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:

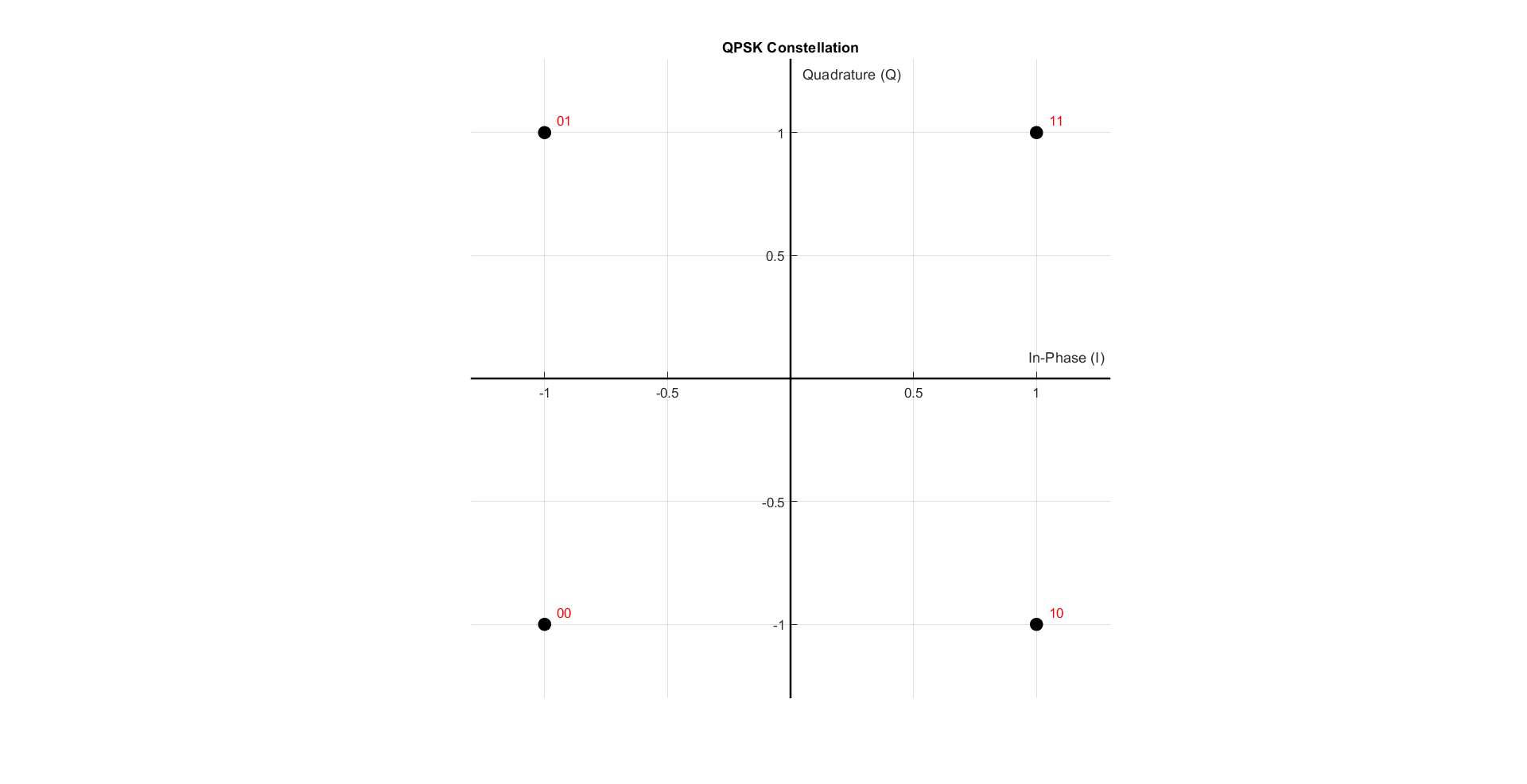


Figure 1 BPSK constellation

Figure 2 QPSK constellation

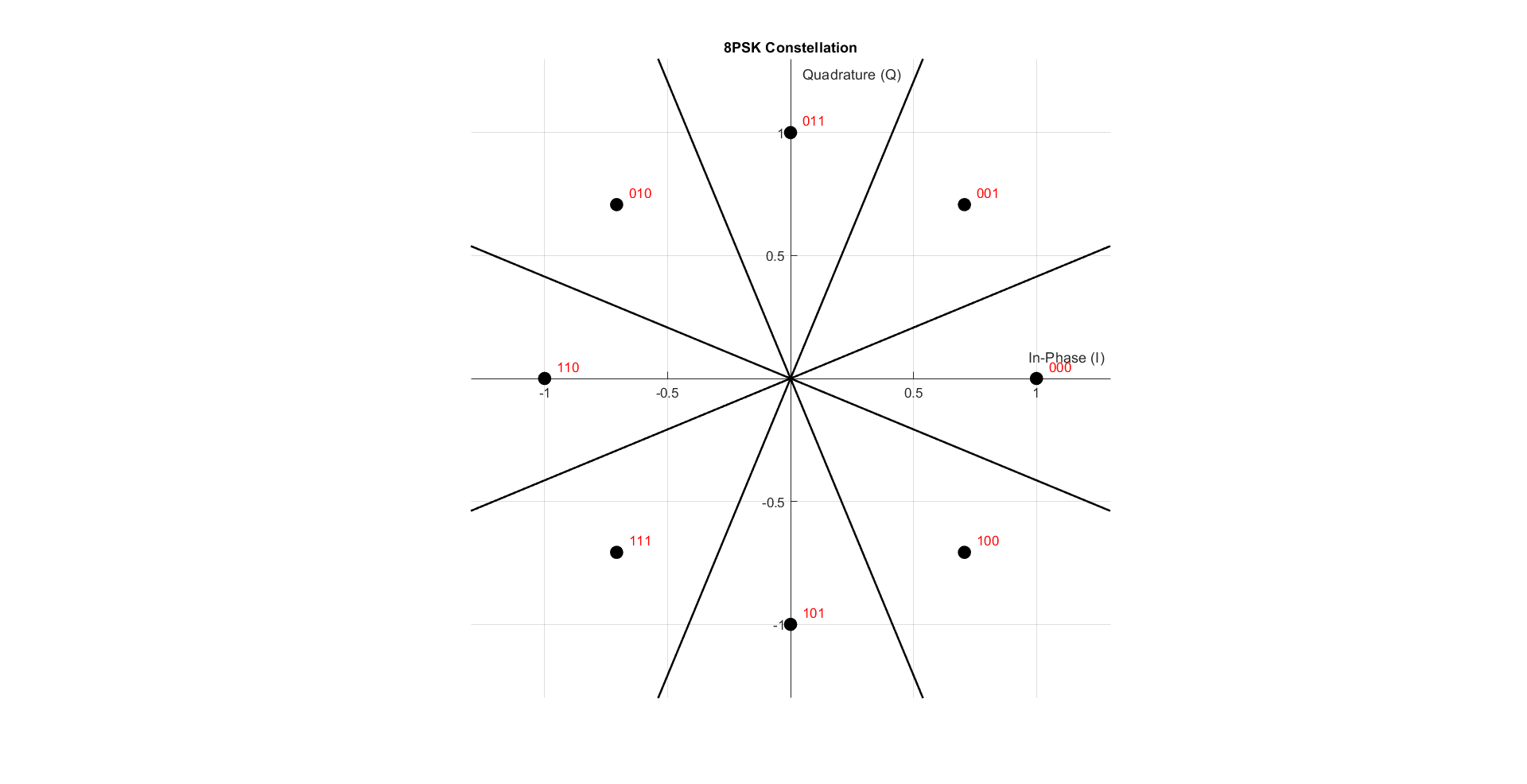
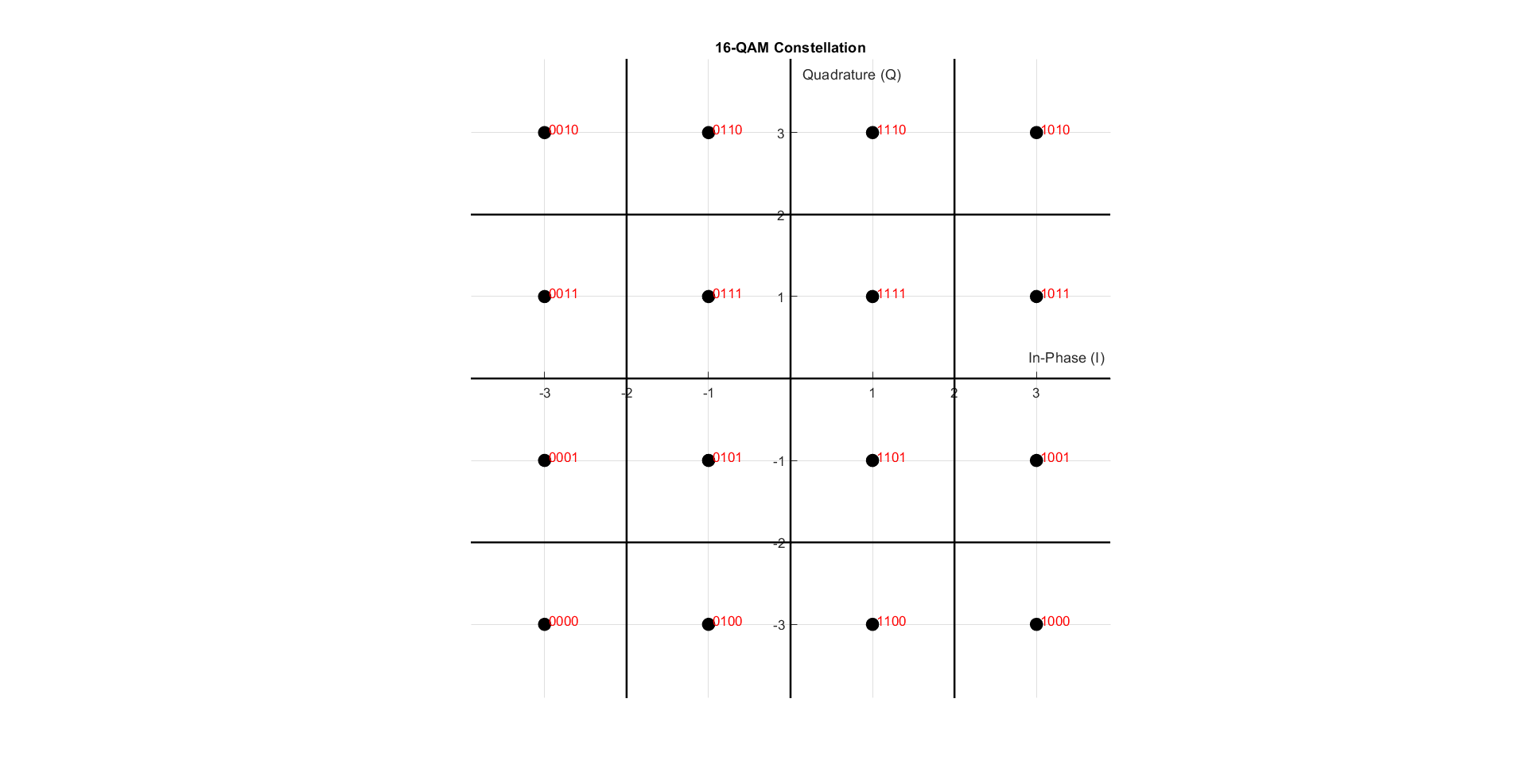


Figure 3 16-QAM constellation

Figure 4 8PSK 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

XBB = XI + j XQ

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

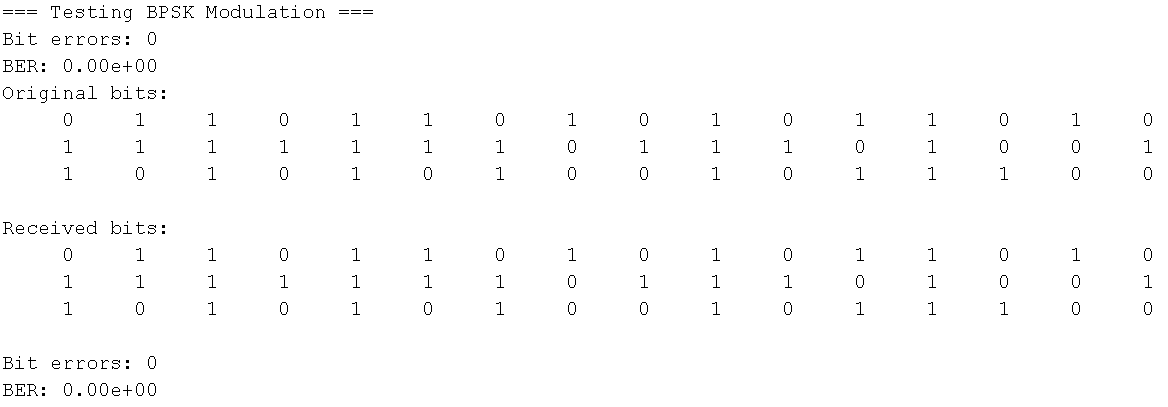
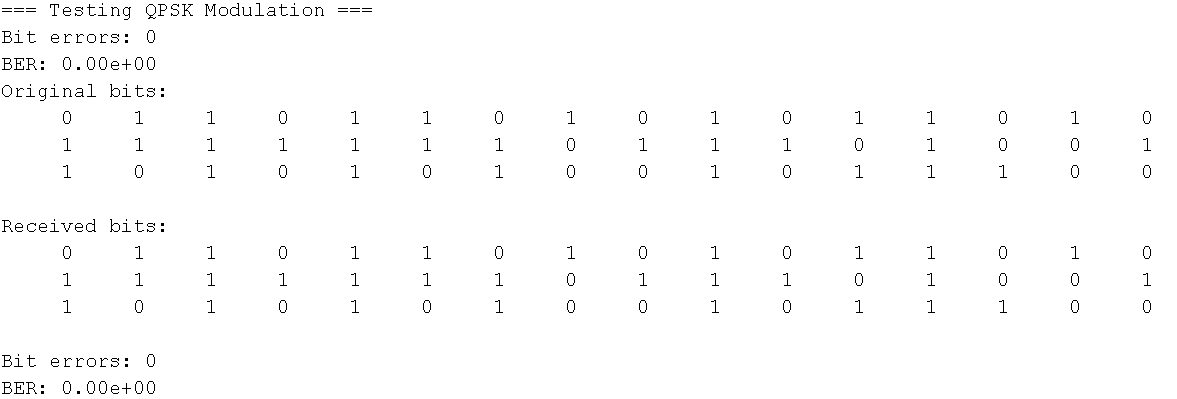
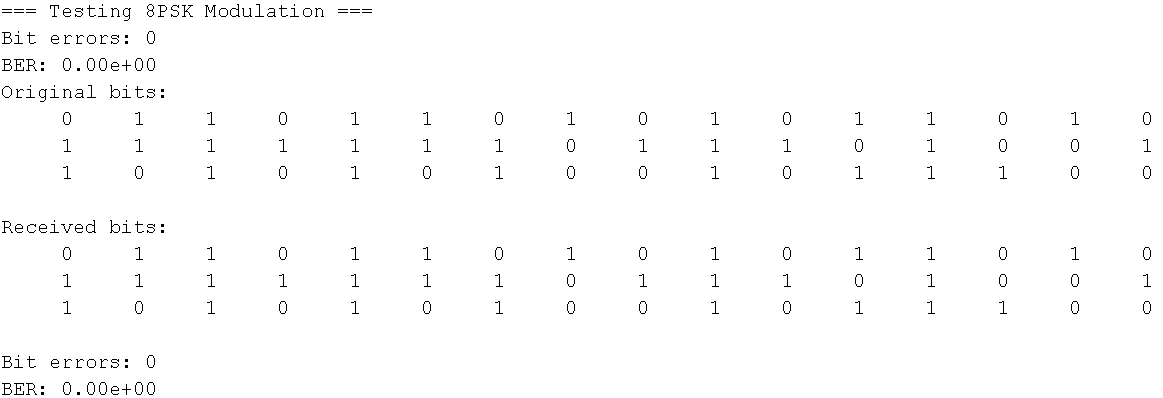
Results:

Figure 5 8PSK Test

Figure 6 QPSK Test

Figure 7 BPSK Test

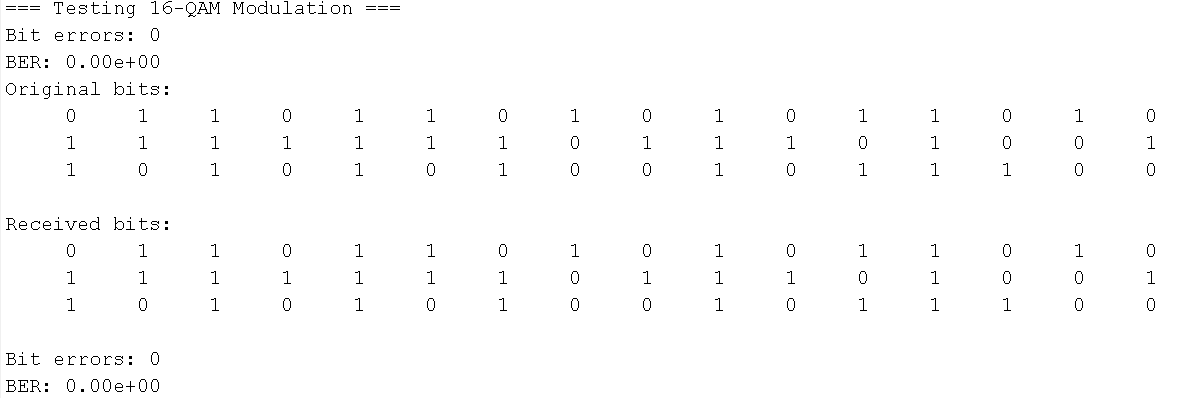


Figure 8 16-QAM Test

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:

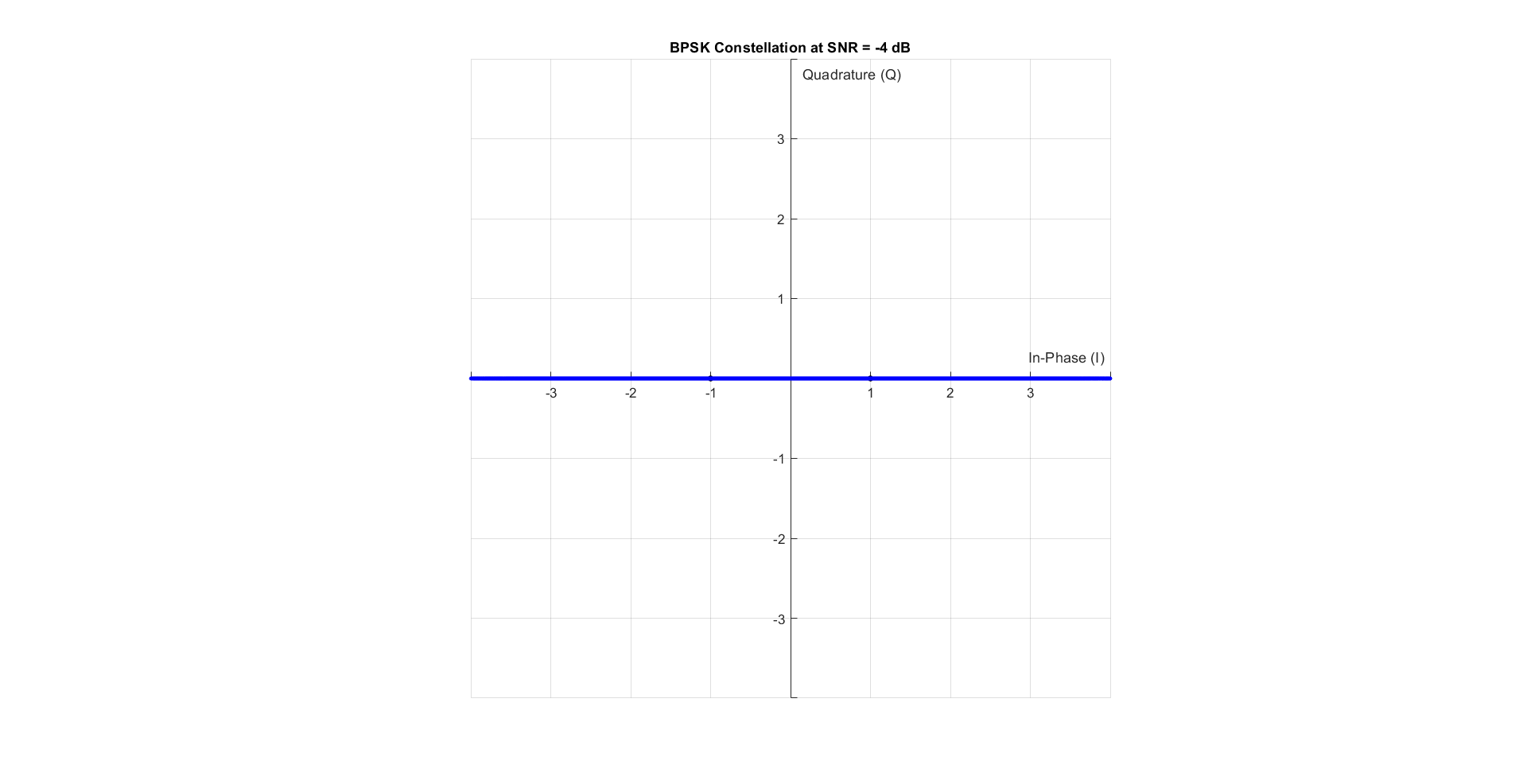
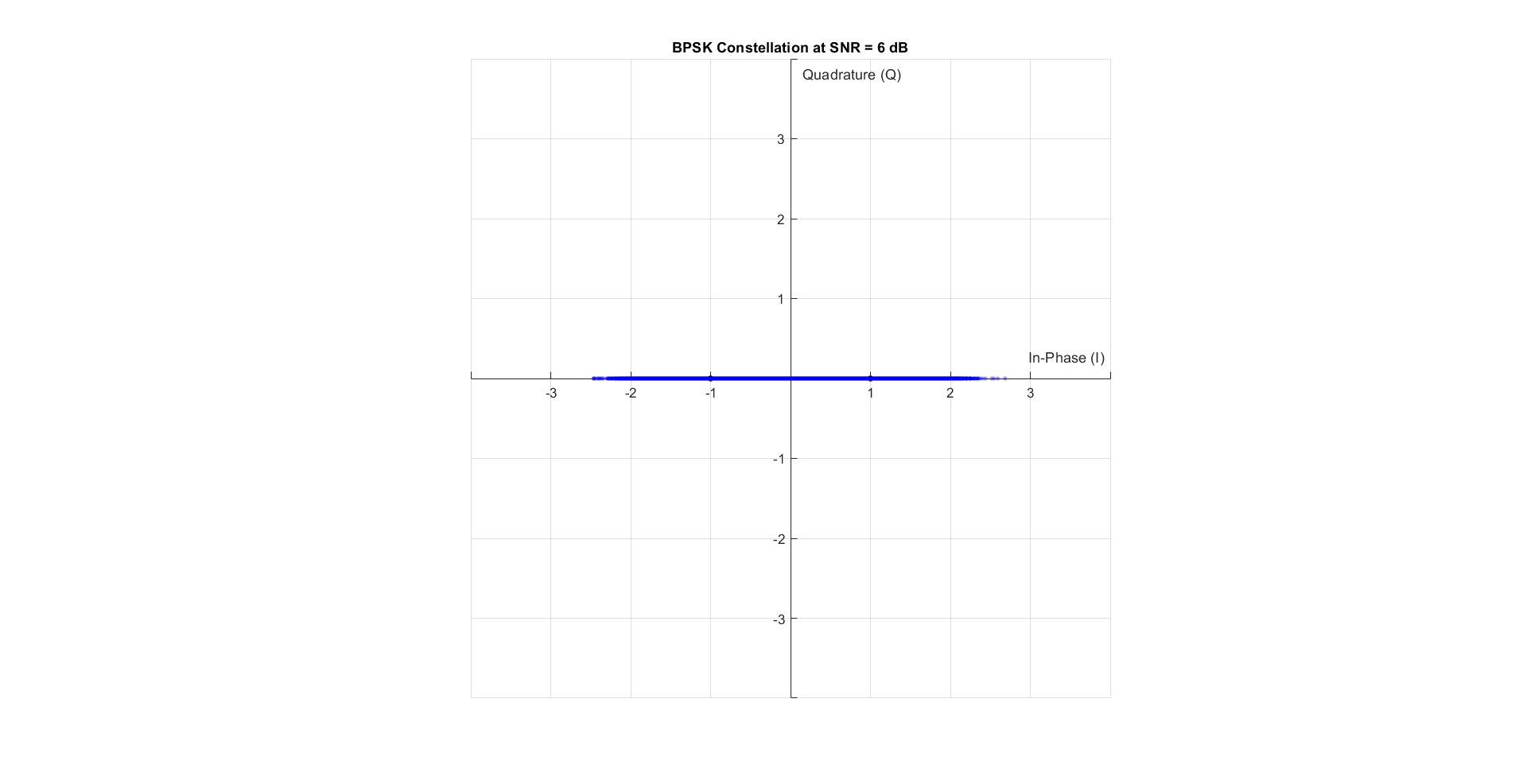
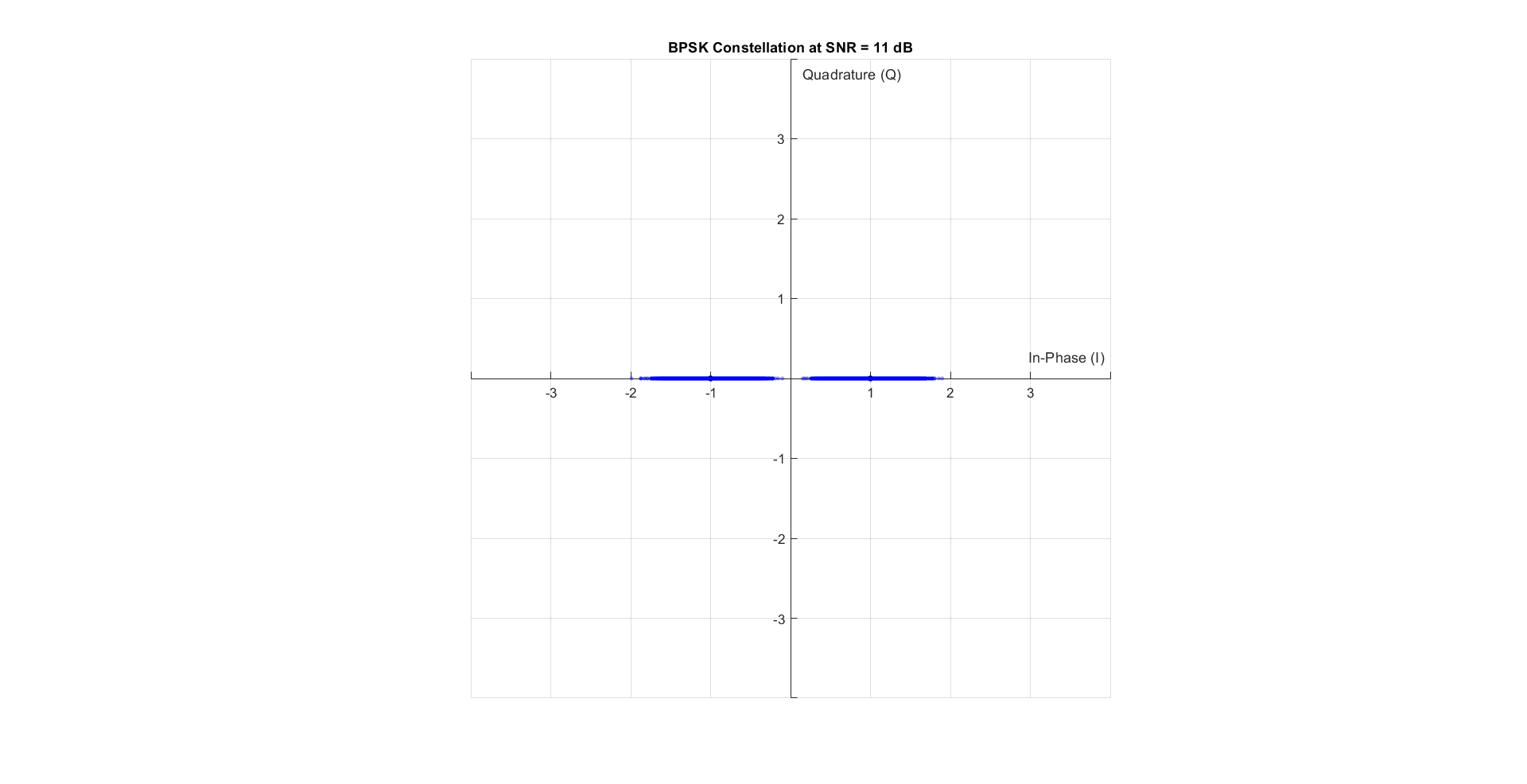
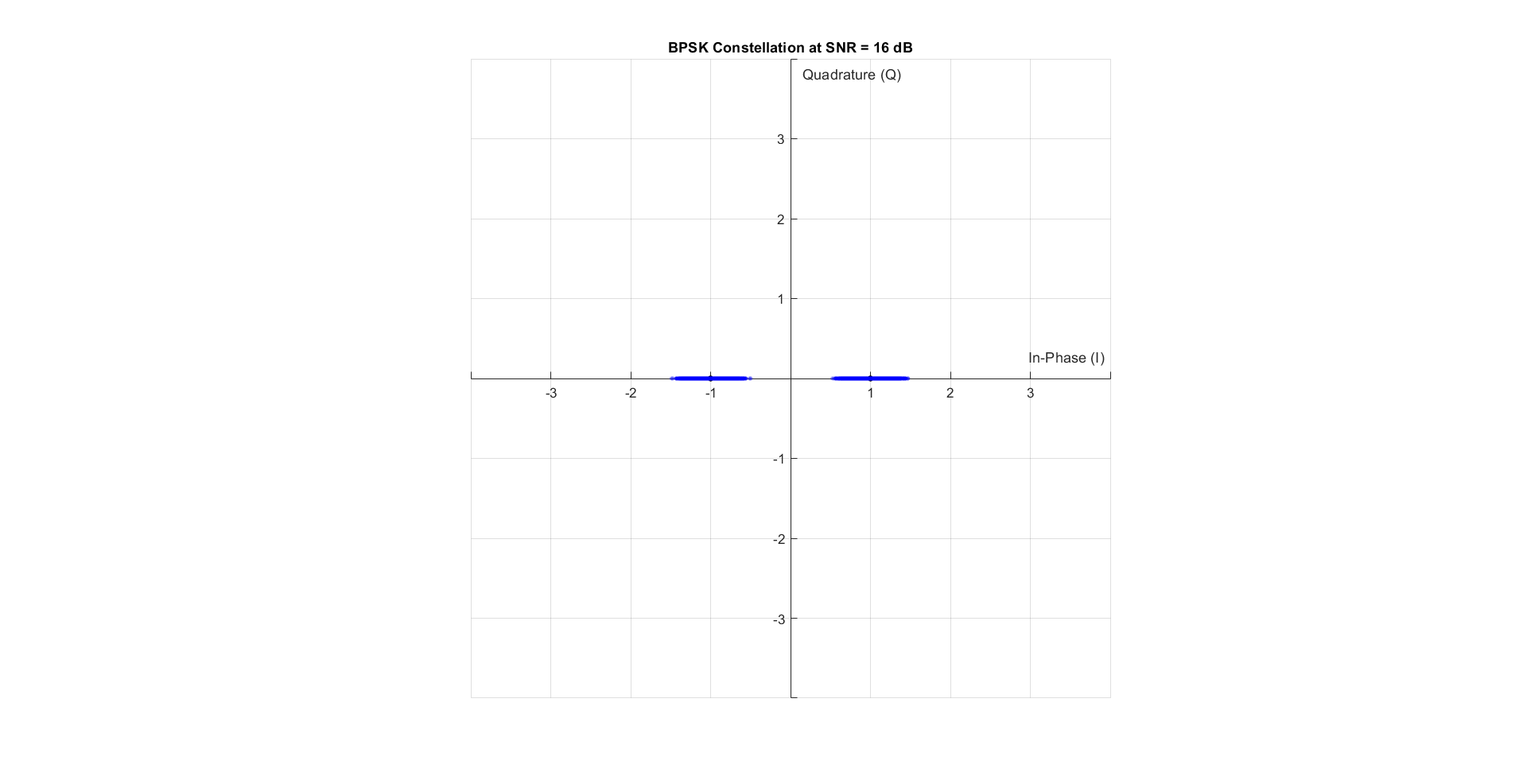


