NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CE3006 Digital Communications

Project Report

HOANG VIET
TOH KWAN HOW RAYMOND
TAN SI EN
TAN QIN KAI
WEI LUOBIN

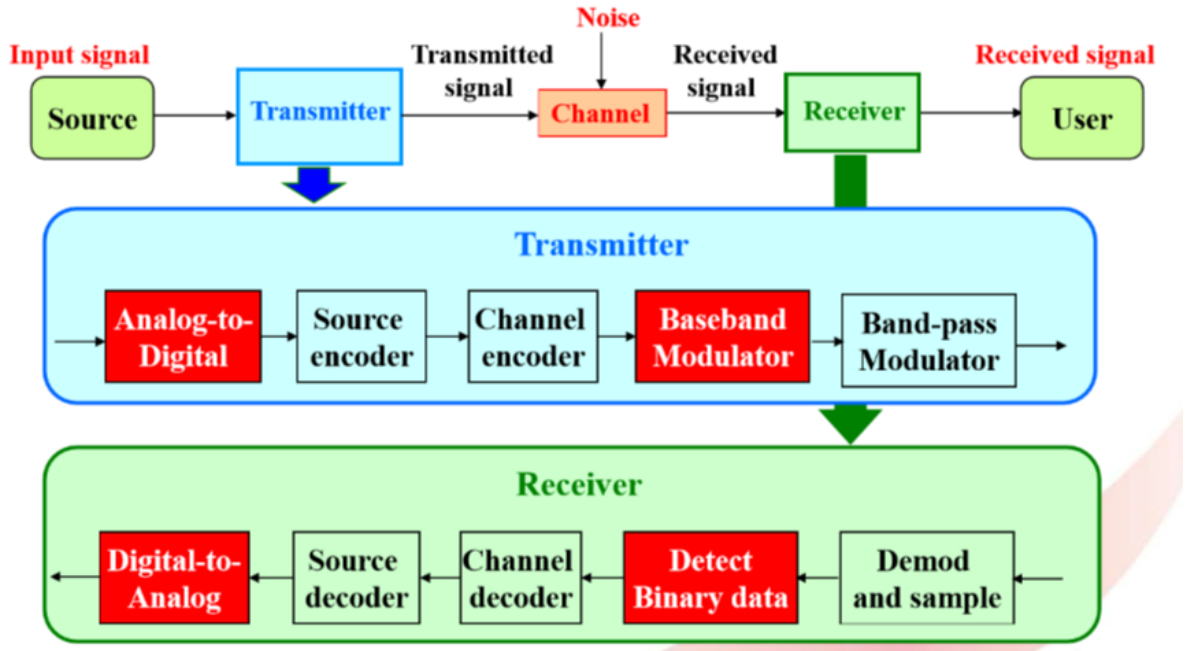
GROUP: 2
SUBMISSION DATE: 05/11/2020

Contents

Introduction 3
Experiments and Methods 4
Phase 1 4
Data Generation 4
Noise Generation 5
Received Data 6
Observation 7
Phase 2 10
Setup 10
OOK 12
BPSK 14
FSK 16
Observations 18
Phase 3 19
Conclusion 22

Introduction

This project aims to replicate real world digital communication system but without the intervention of an antenna for transmitting and receiving. The basic flow of this project is split into 3 phases, signal generation, signal modulation/demodulation, and signal encoding/decoding.
Signal Generation will consist of the generation of data bits and additive white gaussian noise which replicates real world unwanted noise. Signal modulation/demodulation will be shifting signals from low frequency (baseband) to specific higher frequency (bandpass) as real world have frequency allocation from IMDA or FCC. Lastly, signal encoding/decoding will investigate Error Control Coding to improve the performance of the communication model.



The experiments and methods will be covered below. The results from the experiment will also be shown.

Experiments and Methods

Phase 1

The first section is related to baseband modulation and demodulation. The binary data will be generated and will then be transmitted through an additive white Gaussian noise channel with different SNR values. The bit error rate performance will be analysed and plotted against a range of SNR values. The initial starting code is shown below.

```
%---------------Define Declaration---------------%
%Num of bits = 1024
nBits = 1024;

%Signal power = 1
Signal_Power = 1;

%SNR_dB = 10log(Signal_Power/Noise_Power)
%Create matrix to store SNR
SNR_dB = 0:0.5:20;

%SNR = S/N = 10^(SNR_dB/10)
SNR = (10.^(SNR_dB/10));

%Holder value for plotting later
plot_signal = rand(1,nBits);
plot_noise = rand(1,nBits);
plot_receive = rand(1,nBits);

%MODIFY THE VARIABLE BELOW TO CHOOSE AT WHICH SNR VALUE
%TO PLOT SIGNAL,NOISE and RECEIVE
plot_SNR_dB = 15;

%Counter for number of run to calculate BER for each SNR value
Total_Run = 20;
%---------------END OF DEFINE--------------------%
```

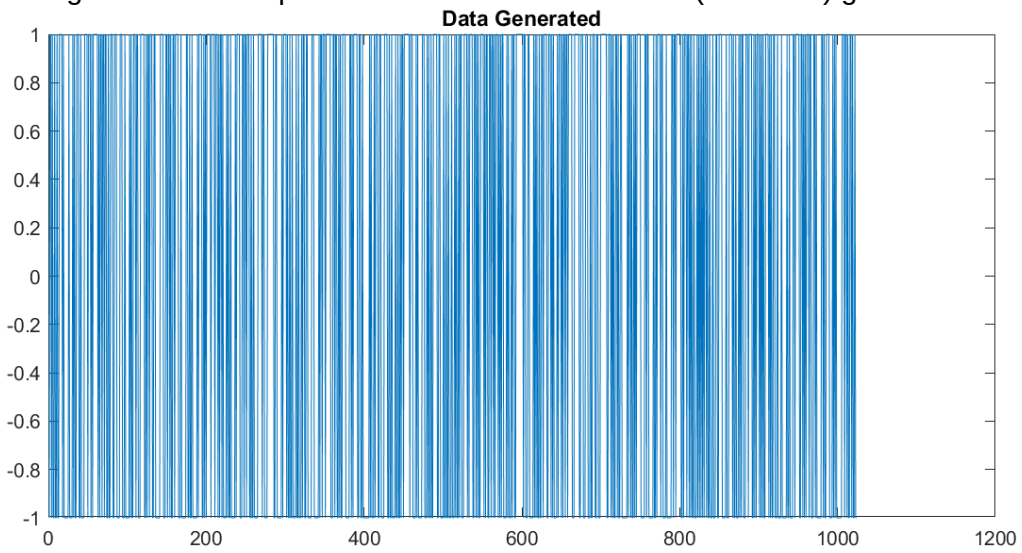
Data Generation

Random binary digits are generated with the following parameters:

