Dhirubhai Ambani University

CT - 216 : Communication Systems

Prof. Yash Vasavada

Mentor : Vivek Patel

Group - 7 : LDPC Codes

PRINCE CHOVATIYA – 202301067

RISHANK DUDHAT – 202301068

YASHASVI ISHWARBHAI JADAV – 202301069

HEET SHAH – 202301070

SIDDHARTH RAMBHIA – 202301072

MEET JAIN – 202301073

KAVISH PATEL – 202301074

TIRTH KANANI – 202301075

PALA AADITYA VIMALKUMAR – 202301076

BHENSADADIA HAPPY – 202301077

Hard Decision Decoding for LDPC Codes

Low-Density Parity-Check (LDPC) codes are highly effective error-correcting codes employed in contemporary communication systems. If LDPC codes are decoded via hard decision decoding, the algorithm in this case only makes binary (0 or 1) decisions from the received signal, not based on probability or likelihood values like in soft decision decoding.

---

1. Hard Decision Decoding: Overview

In **hard decision decoding**, the received signal is first quantized to binary values:

- A transmitted bit `0` is mapped to `+1`, and `1` is mapped to `-1` (using BPSK modulation).
- The received signal (after passing through AWGN) is thresholded at 0:
    - If received value > 0 → decide as `0`
    - If received value ≤ 0 → decide as `1`

This gives a **binary received vector**, which is passed to the LDPC decoder.

---

2. Decoding Algorithm: Bit-Flipping (Hard Decision)

A common method for hard decision decoding of LDPC codes is the **Bit-Flipping algorithm**, which works as follows:

Step 1: Initialization

- Use the hard-decision received vector as the initial estimate of the codeword.

Step 2: Iterative Decoding

- For a fixed number of iterations (e.g., 30):
    - For each **check node** (each parity-check equation):
        - Check if the parity-check is satisfied (i.e., the XOR of connected bits is 0).
    - For each **variable node** (bit):
        - Count how many of the parity-checks involving that bit are unsatisfied.
        - If the number exceeds a threshold, **flip the bit** (i.e., 0 ↔ 1).

- If all parity checks are satisfied (i.e., `H × decodedBits' = 0`), stop decoding.

Step 3: Output

- The final estimated codeword (decoded bits).
- A flag indicating whether decoding was successful (i.e., all parity checks are satisfied).

---

3. Advantages and Limitations

Advantages	Limitations
Low complexity and easy to implement	Poorer performance compared to soft decision decoding
Faster decoding due to simple operations	May fail to converge, especially in noisy channels
Suitable for hardware-constrained systems	Cannot leverage confidence (likelihood) from the channel

---

4. In the Code (`hardDecoding`)

The function `hardDecoding(H, rx, maxIterations, c2v_map, v2c_map)` likely implements this bit-flipping algorithm:

- `rx` is thresholded to generate binary values.
- Iterative updates are done based on the parity-check matrix `H` and the message-passing maps `c2v_map`, `v2c_map`.
- The function returns:
  - `decBits`: the decoded bit vector.
  - `success`: a binary flag indicating whether decoding was successful.

The Shannon Limit

The Shannon limit represents the theoretical minimum E_b/N_0 required to achieve reliable communication at a given code rate R with an arbitrarily low error probability, assuming an infinitely long code.

In the simulation, the Shannon limit is calculated as:

$$\left(\frac{E_b}{N_0}\right)_{\text{min}} = \frac{2^R - 1}{R}$$

and plotted in decibels as:

$$\left(\frac{E_b}{N_0}\right)_{\text{dB}} = 10\log_{10}\left(\frac{2^R - 1}{R}\right)$$

This formula is derived for an AWGN channel with Gaussian input, as follows:

For a complex AWGN channel with Gaussian input, the channel capacity C (in bits per channel use) is:

$$C = \log_2\left(1 + \frac{P}{N_0}\right)$$

where P is the signal power per channel use, and N_0 is the noise power spectral density. For reliable communication, the code rate R must satisfy $R < C$. In the simulation, each transmitted symbol (using BPSK modulation) has energy $E_c = 1$, so $P = E_c$. Since R bits are transmitted per channel use, the energy per information bit E_b is:

$$E_b = \frac{E_c}{R} = \frac{1}{R}$$

Thus, the signal-to-noise ratio is:

$$\frac{P}{N_0} = \frac{E_c}{N_0} = \frac{R \cdot E_b}{N_0} = R \cdot \frac{E_b}{N_0}$$

Setting $R = C$ at the limit of reliable communication:

$$R = \log_2\left(1 + R \cdot \frac{E_b}{N_0}\right)$$

Solving for $\frac{E_b}{N_0}$:

$$2^R = 1 + R \cdot \frac{E_b}{N_0}$$

$$R \cdot \frac{E_b}{N_0} = 2^R - 1$$

$$\frac{E_b}{N_0} = \frac{2^R - 1}{R}$$

This is the formula used in the code. In the plot, it appears as a vertical line, indicating that for $\frac{E_b}{N_0}$ values above this threshold, error-free communication is theoretically possible with sufficiently long codes. Below this limit, reliable communication is not achievable, regardless of coding complexity.

The Normal Approximation

The Shannon limit assumes an infinite block length, which is impractical for real systems. The normal approximation addresses this by estimating the block error rate for codes with finite block length N. It accounts for the channel capacity C and the channel dispersion V, which quantifies the variability in the channel's performance for finite lengths.

In the simulation, the block error rate is approximated as:

$$P_{N_e} \approx Q\left(\frac{\sqrt{N}\left(C - R + \frac{\log_2 N}{2N}\right)}{\sqrt{V}}\right)$$

where:

- $Q(x) = \frac{1}{2}\mathrm{erfc}\left(\frac{x}{\sqrt{2}}\right)$ is the Gaussian Q-function.

- $C = \log_2(1 + P)$, with $P = R \cdot \frac{E_b}{N_0}$.

- $V = (\log_2 e)^2 \cdot \frac{P(P+2)}{(P+1)^2}$ is the channel dispersion.

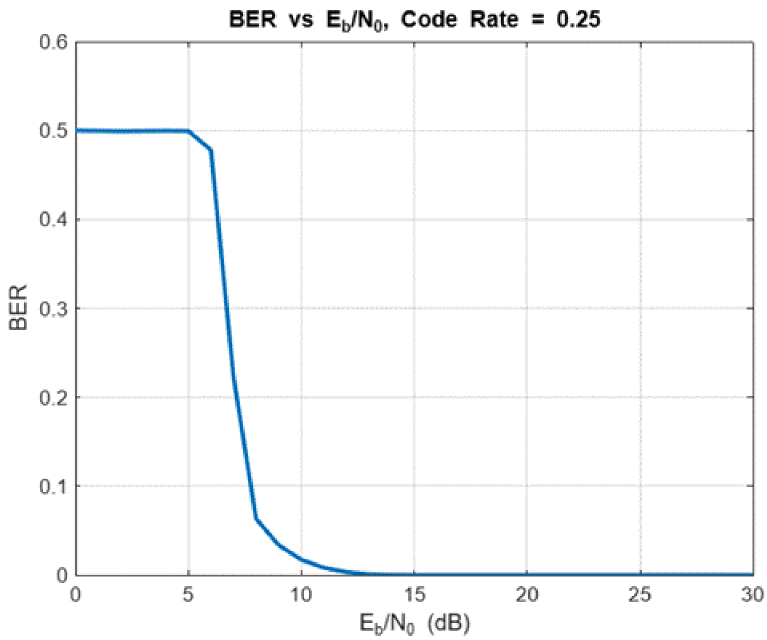
- N is the block length.

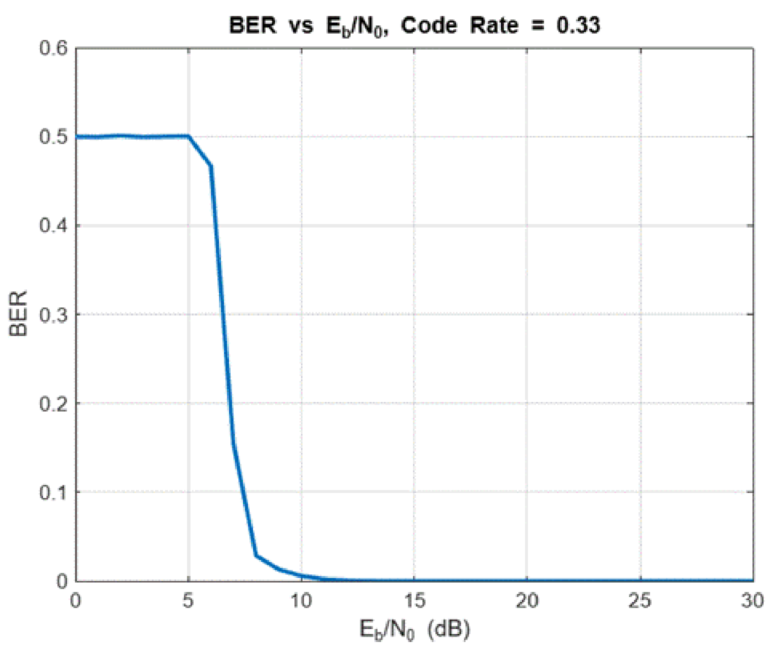
- R is the code rate.

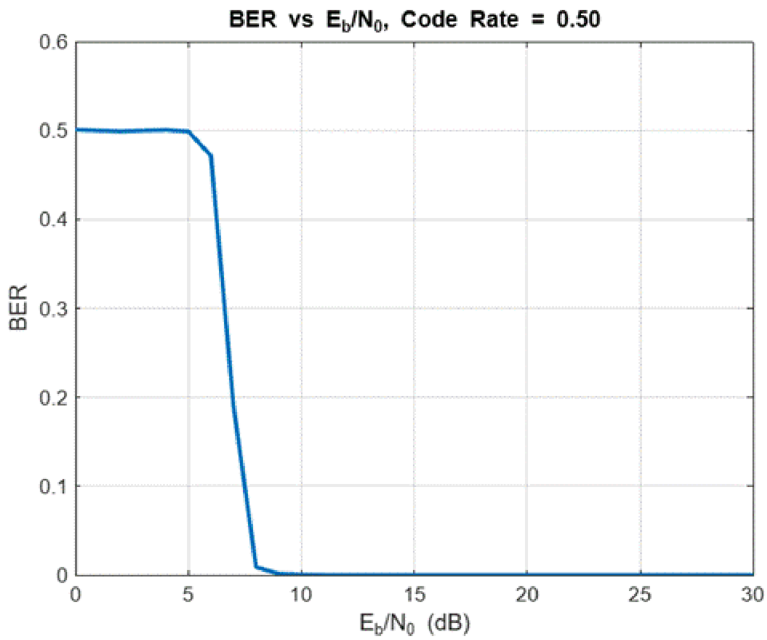
1) Hard Decoding : NR_2_6_52

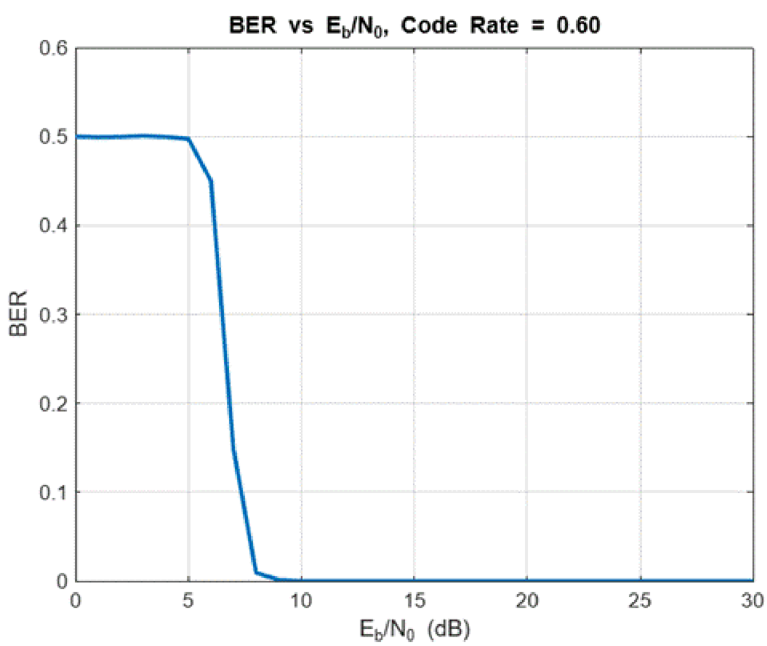
```
clear;
clc;

baseGraph = 'NR_2_6_52';
codeRates = [1/4, 1/3, 1/2, 3/5];
EbN0dBList = 0 : 1 : 30;
numTrials = 500;
maxIterations = 30;

[B, Hfull, z] = nrldpc_Hmatrix(baseGraph);
[mb, nb] = size(B);
kb = nb - mb;
numInfoBits = kb * z;
parityCols = kb - 2;

decodingErr = zeros(length(codeRates), numel(EbN0dBList));
probSuccess = zeros(length(codeRates), numel(EbN0dBList));

for rIdx = 1 : length(codeRates)
        rate = codeRates(rIdx);
        [H, blockLen] = H_matrix(Hfull, z, mb, nb, parityCols, rate);
        c2v_map = get_c2v(H);
        v2c_map = get_v2c(H);

        for eIdx = 1 : length(EbN0dBList)
        EbN0dB = EbN0dBList(eIdx);
```

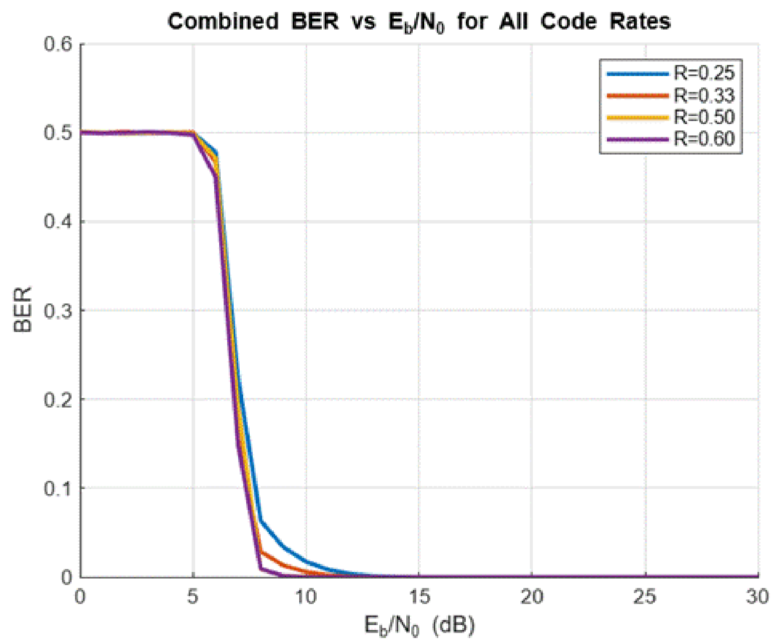
```
        for trial = 1:numTrials

        msg = randi([0, 1], numInfoBits, 1);
        codeword = encodeLDPC(B, z, msg, blockLen);

        tx = 1 - 2 * codeword;
        rx = addAWGN(tx, rate, EbN0dB);

        [decBits, success] = hardDecoding(H, rx, maxIterations, c2v_map, v2c_map);

        decodingErr(rIdx, eIdx) = decodingErr(rIdx, eIdx) + bitErrorRate(codeword, decBits);
        probSuccess(rIdx, eIdx) = probSuccess(rIdx, eIdx) + success;
        end

        decodingErr(rIdx, eIdx) = decodingErr(rIdx, eIdx) / numTrials;
        probSuccess(rIdx, eIdx) = probSuccess(rIdx, eIdx) / numTrials;
        end
end

plotIndividualAndCombined(EbN0dBList, decodingErr, probSuccess, codeRates);
```

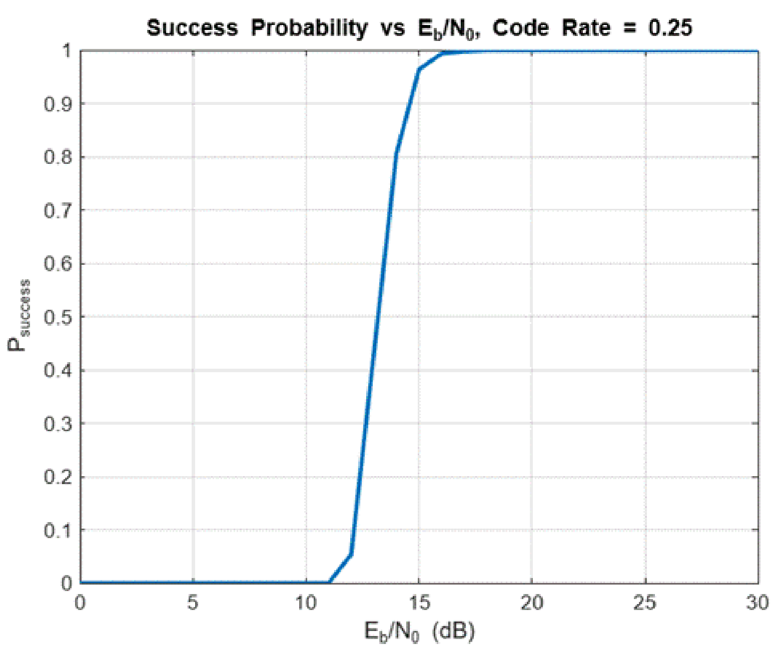

BER vs E_b/N_0, Code Rate = 0.25
BER
E_b/N_0 (dB)
0
0.1
0.2
0.3
0.4
0.5
0.6
0
5
10
15
20
25
30

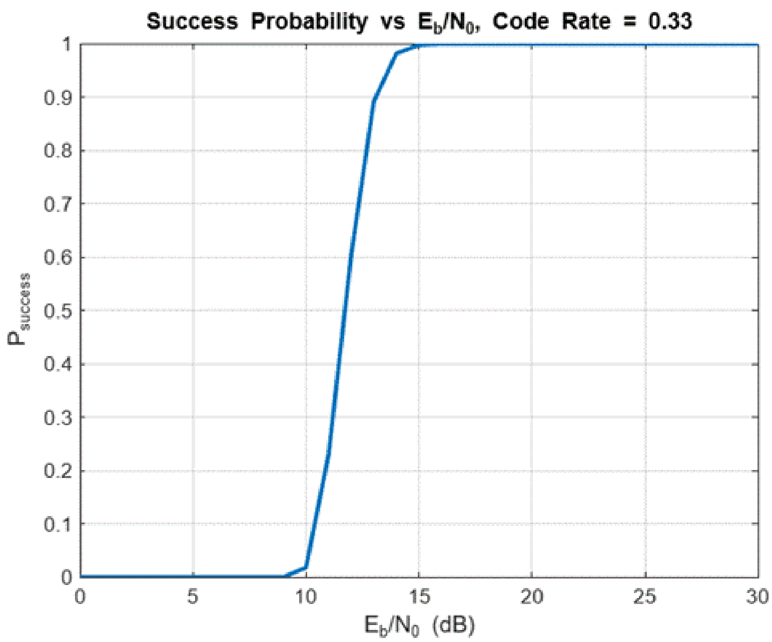

BER vs E_b/N_0, Code Rate = 0.33
BER
E_b/N_0 (dB)
0
0.1
0.2
0.3
0.4
0.5
0.6
0
5
10
15
20
25
30

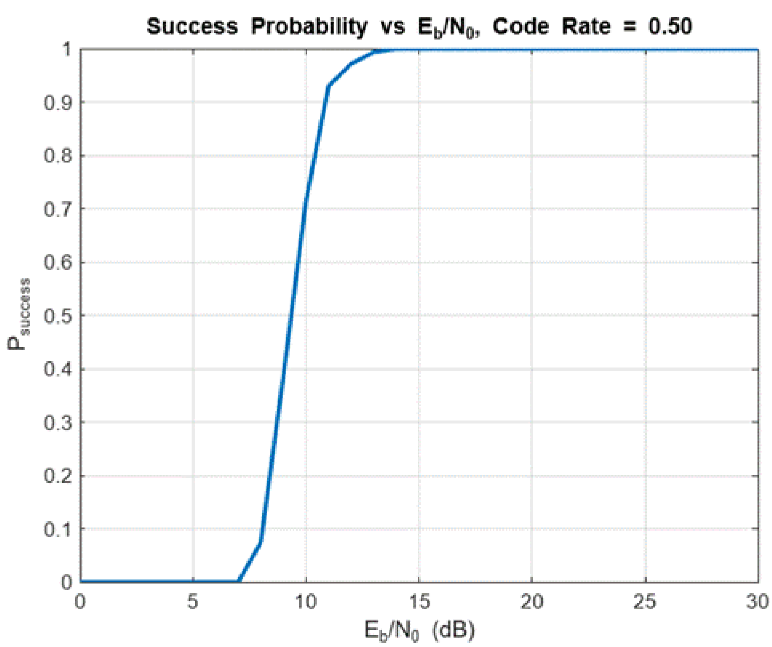

BER vs E_b/N_0, Code Rate = 0.50
BER
E_b/N_0 (dB)
0.6
0.5
0.4
0.3
0.2
0.1
0
0
5
10
15
20
25
30

