Turbo encoder and Decoder implementation

Shiv Prakash, Y9551
Shouvik Ganguly, Y9558

November 15, 2012

Term Paper
EE624

Instructor:
Prof. Aditya K. Jagannatham
DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

Introduction

Turbo codes use the systematic generator matrix $[1, \frac{1+D+D^2}{1+D^2}]$

But the major difference from Convolutional Codes is that there is a random interleaver, which produces a random permutation of the input sequence. Then, both the original input sequence and the interleaved sequence are encoded using the above generator matrix. Finally, the encoded version consists of the original input sequence, the encoded version of the original sequence and the encoded version of the interleaved sequence. Hence it is a code of rate $\frac{1}{3}$.

Encoding

$C_1(D) = X(D)(\frac{1+D+D^2}{1+D^2})$

$\implies C_1(D)(1+D^2) = X(D)(1+D+D^2)$

$\implies C_1^j + C_1^{j-2} = X^j + X^{j-1} + X^{j-2}$

$\implies C_1^j = X^j + X^{j-1} + X^{j-2} + C_1^{j-2}$ (since addition is modulo 2 here).

Using this equation, recursive encoding can be easily done.

Decoding

Due to the presence of the random interleaver, optimal ML decoding is not possible in this case.

So we use two decoders which exchange information and together perform the decoding ("Turbo Principle")

The Algorithm used is the MODIFIED BCJR Algorithm

Let N symbols be sent. Using the encoded symbols (3N), we first modulate them using BPSK (essentially,1's remain 1's and 0's become (-1)'s).

Then we add noise to get $\bar{R}_0^{N-1} = [R_0^0 R_1^0 R_2^0 R_0^1 R_1^1 R_2^1 ...]$

The Algorithm computes the log likelihood ratio for each X^i based on the $\bar{R}_0^{N-1}$

Let S_k denote the state of the trellis at the k^{th} stage, starting from $k = 0$.
$S_k = 0$, 1, 2 or 3, corresponding to the states 00, 01, 10 and 11 respectively.

$P(X^k = i|\bar{R}_0^{N-1}) = \frac{1}{P(\bar{R}_0^{N-1})} P(X^k = i, \bar{R}_0^{N-1})$

$= \frac{1}{P(\bar{R}_0^{N-1})} \sum_{m=0}^{M-1} \sum_{m'=0}^{M-1} P(X^k = i, S_k = m', S_{k+1} = m, \bar{R}_0^{N-1})$

$P(X^k = i, S_k = m', S_{k+1} = m, \bar{R}_0^{k-1}, R^k, \bar{R}_{k-1}^{N-1}) =$

$P[\bar{R}_{k+1}^{N-1}|X^k = i, S_k = m', S_{k+1} = m, \bar{R}_0^{k-1}, R^k] P(X^k = i, S_k = m', S_{k+1} = m, \bar{R}_0^{k-1}, R^k)$

$= P(\bar{R}_{k+1}^{N-1}|S_{k+1} = m) P(S_k = m', \bar{R}_0^{k-1}, X_k = i, S_{k+1} = m, R^k)$

$= P(\bar{R}_{k+1}^{N-1}|S_{k+1} = m) P(X_k = i, S_{k+1} = m, R^k|S_k = m', \bar{R}_0^{k-1}) P(S_k = m', \bar{R}_0^{k-1})$

$= \beta_k(m).\alpha_k(m')\gamma_k^i(m', m)$

where $\beta_k(m) = P(\bar{R}_{k+1}^{N-1}|S_{k+1} = m)$,

$\alpha_k(m') = P(S_k = m', \bar{R}_0^{k-1})$,

$\gamma_k^i(m', m) = P(X_k = i, S_{k+1} = m, R^k|S_k = m')$

$\therefore P(X^k = i|\bar{R}_0^{N-1}) = \frac{1}{P(\bar{R}_0^{N-1})} \sum_{m=0}^{M-1} \sum_{m'=0}^{M-1} \beta_k(m).\alpha_k(m')\gamma_k^i(m', m)$

$$\therefore \Lambda^k = \text{LAPPR} = log \left(\frac{\sum_{m=0}^{M-1} \sum_{m'=0}^{M-1} \beta_k(m).\alpha_k(m')\gamma_k^1(m', m)}{\sum_{m=0}^{M-1} \sum_{m'=0}^{M-1} \beta_k(m).\alpha_k(m')\gamma_k^0(m', m)} \right) \text{——(1)}$$

