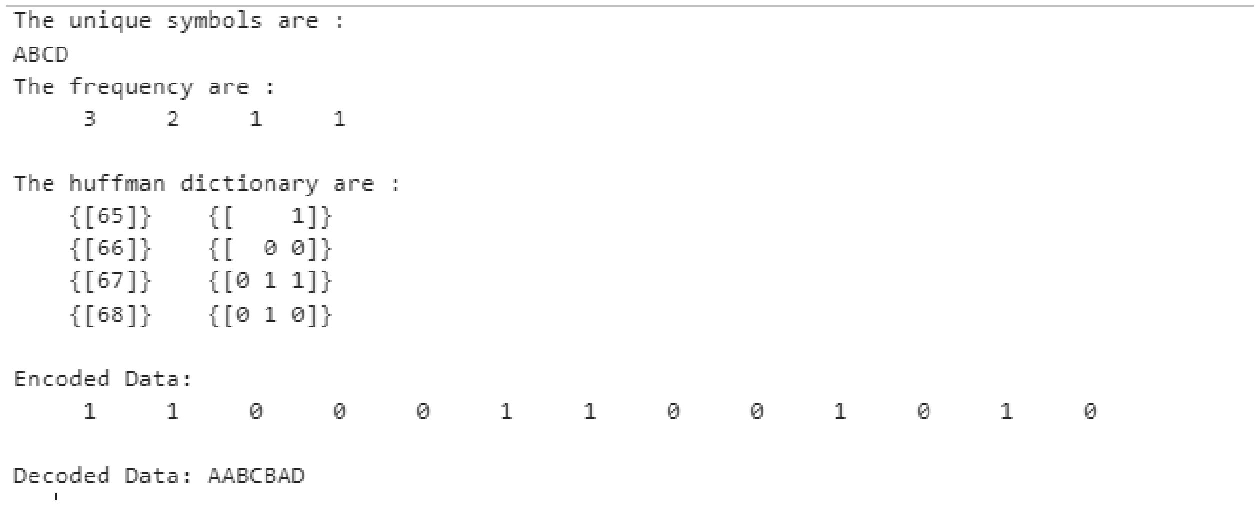
1) Huffman code for encoding and decoding :

Program :

```
input_data = 'AABCBAD';

input_data = upper(input_data);
disp("The unique symbols are :")
symbols = unique(input_data);
disp(symbols)

freq = histcounts(double(input_data), [double(symbols) max(double(input_data))+1]);
disp("The frequency are :")
disp(freq);

probabilities = freq / sum(freq);
 [dict,avglen]=huffmandict(double(symbols),double(probabilities));
disp("The huffman dictionary are :")
disp(dict);


 encoded_data = huffmanenco(input_data, dict);
 disp('Encoded Data: ');
 disp(encoded_data);
 decoded_data = huffmandeco(encoded_data, dict);

 % Display decoded data
 fprintf('Decoded Data: %s\n', char(decoded_data));
```

End*********************

Output:



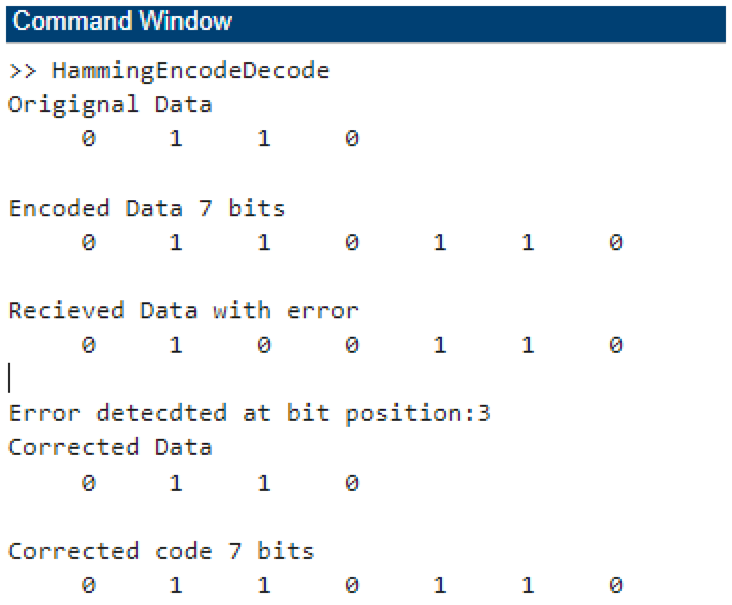
```
The unique symbols are :
ABCD
The frequency are :
     3     2     1     1

The huffman dictionary are :
    {[65]}    {[    1]}
    {[66]}    {[  0 0]}
    {[67]}    {[0 1 1]}
    {[68]}    {[0 1 0]}

Encoded Data:
     1     1     0     0     0     1     1     0     0     1     0     1     0

Decoded Data: AABCBAD
```

