Dhirubhai Ambani University

CT 216
INTRODUCTION TO COMMUNICATION SYSTEM
PROJECT:
LDPC CODES

Under the Guidance of Prof. Yash Vasavda
Mentor TA:

Group 24

HONOR CODE:

- We declare that:
  - The work that we are presenting is our own work.
  - We have not copied the work (the code, the results, etc.) that someone else has done.
  - Concepts, understanding and insights we will be describing are our own.
  - We make this pledge truthfully. We know that violation of this solemn pledge can carry grave consequences.

Members:
202301258 RISHIK YALAMANCHILI
202301259 CHIRAG KATKORIYA
202301260 MAHEK JIKKAR
202301261 PATEL NAKUL JAYMITKUMAR
202301262 PRIYANKA GARG
202301263 YUG SAVALIA
202301264 KRISH PATEL
202301265 JALU RISHABH
202301266 VANSH VORA
202301267 ARAV VAITHA
202301268 SIDDHANT SHEKHAR

CONTENTS:

1. Brief about LDPC codes
2. Importance of Base Matrix and Expansion Factor
3. Brief aboBPSK Modulation, AWGN noise added in the channel encoding
4. Two type of decoding: Soft and Hard
5. Code for Hard decoding
6. Code for Soft decoding
7. Graphs

What are LDPC codes?

LDPC codes (Low-Density Parity-Check codes) are a class of **error-correcting codes** defined by a very **sparse parity-check matrix—**meaning the matrix that enforces the code has mostly zeros and only a few ones. This **sparsity** makes both the representation and the processing of the code much more efficient than with dense-matrix codes. MacKay shows how you can view an LDPC code as a **factor graph**, where simple nodes exchange messages during decoding.

Why are they used?

- **High reliability**: They detect and correct errors introduced by noise or interference in a channel, greatly reducing the chance of undetected mistakes.

- **Near-optimal performance**: Well-designed LDPC codes operate very close to the theoretical **Shannon limit**, squeezing almost every bit of usable capacity out of a noisy link.

- **Fast, low-complexity decoding**: Through an **iterative message-passing algorithm** (often called belief propagation), decoding can be performed quickly on modern hardware with low power and latency.

- **Flexibility**: By choosing the right pattern of ones in the parity-check matrix (regular vs. irregular LDPC), you can tailor the code for different data rates and block lengths.

Where are they used?

- **Satellite and deep-space communications**: Reliable links over vast distances, where retransmission costs are huge.

- **5G cellular and Wi-Fi (IEEE 802.11)**: High-speed wireless links that must cope with interference and fading.
- **Digital Video Broadcasting (DVB-S2)**: Protects video streams over cable and satellite TV.
- **Data storage devices**: SSDs and HDDs use LDPC to correct read/write errors, extending device lifespan and data integrity.

What is a Base Matrix?
A **Base Matrix** (also called a **protograph**) is a small, simple **parity-check matrix** that captures the core connections of an LDPC code. Instead of writing out the full, huge matrix used for actual encoding and decoding, we start with this tiny blueprint. Each row and column in the Base Matrix contains either a zero (no connection) or a small positive integer (the number of parallel edges).

- The Base Matrix defines how bits (variable nodes) link to checks (check nodes) in a **bipartite graph**.
- It is easy to design and analyze because of its small size.
- Changing numbers in the Base Matrix lets you explore different error-correction behaviors before building the full code.

What is the Expansion Factor?
The **Expansion Factor** (sometimes called the **lifting factor**) is the number by which we "blow up" the Base Matrix to get the real LDPC code's parity-check matrix.

- If the Expansion Factor is Z, each entry in the Base Matrix becomes a $Z \times Z$ submatrix.
    - A "-1" turns into a $Z \times Z$ all-zero block (no edges).
    - A "0" becomes a single permutation of the $Z \times Z$ identity matrix (one copy of each connection).
    - A higher integer, say "r," becomes a Z*Z identity matrix with "r" columns shifted;

- By choosing different permutations for each block, we ensure the full matrix stays **sparse** and offers good error-correction.

