UCLA
EE230B
Digital Communication Design Project
Step 1 Report

Alican Salor 404271991
alicansalor@ucla.edu

Darren Reis 804359840
darren.r.reis@gmail.com

February 5, 2014

Contents

1 System Setup

The system is as shown in Figure below:

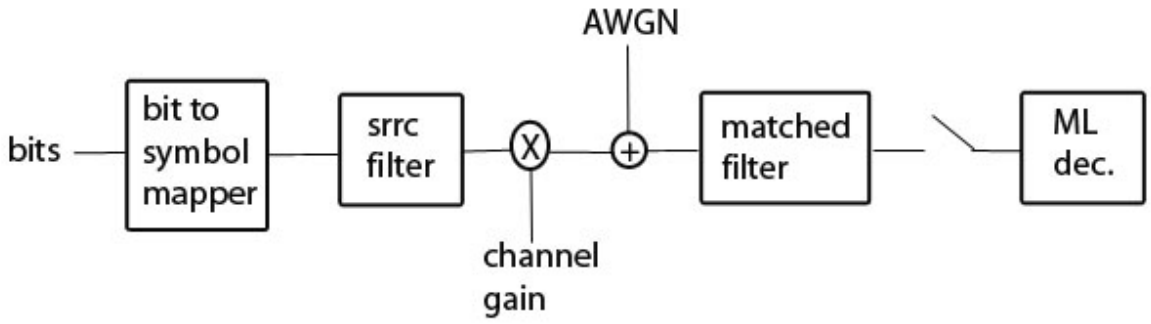

Figure 1: Diagram of the setup used in step 1 of the project

To simulate the system, code was written for each block in the diagram. A random bit generator was used to output a stream of data (Appendix A). In order to make sure enough trials are made for the simulation, the number of the bits generated is set to 48000. The data stream was then passed into a bit-to-symbol mapper, transformed differently based on the modulation scheme (Appendix B). Various schemes were used, including Binary Phase Shift Keying [BPSK], Quadrature Phase Shift Keying [QPSK], and Quadrature Amplitude Modulation [QAM], as shown in Appendices B.1 − B.4. To approximate a real system:

- The system got passed through a square root raised cosine pulse (Appendix C) which has the following impulse response:

$$h(t) = \begin{cases} 1 - \beta + 4\dfrac{\beta}{\pi} & t = 0 \\ \dfrac{\beta}{\sqrt{2}}\left[\left(1+\dfrac{2}{\pi}\right)\sin\left(\dfrac{\pi}{4\beta}\right) + \left(1-\dfrac{2}{\pi}\right)\cos\left(\dfrac{\pi}{4\beta}\right)\right] & t = \pm\frac{T_s}{4\beta} \\ \dfrac{\sin\left[\pi\frac{t}{Ts}\left(1-\beta\right)\right] + 4\beta\frac{t}{Ts}\cos\left[\pi\frac{t}{Ts}\left(1+\beta\right)\right]}{\pi\frac{t}{Ts}\left[1-\left(4\beta\frac{t}{Ts}\right)^2\right]} & otherwise \end{cases}$$

- This filter was then oversampled by four (Appendix D).
- Furthermore, it was broadcast over an Additive White Gaussian Noise (AWGN) channel. (Appendix F). This channel modeled the transmission of the signal from transmitter to receiver. The variance of this channel is determined by the SNR values picked at the beginning of each simulation. In order to achieve this the following equation is used:

  $\sigma^2 = S/(10^{SNR/10})$

  where S is the average signal power of the chosen modulation scheme.
- At the receiver, a matched filter is used for optimal detection of the transmitted signal. As square root raised cosine pulse is used at the transmission side (which a real symmetric pulse), the same square root raised cosine pulse shape is used as the matched filter.

- After sampling the symbol stream back at the original symbol rate, the message is ready for recovery (Appendix G).
- Finally, the data waveform is sent through a decision block, demodulating according to the appropriate scheme (Appendices H.1 − H.4).

An analysis of performance of the system was determined by comparing a theoretical error rate to experimental bit error rate for various schemes. The theoretical derivation is shown in Section 2. On the other hand, the experimental error rate was found by counting the instances of wrongly decoded symbols (or bits as in BPSK case) and dividing this number by the number of total symbols sent through the channel.

2 Mathematical Derivations of Probability of Error

As equiprobable bits are generated and passed through an AWGN channel in this project, the ML decision rule was used. It states:

$\hat{m} = \text{argmin}_{1 \leq m \leq M} ||\underline{r} - \underline{s}_m||$

Thus, in the following subsections, the probability of error for each constellation used in the project is derived using the minimum distance rule.

Furthermore the formulas are used to convert the minimum distance to the average bit energy and the average symbol energy to average bit energy in the M-QAM constellations:

$d_{min} = \sqrt{\frac{6log_2 M}{M-1} E_{bavg}}$

$E_{bavg} = \frac{E_{savg}}{log_2 M}$

2.1 BPSK

The exact probability of bit error is as follows (no interesting decision regions):

$P_{be} = Q\left(\frac{d_{min}/2}{\sqrt{N_o/2}}\right)$

$P_{be} = Q\left(\sqrt{2\frac{E_b}{N_o}}\right)$

2.2 QPSK

The exact probability of symbol error is as follows (found by subtracting the interesting decision regions from the nearest neighbor approx.):

$P_{se} = 2Q\left(\frac{d_{min}/2}{\sqrt{N_o/2}}\right) - Q\left(\frac{d_{min}/2}{\sqrt{N_o/2}}\right)^2$

$P_{se} = 2Q\left(\sqrt{2\frac{E_{bavg}}{N_o}}\right) - Q\left(\sqrt{2\frac{E_{bavg}}{N_o}}\right)^2$

A good approximation of the theoretical bit error can be by dividing the symbol error rate by the number of bits used for each symbol. Then:

$$P_{be} = \frac{1}{2}\left[2Q\left(\sqrt{2\frac{E_{bavg}}{N_o}}\right) - Q\left(\sqrt{2\frac{E_{bavg}}{N_o}}\right)^2\right]$$

2.3 16-QAM

The exact probability of symbol error is as follows (found by subtracting the interesting decision regions from the nearest neighbor approx.):

$$P_{se} = 3Q\left(\frac{d_{min}/2}{\sqrt{N_o/2}}\right) - (9/4)Q\left(\frac{d_{min}/2}{\sqrt{N_o/2}}\right)^2$$

$$P_{se} = 3Q\left(\sqrt{\frac{4}{5}\frac{E_b}{N_o}}\right) - (9/4)Q\left(\sqrt{\frac{4}{5}\frac{E_b}{N_o}}\right)^2$$

A good approximation of the theoretical bit error can be by dividing the symbol error rate by the number of bits used for each symbol. Then:

$$P_{be} = \frac{1}{4}\left[3Q\left(\sqrt{\frac{4}{5}\frac{E_b}{N_o}}\right) - (9/4)Q\left(\sqrt{\frac{4}{5}\frac{E_b}{N_o}}\right)^2\right]$$

2.4 64-QAM

The exact probability of symbol error is as follows (found by subtracting the interesting decision regions from the nearest neighbor approx.):

$$P_{se} = (7/2)Q\left(\frac{d_{min}/2}{\sqrt{N_o/2}}\right) - (49/16)Q\left(\sqrt{\frac{d_{min}/2}{\sqrt{N_o/2}}}\right)^2$$

$$P_{se} = (7/2)Q\left(\sqrt{\frac{18}{63}\frac{E_b}{N_o}}\right) - (49/16)Q\left(\sqrt{\frac{18}{63}\frac{E_b}{N_o}}\right)^2$$

A good approximation of the theoretical bit error can be by dividing the symbol error rate by the number of bits used for each symbol. Then:

$$P_{be} = \frac{1}{6}\left[(7/2)Q\left(\sqrt{\frac{18}{63}\frac{E_b}{N_o}}\right) - (49/16)Q\left(\sqrt{\frac{18}{63}\frac{E_b}{N_o}}\right)^2\right]$$

3 Step 1 Results

In the following sections, the probability of bit error versus the signal to noise ratio is shown on scatter plots for BPSK, QPSK, 16-QAM and 64-QAM constellations .