$\alpha_k(m') = P(S_k = m', \bar{R}_0^{k-1})$

$= \sum_{m''=0}^{M-1} P(S_{k-1} = m'', S_k = m', \bar{R}_0^{k-1})$

$$= \sum_{m''=0}^{M-1} \sum_{i=0}^{1} P(S_{k-1} = m'', X^{k-1} = i, S_k = m', \bar{R}_0^{k-2}, R^{k-1})$$
$$= \sum_{m''=0}^{M-1} \sum_{i=0}^{1} P(S_{k-1} = m'', S_k = m', X^{k-1} = i, R^{k-1} | \bar{R}_0^{k-2}, S_{k-1} = m'') P(\bar{R}_0^{k-2}, S_{k-1} = m'')$$
$$= \sum_{m''=0}^{M-1} \sum_{i=0}^{1} P(S_{k-1} = m'', S_k = m', X^{k-1} = i, R^{k-1} | S_{k-1} = m'') \alpha_{k-1}(m'')$$
$$= \sum_{m''=0}^{M-1} \sum_{i=0}^{1} \gamma_{k-1}^{i}(m'', m') \alpha_{k-1}(m'') \text{ ——(2)}$$

Also, $\alpha_0(m') = P(S_0 = m') = 1$ if $m' = 0$
$= 0$ otherwise

Similarly, $\beta_k(m) = \sum_{m'=0}^{M-1} \sum_{i=0}^{1} \gamma_{k+1}^{i}(m, m') \beta_{k+1}(m')$ ——(3)

Also, $\beta_{N-2}(m) = P(R^{N-1} | S_{N-1} = m)$

and $\beta_{N-1}(m) = 1 \ \forall m$

$\gamma_k^i(m', m) = P(X^k = i, S_{k+1} = m, R^k | S_k = m')$

$= \frac{P(X_k=i, S_{k+1}=m, S_k=m', R^k)}{P(S_k=m')}$

$= \frac{P(R^k | S_{k+1}=m, S_k=m', X^k=i).P(S_{k+1}=m | S_k=m', X^k=i).P(S_k=m', X^k=i)}{P(S_k=m')}$

$= P(R^k | S_{k+1} = m, S_k = m', X^k = i).P(S_{k+1} = m | S_k = m', X^k = i).P(X^k = i)$

The first of these factors depends on the channel, the second factor depends on the structure of the trellis and the third is just the apriori probibility.

Another point is that when we speak of probability for R^k, we actually mean the likelihood, i.e. the probability density function.

After each iteration of the above algorithm, we replace the prior probabilities by the posterior probabilities just obtained, using the relations

$P(X^k = 1) = \frac{e^{\Lambda^k}}{1+e^{\Lambda^k}}$
and
$P(X^k = 0) = \frac{e^{-\Lambda^k}}{1+e^{-\Lambda^k}}$

We have implemented eqns(1), (2) and (3) using matrix multiplications. In each case, we treated α as a row vector and β as a column vector, and got the sum as $\tilde{\alpha}\gamma\tilde{\beta}$ in (1), $\tilde{\alpha}\gamma$ in (2), $\gamma\tilde{\beta}$ in (3).

For this case, using the trellis structure, we get the second factor in the final expression for γ as 1 when $(m = m' = 0, i = 0)$ or $(m = m' = 3, i = 0)$ or $(m = 2, m' = 1, i = 0)$ or $(m = 1, m' = 2, i = 0)$ or $(m = 2, m' = 0, i = 1)$ or $(m = 3, m' = 2, i = 1)$ or $(m = 0, m' = 1)$ or $(m = 1, m' = 3)$, and 0 otherwise.

We have continued the iterations until the minimum absolute value of the LAPPR is less than 10^{-5}, in order to eliminate possible confusion.

Also, since the values of α become progressively less while computing them recursively, we multiply them by a constant in order to keep the values stable.