- Num of bits for transmission :1024
- Binary 1 is converted to +1
- Binary 0 is converted to -1

```
%-----------TRANSMITTER------------%

%Generate random binary digits(0 or 1), INPUT SIGNAL
Data = round(rand(1,nBits));

%Convert binary digit to -1 or +1:
%if Data is 0 -> signal = 2*-0.5 = -1
%else if Data is 1 -> signal = 2*0.5 = +1
Signal = 2 .* (Data - 0.5);
```

The following is the visual representation of the random data(1024 bits) generated:

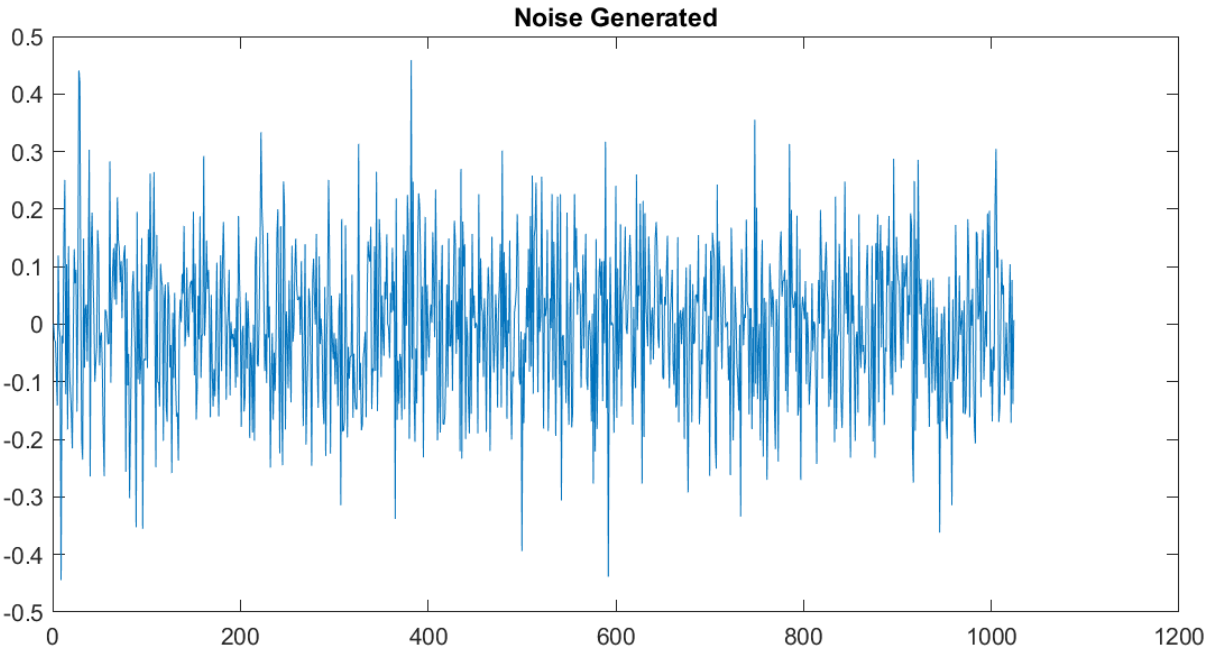


Noise Generation

For noise generation, 1024 bits of noise sample was subsequently generated in a normal distribution with zero mean and a unit variance with respect to SNR (signal to noise ratio).

```
%Get Noise Power from SNR
Noise_Power = Signal_Power ./SNR(i);

%Get Noise Variance from Noise Power
%Change the noise variance with respect to SNR (signal to noise ratio) value
Avg_Noise_Power = Noise_Power/2;

%The randn() function is for normal distribution, generated with equal number of noise samples
%(In Matlab, a.*randn(1000,1) + b is a Gaussian Dist, mean b sd a.)
Noise = sqrt(Avg_Noise_Power) .*randn(1,nBits);

%Receiver side Signal
Receive = Signal+Noise;
```

The graph below is a representation of the Additive white Gaussian noise channel with different SNR values of k.

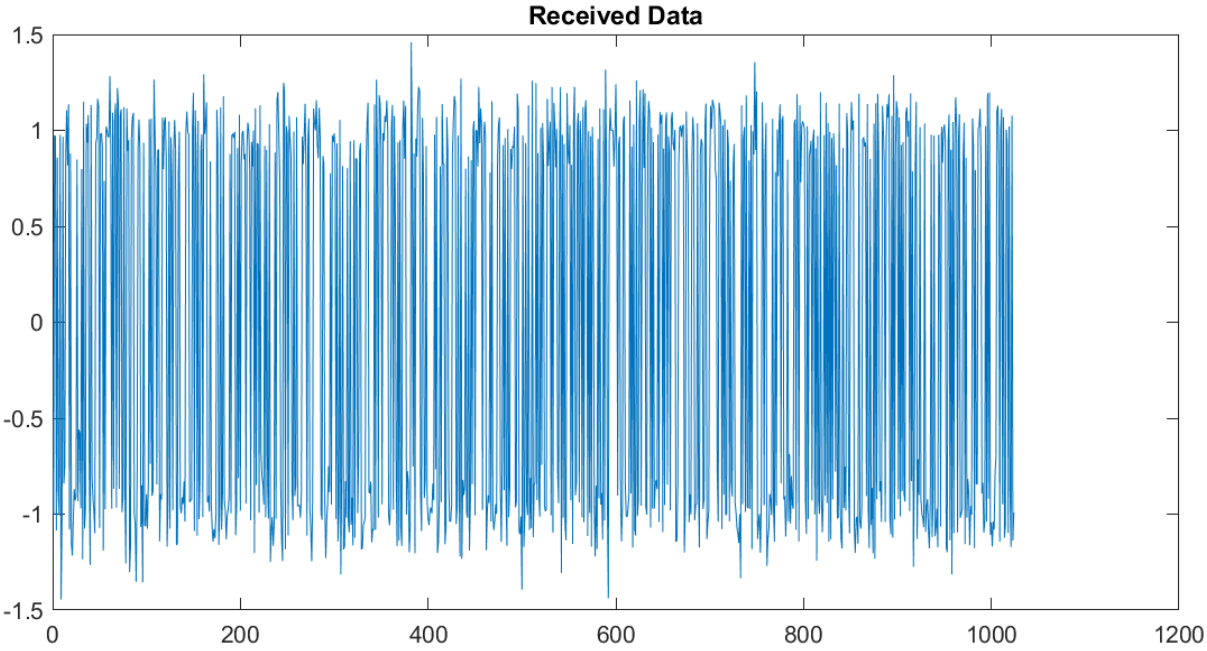


Received Data

For this simulation, it is assumed that signal is received. Noise samples obtained are then added to the transmitted data to simulate receiving of data. A threshold logic is implemented where the threshold value is fixed as 0 with the transmitted data of +1 representing logic value of 1 and transmitted data of -1 representing logic value of 0. We count the wrongly received bit for BER calculation.