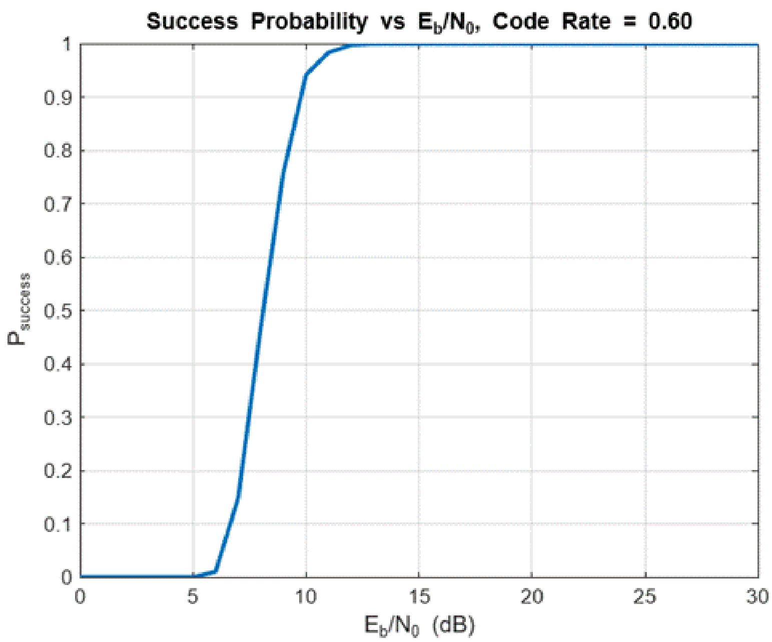

BER vs E_b/N_0, Code Rate = 0.60
BER
E_b/N_0 (dB)
0.6
0.5
0.4
0.3
0.2
0.1
0
0
5
10
15
20
25
30

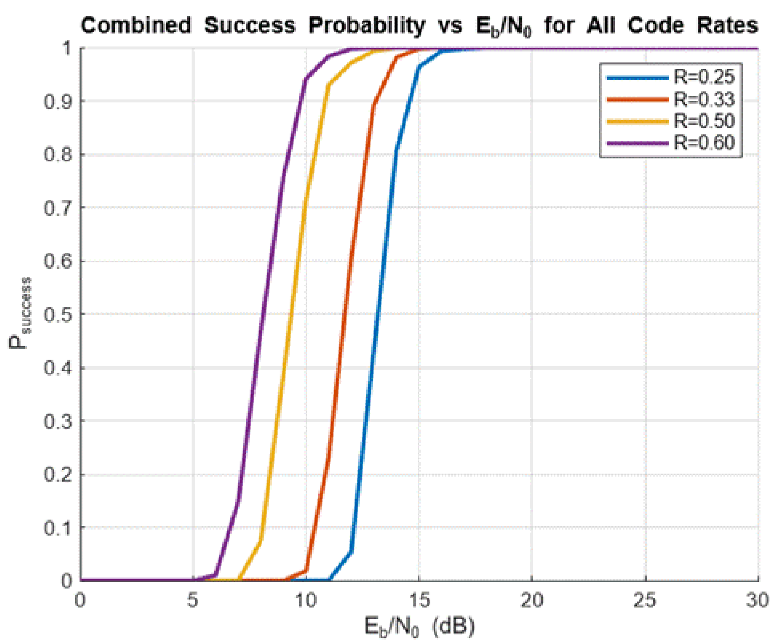

Combined BER vs E_b/N_0 for All Code Rates
BER
E_b/N_0 (dB)
R=0.25
R=0.33
R=0.50
R=0.60
0
0.1
0.2
0.3
0.4
0.5
0.6
5
10
15
20
25
30


Success Probability vs E_b/N_0, Code Rate = 0.25
$P_{success}$
E_b/N_0 (dB)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
5
10
15
20
25
30


Success Probability vs E_b/N_0, Code Rate = 0.33
$P_{success}$
E_b/N_0 (dB)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
5
10
15
20
25
30


Success Probability vs E_b/N_0, Code Rate = 0.50
$P_{success}$
E_b/N_0 (dB)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
5
10
15
20
25
30






```
function plotIndividualAndCombined(EbN0dBList, decodingErr, probSuccess, codeRates)

    for i = 1:numel(codeRates)

    figure;
```

```
plot(EbN0dBList, decodingErr(i, :), 'LineWidth', 2);
grid on;
xlabel('E_b/N_0 (dB)');
ylabel('BER');
title(sprintf('BER vs E_b/N_0, Code Rate = %.2f', codeRates(i)));
end

figure;
hold on;
for i = 1:numel(codeRates)
plot(EbN0dBList, decodingErr(i, :), 'LineWidth', 2);
end

grid on;
xlabel('E_b/N_0 (dB)');
ylabel('BER');
title('Combined BER vs E_b/N_0 for All Code Rates');
legend(arrayfun(@(r) sprintf('R=%.2f', r), codeRates, 'UniformOutput', false));

for i = 1 : length(codeRates)
figure;
plot(EbN0dBList, probSuccess(i, :), 'LineWidth', 2);
grid on;
xlabel('E_b/N_0 (dB)');
ylabel('P_{success}');
title(sprintf('Success Probability vs E_b/N_0, Code Rate = %.2f', codeRates(i)));
end

figure;
```

```
        hold on;
        for i = 1:numel(codeRates)
        plot(EbN0dBList, probSuccess(i, :), 'LineWidth', 2);
        end

        grid on;
        xlabel('E_b/N_0 (dB)');
        ylabel('P_{success}');
        title('Combined Success Probability vs E_b/N_0 for All Code Rates');
        legend(arrayfun(@(r) sprintf('R=%.2f', r), codeRates, 'UniformOutput', false));
end

function [H, nBlockLength] = H_matrix(Hfull, z, mb, nb, parityCols, rate)
        nbRM = ceil(parityCols / rate) + 2;
        nBlockLength = nbRM * z;
        Htrunc = Hfull(:, 1:nBlockLength);
        nChecksRemain = mb * z - nb * z + nBlockLength;
        H = Htrunc(1:nChecksRemain, :);
end

function c = encodeLDPC(B, z, msg, nBlockLength)
        cw = nrldpc_encode(B, z, msg');
        c = cw(1:nBlockLength)';
end

function rx = addAWGN(tx, rate, EbN0dB)
        gamma = 10^(EbN0dB / 10);
        sigma = sqrt(1 / (2 * rate * gamma));
        rx = tx + sigma * randn(size(tx));
```

```
end

function [decodedBits, success] = hardDecoding(H, rx, maxIter, c2v_map, v2c_map)
    nVars   = size(H, 2);
    decoded = (rx < 0);
    prev    = decoded;
    success = false;

    % Initialize VN->CN messages
    v2c_msgs = zeros(size(H));
    for vn = 1:nVars
    for cn = v2c_map{vn}
    v2c_msgs(cn, vn) = decoded(vn);
    end
    end
    c2v_msgs = zeros(size(H));

    for iter = 1:maxIter
    % Check->Variable update
    for cn = 1:numel(c2v_map)
    vn_list = c2v_map{cn};
    xor_val = 0;
    for vn = vn_list
    xor_val = xor(xor_val, v2c_msgs(cn, vn));
    end
    for vn = vn_list
    c2v_msgs(cn, vn) = xor(xor_val, v2c_msgs(cn, vn));
    end
    end
```

```
% Variable->Check update and new estimate
new_est = zeros(nVars, 1);
for vn = 1:nVars
cn_list  = v2c_map{vn};
dv          = numel(cn_list);
totalOnes = decoded(vn) + sum(c2v_msgs(cn_list, vn));

for cn = cn_list
excl = totalOnes - c2v_msgs(cn, vn);
v2c_msgs(cn, vn) = (excl > dv/2);
end

new_est(vn) = (totalOnes > floor(dv/2));
end

% Syndrome check
if all(mod(H * new_est, 2) == 0)
decoded = new_est;
success = true;
break;
end
if isequal(new_est, prev)
decoded = new_est;
break;
end

prev    = new_est;
decoded = new_est;
```

```
    end

    decodedBits = decoded;
end

function c2v = get_c2v(H)
    for i = 1:size(H,1)
    c2v{i} = find(H(i,:));
    end
end

function v2c = get_v2c(H)
    for j = 1:size(H,2)
    v2c{j} = find(H(:,j))';
    end
end

function ber = bitErrorRate(orig, dec)
    ber = sum(xor(orig, dec)) / numel(orig);
end

function [B, H, z] = nrldpc_Hmatrix(BG)
    load(sprintf('%s.txt', BG), BG);
    B = eval(BG);
    [mb, nb] = size(B);
    z     = max(B(:)) + 1;
    Iz    = eye(z);
    I0    = zeros(z);
    H     = zeros(mb*z, nb*z);
```

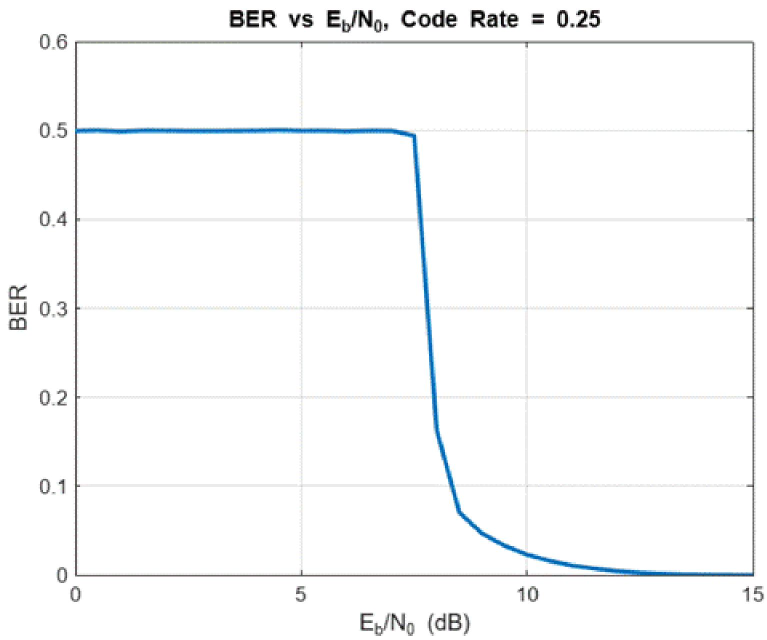
```
        for ii = 1:mb
        rows = (ii-1)*z + (1:z);
        for jj = 1:nb
        cols = (jj-1)*z + (1:z);
        if B(ii,jj) == -1
        H(rows, cols) = I0;
        else
        H(rows, cols) = circshift(Iz, -B(ii,jj));
        end
        end
        end
end

function cword = nrldpc_encode(B, z, msg)
        [m, n] = size(B);
        cword  = zeros(1, n*z);
        cword(1:(n-m)*z) = msg;
        temp   = zeros(1, z);

        for i = 1:4
        for j = 1:(n-m)
        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
        p = B(3, n-m+1);
        else
```

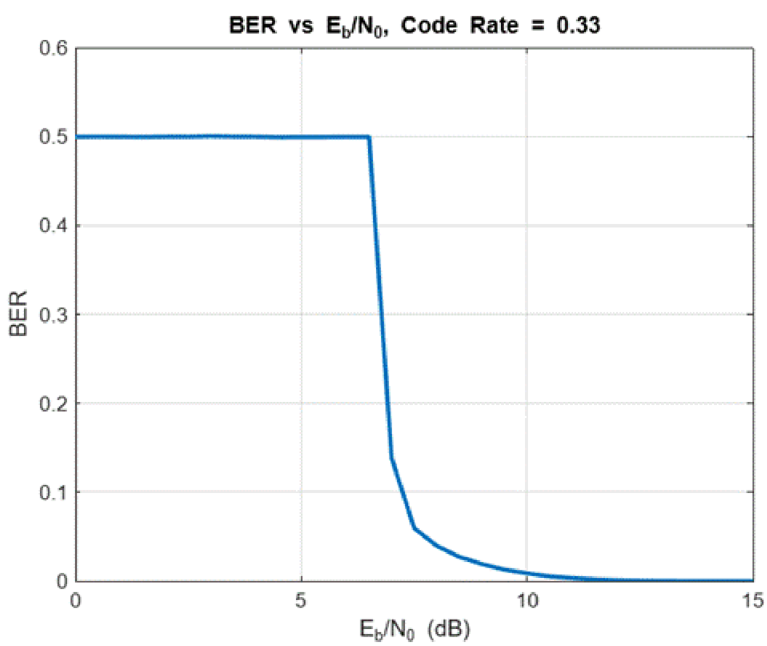
```
        p = B(2, n-m+1);
        end
        cword((n-m)*z+1:(n-m+1)*z) = mul_sh(temp, z-p);

        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

        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
```

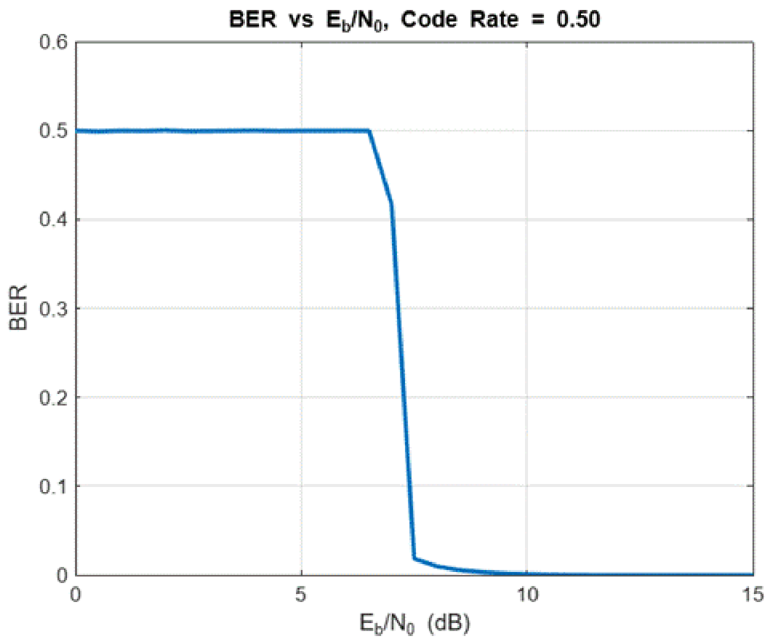
2) Hard Decoding : NR_1_5_352

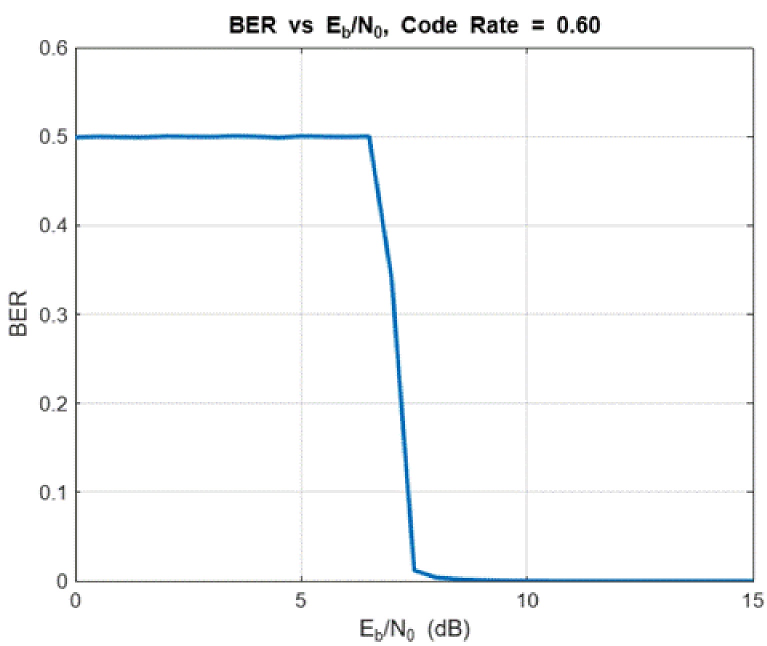
```
clear;
clc;

baseGraph = 'NR_1_5_352';
codeRates = [1/4, 1/3, 1/2, 3/5];
EbN0dBList = 0 : 0.5 : 15;
numTrials = 100;
maxIterations = 20;

[B, Hfull, z] = nrldpc_Hmatrix(baseGraph);
[mb, nb] = size(B);
kb = nb - mb;
numInfoBits = kb * z;
parityCols = kb - 2;

decodingErr = zeros(length(codeRates), numel(EbN0dBList));
probSuccess = zeros(length(codeRates), numel(EbN0dBList));

for rIdx = 1 : length(codeRates)
        rate = codeRates(rIdx);
        [H, blockLen] = H_matrix(Hfull, z, mb, nb, parityCols, rate);
        c2v_map = get_c2v(H);
        v2c_map = get_v2c(H);

        for eIdx = 1 : length(EbN0dBList)
```

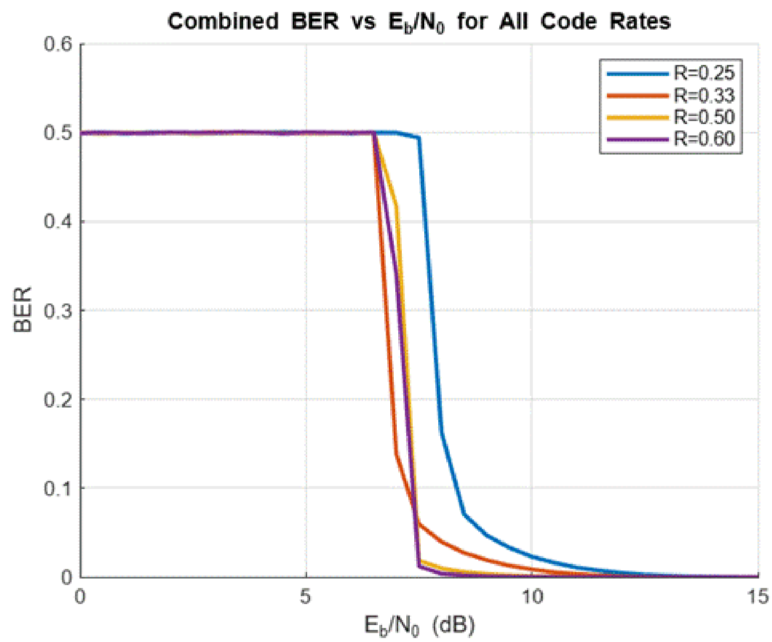
```
        EbN0dB = EbN0dBList(eIdx);

        for trial = 1:numTrials

        msg = randi([0, 1], numInfoBits, 1);
        codeword = encodeLDPC(B, z, msg, blockLen);

        tx = 1 - 2 * codeword;
        rx = addAWGN(tx, rate, EbN0dB);

        [decBits, success] = hardDecoding(H, rx, maxIterations, c2v_map, v2c_map);

        decodingErr(rIdx, eIdx) = decodingErr(rIdx, eIdx) + bitErrorRate(codeword, decBits);
        probSuccess(rIdx, eIdx) = probSuccess(rIdx, eIdx) + success;
        end

        decodingErr(rIdx, eIdx) = decodingErr(rIdx, eIdx) / numTrials;
        probSuccess(rIdx, eIdx) = probSuccess(rIdx, eIdx) / numTrials;
        end
end

plotIndividualAndCombined(EbN0dBList, decodingErr, probSuccess, codeRates);
```

