LDPC Decoding for 5G NR

CT-216: Introduction to Communication System

End of the Semester Project

Instructor: Prof. Yash Vasavda

Group : 32

Group Members:

Name	**ID**
VANSH PATEL	202301443
VAGH DIVYESH	202301444
PRANSU VADSMIYA	202301445
JAINESH PATEL	202301446
ATIK VOHRA	202301447
SHUBHAM VARMORA	202301450
PREM KUKADIYA	202301452
PRANAV BAVADIYA	202301458
AKSHAT BHATT	202301460
CHOVATIYA AYUSH BIPIN	202301461
JOSHI YESHA SNEHAL	202301462

Contents

1 Introduction

1.1 What is an LDPC Code?

Low-Density Parity-Check (LDPC) codes are a class of linear error-correcting codes that were originally proposed by Robert Gallager in the early 1960s. Although initially overlooked due to their computational complexity, LDPC codes gained renewed interest in the 1990s with the advancement of computing power and have since become one of the most powerful and widely used coding techniques in modern communication systems, including Wi-Fi (IEEE 802.11), DVB-S2, and 5G New Radio (NR).

LDPC codes are known for their near-Shannon-limit error-correcting performance and efficient implementation of iterative decoding algorithms. They use a sparse parity-check matrix to represent constraints among code bits. This sparsity allows for fast and scalable decoding using belief propagation or message-passing algorithms.

2 Hard Decision Decoding

2.1 Hard Decision (Bit-Flipping Algorithm)

This algorithm corrects errors in received codewords by iteratively flipping bits that violate parity-check constraints.

Notations

Let:

- $\mathbf{r}$ = Received noisy codeword (binary vector)
- $\mathbf{H}$ = Parity-check matrix

Step 1: Send Initial Messages

Each VN is loaded with the received bit r_i and sends its current bit value to all connected CNs on the Tanner graph.

Step 2: CN $\rightarrow$ VN

For each CN:

- Compute the modulo-2 sum of all incoming messages.
- If the sum = 0:
  - The parity is satisfied; send the same bit values to connected VNs.
- If the sum $\neq$ 0:
  - The parity constraint fails, send back flipped bit values to the connected VNs.

Step 3: VN $\rightarrow$ CN

Each variable node collects feedback from all connected check nodes and applies majority voting:

- If majority of check nodes suggest flipping, update the bit ($0 \leftrightarrow 1$), this is because VNs act as repetition code.
- Else, It remains unchanged.

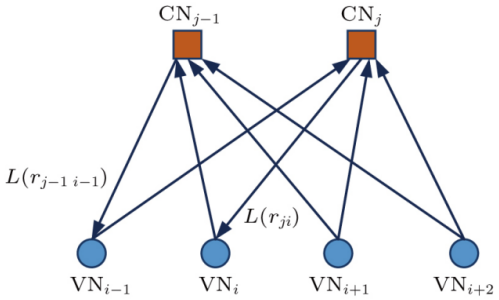


Figure 1: Tanner Graph

Step 4: Termination

Our iterations are terminated if and only if any of the two conditions occurs.

- A valid codeword is found. (if $\mathbf{H} \cdot \mathbf{r} = 0$)
- Maximum number of iterations is reached.

3 Soft Decision Decoding

3.1 Derivation

SISO Decoder SPC(3,2)

Consider codeword as [C1 C2 C3]

$$C_1 = C_2 \oplus C_3$$

$$l_2 = \log \frac{\Pr(C_2 = 0 \mid Y_2)}{\Pr(C_2 = 1 \mid Y_2)}, \quad l_3 = \log \frac{\Pr(C_3 = 0 \mid Y_3)}{\Pr(C_3 = 1 \mid Y_3)}$$

Given p_2 and p_3, what is $\Pr(C_1 = 0 \mid p_2, p_3)$?

$$l_2 = \log \frac{p_2}{1 - p_2}, \quad l_3 = \log \frac{p_3}{1 - p_3}$$

C_1	C_2	C_3
0	0	0
0	1	1
1	0	1
1	1	0

$$p_1 = \Pr(C_1 = 0) = p_2 p_3 + (1 - p_2)(1 - p_3)$$

$$1 - p_1 = \Pr(C_1 = 1) = p_2(1 - p_3) + p_3(1 - p_2)$$

$$p_1 - (1 - p_1) = p_2 p_3 + (1 - p_2)(1 - p_3) - p_2(1 - p_3) - p_3(1 - p_2)$$
$$p_1 - (1 - p_1) = (p_2 - (1 - p_2))(p_3 - (1 - p_3))$$

$$\frac{p_1 - (1 - p_1)}{p_1 + (1 - p_1)} = \frac{p_2 - (1 - p_2)}{p_2 + (1 - p_2)} \cdot \frac{p_3 - (1 - p_3)}{p_3 + (1 - p_3)}$$

$$\frac{1 - \frac{1-p_1}{p_1}}{1 + \frac{1-p_1}{p_1}} = \frac{1 - \frac{1-p_2}{p_2}}{1 + \frac{1-p_2}{p_2}} \cdot \frac{1 - \frac{1-p_3}{p_3}}{1 + \frac{1-p_3}{p_3}}$$

$$l_{\text{ext},1} = \log \frac{p_1}{1 - p_1}$$

$$\Rightarrow \frac{1 - e^{-l_{\text{ext},1}}}{1 + e^{-l_{\text{ext},1}}} = \left(\frac{1 - e^{-l_2}}{1 + e^{-l_2}}\right)\left(\frac{1 - e^{-l_3}}{1 + e^{-l_3}}\right)$$

Check Node Update using Tanh Rule

The check node update in the log-likelihood ratio (LLR) domain uses the following identity:

$$\tanh\left(\frac{l_{\text{ext},1}}{2}\right) = \tanh\left(\frac{l_2}{2}\right) \cdot \tanh\left(\frac{l_3}{2}\right)$$

Sign Rule

$$\text{sign}(l_{\text{ext},1}) = \text{sign}(l_2) \cdot \text{sign}(l_3)$$

Absolute Value Rule

$$\tanh\left(\left|\frac{l_{\text{ext},1}}{2}\right|\right) = \tanh\left(\frac{|l_2|}{2}\right) \cdot \tanh\left(\frac{|l_3|}{2}\right)$$

Log transformation Define the function:

$$f(x) = \log\left(\tanh\left(\frac{x}{2}\right)\right), \quad x > 0$$

This leads to:

$$f(|l_{\text{ext},1}|) = f(|l_2|) + f(|l_3|)$$

Inverse transformation Apply the inverse function:

$$|l_{\text{ext},1}| = f^{-1}\left(f(|l_2|) + f(|l_3|)\right)$$

Final Check Node Update Equation

$$l_{\text{ext},1} = \text{sign}(l_2 \cdot l_3) \cdot f^{-1}\left(f(|l_2|) + f(|l_3|)\right)$$

SISO Decoder Repetition(3,2)

- Message bits: C_1
- Received bits: r_1, r_2, r_3

$$\Pr(C_1 = 0 \mid r_1) = \frac{f(r_1 \mid C_1 = 0)\Pr(C_1 = 0)}{f(r_1)}$$

$$\Pr(C_1 = 1 \mid r_1) = \frac{f(r_1 \mid C_1 = 1)\Pr(C_1 = 1)}{f(r_1)}$$

Posterior Ratio

$$\frac{\Pr(C_1 = 0 \mid r_1)}{\Pr(C_1 = 1 \mid r_1)} = \frac{f(r_1 \mid C_1 = 0)}{f(r_1 \mid C_1 = 1)}$$

$$f(r_1 \mid C_1 = 0) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(r_1-1)^2/(2\sigma^2)}, \quad f(r_1 \mid C_1 = 1) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(r_1+1)^2/(2\sigma^2)}$$

$$\text{LLR} = \log\left(\frac{f(r_1 \mid C_1 = 0)}{f(r_1 \mid C_1 = 1)}\right) = \frac{2}{\sigma^2} \cdot r_1 \quad \text{(intrinsic)}$$

Output LLR

$$\text{LLR} = \log\left(\frac{\Pr(C_1 = 0 \mid r_1, r_2, r_3)}{\Pr(C_1 = 1 \mid r_1, r_2, r_3)}\right)$$

LLR Derivation for the repetition code under AWGN

We want to compute the LLR for a repetition code of length 3 given received values r_1, r_2, r_3 and assuming AWGN with noise variance σ^2.

$$\text{LLR} = \log\left(\frac{f(r_1, r_2, r_3 \mid c = 0)}{f(r_1, r_2, r_3 \mid c = 1)}\right)$$

Assume BPSK: bit $0 \to +1$, bit $1 \to -1$. So for $c = 0$, all transmitted symbols are $+1$, and for $c = 1$, all are -1.

$$= \log \left(\frac{\prod_{i=1}^{3} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-(r_i-1)^2}{2\sigma^2}\right)}{\prod_{i=1}^{3} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-(r_i+1)^2}{2\sigma^2}\right)} \right)$$

$$= \log \left(\exp\left(\frac{2}{\sigma^2}(r_1 + r_2 + r_3) \right) \right)$$

$$= \frac{2}{\sigma^2}(r_1 + r_2 + r_3)$$

General Form for Repetition Code of Length n:

$$L = \frac{2}{\sigma^2} \left(\sum_{i=1}^{n} r_i \right)$$

SPA Code Example

- Encoded: c_1, c_2, c_3
- Received: r_1, r_2, r_3
- Decoded:

$$L_1 = l_1 + L_{\text{ext1}}$$
$$L_2 = l_2 + L_{\text{ext2}}$$
$$L_3 = l_3 + L_{\text{ext3}}$$

Min-Sum Approximation in Soft Decoding

$$L_3 = l + L_{\text{ext}_3}$$
$$\text{sgn}(L_{\text{ext}_1}) = \text{sgn}(l_2) \cdot \text{sgn}(l_3)$$

Define function:

$$f(x) = \left| \log \tanh \left(\left| \frac{x}{2} \right| \right) \right|$$

Then:

$$|L_{\text{ext}_1}| = f\left(f(|l_2|) + f(|l_3|)\right)$$

Now, for large values of l_2 and l_3, the function $f(x)$ is small. So, we approximate for small inputs:

$$f(l_2) + f(l_3) \approx f\left(\min(|l_2|, |l_3|)\right)$$

Therefore:

$$|L_{\text{ext}_1}| = f\left(f\left(\min(|l_2|, |l_3|)\right)\right)$$

Since $f^{-1}(x) = f(x)$, we get:

$$|L_{\text{ext}_1}| = \min(|l_2|, |l_3|)$$

Conclusion: Hence, the **min-sum algorithm** is a simplified approximation for soft decoding:

- SPA for CN $\rightarrow$ VN uses log-tanh
- Min-sum for CN $\rightarrow$ VN uses only minimum operation