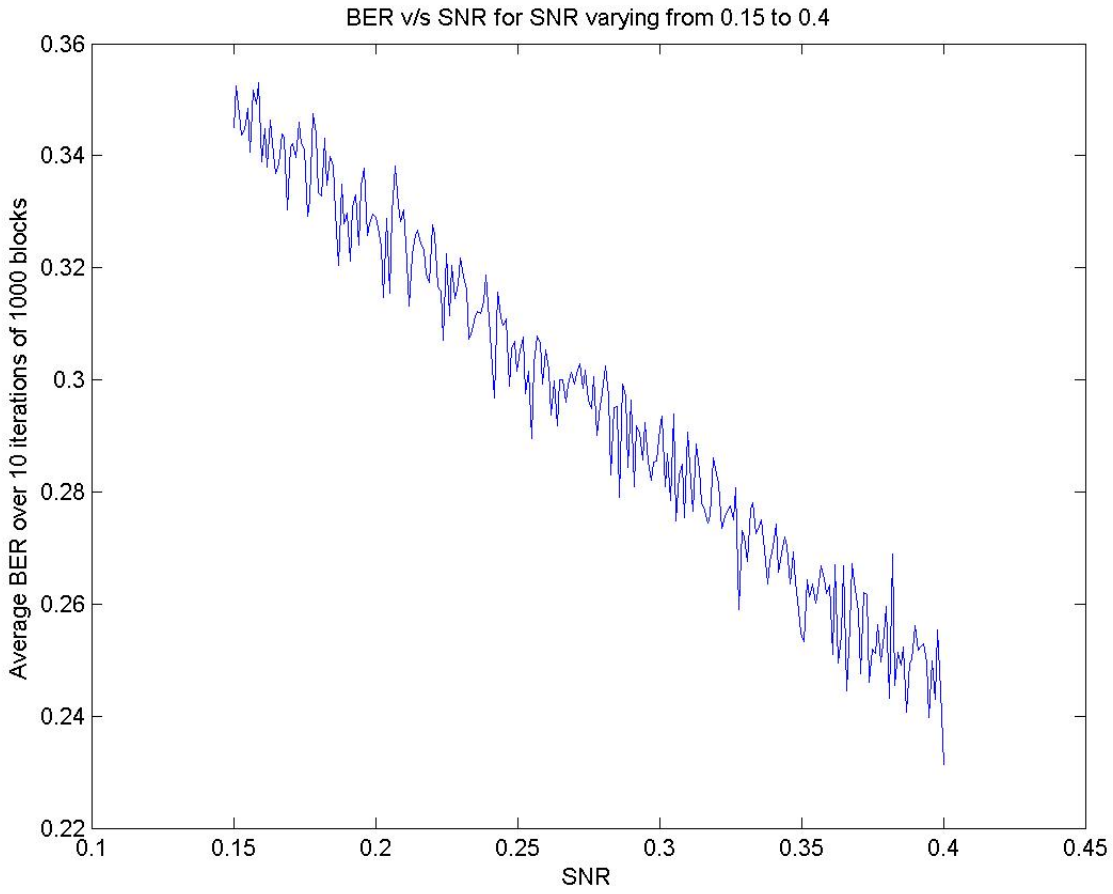
Depending on the SNR, the constant is different. We have used 3 different constants for 3 different SNR regions.

