Part 1 Noise Free:

We’re going to test if our Tx and Rx are working correctly before adding any noise:

The modulation techniques tested are:

BPSK QPSK 8PSK 16-QAM

So we made a function based-code:

Tx Mapper:

Code:

```
function [Tx_Vector, Table] = mapper(bits, mod_type)
    % MAPPER Digital modulation mapper with explicit symbol table
    % Inputs:
    %   bits     - Binary input array (row vector)
    %   mod_type - 'BPSK', 'QPSK', '8PSK', 'BFSK', '16QAM'
    % Outputs:
    %   Tx_Vector - Complex modulated symbols
    %   Table     - Constellation points (M-ary symbols)

    % Ensure bits are row vector
    bits = bits(:)';

    % Define modulation parameters
    switch upper(mod_type)
        case 'BPSK'
            n = 1;  % bits per symbol
            M = 2;   % constellation size
            Table = [-1, 1];  % BPSK symbols (real)

        case 'QPSK'
            n = 2;
            M = 4;
            Table = [-1-1j, -1+1j, 1-1j, 1+1j];  % QPSK symbols

        case '8PSK'
            n = 3;
            M = 8;
            angles =[0, 1, 3, 2, 7, 6, 4, 5]*pi/4;  % Gray-coded 8PSK
            Table = exp(1j*angles);

        case 'BFSK'
            error('BFSK requires time-domain implementation (see alternative)');

        case '16-QAM'
            n = 4;
            M = 16;
            % 16-QAM with unit average power (normalized)
            Table = [-3-3j, -3-1j, -3+3j, -3+1j, ...
                     -1-3j, -1-1j, -1+3j, -1+1j, ...
                      3-3j,  3-1j,  3+3j,  3+1j, ...
                      1-3j,  1-1j,  1+3j,  1+1j];

        otherwise
            error('Unsupported modulation type: %s', mod_type);
    end

    % Pad bits if not multiple of n
    if mod(length(bits), n) ~= 0
        bits = [bits zeros(1, n - mod(length(bits), n))];
    end

    % Reshape into n-bit groups
    bit_groups = reshape(bits, n, [])';

    % Convert to decimal symbols (0 to M-1)
    Array_symbol = bi2de(bit_groups, 'left-msb') + 1;  % MATLAB uses 1-based indexing

    % Map to constellation points
    Tx_Vector = Table(Array_symbol);
end
```

For the Tx mapper, we just convert the bits into decimal values to index it with symbol table, which is grey-coded, from the complex constellations:

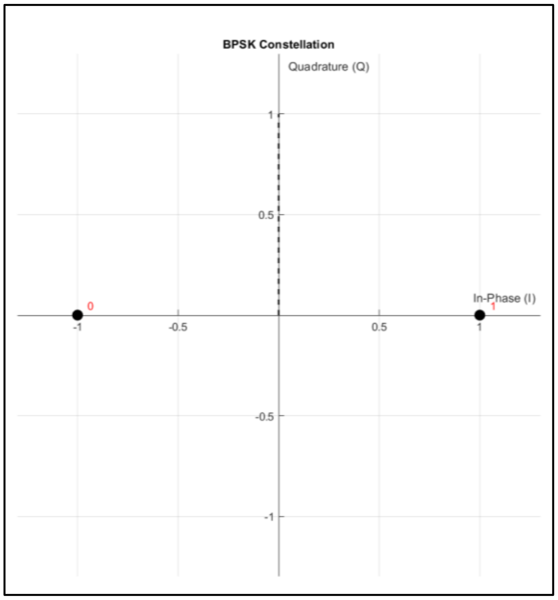

Figure 1 BPSK constellation

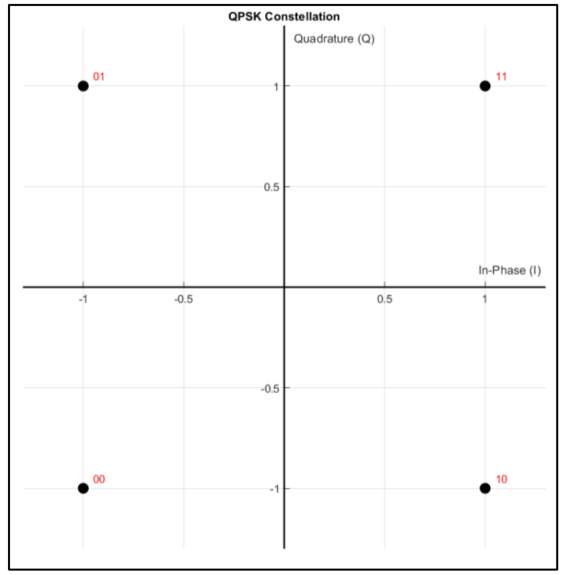

Figure 2 QPSK constellation

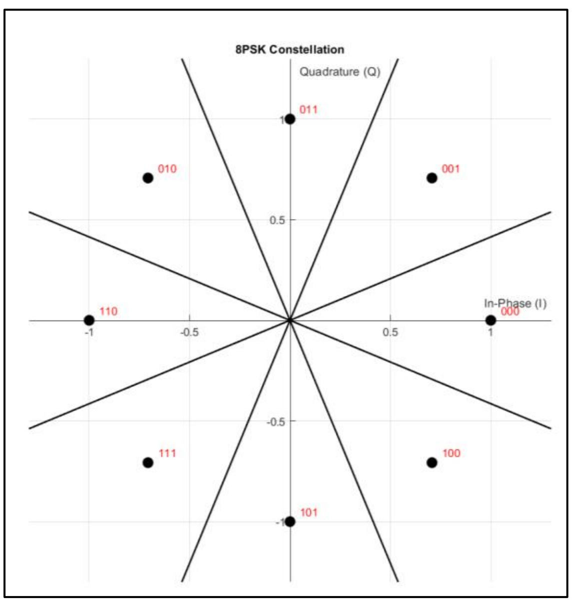

Figure 4 8PSK constellation

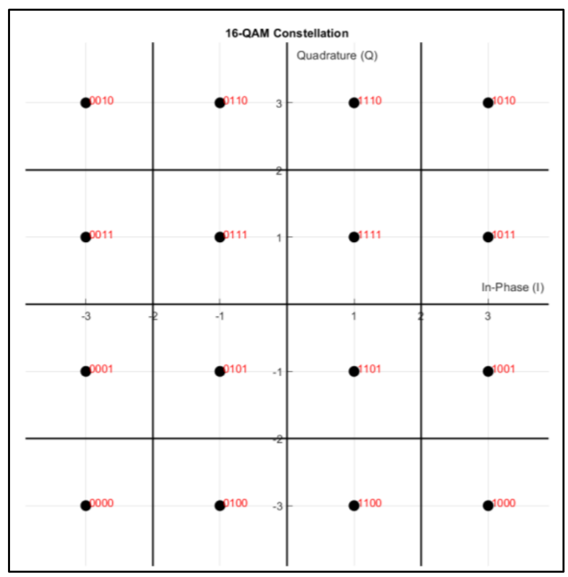

Figure 3 16-QAM constellation

As shown in the figures 1, 2, 3, and 4, we just make some linear algebra operations. As the I is the real part and Q is the imaginary part

$X_{BB} = X_I + j\, X_Q$

Rx Demapper:

```
function [received_bits] = demapper(received_symbols, mod_type)
    % DEMAPPER Digital demodulation demapper
    % Inputs:
    %   received_symbols - Complex received symbols
    %   mod_type         - Modulation type ('BPSK', 'QPSK', etc.)
    % Output:
    %   received_bits    - Demodulated bit stream

    % Get constellation table from mapper
    [~, Table] = mapper([1], mod_type);

    % Determine bits per symbol
    switch upper(mod_type)
        case 'BPSK'
            n = 1;
        case 'QPSK'
            n = 2;
        case '8PSK'
            n = 3;
        case {'16QAM', '16-QAM'}
            n = 4;
        otherwise
            error('Unsupported modulation type');
    end

    % Initialize output bits
    received_bits = zeros(1, length(received_symbols)*n);

    % Demodulate each symbol
    for i = 1:length(received_symbols)
        % Find nearest constellation point
        [~, idx] = min(abs(received_symbols(i) - Table));

        % Convert to binary (0-based index)
        bin_str = dec2bin(idx-1, n);

        % Store bits
        received_bits((i-1)*n+1:i*n) = bin_str - '0';
    end
end
```

For the Rx demapper, we just make inverse Tx mapper operation.

We check the nearest table symbol to the Rx symbol and get it’s index with this index we convert it into bits.

Simulation:

Now we will try a small noise free simulation to make sure that the Rx and Tx runs properly

Code:

```
clear; clc; close all;

%--------Part 1----------
% ========================
% Simulation Parameters
% ========================
bits_Num = 48;                                    % Number of bits to transmit
mod_types = {'BPSK', 'QPSK', '8PSK', '16-QAM'}; % Cell array of modulation types

% Generate random bits (same for all modulations for fair comparison)
Tx_bits = randi([0 1], 1, bits_Num);

% Loop through all modulation types
for mod_idx = 1:length(mod_types)
    mod_type = mod_types{mod_idx};

    fprintf('\n=== Testing %s Modulation ===\n', mod_type);

    % ========================
    % 1. Mapping (Modulation)
    % ========================
    [tx_symbols, constellation] = mapper(Tx_bits, mod_type);

    % ========================
    % 2. Display Constellation
    % ========================
    drawConstellation(constellation, mod_type);
    title(sprintf('%s Constellation', mod_type));

    % ========================
    % 3. Add Channel Noise
    % ========================
    %rx_symbols = awgn(tx_symbols, SNR_dB, 'measured');
    rx_symbols = tx_symbols;
    % ========================
    % 4. Demapping (Demodulation)
    % ========================
    Rx_bits = demapper(rx_symbols, mod_type);

    % ========================
    % 5. Display Results
    % ========================
    % Calculate BER
    [BER, bit_errors] = calculateBER(Tx_bits, Rx_bits);

    % Display input/output comparison
    fprintf('Original bits:\n');
    disp(reshape(Tx_bits, 16, [])'); % Display in 16-bit groups

    fprintf('Received bits:\n');
    disp(reshape(Rx_bits(1:bits_Num), 16, [])'); % Display in 16-bit groups

    fprintf('Bit errors: %d\n', bit_errors);
    fprintf('BER: %.2e\n\n', BER);

end
```

In the simulation we'll generate random bits and modulate it with each type and check if there's an error

Results:

```
=== Testing BPSK Modulation ===
```

```
=== Testing 16-QAM Modulation ===
Bit errors: 0
BER: 0.00e+00
Original bits:
     0     1     1     0     1     1     0     1     0     1     0     1     1     0     1     0
     1     1     1     1     1     1     1     0     1     1     1     0     1     0     0     1
     1     0     1     0     1     0     1     0     0     1     0     1     1     1     0     0

Received bits:
     0     1     1     0     1     1     0     1     0     1     0     1     1     0     1     0
     1     1     1     1     1     1     1     0     1     1     1     0     1     0     0     1
     1     0     1     0     1     0     1     0     0     1     0     1     1     1     0     0

Bit errors: 0
BER: 0.00e+00
```

Figure 7 BPSK Test

Figure 8 16-QAM Test

```
=== Testing QPSK Modulation ===
Bit errors: 0
BER: 0.00e+00
Original bits:
     0     1     1     0     1     1     0     1     0     1     0     1     1     0     1     0
     1     1     1     1     1     1     1     0     1     1     1     0     1     0     0     1
     1     0     1     0     1     0     1     0     0     1     0     1     1     1     0     0

Received bits:
     0     1     1     0     1     1     0     1     0     1     0     1     1     0     1     0
     1     1     1     1     1     1     1     0     1     1     1     0     1     0     0     1
     1     0     1     0     1     0     1     0     0     1     0     1     1     1     0     0

Bit errors: 0
BER: 0.00e+00
```

Figure 6 QPSK Test

```
=== Testing 8PSK Modulation ===
Bit errors: 0
BER: 0.00e+00
Original bits:
     0     1     1     0     1     1     0     1     0     1     0     1     1     0     1     0
     1     1     1     1     1     1     1     0     1     1     1     0     1     0     0     1
     1     0     1     0     1     0     1     0     0     1     0     1     1     1     0     0

Received bits:
     0     1     1     0     1     1     0     1     0     1     0     1     1     0     1     0
     1     1     1     1     1     1     1     0     1     1     1     0     1     0     0     1
     1     0     1     0     1     0     1     0     0     1     0     1     1     1     0     0

Bit errors: 0
BER: 0.00e+00
```

Figure 5 8PSK Test

As shown in the figures 5, 6, 7 and 8, The noise free has zero error which means that the Tx and Rx are working properly.

