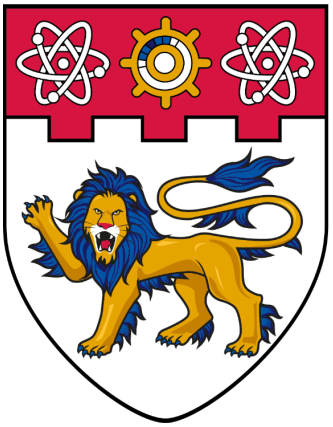

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CE3006 Digital Communications Course Project Report

AY 2022/23 Semester 1

Group 3

Team Members:

NO.	NAME	Matriculation Number
1	CHLOE LIM KE YEE	U1921121C
2	CHOCKALINGAM KASI	U1920428E
3	JARYL CHAN JIA LE	U1920976J
4	LIN XIANG	U1920370D
5	TAN KAH HENG DARRYL	U1921321H

Content Page

1. Project Objectives

The goal of this course project is to develop a basic digital communication system using MATLAB and understand the concepts involved in the process.

2. Digital Communication Systems Overview

Communication Systems can be divided into two types of communication Digital and Analog communication. An analog communication system is a communication system where the information signal sent from point A to point B can only be described as an analog signal. Digital communications is the transfer and reception of data in the form of a digital bitstream or a digitized analog signal transmitted over a point-to-point or point-to-multipoint communication channel.

Digital signals are susceptible to noise as they are sent over a channel. In order to protect the signal integrity, the digital signal can be encoded at the transmitter and decoded when it reaches the receiver. There are three main components of a digital communication system: Transmitter, Channel and Receiver. The signal is received by the transmitter from the source. The sent signal is transmitted through the channel to the receiver.

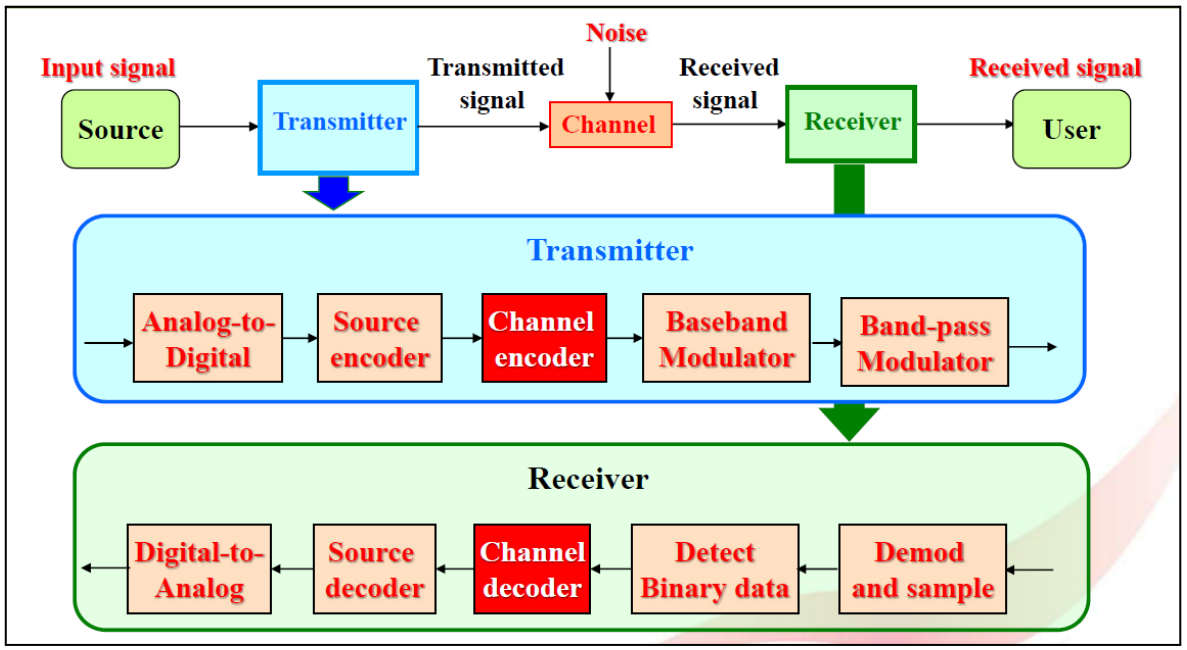

Figure 1: Diagram of communication system

The basic implementation of this project consists of three phases.

- Phase 1 : Data generation
- Phase 2 : Modulation for communication
- Phase 3 : Basic error control coding to improve the performance

3. Project Implementation

Phase 1: Data Generation

This section covers baseband modulation and demodulation where randomly generated binary data is transmitted through the additive white gaussian noise (AWGN) channel with different SNR values ranging from -50 dB to 50 dB with intervals at 5dB. We will monitor the bit error performance as the SNR values are varied during each iteration.

Code Implementation

We start by declaring the variables to be used in the data generation phase. We declare(**N_bits**) as 1024 for transmission. We generate 1024 random bits and store it as an array. The binary digits 1 is converted to +1 and 0 is converted to -1. The 1024 bits of Noise samples are generated with a normal distribution with zero mean and unit variance. The variable **Noise** stores the 1024 bits of noise in an array. We use the variable **SNR_db_Values_Array** to store the Signal to Noise Ratio Values for each iteration. The array consists of SNR values from 0db to 50db at multiples of 5dB. There are 11 iterations, so we create an array **Result** to store the bit error performance rate.

```
N_bits = 1024 ;
Raw_Data = randi([0 1], 1 , N_bits);
Signal =  2 .* (Raw_Data - 0.5);
Noise = randn(1, N_bits);
SNR_db_Values_Array = 0:5:50;
Result = zeros([1 11]);
```

For each iteration of SNR, the noise variance is derived from the equation 10log10(S/N). The Noise power is derived and mixed with noise samples to create the desired noise. The noise sample is mixed with the Signal Data to achieve **Signal_Received**, this is assumed as the received signal with the randomly generated noise values.

```
for k = 1:length(SNR_db_Values_Array)
   SNR = SNR_db_Values_Array(k);
   SignalPower = 1;
   NoisePower_variance = SignalPower ./ (10 .^ (SNR/10));
   Noise = sqrt(NoisePower_variance) .* Noise;
   Signal_Received = Signal + Noise;
```

Declaring Variables to be used while monitoring the received signal values. **Threshold** is declared at 0. The logic is used to determine if the signal received is 1 or 0. **Output** variable is used to store the output values after passing through the Threshold. **Error_Count** is used to compare against the sent signal to track the number of wrong bits. It will be used later to calculate the bit error rate.

```
   Threshold = 0;
   Output_Signal = zeros(1,N_bits);
   Error_Count = 0;
```

Looping the Signal received through a threshold and storing it in the Output_signal variable. If the bit from Signal_Received does not match the bit in Raw_Data, Error_Count is incremented by 1. The Bit_Error_Rate is determined by dividing Error_Count and N_bits. The error rate is stored in the results array.

```
    for i = 1:N_bits
        if (Signal_Received(i) > Threshold)
            Output_Signal(i) = 1;
        else
            Output_Signal(i) = 0;
        end

        if Output_Signal(i) ~= Raw_Data(i)
            Error_Count = Error_Count + 1;
        end
    end
    Bit_Error_Rate = Error_Count ./ N_bits;
    Result(k) = Bit_Error_Rate;
end
```