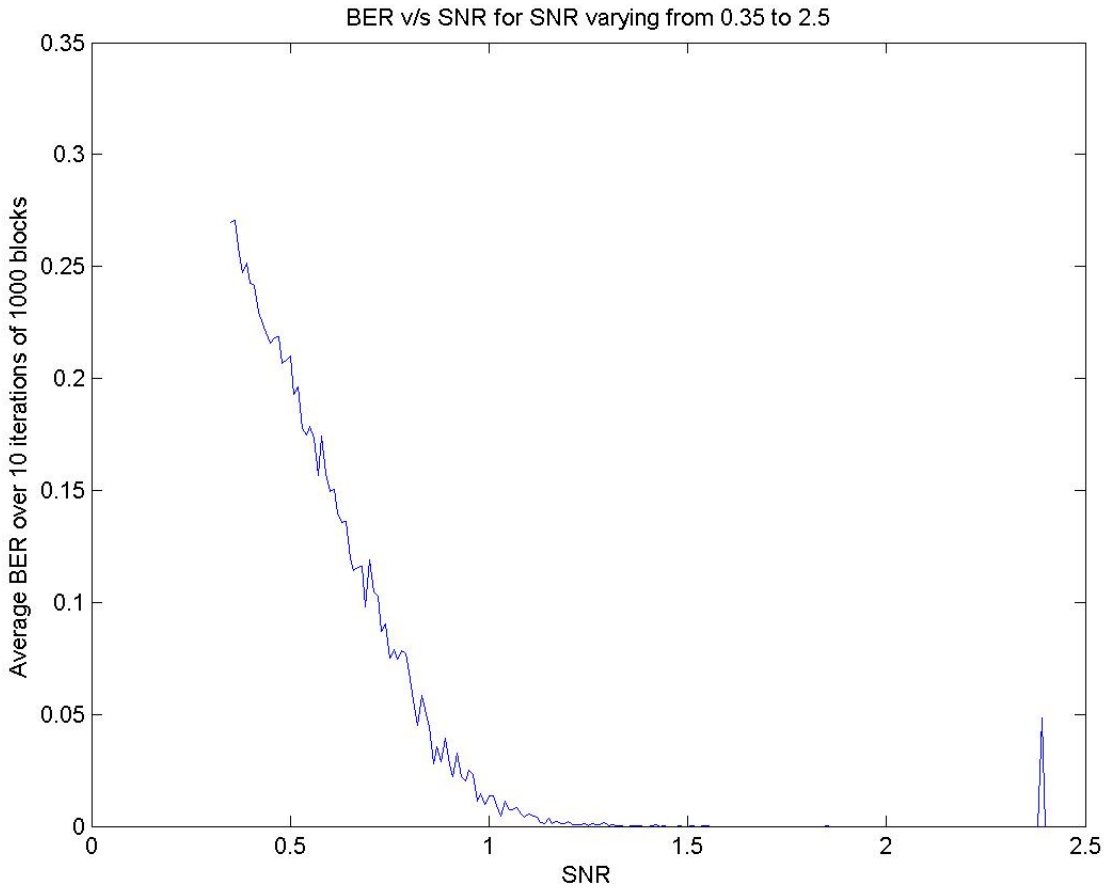
The results obtained are shown below.

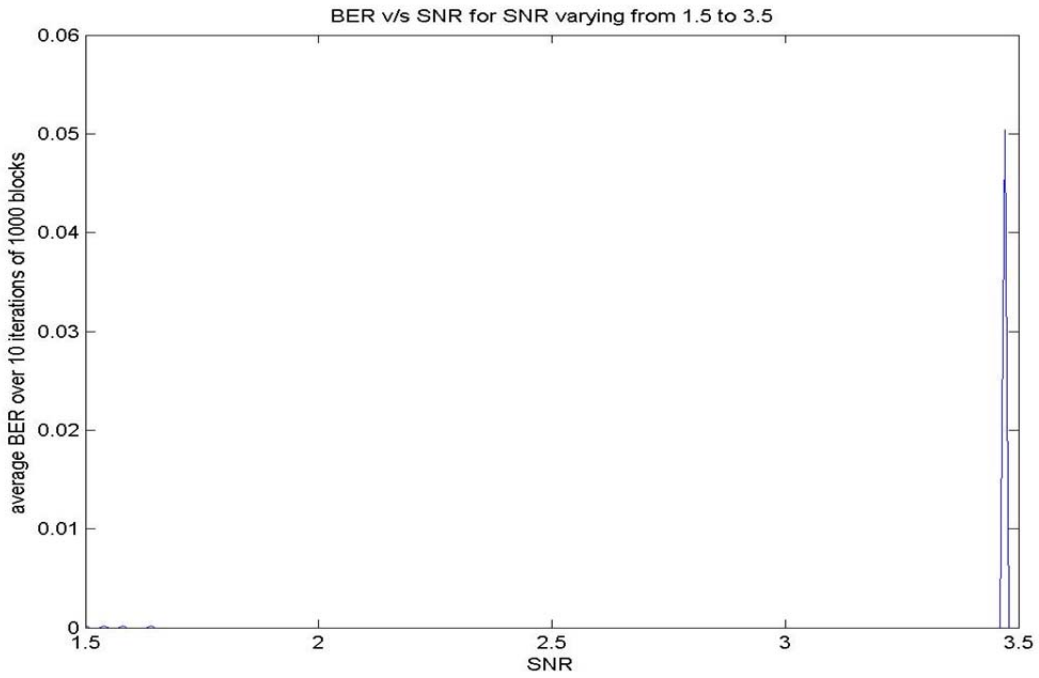
Table 1: Multiplicative Constant

SNR range	Constant
.15 to .4	25
.35 to 2.5	19.8
1.5 to 3.5	17

We took block length of 1000 and ran the code 10 times and plotted the average BER with SNR.