Part 2 **AWGN channel**:

Now we're going to add noise equivalent to the noise in real channel by using Average Energy Bit (Eb)

Code:

```
function noisy_signals = addAWGNChannel(SNR_range_db, clean_signal, Eb)
    % ADDAGWNCHANNEL General AWGN channel noise adder
    % Inputs:
    %   SNR_range_db - Array of SNR values in dB
    %   clean_signal - Input signal (vector or matrix)
    %   Eb - Energy per bit
    % Output:
    %   noisy_signals - Cell array of noisy signals for each SNR

    % Initialize output cell array
    noisy_signals = cell(length(SNR_range_db), 1);

    % Get size of input signal
    signal_size = size(clean_signal);

    % Process each SNR point
    for i = 1:length(SNR_range_db)
        % Convert SNR from dB to linear scale
        SNR_linear = 10^(SNR_range_db(i)/10);

        % Calculate noise power (N0)
        N0 = 1 / SNR_linear;

        % Generate proper noise
        if isreal(clean_signal)
            % Real noise for real signals
            noise = sqrt(Eb*N0/2) * randn(signal_size);
        else
            % Complex noise for complex signals
            noise = sqrt(Eb*N0/2) * (randn(signal_size) + 1j*randn(signal_size));
        end

        % Add noise to the signal
        noisy_signals{i} = clean_signal + noise;
    end

    % If only one SNR point was requested, return array instead of cell
    if length(SNR_range_db) == 1
        noisy_signals = noisy_signals{1};
    end
end
```

So the output is scattered on the constellation graph

Output:

BPSK:

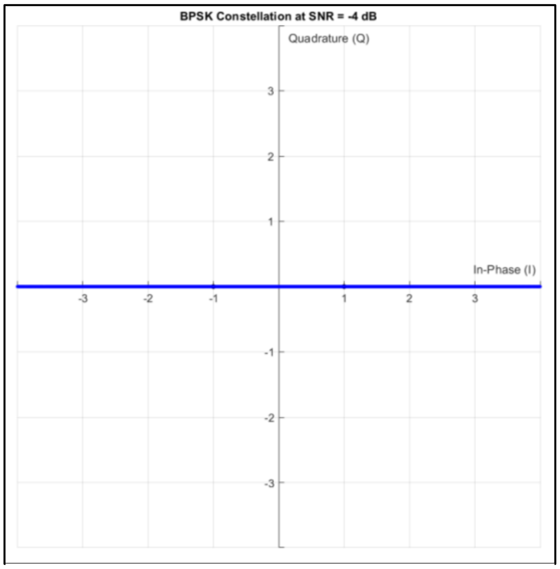

Figure 9 Noise on BPSK with SNR = -4 dB

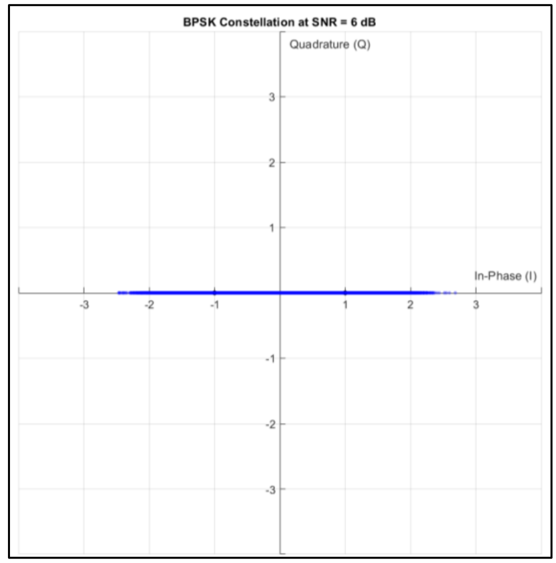

Figure 10 Noise on BPSK with SNR = 6 dB

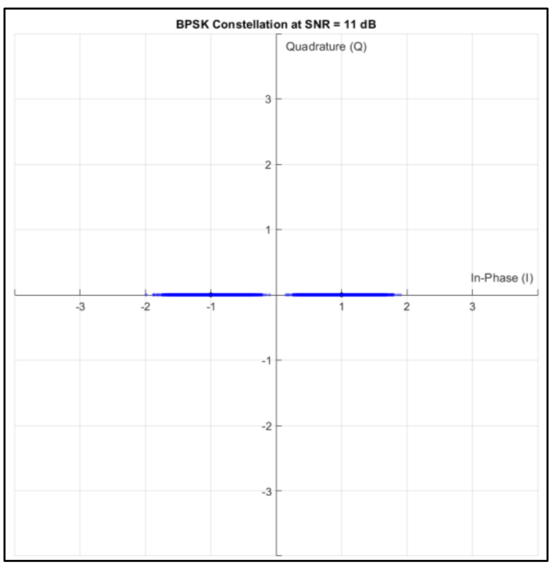

Figure 12 Noise on BPSK with SNR = 11 dB

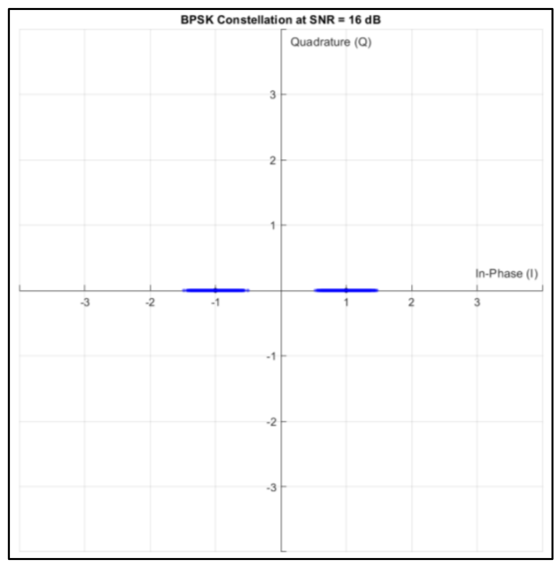

Figure 11 Noise on BPSK with SNR = 16 dB

QPSK:

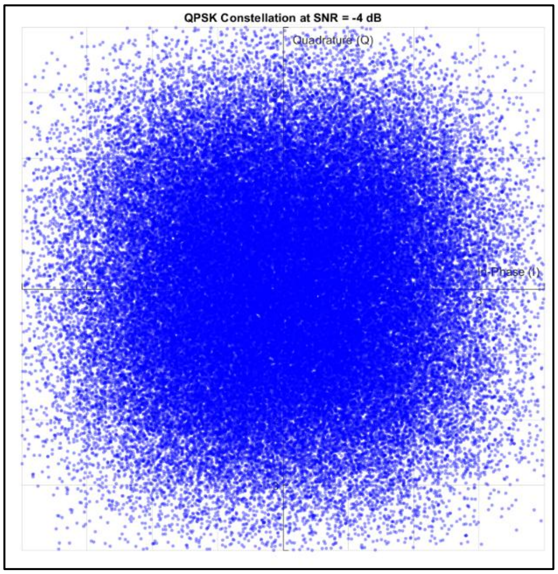

Figure 13 Noise on QPSK with SNR = -4 dB

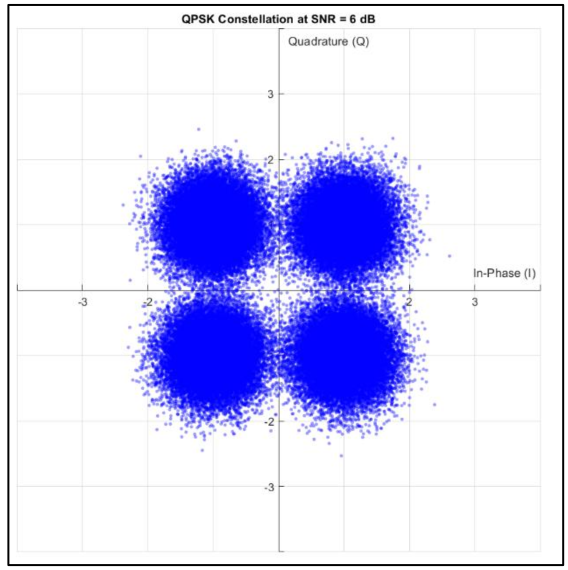

Figure 14 Noise on QPSK with SNR = 6 dB

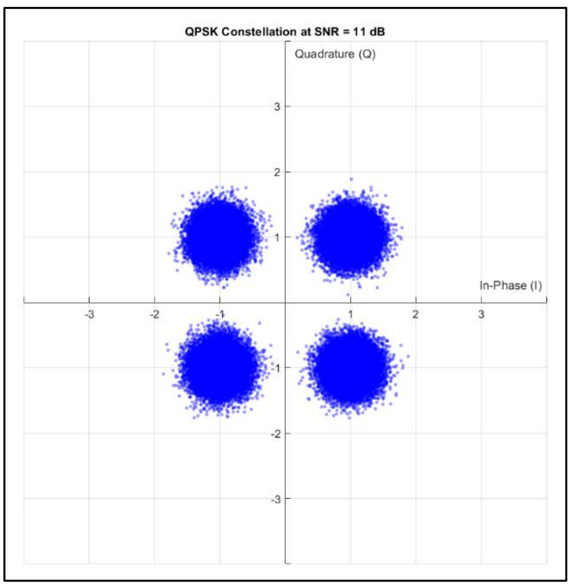

Figure 16 Noise on QPSK with SNR = 11 dB

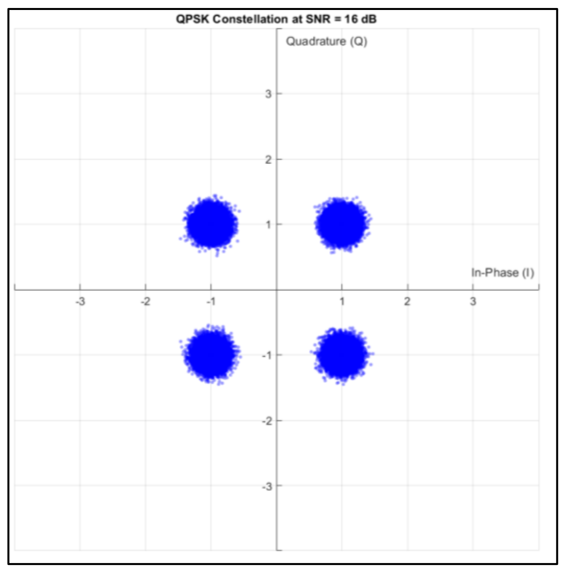

Figure 15 Noise on QPSK with SNR = 16 dB

8PSK:

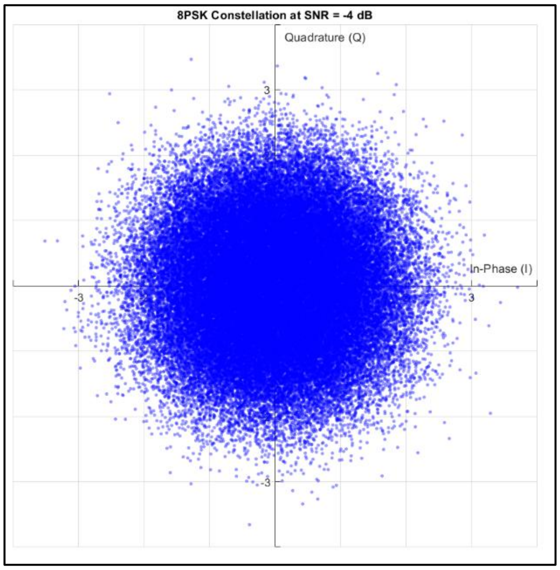

Figure 17 Noise on 8PSK with SNR = -4 dB

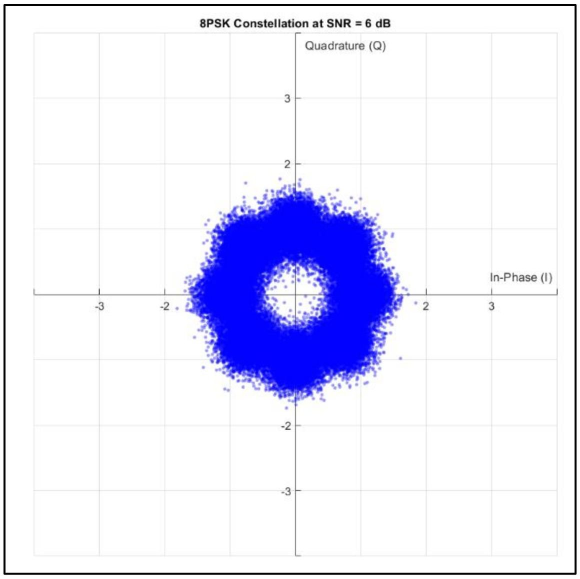

Figure 18 Noise on 8PSK with SNR = 6 dB

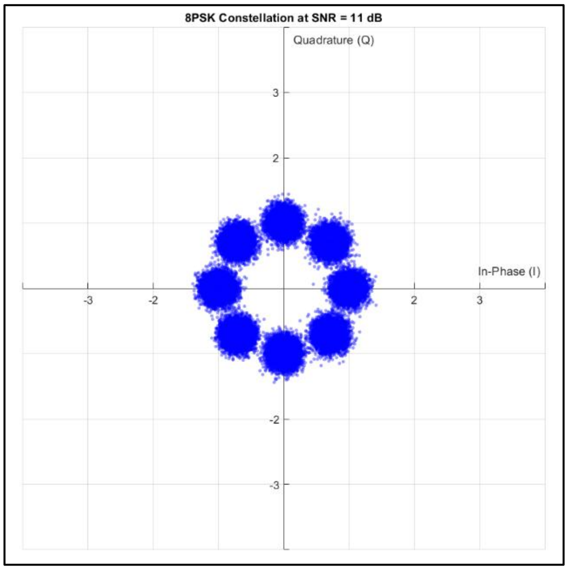

Figure 19 Noise on 8PSK with SNR = 11 dB

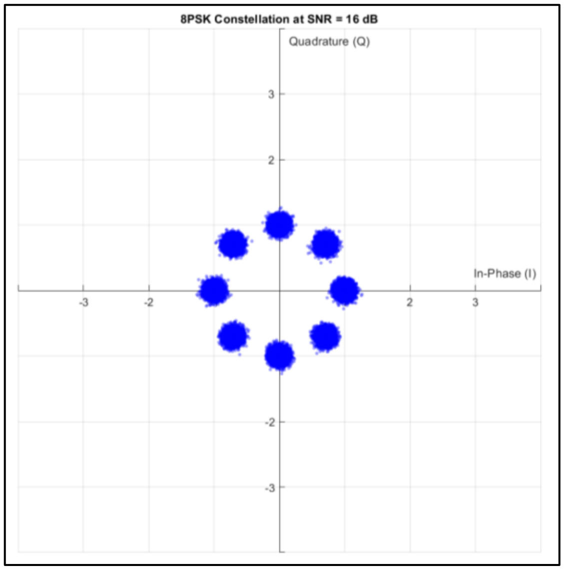

Figure 20 Noise on 8PSK with SNR = 16 dB

16QAM:

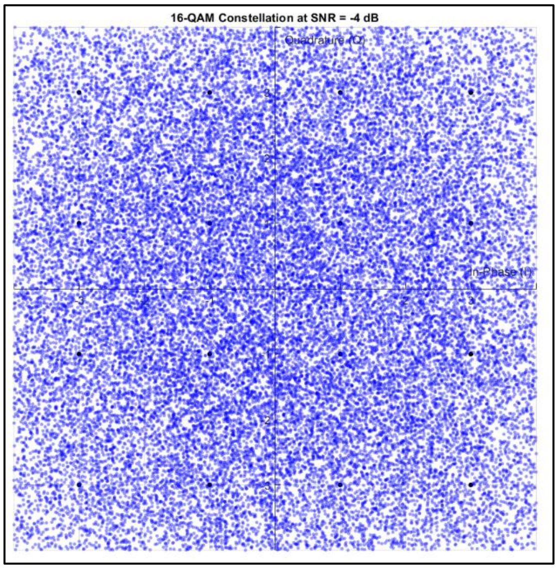

Figure 22 Noise on 16QAM with SNR = -4 dB

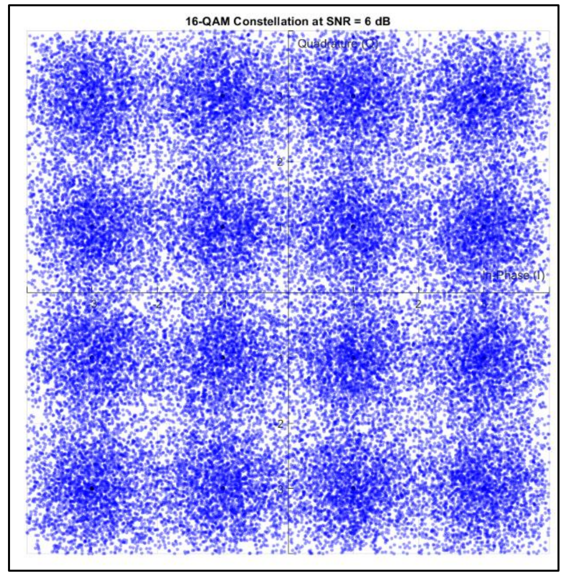

Figure 21 Noise on 16QAM with SNR = 1 dB

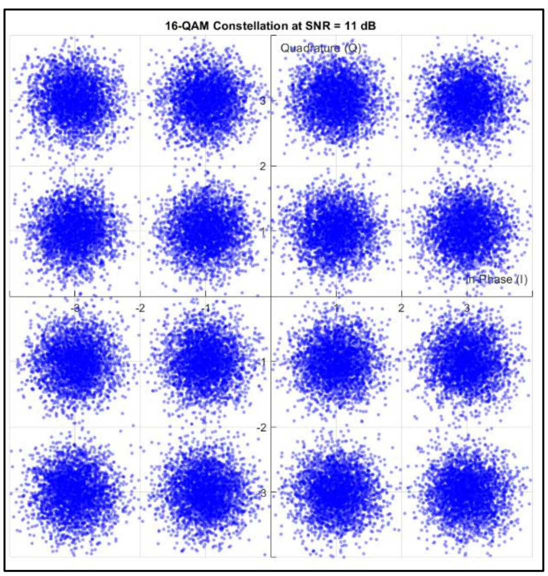

Figure 24 Noise on 16QAM with SNR = 11 dB

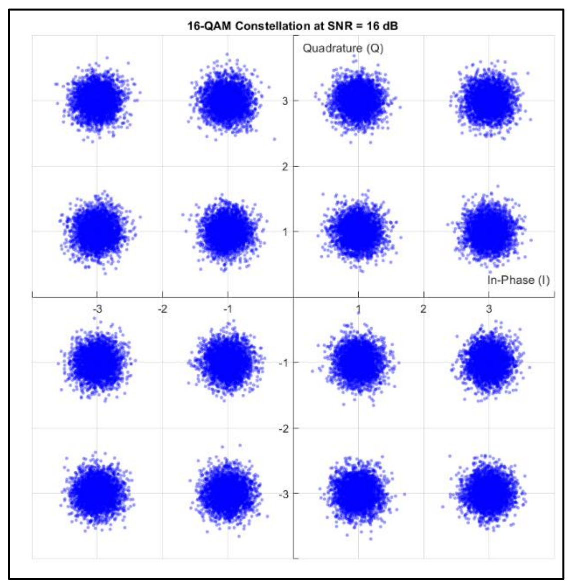

Figure 23 Noise on 16QAM with SNR = 16 dB

It is obvious that noise affects the location of sent symbols on constellation and from the plots we can estimate how good the BER for each scheme based on how good the symbols are well separated where BPSK<QPSK<8PSK16QAM<BFSK.