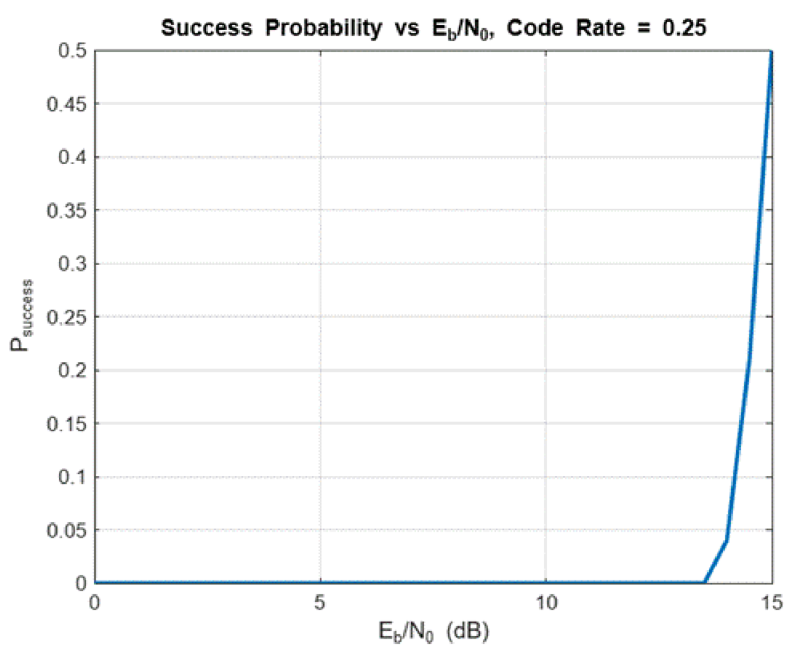

BER vs E_b/N_0, Code Rate = 0.25
BER
E_b/N_0 (dB)
0
0.1
0.2
0.3
0.4
0.5
0.6
0
5
10
15

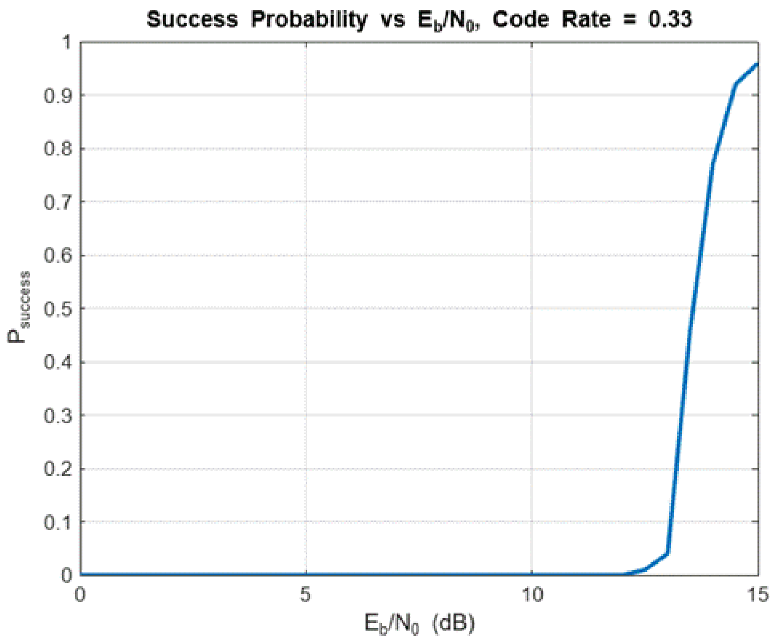

BER vs E_b/N_0, Code Rate = 0.33
BER
E_b/N_0 (dB)
0
0.1
0.2
0.3
0.4
0.5
0.6
0
5
10
15

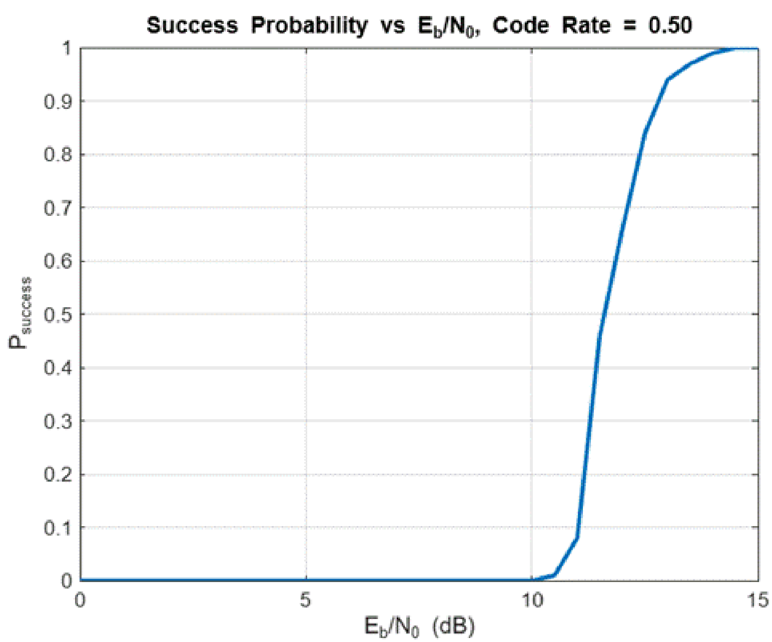

BER vs E_b/N_0, Code Rate = 0.50
0.6
0.5
0.4
0.3
0.2
0.1
0
BER
0
5
10
15
E_b/N_0 (dB)

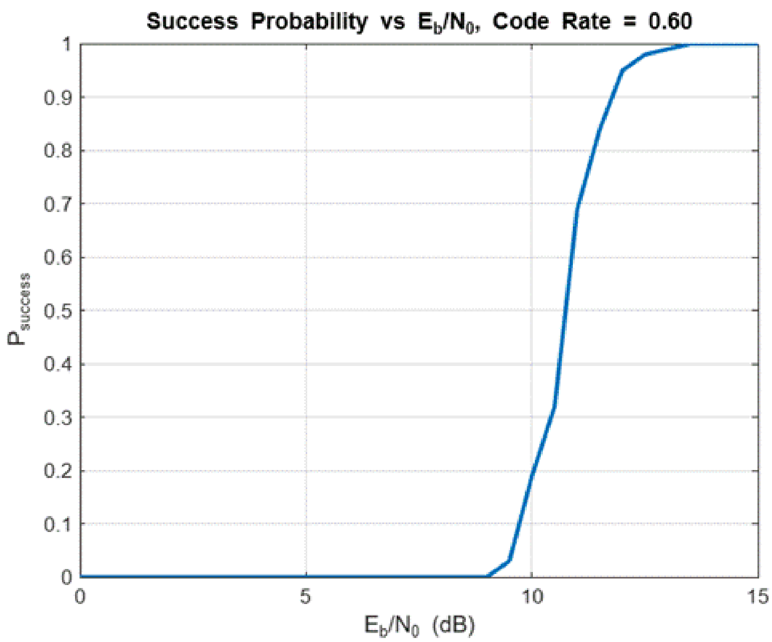

BER vs E_b/N_0, Code Rate = 0.60
0.6
0.5
0.4
0.3
0.2
0.1
0
BER
0
5
10
15
E_b/N_0 (dB)

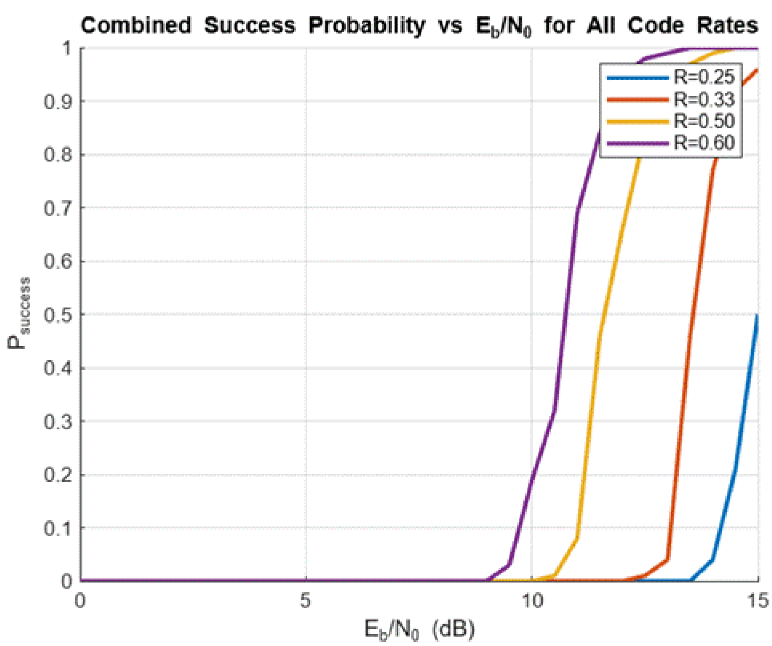

Combined BER vs E_b/N_0 for All Code Rates
BER
E_b/N_0 (dB)
R=0.25
R=0.33
R=0.50
R=0.60
0.6
0.5
0.4
0.3
0.2
0.1
0
5
10
15


Success Probability vs E_b/N_0, Code Rate = 0.25
$P_{success}$
E_b/N_0 (dB)
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
5
10
15


Success Probability vs E_b/N_0, Code Rate = 0.33
$P_{success}$
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
5
10
15
E_b/N_0 (dB)


Success Probability vs E_b/N_0, Code Rate = 0.50
$P_{success}$
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
5
10
15
E_b/N_0 (dB)






```
function plotIndividualAndCombined(EbN0dBList, decodingErr, probSuccess, codeRates)

    for i = 1:numel(codeRates)
```

```
figure;

plot(EbN0dBList, decodingErr(i, :), 'LineWidth', 2);

grid on;

xlabel('E_b/N_0 (dB)');

ylabel('BER');

title(sprintf('BER vs E_b/N_0, Code Rate = %.2f', codeRates(i)));

end


figure;

hold on;

for i = 1:numel(codeRates)

plot(EbN0dBList, decodingErr(i, :), 'LineWidth', 2);

end


grid on;

xlabel('E_b/N_0 (dB)');

ylabel('BER');

title('Combined BER vs E_b/N_0 for All Code Rates');

legend(arrayfun(@(r) sprintf('R=%.2f', r), codeRates, 'UniformOutput', false));


for i = 1 : length(codeRates)

figure;

plot(EbN0dBList, probSuccess(i, :), 'LineWidth', 2);

grid on;

xlabel('E_b/N_0 (dB)');

ylabel('P_{success}');

title(sprintf('Success Probability vs E_b/N_0, Code Rate = %.2f', codeRates(i)));

end
```

```
    figure;
    hold on;
    for i = 1:numel(codeRates)
    plot(EbN0dBList, probSuccess(i, :), 'LineWidth', 2);
    end

    grid on;
    xlabel('E_b/N_0 (dB)');
    ylabel('P_{success}');
    title('Combined Success Probability vs E_b/N_0 for All Code Rates');
    legend(arrayfun(@(r) sprintf('R=%.2f', r), codeRates, 'UniformOutput', false));
end

function [H, nBlockLength] = H_matrix(Hfull, z, mb, nb, parityCols, rate)
    nbRM = min(ceil(parityCols / rate) + 2, nb);
    nBlockLength = nbRM * z;
    Htrunc = Hfull(:, 1:nBlockLength);
    nChecksRemain = mb * z - nb * z + nBlockLength;
    H = Htrunc(1:nChecksRemain, :);
end

function c = encodeLDPC(B, z, msg, nBlockLength)
    cw = nrldpc_encode(B, z, msg');
    c = cw(1:nBlockLength)';
end

function rx = addAWGN(tx, rate, EbN0dB)
    gamma = 10^(EbN0dB / 10);
    sigma = sqrt(1 / (2 * rate * gamma));
```

```
    rx = tx + sigma * randn(size(tx));
end

function [decodedBits, success] = hardDecoding(H, rx, maxIter, c2v_map, v2c_map)
    nVars   = size(H, 2);
    decoded = (rx < 0);
    prev    = decoded;
    success = false;

    % Initialize VN->CN messages
    v2c_msgs = zeros(size(H));
    for vn = 1:nVars
    for cn = v2c_map{vn}
    v2c_msgs(cn, vn) = decoded(vn);
    end
    end
    c2v_msgs = zeros(size(H));

    for iter = 1:maxIter
    % Check->Variable update
    for cn = 1:numel(c2v_map)
    vn_list = c2v_map{cn};
    xor_val = 0;
    for vn = vn_list
    xor_val = xor(xor_val, v2c_msgs(cn, vn));
    end
    for vn = vn_list
    c2v_msgs(cn, vn) = xor(xor_val, v2c_msgs(cn, vn));
    end
```

```
end

% Variable->Check update and new estimate
new_est = zeros(nVars, 1);
for vn = 1:nVars
cn_list  = v2c_map{vn};
dv          = numel(cn_list);
totalOnes = decoded(vn) + sum(c2v_msgs(cn_list, vn));

for cn = cn_list
excl = totalOnes - c2v_msgs(cn, vn);
v2c_msgs(cn, vn) = (excl > dv/2);
end

new_est(vn) = (totalOnes > floor(dv/2));
end

% Syndrome check
if all(mod(H * new_est, 2) == 0)
decoded = new_est;
success = true;
break;
end
if isequal(new_est, prev)
decoded = new_est;
break;
end

prev    = new_est;
```

```
        decoded = new_est;
        end

        decodedBits = decoded;
end

function c2v = get_c2v(H)
        for i = 1:size(H,1)
        c2v{i} = find(H(i,:));
        end
end

function v2c = get_v2c(H)
        for j = 1:size(H,2)
        v2c{j} = find(H(:,j))';
        end
end

function ber = bitErrorRate(orig, dec)
        ber = sum(xor(orig, dec)) / numel(orig);
end

function [B, H, z] = nrldpc_Hmatrix(BG)
        load(sprintf('%s.txt', BG), BG);
        B = eval(BG);
        [mb, nb] = size(B);
        z      = max(B(:)) + 1;
        Iz     = eye(z);
        I0     = zeros(z);
```

```
    H       = zeros(mb*z, nb*z);


    for ii = 1:mb

    rows = (ii-1)*z + (1:z);

    for jj = 1:nb

    cols = (jj-1)*z + (1:z);

    if B(ii,jj) == -1

    H(rows, cols) = I0;

    else

    H(rows, cols) = circshift(Iz, -B(ii,jj));

    end

    end

    end

end


function cword = nrldpc_encode(B, z, msg)

    [m, n] = size(B);

    cword  = zeros(1, n*z);

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

    temp   = zeros(1, z);


    for i = 1:4

    for j = 1:(n-m)

    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

    p = B(3, n-m+1);
```

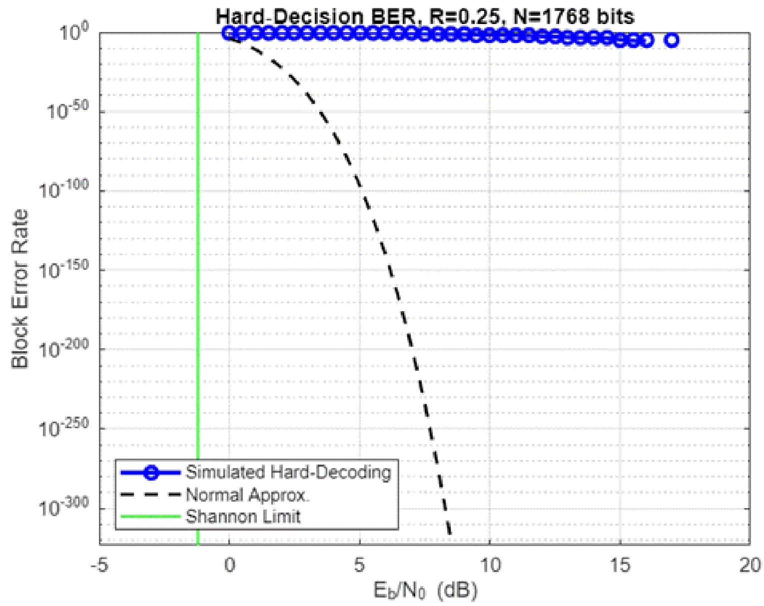
```
        else
        p = B(2, n-m+1);
        end
        cword((n-m)*z+1:(n-m+1)*z) = mul_sh(temp, z-p);

        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

        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
```

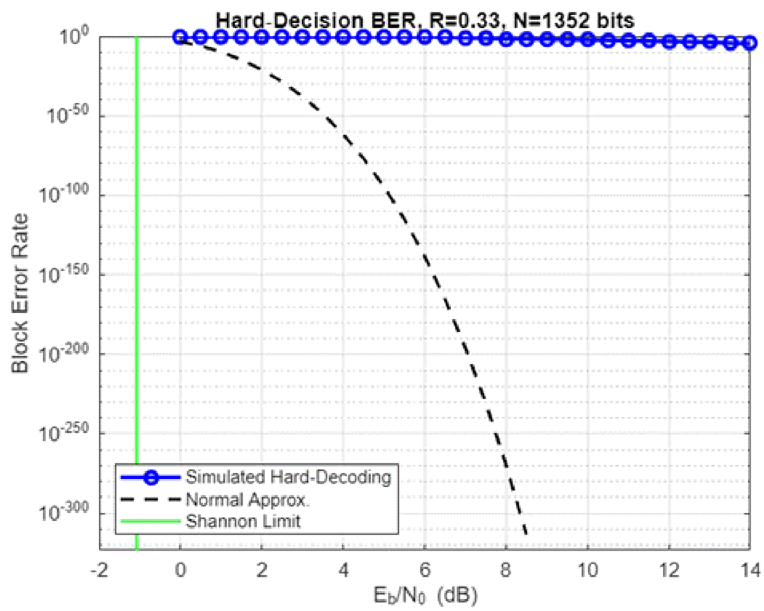
3) Shannon Limit and Normal Approximation for Hard Decoding.

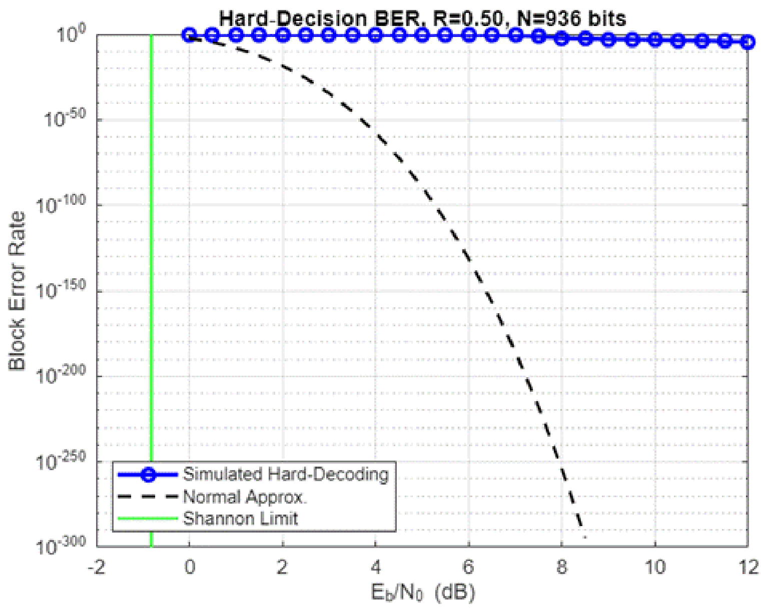
```
clear; clc;
baseGraph       = 'NR_2_6_52';
codeRates       = [1/4, 1/3, 1/2, 3/5];
EbN0dBList      = 0:0.5:30;
numTrials       = 100;
maxIterations = 20;
[B, Hfull, z] = nrldpc_Hmatrix(baseGraph);
[mb, nb]        = size(B);

Rcount = length(codeRates);
Ecount = numel(EbN0dBList);
decErr = zeros(Rcount, Ecount);
probSucc = zeros(Rcount, Ecount);
blockLen = zeros(1, Rcount);

for rIdx = 1:Rcount
    rate = codeRates(rIdx);

    [H, Nbits] = H_matrix(Hfull, z, mb, nb, rate);
    blockLen(rIdx) = Nbits;

    c2v_map = get_c2v(H);
    v2c_map = get_v2c(H);

    for eIdx = 1:Ecount
    EbN0dB = EbN0dBList(eIdx);
```