Plot Bit Error Rate Values against Signal to Noise Ratio as seen in Figure 1 in the Phase 1 Results section

```
figure(1)
plot(SNR_db_Values_Array,Result)
xlabel('SNR Values (dB)');
ylabel('Bit Error Rate (BER)');
title("BER vs SNR");
```

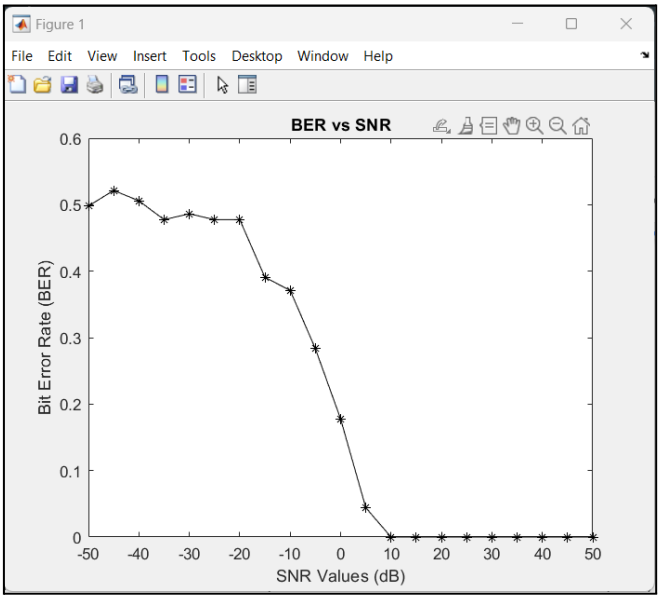

Figure 2: Bit error rate against Signal to noise ratio

In Phase 1, data is generated randomly and transmitted through an additive white noise channel. We monitor the Bit Error Performance by varying SNR from -50 db to 50 db at intervals of 5 db. As signal power becomes stronger than noise, bit error rate decreases. As expected, we can notice in Figure 2 that the Bit Error Rate (BER) decreases as the SNR values increase during each iteration.

Phase 2: Modulation for Communication

In this section, our team has implemented various band-pass modulation algorithms. On-Off Keying (OOK) and Binary Phase Shift Keying (BPSK) are simple modulation methods that we have implemented. Additionally, we have also implemented Binary Frequency Phase Shift Keying (BFSK).

2.1: On-Off Keying (OOK)

OOK is the simplest type of amplitude-shift keying (ASK) modulation. For OOK, digital signals are determined by the presence of a carrier wave. If a carrier wave is detected for a certain period, it indicates the digital signal to be binary 1. Alternatively, if there is no carrier wave detected for the same period, digital signal will be a binary 0.

Code Implementation

First, we declare all the variables that we'll be using.
We declare the number of bits (**N_bits**) as 1024 for transmission.
Carrier frequency (**Fc**) is 10kHz as given which is oversampled by 16 times to give the sampling frequency (**Fs**). **baseband_dataRate** is given as 1 kbps and it can be used to find the samples taken per bit (**`SamplesPerBit`**) as shown in the code below. Carrier signal is defined through the given equation: **A*cos(2*pi*f*t)**, where amplitude(A) = 1, f = Fc and t = the time period range set as shown below. Using 'butter' in MATLAB, a low pass butterworth filter is implemented with a 6th order and cut-off frequency of 0.2. Length of the signal (**signalLen**) is defined as shown below. Lastly, we use the variable **SNR_db_Values_Array** to store the Signal to Noise Ratio Values for each iteration and the error rate of OOK (**Bit_Error_Rate**) is initialized to an array of zeros with a length of **SNR_db_Values_Array.**

```
N_bits = 1024;
Fc = 10000;
Fs = Fc * 16;
baseband_dataRate = 1000;
SamplesPerBit = Fs / baseband_dataRate;
A = 1;
t = 0: 1/Fs : N_bits/baseband_dataRate;

carrier_sig = A .* cos(2*pi*Fc*t);
[b_low, a_low] = butter(6, 0.2);
signalLen = Fs* N_bits /baseband_dataRate + 1;
SNR_db_Values_Array = -50:5:50;
Bit_Error_Rate = zeros(1, length(SNR_db_Values_Array));
```

Next, we designed functions for a decision device and for sampling.
The decision device function: **decision_logic,** takes the sampled signal, number of bits and threshold value as inputs and outputs values that will be used for error calculations.
The sampling function: **sample**, takes the output signal of the low pass filter(**x**), sampling period and number of bits as inputs. It will output a sampled signal which will be passed into the decision_logic function.

```
function Result_Out = decision_logic(sampled,N_bits,threshold)
   Result_Out = zeros(1, N_bits);
   for x = 1:N_bits
       if (sampled(x) > threshold)
           Result_Out(x) = 1;
       else
           Result_Out(x) = 0;
       end
   end
end

function sampled = sample(x, samplingPeriod, numBit)
   sampled = zeros(1, numBit);
   for i = 1:numBit
       sampled(i) = x((2 * i - 1) * samplingPeriod / 2);
   end
end
```

We generate the data bits first. From these data bits, we generate the modulated signal (**Signal**) by multiplying it with the carrier signal. Using the modulated signal power, we calculate the noise power and generate the noise (**Noise**). This noise will be added with the signal to simulate a signal going through a channel. (**Signal_Received**).
At the receiver, we simulate a noncoherent Square Law Detector and pass the signal received through it. This gives our demodulated signal, **Squared**. We then filter this demodulated signal using the MATLAB function: *'filtfilt'*. (**Filtered**)The output will then be sampled and passed into the decision device function.
We repeat this for multiple SNR values, with each SNR value having 100 runs and we average the bit errors for one SNR run over the 100 iterations.

```
Data = randi([0 1], 1 , N_bits);
% Fill the data stream
DataStream = zeros(1, signalLen);
  for i = 1: signalLen - 1
    DataStream(i) = Data(ceil(i*baseband_dataRate/Fs));
  end
DataStream(signalLen) = DataStream(signalLen - 1);

Signal = carrier_sig .* DataStream;
% Generate noise
SignalPower = (norm(Signal)^2)/signalLen;
NoisePower_variance = SignalPower ./ Spower_2_Npower;
Noise = sqrt(NoisePower_variance/2) .*randn(1,signalLen);
Signal_Received = Signal + Noise;
%square law device (detection)
Squared = Signal_Received.^2;
% Filtering of the demodulated signal
Filtered = filtfilt(b_low, a_low, Squared);
% Use the decision threshold logic for decoding of received signals
Sampled = sample(Filtered, SamplesPerBit, N_bits);
Result = decision_logic(Sampled,N_bits,(A*A)/2);

% Calculate the bit error rate performance
No_Bit_Error = 0;
for i = 1: N_bits - 1
```

```
    if(Result(i) ~= Data(i))
        No_Bit_Error = No_Bit_Error + 1;
    end
end
avg_error = No_Bit_Error + avg_error;
```