Tasks

Task 1

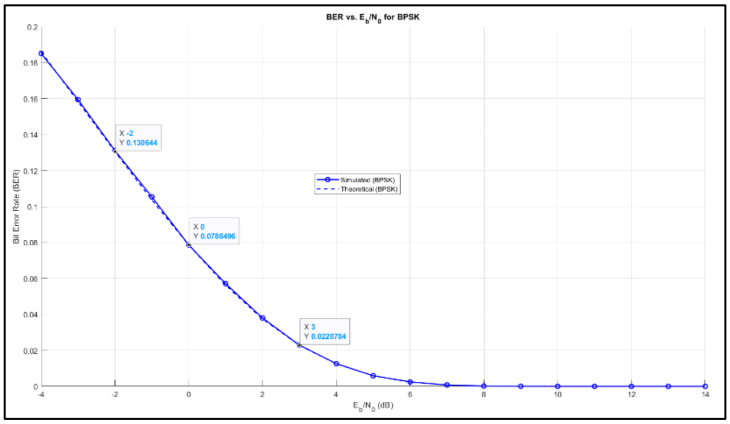

Figure 28 Simulated vs Theoretical BER for BPSK

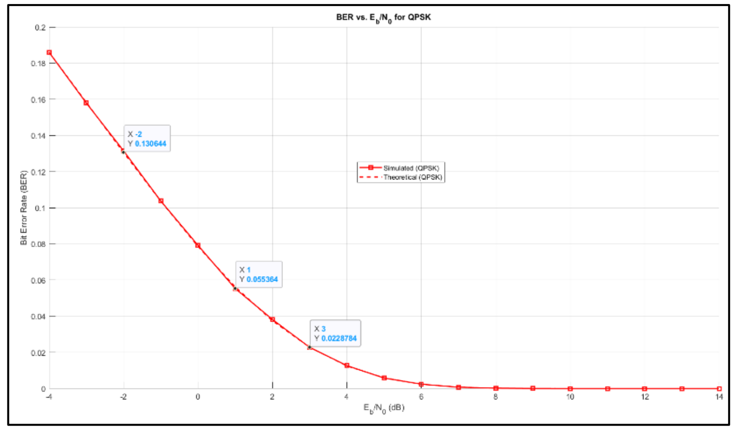

Figure 27 Simulated vs Theoretical BER for QPSK

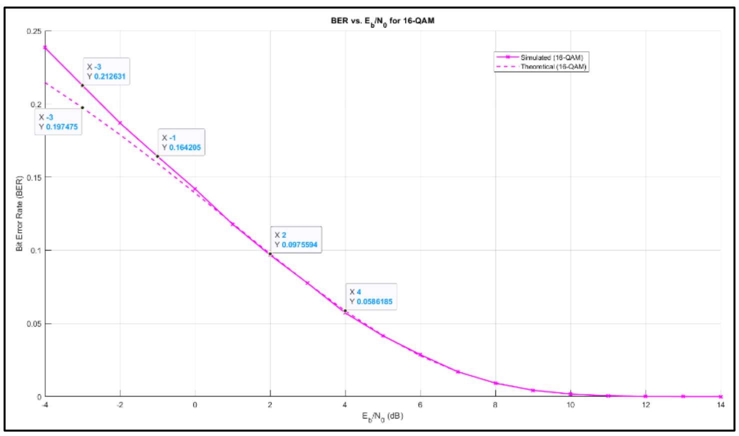

Figure 26 Simulated vs Theoretical BER for 16QAM

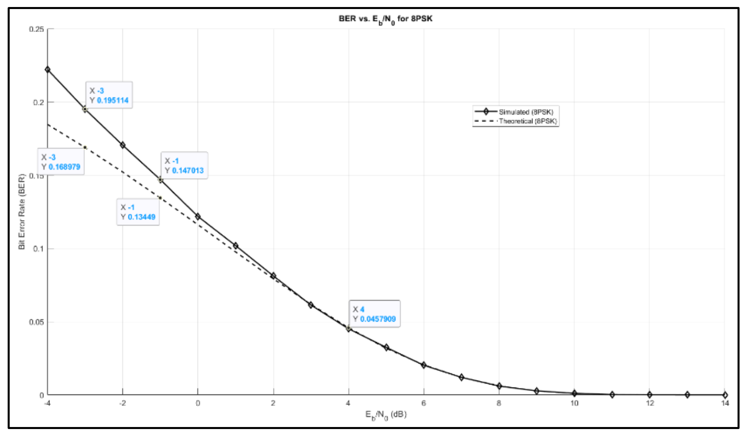

Figure 25 Simulated vs Theoretical BER for 8PSK

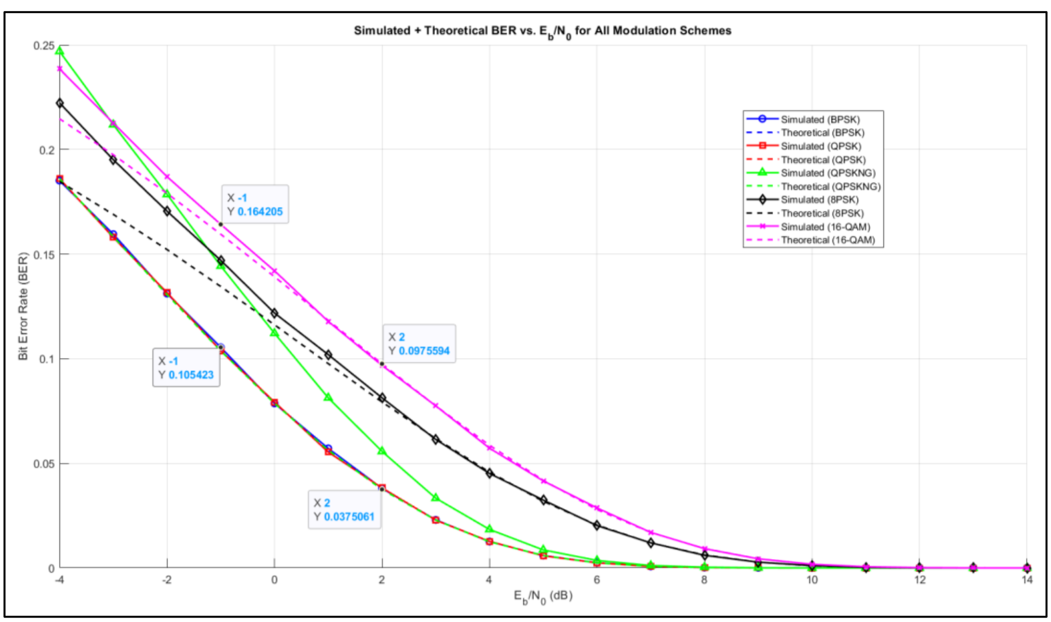

Figure 29 Simulated and Theoritical BER for BPSK, QPSK, 8PSK and 16QAM

Task 2

QPSK not Grey

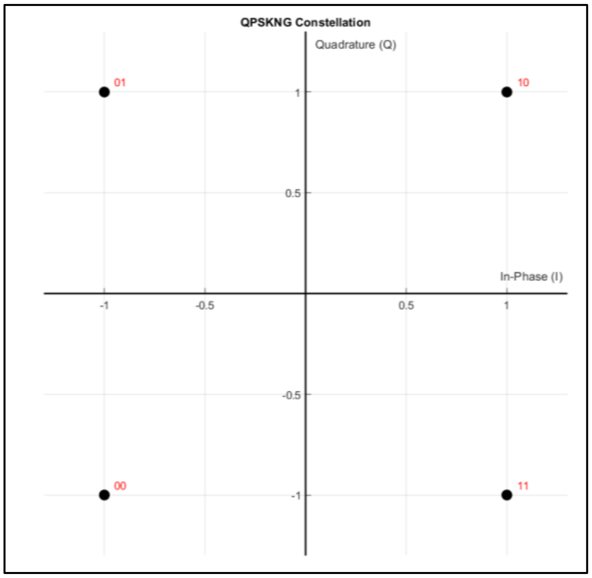

Figure 30 QPSKNG constellation

Output:

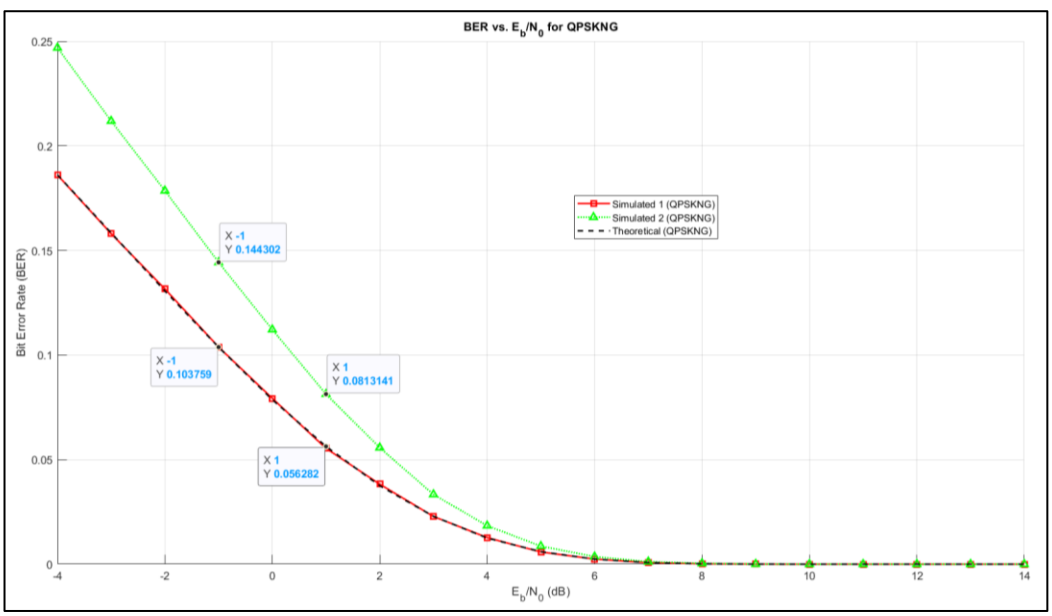

Figure 31 QPSK vs QPSKNG BER

Part 2:

BFSK:

As for the Tx, Rx and BER we used the same functions

Code:

```matlab
% ========================
% Simulation Parameters
% ========================
bits_Num = 6 * 2^15;                                    % Number of bits to transmit
%mod_types = {'BPSK', 'QPSK', 'QPSKNG', '8PSK', '16-QAM', 'BFSK'}; % Cell array of modulation types
mod_types = {'BFSK'};
SNR_db_range = -4:1:16;

% Generate random bits (same for all modulations for fair comparison)
Tx_bits = randi([0 1], 1, bits_Num);

% ========================
% Initialize storage matrices
% ========================

% Initialize rx_symbols_all as 2D cell matrix
% Rows: modulation types, Columns: SNR values
rx_symbols_all = cell(length(mod_types), length(SNR_db_range));

% Initialize storage for Energy Bits
Eb_all = cell(1, length(mod_types));

% Initialize storage for Error
BER_all = zeros(length(mod_types), length(SNR_db_range));
error_count_all = zeros(length(mod_types), length(SNR_db_range));

%   Loop through all modulation types
for mod_idx = 1:length(mod_types)
    mod_type = mod_types{mod_idx};

    fprintf('\n=== %s Modulation ===\n', mod_type);

    % ========================
    % 1. Mapping (Modulation)
    % ========================
    [tx_symbols, constellation,~,Eb] = mapper(Tx_bits, mod_type);

    % Store Energy of bit for this modulation type
    Eb_all{mod_idx} = Eb;

    % ========================
    % 2. Display Constellation
    % ========================
    drawConstellation(constellation, mod_type, 1);
    title(sprintf('%s Constellation', mod_type));

    % ========================
    % 3. Channel Transmission
    % ========================
    % Get noisy symbols for all SNR values
    rx_noisy_symbols = addAWGNChannel(SNR_db_range, tx_symbols, Eb);

    % Store in 2D cell matrix
    rx_symbols_all(mod_idx, :) = rx_noisy_symbols;

    % ========================
    % 4. Demapping (Demodulation)
    % ========================
    Rx_bits = demapper(rx_noisy_symbols, mod_type);

    % ========================
    % 5. Calculate and Store Results
    % ========================
    fprintf('\nSNR Results:\n');
    fprintf('------------\n');

    for snr_idx = 1:1:length(SNR_db_range)
        [BER_all(mod_idx, snr_idx), error_count_all(mod_idx, snr_idx)] = ...
            calculateBER(Tx_bits, Rx_bits{snr_idx});

        % Display results for each SNR
        fprintf('SNR: %6.1f dB | BER: %8.2e | Errors: %4d/%d\n', ...
            SNR_db_range(snr_idx), ...
            BER_all(mod_idx, snr_idx), ...
            error_count_all(mod_idx, snr_idx), ...
            length(Tx_bits));
    end


end
```

Output:

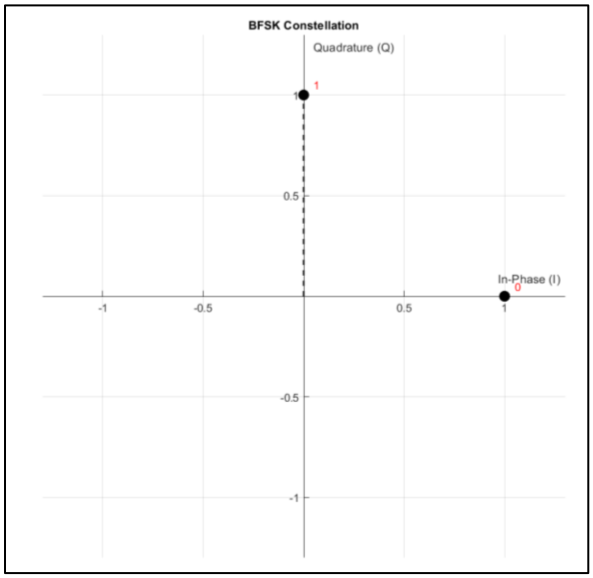

Figure 32 BFSK constellation

Noise:

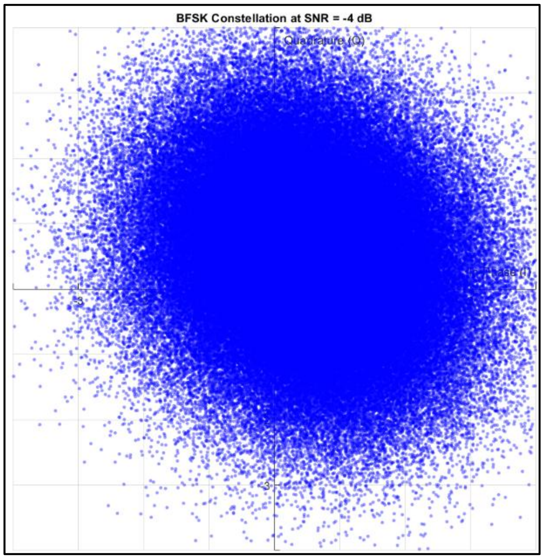

Figure 36 Noise on BFSK with SNR = -4 dB

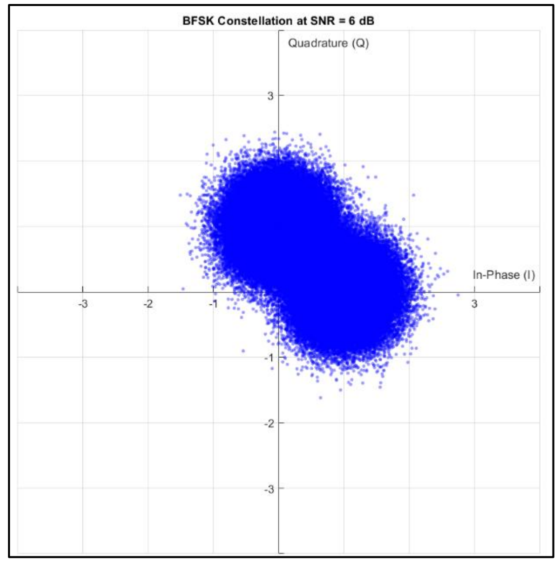

Figure 35 Noise on BFSK with SNR = 6 dB

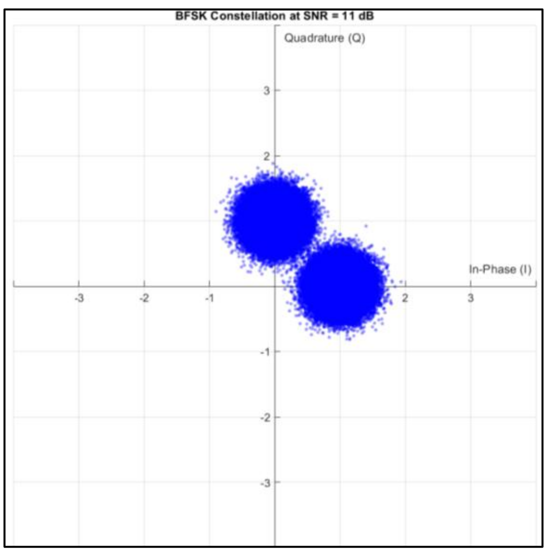

Figure 34 Noise on BFSK with SNR = 11 dB

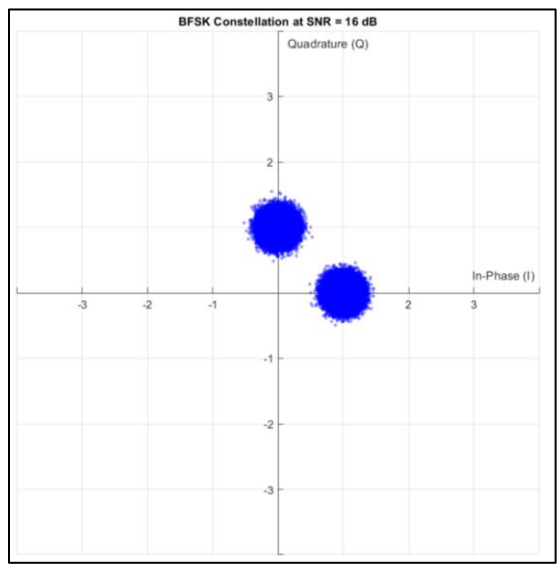

Figure 33 Noise on BFSK with SNR = 16 dB

BER:

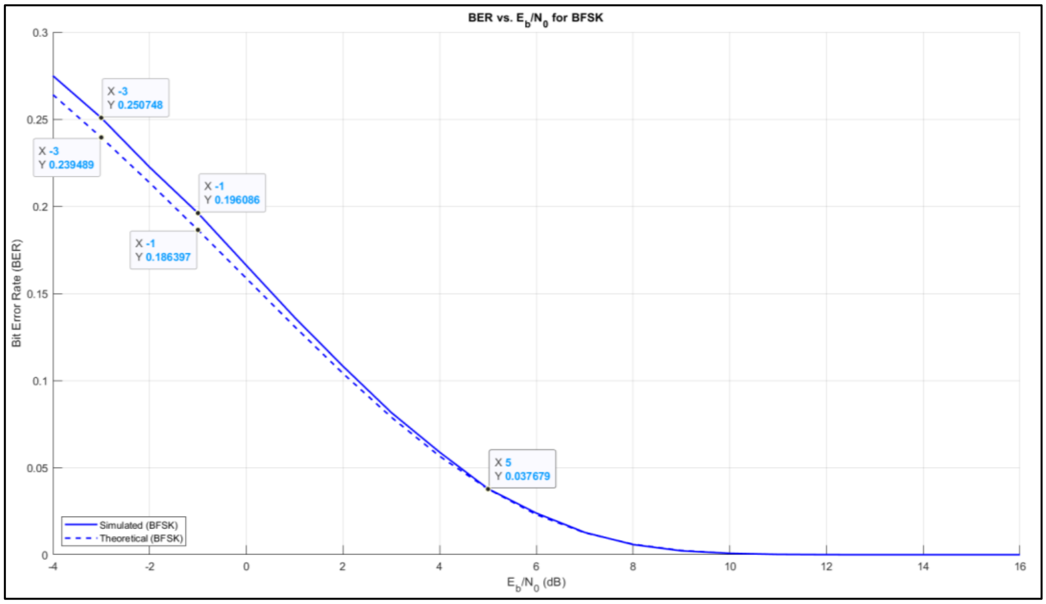

Figure 37 Simulated vs Theoretical BER for BFSK

Base Band:

Code:

```
% ===============================
% declaring parameters (for PSD)
% ===============================
bits_Num = 100;  %less number of bits from the BER
N_realization = 10000;
data = randi([0 1], N_realization, bits_Num + 1);
samples_per_bit=7;
samples_num = samples_per_bit*bits_Num;
sampled_data = repelem(data, 1, samples_per_bit);
Tb = 0.07; % each sample takes 0.01 second

t = 0:Tb/samples_per_bit:Tb;
Fs = 100;
tx_with_delay = zeros(N_realization, 700);

% mapping to BB signals
tx_out = BFSK_BB(bits_Num, N_realization, Tb, Eb, samples_per_bit, sampled_data, t);

% random delay
for i = 1:N_realization
    r = randi([0 (samples_per_bit - 1)]);
    tx_with_delay(i,:) = tx_out(i,r+1:samples_num+r);
end


function [tx_out] = BFSK_BB(bits_Num, N_realization, Tb, Eb, samples_per_bit, sampled_data, t)
% BFSK_BB Generate baseband BFSK time-domain signal
%
% Inputs:
%   bits_Num       - Number of bits per realization
%   N_realization  - Number of realizations
%   Tb             - Bit duration in seconds
%   Eb             - Energy per bit
%
% Output:
%   tx_out         - Baseband BFSK output signal (N_realization x 7*(bits_Num+1))

    % === Derived Parameters ===
    total_samples = samples_per_bit * (bits_Num + 1); % Total samples per realization

    % === Initialize Output Signal ===
    tx_out = zeros(N_realization, total_samples);

    % === Map to Baseband BFSK Signal ===
    for i = 1:N_realization
        for j = 1:samples_per_bit:total_samples
            if sampled_data(i, j) == 0
                tx_out(i, j:j+samples_per_bit-1) = sqrt(2 * Eb / Tb);  % Non-coherent tone for 0
            else
                for k = 1:samples_per_bit
                    tx_out(i, j + k - 1) = sqrt(2 * Eb / Tb) * ...
                        (cos(2 * pi * t(k) / Tb) + 1i * sin(2 * pi * t(k) / Tb));
                end
            end
        end
    end
end
```

Auto Correlation:

Code:

```
function BFSK_autocorr = compute_BFSK_autocorrelation(tx_with_delay)
% COMPUTE_BFSK_AUTOCORRELATION Computes autocorrelation of delayed BFSK signals
% centered at the middle sample.
%
% Input:
%   tx_with_delay   - Matrix of delayed BFSK signals (N_realization × N_samples)
%
% Output:
%   BFSK_autocorr   - Autocorrelation vector (1 × N_samples)

    [~, N_samples] = size(tx_with_delay);

    % Ensure N_samples is even for symmetric range
    if mod(N_samples, 2) ~= 0
        error('N_samples must be even for symmetric autocorrelation.');
    end

    BFSK_autocorr = zeros(1, N_samples);
    center_idx = N_samples / 2;

    for j = -center_idx+1 : center_idx
        i = j + center_idx;
        if i >= 1 && i <= N_samples
            p = conj(tx_with_delay(:, center_idx)) .* tx_with_delay(:, i);
            BFSK_autocorr(i) = sum(p) / length(p);
        end
    end
end
```

Output:

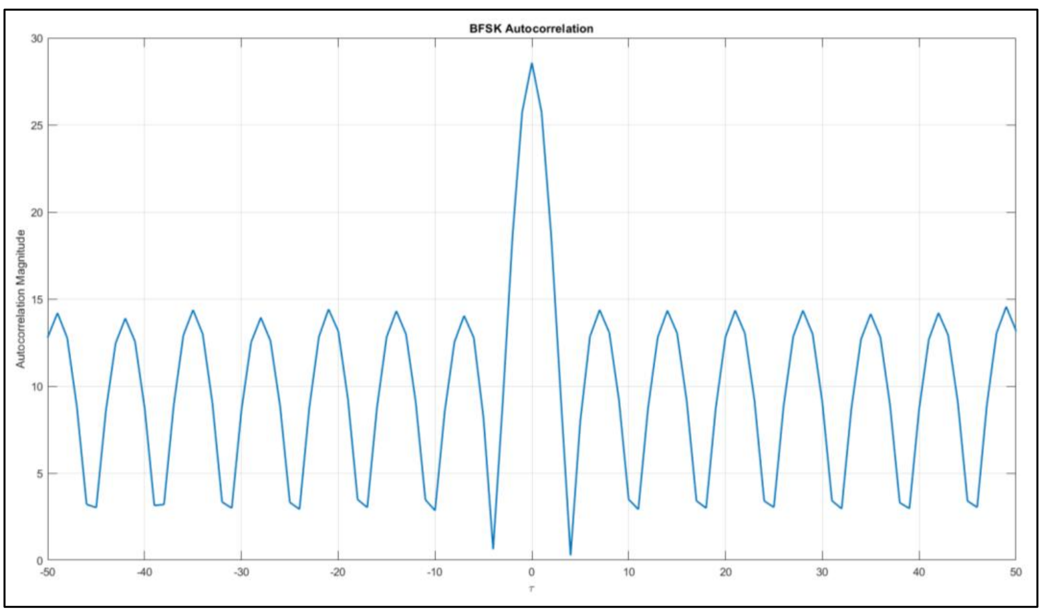

Figure 38 BFSK Auto Correlation

PSD:

```
% Practical PSD
BFSK_PSD = fftshift(fft(Rx_BFSK));     % Use fftshift to center the practical PSD
f = (-350:349) / 700 * Fs;             % Frequency vector for practical PSD
f_normalized = f * Tb;                 % Normalize frequency axis to match the theoretical PSD

% Theoretical PSD
PSD_theoritical = (8 * cos(pi * Tb * f).^2) ./ (pi^2 * (4 * Tb^2 * f.^2 - 1).^2);

% Handle Inf values in the theoretical PSD
idx = PSD_theoritical == Inf;
PSD_theoritical(idx) = 2;  % Change Inf to finite value for plotting
```

