EECS 865 SIMULATION PROJECT

VIJAYA CHANDRAN RAMASAMI, KUID 698659

Contents

List of Figures

1. Overview

This project report is organized as follows.

- **Simulation Setup and Block Diagram** - explains the Simulation Block Diagram, Simulation Parameters and a brief description of each of the Simulation blocks.
- $\pi/4$ **- DQPSK Modulation** - explains the basic math behind $\pi/4$ - DQPSK Modulation and some implementation details.
- **Rayleigh Fading Envelope Generation** - explains the basic methodology adopted to generate the Rayleigh Fading envelope and to implement Spectral Shaping.
- **Results** - provide the results for the simualtion.
- **MATLAB Modules for Simulation Blocks** - provides the MATLAB code for each and every *simulation block*.
- **MATLAB Code** - contains the MATLAB code used for the simulation cases. The code presented in this section uses the simulation blocks presented in the previous section.
- **Appendix A - Raised Cosine Filtering** - explains the method used to generate the square root raised cosine filter coefficents that can be used directly in the code with minor modifications.
- **Appendix B - Rayleigh Fading Generation (Clarke/Gans Model)** - explains the method and the MATLAB code to generate Rayleigh Fading using the *Clarke/Gans Model*.

2. Simulation Setup and Block Diagram

The Block Diagram for this simulation is illustrated in fig(1).

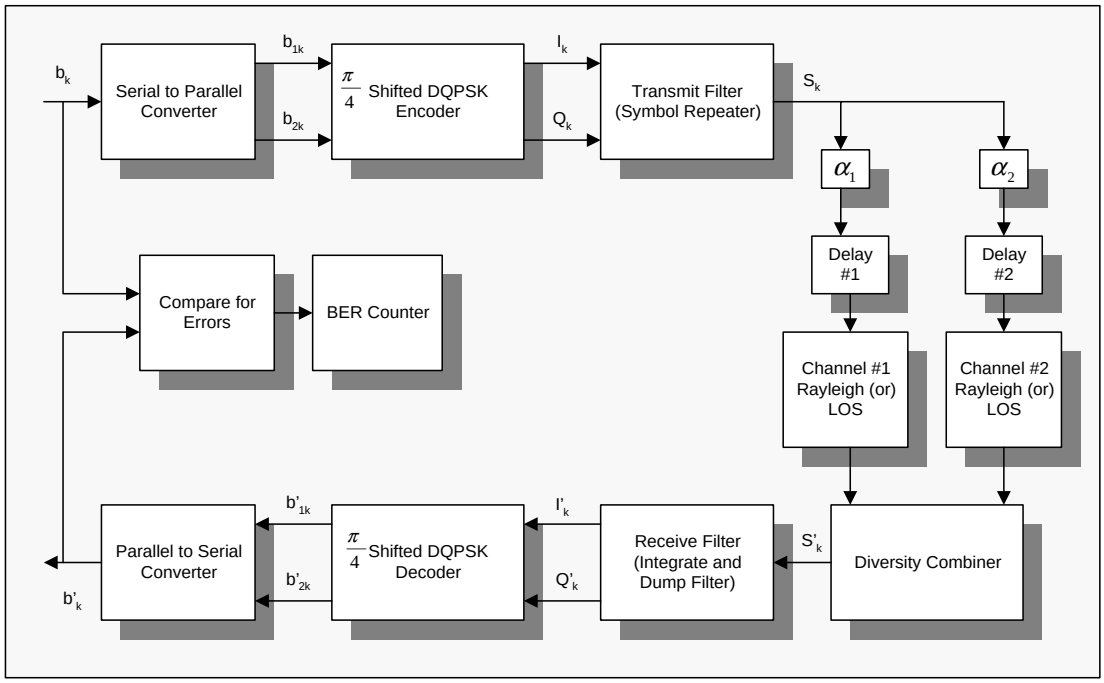

Figure 1. Simulation Block Diagram

2.1. **Monte-Carlo Simulation Technique.** The simulation methodology followed was the well-known Monte-Carlo simulation technique. In the context of BER estimation in Digital Communication Systems, the MC Simulation technique involves the following steps :

(1) Decide on the minimum target BER to be estimated. (Here, it is 10^{-3}).
(2) Set the number of bits per simulation run to be *atleast* 10 times the inverse of the minimum target BER to be estimated. (Here, it is atleast 10^4 bits).
(3) Setup the baseband modulators, demodulators, Transmit/Receive Filters and Channel Simulators.
(4) Run the BER simulation and estimate the BER.
(5) Iterate the simulation for some specified number of iterations and compute the average of the BERs obtained in these iterations. (Here, the number of simulation runs was chosen to be 20).

The various simulation blocks are explained in the following sections.

2.2. **Serial to Parallel Converter.** This block converts the incoming information bits into two streams, one containing the even numbered bits and the other containing the odd numbered bits. The output pair of bits constitute the input symbol stream to the $\pi/4$ - DQPSK Modulator.

MATLAB Prototype.

```
function [BitStreamOne, BitStreamTwo] = SerialToParallel(BitStream)
```

2.3. $\pi/4$ **- DQPSK Encoder.** This block encodes the input information bits $\{b_{1k}, b_{2k}\}$ into Modulation Symbols $\{I_k, Q_k\}$ using $\pi/4$-DQPSK Signal Mapping.

MATLAB Prototype.

```
function [I_SymbolsTx, Q_SymbolsTx] = DQPSKEncoder(BitStreamOne,BitStreamTwo)
```

2.4. **Transmit Filter.** This block converts the Modulation Symbols to *Baseband* Waveforms to be transmitted over the channel. In the given problem, this tranmit filter is just a symbol repeater (represented by an FIR impulse response of all ones for the symbol interval). This block can also be used to perform Raised Cosine filtering if provided with the proper FIR filter coefficients for the SQRC filter.

MATLAB Prototype.

```
function [I_WaveformTx,Q_WaveformTx] = TransmitFilter(I_SymbolsTx,  \\
                        Q_SymbolsTx, hTransmitFilter, numSamplesPerSymbol)
```

2.5. **Delays.** This block introduces a delay of specific duration (represented as the number of samples).

2.6. **Channel Simulators.** These blocks simulate the channel response. They take the I and Q baseband waveforms and apply channel-specific distortions to them. Two types of channel simulators are provided.

- AWGN Channel Simulator.
- Rayleigh Fading Channel Simulator.

Fading and Noise addition are done independently for the I and Q components.

MATLAB Prototypes.

```
function [I_WaveformRx,Q_WaveformRx] = AWGNChannel(I_WaveformTx,Q_WaveformTx,No)
function [I_WaveformOut , Q_WaveformOut] = RayleighFader(I_WaveformIn,  \\
                                                    Q_WaveformIn, AvgFadePower);
```

2.7. **Receive Filter.** This blocks performs *Matched Filtering* of the received signal. In the case of an all-ones FIR impulse response (i.e, symbol repetition), this filter essentially does an integrate and dump operation. The output of this filter are the received symbols that are fed into the demodulator.

MATLAB Prototype.

```
function [I_SymbolsRx,Q_SymbolsRx] = ReceiveFilter(I_WaveformRx,    \\
Q_WaveformRx, hReceiveFilter, numSamplesPerSymbol)
```

2.8. $\pi/4$**-DQPSK Decoder.** This block performs the Maximum-Likelihood decoding of the received symbols and retrieves the information bits.

MATLAB Prototype.

```
function [BitStreamOneRx, BitStreamTwoRx] = DQPSKDecoder(I_SymbolsRx, Q_SymbolsRx)
```

2.9. **Parallel to Serial Converter.** This block does the inverse of the Serial to Parallel Converter block.

MATLAB Prototype.

```
function [BitStream] = ParallelToSerial(BitStreamOneRx, BitStreamTwoRx)
```

2.10. **Comparator and BER Counter.** The blocks compute the Probability of error (Pe), by counting the number of discrepancies between the input and the received bits.

2.11. **Noise Power Calculations.** The value of N_o for a given operating point (i.e, $(E_b/N_o)_o$) can be obtained as follows :

- If N_b is the number of bits used for simulation and I_k and Q_k represent the inphase and quadrature phase components of the baseband signalling waveform, then the Average Energy per Bit (E_b) [1] is given by,

$$E_b = \frac{1}{N_b} \sum_{i=1}^{Nb/2} (I_k^2 + Q_k^2) \tag{1}$$

  It is important to note that I_k and Q_k are represented by their sample values (8 samples/symbol).
- The value of the noise PSD N_o can be found simply as,

$$N_o = \frac{E_b}{(E_b/N_o)_o} \tag{2}$$

[1] This was the value of E_b that gave the desired result during the simulation

3. $\pi/4$ Shifted Differential Quadrature Phase Shift Keying (DQPSK)

This section deals with the math behind the $\pi/4$ Shifted DQPSK Modulation Technique. This digital modulation technique is a special case of Differential M-ary Phase Shift Keying (D-MPSK) where $M = 4$. In such differential techniques, the information bits are mapped into *phase transitions* rather than absolute phase values. This encoding of information in the phase transitions overcomes the *phase ambiguity problems* resulting from the estimation of carrier phase in non-differential PSK systems. (The term "$\pi/4$ Shifted" appears in the context of constellation diagrams and will be explained later). Further, in $\pi/4$-DQPSK, the maximum phase shift is restricted to $\pm 135^o$ (as compared to 180^o for QPSK and 90^o for OQPSK). Hence, the bandlimited $\pi/4$-DQPSK signal preserves the constant envelope property better than bandlimited QPSK, but is more susceptible to envelope variations than OQPSK.