Why they matter together

- **Design flexibility**: You can experiment with a handful of Base Matrices and many Expansion Factors to meet different **code rates** and **block lengths**.

- **Performance tuning**: Adjusting the Base Matrix structure or the Expansion Factor lets you trade off between **decoding speed**, **latency**, and **error-correction strength**.

- **Efficient implementation**: Hardware and software can exploit the regular patterns created by expansion, leading to simple memory layouts and fast **message-passing** during decoding.

Specialities of the **Base Matrix** + **Expansion Factor** approach in LDPC design:

- **Protograph-based structure**
  The small **Base Matrix** (protograph) captures the essential connectivity. This lets designers **analyze** code properties (like cycles and girth) at a tiny scale before expanding—making it far easier to predict performance.

- **Rate and length flexibility**
  You can mix and match different Base Matrices with various **Expansion Factors** (Z) to achieve a wide range of **code rates** and **block lengths**, without redesigning from scratch.

- **Hardware-friendly encoding**
  The quasi-cyclic nature also supports efficient **linear-time encoding** via sparse generator matrices, which is vital for high-throughput applications.

- **Improved girth and cycle control**
  Choosing permutations in the expansion step allows you to **avoid short cycles** in the bipartite graph, boosting the effectiveness of **iterative decoding**.

- **Standard compliance**
  Many communications and storage standards (e.g., **5G**, **Wi-Fi**, **DVB-S2**) adopt protograph-based LDPCs, so leveraging Base Matrix + Expansion Factor designs ensures **interoperability**.

Together, these specialties make the Base Matrix + Expansion Factor method a **powerful**, **scalable**, and **practical** toolkit for crafting LDPC codes that meet diverse performance and implementation demands.

BPSK Modulation with AWGN Channel and Encoding – Summary

BPSK Modulation converts each bit into a carrier wave with one of two **phases**:

- Bit **0** → **+1** (or 0° phase)
- Bit **1** → **–1** (or 180° phase)

The modulated signal is then transmitted over a **noisy channel**—typically modeled as an **AWGN (Additive White Gaussian Noise)** channel.

AWGN Channel
This channel adds **Gaussian noise** to the transmitted signal. So if a symbol $\mathbf{x} \in \{+1, -1\}$ is transmitted, the received signal is:
$\mathbf{y = x + n}$
where **n** is a random value drawn from a **normal distribution** with mean zero and variance $\boldsymbol{\sigma^2}$. The noise is independent across symbols.

Encoding Before Modulation
Before BPSK, the input data is typically passed through a **channel encoder** (e.g., **LDPC**, **Hamming**, or **convolutional** code).

- **Purpose**: Add **redundancy** to detect and correct errors caused by noise
- **Output**: A longer sequence of encoded bits, which are then modulated using BPSK
- Encoding improves the system's **bit error rate (BER)** under noisy conditions

Demodulation and Decoding

- The receiver uses **coherent detection**, comparing the noisy value **y** to 0:

    - **y > 0** → decide bit **0**
    - **y < 0** → decide bit **1**

- The detected bitstream is passed through a **channel decoder** to correct errors using the added redundancy

Key Specialities

- **Robustness**: BPSK combined with encoding resists noise effectively
- **Simple decision rule**: One threshold for demodulation
- **Optimality**: For binary signaling over AWGN, BPSK is near optimal in terms of **error performance**

Soft vs. Hard Decoding – Summary

In digital communication over noisy channels like AWGN, decoding is the process of estimating the original bit sequence from the received noisy signal. There are two main approaches:

Hard Decoding

- The receiver first makes a binary decision for each received symbol (e.g., “0” or “1”)
- These decisions are then passed to the channel decoder
- This process treats the channel as a binary symmetric channel with fixed probability of error