4 Code

4.1 For - Hard Decision Decoding

Using NR_2_6_52

```
%encoding functions
function y = mul_sh(x, k)
    if k == -1
        y = zeros(1, length(x));
    else
        y = [x(k+1:end) x(1:k)];
    end
end

function [B,H,z] = nrldpc_Hmatrix(BG)
    load(sprintf('%s.txt',BG),BG);
    B = NR_2_6_52;
    [mb,nb] = size(B);
    z = 52;
    H = zeros(mb*z,nb*z);
    Iz = eye(z); I0 = zeros(z);
    for kk = 1:mb
        tmpvecR = (kk-1)*z+(1:z);
        for kk1 = 1:nb
            tmpvecC = (kk1-1)*z+(1:z);
            if B(kk,kk1) == -1
                H(tmpvecR,tmpvecC) = I0;
            else
                H(tmpvecR,tmpvecC) = circshift(Iz,-B(kk,kk1));
            end
        end
    end

    [U,N]=size(H); K = N-U;
    P = H(:,1:K);
    G = [eye(K); P];
    Z = H*G;
end

function cword = nrldpc_encode(B,z,msg)
    %B: base matrix
    %z: expansion factor
    %msg: message vector, length = (#cols(B)-#rows(B))*z
    %cword: codeword vector, length = #cols(B)*z

    [m,n] = size(B);

    cword = zeros(1,n*z);
    cword(1:(n-m)*z) = msg;

    %double-diagonal encoding
    temp = zeros(1,z);
    for i = 1:4 %row 1 to 4
```

```
        for j = 1:n-m %message columns
            temp = mod(temp + mul_sh(msg((j-1)*z+1:j*z),B(i,j)),2);
        end
    end
    if B(2,n-m+1) == -1
        p1_sh = B(3,n-m+1);
    else
        p1_sh = B(2,n-m+1);
    end
    cword((n-m)*z+1:(n-m+1)*z) = mul_sh(temp,z-p1_sh); %p1
    %Find p2, p3, p4
    for i = 1:3
        temp = zeros(1,z);
        for j = 1:n-m+i
            temp = mod(temp + mul_sh(cword((j-1)*z+1:j*z),B(i,j)),2);
        end
        cword((n-m+i)*z+1:(n-m+i+1)*z) = temp;
    end
    %Remaining parities
    for i = 5:m
        temp = zeros(1,z);
        for j = 1:n-m+4
            temp = mod(temp + mul_sh(cword((j-1)*z+1:j*z),B(i,j)),2);
        end
        cword((n-m+i-1)*z+1:(n-m+i)*z) = temp;
    end
end
```

```
baseGraph5GNR = 'NR_2_6_52';

% Number of Simulation (A new message generated, encoded and decoded) for each
coderate
Nsim = 500;

%Max iterations of the tanner graph to decode the received codeword
max_itr = 20;

colors = ['r', 'g', 'b', 'm'];

%EbN0dB range
EbNodB = 0:0.5:10;

%Various Coderates
R = [1/4 1/3 1/2 3/5];

% Prob of bit errors vs EbN0dB
Bit_Error_Rate = zeros(length(R), length(EbNodB));

% Prob of Error Probability vs Eb/No(dB)
```

```
p_error = zeros(length(R), length(EbNodB));

% Success Probabilty vs No of Iteration
p_success_itr = zeros(length(EbNodB), max_itr);

% coderate iterator
coderate_iterator = 1;

for codeRate = R

    [B, Hfull, z] = nrldpc_Hmatrix(baseGraph5GNR);
    [mb, nb] = size(B);
    kb = nb - mb;
    kNumInfoBits = kb * z;
    k_pc = kb - 2;

    %Finding the number of bits to be considered for the given coderate
    %rest all parity bits to be punctured (Not to be Considered)
    nbRM = ceil(k_pc / codeRate) + 2;
    nBlockLength = nbRM * z;

    %Shortening the H matrix accordingly
    H = Hfull(:, 1:nBlockLength);
    nChecksNotPunctured = mb*z - nb*z + nBlockLength;
    H = H(1:nChecksNotPunctured, :);

    Nchecks = size(H, 1);

    %No of rows and columns in the shortened H-Matrix
    Rows = size(H, 1);
    Cols = size(H, 2);

    %EbNodB iterator
    ebno_iterator=1;

    for EbNoindex = 1:length(EbNodB)

        EbNo = 10^(EbNodB(EbNoindex) / 10);
        sigma = sqrt(1 / (2 * codeRate * EbNo));
        ErrorinBits = 0;

        for ksim = 1:Nsim

            %Random message generation
            b = randi([0 1], [kNumInfoBits 1]);

            %encoding the message using Base Matrix
            c = nrldpc_encode(B, z, b');
            c = c(1:nBlockLength)';
```

```
            % BPSK Modulation (0-> 1V, 1-> -1V)
            s = 1 - 2*c;

            %Passing the encoded codeword through AWGN noise channel with
            %specified EbN0dB
            received_vector = s + sigma * randn(size(s));
            received_vector = (received_vector < 0);

            %Message Passing Matrices
            M = zeros(Rows, Cols);  % VN->CN
            L = zeros(Rows, Cols);  % CN->VN
            decoded_msg = zeros(1, Cols);

            % Firstly sending received bits form VN->CN
            for Col = 1 : Cols
                Connected_CNs = find(H(:, Col));
                for j = 1 : length(Connected_CNs)
                    M(Connected_CNs(j), Col) = received_vector(Col);  % Initial
message is received bit
                end
            end

            % Decoding iterations
            for itr = 1 : max_itr

                % SPC
                for ir = 1 : Rows
                    Connected_VNs = find(H(ir, :));
                    for j = 1 : length(Connected_VNs)
                        others = M(ir, Connected_VNs([1:j-1, j+1:end]));
                        L(ir, Connected_VNs(j)) = mod(sum(others), 2);  % Parity
excluding self
                    end
                end

                % Majority Voting
                for Col = 1 : Cols
                    Connected_CNs = find(H(:, Col));
                    total_vote = sum(L(Connected_CNs, Col)) + received_vector(Col);
                    decoded_msg(Col) = total_vote > ((length(Connected_CNs) + 1) /
2);

                    for j = 1 : length(Connected_CNs)
                        other_votes = L(Connected_CNs([1:j-1, j+1:end]), Col);
                        count = sum(other_votes) + received_vector(Col);
                        M(Connected_CNs(j), Col) = count > (length(Connected_CNs) /
2);
                    end
                end
```

```
                if (isequal(decoded_msg(1:kNumInfoBits), c(1:kNumInfoBits)') &&
coderate_iterator==2)
                    p_success_itr(ebno_iterator,itr) =
p_success_itr(ebno_iterator,itr) + 1;
                end

             end

            % Count bit errors
            bitError = sum(decoded_msg(1:kNumInfoBits) ~= c(1:kNumInfoBits)');
            if(bitError > 0)
                Bit_Error_Rate(coderate_iterator, ebno_iterator) =
Bit_Error_Rate(coderate_iterator, ebno_iterator) + bitError;
                p_error(coderate_iterator, ebno_iterator) =
p_error(coderate_iterator, ebno_iterator) + 1;
            end
        end
        Bit_Error_Rate(coderate_iterator, ebno_iterator) =
Bit_Error_Rate(coderate_iterator, ebno_iterator) / (Cols-Rows) / Nsim;
        p_error(coderate_iterator, ebno_iterator) = p_error(coderate_iterator,
ebno_iterator) / Nsim;

        ebno_iterator=ebno_iterator+1;
    end
    coderate_iterator=coderate_iterator+1;
end

p_success_itr = p_success_itr/Nsim;

EbNo = 10 .^ (EbNodB ./ 10);

% Calculate Bit Error Rate (Bit_Error_Rate) for each SNR
ber_uncoded = 0.5 * erfc(sqrt(EbNo ./ 2));

figure;
for idx = 1:length(R)
    semilogy(EbNodB, Bit_Error_Rate(idx, :),'Color', colors(idx), 'LineWidth', 2);
    hold on;
end
semilogy(EbNodB, ber_uncoded,'Color','k', 'LineWidth', 2);
xlabel('EbNo in dB');
ylabel('BER');
title('BER Performance (Log Scale)');
legend('Rate = 1/4', 'Rate = 1/3', 'Rate = 1/2', 'Rate = 3/5','Uncoded BPSK');
hold off;

figure;
hold on;
grid on;
for idx = 1:length(R)
```

```
    plot(EbNodB, Bit_Error_Rate(idx, :), 'Color', colors(idx), 'LineWidth', 2);
end
plot(EbNodB, ber_uncoded,'Color','k', 'LineWidth', 2);
xlabel('E_b/N_0 (dB)');
ylabel('BER');
title('BER Performance (Linear Scale)');
legend('Rate = 1/4', 'Rate = 1/3', 'Rate = 1/2', 'Rate = 3/5', 'Uncoded BPSK');
hold off;

figure;
hold on;
legendEntries = cell(1, length(EbNodB));
for i=1:length(EbNodB)
    plot(1:max_itr,p_success_itr(i,:),'LineWidth',2);
    legendEntries{i} = sprintf('Eb/N0 = %.1f dB', EbNodB(i));
end
xlabel('Iteration');
ylabel('Success Probability');
title('Success Probability vs Iteration for Different Eb/N0');
legend(legendEntries, 'Location', 'northeast');
grid on;
hold off;

figure;
hold on;
grid on;
for idx = 1:length(R)
    plot(EbNodB, 1-p_error(idx, :), 'Color', colors(idx), 'LineWidth', 2);
end
xlabel('E_b/N_0 (dB)');
ylabel('Probability of Success');
title('Probability of Success vs E_b/N_0 (dB)');
legend('Rate = 1/4', 'Rate = 1/3', 'Rate = 1/2', 'Rate = 3/5');
hold off;

figure;
hold on;
grid on;
for idx = 1:length(R)
    plot(EbNodB, p_error(idx, :), 'Color', colors(idx), 'LineWidth', 2);
end
xlabel('E_b/N_0 (dB)');
ylabel('Probability Of Decoding Failure');
title('Probability Of Decoding Failure vs E_b/N_0 (dB)');
legend('Rate = 1/4', 'Rate = 1/3', 'Rate = 1/2', 'Rate = 3/5');
hold off;
```

Using NR_1_5_352

```
%encoding functions
function y = mul_sh(x, k)
    if k == -1
        y = zeros(1, length(x));
    else
        y = [x(k+1:end) x(1:k)];
    end
end

function [B,H,z] = nrldpc_Hmatrix(BG)
    load(sprintf('%s.txt',BG),BG);
    B = NR_1_5_352;
    [mb,nb] = size(B);
    z = 352;
    H = zeros(mb*z,nb*z);
    Iz = eye(z); I0 = zeros(z);
    for kk = 1:mb
        tmpvecR = (kk-1)*z+(1:z);
        for kk1 = 1:nb
            tmpvecC = (kk1-1)*z+(1:z);
            if B(kk,kk1) == -1
                H(tmpvecR,tmpvecC) = I0;
            else
                H(tmpvecR,tmpvecC) = circshift(Iz,-B(kk,kk1));
            end
        end
    end

    [U,N]=size(H); K = N-U;
    P = H(:,1:K);
    G = [eye(K); P];
    Z = H*G;
end

function cword = nrldpc_encode(B,z,msg)
    %B: base matrix
    %z: expansion factor
    %msg: message vector, length = (#cols(B)-#rows(B))*z
    %cword: codeword vector, length = #cols(B)*z

    [m,n] = size(B);

    cword = zeros(1,n*z);
    cword(1:(n-m)*z) = msg;

    %double-diagonal encoding
    temp = zeros(1,z);
    for i = 1:4 %row 1 to 4
```

```
        for j = 1:n-m %message columns
            temp = mod(temp + mul_sh(msg((j-1)*z+1:j*z),B(i,j)),2);
        end
    end
    if B(2,n-m+1) == -1
        p1_sh = B(3,n-m+1);
    else
        p1_sh = B(2,n-m+1);
    end
    cword((n-m)*z+1:(n-m+1)*z) = mul_sh(temp,z-p1_sh); %p1
    %Find p2, p3, p4
    for i = 1:3
        temp = zeros(1,z);
        for j = 1:n-m+i
            temp = mod(temp + mul_sh(cword((j-1)*z+1:j*z),B(i,j)),2);
        end
        cword((n-m+i)*z+1:(n-m+i+1)*z) = temp;
    end
    %Remaining parities
    for i = 5:m
        temp = zeros(1,z);
        for j = 1:n-m+4
            temp = mod(temp + mul_sh(cword((j-1)*z+1:j*z),B(i,j)),2);
        end
        cword((n-m+i-1)*z+1:(n-m+i)*z) = temp;
    end
end
```

```
baseGraph5GNR = 'NR_1_5_352';

% Number of Simulation (A new message generated, encoded and decoded) for each
coderate
Nsim = 500;

%Max iterations of the tanner graph to decode the received codeword
max_itr = 20;

colors = ['r', 'g', 'b', 'm'];

%EbN0dB range
EbNodB = 0:0.5:10;

%Various Coderates
R = [1/3 1/2 3/5 4/5];
% Prob of bit errors vs EbN0dB
Bit_Error_Rate = zeros(length(R), length(EbNodB));

% Prob of Error Probability vs Eb/No(dB)
p_error = zeros(length(R), length(EbNodB));
```

```
% Success Probabilty vs No of Iteration
p_success_itr = zeros(length(EbNodB), max_itr);

% coderate iterator
coderate_iterator = 1;

for codeRate = R
    [B, Hfull, z] = nrldpc_Hmatrix(baseGraph5GNR);
    [mb, nb] = size(B);
    kb = nb - mb;
    kNumInfoBits = kb * z;
    k_pc = kb - 2;

    %Finding the number of bits to be considered for the given coderate
    %rest all parity bits to be punctured (Not to be Considered)
    nbRM = ceil(k_pc / codeRate) + 2;
    nBlockLength = nbRM * z;

    %Shortening the H matrix accordingly
    H = Hfull(:, 1:nBlockLength);
    nChecksNotPunctured = mb*z - nb*z + nBlockLength;
    H = H(1:nChecksNotPunctured, :);

    Nchecks = size(H, 1);

    %No of rows and columns in the shortened H-Matrix
    Rows = size(H, 1);
    Cols = size(H, 2);

    %EbN0dB iterator
    ebno_iterator=1;

    for EbNoindex = 1:length(EbNodB)

        EbNo = 10^(EbNodB(EbNoindex) / 10);
        sigma = sqrt(1 / (2 * codeRate * EbNo));
        ErrorinBits = 0;

        for ksim = 1:Nsim

            %Random message generation
            b = randi([0 1], [kNumInfoBits 1]);

            %encoding the message using Base Matrix
            c = nrldpc_encode(B, z, b');
            c = c(1:nBlockLength)';

            % BPSK Modulation (0-> 1V, 1-> -1V)
            s = 1 - 2*c;
```

```
            %Passing the encoded codeword through AWGN noise channel with
            %specified EbN0dB
            recieved_vector = s + sigma * randn(size(s));
            recieved_vector = (recieved_vector < 0);

            %Message Passing Matrices
            M = zeros(Rows, Cols);  % VN->CN
            L = zeros(Rows, Cols);  % CN->VN
            decoded_msg = zeros(1, Cols);

            % Firstly sending received bits form VN->CN
            for ic = 1 : Cols
                connected_checks = find(H(:, ic));
                for j = 1 : length(connected_checks)
                    M(connected_checks(j), ic) =recieved_vector(ic);
                end
            end


            %Decoding Iterations
            for itr = 1 : max_itr

                % SPC
                for ir = 1 : Rows
                    connected_vars = find(H(ir, :));
                    for j = 1 : length(connected_vars)
                        others = M(ir, connected_vars([1:j-1, j+1:end]));
                        L(ir, connected_vars(j)) = mod(sum(others), 2);  % Parity
excluding self
                    end
                end

                % Majority Logic
                for ic = 1 : Cols
                    connected_checks = find(H(:, ic));
                    total_vote = sum(L(connected_checks, ic)) + recieved_vector(ic);
                    decoded_msg(ic) = total_vote > ((length(connected_checks) +
1) / 2);

                    for j = 1 : length(connected_checks)
                        other_votes = L(connected_checks([1:j-1, j+1:end]), ic);
                        cnt = sum(other_votes) + recieved_vector(ic);
                        M(connected_checks(j), ic) = cnt >
(length(connected_checks) / 2);
                    end
                end

                if (isequal(decoded_msg(1:kNumInfoBits), c(1:kNumInfoBits)') &&
coderate_iterator==2)
```

```
                    p_success_itr(ebno_iterator,itr) =
p_success_itr(ebno_iterator,itr) + 1;
                end

             end

            % Count bit errors
            bitError = sum(decoded_msg(1:kNumInfoBits) ~= c(1:kNumInfoBits)');
            if(bitError > 0)
                Bit_Error_Rate(coderate_iterator, ebno_iterator) =
Bit_Error_Rate(coderate_iterator, ebno_iterator) + bitError;
                p_error(coderate_iterator, ebno_iterator) =
p_error(coderate_iterator, ebno_iterator) + 1;
            end
        end
        Bit_Error_Rate(coderate_iterator, ebno_iterator) =
Bit_Error_Rate(coderate_iterator, ebno_iterator) / (Cols-Rows) / Nsim;
        p_error(coderate_iterator, ebno_iterator) = p_error(coderate_iterator,
ebno_iterator) / Nsim;

        ebno_iterator=ebno_iterator+1;
    end
    coderate_iterator=coderate_iterator+1;
end

p_success_itr = p_success_itr/Nsim;

EbNo = 10 .^ (EbNodB ./ 10);

% Calculate Bit Error Rate (Bit_Error_Rate) for each SNR
ber_uncoded = 0.5 * erfc(sqrt(EbNo ./ 2));

figure;
for idx = 1:length(R)
    semilogy(EbNodB, Bit_Error_Rate(idx, :),'Color', colors(idx), 'LineWidth', 2);
    hold on;
end
semilogy(EbNodB, ber_uncoded,'Color','k', 'LineWidth', 2);
xlabel('EbNo in dB');
ylabel('BER');
title('BER Performance (Log Scale)');
legend('Rate = 1/4', 'Rate = 1/3', 'Rate = 1/2', 'Rate = 3/5','Uncoded BPSK');
hold off;

figure;
hold on;
grid on;
for idx = 1:length(R)
    plot(EbNodB, Bit_Error_Rate(idx, :), 'Color', colors(idx), 'LineWidth', 2);
end
```

```
plot(EbNodB, ber_uncoded,'Color','k', 'LineWidth', 2);
xlabel('E_b/N_0 (dB)');
ylabel('BER');
title('BER Performance (Linear Scale)');
legend('Rate = 1/4', 'Rate = 1/3', 'Rate = 1/2', 'Rate = 3/5', 'Uncoded BPSK');
hold off;

figure;
hold on;
legendEntries = cell(1, length(EbNodB));
for i=1:length(EbNodB)
    plot(1:max_itr,p_success_itr(i,:),'LineWidth',2);
    legendEntries{i} = sprintf('Eb/N0 = %.1f dB', EbNodB(i));
end
xlabel('Iteration');
ylabel('Success Probability');
title('Success Probability vs Iteration for Different Eb/N0');
legend(legendEntries, 'Location', 'northeast');
grid on;
hold off;

figure;
hold on;
grid on;
for idx = 1:length(R)
    plot(EbNodB, 1-p_error(idx, :), 'Color', colors(idx), 'LineWidth', 2);
end
xlabel('E_b/N_0 (dB)');
ylabel('Probability of Success');
title('Probability of Success vs E_b/N_0 (dB)');
legend('Rate = 1/4', 'Rate = 1/3', 'Rate = 1/2', 'Rate = 3/5');
hold off;

figure;
hold on;
grid on;
for idx = 1:length(R)
    plot(EbNodB, p_error(idx, :), 'Color', colors(idx), 'LineWidth', 2);
end
xlabel('E_b/N_0 (dB)');
ylabel('Probability Of Decoding Failure');
title('Probability Of Decoding Failure vs E_b/N_0 (dB)');
legend('Rate = 1/4', 'Rate = 1/3', 'Rate = 1/2', 'Rate = 3/5');
hold off;
```

