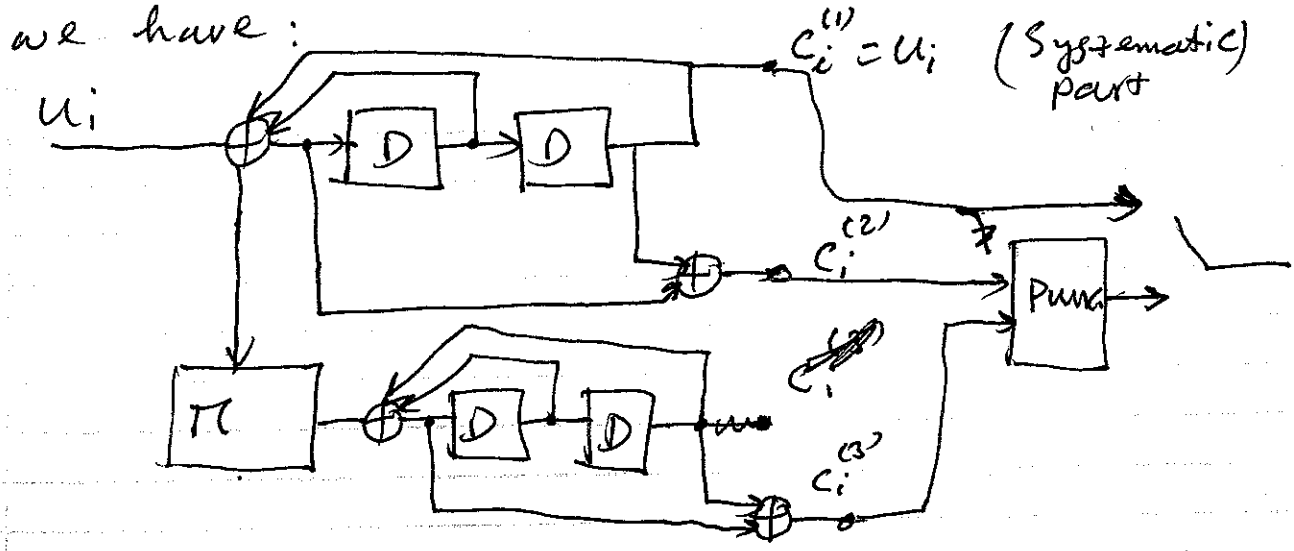
As seen, there are two encoders (they can be the same (usually they are) or different. They work on the input data: one on the data directly; the other on a shuffled (interleaved) version of data. Input Data itself (the system part) plus the output of the two encoders are sent. If the codes are of rate $\frac{1}{2}$, the overall rate is $\frac{1}{3}$ (since only one

systematic part is sent). Of course, the rate can increased by puncturing (not sending some of the non-systematic output). For example, if we delete half of the P_1 and half of the 2nd parity (P_2), then the rate is $\frac{1}{2}$.

The codes used are recursive with a generator matrix:

$$G(D) = \left[1 \quad \frac{g_2(D)}{g_1(D)}\right]$$

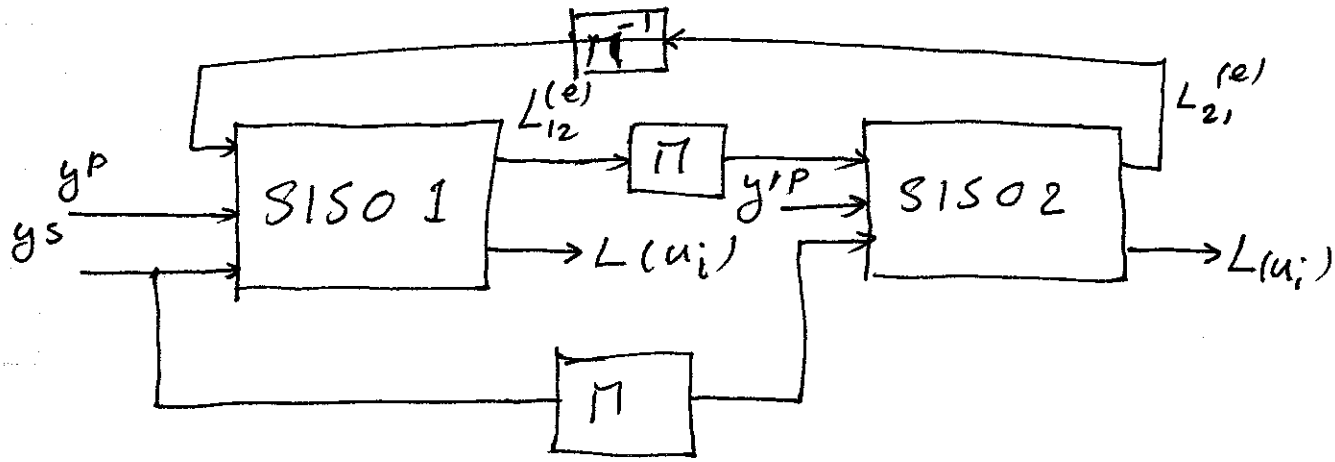
Example: Consider the $K=3$, rate $\frac{1}{2}$ code with $g_1(D) = D^2+D$ and $g_2(D) = 1+D^2$. Then we have:



Decoding of Turbo Codes:

Unlike conventional codes, where the input to the encoder is assumed to have equally probability and this is used in the decoding resulting in ML detection to be optimum (equivalent to MAP in the case of Turbo codes we have two encoders working in parallel and therefore at the receiver we will have two decoders working in parallel. The two decoders have access to the same systematic information plus one set of parity and therefore they each can try to find the original input to the encoder. But, if one of them does that and declares an output (a hard decision there would be no point in having sent extra parities and having the other decoder work. So, we require that instead (or in addition to) hard decision, each encoder provide some soft output (some reliability value) indicating how likely is it that that its

hard decision is correct. This would be an a priori information for the other decoder. The two decoders take turn in decoding and passing soft values (likelihood ratios) to each other until they have little to give. Then, we make the final decision The decoding structure is like this



where SISO stands for Soft-Input-Soft-Output decoder. This means that like conventional (soft decision) Viterbi decoder it takes soft input (the channel output not yet demodulated) but unlike Viterbi decoder it gives soft-output. This soft output is the like-lihood ratio of the corresponding bit.

