

Wireless Communications

Final CA

Instructor: Dr. Sabbaghian

By: Narjes Noorzad

SudentT ID: 810196626

Contents

Introduction.....	4
Transmitter.....	5
Input data	5
QPSK modulation	5
Frame Divider.....	5
TX Serial to Parallel	7
DPSK modulation.....	7
IFFT bins allocation	8
IFFT	9
CP addition.....	9
TX Parallel to serial.....	9
TX Cascade Frames	9
Power Amplifier * (clipping effect).....	10
Chaining TX modules to each other.....	10
Channel	11
AWGN	11
Receiver.....	12
Frames Detection.....	12
RX Serial to Parallel.....	12
CP removal	12
FFT	13
Extract Carrier from FFT bins.....	13

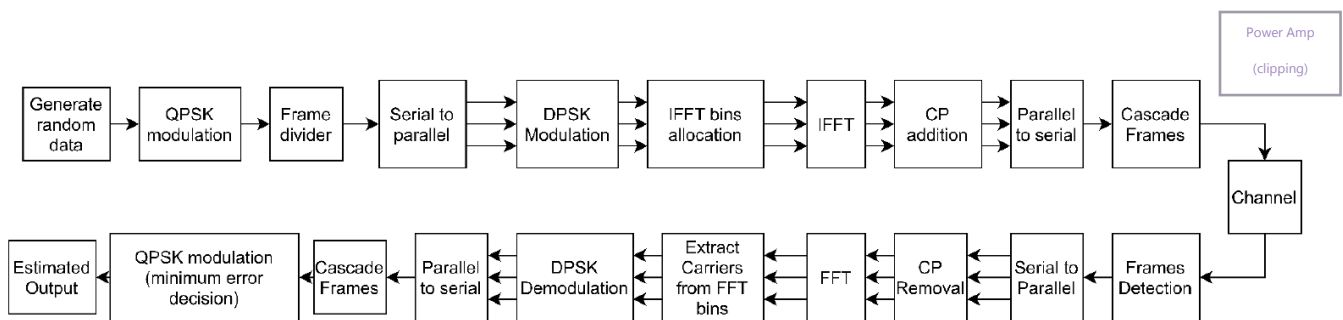
DPSK Demodulation.....	13
RX Parallel to Serial.....	14
RX Cascade Frames.....	14
QPSK Demodulation.....	15
Evaluating Estimated Output	15
Chaining channel and receiver to the system.....	15
Results.....	16
A) Results for SNR = 40dB, clipping = 3dB, IFFT length = 1024, carrier num = 400	16
B) BER & SER over SNR (dB) (clipping = 3dB, IFFT length = 1024, carrier num = 400).....	16
C) BER & SER over Clipping Level (dB) (Setting SNR to medium level of 40dB)	17
D) BER & SER over SNR (dB) for Rayleigh Fading (Without MMSE equalizer)	17
E) BER & SER over SNR (dB) for Rayleigh Fading with MMSE Equalizer	18
F) Header Detection and Synchronization (Bonus).....	19

Introduction

This report will cover the implementation of an OFDM system.

Each module will be comprehensively explained throughout this report along with its MATLAB implementation code.

According to the block diagram in the instructions, ten difference modules should be implemented both in the transmitter and receiver side. In addition to every module in the transmitter, in the receiver side, there is also an estimator module in order to detect the symbols effected by unwanted noise, fading, etc.



Transmitter

Input data

Input message should be generated using rand.

In order to do this, a randomly generated bitstream is reshaped into a matrix with $\log_2^M = 2$ columns so that each row represents a symbol.

```
total_bits = 10^7;  
M = 4;  
total_symbols = total_bits/log2(M);
```

```
rand bit = randi([0 1], total_symbols, log2(M));
```

QPSK modulation

To modulate the symbols, QPSK modulation is employed.