4.2 For - Soft Decision Decoding

```
% Load 5G NR LDPC base matrix
baseGraph5GNR = 'NR_2_6_52';
[B, H, z] = nrldpc_Hmatrix(baseGraph5GNR);

% Code rates to simulate
R = [1/4, 1/3, 1/2, 3/5];
EbNo_dB_range = 0:0.01:1;
Nsim = 100;
max_iter = 10;


BER = zeros(length(R), length(EbNo_dB_range));
p_error = zeros(length(R), length(EbNo_dB_range));

for crIdx = 1:length(R)
    codeRate = R(crIdx);
    [mb, nb] = size(B);
    k = (nb - mb) * z; % Total info bits before shortening
    numShortened = 2 * z; % Shorten first 2*z info bits

    % Calculating required number of punctured parity bits to achieve Rate
totalBitsAfterShortening = nb*z - numShortened;
actualInfoBits = k - numShortened;
desiredCodewordLength = actualInfoBits / codeRate;

% Ensuring desiredCodewordLength
desiredCodewordLength = max(desiredCodewordLength, actualInfoBits);

% Calculating required puncturing
numPunctured = totalBitsAfterShortening - desiredCodewordLength;
numPunctured = max(0, round(numPunctured));

% Available parity bits (after shortening)
availableParity = mb*z - (numShortened - (nb - mb)*z); % Adjust based on base graph
availableParity = max(0, availableParity); % Ensure non-negative

% Limit puncturing
numPunctured = min(numPunctured, availableParity);

% Adjusting
numPunctured = floor(numPunctured / z) * z;

% Actual code rate after puncturing
actualCodeRate = actualInfoBits / (totalBitsAfterShortening - numPunctured);


    fprintf('Target R=%.2f, Actual R=%.4f, Punctured %d bits\n', codeRate,
actualCodeRate, numPunctured);

    for ebNoIdx = 1:length(EbNo_dB_range)
```

```
        EbNo_linear = 10^(EbNo_dB_range(ebNoIdx)/10);
        sigma = sqrt(1 / (2 * actualCodeRate * EbNo_linear)); % Use actual code rate

        errorCount = 0;
        bitError = 0;

        for sim = 1:Nsim
            % Generating random info bits
            infoBits = randi([0 1], actualInfoBits, 1);

            % Encoding using 5G NR encoder (custom function)
            paddedInfo = [zeros(numShortened, 1); infoBits]';
            codeword = nrldpc_encode(B, z, paddedInfo);

            if any(mod(H * codeword', 2))
               error('Invalid codeword generated');
            end
            % Removing shortened and punctured bits
            tx_codeword = codeword(numShortened + 1 : end - numPunctured);

            % BPSK modulation
            txSig = 1 - 2 * tx_codeword;

            % AWGN channel
            rxSig = txSig + sigma * randn(size(txSig));

            % LLR calculation
            llr_transmitted = 2 * rxSig / (sigma^2);

            % LLRs for shortened  and punctured (zeros)
            llr_shortened = 1e20 * ones(1, numShortened);
            llr_punctured = zeros(1, numPunctured);
            llr = [llr_shortened, llr_transmitted, llr_punctured];

            % LDPC Decoder (Min-Sum)
            decodedBits = min_sum_ldpc_decoder(llr, H, max_iter);

            % Extracting info bits
            receivedInfo = decodedBits(numShortened + 1 : numShortened +
actualInfoBits);

            % Counting errors
            bitError = bitError + sum(receivedInfo ~= infoBits');
            errorCount = errorCount + (any(receivedInfo ~= infoBits'));
        end

        BER(crIdx, ebNoIdx) = bitError / (Nsim * actualInfoBits);
        p_error(crIdx, ebNoIdx) = errorCount / Nsim;
        fprintf("CodeRate=%.2f | Eb/No=%.1f dB | BER=%.4e | BLER=%.4f\n", ...
```

```
            actualCodeRate, EbNo_dB_range(ebNoIdx), BER(crIdx, ebNoIdx),
p_error(crIdx, ebNoIdx));
    end
end
```

```
Target R=0.25, Actual R=0.2500, Punctured 936 bits
```

```
% Ploting BER and BLER
figure;
semilogy(EbNo_dB_range, BER', 'LineWidth', 2);
legend("R=1/4", "R=1/3", "R=1/2", "R=3/5");
title('BER vs Eb/No'); xlabel('Eb/No (dB)'); ylabel('BER'); grid on;

figure;
plot(EbNo_dB_range, p_error', 'LineWidth', 2);
legend("R=1/4", "R=1/3", "R=1/2", "R=3/5");
title('BLER vs Eb/No'); xlabel('Eb/No (dB)'); ylabel('BLER'); grid on;

% Min-Sum Decoder Function
function decodedBits = min_sum_ldpc_decoder(llr, H, max_iter)
    [m, n] = size(H);
    llr_in = llr(:)';
    msg_c2v = zeros(m, n); % Check-to-variable messages
    msg_v2c = zeros(m, n); % Variable-to-check messages


    % Initializing v2c messages with LLRs
    [row, col] = find(H);
    for idx = 1:length(row)
        i = row(idx);
        j = col(idx);
        msg_v2c(i,j) = llr_in(j);
    end



    for iter = 1:max_iter

        % Storing previous iteration messages
       prev_msg_v2c = msg_v2c;

        % Check Node Update
        for i = 1:m
            cols = find(H(i, :));
            signs = sign(msg_v2c(i, cols));
            abs_msg = abs(msg_v2c(i, cols));
            for j = 1:length(cols)
                min1 = Inf;
                min2 = Inf;
                for k = 1:length(cols)
```

```
                    if k ~= j
                        if abs_msg(k) < min1
                            min2 = min1;
                            min1 = abs_msg(k);
                        elseif abs_msg(k) < min2
                            min2 = abs_msg(k);
                        end
                    end
                end
                alpha = 0.8; % Normalization factor
                msg_c2v(i, cols(j)) = prod(signs([1:j-1, j+1:end])) * min1 * alpha;
            end
        end

        % Variable Node Update
        for j = 1:n
            rows = find(H(:, j));
            sum_llr = llr_in(j);
            for i = 1:length(rows)
                sum_llr = sum_llr + msg_c2v(rows(i), j);
            end
            for i = 1:length(rows)
                msg_v2c(rows(i), j) = sum_llr - msg_c2v(rows(i), j);
            end
        end




for i = 1:m
    for j = find(H(i,:))

        if sign(msg_v2c(i,j)) ~= sign(prev_msg_v2c(i,j)) && iter > 1
            msg_v2c(i,j) = 0;
        end
    end
end

        % Computing posterior LLRs
        posterior_llr = llr_in + sum(msg_c2v, 1);
        decodedBits = posterior_llr < 0;

        % Early stopping
        if all(mod(H * decodedBits', 2) == 0)
            break;
        end
    end
end

function [B,H,z] = nrldpc_Hmatrix(BG)
```

```
load(sprintf('%s.txt',BG),BG);
B = NR_2_6_52; % change for 2nd Base matrix
[mb,nb] = size(B);
z = 52; % change for 2nd Base matrix
H = zeros(mb*z,nb*z);
Iz = eye(z); I0 = zeros(z);
for kk = 1:mb
    tmpvecR = (kk-1)*z+(1:z);
    for kk1 = 1:nb
        tmpvecC = (kk1-1)*z+(1:z);
        if B(kk,kk1) == -1
            H(tmpvecR,tmpvecC) = I0;
        else
            H(tmpvecR,tmpvecC) = circshift(Iz,-B(kk,kk1));
        end
    end
end

[U,N]=size(H); K = N-U;
P = H(:,1:K);
G = [eye(K); P];
Z = H*G;

end

% Function to encode message using Base Matrix
function cword = nrldpc_encode(B,z,msg)

%B: base matrix
%z: expansion factor
%msg: message vector, length = (#cols(B)-#rows(B))*z
%cword: codeword vector, length = #cols(B)*z


[m,n] = size(B);

cword = zeros(1,n*z);
cword(1:(n-m)*z) = msg;


%double-diagonal encoding
temp = zeros(1,z);
for i = 1:4 %row 1 to 4
    for j = 1:n-m %message columns
        temp = mod(temp + mul_sh(msg(((j-1)*z+1):(j*z)),B(i,j)),2);
    end
end
if B(2,n-m+1) == -1
    p1_sh = B(3,n-m+1);
```