3.1. I and Q Components.

The In-phase (I) and Quadrature Phase (Q) components of $\pi/4$ DQPSK can be expressed as :

$$I(i) = I(i-1) * cos(\Delta\theta_i) - Q(i-1) * sin(\Delta\theta_i) \tag{3}$$

$$Q(i) = I(i-1) * sin(\Delta\theta_i) + Q(i-1) * cos(\Delta\theta_i) \tag{4}$$

Or,

$$S(i) = S(i-1) * e^{j\Delta\theta_i} \tag{5}$$

Where,

- $I(i)$ and $Q(i)$ are the in-phase and the quadrature phase components of the $\pi/4$ DPSK Modulated Symbol at the i^{th} signalling interval.
- $S(i)$ is the i^{th} modulated symbol.
- $\Delta\theta_i$ is the phase difference between the symbols at the i^{th} and the $(i-1)^{th}$ signalling intervals.

3.2. Phase Shift Mapping.

The Phase shift $\Delta\theta_i$ depends on the *input symbol* $d_i = \{00,01,11,10\}$. Thus, the information bits that consitute the input symbol are encoded into one of the 4 possible phase transitions, defined by the following table.

b_{i1}	b_{i2}	$\Delta\theta_i = f(b_{i1}, b_{i2})$
0	0	$\pi/4$
0	1	$3\pi/4$
1	0	$-3\pi/4$
1	1	$-\pi/4$

Where, b_{i1} and b_{i2} are the information bits that constitute the input symbol d_i.

3.3. Constellation.

The Constellation of $\pi/4$ DQPSK Modulation is shown in fig(2). The constellation diagram shows that $\pi/4$ DQPSK is a *combination of 2 QPSK constellations* shifted with respect to each other by $\pi/4$ (and hence the name). Further, in $\pi/4$ DQPSK, there is an inherent *feedback*, since the decoding of the present symbol depends upon the decoding (i.e, the phase) of the past symbol.

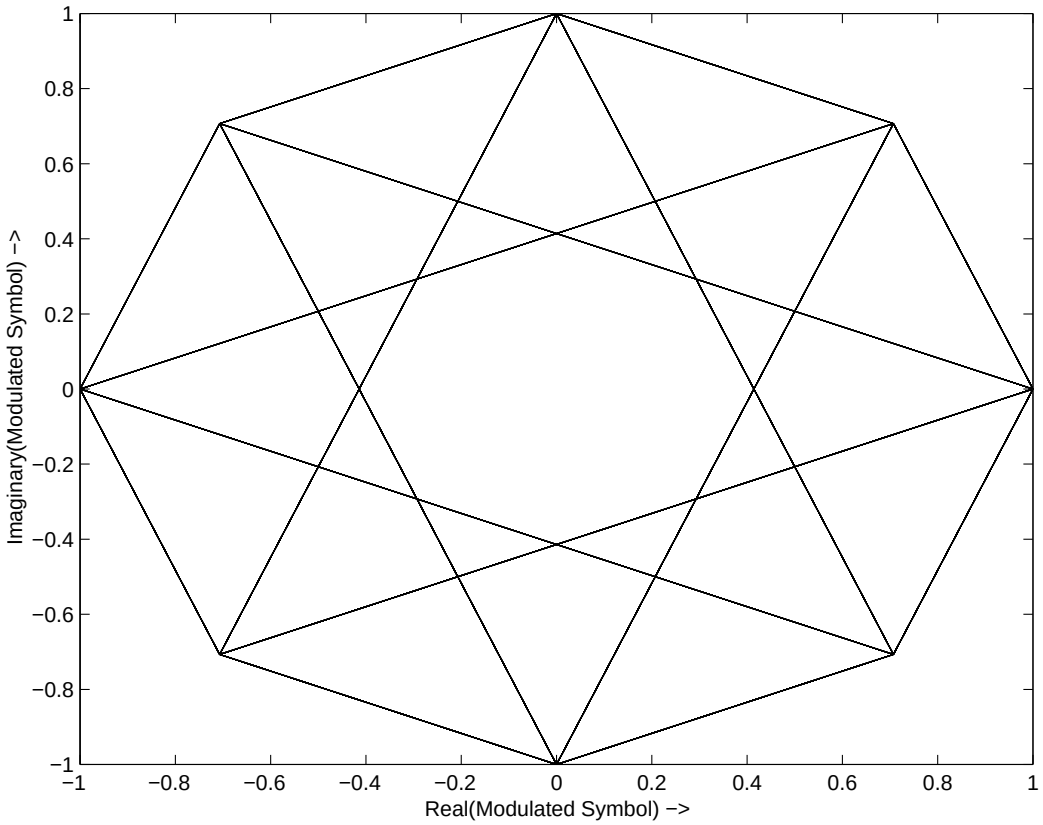

FIGURE 2. $\pi/4$-DQPSK Constellation

3.4. **$\pi/4$-DQPSK Encoder Implementation.** The encoding operation can be easily implemented using the *recursive* relations given in equations (4).

3.5. **$\pi/4$-DQPSK Decoder Implementation.** $\pi/4$-DQPSK decoding is generally accomplished using differentially coherent detection.

- Compute the phase angles of all the received baseband symbols.
- Compute the difference in phase angles between succesive symbols.

$$\Delta\theta_k = \angle S_k - \angle S_{k-1} \tag{6}$$

- Demap the phase values into information bits using the inverse of the phase mapping function.

Once the value of $\Delta\theta_k$ is obtained, the $\pi/4$-DQPSK *decision* rule becomes,

$$b_{1k} = (\sin \Delta\theta_k > 0) \tag{7}$$

$$b_{2k} = (\cos \Delta\theta_k > 0) \tag{8}$$

4. Rayleigh Fading Envelope Generation

The generation of Rayleigh Fading envelopes follow from the basic fact that the envelope of a complex gaussian process (with independent real and imaginary parts) has a Rayleigh Distribution. The general method to generate a Rayleigh Fading envelope is illustrated in fig (3).

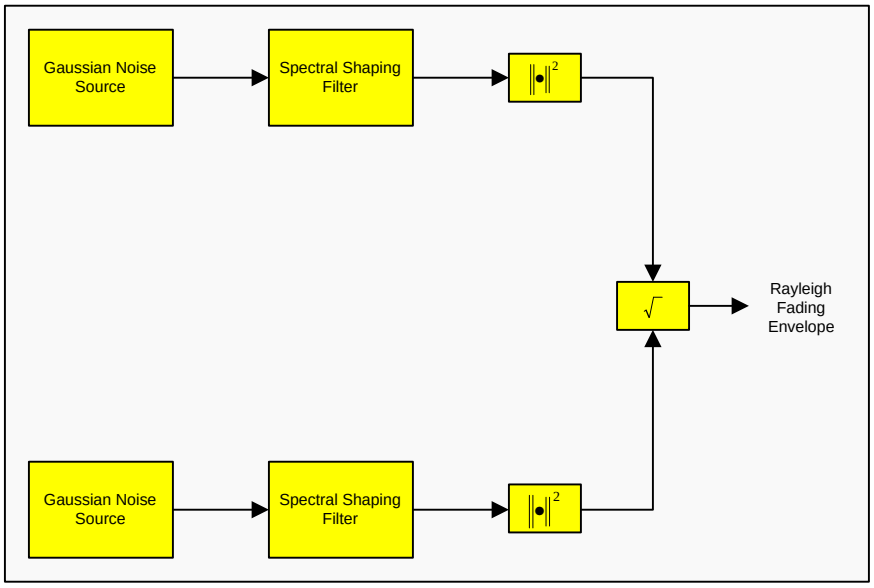

Figure 3. Rayleigh Fading Generation at Baseband

The Spectral Shaping filter is needed to introduce a desired amount of correlation into the gaussian samples that produce the rayleigh distribution. In case of Mobile Communication Systems where Rayleigh fading has to be generated for a particular *speed* of the mobile, the spectral shaping filter takes the form of a *Doppler Filter* with the maximum doppler spread specified by the Mobile Speed (Clarke/Gans Model).

If $I_g(n)$ and $Q_g(n)$ represent the in-phase and quadrature phase components of the complex gaussian process (*after* spectral shaping), the rayleigh fading envelope can be generated as,

$$\alpha(n) = \sqrt{I_g(n)^2 + Q_g(n)^2} \tag{9}$$

4.1. Spectral Shaping Filter.

The Spectral Shaping filter is usually specified in terms of its Autocorrelation function or Power Spectral density. When a Power Spectral Density is specified, the *colored* Gaussian samples can be generated by passing the *white* Gaussian Noise samples through a filter whose transfer function $H(f)$ can be obtained by solving,

$$H(f)H^*(f) = S_{xx}(f) \tag{10}$$

Where, $S_{xx}(f)$ is the power spectral density of the filter. The digital implementation of $H(f)$ can be done either using FFT Techniques or FIR/IIR filtering depending on the situation and form of $H(f)$ obtained.