According to QPSK modulation constellation with an initial angle of $\alpha = 0^\circ$, signal space points are defined as follows:

$$s_1 = (1, 0), \quad s_2 = (0, j), \quad s_3 = (-1, 0), \quad s_4 = (0, -j)$$

To map symbols to the constellation points, gray coding is applied to the decimal equivalent of each symbol. This is done to minimize the detection error.

```
function [s]=QPSK(d)  
    s = bi2de(d,2);  
    s = changem(s, [0 1 3 2], [0 1 2 3]);%Gray coding  
end
```

The total number of symbols is calculated using a simple division:

$$\text{total symbols} = \frac{\text{total bits}}{\log_2^M} = \frac{10^7}{2} = 5 \times 10^6 \text{ (sym)}$$

Frame Divider

An OFDM message is separated into multiple "frames" in practical systems. Frames are transmitted over the channel with a guard time interval, including cyclic prefixes which might be all zero.

In this assignment, however, all the frames are going to be transmitted at once. Two headers which are eight times as long as data frames (without CP), are appended to the sequence of data frames at the beginning and the end.

To implement a frame divider, we receive the symbol stream from the previous module and separate symbols into groups of "frame_size."

The frame length is calculated according to the total number of symbols, "sfc" (symbol / (frame* carrier)) rate, and the number of carriers (n_c):

$$frame\ length = sfc \times n_c$$

Substituting values, we obtain:

$$sfc = \text{ceil} \left(\frac{2^{13}}{n_c} \right) = \text{ceil} \left(\frac{2^{13}}{400} \right) = 21$$

$$frame\ length = 21 \times 400 = 8400\ (sym)$$

By dividing the total number of symbols, we obtain the number of frames needed:

$$data\ frames = \text{ceil} \left(\frac{total\ symbols}{frame\ length} \right) = \text{ceil} \left(\frac{5 \times 10^6}{8400} \right) = 596$$

There are two steps to follow for dividing the symbol stream into frames :

1. The symbol stream is separated into groups of 8400. (Zeros are padded to the last group if necessary - to form the 596'th frame.)
2. Next, we reshape the bitstream into a matrix, such that each row of that matrix represents a frame. So, the symbol stream is now a matrix with 596 rows, each corresponding to a frame.

```
function [output] = Frame_Divider(input, nc)
    sfc = ceil(2^13/nc);
    sf = sfc*nc;
    reminder = sf - mod(length(input), sf);
    extended_signal = cat(2, input, zeros(1, reminder));
    output = reshape(extended_signal, sf, length(extended_signal)/sf);
end
```

TX Serial to Parallel

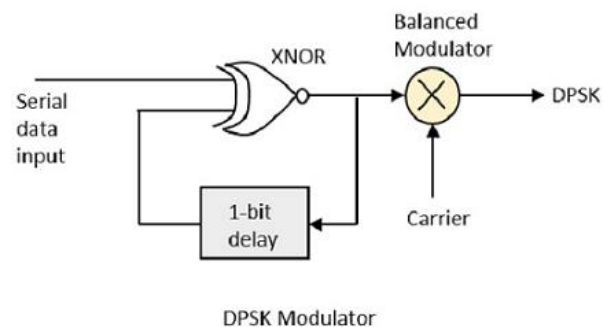
This module converts each frame from a row to a matrix. The total message will now appear as a tensor in which each frame is a layer of that tensor in form of a matrix.

```
function [out] = TX_Serial_to_parallel(in, n_c)
    sfc = ceil(2^13/n_c);
    frame_numbers = size(in);
    frame_numbers = frame_numbers(2);
    out = reshape(in, sfc, n_c, frame_numbers);
end
```

DPSK modulation

DPSK modulation is based on a recursive behavior modeled as a feedback loop. Therefore, it needs an initialization value.

A reference row is inserted at the beginning of each frame (resulting in an appended layer on the top of the data tensor). This row is filled with random symbols.