Phase 2: OOK Results

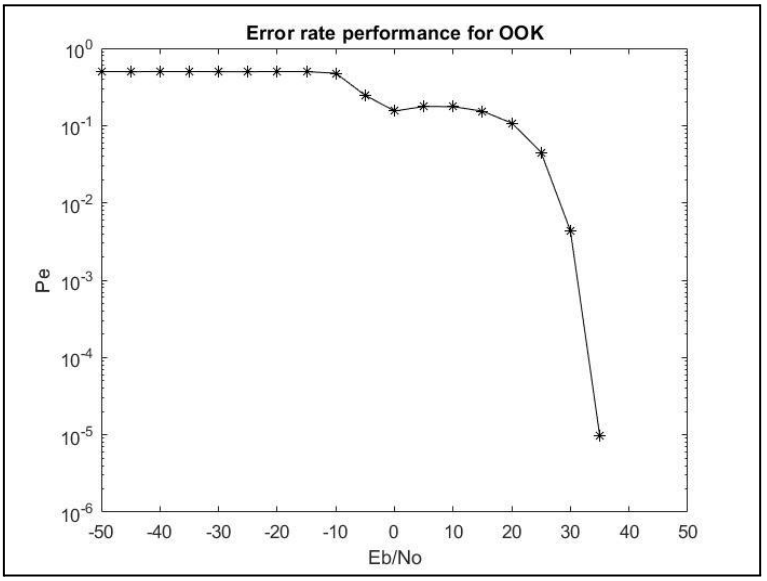

Figure 3: Bit error rate performance for OOK

Figure 3 shows the error rate performance for OOK when error is averaged over 100 times for each SNR value. It is observed that the plot follows the expected output as the bit error rate(Pe) and SNR(Eb/No) has an inverse relation.

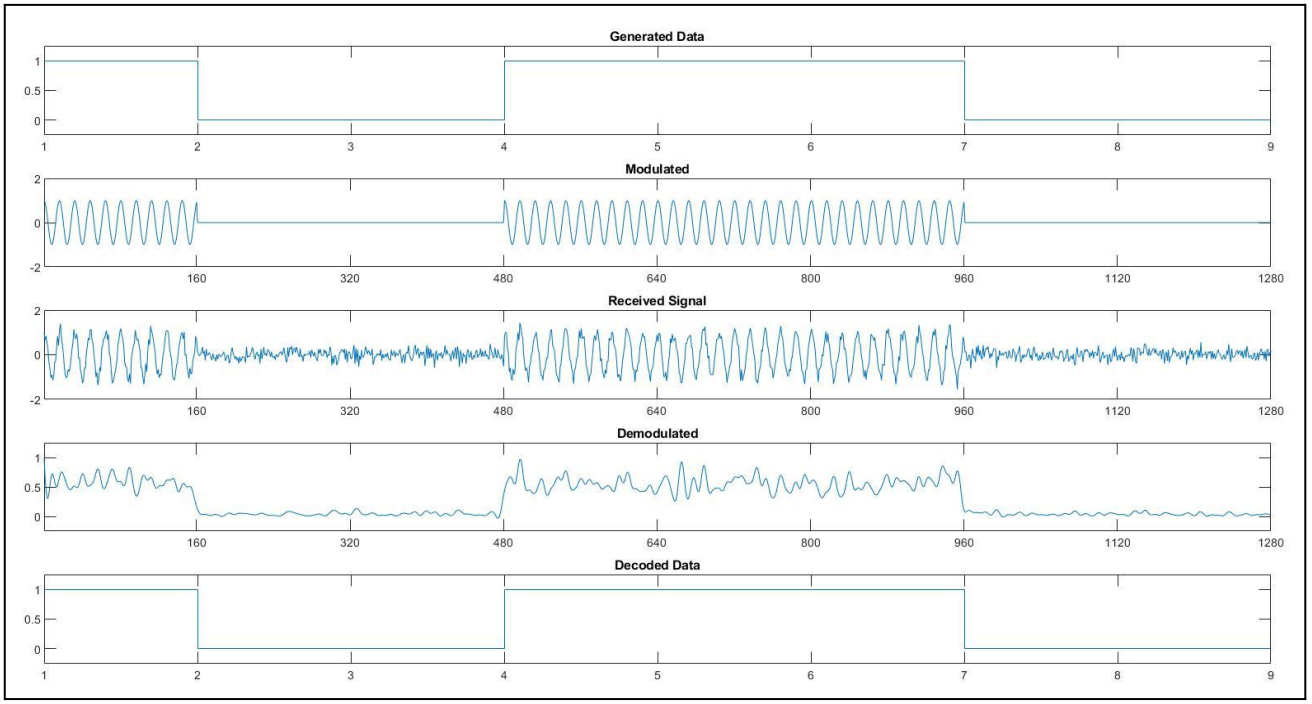

Figure 4: Results for OOK signal after modulation and demodulation

Figure 4 shows the signals at the various stages. We have set a range to see the signal for the first 8 bits to get a clearer picture of how the signal looks at each stage.

As seen in the graph, the amplitude of the modulated data is 1 and 0 when generated data is '1' bit and '0' bit respectively which is expected for OOK. After demodulation, the Decoded data is the same as the original generated data, meaning that this implementation of OOK works.

2.2: Binary Phase-shift Keying (BPSK)

BPSK is the most primitive form of phase shift keying(PSK). For BPSK, the data signals are in the +1 or -1 format.

The steps for initialization, modulation and demodulation are the same as OOK and the sample and decision_logic functions are used in BPSK as well.

Code Implementation

First, we declare all the variables that we'll be using. We declare the number of bits (**N_bits**) as 1024 for transmission. Carrier frequency (**Fc**) is 10kHz as given which is oversampled by 16 times to give the sampling frequency (**Fs**). **baseband_dataRate** is given as 1 kbps and it can be used to find the samples taken per bit (**SamplesPerBit**) as shown in the code below. Carrier signal is defined through the given equation: **A*cos(2*pi*f*t)**, where amplitude(A) = 1, f = Fc and t = the time period range set as shown below. Using 'butter' in MATLAB, a low pass butterworth filter is implemented with a 6th order and cut-off frequency of 0.2. Length of the signal (**signalLen**) is defined as shown below. Lastly, we use the variable **SNR_db_Values_Array** to store the Signal to Noise Ratio Values for each iteration and the error rate of OOK (**Bit_Error_Rate**) is initialized to an array of zeros with a length of **SNR_db_Values_Array.**

```
N_bits = 1024;
Fc = 10000;
Fs = Fc * 16;
baseband_dataRate = 1000;
SamplesPerBit = Fs / baseband_dataRate;
A = 1;
t = 0: 1/Fs : N_bits/baseband_dataRate;
carrier_sig = A .* cos(2*pi*Fc*t);
[b_low, a_low] = butter(6, 0.2);
signalLen = Fs* N_bits /baseband_dataRate + 1;
SNR_db_Values_Array = -50:5:50;
Bit_Error_Rate = zeros(1, length(SNR_db_Values_Array));
```

Next, we designed functions for a decision device and for sampling. The decision device function: **decision_logic,** takes the sampled signal, number of bits and threshold value as inputs and outputs values that will be used for error calculations. The sampling function: **sample**, takes the output signal of the low pass filter(**x**), sampling period and number of bits as inputs. It will output a sampled signal which will be passed into the decision_logic function.

```
function Result_Out = decision_logic(sampled,N_bits,threshold)
   Result_Out = zeros(1, N_bits);
   for x = 1:N_bits
       if (sampled(x) > threshold)
           Result_Out(x) = 1;
       else
```

```
            Result_Out(x) = 0;
        end
    end
end
function sampled = sample(x, samplingPeriod, numBit)
    sampled = zeros(1, numBit);
    for i = 1:numBit
        sampled(i) = x((2 * i - 1) * samplingPeriod / 2);
    end
end
```