Π is the interleaver and Π^{-1} is de-interleaver. We need these to align the extrinsic information (coming from another decoder) with the input from the channel.

The main component of a turbo decoder is the SISO. There are different ways to implement this. The most common one is BCJR (Bahl, Cocke, Jelinek and Raviv: 1974). Another method is Soft output VA (SOVA). Here, we only discuss BCJR.

BCJR decoding technique

We said earlier that the Viterbi Algorithm finds the optimum sequence, i.e., the sequence of bits that results in the least number of errors. BCJR is a bit-by-bit decoder. That is, for each bit, u_i, it looks at the whole sequences of channel outputs and finds the bit value that results in minimum probability of error. BCJR, in

addition, gives a soft value comparing the probability of this bit being 1 or 0.

Let's write the output- and state- equations governing ~~the~~ each component code:

$$c_i = f_c(u_i, \sigma_{i-1})$$

$$\sigma_i = f_s(u_i, \sigma_{i-1}) \qquad u_i \in \{0, 1\}$$

The decoder receives $\underline{y} = (y_1, y_2, \cdots, y_N)$

where $y_{im} = \sqrt{E_b}(2c_{im} - 1) + n_{im}$

and $\underline{y}_i = (y_{i1}, y_{i2}, \cdots, y_{in})$

in the case of the Turbo code we discussed

$$\underline{y}_i = (y_i^s, y_i^p, y_i^{p'})$$

Having $\underline{y} = (y_1, \cdots y_N)$ we have to maximize $P(u_i | \underline{y})$, i.e., we need to decide whether $u_i = 0$ or $u_i = 1$ based on which results in a higher $P(u_i | \underline{y})$ [is $P(u_i = 0 | \underline{y}) > P(u_i = 1 | \underline{y})$ or not?]

So, the problem is to find $\hat{u}_i$ as

$$\hat{u}_i = \arg \max_{u_i \in \{0,1\}} P(u_i | \underline{y})$$

Which can be written as,

$$\hat{u}_i = \underset{u_i \in \{0,1\}}{\arg\max} \frac{P(u_i, \underline{y})}{P(\underline{y})}$$

$$= \underset{u_i \in \{0,1\}}{\arg\max}\ p(u_i, \underline{y})$$

$$= \underset{\ell \in \{0,1\}}{\arg\max} \sum_{(\sigma_{i-1}, \sigma_i) \in S_\ell} p(\sigma_{i-1}, \sigma_i, \underline{y})$$

where S_ℓ is the set of all transitions from stage $\underline{i-1}$ to stage $\underline{i}$ that corresponds to an input $u_i = \ell$. So

$$S_0 = \{(\sigma_{i-1}, \sigma_i) \text{ s.t. } \sigma_i = f_s(u_i = 0, \sigma_{i-1})\}$$

and

$$S_1 = \{(\sigma_{i-1}, \sigma_i) \text{ s.t. } \sigma_i = f_s(u_i = 1, \sigma_{i-1})\}$$

We can write:

$$\underline{y} = (y_1, y_2, \cdots y_{i-1}, \underline{y_i}, y_{i+1}, \cdots y_N)$$

Define

$$y_1^{(i-1)} = (y_1, y_2, \cdots y_{i-1})$$

i.e., the vector of channel outputs coming before y_i and define

$$y_{i+1}^{(N)} = (y_{i+1}, y_{i+2}, \cdots y_N)$$

i.e. the vector of values following y_i. Then

$$\underline{y} = (y_1^{(i-1)}, y_i, y_{i+1}^{(N)})$$

Then:

$$p(\sigma_{i-1}, \sigma_i, \underline{y}) = p(\sigma_{i-1}, \sigma_i, y_1^{(i-1)}, y_i, y_{i+1}^{(N)})$$

$$= p(\sigma_{i-1}, \sigma_i, y_1^{(i-1)}, y_i) p(y_{i+1}^{(N)} | \sigma_{i-1}, \sigma_i, y_1^{(i-1)}, y_i)$$

$$= p(\sigma_{i-1}, y_1^{(i-1)}) p(\sigma_i, y_i | \sigma_{i-1}, y_1^{(i-1)}) p(y_{i+1}^{(N)} | \sigma_{i-1}, \sigma_i, y_i)$$

$$= p(\sigma_{i-1}, y_1^{(i-1)}) p(\sigma_i, y_i | \sigma_{i-1}) p(y_{i+1}^{(N)} | \sigma_i)$$

Denote

$$\alpha_{i-1}(\sigma_{i-1}) = p(\sigma_{i-1}, y_1^{(i-1)})$$

$$\beta_i(\sigma_i) = p(y_{i+1}^{(N)} | \sigma_i)$$

$$\gamma_i(\sigma_{i-1}, \sigma_i) = p(\sigma_i, y_i | \sigma_{i-1})$$

Using these symbols, we have

$$p(\sigma_{i-1}, \sigma_i, \underline{y}) = \alpha_{i-1}(\sigma_{i-1}) \gamma_i(\sigma_{i-1}, \sigma_i) \beta_i(\sigma_i)$$