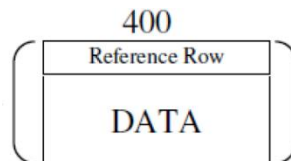
The figure above, shows the implementation of a DPSK modulation block in terms of common blocks we use.

```

function [out] = DPSK_modulation(in)
    shape = size(in);
    random_first_row = randi([0 3], shape(2), shape(3));
    extended_in_signal = zeros(shape(1)+1, shape(2), shape(3));
    extended_in_signal(1, :, :) = random_first_row;
    extended_in_signal(2:(shape(1)+1), :, :) = in;
    out = zeros(shape(1)+1, shape(2), shape(3));
    for row = 2:(shape(1)+1)
        out(row, :, :) = mod(extended_in_signal(row, :, :) + extended_in_signal(row-1, :, :), 4);
    end
    out(1, :, :) = extended_in_signal(1, :, :);
    out = changem(out, [1 -1 1 -1], [0 2 1 3]);
end

```

The output of the DPSK modulation should be a complex signal (symbol stream) so a mapping from decimal gray coded values to complex points, corresponding to the modulation is added.

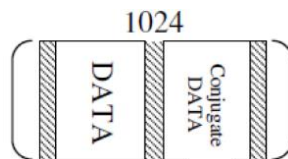


IFFT bins allocation

Input of the IFFT module should be the same length as the FFT module with 1024 points.

Matrix at our hand is a 400-column matrix which should be transferred to 1024 column matrix to be fit for the IFFT module. Thus, a conjugate matrix is added to it and also add some zero columns in between.

$$\#added\ columns = 1024 - 2 \times 400 = 224$$



```

function [out] = IFFT_bins_allocation(in)
    conj_in_signal = conj(in);
    shape = size(in);
    out = zeros(shape(1), 1024, shape(3));
    out(:, (76:76+shape(2)-1), :) = in;
    out(:, (550:550+shape(2)-1), :) = conj_in_signal;
end

```


IFFT

The IFFT module simply takes the input frames one by one and computes a row-wise IFFT (because a row contains part of the data frame spread among subcarriers)

Due to the complex symmetry of the frequency-domain signal the IFFT output is expected to be real.

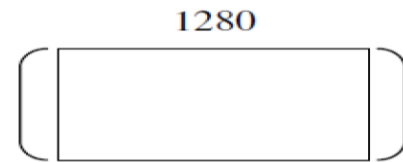
```
function [out] = IFFT(in, IFFT_len)
    out = ifft(in, IFFT_len, 2);
```

CP addition

To prevent interference, a frame guard should be added to the beginning of each frame.

Filling these columns with cyclic prefix ensures circular convolution and frequency-domain multiplication as a pre-requisite to analyze channel distortions and implement further techniques such as equalization.

```
function [out] = CP_addition(in)
    shape = size(in);
    out = zeros(shape(1), 1280, shape(3));
    out(:, (257:1280), :) = in;
    out(:, (1:256), :) = in(:, (769:1024), :);
end
```



TX Parallel to serial

Eventually the frames are transmitted in a flattened shape over the channel, so the data should be converted a matrix .

```
function [out] = TX_parallel_to_serial(in)
    shape = size(in);
    out = reshape(in, shape(1)*shape(2), shape(3));
end
```

TX Cascade Frames

Matrix should now be converted to a vector. This vector contains frames cascaded to another (rows put beside one another) and also 2 headers, each 8 times larger than a single frame (without CP).

```

function [out] = TX_cascade_frames(in, n_c)
    shape = size(in);
    sfc = ceil(2^13/n_c);
    header = zeros(1, 8*1024*(sfc+1));
    out = reshape(in, 1, shape(1)*shape(2));
    out = cat(2, header, out);
    out = cat(2, out, header);
end

```

Power Amplifier * (clipping effect)

This module is added to model the clipping effect. The clipping effect caused by the non-linearity of the power amplifier (PA), is simulated by finding the highest peak in the signal and decreasing it according to the clipping level. All the other values remaining higher than the clipped level will also be lowered to this level.

```

function [out] = clipping(in,clipping_db)
    clipping = 10^(clipping_db/10);
    max_value = max(in);
    in(in>max_value/clipping)= max_value/clipping;
    out = in;
end