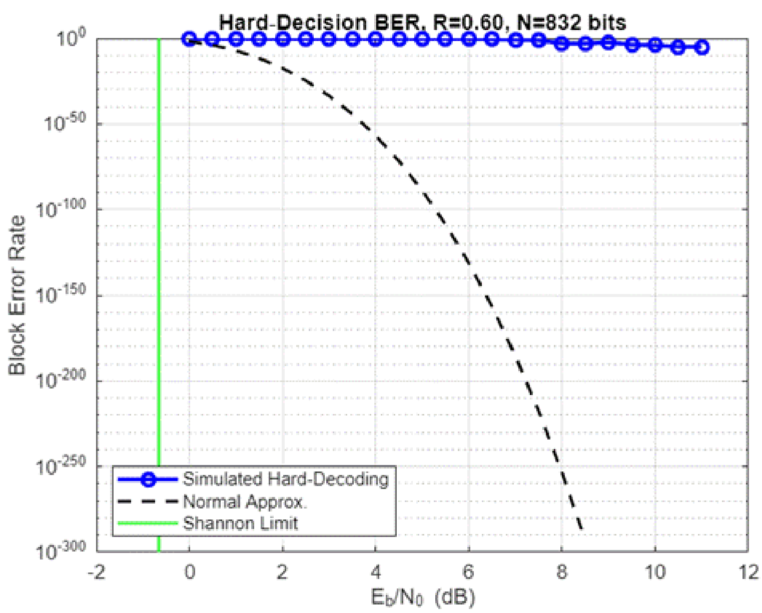
```
        for tr = 1:numTrials

        msg = randi([0,1], (nb-mb)*z, 1);
        cw  = encodeLDPC(B, z, msg, Nbits);

        tx = 1 - 2*cw;
        rx = addAWGN(tx, rate, EbN0dB);

        [decBits, success] = hardDecoding(H, rx, maxIterations, c2v_map, v2c_map);

        decErr(rIdx,eIdx)   = decErr(rIdx,eIdx)   + bitErrorRate(cw, decBits);
        probSucc(rIdx,eIdx) = probSucc(rIdx,eIdx) + success;
        end

        decErr(rIdx,eIdx)   = decErr(rIdx,eIdx)   / numTrials;
        probSucc(rIdx,eIdx) = probSucc(rIdx,eIdx) / numTrials;
        end
end

for i = 1:Rcount
        rate = codeRates(i);
        N    = blockLen(i);

        gammaLin  = 10.^(EbN0dBList/10);
        P         = rate .* gammaLin;
        C         = log2(1 + P);
        V         = (log2(exp(1))).^2 .* P .* (P+2) ./ (P+1).^2;
        arg       = sqrt(N) .* ( C - rate + log2(N)./(2*N) ) ./ sqrt(V);
```

```
    PN_e      = 0.5 * erfc(arg./sqrt(2));


    EbN0_sh = 10*log10((2^rate - 1)/rate);


    figure;
    semilogy(EbN0dBList, decErr(i,:),'b-o','LineWidth',2); hold on;
    semilogy(EbN0dBList, PN_e, 'k--','LineWidth',1.5);
    xline(EbN0_sh, 'g-','LineWidth',1.5);
    grid on;
    xlabel('E_b/N_0 (dB)');
    ylabel('Block Error Rate');
    title (sprintf('Hard-Decision BER, R=%.2f, N=%d bits', rate, N));
    legend('Simulated Hard-Decoding','Normal Approx.','Shannon Limit','Location','southwest');
end
```


Hard-Decision BER, R=0.25, N=1768 bits
Block Error Rate
10^{0}
10^{-50}
10^{-100}
10^{-150}
10^{-200}
10^{-250}
10^{-300}
Simulated Hard-Decoding
Normal Approx.
Shannon Limit
-5
0
5
10
15
20
E_b/N_0 (dB)


Hard-Decision BER, R=0.33, N=1352 bits
Block Error Rate
10^{0}
10^{-50}
10^{-100}
10^{-150}
10^{-200}
10^{-250}
10^{-300}
Simulated Hard-Decoding
Normal Approx.
Shannon Limit
-2
0
2
4
6
8
10
12
14
E_b/N_0 (dB)






```
function [decoded, success] = hardDecoding(H, rx, maxIter, c2v_map, v2c_map)

        nVars = size(H,2);

        decoded = rx < 0;

        prev = decoded;

        success = false;



        v2c_msgs = zeros(size(H));
```

```
for vn = 1:nVars
for cn = v2c_map{vn}
v2c_msgs(cn,vn) = decoded(vn);
end
end

for iter = 1:maxIter
c2v_msgs = zeros(size(H));
for cn = 1:numel(c2v_map)
vs = c2v_map{cn};
xor_all = 0;
for v = vs, xor_all = xor(xor_all, v2c_msgs(cn,v)); end
for v = vs
c2v_msgs(cn,v) = xor(xor_all, v2c_msgs(cn,v));
end
end

new_est = zeros(nVars,1);
for vn = 1:nVars
cs = v2c_map{vn};
dv = numel(cs);
totalOnes = decoded(vn) + sum(c2v_msgs(cs,vn));
for cn = cs
excl = totalOnes - c2v_msgs(cn,vn);
v2c_msgs(cn,vn) = excl > dv/2;
end
new_est(vn) = totalOnes > floor(dv/2);
end
```

```
        if all(mod(H*new_est,2)==0)
        decoded = new_est;
        success = true;
        return
        end
        if isequal(new_est, prev)
        decoded = new_est;
        return
        end
        prev = new_est;
        decoded = new_est;
        end
end

function rx = addAWGN(tx, rate, EbN0dB)
        gamma = 10^(EbN0dB/10);
        sigma = sqrt(1/(2*rate*gamma));
        rx = tx + sigma*randn(size(tx));
end

function [H, Nbits] = H_matrix(Hfull, z, mb, nb, rate)
        parityCols = (nb-mb) - 2;
        nbRM = min( ceil(parityCols/rate)+2, nb );
        Nbits = nbRM * z;
        Htrunc = Hfull(:,1:Nbits);
        chkRows = mb*z - nb*z + Nbits;
        H = Htrunc(1:chkRows,:);
end
```

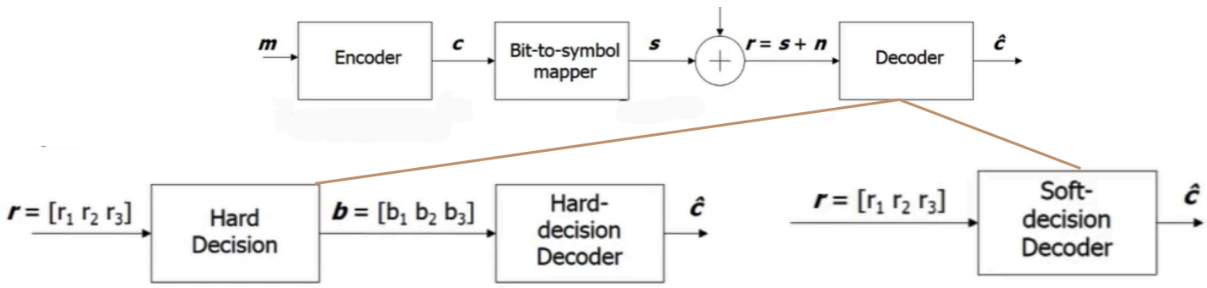
```
function c = encodeLDPC(B, z, msg, Nbits)
    cw_full = nrldpc_encode(B, z, msg');
    c = cw_full(1:Nbits)';
end

function ber = bitErrorRate(orig, dec)
    ber = sum(xor(orig,dec)) / numel(orig);
end

function c2v = get_c2v(H)
    for i = 1:size(H,1)
    c2v{i} = find(H(i,:));
    end
end

function v2c = get_v2c(H)
    for j = 1:size(H,2)
    v2c{j} = find(H(:,j))';
    end
end

function [B, H, z] = nrldpc_Hmatrix(BG)
    load(sprintf('%s.txt',BG),BG);
    B = eval(BG);
    [mb,nb] = size(B);
    z = max(B(:)) + 1;
    Iz = eye(z); I0 = zeros(z);
    H = zeros(mb*z, nb*z);
    for ii=1:mb
```

```
        rows = (ii-1)*z + (1:z);
        for jj=1:nb
        cols = (jj-1)*z + (1:z);
        if B(ii,jj)==-1
        H(rows,cols)=I0;
        else
          H(rows,cols)=circshift(Iz,-B(ii,jj));
        end
        end
        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

function cw = nrldpc_encode(B, z, msg)
        [m,n] = size(B);
        cw = zeros(1,n*z);
        cw(1:(n-m)*z) = msg;
        temp = zeros(1,z);
        for i=1:4
        for j=1:(n-m)
        temp = mod(temp + mul_sh(msg((j-1)*z+1:j*z), B(i,j)),2);
        end
```

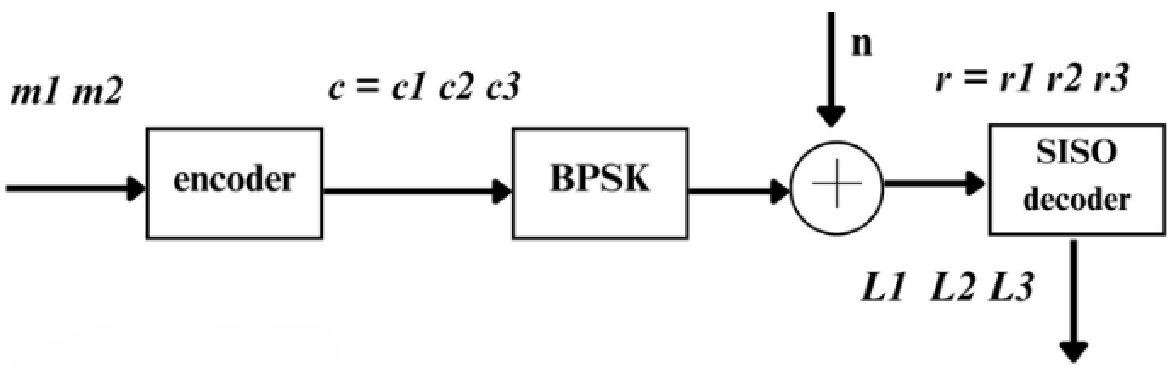
```
        end


        if B(2,n-m+1)==-1

                p=B(3,n-m+1);

   else

                p=B(2,n-m+1);

   end


        cw((n-m)*z+1:(n-m+1)*z) = mul_sh(temp, z-p);

        for i=1:3

        temp = zeros(1,z);

        for j=1:(n-m+i)

        temp = mod(temp + mul_sh(cw((j-1)*z+1:j*z), B(i,j)),2);

        end

        cw((n-m+i)*z+1:(n-m+i+1)*z) = temp;

        end

        for i=5:m

        temp = zeros(1,z);

        for j=1:(n-m+4)

        temp = mod(temp + mul_sh(cw((j-1)*z+1:j*z), B(i,j)),2);

        end

        cw((n-m+i-1)*z+1:(n-m+i)*z) = temp;

        end

end
```

Soft Decision Decoding for LDPC Codes

- Soft decision decoding is a technique used in error correction coding that uses the real-valued outputs from the channel (not just hard 0 or 1 decisions).
- Instead of making an early decision on each bit, the decoder processes probabilistic information—usually in the form of Log-Likelihood Ratios (LLRs).



SISO decoder for SPC Code



- Li = l1 + lext, 1
- LLR l1 = (2/(sigma)^2)*r1
- lext,1 = sign(l2)*sign(l3)*min(|l2|,|l3|))

Log likelyhood ratio

$$P(C_i = 1|r_i) = \frac{P(r_i|\ C_i = 1)P(C_i = 1)}{P(r_i)}$$

$$P(C_i = 0|r_i) = \frac{P(r_i|\ C_i = 0)P(C_i = 0)}{P(r_i)}$$

$$\lambda_i = \frac{P(C_i = 1|r_i)}{P(C_i = 0|r_i)} = \frac{P(r_i|\ C_i = 1)}{P(r_i|\ C_i = 0)}$$

$$\lambda_i = \frac{\left(\frac{1}{\sqrt{2\pi\sigma}}\right)\left(e^{-\frac{(r_i+1)^2}{2\sigma^2}}\right)}{\left(\frac{1}{\sqrt{2\pi\sigma}}\right)\left(e^{-\frac{(r_i-1)^2}{2\sigma^2}}\right)}$$

$$\lambda_i = e^{\frac{2r_i}{\sigma^2}}$$

$$\iota_i = \ln(\lambda_i) = \frac{2r_i}{\sigma^2}$$ intrinsic LLR

$where\ P(C_i = 1) = P(C_i = 0) = ½$

SISO decoder for SPC

Ex – (3, 2)

C1	C2	C3
0	0	0
0	1	1
1	0	1
1	1	0

$$c1 = c2 \oplus c3$$

Where Pi = (Ci=1)

$$P1 = P2(1 - P3) + P3(1 - P2) \quad \text{....... (1)}$$

$$(1 - P1) = P2P3 + (1 - P2)(1 - P3) \quad \text{....... (2)}$$

Now Subtract Equations (1-2)

$$P1 - (1 - P1) = P2(P3(1 - p3)) - (1 - P2)\,(P3(1 - p3))$$

$$P1 - (1 - P1) = (P2 - (1 - P2))\,(P3 - (1 - p3))$$

$$\frac{P1 - (1 - P1)}{P1 + (1 - P1)} = \frac{(P2 - (1 - P2))}{P2 + (1 - P2)}\,\frac{(P3 - (1 - p3))}{P3 + (1 - P3)}$$

$$\frac{1 - \frac{(1 - P1)}{P1}}{1 + \frac{(1 - P1)}{P1}} = \frac{1 - \frac{(1 - P2)}{P2}}{1 + \frac{(1 - P2)}{P2}}\,\frac{1 - \frac{(1 - P3)}{P3}}{1 + \frac{(1 - P3)}{P3}}$$

$$\frac{1 - e^{-l_{ext,1}}}{1 + e^{-l_{ext,1}}} = \frac{1 - e^{-l_2}}{1 + e^{-l_2}}\,\frac{1 - e^{-l_3}}{1 + e^{-l_3}}$$

$$\tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

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

$l_{ext,1}$ can be written in two-part Magnitude and sign

$$sgn(l_{ext,1}) = sgn(l_2)\,sgn(l_3)$$

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

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

Now $f(x)$ is,

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

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

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

Because of the characteristics of the f function

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

$$|l_{ext,1}| = f(f(min(|l_2|, |l_3|)))$$

Now, f is inverse of its own

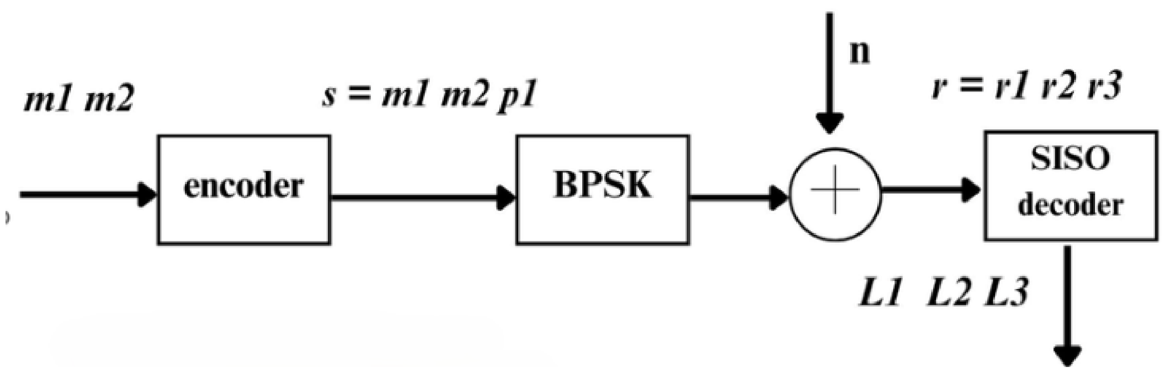
$$|l_{ext,1}| = min(|l_2|, |l_3|)$$
$$l_{ext,1} = sgn(l_{ext,1}) * |l_{ext,1}|$$

Similarly for n

$$|l_{ext,1}| = min(|l_2|, |l_3|, \ldots\ldots, |l_n|)$$
$$sgn(l_{ext,1}) = sgn(l_2)\, sgn(l_3) \ldots\ldots sgn(l_n)$$
$$l_{ext,1} = sgn(l_{ext,1}) * |l_{ext,1}|$$

This is the return value after the mean operation.

SISO decoder for repetition Code



- Li = belief that bit is 0
- L1 = computed using r1,r2,r3
- L1 = r1+r2+r3
- for L1, r1 = intrinsic, r2+r3 = extrinsic

SISO decoder for Repetition Code

Calculation for L1

$$l_i = \frac{P(C_i = 1 \mid r_1, r_2, \ldots\ldots, r_n)}{P(C_i = 0 \mid r_1, r_2, \ldots\ldots, r_n)} = \frac{P(r_1, r_2, \ldots\ldots, r_n \mid C_i = 1)}{P(r_1, r_2, \ldots\ldots, r_n \mid C_i = 0)}$$

$$l_i = \frac{P(r_1 \mid C_1 = 1)\, P(r_2 \mid C_2 = 1) \ldots P(r_n \mid C_n = 1)}{P(r_1 \mid C_1 = 0)\, P(r_2 \mid C_2 = 0) \ldots P(r_n \mid C_n = 0)}$$ (Because all r 1 ... independent from each other)

$$= \frac{\left(e^{-\frac{(r_1+1)^2}{2\sigma^2}}\right)}{\left(e^{-\frac{(r_1-1)^2}{2\sigma^2}}\right)} \frac{\left(e^{-\frac{(r_2+1)^2}{2\sigma^2}}\right)}{\left(e^{-\frac{(r_2-1)^2}{2\sigma^2}}\right)} \frac{\left(e^{-\frac{(r_3+1)^2}{2\sigma^2}}\right)}{\left(e^{-\frac{(r_3-1)^2}{2\sigma^2}}\right)} \qquad \left(\because \quad \lambda = \frac{\left(\frac{1}{\sqrt{2\pi}\sigma}\right)\left(e^{-\frac{(r_i-1)^2}{2\sigma^2}}\right)}{\left(\frac{1}{\sqrt{2\pi}\sigma}\right)\left(e^{-\frac{(r_i+1)^2}{2\sigma^2}}\right)} \right)$$

$$= e^{\frac{2r_1}{\sigma^2}}\, e^{\frac{2r_2}{\sigma^2}}\, e^{\frac{2r_3}{\sigma^2}}$$

$$L_i = r_1 + r_2 + \ldots\ldots + r_n \quad \text{ignore } 2/\sigma^2 \; factor$$

Message Passing in Tanner Graph Using Log-Likelihood Ratios (LLRs)

Initialization (Intrinsic Information from Variable Nodes)

- Initially, each VN computes its intrinsic LLR based on the received channel information (e.g., from a noisy BPSK signal).
- Each VN sends this intrinsic LLR as a message to all connected Check Nodes (CNs)**.**

Check Node to Variable Node Message Passing (SISO SPC)

- Upon receiving LLR messages from all connected VNs, the CN computes extrinsic LLRs to send back to each VN.
- This computation at each CN can be viewed as a Soft-In Soft-Out (SISO) decoding of a Single Parity Check (SPC) code, using the incoming LLRs.
- The CN sends the computed extrinsic LLR to each VN.

Variable Node Update (SISO Repetition)

- After receiving extrinsic LLRs from all connected CNs, each VN combines them with its intrinsic LLR.
- This combination represents a Soft-In Soft-Out decoding of a repetition code, since each VN is connected to multiple CNs and thus receives multiple estimates of the same bit.
- These updated messages are sent back to each connected CN excluding the one from which the message was received (i.e., again, extrinsic information).

Iterative Process

- Steps 2 and 3 are repeated iteratively.
- The goal is for the LLRs at each VN to converge to a stable value representing a strong belief in the bit's value.

Soft Decision Decoding - MINSUM Algorithm

Storage Matrix L:

- L is a sparse matrix with the same dimensions as the parity check matrix H.
- An entry in Lis zero if the corresponding entry in H is zero.
- An entry in L is non-zero only if the corresponding entry in H is 1.
- Each non-zero entry in a row of L is initialized with the received value corresponding to that variable node (bit) from the received codeword.

$$\boldsymbol{r} = [\, r_1 \quad r_2 \quad r_3 \quad r_4 \quad r_5 \quad r_6 \quad r_7 \,]$$

$$L = \begin{bmatrix} r_1 & r_2 & r_3 & 0 & r_5 & 0 & 0 \\ 0 & r_2 & r_3 & r_4 & 0 & r_6 & 0 \\ r_1 & r_2 & 0 & r_4 & 0 & 0 & r_7 \\ r_1 & 0 & r_3 & 0 & r_5 & r_6 & r_7 \end{bmatrix}$$

- In soft decision decoding, the received signal is used without hard demodulation, preserving the real-valued information.
- Each Variable Node (VN) sends a belief (soft information) to its connected Check Nodes (CNs).
- Check Nodes also send updated beliefs back to the VNs in a two-way iterative process.
- Message transfer:
  - VN to CN: Uses sum rule
  - CN to VN: Uses the Min-Sum algorithm.