BER v/s SNR for SNR varying from 0.15 to 0.4
Average BER over 10 iterations of 1000 blocks
SNR
0.36
0.34
0.32
0.3
0.28
0.26
0.24
0.22
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45


BER v/s SNR for SNR varying from 0.35 to 2.5
Average BER over 10 iterations of 1000 blocks
SNR
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0
0.5
1
1.5
2
2.5



We see that for SNR greater than 1.5, the BER is almost 0 everywhere.

Code

```
clc
clear all;
close all

N = 1000;
energy = 1.5:.01:3.5;

BER = zeros(size(energy));
avgerr=zeros(1,10);




for k = 1:length(energy)
    tic

    for l=1:10
        X = randi([0 1],1,N);
        C0 = zeros(size(X));
        C1 = C0;
        C2 = C1;
        C0 = X;

        pii = randperm(N);

        Xpii(1:N) = X(pii);
```

```
        C1(1) = X(1);
        C1(2) = mod(X(2)+X(1),2);
        C2(1) = Xpii(1);
        C2(2) = mod(Xpii(2)+Xpii(1),2);

        for m = 3:N
            C1(m) = mod(X(m)+X(m-1)+X(m-2)+C1(m-2),2);
            C2(m) = mod(Xpii(m)+Xpii(m-1)+Xpii(m-2)+C2(m-2),2);
        end

        C(1:3:(3*N - 2)) = C0(1:N);
        C(2:3:(3*N - 1)) = C1(1:N);
        C(3:3:(3*N )) = C2(1:N);
        R = (C-(C==0))+((1/sqrt(energy(k)))*(randn(1,3*N)));

        %decoding starts here

        R_syst = R(1:3:3*N-2);

        R1(1:2:2*N-1) = R_syst ;
        R1(2:2:2*N) = R(2:3:3*N-1);

        R2(1:2:2*N-1) = R_syst(pii);
        R2(2:2:2*N) = R(3:3:3*N);

        lambda = decode_turbo(R1,zeros(1,N),energy(k));
        lambda1 = zeros(1,N);
        while ((min(min(abs(lambda),abs(lambda1))))<=1e-5)
            lambda = lambda1;

            lambda_new = decode_turbo(R1,lambda,energy(k));

            lambda1(pii) = decode_turbo(R2,lambda_new(pii),energy(k));


        end

        X_decoded = lambda1>0;
        avgerr(l)= (sum(X_decoded~=X))/N;
    end
    BER(k) = mean(avgerr);
    toc
end

plot(energy,BER);
```