So:

$$\hat{u}_i = \arg\max_{\ell \in \{0,1\}} \sum_{(\sigma_{i-1}, \sigma_i) \in S_\ell} \alpha_{i-1}(\sigma_{i-1}) \gamma_i(\sigma_{i-1}, \sigma_i) \beta_i(\sigma_i)$$

Now, we find two recursion formulas for computing $\alpha_{i-1}(\sigma_{i-1})$ and $\beta_i(\sigma_i)$ for $i \in \{1, \cdots, N\}$

Forward recursion for $\alpha_i(\sigma_i)$:

$$\alpha_i(\sigma_i) = \sum_{\alpha_{i-1} \in \Sigma} \gamma_i(\sigma_{i-1}, \sigma_i)\, \alpha_{i-1}(\sigma_{i-1}) \qquad 1 \le i \le N$$

So, we start from $\alpha_0(\sigma_0)$ and recursively find $\alpha_i(\sigma_i)$ $i = 1, 2, \cdots, N$

The initial point is:

$$\alpha_0(\sigma_0) = P(\sigma_0) = \begin{cases} 1 & \sigma_0 = 0 \\ 0 & \sigma_0 \neq 0 \end{cases}$$

Proof:

$$\alpha_i(\sigma_i) = p(\sigma_i, y_1^{(i)}) = \sum_{\sigma_{i-1} \in \Sigma} p(\sigma_{i-1}, \sigma_i, y_1^{(i-1)}, y_i)$$

$$= \sum_{\sigma_{i-1} \in \Sigma} p(\sigma_{i-1}, y_1^{i-1})\, p(\sigma_i, y_i \mid \sigma_{i-1}, y_1^{(i-1)})$$

$$= \sum_{\sigma_{i-1} \in \Sigma} p(\sigma_{i-1}, y_1^{(i-1)})\, p(\sigma_i, y_i \mid \sigma_{i-1})$$

$$= \sum_{\sigma_{i-1} \in \Sigma} \alpha_{i-1}(\sigma_{i-1})\, \gamma_i(\sigma_{i-1}, \sigma_i)$$

Backward Recursion for $\beta_i(\sigma_i)$

$$\beta_{i-1}(\sigma_{i-1}) = \sum_{\sigma_i \in \Sigma} \beta_i(\sigma_i)\,\gamma_i(\sigma_{i-1}, \sigma_i) \qquad 1 \le i \le N$$

with the starting condition

$$\beta_N(\sigma_N) = \begin{cases} 1 & \sigma_N = 0 \\ 0 & \sigma_N \neq 0 \end{cases}$$

That is, we assume that the trellis terminate at state zero.

Proof: $\beta_{i-1}(\sigma_{i-1}) = p(y_i^{(N)} \mid \sigma_{i-1})$

so:

$$\beta_{i-1}(\sigma_{i-1}) = \sum_{\sigma_i \in \Sigma} p(y_i, y_{i+1}^{(N)}, \sigma_i \mid \sigma_{i-1})$$

$$= \sum_{\sigma_i \in \Sigma} p(y_i, \sigma_i \mid \sigma_{i-1})\, p(y_{i+1}^{(N)} \mid \sigma_i, y_i, \sigma_{i-1})$$

$$= \sum_{\sigma_i \in \Sigma} p(\sigma_i, y_i \mid \sigma_{i-1})\, p(y_{i+1}^{(N)} \mid \sigma_i)$$

$$= \sum_{\sigma_i \in \Sigma} \gamma_i(\sigma_{i-1}, \sigma_i)\,\beta_i(\sigma_i)$$

Computing $\gamma_i(\sigma_{i-1}, \sigma_i)$

$$\gamma_i(\sigma_{i-1}, \sigma_i) = p(\sigma_i, y_i \mid \sigma_{i-1})$$

$$= p(\sigma_i \mid \sigma_{i-1})\, p(y_i \mid \sigma_i, \sigma_{i-1})$$

$$= p(u_i)\, p(y_i \mid u_i) = p(u_i)\, p(y_i \mid c_i)$$

In addition to giving out the most likely value (zero or one) BCJR provide $P(u_i | \underline{y})$ which can be used to define the log-likelihood ratio:

$$L(u_i) = \ln \frac{P(u_i=1 | \underline{y})}{P(u_i=0 | \underline{y})}$$

$$= \ln \frac{P(u_i=1, \underline{y})}{P(u_i=0, \underline{y})}$$

$$= \ln \frac{\sum_{(\sigma_{i-1}, \sigma_i) \in S_1} \alpha_{i-1}(\sigma_{i-1}) \gamma_i(\sigma_{i-1}, \sigma_i) \beta_i(\sigma_i)}{\sum_{(\sigma_{i-1}, \sigma_i) \in S_0} \alpha_{i-1}(\sigma_{i-1}) \gamma_i(\sigma_{i-1}, \sigma_i) \beta_i(\sigma_i)}$$

At any point, if one wants to decide on u_i, he makes use of the following rule

$$\hat{u}_i = \begin{cases} 1 & L(u_i) \geq 0 \\ 0 & L(u_i) < 0 \end{cases}$$

For the Gaussian channel:

$$\gamma_i(\sigma_{i-1}, \sigma_i) = \frac{P(u_i)}{(\pi N_0)^{n/2}} \exp\left[- \frac{\| \underline{y}_i - \underline{c}_i \|^2}{N_0} \right]$$

Example: Let's consider $n = 2$ then

$\underline{y}_i = (y_i^s, y_i^p)$.