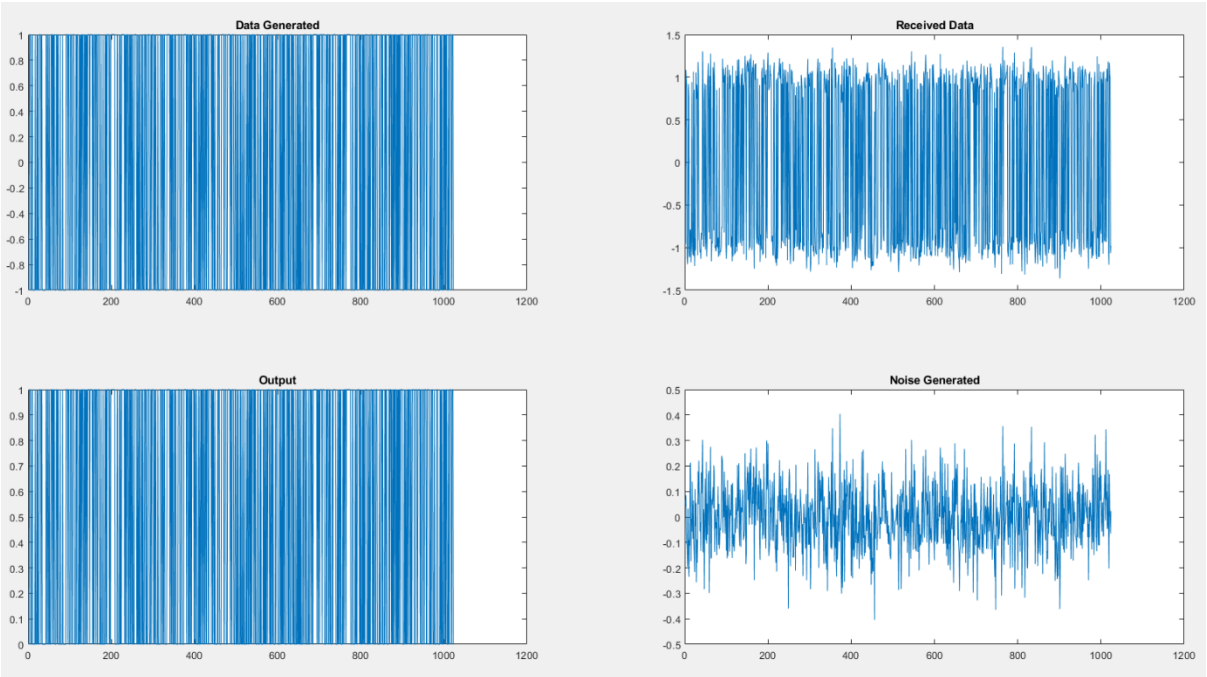
```
%-----RECEIVER------%

%Initialize threshold
Threshold = 0;

%Initialize Error for this run
Error = 0;

%Fix the threshold value as 0
%If received signal >= threshold value, threshold = 1
%If received signal < threshold value, threshold = 0
for k= 1 : nBits
    if (Receive(k)>= Threshold) && Data(k)==0||(Receive(k)<Threshold && Data(k)==1)
        Error = Error+1;
    end
end
```

The following is the visual representation of the received data(output data):



Observation

This data is then compared to the input binary digit in the following graph:



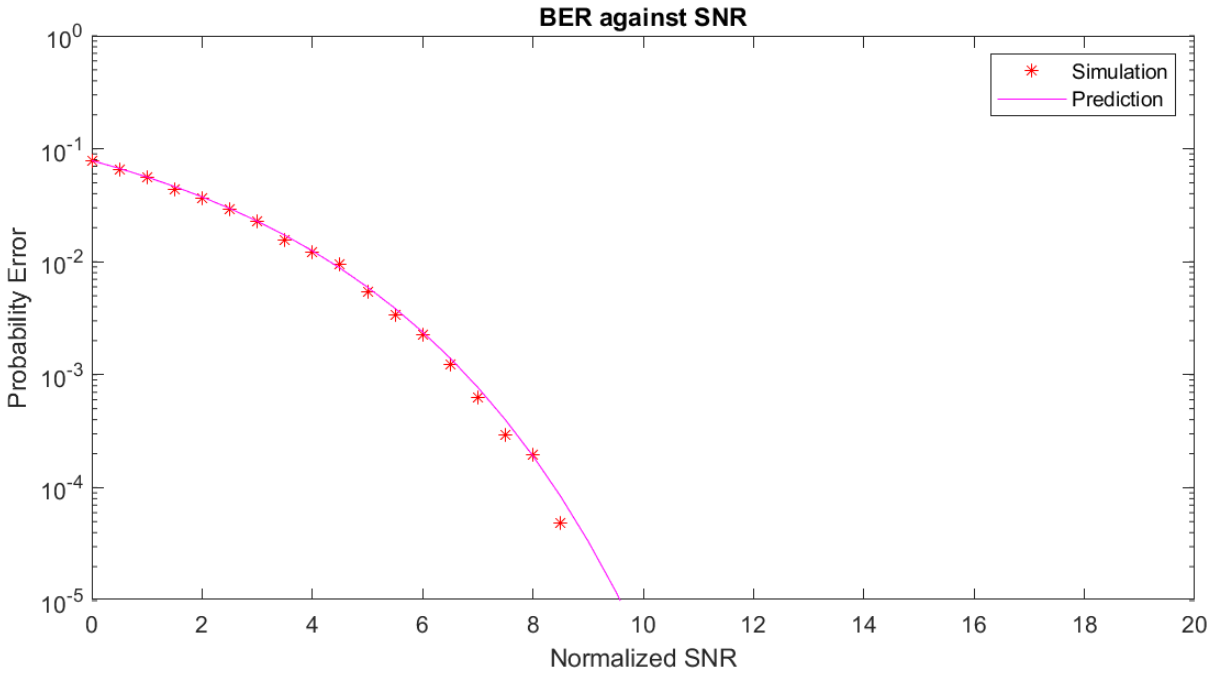
Bit error rate is then calculated with :