In the given problem, the PSD can be obtained taking the fourier transform of the specified autocorrelation as,

$$R_{xx}(\tau) = e^{-1000\tau} \Rightarrow S_{xx}(f) = \frac{2000}{1000^2 + (2\pi f)^2} \tag{11}$$

Further, the $H(f)$ can be obtained from the power spectral density as,

$$H(f) = \frac{\sqrt{2000}}{1000 + j2\pi f} \tag{12}$$

The digital implementation of the above transfer function can be done in a plethora of ways. But the best (and the most relevant) method is the IIR implementation of the above filter using AR models. Even then, a choice has to be made between IIR filter synthesizing techniques such bilinear, impulse-invariance, backward/forward difference methods etc. Since the filter has a simple one-pole type transfer funtion, it is much better to use impulse invariance rather then other techniques. Proceeding futher, the impulse response of the digital filter is obtained as,

$$H(s) = \frac{\sqrt{2000}}{s + 1000} \Rightarrow H(z) = \frac{\sqrt{2000}}{1 - e^{-\frac{1000}{T}} z^{-1}} \tag{13}$$

Where T represents the sampling duration. In our case, because of the *slow fading* assumption, we have to generate *one* rayleigh fading envelope sample per symbol (i.e, 8000 samples per second). This works to to a sampling duration of $T = 0.125ms$. Thus we get,

$$H(z) = \frac{44.72}{1 - 0.8825z^{-1}} \tag{14}$$

Which converts to the simple differential equation,

$$y[n] = 0.8825y[n-1] + 44.72x[n] \tag{15}$$

This system can be easily implemented using the "filter" command in MATLAB.

4.2. **Fade Power Adjustment.** Suppose we require a specified average fade power P. Given the generated fading samples $\{\alpha_k\}$, we can generate the fading samples with the given average fade power P using the transformation,

$$\tilde{\alpha}_k = \frac{\sqrt{P}}{\sqrt{E[\alpha^2]}} \times \alpha_k \tag{16}$$

4.3. **Simulated Envelope.** The simulated rayleigh fading envelope at baseband is shown in fig(4).

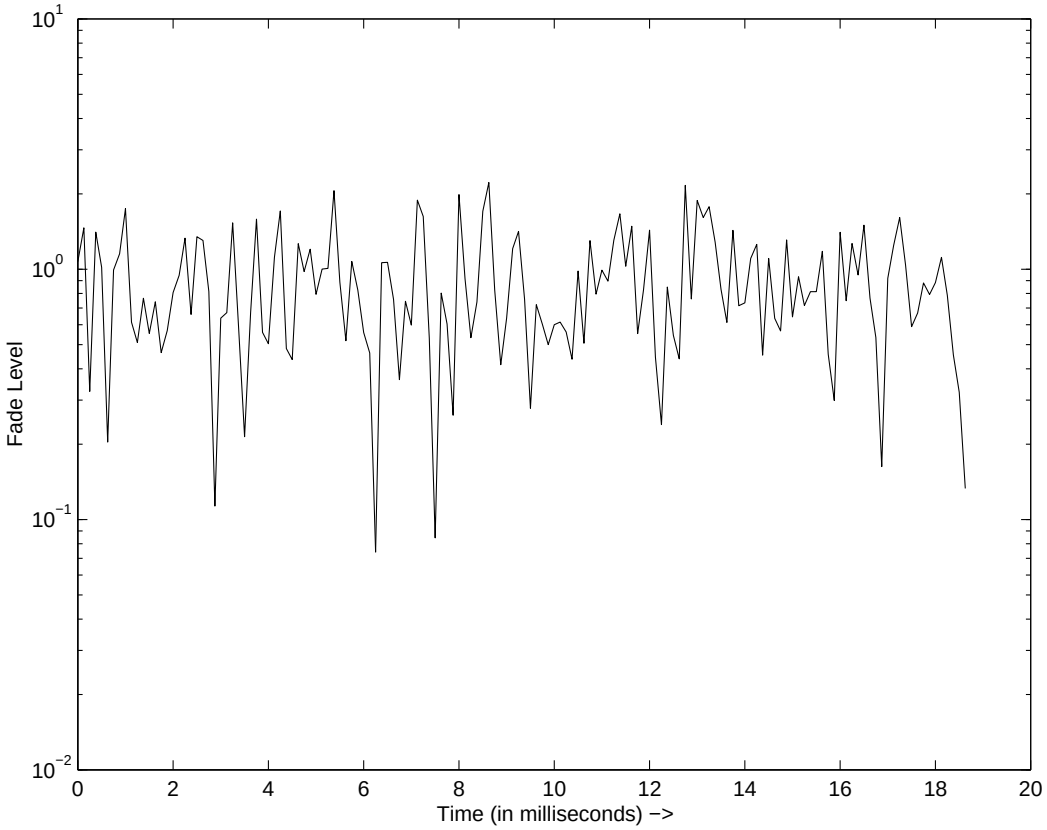

FIGURE 4. Simulated Rayleigh Fading Signal at Baseband (E[Power] = 1)

4.4. **Faded SNR per bit.** When fading is considered, the average signal to noise ratio per bit (γ_b) has to be redefined as,

$$\gamma_b = E\{\alpha_k^2\}\frac{E_b}{N_o} \tag{17}$$

The operating points during the simulation are chosen based on the above expression for γ_b. Since the SNR per bit is a product of the average fade power ($E\{\alpha_k^2\}$) and the unfaded E_b/N_o, it is an implementation issue to decide on their values given a value of γ_b as an operating point.

5. Simulation Results

5.1. **Simulation Parameters.**

- Method - Monte Carlo Simulation.
- Number of Bits per iteration = 10000 (Case I), 5000 (Case II), 3000 (Case III).
- Number of iterations = 20.

5.2. **Case - I.** The channel used for this simulation is an single ray, AWGN channel with an average power gain of 1.

5.2.1. *Calibration.* A $\pi/4$- QPSK Encoder/Decoder combination (*without* differential encoding/decoding) was used to calibrate the system. The Pe for QPSK is $Q(\sqrt{2 * E_b/N_o})$. The theoritical and the practical curves for QPSK matched properly which implies that all the other modules are functioning properly. Thus the system is properly calibrated.

The Pe for $\pi/4$-DQPSK using *differentially coherent* demodulation is different from that of QPSK. The simulation results are shown in fig(5).

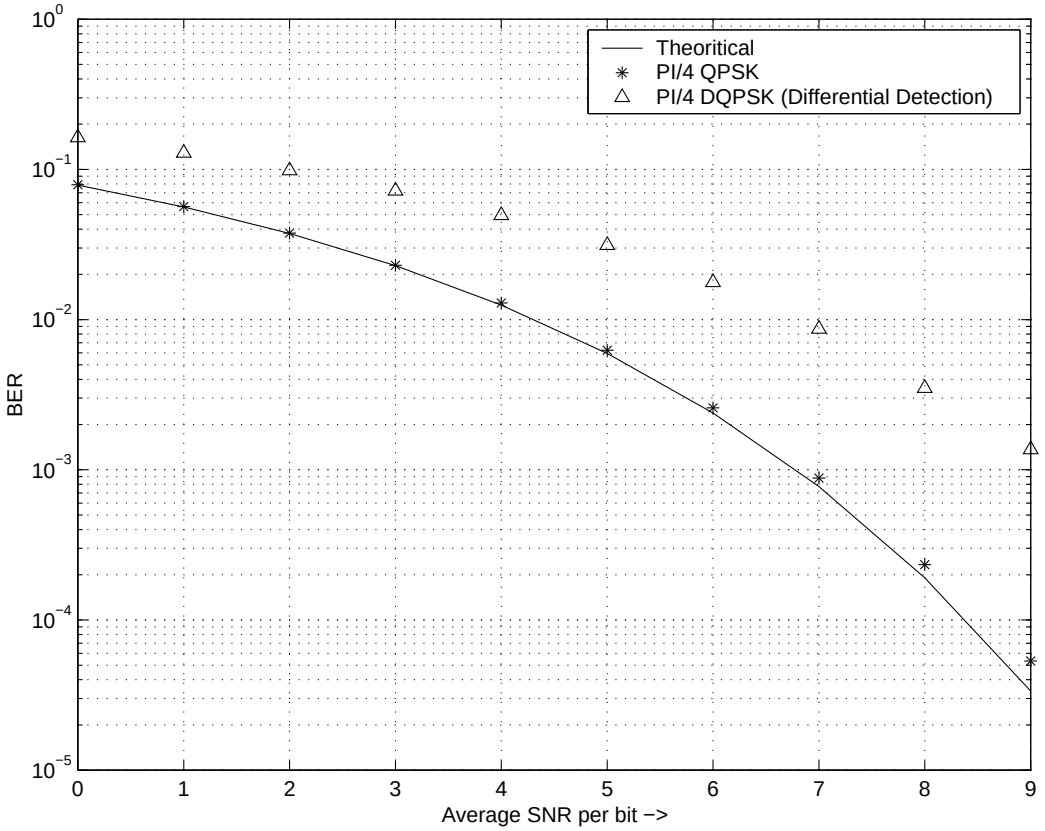

Figure 5. BER vs E_b/N_o plot for $\pi/4$-DQPSK (Case-1)

It is seen from the graph that the performance of $\pi/4$-DQPSK with differential detection is about *3-dB poorer* that of $\pi/4$ QPSK operating under the same conditions.

5.3. **Case II.** The channel used for this simulation is a 2-ray channel consisting of:

- An LOS AWGN channel with an average power gain of 0.5.
- A Rayleigh Fading Channel with an average power gain of 0.5.

There is no time delay in the response of these channels. The simulation results are shown in fig(6).

