UCLA
EE230B
Digital Communication Design Project
Step 1 Report

Alican Salor 404271991
alicansalor@ucla.edu

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

February 1, 2014

Contents

1 System Setup

The system is as shown in Figure BLAH. 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 B). The data stream then was passed into a bit-to-symbol mapper, transformed differently based on the modulation scheme (Appendix C). Various schemes were used, including Binary Phase Shift Keying [BPSK], Quadrature Phase Shift Keying [QPSK], and Quadrature Amplitude Modulation [QAM], as shown in Appendices C.1 − C.4. To approximate a real system, the system got passed through a square root raised cosine pulse (Appendix D). This filter was then oversampled by four in order to DO BLAH (Appendix E). To simulate world noises, it was broadcast over a channel of additive white gaussian noises (Appendix G). For different SNR levels, the AWGN channel had different gain and variance. This channel modeled the transmission of the signal from transmitter to receiver. The receiver then used a matched filter version of the raised cosine pusle-shape. After sampling the symbol stream back at the original symbol rate, the message was ready for recovery (Appendix H). Finally, the data waveform was sent through a decision block, demodulating according to the appropriate scheme (Appendices I.1 − I.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. The experimental error rate was found by counting the instances of wrongly decoded bits.

2 Mathematical Derivations of Probability of Error

As equiprobable bits are generated and passed through an AWGN channel in this project we have used ML decision rule which 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 interseting decision regions):

$$P_e = Q(\frac{d_{min}/2}{\sqrt{N_o/2}})$$

$$P_e = Q(\sqrt{2\frac{E_b}{N_o}})$$

2.2 QPSK

The exact probability of bit error is as follows (found by subtracting the interseting decision regions from the nearest neighbour approx.):

$$P_e = 2Q(\frac{d_{min}/2}{\sqrt{N_o/2}}) - Q(\frac{d_{min}/2}{\sqrt{N_o/2}})^2$$

$$P_e = 2Q(\sqrt{2\frac{E_{bavg}}{N_o}}) - Q(\sqrt{2\frac{E_{bavg}}{N_o}})^2$$

2.3 16-QAM

The exact probability of bit error is as follows (found by subtracting the interseting decision regions from the nearest neighbour approx.):

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

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

2.4 64-QAM

The exact probability of bit error is as follows (found by subtracting the interseting decision regions from the nearest neighbour approx.):

$$P_e = (7/2)Q(\frac{d_{min}/2}{\sqrt{N_o/2}}) - (49/16)Q(\sqrt{\frac{d_{min}/2}{\sqrt{N_o/2}}})^2$$
$$P_e = (7/2)Q(\sqrt{\frac{18}{63}\frac{E_b}{N_o}}) - (49/16)Q(\sqrt{\frac{18}{63}\frac{E_b}{N_o}})^2$$