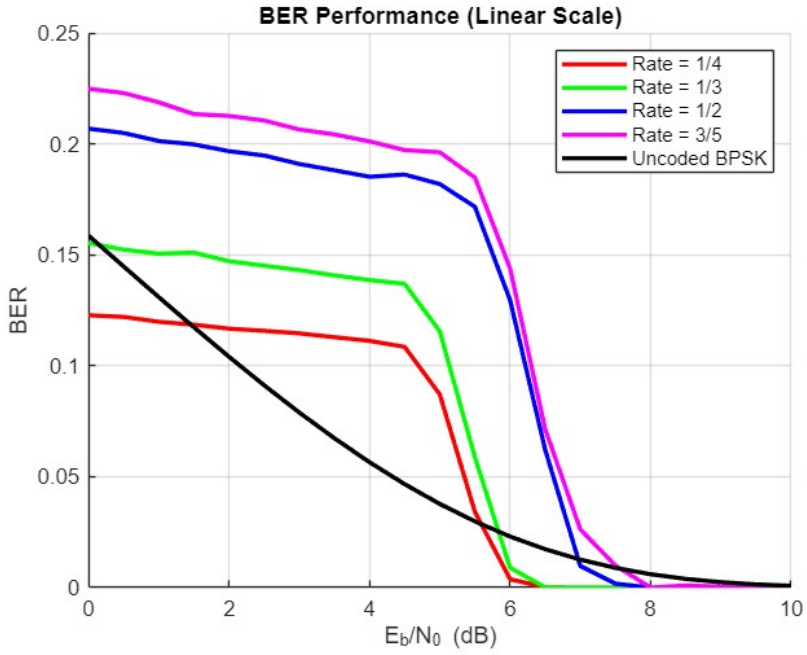
```
else
    p1_sh = B(2,n-m+1);
end


cword((n-m)*z+1:(n-m+1)*z) = mul_sh(temp,z-p1_sh); %p1
%Find p2, p3, p4
for i = 1:3
    temp = zeros(1,z);
    for j = 1:n-m+i
        temp = mod(temp + mul_sh(cword(((j-1)*z+1):(j*z)),B(i,j)),2);
    end
    cword((n-m+i)*z+1:(n-m+i+1)*z) = temp;
end


%Remaining parities
for i = 5:m
    temp = zeros(1,z);
    for j = 1:n-m+4
        temp = mod(temp + mul_sh(cword(((j-1)*z+1):(j*z)),B(i,j)),2);
    end
    cword((n-m+i-1)*z+1:(n-m+i)*z) = temp;
end



end

function y = mul_sh(x, k) % Function for shifting matrix x by k positions

 if(k == -1)
    y = zeros(1, length(x));
 else
    y = [x(k+1 : end) x(1 : k)];
 end
 end
```

5 Graphs

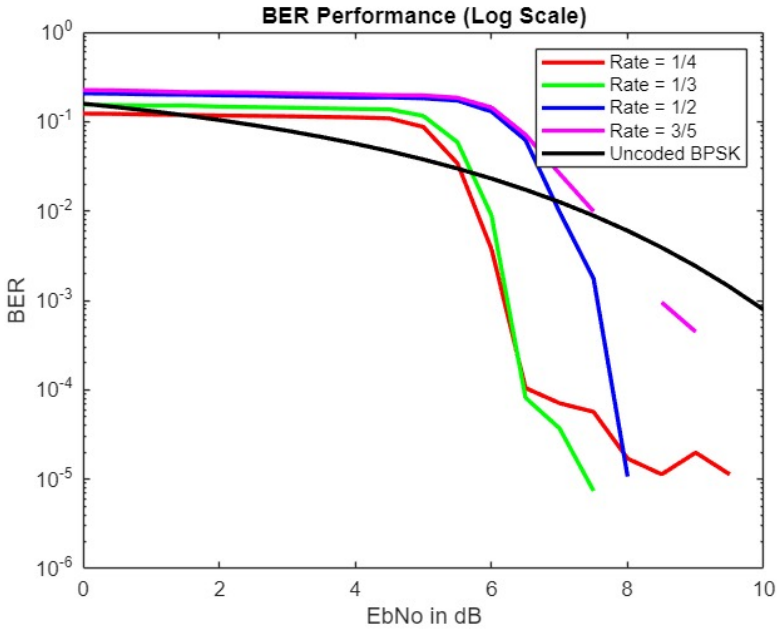
5.1 NR_2_6_52

5.1.1 For Hard Decision Decoding

- **Bit Error Rate vs E_b/N_0 (dB) in Linear Scale for NR_2_6_52**

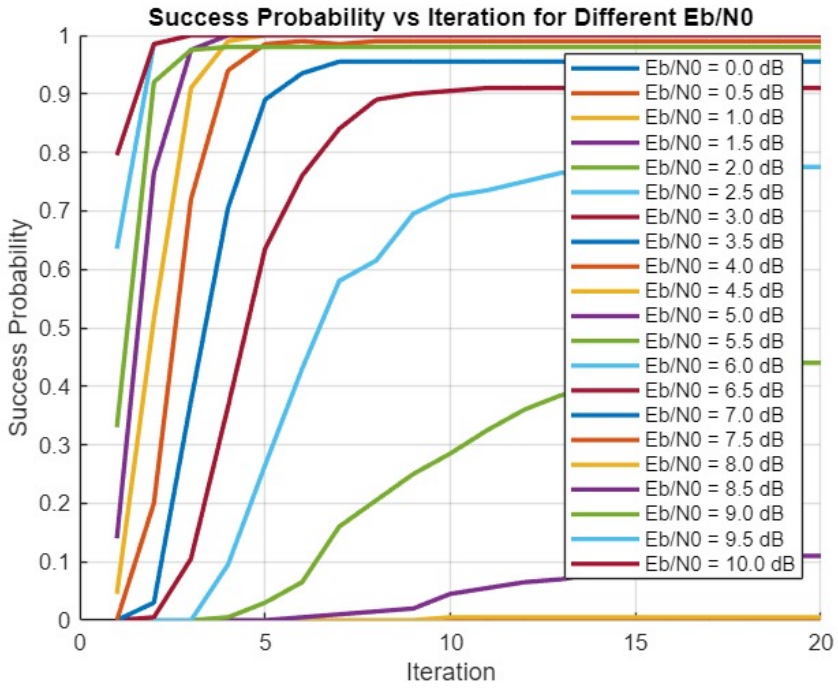


- **Bit Error Rate vs E_b/N_0 (dB) in Log Scale for NR_2_6_52**



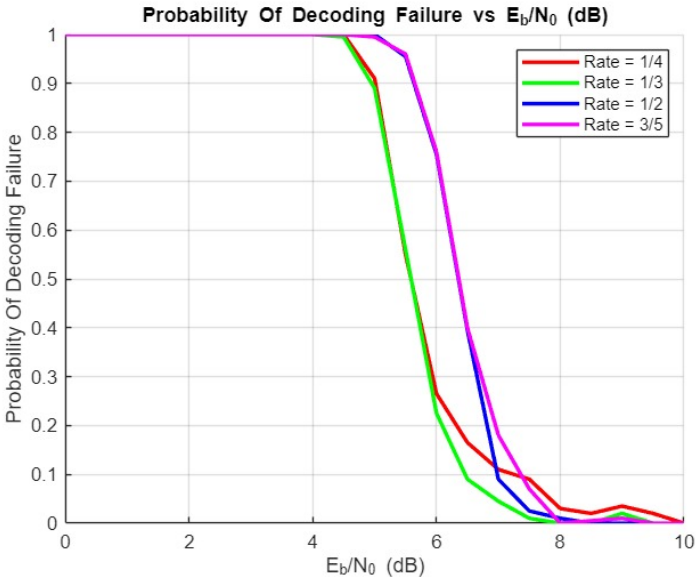
The number of bit errors decreases drastically once the value of Eb/No exceeds approximately 5 dB

- **Success Probability vs Number of Iterations for CodeRate=0.25 for NR_2_6_52**



From the graph, it is evident that for Eb/No values greater than or equal to 8 dB, the decoder is able to successfully decode all transmitted messages without any bit errors.

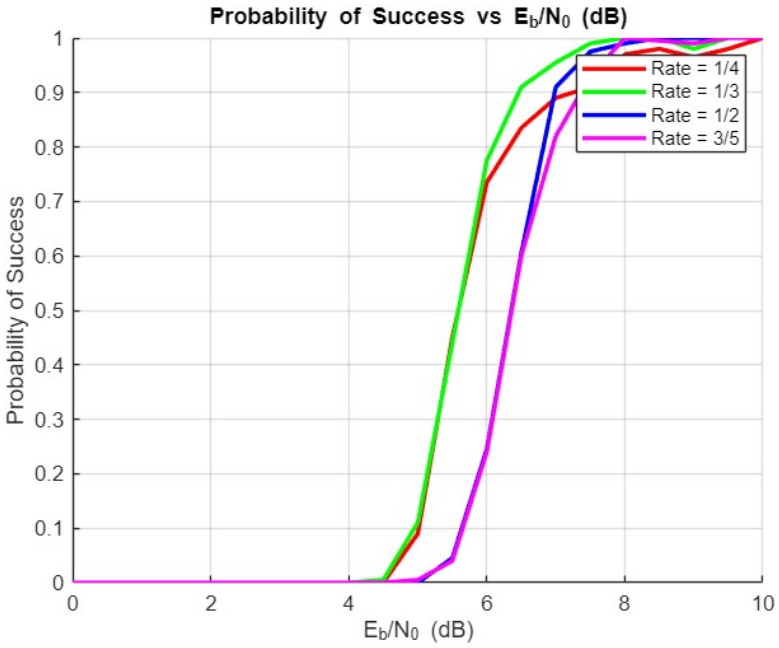
- **Probability of Decoding Failure vs E_b/N_0 (dB) for NR_2_6_52**



From the graph, we can clearly observe a comparison between various code rates. A

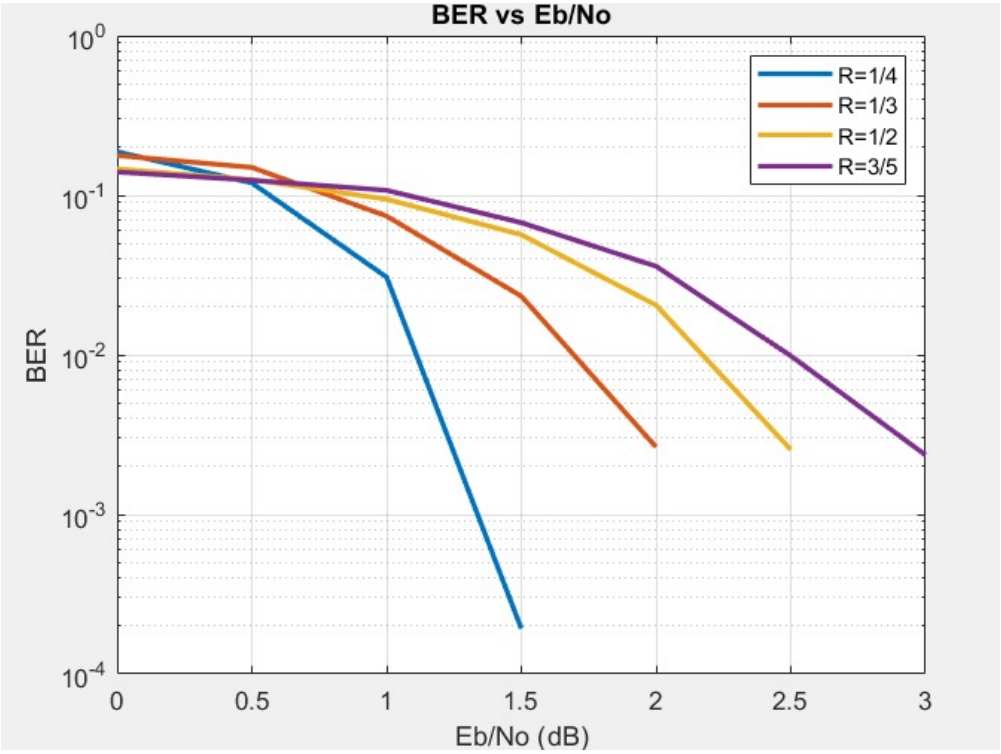
lower code rate indicates a more efficient channel. This is because, with lower code rates, the number of parity bits increases, making the decoding more reliable. A lower Eb/No value required for successful decoding indicates better code performance.

- **Probability of Success vs E_b/N_0 (dB) for NR_2_6_52**



5.1.2 For Soft Decision Decoding

- **BER vs** E_b/N_0



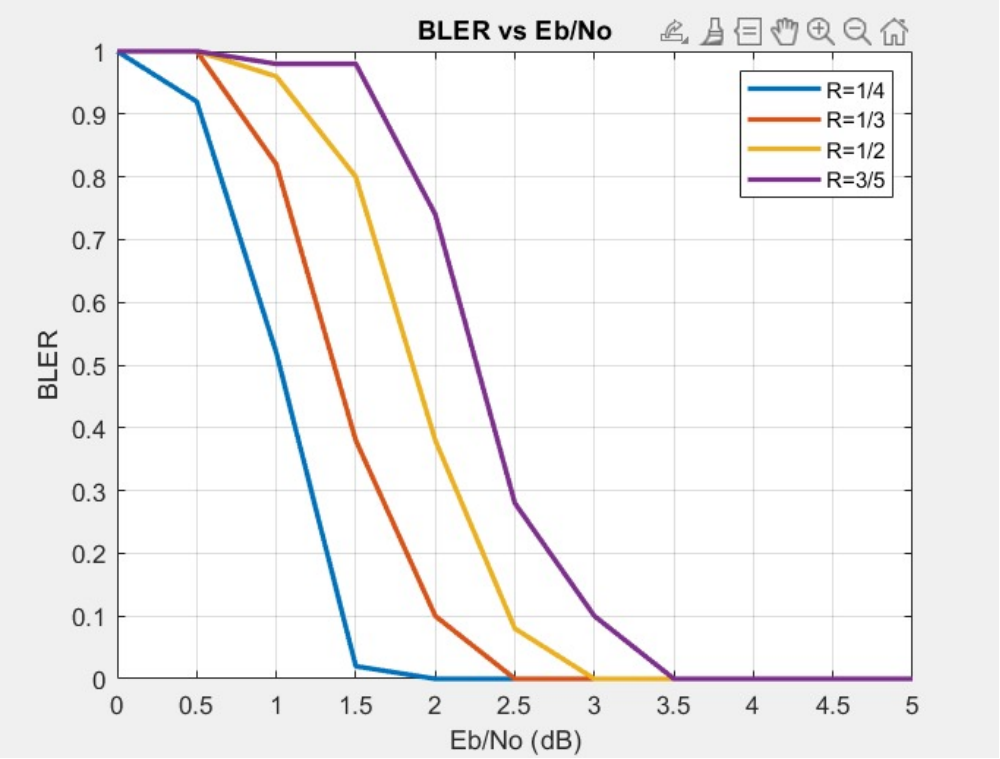
This plot presents the Bit Error Rate (BER) performance of LDPC codes using 5G NR Base Graph 2 (BG2) with a lifting size of Z = 52 over an AWGN channel with BPSK modulation. Results are shown for various code rates (R = 1/4, 1/3, 1/2, and 3/5). As Eb/No increases, BER decreases, with lower-rate codes (e.g., R=1/4) generally outperforming higher-rate ones at the same Eb/No.

