EE624: Information and Coding Theory

TURBO encoder & decoder implementation and its performance over AWGN channel for BPSK modulated symbols

YOGESH KUMAR SONIWAL

Roll No.:Y9672

1. Introduction

Turbo codes are a class of high-performance Forward Error Correction Codes (FEC) developed in 1993. These are derived from parent convolution codes. Turbo codes uses Recursive Systematic Convolution (RSC) Encoder and Iterative BCJR (Bahl Cocke Jelinek Raviv) decoder. Its performance is much better than the corresponding convolution code.

In this term paper the RSC encoder and BCJR decoder are described in sections 2 to 4. Algorithm design is described in section 5. Simulation results are presented in section 6. Conclusion is drawn in section 7.

2. RSC (Recursive Systematic Convolution) Turbo Encoder:

Turbo encoder is comprised of 2 Rate ½ RSC encoder as shown in Figure 1.The first encoder takes the input information bits and generates Parity bits $\overline{P_0}$. The interleaver π interleaves the information bits $\bar{X}$ to generate interleaved information $\overline{X_\pi}$. The second encoder uses $\overline{X_\pi}$ and generates Parity bits $\bar{P}_1$.

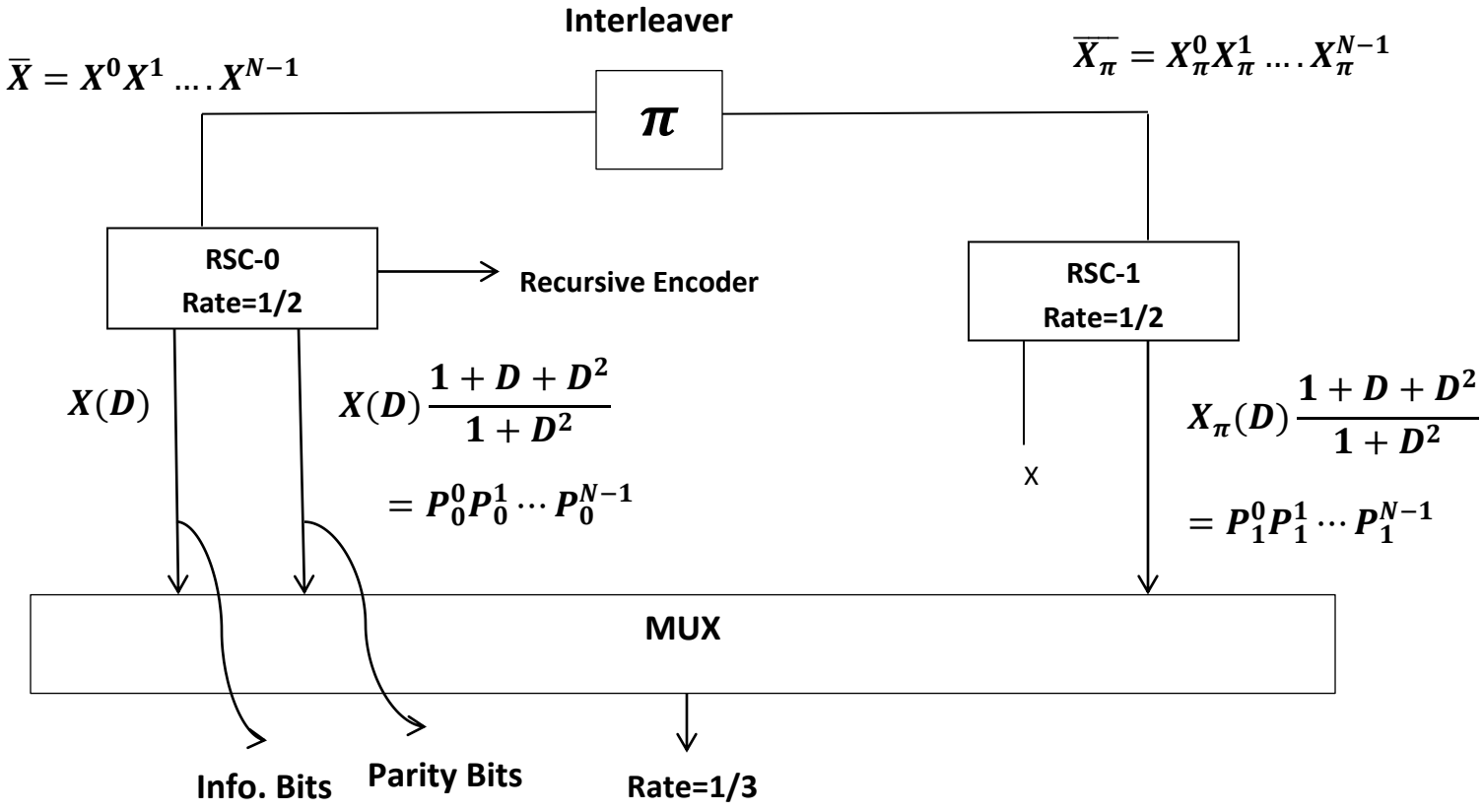


Figure 1: Turbo Encoder Architecture for Code $\boldsymbol{G(D) = \left[1 \quad \frac{1+D+D^2}{1+D^2}\right]}$

Encoding for each Rate ½ RSC encoder (RSC-0 & RSC-1) is done as follows:

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

Which gives $C_0(D) = X(D)$ and $C_1(D) = X(D)\frac{1+D+D^2}{1+D^2}$

Now assume $F(D) = \frac{X(D)}{1+D^2}$,

From these relations we obtain $C_0^j = X^j$ and $C_1^j = X^j + F^{j-1}$,where $F^j = F^{j-2} + X^j$

If we consider (F^{j-1} F^{j-2}) to be current state the state diagram (Figure: 2) and Trellis Diagram (Figure: 3) are obtained

Encoded Codeword from Turbo Encoder: $\overline{\boldsymbol{C}} = \boldsymbol{X^0 P_0^0 P_1^0}\ .\ \boldsymbol{X^1 P_0^1 P_{1\ldots}^1}\ .\boldsymbol{X^{N-1} P_0^{N-1} P_1^{N-1}}$