```

Chaining TX modules to each other

Transmitter modules are chained together in another function called OFDM, followed by the channel and receiver modules which are discussed afterwards.

```

function [ber, ser] = OFDM(in, n_c, FFT_len, SNR_db, clp_lvl)

    num_carriers = n_c;
    IFFT_len = FFT_len;


    % Transmitter Modules:
    out_QPSK = QPSK(in);
    framed = Frame_Divider(out_QPSK', num_carriers);
    out_tx_s2p = TX_Serial_to_parallel(framed, num_carriers);
    dp_out = DPSK_modulation(out_tx_s2p);
    out_IFFT_bins = IFFT_bins_allocation(dp_out);
    out_IFFT = IFFT(out_IFFT_bins, IFFT_len);
    out_CP = CP_addition(out_IFFT);
    out_tx_p2s = TX_parallel_to_serial(out_CP);
    out_tx_cas = TX_cascade_frames(out_tx_p2s,num_carriers);
    out_tx_cas = clipping_1(out_tx_cas, clp_lvl);

```

Channel

AWGN

The channel is modeled as an additive white gaussian noise (AWGN) channel. The noise variance is automatically calculated according to the SNR value fed to the AWGN function.

```
 function [out] = channel(in, SNR)  
    out = awgn(in, SNR);  
end
```

Receiver

Modules implemented in the receiver almost do the same thing as their equivalent modules in the transmitter.

Received signal is distorted due to the noisy channel therefore an MSE detection unit should be implemented the first demodulation. (DPSK demodulation)

Frames Detection

Reversing what had been done, data vector should be converted into a matrix, each row corresponding to a frame + guard. So the vector should be reshaped to groups of $frame_length \times CP_ratio$ symbols.

Its important to remove the headers from the beginning and ending of the vector according to the calculated header length.

```
function out = RX_Frames_Detection(in, n_c)
    sfc = ceil(2^13/n_c);
    frame_len = (1024 * (sfc+1));
    header_len = 8 * frame_len;
    in = in(header_len+1:length(in)- header_len); % throwing the header away
    out = reshape(in, frame_len*(5/4), []); % separating frames by rows
end
```

RX Serial to Parallel

Each frame should be reconverted to a matrix, resulting in a data tensor as the output.

```
function [out] = RX_serial_to_parallel(in)
    shape = size(in);
    out = reshape(in, [], (1024*5/4), shape(2));
end
```

CP removal

The added CP columns to the data should be removed from the beggining.

```
function [out] = CP_Removal(in)
    shape = size(in);
    out = in(:, (shape(2)/5+1):shape(2), :);
end
```

FFT

Applying a row-wise Fast Fourier Transform on each frame returns our data back into frequency domain.

```
function [out] = FFT(in, FFT_len)
    out = fft(in, FFT_len, 2);
```

Extract Carrier from FFT bins

In order to store modulated data, 400 column data was employed before mixing with zeros to be fit for the IFFT module. Extracting the exact data is a must at the receiver side.

Obviously, the values will be affected by the AWGN noise the signal has experienced through the channel.

```
function [out] = Extract_Carriers(in, n_c)
    out = in(:, 76:76+n_c-1, :);
end
```

DPSK Demodulation

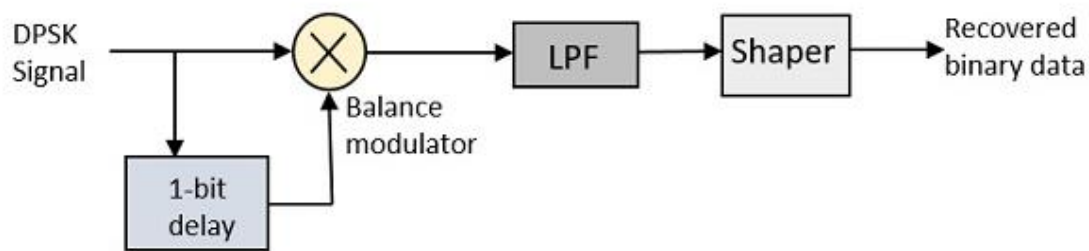
Next, complex signals should be demodulated into decimal values mod 4.