3.1 Probability of Error vs SNR plots

In the sections given below considering BPSK, QPSK, 16-QAM and 64-QAM constellations the following are plotted using the MATLAB functions given in the appendix:

- Theoretical Bit Error Rate/Symbol Error Rate curve as a function of the symbol SNR

- The theoretical Bit Error Rate/Symbol Error Rate curve as a function of E_b/N_o
- The Bit Error Rate/Symbol Error Rate curve from the simulation as a function of the received signal's SNR

3.1.1 BPSK

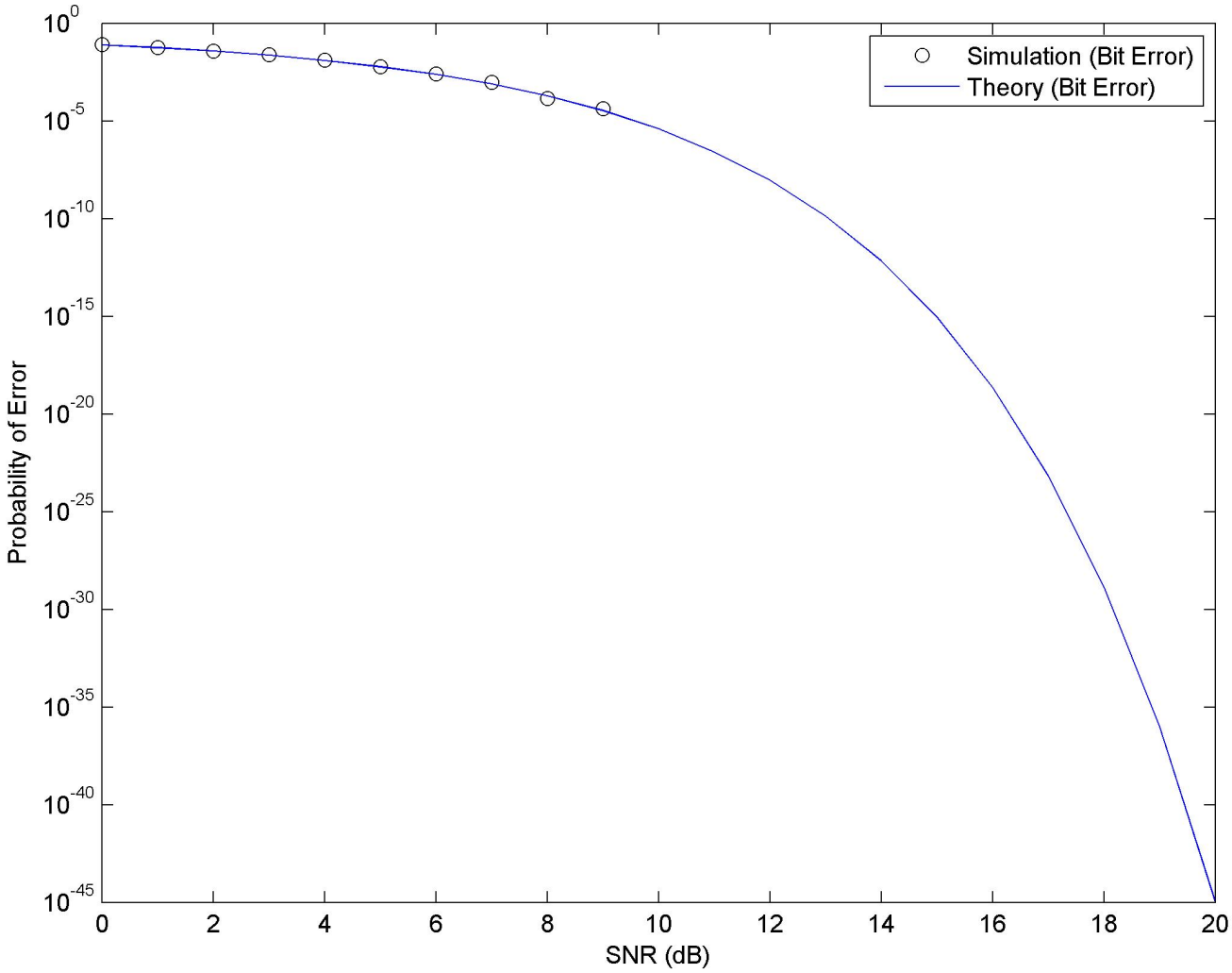

Figure 2: Theoretical and Experimental error rates versus different SNR levels at which the BPSK modulation is run

3.1.2 QPSK

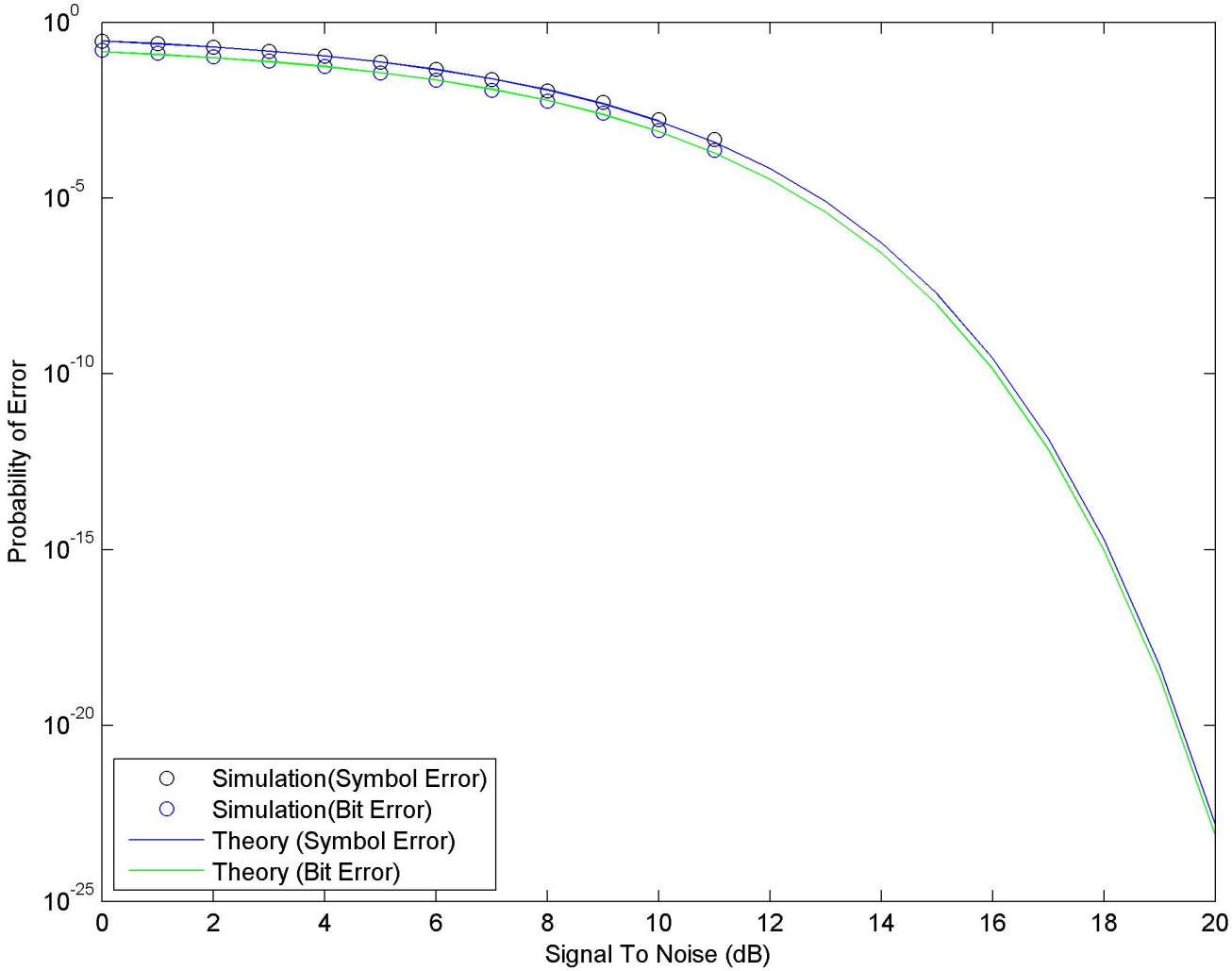

Figure 3: Theoretical and Experimental error rates versus different SNR levels at which the QPSK modulation is run