Figure 9 Noise on BPSK with SNR = -4 dB

Figure 10 Noise on BPSK with SNR = 6 dB

Figure 11 Noise on BPSK with SNR = 16 dB

Figure 12 Noise on BPSK with SNR = 11 dB

QPSK:

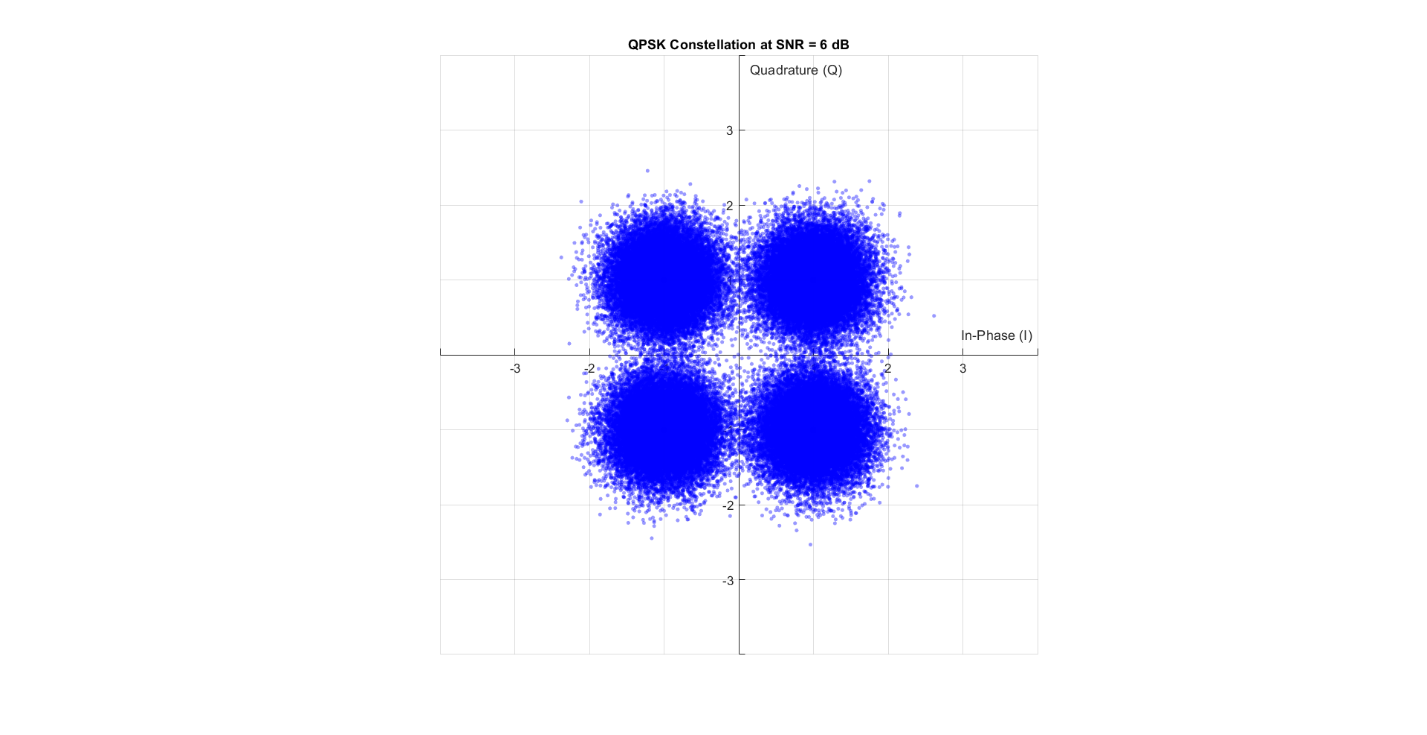
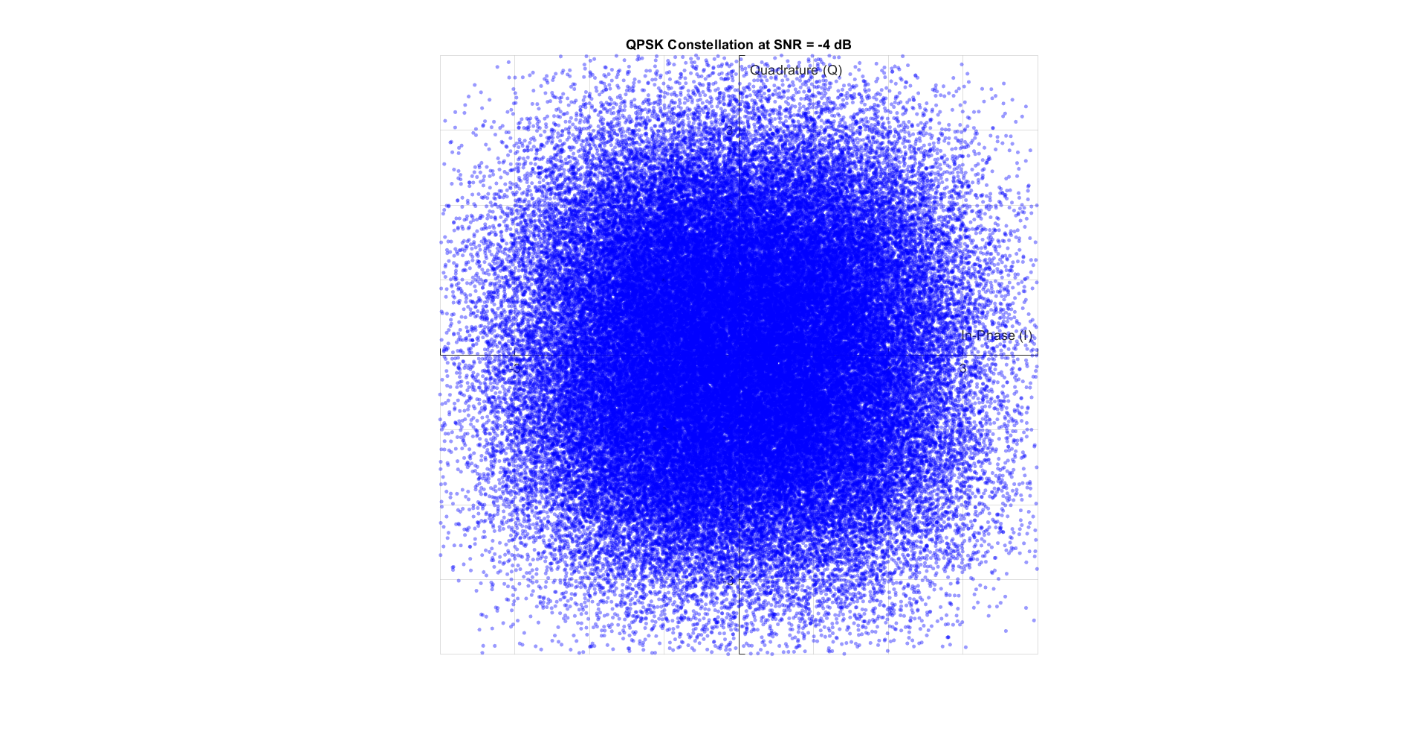