2. Hamming encoding and decoding

```
data=[0 1 1 0];
disp("Origignal Data");
disp(data);
encodedData=hammingEncode(data);
disp("Encoded Data 7 bits");
disp(encodedData);

recievedData=encodedData;
recievedData(3)=mod(recievedData(3)+1,2);
disp("Recieved Data with error");
disp(recievedData);

[correctedData,correctedCode]=hammingDecode(recievedData);
disp("Corrected Data");
disp(correctedData);
disp("Corrected code 7 bits");
disp(correctedCode);


function encoded=hammingEncode(data)
if length(data)~=4
    error("Input data must be 4 bit binary vector");
end


 G=[1 0 0 0 1 1 0;
    0 1 0 0 1 0 1;
    0 0 1 0 0 1 1;
    0 0 0 1 1 1 1];

 encoded=mod(data*G,2);
end

function[correctedData,correctedCode]=hammingDecode(recieved)
if length(recieved)~=7
    error("Recieved code must be 7 bits binary vector")
end

H=[ 1 1 0 1 1 0 0;
    1 0 1 1 0 1 0;
    0 1 1 1 0 0 1];


syndrome=mod(H*recieved',2);
if any(syndrome)
    errorPos=bi2de(syndrome',"left-msb");
    fprintf("Error detecdted at bit position:%d\n",errorPos);
    recieved(errorPos)=mod(recieved(errorPos)+1,2);

else
    disp("No errors detected.");
end
```

```
correctedCode=recieved;
correctedData=recieved(1:4);
end
```

output:



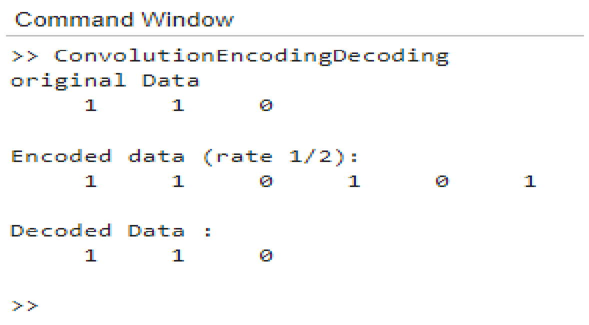
Command Window

```
>> HammingEncodeDecode
Origignal Data
     0     1     1     0

Encoded Data 7 bits
     0     1     1     0     1     1     0

Recieved Data with error
     0     1     0     0     1     1     0

Error detecdted at bit position:3
Corrected Data
     0     1     1     0

Corrected code 7 bits
     0     1     1     0     1     1     0
```

3. Convolution encoding and decoding

```
data=[1 1 0];
disp("original Data");
disp(data);

encodedData=convEncode(data);
disp("Encoded data (rate 1/2):");
disp(encodedData);

%simulate noicy channel
noicyData=encodedData;
noicyData(3)=mod(noicyData(5)+1,2);

decodedData=viterbiDecode(noicyData);
disp("Decoded Data :")
disp(decodedData);

% function for encode
function encoded=convEncode(data)
state=[0 0];
encoded=[];
%loop each input bit

for i=1:length(data)
    state=[data(i) state(1:2)];

    output1=mod(state(1)+state(2)+state(3),2);
    output2=mod(state(1)+state(3),2);
    %Append the output message to encoder
    encoded=[encoded output1 output2];
end
end


% function for decoding
function decoded=viterbiDecode(encoded)

trellis=poly2trellis(3,[7,5]);
decoded=vitdec(encoded,trellis,2,"trunc","hard");
end
```

output:



```
Command Window
>> ConvolutionEncodingDecoding
original Data
     1     1     0

Encoded data (rate 1/2):
     1     1     0     1     0     1

Decoded Data :
     1     1     0

>>
```

