Output & Interpretation:

* The BER vs SNR plot will show a decreasing trend in the bit error as the SNR increases, which is expected. A higher SNR means the signal is stronger compared to noise, leading to fewer errors in detection.
* The use of a matched filter maximises the signal-to noise (SNR) ratio at the receiver, which improves the detection of transmitted bits, especially at lower SNRs.

Example plot description:

X-axis : SNR in dB
Y-axis : Bit error rate (BER), shown on a logarithmic scale.

As SNR increases, the BER will decrease due to the reduction in noise impact, resulting in a cleaner signal & more accurate bit recovery.

Experiment No: 6

Aim: Simulation of binary baseband signals using a rectangular pulse & estimate the BER for AWGN channel using matched filter receiver.

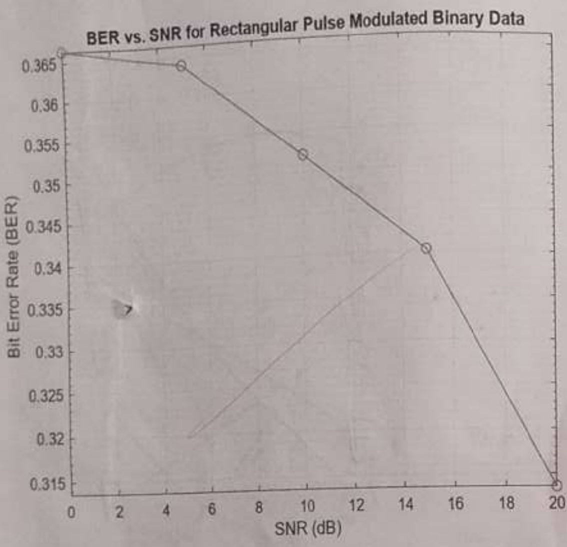
Theory: In digital communication systems, binary baseband signals are used to transmit binary data (0s & 1s) over a communication channel. One of the simplest way to represent binary data is using a rectangular pulse. A rectangular pulse for bit '1' or '0' is transmitted over a fixed duration, with the amplitude of pulse being constant during that period.

Code:

```
N = 1e4;
SNR_dB = 0:5:20;
pulse_width = 1;
data = randi([0,1],N,1);
t = 0:0.01:pulse_width;
rect_pulse = ones(size(t));
BER = zeros(length(SNR_dB),1);
for snr_idx = 1:length(SNR_dB)
    tx_signal = [];
    for i = 1:N
        if data(i)==1
            tx_signal = [tx_signal; ~~zeros(size(~~rect_pulse)];
        else
            tx_signal = [tx_signal; zeros(size(rect_pulse))];
        end
    end
```

EXPERIMENT 6

2KE22EC095



```
SNR = 10^(SNR_dB(snr_idx)/10);
noice power = 1/(2*SNR);
noise=sqrt(noise power)* randn(length(tx_signal), 1);
rx_signal= tx_signal + noise;
matched_filter = rect_pulse;
filtered_signal= conv(rx_signal, matched_~~p~~filter, 'same');
sample interval= round (length (filtered_signal) /N);
sampled_signal = filtered_signal (1:sample_interval: end);
estimated_bits = sampled_signal >0.5;
num_errors = sum(estimated_bits ~= data);
BER (snr_idx) = num_errors/N;
end
figure;
semilogy( SNR_dB, BER, 'b-o');
grid on;
xlabel ('SNR(dB)');
ylabel('Bit Error Rate(BER)');
title('BER vs SNR for Rectangular Pulse Modulated
        Binary Data');
```

* Conclusion:-

Simulation of binary baseband signals ~~&~~ using a rectangular pulse & estimation of BER for AWGN channel using matched filter has performed & verified.

Output:

Resulting plots:

1. Binary input sequence: The first plot shows the binary i/p sequence as pulses.

2. Odd & even components: The second & 3rd plots show the sine (odd) & cosine (even) components of QPSK signal.

3. QPSK modulated signal: The fourth plot displays the final QPSK signal.

4. QPSK constellation: The fifth plot shows the QPSK constellation with the four possible constellation points.

Experiment. No:- 7

Aim: Perform the QPSK modulation and demodulation Display the signal and its constellation.