Figure 13 Noise on QPSK with SNR = -4 dB

Figure 14 Noise on QPSK with SNR = 6 dB

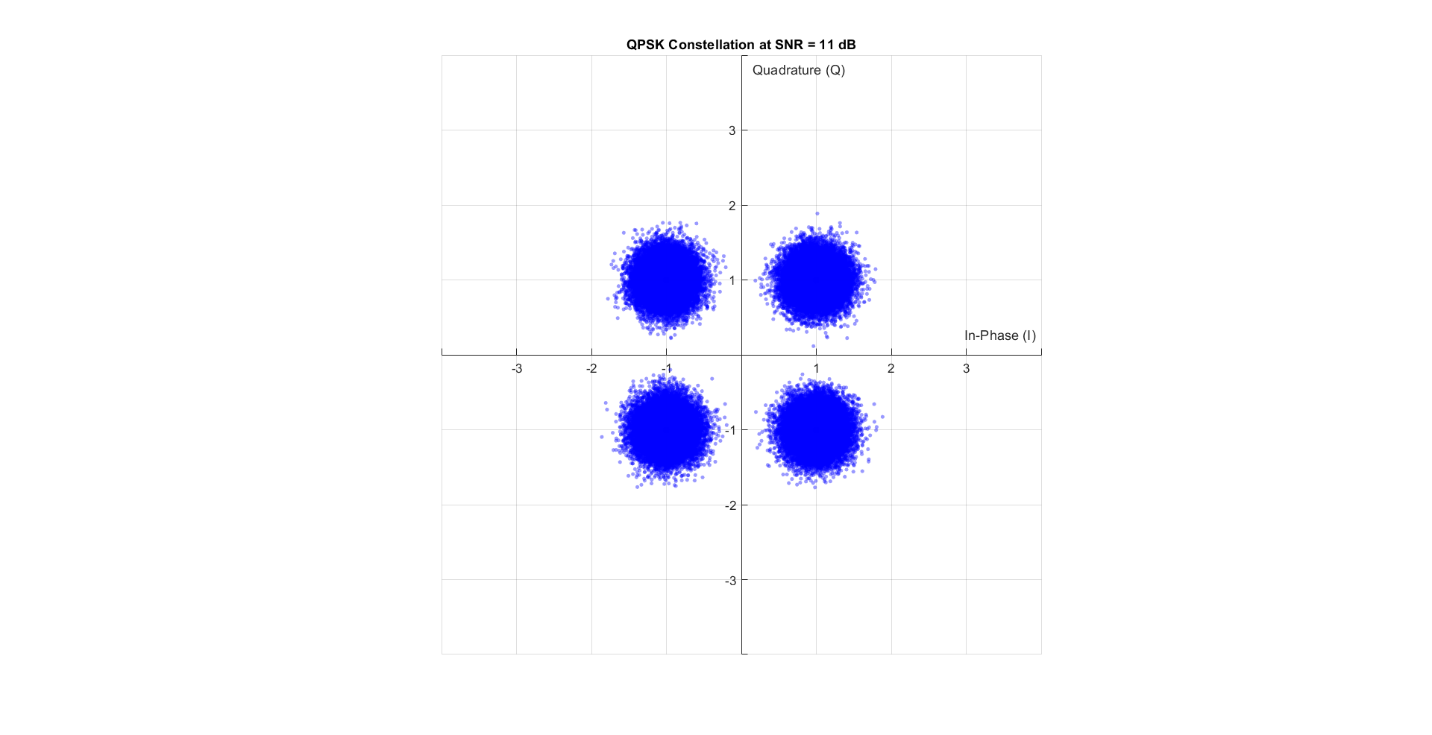
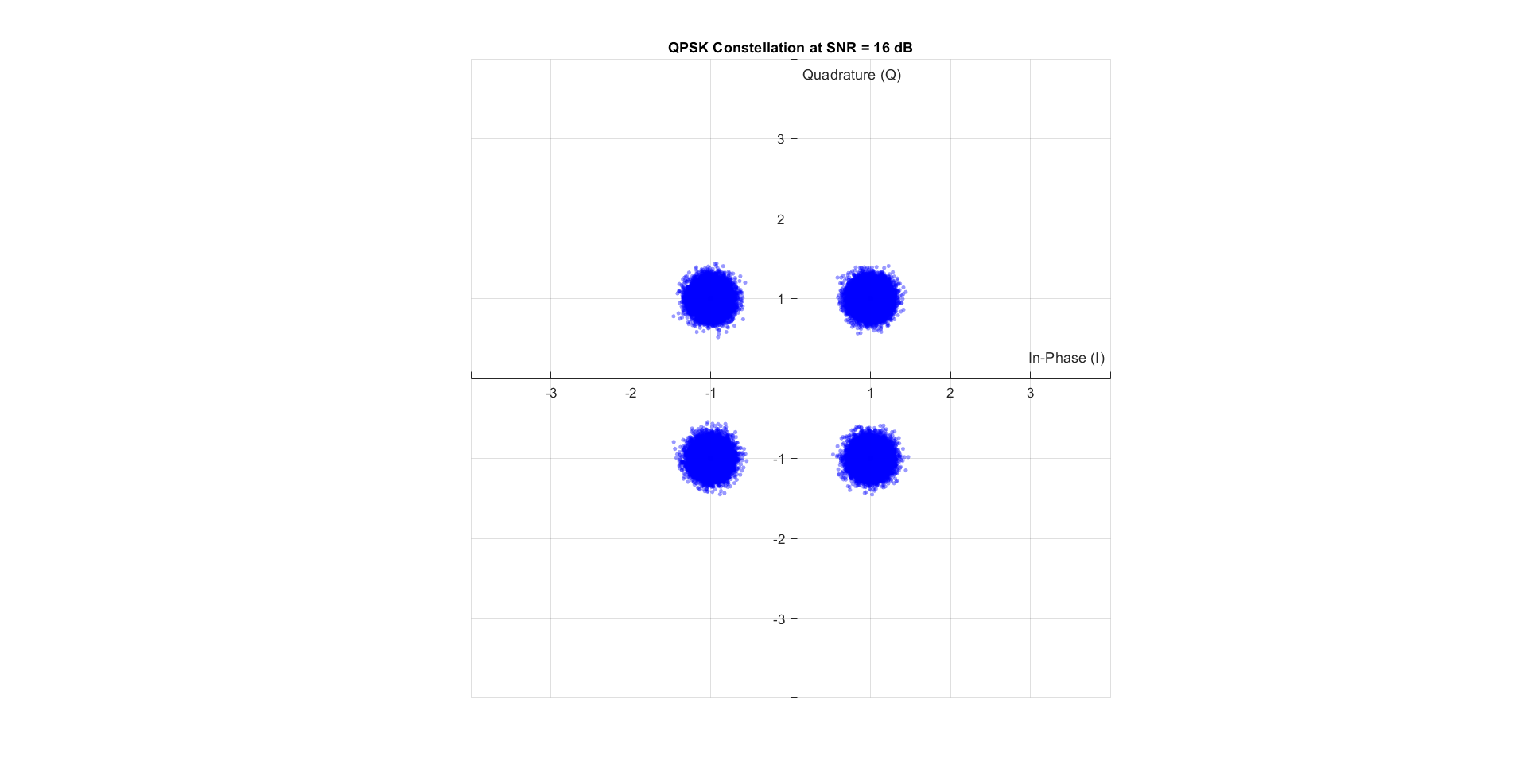


Figure 15 Noise on QPSK with SNR = 16 dB

Figure 16 Noise on QPSK with SNR = 11 dB

8PSK:

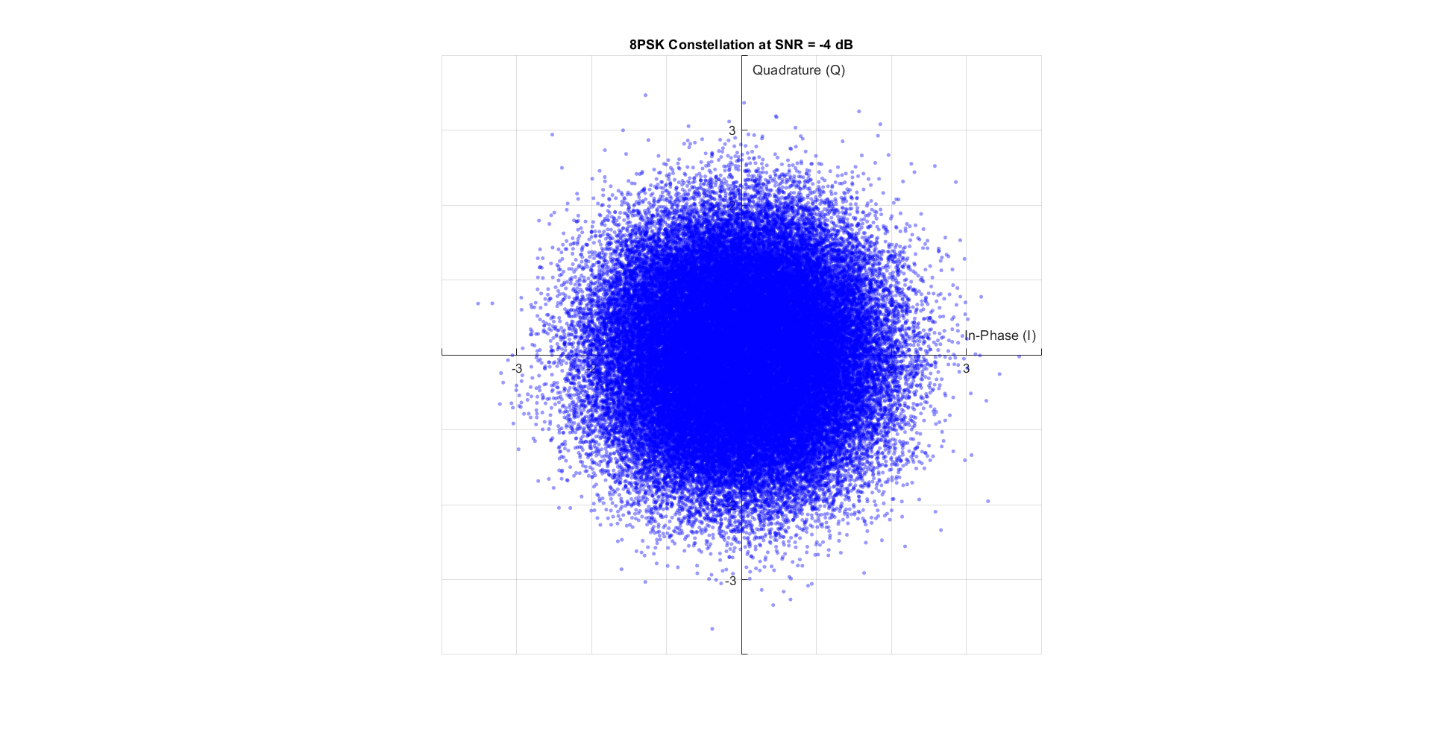


Figure 17 Noise on 8PSK with SNR = -4 dB



Figure 18 Noise on 8PSK with SNR = 6 dB

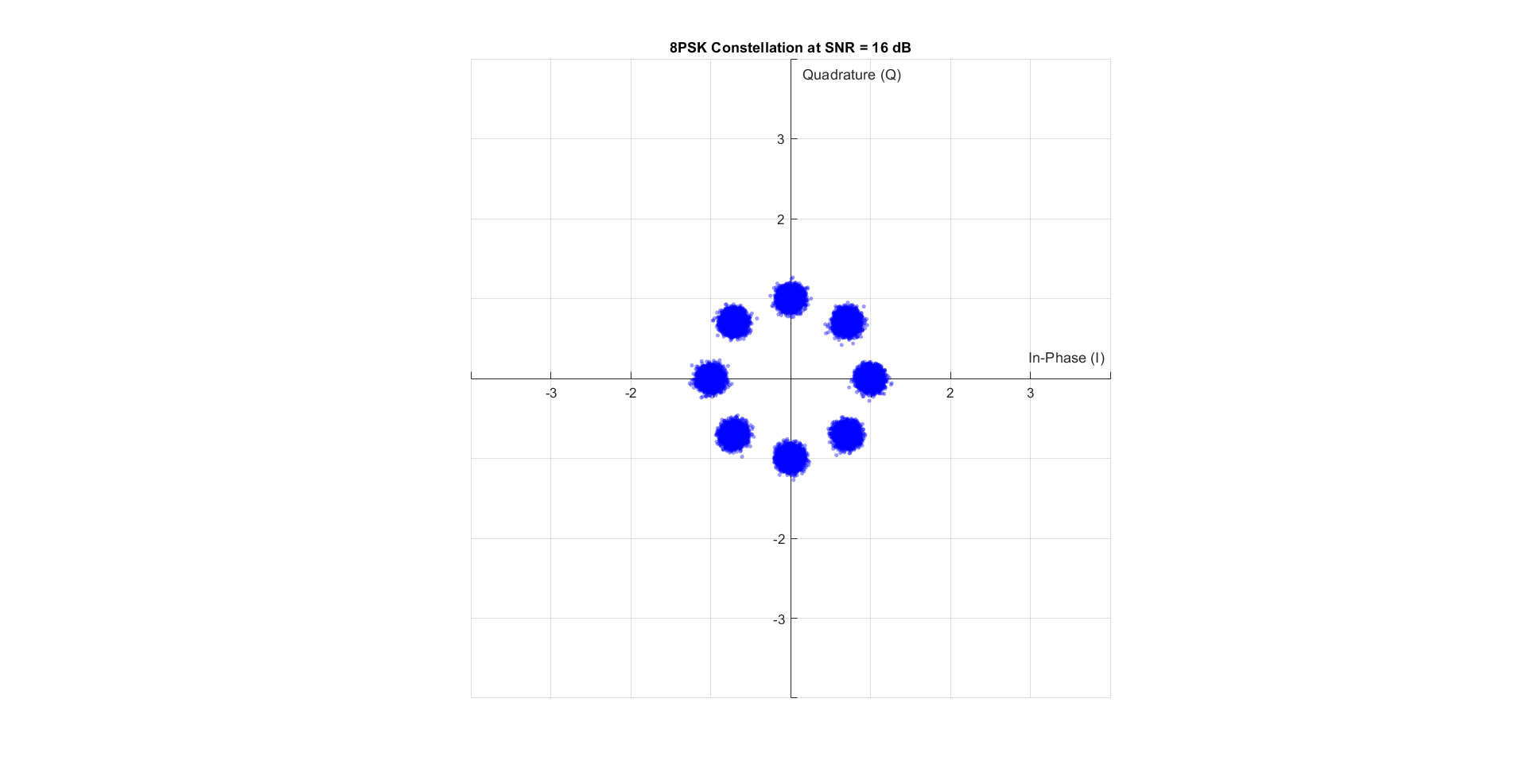
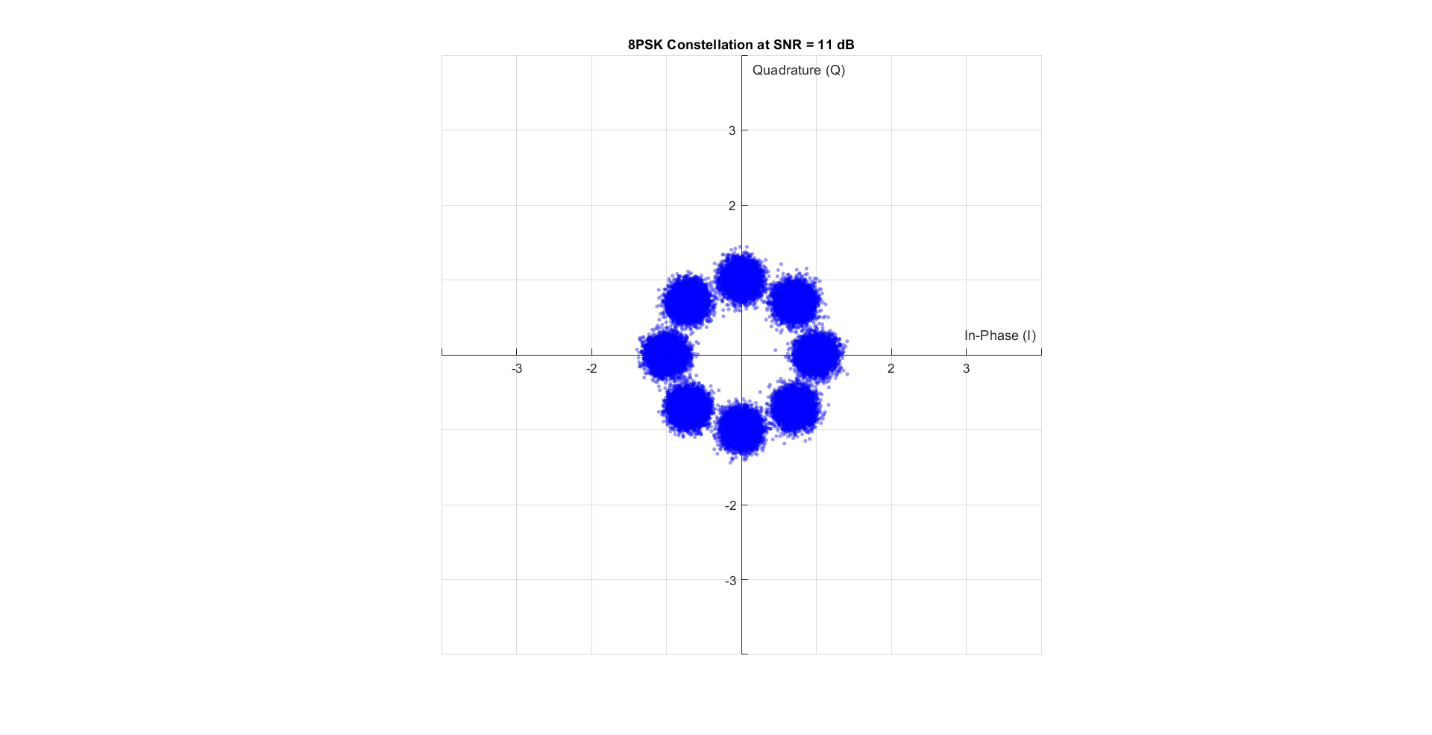