Advantages:

- Low complexity
- Fast implementation

Limitations:

- Discards soft information (e.g., signal reliability)
- Lower error correction performance

Soft Decoding

- The receiver forwards the raw channel values or likelihoods directly to the decoder
- The decoder interprets these values as probabilistic evidence
- This aligns with Bayesian inference and uses log-likelihood ratios to improve decision accuracy

Advantages:

- More accurate decoding by using full channel information
- Closer to optimal in terms of error performance

Limitations:

- Higher computational complexity
- Requires more memory and processing

Key Difference:
Hard decoding uses binary estimates, while soft decoding uses real-valued beliefs, making it more effective when working with iterative decoders like those used in LDPC or turbo codes.

In information-theoretic terms, soft decoding better preserves and utilizes the mutual information between the transmitted and received data, achieving performance closer to channel capacity.

Hard Decoding- Code

```
colors = [ 0.0, 0.7, 0.8;

 0.12, 0.34, 0.57;
```

```
0.91, 0.15, 0.76;

0.31, 0.12, 0.77;

0.93, 0.13, 0.65;

0.55, 0.51, 0.87;

0.61, 0.78, 0.79;

0.01, 0.31, 0.39;

0.71, 0.25, 0.81;

0.83, 0.69, 0.44;

0.06, 0.40, 0.74;

0.18, 0.18, 0.53;

0.34, 0.72, 0.53;

0.94, 0.38, 0.64;

0.70, 0.15, 0.88;

0.60, 0.67, 0.09;

0.91, 0.29, 0.31;

0.80, 0.86, 0.31;

0.19, 0.93, 0.42;

0.95, 0.79, 0.21;

0.14, 0.41, 0.05

];


baseGraph5GNR = 'NR_1_5_352'; % load 5G NR LDPC base H matrix, use both NR_2_6_52
and NR_1_5_352

codeRates = [2/3,1/2,1/3,3/4];
```

```
[B,Hfull,z] = nrldpc_Hmatrix(baseGraph5GNR);



EbNodb= 0:0.5:10;

Nsim=50;

max_iter=20;

ef=1:1:20;

for cr=codeRates


    [mb,nb] = size(B);

    kb = nb - mb; % 5G NR specific details

    kNumInfoBits = kb * z;

    k_pc = kb-2;

    nbRM = ceil(k_pc/cr)+2;

    nBlockLength = nbRM * z;


    H = Hfull(:,1:nBlockLength);

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

    H = H(1:nChecksNotPunctured,:);


    [row,col]=size(H);

    k1=(col-row);

    msg = zeros(row, col); % initialising msg(matrix of binary messages exchanged between CNs
and VNs in each iteration)

    cn_to_vn = cn_vn(H);     % shows ith cn connected to which all vns
```

```
    vn_to_cn = vn_cn(H); %shows jth vn connected to which all cns


    decoding_error = zeros(1, length(EbNodb));

    bit_error=zeros(1,length(EbNodb));

    d_iter=1;


    dcolor=1;

   f = figure;

set(f, 'Units', 'normalized', 'OuterPosition', [0 0 1 1]);

    for SNR=EbNodb

        SNR

        eb_no=10^(SNR/10);


        sigma= (1/(sqrt(2 * cr* eb_no)));


        error1=0;

        %itr_success=zeros(1,max_iter);

        itr_success=Nsim.*ones(1,max_iter);


        success = 0;

        vn_sum_vec=zeros(1,col);


        for nsim=1:Nsim

           org_msg = randi([0 1],1, (nb-mb)*z); % Generate information (or message) bit vector
```

```
%org_msg = randi([0 1], (nb-mb)*z, 1);

%b=zeros(1,kNumInfoBits);

 encoded_msg = nrldpc_encode(B,z,org_msg); % Encode using 5G NR LDPC base
matrix

 encoded_msg = encoded_msg(1:nBlockLength);

 n=length(encoded_msg);

 decoded_msg = zeros(1,col);

 bpsk_msg = 1 - 2.*encoded_msg;


 noise = sigma * randn(1,n);


 r = bpsk_msg + noise; %received_bpsk


 received = (r<0);


 prev_msg1=received;


 for iter=1:1:max_iter


    % First iteration: Load received bits into VNs and send to CNs

    if (iter == 1)

        for i = 1:col

            for j = vn_to_cn{i,1}

                msg(j, i) = received(1, i);

            end
```

```
        end

% Other iterations: VNs perform majority voting from CN messages
      else