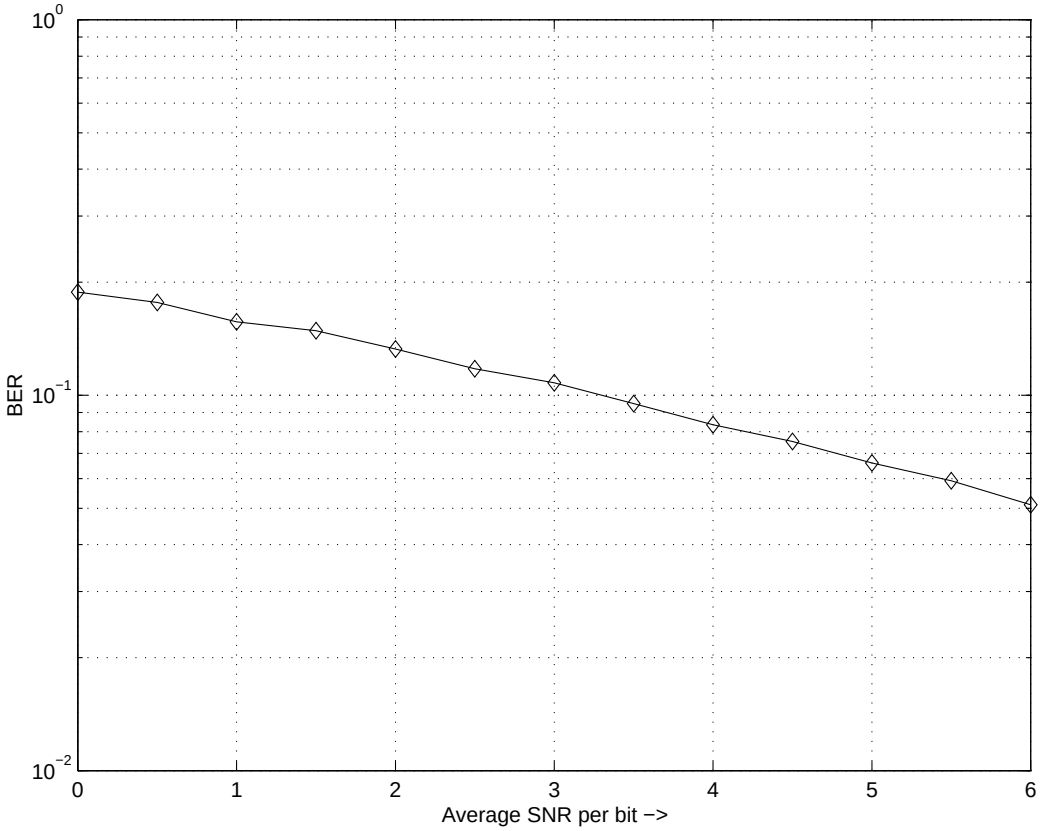

FIGURE 6. BER vs E_b/N_o plot for $\pi/4$-DQPSK (Case-2)

5.4. **Case III.** The channel used for this simulation consists of 2 independent rayleigh faded channels with an average power gain of 0.5. Three different values of delays (0,0.1T_s,0.2T_s were simulated. *Only* for this case, the number of samples per symbol was chosen to be 10, so that an integer value is obtained for the delay in terms of samples. The results are shown in fig(7).

5.5. **Comprehensive Plot.** Combining all the three plots, we get fig(8).

5.6. **Notes.**

- The simulations were repeated using Square-Root Raised Cosine Filters and identical results were obtained.

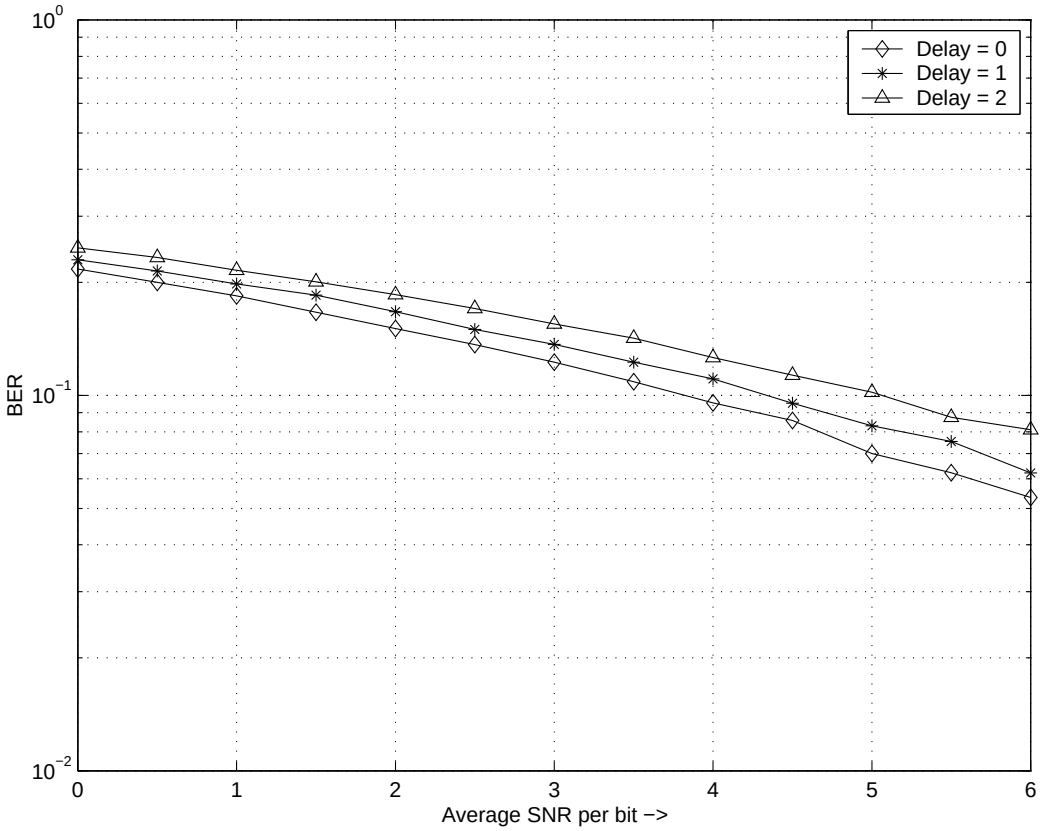

FIGURE 7. BER vs E_b/N_o plot for $\pi/4$-DQPSK (Case-3)

- All the plots were generated by storing the simulation results in a data file and plotting them offline.

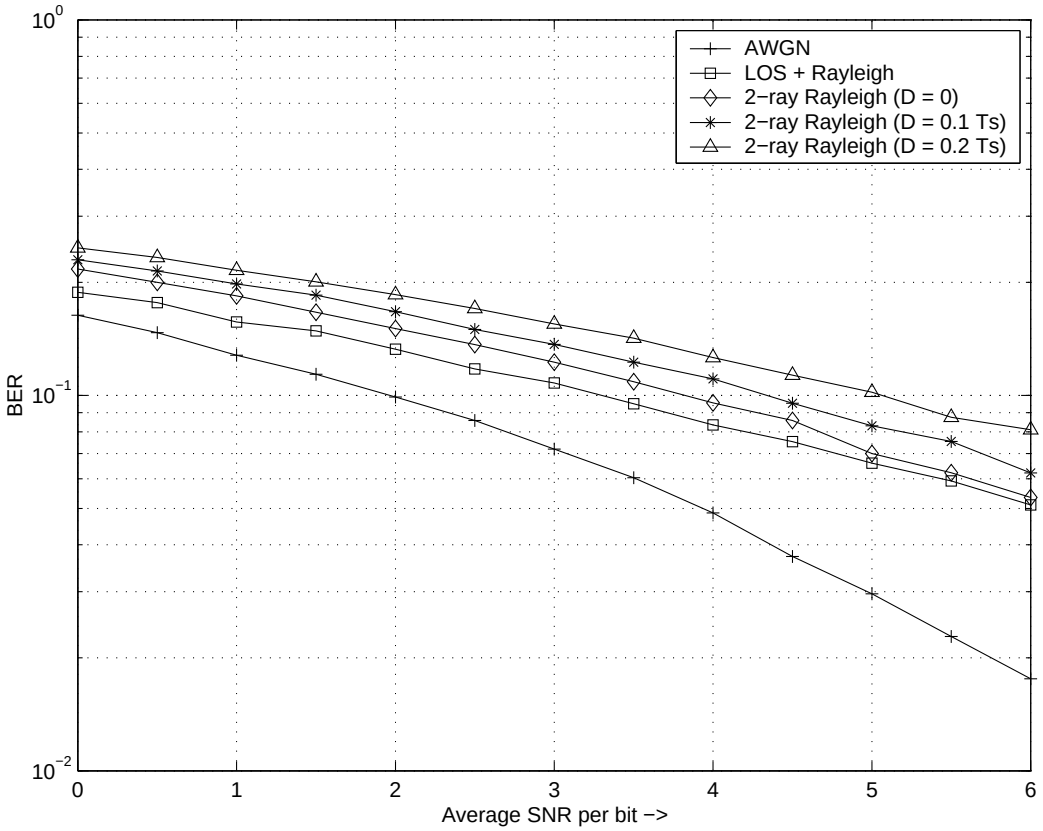

FIGURE 8. Comprehensive plot for all the Results

6. MATLAB Modules for Simulation Blocks

Some implemenation issues and MATLAB code for all the simulation blocks are presented in this section.

