Exp :05

```
% Define the set of vectors
A = [1 0 0; 1 1 1; 0 0 1]';

% Perform Gram-Schmidt Orthogonalization
Q = zeros(size(A));
for j = 1:size(A, 2)
    v = A(:, j);
    for i = 1:j-1
        v = v - (Q(:, i)' * v) * Q(:, i);
    end
    Q(:, j) = v / norm(v);
end
disp('Orthonormal basis Q:');
disp(Q);

% Plot the orthonormal vectors
figure;
hold on;
for i = 1:size(Q, 2)
    quiver3(0, 0, 0, Q(1,i), Q(2,i), Q(3,i), 'LineWidth', 2); % Plot each vector
end
xlabel('X-axis'); ylabel('Y-axis'); zlabel('Z-axis');
title('Orthonormal Vectors');
grid on;
hold off;
```

Exp: 06

```
% Parameters
N = 10000;  % Number of bits
Eb_N0_dB = 0:5:20;  % Eb/N0 range in dB
Tb = 1;  % Bit duration

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

% Generate rectangular pulse
t = 0:0.01:Tb;
pulse = ones(size(t));

% Modulate bits using rectangular pulse
signal = zeros(size(t, 2)*N, 1);
for i = 1:N
    if bits(i) == 1
        signal((i-1)*size(t, 2)+1:i*size(t, 2)) = pulse;
    end
end

% Add AWGN noise
BER = zeros(size(Eb_N0_dB));
for i = 1:length(Eb_N0_dB)
    Eb_N0 = 10^(Eb_N0_dB(i)/10);
    noise = sqrt(1/(2*Eb_N0))*randn(size(signal));
    received_signal = signal + noise;

    % Matched filter receiver
    matched_filter = pulse(end:-1:1);  % Time-reversed pulse
    filtered_signal = conv(received_signal, matched_filter, 'same');

    % Detect bits
    detected_bits = zeros(size(bits));
    for j = 1:N
        if filtered_signal(j*size(t, 2)) > 0.5
```

```
            detected_bits(j) = 1;
        end
    end

    % Calculate BER
    errors = sum(bits ~= detected_bits);
    BER(i) = errors/N;
end

% Plot BER vs Eb/N0
figure;
semilogy(Eb_N0_dB, BER);
xlabel('Eb/N0 (dB)');
ylabel('BER');
title('BER Performance of Binary Baseband Signaling');
grid on;
```

Exp:08

```
% Parameters
M = 16;  % Number of symbols
N = 1000;  % Number of bits

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

% Convert bits to symbols
symbols = zeros(1, N/4);
for i = 1:N/4
   temp = bits(4*i-3:4*i);
   symbols(i) = (2*temp(1)-1) + 1j*(2*temp(2)-1) + 2*(2*temp(3)-1) + 2j*(2*temp(4)-1);
end

% Plot 16-QAM constellation
figure;
scatter(real(symbols), imag(symbols),'bo');
xlabel('In-phase');
ylabel('Quadrature');
title('16-QAM Constellation');
grid on;
```

exp:10

```
% Hamming Encoding
% Define the data to encode
data = [1 0 1 0];
% Calculate the parity bits
p1 = mod(data(1) + data(3) + data(4), 2);
p2 = mod(data(1) + data(2) + data(4), 2);
p3 = mod(data(1) + data(2) + data(3), 2);

% Create the encoded data
encoded_data = [p1 p2 data(1) p3 data(2) data(3) data(4)];

disp('Encoded Data:');
disp(encoded_data);

% Hamming Decoding
% Define the encoded data with error
encoded_data = [1 0 1 0 1 0 1];
% Calculate the syndrome
s1 = mod(encoded_data(1) + encoded_data(3) + encoded_data(5) + encoded_data(7), 2);
s2 = mod(encoded_data(2) + encoded_data(3) + encoded_data(6) + encoded_data(7), 2);
s3 = mod(encoded_data(4) + encoded_data(5) + encoded_data(6) + encoded_data(7), 2);

% Determine the error location
error_location = bin2dec([num2str(s1) num2str(s2) num2str(s3)]);
% Correct the error
if error_location ~= 0
   encoded_data(error_location) = mod(encoded_data(error_location) + 1, 2);
end

% Extract the decoded data
decoded_data = encoded_data([3 5 6 7]);

disp('Decoded Data:');
disp(decoded_data);
```