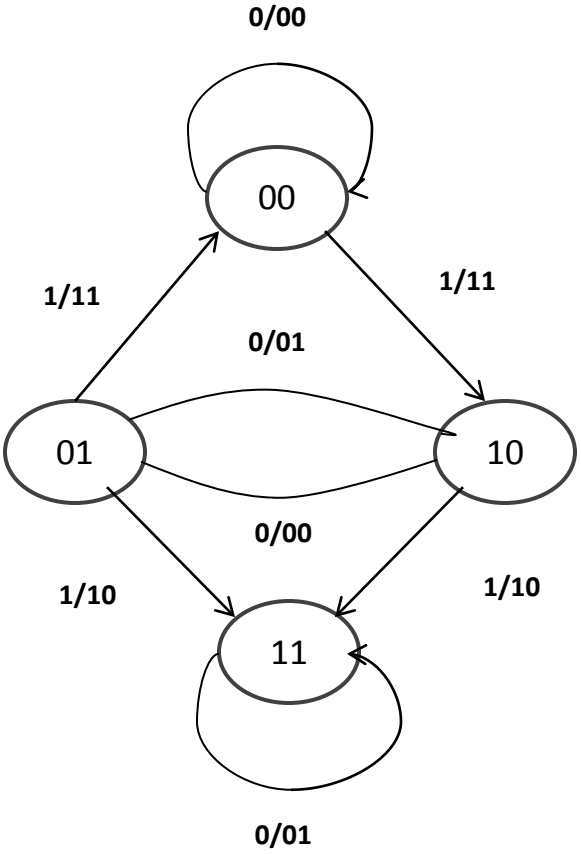


Figure 2: State Diagram

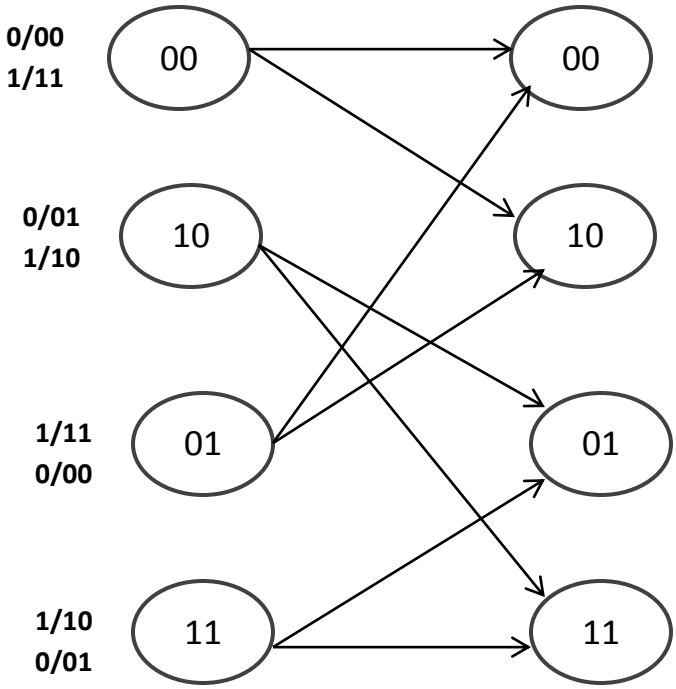


Figure 3: Trellis Diagram

3. Transmission of BPSK modulated symbols through AWGN channel:

Modulation using BPSK modulation:

$$u_0^j = 2X^j - 1$$
$$u_1^j = 2P_0^j - 1$$
$$u_2^j = 2P_1^j - 1$$

For j=0,1….N-1

Transmission through AWGN channel

$$R_i^j = \sqrt{P}u_i^j + n_i^j$$

For j=0,1,….,N-1 and i=0,1,2. Here P is signal power. Noise variance is assumed to be unity.