Output:

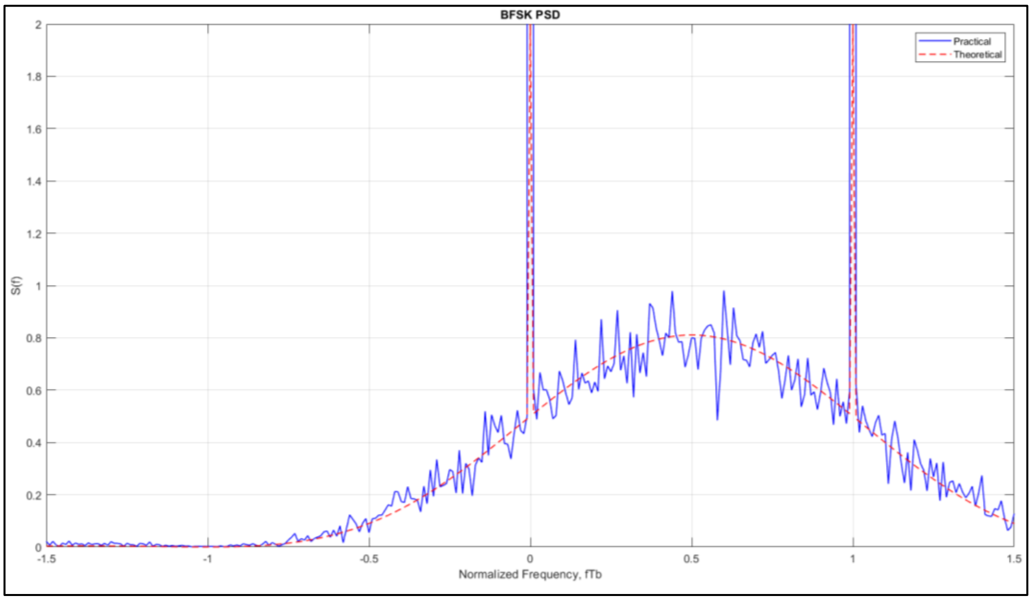

Figure 39 BFSK PSD

Appendix:

Full Code:

```matlab
clear; clc; close all;

%--------Part 1----------
% ========================
% Simulation Parameters
% ========================
bits_Num = 6 * 2^15;                                    % Number of bits to transmit
mod_types = {'BPSK', 'QPSK', 'QPSKNG', '8PSK', '16-QAM', 'BFSK'}; % Cell array of modulation types
SNR_db_range = -4:1:16;

% Generate random bits (same for all modulations for fair comparison)
Tx_bits = randi([0 1], 1, bits_Num);

% ========================
% Initialize storage matrices
% ========================

% Initialize rx_symbols_all as 2D cell matrix
% Rows: modulation types, Columns: SNR values
rx_symbols_all = cell(length(mod_types), length(SNR_db_range));

% Initialize storage for Energy Bits
Eb_all = cell(1, length(mod_types));

% Initialize storage for Error
BER_all = zeros(length(mod_types), length(SNR_db_range));
error_count_all = zeros(length(mod_types), length(SNR_db_range));

%   Loop through all modulation types
for mod_idx = 1:length(mod_types)
    mod_type = mod_types{mod_idx};

    fprintf('\n=== %s Modulation ===\n', mod_type);

    % ========================
    % 1. Mapping (Modulation)
    % ========================
    [tx_symbols, constellation,~,Eb] = mapper(Tx_bits, mod_type);

    % Store Energy of bit for this modulation type
    Eb_all{mod_idx} = Eb;

    % ========================
    % 2. Display Constellation
    % ========================
    drawConstellation(constellation, mod_type, 1);
    title(sprintf('%s Constellation', mod_type));

    % ========================
    % 3. Channel Transmission
    % ========================
    % Get noisy symbols for all SNR values
    rx_noisy_symbols = addAWGNChannel(SNR_db_range, tx_symbols, Eb);

    % Store in 2D cell matrix
    rx_symbols_all(mod_idx, :) = rx_noisy_symbols;

    % ========================
    % 4. Demapping (Demodulation)
    % ========================
    Rx_bits = demapper(rx_noisy_symbols, mod_type);

    % ========================
    % 5. Calculate and Store Results
    % ========================
    fprintf('\nSNR Results:\n');
    fprintf('------------\n');

    for snr_idx = 1:1:length(SNR_db_range)
        [BER_all(mod_idx, snr_idx), error_count_all(mod_idx, snr_idx)] = ...
            calculateBER(Tx_bits, Rx_bits{snr_idx});

        % Display results for each SNR
        fprintf('SNR: %6.1f dB | BER: %8.2e | Errors: %4d/%d\n', ...
            SNR_db_range(snr_idx), ...
            BER_all(mod_idx, snr_idx), ...
            error_count_all(mod_idx, snr_idx), ...
            length(Tx_bits));
    end


end
```

```
%   Display Noise
drawNoisyConstellations(rx_symbols_all, SNR_db_range, mod_types);

%   Graph BER Vs SNR (task 1)
plot_BER_vs_SNR(BER_all, SNR_db_range, mod_types);

%   Graph BER grey vs not grey QPSK (task 2)
plot_BER_vs_SNR_dual(BER_all(2, :), BER_all(3, :), SNR_db_range, mod_types(2:3));

%   Graph BER Vs SNR (task 1)
plot_BER_vs_SNR_all(BER_all, SNR_db_range, mod_types);

% ========================
%    BFSK
% ========================

% ===============================
% declaring parameters (for PSD)
% ===============================
bits_Num = 100;  %less number of bits from the BER
N_realization = 10000;
data = randi([0 1], N_realization, bits_Num + 1);
samples_per_bit=7;
samples_num = samples_per_bit*bits_Num;
sampled_data = repelem(data, 1, samples_per_bit);
Tb = 0.07; % each sample takes 0.01 second

t = 0:Tb/samples_per_bit:Tb;
Fs = 100;
tx_with_delay = zeros(N_realization, 700);

% mapping to BB signals
tx_out = BFSK_BB(bits_Num, N_realization, Tb, Eb, samples_per_bit, sampled_data, t);

% random delay
for i = 1:N_realization
    r = randi([0 (samples_per_bit - 1)]);
    tx_with_delay(i,:) = tx_out(i,r+1:samples_num+r);
end

% Autocorrelation
BFSK_autocorr = compute_BFSK_autocorrelation(tx_with_delay);
Rx_BFSK = BFSK_autocorr;

% plt auto correlation
draw_autocorr(Rx_BFSK);

Practical PSD
BFSK_PSD = fftshift(fft(Rx_BFSK));    % Use fftshift to center the practical PSD
f = (-350:349) / 700 * Fs;            % Frequency vector for practical PSD
f_normalized = f * Tb;                % Normalize frequency axis to match the theoretical PSD

% Theoretical PSD
PSD_theoritical = (8 * cos(pi * Tb * f).^2) ./ (pi^2 * (4 * Tb^2 * f.^2 - 1).^2);

% Handle Inf values in the theoretical PSD
idx = PSD_theoritical == Inf;
PSD_theoritical(idx) = 2;  % Change Inf to finite value for plotting

% Plot PSD
draw_psd(f_normalized, BFSK_PSD, PSD_theoritical);
```