3.1.3 16-QAM

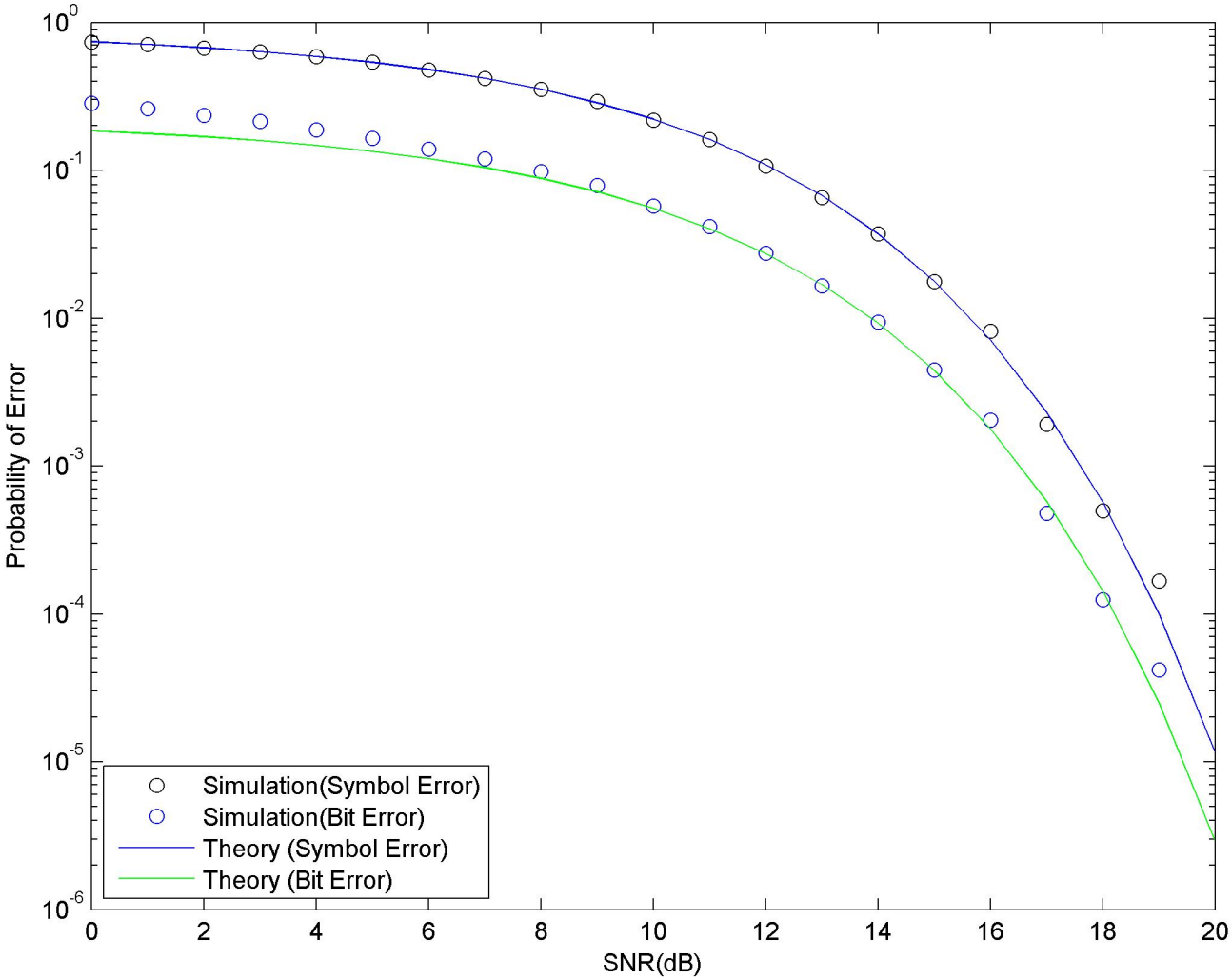

Figure 4: Theoretical and Experimental error rates versus different SNR levels at which the 16-QAM modulation is run

3.1.4 64-QAM

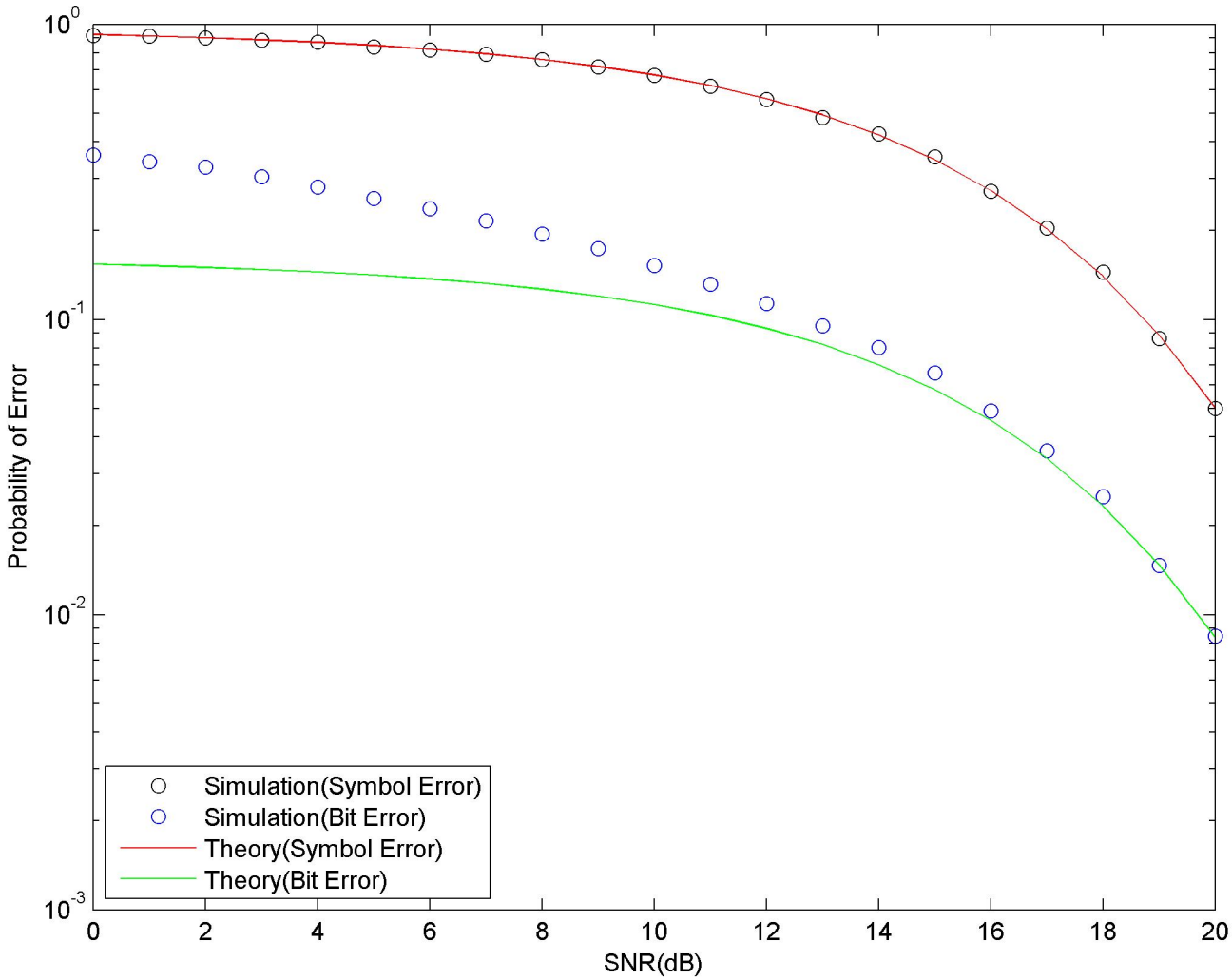

Figure 5: Theoretical and Experimental error rates versus different SNR levels at which the 64-QAM modulation is run

3.2 Constellation Plots

In the following sections scatter plots of BPSK, QPSK, 16-QAM and 64-QAM constellations are plotted at symbol/input SNRs of 3 dB, 6dB, 10dB and 20dB.