Theory:- Quadrature Phase Shift Keying is a digital modulation technique used to transmit data efficiently over communication channels. It is a type of phase modulation where data is represented by four distinct phase shifts a carrier signal QPSK uses two bits per second meaning each symbol carries two bits of information. In QPSK, the carrier signal is modulated by varying its phase. The four possible phase shifts (0°, 90°, 180° & 270°) correspond to different pairs of bits.

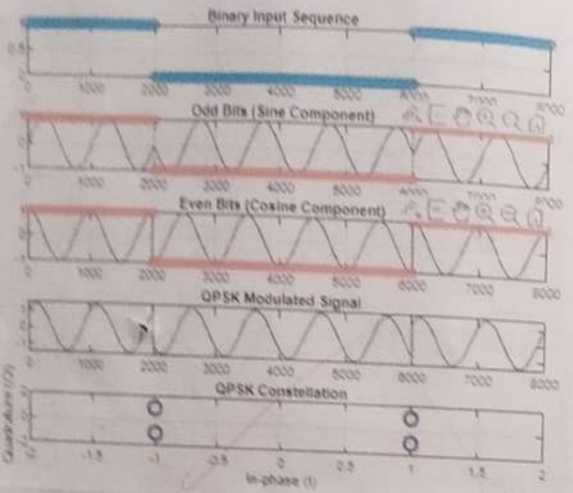
00 → 0°, 01 → 90°, 10 → 180°, 11 → 270°

Code:

```
clc; clear all; close all;
bit seq = [1 1 0 0 0 0 1 1];
N=length(bit_seq);
fc=1; % carrier frequency
t = 0:0.001:2 ; % time vector
b = []; % input bit sequence as a waveform
qpsk1 = []; % QPSK signal
bec = []; % even bit cosine waveform
bes = []; % odd bit sine waveform
b_o = []; % odd bit stream
b_e = []; % even bit stream
bit_e = []; % even bit stream (for plotting)
bit_o = []; % odd bit stream (for plotting)
for i=1:N
    bx = bit_seq(i)* ones(1,1000); %Generate pulse for each bit
```

EXPERIMENT 7

2KE22EC095



```
    b = [b, bx];
end
for i = 1:N
  if bit_seq(i) = = 0
     bit_seq(i) = -1;
  end
  if mod(i,2) = = 0
     e_bit = bit_seq(i);
     b_e = [b_e, e_bit];
  else
     o_bit = bit_seq(i);
     b_o = [b_o, o_bit];
  end
end
for i = 1: N/2
     be_c = (b_e(i)* cos(2*pi*fc*t));
     bo_s = (b_o(i)* sin(2*pi*fc*t));
     q = be_c + bo_s;
     even = b_e(i)* ones(1,2000);
     odd = b_o(i)* ones(1,2000);
     bit_e = [bit_e, even];
     bit_o = [bit_o, odd];
     qpsk1 = [qpsk1, q];
     bec = [bec, be_c];
     bes = [bes, bo_c];
foend
figure('name', 'QPSK modulation');
subplot(5,1,1);
plot(b,'o');
grid on;
axis([0 N*1000 0 1]);
```

```
title('Binary Input Sequence');
subplot(5,1,2);
plot(bes);
hold on;
plot(bit_o,'rs:');
grid on;
axis([0 N*1000-1 1]);
title('Odd Bits(Sine Component)');
subplot(5,1,3);
plot(bec);
hold on;
plot(bit_e,'rs:');
grid on;
axis([0 N*1000-1 1]);
title('Even Bits(cosine Component)');
subplot(5,1,4);
plot(qpsk1);
axis([0 N*1000-1.5 1.5]);
title('QPSK Modulated Signal');
subplot(5,1,5);
constellation.[1+1j,-1+1j ,-1,-1j, 1-1j];
plot(real(constellation), imag(constellation),'bo','MarkerSize',
     8,'LineWidth',2);
grid on;
axis([-2 2 -2 2]);
title('QPSK Constellation');
xlabel('In-phase(I)');
ylabel('Quadrature (Q)');
```

[illegible]

Conclusion: QPSK modulation & demodulation & display of signal & its constellation has bee performed.