3 Step 1 Results

3.1 Probability of Error vs SNR plots

3.1.1 BPSK

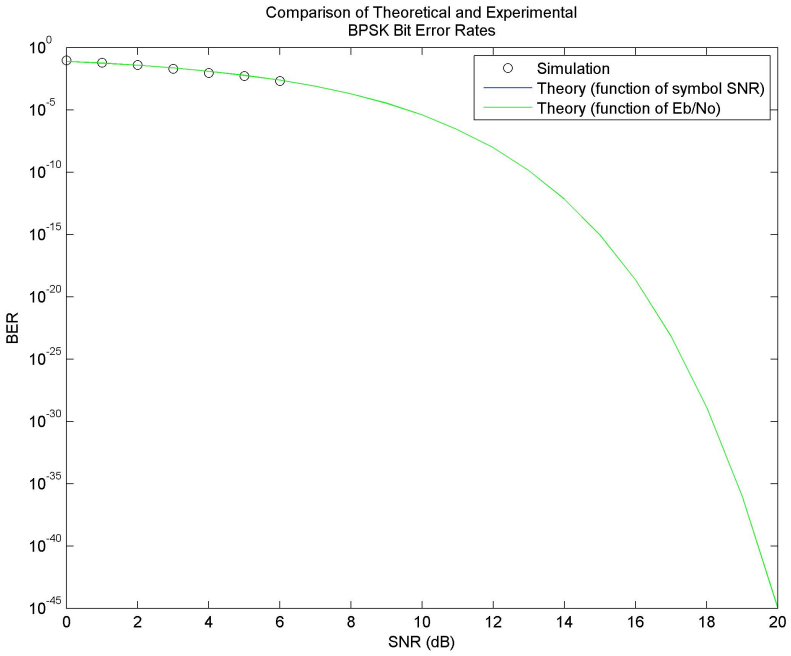


Figure 1

3.1.2 QPSK

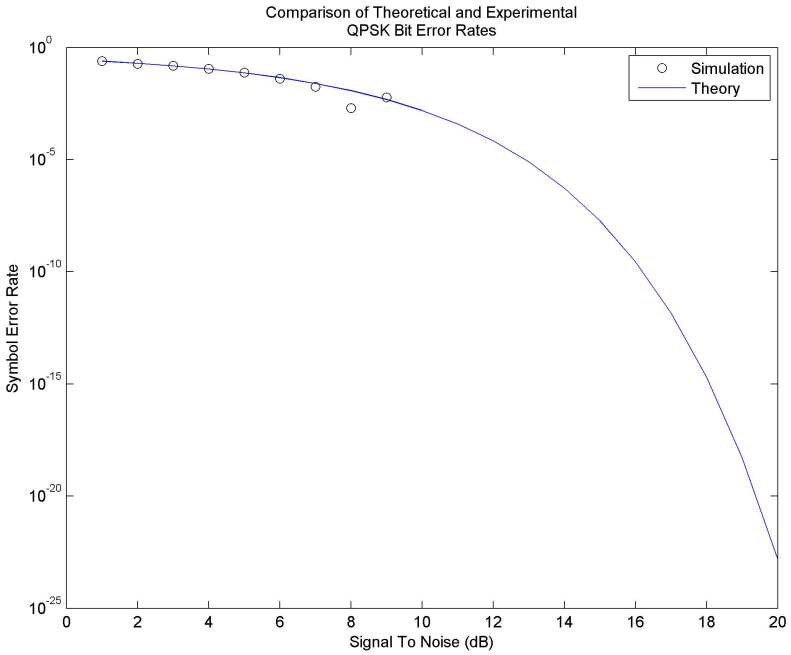


Figure 2

3.1.3 16-QAM

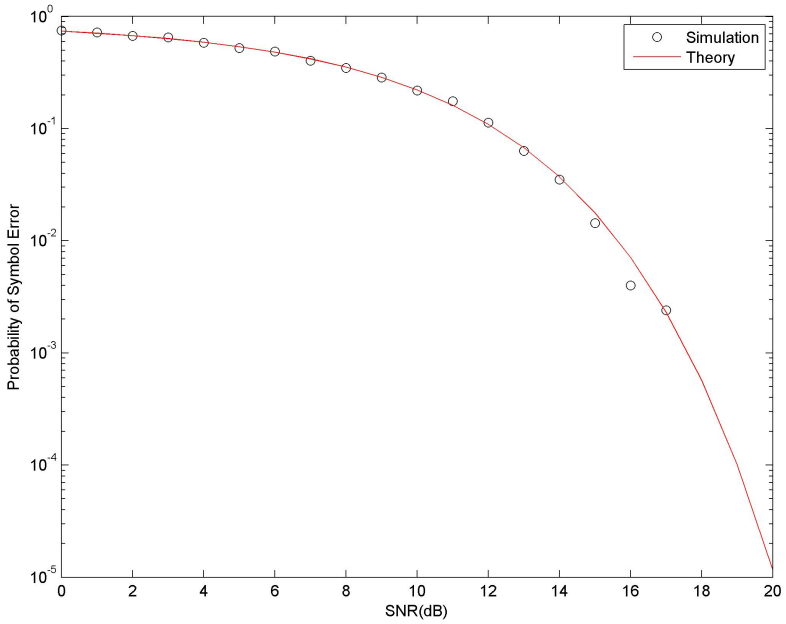


Figure 3

3.1.4 64-QAM

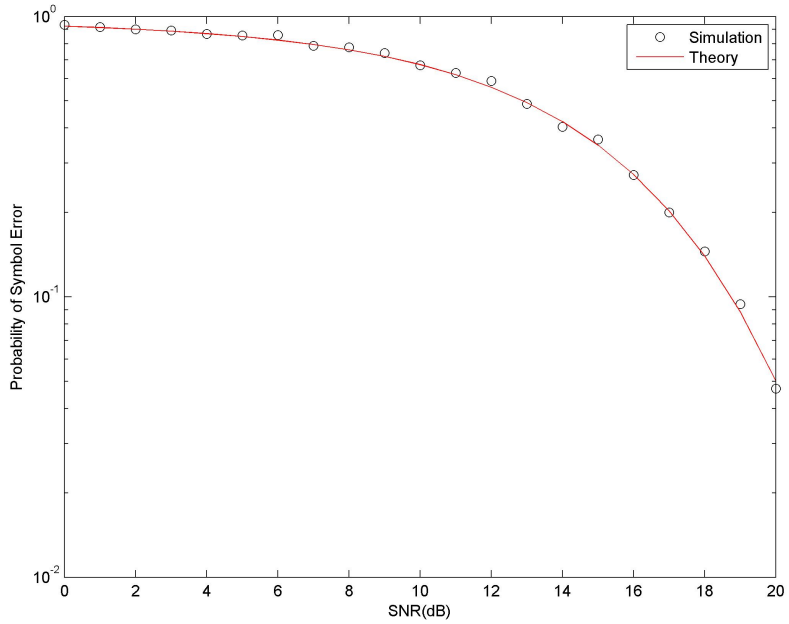


Figure 4

3.2 Constellation Plots

3.2.1 BPSK

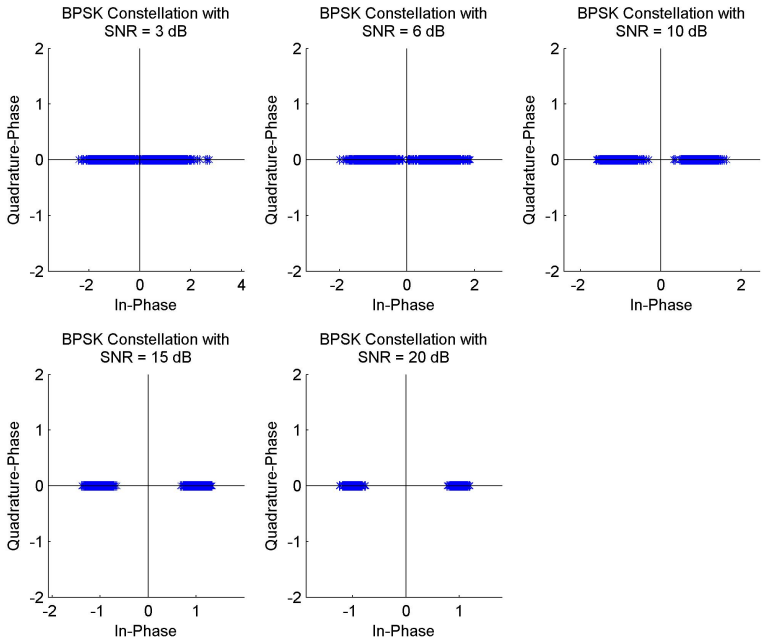


Figure 5

3.2.2 QPSK

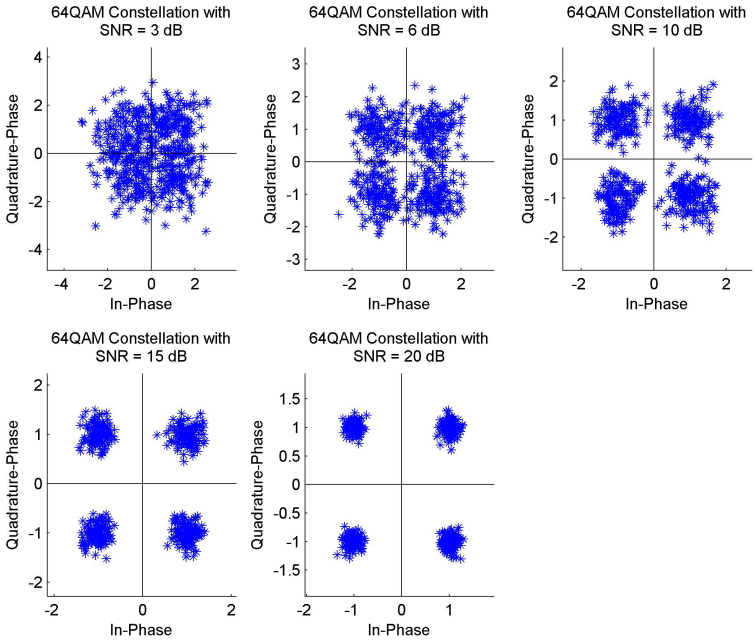


Figure 6

3.2.3 16-QAM

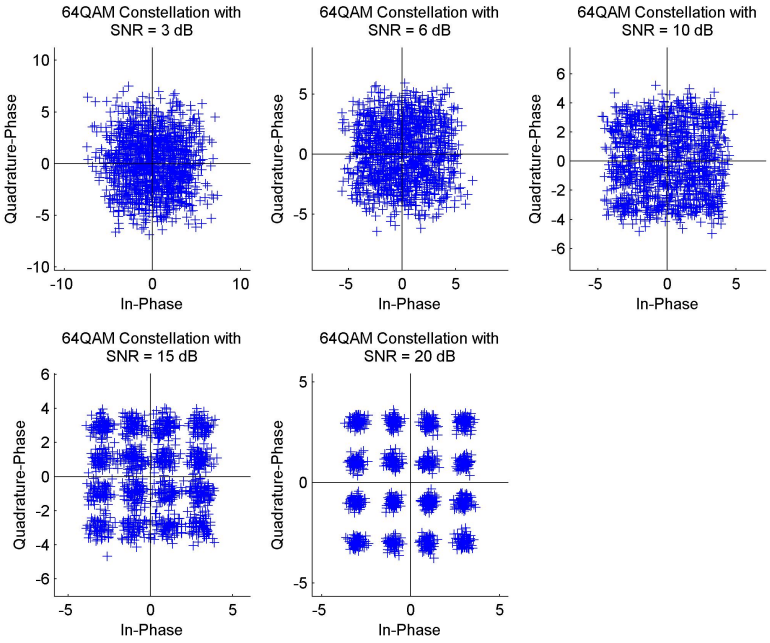


Figure 7

3.2.4 64-QAM