3.2.1 BPSK

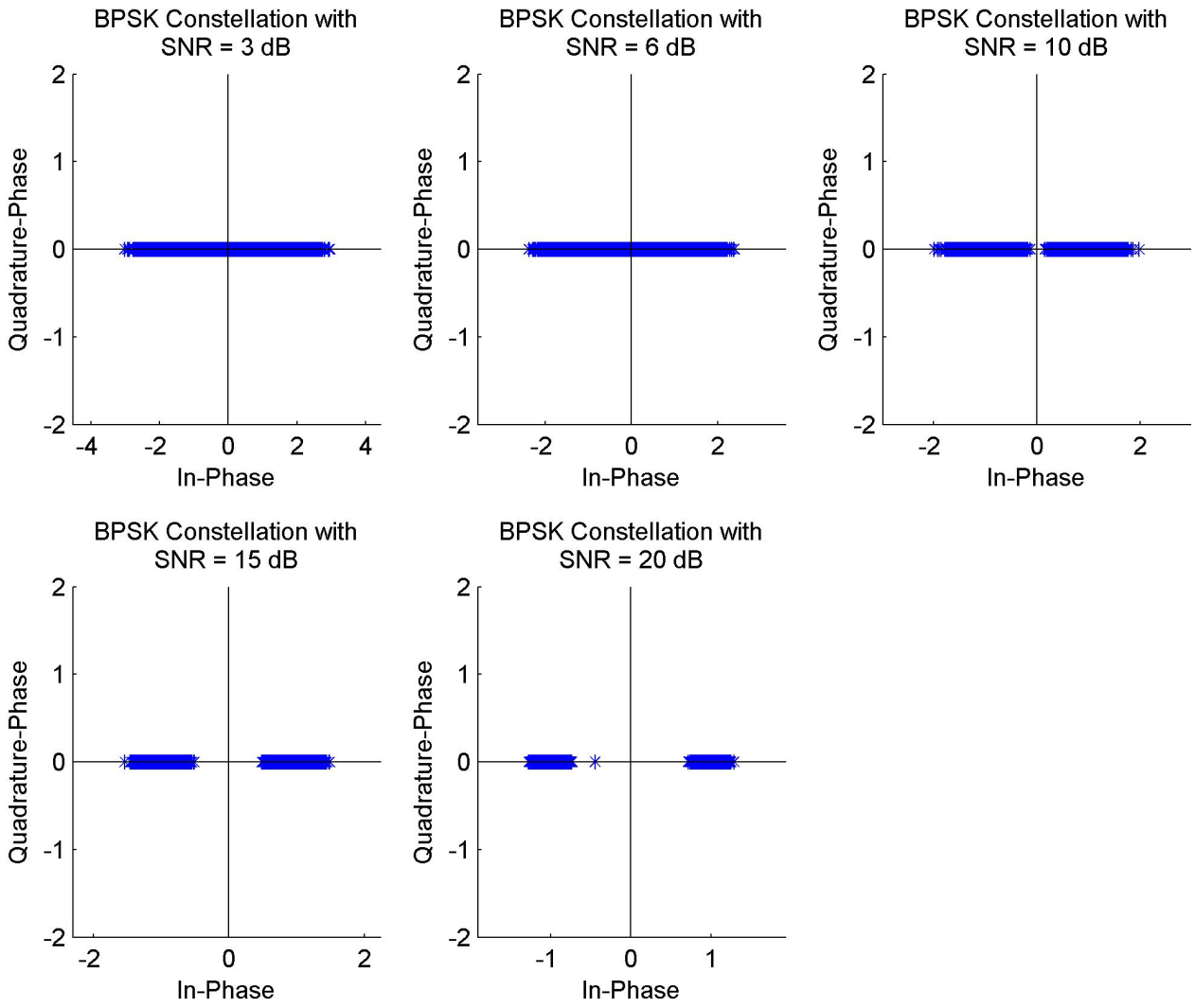

Figure 6: The constellation plots for different levels of SNR at which the BPSK modulation is run

3.2.2 QPSK

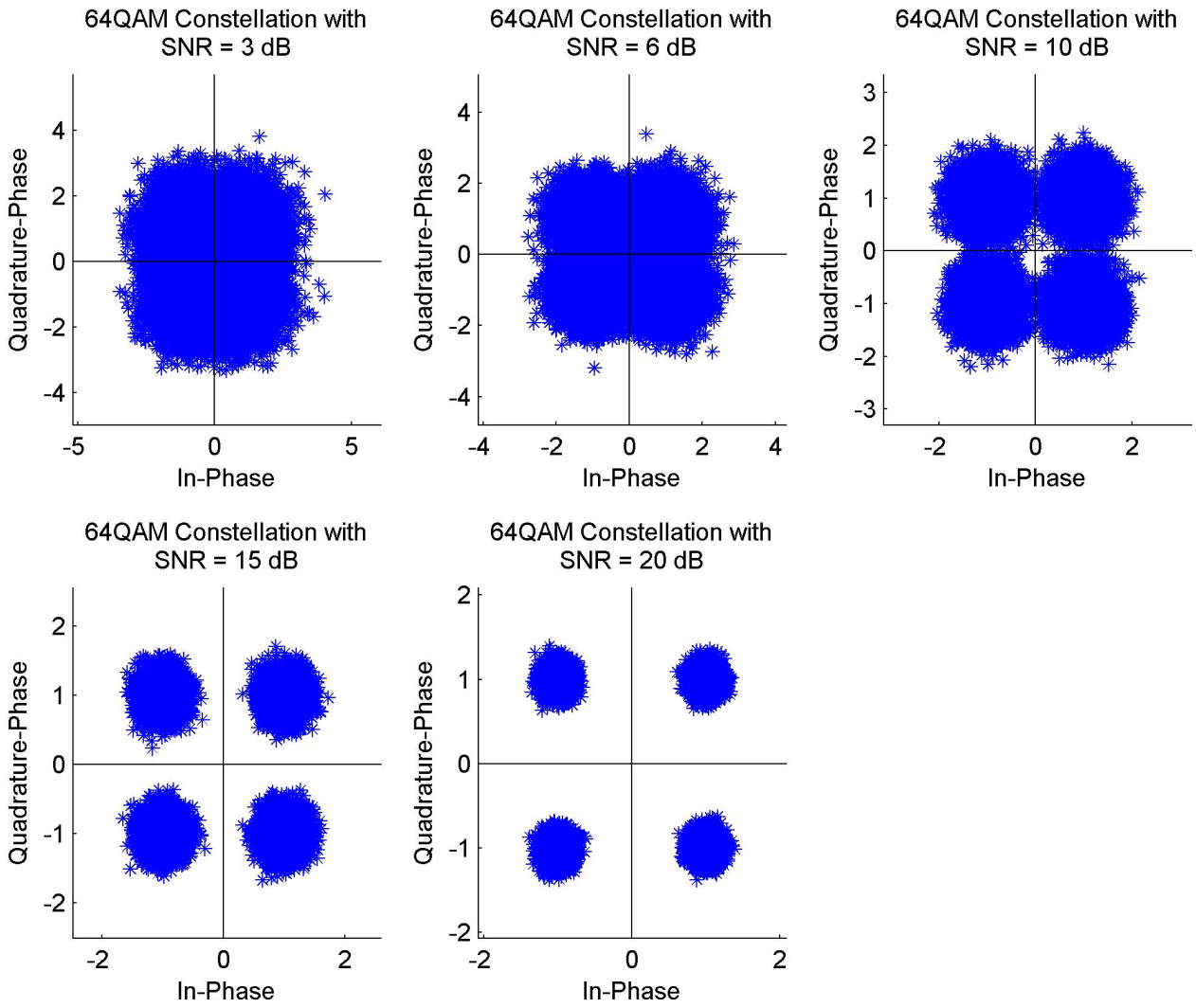

Figure 7: The constellation plots for different levels of SNR at which the QPSK modulation is run