$$BER = \frac{No.of\ Errors\ made\ during\ Transmission}{Total\ No.of\ Bits}$$

The following shows the calculation implementation of the bit error rate:

```
        %---------------ERROR STATISTICS CALCULATION--------------------%
        %BER = TotalError divide by number of bits
        Error = Error ./nBits;

        %Accumulate the error of each run
        Sum_Error = Error + Sum_Error;

        %---------------Choose the value for plotting-----------------%
        if (plot_SNR_dB == SNR_dB(i))
            plot_signal = Signal;
            plot_noise = Noise;
            plot_receive = Receive;
        end

    end

    %Average Error
    Error_Rate(i) = Sum_Error / Total_Run;
end

%Predict BER using
Pred_BER=(1/2)*erfc(sqrt(SNR));
```

This process is then repeated for different SNR values from 0dB to 9dB in steps of 0.5dB. Bit error rate (BER is then plotted against Signal to Noise ratio (SNR) in the following graph below.



It is observed that in general the higher the SNR, the lower the BER. The Signal will be more accurate and resilient to noise when transmitting the signal.

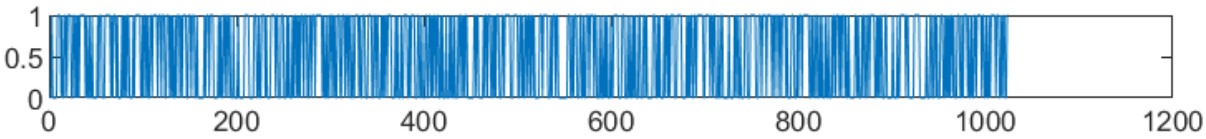
Phase 2

The second section investigates band-pass modulation and demodulation of the signal. The following modulation schemes are implemented to the generated baseband data for simulated transmission:

- **On-Off Keying (OOK)**
- **Binary Phase Shift Keying (BPSK)**
- **Frequency Shift Keying (FSK)**

Setup

A set of 1024bit of data is generated as shown in the figure below:



(cos(2πft)) is utilized as the carrier signal to carry the signal over to the receiver after modulation with a frequency (fc) of 10KHz. The carrier signal will be oversampled by 16 times (fs=16fc) at a data rate of 1kbps.Therefore there will be effectively 1 bit of data represented by 10 cosine waves. With 16 samples taken for every cosine wave, there would be 1600 sampling points per bit of data.

A 6th order filter (Butterworth) with cut-off frequency of normalized cut-off 0.2 is implemented in the receiving function to filter the signal.
The following is the implementation code for the simulation:

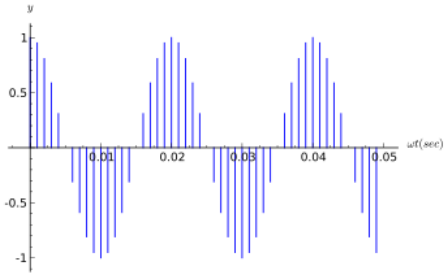
```
%define carrier frequency
fc = 10000; %10kHz
%16 times oversampled -> sample freq = 16 fc
fs = 16 * fc;

%define data rate of 1kbps
dataRate = 1000;
%define number of data bits
nBits = 1024;
%define sampling rate
samplingPeriod = fs / dataRate;

%define Amplitude
Amp = 5;
%define time steps
t = 0: 1/fs : nBits/dataRate;

%define 6th order LP butterworth filter with 0.2 normalized cutoff frequency
[b_low,a_low] = butter(6, 0.2);
%define 6th order HP butterworth filter with 0.2 normalized cutoff frequency
[b_high,a_high] = butter(6, 0.2, 'high');
```

```
%generate carrier frequency
Carrier = Amp .* cos(2*pi*fc*t);
Carrier_2 = Amp .* cos(2*pi*10*fc*t);
%calculate signal length
SignalLength = fs*nBits/dataRate + 1;

%SNR_dB = 10 log (Signal_Power/Noise_Power)
SNR_dB = -20:1:20;
%==> SNR = Signal_Power/Noise_Power = 10^(SNR_dB/10)
SNR = (10.^(SNR_dB/10));

%MODIFY THE VARIABLE BELOW TO CHOOSE AT WHICH SNR VALUE
%TO PLOT SIGNAL,NOISE and RECEIVE
plot_SNR_dB = 15;


%set run times
Total_Run = 10;

%define placeholder for error calculation
Error_RateOOK = zeros(length(SNR));
Error_RateBPSK = zeros(length(SNR));
Error_RateFSK = zeros(length(SNR));

%for each SNR value
for i = 1 : length(SNR)
    Avg_ErrorOOK = 0;
    Avg_ErrorBPSK = 0;
    Avg_ErrorFSK = 0;

    %for each SNR value, average the error over %Total_Run times
    for j = 1 : Total_Run

        %-----Data generation-----%
        Data = round(rand(1,nBits));

        %fill the data stream
        DataStream = zeros(1, SignalLength);
        for k = 1: SignalLength - 1
            DataStream(k) = Data(ceil(k*dataRate/fs));
        end
        DataStream(SignalLength) = DataStream(SignalLength - 1);
```

OOK

The following is the implementation of the OOK signal modulation scheme for communication simulation:

Data generation



Variable [Signal_OOK] is implemented to simulate a total of 163841 data points of the signal sent out whereby the carrier frequency [Carrier] is multiplied against [DataStream], obtaining a discrete time signal representation where every baseband bit is represented in steps of 160 sampled datapoints.