MINSUM - Algorithm

- (Min-Sum SPC SISO)
- For each row:

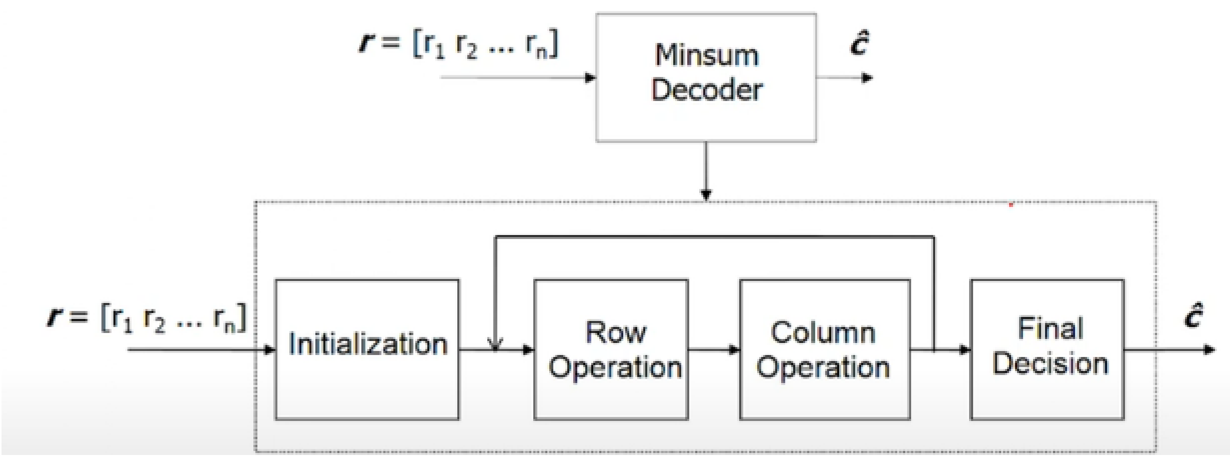
Magnitude

- Min1 = Minimum absolute value of all nonzero entries in the row
- Min2 = Next higher absolute value
- Set magnitude of all values (except minimum) = Min1
- Set magnitude of minimum value = Min2

Sign

- Parity = Product of signs of entries in the row
- New sign of an entry = (Old Sign) × (Parity)

- (Min-Sum repetition SISO)
- For each column:
- sum_j = r_j + sum of all entries in column j
- new entry = sum - (old entry)



4) Soft Decoding : NR_2_6_52

```
clear all; close all; clc;


%% Simulation Parameters

baseGraph5GNR = 'NR_2_6_52';

codeRates = [1/4, 1/3, 1/2, 3/5]; % Code rates to simulate

EbN0dB_vec = -1.50:0.50:5.00;

max_iterations = 20; % Maximum decoding iterations

Nsim = 100;

K = 22*52; % Information bits

N = 68*52; % Codeword length


%% Initialize Results Storage

results = struct();

for cr = 1:length(codeRates)

        results(cr).rate = codeRates(cr);

        results(cr).BER = zeros(size(EbN0dB_vec));

        results(cr).FER = zeros(size(EbN0dB_vec));

        results(cr).iteration_success = zeros(max_iterations, length(EbN0dB_vec));

        results(cr).Pc = zeros(size(EbN0dB_vec)); % Probability of successful decoding

end


%% Calculate Theoretical Benchmarks (Shannon Limit and Normal Approximation)

shannon_limit = 10*log10((2.^codeRates-1)./codeRates); % In dB


% Pre-calculate Normal Approximation for each code rate

for cr = 1:length(codeRates)
```

```
    c_r = codeRates(cr);
    P_NA = zeros(size(EbN0dB_vec));
    for snr_idx = 1:length(EbN0dB_vec)
    EbN0dB = EbN0dB_vec(snr_idx);
    EbN0 = 10^(EbN0dB/10);
    P = c_r * EbN0;
    C = log2(1 + P);
    V = (log2(exp(1)))^2 * (P*(P + 2))/(2*(P + 1)^2);
    argument = sqrt(N/V) * (C - c_r + log2(N)/(2*N));
    P_NA(snr_idx) = qfunc(argument);
    end
    results(cr).P_NA = P_NA;
end

%% Main Simulation Loop
for cr_idx = 1:length(codeRates)
    c_r = codeRates(cr_idx);
    fprintf('\nSimulating code rate %.2f (%d/%d)\n', c_r, cr_idx, length(codeRates));

    % Generate parity check matrix
    [B, Hfull, z] = nrldpc_Hmatrix(baseGraph5GNR);
    [mb, nb] = size(B);
    kb = nb - mb;
    kNumInfoBits = kb * z;

    % Rate matching
    k_pc = kb-2;
    nbRM = ceil(k_pc/c_r)+2;
    nBlockLength = nbRM * z;
```

```
H = Hfull(:,1:nBlockLength);
nChecksNotPunctured = mb*z - nb*z + nBlockLength;
H = H(1:nChecksNotPunctured,:);

% Build Tanner graph
[VN_to_CN_map, CN_to_VN_map] = build_tanner_graph(H);

for snr_idx = 1:length(EbN0dB_vec)
EbN0dB = EbN0dB_vec(snr_idx);
EbN0 = 10^(EbN0dB/10);
EsN0 = c_r * EbN0;
noise_var = 1/(2*EsN0);

bit_errors = 0;
frame_errors = 0;
iteration_success = zeros(1, max_iterations);
success_count = 0; % For Pc calculation

for sim = 1:Nsim
% Generate and encode message
msg_bits = randi([0 1], 1, kNumInfoBits);
cword = nrldpc_encode(B, z, msg_bits);
cword = cword(1:nBlockLength);

% BPSK modulation and AWGN channel
tx_signal = 1 - 2*cword;
noise = sqrt(noise_var) * randn(1, nBlockLength);
rx_signal = tx_signal + noise;
```

```
        % LLR calculation and decoding

        llr = (2/noise_var) * rx_signal;

        [decoded_bits, iter_hist, final_success] = ...

        ldpc_decode_c332(llr, H, VN_to_CN_map, CN_to_VN_map, max_iterations, msg_bits);


        % Update statistics

        iteration_success = iteration_success + iter_hist;

        success_count = success_count + final_success;


        % Calculate errors

        bit_errors = bit_errors + sum(decoded_bits(1:kNumInfoBits) ~= msg_bits);

        frame_errors = frame_errors + (sum(decoded_bits(1:kNumInfoBits) ~= msg_bits) > 0);

        end


        % Store results

        results(cr_idx).BER(snr_idx) = bit_errors / (Nsim * kNumInfoBits);

        results(cr_idx).FER(snr_idx) = frame_errors / Nsim;

    results(cr_idx).iteration_success(:,snr_idx) = iteration_success' / Nsim;

        results(cr_idx).Pc(snr_idx) = success_count / Nsim; % C.3.2 Pc(imax)


        fprintf('  SNR %.1f dB: FER=%.2e, Pc=%.2f\n', EbN0dB, results(cr_idx).FER(snr_idx),
results(cr_idx).Pc(snr_idx));

        end


  %% Generate Individual Plots for Each Code Rate


        % Plot 1: Decoding Error Probability vs Eb/No with theoretical curves

        figure(cr_idx*10 + 1);

        set(gcf, 'Position', [100, 100, 800, 600]);
```

```
% Plot simulation results

semilogy(EbN0dB_vec, results(cr_idx).FER, 'b-o', ...

'LineWidth', 2, 'MarkerFaceColor', 'b', 'MarkerSize', 8, ...

'DisplayName', 'LDPC Simulation');

hold on;


% Plot Normal Approximation

semilogy(EbN0dB_vec, results(cr_idx).P_NA, 'r--', ...

'LineWidth', 2, 'DisplayName', 'Normal Approximation');


% Plot Shannon limit

shannon_line = ones(size(EbN0dB_vec)) * 0.5; % Placeholder for vertical line

semilogy([shannon_limit(cr_idx), shannon_limit(cr_idx)], [1e-4 1], 'k:', ...

'LineWidth', 2, 'DisplayName', 'Shannon Limit');


hold off;

grid on;

xlabel('Eb/No (dB)', 'FontSize', 12);

ylabel('Block Error Rate (BLER)', 'FontSize', 12);

title(sprintf('Rate %.2f LDPC Code (K=%d, N=%d)', c_r, K, N), 'FontSize', 14);

legend('Location', 'southwest', 'FontSize', 10);

ylim([1e-4 1]);

xlim([min(EbN0dB_vec) max(EbN0dB_vec)]);


% Plot 2: Success Rate vs Iteration Number (unchanged)

figure(cr_idx*10 + 2);

set(gcf, 'Position', [100, 100, 800, 600]);

hold on;

colors = jet(length(EbN0dB_vec));
```

```
        for snr_idx = 1:length(EbN0dB_vec)
        plot(1:max_iterations, results(cr_idx).iteration_success(:,snr_idx), ...
        'Color', colors(snr_idx,:), ...
        'LineWidth', 2, ...
        'DisplayName', sprintf('%.1f dB', EbN0dB_vec(snr_idx)));
        end
        hold off;
        grid on;
        xlabel('Iteration Number', 'FontSize', 12);
        ylabel('Success Rate', 'FontSize', 12);
        title(sprintf('Success Rate vs Iteration (Rate %.2f)', c_r), 'FontSize', 14);
        legend('Location', 'eastoutside', 'FontSize', 10);
        ylim([0 1]);
end

%% Generate Combined Performance Plot with Theoretical Benchmarks
figure(100);
set(gcf, 'Position', [100, 100, 900, 700]);
hold on;

colors = lines(length(codeRates));
markers = {'o', 's', 'd', '^'};

for cr = 1:length(codeRates)
        % Plot simulation results
        semilogy(EbN0dB_vec, results(cr).FER, ...
        'Color', colors(cr,:), ...
        'Marker', markers{cr}, ...
        'LineWidth', 2, ...
```

```
        'MarkerFaceColor', colors(cr,:), ...
        'DisplayName', sprintf('Rate %.2f LDPC', codeRates(cr)));

        % Plot Normal Approximation
        semilogy(EbN0dB_vec, results(cr).P_NA, '--', ...
        'Color', colors(cr,:), ...
        'LineWidth', 1.5, ...
        'DisplayName', sprintf('Rate %.2f NA', codeRates(cr)));
end

% Plot Shannon limits
for cr = 1:length(codeRates)
        plot([shannon_limit(cr), shannon_limit(cr)], [1e-4 1], ':', ...
        'Color', colors(cr,:), ...
        'LineWidth', 1.5, ...
        'DisplayName', sprintf('Rate %.2f Shannon', codeRates(cr)));
end

hold off;
grid on;
xlabel('Eb/No (dB)', 'FontSize', 14);
ylabel('Block Error Rate (BLER)', 'FontSize', 14);
title('LDPC Performance vs Theoretical Benchmarks', 'FontSize', 16);
legend('Location', 'southwest', 'FontSize', 10);
set(gca, 'FontSize', 12);
ylim([1e-4 1]);
xlim([min(EbN0dB_vec) max(EbN0dB_vec)]);

%% Helper Functions
```

```
function [decoded_bits, iteration_history, final_success] = ...
ldpc_decode_c332(llr, H, VN_to_CN_map, CN_to_VN_map, max_iter, original_msg)

[num_CNs, num_VNs] = size(H);
VN_msgs = zeros(num_CNs, num_VNs);
CN_msgs = zeros(num_CNs, num_VNs);
iteration_history = zeros(1, max_iter);
kNumInfoBits = length(original_msg);

% Initialize with channel LLRs
for vn = 1:num_VNs
cn_list = VN_to_CN_map{vn};
VN_msgs(cn_list, vn) = llr(vn);
end

final_success = 0;
for iter = 1:max_iter
% Check node updates (min-sum with scaling)
for cn = 1:num_CNs
vn_list = CN_to_VN_map{cn};
incoming = VN_msgs(cn, vn_list);
sign_prod = prod(sign(incoming));
abs_incoming = abs(incoming);

for i = 1:length(vn_list)
vn = vn_list(i);
min1 = min(abs_incoming([1:i-1, i+1:end]));
CN_msgs(cn, vn) = 0.8 * sign_prod * sign(incoming(i)) * min1;
end
```

```
        end

        % Variable node updates
        decoded_bits = zeros(1, num_VNs);
        for vn = 1:num_VNs
        cn_list = VN_to_CN_map{vn};
        total = llr(vn) + sum(CN_msgs(cn_list, vn));

        for cn = cn_list
        VN_msgs(cn, vn) = total - CN_msgs(cn, vn);
        end

        decoded_bits(vn) = (total < 0);
        end

        % Track success at each iteration
        current_success = isequal(decoded_bits(1:kNumInfoBits), original_msg);
        iteration_history(iter) = current_success;

        if current_success
        final_success = 1;
        iteration_history(iter+1:end) = 1; % Fill remaining iterations
        break;
        end
        end
end

%% Rest of the helper functions remain unchanged
function [VN_to_CN_map, CN_to_VN_map] = build_tanner_graph(H)
```

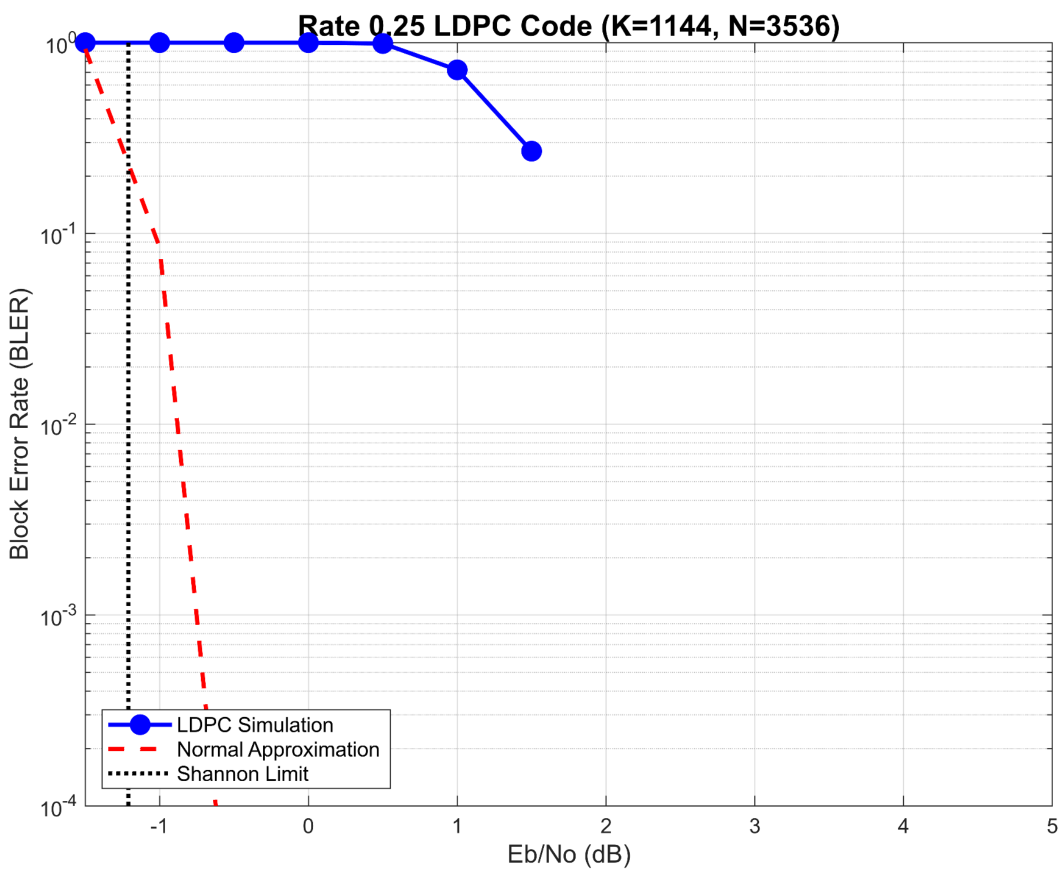
```
	[num_CNs, num_VNs] = size(H);
	VN_to_CN_map = cell(num_VNs, 1);
	CN_to_VN_map = cell(num_CNs, 1);

	for vn = 1:num_VNs
	VN_to_CN_map{vn} = find(H(:, vn))';
	end

	for cn = 1:num_CNs
	CN_to_VN_map{cn} = find(H(cn, :));
	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
```

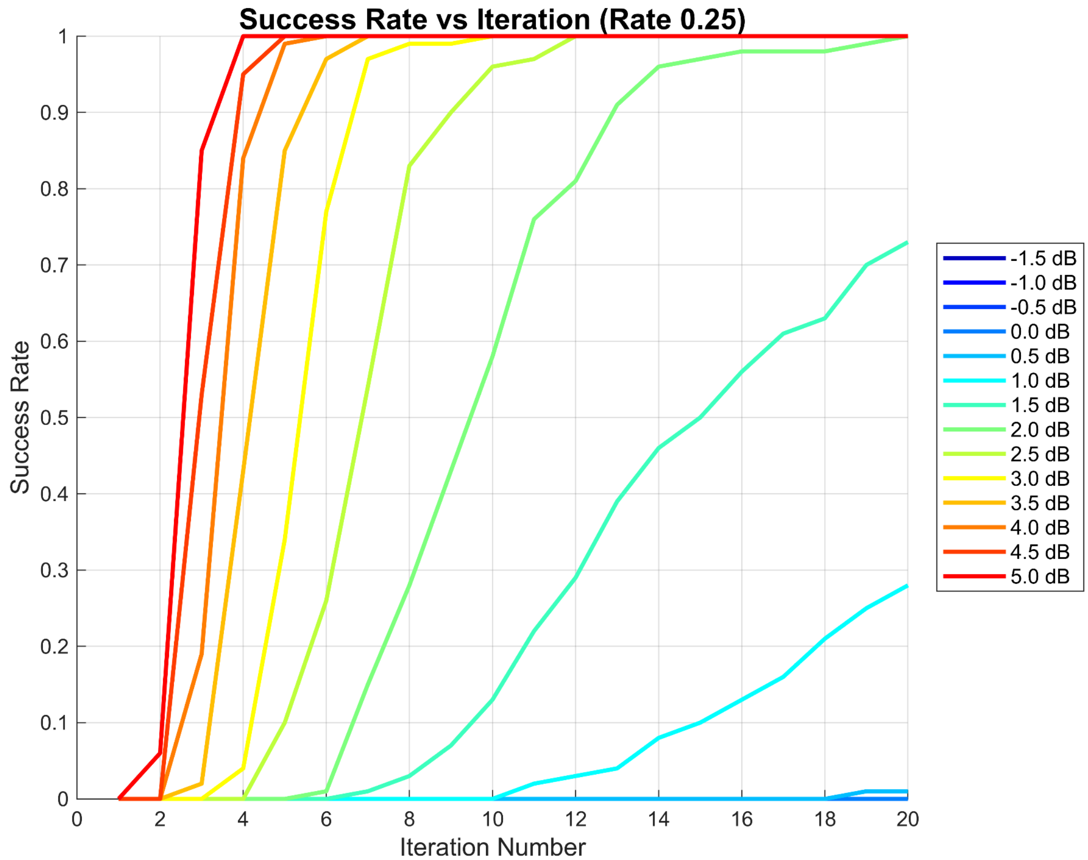
```
        end
        end
end

function cword = nrldpc_encode(B,z,msg)
        [m,n] = size(B);
        cword = zeros(1,n*z);
        cword(1:(n-m)*z) = msg;

        temp = zeros(1,z);
        for i = 1:4
        for j = 1:n-m
        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);

        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


        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
```

OUTPUT:

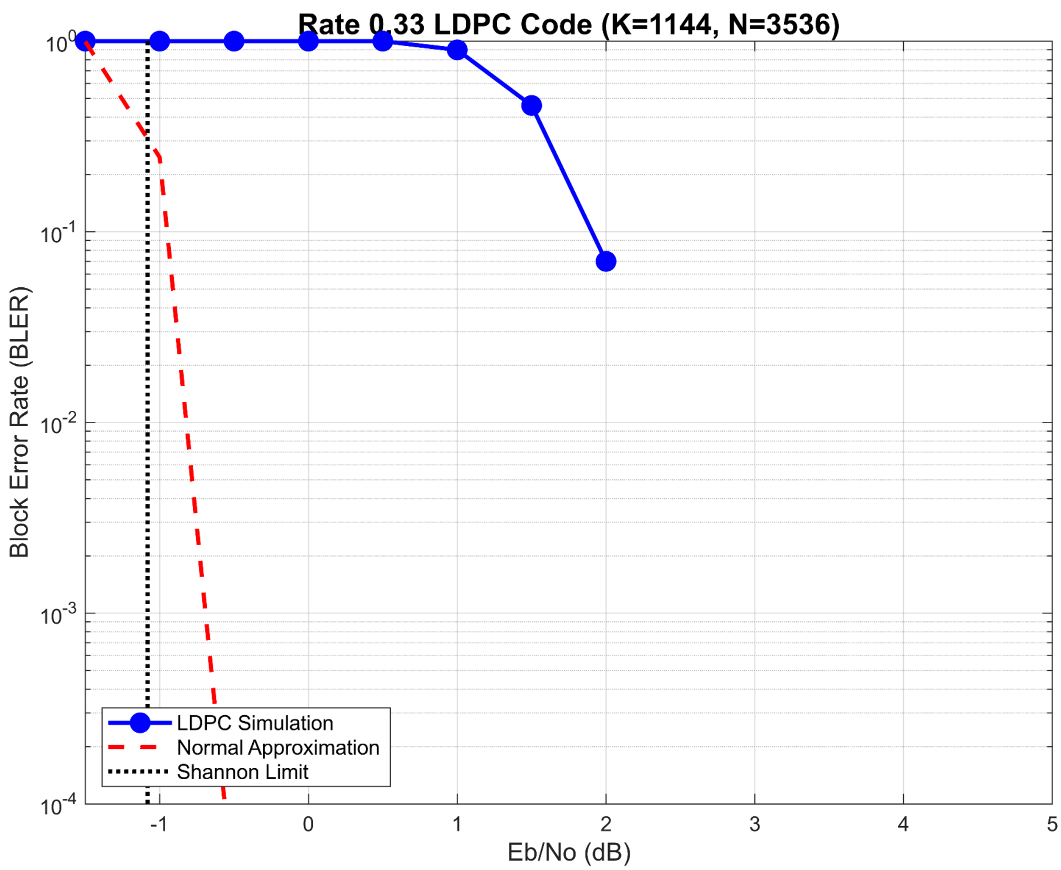
Simulating code rate 0.25 (1/4)
SNR -1.5 dB: FER=1.00e+00, Pc=0.00
SNR -1.0 dB: FER=1.00e+00, Pc=0.00
SNR -0.5 dB: FER=1.00e+00, Pc=0.00
SNR 0.0 dB: FER=1.00e+00, Pc=0.00
SNR 0.5 dB: FER=9.90e-01, Pc=0.01
SNR 1.0 dB: FER=7.20e-01, Pc=0.28
SNR 1.5 dB: FER=2.70e-01, Pc=0.73
SNR 2.0 dB: FER=0.00e+00, Pc=1.00
SNR 2.5 dB: FER=0.00e+00, Pc=1.00
SNR 3.0 dB: FER=0.00e+00, Pc=1.00
SNR 3.5 dB: FER=0.00e+00, Pc=1.00
SNR 4.0 dB: FER=0.00e+00, Pc=1.00
SNR 4.5 dB: FER=0.00e+00, Pc=1.00
SNR 5.0 dB: FER=0.00e+00, Pc=1.00

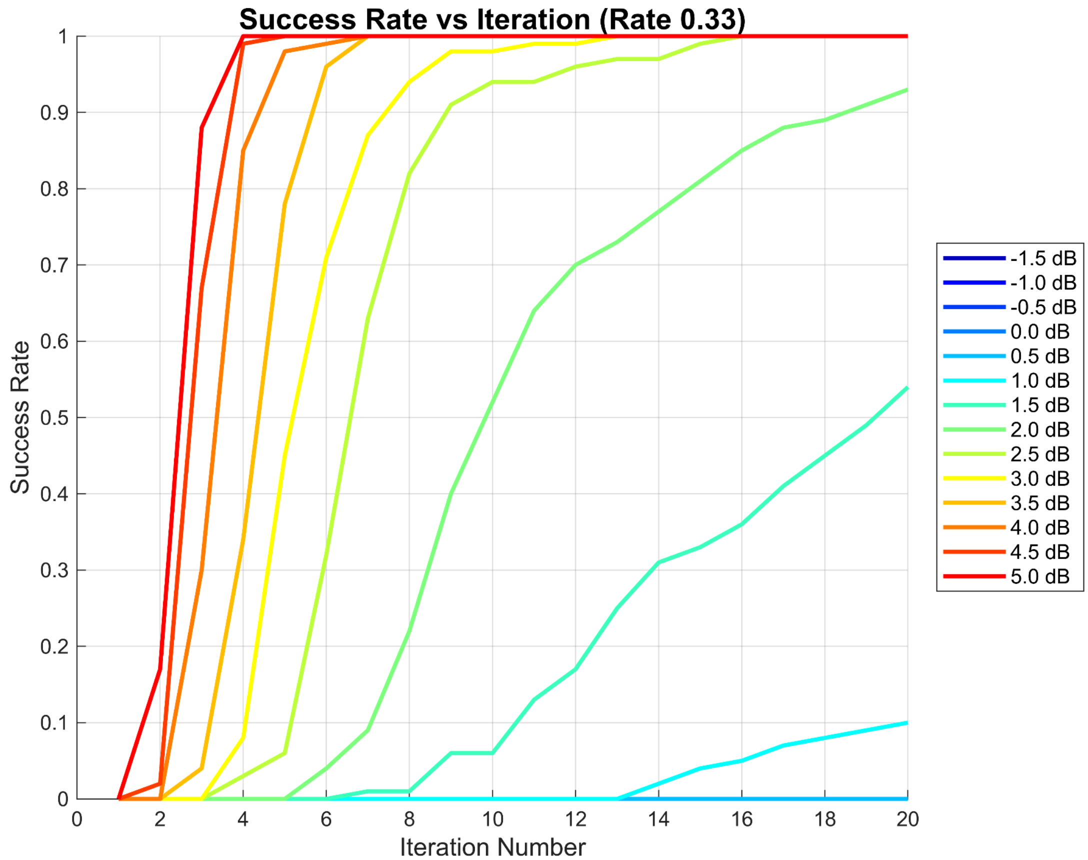




Simulating code rate 0.33 (2/4)
SNR -1.5 dB: FER=1.00e+00, Pc=0.00
SNR -1.0 dB: FER=1.00e+00, Pc=0.00

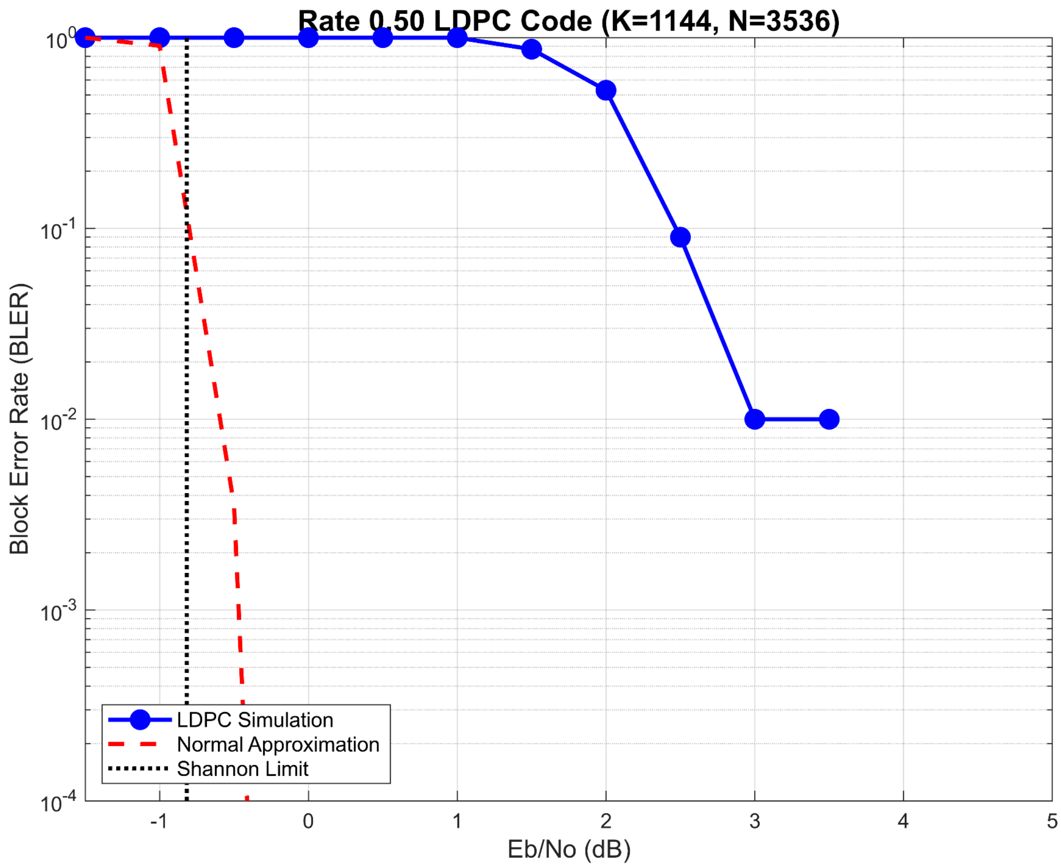
SNR -0.5 dB: FER=1.00e+00, Pc=0.00
SNR 0.0 dB: FER=1.00e+00, Pc=0.00
SNR 0.5 dB: FER=1.00e+00, Pc=0.00
SNR 1.0 dB: FER=9.00e-01, Pc=0.10
SNR 1.5 dB: FER=4.60e-01, Pc=0.54
SNR 2.0 dB: FER=7.00e-02, Pc=0.93
SNR 2.5 dB: FER=0.00e+00, Pc=1.00
SNR 3.0 dB: FER=0.00e+00, Pc=1.00
SNR 3.5 dB: FER=0.00e+00, Pc=1.00
SNR 4.0 dB: FER=0.00e+00, Pc=1.00
SNR 4.5 dB: FER=0.00e+00, Pc=1.00
SNR 5.0 dB: FER=0.00e+00, Pc=1.00

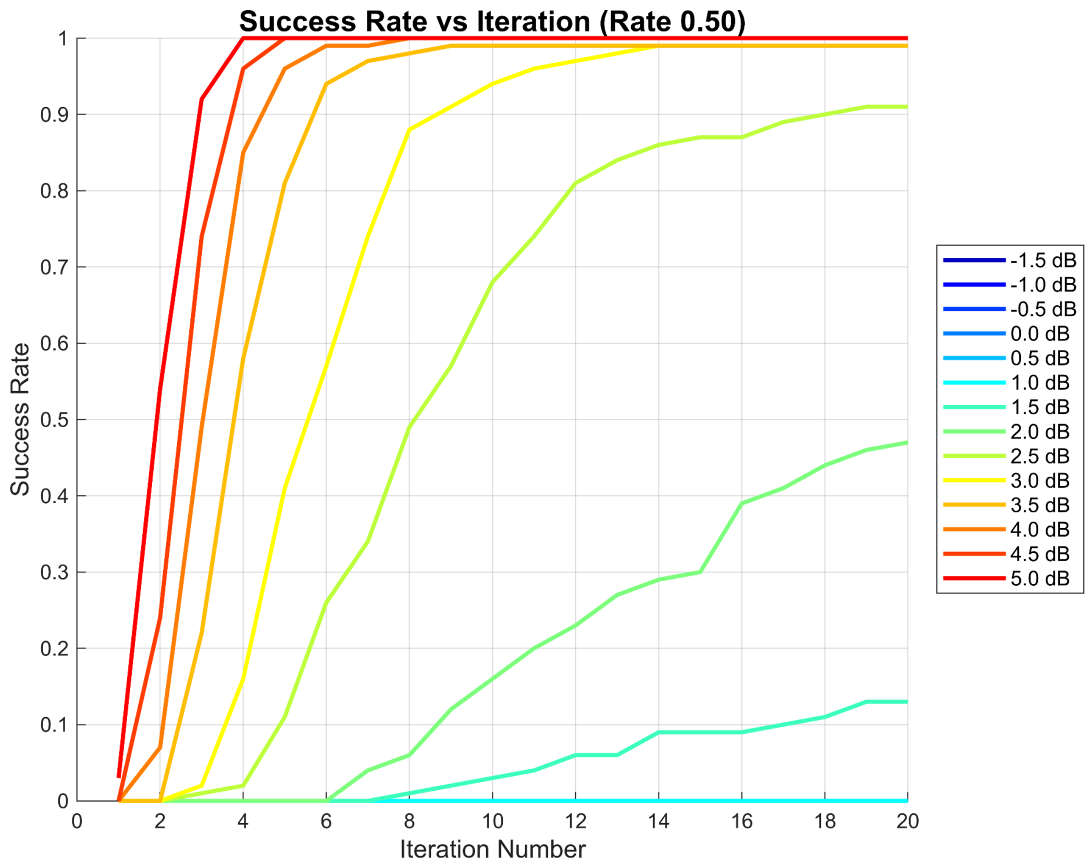




Simulating code rate 0.50 (3/4)
SNR -1.5 dB: FER=1.00e+00, Pc=0.00
SNR -1.0 dB: FER=1.00e+00, Pc=0.00
SNR -0.5 dB: FER=1.00e+00, Pc=0.00

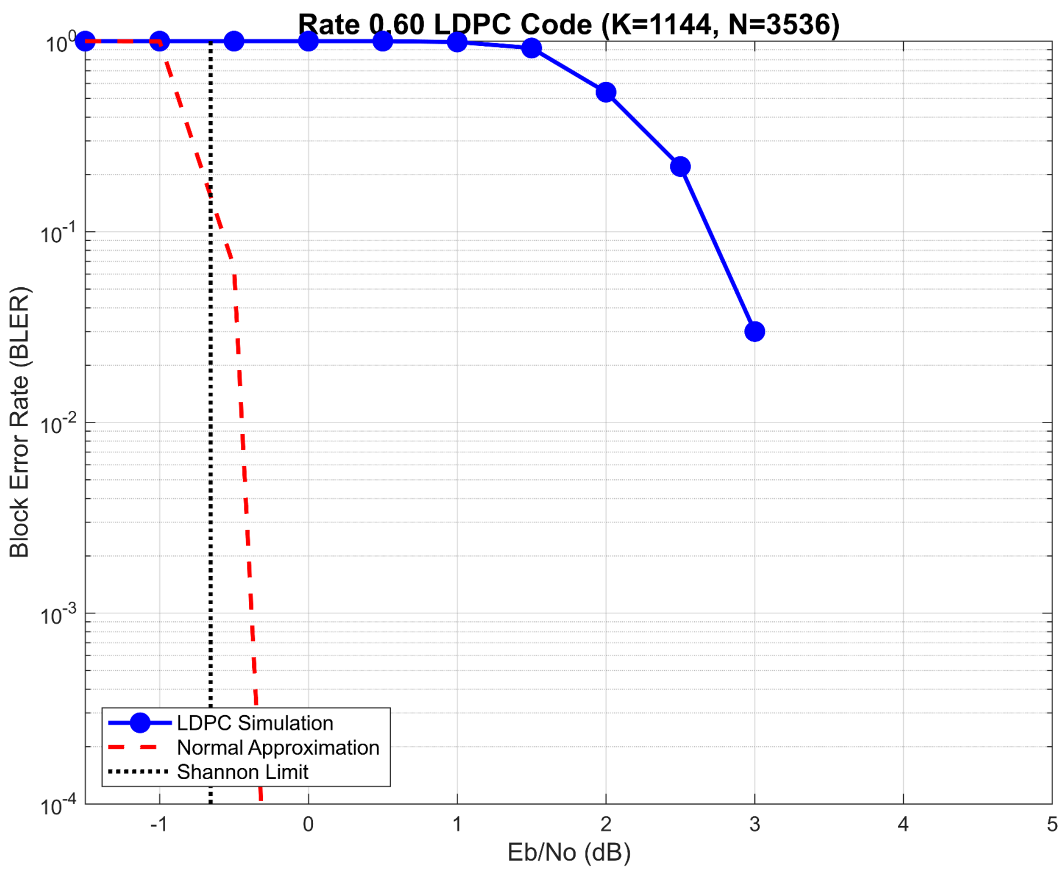
SNR 0.0 dB: FER=1.00e+00, Pc=0.00
SNR 0.5 dB: FER=1.00e+00, Pc=0.00
SNR 1.0 dB: FER=1.00e+00, Pc=0.00
SNR 1.5 dB: FER=8.70e-01, Pc=0.13
SNR 2.0 dB: FER=5.30e-01, Pc=0.47
SNR 2.5 dB: FER=9.00e-02, Pc=0.91
SNR 3.0 dB: FER=1.00e-02, Pc=0.99
SNR 3.5 dB: FER=1.00e-02, Pc=0.99
SNR 4.0 dB: FER=0.00e+00, Pc=1.00
SNR 4.5 dB: FER=0.00e+00, Pc=1.00
SNR 5.0 dB: FER=0.00e+00, Pc=1.00

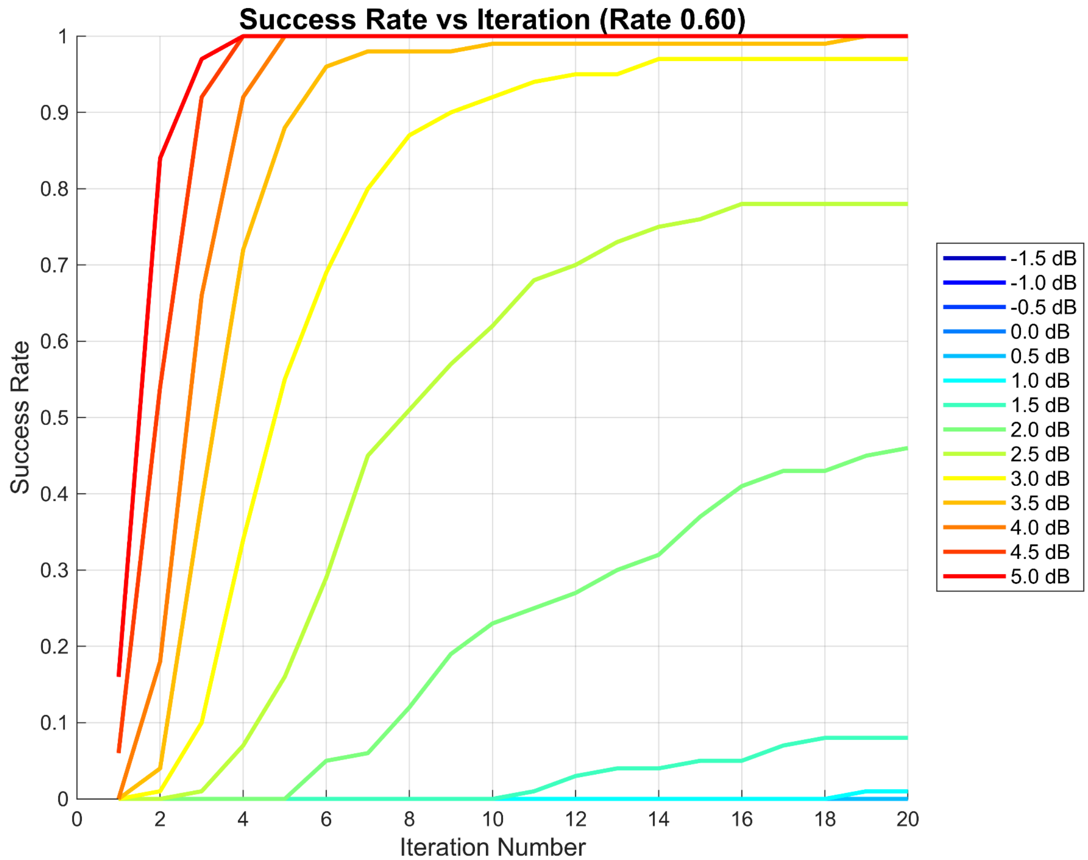




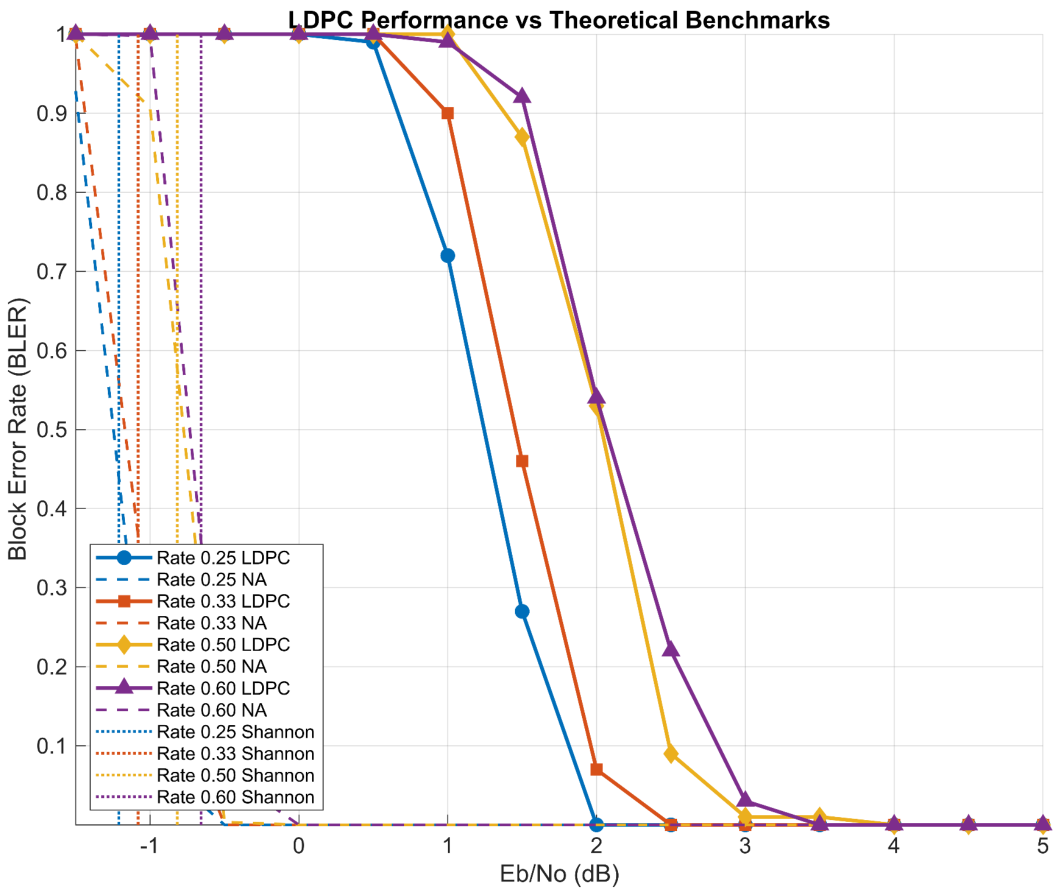
Simulating code rate 0.60 (4/4)
SNR -1.5 dB: FER=1.00e+00, Pc=0.00
SNR -1.0 dB: FER=1.00e+00, Pc=0.00
SNR -0.5 dB: FER=1.00e+00, Pc=0.00