Output plots:

1. 16-QAM constellation Diagram (Before noise):

This plot shows the ideal constellation of the 16-QAM symbols with the points at their expected locations. The real part (I) is plotted on the y-axis.

Plot description:

Blue circles represent the 16-QAM symbols. The symbols are placed at positions like (-3, -3), (-3, -1), (3, 3), etc., as defined by the qam_levels.

Expected plot output:

16 well-seperated points forming the 16-QAM constellation
16-QAM constellation diagram (after noise with SNR = 20dB
This plot shows how the constellation looks after noise is added with an SNR of 20 dB.

Experiment No :- 8

Aim: General 16-QAM modulation & obtain the QAM constellation.

Theory: Quadrature Amplitude Modulation (QAM) is a modulation technique that combines both amplitude modulation & phase modulation (PM) to encode data in digital communication systems. It uses two carrier signals that are 90° out of phase with each other, known as in-phase (I) & quadrature (Q) components. By variation the amplitude of both components simultaneously, multiple distinct symbols can be created, each representing a unique combination of I & Q values.

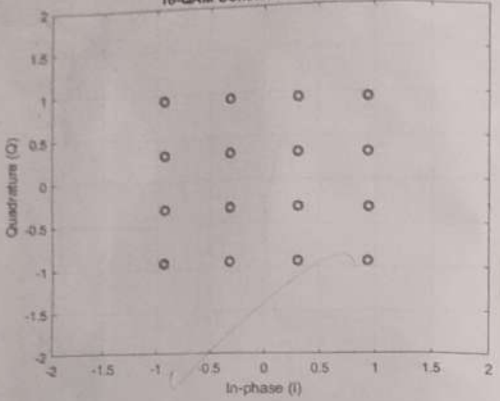
Code:

```
clc; clear; close all;
N=1000;
M=16;
k=log2(M);
SNR_dB=20;
data_bits=randi([0,1],1,N);
symbols_bits=reshape(data bits, k, []).');
qam_levels=[-3 -1 1 3];
qam symbols=zeros(1, size(symbols_bits, 1));
for i=1:size(symbols_bits, 1)
    I_bits=bi2de(symbols_bits}
    I_val=qam_levels(I_bits+1);
    Q_bits=bi2de(symbols_bits(i, 3:4), 'left-msb');
    Q_val=qam_levels(Q_bits+1);
    qam_symbols(i)=I_val+1i*Q_val;
end
qam_symbols=qam_symbols/sqrt(mean(abs(qam_symbols).^2));
```

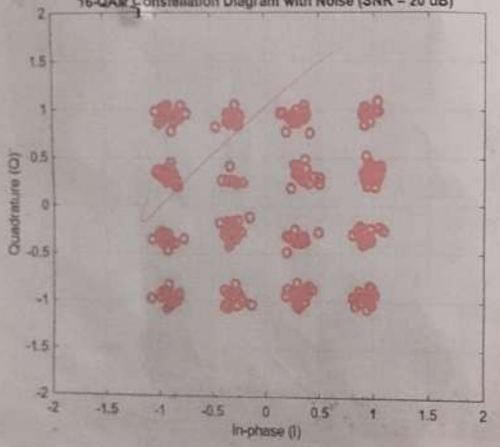
Experiment 8

2KE22EC095

16-QAM Constellation Diagram



16-QAM Constellation Diagram with Noise (SNR = 20 dB)



```
figure;
plot(real(qam symbols), imag (qam_symbols), 'bo','MarkerSize',
     6, 'LineWidth', 2);
xlabel('In-Phase(I)');
ylabel('Quadrature(Q)');
title('16-QAM Constellation Diagram');
grid on;
axis([-2 2 -2 2]);
rx_signal = awgn(qam_symbols, SNR_dB,'measured');
figure;
plot(real(rx_signal), imag (rx_signal),'ro', 'MarkerSize',6,
     'LineWidth', 2);
xlabel ('In-phase(I)');
ylabel ('Quadrature (Q)');
title(['16-QAM Constellation Diagram with Noise (SNR = 'num2str
     (SNR_dB)', dB']);
grid on;
axis([-2 2 -2 2]);
```

* Conclusion:-

Generation of 16 QAM & QAM constellation has been performed & verified.