As a group, we attempted to validate the performance across a wide range of Eb/No values. However, we were not able to obtain reliable or consistent results in the 0 to 1 dB range, where the curves do not clearly separate as expected.

We believe this behavior may be due to one or more of the following factors:

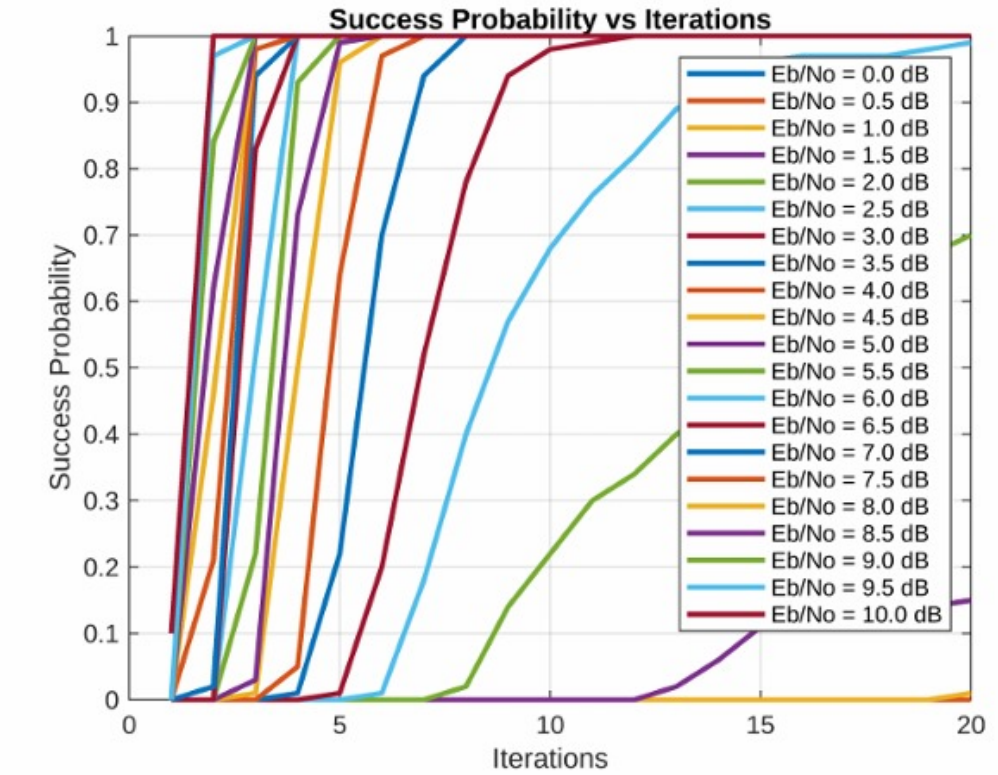
Insufficient number of simulations. Decoder convergence issues, such as too few iterations or early stopping conditions. LLR saturation or quantization effects limiting decoder precision.

- **BLER vs** E_b/N_0



The Block Error Rate (BLER) performance of LDPC codes using BPSK modulation over an AWGN channel is plotted here for a range of code rates (R = 1/4, 1/3, 1/2, and 3/5). The 5G NR Base Graph 2 (BG2) with a lifting size of Z = 52 serves as the basis for the simulations. As anticipated, higher-rate codes (e.g., R=3/5) need higher Eb/No to achieve the same BLER performance, whereas lower-rate codes (e.g., R=1/4) show better error correction capability at lower Eb/No values.

- **Success Probability vs Iterations**



This graph shows how the success probability of a decoding algorithm improves over iterations for different Eb/No (signal-to-noise) levels.

Higher Eb/No $\rightarrow$ faster and more reliable success (steeper curves).

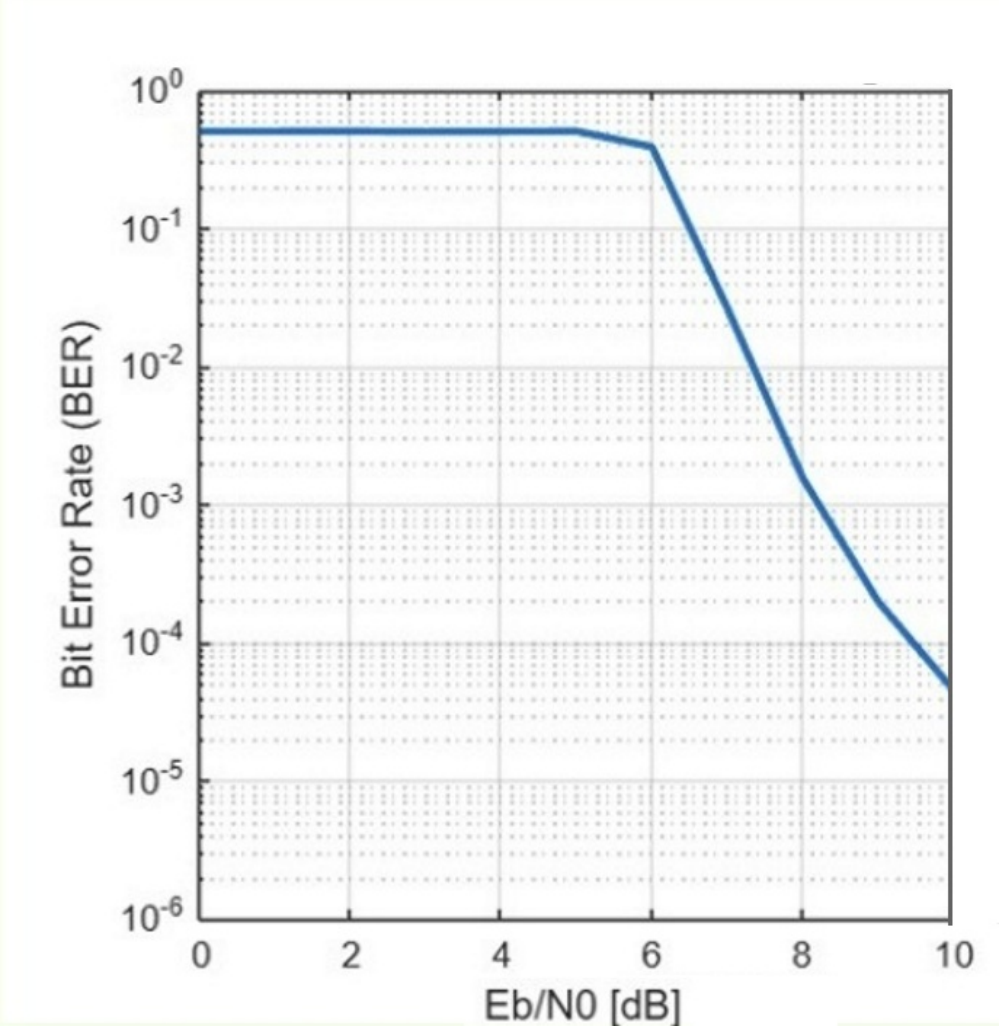
Lower Eb/No $\rightarrow$ slower convergence or poor success rate.

It highlights that better signal quality leads to quicker and more accurate decoding..

5.2 NR_1_5_352

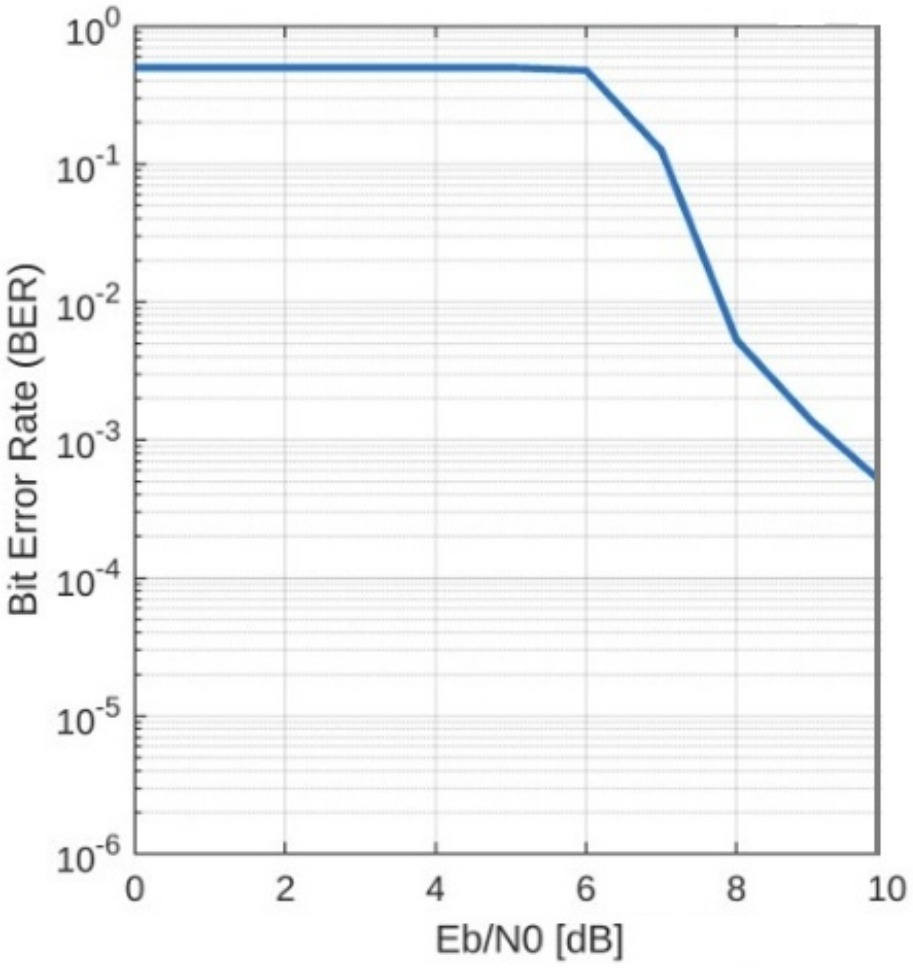
5.2.1 For Hard Decision Decoding

- **Bit Error Rate vs E_b/N_0 (dB) in Log Scale for CodeRate = 1/3**



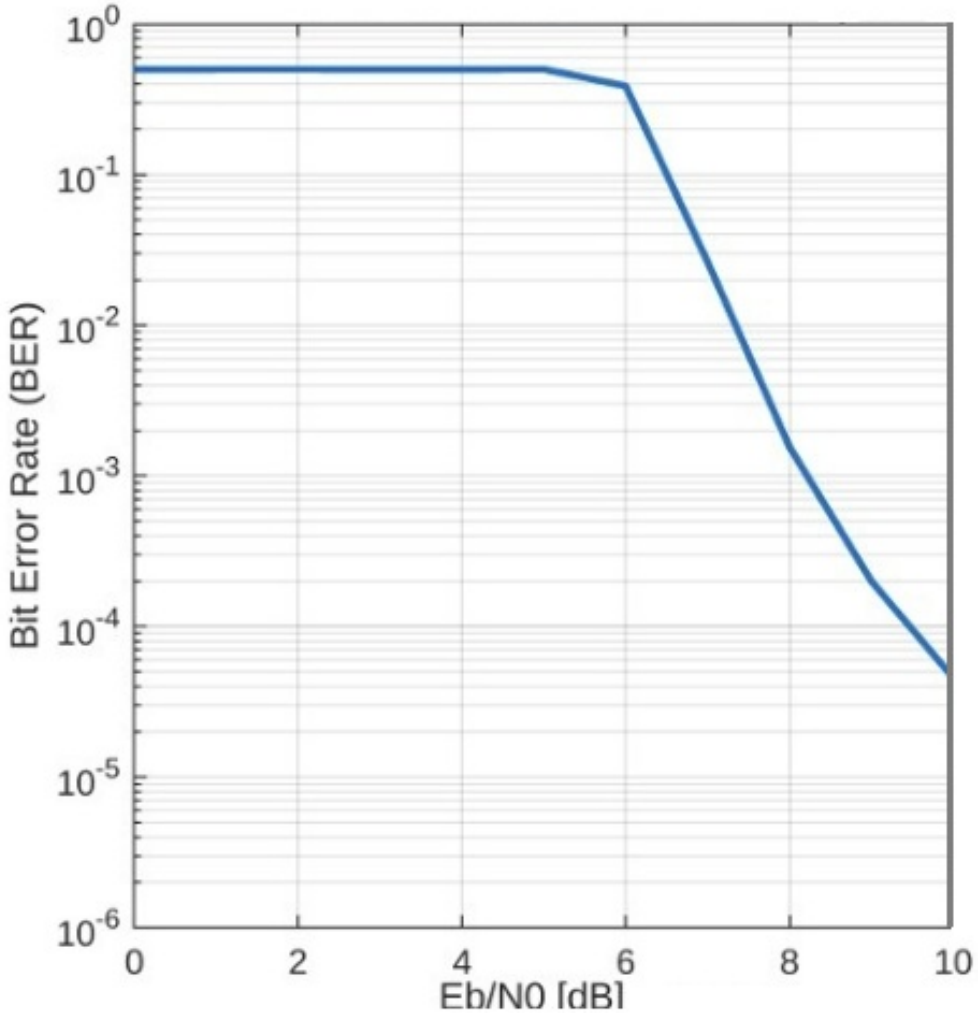
The number of bit errors decreases drastically once the value of Eb/No exceeds approximately 6 dB.