4. Gram-schmidt orthogonalization.

File name : gramSchmidtOrthogonalBasis.m

```
function orthogonal_basis = gramSchmidtOrthogonalBasis(vectors)

    % Input: vectors - a matrix where each column is a vector (n x m matrix)

    % Output: orthogonal_basis - a matrix of orthogonal basis vectors

    [n, m] = size(vectors);  % n is the dimension, m is the number of vectors

    orthogonal_basis = zeros(n, m); % Initialize the orthogonal basis matrix


    for i = 1:m

        % Start with the original vector

        orthogonal_vector = vectors(:, i);

        % Subtract projections of previous orthogonal vectors

        for j = 1:i-1

            orthogonal_vector = orthogonal_vector - (dot(orthogonal_basis(:, j),
vectors(:, i)) / dot(orthogonal_basis(:, j), orthogonal_basis(:, j))) *
orthogonal_basis(:, j);

        end

         % Store the orthogonalized vector

        orthogonal_basis(:, i) = orthogonal_vector;

    end


    % Remove any zero columns (if vectors were linearly dependent)

    orthogonal_basis = orthogonal_basis(:, any(orthogonal_basis));

end
```

 ****

file name : orthogonal.m

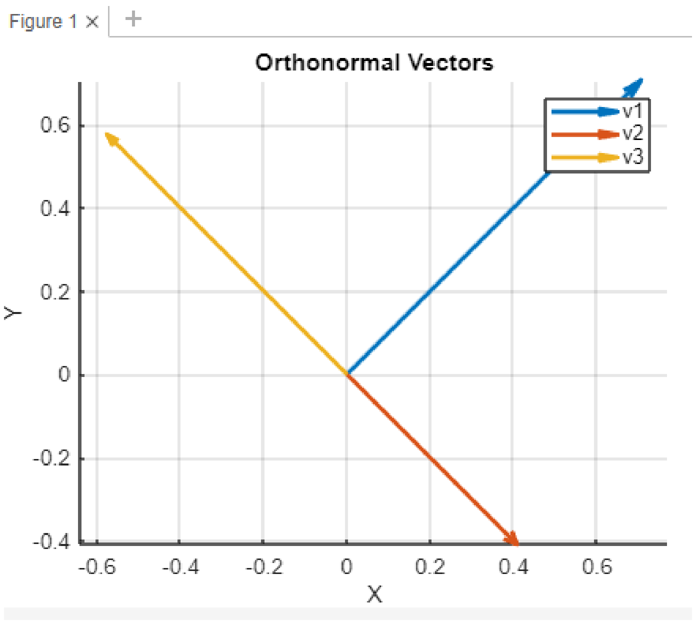
```
vectors=[1 1 0;1 0 1;0 1 1];
```

```
orthogonal_basis=gramSchmidtOrthogonalBasis(vectors);

 % gramSchmidtOrthogonalBasis this function is called by another
file(gramSchmidtOrthogonalBasis.m) with

 % name as the function name

disp('orthogonal vectors are:')

disp(orthogonal_basis);

% Orthonormal function starts from here


% Given set of vectors (Example input as columns)

A = [1 1 0;1 0 1;0 1 1]; % 3 vectors in 3D space


% Number of vectors (columns in A)

[m, n] = size(A);


% Initialize matrix to store the orthonormal vectors

Q = zeros(m, n);


% Gram-Schmidt Process to generate orthonormal basis

for i = 1:n

    % Start with the current vector in the input set

    v = A(:, i);


    % Subtract projections onto the previous orthonormal vectors

    for j = 1:i-1

        v = v - (Q(:, j)' * A(:, i)) * Q(:, j);

    end


    % Normalize the resulting vector to make it unit length
    Q(:, i) = v / norm(v);

end



% Output the orthonormal vectors (columns of Q)

disp('Orthonormal Vectors (columns of Q):');

disp(Q);



% Plot the orthonormal vectors (for 3D case in this example)

figure;

hold on;

axis equal;

grid on;

xlabel('X');

ylabel('Y');

zlabel('Z');

title('Orthonormal Vectors');



% Plot each orthonormal vector

for i = 1:n

    quiver3(0, 0, 0, Q(1, i), Q(2, i), Q(3, i), 0, 'LineWidth', 2, 'DisplayName',
['v' num2str(i)]);

end



% Add legend

legend show;



hold off;
```