We generate the data bits first. From these data bits, we generate the modulated signal (**Signal**) by multiplying it with the carrier signal. For "1", the phase will follow the carrier signal. However, if it is "0", then the phase will be shifted 180 degrees. Using the modulated signal power, we calculate the noise power and generate the noise (**Noise**). This noise will be added with the signal to simulate a signal going through a channel. (**Signal_Received**). At the receiver, we simulate a coherent detector which multiplies with the carrier signal before filtering and passes the signal received through it. This gives our demodulated signal, **Mul_with_carrier**. We then filter this demodulated signal using the MATLAB function: *'filtfilt'*. (**Filtered**)The output will then be sampled and passed into the decision device function. We repeat this for multiple SNR values, with each SNR value having 100 runs and we average the bit errors for one SNR run over the 100 iterations.

```
Data = randi([0 1], 1 , N_bits);
% Fill the data stream
DataStream = zeros(1, signalLen);
  for i = 1: signalLen - 1
    DataStream(i) = Data(ceil(i*baseband_dataRate/Fs));
  end
DataStream(signalLen) = DataStream(signalLen - 1);
DataStream = DataStream .*2 - 1;

Signal = carrier_sig .* DataStream;
% Generate noise
SignalPower = bandpower(Signal);
NoisePower_variance = SignalPower ./ Spower_2_Npower;
Noise = sqrt(NoisePower_variance/2) .*randn(1,signalLen);
Signal_Received = Signal + Noise;

Mul_with_carrier = Signal_Received .* carrier_sig;
% Filtering of the demodulated signal
Filtered = filtfilt(b_low, a_low, Mul_with_carrier);
% Use the decision threshold logic for decoding of received signals
Sampled = sample(Filtered, SamplesPerBit, N_bits);
Result = decision_logic(Sampled,N_bits,(A*A)/2);

% Calculate the bit error rate performance
No_Bit_Error = 0;
for i = 1: N_bits - 1
   if(Result(i) ~= Data(i))
      No_Bit_Error = No_Bit_Error + 1;
   end
```

```
end
avg_error = No_Bit_Error + avg_error;
```

BPSK Results

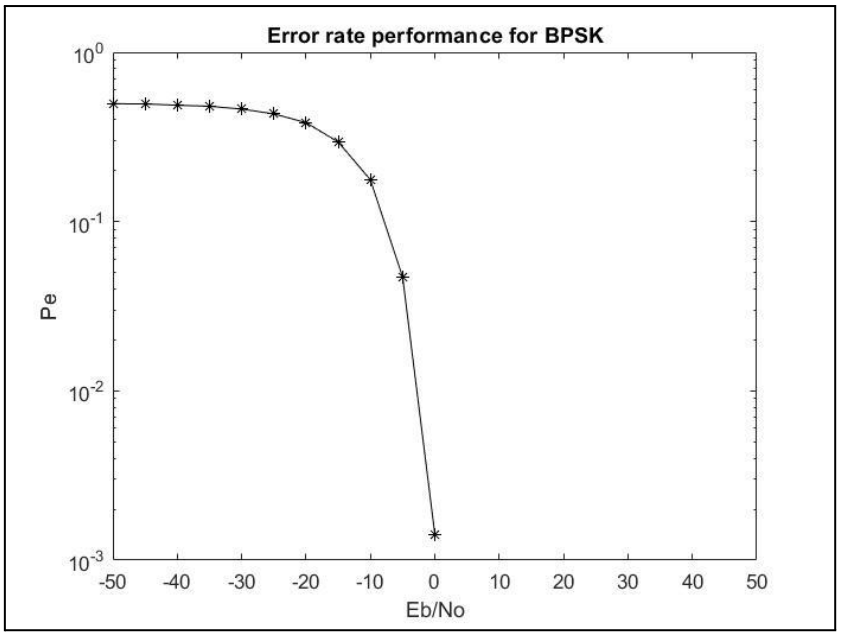

Figure 5: Bit error rate performance for BPSK

Figure 5 shows the error rate performance for BPSK when error is averaged over 100 times for each SNR value. It is observed that the plot follows the expected output as the bit error rate(Pe) and SNR(Eb/No) has an inverse relation.

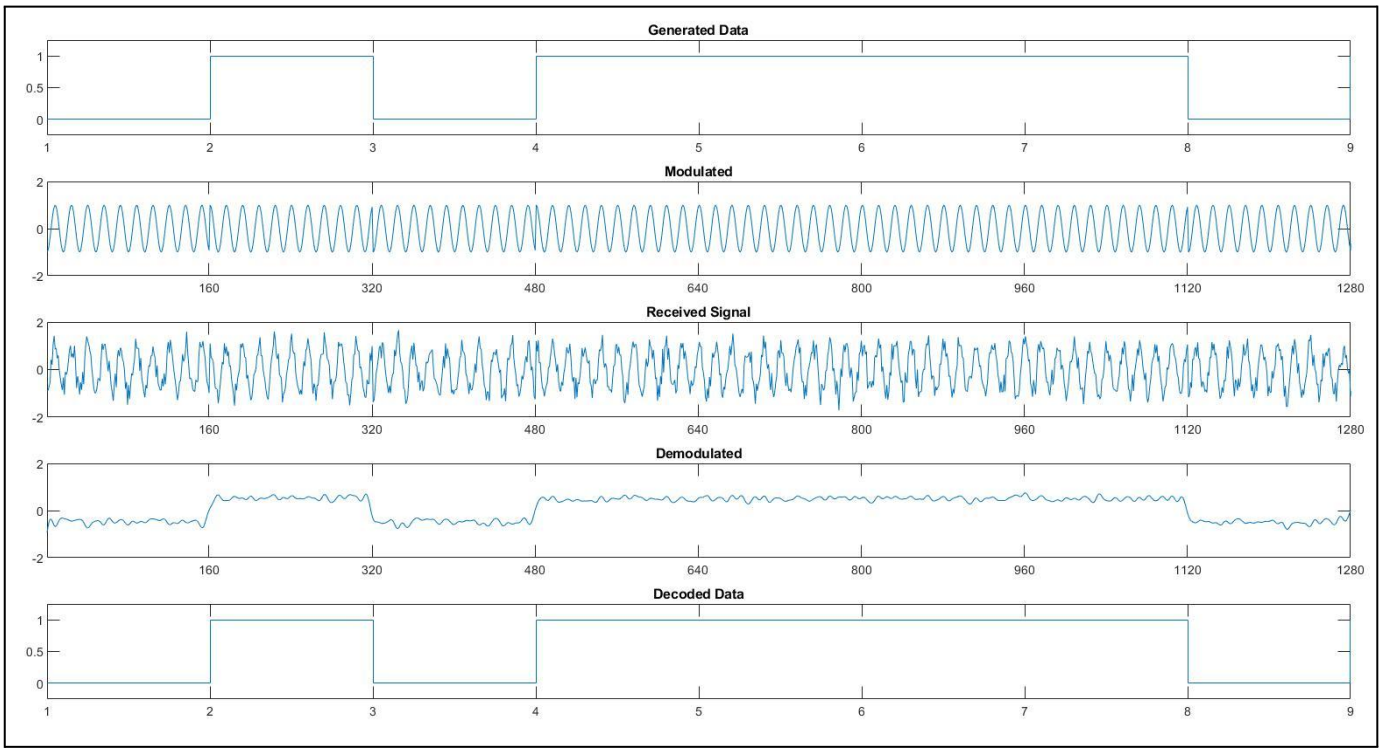

Figure 6: Results for BPSK signal after modulation and demodulation

Figure 6 shows the signals at the various stages. We have set a range to see the signal for the first 8 bits to get a clearer picture of how the signal looks at each stage.