The function decode_turbo

```
function[lambda_new] = decode_turbo(R,lambda,energy)
N = length(lambda);

temp=17;
sigma = 1/sqrt(energy);
gamma0 = zeros(N+1,4,4);
gamma1 = gamma0;
alpha = zeros(N+1,4);
beta = alpha;
```

```
gamma0(2:N+1,1,1) = temp*((1/(2*pi*sigma))*exp(-0.5*(((R(1:2:2*N-1)-((-
1)*ones(1,N))).^2)+((R(2:2:2*N)-(-
1)*ones(1,N)).^2))/(sigma^2))).*(ones(1,N)+exp(lambda(1:N)));
gamma0(2:N+1,3,2) = temp*((1/(2*pi*sigma))*exp(-0.5*(((R(1:2:2*N-1)-(-
1)*ones(1,N)).^2)+((R(2:2:2*N)-
(+1)*ones(1,N)).^2))/(sigma^2))).*(ones(1,N)+exp(lambda(1:N)));
gamma0(2:N+1,2,3) = temp*((1/(2*pi*sigma))*exp(-0.5*(((R(1:2:2*N-1)-(-
1)*ones(1,N)).^2)+((R(2:2:2*N)-(-
1)*ones(1,N)).^2))/(sigma^2))).*(ones(1,N)+exp(lambda(1:N)));
gamma0(2:N+1,4,4) = temp*((1/(2*pi*sigma))*exp(-0.5*(((R(1:2:2*N-1)-(-
1)*ones(1,N)).^2)+((R(2:2:2*N)-
(+1)*ones(1,N)).^2))/(sigma^2))).*(ones(1,N)+exp(lambda(1:N)));

gamma1(2:N+1,1,3) = temp*((1/(2*pi*sigma))*exp(-0.5*(((R(1:2:2*N-1)-
(+1)*ones(1,N)).^2)+((R(2:2:2*N)-
(+1)*ones(1,N)).^2))/(sigma^2))).*(exp(lambda(1:N))./(ones(1,N)+exp(lambda
(1:N)));
gamma1(2:N+1,3,4) = temp*((1/(2*pi*sigma))*exp(-0.5*(((R(1:2:2*N-1)-
(+1)*ones(1,N)).^2)+((R(2:2:2*N)-(-
1)*ones(1,N)).^2))/(sigma^2))).*(exp(lambda(1:N))./(ones(1,N)+exp(lambda(1
:N)));
gamma1(2:N+1,2,1) = temp*((1/(2*pi*sigma))*exp(-0.5*(((R(1:2:2*N-1)-
(+1)*ones(1,N)).^2)+((R(2:2:2*N)-
(+1)*ones(1,N)).^2))/(sigma^2))).*(exp(lambda(1:N))./(ones(1,N)+exp(lambda
(1:N)));
gamma1(2:N+1,4,2) = temp*((1/(2*pi*sigma))*exp(-0.5*(((R(1:2:2*N-1)-
(+1)*ones(1,N)).^2)+((R(2:2:2*N)-(-
1)*ones(1,N)).^2))/(sigma^2))).*(exp(lambda(1:N))./(ones(1,N)+exp(lambda(1
:N)));

alpha(1,:) = [0,0,0,0];
alpha(2,:) = [1,0,0,0];

gamma_add = gamma0 + gamma1;

beta(N,1) = (1/(2*pi*sigma))*((1/(1+exp(lambda(N))))*(exp(-0.5*(((R(2*N-1)-
(-1))^2) + ((R(2*N)-(-1))^2))/(sigma^2))) + ((1/(1+exp(-lambda(N))))*exp(-
0.5*(((R(2*N-1)-(+1))^2) + ((R(2*N)-(+1))^2))/(sigma^2))));
beta(N,2) = beta(N,1);
beta(N,3) = (1/(2*pi*sigma))*((1/(1+exp(lambda(N))))*(exp(-0.5*(((R(2*N-1)-
(-1))^2) + ((R(2*N)-(+1))^2))/(sigma^2))) + ((1/(1+exp(-lambda(N))))*exp(-
0.5*(((R(2*N-1)-(+1))^2) + ((R(2*N)-(-1))^2))/(sigma^2))));
beta(N,4) = beta(N,3);
beta(N+1,:) = (1e10)*ones;
for k = 1:N-1
    k1 = k + 1;
    alpha0 = alpha(k1,:);
    beta0 = beta(N-k1+2,:)';
    gamma_add0 = squeeze(gamma_add(k1,:,:));
    gamma_add1 = squeeze(gamma_add(N-k1+2,:,:));

    alpha(k1+1,:) = alpha0*gamma_add0;
    beta(N-k1+1,:) = (gamma_add1*beta0)';

end

lambda_new = zeros(1,N);
```

```
for k = 2:N+1
    alpha1 = alpha(k,:);
    beta1 = (beta(k,:))';
    gamma10 = squeeze(gamma0(k,:,:));
    gamma11 = squeeze(gamma1(k,:,:));
    lambda_new(k-1) =
log(((alpha1*gamma11)*beta1)/((alpha1*gamma10)*beta1));

end
```

For both encoding and decoding, when the number of iterations and the length of the vector 'energy' remain fixed, the computational complexity goes as O(N) where N is the block length.

Hence the complexity is roughly, k*N*n*p, where k is a constant, N is the block length, n is the number of times we run it for the same energy value (10 in our case) and p is the length of the energy vector