To inverse the DPSK modulation, consecutive rows are subtracted from one another.

The presence of the reference row solves the problem of initialization.

```
function [out] = DPSK_Demodulation(in)
    Re_in_signal = real(in);
    Im_in_signal = imag(in);
    shape = size(in);
    dist = zeros(shape(1), shape(2), shape(3), 4);
    dist(:, :, :, 1) = (Re_in_signal-1).^2 + Im_in_signal.^2;
    dist(:, :, :, 2) = (Re_in_signal).^2 + (Im_in_signal-1).^2;
    dist(:, :, :, 3) = (Re_in_signal+1).^2 + Im_in_signal.^2;
    dist(:, :, :, 4) = (Re_in_signal).^2 + (Im_in_signal+1).^2;
    [~, detected] = min(dist, [], 4);
    detected = detected - 1;
    for row = 2:shape(1)
        detected(row, :, :) = mod(detected((row), :, :) - detected((row-1), :, :), 4);
    end
    out = detected((2:shape(1)), :, :);
end
```

But first, each value should be mapped to the closest signal space points on the DPSK modulation in order to ensure MSE error criteria. To do this, the distance from all 4 points of the constellation is computed and the point with the minimum distance is selected.



The DPSK module is implemented via a feed forward loop in digital communication, followed by a LPF and a shaper which is not implemented.

RX Parallel to Serial

The data should be reconverted to a single data matrix, each row representing a frame.

```

function [out] = RX_parallel_to_serial(in)
    shape = size(in);
    out = reshape(in, shape(1)*shape(2), shape(3));
end

```

RX Cascade Frames

The data matrix is reconverted into the data vector, cascading frames which each are located in different rows beside one another and ready for demodulation, comparison and evaluation.

```

function [out] = RX_cascade_frames(in)
    shape = size(in);
    out = reshape(in, shape(1)*shape(2), 1);
end

```

QPSK Demodulation

After each data point (symbol) is mapped to the closest valid symbol, and the data is converted back to a vector, it is demodulated by converting the gray coded values back to sequences of 2 grouped bits. The output should be a bitstream which can now easily be compared with the initial bit stream generated in the transmitter.

```
function [predicted_bit, predicted_symbol] = QPSK_Demodulation(in)
    predicted_symbol = changem(in, [0 1 2 3], [0 1 3 2]); %Gray coding
    predicted_bit = de2bi(predicted_symbol);
end
```

Evaluating Estimated Output

By comparing the mentioned bit stream and symbol streams (estimated and true), BER and SER are obtained.

```
function [ber, ser] = Estimation_output(predicted_bit, labels, predicted_symbols, symbols)
    shape = size(labels);
    ber = sum(sum(predicted_bit(1:shape(1),:) ~= labels))/(shape(1)*shape(2));
    ser = sum(predicted_symbols(1:shape(1)) ~= symbols)/shape(1);
end
```

Chaining channel and receiver to the system

The channel module and receiver modules are chained in the OFDM function and they follow the chain of transmitter modules, completing the OFDM system. This function can be called every time an OFDM message is wished to be transferred over the channel.

```
% Receiver Modules:
out_rx_frddet = RX_Frames_Detection(out_channel, num_carriers);
out_rx_s2p = RX_serial_to_parallel(out_rx_frddet);
out_CPrem = CP_Removal(out_rx_s2p);
out_FFT = FFT(out_CPrem, FFT_len);
out_ExCar = Extract_Carriers(out_FFT, num_carriers);
dp_demod_out = DPSK_Demodulation(out_ExCar);
out_rx_p2s = RX_parallel_to_serial(dp_demod_out);
out_rx_cas = RX_cascade_frames(out_rx_p2s);
out_QPSK_demod = QPSK_Demodulation(out_rx_cas);
[ber, ser] = Estimation_output(out_QPSK_demod, in, out_rx_cas, out_QPSK);
```