Then:

$$\gamma(\sigma_{i-1}, \sigma_i) = \frac{P(u_i)}{\pi N_0} \exp\left[-\frac{(y_i^s - c_i^s)^2 + (y_i^p - c_i^p)^2}{N_0}\right]$$

where c_i^s and c_i^p are either $+\sqrt{E_c}$ or $-\sqrt{E_c}$ depending on the value of u_i (and the parity, respectively.

Therefore,

$$\gamma_i(\sigma_{i-1}, \sigma_i) = \frac{P(u_i)}{\pi N_0} \exp\left[-\frac{(y_i^s)^2 + (y_i^p)^2 + 2E_c}{N_0}\right] \exp\left[\frac{2y_i^s c_i^s + 2y_i^p c_i^p}{N_0}\right]$$

The part $\frac{1}{\pi N_0} \exp\left[-\frac{(y_i^s)^2 + (y_i^p)^2 + 2E_c}{N_0}\right]$ does not depend on u_i can be discarded.

So,

$$L(u_i) = \ln \frac{\sum_{s_1} \alpha_{i-1}(\sigma_{i-1}) P(u_i) \exp\left(\frac{2y_i c_i^s}{N_0}\right) \exp\left[\frac{2y_i^p c_i^p}{N_0}\right] \beta_i(\sigma)}{\sum_{s_0} \alpha_{i-1}(\sigma_{i-1}) P(u_i) \exp\left(\frac{2y_i c_i^s}{N_0}\right) \exp\left[\frac{2y_i^p c_i^p}{N_0}\right] \beta_i(\sigma)}$$

$$= \frac{4\sqrt{E_c} y_i^s}{N_0} + \ln \frac{P(u_i=1)}{P(u_i=0)} + \ln \frac{\sum_{s_1} \alpha_{i-1}(\sigma_{i-1}) \exp\left(\frac{2y_i^p c_i^p}{2N_0}\right) \beta_i}{\sum_{s_0} \alpha_{i-1}(\sigma_{i-1}) \exp\left(\frac{2y_i^p c_i^p}{2N_0}\right) \beta_i}$$

As seen, the Log-likelihood value has three components: one depends on the channel output, 2nd depends on the

a priori probability, and the third being new information or extrinsic information (to be used by the other encoder as a priori informat in the next iteration.

So:

$$L(u_i) = L_c + L^a(u_i) + L^e(u_i)$$

where

$$L^a(u_i) = \ln \frac{P(u_i=1)}{P(u_i=0)}$$

$$L_c = \frac{4\sqrt{E_c}\, y_i^s}{N_0}$$

and

$$L^e(u_i) = \ln \sum_{S_1} \alpha_{i-1}(\sigma_{i-1}) \exp\left(\frac{2 y_i^p c_i^p}{N_0}\right) \beta_i(\sigma_i) - \ln \sum_{S_0} \alpha_{i-1}(\sigma_{i-1}) \exp\left(\frac{2 y_i^p c_i^p}{N_0}\right) \beta_i(\sigma_i)$$

Log-APP algorithm:

The BCJR algorithm, when the trellis is long is not numerically stable. So, it is better to perform the computations in log domain. Log-APP algorithm is the equivalent of BCJR

So, the use of Log_APP does not entail any loss of performance, but avoids computational problems.

In Log_APP method we define:

$$\tilde{\alpha}_i(\sigma_i) = \ln(\alpha_i(\sigma_i))$$

$$\tilde{\beta}_i(\sigma_i) = \ln(\beta_i(\sigma_i))$$

and

$$\tilde{\gamma}_i(\sigma_{i-1}, \sigma_i) = \ln(\gamma_i(\sigma_{i-1}, \sigma_i))$$

Then the backward and forward recursions are

$$\tilde{\alpha}_i(\sigma_i) = \ln\left(\sum_{\sigma_{i-1} \in \Sigma} \exp(\tilde{\alpha}_{i-1}(\sigma_{i-1}) + \tilde{\gamma}_i(\sigma_{i-1}, \sigma_i))\right)$$

and

$$\tilde{\beta}_{i-1}(\sigma_{i-1}) = \ln\left(\sum_{\sigma_i \in \Sigma} \exp(\tilde{\beta}_i(\sigma_i) + \tilde{\gamma}_i(\sigma_{i-1}, \sigma_i))\right)$$

with the initial condition:

$$\tilde{\alpha}_0(\sigma_0) = \begin{cases} 0 & \sigma_0 = 0 \\ -\infty & \sigma_0 \neq 0 \end{cases}$$

and

$$\tilde{\beta}_N(\sigma_N) = \begin{cases} 0 & \sigma_N = 0 \\ -\infty & \sigma_N \neq 0 \end{cases}.$$

Then:

$$L(u_i) = \ln\left[\sum_{S_1} \exp\left(\tilde{\alpha}_{i-1}(\sigma_{i-1}) + \tilde{\gamma}_i(\sigma_{i-1}, \sigma_i) + \tilde{\beta}_i(\sigma_i)\right)\right] - \ln\left[\sum_{S_0} \exp\left(\tilde{\alpha}_{i-1}(\sigma_{i-1}) + \tilde{\gamma}_i(\sigma_{i-1}, \sigma_i) + \tilde{\beta}_i(\sigma_i)\right)\right].$$

While computationally stable, the Log-APP is not computationally efficient.