Functions:

```
% ========================
% Functions
% ========================

function [Tx_Vector, Table, Eavg, Eb] = mapper(bits, mod_type)
    % MAPPER Digital modulation mapper with explicit symbol table and energy calculation
    % Inputs:
    %   bits     - Binary input array (row vector)
    %   mod_type - 'BPSK', 'QPSK', 'QPSKNG', '8PSK', 'BFSK', '16-QAM'
    % Outputs:
    %   Tx_Vector - Complex modulated symbols
    %   Table     - Constellation points (M-ary symbols)
    %   Eavg      - Average symbol energy (normalized)
    %   Eb        - Energy per bit

    % Ensure bits are row vector
    bits = bits(:)';

    % Define modulation parameters
    switch upper(mod_type)
        case 'BPSK'
            n = 1;  % bits per symbol
            M = 2;   % constellation size
            Table = [-1, 1];  % BPSK symbols (real)

        case 'QPSK'
            n = 2;
            M = 4;
            Table = [-1-1j, -1+1j, 1-1j, 1+1j];  % QPSK symbols

        case 'QPSKNG'
            n = 2;
            M = 4;
            Table = [-1-1j, -1+1j, 1+1j, 1-1j];  % QPSKNG symbols

        case '8PSK'
            n = 3;
            M = 8;
            angles =[0, 1, 3, 2, 7, 6, 4, 5]*pi/4;  % Gray-coded 8PSK
            Table = exp(1j*angles);

        case 'BFSK'
            n=1;
            M=2;
            Table = [ 1, 1j];

        case '16-QAM'
            n = 4;
            M = 16;
            % 16-QAM with unit average power (normalized)
            Table = [-3-3j, -3-1j, -3+3j, -3+1j, ...
                     -1-3j, -1-1j, -1+3j, -1+1j, ...
                      3-3j,  3-1j,  3+3j,  3+1j, ...
                      1-3j,  1-1j,  1+3j,  1+1j];

        otherwise
            error('Unsupported modulation type: %s', mod_type);
    end

    % Pad bits if not multiple of n
    if mod(length(bits), n) ~= 0
        bits = [bits zeros(1, n - mod(length(bits), n))];
    end

    % Calculate average symbol energy
    Eavg = mean(abs(Table).^2);

    % Calculate average bit energy
    Eb =  Eavg / n;

    % Reshape into n-bit groups
    bit_groups = reshape(bits, n, [])';

    % Convert to decimal symbols (0 to M-1)
    Array_symbol = bi2de(bit_groups, 'left-msb') + 1;  % MATLAB uses 1-based indexing

    % Map to constellation points
    Tx_Vector = Table(Array_symbol);
end
```

```matlab
function drawConstellation(Table, mod_type, showdetails)
    % DRAWCOnSTELLATION Enhanced constellation visualization
    % Inputs:
    %   Table - Constellation points (complex numbers)
    %   mod_type - Modulation type ('BPSK', 'QPSK', etc.)
    %   showdetails- true to show colored regions, false for boundaries only

    if nargin < 3
        show_regions = true; % Default to showing regions
    end

    figure;
    hold on;

    % Ensure Table is column vector and get points
    Table = Table(:);
    points = [real(Table), imag(Table)];

    % Create grid for visualization
    x_range = linspace(min(points(:,1))-1, max(points(:,1))+1, 200);
    y_range = linspace(min(points(:,2))-1, max(points(:,2))+1, 200);
    [x_grid, y_grid] = meshgrid(x_range, y_range);
    grid_points = x_grid(:) + 1j*y_grid(:);

    % ============================================
    % 1. Decision Visualization
    % ============================================
    if showdetails == 1
        if length(Table) > 2  % Voronoi needs at least 3 points
            [vx, vy] = voronoi(points(:,1), points(:,2));
            plot(vx, vy, 'k-', 'LineWidth', 1.5);
        else
            % For BPSK, draw simple decision boundary
            plot([0 0], ylim, 'k--', 'LineWidth', 1.5);
        end
    end

    % ============================================
    % 2. Constellation Points
    % ============================================
    if showdetails == 1
        scatter(points(:,1), points(:,2), 100, 'filled', 'k');
    else
        scatter(points(:,1), points(:,2), 20, 'filled', 'k');
    end
    % ============================================
    % 3. Binary Labels
    % ============================================
    switch upper(mod_type)
        case 'BPSK'
            n = 1;
        case 'QPSK'
            n = 2;
        case 'QPSKNG'
            n = 2;
        case '8PSK'
            n = 3;
        case {'16QAM', '16-QAM'}
            n = 4;
        case 'BFSK'
            n=1;
        otherwise
            error('Unsupported modulation type');
    end

    if showdetails == 1
        for i = 1:length(Table)
            bin_str = dec2bin(i-1, n);
            % Position text slightly offset from the point
            text(real(Table(i)) + 0.05, imag(Table(i)) + 0.05, bin_str, ...
                'FontSize', 10, 'Color', 'r');
        end
    end
    % ============================================
    % 4. Plot Formatting
    % ============================================
    title(sprintf('%s Constellation', mod_type));
    xlabel('In-Phase (I)'); ylabel('Quadrature (Q)');
    grid on;
    axis equal;

    % Center axes
    ax = gca;
    ax.XAxisLocation = 'origin';
    ax.YAxisLocation = 'origin';

    % Set axis limits
    max_val = max([abs(points(:))]) * 1.3;
    xlim([-max_val, max_val]);
    ylim([-max_val, max_val]);


    hold off;
end
```

```matlab
function drawNoisyConstellations(rx_symbols_all, SNR_db_range, mod_types)
    % DRAWNOISYCONSTELLATIONS Plot constellations with noisy received points
    % Inputs:
    %   rx_symbols_all - Cell array, rx_symbols_all{mod_idx, snr_idx}
    %   SNR_db_range   - Vector of SNR values (dB)
    %   mod_types      - Cell array of modulation type strings (e.g., {'BPSK', 'QPSK'})

    % Validate inputs
    if ~iscell(rx_symbols_all) || ~iscell(mod_types)
        error('rx_symbols_all and mod_types must be cell arrays.');
    end

    num_mods = numel(mod_types);
    num_snr = numel(SNR_db_range);

    for mod_idx = 1:num_mods
        mod_type = mod_types{mod_idx};

        % Generate constellation table for this modulation
        [~, Table] = mapper([1], mod_type);

        for snr_idx = 1:floor(num_snr/4):num_snr
            rx_symbols = rx_symbols_all{mod_idx, snr_idx};
            snr_db = SNR_db_range(snr_idx);

            % Center axes
            ax = gca;
            ax.XAxisLocation = 'origin';
            ax.YAxisLocation = 'origin';

            % Plot decision regions and ideal points
            drawConstellation(Table, mod_type, 0);
            title(sprintf('%s Constellation at SNR = %d dB', mod_type, snr_db));
            xlabel('In-Phase (I)'); ylabel('Quadrature (Q)');
            grid on;
            axis equal;
            hold on;
            % Plot noisy received symbols
            scatter(real(rx_symbols), imag(rx_symbols), 10, 'b', 'filled', 'MarkerFaceAlpha', 0.4);

            % Set axis limits a bit bigger to fit noisy points
            max_val = 4;
            xlim([-max_val, max_val]);
            ylim([-max_val, max_val]);

            hold off;
        end
    end
end

function [received_bits] = demapper(received_symbols, mod_type)
    % DEMAPPER Digital demodulation demapper
    % Inputs:
    %   received_symbols - Complex received symbols (array or cell array)
    %   mod_type         - Modulation type ('BPSK', 'QPSK', etc.)
    % Output:
    %   received_bits    - Demodulated bit stream (array or cell array)

    % Check if input is cell array (multiple SNR cases)
    if iscell(received_symbols)
        % Process each SNR case
        received_bits = cell(size(received_symbols));
        for i = 1:numel(received_symbols)
            received_bits{i} = demodulate_symbols(received_symbols{i}, mod_type);
        end
    else
        % Single SNR case
        received_bits = demodulate_symbols(received_symbols, mod_type);
    end
end
```

```matlab
function bits = demodulate_symbols(symbols, mod_type)
    % Helper function for actual demodulation

    % Determine bits per symbol
    switch upper(mod_type)
        case 'BPSK'
            n = 1;
        case 'QPSK'
            n = 2;
        case 'QPSKNG'
            n = 2;
        case '8PSK'
            n = 3;
        case {'16QAM', '16-QAM'}
            n = 4;
        case 'BFSK'
            n=1;
        otherwise
            error('Unsupported modulation type');
    end

    % Initialize output bits
    bits = zeros(1, length(symbols)*n);

    % ======================
    % Special case for BFSK
    % ======================
    if strcmpi(mod_type, 'BFSK')
        for i = 1:length(symbols)
            theta = angle(symbols(i));
            if (theta > pi/4 && theta < 5*pi/4)
                bits(i) = 1;
            else
                bits(i) = 0;
            end
        end
        return;
    end

    % ======================
    % General case
    % ======================

    % Get constellation table from mapper
    [~, Table] = mapper([1], mod_type);

    % Demodulate each symbol
    for i = 1:length(symbols)
        % Find nearest constellation point
        [~, idx] = min(abs(symbols(i) - Table));

        % Convert to binary (0-based index)
        bin_str = dec2bin(idx-1, n);

        % Store bits
        bits((i-1)*n+1:i*n) = bin_str - '0';
    end
end

function noisy_signals = addAWGNChannel(SNR_range_db, clean_signal, Eb)
    % ADDAGWNCHANNEL General AWGN channel noise adder
    % Inputs:
    %   SNR_range_db - Array of SNR values in dB
    %   clean_signal - Input signal (vector or matrix)
    %   Eb - Energy per bit
    % Output:
    %   noisy_signals - Cell array of noisy signals for each SNR

    % Initialize output cell array
    noisy_signals = cell(length(SNR_range_db), 1);

    % Get size of input signal
    signal_size = size(clean_signal);

    % Process each SNR point
    for i = 1:length(SNR_range_db)
        % Convert SNR from dB to linear scale
        SNR_linear = 10^(SNR_range_db(i)/10);

        % Calculate noise power (N0)
        N0 = 1 / SNR_linear;

        % Generate proper noise
        if isreal(clean_signal)
            % Real noise for real signals
            noise = sqrt(Eb*N0/2) * randn(signal_size);
        else
            % Complex noise for complex signals
            noise = sqrt(Eb*N0/2) * (randn(signal_size) + 1j*randn(signal_size));
        end

        % Add noise to the signal
        noisy_signals{i} = clean_signal + noise;
    end

    % If only one SNR point was requested, return array instead of cell
    if length(SNR_range_db) == 1
        noisy_signals = noisy_signals{1};
    end
end
```

```
function [BER, bit_errors] = calculateBER(original_bits, received_bits)
    % CALCULATEBER Compute Bit Error Rate for single or multiple SNR cases
    % Inputs:
    %   original_bits - Transmitted bit sequence (1D array)
    %   received_bits - Received bits (1D array or cell array for multiple SNR)
    % Outputs:
    %   BER - Bit Error Rate (scalar or array matching received_bits input)
    %   bit_errors - Number of errors (scalar or array)

    % Ensure original bits are row vector
    original_bits = original_bits(:)';

    % Handle cell array input (multiple SNR cases)
    if iscell(received_bits)
        BER = zeros(size(received_bits));
        bit_errors = zeros(size(received_bits));

        for i = 1:numel(received_bits)
            [BER(i), bit_errors(i)] = calculateSingleBER(original_bits, received_bits{i});
        end
    else
        % Single SNR case
        [BER, bit_errors] = calculateSingleBER(original_bits, received_bits);
    end
end

function [BER, bit_errors] = calculateSingleBER(original_bits, received_bits)
    % Helper function for single SNR case BER calculation

    % Ensure received bits are row vector
    received_bits = received_bits(:)';

    % Trim received bits if longer (due to padding)
    if length(received_bits) > length(original_bits)
        received_bits = received_bits(1:length(original_bits));
    end

    % Calculate errors
    bit_errors = sum(original_bits ~= received_bits);
    BER = bit_errors / length(original_bits);
End

function displayBitComparison(Tx_bits, Rx_bits, bit_errors, BER, bits_per_group)
    % DISPLAYBITCOMPARISON Display input/output bit comparison and BER results
    %
    % Inputs:
    %   Tx_bits - Transmitted bit sequence
    %   Rx_bits - Received bit sequence
    %   bit_errors - Number of bit errors
    %   BER - Bit Error Rate
    %   bits_per_group - Number of bits to display per row (default: 16)

    if nargin < 5
        bits_per_group = 16; % Default to 16-bit groups
    end

    % Ensure inputs are row vectors
    Tx_bits = Tx_bits(:)';
    Rx_bits = Rx_bits(:)';

    % Display original bits
    fprintf('Original bits:\n');
    disp(reshape(Tx_bits, bits_per_group, [])');

    % Display received bits (trimmed to original length)
    fprintf('\nReceived bits:\n');