        for i = 1:col
          for j=vn_to_cn{i,1}
            sum_others = vn_sum_vec(1,i) - msg(j,i);
            msg(j,i) = sum_others > (length(vn_to_cn{i,1})/2);
          end
        end
      end

      % CN to VN message passing using XOR
      for i = 1:row
        xor_val = 0;
        %computing xor of all the values received by CN
        for j=cn_to_vn{i,1}
          xor_val = mod((xor_val + msg(i,j)),2);
        end

        %sending the message to particular VNs connected
        for j=cn_to_vn{i,1}
```

```
            msg(i,j) = mod((xor_val + msg(i,j)),2);
        end
    end

% VN makes final decision using majority voting with original received bit
    for i = 1:col
        sum1 = received(1,i);
        temp = msg(:,i);
        sum1 = sum1 + sum(temp);
        vn_sum_vec(1,i)=sum1;
        decoded_msg(1,i) = sum1>((length(vn_to_cn{i,1})+1)/2);
    end

   % if(sum(xor(decoded_msg(1:k1),org_msg))==0)
    %       success = success+1;
    %       break;
    %else
    %    itr_success(1,iter)=itr_success(1,iter)-1;

    %end
    check = 1;
```

```
for i=1:(col-row)
    if(decoded_msg(i) ~= org_msg(i))
        check = 0;
        break
    end
end
if check==1
    success = success+1;
    for j=iter:max_iter
        itr_success(1, j) = itr_success(1, j)+1;
    end
    break;
end

 check2 = 1;
for i=1:col
    if decoded_msg(1, i) ~= prev_msg1(1, i)
        check2 = 0;
        break;
    end
end

%if(sum(xor(prev_msg1,decoded_msg))==0)
 %     for tmp_itr=iter+1 : max_iter
```

```
            %        itr_success(1,tmp_itr)=itr_success(1,tmp_itr)-1;
             %   end
              %  break;
          % end
           %   prev_msg1 = decoded_msg;
           if check2==1
               break;
           end


           prev_msg1 = decoded_msg;
       end
       for i=1:col
           if decoded_msg(1, i)~=encoded_msg(1, i)
               error1 = error1+1;
           end
       end
   end



%hold off;
%plot(EbNodb,decoding_error,'LineWidth',2);
%plot(ef,itr_success./Nsim);
%xlabel("Iteration number");
%ylabel('Success Probability at each iteration');
```

```
    %title('Success Probability v/s iteration for Hard Decoding');

%legend('0.0','0.5','1.0','1.5','2.0','2.5','3.0','3.5','4.0','4.5','5.0','5.5','6.0','6.5','7.0','7.5','8.0','8.5','9.
0','9.5','10.0','northeastoutside');

    %grid on;

    hold on;

    decoding_error(1, d_iter) = (Nsim-success)/Nsim;

            bit_error(1,d_iter)=error1/(Nsim*col);

            d_iter=d_iter+1;


    end

    %plot(EbNodb,bit_error,'LineWidth',2);


    %Nchecks = size(H,1);

    %hold on;

    plot(EbNodb, bit_error, 'LineWidth', 2);

    xticks(0:1:10);

    xlabel("Eb/No (dB)");

    ylabel("Bit error probability");

    title(['Hard Decision Bit Error Probability, Coderate = ', num2str(cr)]);

    grid on;

    %hold on;




end
```

```
    %xlabel("Eb/No (dB)");

    %ylabel("Decoding error probability");

    %title("Hard Decision Decoding error probability");

    %legend('Coderate = 1/4','Coderate = 1/3', 'Coderate = 1/2',...

    %'Coderate = 3/5');


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

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

function out = cn_vn(H)
  [row, col]=size(H);
  out=cell(row,1);
  for i = 1:row
      out{i,1} = [];
  end


  for i=1:row
```

```
    for j=1:col
        if(H(i,j)==1)
            out{i,1} = [out{i,1} j];
        end
    end
  end
end


function out = vn_cn(H)
  [row, col]=size(H);
  out=cell(col,1);
  for i = 1:col
      out{i,1} = [];
  end


  for i=1:col
      for j=1:row
          if(H(j,i)==1)
              out{i,1} = [out{i,1} j];
          end
      end
  end
```

```
end
```