One can simplify it by observing that

$$\ln(e^x + e^y) \simeq \max(x, y)$$

$$\ln(e^x + e^y + e^z) \simeq \max(x, y, z)$$

The approximation is tight when one of the argument is much larger than the others (which is usually the case). However, one can add a correction factor to reduce the error.

Invention

The new simplified algorithm is called Max-log method. Here, we have

$$\tilde{\alpha}_i(\sigma_i) = \max_{\sigma_{i-1} \in \Sigma} \left[\tilde{\alpha}_{i-1}(\sigma_{i-1}) + \tilde{\gamma}_i(\sigma_{i-1}, \sigma_i)\right]$$

$$\tilde{\beta}_{i-1}(\sigma_{i-1}) = \max_{\sigma_i \in \Sigma} \left[\tilde{\beta}_i(\sigma_i) + \gamma_i(\sigma_{i-1}, \sigma_i)\right]$$

The log-likelihood ratio (the L-value) is:

$$L(u_i) = \max_{(\sigma_{i-1},\sigma_i)\in S_1} \left[\tilde{\alpha}_{i-1}(\sigma_{i-1}) + \tilde{\gamma}_i(\sigma_{i-1},\sigma_i) + \tilde{\beta}_i(\sigma_i)\right] - \max_{(\sigma_{i-1},\sigma_i)\in S_0} \left[\tilde{\alpha}_{i-1}(\sigma_{i-1}) + \tilde{\gamma}_i(\sigma_{i-1},\sigma_i) + \tilde{\beta}_i(\sigma_i)\right]$$

For the example of Turbo code with rate $\frac{1}{2}$ component codes and Gaussian channel;

$$L(u_i) = \frac{4\sqrt{Ee}\, y_i^s}{N_0} + L^a(u_i) + \max\left[\tilde{\alpha}_{i-1}(\sigma_{i-1}) + \frac{2y_i^p c_i^p}{N_0} + \tilde{\beta}_i(\sigma_i)\right] - \max\left[\tilde{\alpha}_{i-1}(\sigma_{i-1}) + \frac{2y_i^p c_i^p}{N_0} + \tilde{\beta}_i(\sigma_i\right.$$

where

$$L^a(u_i) = \ln\frac{P(u_i=1)}{P(u_i=0)}$$

Iterative Procedure:

The first decoder starts decoding based on (y^s, y^p) with $L^a(u_i)=0$ (since at the beginning there is no information about prob. of input and we assume they are equally likely). It uses BCJR to find $L(u_i)$. It subtracts $L_c = \frac{4\sqrt{Ee}\, y_i^s}{N_0}$

from it and find $L_{12}^{e}(u_i)$ and passes it to the decoder 2. Of course it interleaves the $L_{12}^{(e)}$ in order to be aligned with its channel input stream.

Decoder 2 uses L_c and $L_{12}^{e}(u_i)$ as $L^a(u_i)$ to find $L(u_i)$. Then it substracts L_c and $L^a(u_i)$ to get the extrinsic information $L_{21}^{e}(u_i)$ which is given to decoder 1 to be used as its new $L^a(u_i)$. This iterative procedure continues until no further improvement is observed. This is usually 4-6 iterations.

LDPC (Low Density Parity Check) Codes

LDPC Codes were invented by R. G. Gallager in 1960 as his Ph.D. thesis.
They were soon forgotten, possibly, due to the fact that at that time there were not processors powerfull enough to take advantage of them.
Latter in 1990's they were "re-invented by at least two group of researchers and became popular.
LDPC codes are linear block codes with sparse parity check matrices. That is, in forming each parity only a small number of terms are involved. This results in long codes being with rather moderate complexity being decoded.
To start, with something familiar and easy to work with, we choose the (7, 4) Hamming code.

The (7,4) Hamming code has the generator matrix:

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

An information vector $(u_1, u_2, u_3, u_4) = \underline{u}$ is encoded as

$$\underline{c} = \underline{u} G$$

i.e.,

$$c_1 = u_1$$

$$c_2 = u_2$$

$$c_3 = u_3$$

$$c_4 = u_4$$

$$c_5 = u_1 + u_2 + u_3$$

$$c_6 = u_2 + u_3 + u_4$$

$$c_7 = u_1 + u_2 + u_4$$

While any (n,k) code has a $k \times n$ generator matrix, it can also be defined in terms of an $(n-k) \times n$ matrix called a parity check matrix. This matrix, denoted by H is

orthogonal to G in the sense that

$$GH^T = 0$$

since each code word is represented as $\underline{c} = \underline{u}G$, we have

$$\underline{c}H^T = \underline{u}GH^T = \underline{u}\,\underline{0} = \underline{0}.$$

That is, for valid codes, the parity check matrix can verify the validity.