4. Decoding of received codewords at the receiver using BCJR algorithm:

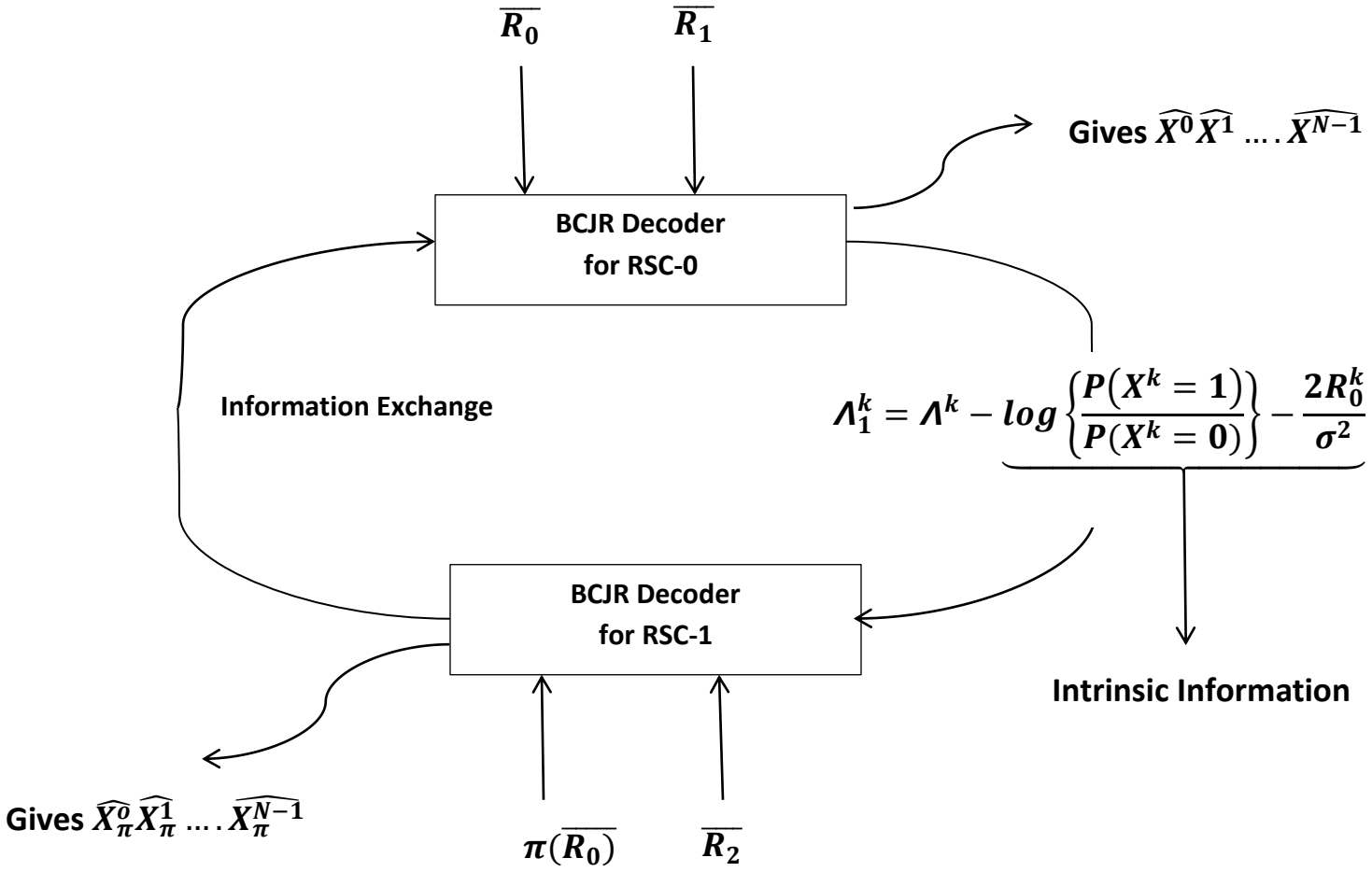


Figure 4: BCJR Decoder Architecture

BCJR (Bahl Cocke Jelinek Raviv) decoder Architecture is shown in Figure: 4. There are two decoders, BCJR-0 and BCJR-1. The first decoder takes $\overline{R_0}$ and $\overline{R_1}$ as inputs and second decoder takes $\pi(\overline{R_0})$ and $\overline{R_2}$ as inputs.

We define Log Aposteriori Probability Ratio(LAPPR) as follows:

$$\Lambda^k = log\left\{\frac{P(X^k = 1|\overline{R})}{P(X^k = 0|\overline{R}}\right\}$$

For k=0,1,….N-1.

where $\overline{R}$ is the input to any of the decoder. Consider BCJR-0, then $\overline{R}$ = $\overline{R_0}$ $\overline{R_1}$
It can be shown that

$$P(X^k = i|\overline{R}) = \frac{1}{P(R_0^{N-1})}\sum_{m=0}^{M-1}\sum_{m'=0}^{M-1}\alpha_k(m')\gamma_k^i(m',m)\beta_k(m)$$

Where i=0 or 1 (input), m and m' are stages (S_k), M is total number of stages. Stage m=0, 1, 2 and 3 respectively represents stage 00, 01 10 and 11.

The description of terms inside summation is given as follows:

$\alpha_k(m')$ is forward state metric

$$\alpha_k(m') = P(S_k = m' | \overline{R_0^{k-1}})$$

Solving $\alpha_k(m')$ gives

$$\alpha_k(m') = \sum_{m''=0}^{M-1} \sum_{i=0}^{1} \gamma_{k-1}^{i}(m'', m') \alpha_{k-1}(m'')$$

$\beta_k(m)$ is backward state metric

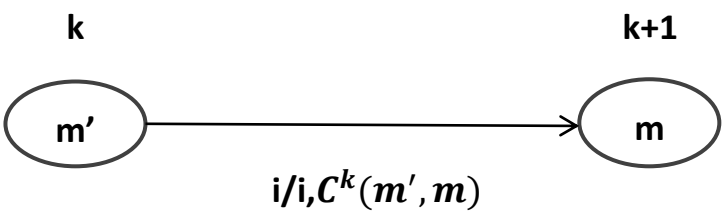
$$\beta_k(m) = P(\overline{R_{K+1}^{N-1}} | S_{k+1} = m)$$

Solving $\beta_k(m')$ gives

$$\beta_k(m) = \sum_{m'=0}^{M-1} \sum_{i=0}^{1} \gamma_{k+1}^{i}(m', m) \beta_{k+1}(m')$$

$\gamma_k^i(m', m)$ is transition probability from state m' to m at stage k for input i

$$\gamma_k^i(m', m) = \underbrace{P(X^k = i)}_{\text{Apriori Probability at time k for input i}} \underbrace{P(S_{k+1} = m | S_k = m', X^k = i)}_{\text{Obtained from trellis}} \underbrace{P(\overline{R^k} | S_{k+1} = m', S_k = m, X^k = i)}_{\text{Obtained from likelihood}}$$



$$= \frac{1}{\sqrt{2\pi\sigma^2}} exp\left(-\frac{1}{2\sigma^2}\left(R_0^k - (2i-1)\right)^2\right) X \frac{1}{\sqrt{2\pi\sigma^2}} exp\left(-\frac{1}{2\sigma^2}\left(R_1^k - (2C^k(m', m) - 1)\right)^2\right)$$

Using the equations described above and assuming some Apriori Probabilities for inputs 0 and 1 at all stages the first decoder calculates LAPPR for all stages and decodes the received codebits. Then it passes the LAPPR values excluding the intrinsic information to the second decoder (As described in Figure 4), which uses these LAPPR values to calculate Priory probabilities at all stages for both inputs. And now the second decoder repeats the same process incorporated by first decoder. This iteration stops when decoded bits using both the decoders become same or Priory Probability for one of the input becomes 1.

5. Algorithm Design

After receiving transmitted bits at the receiver with Additive White Gaussian Noise (AWGN), we start with initializing apriori probabilities to 0.5 for both input 0 and 1. Then we calculate ϒ probabilities at all stages for both inputs. The initialization of α probabilities for first stage is done assuming encoder to be in state 00 initially. So α is initialized as $\alpha_0(00) = 1, \alpha_0(01) = 0, \alpha_0(10) = 0, \alpha_0(11) = 0$. Then we use recursive relation of α to calculate α of next stage. Similarly β probabilities are initialized at the final stage i.e. N-1, Where N is block length. If we assume the next encoder to be in state 00 at start then β can be initialized as $\beta_{N-1}(00) = 1, \beta_{N-1}(01) = 0, \beta_{N-1}(10) = 0, \beta_{N-1}(11) = 0$. After calculation of α, β, and ϒ the ratio Λ^k is calculated for all stages. Observing the values of Λ^k , whether they are positive or negative the bits are decoded into 1 and 0 respectively.

After the first iteration the first decoder (BCJR-0) passed Λ^k values to second decoder excluding intrinsic information. The second decoder uses these probabilities to calculate priory probabilities at all stages using the following relation

$$\boldsymbol{P(X^k = 1) = \frac{e^{\Lambda_1^k}}{1 + e^{\Lambda_1^k}}}$$

$$\boldsymbol{P(X^k = 0) = \frac{1}{1 + e^{\Lambda_1^k}}}$$

The second decoder repeats the same process to decode the bits. In simulation this iterative process is terminated when Priory Probability for one of the input becomes 1.

6. Simulation Results

For the simulation I have used **N=1 million bits** of information and varied SNR from 0 dB to 9 dB.

Total time taken for 4 iterations and all values of SNR ranging with an increment of 0.5 dB (i.e. 19 values of SNRs) it took 246.510875 sec. Hence time taken for each SNR value=12.974 sec. and for each iteration it is **3.24 sec**, which is quite small considering the large number of information bits.