Results

A) Results for SNR = 40dB, clipping = 3dB, IFFT length = 1024, carrier num = 400

- **Number of Frames:** We are transmitting a total number of **596** data frames over the channel. If we include headers as frames as well, we are actually transmitting a total number of 612 frames. To show this in practice, we can add an output to the OFDM function which is passed back to the main code and stored in our workspace:

```
function [ber, ser, framed] = OFDM(in, n_c, FFT_len, SNR_db, clp_lvl)
[ber_1, ser_1, frame] = OFDM(rand_bit, 400, 1024, 20, 3);
frame size = size(frame);
```

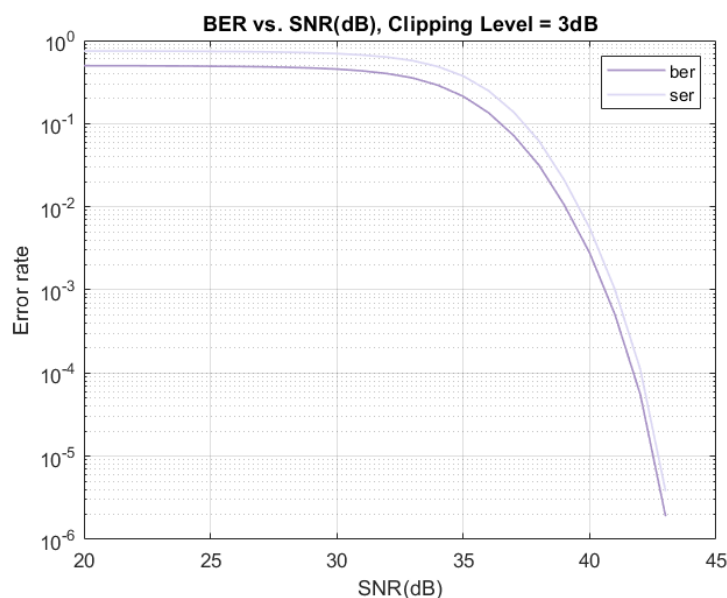
 **frame** 8400x596 double

- **BER:** The BER for the given parameter is 1% which is not awful, but not good either. Clipping may also have degraded our performance due to signal distortion and out of band radiation.

BER = 0.0105

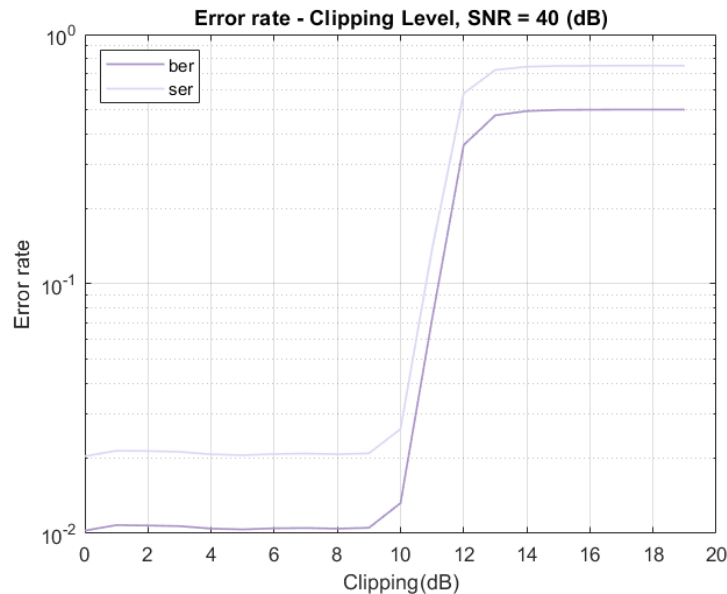
B) BER & SER over SNR (dB) (clipping = 3dB, IFFT length = 1024, carrier num = 400)

- As we can see in the figure below, both BER and SER decrease exponentially from a value around 50% to 0%, when we increase SNR from 20 to 50 dB with 1dB Steps.