- **Bit Error Rate vs E_b/N_0 (dB) in Log Scale for CodeRate = $1/2$**



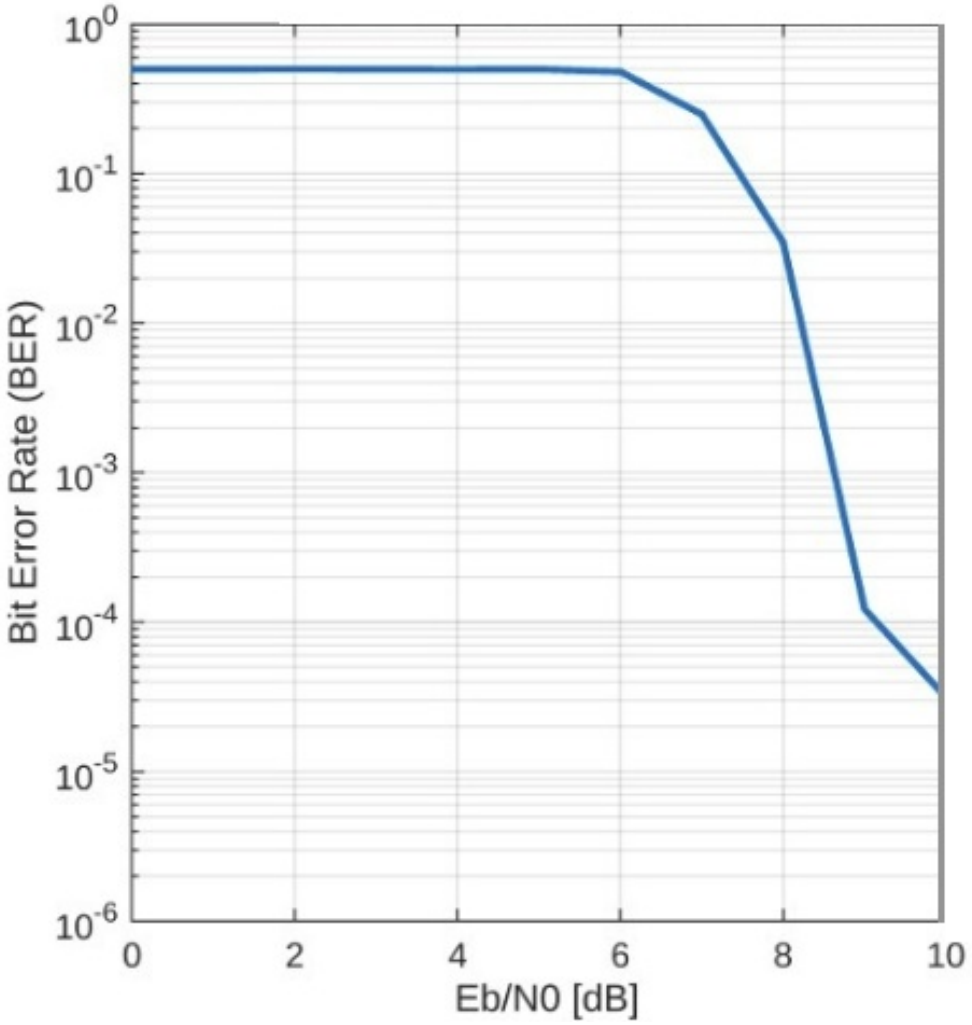
The number of bit errors decreases drastically once the value of Eb/No exceeds approximately 6.5 dB.

- **Bit Error Rate vs E_b/N_0 (dB) in Log Scale for CodeRate = $3/5$**



The number of bit errors decreases drastically once the value of Eb/No exceeds approximately 6 dB.

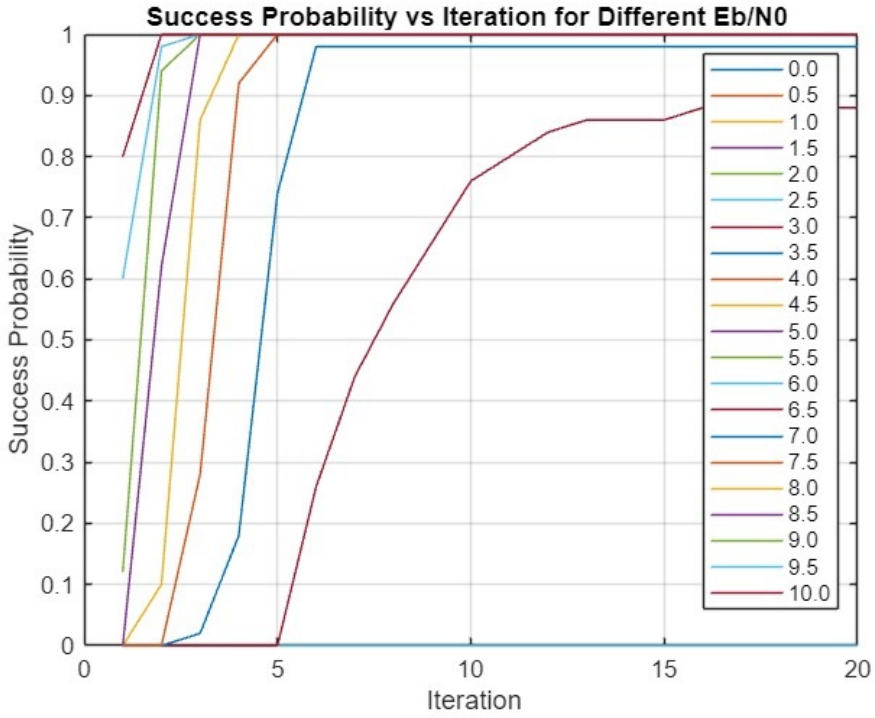
- **Bit Error Rate vs E_b/N_0 (dB) in Log Scale for CodeRate = $4/5$**



The number of bit errors decreases drastically once the value of Eb/No exceeds approximately 8 dB.

For different coderates, we observe that Bit Error Rate(BER) is directly proportional to Eb/No. As coderate increases, number of parity bits decreases making decoding less efficient thus more Eb/No(SNR) required to decode it.

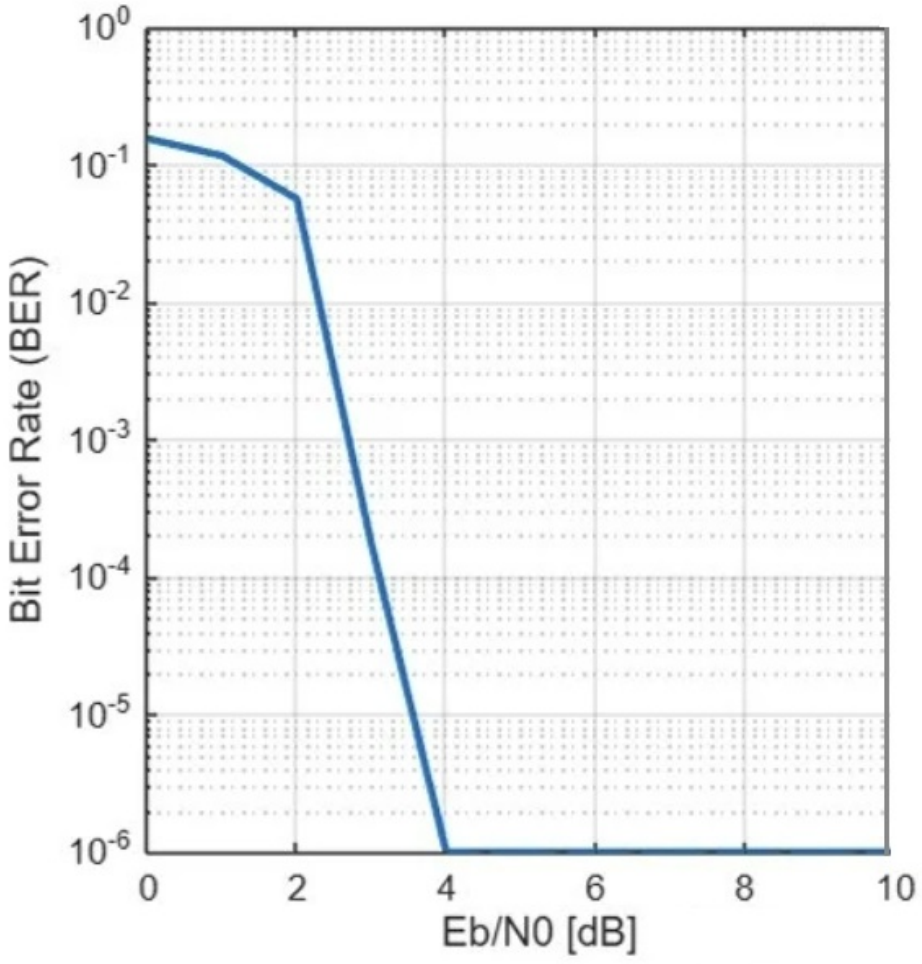
- **Success Probability vs Number of Iterations for CodeRate=0.5 for NR_1_5_352**



From the graph, it is evident that for Eb/No values greater than or equal to 7.5 dB, the decoder is able to successfully decode all transmitted messages without any bit errors.