For the (7,4) Hamming code the H matrix is:

$$H = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Using the parity check equation above ($cH^T = 0$), we see that, for any valid codeword:

$$c_1 + c_2 + c_3 + c_5 = 0$$

$$c_2 + c_3 + c_4 + c_6 = 0$$

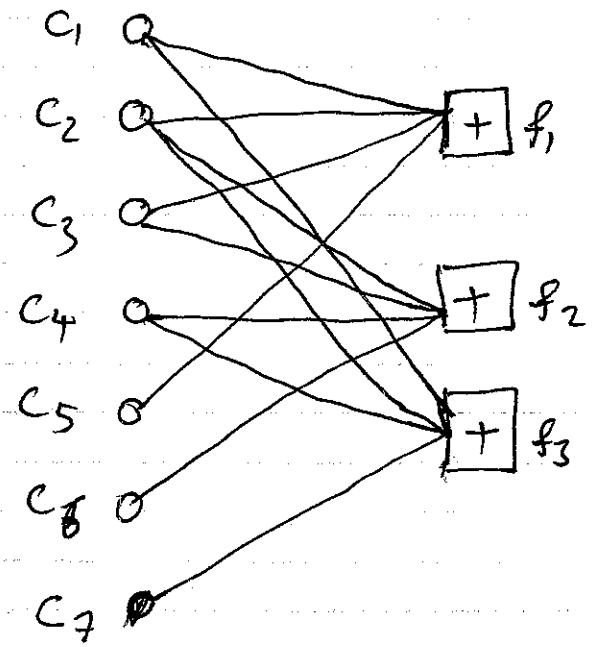
$$c_1 + c_2 + c_4 + c_7 = 0$$

The parity relationships for a linear block codes can be graphically represented by a bipartite graph called a Tanner graph.

A bipartite graph is a graph whose node can be divided into two sets ~~each~~ containing N_1 and N_2 nodes, respectively. The node in one set are only connected to nodes in the other set.

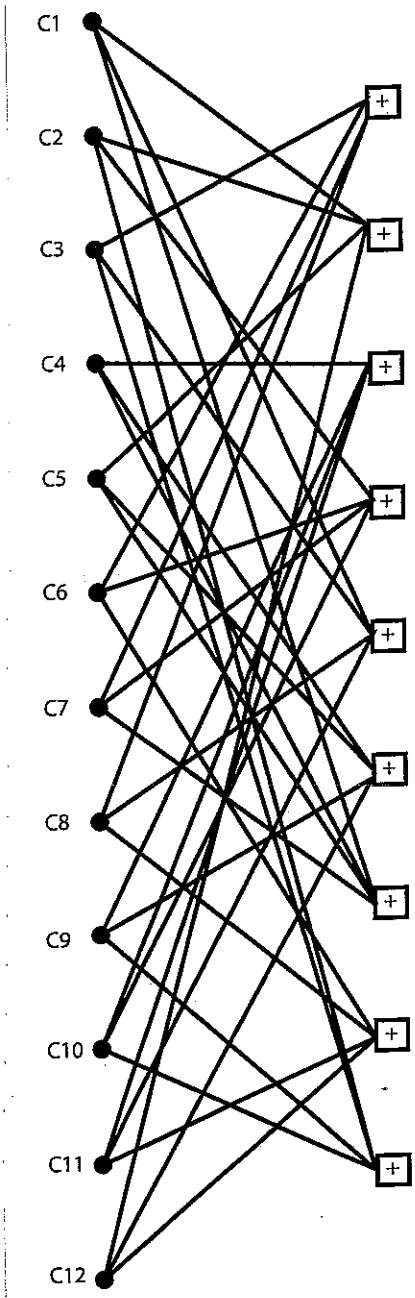
For representing block codes using Tanner Graphs. we have two sets of nodes, variable nodes (one for each component of the codeword) and check nodes (one for each parity check equation)

For the (7, 4), as an example, we have:



As another example, Consider the code with the parity check matrix:

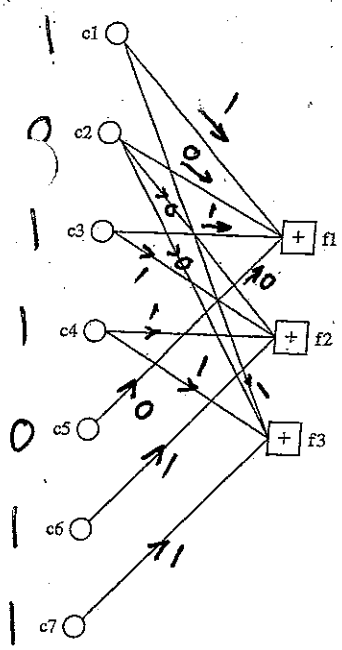
$$H = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

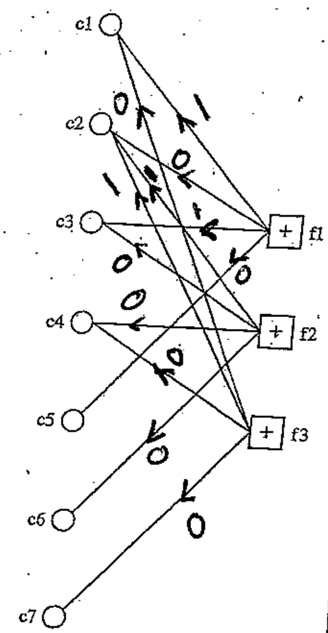


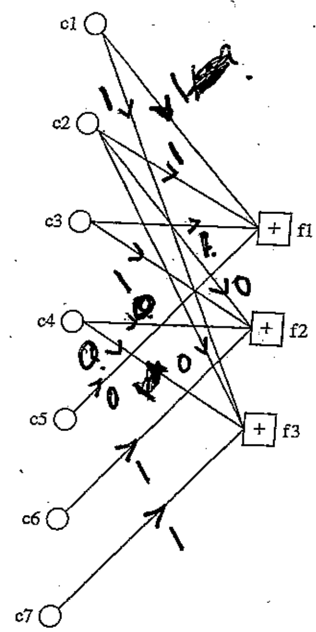
The Decoding algorithm for LDPC Codes.