As seen in the graph, the phase of the modulated data changes 180 degrees between the binary symbols of “0” and “1”. After demodulation, the Decoded data is the same as the original generated data, meaning that this implementation of BPSK works.

2.3: Binary Frequency-shift Keying (BFSK)

BFSK is the most primitive form of frequency shift keying(FSK). For BFSK , the data signals are in the +1 or 0 format. The steps for initialization, modulation and demodulation are the same as OOK and the sample and decision_logic functions are used in BFSK as well.

Code Implementation

First, we declare all the variables that we’ll be using. We declare the number of bits (**N_bits**) as 1024 for transmission.

Carrier frequency (**Fc**) is 10kHz as given which is oversampled by 16 times to give the sampling frequency (**Fs**). **baseband_dataRate** is given as 1 kbps and it can be used to find the samples taken per bit (**`SamplesPerBit`**) as shown in the code below. Carrier signal is defined through the given equation: **A*cos(2*pi*f*t)**, where amplitude(A) = 1, f = Fc and t = the time period range set as shown below. Length of the signal (**signalLen**) is defined as shown below. Lastly, we use the variable **SNR_db_Values_Array** to store the Signal to Noise Ratio Values for each iteration and the error rate of OOK (**Bit_Error_Rate**) is initialized to an array of zeros with a length of **SNR_db_Values_Array.**

Unlike OOK and BPSK, we need to generate 2 carrier signals, each with different frequency to correspond to the 2 symbols in the data. The detector we are designing requires 2 filters. Using ‘butter’ in MATLAB, a low pass and a high pass butterworth filters are implemented with a 6th order and cut-off frequency of 0.2.

```
N_bits = 1024;
Fc = 10000;
Fs = Fc * 16;
baseband_dataRate = 1000;
SamplesPerBit = Fs / baseband_dataRate;
A = 1;
t = 0: 1/Fs : N_bits/baseband_dataRate;

carrier_sig = Amp .* cos(2*pi*Fc*t);
carrier_sig2 = Amp .* cos(2*pi*(10*Fc)*t);
% Assume a 6th order filter with cut-off frequency 0.2 in the function
[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');
signalLen = Fs* N_bits /baseband_dataRate + 1;
SNR_db_Values_Array = -50:5:50;
Bit_Error_Rate = zeros(1, length(SNR_db_Values_Array));
```

Next, we designed functions for a decision device and for sampling.

The decision device function: **decision_logic,** takes the sampled signal, number of bits and threshold value as inputs and outputs values that will be used for error calculations.

The sampling function: **sample**, takes the output signal of the low pass filter(**x**), sampling period and number of bits as inputs. It will output a sampled signal which will be passed into the decision_logic function.

```
function Result_Out = decision_logic(sampled,N_bits,threshold)
   Result_Out = zeros(1, N_bits);
   for x = 1:N_bits
       if (sampled(x) > threshold)
           Result_Out(x) = 1;
       else
           Result_Out(x) = 0;
       end
   end
end
function sampled = sample(x, samplingPeriod, numBit)
   sampled = zeros(1, numBit);
   for i = 1:numBit
       sampled(i) = x((2 * i - 1) * samplingPeriod / 2);
   end
end
```

We generate the data bits first. From these data bits, we generate the modulated signal (**Signal**) by multiplying it with the carrier signal. For “0”, the frequency will follow the carrier signal. However, if it is “1”, then the frequency will be multiplied 10 times. Using the modulated signal power, we calculate the noise power and generate the noise (**Noise**). This noise will be added with the signal to simulate a signal going through a channel. (**Signal_Received**). At the receiver, we simulate a noncoherent detector and pass the signal received through it. This gives our demodulated signal, **Summation**. We then filter this demodulated signal using the MATLAB function: *‘filtfilt’*. (**Filtered**)The output will then be sampled and passed into the decision device function. We repeat this for multiple SNR values, with each SNR value having 100 runs and we average the bit errors for one SNR run over the 100 iterations.

```
Data = randi([0 1], 1 , N_bits);
% Fill the data stream
DataStream = zeros(1, signalLen);
  for i = 1: signalLen - 1
    DataStream(i) = Data(ceil(i*baseband_dataRate/Fs));
  end
DataStream(signalLen) = DataStream(signalLen - 1);

Signal_1 = DataStream .* carrier_sig2;
Signal_0 = (1 - DataStream) .* carrier_sig;
Signal = Signal_0 + Signal_1;
% Generate noise
SignalPower = bandpower(Signal);
NoisePower_variance = SignalPower ./ Spower_2_Npower;
Noise = sqrt(NoisePower_variance/2) .*randn(1,signalLen);
Signal_Received = Signal + Noise;
% Filter the signal received first.
Filtered_0 = filtfilt(b_low,a_low,Signal_Received);
Filtered_1 = filtfilt(b_high,a_high,Signal_Received);
% The squaring of output;
Squared_0 = Filtered_0.^2;
```

```
Squared_1 = Filtered_1.^2;
% Adding both;
Summation = Squared_1 - Squared_0;
% Use the decision threshold logic for decoding of received signals
Sampled = sample(Summation, SamplesPerBit, N_bits);
Result = decision_logic(Sampled,N_bits,(A*A)/2);
% Calculate the bit error rate performance
No_Bit_Error = 0;
for i = 1: N_bits - 1
   if(Result(i) ~= Data(i))
      No_Bit_Error = No_Bit_Error + 1;
   end
end
avg_error = No_Bit_Error + avg_error;
```

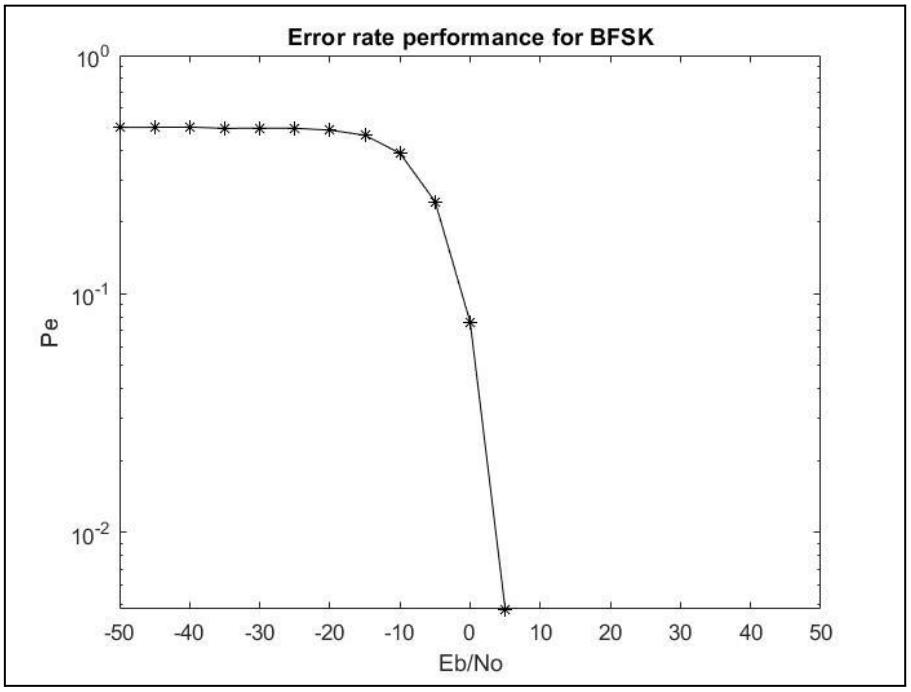

Figure 7: Bit error rate performance for BFSK

Figure 7 shows the error rate performance for BFSK when error is averaged over 100 times for each SNR value. It is observed that the plot follows the expected output as the bit error rate(Pe) and SNR(Eb/No) has an inverse relation.