Figure 19 Noise on 8PSK with SNR = 11 dB

Figure 20 Noise on 8PSK with SNR = 16 dB

16QAM:

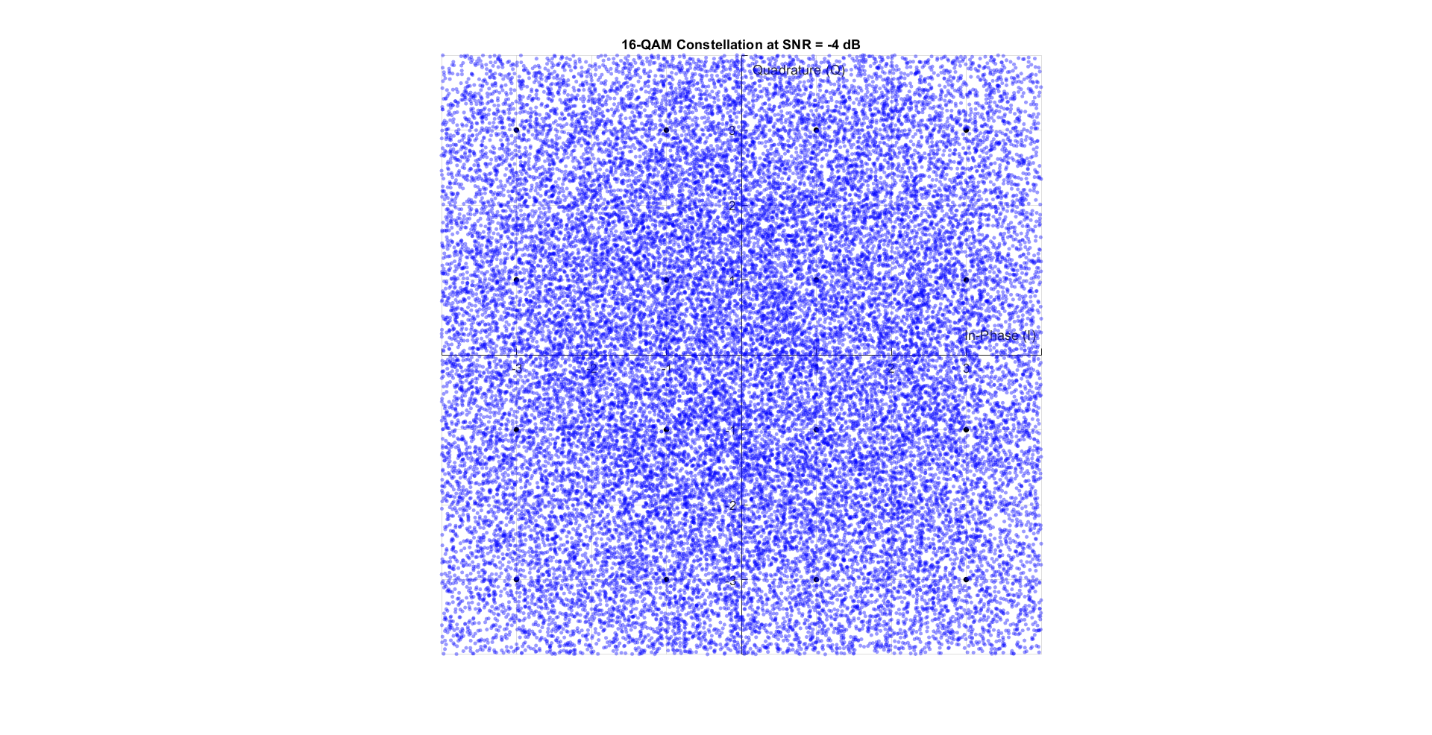
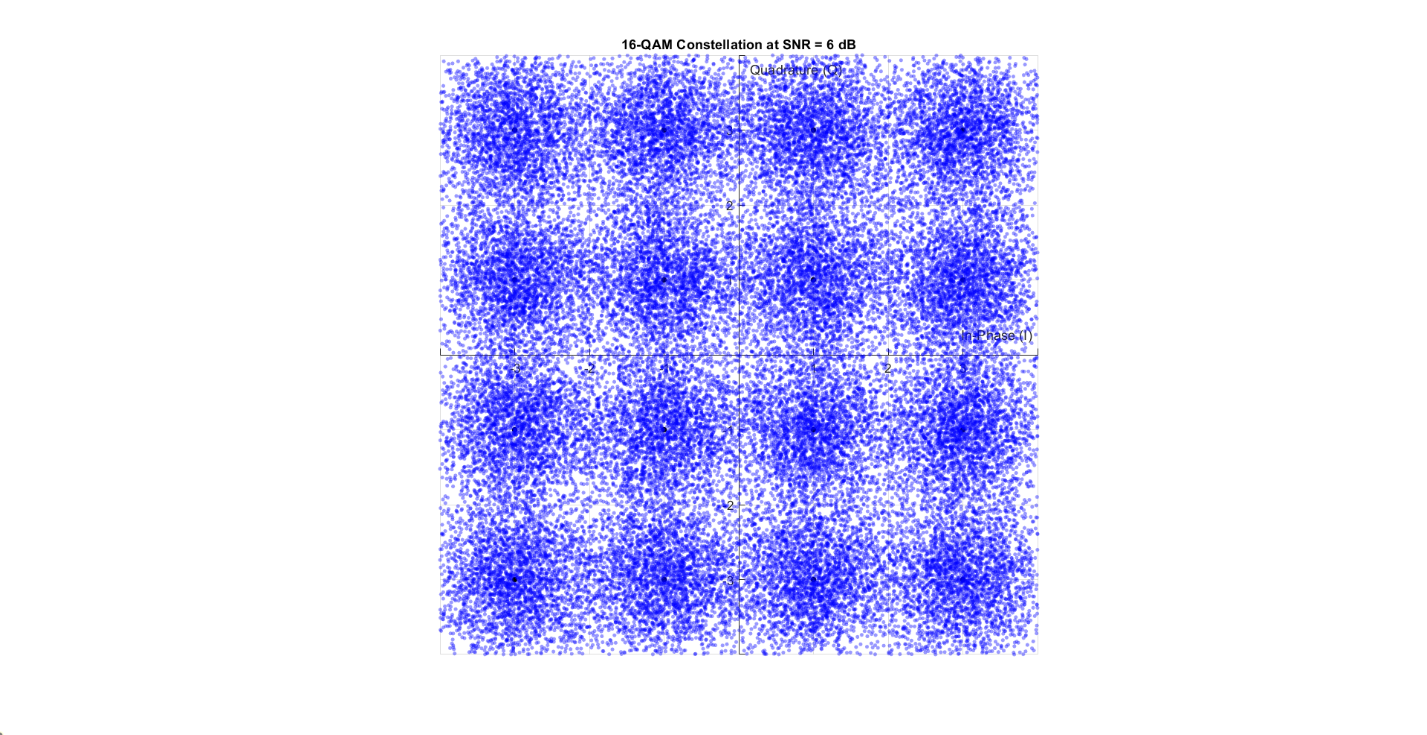


Figure 21 Noise on 16QAM with SNR = 1 dB

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

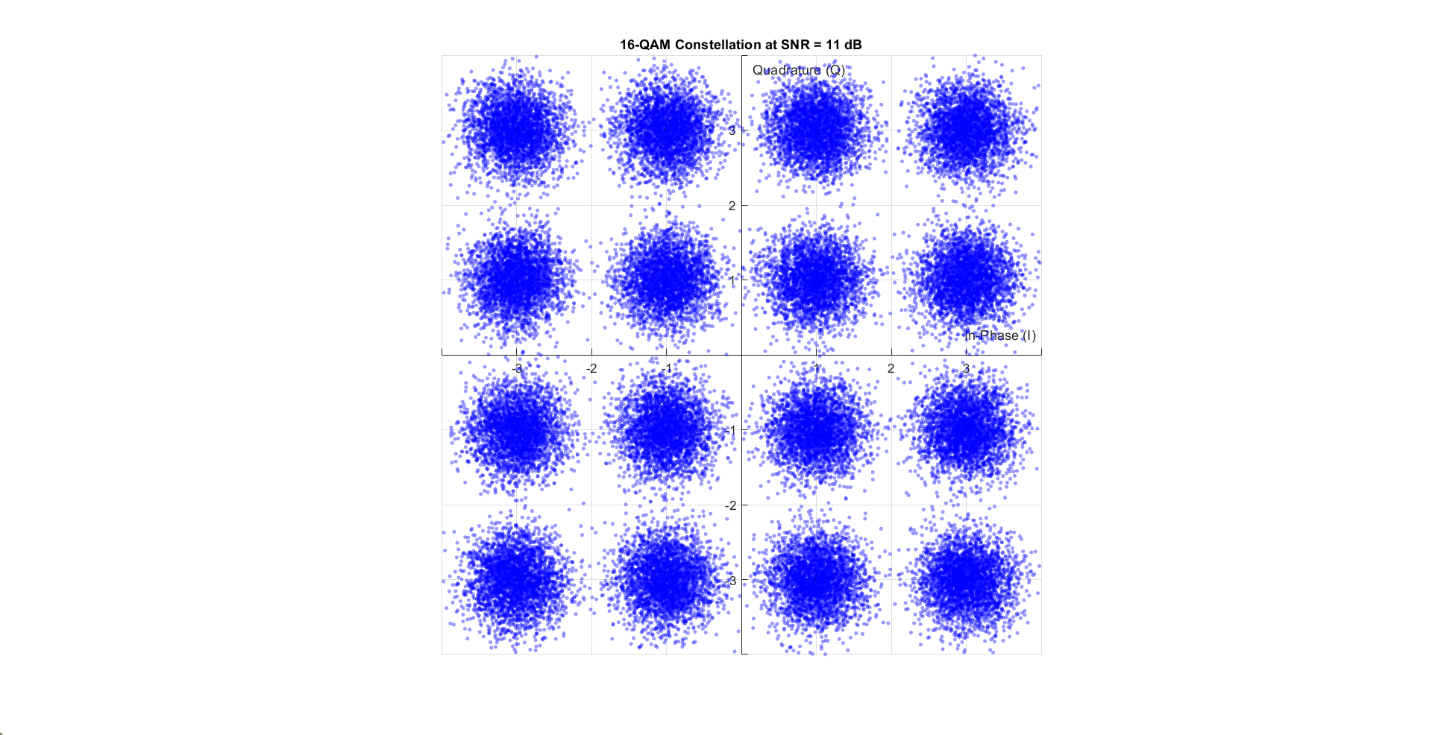
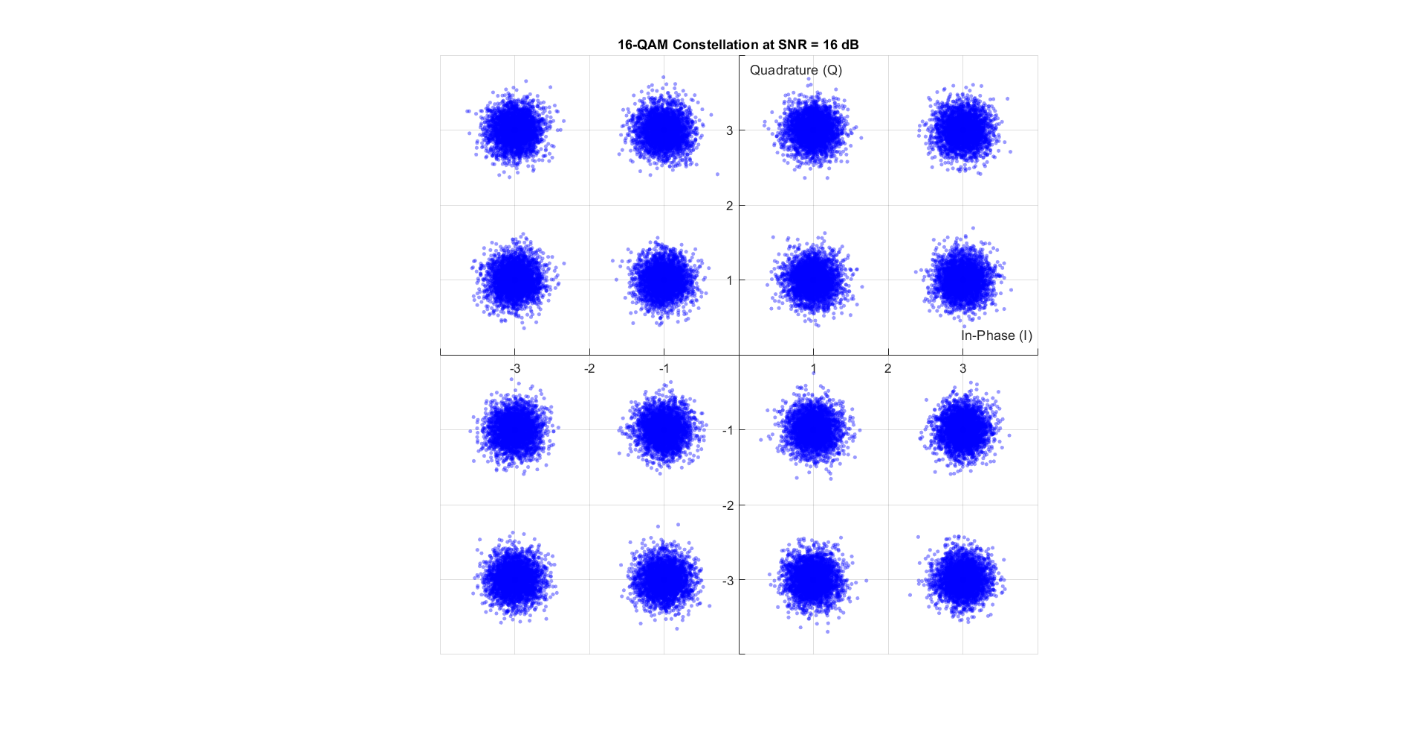