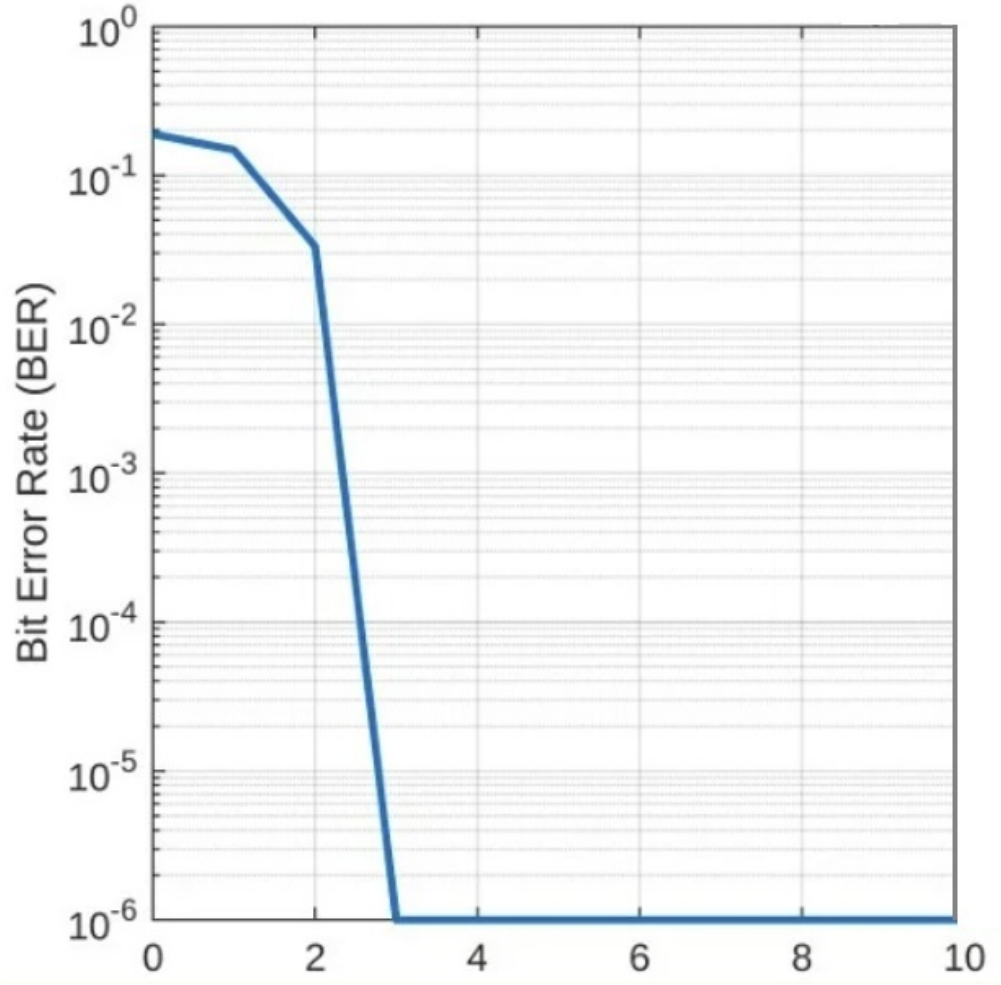
5.2.2 For Soft Decision Decoding

- **Bit Error Rate vs E_b/N_0 (dB) in Log Scale for CodeRate = $1/3$**



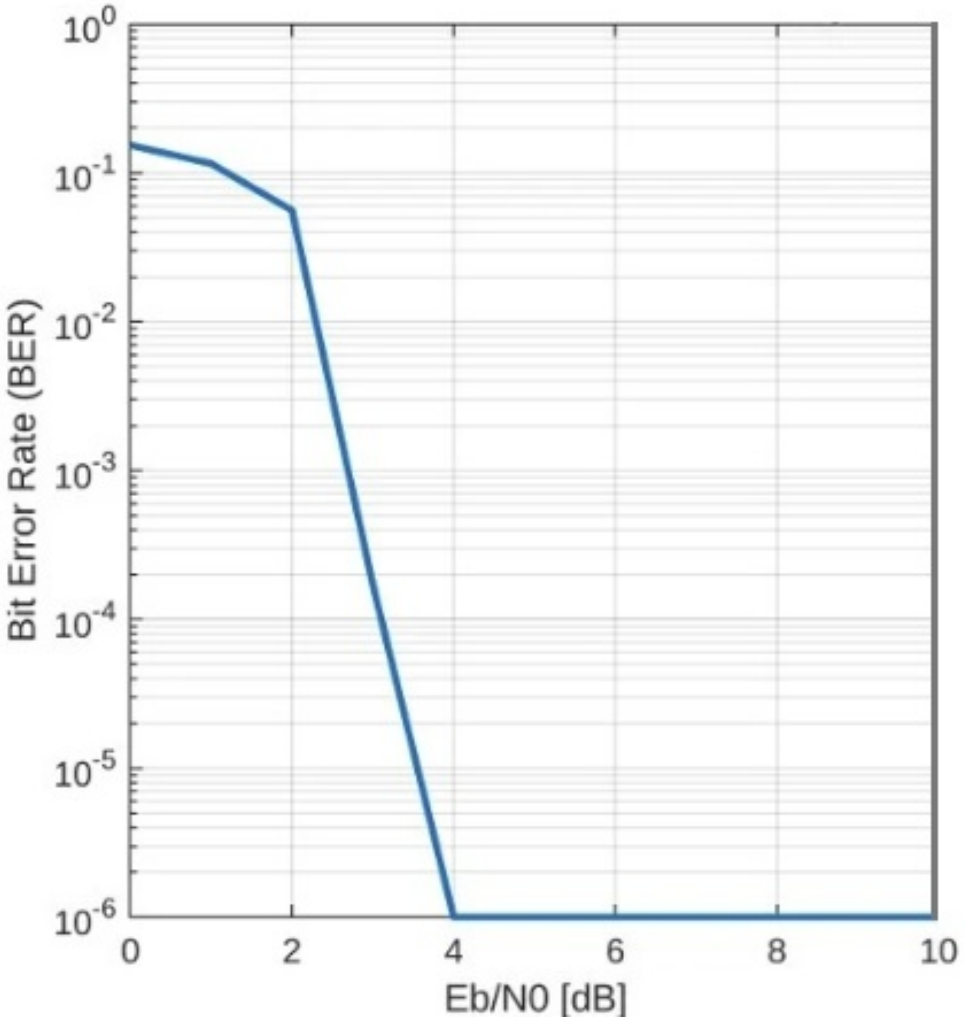
The number of bit errors decreases drastically once the value of Eb/No exceeds approximately 2 dB.

- **Bit Error Rate vs E_b/N_0 (dB) in Log Scale for CodeRate = $1/2$**



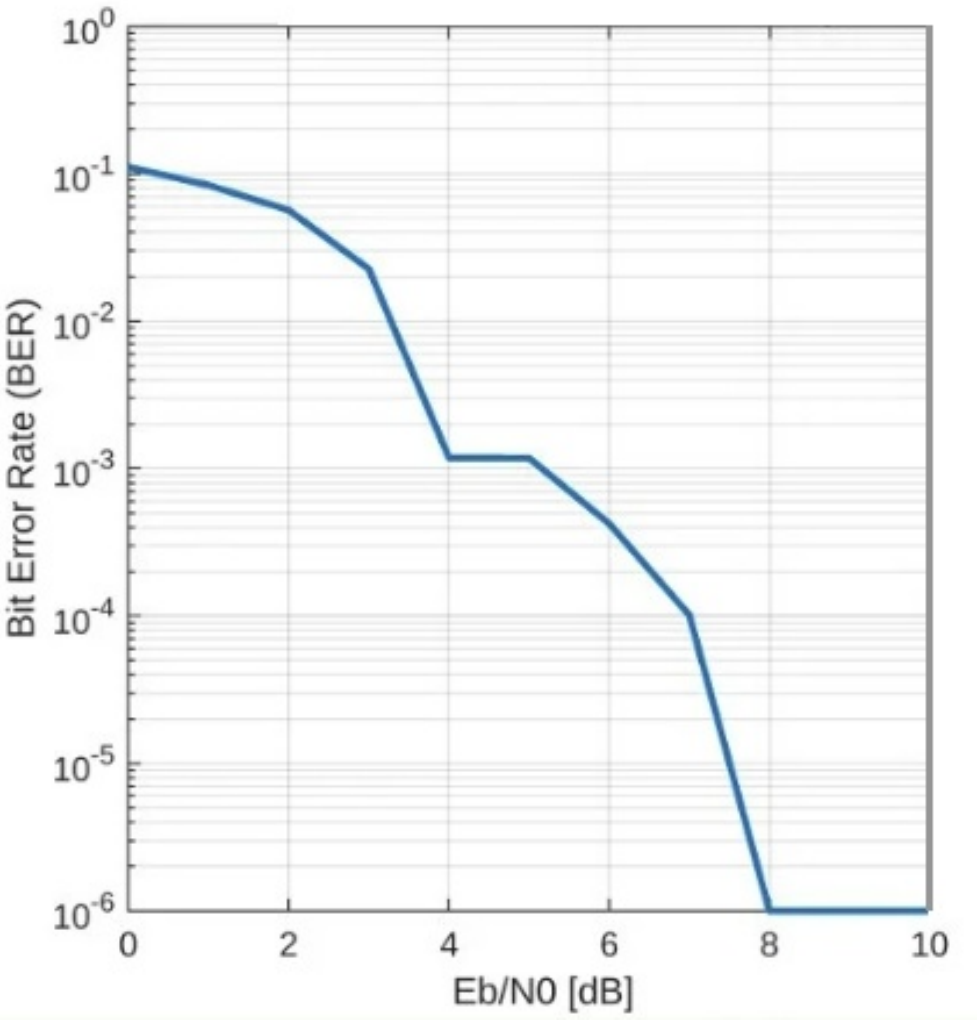
The number of bit errors decreases drastically once the value of Eb/No exceeds approximately 2 dB.

- **Bit Error Rate vs E_b/N_0 (dB) in Log Scale for CodeRate = $3/5$**



The number of bit errors decreases drastically once the value of Eb/No exceeds approximately 2 dB.

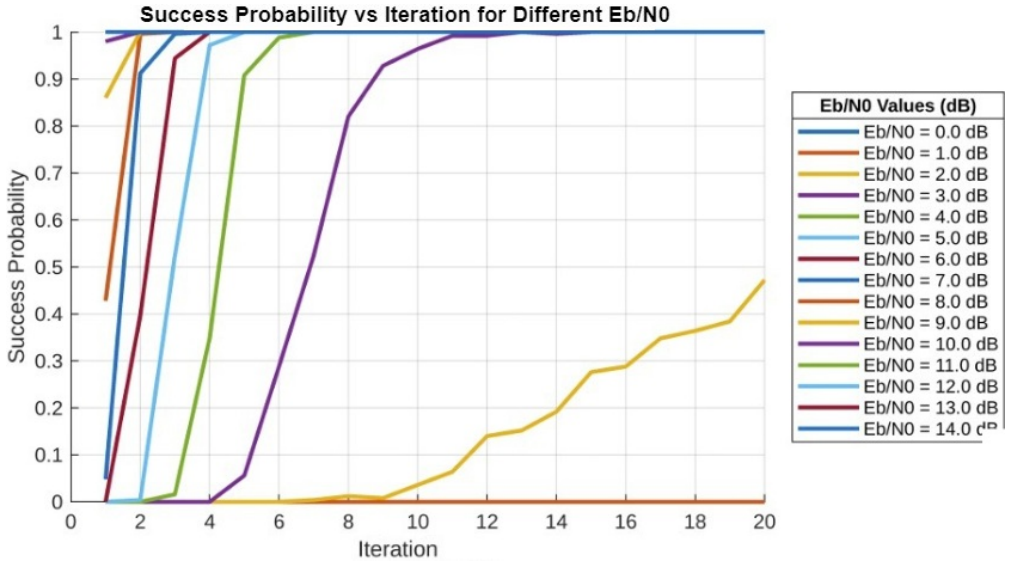
- **Bit Error Rate vs E_b/N_0 (dB) in Log Scale for CodeRate = $4/5$**



The number of bit errors decreases drastically once the value of Eb/No exceeds approximately 5 dB.

For different coderates, we observe that Bit Error Rate(BER) is directly proportional to Eb/No. As coderate increases, number of parity bits decreases making decoding less efficient thus more Eb/No(SNR) required to decode it.

- **Success Probability vs Number of Iterations for CodeRate=0.5 for NR_1_5_352**



From the graph, it is evident that for Eb/No values greater than or equal to 3 dB, the decoder is able to successfully decode all transmitted messages without any bit errors. However, the number of decoding iterations required to achieve successful decoding varies.

Conclusion:

- Soft decoding proves to be significantly more efficient than hard decoding, as it leverages probabilistic information (beliefs) from other received bits, not just binary decisions.
- With each iteration, the accuracy of decoding improves, increasing the likelihood of successful correction of errors.
- The performance further enhances with lower code rates, as a higher number of parity bits provide more constraints, leading to faster and more reliable decoding.