exp:12

```
msg = [1 0 1 1 0 1 0 0];

% Define constraint length and generator polynomial

constraint_length = 3;

generator_polynomials = [7 5];

% Create trellis structure for the convolutional encoder

trellis = poly2trellis(constraint_length, generator_polynomials);

% Encode the message using convolutional encoder

encoded_msg = convenc(msg, trellis);

% Simulate noise by flipping a bit in the encoded message

encoded_msg_noisy = encoded_msg;

encoded_msg_noisy(4) = ~encoded_msg_noisy(4); % Flip the 4th bit to simulate noise

% Perform Viterbi decoding on the noisy message

traceback_length = 5;

decoded_msg = vitdec(encoded_msg_noisy, trellis, traceback_length, 'trunc', 'hard');

% Display results

disp('Original Message:');

disp(msg);

disp('Encoded Message:');

disp(encoded_msg);

disp('Noisy Encoded Message (with bit flip):');

disp(encoded_msg_noisy);

disp('Decoded Message:');

disp(decoded_msg);
```

Exp:07

```
data=[0  1 0 1 1 1 0 0 1 1]; % information


figure(1)

stem(data, 'linewidth',3), grid on;

title('  Information before Transmiting ');

axis([ 0 11 0 1.5]);

data_NZR=2*data-1; % Data Represented at NZR form for QPSK modulation

s_p_data=reshape(data_NZR,2,length(data)/2);  % S/P convertion of data


br=10.^6; %Let us transmission bit rate  1000000

f=br; % minimum carrier frequency

T=1/br; % bit duration

t=T/99:T/99:T; % Time vector for one bit information


%  QPSK modulation

y=[];

y_in=[];

y_qd=[];

for(i=1:length(data)/2)

   y1=s_p_data(1,i)*cos(2*pi*f*t); % inphase component

   y2=s_p_data(2,i)*sin(2*pi*f*t) ;% Quadrature component

   y_in=[y_in y1]; % inphase signal vector

   y_qd=[y_qd y2]; %quadrature signal vector

   y=[y y1+y2]; % modulated signal vector

end

Tx_sig=y; % transmitting signal after modulation

tt=T/99:T/99:(T*length(data))/2;


figure(2)
```

```
subplot(3,1,1);
plot(tt,y_in,'linewidth',3), grid on;
title(' wave form for inphase component in QPSK modulation ');
xlabel('time(sec)');
ylabel(' amplitude(volt0');

subplot(3,1,2);
plot(tt,y_qd,'linewidth',3), grid on;
title(' wave form for Quadrature component in QPSK modulation ');
xlabel('time(sec)');
ylabel(' amplitude(volt0');


subplot(3,1,3);
plot(tt,Tx_sig,'r','linewidth',3), grid on;
title('QPSK modulated signal (sum of inphase and Quadrature phase signal)');
xlabel('time(sec)');
ylabel(' amplitude(volt0');
```

Exp:09

```
p = [0.4 0.3 0.2 0.1];
n=length(p);
symbols=[1:n];
[dict,avglen]=huffmandict(symbols,p);
temp=dict;
t=dict(:,2);
for i=1:length(temp)
   temp{i,2}=num2str(temp{i,2});
end
disp('The huffman code dict:');
disp(temp)
fprintf('Enter the symbols between 1 to %d in[]',n);
sym=input(':')
encod=huffmanenco(sym,dict);
disp('The encoded output:');
disp(encod);
bits=input('Enter the bit stream in[];');
decod=huffmandeco(bits,dict);
disp('The symbols are:');
disp(decod);
H=0;
Z=0;
for(k=1:n)
   H=H+(p(k)*log2(1/p(k)));
 end
fprintf(1,'Entropy is %f bits',H);
N=H/avglen;
fprintf('\n Efficiency is:%f',N);
for(r=1:n)
  l(r)=length(t{r});
end
m=max(l)
s=min(l)
v=m-s;
fprintf('the variance is:%d',v);
```