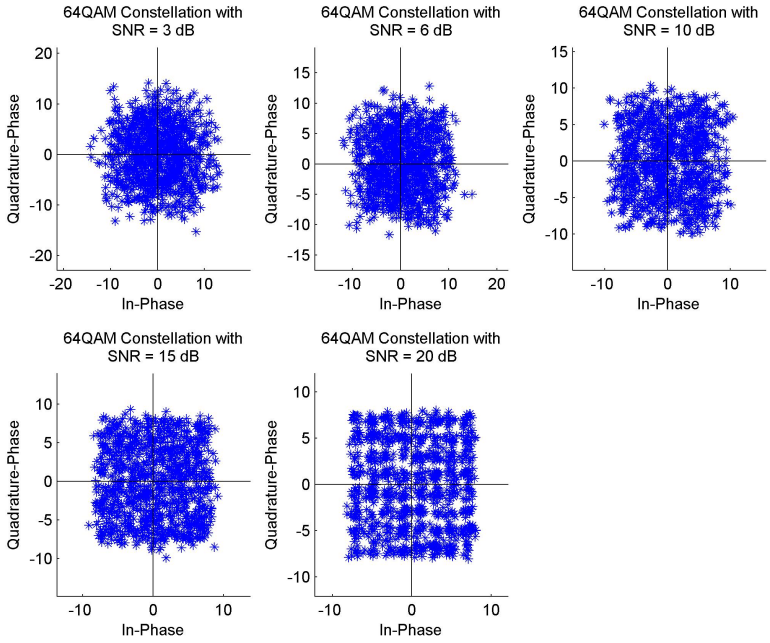


Figure 8

4 Conclusion

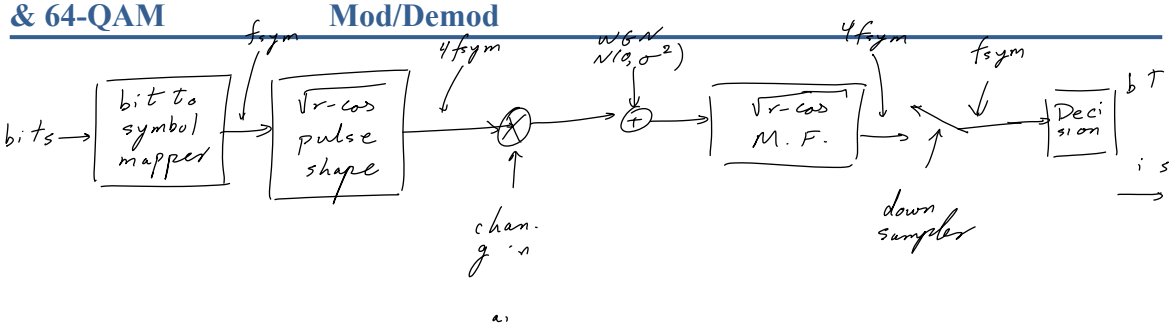
A Project Assignment

Project: EE230B

M-QAM Mod/Demod

The purpose of this project is to provide simulation based validation of the theory being studied in class. In addition we introduce some impairments and look at the impact of these impairments on the performance of a digitally modulated signal.

Step1: BPSK , QPSK, & 16-QAM & 64-QAM Mod/Demod



In Matlab (not simulink), build up a simulation where a random vector of bits are generated. These should then be passed through a bit to symbol mapper, a sqrt-raised cosine pulse shaping filter (oversample by 4 relative to the symbol frequency). Pass the resulting waveform through a channel with unity gain and AWGN with variance σ^2

Given a desired SNR, your simulation must automatically set the channel gain and the AWGN variance.

At the receiver model the receiver matched filter, the symbol slicer, and the symbol to bit mapper.

Repeat the above for a BPSK, QPSK, and 16-QAM, 64-QAM systems.

For each of the modulation formats BPSK, QPSK, 16-QAM, 64-QAM

- plot on a single graph
  - The theoretical Bit Error Rate curve as a function of the symbol SNR
  - The theoretical Bit Error Rate curve as a function of Eb/No
  - The Bit Error rate curve from your simulation as a function of the received signal's SNR
- Please note, I am looking for four graphs (for each of the four modulation formats BPSK, QPSK, 16-QAM, 64-QAM) with each one having 3 curves on it.
- Plot the following constellations by looking at the signal just before the decision block
  - BPSK, QPSK, 16-QAM and 64-QAM at Symbol SNRs of 15 dB SNR
  - QPSK and 64-QAM at input SNRs of 3 dB, 6 dB, 10 dB, 20 dB