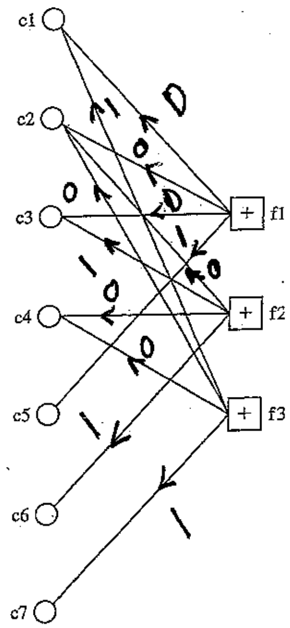
LDPC codes can be decoded iteratively, using a technique called message passing or belief propagation.

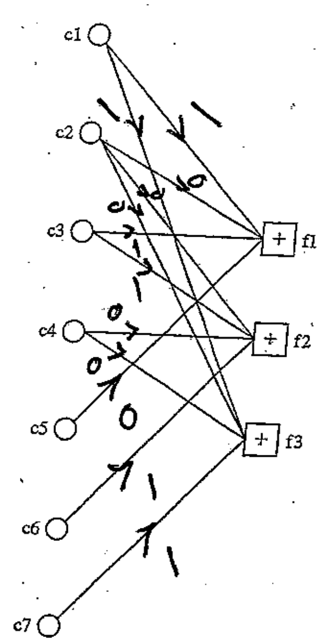
In each iteration each variable node sends its observation plus observations it has receive from other check nodes (all except the one it wants to send a message to) in form of a message specifying the likelihood of its value. In return each check node sends to each variable node what it believes is the likelihood of the value of that variable node. This belief is formed based on the parity check equation and the value received from all other variable nodes (except the targetted variable node).

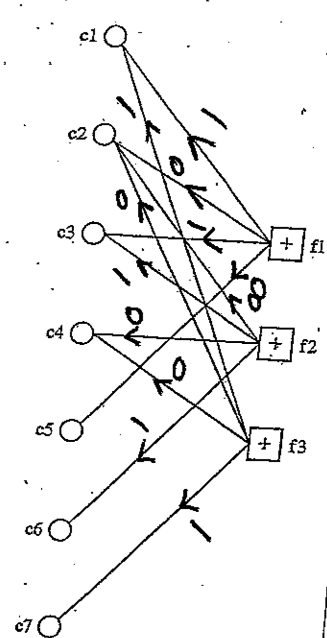

c1
c2
c3
c4
c5
c6
c7
f1
f2
f3

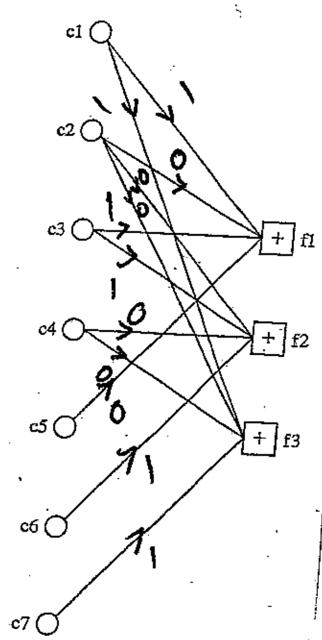

c1
c2
c3
c4
c5
c6
c7
f1
f2
f3

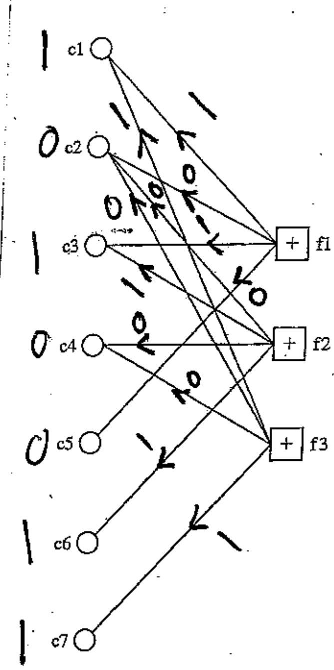

c1
c2
c3
c4
c5
c6
c7
f1
f2
f3


c1
c2
c3
c4
c5
c6
c7
f1
f2
f3


c1
c2
c3
c4
c5
c6
c7
f1
f2
f3


c1
c2
c3
c4
c5
c6
c7
f1
f2
f3


c1
c2
c3
c4
c5
c6
c7
f1
f2
f3


c1
c2
c3
c4
c5
c6
c7
f1
f2
f3

LDPC decoding: Belief Propagation algorithm.

The variable nodes initially have the channel values $y_1, y_2, \cdots y_n$ corresponding to (a noisy version of) the codeword $(c_1, c_2, \cdots c_n)$.

Given these values of $y_1, \cdots y_n$, the variable nodes can calculate the likelihood (or log-likelihoo rations of $(c_1, \cdots, c_n)$ as

$$\ell(c_i) = \log \frac{P(c_i=1|y_i)}{P(c_i=0|y_i)} = \log \frac{P(y_i|c_i=1)}{P(y_i|c_i=0)}$$

where we have assumed that the transmitted bits are equiprobable.