3.2.3 16-QAM

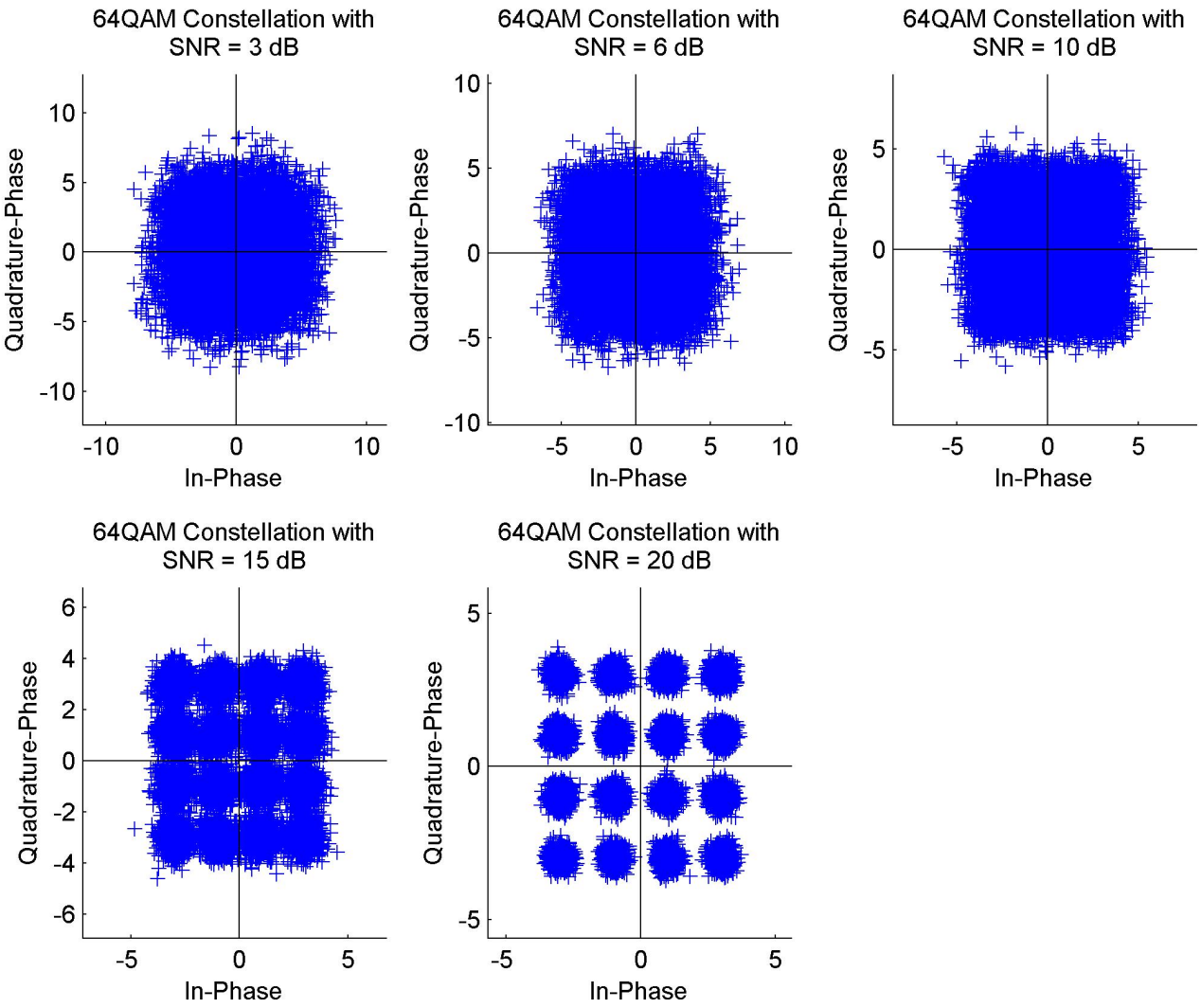

Figure 8: The constellation plots for different levels of SNR at which the 16-QAM modulation is run

3.2.4 64-QAM

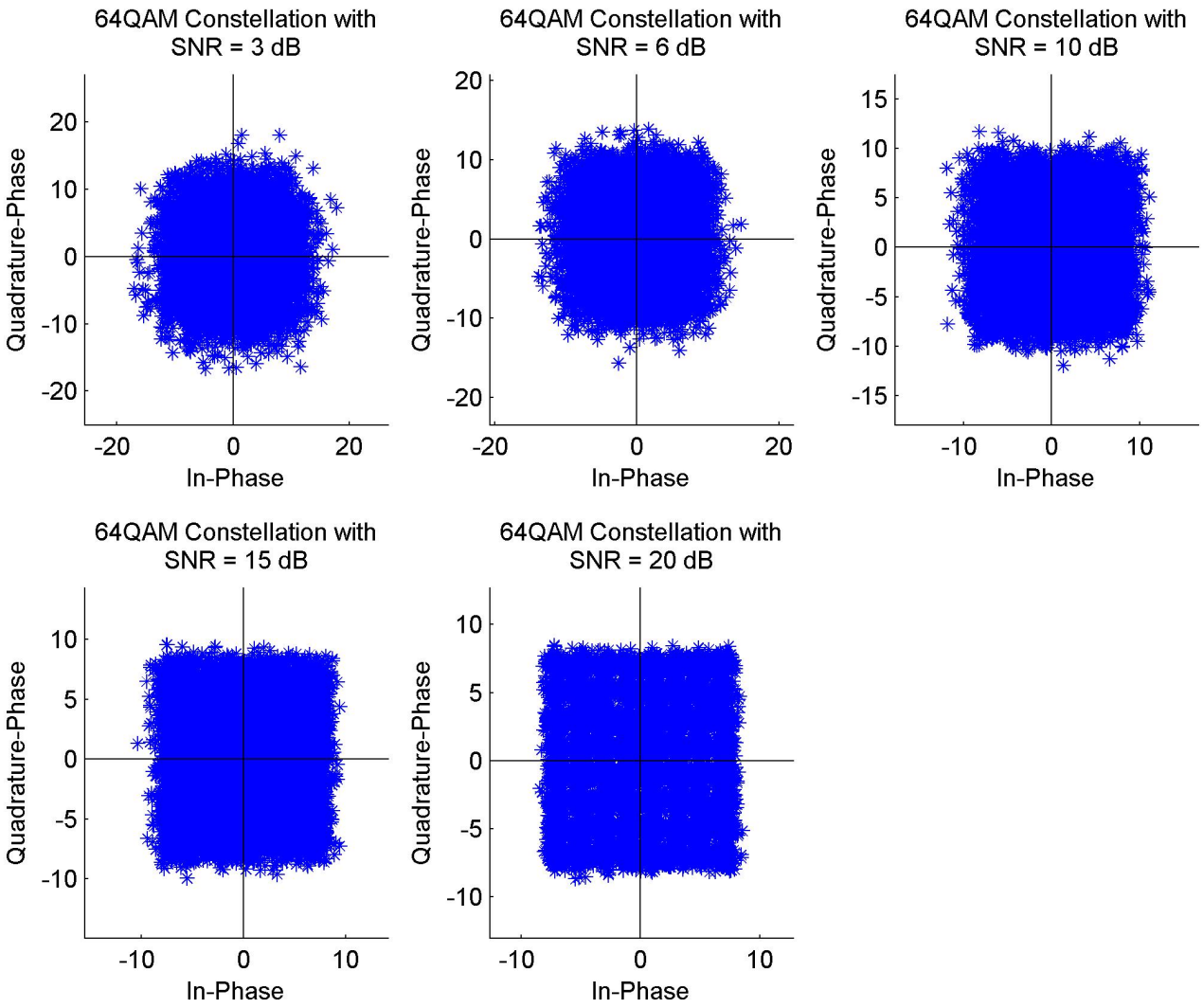

Figure 9: The constellation plots for different levels of SNR at which the 64-QAM modulation is run

4 Conclusion

This project was a demonstration of a digital communication with various modulation schemes which are:

- BPSK
- QPSK
- 16-QAM
- 64-QAM

Randomly generated and equally probable bits are modulated using the given schemes above. They are filtered with a square-root raised cosine pulse and passed through an AWGN channel whose noise is varied from tolerable to overpowering. This can be seen in the both sets of plots. These plots were made by sending 48,000 bits through the system and measuring the error rate, as explained through the report.

The following are deduced from this step of the project:

- For each modulation scheme, *increasing* the input SNR *decreases* the probability of symbol error
- *Increasing* the constellation size *increases* the bit rate. That being said, the plots show the probability of symbol error *increases* as well.
- With sufficient simulation test bits, experimental error rates match up with the theoretical probability of symbol error.
- As the constellation size *increases*, the difference between experimental and theoretical bit error rates at low SNR values *amplifies*. However, at high SNR levels, theoretical and experimental bit error rates are almost identical. This is due to the theoretical bit error rates being crudely approximated (dividing the theoretical symbol error rates by the number of bits used per symbol). By doing so, we assume that bit errors are only caused by a one-bit difference. This underestimates the error rate - simulation symbol errors can occur with more than one bit error. Therefore we see that this approximation works best at high SNR values.
- Finally, it must also be mentioned that the theoretical error rates are calculated using Eb/No rather than the symbol SNR. This conversion is made by dividing the SNR by the number of bits used per symbol. Thus, if the theoretical error rates where plotted with Eb/No values as the x-axis the graphs would shift left.