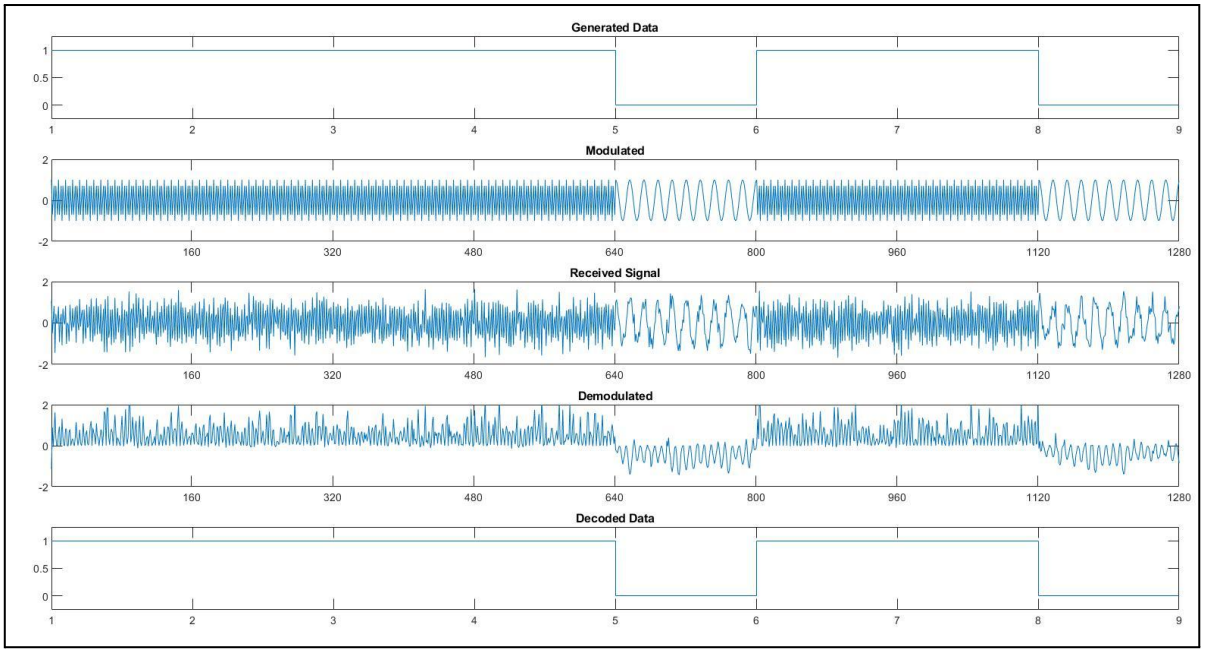

Figure 8: Results of BFSK signal after modulation and demodulation

Figure 8 shows the signals at the various stages. We have set a range to see the signal for the first 8 bits to get a clearer picture of how the signal looks at each stage. As seen in the graph, the frequency of the modulated data changes between the binary symbols of “0” and “1”. After demodulation, the Decoded data is the same as the original generated data, meaning that this implementation of BFSK works.

Phase 3: Basic Error Control Coding to Improve the Performance

In our proposed communication, there is prevalent noise introduced into the channel. Hence, the message that is passed through the channel will be distorted or corrupted to a certain extent. This increases the difficulty of the receiver to decipher the intended message.

Therefore, error control coding is introduced to control the erroneous bits in the message, making it easier for the receiver to obtain an ideal message. Error control is done by having an encoder and a decoder. The encoder introduces redundancy in the message binary sequence, making the original message into code words. Meanwhile, the decoder utilizes these added redundancies, the code words, to alleviate the errors caused by the presence of noise, interference and other distortions in the channel.

In this section we will cover some of the error control techniques using the tools available within MATLAB software. The functions used are from the MATLAB error detection and correction category, where the function encode is used as the encoder and the function decode is used as the decoder. These techniques implemented are Hamming code, linear block code and cyclic block code. For each type of channel coding, we will implement it on OOK, BPSK and BFSK modulations. Then we compare these bit error rate performance of the error control techniques by plotting using MATLAB semilogy function.

Hamming Code

Hamming Codes are the first class of binary linear block codes. They are widely used for digital storage systems and long distance telephony. They use a block parity mechanism where the data is divided into blocks, and parity is added to the block. Hamming code can correct single-bit errors and detect the presence of two-bit errors in a data block.

The code implementation will be as follows, where n is the codeword length and k is the message length.

In the process of encoding, we would carry out the following process:

1. Calculate the number of redundant bits
2. Position the redundant bits
3. Calculate the values of each redundant bit.

Next, the steps to decode a hamming code would be as follow:

1. Calculate the number of redundant bits
2. Position the redundant bits
3. Carry out parity checking
4. Carry out error detection and correction.

Matlab Code :

Encoding: encData = encode(data,n,k,'hamming/binary');

Decoding: decData = decode(encData,n,k,'hamming/binary');

Results for the Hamming code implementation are shown in the Phase 3 Results section for OOK, BPSK & BFSK

Linear Block Code

Linear block code is a form of error correction method. The original message will be split into code blocks. For each code block, a parity check portion will be added to the original segment of the message to form a code word. This method is called linear block codes because the vector sum of two code words is always a code word.

Linear block code can be classified as systematic or unsystematic. An (n,k) linear block code is systematic only when it satisfies the condition where every code word consists of a message part and a parity check part. Else, it would be unsystematic.

The message part will contain k unaltered message bits and the parity check part will contain (n-k) parity check bits. Together, they form a codeword. An example of a linear code is shown below, taken from the lecture slides.

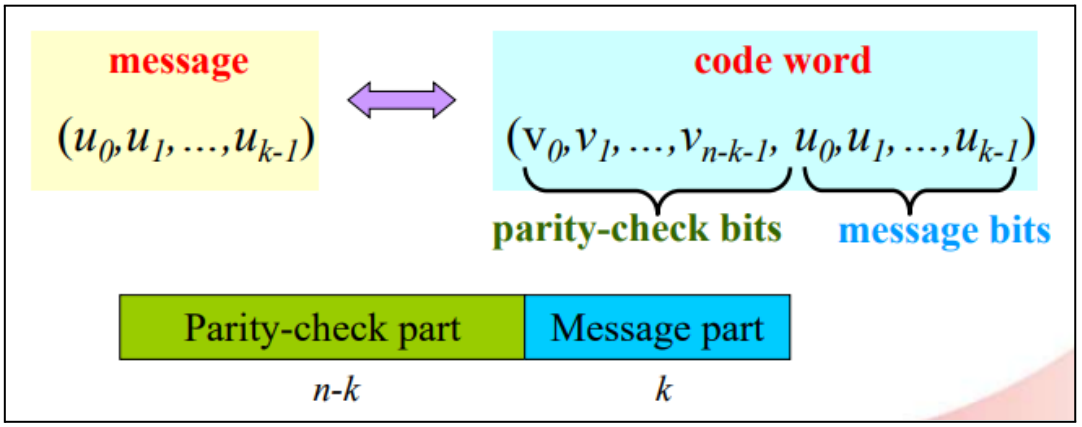

Figure 9: Message to code word diagram

Linear block code uses a generator matrix to create the code word from a message. Using this generator matrix, a parity check matrix, H, can be created. This parity check matrix uniquely identifies the code. A syndrome table can be created from this parity check matrix to reflect the erroneous part of the message.
The receiver can identify the error bits by using the syndrome table to calculate the syndrome digits. Through the syndrome digits, the receiver obtains important information about the erroneous digits and this information can be used for error correction.
The crucial steps for the creation of the generator matrix, the syndrome table, and the encoding and decoding of the code words are shown below.