Plots for simulated Bit Error Rates iteration-vise are shown in Figure 5 to 8.

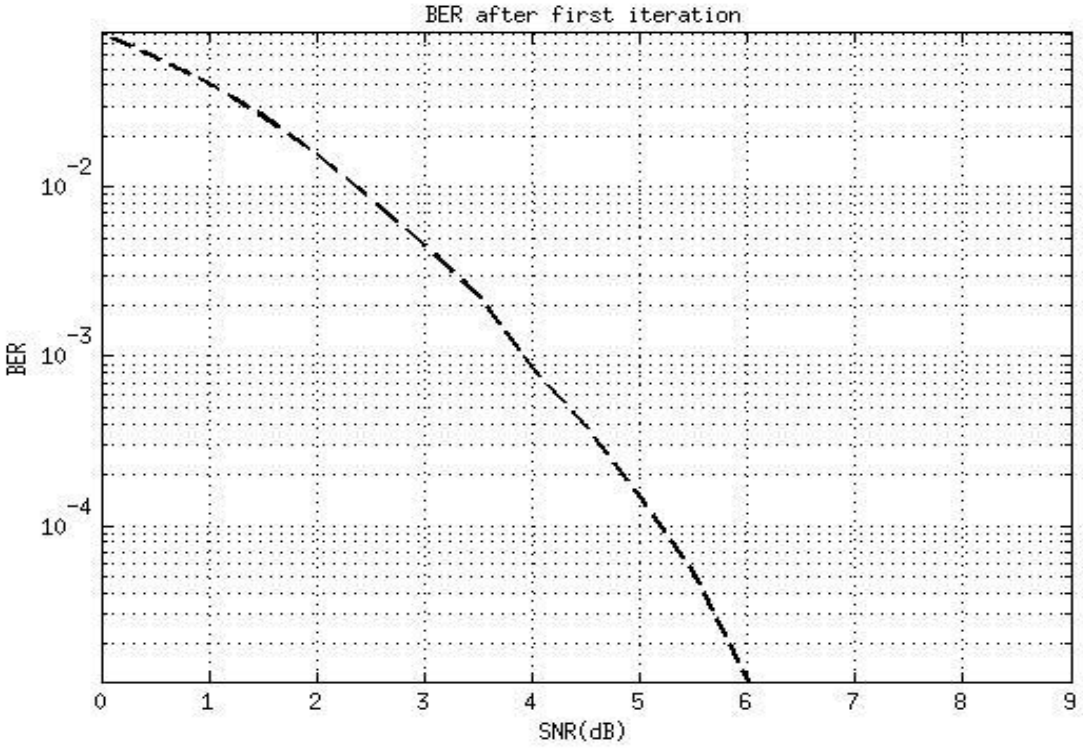


Figure 5: BER after first iteration

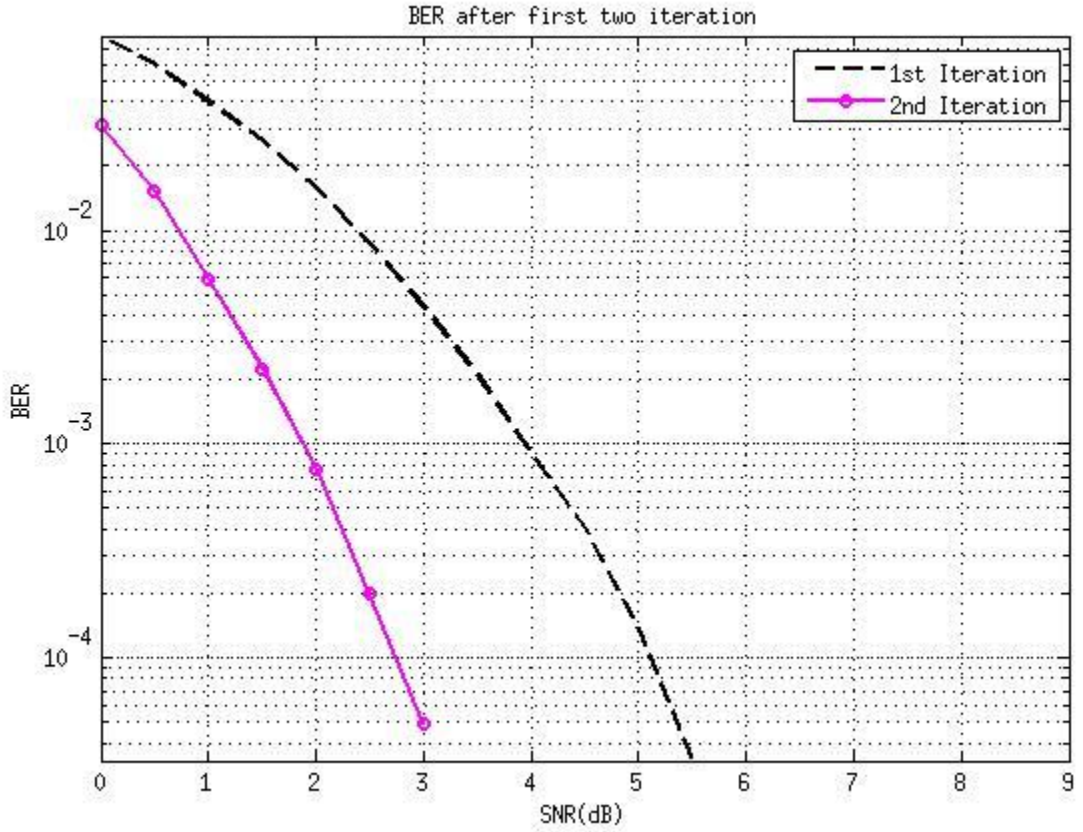


Figure 6: BER after second iteration

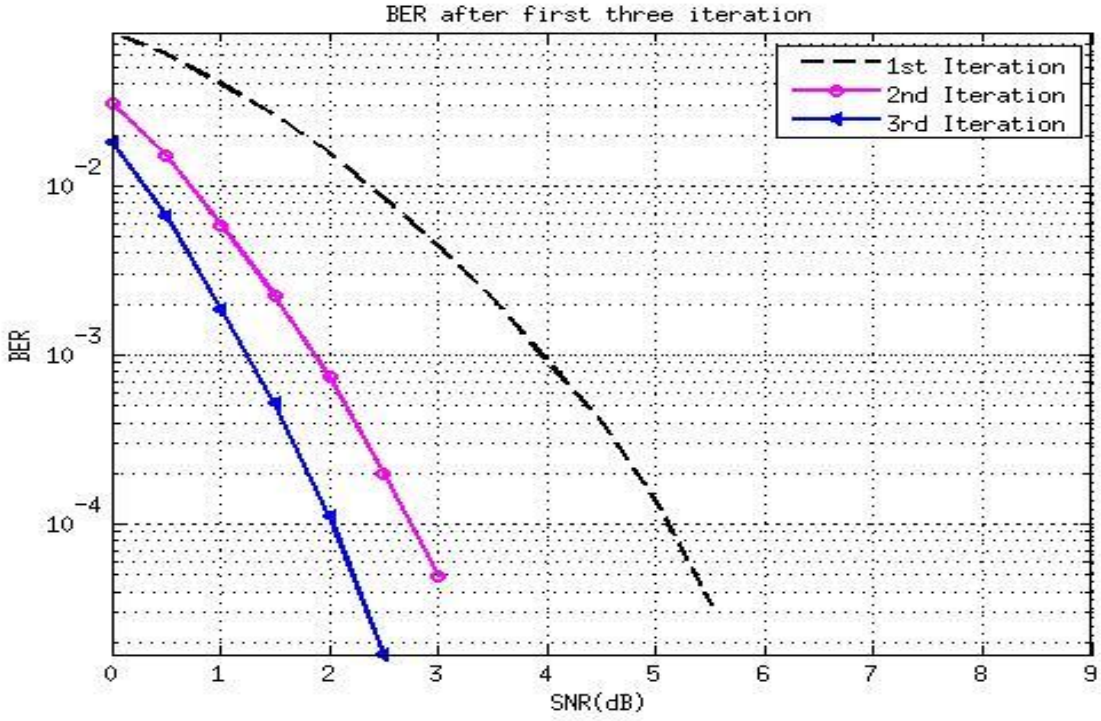


Figure 7: BER after third iteration

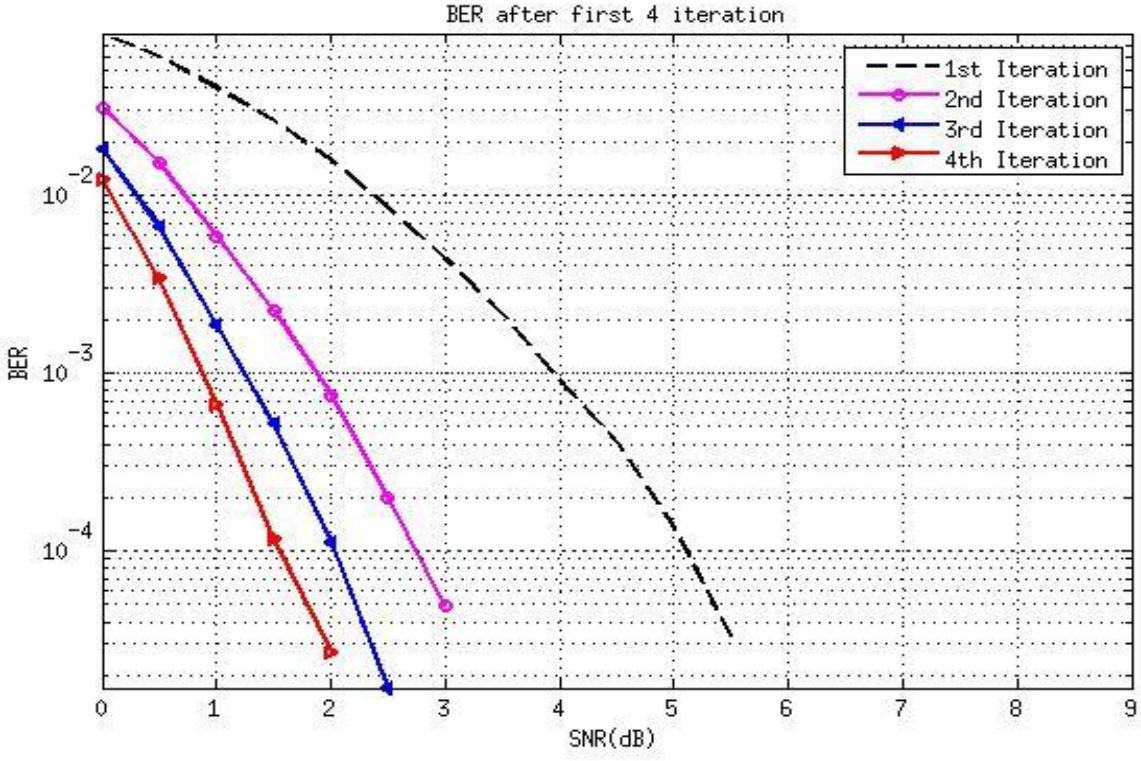


Figure 8: BER after fourth iteration

From these plots we can observe that BER after each iteration decreases for all SNR's. After 4th iteration BER remains almost same, so the BCJR algorithm converges in almost 4 iterations. Also BER become exactly zero after 2 dB in 4 iterations. So Turbo code is much better compared to corresponding parent convolution code.

Plot for comparison of performance of Turbo code and convolution code is shown in Figure: 9. The expression used for theoretical union bound on BER for parent convolution code is following

$$P_e^S = \sum_{j=0}^{\infty} 2^{j+1}(j+1)Q(\sqrt{(j+5)SNR})$$

Superscript S denotes soft decoding. First 10 terms are considered for BER calculation. From the plot it can be concluded that performance of Turbo decoder drastically improves compared to convolution code.