A Random Bit Sequence Generator

```
function [bits] = random_bit_generator(N)
%FUNCTION - randomly generates given number of uniformly distributed bits

% INPUTS
% N - number of bits to make
% OUTPUTS
% bits - the random set of bits generated

bits_num = round(rand(1,N));
bits = '';

for i=1:N
  bits = strcat(bits,num2str(bits_num(i)));
end
end
```

B Bit to Symbol Mappers

B.1 BPSK Modulation

```
function [ sym ] = bpsk_mod( bits, N )
% FUNCTION
%   convert the bits into symbols on bpsk scheme

% INPUTS
%   bits - the data input bitstream
%   N - the number of symbols to make

% OUTPUTS
%   sym - the symbol stream after modulation

sym = zeros(1,N);
for i=1:N
    currentWord = bits(i);
    switch currentWord
        case '0'
            sym(i) = -1;
        case '1'
            sym(i) = 1;
    end
end
end
```

B.2 QPSK Modulation

```
function [ sym_quad, sym_inp ] = qpsk_mod( bits, N )
% FUNCTION
%   convert the bits into symbols on qpsk scheme
```

```
% INPUTS
%   bits - the data input bitstream
%   N - the number of symbols to make

% OUTPUTS
%   sym_quad - the quadrature component of the symbol (sin)
%   sym_inp - the in-phase component of the symbol (cos)

sym_quad = zeros(1,N);
sym_inp = zeros(1,N);

for i=1:N

    currentWord = bits((i-1)*2+1:i*2);

    switch currentWord

        case '00'
            sym_quad(i) = -1;
            sym_inp(i) = -1;
        case '01'
            sym_quad(i) = 1;
            sym_inp(i) = -1;
        case '10'
            sym_quad(i) = -1;
            sym_inp(i) = 1;
        case '11'
            sym_quad(i) = 1;
            sym_inp(i) = 1;
    end
end
end
```

B.3 16-QAM Modulation

```
function [sym_quad sym_inp] = QAM_16_mod(bits,N)
% FUNCTION
%   convert the bits into symbols on QAM16 scheme

% INPUTS
%   bits - the data input bitstream
%   N - the number of symbols to make

% OUTPUTS
%   sym_quad - the quadrature component of the symbol
%   sym_inp - the in-phase component of the symbol


sym_quad = zeros(1,N);
sym_inp = zeros(1,N);

for i=1:N

    currentWord = bits((i-1)*4+1:i*4);
```

```
    switch currentWord

        case '0000'
            sym_quad(i) = 3;
            sym_inp(i) = -3;
        case '0001'
            sym_quad(i) = 1;
            sym_inp(i) = -3;
        case '0010'
            sym_quad(i) = -3;
            sym_inp(i) = -3;
        case '0011'
            sym_quad(i) = -1;
            sym_inp(i) = -3 ;
        case '0100'
            sym_quad(i) = 3;
            sym_inp(i) = -1;
        case '0101'
            sym_quad(i) = 1;
            sym_inp(i) = -1;
        case '0110'
            sym_quad(i) = -3;
            sym_inp(i) = -1;
        case '0111'
            sym_quad(i) = -1;
            sym_inp(i) = -1;
        case '1000'
            sym_quad(i) = 3;
            sym_inp(i) = 3;
        case '1001'
            sym_quad(i) = 1;
            sym_inp(i) = 3;
        case '1010'
            sym_quad(i) = -3;
            sym_inp(i) = 3;
        case '1011'
            sym_quad(i) = -1;
            sym_inp(i) = 3;
        case '1100'
            sym_quad(i) = 3;
            sym_inp(i) = 1;
        case '1101'
            sym_quad(i) = 1;
            sym_inp(i) = 1;
        case '1110'
            sym_quad(i) = -3;
            sym_inp(i) = 1;
        case '1111'
            sym_quad(i) = -1;
            sym_inp(i) = 1;
    end
end
end
```

B.4 64-QAM Modulation

```
function [sym_quad sym_inp] = QAM_64_mod(bits,N)
% FUNCTION - take the binary stream and modulate into QAM64 scheme

% INPUTS
%   bits - the data input bitstream
%   N - the number of symbols to make

% OUTPUTS
%   sym_quad - the quadrature component of the symbol
%   sym_inp - the in-phase component of the symbol

sym_quad = zeros(1,N);
sym_inp = zeros(1,N);

for i=1:N

    currentWord = bits((i-1)*6+1:i*6);
    switch currentWord

        case '000000'
            sym_quad(i) = 7;
            sym_inp(i) = -7;
        case '000001'
            sym_quad(i) = 7;
            sym_inp(i) = -5;
        case '000010'
            sym_quad(i) = 7;
            sym_inp(i) = -1;
        case '000011'
            sym_quad(i) = 7;
            sym_inp(i) = -3;
        case '000100'
            sym_quad(i) = 7;
            sym_inp(i) = 7;
        case '000101'
            sym_quad(i) = 7;
            sym_inp(i) = 5;
        case '000110'
            sym_quad(i) = 7;
            sym_inp(i) = 1;
        case '000111'
            sym_quad(i) = 7;
            sym_inp(i) = 3;
        case '001000'
            sym_quad(i) = 5;
            sym_inp(i) = -7;
        case '001001'
            sym_quad(i) = 5;
            sym_inp(i) = -5;
        case '001010'
            sym_quad(i) = 5;
            sym_inp(i) = -1;
        case '001011'
            sym_quad(i) = 5;
```

```
    sym_inp(i) = -3;
case '001100'
    sym_quad(i) = 5;
    sym_inp(i) = 7;
case '001101'
    sym_quad(i) = 5;
    sym_inp(i) = 5;
case '001110'
    sym_quad(i) = 5;
    sym_inp(i) = 1;
case '001111'
    sym_quad(i) = 5;
    sym_inp(i) = 3;
case '010000'
    sym_quad(i) = 1;
    sym_inp(i) = -7;
case '010001'
    sym_quad(i) = 1;
    sym_inp(i) = -5;
case '010010'
    sym_quad(i) = 1;
    sym_inp(i) = -1;
case '010011'
    sym_quad(i) = 1;
    sym_inp(i) = -3;
case '010100'
    sym_quad(i) = 1;
    sym_inp(i) = 7;
case '010101'
    sym_quad(i) = 1;
    sym_inp(i) = 5;
case '010110'
    sym_quad(i) = 1;
    sym_inp(i) = 1;
case '010111'
    sym_quad(i) = 1;
    sym_inp(i) = 3;
case '011000'
    sym_quad(i) = 3;
    sym_inp(i) = -7;
case '011001'
    sym_quad(i) = 3;
    sym_inp(i) = -5;
case '011010'
    sym_quad(i) = 3;
    sym_inp(i) = -1;
case '011011'
    sym_quad(i) = 3;
    sym_inp(i) = -3;
case '011100'
    sym_quad(i) = 3;
    sym_inp(i) = 7;
case '011101'
    sym_quad(i) = 3;
    sym_inp(i) = 5;
case '011110'
```

```
    sym_quad(i) = 3;
    sym_inp(i) = 1;
case '011111'
    sym_quad(i) = 3;
    sym_inp(i) = 3;
case '100000'
    sym_quad(i) = -7;
    sym_inp(i) = -7;
case '100001'
    sym_quad(i) = -7;
    sym_inp(i) = -5;
case '100010'
    sym_quad(i) = -7;
    sym_inp(i) = -1;
case '100011'
    sym_quad(i) = -7;
    sym_inp(i) = -3;
case '100100'
    sym_quad(i) = -7;
    sym_inp(i) = 7;
case '100101'
    sym_quad(i) = -7;
    sym_inp(i) = 5;
case '100110'
    sym_quad(i) = -7;
    sym_inp(i) = 1;
case '100111'
    sym_quad(i) = -7;
    sym_inp(i) = 3;
case '101000'
    sym_quad(i) = -5;
    sym_inp(i) = -7;
case '101001'
    sym_quad(i) = -5;
    sym_inp(i) = -5;
case '101010'
    sym_quad(i) = -5;
    sym_inp(i) = -1;
case '101011'
    sym_quad(i) = -5;
    sym_inp(i) = -3;
case '101100'
    sym_quad(i) = -5;
    sym_inp(i) = 7;
case '101101'
    sym_quad(i) = -5;
    sym_inp(i) = 5;
case '101110'
    sym_quad(i) = -5;
    sym_inp(i) = 1;
case '101111'
    sym_quad(i) = -5;
    sym_inp(i) = 3;
case '110000'
    sym_quad(i) = -1;
    sym_inp(i) = -7;
```

```
        case '110001'
            sym_quad(i) = -1;
            sym_inp(i) = -5;
        case '110010'
            sym_quad(i) = -1;
            sym_inp(i) = -1;
        case '110011'
            sym_quad(i) = -1;
            sym_inp(i) = -3;
        case '110100'
            sym_quad(i) = -1;
            sym_inp(i) = 7;
        case '110101'
            sym_quad(i) = -1;
            sym_inp(i) = 5;
        case '110110'
            sym_quad(i) = -1;
            sym_inp(i) = 1;
        case '110111'
            sym_quad(i) = -1;
            sym_inp(i) = 3;
        case '111000'
            sym_quad(i) = -3;
            sym_inp(i) = -7;
        case '111001'
            sym_quad(i) = -3;
            sym_inp(i) = -5;
        case '111010'
            sym_quad(i) = -3;
            sym_inp(i) = -1;
        case '111011'
            sym_quad(i) = -3;
            sym_inp(i) = -3;
        case '111100'
            sym_quad(i) = -3;
            sym_inp(i) = 7;
        case '111101'
            sym_quad(i) = -3;
            sym_inp(i) = 5;
        case '111110'
            sym_quad(i) = -3;
            sym_inp(i) = 1;
        case '111111'
            sym_quad(i) = -3;
            sym_inp(i) = 3;
    end

end
end
```