Matlab Code implementation:

Create a cyclic generator polynomial: pol = cyclpoly(n,k);
Create a parity-check matrix using the generator polynomial: parmat = cyclgen(n,pol);
Create a generator matrix: genmat= gen2par(parmat);
Create a syndrome table: trt = syndtable(parmat);

Encoding: encData = encode(data,n,k,linear/binary',genmat);
Decoding: decData = decode(encData,n,k,linear/binary',genmat,trt);
Results for the Hamming code implementation are shown in the Phase 3 Results section for OOK, BPSK & BFSK

Cyclic Block Code

Cyclic Code is a subclass of linear block codes.
An (n,k) linear code is considered cyclic only if every cyclic shift of each code word results in another code vector.
In cyclic code, the code structure is represented by code polynomials. This means that cyclic codes will be subjected to binary field arithmetic operation where addition is an XOR operation, multiplication is an AND operation, and lastly subtraction and addition are the same operation.
As a result, this code polynomial structure makes the encoding and syndrome decoding simpler for communication.
Instead of a generator matrix, cyclic block code has a generator polynomial to create the code words. Similarly to linear block codes, it also has systematic and unsystematic code words.
Creation of the generator polynomial, the parity check portion, along with the syndrome table are shown below.

Matlab Code implementation:

Create a cyclic generator polynomial: pol = cyclpoly(n,k);
Create a parity-check matrix using the generator polynomial: parmat = cyclgen(n,pol);
Create a syndrome table: trt = syndtable(parmat);
Encoding: encData = encode(data,n,k,'cyclic/binary',genpoly);
Decoding: decData = decode(encData,n,k,'cyclic/binary',genpoly,trt);
Results for the Hamming code implementation are shown in the Phase 3 Results section for OOK, BPSK & BFSK

3.1 OOK Error Control Results

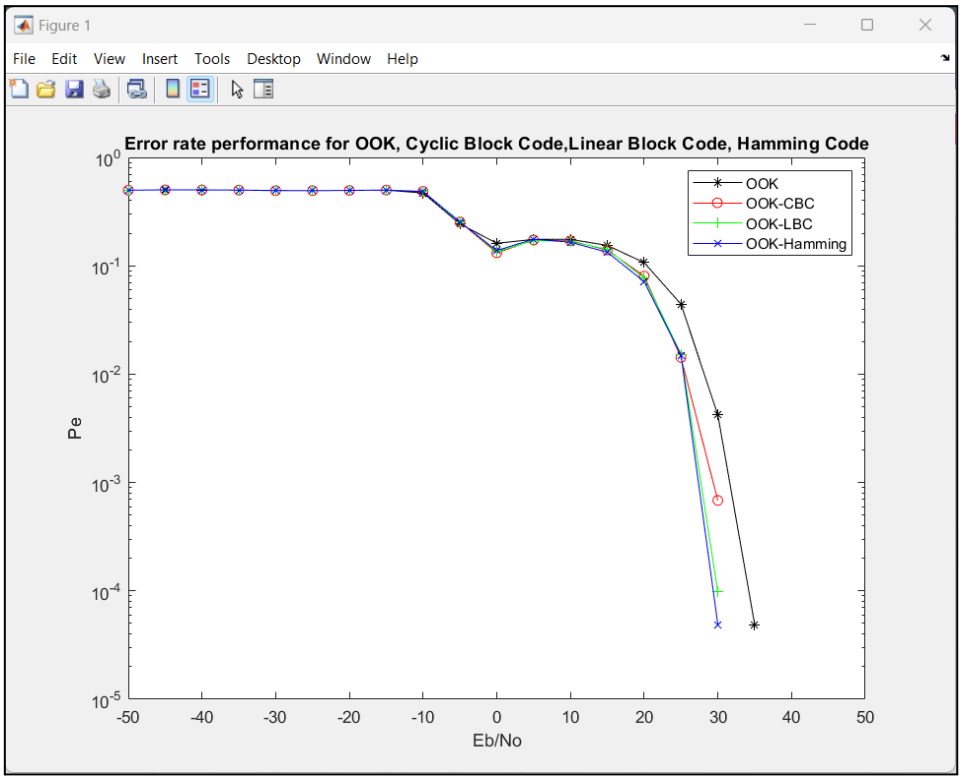

Figure 10: Bit Error rate performance against SNR with Error Control Method for OOK

When the signal to Noise Ratio is less than -10, the noise is stronger than the signal. Therefore, we are not able to decode the correct signal values. In Figure 10, as signal values become stronger over noise values, we notice that the Bit Error Rate performance improves and there are fewer errors.

We can observe that channel coding is effective in protecting the integrity of the signal that is being sent from the Transmitter through the Channel to the Receiver. The results after implementing Cyclic Block code and Hamming Code are effective as seen from the figure above.

As the Signal to Noise Ratio increases, the bit error rate after implementing the channel coding is lower than the original On-Off Keying Method used in Amplitude Shift Keying.