6 Shannon Approximation

- **Code :**

```
% Load 5G NR LDPC base matrix
baseGraph5GNR = 'NR_2_6_52';
[B, H, z] = nrldpc_Hmatrix(baseGraph5GNR);

% Code rates to simulate - CORRECTED: Define all four rates
R = [1/4, 1/3, 1/2, 3/5];
% EbNo range
EbNo_dB_range = -3:0.25:4;
%  simulation
Nsim = 200;
%  decoding iterations
max_iter = 20;


BER = zeros(length(R), length(EbNo_dB_range));
p_error = zeros(length(R), length(EbNo_dB_range));
NA_BLER = zeros(length(R), length(EbNo_dB_range));

%  Shannon limit for BPSK
shannon_limit_dB = 10*log10((2.^R - 1) ./ (2*R));

for crIdx = 1:length(R)
    codeRate = R(crIdx);
    [mb, nb] = size(B);
    k = (nb - mb) * z;
    numShortened = 2 * z;

    % Calculate puncturing
    totalBitsAfterShortening = nb*z - numShortened;
    actualInfoBits = k - numShortened;
    desiredCodewordLength = actualInfoBits / codeRate;
    desiredCodewordLength = max(desiredCodewordLength, actualInfoBits);
    numPunctured = totalBitsAfterShortening - desiredCodewordLength;
    numPunctured = max(0, round(numPunctured));
    availableParity = mb*z - (numShortened - (nb - mb)*z);
    availableParity = max(0, availableParity);
    numPunctured = min(numPunctured, availableParity);
    numPunctured = floor(numPunctured / z) * z;

    actualCodeRate = actualInfoBits / (totalBitsAfterShortening - numPunctured);
    N = totalBitsAfterShortening - numPunctured;

    fprintf('Target R=%.2f, Actual R=%.4f, N=%d\n', codeRate, actualCodeRate, N);

    for ebNoIdx = 1:length(EbNo_dB_range)
        EbNo_dB = EbNo_dB_range(ebNoIdx);
        EbNo_linear = 10^(EbNo_dB/10);


        % For BPSK over AWGN, SNR = 2*R*Eb/N0
```

```
        SNR_linear = 2 * actualCodeRate * EbNo_linear;
        C = 0.5 * log2(1 + SNR_linear); % Capacity for real AWGN channel

        % Dispersion required for BPSK
        V = 0;
        if SNR_linear > 0
            V = SNR_linear * (2 + SNR_linear) / (2 * (1 + SNR_linear)^2) *
(log2(exp(1)))^2;
        end

        % Normal approximation
        if V > 0
            argument = sqrt(N/V) * (C - actualCodeRate);
            NA_BLER(crIdx, ebNoIdx) = qfunc(argument);
        else
            NA_BLER(crIdx, ebNoIdx) = 1; % If dispersion is zero, set BLER to 1
        end

        % Simulating with early stopping
        sigma = sqrt(1 / (2 * actualCodeRate * EbNo_linear));
        errorCount = 0;
        bitError = 0;

        % Early stopping for high SNR points to speed up simulation
        target_errors = 100;
        max_frames = Nsim;

        sim = 0;
        while (errorCount < target_errors) && (sim < max_frames)
            sim = sim + 1;

            % random information bits
            infoBits = randi([0 1], actualInfoBits, 1);

            % Padding with shortened bits
            paddedInfo = [zeros(numShortened, 1); infoBits]';

            % Encoding using LDPC
            codeword = nrldpc_encode(B, z, paddedInfo);

            % Removing shortened and punctured bits
            tx_codeword = codeword(numShortened + 1 : end - numPunctured);

            % BPSK modulation
            txSig = 1 - 2 * tx_codeword;

            % Adding AWGN
            rxSig = txSig + sigma * randn(size(txSig));

            % LLR calculation
```

```
            llr_transmitted = 2 * rxSig / (sigma^2);

            % Preparing LLRs for decoder
            llr = [1e5*ones(1, numShortened), llr_transmitted, zeros(1,
numPunctured)];

            % Decoding using Min-Sum
            decodedBits = min_sum_ldpc_decoder(llr, H, max_iter);

            % Extracting information bits
            receivedInfo = decodedBits(numShortened + 1 : numShortened +
actualInfoBits);

            % Checking for errors
            has_errors = any(receivedInfo ~= infoBits');

            % Counting errors
            bit_errors_in_frame = sum(receivedInfo ~= infoBits');
            bitError = bitError + bit_errors_in_frame;
            errorCount = errorCount + has_errors;


            if mod(sim, 100) == 0
                fprintf('  R=%.2f, Eb/No=%.2f dB: %d/%d frames, %d errors\n', ...
                    codeRate, EbNo_dB, sim, max_frames, errorCount);
            end
        end

        % Calculating error rates
        BER(crIdx, ebNoIdx) = bitError / (sim * actualInfoBits);
        p_error(crIdx, ebNoIdx) = errorCount / sim;


        if p_error(crIdx, ebNoIdx) == 0
            p_error(crIdx, ebNoIdx) = 1/(10*sim);
        end
    end
end


% Ploting BLER
figure('Position', [100, 100, 800, 600]);
colors = {[1 1 0], 'r', 'b', [0.5 0 0.5]}; % Yellow, Red, Blue, Purple

hold on;
h_all = []; % Combined handle list for legend
legend_labels = {}; %  labels

% Ploting simulated BLER (solid lines)
for i = 1:length(R)
```

```
    h_sim = semilogy(EbNo_dB_range, p_error(i,:), ...
        'Color', colors{i}, 'LineStyle', '-', 'LineWidth', 2);
    h_all = [h_all, h_sim];
    legend_labels{end+1} = sprintf('R=%.2f Sim', R(i));
end

% Ploting NA BLER (dashed lines)
for i = 1:length(R)
    h_na = semilogy(EbNo_dB_range, NA_BLER(i,:), ...
        'Color', colors{i}, 'LineStyle', '--', 'LineWidth', 2);
    h_all = [h_all, h_na];
    legend_labels{end+1} = sprintf('R=%.2f NA', R(i));
end

% Ploting Shannon limits (dotted vertical lines)
for i = 1:length(R)
    h_sh = xline(shannon_limit_dB(i), ':', ...
        sprintf('Shannon R=%.2f', R(i)), ...
        'LineWidth', 1.5, 'Color', colors{i});
    h_all = [h_all, h_sh];
    legend_labels{end+1} = sprintf('Shannon R=%.2f', R(i));
end

% Final plot
legend(h_all, legend_labels, 'Location', 'southwest', 'FontSize', 10);
title('BLER vs Eb/No (BPSK over AWGN)', 'FontSize', 14, 'FontWeight', 'bold');
xlabel('Eb/No (dB)', 'FontSize', 12, 'FontWeight', 'bold');
ylabel('BLER', 'FontSize', 12, 'FontWeight', 'bold');
grid on;
set(gca, 'FontSize', 11);
axis([min(EbNo_dB_range) max(EbNo_dB_range) 1e-4 1]);

% Adding textbox
dim = [0.15 0.7 0.3 0.2];
str = sprintf('Simulation Parameters:\nNsim = %d\nIterations = %d\nBase Graph: %s',
...
             Nsim, max_iter, baseGraph5GNR);
annotation('textbox', dim, 'String', str, 'FitBoxToText', 'on', ...
           'BackgroundColor', 'white', 'EdgeColor', 'black');

hold off;

% Saving the figure
saveas(gcf, 'LDPC_BLER_vs_Shannon_clean.png');
fprintf('Plot saved as LDPC_BLER_vs_Shannon_clean.png\n');


% Min-Sum Decoder Function
function decodedBits = min_sum_ldpc_decoder(llr, H, max_iter)
    [m, n] = size(H);
```

```
    llr_in = llr(:)';
    msg_c2v = zeros(m, n);
    msg_v2c = zeros(m, n);

    % Find non-zero elements in H
    [row, col] = find(H);


    edges = [row, col];
    num_edges = length(row);

    % variable node's connected check nodes
    var_to_check = cell(n, 1);
    for j = 1:n
        var_to_check{j} = find(H(:, j));
    end

    % check node's connected variable nodes
    check_to_var = cell(m, 1);
    for i = 1:m
        check_to_var{i} = find(H(i, :));
    end

    % Initialize variable-to-check messages
    for idx = 1:num_edges
        i = row(idx); j = col(idx);
        msg_v2c(i,j) = llr_in(j);
    end

    % Normalized Min-Sum parameters
    alpha = 0.8;  % Scaling factor for check node update
    beta = 0.5;   % Self-correction factor

    % Decoding iterations
    for iter = 1:max_iter
        % Storing previous messages for self-correction
        if iter > 1
            prev_msg_v2c = msg_v2c;
        end

        % Check Node Update
        for i = 1:m
            cols = check_to_var{i};
            if length(cols) < 2
                continue;  % Skiping CNs with less than 2 connections
            end

            % Calculating sign product and find minimum magnitudes
            signs = sign(msg_v2c(i, cols));
            abs_msgs = abs(msg_v2c(i, cols));
```

```
        sign_product = prod(signs);

        % Finding the two smallest values
        [min1, idx1] = min(abs_msgs);
        abs_msgs(idx1) = Inf;
        min2 = min(abs_msgs);

        % Applying normalized min-sum to each edge
        for j = 1:length(cols)
            col_idx = cols(j);


            edge_sign = sign_product * signs(j);

            % Calculating magnitude
            if j == idx1
                min_val = min2;
            else
                min_val = min1;
            end

            % Applying  normalization factor alpha
            msg_c2v(i, col_idx) = edge_sign * min_val * alpha;
        end
    end

    % VN Update
    for j = 1:n
        rows = var_to_check{j};
        if isempty(rows)
            continue;  % Skip isolated VNs
        end

        % Calculate sum of all check-to-variable messages
        sum_c2v = sum(msg_c2v(rows, j));


        for i = rows'
            msg_v2c(i, j) = llr_in(j) + (sum_c2v - msg_c2v(i, j));
        end
    end

    % Self-correction
    if iter > 1
        for idx = 1:num_edges
            i = row(idx); j = col(idx);
            if sign(msg_v2c(i,j)) ~= sign(prev_msg_v2c(i,j))
```

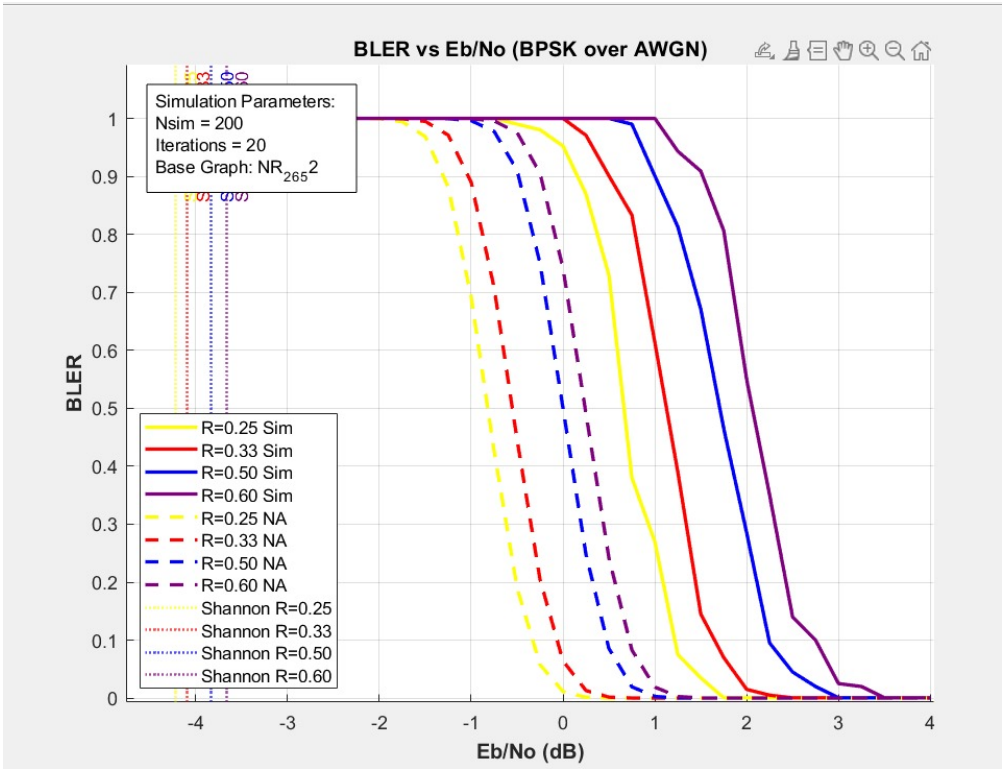
```
                    msg_v2c(i,j) = beta * msg_v2c(i,j) + (1-beta) *
prev_msg_v2c(i,j);
                end
            end
        end

        % Computing posterior LLRs
        posterior_llr = llr_in;
        for j = 1:n
            rows = var_to_check{j};
            posterior_llr(j) = llr_in(j) + sum(msg_c2v(rows, j));
        end

        % Making hard decisions
        decodedBits = (posterior_llr < 0);

        % Check if valid codeword (H * c' = 0 in GF(2))
        syndrome = mod(H * decodedBits', 2);
        if all(syndrome == 0)
            % Valid codeword found - early termination
            break;
        end
    end
end


function [B,H,z] = nrldpc_Hmatrix(BG)

load(sprintf('%s.txt',BG),BG);
B = NR_2_6_52; % change for 2nd Base matrix
[mb,nb] = size(B);
z = 52; % change for 2nd Base matrix
H = zeros(mb*z,nb*z);
Iz = eye(z); I0 = zeros(z);
for kk = 1:mb
    tmpvecR = (kk-1)*z+(1:z);
    for kk1 = 1:nb
        tmpvecC = (kk1-1)*z+(1:z);
        if B(kk,kk1) == -1
            H(tmpvecR,tmpvecC) = I0;
        else
            H(tmpvecR,tmpvecC) = circshift(Iz,-B(kk,kk1));
        end
    end
end

[U,N]=size(H); K = N-U;
P = H(:,1:K);
G = [eye(K); P];
Z = H*G;
```

```
end

% Function to encode message using Base Matrix
function cword = nrldpc_encode(B,z,msg)

%B: base matrix
%z: expansion factor
%msg: message vector, length = (#cols(B)-#rows(B))*z
%cword: codeword vector, length = #cols(B)*z


[m,n] = size(B);

cword = zeros(1,n*z);
cword(1:(n-m)*z) = msg;


%double-diagonal encoding
temp = zeros(1,z);
for i = 1:4 %row 1 to 4
    for j = 1:n-m %message columns
        temp = mod(temp + mul_sh(msg(((j-1)*z+1):(j*z)),B(i,j)),2);
    end
end
if B(2,n-m+1) == -1
    p1_sh = B(3,n-m+1);
else
    p1_sh = B(2,n-m+1);
end


cword((n-m)*z+1:(n-m+1)*z) = mul_sh(temp,z-p1_sh); %p1
%Find p2, p3, p4
for i = 1:3
    temp = zeros(1,z);
    for j = 1:n-m+i
        temp = mod(temp + mul_sh(cword(((j-1)*z+1):(j*z)),B(i,j)),2);
    end
    cword((n-m+i)*z+1:(n-m+i+1)*z) = temp;
end


%Remaining parities
for i = 5:m
    temp = zeros(1,z);
    for j = 1:n-m+4
        temp = mod(temp + mul_sh(cword(((j-1)*z+1):(j*z)),B(i,j)),2);
    end
    cword((n-m+i-1)*z+1:(n-m+i)*z) = temp;
```

```
end


end

function y = mul_sh(x, k)

 if(k == -1)
    y = zeros(1, length(x));
 else
    y = [x(k+1 : end) x(1 : k)];
 end
 end
```

```
Target R=0.25, Actual R=0.2500, Punctured 936 bits
```

- **Graph for BLER vs E_b/N_0 (BPSK over AWGN)**



This plot shows how the Block Error Rate (BLER) changes with increasing Eb/No (energy per bit to noise power spectral density ratio) for different code rates of LDPC codes. The simulation is performed using 5G NR Base Graph 2 (BG2) with a lifting size of Z = 52, over an AWGN channel with BPSK modulation.

1.As Eb/No increases, the BLER decreases, meaning the communication becomes more reliable.

2.Lower code rates (like R = 0.25) achieve lower BLER at smaller Eb/No values because they add more redundancy, which helps correct errors.

3.Higher code rates (like R = 0.60) require higher Eb/No to reach the same level of reliability.

4.The simulated performance curves should ideally lie above the Shannon limit lines, since no real system can surpass the theoretical limit.

7 References

1. Yash Vasavada, *Lecture Slides of Channel Coding*

2. Andrew Thangaraj, *NPTEL Video Lectures on LDPC Codes*, IIT Madras.

3. Lifang Wang, *Implementation of Low-Density Parity-Check codes for 5G NR shared channels*