Soft Decoding- Code

```
colors = [ 0.0, 0.7, 0.8;
           0.12, 0.34, 0.57;
           0.91, 0.15, 0.76;
           0.31, 0.12, 0.77;
           0.93, 0.13, 0.65;
           0.55, 0.51, 0.87;
           0.61, 0.78, 0.79;
           0.01, 0.31, 0.39;
           0.71, 0.25, 0.81;
           0.83, 0.69, 0.44;
           0.06, 0.40, 0.74;
```

```
        0.18, 0.18, 0.53;
        0.34, 0.72, 0.53;
        0.94, 0.38, 0.64;
        0.70, 0.15, 0.88;
        0.60, 0.67, 0.09;
        0.91, 0.29, 0.31;
        0.80, 0.86, 0.31;
        0.19, 0.93, 0.42;
        0.95, 0.79, 0.21;
        0.14, 0.41, 0.05
       ];

%% starting
%for matrix NR_2_6_52
baseGraph5GNR = 'NR_2_6_52'; % load 5G NR LDPC base H matrix
coderate = [1/4 1/3 1/2 3/5];  % Different code rates more commonly used with this base graph
eb_no_dbvec = 0:0.5:10;
[B,Hfull,z] = nrldpc_Hmatrix(baseGraph5GNR,52); % Convert the base H matrix to binary H matrix

% Modified simulation parameters as requested
nsim = 10;
max_it = 50;
iterations = 1:1:max_it;
```

```
% Arrays to store simulation results for all code rates
success_prob_vs_ebno = zeros(length(coderate), length(eb_no_dbvec));
ber_vs_ebno = zeros(length(coderate), length(eb_no_dbvec));
shannon_bounds = zeros(length(coderate), length(eb_no_dbvec));

% Create main figures for final results
figure(100); % Figure for Success Probability vs Eb/No plots
hold on;

figure(200); % Figure for BER vs Eb/No plots
hold on;


%% for different coderates and monte carlo
cr_idx = 1;
for cr = coderate
    % Create a new figure for success probability vs iteration for this code rate
    % Use figure number = cr_idx to create separate figures for each code rate
    figure(cr_idx);
    clf; % Clear the figure
    hold on;

    %performing rate matching
    [mb,nb] = size(B); kb = nb - mb; % 5G NR specific details
```

```
kNumInfoBits = kb * z; % Number of information bits

k_pc = kb-2; nbRM = ceil(k_pc/cr)+2; % Some 5G NR specific details

nBlockLength = nbRM * z; % Number of encoded bits
H = Hfull(:,1:nBlockLength);
nChecksNotPunctured = mb*z - nb*z + nBlockLength;
H = H(1:nChecksNotPunctured,:); % this is the binary H matrix

%rate matching done
[row,col] = size(H);
L = zeros(size(H));
k = col - row;
cn_to_vn_map = cn_vn(H); %shows ith cn connected to which all vns
vn_to_cn_map = vn_cn(H); %shows ith vn connected to which all cns



%performing soft decoding
d_iter = 1;
decoding_error = zeros(1,length(eb_no_dbvec));
bit_error = zeros(1,length(eb_no_dbvec));

% Calculate Shannon bound for this code rate
% Shannon bound for BPSK over AWGN: Eb/N0 = (2^R - 1)/(R*2)
```

```
    shannon_bound_ebno_linear = (2^cr - 1)/(cr*2);
    shannon_bound_ebno_db = 10*log10(shannon_bound_ebno_linear);

    eb_no_idx = 1;
    for eb_no_db = eb_no_dbvec
        eb_no = 10^(eb_no_db/10);
        sigma = sqrt(1/(2*cr*eb_no));
        success = 0;
        error1 = 0;  % Total bit errors
        total_bits = 0;  % Total bits transmitted
        itr_success = nsim.*ones(1,max_it);
        vn_sum_vec = zeros(1,col);

        for sim=1:nsim

            org_msg = randi([0 1],[k 1]); % Generate information (or message) bit vector