Figure 23 Noise on 16QAM with SNR = 16 dB

Figure 24 Noise on 16QAM with SNR = 11 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 25 Simulated vs Theoretical BER for 8PSK

Figure 26 Simulated vs Theoretical BER for 16QAM

Figure 27 Simulated vs Theoretical BER for QPSK

Figure 28 Simulated vs Theoretical BER for BPSK

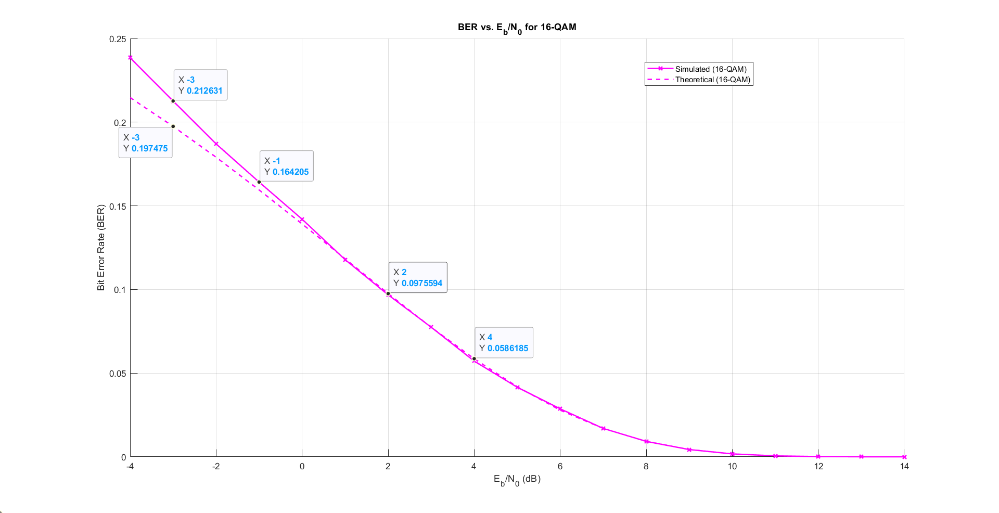
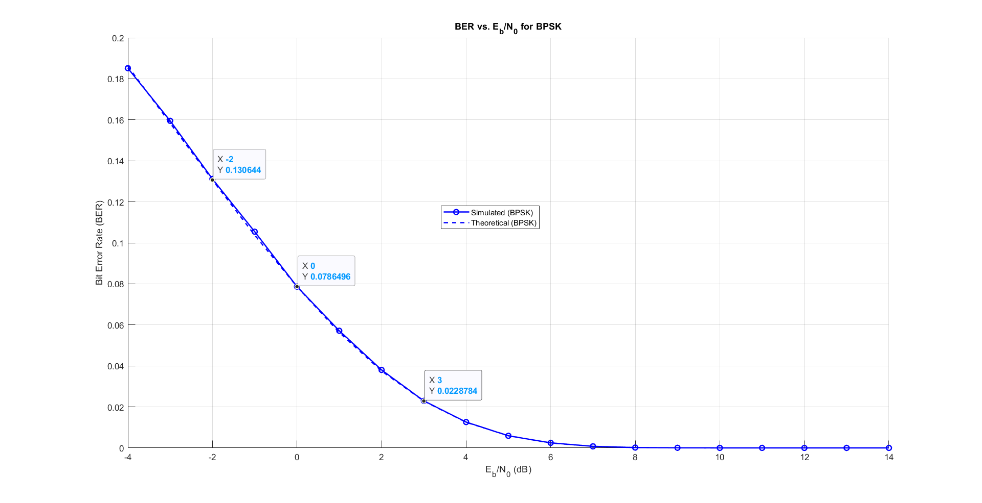
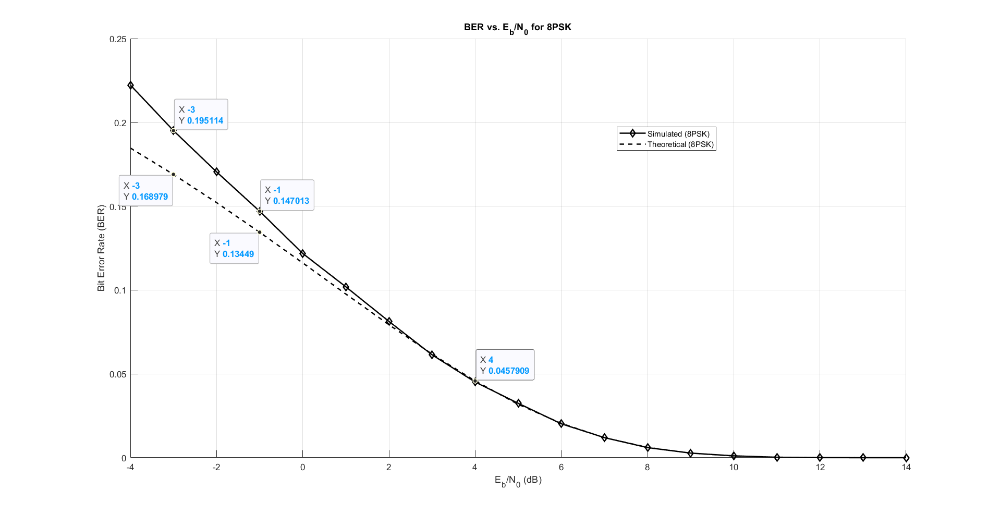
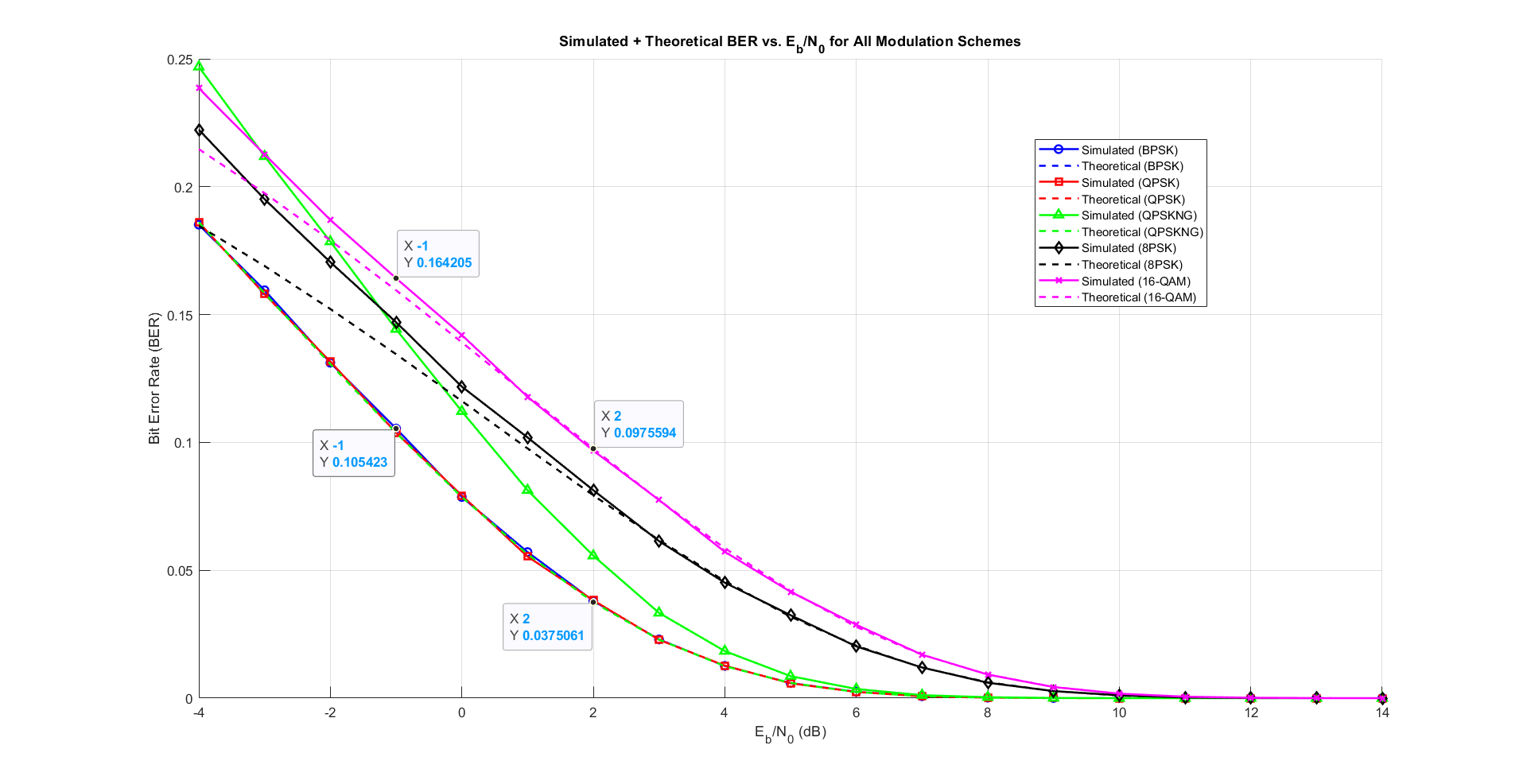


Figure 29 Simulated and Theoritical BER for BPSK, QPSK, 8PSK and 16QAM

Task 2

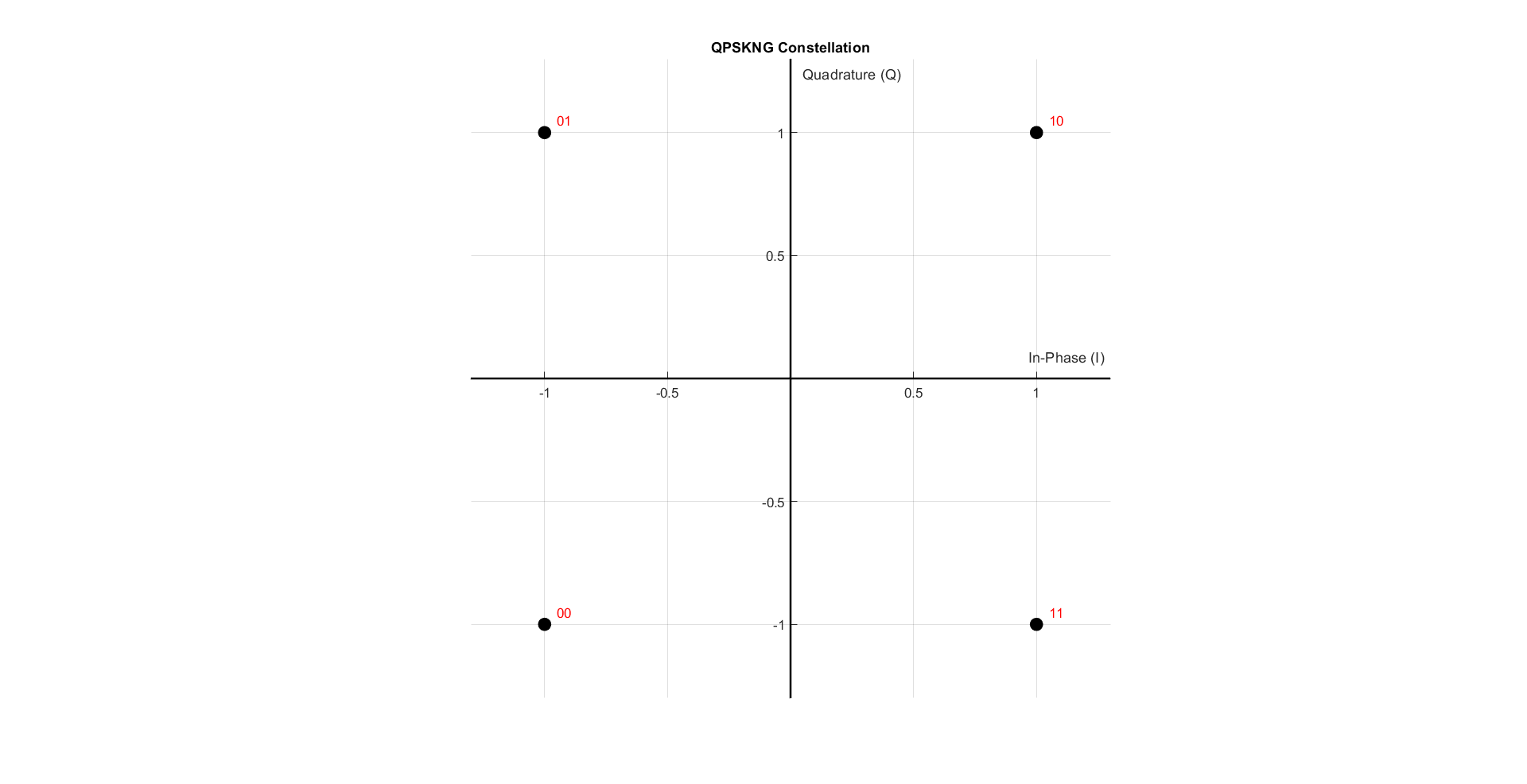
QPSK not Grey

Figure 30 QPSKNG constellation

Output:

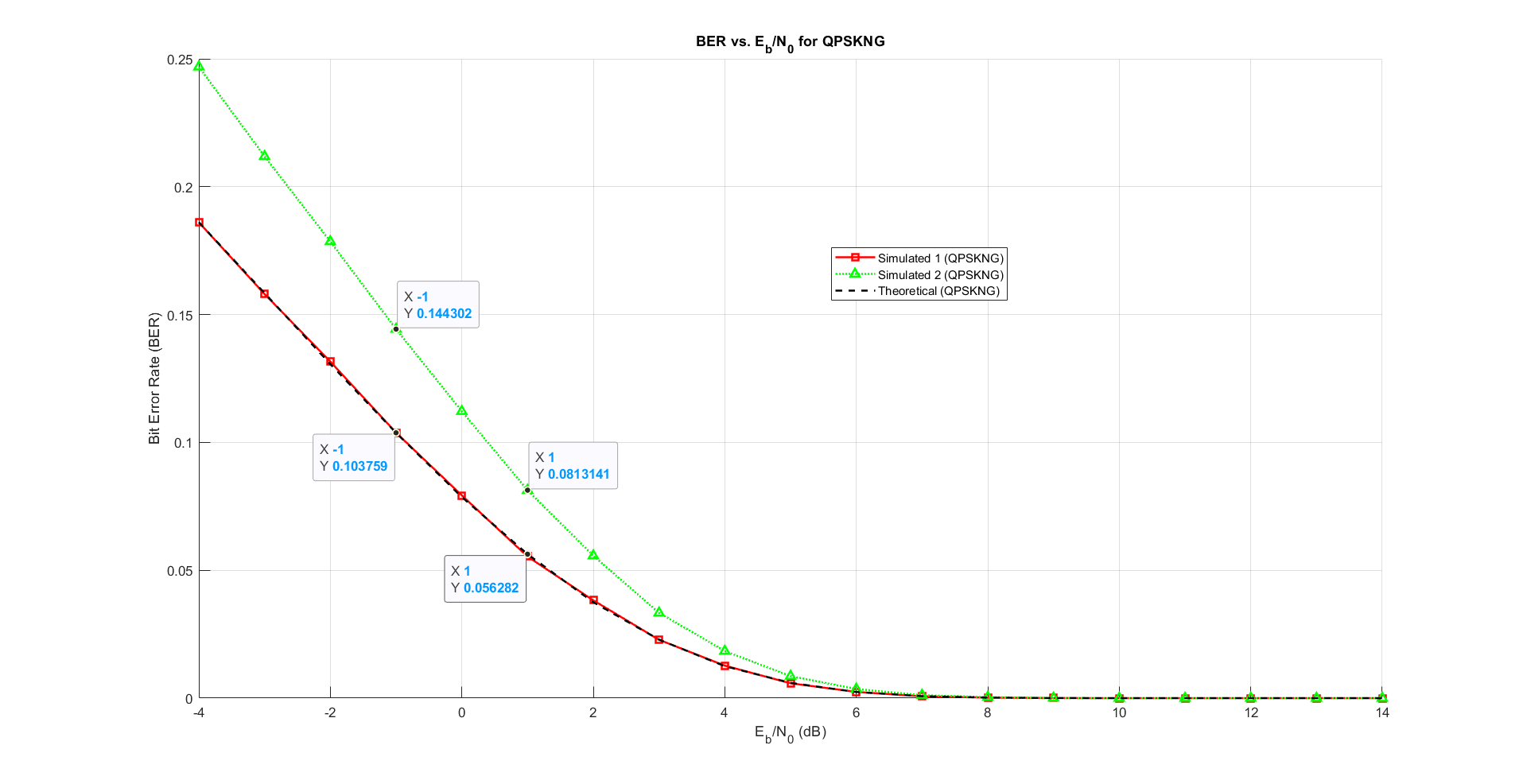


Figure 31 QPSK vs QPSKNG BER

Part 2:

BFSK:

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

Code:

% ========================

% 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 BFSK constellation

Noise:

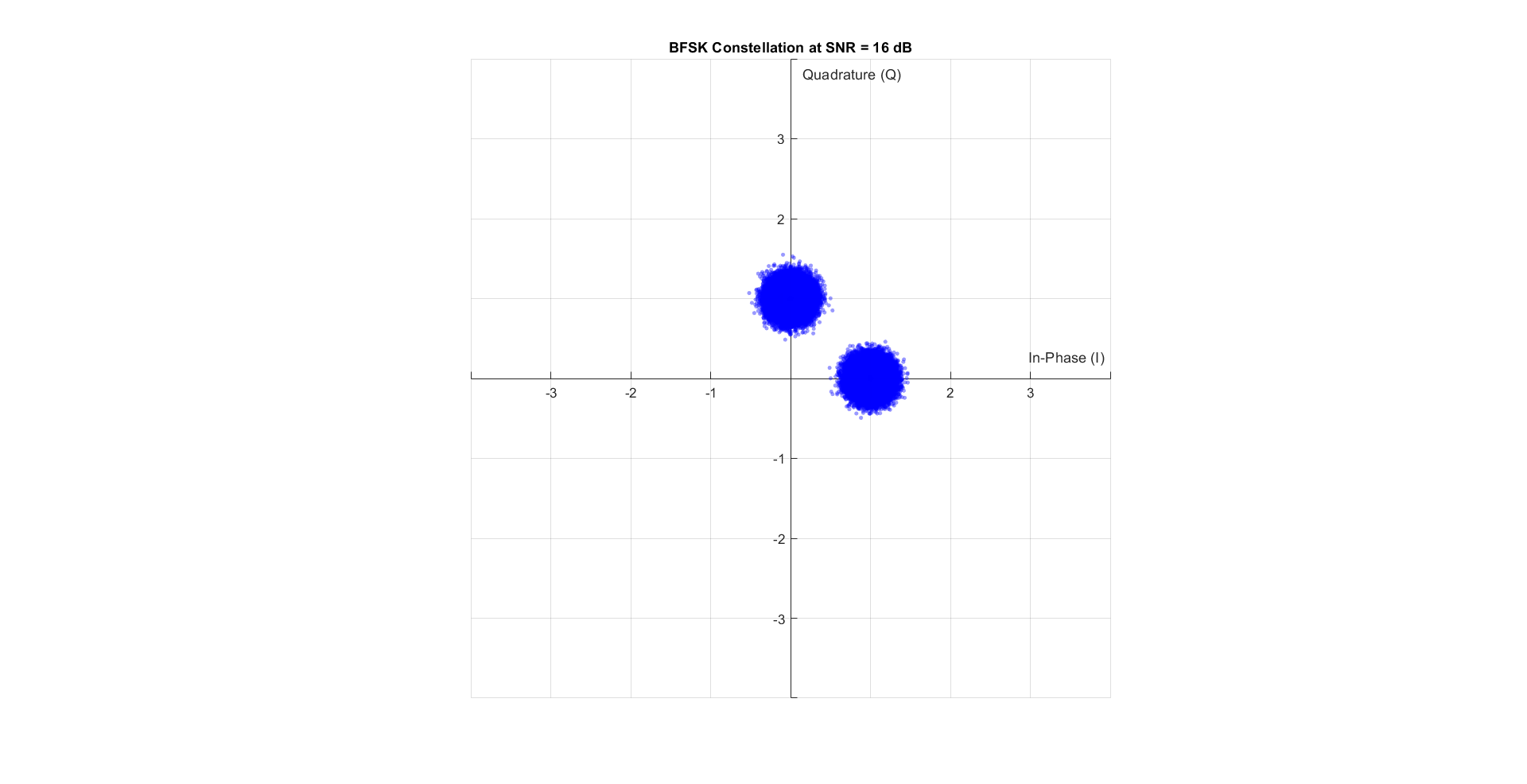
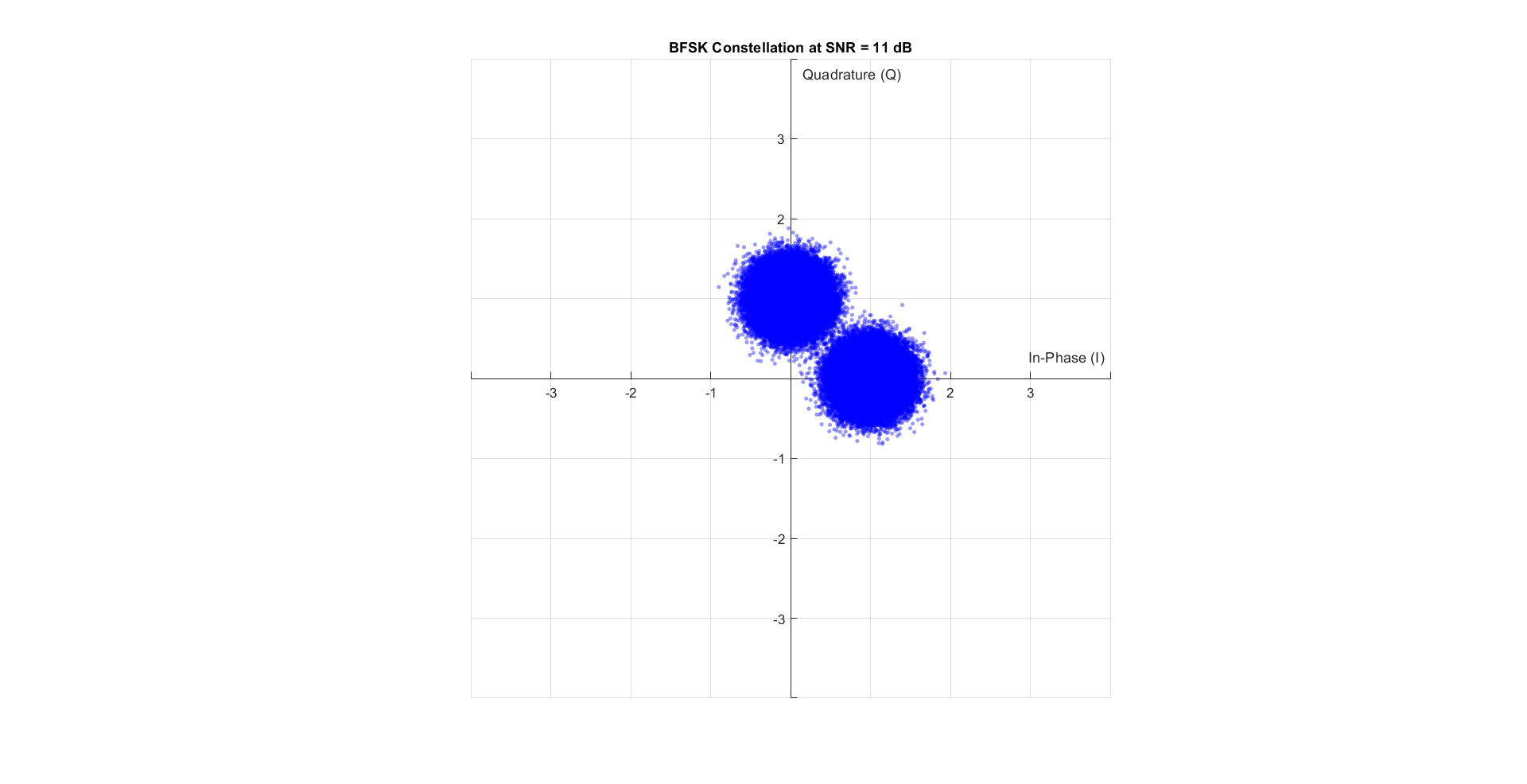


Figure Noise on BFSK with SNR = 16 dB

Figure Noise on BFSK with SNR = 11 dB

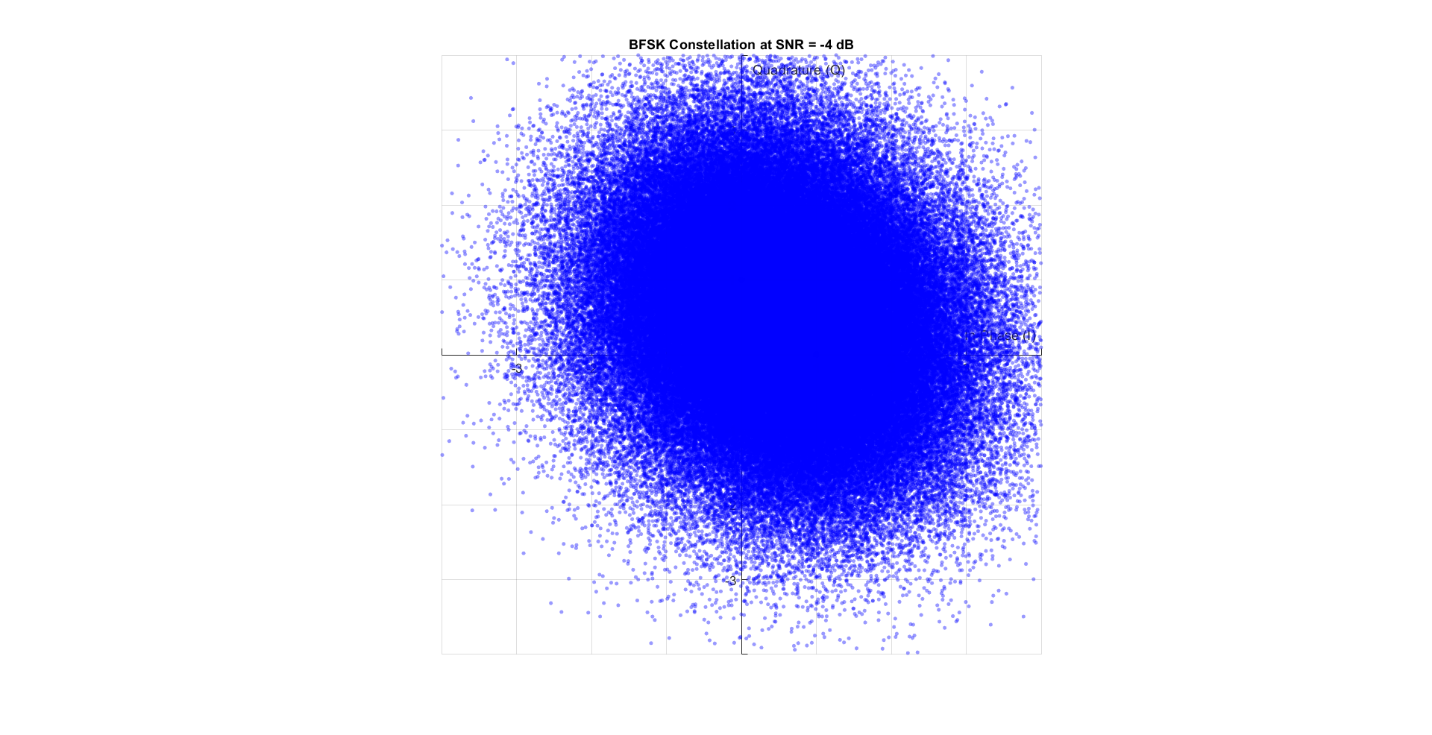
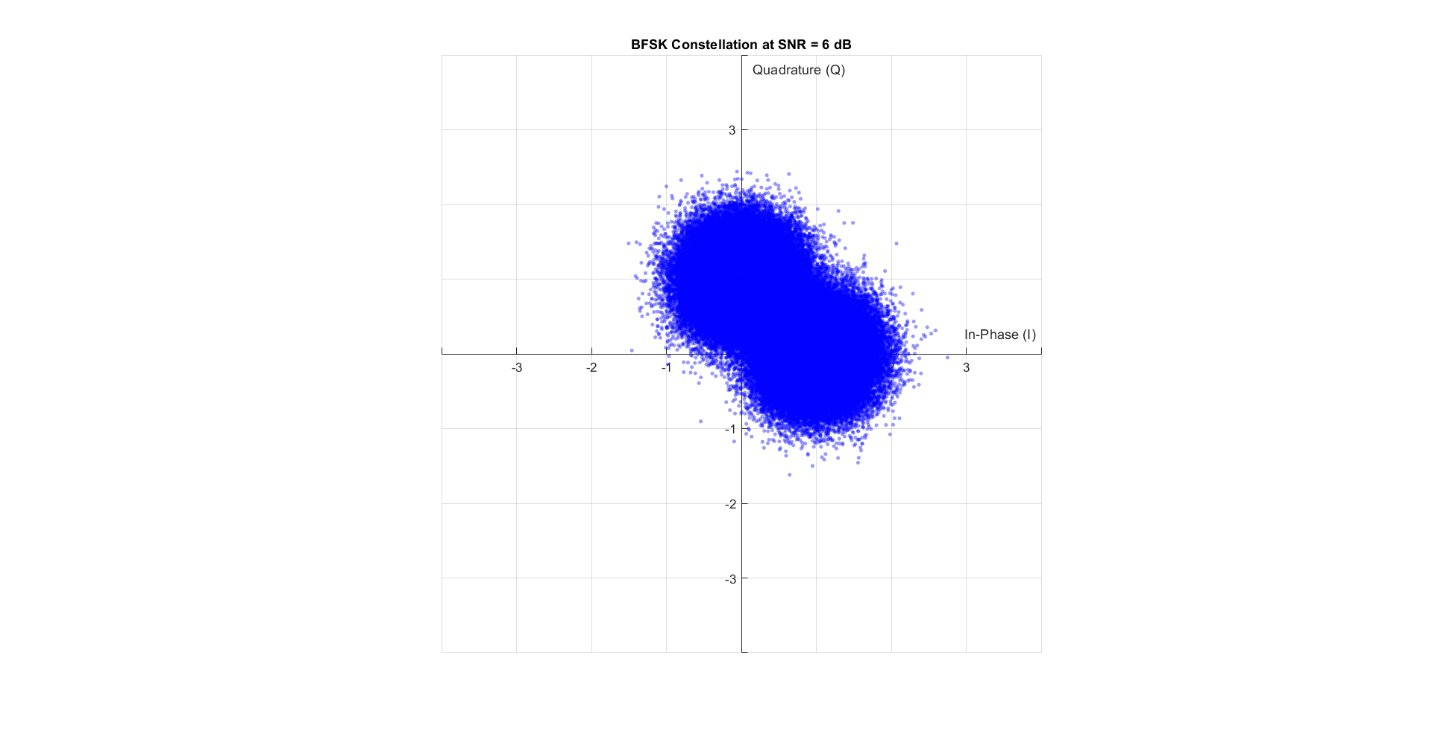