What to hand in:

- Derivation of the theoretical bit error rate curves

- Explanation of how you measured the SNR in the simulation and why this is the correct way to measure it
- The BER/SER plots asked for above
- The Constellation plots asked for above
- Matlab source code for:
    - Bit to symbol mapper
    - Sqrt Raised cosine pulse shape
    - Channel gain and noise source
    - Sampler
    - Decision block
    - Note: the code that you submit must be well documented and commented so that it is easily understood by a novice Matlab programmer
- A short text (no more than a page) explaining
    - How many bits did you pass through your simulation to get the desired BER curves?
    - any discrepancies between the theoretical and the simulation results.
    - the challenges, if any, that you encounters and lessons learned.
    - What was the most difficult part of this step in the project?

Step 2: QPSK & 16-QAM Mod/Demod with Carrier Offset

Augment the system of step 1 with a block to model

1. Carrier phase offset
2. Carrier frequency offset

For a carrier phase offset of 5, 10, 20, and 45 degrees

- Plot the constellation for a 16-QAM system operating at 20 dB SNR
- Plot, using your simulation, the BER of BPSK, QPSK, 16-QAM and 64-QAM systems as a function of the input SNR, repeat using Eb/No
- Plot the theoretical BER curves for BPSK, QPSK, 16-QAM and 64-QAM (you can either derive this on your own, or find a paper where it is derived, either way you will need to clearly show the derivation in your report).
- Compare the BERs obtained here with those of step one. Comment on the difference, and explain hour carrier phase offset can impact a system's performance

For a carrier frequency offset of 0.01, 0.1, 1, 10 Hz plot the constellation for BPSK and QPSK at SNRs of 6 dB and 20 dB. You can assume that the input bit stream is at 1 Mbps.

What to hand in:

- Matlab source code the describes how you generated the carrier frequency and phase offsets
- All plots and explanations asked for above
- A short text describing the major issues you faced and how you overcame them.

Step 3: Frequency Recovery

This step augments the simulation setup of Step 2. We look to add a carrier recovery loop so that the receiver can estimate and later correct both carrier phase and frequency offsets.

You are to implement two types of frequency correction loops. The first is a costas loop for QPSK based signals, and the second is a decision directed frequency recovery loop.

Costas Loop for QPSK based signals:

- Plot the S-curve of the phase detector as a function of phase offset
- Plot the s-curve of the carrier detector as a function of carrier offset
- With the loop closed plot the transience of the carrier recovery loop for the following four combinations assuming a phase offset of 30 degrees
    - QPSK
    - an SNR of 30 dB and 6 dB
- Plot the transience of the carrier recovery loop for a carrier offset of 1 ppm (parts per million) and 30 ppm relative to the symbol rate
- Plot the BER curves, where should you start measuring the bit errors?

Repeat the above for a decision directed frequency recovery loop.

Step 4: Data Converters

Augment the simulation of step 1 to include the effect of data converters.

Model the D/A converter as a zero order hold followed by a low pass anti-aliasing filter

- A zero order hold converts the digital sample to an analog voltage and then "maintains" it for the duration of a sample period.
- In your simulation make the sampling period of the analog portion of the simulation to be 80 times the symbol rate
- The low pass filter should be a 4^{th} order butterworth filter.

The A/D converter is effectively a sub-sampling circuit which converts the analog signal (modeled as an oversampled digital signal in your simulation) to a digital stream with sampling frequency equal to 4*fsym.

- You must include noise-limiting lowpass filter in front of the A/D converter. Use the same 4^{th} order Butterworth filter as on the TX side (although it need not be the same exact filter)

- What is the impact on the BER of choosing a good or bad sampling instant for the A/D converter? Explain.
- What is the impact of shifting the cut off frequency of the TX anti-aliasing LPF? Explain.
- What is the impact of shifting the cut off frequency of the RX noise-limiting filter? Explain.
- Compare the BER curves for the system with the data converters with those of the system without data converters. Comment on the difference

- What would you have to do to perfectly match the BER curves of the system with the data converters with the theoretical BER curves?
    - Try out your guess in simulation and present the results. Explain exactly why or why not you were able to achieve the theoretical limit?