****

output:

```
>> orthogonal
orthogonal vectors are:
    1.0000    0.5000   -0.6667
    1.0000   -0.5000    0.6667
         0    1.0000    0.6667

Orthonormal Vectors (columns of Q):
    0.7071    0.4082   -0.5774
    0.7071   -0.4082    0.5774
         0    0.8165    0.5774

``
```

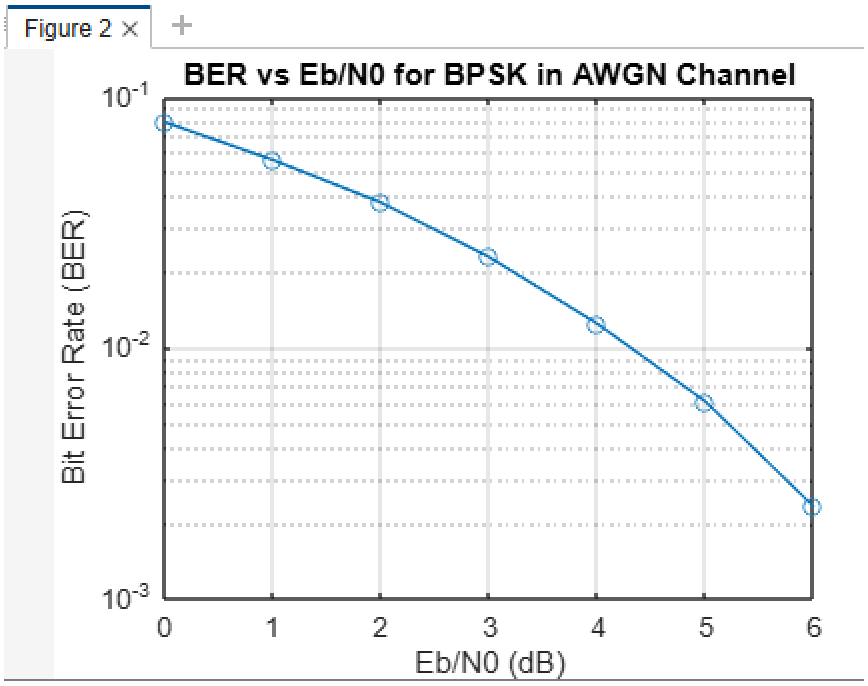


****

Simulation bit error rate for AWGN channel

```
% Parameters
N = 1e5; % Number of bits
EbN0_dB = 0:6; % Eb/N0 range in dB
M = 2; % Binary modulation (BPSK)
k = log2(M); % Bits per symbol

% Generate random binary data
data = randi([0 1], N, 1);

% BPSK modulation
txSignal = 2*data - 1; % Map 0 -> -1, 1 -> 1

% Rectangular pulse shaping
pulseShape = ones(1, 1); % Rectangular pulse
txSignal = conv(txSignal, pulseShape, 'same');

% Initialize BER array
BER = zeros(length(EbN0_dB), 1);

for i = 1:length(EbN0_dB)
    % Calculate noise variance
    EbN0 = 10^(EbN0_dB(i)/10);
    noiseVar = 1/(2*EbN0);

    % Generate AWGN noise
    noise = sqrt(noiseVar) * randn(size(txSignal));

    % Received signal
    rxSignal = txSignal + noise;

    % BPSK demodulation
    rxData = rxSignal > 0;

    % Calculate BER
    BER(i) = sum(data ~= rxData) / N;
end

% Plot BER vs Eb/N0
figure;
semilogy(EbN0_dB, BER, 'o-');
xlabel('Eb/N0 (dB)');
ylabel('Bit Error Rate (BER)');
title('BER vs Eb/N0 for BPSK in AWGN Channel');
grid on;
```

output:



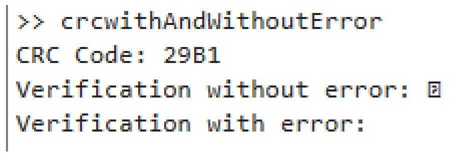
**************************************AWGN END**********************************