SNR 0.0 dB: FER=1.00e+00, Pc=0.00
SNR 0.5 dB: FER=1.00e+00, Pc=0.00
SNR 1.0 dB: FER=9.90e-01, Pc=0.01
SNR 1.5 dB: FER=9.20e-01, Pc=0.08
SNR 2.0 dB: FER=5.40e-01, Pc=0.46
SNR 2.5 dB: FER=2.20e-01, Pc=0.78
SNR 3.0 dB: FER=3.00e-02, Pc=0.97
SNR 3.5 dB: FER=0.00e+00, Pc=1.00
SNR 4.0 dB: FER=0.00e+00, Pc=1.00
SNR 4.5 dB: FER=0.00e+00, Pc=1.00
SNR 5.0 dB: FER=0.00e+00, Pc=1.00





Success Rate vs Iteration (Rate 0.60)
Success Rate
Iteration Number
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
2
4
6
8
10
12
14
16
18
20
-1.5 dB
-1.0 dB
-0.5 dB
0.0 dB
0.5 dB
1.0 dB
1.5 dB
2.0 dB
2.5 dB
3.0 dB
3.5 dB
4.0 dB
4.5 dB
5.0 dB


LDPC Performance vs Theoretical Benchmarks
Block Error Rate (BLER)
Eb/No (dB)
Rate 0.25 LDPC
Rate 0.25 NA
Rate 0.33 LDPC
Rate 0.33 NA
Rate 0.50 LDPC
Rate 0.50 NA
Rate 0.60 LDPC
Rate 0.60 NA
Rate 0.25 Shannon
Rate 0.33 Shannon
Rate 0.50 Shannon
Rate 0.60 Shannon

5) Soft Decoding : NR_1_5_352

```
clear all; close all; clc;


%% Simulation Parameters (Optimized for NR_1_5_352)

baseGraph5GNR = 'NR_1_5_352';

codeRates = [1/3, 1/2, 3/5, 4/5]; % Supported code rates for BG1

EbN0dB_vec = -1.50:0.50:5.00;

max_iterations = 20;

Nsim = 10;

z = 352; % Expansion factor for NR_1_5_352

K = 10*z; % Information bits (22*52 was for NR_2_6_52)

N = 52*z; % Codeword length (68*52 was for NR_2_6_52)


%% Initialize Results Storage

results = struct();

for cr = 1:length(codeRates)

        results(cr).rate = codeRates(cr);

        results(cr).BER = zeros(size(EbN0dB_vec));

        results(cr).FER = zeros(size(EbN0dB_vec));

        results(cr).iteration_success = zeros(max_iterations, length(EbN0dB_vec));

        results(cr).Pc = zeros(size(EbN0dB_vec));

end


%% Theoretical Benchmarks

shannon_limit = 10*log10((2.^codeRates-1)./codeRates);


for cr = 1:length(codeRates)

        c_r = codeRates(cr);
```

```
        P_NA = zeros(size(EbN0dB_vec));
        for snr_idx = 1:length(EbN0dB_vec)
        EbN0dB = EbN0dB_vec(snr_idx);
        EbN0 = 10^(EbN0dB/10);
        P = c_r * EbN0;
        C = log2(1 + P);
        V = (log2(exp(1)))^2 * (P*(P + 2))/(2*(P + 1)^2);
        argument = sqrt(N/V) * (C - c_r + log2(N)/(2*N));
        P_NA(snr_idx) = qfunc(argument);
        end
        results(cr).P_NA = P_NA;
end

%% Main Simulation Loop
for cr_idx = 1:length(codeRates)
        c_r = codeRates(cr_idx);
        fprintf('\nSimulating code rate %.2f (%d/%d)\n', c_r, cr_idx, length(codeRates));

        % Generate parity check matrix
        [B, Hfull, z] = nrldpc_Hmatrix_352(baseGraph5GNR);
        [mb, nb] = size(B);
        kb = nb - mb;
        kNumInfoBits = kb * z;

        % Rate matching
        k_pc = kb-2;
        nbRM = ceil(k_pc/c_r)+2;
        nBlockLength = nbRM * z;
        H = Hfull(:,1:nBlockLength);
```

```
nChecksNotPunctured = mb*z - nb*z + nBlockLength;

H = sparse(H(1:nChecksNotPunctured,:)); % Use sparse matrix


% Build memory-efficient Tanner graph

[VN_to_CN_map, CN_to_VN_map] = build_tanner_graph_sparse(H);


for snr_idx = 1:length(EbN0dB_vec)

EbN0dB = EbN0dB_vec(snr_idx);

EbN0 = 10^(EbN0dB/10);

EsN0 = c_r * EbN0;

noise_var = 1/(2*EsN0);


bit_errors = 0;

frame_errors = 0;

iteration_success = zeros(1, max_iterations);

success_count = 0;


parfor sim = 1:Nsim % Parallel processing

% Generate and encode message

msg_bits = randi([0 1], 1, kNumInfoBits);

cword = nrldpc_encode_352(B, z, msg_bits);

cword = cword(1:nBlockLength);


% BPSK modulation and AWGN channel

tx_signal = 1 - 2*cword;

noise = sqrt(noise_var) * randn(1, nBlockLength);

rx_signal = tx_signal + noise;


% LLR calculation and decoding
```

```
        llr = (2/noise_var) * rx_signal;

        [decoded_bits, iter_hist, final_success] = ...

          ldpc_decode_memory_optimized(llr, H, VN_to_CN_map, CN_to_VN_map, max_iterations, msg_bits);



        % Update statistics

        iteration_success = iteration_success + iter_hist;

        success_count = success_count + final_success;

        bit_errors = bit_errors + sum(decoded_bits(1:kNumInfoBits) ~= msg_bits);

        frame_errors = frame_errors + (sum(decoded_bits(1:kNumInfoBits) ~= msg_bits) > 0);

        end



        % Store results

        results(cr_idx).BER(snr_idx) = bit_errors / (Nsim * kNumInfoBits);

        results(cr_idx).FER(snr_idx) = frame_errors / Nsim;

    results(cr_idx).iteration_success(:,snr_idx) = iteration_success' / Nsim;

        results(cr_idx).Pc(snr_idx) = success_count / Nsim;



        fprintf('  SNR %.1f dB: FER=%.2e, Pc=%.2f\n', EbN0dB, results(cr_idx).FER(snr_idx),
results(cr_idx).Pc(snr_idx));

        end



  %% Plotting

        figure(cr_idx*10 + 1);

        set(gcf, 'Position', [100, 100, 800, 600]);

        semilogy(EbN0dB_vec, results(cr_idx).FER, 'b-o', 'LineWidth', 2, 'MarkerFaceColor', 'b', 'DisplayName',
'LDPC Simulation');

        hold on;

        semilogy(EbN0dB_vec, results(cr_idx).P_NA, 'r--', 'LineWidth', 2, 'DisplayName', 'Normal Approximation');

        semilogy([shannon_limit(cr_idx), shannon_limit(cr_idx)], [1e-4 1], 'k:', 'LineWidth', 2, 'DisplayName',
'Shannon Limit');

        hold off;
```

```
        grid on;
        xlabel('Eb/No (dB)'); ylabel('Block Error Rate (BLER)');
        title(sprintf('Rate %.2f LDPC Code (K=%d, N=%d)', c_r, K, N));
        legend('Location', 'southwest');
        ylim([1e-4 1]); xlim([min(EbN0dB_vec) max(EbN0dB_vec)]);

        figure(cr_idx*10 + 2);
        set(gcf, 'Position', [100, 100, 800, 600]);
        hold on;
        colors = jet(length(EbN0dB_vec));
        for snr_idx = 1:length(EbN0dB_vec)
        plot(1:max_iterations, results(cr_idx).iteration_success(:,snr_idx), ...
        'Color', colors(snr_idx,:), 'LineWidth', 2, ...
        'DisplayName', sprintf('%.1f dB', EbN0dB_vec(snr_idx)));
        end
        hold off;
        grid on;
        xlabel('Iteration Number'); ylabel('Success Rate');
        title(sprintf('Success Rate vs Iteration (Rate %.2f)', c_r));
        legend('Location', 'eastoutside');
        ylim([0 1]);
end
%% Combined Performance Plot
figure(100);
set(gcf, 'Position', [100, 100, 900, 700]);
hold on;
colors = lines(length(codeRates));
markers = {'o', 's', 'd', '^'};
```

```matlab
for cr = 1:length(codeRates)

    semilogy(EbN0dB_vec, results(cr).FER, ...

    'Color', colors(cr,:), 'Marker', markers{cr}, ...

    'LineWidth', 2, 'MarkerFaceColor', colors(cr,:), ...

    'DisplayName', sprintf('Rate %.2f LDPC', codeRates(cr)));


    semilogy(EbN0dB_vec, results(cr).P_NA, '--', ...

    'Color', colors(cr,:), 'LineWidth', 1.5, ...

    'DisplayName', sprintf('Rate %.2f NA', codeRates(cr)));

end


for cr = 1:length(codeRates)

    plot([shannon_limit(cr), shannon_limit(cr)], [1e-4 1], ':', ...

    'Color', colors(cr,:), 'LineWidth', 1.5, ...

    'DisplayName', sprintf('Rate %.2f Shannon', codeRates(cr)));

end


hold off;

grid on;

xlabel('Eb/No (dB)'); ylabel('Block Error Rate (BLER)');

title('LDPC Performance vs Theoretical Benchmarks');

legend('Location', 'southwest');

set(gca, 'FontSize', 12);

ylim([1e-4 1]); xlim([min(EbN0dB_vec) max(EbN0dB_vec)]);


%% Memory-Optimized Functions for NR_1_5_352

function [B,H,z] = nrldpc_Hmatrix_352(BG)

    % Load the base graph file

    load('NR_1_5_352.txt', 'NR_1_5_352');
```

```
B = NR_1_5_352;
[mb,nb] = size(B);
z = 352;

% Create sparse matrix directly
[rows, cols, shifts] = find(B ~= -1);
num_entries = length(rows);

% Pre-calculate total number of non-zero entries
total_nnz = num_entries * z;
row_inds = zeros(total_nnz, 1);
col_inds = zeros(total_nnz, 1);

% Build indices for sparse matrix
current_pos = 1;
for idx = 1:num_entries
r = rows(idx);
c = cols(idx);
s = B(r,c);

% Calculate base positions
base_row = (r-1)*z;
base_col = (c-1)*z;

% Create shifted indices for this block
block_rows = (1:z) + base_row;
block_cols = mod((1:z) + s - 1, z) + 1 + base_col;

% Store indices
```

```
        end_pos = current_pos + z - 1;

        row_inds(current_pos:end_pos) = block_rows;

        col_inds(current_pos:end_pos) = block_cols;


        current_pos = end_pos + 1;

        end


        H = sparse(row_inds, col_inds, 1, mb*z, nb*z);

end


function [VN_to_CN_map, CN_to_VN_map] = build_tanner_graph_sparse(H)

        [num_CNs, num_VNs] = size(H);

        [rows, cols] = find(H);


        % Pre-allocate

        VN_to_CN_map = cell(num_VNs,1);

        CN_to_VN_map = cell(num_CNs,1);


        % Build VN connections

        for vn = 1:num_VNs

        VN_to_CN_map{vn} = rows(cols == vn)';

        end


        % Build CN connections

        for cn = 1:num_CNs

        CN_to_VN_map{cn} = cols(rows == cn)';

        end

end
```

```
function [decoded_bits, iteration_history, final_success] = ...
    ldpc_decode_memory_optimized(llr, H, VN_to_CN_map, CN_to_VN_map, max_iter, original_msg)

    num_VNs = length(VN_to_CN_map);
    num_CNs = length(CN_to_VN_map);
    kNumInfoBits = length(original_msg);

    % Use cell arrays for message storage
    VN_to_CN_msgs = cell(num_VNs,1);
    CN_to_VN_msgs = cell(num_CNs,1);

    % Initialize VN messages
    for vn = 1:num_VNs
    cn_list = VN_to_CN_map{vn};
    VN_to_CN_msgs{vn} = llr(vn) * ones(size(cn_list));
    end

    iteration_history = zeros(1, max_iter);
    final_success = 0;

    for iter = 1:max_iter
    % Check node updates
    for cn = 1:num_CNs
    vn_list = CN_to_VN_map{cn};
    incoming = zeros(size(vn_list));

    % Collect incoming messages
    for i = 1:length(vn_list)
    vn = vn_list(i);
```

```
cn_pos_in_vn = find(VN_to_CN_map{vn} == cn, 1);

incoming(i) = VN_to_CN_msgs{vn}(cn_pos_in_vn);

end


sign_prod = prod(sign(incoming));

abs_incoming = abs(incoming);


% Compute outgoing messages

if isempty(CN_to_VN_msgs{cn})

CN_to_VN_msgs{cn} = zeros(size(vn_list));

end


for i = 1:length(vn_list)

min1 = min(abs_incoming([1:i-1, i+1:end]));

CN_to_VN_msgs{cn}(i) = 0.8 * sign_prod * sign(incoming(i)) * min1;

end

end


% Variable node updates and hard decision

decoded_bits = zeros(1, num_VNs);

for vn = 1:num_VNs

cn_list = VN_to_CN_map{vn};

total = llr(vn);


% Sum all incoming messages

for i = 1:length(cn_list)

cn = cn_list(i);

vn_pos_in_cn = find(CN_to_VN_map{cn} == vn, 1);

total = total + CN_to_VN_msgs{cn}(vn_pos_in_cn);
```

```
        end

        % Update outgoing messages
        for i = 1:length(cn_list)
        cn = cn_list(i);
        vn_pos_in_cn = find(CN_to_VN_map{cn} == vn, 1);
        VN_to_CN_msgs{vn}(i) = total - CN_to_VN_msgs{cn}(vn_pos_in_cn);
        end

        decoded_bits(vn) = (total < 0);
        end

        % Check for success
        current_success = isequal(decoded_bits(1:kNumInfoBits), original_msg);
        iteration_history(iter) = current_success;

        if current_success
        final_success = 1;
        iteration_history(iter+1:end) = 1;
        break;
        end
        end
end

function cword = nrldpc_encode_352(B,z,msg)
        [m,n] = size(B);
        cword = zeros(1,n*z);
        cword(1:(n-m)*z) = msg;
```

```
temp = zeros(1,z);
for i = 1:4
for j = 1:n-m
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);

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

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
```

OUTPUT:

Simulating code rate 0.33 (1/4)

SNR -1.5 dB: FER=1.00e+00, Pc=0.00

SNR -1.0 dB: FER=1.00e+00, Pc=0.00

SNR -0.5 dB: FER=1.00e+00, Pc=0.00

SNR 0.0 dB: FER=1.00e+00, Pc=0.00

SNR 0.5 dB: FER=1.00e+00, Pc=0.00

SNR 1.0 dB: FER=1.00e+00, Pc=0.00

SNR 1.5 dB: FER=0.00e+00, Pc=1.00

SNR 2.0 dB: FER=0.00e+00, Pc=1.00

SNR 2.5 dB: FER=0.00e+00, Pc=1.00

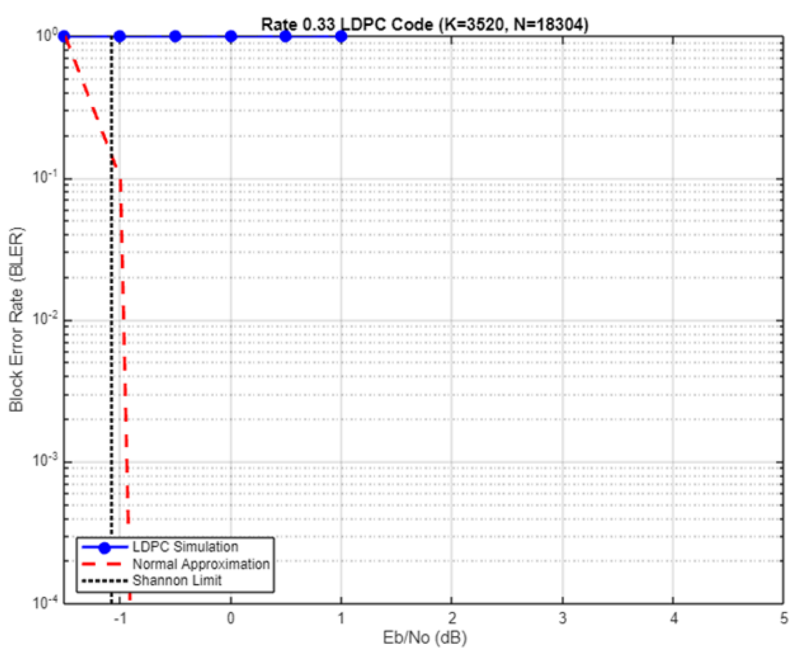
SNR 3.0 dB: FER=0.00e+00, Pc=1.00

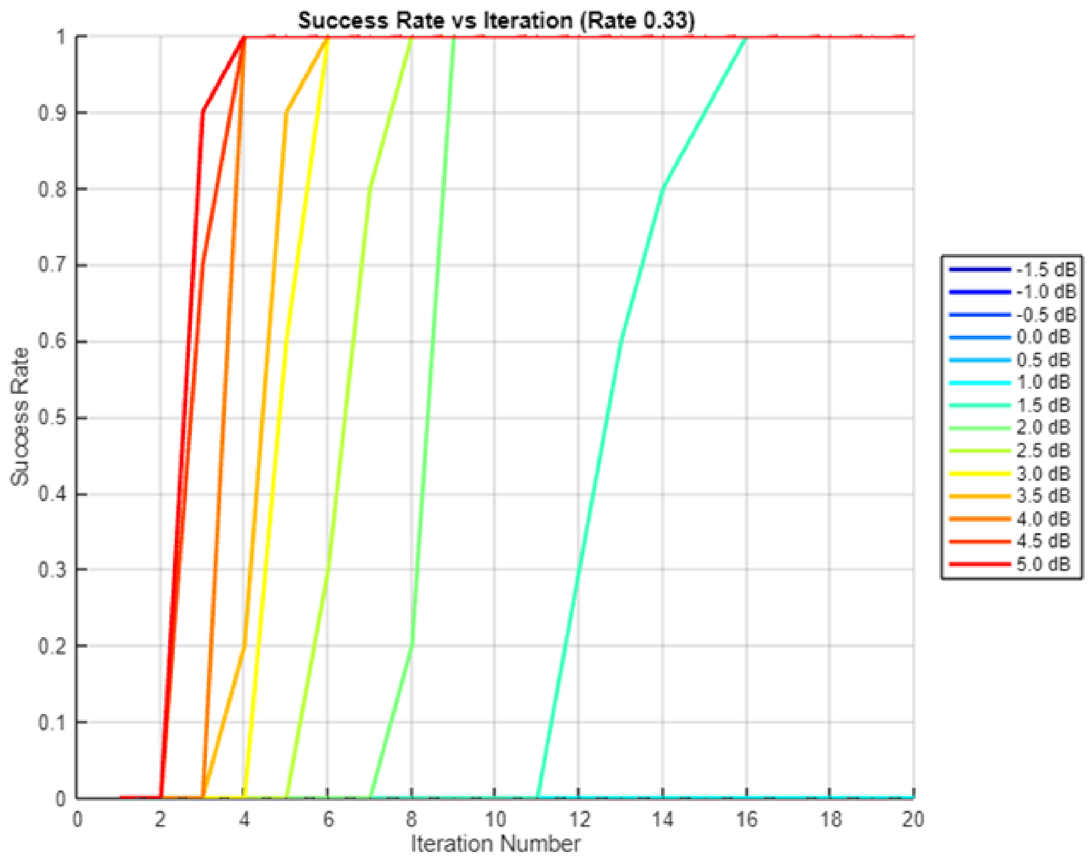
SNR 3.5 dB: FER=0.00e+00, Pc=1.00

SNR 4.0 dB: FER=0.00e+00, Pc=1.00

SNR 4.5 dB: FER=0.00e+00, Pc=1.00

SNR 5.0 dB: FER=0.00e+00, Pc=1.00





Success Rate vs Iteration (Rate 0.33)
Success Rate
Iteration Number
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
2
4
6
8
10
12
14
16
18
20
-1.5 dB
-1.0 dB
-0.5 dB
0.0 dB
0.5 dB
1.0 dB
1.5 dB
2.0 dB
2.5 dB
3.0 dB
3.5 dB
4.0 dB
4.5 dB
5.0 dB