6.1. **Serial To Parallel Converter.**

```
% Serial to Parallel Converter.
function [BitStreamOne,BitStreamTwo] = SerialToParallel(BitStream)

BitStreamOne = BitStream(1:2:length(BitStream));
BitStreamTwo = BitStream(2:2:length(BitStream));
```

6.2. $\pi/4$**-DQPSK Encoder.**

```
% pi/4 shifter DQPSK Encoder.
function [I_Symbols, Q_Symbols] = DQPSKEncoder(BitStreamOneTx, BitStreamTwoTx)

% This is supposed to be (I(-1) + jQ(-1)).
InitialSymbol = (1+0i);
I_StreamLength = length(BitStreamOneTx);
Q_StreamLength = I_StreamLength;

% ------------------> Differential Modulation <------------------------------ %
% Do the Differential Modulation and generate the baseband-complex signal.
% Calculate the phase-shift due to the first symbol.
PhaseShift(1) = CalcPhase([BitStreamOneTx(1),BitStreamTwoTx(1)]);
% The first modulated symbol.
ModulatedSymbol(1) = (InitialSymbol)*exp(i*PhaseShift(1));
% The other symbols calculated iteratively.
for index = 2:1:I_StreamLength
  % The phase shift due to the ith modulated symbol.
  PhaseShift(index) = CalcPhase([BitStreamOneTx(index),BitStreamTwoTx(index)]);
  % Rotated the modulated symbol phasor to the new position.
  ModulatedSymbol(index) = ModulatedSymbol(index-1)*exp(j*PhaseShift(index));
end

% Extract the I and Q parts of the Symbol.
I_Symbols = real(ModulatedSymbol);
Q_Symbols = imag(ModulatedSymbol);
%-----------------> End of Code <-------------------------------------------%

% CalcPhase -> does the phase mapping.
function Phase = CalcPhase(BitVector)
switch BitVector(1)
case 0,
   switch BitVector(2)
   case 0, % [0,0] case
```

```
      Phase = pi/4;
   case 1, % [0,1] case
      Phase = 3*pi/4;
   end
case 1,
   switch BitVector(2)
   case 0, % [1,0] case
      Phase = -pi/4;
   case 1, %[1,1] case
      Phase = -3*pi/4;
   end
end
```

6.3. **Transmit Filter.**

```
function [I_TxWaveform, Q_TxWaveform] = TransmitFilter(I_Symbols,Q_Symbols, \\
                                         hTransmitFilter,numSamplesPerSymbol)

% The first step is to convert the symbol stream into a digital signal so that it can
% be filtered.
I_Waveform = SymbolToWaveform(I_Symbols,numSamplesPerSymbol);
Q_Waveform = SymbolToWaveform(Q_Symbols,numSamplesPerSymbol);

% The next step is to filter the signal to obtain the transmit waveforms.
I_TxWaveform = conv(I_Waveform,hTransmitFilter);
Q_TxWaveform = conv(Q_Waveform,hTransmitFilter);
%-------------------------------------------------------------------------%

% Converts a Symbol stream to a Waveform containing impulses.
function [Waveform] = SymbolToWaveform(SymbolStream,numSamplesPerSymbol)

lenWaveform = length(SymbolStream)*numSamplesPerSymbol;
Waveform = zeros(1,lenWaveform);
Waveform(1:numSamplesPerSymbol:lenWaveform) = SymbolStream;
```

6.4. **Rayleigh Fading Generator.** The actual routine that generates the Rayleigh Fading Envelope is :

```
% Rayleigh Fading generator
function [fadeEnvelope] = GenerateRayleighFade(NumSamples, AvgPower)

% First generate the I and Q Gaussian Sequences.
I_Gaussian = randn(1,NumSamples);
Q_Gaussian = randn(1,NumSamples);

% Pass the samples thro a spectral shaping filter.
If_Gaussian = filter([44.72], [1 -0.8825], I_Gaussian, randn);
```

```
Qf_Gaussian = filter([44.72], [1 -0.8825], Q_Gaussian, randn);

% Generate the Fade Envelope.
fadeEnvelope = sqrt(If_Gaussian.*If_Gaussian + Qf_Gaussian.*Qf_Gaussian);

% Adjust the fade power level to the one specified.
rmsEnvelope = sqrt(mean(fadeEnvelope.*fadeEnvelope));
fadeEnvelope = fadeEnvelope/rmsEnvelope;
fadeEnvelope = fadeEnvelope*sqrt(AvgPower);
```

The routine that applies this fading signal generated to the input waveform are :

```
function [I_WaveformOut, Q_WaveformOut] = RayleighFader(I_WaveformIn,     \\
                                                        Q_WaveformIn, AvgFadePower)

lenWaveform = length(I_WaveformIn);

I_Fade = GenerateRayleighFade(lenWaveform, AvgFadePower);
Q_Fade = GenerateRayleighFade(lenWaveform, AvgFadePower);

I_WaveformOut = I_WaveformIn.*I_Fade;
Q_WaveformOut = Q_WaveformIn.*Q_Fade;
```

6.5. **Receive Filter.**

```
function [I_RxSymbols,Q_RxSymbols] = ReceiveFilter(I_RxWaveform,Q_RxWaveform, \\
                                                   hReceiveFilter,numSamplesPerSymbol)

% Do the matched filtering first.
I_FilterOutput = conv(I_RxWaveform,hReceiveFilter);
Q_FilterOutput = conv(Q_RxWaveform,hReceiveFilter);

% It's assumed here that the transmit and the receive filters are of the same length.
% In actual it is ((Nt+Nr-1)-1)/2 = (Nt-1) = (Nr-1) (if Nt = Nr).
N_FilterTrailer = length(hReceiveFilter)-1;

% Convert to a symbol stream, taking into account the trailers due to the transmit and
% receive filter impulse responses.
SymbolRange = N_FilterTrailer+1:length(I_FilterOutput)-N_FilterTrailer;
I_RxSymbols = WaveformToSymbol(I_FilterOutput(SymbolRange),numSamplesPerSymbol);
Q_RxSymbols = WaveformToSymbol(Q_FilterOutput(SymbolRange),numSamplesPerSymbol);

%-------------------------------------------------------------------------------%

% Converts a Waveform to a symbol stream.
function [SymbolStream] = WaveformToSymbol(Waveform, numSamplesPerSymbol)

SymbolStream = Waveform(1:numSamplesPerSymbol:length(Waveform));
```

6.6. **Coherent $\pi/4$-DQPSK Decoder.**

```
function [BitStreamOne,BitStreamTwo] = DQPSKDecoder(I_SymbolsRx,Q_SymbolsRx)

InitialSymbol = 1+j*0; % Corresponding to a Phase of Zero.
streamLength = length(I_SymbolsRx);

ModSymbols = (I_SymbolsRx + j*Q_SymbolsRx);

ModAngles = angle(ModSymbols);
DiffAngles = [ModAngles 0] - [0 ModAngles];
DiffAngles = DiffAngles(1:end-1);
BitStreamOne = sin(DiffAngles)<0;
BitStreamTwo = cos(DiffAngles)<0;
```

6.7. $\pi/4$**-QPSK Encoder/Decoder (without Differential Encoding/Decoding).** These modules are purely for calibrating the simulation setup.

6.7.1. $\pi/4$*-QPSK Encoder.*

```
function [I_Symbols, Q_Symbols] = QPSKEncoder(BitStreamOne,BitStreamTwo)

I_Symbols = 2*BitStreamOne-1;
Q_Symbols = 2*BitStreamTwo-1;
```

6.7.2. $\pi/4$*-QPSK Decoder.*

```
function [BitStreamOne, BitStreamTwo] = QPSKDecoder(I_Symbols, Q_Symbols)

BitStreamOne = I_Symbols > 0;
BitStreamTwo = Q_Symbols > 0;
```

7. MATLAB Source Code

7.1. **Case-I : AWGN Channel.**

```
clear;
numSamplesPerSymbol = 8;
BitRate = 16000;
BaudRate = BitRate/2;
SymbolDuration = 1/BaudRate;
SamplingFrequency = numSamplesPerSymbol*BaudRate;

nIters = 20;
LogEbNo = [0 1 2 3 4 5 6 7 8 9];
% LogEbNo = 0:0.5:6 -> for the comprehensive plot..

lenSim = length(LogEbNo);

% <-- Complete the filter construction part first -->

% Uncomment the following portion of the code to implement
% Sqrt Raised Cosine Filtering...

% Nf = 61;
% hSqRCFilter = SqRCFilter(Nf,0.25,SymbolDuration,SamplingFrequency);
% Nt = 31;
% nStart = (Nf-1)/2 - (Nt-1)/2 + 1;
% nEnd   = (Nf-1)/2 + (Nt-1)/2 + 1;
% hTransmitFilter = hSqRCFilter(nStart:nEnd);
% hTransmitFilter = hTransmitFilter/max(hTransmitFilter);
% hReceiveFilter = hTransmitFilter;

% Symbol Repetition - comment out this part for using RC filtering.
hTransmitFilter = ones(1,numSamplesPerSymbol);
hReceiveFilter = hTransmitFilter;
% <-- System test begins -->

for EbNoIndex = 1:lenSim
   for iters = 1:nIters

      % The Transmitter.
      BitStreamLength = 10000;
      BitStream = rand(1,BitStreamLength)>0.5;
      [BitStreamOne,BitStreamTwo] = SerialToParallel(BitStream);
      [I_SymbolsTx,Q_SymbolsTx] = DQPSKEncoder(BitStreamOne,BitStreamTwo);
      [I_WaveformTx,Q_WaveformTx] = TransmitFilter(I_SymbolsTx,Q_SymbolsTx, \\
                                          hTransmitFilter,numSamplesPerSymbol);
```

```
      % Power Calculations.
      EbNo = 10^(LogEbNo(EbNoIndex)/10);
      Eb1 = sum(I_WaveformTx.*I_WaveformTx)/(BitStreamLength);
      Eb2 = sum(Q_WaveformTx.*Q_WaveformTx)/(BitStreamLength);
      No = (Eb1+Eb2)/(EbNo);

      % The Receiver.
      [I_WaveformRx,Q_WaveformRx] = AWGNChannel(I_WaveformTx,Q_WaveformTx,No);
      [I_SymbolsRx,Q_SymbolsRx] = ReceiveFilter(I_WaveformRx,Q_WaveformRx, \\
                                          hReceiveFilter,numSamplesPerSymbol);
      [BitStreamOneRx,BitStreamTwoRx] = DQPSKDecoder(I_SymbolsRx,Q_SymbolsRx);
      BitStreamRx = ParallelToSerial(BitStreamOneRx,BitStreamTwoRx);
      Errors = sum(BitStream~=BitStreamRx);
      BER(iters) = Errors/BitStreamLength;
   end
   AvBER(EbNoIndex) = sum(BER)/nIters
end
```