Modulation

The following shows the code of implementation

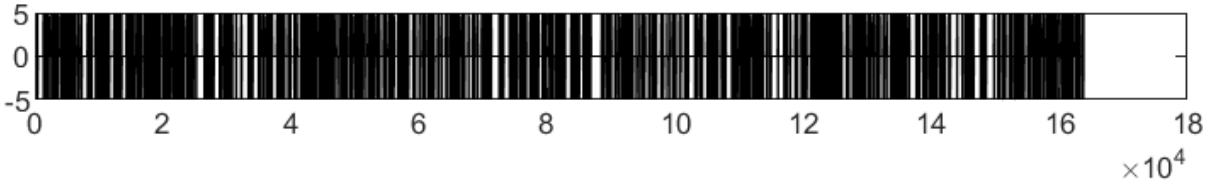
```
%----- OOK -----%
Signal_OOK = Carrier .* DataStream;
```

As our DataStream is 1 and 0, multiplying it with Carrier frequency will immediately yield OOK characteristic with bit 1's waveform is the carrier waveform and bit 0's waveform have 0 amplitude.

Additive white noise is then generated with the required SNR value and added to the modulated signal. The following shows the code of implementation.

```
%generate noise
Signal_Power_OOK = (norm(Signal_OOK)^2)/SignalLength;  %Sum of squared signal amp over signal length
Noise_Power_OOK = Signal_Power_OOK ./SNR(i);
NoiseOOK = sqrt(Noise_Power_OOK/2) .*randn(1,SignalLength);
```

The following shows the visual representation of the modulated signal.

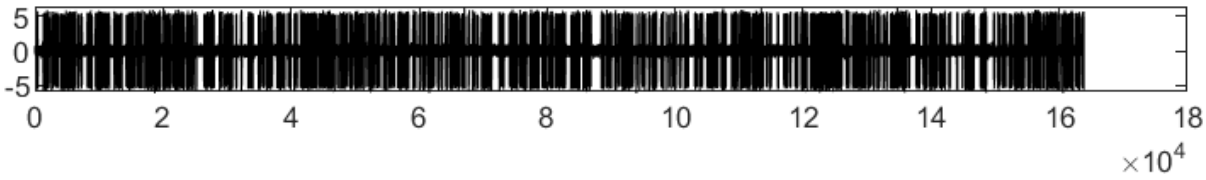


Transmission

Additive white noise is added to the transmitted signal as shown in the implementation code below:

```
%transmission
ReceiveOOK = Signal_OOK+NoiseOOK;
```

The following shows the visual representation of the transmitted signal.

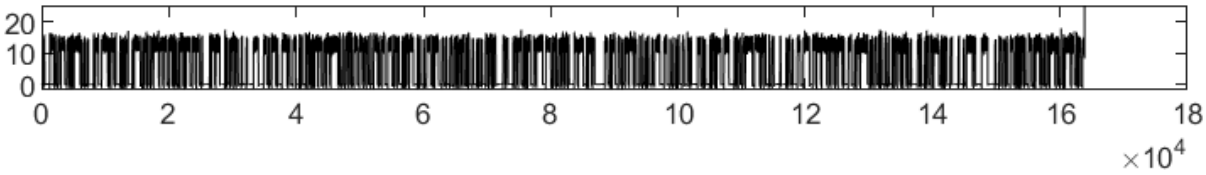


Demodulation

The signal is then demodulated non-coherently via square law detector and passed through a low pass filter as shown in the implementation code below:

```
%detection -- square law device
SquaredOOK = ReceiveOOK .* ReceiveOOK;
%low pass filter
FilteredOOK = filtfilt(b_low, a_low, SquaredOOK);
```

The following shows the visual representation of the demodulated signal.

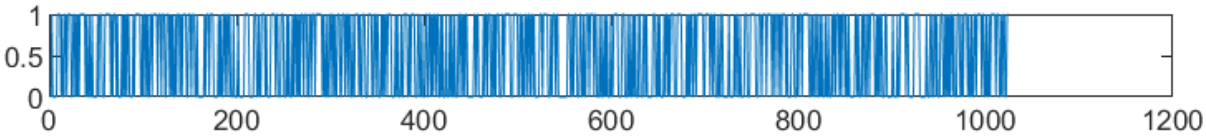


Decoded Signal

The filtered signal is then sampled and hold to yield the test statistic. A decision threshold logic of $0.5 * (A + 0)$ is implemented, where [A] is the amplitude of the signal, to decode the test statistic. The following is the implementation code:

```
%sample and decision device
sampledOOK = sample(FilteredOOK, samplingPeriod, nBits);
result_OOK = decision_device(sampledOOK,nBits, Amp/2);  %--OOK threshold is 0.5*(A+0)
```

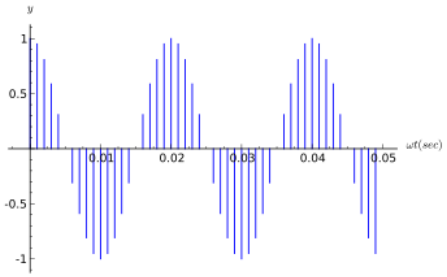
The detailed implementation of these functions can be found at the end of our .m files.
The following shows the visual representation of the decoded signal.



BPSK

The following is the implementation of BPSK signal modulation for communication simulation:

Data generation



Variable [Signal_BPSK] is implemented to simulate a total of 163841 data points of the signal sent out whereby the carrier frequency [Carrier] is multiplied against [DataStream], obtaining a discrete time signal representation where every baseband bit is represented in steps of 160 sampled datapoints.

Modulation

The following shows the code of implementation, [DataStream] is converted to a format of +1 or -1. As our carrier is Cosine, multiply it with -1 will yield a 180 phase delayed waveform. Therefore we can yield bit 1 ‘s waveform as carrier and bit 0’s waveform as 180-delayed waveform.

```
%----- binary phase shift keying -----%
DataStream_BPSK = DataStream .* 2 - 1;
Signal_BPSK = Carrier .* DataStream_BPSK;
```

Additive white noise is then generated with the required SNR value to the modulated signal. The following shows the code of implementation.

```
%generate noise
Signal_Power_BPSK = (norm(Signal_BPSK)^2)/SignalLength;
Noise_Power_BPSK = Signal_Power_BPSK ./SNR(i);
NoiseBPSK = sqrt(Noise_Power_BPSK/2) .*randn(1,SignalLength);
```

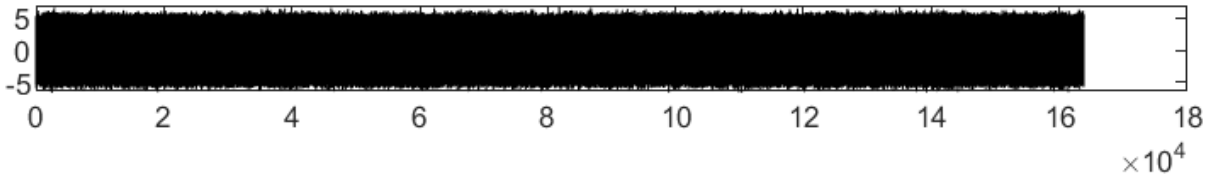
The following shows the visual representation of the modulated signal.