[illegible]

Comple 1:

Input:

```
Enter the probabilities : [0.4  0.3  0.2  0.1]
Enter the symbols between 1 to 4 in[] : [1 2 3 4]
Enter the bit stream in [] : [1 0 1 1 0 0 1 0]
```

Huffman code dictionary

```
The huffman code dict :
{
  [1] '0'
  [2] '10'
  [3] '11'
  [4] '010'
}
The huffman codes will be:
    symbol 1: 0
    symbol 2: 10
    symbol 3: 11
    symbol 4: 010
The encoded output: 01011010
The symbols are : [1 2 3 4]
The bitstream [1 0 1 1 0 0 1 0] will be decoded back to the
original sequence of symbols [1, 2, 3, 4].
Entropy is 1.846 bits.
Efficiency is : 0.921 bits.
The variance is : 2
```

Experiment No.: 7

Aim: Encoding & Decoding of Huffman code.

Theory: Huffman code is a widely used algorithm for data compression, designed to minimize the average length of codes assigned to characters based on their frequencies of occurrence. It is a type of prefix coding, where no code is a prefix of another, ensuring that the codeing is uniquely decodable.

Code:

```
clc;
p=input('Enter the probabilities : ');
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(':');
encode=huffmanenco(sym,dict);
disp('The encoded output: ');
disp(encode);
bits=input('Enter the bit stream in[] : ');
decode=huffmandeco(bits,dict);
```

EXPERIMENT 9

2KE22EC095

```
The huffman code dict:

    {[1]}    {'1'       }
    {[2]}    {'0  1'    }
    {[3]}    {'0  0  0'}
    {[4]}    {'0  0  1'}

Enter the symbols between 1 to 4 in[]

sym = 1:4
     1     2     3     4

The encoded output:

     1     0     1     0     0     0     0     0     1

The symbols are:

     1     2     3     4

Entropy is 1.846439 bits

 Efficiency is:0.971810

m = 3

s = 1

the variance is:2
```

```
disp ('The symbols are:');
disp (decod);
H=0;
l=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);
```

~~*Constell~~

* Conclusion:-
Encoding & decoding of Huffman code has been performed & verified.

[illegible]

Sample Input & outputs:

1) Input data: [1 0 0 1] (4-bits)

2) Encoded data: After encoding with the generator matrix, the result will be a 7-bit string.

3) Simulated error: The 5th bit is flipped to simulate an error.

4) Decoded error: The received data is decoded, errors are detected & corrected & the original 4-bit data [1 0 0 1] is recovered.

Experiment No: 10

Aim: Encoding & decoding of binary data using a Hamming code.

Theory: The 7, 4 hamming code is an error detecting & error correcting code that encodes 4 data bits into 7 bits by adding 3 parity bits. It is designed to detect & correct single bit errors in the transmitted data.

Code:

```
function hamming74()
    data = [1 0 1 0];
    encoded_data = hamming74_encode(data);
    disp('Encoded Data');
    disp(encoded_data);
end
function encoded_data = hamming74_encode(data)
    if length(data) ~= 4
        error('Input data must be 4 bits long');
    end
    G = [1 0 0 0 1 1 1;
         0 1 0 0 1 0 1;
         0 0 1 0 0 1 1;
         0 0 0 1 1 1 0;
    encoded_data = mod(data * G, 2);
end
function decoded_data = hamming74_decode(received_data)
    if length(received_data) ~= 7
        error('Received data must be 7 bits long');
    end
```