Step 5: ISI and Equalization

Replace the channel gain block developed in step 1 to a convolution with a channel representing each of the following three channels:

$$h_1(t) = 1\delta(t) + 0.25\delta(t - T_{sym})$$

$$h_2(t) = 1\delta(t) - 0.25\delta(t - T_{sym}) + 0.125\delta(t - 2T_{sym})$$

$$h_3(t) = 0.1\delta(t + T_{sym}) + 1\delta(t) - 0.25\delta(t - T_{sym})$$

For each of the channels, plot the constellation diagram for a BPSK and QPSK transmitted signal and comment on what you see.

For each of the channels plot the BER for a BPSK system and comment on the difference between the BER curves for this system and those obtained with the single tap channel of Step 1

Equalize the system using both the ZF, MMSE, and the MMSE-DFE equalizers. Plot the BER curves for a BPSK system and comment on the difference.

Step 6: Final Report

For this step we look to setup a complete end to end system that comprises all the elements developed in steps 1 through 5.

Setup an end to end simulation of an M-QAM system, that can operate in either QPSK or 16-QAM modes. Your simulation should model the following elements:

1. Signal bandwidth of 10 MHz
2. Raised-cosine pulse shape, broken down into two square-root raised cosine filters. One at the TX and one at the RX. The roll-off factor for the raised-cosine pulse is 0.75.
3. Data converters, both D/A and A/D with sufficient oversampling rate to accurately model the analog anti-aliasing filters, but not so excessive that the simulation speed is compromised
4. Carrier frequency offset of 100 KHz
5. Anti aliasing filters should be 4th order butterworth filters with cut-off frequency of 5 MHz.
6. You should use the following channel response:

$$h(t) = 0.1\delta(t + 100ns) + 0.8\delta(t + 40ns) + 0.95\delta(t - 30ns) + 0.7\delta(t - 100ns) - 0.35\delta(t - 170ns)$$

You can assume perfect knowledge of the channel, but not the frequency offset. Design the system so as to achieve a BER performance as close as possible to the ideal performance obtained in step 1 of the project for QPSK and 16-QAM.

Plots to include in your report:

- Phase error between the local oscillator at the RX and the incoming signal
- Eye diagram of the signal after the receive matched filter, and after the equalizer
- Eye diagram of the system with an AWGN channel.
- BER curves of the system with an AWGN channel and with the channel response given above.

Commentary:

- Admittedly, there are no elements in this final step that you had not implemented and/or tested in the first 5 steps. Comment on the relative ease (or difficulty) of achieving end to end communication for the given system. Why was it easy or difficult.
- Write a report similar to a technical document that might be written in a company that you'd be working at. This report should be written from the vantage point that someone in two years is to replicate exactly what you implemented. They need to know what you did exactly so that they could build the system exactly as you built it. Moreover, you should provide them with sufficient test data and instructions so that once they have assembled all the pieces they can be sure that it is the same exact performance as you. Also provide suggestions as to how the performance of the system can be improved (if at all).

B 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
```

C Bit to Symbol Mappers

C.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 -

% 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
```

C.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 input binary datastream length

% 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
```

C.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 -

% 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


%normalization
% sym_quad = sym_quad.*(1/sqrt(10));
% sym_inp = sym_inp.*(1/sqrt(10));


end
```

C.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 -

% 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
```

D 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*rollOffF
    end

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

end
```

E 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
```

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 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
```

H 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
```

I Decision Blocks

I.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
```

I.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
```

I.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);

% inphase_sig = inphase_sig.*(sqrt(10));
% quadrature_sig = quadrature_sig.*(sqrt(10));

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
```

I.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
```