Simulating code rate 0.50 (2/4)

SNR -1.5 dB: FER=1.00e+00, Pc=0.00

SNR -1.0 dB: FER=1.00e+00, Pc=0.00

SNR -0.5 dB: FER=1.00e+00, Pc=0.00

SNR 0.0 dB: FER=1.00e+00, Pc=0.00

SNR 0.5 dB: FER=1.00e+00, Pc=0.00

SNR 1.0 dB: FER=1.00e+00, Pc=0.00

SNR 1.5 dB: FER=9.00e-01, Pc=0.10

SNR 2.0 dB: FER=0.00e+00, Pc=1.00

SNR 2.5 dB: FER=0.00e+00, Pc=1.00

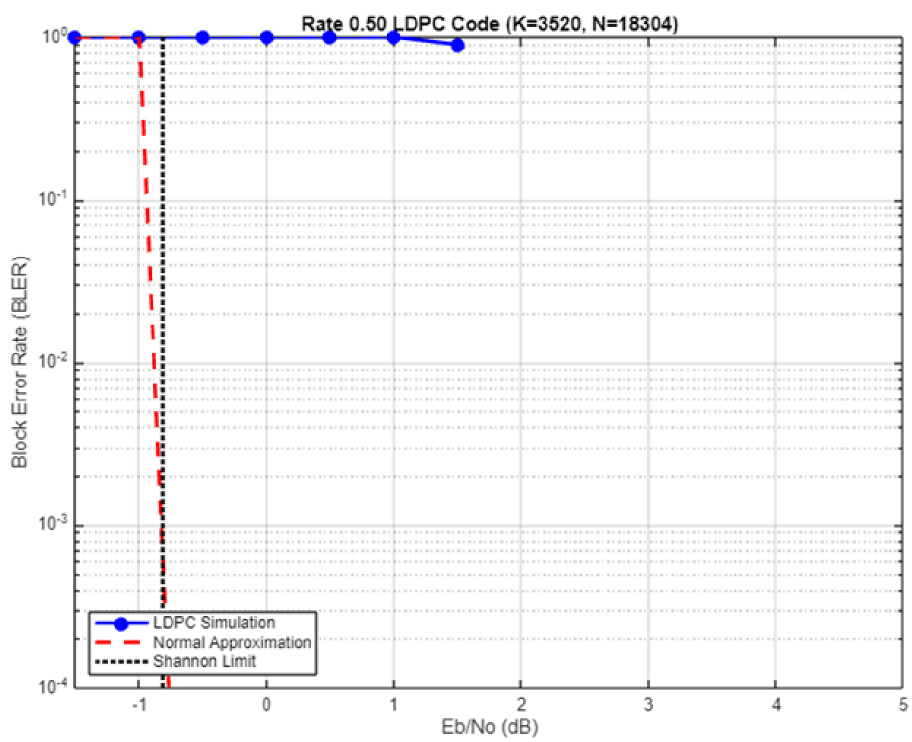
SNR 3.0 dB: FER=0.00e+00, Pc=1.00

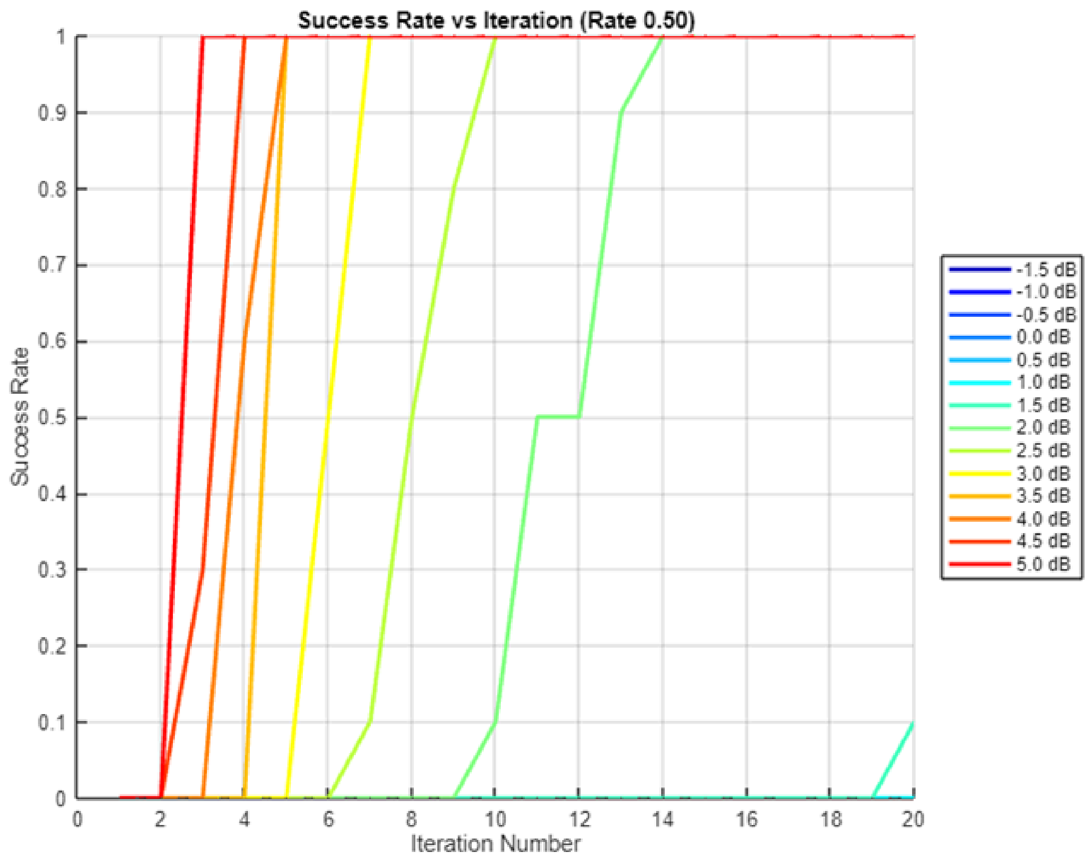
SNR 3.5 dB: FER=0.00e+00, Pc=1.00

SNR 4.0 dB: FER=0.00e+00, Pc=1.00

SNR 4.5 dB: FER=0.00e+00, Pc=1.00

SNR 5.0 dB: FER=0.00e+00, Pc=1.00





Success Rate vs Iteration (Rate 0.50)
Success Rate
Iteration Number
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
2
4
6
8
10
12
14
16
18
20
-1.5 dB
-1.0 dB
-0.5 dB
0.0 dB
0.5 dB
1.0 dB
1.5 dB
2.0 dB
2.5 dB
3.0 dB
3.5 dB
4.0 dB
4.5 dB
5.0 dB

Simulating code rate 0.60 (3/4)

SNR -1.5 dB: FER=1.00e+00, Pc=0.00

SNR -1.0 dB: FER=1.00e+00, Pc=0.00

SNR -0.5 dB: FER=1.00e+00, Pc=0.00

SNR 0.0 dB: FER=1.00e+00, Pc=0.00

SNR 0.5 dB: FER=1.00e+00, Pc=0.00

SNR 1.0 dB: FER=1.00e+00, Pc=0.00

SNR 1.5 dB: FER=1.00e+00, Pc=0.00

SNR 2.0 dB: FER=3.00e-01, Pc=0.70

SNR 2.5 dB: FER=0.00e+00, Pc=1.00

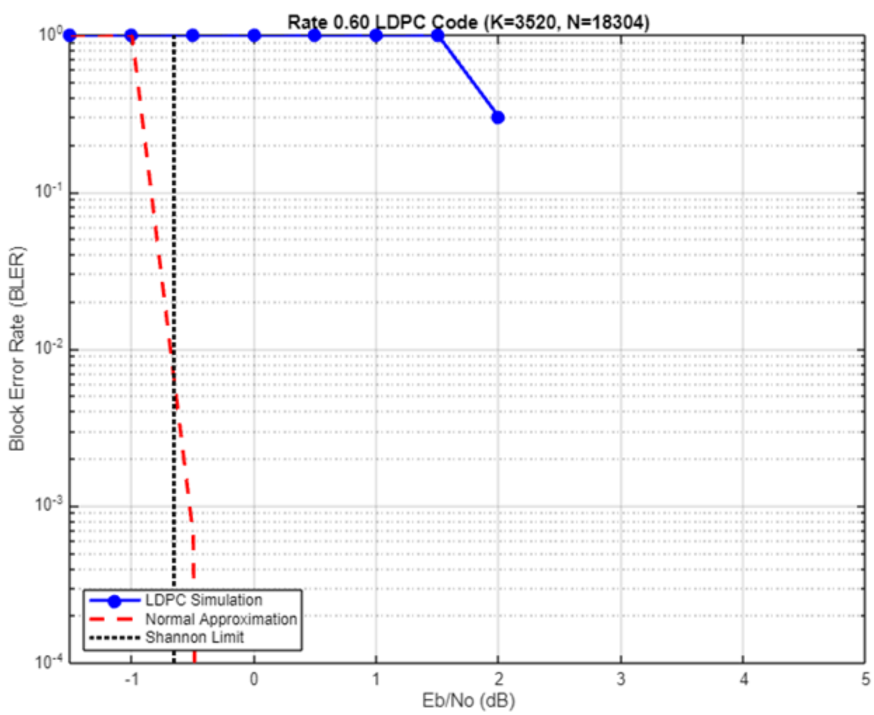
SNR 3.0 dB: FER=0.00e+00, Pc=1.00

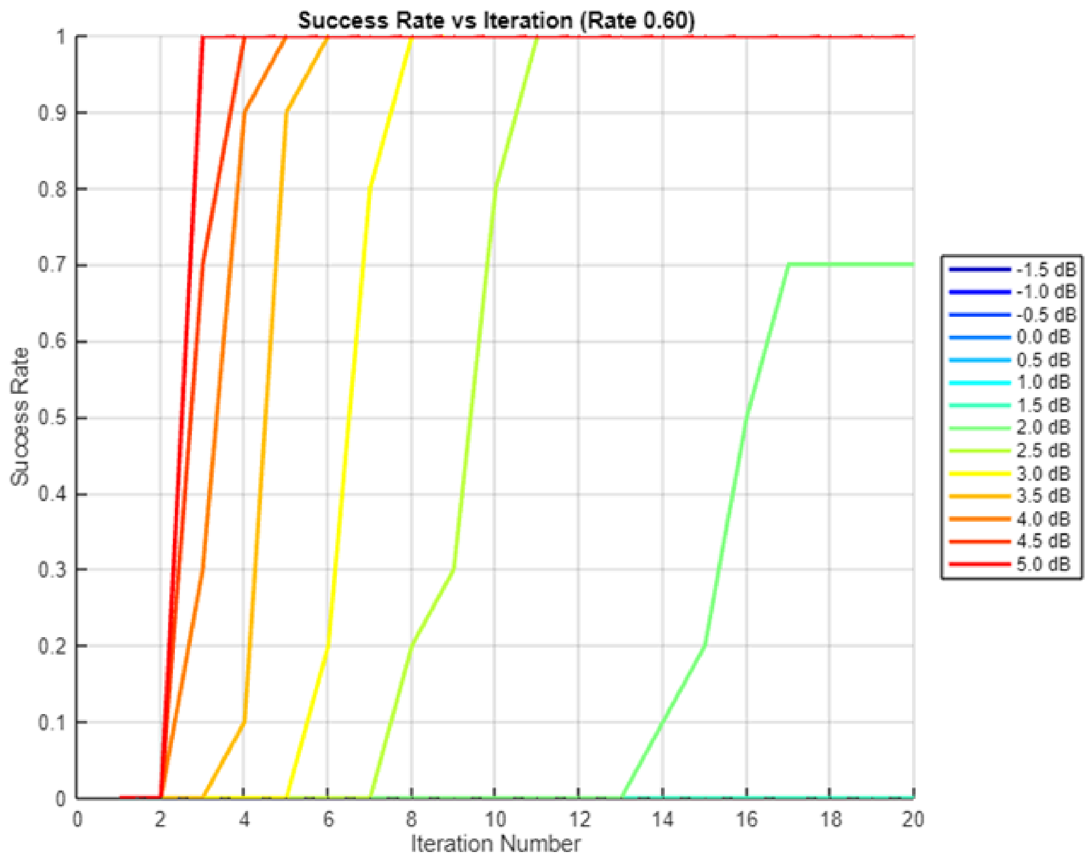
SNR 3.5 dB: FER=0.00e+00, Pc=1.00

SNR 4.0 dB: FER=0.00e+00, Pc=1.00

SNR 4.5 dB: FER=0.00e+00, Pc=1.00

SNR 5.0 dB: FER=0.00e+00, Pc=1.00





Success Rate vs Iteration (Rate 0.60)
Success Rate
Iteration Number
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
2
4
6
8
10
12
14
16
18
20
-1.5 dB
-1.0 dB
-0.5 dB
0.0 dB
0.5 dB
1.0 dB
1.5 dB
2.0 dB
2.5 dB
3.0 dB
3.5 dB
4.0 dB
4.5 dB
5.0 dB

Simulating code rate 0.80 (4/4)

SNR -1.5 dB: FER=1.00e+00, Pc=0.00

SNR -1.0 dB: FER=1.00e+00, Pc=0.00

SNR -0.5 dB: FER=1.00e+00, Pc=0.00

SNR 0.0 dB: FER=1.00e+00, Pc=0.00

SNR 0.5 dB: FER=1.00e+00, Pc=0.00

SNR 1.0 dB: FER=1.00e+00, Pc=0.00

SNR 1.5 dB: FER=1.00e+00, Pc=0.00

SNR 2.0 dB: FER=1.00e+00, Pc=0.00

SNR 2.5 dB: FER=1.00e+00, Pc=0.00

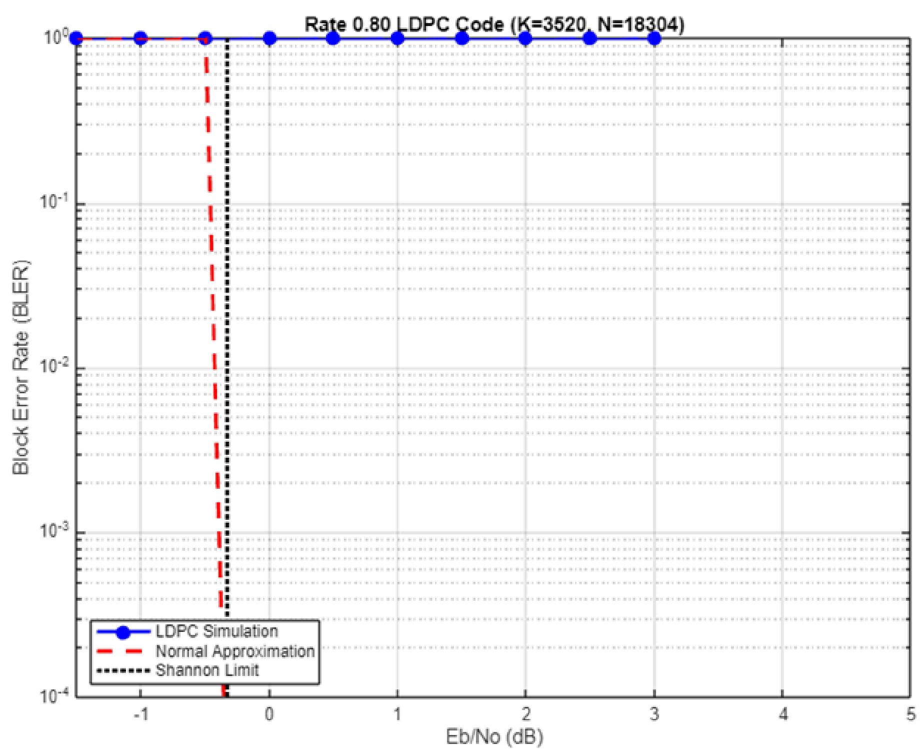
SNR 3.0 dB: FER=1.00e+00, Pc=0.00

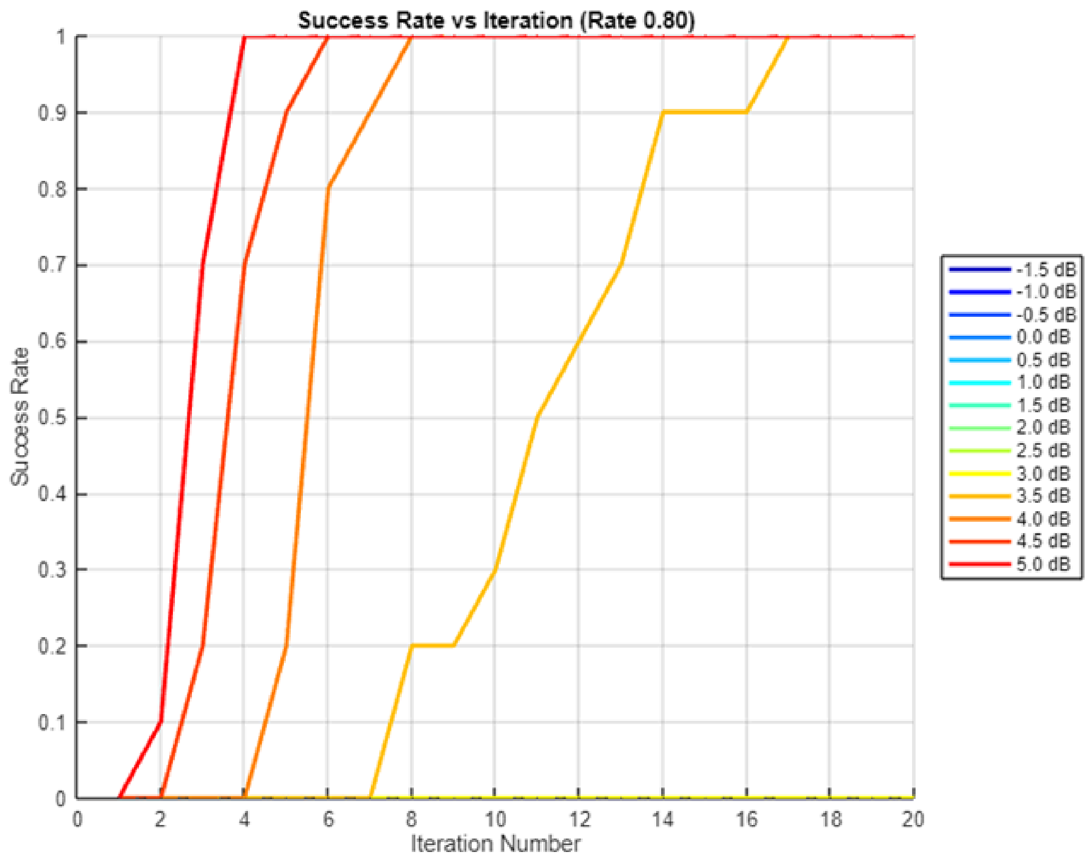
SNR 3.5 dB: FER=0.00e+00, Pc=1.00

SNR 4.0 dB: FER=0.00e+00, Pc=1.00

SNR 4.5 dB: FER=0.00e+00, Pc=1.00

SNR 5.0 dB: FER=0.00e+00, Pc=1.00




Success Rate vs Iteration (Rate 0.80)
Success Rate
Iteration Number
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
2
4
6
8
10
12
14
16
18
20
-1.5 dB
-1.0 dB
-0.5 dB
0.0 dB
0.5 dB
1.0 dB
1.5 dB
2.0 dB
2.5 dB
3.0 dB
3.5 dB
4.0 dB
4.5 dB
5.0 dB