EXPERIMENT 10

2KE22EC095

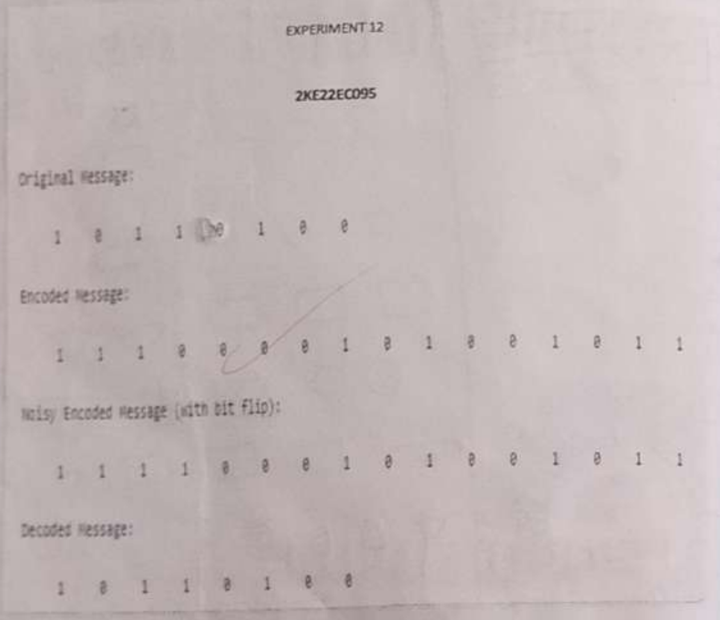
```
Original Data:

     1      0      1      1

Encoded Data:

     1      0      1      1      0      1      0

Received Data with Error:

     1      0      0      1      0      1      0

Decoded Data:

     1      0      0      1

Error detected at position: 0
Error detected at position: 1
Error detected at position: 1
```

```
H = [1 1 1 0 1 0 0;
     1 1 0 1 0 1 0;
     1 0 1 1 0 0 1];
syndrome = mod(received_data' * H', 2);
if any(syndrome) == 1
    error_position = bi2de(syndrome, 'left-msb') + 1;
    disp(['Error detected at position: ', num2str(error_
                                              position)]);
    received_data(error_position) = mod(received_data
                        (error_position) + 1, 2);
else
    disp('No error detected');
end
decoded_data = received_data(1:4);
end.
```

*Conclusion:
Encoding & decoding of binary data using a Hamming code has been performed & verified.

[illegible]

Sample outputs:
After running the code, you should see o/p similar to following:
```
original messages: 1 0 1 1 0 1 0 0
encoded message: 1 0 1 1 1 0 1 1 1 0 0
noisy encoded msg: 1 0 1 0 1 0 1 1 1 0 0
decoded message: 1 0 1 1 0 1 0 0.
```



EXPERIMENT 12

2KE22EC095

```
Original Message:

    1    0    1    1    0    1    0    0

Encoded Message:

    1    1    1    0    0    0    0    1    0    1    0    0    1    0    1    1

Noisy Encoded Message (with bit flip):

    1    1    1    1    0    0    0    1    0    1    0    0    1    0    1    1

Decoded Message:

    1    0    1    1    0    1    0    0
```

Experiment No :- 12

Aim: Encoding & Decoding of convolution code.

Theory: Convolution coding is a method of error correction where each bit of the input message is transformed into multiple output bits based on a set of generator polynomials. It is widely used in communication systems to improve the reliability of data transmission over noisy channels.

Code:
```
clc; clear; close all;
msg = [1 0 1 1 0 1 0 0];
constraint_length = 3;
generator_polynomials = [7 5];
trellis = poly2trellis(constraint_length; generator_polynomials);
encoded_msg = convenc(msg, trellis);
encoded_msg_noisy = encoded_msg;
encoded_msg_noisy(4) = ~encoded_msg_noisy(4);
traceback_length = 5;
decoded_msg = vitdec(encoded_msg_noisy, trellis, traceback_length, 'trunc', 'hard');
disp('original message:'); disp(msg);
disp('Encoded message:'); disp(decoded_msg);
disp('Noisy Encoded Message (with bit flip):');
disp(encoded_msg_noisy);
disp('Decoded Message:'); disp(decoded_msg);
```

Conclusion: Encoding & decoding of convolution code has been performed & verified.