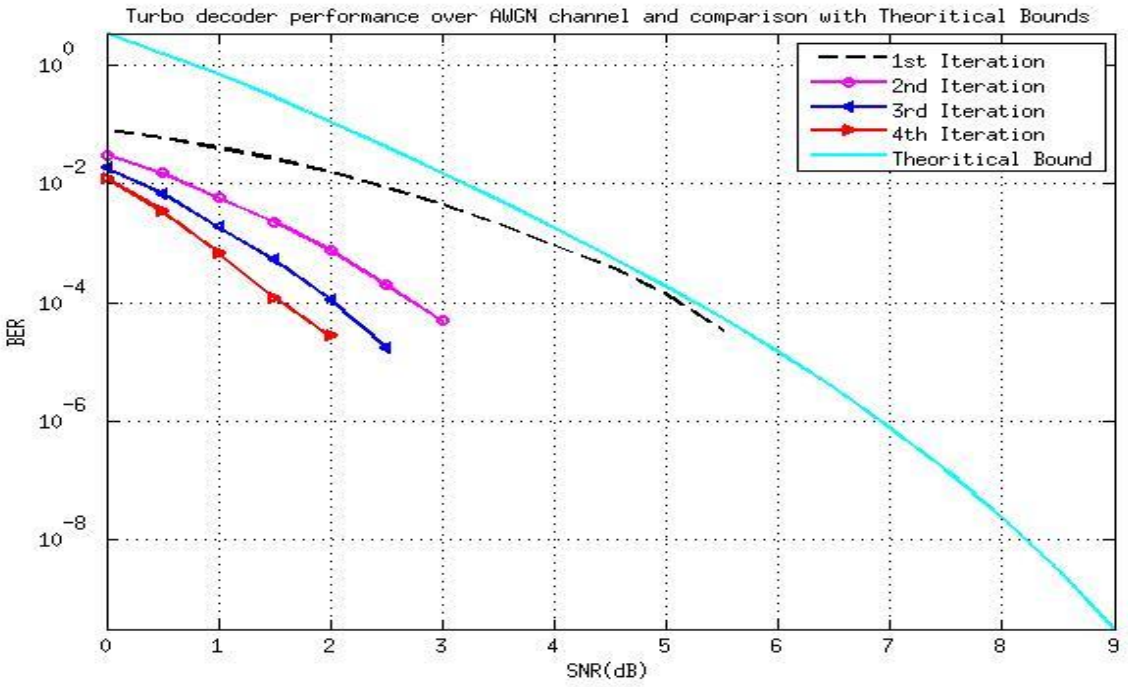


Figure 9: BER comparison with theoretical bounds

The BER values for all 4 iteration and theoretical bounds on convolution are shown in Table1.

SNR(dB)	1st iteration	2nd iteration	3rd iteration	4th iteration	Theoretical BER of Convolution code*
0	0.0810	0.0309	0.0182	0.0122	3.4532
0.5	0.0585	0.0151	0.0067	0.0034	1.6124
1.0	0.0410	0.0059	0.0018	$6.69X10^{-4}$	0.7065

1.5	0.0262	0.0022	$5.18X10^{-4}$	$1.17X10^{-4}$	0.2911
2.0	0.0154	$7.54X10^{-4}$	$1.11X10^{-4}$	$2.7X10^{-5}$	0.1134
2.5	0.0084	$1.98X10^{-4}$	$1.7X10^{-5}$	0	0.0422
3.0	0.0045	$4.90X10^{-5}$	0	0	0.0151
3.5	0.0023	0	0	0	0.0053
4.0	$8.38X10^{-4}$	0	0	0	0.0018
4.5	$3.86X10^{-4}$	0	0	0	$5.8542X10^{-4}$
5.0	$1.46X10^{-4}$	0	0	0	$1.8331X10^{-4}$
5.5	$5.30X10^{-5}$	0	0	0	$5.3848X10^{-5}$
6.0	$1.2X10^{-5}$	0	0	0	$1.4566X10^{-5}$
6.5	0	0	0	0	$3.5593X10^{-6}$
7.0	0	0	0	0	$7.6999X10^{-6}$
7.5	0	0	0	0	$1.4434X10^{-7}$
8.0	0	0	0	0	$2.2907X10^{-8}$
8.5	0	0	0	0	$2.9997X10^{-9}$
9.0	0	0	0	0	$3.1504X10^{-10}$

Table 1: BER values for 4 iteration and comparison with theoretical BER of Corresponding convolution code

****Some value are higher than 1, as this is a union bound***

7. Conclusion

From the simulation results for BER for Turbo code, we can observe that performance of BCJR iterative decoder is much better than the decoding using Viterbi algorithm for the convolution code. BCJR algorithm is able to decode the transmitted bits with 0 probability of error even at very high SNR. Although the time complexity may be a factor in the case of large number of bits, as a significant amount of trellis calculations are involved. To improve the performance of BCJR decoding in terms of time many reduced state BCJR algorithms such as BCJR-M and BCJR-T are proposed, which takes average number of live states per trellis into account and reduces calculation. Hence Turbo code is better in terms of performance and time complexity than other codes.