C Square Root Raised Cosine Filter

```
function [srrc_normalized_response] = sqrt_raised_cosine(overSampleFactor,
    rollOffFactor,limits,Ts)
% FUNCTION - this pulse is used for pulse shaping, to eliminate ISI.

% INPUTS
% overSamplingFactor - the amount to over sample
% rollOffFactor - the BW metric, roughly measure of the residual ripples
% limits - the upper limits (and negative for lower)
% Ts - the symbol interval

% OUTPUTS
% srrc_normalized_response - the data signal after the srrc

t = -limits:1/overSampleFactor:limits;
response = zeros(1,length(t));

for i=1:length(t)

    if (t(i) == 0)
        response(i) = 1 - rollOffFactor + 4*rollOffFactor/pi;
    elseif (t(i)== Ts/(4*rollOffFactor) || t(i)== -Ts/(4*rollOffFactor))
        response(i) = (rollOffFactor/sqrt(2))*((1+2/pi)*sin(pi/(4*rollOffFactor)) + (1-
            2/pi)*cos(pi/(4*rollOffFactor)));
    else
       response(i) = (sin((pi*t(i)/Ts)*(1-rollOffFactor)) +
           4*rollOffFactor*(t(i)/Ts)*cos(pi*(t(i)/Ts)*(1+rollOffFactor)))/(pi*(t(i)/Ts)*(1-(4*rollOffFactor*t(i)
    end

    %normalize to unit energy
    srrc_normalized_response = response./sqrt(sum(response.^2));
end
end
```

D Up Sampler

```
function [impulseTrain] = impulse_train(overSampleSize,N,symbols)
% FUNCTION
%   Take in the symbol signal and zero-pad to create a new data waveform

% INPUT
% overSampleSize - the number or zeros to pad
% N - the number of symbols transmitted
% symbols - the symbol waveform

% OUTPUT
% impulseTrain - the upsampled waveform


impulseTrain = zeros(1,length(symbols)*overSampleSize);

for i=1:N
```

```
   impulseTrain((i-1)*overSampleSize+1) = symbols(i);
end
end
```

E Additive Gaussian White Noise Channel

```
function [channel_output] = awgn_channel(transmitted_sig,snr,S)
% FUNCTION
% this models WGN noise sequence added to the signal waveform

% INPUT
% transmitted_sig - the signal sent over the channel
% snr - the signal to noise ratio comparing noise and white noise variance
% S - the average symbol power

% OUTPUT
% channel_output - the sum of input signal and the white noise

variance = S/(10^(snr/10));
noise = sqrt(variance/2)*randn(size(transmitted_sig));

channel_output = transmitted_sig + noise;
end
```

F Additive Gaussian White Noise Channel

```
function [channel_output] = awgn_channel(transmitted_sig,snr,S)
% FUNCTION
% this models WGN noise sequence added to the signal waveform

% INPUT
% transmitted_sig - the signal sent over the channel
% snr - the signal to noise ratio comparing noise and white noise variance
% S - the average symbol power

% OUTPUT
% channel_output - the sum of input signal and the white noise

variance = S/(10^(snr/10));
noise = sqrt(variance/2)*randn(size(transmitted_sig));

channel_output = transmitted_sig + noise;
end
```

G Sampler

```
function [output] = sampler(input_signal, overSamplingFactor, Ts)
% FUNCTION - this takes the cts waveform and samples it
%             (with upconversion) at periods of Ts

% INPUTS
% input_signal - the data stream received
% overSamplingFactor - the amount to over sample
% Ts - the symbol interval

% OUTPUTS
% output - the sampled data signal

N = length(input_signal);
output = zeros(1,N/overSamplingFactor);
for i=1:N/overSamplingFactor
    output(i) = input_signal((i-1)*overSamplingFactor*Ts+1);
end
end
```

H Decision Blocks

H.1 BPSK Demodulation

```
function [bits] = bpsk_demod(sig)
% FUNCTION - this takes in the real and imaginary signals and maps
%             back into binary chips

% INPUTS
% inphase_sig - the real component of the incoming waveform

% OUTPUTS
% bits - the binary stream of data received

bits = '';
loopSize = length(sig);
for i=1:loopSize
    %decision based on regions of constellation
    if(sig(i) >= 0)
        bits = strcat(bits,'1');
    elseif(sig(i) < 0)
        bits = strcat(bits,'0');
    end
end
end
```

H.2 QPSK Demodulation

```
function [bits] = qpsk_demod(inphase_sig,quadrature_sig)
% FUNCTION - this takes in the real and imaginary signals and maps
%             back into binary chips

% INPUTS
% inphase_sig - the real component of the incoming waveform
% quadrature_sig - the imaginary component of the waveform

% OUTPUTS
% bits - the binary stream of data received

bits = '';
loopSize = length(inphase_sig);


for i=1:loopSize

    %decision based on regions of constellation

    if(inphase_sig(i) >= 0)
        if (quadrature_sig(i) >= 0)
            bits = strcat(bits,'11');
        elseif (quadrature_sig(i) < 0)
            bits = strcat(bits,'10');
        end
    elseif(inphase_sig(i) < 0)
        if (quadrature_sig(i) >= 0)
            bits = strcat(bits,'01');
        elseif (quadrature_sig(i) < 0)
            bits = strcat(bits,'00');
        end
    end
end
```

H.3 16-QAM Demodulation

```
function [bits] = QAM_16_demod(inphase_sig,quadrature_sig)
% FUNCTION - this takes in the real and imaginary signals and maps
%             back into binary chips

% INPUTS
% inphase_sig - the real component of the incoming waveform
% quadrature_sig - the imaginary component of the waveform

% OUTPUTS
% bits - the binary stream of data received

bits = '';
loopSize = length(inphase_sig);

for i=1:loopSize
```

```
    %decision based on regions of constellation
    if(quadrature_sig(i) >= 2)

        if (inphase_sig(i) <= -2)
           bits = strcat(bits,'0000');
        elseif (inphase_sig(i) > -2 && inphase_sig(i) < 0)
           bits = strcat(bits,'0100');
        elseif (inphase_sig(i) >= 0 && inphase_sig(i) < 2)
           bits = strcat(bits,'1100');
        elseif (inphase_sig(i) >= 2 )
           bits = strcat(bits,'1000');
        end

    elseif (quadrature_sig(i) >= 0 && quadrature_sig(i) < 2)

        if (inphase_sig(i) <= -2)
           bits = strcat(bits,'0001');
        elseif (inphase_sig(i) > -2 && inphase_sig(i) < 0)
           bits = strcat(bits,'0101');
        elseif (inphase_sig(i) >= 0 && inphase_sig(i) < 2)
           bits = strcat(bits,'1101');
        elseif (inphase_sig(i) >= 2 )
           bits = strcat(bits,'1001');
        end

    elseif (quadrature_sig(i) < 0 && quadrature_sig(i) >= -2 )

         if (inphase_sig(i) <= -2)
           bits = strcat(bits,'0011');
        elseif (inphase_sig(i) > -2 && inphase_sig(i) < 0)
           bits = strcat(bits,'0111');
        elseif (inphase_sig(i) >= 0 && inphase_sig(i) < 2)
           bits = strcat(bits,'1111');
        elseif (inphase_sig(i) >= 2 )
           bits = strcat(bits,'1011');
        end

    elseif(quadrature_sig(i) < -2)

        if (inphase_sig(i) <= -2)
           bits = strcat(bits,'0010');
        elseif (inphase_sig(i) > -2 && inphase_sig(i) < 0)
           bits = strcat(bits,'0110');
        elseif (inphase_sig(i) >= 0 && inphase_sig(i) < 2)
           bits = strcat(bits,'1110');
        elseif (inphase_sig(i) >= 2 )
           bits = strcat(bits,'1010');
         end
    end
end
end
```