7.2. **Case-II : LOS + Rayleigh Fading.**

```
clear;
numSamplesPerSymbol = 8;

nIters = 20;
LogFadeEbNo = 0:0.5:6;
lenSim = length(LogFadeEbNo);

hTransmitFilter = ones(1,numSamplesPerSymbol);
hReceiveFilter = hTransmitFilter;

% <-- System test begins -->

for EbNoIndex = 1:lenSim
   for iters = 1:nIters
      % The Transmitter first
      BitStreamLength = 5000;
      BitStream = randn(1,BitStreamLength)>0.5;
      [BitStreamOne,BitStreamTwo] = SerialToParallel(BitStream);
      [I_SymbolsTx,Q_SymbolsTx] = DQPSKEncoder(BitStreamOne,BitStreamTwo);
      [I_WaveformTx,Q_WaveformTx] = TransmitFilter(I_SymbolsTx,Q_SymbolsTx, \\
                                          hTransmitFilter,numSamplesPerSymbol);

      % Eb/No and Avg Fade Power Calculations.
      AvgFadePower = 10^(LogFadeEbNo(EbNoIndex)/10);
      Eb1 = sum(I_WaveformTx.*I_WaveformTx)/(BitStreamLength);
```

```
      Eb2 = sum(Q_WaveformTx.*Q_WaveformTx)/(BitStreamLength);
      Eb = Eb1 + Eb2;

      % Channel Power Gains..
      I_WaveformTx = I_WaveformTx/sqrt(2); Q_WaveformTx = Q_WaveformTx/sqrt(2);

      % The Ray #1 (Rayleigh Faded).
      %  Multipath-Fading...
      [I_FadedWaveform, Q_FadedWaveform] = RayleighFader(I_WaveformTx, \\
                                                  Q_WaveformTx,AvgFadePower);
      % AWGN Channel...
      EbNo = 1; No = Eb/EbNo;
      [I_WaveformRay1,Q_WaveformRay1] = AWGNChannel(I_FadedWaveform,  \\
                                                       Q_FadedWaveform,No);

      %The Ray #2 (Line of Sight).
      EbNo = 10^(LogFadeEbNo(EbNoIndex)/10); No = Eb/EbNo;
      [I_WaveformRay2,Q_WaveformRay2] = AWGNChannel(I_WaveformTx,Q_WaveformTx,No);

      % Diversity Combining.
      I_WaveformRx = I_WaveformRay1+I_WaveformRay2;
      Q_WaveformRx = Q_WaveformRay1+Q_WaveformRay2;

      % THe Receiver..
      [I_SymbolsRx, Q_SymbolsRx] = ReceiveFilter(I_WaveformRx,Q_WaveformRx, \\
                                          hReceiveFilter,numSamplesPerSymbol);
      [BitStreamOneRx,BitStreamTwoRx] = DQPSKDecoder(I_SymbolsRx,Q_SymbolsRx);
      BitStreamRx = ParallelToSerial(BitStreamOneRx,BitStreamTwoRx);
      Errors = sum(BitStream~=BitStreamRx);
      BER(iters) = Errors/BitStreamLength;
   end
   AvBER(EbNoIndex) = sum(BER)/nIters
end
```

7.3. **Case-III : 2-ray Rayleigh (variable delays).**

```
clear;
numSamplesPerSymbol = 10;
delay = 0; %change this param for varying delays.

nIters = 20;
LogFadeEbNo = 0:0.5:6;
lenSim = length(LogFadeEbNo);

hTransmitFilter = ones(1,numSamplesPerSymbol);
hReceiveFilter = hTransmitFilter;
```

```
% <-- System test begins -->

for EbNoIndex = 1:lenSim
   for iters = 1:nIters
      % The Transmitter first
      BitStreamLength = 3000;
      BitStream = rand(1,BitStreamLength)>0.5;
      [BitStreamOne,BitStreamTwo] = SerialToParallel(BitStream);
      [I_SymbolsTx,Q_SymbolsTx] = DQPSKEncoder(BitStreamOne,BitStreamTwo);
      [I_WaveformTx,Q_WaveformTx] = TransmitFilter(I_SymbolsTx,Q_SymbolsTx, \\
                                          hTransmitFilter,numSamplesPerSymbol);

      % Eb/No and Avg Fade Power Calculations.
      AvgFadePower = 1;
      EbNo = 10^(LogFadeEbNo(EbNoIndex)/10);
      Eb1 = sum(I_WaveformTx.*I_WaveformTx)/(BitStreamLength);
      Eb2 = sum(Q_WaveformTx.*Q_WaveformTx)/(BitStreamLength);
      Eb = Eb1 + Eb2;

      % Channel Power Gains..
      I_WaveformTx = I_WaveformTx/sqrt(2); Q_WaveformTx = Q_WaveformTx/sqrt(2);

      % The Ray #1 (Rayleigh Faded).
      %  Multipath-Fading...
      [I_FadedWaveform1, Q_FadedWaveform1] = RayleighFader(I_WaveformTx,  \\
                                                  Q_WaveformTx,AvgFadePower);
      % AWGN Channel...
      No = Eb/EbNo;
      [I_WaveformRay1,Q_WaveformRay1] = AWGNChannel(I_FadedWaveform1,     \\
                                                    Q_FadedWaveform1,No);

      %The Ray #2 (Rayleigh Faded).
      %  Multipath-Fading... (independent of the other ray).
      [I_FadedWaveform2, Q_FadedWaveform2] = RayleighFader(I_WaveformTx, \\
                                                   Q_WaveformTx,AvgFadePower);
      % AWGN Channel...
      No = Eb/EbNo;
      [I_WaveformRay2,Q_WaveformRay2] = AWGNChannel(I_FadedWaveform2,   \\
                                                    Q_FadedWaveform2,No);

      % Diversity Combining (Delay is introduced here).
      I_WaveformRx = I_WaveformRay1+[zeros(1,delay) I_WaveformRay2(1:end-delay)];
      Q_WaveformRx = Q_WaveformRay1+[zeros(1,delay) Q_WaveformRay2(1:end-delay)];
```

```
      % THe Receiver..
      [I_SymbolsRx, Q_SymbolsRx] = ReceiveFilter(I_WaveformRx,Q_WaveformRx, \\
                                                 hReceiveFilter,numSamplesPerSymbol);
      [BitStreamOneRx,BitStreamTwoRx] = DQPSKDecoder(I_SymbolsRx,Q_SymbolsRx);
      BitStreamRx = ParallelToSerial(BitStreamOneRx,BitStreamTwoRx);
      Errors = sum(BitStream~=BitStreamRx);
      BER(iters) = Errors/BitStreamLength;
   end
   AvBER(EbNoIndex) = sum(BER)/nIters
end
```

8. Appendix A - Raised Cosine Filtering

As explained earlier, Raised Cosine Filtering can be easily included in the simulation once the *filter coefficients* are known. (These coefficients must be passed as parameters to the "TransmitFilter(...)" and the "ReceiveFilter(...)" functions). This section explains the method used to generate the SQRC filter coefficients.