Transmission

Additive white noise is then added to the transmitted signal as shown in the implementation code below:

```
%transmission
ReceiveBPSK = Signal_BPSK+NoiseBPSK;
```

The following shows the visual representation of the transmitted signal.

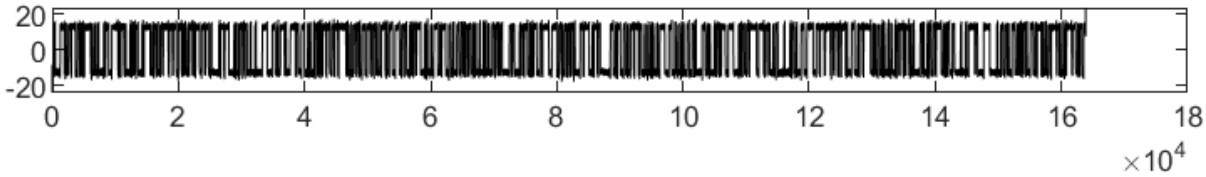


Demodulation

The signal is then demodulated similarly as OOK (noncoherently, using square law detector) and passed through a low pass filter as shown in the implementation code below

```
%non-coherent detection -- square law
SquaredBPSK = ReceiveBPSK .* Carrier;
%
OutputBPSK = filtfilt(b_low, a_low, SquaredBPSK);
```

The following shows the visual representation of the demodulated signal.

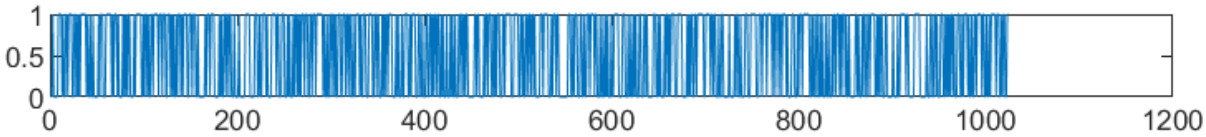


Decoding

A filter with a cut-off frequency is utilised when filtering the demodulated signal. A decision threshold logic of 0 is implemented. The following is the implementation code:

```
%sample and decision device
sampledBPSK = sample(OutputBPSK, samplingPeriod, nBits);
resultBPSK = decision_device(sampledBPSK,nBits,0);          %-- bipolar -- threshold 0
```

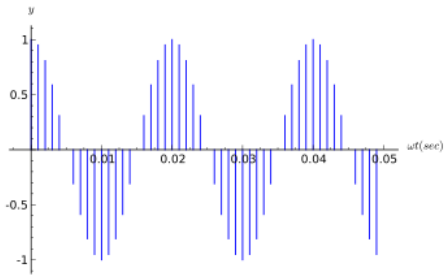
The following shows the visual representation of the decoded signal.



FSK

The following is the implementation of FSK signal modulation for communication simulation:

Data generation



Variable [Signal_FSK] is implemented to simulate a total of 163841 data points of the signal sent out whereby two carrier frequencies, [Carrier] and [Carrier_2], are multiplied against two different data streams [DataStream] and [DataStreamFSK], obtaining a discrete time signal representation where every baseband bit is represented in steps of 160 sampled datapoints. [Carrier] is Fc whereas [Carrier_2] is 10*Fc.

Modulation

The following shows the code of implementation, we create another variable [DataStreamFSK] that is the bit flip version of [DataStream]. Multiply the high frequency carrier [Carrier_2] with [DataStream], and multiply the low frequency carrier [Carrier] with the [DataStreamFSK] will yield a waveform where the bit 1 in [DataStream] is high-frequency and the bit 0 in [DataStream] (bit 1 in [DataStreamFSK]) is low-frequency.

```
%--FSK--%
DataStreamFSK = -1.*DataStream + 1; %bit flip
Signal_FSK = Carrier_2 .* DataStream + Carrier.*DataStreamFSK; % 1-- High F, 0 -- Low F
```

Additive white noise is then generated with the required SNR value and added to the modulated signal. The following shows the code of implementation.

```
%generate noise
Signal_Power_FSK = (norm(Signal_FSK)^2)/SignalLength;
Noise_Power_FSK = Signal_Power_FSK ./SNR(i);
NoiseFSK = sqrt(Noise_Power_FSK/2) .*randn(1,SignalLength);
```

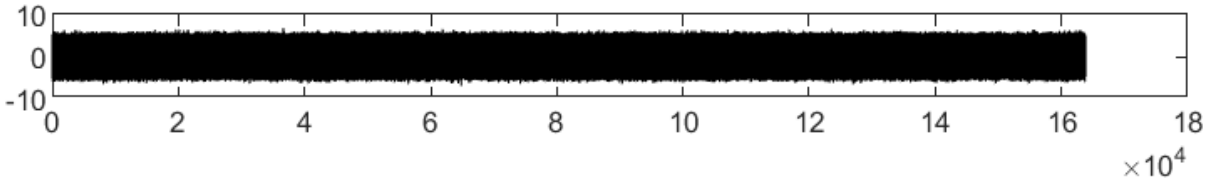
The following shows the visual representation of the modulated signal.

Transmission

Additive white noise is then added to the transmitted signal as shown in the implementation code below:

```
%transmission
ReceiveFSK = Signal_FSK+NoiseFSK;
```

The following shows the visual representation of the transmitted signal.