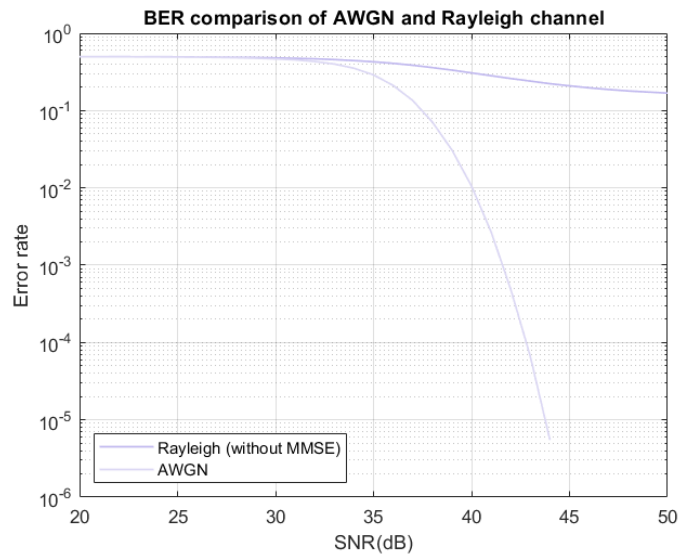
C) BER & SER over Clipping Level (dB) (Setting SNR to medium level of 40dB)

- As we expected with increasing the clipping value, the error rate will increase due to signal distortion and probable out of band radiation.



D) BER & SER over SNR (dB) for Rayleigh Fading (Without MMSE equalizer)

- Rayleigh fading degrades performance as a random variable is being multiplied in the channel gain which was 1 (flat) before this. We haven't predicted the channel fading with pilot signals, thus the BER, SER will be severely higher than AWGN channel. Reliable communication won't ever be provided as the variance of the Rayleigh R.V limits BER, SER rates.



$$f_H(h) = \frac{h}{\sigma^2} e^{\left(\frac{-h^2}{2\sigma^2}\right)}$$

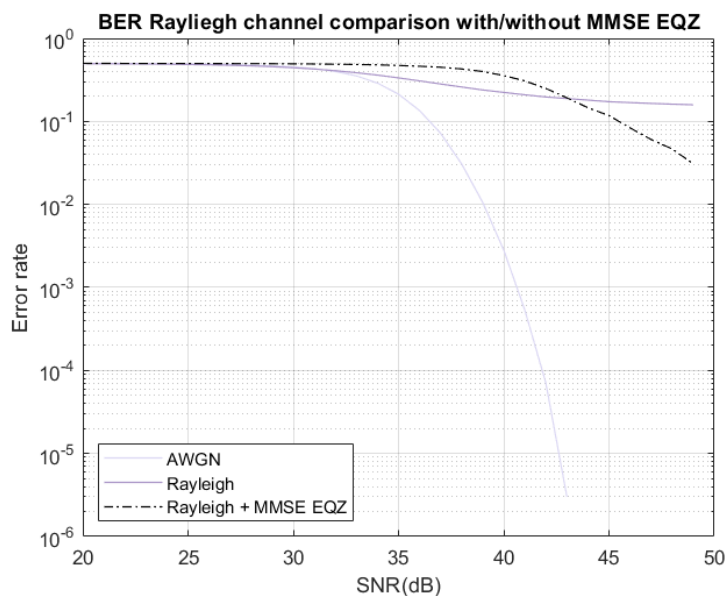
```
function [out, H_f] = rayleigh(in)
len = length(in);
ray_smp1 = raylrnd(1, 1, len);
H_f = ray_smp1;
out = ray_smp1.*in;
end
```

E) BER & SER over SNR (dB) for Rayleigh Fading with MMSE Equalizer

Assume we are estimating channel fading coefficients h_k , using pilot signals in a sufficiently small period of time grounding on the fact that the receiver knows exactly all the channel fading coefficients.

As we are using a Rayleigh with variance = 1, the channel fading that's multiplied in the signal before adding AWGN noise will deteriorate performance, and low SNR regions won't let the equalizer compensate channel's behavior.

In fact, MMSE equalizer's performance would be worse than the mode which we don't use an equalizer at all in these regions. But by increasing SNR, MMSE can decrease BER and stay somewhere between AWGN and Rayleigh channel.