            encoded_msg = nrldpc_encode(B,z,org_msg'); % Encode using 5G NR LDPC base
matrix
            encoded_msg = encoded_msg(1:nBlockLength);

            n = length(encoded_msg);
            %performing bpsk modulation
            bpsk_msg = 1 - 2.*encoded_msg;
            %generating noise
            noise = sigma * randn(1,n);
```

```
received_bpsk = bpsk_msg + noise;

%changing message back to bits for received msg only

received_bits = (received_bpsk<0);

prev_msg = received_bits;

c_hat = zeros(1,col);


for it = 1:max_it


    %message from VN to CN


    %for 1st iteration, load all received values into VN and

    %send them directly to CN

    if(it==1)

        for i=1:col

            for j=vn_to_cn_map{i,1}

                L(j,i) = received_bpsk(1,i);

            end

        end


    %otherwise subtract the current value from the total sum vec.

    else

        for i = 1:col

            for j=vn_to_cn_map{i,1}
```

```
            L(j,i) = vn_sum_vec(1,i) - L(j,i);
        end
    end
end


%message from CN to VN using minsum approximation
for i=1:row
    min1=1e9;           %first minimum
    min2=1e9;           %second minimum
    pos=-1;             %VN number which has minimum1 value
    total_sign=1;       % the sign obtained by multiplying all the non-zero elemnts in
the row

    for j=cn_to_vn_map{i,1}
        ele = abs(L(i,j));

        %computing the minimums
        if(ele<=min1)
            min2=min1;
            min1=ele;
            pos = j;
        elseif(ele<=min2 && ele>min1)
            min2=ele;
        end
```

```
        %computing overall sign
        if(L(i,j)~=0)
            total_sign = total_sign*(sign(L(i,j)));
        end
    end

    %sending the message
    for j=cn_to_vn_map{i,1}
        if(j~=pos)
            L(i,j) = total_sign * sign(L(i,j)) * min1;
        else
            L(i,j) = total_sign * sign(L(i,j)) * min2;
        end
    end
end

%finding sum of values received by each vn
for i = 1:col
    sum1 = received_bpsk(1,i);
    temp = L(:,i);
    sum1 = sum1 + sum(temp);
    vn_sum_vec(1,i)=sum1;
end
```

```
        c_hat = (vn_sum_vec<0);


        if(sum(xor(c_hat(1:k),org_msg'))==0)
            success = success+1;
            break;
        else
            itr_success(1,it)=itr_success(1,it)-1;
        end
        % Calculate bit errors for BER calculation
        bit_errors_this_frame = sum(c_hat ~= encoded_msg);
        error1 = error1 + bit_errors_this_frame;
        total_bits = total_bits + col;
        if(sum(xor(prev_msg,c_hat))==0)
            for tmp_itr=it+1:max_it
                itr_success(1,tmp_itr)=itr_success(1,tmp_itr)-1;
            end
            break;
        end
        prev_msg = c_hat;
    end
end
        % Plot success probability vs iteration for this Eb/No value
% Using the current figure for this code rate
figure(cr_idx);
```

```
        plot(iterations, itr_success./nsim, 'Color', colors(eb_no_idx,:), 'DisplayName',
[num2str(eb_no_db) ' dB']);

            % Store results for this Eb/No point

        decoding_error(1, eb_no_idx) = (nsim-success)/nsim;