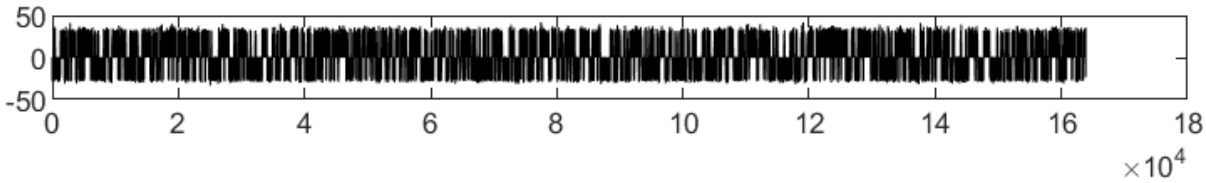


Demodulation

The signal is then demodulated and passed through two filters (one high pass, to capture the high frequency waveform; and a lowpass, to capture the low frequency waveform). The respective waveform is then noncoherently detected using square law detector before they are added together (same demodulation method as mentioned in the slides).

```
%non coherent detection -- bandpass filters
ReceiveFSK_LOW = filtfilt(b_low,a_low,ReceiveFSK);
ReceiveFSK_HIGH = filtfilt(b_high,a_high,ReceiveFSK);
%Square Law
SquaredFSK_LOW = ReceiveFSK_LOW.*ReceiveFSK_LOW;
SquaredFSK_HIGH = ReceiveFSK_HIGH.*ReceiveFSK_HIGH;
SquaredFSK = SquaredFSK_HIGH - SquaredFSK_LOW;
```

The following shows the visual representation of the demodulated signal.

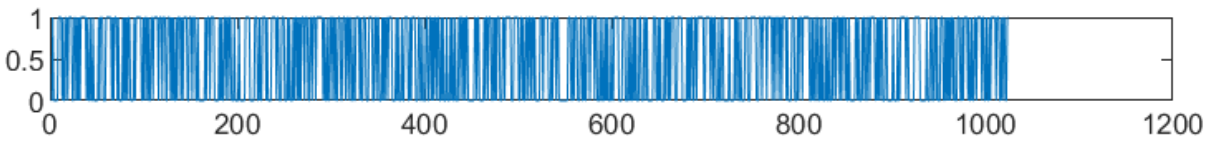


Decoding

We carry out decoding similarly to previous demodulation methods.

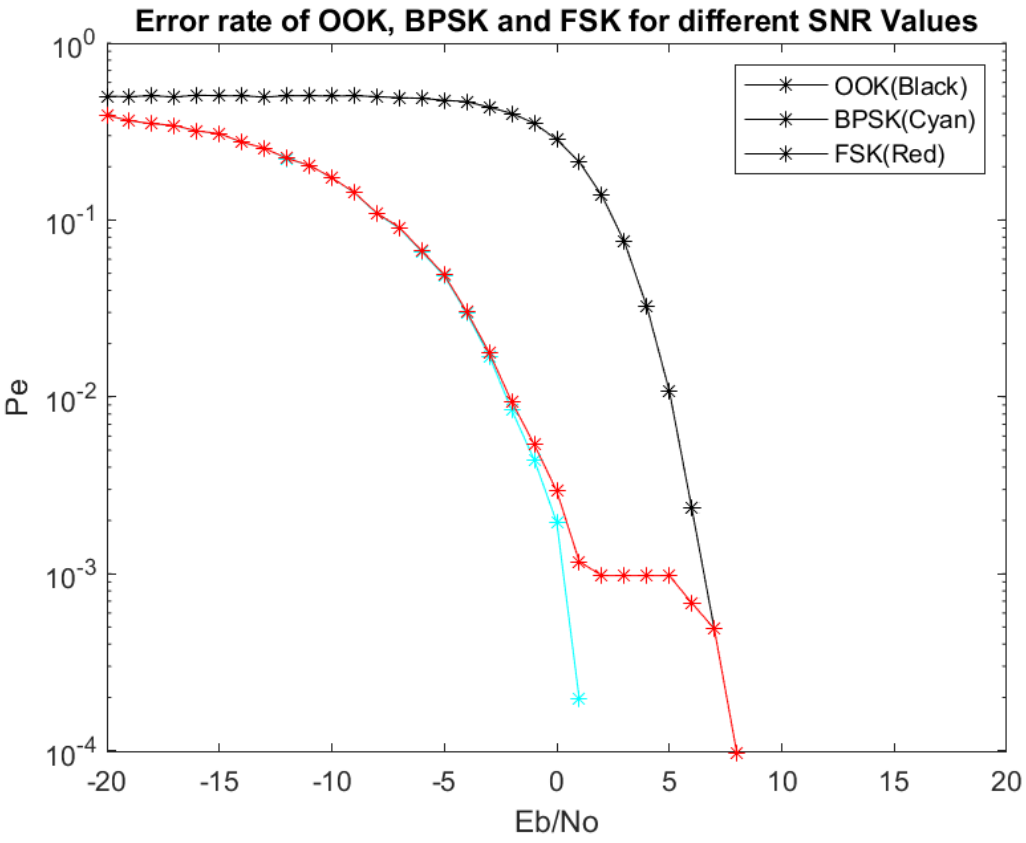
```
%sample and decision device
SampledFSK = sample(SquaredFSK, samplingPeriod, nBits);
resultFSK = decision_device(SampledFSK,nBits, 0);
```

The following shows the visual representation of the decoded signal.



Observations

Bit error rate performance of each modulation scheme on different SNR values is then calculated and plotted against a graph. The following is the plotted graph



OOK maintained a $10^{-0.3}$ Error Rate till after SNR of 0 before sharply dropping

FSK remained at $10^{-0.3}$ till after SNR of 5 and above before having a sharp drop

BPSK is steadily more accurate and consistent in error rate among all 3 schemes.

It is observed that BPSK has the lowest bit error rate followed by FSK and OOK.

This is the case as BPSK is the most immune to aptitude noise as it is depending more on delay in frequency compared to