Figure Noise on BFSK with SNR = 6 dB

Figure Noise on BFSK with SNR = -4 dB

BER:

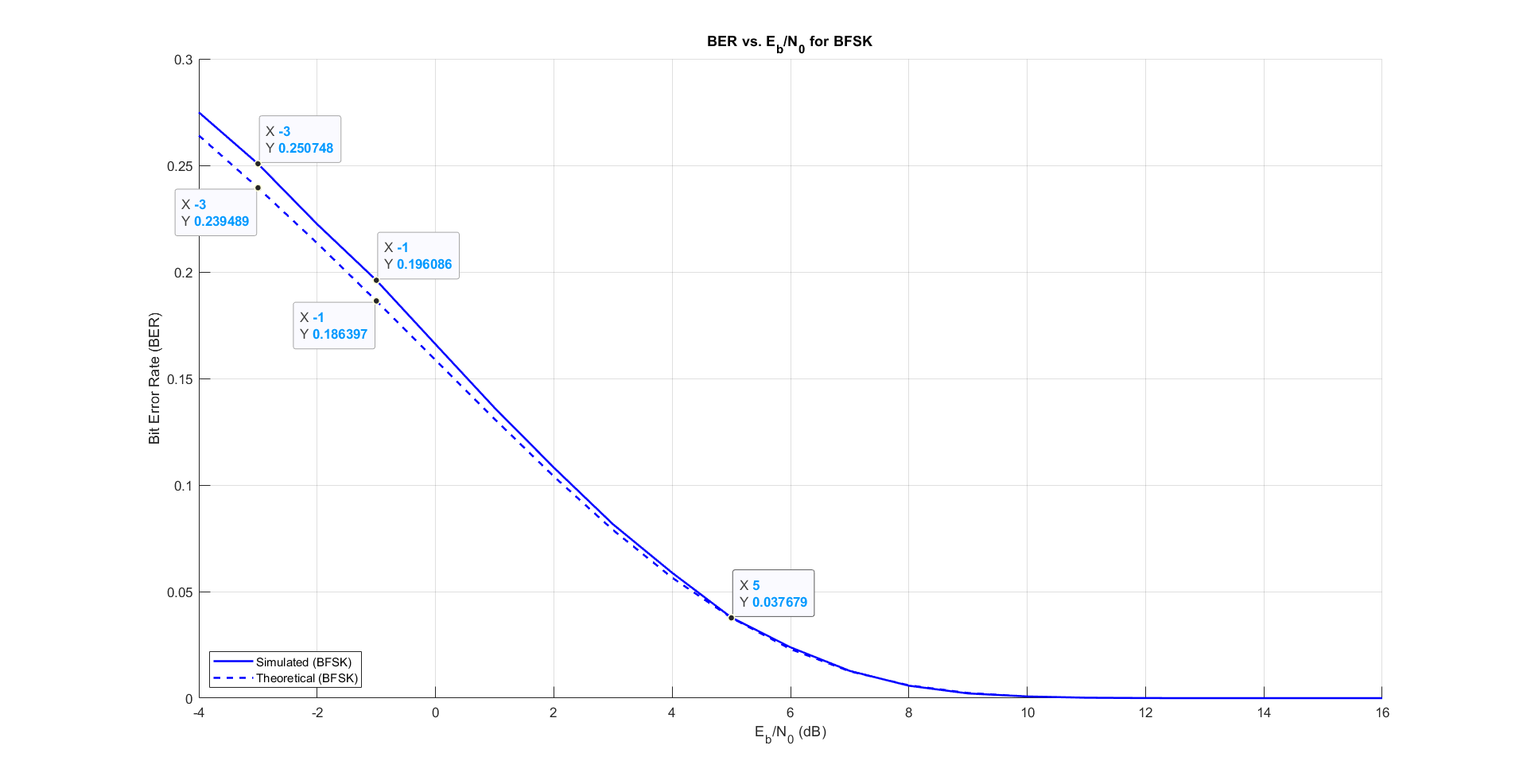


Figure Simulated vs Theoretical BER for BFSK

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:

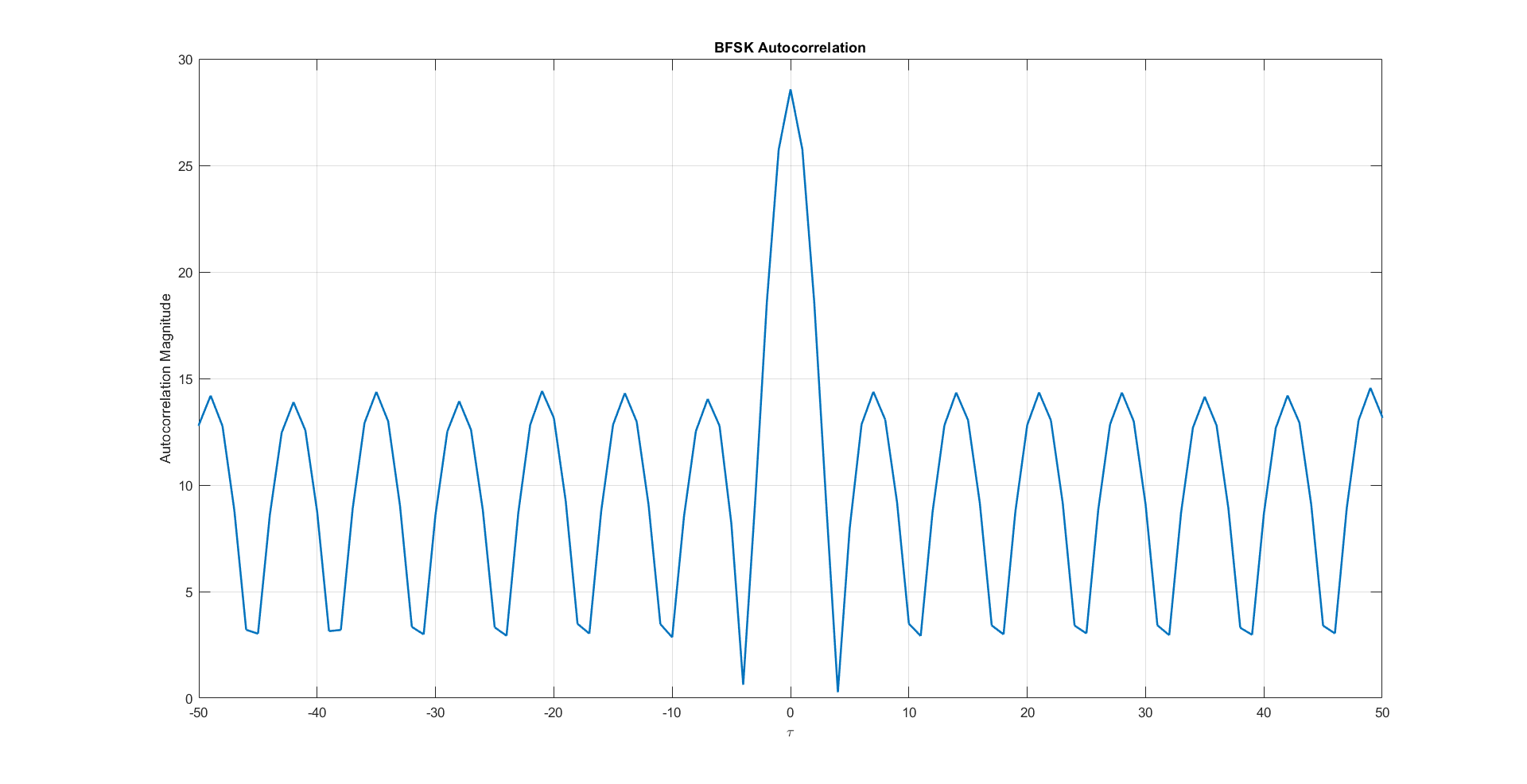


Figure BFSK Auto Correlation

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:

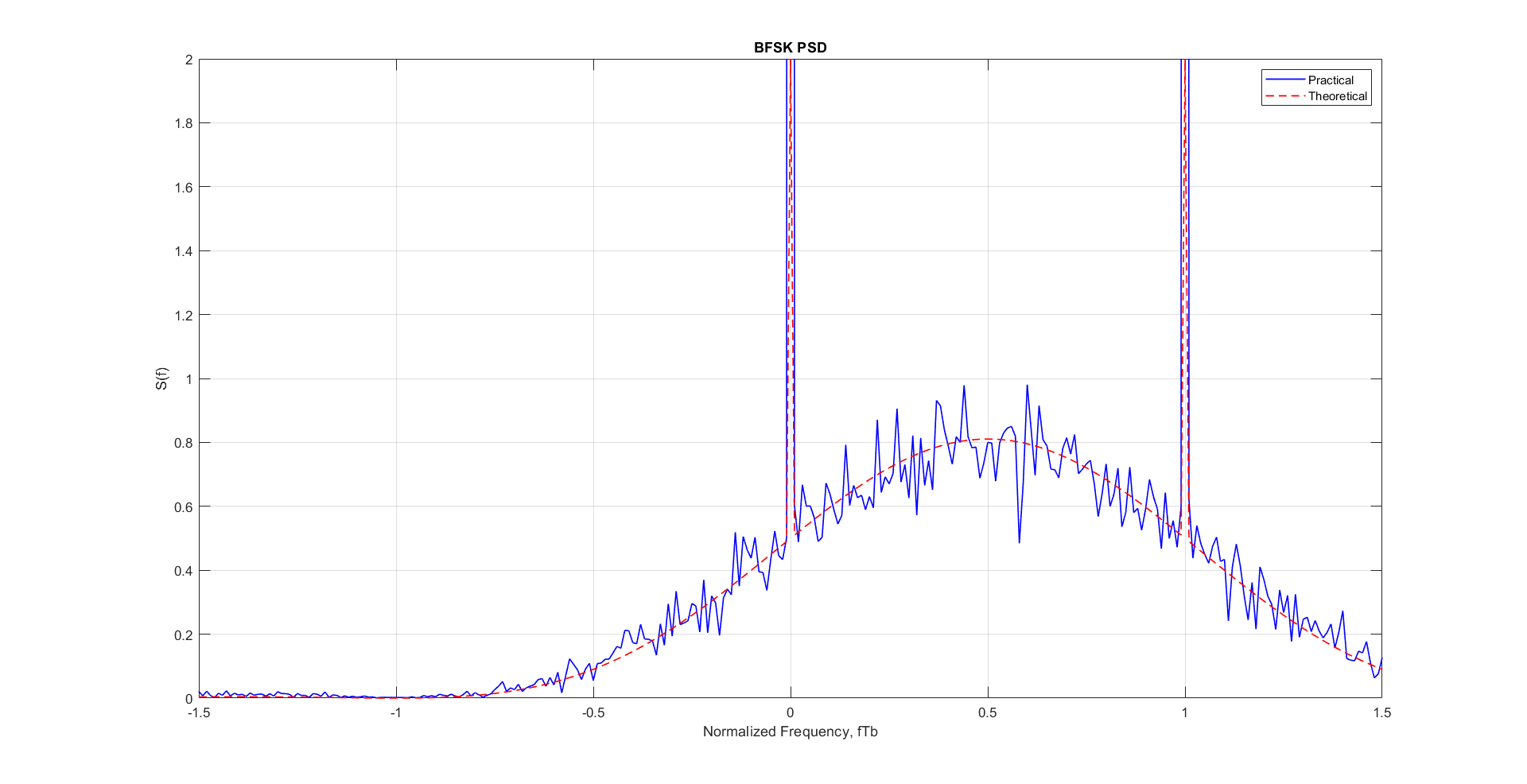


Figure BFSK PSD

Appendix:

Full Code:

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

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

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

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

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

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

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

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

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