MATLAB Code:

```
close all;clear all;clc;
N=10^(6);                          %No. of bits(Block length)
X=floor(2*rand(1,N));              %Information bit generation
Interleaver=randperm(N);           %Interleaver(random permutation of first N numbers)
SNRdB=0:0.5:9;                     %SNR in dB
SNR=10.^(SNRdB/10);                %SNR in linear scale
Iteration=4;
ber=zeros(length(SNR),Iteration);   %For storing simulated BER(Each column corresponds to an iteration)
%% Encoding Part
% At all places state 0,1,2,3 represents 00,01,10 and 11 respectively

Input_matrix=2*[0,1;0,1;0,1;0,1]-1; %First column represents input=0 and second column represents
input=1
                                        %Each row represents state 00,10,01 and 11 respectively
Parity_bit_matrix=2*[0,1;1,0;0,1;1,0]-1;      %Parity bits corresponding to inputs of above matrix

current_state=0;              %Initializing first state to 00
P0=zeros(1,N);                %Corresponding parity bits(For encoder RSC-0)
for i=1:N
   [current_state,P0(i)]=parity_bit(current_state,X(i)); %Finding next state and corresponding parity bit
   using current state and input bit
end

X_pi(1:N)=X(Interleaver(1:N));     %Interleaving the input bits for RSC-1 encoder
current_state=0;                   %Initializing first state to 00
P1=zeros(1,N);                     %Corresponding parity bits (For encoder RSC-1)
for i=1:N
   [current_state,P1(i)]=parity_bit(current_state,X_pi(i));
end

mod_code_bit0=2*X-1;       %Modulating Code Bits using BPSK Modulation
mod_code_bit1=2*P0-1;
mod_code_bit2=2*P1-1;

for k=1:length(SNR)            %Simulation starts here

   R0=(sqrt(SNR(k))*mod_code_bit0)+randn(1,N); %Received Codebits Corresponding to input bits
   R1=(sqrt(SNR(k))*mod_code_bit1)+randn(1,N); %Received bits Corresponding to parity bits of RSC-0
   R2=(sqrt(SNR(k))*mod_code_bit2)+randn(1,N); %Received bits Corresponding to parity bits of RSC-1

   R0_pi(1:N)=R0(Interleaver(1:N)); %Interleaved received bits for input to be used by RSC-1

   %% Decoding Part

   BCJR=0;                  %First iteration will be done by BCJR-0
   Apriori=ones(2,N);       %First row for prob. of i/p 0 and second row for prob. of i/p 1
   Apriori=Apriori*0.5;     %Initializing all priory to 1/2

   for iter=1:Iteration     %Iterative process starts here

      if BCJR==0    %If BCJR is 0 then pass R0 and R1 to calculate GAMMA
         GAMMA=gamma_1(Apriori,N,Input_matrix,Parity_bit_matrix,R0,R1,SNR(k));
      else             %If BCJR is 1 then pass R0_pi and R2 to calculate GAMMA
         GAMMA=gamma_1(Apriori,N,Input_matrix,Parity_bit_matrix,R0_pi,R2,SNR(k));
```

```
        end

        ALPHA=alpha_1(GAMMA,N); %Calculation of ALPHA at each stage using GAMMA and ALPHA of
previous stage
        BETA=beta_1(GAMMA,N); %Calculation of BETA at each stage using GAMMA and BETA of previous
stage
        %Calculating LAPPR using ALPHA,BETA and GAMMA
        [P_X0,P_X1,LAPPR_1]=lappr(ALPHA,BETA,GAMMA,N);

        if BCJR==0      %Subtracting intrinsic information from LAPPR
            LAPPR_mod=LAPPR_1-log(Apriori(1,:)./Apriori(2,:))-2*R0;
        else
            LAPPR_mod=LAPPR_1-log(Apriori(1,:)./Apriori(2,:))-2*R0_pi;
        end

        decoded_bits=zeros(1,N);
        decoded_bits(LAPPR_mod>0)=1;  %Decoding is done using LAPPR values

        if BCJR==0      %If the decoder is BCJR-0 then
            ber(k,iter)=sum(abs((decoded_bits-X)));   %calculate BER using input X
            lappr_2(1:N)=LAPPR_mod(Interleaver(1:N)); %Interleave the LAPPR values and pass to BCJR-1
            LAPPR_mod=lappr_2;
        else            %If the decoder is BCJR-1 then
            ber(k,iter)=sum(abs((decoded_bits-X_pi))); %calculate BER using input X_pi
           lappr_2(Interleaver(1:N))=LAPPR_mod(1:N);%Re-interleave the LAPPR values and pass to BCJR-0
            LAPPR_mod=lappr_2;
        end

        Apriori(1,1:N)=1./(1+exp(LAPPR_mod));                 %Priory corresponding to input 0
        Apriori(2,1:N)=exp(LAPPR_mod)./(1+exp(LAPPR_mod)); %Priory corresponding to input 1

        BCJR=~BCJR;   %Changing the state of the decoder for the next iteration
    end             %One iteration ends here
end
ber=ber/N;
figure;
%% Plots for simulated BER
semilogy(SNRdB,ber(:,1),'k--','linewidth',2.0);
hold on
semilogy(SNRdB,ber(:,2),'m-o','linewidth',2.0);
hold on
semilogy(SNRdB,ber(:,3),'b-<','linewidth',2.0);
hold on
semilogy(SNRdB,ber(:,4),'c->','linewidth', 2.0);
hold on
%% Theoritical expression for BER
BER=zeros(1,length(SNR));
for j=1:1:10
    BER=BER+(2^j)*(j)*qfunc(sqrt((j+4)*SNR));
end
semilogy(SNRdB,BER,'k-','linewidth',2.0)
title('Turbo Decoder performance over AWGN channel for BPSK modulated symbols');
xlabel('SNR(dB)');ylabel('BER');
legend('1st Iteration','2nd Iteration','3rd Iteration','4th Iteration','Theoritical Bound');
grid on
axis tight
```

Functions Used

1) gamma_1

```
%This function uses Apriori probabilities, Input matrix, Parity Bit Matrix,
%and received code bits (R0 and R1) and returns GAMMA probabilities at all
%stages. The formula used for calculating gamma at a particular stage and
%given input i (0 or 1) is following
%GAMMA(m',m)=P(X=i) * P(S(k+1)=m|S(k)=m',X=i) * P(R0(k) R1(k) | S(k+1)=m, S(k)=m', X=i))
%Due to the second term only 8 probabilities will survive corresponding to
%8 transitions of Trellis. GAMMA is constructed in following way:
%At each stage(fixed k), there are 2 columns, 1st for i/p 0 and 2nd for i/p
%1. The 4 entries for 1st columns are trasitions from 00 to 00, 10 to 01,
%01 to 10, 11 to 11 respectively and for 2nd column transitions from 00 to
%10, 10 to 11, 01 to 00, 11 to 01 respectively

function [g]=gamma_1(Apriori,N,Input_matrix,Parity_bit_matrix,R0,R1,SNR)

   g=zeros(4,2*N);
   for i=1:N
      T1=((R0(i)-sqrt(SNR)*Input_matrix(1,1))^2)+((R1(i)-sqrt(SNR)*Parity_bit_matrix(1,1))^2);
      T2=((R0(i)-sqrt(SNR)*Input_matrix(1,2))^2)+((R1(i)-sqrt(SNR)*Parity_bit_matrix(1,2))^2);
      T3=((R0(i)-sqrt(SNR)*Input_matrix(2,1))^2)+((R1(i)-sqrt(SNR)*Parity_bit_matrix(2,1))^2);
      T4=((R0(i)-sqrt(SNR)*Input_matrix(2,2))^2)+((R1(i)-sqrt(SNR)*Parity_bit_matrix(2,2))^2);
      T5=((R0(i)-sqrt(SNR)*Input_matrix(3,1))^2)+((R1(i)-sqrt(SNR)*Parity_bit_matrix(3,1))^2);
      T6=((R0(i)-sqrt(SNR)*Input_matrix(3,2))^2)+((R1(i)-sqrt(SNR)*Parity_bit_matrix(3,2))^2);
      T7=((R0(i)-sqrt(SNR)*Input_matrix(4,1))^2)+((R1(i)-sqrt(SNR)*Parity_bit_matrix(4,1))^2);
      T8=((R0(i)-sqrt(SNR)*Input_matrix(4,2))^2)+((R1(i)-sqrt(SNR)*Parity_bit_matrix(4,2))^2);

      g(1,2*i-1)= Apriori(1,i)*(1/(2*pi))*exp(-0.5*T1);      %Current State=0(00), i/p=0
      g(1,2*i)=   Apriori(2,i)*(1/(2*pi))*exp(-0.5*T2);      %Current State=0(00), i/p=1
      g(2,2*i-1)= Apriori(1,i)*(1/(2*pi))*exp(-0.5*T3);      %Current State=2(10), i/p=0
      g(2,2*i)=   Apriori(2,i)*(1/(2*pi))*exp(-0.5*T4);      %Current State=2(10), i/p=1
      g(3,2*i-1)= Apriori(1,i)*(1/(2*pi))*exp(-0.5*T5);      %Current State=1(01), i/p=0
      g(3,2*i)=   Apriori(2,i)*(1/(2*pi))*exp(-0.5*T6);      %Current State=1(01), i/p=1
      g(4,2*i-1)= Apriori(1,i)*(1/(2*pi))*exp(-0.5*T7);      %Current State=3(11), i/p=0
      g(4,2*i)=   Apriori(2,i)*(1/(2*pi))*exp(-0.5*T8);      %Current State=3(11), i/p=1
   end

end
```