8.1. **Description.** The Raised Cosine filter is defined using the following frequency domain transfer function.

$$H_{rc}(f) = \begin{cases} T, & 0 \le |f| \le \frac{(1-\alpha)}{2T} \\ \frac{T}{2}\left[1 + \cos\frac{\pi T}{\alpha}\left(|f| - \frac{(1-\alpha)}{2T}\right)\right], & \frac{(1+\alpha)}{2T} \le |f| \le \frac{(1-\alpha)}{2T} \\ 0, & |f| \ge \frac{(1-\alpha)}{2T} \end{cases} \tag{18}$$

Where, the parameter α denotes the *roll-off factor* of the RC filter and T is the symbol rate. The roll-off factor is an important paramter that determines the *bandwidth* of th pulse waveform and the time-sidelobe levels in adjacent symbol slots. A filter defined using the above transfer function produces zero-ISI over a low-pass channel. Generally, in Digital

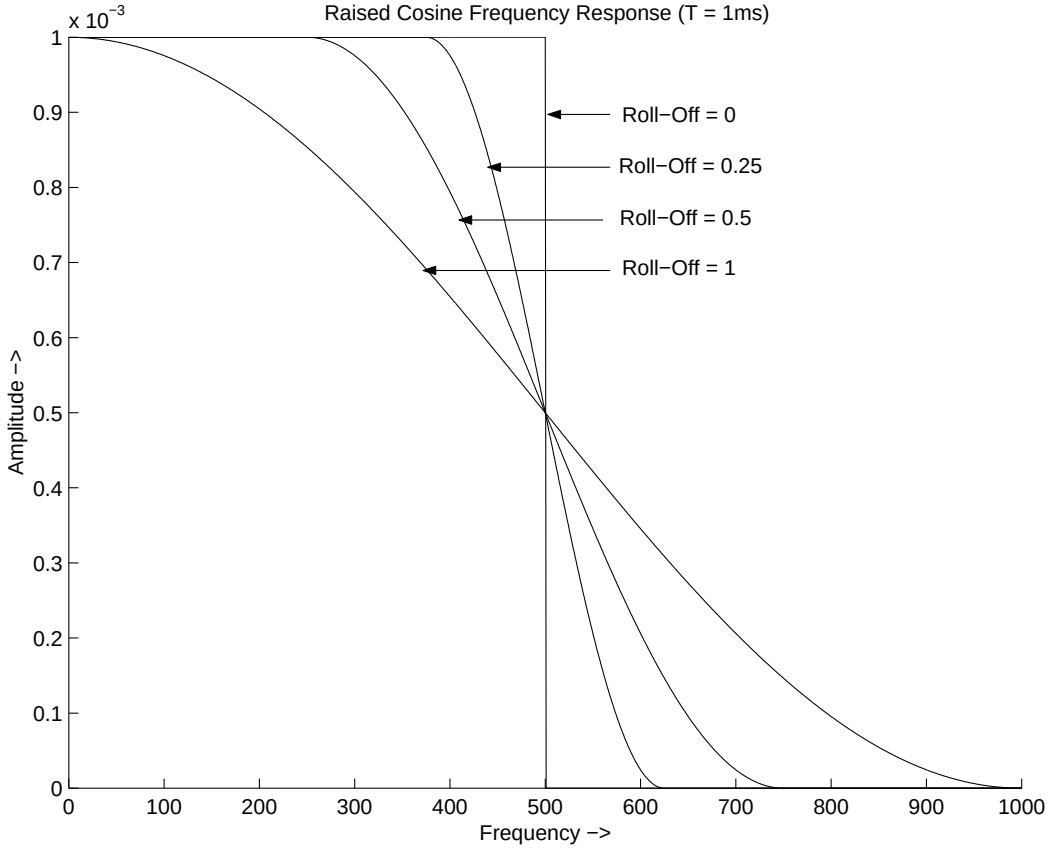

Figure 9. Raised Cosine Frequency Response for T = 1ms

Communication Systems, the trasmit and the receive filters are jointly designed to produce zero-ISI. For example, if $H_T(f)$ is the transmit filter and $H_R(f)$ is the receive filter, then the product (which is indeed, a casade of these two filters) $H_T(f)H_R(f)$ is designed to yeild zero-ISI. If we design $H_T(f)$ and $H_R(f)$ as :

$$H_T(f)H_R(f) = H_{rc}(f) \tag{19}$$

then the combined effect if these two filters effectively produces zero ISI.

8.2. **Design.** The simplest way to design $H_T(f)$ and $H_R(f)$, for simulation purposes, is to use linear-phase (FIR) digital filters with a magnitude repsonse given by,

$$|H_T(f)| = |H_R(f)| = \sqrt{H_{rc}(f)} \tag{20}$$

The most commonly used design methodology is the *Frequency Sampling design*, where the frequency response of the filter is sampled at constant intervals and an inverse fourier transform (IDFT) is applied to the frequency samples to obtain the filter coeffcients. More the number of frequency samples taken, more will the actual response match the desired response.

Design Specifications. There are 3 fundemental parameters that one must specify for the filter design : the response type (Raised Cosine or Square Root Raised Cosine), the Symbol/Bit duration (T_b) and the roll-off factor (α). In addition, for the frequency sampling design, there are 3 more additional parameters required : the sampling frequency (F_s), the Frequency Resolution factor ($N = Fs/\Delta f$) and the number of filter taps (N_t) . The explanation of these parameters are given below:

- Type of Response - This can be either Raised Cosine (RC) or Square Root Raised Cosine (SQRC).
- Symbol/Bit duration (T_b)- This is a fundemental parameter that determines the frequency response of the filter. The designed (RC) filter will have zeros at $t = nT_b$
- Roll-Off factor (α) - This determines the sharpness of the frequency response.
- Sampling Frequency (F_s) - Required for the digital implementation. The Sampling frequency should be atleast $2/T_b$.
- Frequency Resolution factor (N) - This determines the number of samples of the frequency response that are used in the design. Higher frequency resolution leads to a response that matches the desired response with more accuracy. Generally, N will be specified as an odd integer.
- Number of Filter Taps (N_t) - If more filter taps are used in the FIR filter, a more accurate response is obtained. Generally, N_t will be specified as an odd integer.

An example design specification can be like ($Type = RC, T_b = 1ms, \alpha = 0.25, F_s = 4kHz (= 4/T_b), N = 61, N_t = 31$).

Design Procedure. The design procedure can be split into the following steps :

- **Step 1:** From the given frequency response $H(f)$ (this can either be the RC or the SQRC Response), form the frequency samples at uniform intervals of $\Delta f = F_s/N$, for the first $(N-1)/2$ samples.

$$H_d(k) = H(kF_s/N), \qquad k = 0, 1, \ldots, (N-1)/2. \tag{21}$$

- **Step 2:** For a real-valued impulse response, the frequency response must obey the following symmetry condition :

$$H_d(k) = H_d(N-k), \qquad k = 0, 1, \ldots, N-1. \tag{22}$$

The remaining $(N-1)/2$ samples (from $(N+1)/2$ to $N-1$), are filled using the above symmetry condition.

- **Step 3:** Once the frequency samples are formed, the inverse discrete fourier transform (IDFT) of the samples is taken as follows :

$$h_d(n) = \sum_{i=0}^{N-1} H_d(k) e^{j(\frac{2*\pi}{N})kn}, \quad n = \frac{-(N_t-1)}{2}, \ldots, \frac{(N_t-1)}{2}; \tag{23}$$

Using the symmetry of the frequency response, the above equation can be rewritten as,

$$h_d(n) = H_d(0) + \sum_{i=0}^{\frac{N-1}{2}} H_d(k) \cos(\frac{2*\pi}{N}kn), \quad n = \frac{-(N_t-1)}{2}, \ldots, \frac{(N_t-1)}{2}; \tag{24}$$

Thus, using the above equation, we compute the impulse response values $h_d(-(N_t-1)/2), \ldots, h_d((N_t-1)/2)$.

- **Step 4:** The final step is to convert the computed (non-causal) impulse response into a realizable causal reponse by shifting the impulse response by $(N_t-1)/2$ samples, i.e, obtaining $g_d(n)$ as :

$$g_d(n) = h_d(n - (N_t-1)/2) \tag{25}$$

Now, $g_d(n)$ exists from $0, 1, \ldots, (N_t-1)$ and is causal and realizable.

The filter taps $g_d(n)$ for the above mentioned example specification is shown in fig(10). The response does not have zeros at the symbol intervals, because it is a SQRC filter and not a RC filter. A cascade to two SQRC filters does yeild a zero ISI.

8.3. **MATLAB Code.**

```
function [h] = SqRCFilter(N,Alpha,Tb,Fs)

H(1) = sqrt(RaisedCosineResponse(0,Alpha,Tb));
for k = 1:(N-1)/2,
   H(k+1) = sqrt(RaisedCosineResponse(k*Fs/N,Alpha,Tb));
   H(N-k+1) = H(k+1);
end
for n = -(N-1)/2:(N-1)/2
   h(n+((N-1)/2)+1) = H(0+1);
   for m = 1:(N-1)/2,
      h(n+((N-1)/2)+1) = h(n+((N-1)/2)+1) + 2*H(m+1)*cos(2*pi*m*n/N);
   end
end
```