H.4 64-QAM Demodulation

```
function [bits] = QAM_64_demod(inphase_sig,quadrature_sig)
% FUNCTION - this takes in the real and imaginary signals and maps
%             back into binary chips

% INPUTS
% inphase_sig - the real component of the incoming waveform
% quadrature_sig - the imaginary component of the waveform

% OUTPUTS
% bits - the binary stream of data received

bits = '';
loopSize = length(inphase_sig);


for i=1:loopSize

    %decision
    if(quadrature_sig(i) >= 6)

        if (inphase_sig(i) <= -6)
           bits = strcat(bits,'000000');
        elseif (inphase_sig(i) > -6 && inphase_sig(i) <= -4)
           bits = strcat(bits,'000001');
        elseif (inphase_sig(i) > -4 && inphase_sig(i) <= -2)
           bits = strcat(bits,'000011');
        elseif (inphase_sig(i) > -2 && inphase_sig(i) <= 0)
           bits = strcat(bits,'000010');
        elseif (inphase_sig(i) > 0 && inphase_sig(i) <= 2)
           bits = strcat(bits,'000110');
        elseif (inphase_sig(i) > 2 && inphase_sig(i) <= 4)
           bits = strcat(bits,'000111');
        elseif (inphase_sig(i) > 4 && inphase_sig(i) <= 6)
           bits = strcat(bits,'000101');
        elseif (inphase_sig(i) > 6 )
           bits = strcat(bits,'000100');
        end

    elseif(quadrature_sig(i) >= 4 && quadrature_sig(i) < 6)

        if (inphase_sig(i) <= -6)
           bits = strcat(bits,'001000');
        elseif (inphase_sig(i) > -6 && inphase_sig(i) <= -4)
           bits = strcat(bits,'001001');
        elseif (inphase_sig(i) > -4 && inphase_sig(i) <= -2)
           bits = strcat(bits,'001011');
        elseif (inphase_sig(i) > -2 && inphase_sig(i) <= 0)
           bits = strcat(bits,'001010');
        elseif (inphase_sig(i) > 0 && inphase_sig(i) <= 2)
           bits = strcat(bits,'001110');
        elseif (inphase_sig(i) > 2 && inphase_sig(i) <= 4)
           bits = strcat(bits,'001111');
        elseif (inphase_sig(i) > 4 && inphase_sig(i) <= 6)
           bits = strcat(bits,'001101');
```

```
    elseif (inphase_sig(i) > 6 )
       bits = strcat(bits,'001100');
    end

elseif (quadrature_sig(i) >= 2 && quadrature_sig(i) < 4)

    if (inphase_sig(i) <= -6)
       bits = strcat(bits,'011000');
    elseif (inphase_sig(i) > -6 && inphase_sig(i) <= -4)
       bits = strcat(bits,'011001');
    elseif (inphase_sig(i) > -4 && inphase_sig(i) <= -2)
       bits = strcat(bits,'011011');
    elseif (inphase_sig(i) > -2 && inphase_sig(i) <= 0)
       bits = strcat(bits,'011010');
    elseif (inphase_sig(i) > 0 && inphase_sig(i) <= 2)
       bits = strcat(bits,'011110');
    elseif (inphase_sig(i) > 2 && inphase_sig(i) <= 4)
       bits = strcat(bits,'011111');
    elseif (inphase_sig(i) > 4 && inphase_sig(i) <= 6)
       bits = strcat(bits,'011101');
    elseif (inphase_sig(i) > 6 )
       bits = strcat(bits,'011100');
    end

elseif (quadrature_sig(i) >= 0 && quadrature_sig(i) < 2 )

    if (inphase_sig(i) <= -6)
       bits = strcat(bits,'010000');
    elseif (inphase_sig(i) > -6 && inphase_sig(i) <= -4)
       bits = strcat(bits,'010001');
    elseif (inphase_sig(i) > -4 && inphase_sig(i) <= -2)
       bits = strcat(bits,'010011');
    elseif (inphase_sig(i) > -2 && inphase_sig(i) <= 0)
       bits = strcat(bits,'010010');
    elseif (inphase_sig(i) > 0 && inphase_sig(i) <= 2)
       bits = strcat(bits,'010110');
    elseif (inphase_sig(i) > 2 && inphase_sig(i) <= 4)
       bits = strcat(bits,'010111');
    elseif (inphase_sig(i) > 4 && inphase_sig(i) <= 6)
       bits = strcat(bits,'010101');
    elseif (inphase_sig(i) > 6 )
       bits = strcat(bits,'010100');
    end

elseif(quadrature_sig(i) < 0 && quadrature_sig(i) >= -2)

    if (inphase_sig(i) <= -6)
       bits = strcat(bits,'110000');
    elseif (inphase_sig(i) > -6 && inphase_sig(i) <= -4)
       bits = strcat(bits,'110001');
    elseif (inphase_sig(i) > -4 && inphase_sig(i) <= -2)
       bits = strcat(bits,'110011');
    elseif (inphase_sig(i) > -2 && inphase_sig(i) <= 0)
       bits = strcat(bits,'110010');
    elseif (inphase_sig(i) > 0 && inphase_sig(i) <= 2)
       bits = strcat(bits,'110110');
```

```
    elseif (inphase_sig(i) > 2 && inphase_sig(i) <= 4)
       bits = strcat(bits,'110111');
    elseif (inphase_sig(i) > 4 && inphase_sig(i) <= 6)
       bits = strcat(bits,'110101');
    elseif (inphase_sig(i) > 6 )
       bits = strcat(bits,'110100');
    end

elseif(quadrature_sig(i) < -2 && quadrature_sig(i) >= -4)

    if (inphase_sig(i) <= -6)
       bits = strcat(bits,'111000');
    elseif (inphase_sig(i) > -6 && inphase_sig(i) <= -4)
       bits = strcat(bits,'111001');
    elseif (inphase_sig(i) > -4 && inphase_sig(i) <= -2)
       bits = strcat(bits,'111011');
    elseif (inphase_sig(i) > -2 && inphase_sig(i) <= 0)
       bits = strcat(bits,'111010');
    elseif (inphase_sig(i) > 0 && inphase_sig(i) <= 2)
       bits = strcat(bits,'111110');
    elseif (inphase_sig(i) > 2 && inphase_sig(i) <= 4)
       bits = strcat(bits,'111111');
    elseif (inphase_sig(i) > 4 && inphase_sig(i) <= 6)
       bits = strcat(bits,'111101');
    elseif (inphase_sig(i) > 6 )
       bits = strcat(bits,'111100');
    end

elseif(quadrature_sig(i) < -4 && quadrature_sig(i) >= -6)

    if (inphase_sig(i) <= -6)
       bits = strcat(bits,'101000');
    elseif (inphase_sig(i) > -6 && inphase_sig(i) <= -4)
       bits = strcat(bits,'101001');
    elseif (inphase_sig(i) > -4 && inphase_sig(i) <= -2)
       bits = strcat(bits,'101011');
    elseif (inphase_sig(i) > -2 && inphase_sig(i) <= 0)
       bits = strcat(bits,'101010');
    elseif (inphase_sig(i) > 0 && inphase_sig(i) <= 2)
       bits = strcat(bits,'101110');
    elseif (inphase_sig(i) > 2 && inphase_sig(i) <= 4)
       bits = strcat(bits,'101111');
    elseif (inphase_sig(i) > 4 && inphase_sig(i) <= 6)
       bits = strcat(bits,'101101');
    elseif (inphase_sig(i) > 6 )
       bits = strcat(bits,'101100');
    end

 elseif(quadrature_sig(i) < -6)

    if (inphase_sig(i) <= -6)
       bits = strcat(bits,'100000');
    elseif (inphase_sig(i) > -6 && inphase_sig(i) <= -4)
       bits = strcat(bits,'100001');
    elseif (inphase_sig(i) > -4 && inphase_sig(i) <= -2)
       bits = strcat(bits,'100011');
```

```
        elseif (inphase_sig(i) > -2 && inphase_sig(i) <= 0)
           bits = strcat(bits,'100010');
        elseif (inphase_sig(i) > 0 && inphase_sig(i) <= 2)
           bits = strcat(bits,'100110');
        elseif (inphase_sig(i) > 2 && inphase_sig(i) <= 4)
           bits = strcat(bits,'100111');
        elseif (inphase_sig(i) > 4 && inphase_sig(i) <= 6)
           bits = strcat(bits,'100101');
        elseif (inphase_sig(i) > 6 )
           bits = strcat(bits,'100100');
        end

    end
end
end
```