CRC:

```
function crc_code = crc_ccitt(data)
    % CRC-CCITT polynomial
    poly = hex2dec('1021');
    crc = uint16(0xFFFF); % Initial value

    for i = 1:length(data)
        crc = bitxor(crc, bitshift(uint16(data(i)), 8));
        for j = 1:8
            if bitand(crc, 32768) % 0x8000
                crc = bitxor(bitshift(crc, 1), poly);
            else
                crc = bitshift(crc, 1);
            end
        end
    end

    crc_code = bitand(crc, 65535); % 0xFFFF
end

% Example data
data = uint8('123456789');

crc_code = crc_ccitt(data);
```

```
fprintf('CRC Code: %04X\n', crc_code);

% Verification without error
data_no_error = uint8('123456789');
crc_no_error = crc_ccitt(data_no_error);
fprintf('Verification without error: %s\n', isequal(crc_code, crc_no_error));

% Verification with error
data_with_error = uint8('123456780');
crc_with_error = crc_ccitt(data_with_error);
fprintf('Verification with error: %s\n', isequal(crc_code, crc_with_error));
```

output:



```
>> crcwithAndWithoutError
CRC Code: 29B1
Verification without error: □
Verification with error:
```

End********************

16 QAM Modulation:

```
% MATLAB R2015 code for 16-QAM modulation and constellation diagram

% Parameters
M = 16;                 % Modulation order (16-QAM)
k = log2(M);            % Bits per symbol
nSymbols = 1000;        % Number of symbols

% Generate random data
data = randi([0 M-1], nSymbols, 1);

% QAM Modulation (normalize symbols for unit average power)
modulatedSignal = qammod(data, M);       % Perform QAM modulation
modulatedSignal = modulatedSignal / sqrt(mean(abs(modulatedSignal).^2)); %
Normalize to unit power

% Constellation Plot
scatterplot(modulatedSignal);
title('16-QAM Constellation (R2015)');
xlabel('In-Phase');
ylabel('Quadrature');

% Adding grid and axis labels for better visualization
grid on;
axis([-1.5 1.5 -1.5 1.5]);
```

output:

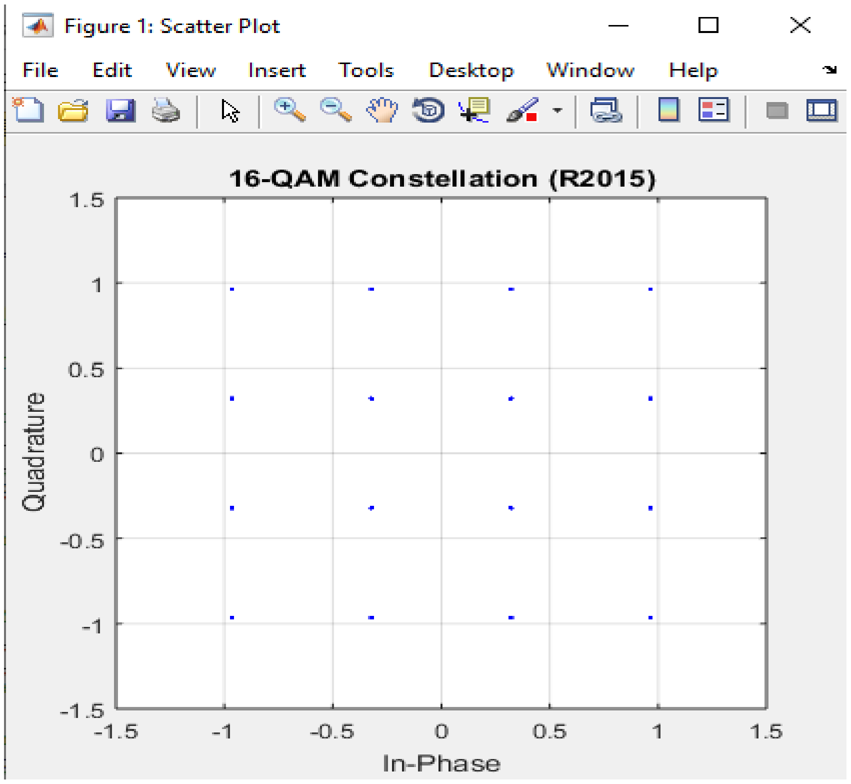


Figure 1: Scatter Plot
File Edit View Insert Tools Desktop Window Help
16-QAM Constellation (R2015)
Quadrature
In-Phase
1.5
1
0.5
0
-0.5
-1
-1.5