Note that

Progpiprogram

$$\frac{P(c_i=1|y_i)}{P(c_i=0|y_i)} = e^{\ell(c_i)}$$

$$\frac{1-P(c_i=0|y_i)}{P(c_i=0|y_i)} = e^{\ell(c_i)}$$

or

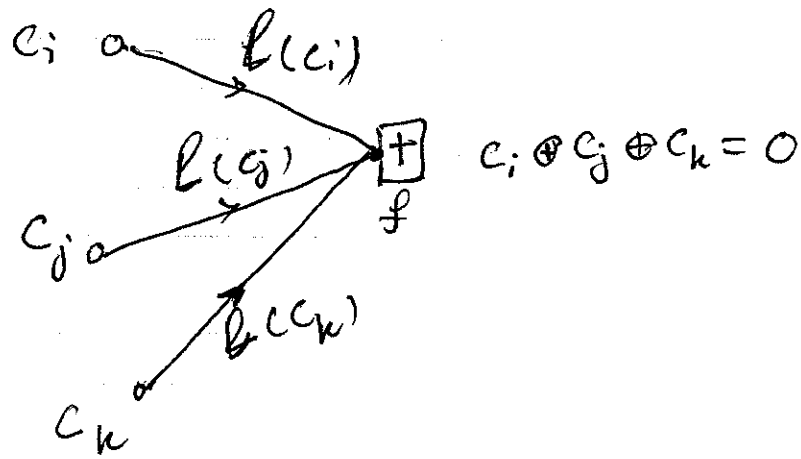
$$P(c_i=0|y_i) = \frac{1}{1+e^{\ell(c_i)}}$$

or

and

$$P(c_i = 1 \mid y_i) = \frac{e^{\ell(c_i)}}{1+e^{\ell(c_i)}} = \frac{1}{1+e^{-\ell(c_i)}}$$

The check nodes, after receiving these ~~liked~~ log-likelihood ratios from the variable nodes combine them to form a response for each variable node that has sent a message. Assume that a check node f receives messages $\ell(c_i), \ell(c_j)$ and $\ell(c_k)$ from variable node c_i, c_j and c_k,



f has the check equation $c_i + c_j + c_k = 0$ So, it knows that $c_i = c_j \oplus c_k$. So, it should send back to c_i, $\ell_f(c_j \oplus c_k)$. That is, the check node sends to the variable node what is learns

from other nodes (all connected nodes to it except c_i)
similarly, it sends to c_j and c_k,
$l_f(c_i \oplus c_k)$ and $l_g(c_j \oplus c_i)$, respectively.

Remember

$$P(c_j = 0) = \frac{1}{1 + e^{+l(c_j)}} =$$

and

$$P(c_j = 1) = \frac{1}{1 + e^{-l(c_j)}}$$

$$\tanh\left(\frac{l(c_j)}{2}\right) = \frac{e^{l(c_j)} - 1}{e^{l(c_j)} + 1} = \frac{1}{1 + e^{-l(c_j)}} - \frac{1}{1 + e^{l(c_j)}}$$

$$= P(c_j = 1) - P(c_j = 0)$$

$$= 1 - 2P(c_j = 0)$$

or

$$2P(c_j = 0) = 1 - \tanh\left(\frac{l(c_j)}{2}\right)$$

$$2P(c_j = 1) = 1 + \tanh\left(\frac{l(c_j)}{2}\right)$$

Now consider:

$$l(c_j \oplus c_k) = \log \frac{P(c_j \oplus k = 1)}{P(c_j \oplus c_k = 0)}$$

First let's calculate

$$P(c_j \oplus c_k = 0) = P(c_j = 0)\,P(c_k = 0) + P(c_j = 1)P(c_k =$$

$$= \frac{1 - \tanh\left(\frac{l(c_j)}{2}\right)}{2} \cdot \frac{1 - \tanh\left(\frac{l(c_k)}{2}\right)}{2} +$$

$$+ \frac{1 + \tanh\left(\frac{l(c_j)}{2}\right)}{2} \cdot \frac{1 + \tanh\left(\frac{l(c_k)}{2}\right)}{2}$$

$$= \frac{1 + \tanh\left(\frac{l(c_j)}{2}\right)\tanh\left(\frac{l(c_k)}{2}\right)}{2}$$

Similarly,

$$P(c_j \oplus c_k = 1) = \frac{1 - \tanh\left(\frac{l(c_j)}{2}\right)\tanh\left(\frac{l(c_k)}{2}\right)}{2}$$

Therefore,

$$l(c_j \oplus c_k) = \ln \frac{1 + \tanh\left(\frac{l(c_j)}{2}\right)\tanh\left(\frac{l(c_k)}{2}\right)}{1 + \tanh\left(\frac{l(c_j)}{2}\right)\tanh\left(\frac{l(c_k)}{2}\right)}$$

In general:

$$l(c_1 \oplus c_2 \oplus \cdots \oplus c_e | y_1, \cdots y_e) = \ln \frac{1 - \prod_{i=1}^{e} \tanh\left(\frac{l(c_i)}{2}\right.}{1 + \prod_{i=1}^{e} \tanh\left(\frac{l(c_i)}{2}\right.}$$

So, the belief propagation algorithm works like this, each variable node, say c, sends to each check node connected to it, say, f:

$$m_{cf}^{(\ell)} = \begin{cases} m_c & \text{if } \ell = 0 \\ m_c + \sum_{f' \in F_c \setminus \{f\}} m_{f'c}^{(\ell-1)} & \text{if } \ell \geq 1 \end{cases}$$

and each check node sends to each variable node connected to it

$$m_{fc}^{(\ell)} = h\left(\frac{1 + \prod_{c' \in C_f \setminus \{c\}} \tanh\left(m_{c'f}^{(\ell)}/2\right)}{1 - \prod_{c' \in C_f \setminus \{c\}} \tanh\left(m_{c'f}^{(\ell)}/2\right)} \right)$$

where F_c is the set of check node incident on the message node c and C_f is the set of message notes incident on a check node f.