Phase 3

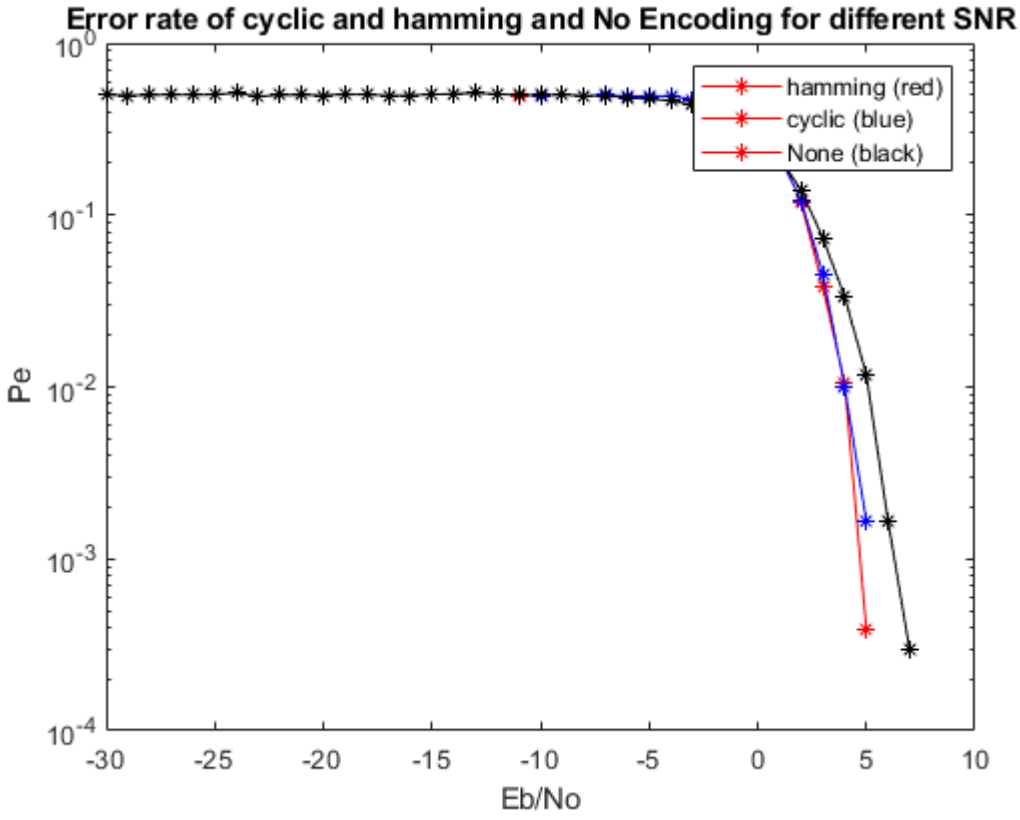
In the final section, 3 Error Control Codes (ECC) are implemented to improve the performance during modulation and demodulation. This aims to improve the BER of the system. The resultant graphs for the OOK scheme along with their respective BER are shown below along with their representation codes. The 3 ECC schemes employed are:

1. **Hamming**
2. **Cyclic**
3. **No ECC**

```
%set run times
Total_Run = 10;

%define placeholder for error calculation
Error_Rate_Hamming = zeros(length(SNR));
Error_Rate_Cyclic = zeros(length(SNR));
Error_Rate_NoEncode = zeros(length(SNR));

%for each SNR value
for i = 1 : length(SNR)
    Avg_Error_Hamming = 0;
    Avg_Error_Cyclic = 0;
    Avg_Error_NoEncode = 0;

    %for each SNR value, average the error over %Total_Run times
    for j = 1 : Total_Run

        %-----Data generation-----%
        Data = round(rand(1,nBits));
        EncodeHamming = encode(Data, 7, 4, 'hamming');
        EncodeCyclic = encode(Data,7,4,'cyclic');

        %fill the data stream
        DataStream_Hamming = zeros(1, SignalLength);
        DataStream_Cyclic = zeros(1, SignalLength);

        for k = 1: SignalLength - 1
            DataStream_Hamming(k) = EncodeHamming(ceil(k*dataRate/fs));
            DataStream_Cyclic(k) = EncodeCyclic(ceil(k*dataRate/fs));

        end
        DataStream_Hamming(SignalLength) = DataStream_Hamming(SignalLength - 1);
        DataStream_Cyclic(SignalLength) = DataStream_Cyclic(SignalLength-1);

        DataStream_NoEncode = zeros(1, SignalLength_pure);
        for k = 1:SignalLength_pure -1
            DataStream_NoEncode(k)= Data(ceil(k*dataRate/fs));
        end
        DataStream_NoEncode(SignalLength_pure) = DataStream_NoEncode(SignalLength_pure-1);
```

From the implementation code above, 2 encoding techniques (Hamming and cyclic) are used to encode the data. The data is passed through the system and on the receiving end, it is demodulated respectively for their ECC technique. No ECC was also performed to evaluate the results against the 2 ECCs employed.

```
        %----- OOK -----%
        resultOOK_Hamming = OOK_transmission(DataStream_Hamming,SNR(i),Carrier,SignalLength,samplingPeriod,Enc_nBits,Amp);
        resultOOK_Cyclic = OOK_transmission(DataStream_Cyclic,SNR(i),Carrier,SignalLength,samplingPeriod,Enc_nBits,Amp);
        resultOOK_NoEncode = OOK_transmission(DataStream_NoEncode,SNR(i),Carrier_pure,SignalLength_pure,samplingPeriod,nBits,Amp);

        decodedHamming = decode(resultOOK_Hamming,7,4,'hamming');
        decodedCyclic = decode(resultOOK_Cyclic,7,4,'cyclic');

        %--Calculate Error--%
        ErrorHamming = 0;
        ErrorCyclic = 0;
        ErrorNoEncode = 0;
        for k = 1: nBits
            if(decodedHamming(k) ~= Data(k))
                ErrorHamming = ErrorHamming + 1;
            end
            if(decodedCyclic(k) ~= Data(k))
                ErrorCyclic = ErrorCyclic + 1;
            end
            if (resultOOK_NoEncode(k) ~= Data(k));
                ErrorNoEncode = ErrorNoEncode + 1;
            end
        end
        Avg_Error_Hamming = ErrorHamming + Avg_Error_Hamming;
        Avg_Error_Cyclic = ErrorCyclic + Avg_Error_Cyclic;
        Avg_Error_NoEncode = ErrorNoEncode + Avg_Error_NoEncode;
    end
    Error_Rate_Hamming(i) = Avg_Error_Hamming/Total_Run/nBits;
    Error_Rate_Cyclic(i) = Avg_Error_Cyclic/Total_Run/nBits;
    Error_Rate_NoEncode(i) = Avg_Error_NoEncode/Total_Run/nBits;
end
```



It can be seen from the graph above that when an ECC technique is employed, the resultant BER improves slightly. This is due to the errors being detected are not simply neglected but being corrected. Hamming ECC performs the best and gives the lower BER compared to Cyclic and when ECC is not applied.

Conclusion

Throughout this project, our team has learnt and applied various modulation, demodulation and ECC techniques that can be applied to real-world digital signals during communication. The importance of BER and how it affects data integrity is also stressed. Proper analysis by the means of graphs. This project gave us a glimpse of how real-world scenarios and applications of digital communications are used.