        bit_error(1, eb_no_idx) = error1/total_bits;

            % Store results in our arrays for later plotting

        success_prob_vs_ebno(cr_idx, eb_no_idx) = success/nsim;  % Success probability

        ber_vs_ebno(cr_idx, eb_no_idx) = error1/total_bits;      % BER


        eb_no_idx = eb_no_idx + 1;

    end

     % Finalize the Success Probability vs Iteration plot for this code rate

    figure(cr_idx);

    xlabel("Iteration number");

    ylabel('Success Probability at each iteration');

    title(['Success Probability vs. Iteration for Soft Decoding, Code Rate = ', num2str(cr)]);

    grid on;

    legend('show', 'Location', 'southeast');

    hold off;

        % Plot Success Probability vs Eb/No for this code rate in the main figure

    figure(100);

    plot(eb_no_dbvec, success_prob_vs_ebno(cr_idx, :), 'LineWidth', 2, 'DisplayName',
['Coderate = ', num2str(cr)]);

        % Plot BER vs Eb/No for this code rate in the main figure

    figure(200);
```

```
    semilogy(eb_no_dbvec, ber_vs_ebno(cr_idx, :), 'LineWidth', 2, 'DisplayName', ['Coderate = ', num2str(cr)]);


    cr_idx = cr_idx + 1;

end

% Finalize Success Probability vs Eb/No plot

figure(100);

xlabel("Eb/No (dB)");

ylabel("Success Probability");

title("Success Probability vs Eb/No for Different Code Rates");

legend('show');

grid on;

hold off;

%% Plot Eb/No vs Bit Error Rate with Shannon Capacity Limits

figure(200);

hold on;

% Add Shannon bounds to the BER plot

for i = 1:length(coderate)

    semilogy(eb_no_dbvec, shannon_bounds(i,:), 'k--', 'LineWidth', 1.5, 'DisplayName', ['Shannon bound, R = ', num2str(coderate(i))]);

end

xlabel("Eb/No (dB)");

ylabel("Bit Error Rate (BER)");

title("Bit Error Rate vs Eb/No with Shannon Capacity Bounds");

legend('show');
```

```
ylim([10^(-5) 1]);

grid on;

hold off;

%% added functionss and plots of normal approximation

% Add this code after the existing color definitions but before the simulation starts

% Normal approximation function

function [P_err] = normal_approximation(n, k, eb_no_db, rate)

    % Calculating normal approximation for finite block length codes

    % Based on Polyanskiy-Poor-Verdú (PPV) bound


    % Parameters

    n = double(n);       % Block length (bits)

    k = double(k);       % Information bits

    R = double(k/n);     % Code rate


    % Convert Eb/N0 from dB to linear scale

    eb_no = 10.^(eb_no_db/10);


    % Calculate SNR

    snr = rate * eb_no;


    % Channel capacity for AWGN (BPSK)

    C = 0.5 * log2(1 + 2*snr);
```

```
    % Channel dispersion for AWGN (BPSK)
    V = 0;
    for y = -10:0.01:10
        py = 1/sqrt(2*pi) * exp(-y.^2/2);
        for x = [-1, 1]
            px = 0.5;  % BPSK probability
            pyx = 1/sqrt(2*pi) * exp(-(y-x*sqrt(2*snr)).^2/2);
            if pyx > 0
                V = V + px * pyx * (log2(pyx/(py*px)) - C)^2 * 0.01;
            end
        end
    end

    % Calculate the normal approximation bound
    Q = @(x) 0.5 * erfc(x/sqrt(2));
    epsilon = Q(sqrt(n/V) * (C - R + log2(n)/(2*n)));



    % Return error probability
    P_err = epsilon;