    disp(reshape(Rx_bits(1:length(Tx_bits)), bits_per_group, [])');

    % Display error statistics
    fprintf('\nError Analysis:\n');
    fprintf('Bit errors: %d\n', bit_errors);
    fprintf('BER: %.2e\n', BER);

end
```

```matlab
function plot_BER_vs_SNR(BER_all, SNR_Range, Mod_Types)
    % This function plots BER vs SNR for multiple modulation types
    % Inputs:
    %   BER_all    : matrix (SNR points × modulation types)
    %   SNR_Range  : vector of SNR values in dB
    %   Mod_Types  : cell array of modulation type names (strings)

    % Transpose BER_all if it has the wrong dimensions
    if size(BER_all, 1) ~= length(SNR_Range)
        BER_all = BER_all.';
    end

    % Number of modulation types
    num_mods = length(Mod_Types);

    % Define colors and markers for different mod types
    colors = ['b', 'r', 'g', 'k', 'm', 'c', 'y'];
    %markers = ['o', 's', '^', 'd', 'x', '+', '*'];

    % Loop over each modulation type and create a new figure for each
    for idx = 1:num_mods
        % Create a new figure for each modulation type
        figure;
        hold on;
        grid on;

        % Plot simulated BER
        semilogy(SNR_Range, BER_all(:, idx), ...
                 [colors(mod(idx-1,length(colors))+1) ], ...
                 'LineWidth', 1.5);

        EbNo = 10.^(SNR_Range/10); % Convert SNR from dB to linear

        % Plot theoretical or tight upper bound BER
        switch Mod_Types{idx}
            case 'BPSK'
                BER_theory = 0.5 * erfc(sqrt(EbNo));
            case 'QPSK'
                BER_theory = 0.5 * erfc(sqrt(EbNo)); % same as BPSK
            case 'QPSKNG'
                BER_theory = 0.5 * erfc(sqrt(EbNo)); % same as QPSK
            case '8PSK'
                BER_theory = erfc(sin(pi/8) * sqrt(3 * EbNo)) / 3;
            case '16-QAM'
                BER_theory = (3/8)*erfc(sqrt((2/5)*EbNo));
            case '64qam'
                BER_theory = (7/24)*erfc(sqrt((7/21)*EbNo));
            case 'BFSK'
                BER_theory = 0.5*erfc(sqrt(0.5*EbNo));
            otherwise
                warning('No theoretical curve for %s. Skipping.', Mod_Types{idx});
                BER_theory = nan(size(EbNo));
        end

        % If theoretical BER is computed, plot it
        if ~any(isnan(BER_theory))
            semilogy(SNR_Range, BER_theory, ...
                     [colors(mod(idx-1,length(colors))+1) '--'], ...
                     'LineWidth', 1.5);
        end

        % Labels and title
        xlabel('E_b/N_0 (dB)');
        ylabel('Bit Error Rate (BER)');
        title(['BER vs. E_b/N_0 for ' Mod_Types{idx}]);

        % Add a legend
        legend_entries = {['Simulated (' Mod_Types{idx} ')'], ['Theoretical (' Mod_Types{idx} ')']};
        legend(legend_entries, 'Location', 'southwest');

        % Set plot limits
        %ylim([1e-5 1]);
        xlim([min(SNR_Range) max(SNR_Range)]);

        hold off;
    end
end
```

```matlab
function plot_BER_vs_SNR_dual(BER1, BER2, SNR_Range, Mod_Types)
    % Plots BER vs SNR for two BER datasets + theoretical for multiple mod types
    % Inputs:
    %   BER1       : matrix (SNR points × modulation types) - first BER dataset
    %   BER2       : matrix (SNR points × modulation types) - second BER dataset
    %   SNR_Range  : vector of SNR values in dB
    %   Mod_Types  : cell array of modulation type names (strings)

    % Transpose if needed
    if size(BER1, 1) ~= length(SNR_Range)
        BER1 = BER1.';
    end
    if size(BER2, 1) ~= length(SNR_Range)
        BER2 = BER2.';
    end

    num_mods = length(Mod_Types);
    colors = ['b', 'r', 'g', 'k', 'm', 'c', 'y'];
    %markers = ['o', 's', '^', 'd', 'x', '+', '*'];

    for idx = 1:num_mods
        figure;
        hold on;
        grid on;

        EbNo = 10.^(SNR_Range/10); % Convert SNR from dB to linear

        % Plot BER1 (e.g., baseline)
        semilogy(SNR_Range, BER1, ...
                 [colors(mod(idx-1,length(colors))+1) ], ...
                 'LineWidth', 1.5);

        % Plot BER2 (e.g., improved method)
        semilogy(SNR_Range, BER2, ...
                 [colors(mod(idx,length(colors))+1) ], ...
                 'LineWidth', 1.5);

        % Compute theoretical BER
        switch Mod_Types{idx}
            case 'BPSK'
                BER_theory = 0.5 * erfc(sqrt(EbNo));
            case 'QPSK'
                BER_theory = 0.5 * erfc(sqrt(EbNo));
            case 'QPSKNG'
                BER_theory = 0.5 * erfc(sqrt(EbNo));
            case '8PSK'
                BER_theory = erfc(sin(pi/8) * sqrt(3 * EbNo)) / 3;
            case '16-QAM'
                BER_theory = (3/8)*erfc(sqrt((2/5)*EbNo));
            case '64qam'
                BER_theory = (7/24)*erfc(sqrt((7/21)*EbNo));
            case 'BFSK'
                BER_theory = 0.5*erfc(sqrt(0.5*EbNo));
            otherwise
                warning('No theoretical curve for %s. Skipping.', Mod_Types{idx});
                BER_theory = nan(size(EbNo));
        end

        % Plot theoretical BER if available
        if ~any(isnan(BER_theory))
            semilogy(SNR_Range, BER_theory, ...
                     [colors(mod(idx+1,length(colors))+1) '--'], ...
                     'LineWidth', 1.5);
        end

        % Labels and title
        xlabel('E_b/N_0 (dB)');
        ylabel('Bit Error Rate (BER)');
        title(['BER vs. E_b/N_0 for ' Mod_Types{idx}]);

        % Legend
        legend_entries = {['Simulated 1 (' Mod_Types{idx} ')'], ...
                          ['Simulated 2 (' Mod_Types{idx} ')'], ...
                          ['Theoretical (' Mod_Types{idx} ')']};
        legend(legend_entries, 'Location', 'southwest');

        xlim([min(SNR_Range) max(SNR_Range)]);
        %ylim([1e-5 1]);

        hold off;
    end
end
```

```matlab
function plot_BER_vs_SNR_all(BER_all, SNR_Range, Mod_Types)
    % This function plots:
    % 1. All simulated BER curves in one figure
    % 2. All simulated + theoretical BER curves in another figure
    %
    % Inputs:
    %   BER_all    : matrix (SNR points × modulation types)
    %   SNR_Range  : vector of SNR values in dB
    %   Mod_Types  : cell array of modulation type names (strings)

    % Transpose if needed
    if size(BER_all, 1) ~= length(SNR_Range)
        BER_all = BER_all.';
    end

    colors = ['b', 'r', 'g', 'k', 'm', 'c', 'y'];
    %markers = ['o', 's', '^', 'd', 'x', '+', '*'];
    EbNo = 10.^(SNR_Range / 10); % Convert to linear

    % 1. PLOT ONLY SIMULATED BER
    figure;
    hold on; grid on;
    legend_entries = {};

    for idx = 1:length(Mod_Types)
        color = colors(mod(idx-1, length(colors)) + 1);
        %marker = markers(mod(idx-1, length(markers)) + 1);

        semilogy(SNR_Range, BER_all(:, idx), ...
                 [color], ...
                 'LineWidth', 1.5);

        legend_entries{end+1} = ['Simulated (' Mod_Types{idx} ')'];
    end

    xlabel('E_b/N_0 (dB)');
    ylabel('Bit Error Rate (BER)');
    title('Simulated BER vs. E_b/N_0 for All Modulation Schemes');
    legend(legend_entries, 'Location', 'southwest');
    xlim([min(SNR_Range), max(SNR_Range)]);
    hold off;

    % 2. PLOT SIMULATED + THEORETICAL BER
    figure;
    hold on; grid on;
    legend_entries = {};

    for idx = 1:length(Mod_Types)
        color = colors(mod(idx-1, length(colors)) + 1);
        %marker = markers(mod(idx-1, length(markers)) + 1);

        % Simulated
        semilogy(SNR_Range, BER_all(:, idx), ...
                 [color], ...
                 'LineWidth', 1.5);
        legend_entries{end+1} = ['Simulated (' Mod_Types{idx} ')'];

        % Theoretical
        switch Mod_Types{idx}
            case {'BPSK', 'QPSK', 'QPSKNG'}
                BER_theory = 0.5 * erfc(sqrt(EbNo));
            case '8PSK'
                BER_theory = erfc(sin(pi/8) * sqrt(3 * EbNo)) / 3;
            case '16-QAM'
                BER_theory = (3/8) * erfc(sqrt((2/5)*EbNo));
            case '64qam'
                BER_theory = (7/24) * erfc(sqrt((7/21)*EbNo));
            case 'BFSK'
                BER_theory = 0.5 * erfc(sqrt(0.5*EbNo));
            otherwise
                BER_theory = nan(size(EbNo));
        end

        if ~any(isnan(BER_theory))
            semilogy(SNR_Range, BER_theory, ...
                     [color '--'], ...
                     'LineWidth', 1.5);
            legend_entries{end+1} = ['Theoretical (' Mod_Types{idx} ')'];
        end
    end

    xlabel('E_b/N_0 (dB)');
    ylabel('Bit Error Rate (BER)');
    title('Simulated + Theoretical BER vs. E_b/N_0 for All Modulation Schemes');
    legend(legend_entries, 'Location', 'southwest');
    xlim([min(SNR_Range), max(SNR_Range)]);
    hold off;
end
```

```matlab
function [tx_out] = BFSK_BB(bits_Num, N_realization, Tb, Eb, samples_per_bit, sampled_data, t)
% BFSK_BB Generate baseband BFSK time-domain signal
%
% Inputs:
%   bits_Num       - Number of bits per realization
%   N_realization  - Number of realizations
%   Tb             - Bit duration in seconds
%   Eb             - Energy per bit
%
% Output:
%   tx_out         - Baseband BFSK output signal (N_realization x 7*(bits_Num+1))

    % === Derived Parameters ===
    total_samples = samples_per_bit * (bits_Num + 1); % Total samples per realization

    % === Initialize Output Signal ===
    tx_out = zeros(N_realization, total_samples);

    % === Map to Baseband BFSK Signal ===
    for i = 1:N_realization
        for j = 1:samples_per_bit:total_samples
            if sampled_data(i, j) == 0
                tx_out(i, j:j+samples_per_bit-1) = sqrt(2 * Eb / Tb);  % Non-coherent tone for 0
            else
                for k = 1:samples_per_bit
                    tx_out(i, j + k - 1) = sqrt(2 * Eb / Tb) * ...
                        (cos(2 * pi * t(k) / Tb) + 1i * sin(2 * pi * t(k) / Tb));
                end
            end
        end
    end
end

function [tx_with_delay] = apply_random_delay(tx_out, samples_per_bit)
% APPLY_RANDOM_DELAY Applies random symbol-aligned delay to each realization
%
% Inputs:
%   tx_out           - Original signal matrix (N_realization × total_samples)
%   samples_per_bit  - Number of samples per bit (e.g., 7)
%
% Output:
%   tx_with_delay    - Delayed signals, trimmed to same size (N_realization × trimmed_samples)

    [N_realization, total_samples] = size(tx_out);
    trimmed_samples = total_samples - samples_per_bit;
    tx_with_delay = zeros(N_realization, trimmed_samples);

    for i = 1:N_realization
        r = randi([0 (samples_per_bit - 1)]);  % Random delay in samples
        tx_with_delay(i, :) = tx_out(i, r + 1 : r + trimmed_samples);
    end
end

function BFSK_autocorr = compute_BFSK_autocorrelation(tx_with_delay)
% COMPUTE_BFSK_AUTOCORRELATION Computes autocorrelation of delayed BFSK signals
% centered at the middle sample.
%
% Input:
%   tx_with_delay   - Matrix of delayed BFSK signals (N_realization × N_samples)
%
% Output:
%   BFSK_autocorr   - Autocorrelation vector (1 × N_samples)

    [~, N_samples] = size(tx_with_delay);

    % Ensure N_samples is even for symmetric range
    if mod(N_samples, 2) ~= 0
        error('N_samples must be even for symmetric autocorrelation.');
    end

    BFSK_autocorr = zeros(1, N_samples);
    center_idx = N_samples / 2;

    for j = -center_idx+1 : center_idx
        i = j + center_idx;
        if i >= 1 && i <= N_samples
            p = conj(tx_with_delay(:, center_idx)) .* tx_with_delay(:, i);
            BFSK_autocorr(i) = sum(p) / length(p);
        end
    end
end
```

```matlab
function draw_autocorr(Rx_BFSK)
% DRAW_AUTOCORR Plots the magnitude of the symmetric autocorrelation
%
% Input:
%   Rx_BFSK - 1 × N vector of autocorrelation values (only one-sided)

    N = length(Rx_BFSK);
    tau = (-N+1):(N-1);

    % plot the graph
    figure('Name', 'Autocorrelation');
    plot(tau-N/2, abs(fliplr([Rx_BFSK Rx_BFSK(2:end)])), 'LineWidth', 1.5);
    xlabel('\tau');
    ylabel('Autocorrelation Magnitude');
    xlim([-50 50]);
    title('BFSK Autocorrelation');
    grid on;
end

function draw_psd(f_normalized, BFSK_PSD, PSD_theoretical)
% DRAW_PSD Plots the practical and theoretical PSD of a BFSK signal
%
% Inputs:
%   f_normalized     - Frequency axis (normalized by bit rate)
%   BFSK_PSD         - Practical PSD values (1 × N)
%   PSD_theoretical  - Theoretical PSD values (1 × N), aligned with f_normalized

    figure('Name', 'PSD');
    plot(f_normalized, abs(BFSK_PSD) / 100, 'b', 'LineWidth', 1);   % Practical PSD
    hold on;
    plot(f_normalized + 0.5, abs(PSD_theoretical), 'r--', 'LineWidth', 1); % Shifted theoretical PSD
    hold off;

    xlabel('Normalized Frequency, fTb');
    ylabel('S(f)');
    title('BFSK PSD');
    xlim([-1.5 1.5]);
    ylim([0 2]);
    legend('Practical', 'Theoretical');
    grid on;
end
```