To implement the MMSE equalizer, the weights are derived using the following formula:

$$W_k = \frac{H_k^*}{|H_k|^2 + \frac{\sigma_n^2}{\sigma_s^2}} \xrightarrow{H_k \in R} \frac{1}{W_k} = H_k + \frac{1}{SNR}$$

```
function [out, W] = MMSE_Equalizer(sig, SNR_db, H_f)
    SNR = 10^(SNR_db/10);
    W = conj(H_f)./(abs(H_f).^2+1./(length(sig)*SNR));
    out = sig.*W;
end
```

MMSE equalizer thinks it's compensating the channel, but actually it's compensating a noisy estimation of the channels fading coefficients, even when the term in the denominator vanishes in high SNR regions.

F) Header Detection and Synchronization (Bonus)

To implement the channel delay, we simply generate a random number for the delay length and then fill the received bits during delay with a random bitstream as well (to model AWGN noise). Then we choose a pilot stream which is both known in the transmitter and the receiver.

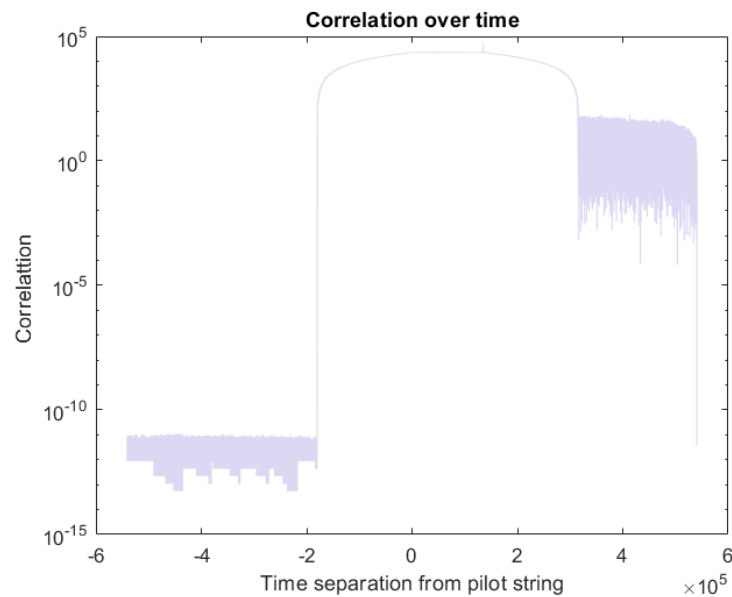
1. The "TX_cascade_frames" module should send the pilot stream as well inside the first part of the header.

```
function [out, header_f] = TX_cascade_frames(in, n_c)
    shape = size(in);
    sfc = ceil(2^13/n_c);
    out = reshape(in, 1, shape(1)*shape(2));
    header_f = randi([0 1], 1, 8*1024*(sfc+1)); % for the bonus part
    out = cat(2, header_f, out);
    out = cat(2, out, header_f);
    delay = randi([1 8*1024*(sfc+1)]); % for the bonus part
    out = cat(2, randi([0 1], 1, delay), out);
end
```

2. A module should be added before "RX_frame_detection" called "delay estimation" which will first compute the cross correlation of the received signal with the pilot string and then identify where this correlation is maximized (reaching the power of the pilot signal).
 - Cross correlation is computed for a length 3 times larger than the pilot stream
 - The pilot stream which was generated by the transmitter and fed to the receiver as an argument

```
function [idx, v, Lags] = delay_estimation(in, pilot)
len = length(pilot);
[corr_hdr_rx, Lags] = xcorr(in(1:3*len), pilot);
v = corr_hdr_rx;
corr_hdr_rx = corr_hdr_rx(Lags >= 0);
idx = find(corr_hdr_rx == max(corr_hdr_rx));
end
```

3. The peak shows the start of the header and the message is synchronized between the receiver and the transmitter



4. The part of the message which has an index bigger than the peak index will be passed through the system as data and the BER/SNR curves will be like the first part as the signal is being detected correctly (the corresponding bits are being compared correctly)