3.2 BPSK Error Control Results

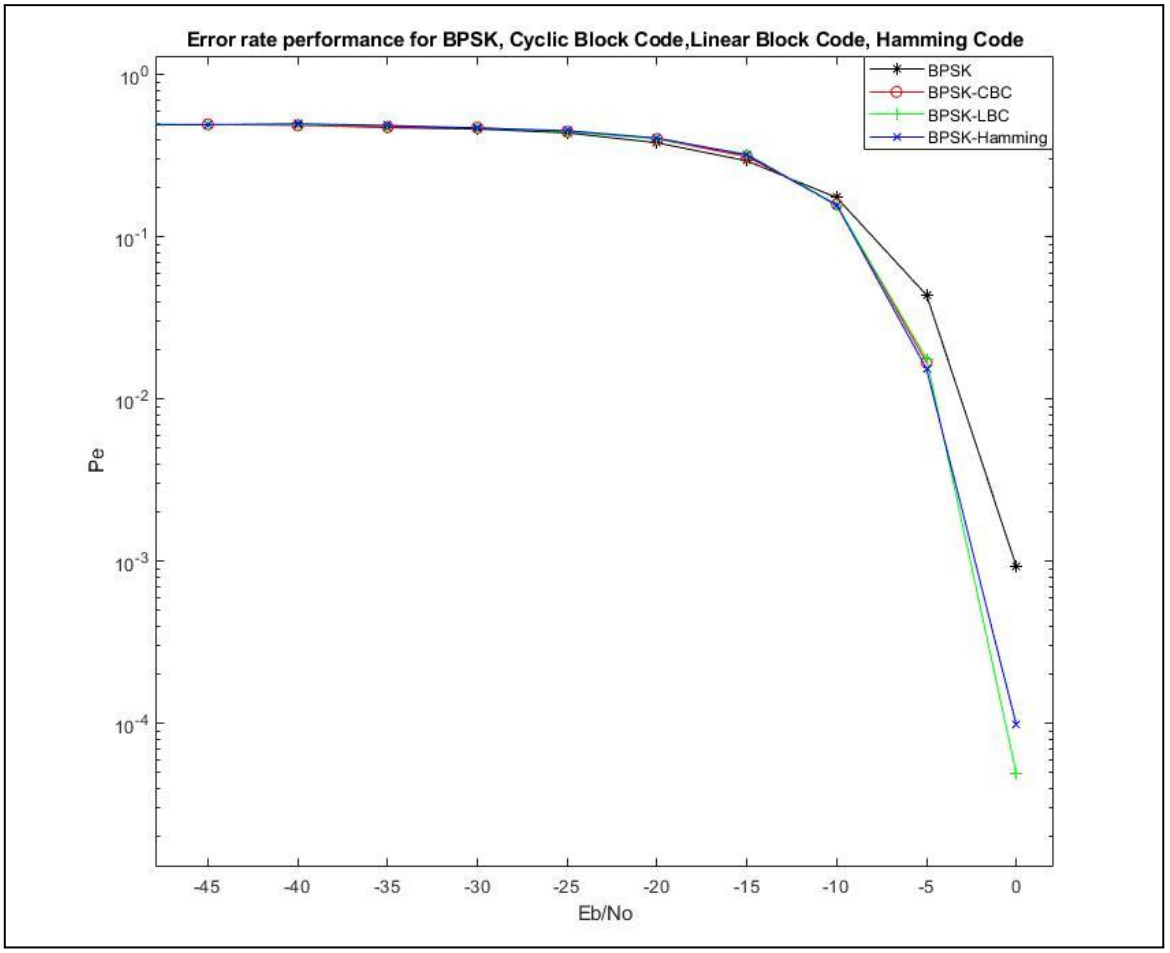

Figure 11: Bit Error rate performance against SNR with Error Control Method for BPSK

From Figure 11, it showed a lower bit error rate when error control is implemented on the BPSK signal. Out of all the error control techniques, we can see that the linear block coding was most effective at correcting the errors due to the noise from the channel. Similar to the OOK results from Figure 10, applying error correction improves the overall communication performance.

However, we can see that starting from an SNR value of around 0 the BER of the BPSK after implementing the error control code is close to zero. Hence, we can conclude that the encoding and decoding of BPSK signals will not be effective if the SNR is more than zero.

3.3 BFSK Error Control Results

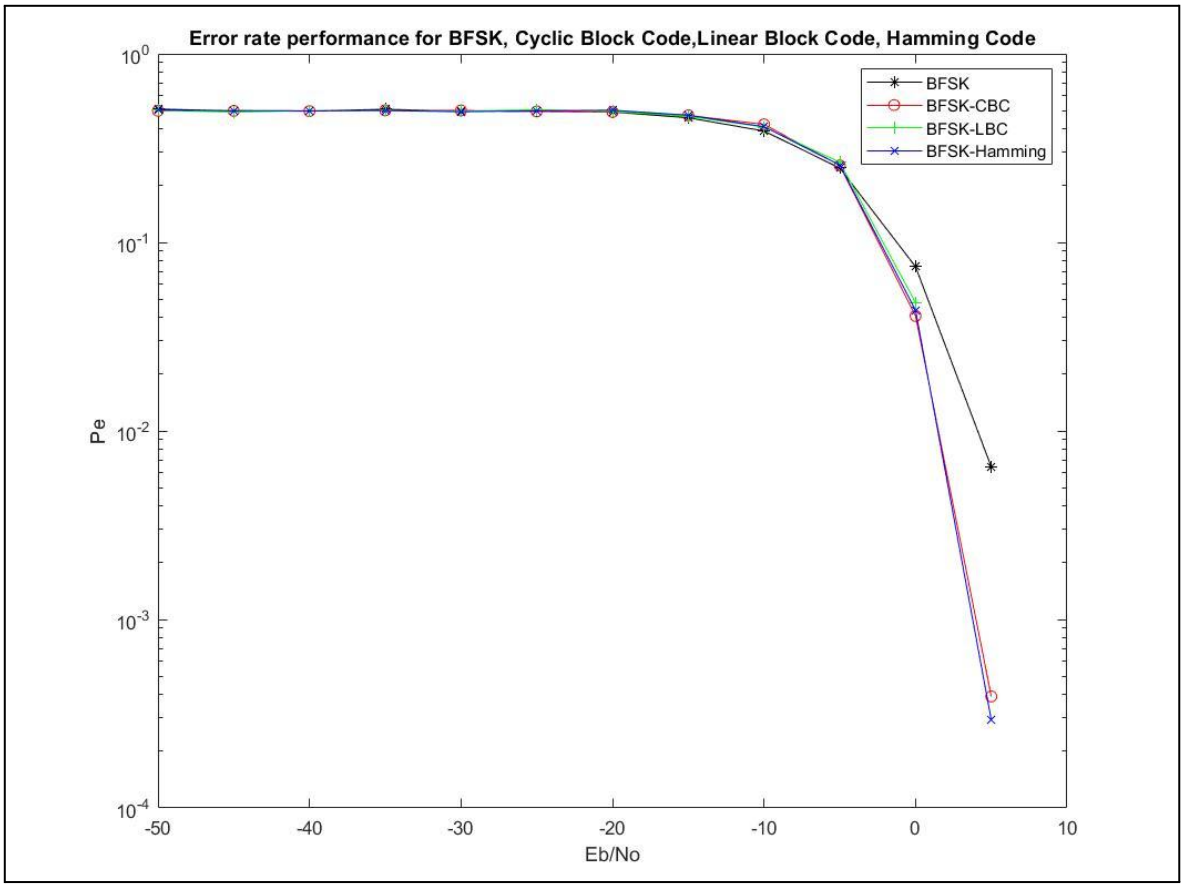

Figure 12: Bit Error rate performance against SNR with Error Control Method for BFSK

From Figure 12, it can be seen that these error control methods help to improve performance, which is consistent with the OOK and BPSK performance patterns seen above. Hamming codes work the best amongst the 3 error coding methods.

4. Conclusion

Through this project, we have learnt how noise affects signal transmission from the transmitter to receiver and how to implement various techniques to reduce the bit error rate of transmission.

In Phase 1, we were able to observe how the Bit Error Rate performance varied as the SNR values change during each iteration. As SNR value increases, the Bit Error Rate reduces hence showing an improvement.

As observed from phase 2, OOK performed the worst as bit error rate only reached 0 when SNR is at 35 dB. Meanwhile, BPSK performed the best as the bit error rate reached 0 at 5 dB followed by BFSK where bit error rate reached 0 at 10 dB. This could be because BFSK and BPSK are insensitive to amplitude variations in the channel unlike OOK which is an ASK modulation.

By applying channel coding techniques as shown in phase 3, it is able to further improve the performance of the various modulation techniques used in phase 2. In general, Hamming Codes seemed to have performed better than the 2 other error controls. However, either way, implementing error control codes will definitely help to improve performance by detecting and correcting errors.