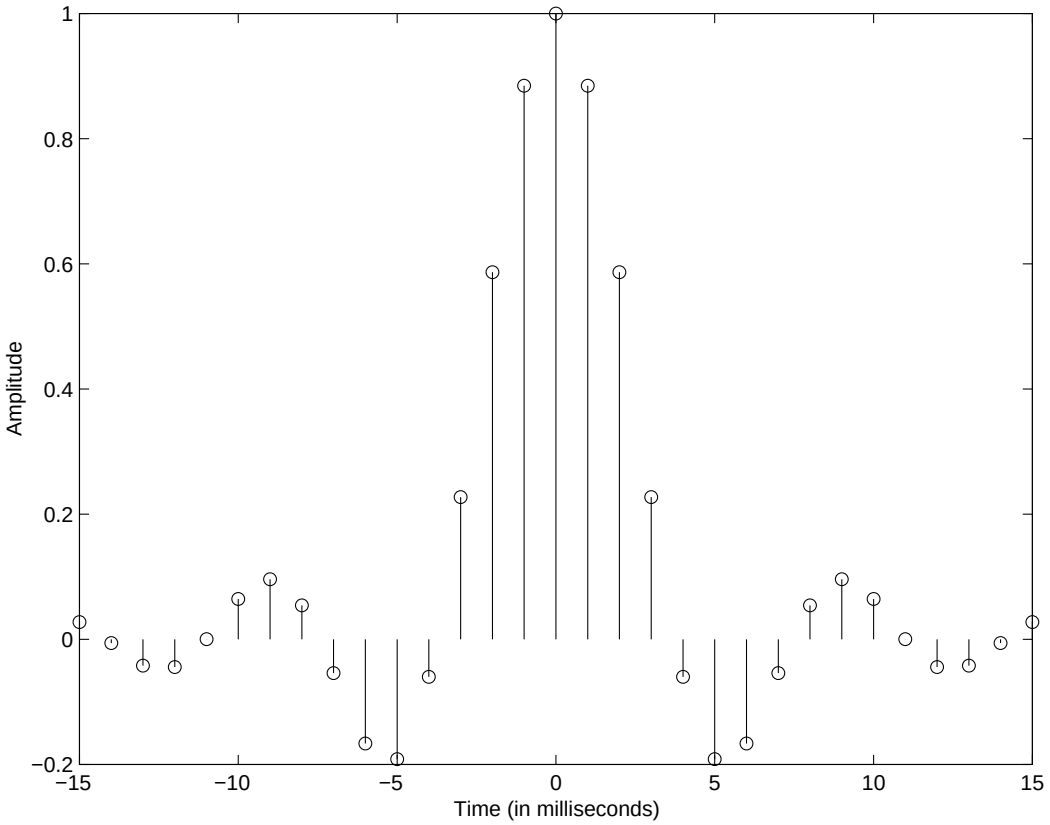

FIGURE 10. An Example Impulse Response

9. Appendix B - Rayleigh Fading Generation (Clarke/Gans Model)

During the course of the project, a rayleigh fading generator using the Clarke/Gans Model was developed using the method described in Rappaport's book on Wireless Communication. This section presents the code and an example result for USDC mobile transeiver travelling at 100 miles/hr.

9.1. **Result.** The Rayleigh Fading envelope for a USDC mobile receiver travelling at 100 miles/hr is plotted in fig(11).

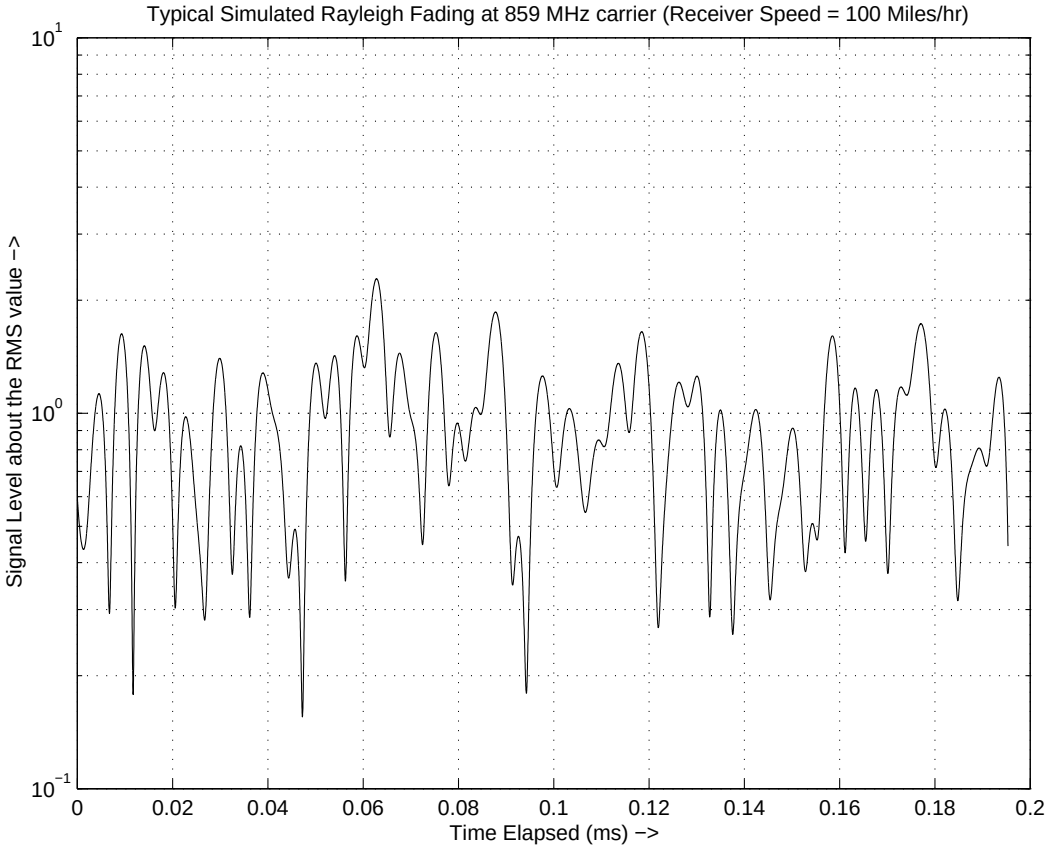

Figure 11. Typical Simulated Rayleigh Fading at 859 MHz carrier (Receiver Speed = 100 Miles/hr)

9.2. **MATLAB Code.**

```
function [rayEnvelope] = RayleighEnvelope(MobileSpeed, CarrierFrequency,  \\
                                                    SamplingFrequency, NumSamples)

% Step #1.
% ---------
% Calculate the doppler shift.
```

```
% 1. Calculate the Wavelength from the carrier frequency.
% 2. Calulate the Maximum Doppler Shift 'fd', using fd = V/Lambda.
% Convert the given miles/hr to meters/s first.
waveLength = 3e+08/CarrierFrequency;
dopplerFrequency = 1.609*MobileSpeed*1000/(3600*waveLength);

% Step #2
% --------
% Two FFT's are required for generating the spectra of the gaussian random process.
% Make them simple by taking a FFT with some 2^n points.
nActualPoints = ((2*dopplerFrequency)/SamplingFrequency)*NumSamples;
Nfft = 8; % Start with 8, always better.
while (Nfft < nActualPoints),
   Nfft = Nfft*2;
end
Nfft

% The Number of samples for the IFFT will be more than the number of samples for
% the FFT by factor of (fs/2*fd). This will smoothen the fading envelope based on
% how much the sampling frequency is larger than 2*fd. A large sampling frequency
% must be choosen to resolve even the deepest of fades properly.

Nifft = ceil(Nfft*(SamplingFrequency/(2*dopplerFrequency)))

% Step #3.
% ---------
% Generate the complex gaussian random process. The pair of Gaussian random
% processes are stationary inependent consisting of identically distributed
% random variables.
I_Gaussian = randn(1,Nfft);
Q_Gaussian = randn(1,Nfft);

% Take FFT's of these Gaussian Random Variables.
I_Gaussian_FFT = fft(I_Gaussian);
Q_Gaussian_FFT = fft(Q_Gaussian);

% Step #4.
% ---------
% The most difficult part. Generating the Doppler filters frequency response.
% Since the doppler filter has INFINITE response at f = fd, the filter response
% at fd can be obtained using polynomial fitting of the filter response curve.

deltaF = 2*dopplerFrequency/Nfft; % Frequency Spacing.

% The DC component first.
```

```
dopplerFilter(1) = 1.5/(pi*dopplerFrequency);
frequencyIndex(1) = 0;

% The other components for ONE side the spectrum.
% Store the frequency indices for polynomial fitting.
for index = 2:Nfft/2,
   frequencyIndex(index) = (index-1)*deltaF;
   dopplerFilter(index) = 1.5/(pi*dopplerFrequency*sqrt(1-     \\
                                  (frequencyIndex(index)/dopplerFrequency)^2));
   dopplerFilter(Nfft-index+2) = dopplerFilter(index);
end

% polynomial fitting using the last 3 frequency samples.
nFitPoints = 3 % good enough.
polyFreq = polyfit(frequencyIndex((Nfft/2)-nFitPoints:(Nfft/2)),  \\
                        dopplerFilter((Nfft/2)-nFitPoints:(Nfft/2)),nFitPoints);
dopplerFilter((Nfft/2)+1) = polyval(polyFreq,frequencyIndex(Nfft/2)+deltaF);

% Step #5.
% -------
% Do the filtering of the gaussian random variables here.
I_Filtered_Gaussian = I_Gaussian_FFT.*sqrt(dopplerFilter);
Q_Filtered_Gaussian = Q_Gaussian_FFT.*sqrt(dopplerFilter);

% Take the IFFT.
% First to smoothen the points out, ZERO-PAD the frequency response
% to Nifft points (as explained before).
I_Freq_Points = [I_Filtered_Gaussian(1:Nfft/2) zeros(1,Nifft-Nfft)  \\
                                          I_Filtered_Gaussian(Nfft/2+1:Nfft)];
I_Time_Response = ifft(I_Freq_Points);
Q_Freq_Points = [Q_Filtered_Gaussian(1:Nfft/2) zeros(1,Nifft-Nfft)  \\
                                          Q_Filtered_Gaussian(Nfft/2+1:Nfft)];
Q_Time_Response = ifft(Q_Freq_Points);

% Step #6 - Finishing off
% -----------------------
% Take the magnitude squared of the I and Q components and add them together.
rayEnvelope = sqrt(((abs(I_Time_Response)).^2) + ((abs(Q_Time_Response)).^2));
%for i = 1:Nifft
%   rayEnvelope(i) = sqrt(abs(I_Time_Response(i))^2 + abs(Q_Time_Response(i))^2);
%end

% Compute the Root Mean Squared Value and Normalize the envelope.
rayRMS = sqrt(mean(rayEnvelope(1:NumSamples).*rayEnvelope(1:NumSamples)));
rayEnvelope = rayEnvelope(1:NumSamples)/rayRMS;
```

EECS Department, The University of Kansas