end



for i = 1:length(coderate)
```

```
    % Calculate normal approximation for each Eb/N0 point
    n = nBlockLength;  % Use the block length from the simulation
    k = kNumInfoBits;  % Use the info bits from the simulation
    normal_approx_ber = zeros(1, length(eb_no_dbvec));

    for j = 1:length(eb_no_dbvec)
        normal_approx_ber(j) = normal_approximation(n, k, eb_no_dbvec(j), coderate(i));
    end

    % Plot normal approximation
    semilogy(eb_no_dbvec, normal_approx_ber, linestyles{mod(i-1,4)+1}, 'Color', [0.5 0.5 0.5],
'LineWidth', 1.5, ...
        'DisplayName', ['Normal Approx, R = ', num2str(coderate(i))]);
end

% Create a dedicated figure for normal approximation comparison
figure(300);
hold on;
for i = 1:length(coderate)
    % Plot actual BER
    semilogy(eb_no_dbvec, ber_vs_ebno(i,:), '-', 'LineWidth', 2, 'Marker', markers(i), 'MarkerSize',
6, ...
        'DisplayName', ['Actual BER, R = ', num2str(coderate(i))]);

    % Calculate normal approximation for each Eb/N0 point
```

```
    n = nBlockLength;  % Use the block length from the simulation
    k = kNumInfoBits;  % Use the info bits from the simulation
    normal_approx_ber = zeros(1, length(eb_no_dbvec));

    for j = 1:length(eb_no_dbvec)
        normal_approx_ber(j) = normal_approximation(n, k, eb_no_dbvec(j), coderate(i));
    end

    % Plot normal approximation
    semilogy(eb_no_dbvec, normal_approx_ber, '--', 'Color', colors(i,:), 'LineWidth', 1.5, ...
        'DisplayName', ['Normal Approx, R = ', num2str(coderate(i))]);

    % Calculate and plot Shannon bound for this code rate
    shannon_bound_ebno_linear = (2^coderate(i) - 1)/(coderate(i)*2);
    shannon_bound_ebno_db = 10*log10(shannon_bound_ebno_linear);

    % Plot vertical line at Shannon bound
    xline(shannon_bound_ebno_db, ':', ['Shannon limit, R = ', num2str(coderate(i))], 'LineWidth',
1.5);
end
xlabel("Eb/No (dB)");
ylabel("Bit Error Rate (BER)");
title("BER Performance with Normal Approximation and Shannon Limits in log scale");
legend('show', 'Location', 'southwest');
ylim([10^(-5) 1]);
```

```
grid on;

hold off;


%% starter code


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

    load(sprintf('%s.txt',BG),BG);

    B = NR_2_6_52;

    [mb,nb] = size(B);

    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; % n = length of codeword, u = number of CNs or parities, k = length
of original message
```

```
    P = H(:,1:K);

    G = [eye(K); P];

    Z = H*G;

end


function out=cn_vn(H)

    [row, col]=size(H);

    out=cell(row,1);

    for i = 1:row

        out{i,1} = [];

    end

    for i=1:row

        for j=1:col

            if(H(i,j)==1)

                out{i,1} = [out{i,1} j];

            end

        end

    end

end



function out=vn_cn(H)

    [row, col]=size(H);

    out=cell(col,1);
```

```
    for i = 1:col
        out{i,1} = [];
    end
    for i=1:col
        for j=1:row
            if(H(j,i)==1)
                out{i,1} = [out{i,1} j];
            end
        end
    end
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

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
```