---

2) alpha_1

```
%This function calculates ALPHA probabilities at each stage for all states,
%using GAMMA probabilities already calculated. Uses recursion formula for
%alpha to calculate alpha of next stage. Each column is for states 00,10,01
%and 11 respectively. As we move ahead in the block Alpha will become very
%less because 1st term in calculation of gamma can we very less(of the order
%of 10^(-15)) so alpha will keep on becoming even lesser and lesser. And
%after certain recursion it will become exactly 0. So to avoid that we can
%multiply each alpha by 10^(-20) at a stage where they all become less than
%10^(-20). As we need alpha in calculation of LAPPR. So scaling won't affect the ratio

function [ALPHA]=alpha_1(GAMMA,N)
```

```
    ALPHA=zeros(4,N);

    %Initialization of alpha assuming first state to be 00
    ALPHA(1,1)=1;ALPHA(2,1)=0;ALPHA(3,1)=0;ALPHA(4,1)=0;

    j=1;
    for i=2:N
        ALPHA(1,i)=((GAMMA(1,j)*ALPHA(1,i-1))+(GAMMA(3,j+1)*ALPHA(3,i-1)));
        ALPHA(2,i)=((GAMMA(3,j)*ALPHA(3,i-1))+(GAMMA(1,j+1)*ALPHA(1,i-1)));
        ALPHA(3,i)=((GAMMA(2,j)*ALPHA(2,i-1))+(GAMMA(4,j+1)*ALPHA(4,i-1)));
        ALPHA(4,i)=((GAMMA(4,j)*ALPHA(4,i-1))+(GAMMA(2,j+1)*ALPHA(2,i-1)));
        j=j+2;

        if (ALPHA(1,i)<10^(-20) && ALPHA(2,i)<10^(-20) &&...
                ALPHA(3,i)<10^(-20) && ALPHA(4,i)<10^(-20) )
            ALPHA(:,i)=10^(20)*ALPHA(:,i);   %Scaling Alpha if became very less
        end
    end

end
```

---

3) beta_1

```
%This function calculates BETA probabilities at each stage for all states,
%using GAMMA probabilities already calculated. Uses recursion formula for
%beta to calculate beta of previous stage. Each column is for states 00,10,01
%and 11 respectively. As we move backward in the block Beta will become very
%less because 1st term in calculation of gamma can we very less(of the order
%of 10^(-15)) so beta will keep on becoming even lesser and lesser. And
%after certain recursion it will become exactly 0. So to avoid that we can
%multiply each beta by 10^(-20) at a stage where they all become less than
%10^(-20). As we need beta in calculation of LAPPR. So scaling wont affect the ratio

function [BETA]=beta_1(GAMMA,N)

    BETA=zeros(4,N);
    %Initialization assuming the final stage to be 00
    BETA(1,N)=1;BETA(2,N)=0;BETA(3,N)=0;BETA(4,N)=0;

    j=2*N-1;
    for i=N-1:-1:1
      BETA(1,i)=(GAMMA(1,j)*BETA(1,i+1))+(GAMMA(1,j+1)*BETA(2,i+1));
      BETA(2,i)=(GAMMA(2,j)*BETA(3,i+1))+(GAMMA(2,j+1)*BETA(4,i+1));
      BETA(3,i)=(GAMMA(3,j)*BETA(2,i+1))+(GAMMA(3,j+1)*BETA(1,i+1));
      BETA(4,i)=(GAMMA(4,j)*BETA(4,i+1))+(GAMMA(4,j+1)*BETA(3,i+1));
      j=j-2;

      if (BETA(1,i)<10^(-20) && BETA(2,i)<10^(-20) &&...
              BETA(3,i)<10^(-20) && BETA(4,i)<10^(-20) )
          BETA(:,i)=10^(20)*BETA(:,i);        %Scaling beta if became very less
      end
    end

end
```

---

4) lappr_1

```
%This function calculates LAPPR at each stage using ALPHA, BETA and GAMMA
%probabilities. Returns P(X(k)=0), P(X(k)=1) and LAPPR(k) for all k's

function [p_x0,p_x1,lappr_1]=lappr(ALPHA,BETA,GAMMA,N)
   p_x0=zeros(1,N);
   p_x1=zeros(1,N);
   lappr_1=zeros(1,N);

   for i=1:N
      p_x1(i)=(ALPHA(1,i)*GAMMA(1,2*i)*BETA(2,i))+(ALPHA(2,i)*GAMMA(2,2*i)*BETA(4,i))+...
         (ALPHA(3,i)*GAMMA(3,2*i)*BETA(1,i))+(ALPHA(4,i)*GAMMA(4,2*i)*BETA(3,i));

      p_x0(i)=(ALPHA(1,i)*GAMMA(1,2*i-1)*BETA(1,i))+(ALPHA(2,i)*GAMMA(2,2*i-1)*BETA(3,i))+...
         (ALPHA(3,i)*GAMMA(3,2*i-1)*BETA(2,i))+(ALPHA(4,i)*GAMMA(4,2*i-1)*BETA(4,i));

      lappr_1(i)=log(p_x1(i)/p_x0(i));

   end

end
```

---

5) parity_bits

```
%Function for finding next state and corresponding parity bit using the
%TRELLIS Diagram. State 0,1,2,3 represents state 00,01,10 and 11
%respectively
function [next_state,parity_bit]=parity_bit(current_state,input)
   if(current_state==0)
      if(input==0)
         next_state=0;parity_bit=0;
      else
         next_state=2;parity_bit=1;
      end
   end

   if(current_state==1)
      if(input==0)
         next_state=2;parity_bit=0;
      else
         next_state=0;parity_bit=1;
      end
   end

   if(current_state==2)
      if(input==0)
         next_state=1;parity_bit=1;
      else
         next_state=3;parity_bit=0;
      end
   end

   if(current_state==3)
      if(input==0)
```

```
            next_state=3;parity_bit=1;
        else
            next_state=1;parity_bit=0;
        end
    end
end
```